Android平台开发技术

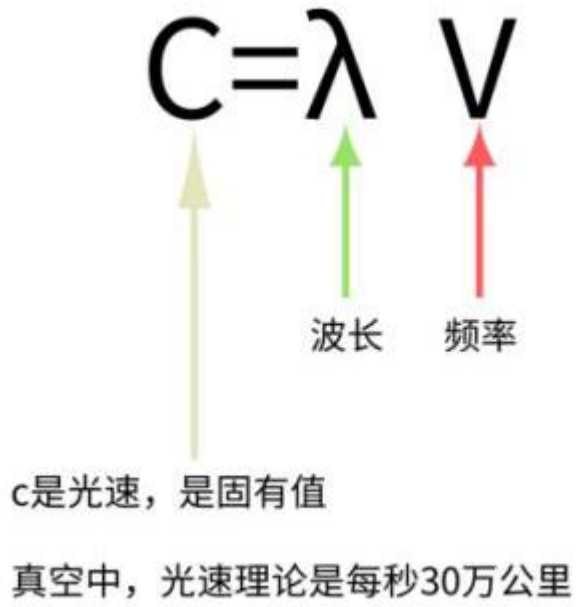
李莹
liying@buaa.edu.cn

学习方法

- 建立开发环境
- 阅读SDK 文档
- 背景知识
  - Java
  - 面向对象
  - 设计模式
  - J2ME等其它手机编程(选)

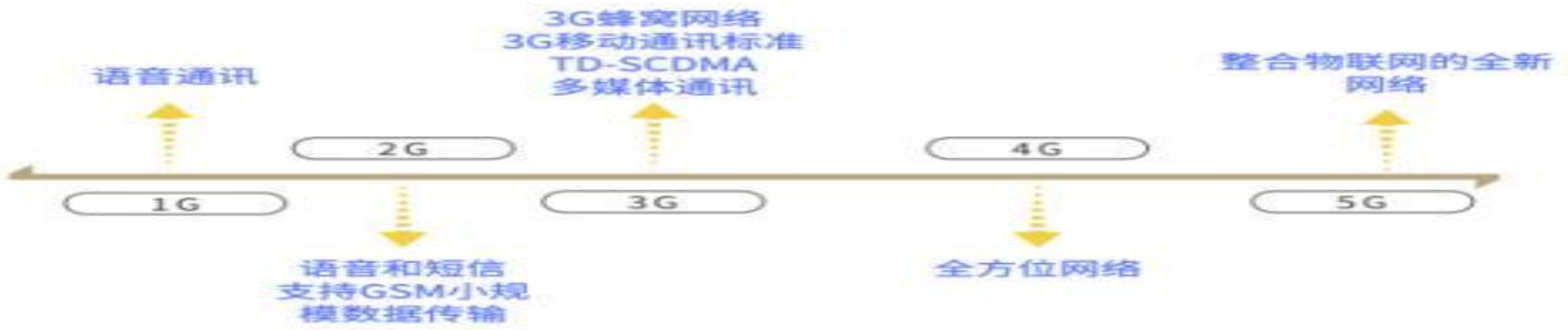
无线通讯技术发展

- 从1G到5G
  - 无线通讯：传输电磁波
  - 频率越来越高，波长原来越小
  - 基站越来越密



无线通讯技术发展

- 第一代：1G(1985)
  - 模拟信号
  - 只能通话
- 第二代：2G(1990)
  - 数字信号(32Kbps)
  - 上网、发送短信
- 第三代：3G(2000)
  - 较高频宽(2Mbps)
  - 视频通讯
- 第四代：4G(2010)
  - 高频宽(100Mbps)
  - 移动通讯
- 第五代：5G(2017)
  - 超高频宽(2.5Gbps)
  - 物联网

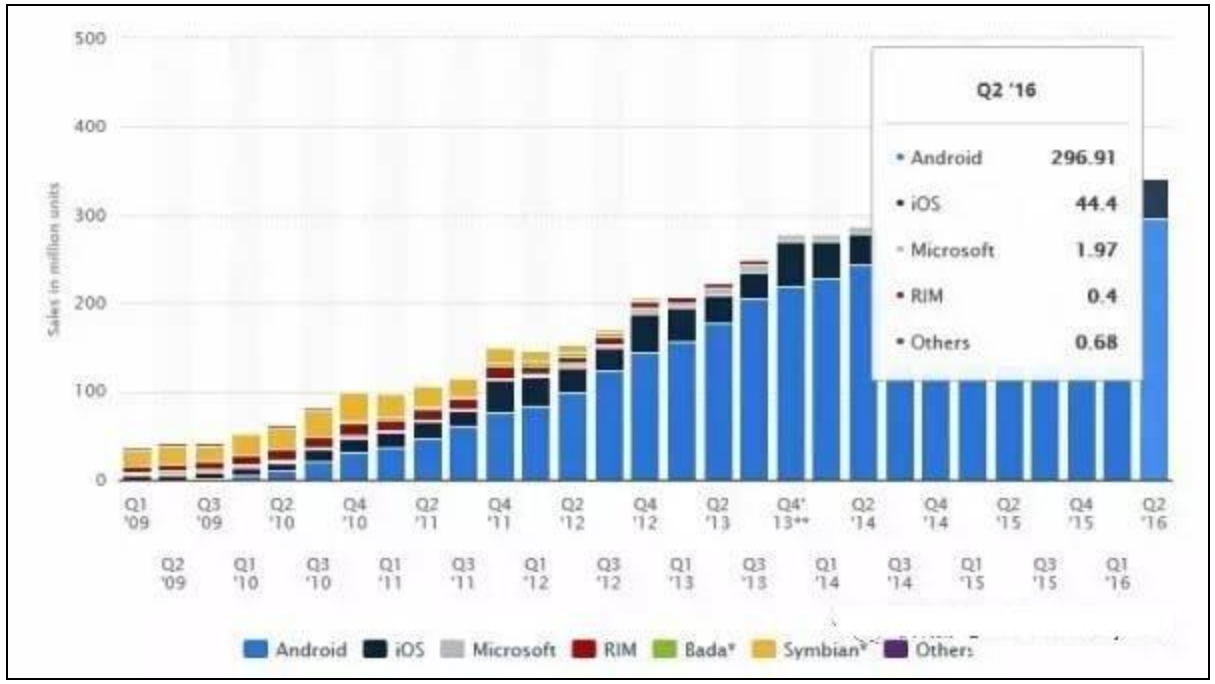


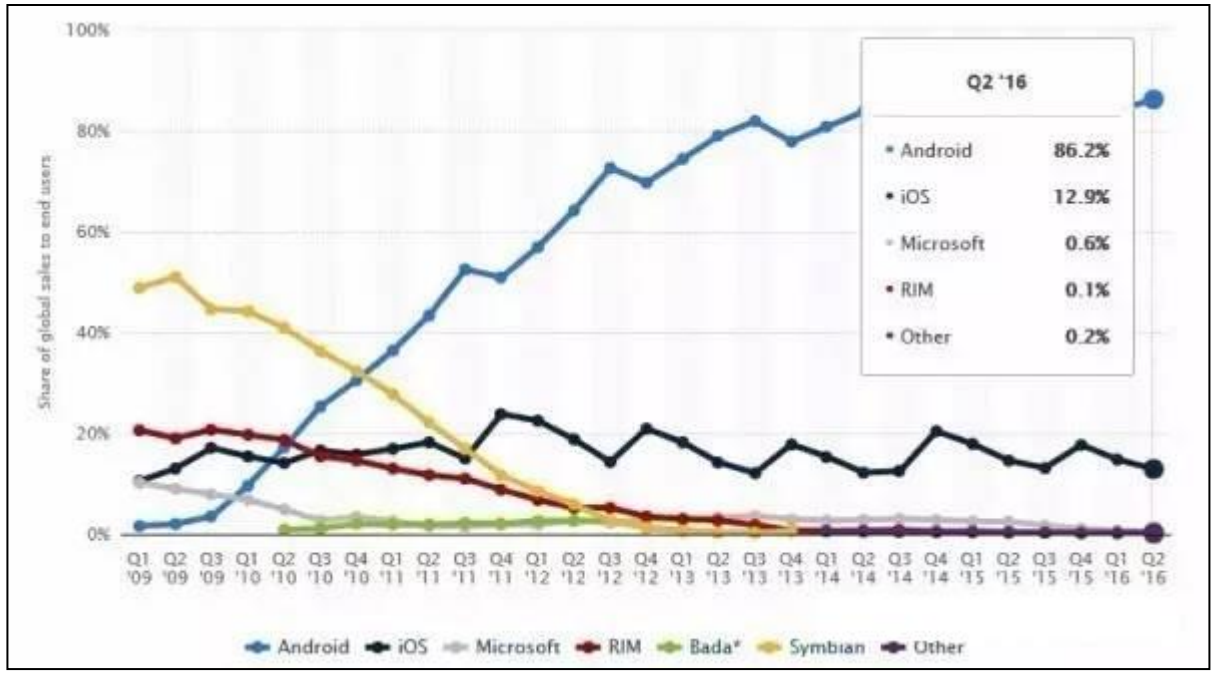
移动操作系统

- 支持移动设备APP开发

  Android、iOS、Windows Phone 7、Symbian和 BlackBerry OS

  - 应用软件互补兼容





Android和IOS

安卓系统和苹果系统

类型	开发平台不同	开发理念不同	运行机制不同	硬件使用效率不同	程序优化机制不同	系统安全性差异
安卓	Linux	免费、开源	无法强制限制内存资源使用	硬件设计和软件开发相对独立	应用软件缺乏监管，优化不到位	开源缺乏管控，极易被破解和破坏
苹果	Unix	收费、闭源	严格控制内存资源使用（墓碑机制、沙漏机制）	软件适配硬件资源	有严格审核机制，利于版本优化	系统封闭，漏洞极少，很难被攻击

开放手机联盟OHA

OHA（Open Handset Alliance）

开放手机联盟是美国Google公司2007年11月5日宣布组建的一个全球性联盟组织。

- 加速智能移动设备的发展、提供用户更多更好更便宜的服务
- 支持Google发布的Android手机操作系统或者应用软件，共同开发名为Android的开放源代码移动系统
- 开放手机联盟包括手机制造商、手机芯片厂商和移动运营商几类。目前，联盟成员数量已经达到了43 家
- 由OHA开发的Android™，是第一个完整的、开放的、免费的智能移动开发平台

Handset Manufacturers
MOTOROLA
htc
smart mobility
SAMSUNG
LG Electronics
Software
Ascender Corporation
ebay
esmertec
Google
livewire
Living Image
NUANCE
pv
SkyPoP
SONiVOX
sound that rocks.
Mobile Operators
中国移动通信
CHINA MOBILE
Designing The Future
KDDI
NTT DoCoMo
Sprint
TELECOM
Telefonica
T-Mobile
open handset alliance
Semiconductor
Audience
BROADCOM.
intel
MARVELL
nVIDIA.
QUALCOMM
SiRF
Synaptics
Texas Instruments
Commercialization
Aplix Corporation
noser
tat
WIND RIVER

Android历史（1）

- 你了解Android吗?
  - 名字由来
    - 外表像人类的机器人——《未来夏娃》，1886
    - Droid字体——Ascender设计的Logo，2010
    - 体现开源、直接面向开发者精神
  - 版本发展
    - 阿童木（Android Beta），2007年11月
    - 发条机器人（Android 1.0）
    - Android第一版Android 1.1，2008年9月

Android历史（2）

- 你了解Android吗？（续）
  - 甜点作为版本号：纸杯蛋糕（Cupcake）、甜甜圈（Donut）、闪电泡芙（Éclair）、冻酸奶（Froyo）、姜饼（Gingerbread）、蜂巢（Honeycomb）、冰淇淋三明治（Ice Cream Sandwich）、果冻豆（Jelly Bean）、奇巧（KitKat）、棒棒糖（Lollipop）、棉花糖（Marshmallow）、牛轧糖（Nougat）、奥利奥（Oreo）、馅饼（Pie）。

猜测下一代Android命名？

Android历史（2）

版本名称	版本号	版本发布时间	对应API
(no code name)	1.0	2008年9月23日	API level 1
(no code name)	1.1	2009年2月2日	API level 2
Cupcake	1.5	2009年4月17日	API level 3,NDK 1
Donut	1.6	2009年9月15日	API level 4,NDK 2
Eclair	2.0.1	2009年12月3日	API level 6
Eclair	2.1	2010年1月12日	API level 7,NDK3
Froyo	2.2.x	2010年1月12日	API level 8,NDK 4
Gingerbread	2.3 – 2.3.2	2011年1月1日	API level 9,NDK5
Gingerbread	2.3.3 – 2.3.7	2011年9月2日	API level 10
Honeycomb	3.0	2011年2月24日	API level 11
Honeycomb	3.1	2011年5月10日	API level 12,NDK 6
Honeycomb	3.2.x	2011年7月15日	API level 13
Ice Cream Sandwich	4.0.1 – 4.0.2	2011年10月19日	API level 14,NDK 7
Ice Cream Sandwich	4.0.3 – 4.0.4	2012年2月6日	API level 15,NDK 8
Jelly Bean	4.1	2012年6月28日	API level 16
Jelly Bean	4.1.1	2012年6月28日	API level 16
Jelly Bean	4.2-4.2.2	2012年11月	API level 17
Jelly Bean	4.3	2013年7月	API level 18
KitKat	4.4	2013年7月24日	API level 19
Kitkat Watch	4.4W	2014年6月	API level 20
Lollipop(Android L)	5.0/5.1	2014年6月25日	API level 21/API level 22
Marshmallow(Android M)	6.0	2015年5月28日	API level 23

三大优势：

- 开放性
- 定制化
- 免费用

开发趋势：

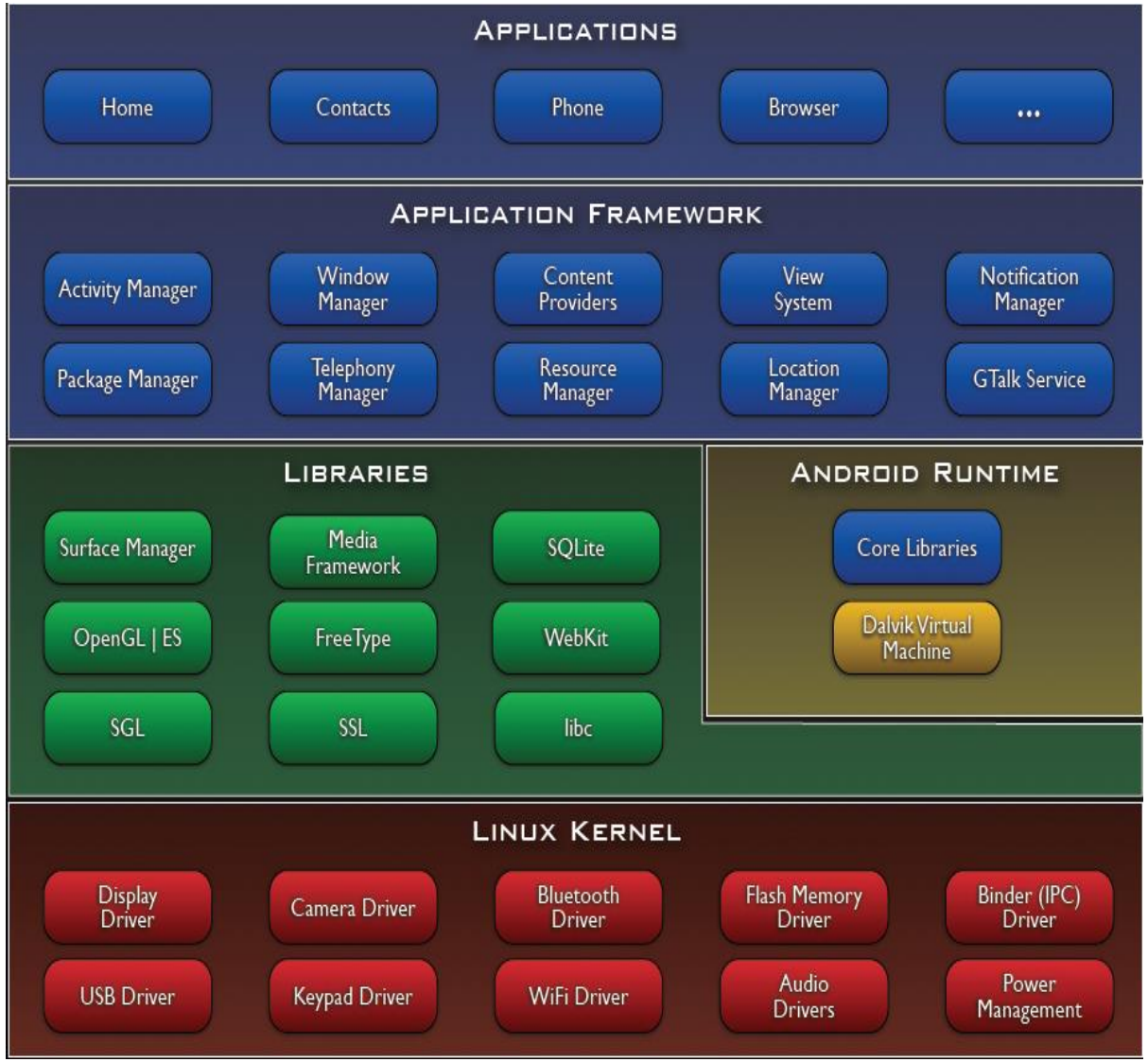
- 系统碎片化：Rom定制
- 屏幕碎片化：适配尺寸

Android系统架构（1）

- 系统架构设计方法

  软件层叠方式（Software Stack）

  - 层内高内聚、层间低耦合
  - 分工协作、易于维护



Android系统架构（2）

- Linux内核层

  提供核心服务

  - 硬件驱动
  - 进程管理
  - 内存管理
  - 网络连接
  - 安全防护

Android系统架构（2）

- 系统运行库层

  提供主要特性支持

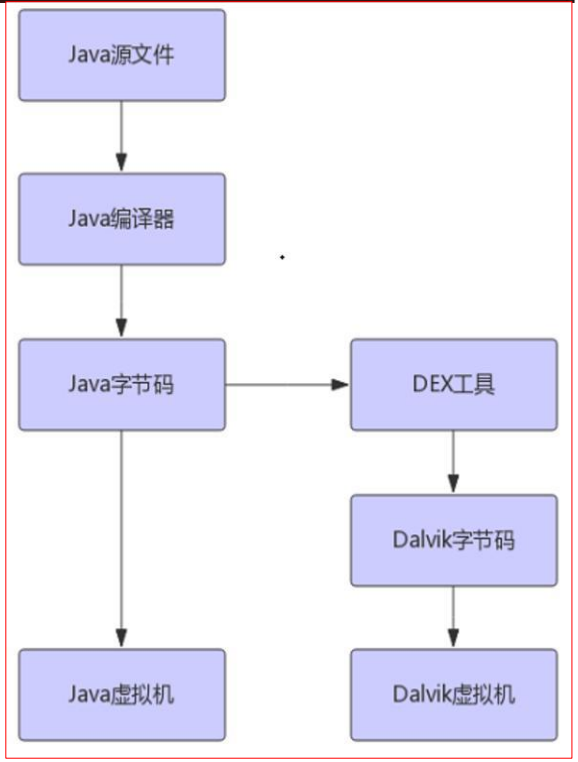
  - 数据库
  - 3D绘图
  - 浏览器
  - JAVA编程
  - 虚拟机

DVM (5.0后升级为ART)

- Google
- Dalvik excutable (*.dex)

JVM

- Oracle (Sun)
- Java bytecode (*.class)



Android系统架构（2）

- 应用框架层

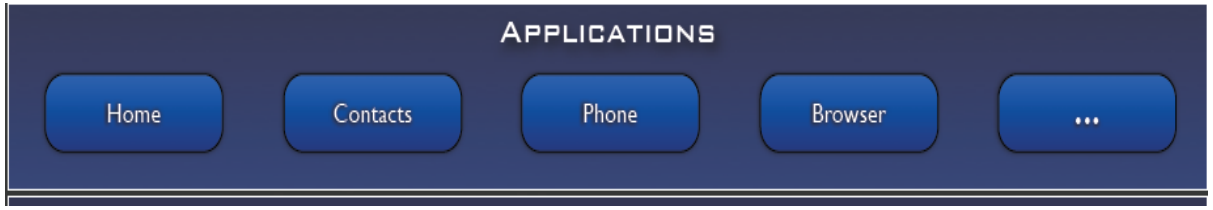
  提供各种API

  - 供系统程序调用
  - 供开发程序调用

- 应用层

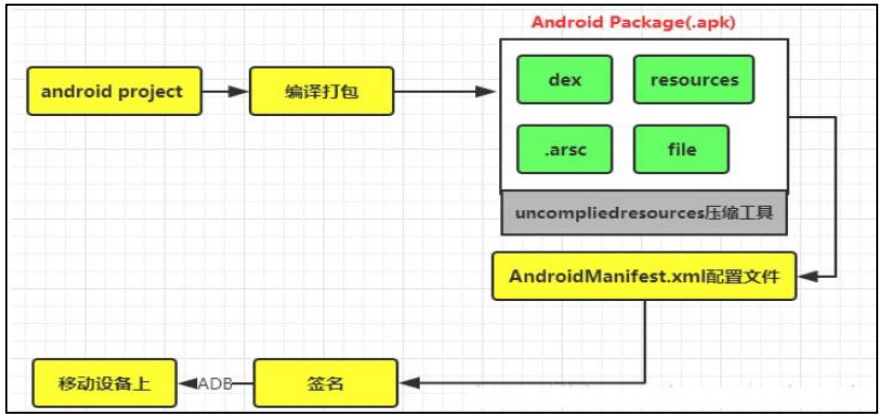
  提供各种应用程序

  - 邮件
  - 日历
  - 地图
  - ……



Android开发环境

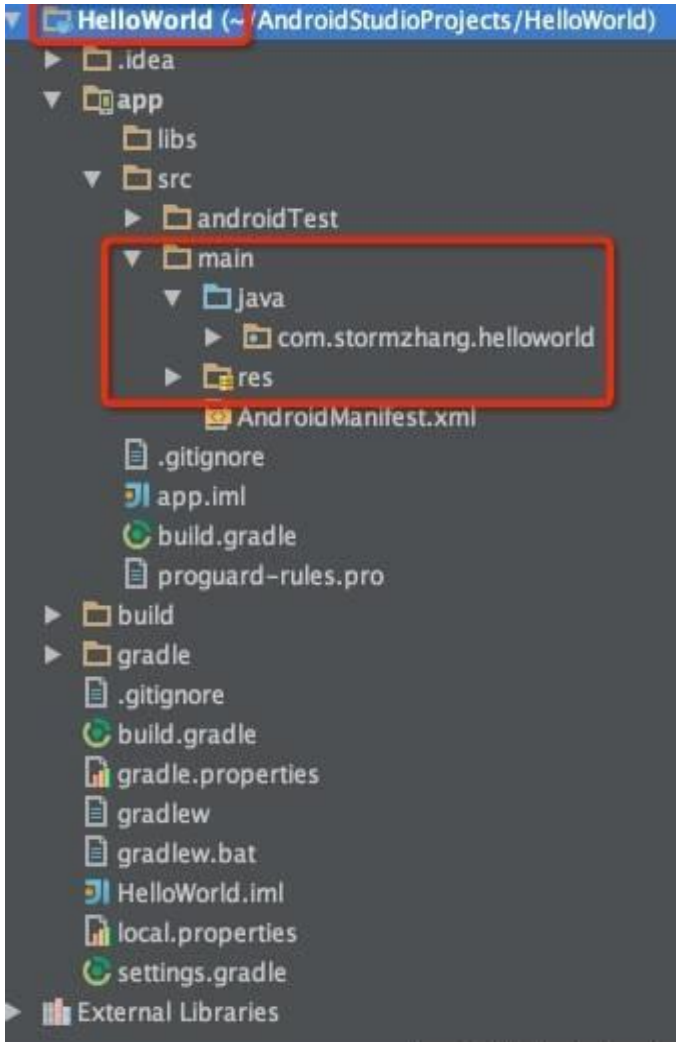
- Android主流开发环境（3种）
  - Eclipse + ADT + SDK（被取代）
  - Android Studio + SDK（建议用）
  - IntelliJ IDEA + SDK（轻量级）



序	名称	说明
1	Dalvik	安卓特有虚拟机
2	AVD	Android virtual device，安卓模拟器
3	ADT	Android development tools，安卓开发工具
4	SDK	Software development kit，软件开发工具包
5	DDMS	Dalvik debug monitor service，安卓调式工具
6	adb	安卓调试桥，在sdk的platform-tools目录下，功能很多,命令行必备
7	DX工具	将.class转换成.dex文件
8	AAPT	android asset packing tool，安卓资源打包工具
9	R.java文件	由AAPT工具根据App中的资源文件自动生成,可以理解为资源字典
10	AndroidManifest.xml	app包名 + 组件声明 + 程序兼容的最低版本 + 所需权限等程序的配置文件

Android Studio开发APP

- 项目结构
  - build：编译时自动生成的文件
  - libs：使用的第三方jar包
  - AndroidTest：编写测试用例
  - java：编写java代码
  - res：使用的资源（图片、布局、字符串）
  - AndroidManifest.xml：配置文件
  - test：编写Unit测试用例
  - . gitignore：将指定目录或文件排除在版本控制外
  - app.iml：IntelliJ IDEA 项目自动生成的文件
  - build.gradle：指定很多项目构建相关的配置
  - proguard-rules.pro：指定项目代码的混淆规则



注意：资源文件只能以小写字母和下划线做首字母，随后的名字中只能出现 [a-z0-9_.]

Android四大组件

- 四大组件
  - 活动：Activity，界面上显示的内容都要放在活动里
  - 服务：Service，后台运行，不可见
  - 广播接收器：Broadcast Receiver，接收和发送广播消息
  - 内容提供器：Content Provider，跨应用程序间共享数据的唯一方法

Android四大组件——活动（1）

- 活动组件（Activity）

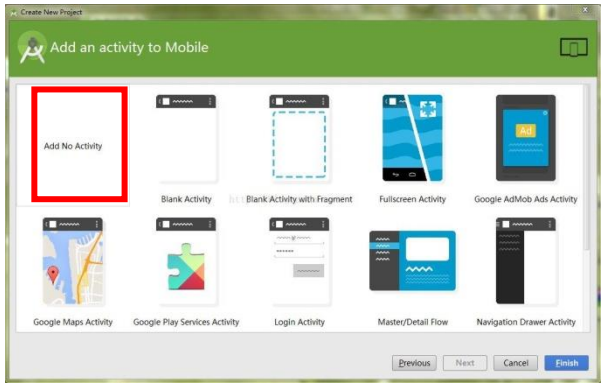
  一个APP可以包含0个或多个活动

  - 当做屏幕、显示界面、用于交互
  - 用栈存储（任务栈）

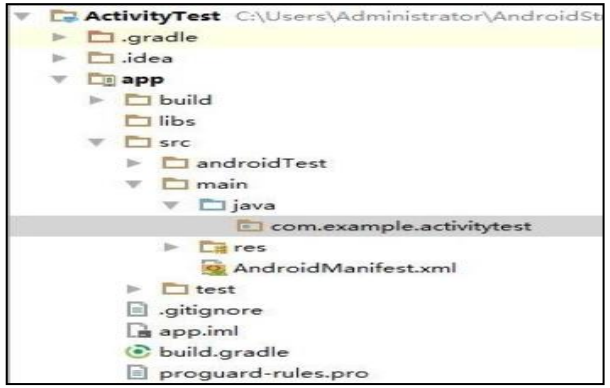
- 新建活动

  逻辑和视图分离，一个活动对应一个布局

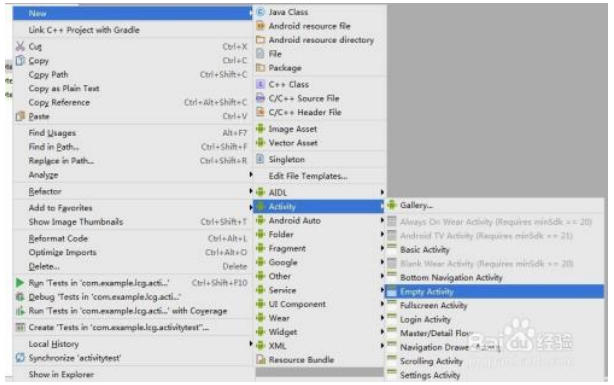
  - 新建活动：自动\手动，在src中新建类，继承Activity类
  - 新建布局：显示界面内容，在res/layout中新建xml布局文件
  - 加载布局：重写活动onCreate()方法，关联布局和活动
  - 注册活动：在AndroidManifest.xml中注册活动(自动完成)
  - 设主活动：在AndroidManifest.xml中设置主活动



(1)选择【空活动】



(2)右击”src->java->项目名称”



(3)选择” New→Activity→Empty Activity”

Android四大组件——活动（2）

- 新建活动（续）

  重写活动的onCreate()方法

```
public class FirstActivity extends Activity {

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

}
```

```
public class FirstActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.first_layout);
    }

}
```

- 调用setContentView()方法加载布局，参数为布局文件的id
- 任何资源都会在R文件中自动生成一个资源id
- R和Android.R不同
- res文件夹下的所有文件不要使用大写字母命名
- 不要试图修改R类中的内容，Android开发工具会自动生成会维护该静态类

Android四大组件——活动（3）

- 新建活动（续）

在AndroidManifest.xml中注册活动

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.activitytest"
    android:versionCode="1"
    android:versionName="1.0" >
    <uses-sdk
        android:minSdkVersion="14"
        android:targetSdkVersion="19" />
    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=".FirstActivity"
            android:label="This is FirstActivity" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

- 在<application>标签注册活动
- 在<activity>标签完成活动注册
- 在<intent-filter>标签设置主活动和将该活动显示到界面应用列表

Android四大组件——活动（4）

新建活动 class MyActivity extends AppCompatActivity{…}

新建布局 在res中创建布局文件xml，设计活动要显示的界面

重写onCreate setContentView(R.layout.first_layout);

注册活动 在AndroidManifest.xml的<application>标签内<activity>标签注册活动：<activity android:name=".活动名"></activity>

设主活动 在AndroidManifest.xml的<application>标签内<activity> 的<intent-filter>设置主活动

启动活动 startActivity(it);//it是一个Intent对象

关闭活动 finish();或Back键

Android四大组件——活动（5）

- 活动的生命周期

  活动是可以堆叠的（隐含时间序）

  - 栈：存储一组活动
    - 入栈：启动活动，startActivity(it)；
    - 出栈：按Back键或者销毁活动，finish()；
  - 任务：管理一组活动，一个任务就是一个栈（活动的集合）

Android四大组件——活动（6）

- 活动的生命周期（续）

  活动的生命周期包含4种状态

  - 运行：活动处于栈顶
  - 暂停：活动不在栈顶，但仍可见
  - 停止：活动不在栈顶，且完全不可见
  - 销毁：活动从栈移除

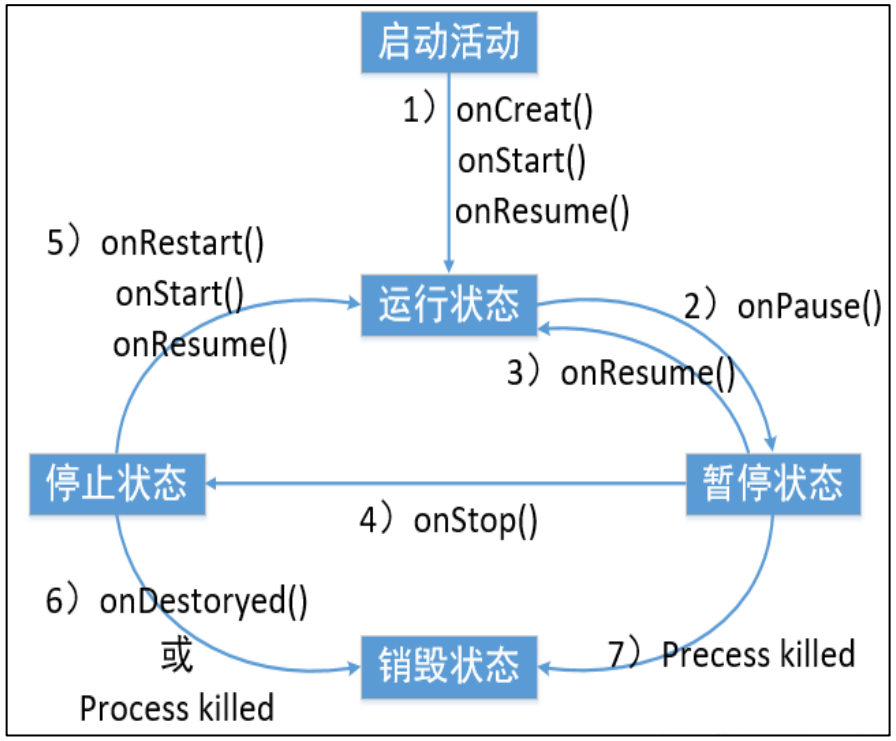
生命周期	被系统回收优先级
运行	1，最不可能被回收
暂停	2，内存极低时被回收
停止	3，内存不足时被回收
销毁	4，最先被回收

Android四大组件——活动（7）

- 活动的生命周期（续）

Activity类定义7个回调方法，覆盖活动生命周期4种状态，回调方法是在事件发生时被自动调用，具体触发事件如下：

- onCreate()：活动创建时
- onStart()：活动由不可见变为可见时
- onResume()：准备好和用户进行交互时
- onPause()：系统准备去启动或恢复另一个活动时
- onStop()：活动完全不可见时被停止时
- onRestart()：活动由停止状态变为运行状态之前
- onDestroy()：活动被销毁之前

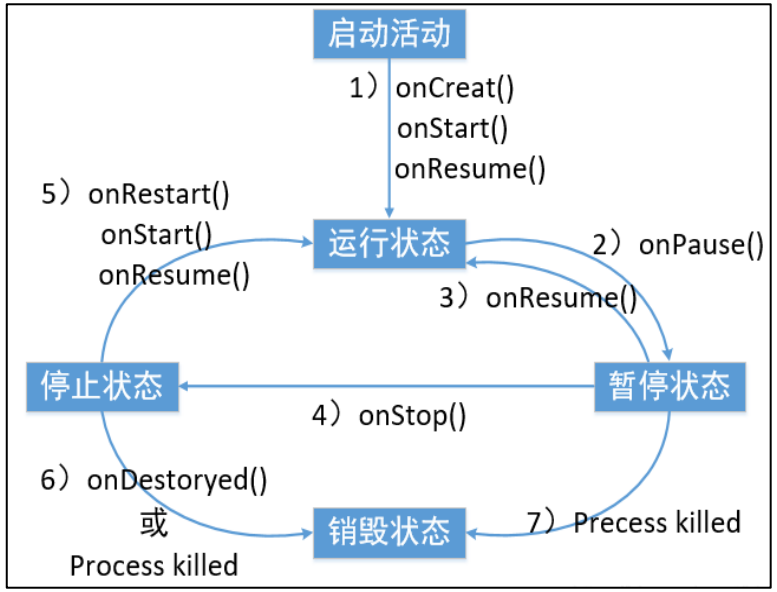


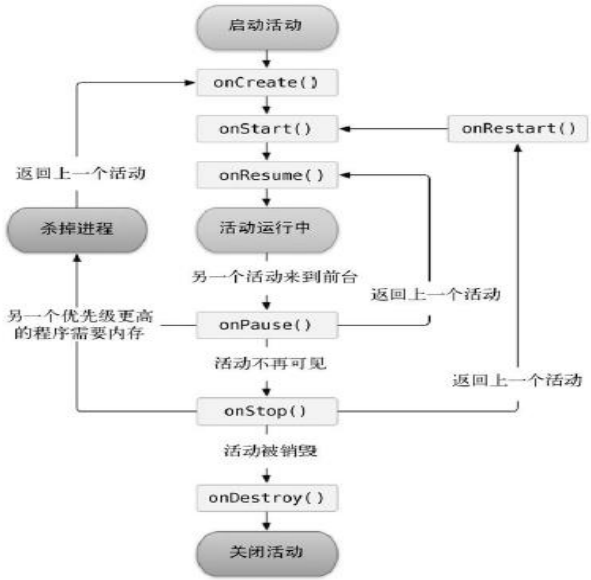
Android四大组件——活动（8）

- 活动的生命周期（续）

活动的3种生存期

- 完整生存期：活动在onCreate()方法和onDestory()方法之间的过程
- 可见生存期：活动在onStart()方法和onStop()方法之间的过程（活动对用户可见，但不一定能交互）
- 前台生存期：活动在onResume()方法和onPause()方法之间的过程（活动既可见，又能与用户交互）





Android四大组件——活动（9）

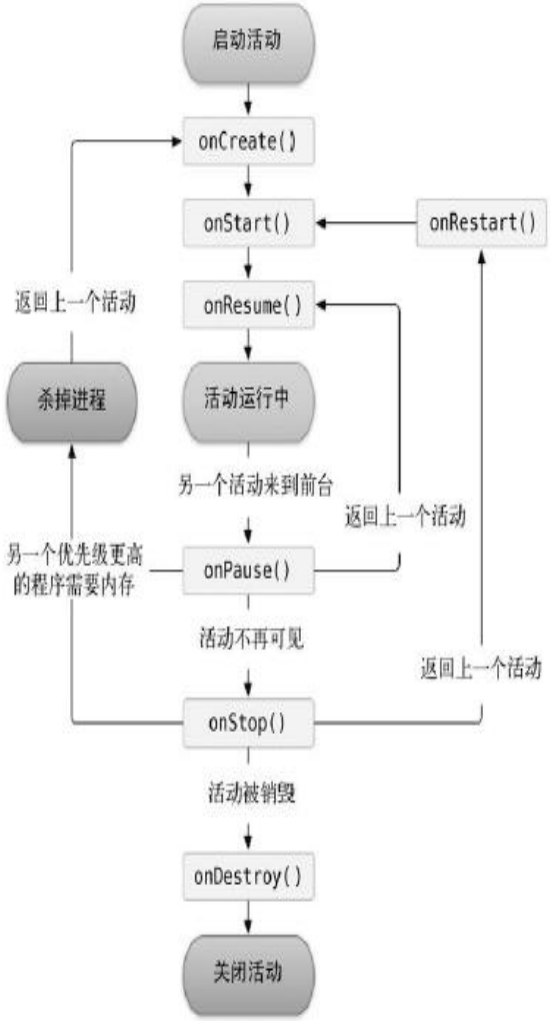
- 活动的生命周期（续）

活动的几种场景（类比舞台）

- 创建活动：onCreate()在后台生成，不可见
- 启动活动：onStart()到等候区，可见但不表演
- 恢复活动：onResume()到台中央，可见且开始表演
- 暂停活动：onPause()到等候区，可见但不表演
- 停止活动：onStop()到台下，不可见
- 重启活动：onRestart()从台下重新返回后台，不可见
- 销毁活动：onDestory()活动消亡

➢ 活动要跟用户交互，一定处于栈顶

➢ 每次只能转换一个状态，状态间不能跳转



Android四大组件——活动（10）

- 活动生命周期的场景
  - 场景1：主活动启动子活动1（普通界面）

- 第一次创建主活动时：
  - 依次执行onCreate() 、 onStart() 和onResume()
- 启动活动1时，子活动1会覆盖主活动：
  - 依次执行主活动的onPause()和onStop()
- 在子活动1中按Back键返回主活动
  - 依次执行主活动的onRestart()、 onStart()和onResume()，但不会执行onCreate()

Android四大组件——活动（11）

- 活动生命周期的场景
  - 场景2：主活动启动子活动2（对话框）

```
<activity android:name=".NormalActivity">
</activity>
<activity android:name=".DialogActivity"
    android:theme="@android:style/Theme.Dialog">
</activity>
```

- 第一次创建主活动时：
  - 依次执行onCreate()、onStart()和onResume()
- 启动活动2时，子活动2不会覆盖主活动：
  - 只执行主活动的onPause()，但不执行onStop()
- 在子活动2中按Back键返回主活动
  - 只执行主活动的onResume()

- android:theme=”@android:style/Theme.Dialog” // 设置主题类型
- android:theme=” Theme.Black” // 设置主题属性

Android四大组件——活动（12）

- 活动生命周期的场景
  - 场景3：主活动按Back键退出

- 依次执行主活动的onPause()、onStop()和onDestroy()
- 最终销毁MainActivity

Android四大组件——活动（13）

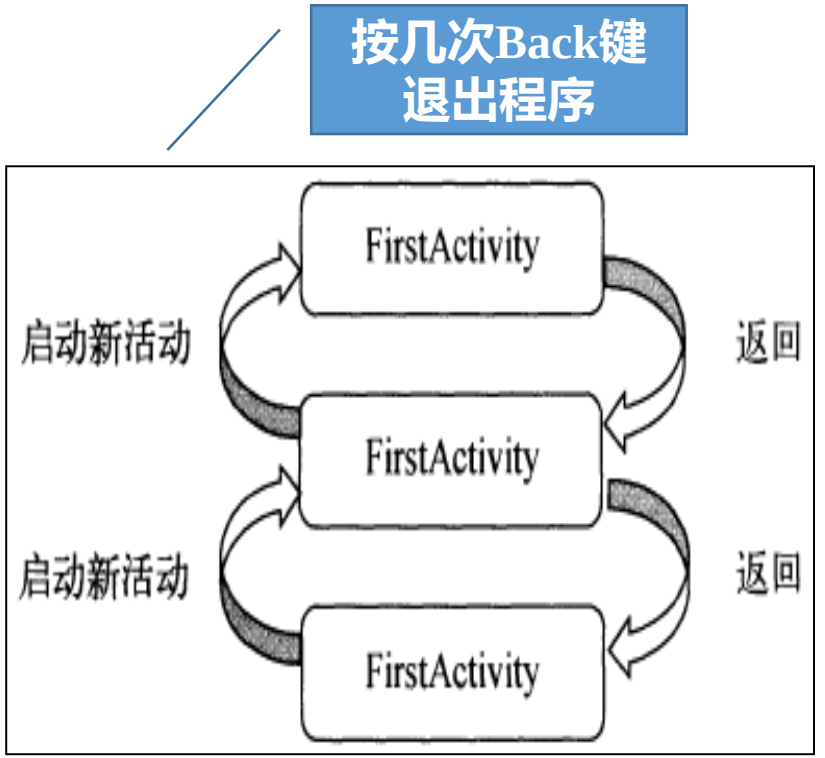
- 活动的启动模式
  - 四种启动模式：standard、singleTop、singleTask、singleInstance
    - standard：默认，每次启动创建该活动一个新实例，无论活动是否已经在栈中
    - singleTop：如果栈顶有该活动就直接使用，则不再创建新实例
    - singleTask：如果栈中有该活动，则将该活动置于栈顶（之上的所有活动全部出栈）
    - singleInstance：用一个新栈管理该活动，实现活动共享
  - 设置启动模式：在AndroidManifest.xml中给<activity>指定android:launchMode属性

```
<activity
    android:name=".FirstActivity"
    android:launchMode="singleTask"
    android:label="This is FirstActivity">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

Android四大组件——活动（14）

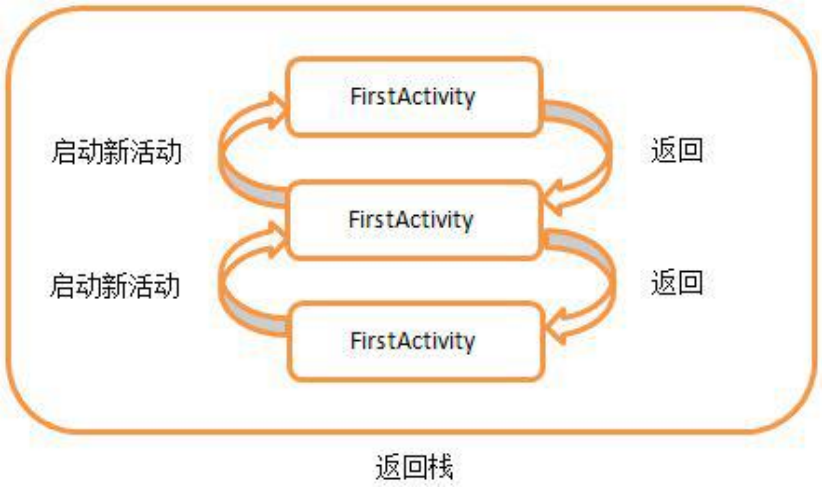
- 活动的启动模式——standard
  - 场景：当前活动自己启动自己
  - 操作：连续点击两次按钮
  - onCreate()调用次数：点击一次，调用一次

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    Log.d("FirstActivity", this.toString());
    setContentView(R.layout.first_layout);
    Button button1 = (Button) findViewById(R.id.button_1);
    button1.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            Intent intent = new Intent(FirstActivity.this, FirstActivity.class);
            startActivity(intent);
        }
    });
}
```



Android四大组件——活动（15）

- 活动的启动模式（续）
- standard场景：撰写10封邮件，希望打开10个新界面，需要启动10个独立的新activity



standard生命周期流程

- 第一次进入：onCreate onStart onResume
- 在栈顶再次进入： onCreate onStart onResume
- 不在栈顶再次进入：onCreate onStart onResume

Android四大组件——活动（16）

- 活动的启动模式（续）
- singleTop场景：搜索栏，在搜索结果界面可以直接继续新的搜索，返回一次就能到主界面

情况一：栈顶已经存在

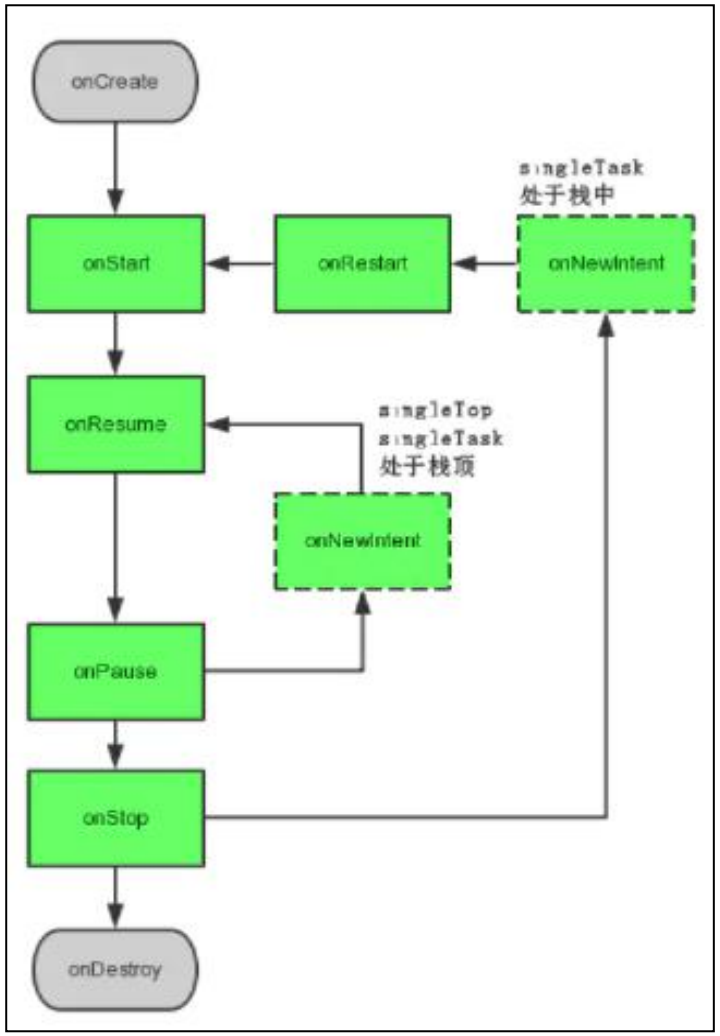
情况二：栈顶不存在

singleTop生命周期流程

- 第一次进入：onCreate onStart onResume
- 在栈顶再次进入：onNewIntent
- 不在栈顶再次进入：onCreate onStart onResume

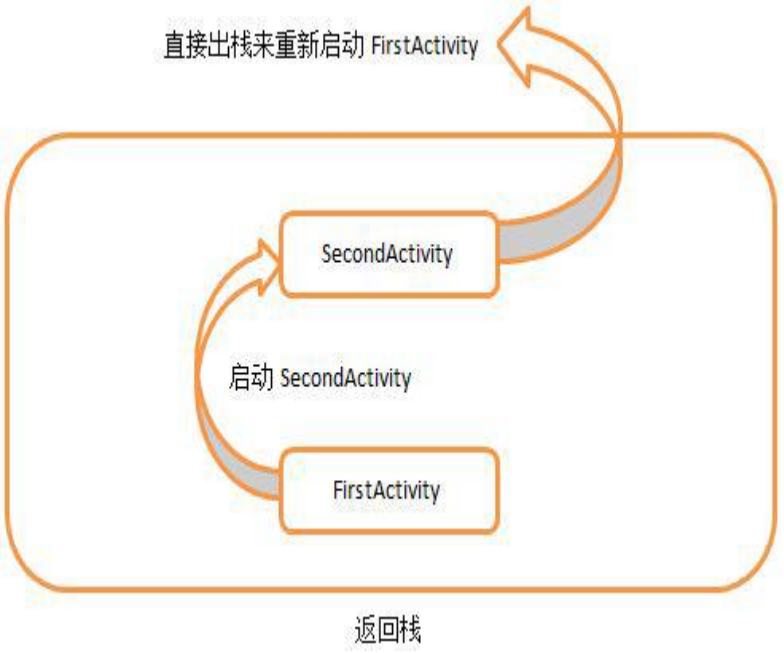
课外补充

- 理解onNewIntent()
  - 存在问题：通过Intent启动一个活动时，如果栈中已经存在该活动，系统仍会创建一个新的活动实例并显示
  - 解决方法：避免活动被多次实例化，将活动的加载模式设置为单任务模式
  - 到达效果：再次启动活动时，不会调用onCreate()产生新实例，而是调用onNewIntent()并重用栈中该活动的实例
  - 具体实现：在AndroidManifest.xml中将活动launchMode属性设置为singleTask或SingleTop



Android四大组件——活动（17）

- 活动的启动模式（续）
- singleTask场景：微信主页面，只要打开过，就只显示这一个。

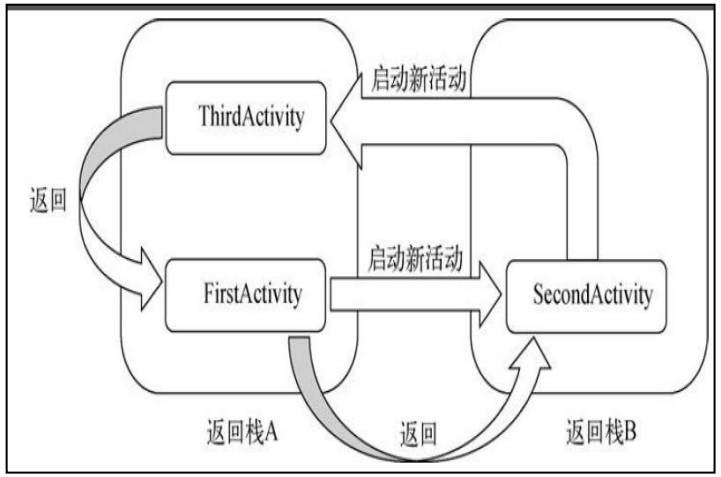


singleTask生命周期流程

- 第一次进入：onCreate onStart onResume
- 在栈顶再次进入：onNewIntent
- 不在栈顶再次进入：onNewIntent onRestart onStart onResume

Android四大组件——活动（18）

- 活动的启动模式（续）
- singleInstance场景：闹铃界面进入设置闹铃界面，按Home返回闹铃界面而非主界面



singleInstanc生命周期流程

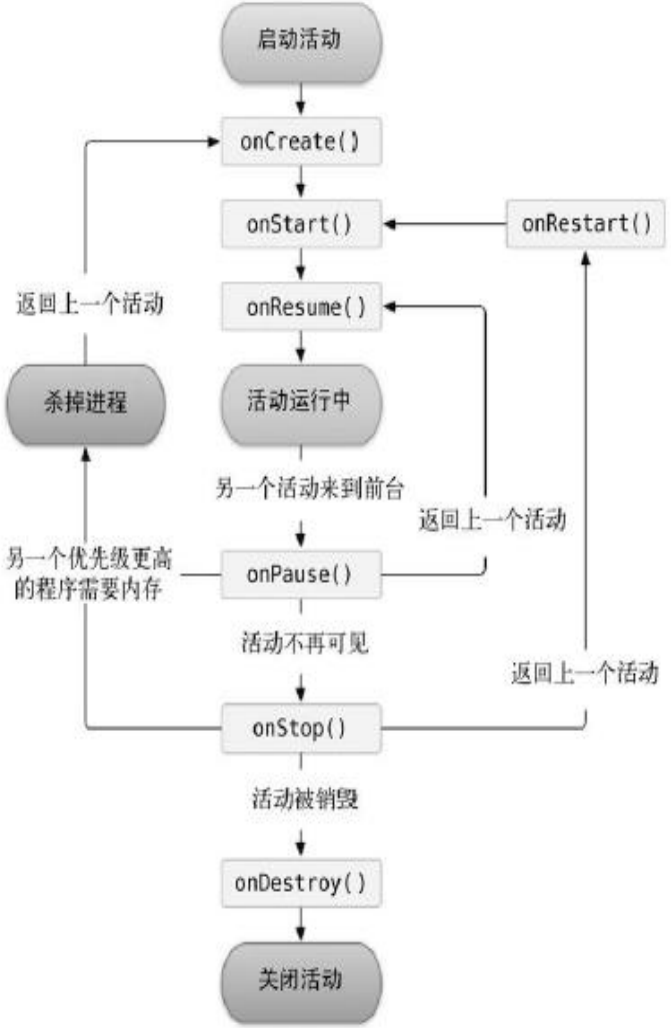
- 活动1启动活动2：1) 先创建栈B，再创建活动2的activity，依次调用onCreate onStart onResume
  2) 创建Intent，实现活动1到活动2跳转
- 活动2启动活动3：1) 在栈A顶部，创建活动3的activity，依次调用onCreate onStart onResume
  2) 创建Intent，实现活动2到活动3跳转
- 活动3按Back键：活动1处于栈A顶部
- 活动1按Back键：栈A为空，返回栈B

Android四大组件——活动（19）

- 活动被回收问题

  活动处于停止状态，可能会被系统回收

  - 场景：活动A启动活动B，然后按Back再返回活动A
  - 现象：活动A正常显示，但数据全部丢失
  - 原因：执行onCreate()方法而不是onRestart()方法
  - 解决：使用onSaveInstanceState()回调方法



Android四大组件——活动（20）

- 活动被回收问题（续）
  - 活动回收前保存数据：活动回收前会自动调用onSaveInstanceState()方法
  - 活动回收后恢复数据：系统回收方法前会有通过onSaveInstanceState()方法将数据保存在Bundle参数中

```
@Override
protected void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);
    String tempData = "Something you just typed";
    outState.putString("data_key", tempData);
}
```

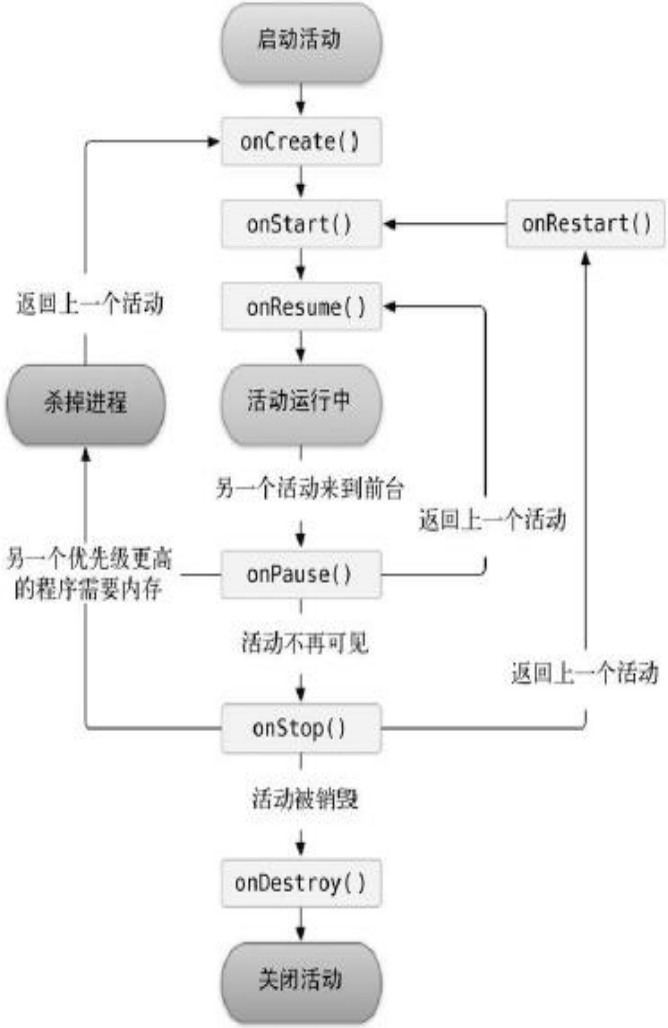
```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    Log.d(TAG, "onCreate");
    setContentView(R.layout.activity_main);
    if (savedInstanceState != null) {
        String tempData = savedInstanceState.getString("data_key");
        Log.d(TAG, tempData);
    }
    ...
}
```

Android四大组件——活动（21）

- 活动被回收问题

  活动处于停止状态，可能会被系统回收

  - 场景：活动A启动活动B，然后按Back再返回活动A
  - 现象：活动A正常显示，但数据全部丢失
  - 原因：执行onCreate()方法而不是onRestart()方法
  - 解决：使用onSaveInstanceState()回调方法



Android四大组件——活动（22）

- 活动被回收问题（续）
  - 活动回收前保存数据：活动回收前会自动调用onSaveInstanceState()方法
  - 活动回收后恢复数据：系统回收方法前会有通过onSaveInstanceState()方法将数据保存在Bundle参数中

```
@Override
protected void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);
    String tempData = "Something you just typed";
    outState.putString("data_key", tempData);
}
```

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    Log.d(TAG, "onCreate");
    setContentView(R.layout.activity_main);
    if (savedInstanceState != null) {
        String tempData = savedInstanceState.getString("data_key");
        Log.d(TAG, tempData);
    }
    ...
}
```

Android四大组件——活动（23）

- 思考

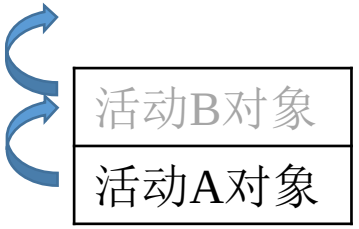
1、活动A启动活动B，活动B再启动活动A，分别对standard、singleTop、singleTask

2、活动A启动活动B，活动B再启动活动C，活动B是singleInstance，活动A和C是默认

- 栈变化情况
- 退出程序需要按几次Back键

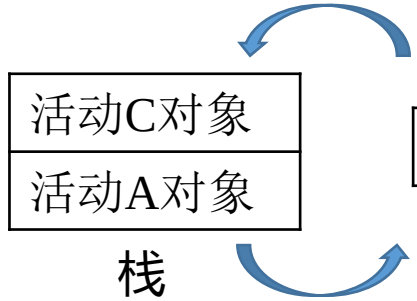
standard栈

singleTop栈
判断栈顶是否存在需要启动的活动



活动B出栈
重新启动活动A

singleTask栈
判断栈中是否存在需要启动的活动



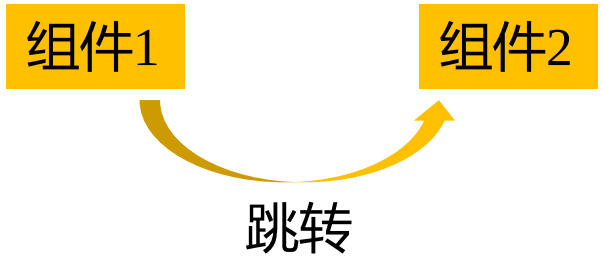
栈

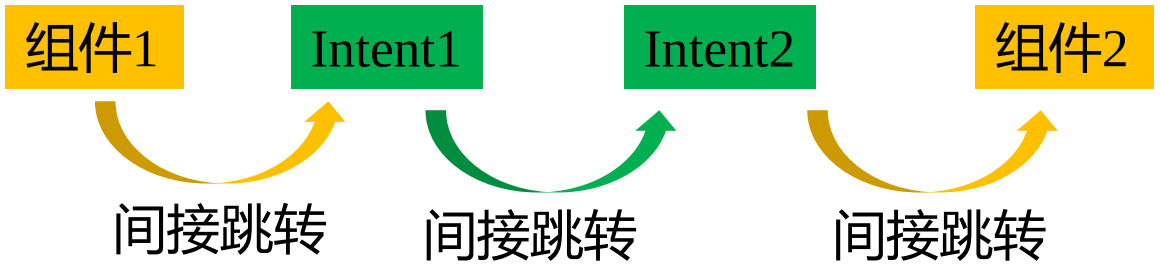
singleInstance栈

问题

- 如何实现两个activity组件间的调转?

  直接跳转 or 间接跳转





Intent作用

- 提供一致的编程模型
- 降低编码耦合度

Android通信桥梁——Intent（1）

- Intent（意图）

  Intent负责Android各组件间跳转和通信

  - 原理：Android的一种消息机制，运行时绑定请求者与实现者。
  - 作用： 在程序内部或者程序之间调用不同的组件，启动活动、服务和发送广播等组件
  - 方式：显式Intent和隐式Intent
    - 显式Intent：意图明显，带两个参数，第二个参数明确要启动的活动
    - 隐式Intent：意图含蓄，可带一个参数或不带参数，由系统分析并确定合适的活动

Intent(Context packageContext, Class **)//显式Intent

- 第一个参数Context：提供一个启动活动的上下文
- 第二个参数Class：指定想要启动的目标活动类

Intent()//无参隐式Intent

Intent(action)//带参隐式Intent，action表示被启动活动具有的动作信息

Android通信桥梁——Intent（2）

- 显式Intent和隐式Intent

```
button1.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        Intent intent = new Intent(FirstActivity.this, SecondActivity.class);
        startActivity(intent);
    }
});
```

显式Intent

```
button1.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        Intent intent = new Intent("com.example.activitytest.ACTION_START");
        intent.addCategory("com.example.activitytest.MY_CATEGORY");
        startActivity(intent);
    }
});
```

隐式Intent

```
<activity android:name=".SecondActivity" >
    <intent-filter>
        <action android:name="com.example.activitytest.ACTION_START" />
        <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>
</activity>
```

隐式Intent

- 在AndroidManifest.xml中通过在<activity> 标签下配置<intent-filter> 内容，可以指定当前活动能够响应的action和category
- 在创建活动java文件中通过Intent构造函数设置指定的action 和category
- 只有action和category都匹配，活动才会响应该Intent

Android通信桥梁——Intent（3）

- 显式Intent和隐式Intent（续）

```
button1.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        Intent intent = new Intent("com.example.activitytest.ACTION_START");
        startActivity(intent);
    }
});
```

```
<activity android:name=".SecondActivity" >
    <intent-filter>
        <action android:name="com.example.activitytest.ACTION_START" />
        <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>
</activity>
```

```
<activity android:name=".SecondActivity" >
    <intent-filter>
        <action android:name="com.example.activitytest.ACTION_START" />
        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="com.example.activitytest.MY_CATEGORY"/>
    </intent-filter>
</activity>
```

category.DEFAULT

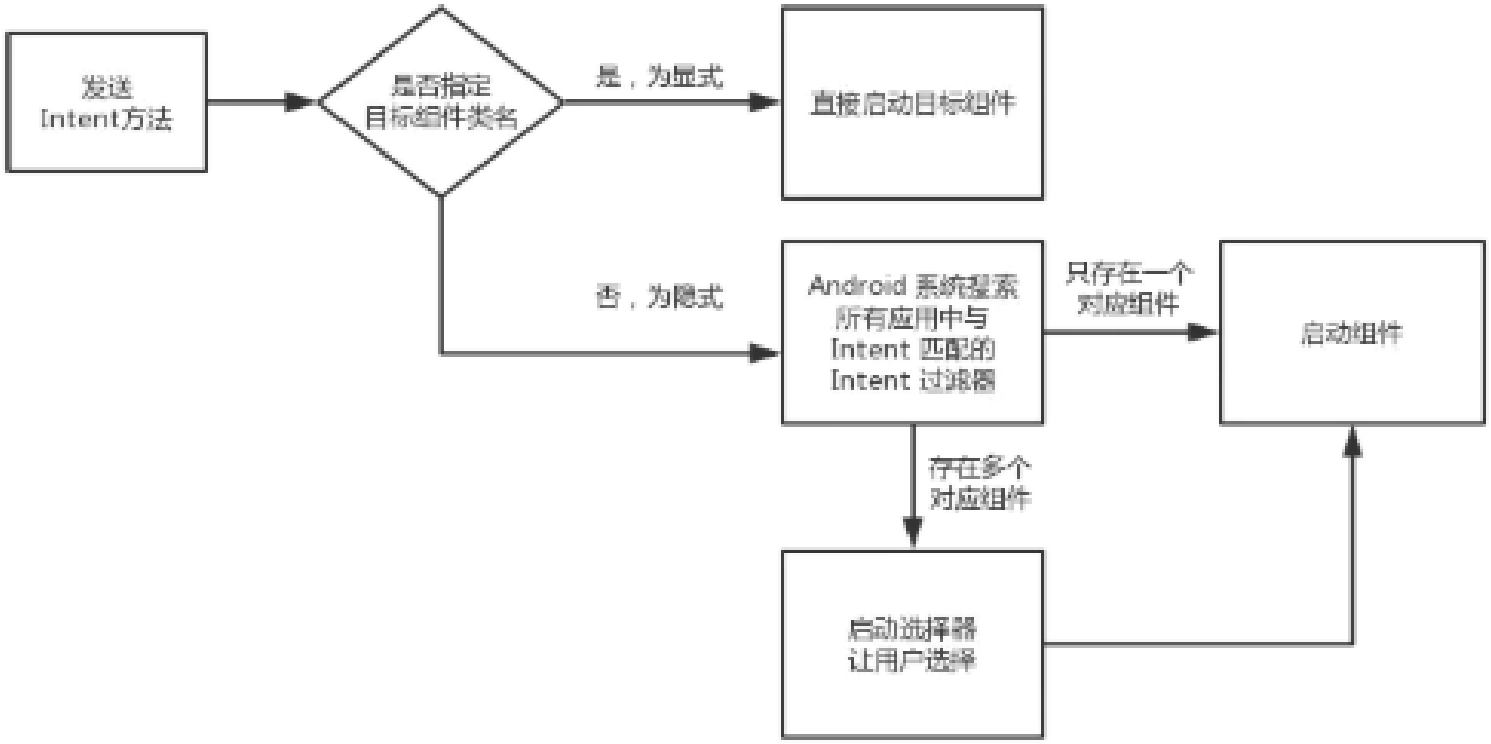
- 是一种默认category类型
- 不需要手动添加到Intent中
- 调用startActivity方法时自动将它 添加到Intent中

注意：

- 一个活动可以添加多个action
- 一个活动可以添加多个category
- 一个intent只能设置一个action
- 一个intent可以设置多个category
- 隐式Intent需要在AndroidManifest.xml要有与该intent匹配的action和category，否则报错

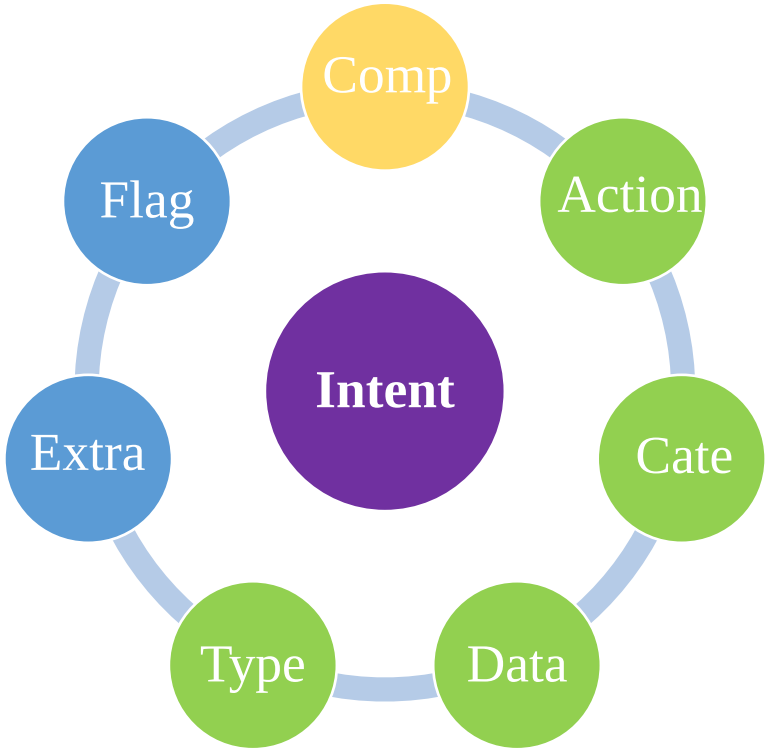
Android通信桥梁——Intent（4）

- Intent工作流程



Android通信桥梁——Intent（5）

- Intent（续）
  - 属性：7种，分别是Component，Action，Category，Data，Type，Extra，Flag



Intent包含的信息

- 显式intent:：必须通过Component指定打开的activity
- 隐式intent：主要通过Action，Category，Data，Type筛选打开的activity
- 附加信息：Extra、Flag

Android通信桥梁——Intent（6）

- Intent的属性
  - Component：目标组件名称，使intent仅传给指定的组件
  - Action：使Intent启动能够响应指定“抽象动作”的组件
  - Category：为Action增加额外的附加类别信息
  - Data：为Action提供待操作数据的路径，即URI
  - Type：为Data所指定的Uri提供对应的数据类型，即MIME
  - Extra：携带的附加数据，实现组件间传递信息
  - Flag：为Intent添加额外的控制旗标，指定Android系统如何启动Activity

Android通信桥梁——Intent（7）

【例】Intent构造示例

```
//显式Intent构造示例
Intent intent = new Intent(this,TagerActivity.class);
```

```
//隐式Intent构造示例
Intent intent=new Intent(Intent.ACTION_VIEW);//设定action为展示内容
intent.setData(Uri.parse("http://www.baidu.com"));//设置data的uri为一个网址
intent.addCategory(Intent.CATEGORY_LAUNCHER);//目标组件为某个应用的首页面
intent.setFlags(Intent.FLAG_ACTIVITY_SINGLE_TOP);//目标Activity的启动方式为single top
intent.putExtra("extra_my","Tom");//附加String数据
```

Android通信桥梁——Intent（8）

【例】Intent隐式应用

```
Intent intent=new Intent(Intent.ACTION_VIEW);
等价
//Intent intent=new Intent();
//intent.setAction ("android.intent.action.View"));
startActivity(intent)
```

Android系统筛选出所有满足Intent过滤器的组件

Android通信桥梁——Intent（9）

【例】点击按钮，返回系统Home桌面

分析：用intent启动满足条件的activity。

```
@Override
public void onClick(View v)
{
      // 创建 Intent 对象
      Intent intent = new Intent();
      // 为 Intent 设置 Action、Category 属性
      intent.setAction(Intent.ACTION_MAIN);
      intent.addCategory(Intent.CATEGORY_HOME);
      startActivity(intent);
}
```

【例】点击按钮，打开指定的网页

```
@Override
public void onClick(View v)
{
      // 创建 Intent
      Intent intent = new Intent();
      String data = "http://www.crazyit.org";
      // 根据指定字符串解析出 Uri 对象
      Uri uri = Uri.parse(data);
      // 为 Intent 设置 Action 属性
      intent.setAction(Intent.ACTION_VIEW);
      // 设置 Data 属性
      intent.setData(uri);
      startActivity(intent);
}
```

Android通信桥梁——Intent（10）

【例】点击按钮，拨打指定电话号码

```
@Override
public void onClick(View v)
{
      // 创建 Intent
      Intent intent = new Intent();
      // 为 Intent 设置 Action 属性（动作为：拨号）
      intent.setAction(Intent.ACTION_DIAL);
      String data = "tel:13800138000";
      // 根据指定字符串解析出 Uri 对象
      Uri uri = Uri.parse(data);
      // 设置 Data 属性
      intent.setData(uri);
      startActivity(intent);
}
```

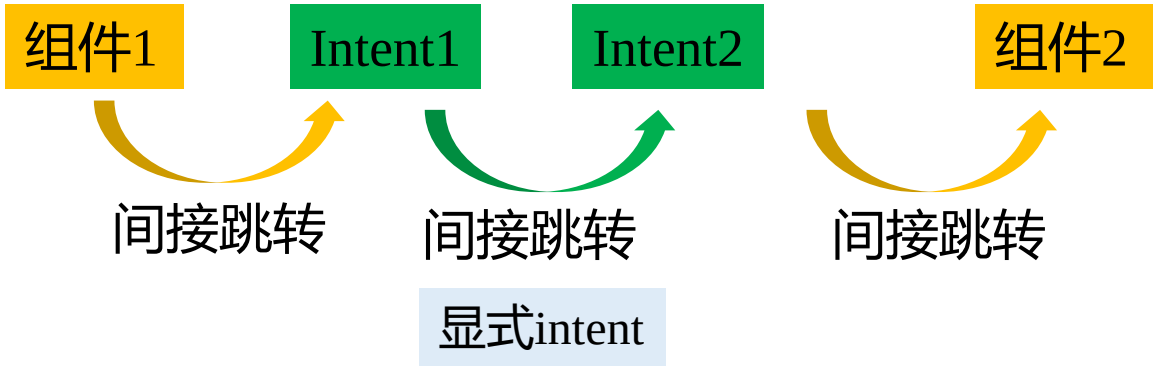
Android通信桥梁——Intent（11）

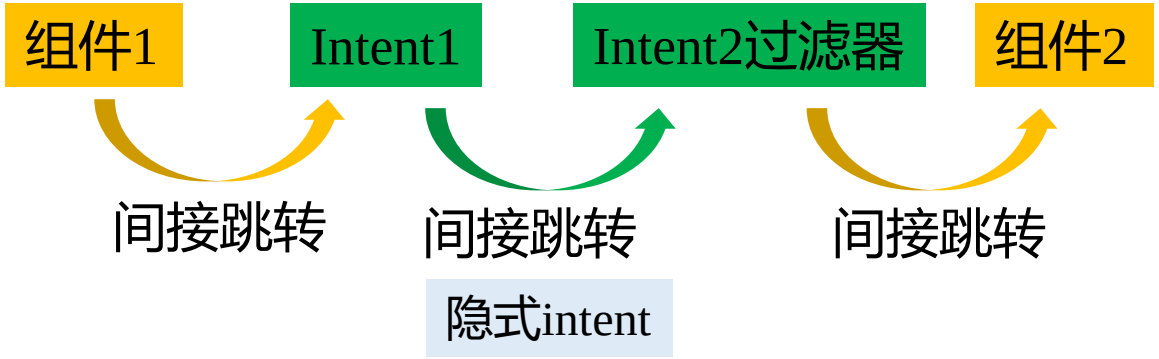
Action 常量	对应字符串	简单说明
ACTION_MAIN	android.intent.action.MAIN	应用程序入口
ACTION_VIEW	android.intent.action.VIEW	显示指定数据
ACTION_ATTACH_DATA	android.intent.action.ATTACH_DATA	指定某块数据将被附加到其他地方
ACTION_EDIT	android.intent.action.EDIT	编辑指定数据
ACTION_PICK	android.intent.action.PICK	从列表中选择某项，并返回所选的数据
ACTION_CHOOSER	android.intent.action.CHOOSER	显示一个 Activity 选择器
ACTION_GET_CONTENT	android.intent.action.GET_CONTENT	让用户选择数据，并返回所选数据
ACTION_DIAL	android.intent.action.DIAL	显示拨号面板
ACTION_CALL	android.intent.action.CALL	直接向指定用户打电话
ACTION_SEND	android.intent.action.SEND	向其他人发送数据
ACTION_SENDTO	android.intent.action.SENDTO	向其他人发送消息
ACTION_ANSWER	android.intent.action.ANSWER	应答电话
ACTION_INSERT	android.intent.action.INSERT	插入数据
ACTION_DELETE	android.intent.action.DELETE	删除数据
ACTION_RUN	android.intent.action.RUN	运行数据
ACTION_SYNC	android.intent.action.SYNC	执行数据同步
ACTION_PICK_ACTIVITY	android.intent.action.PICK_ACTIVITY	用于选择 Activity
ACTION_SEARCH	android.intent.action.SEARCH	执行搜索
ACTION_WEB_SEARCH	android.intent.action.WEB_SEARCH	执行 Web 搜索
ACTION_FACTORY_TEST	android.intent.action.FACTORY_TEST	工厂测试的入口点

Category 常量	对应字符串	简单说明
CATEGORY_DEFAULT	android.intent.category.DEFAULT	默认的 Category
CATEGORY_BROWSABLE	android.intent.category.BROWSABLE	指定该 Activity 能被浏览器安全调用
CATEGORY_TAB	android.intent.category.TAB	指定 Activity 作为 TabActivity 的 Tab 页
CATEGORY_LAUNCHER	android.intent.category. LAUNCHER	Activity 显示顶级程序列表中
CATEGORY_INFO	android.intent.category. INFO	用于提供包信息
CATEGORY_HOME	android.intent.category. HOME	设置该 Activity 随系统启动而运行
CATEGORY_PREFERENCE	android.intent.category.PREFERENCE	该 Activity 是参数面板
CATEGORY_TEST	android.intent.category.TEST	该 Activity 是一个测试
CATEGORY_CAR_DOCK	android.intent.category.CAR_DOCK	指定手机被插入汽车底座（硬件）时运行该 Activity
CATEGORY_DESK_DOCK	android.intent.category.DESK_DOCK	指定手机被插入桌面底座（硬件）时运行该 Activity
CATEGORY_CAR_MODE	android.intent.category.CAR_MODE	设置该 Activity 可在车载环境下使用

Android通信桥梁——Intent（12）

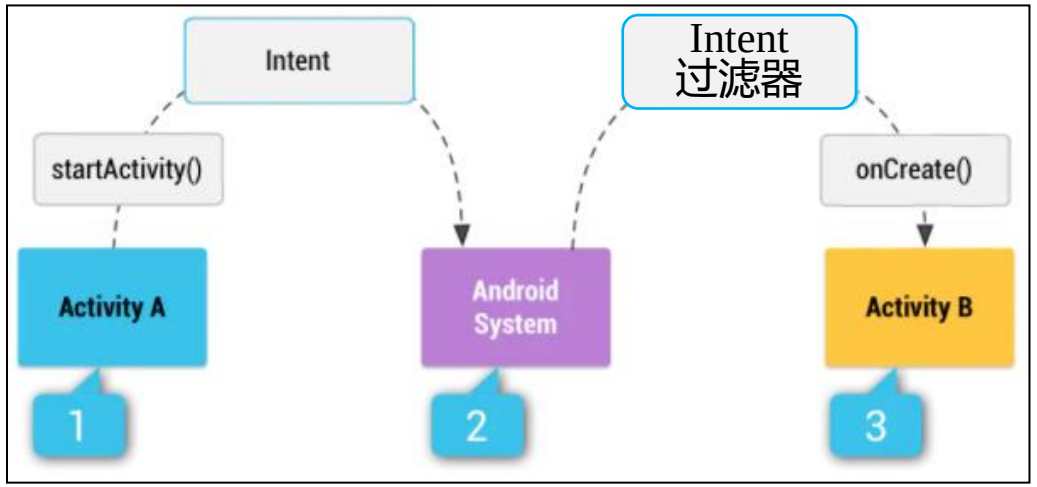
- Intent机制理解
  - 充当中转媒介
  - 调用者只关注提意图
  - 实现调用者和响应者的解耦
  - 统一接口





Android通信桥梁——Intent（13）

- Intent和Intent过滤器
  - Intent针对请求者，用于描述“我的需求”
  - Intent过滤器针对实现者，用于描述“让系统筛选谁能满足我的需求”
  - 两者匹配，则启动相应的组件（准确地说是响应相应的Intent）



隐式Intent通过系统传递以启动其他Activity流程：
步骤1： 请求方A创建包含条件描述Intent，并将其传递给 startActivity()；
步骤2：Android 系统搜索所有应用中与 请求方A中Intent匹配的Intent过滤器；
步骤3：找到匹配项之后，系统通过调用匹配 的实现方（Activity B）的 onCreate() 方法，并将其传递给 Intent，以此启动匹配 Activity。

Android通信桥梁——Intent（14）

- Activity组件响应隐式 Intent需要注意

1、必须在AndroidManifest.xml中的Intent过滤器<intent-filter>…</intent-filter>中添加action和category属性（category.DEFAULT不可缺省）

2、Intent中一定要有action和category属性（category.DEFAULT可缺省）

思考

- 对于组件：
  一个组件可以建立几个Intent
- 对于组件的Intent：
  一个Intent可以设置一个action，多个category、多个data…
- 对于组件的Intent过滤器：
  一个Intent过滤器可以匹配多个action、多个category、多个data

Android通信桥梁——Intent（15）

```
<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="ShareActivity">
    <intent-filter>
        <action android:name="android.intent.action.SEND"/>
        <category android:name="android.intent.category.DEFAULT"/>
        <data android:mimeType="text/plain"/>
    </intent-filter>
    <intent-filter>
        <action android:name="android.intent.action.SEND"/>
        <action android:name="android.intent.action.SEND_MULTIPLE"/>
        <category android:name="android.intent.category.DEFAULT"/>
        <data android:mimeType="application/vnd.google.panorama360+jpg"/>
        <data android:mimeType="image/*"/>
        <data android:mimeType="video/*"/>
    </intent-filter>
</activity>
```

Android通信桥梁——Intent（16）

- 隐式Intent匹配规则
  - 隐式 Intent会与所有设备所有应用组件的 Intent 过滤器进行比对
  - 一个组件可以设置多个Intent过滤器，某个Intent对象只需与其中任何一个匹配即可启动组件
  - 系统会根据Intent与Intent过滤器的action、category、data进行三次比对，全部通过才能匹配
  - 被调活动Intent过滤器中各元素必须是主调活动Intent对象中各个元素的子集
  - 主调活动Intent对象：一个action、多个category、多个data
  - 被调活动Intent过滤器：多个action，多个category、多个data

Android通信桥梁——Intent（17）

- Intent过滤器匹配规则
  - Action匹配规则
    - Intent 只能包含一项 action，而 Intent 过滤器可以包含多项 action
    - 只要 Intent 的 action 可以在 Intent 过滤器中找到对应项即可通过测试
    - 当 Intent 过滤器未声明任何 action 时，任何 Intent 都不能通过匹配
    - 当 Intent 未指定 Action 时，无法通过匹配
  - Category匹配规则
    - 只有Intent中的所有category是Activity中Intent过滤器中category的子集，才算匹配成功
    - 当目标组件为 Activity并且响应隐式 Intent时，必须添加 "android.intent.category.DEFAULT" 到 Intent 过滤器中
    - 当目标组件为广播时，Intent 和 Intent 过滤器都不设置 category，可通过匹配。

Android通信桥梁——Intent（18）

- Intent传递数据
  - 一次传单个数据：putExtra(“键”，值)
  - 一次传一组数据：先存入Bundle 对象，再putExtra(Bundle对象)

Android通信桥梁——Intent（19）

```
//添加 Extra 示例
Intent intent = new Intent(this,TagerActivity.class);
intent.putExtra("extra_int",1000);
int[] ints={1,2,3};
intent.putExtra("extra_int_array",ints);
intent.putExtra("extra_string","teger Activity");
startActivity(intent);
```

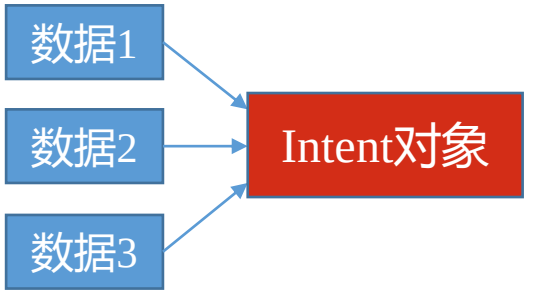
```
//对应取出数据示例
Intent intent = getIntent();
int i=intent.getIntExtra("extra_int",0);
int[] ints=intent.getIntArrayExtra("extra_int_array");
String a = intent.getStringExtra("extra_string");
```

Android通信桥梁——Intent（20）

- Intent一次传递单个数据动作分解

  上一个活动（主）向下一个活动（子）传递数据

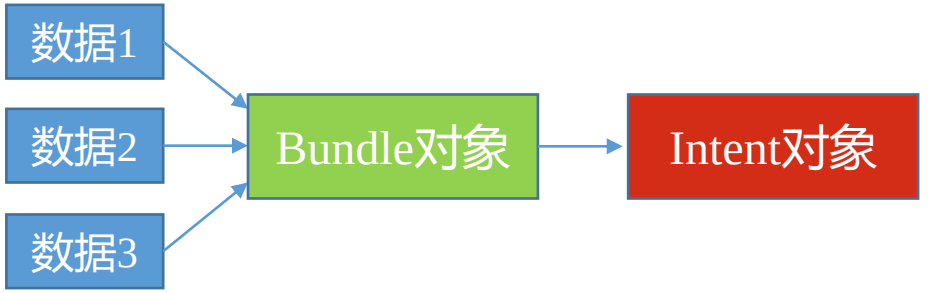
  - 一次传递一个数据：将数据直接嵌入Intent对象中发送
    - 主活动启动子活动：new Intent()
    - 主活动存储一个数据：Intent的putExtra()方法
    - 主活动发送Intent对象：startActivity()
    - 子活动获取Intent对象：getIntent()
    - 子活动接收一个数据：Intent的getStringExtra()方法（或getIntExtra() 或getBooleanExtra()）




```
button1.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        String data = "Hello SecondActivity";
        Intent intent = new Intent(FirstActivity.this, SecondActivity.class);
        intent.putExtra("extra_data", data);
        startActivity(intent);
    }
});
```

```
public class SecondActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.second_layout);
        Intent intent = getIntent();
        String data = intent.getStringExtra("extra_data");
        Log.d("SecondActivity", data);
    }

}
```

Android通信桥梁——Intent（21）



- Intent传递数据（续）

  上一个活动（主）向下一个活动（子）传递数据

  - 一次传递多个数据：将数据间接存入Bundle对象，再用Intent发送Bundle对象
    - 主活动启动子活动：new Intent()
    - 主活动存储多个数据：Bundle的putInt()方法（或putString()等）
    - 主活动嵌入Bundle对象：Intent的putExtras()方法
    - 主活动发送Bundle对象：startActivity()
    - 子活动获取Intent对象：getIntent()
    - 子活动获取Bundle对象：Intent的getExtras()方法
    - 子活动接收多个数据：Bundle的getInt()方法（或getString()等）

```
Intent it1 =  new Intent (A.this, B.class);
Bundle bd = new Bundle();
bd.putInte("num",1);
bd.putSting("detail","hello");
it1.putEntras(bd);
startActivity(it1);
```

```
Intent it2 = getIntent();
Bundle bd = it2.getExtras();
int n = bd.getInt("num");
String d = bd.getString("detail");
```

注意：使用Bundle传递数据的大小<0.5MB，否则会报TransactionTooLargeException异常

Android通信桥梁——Intent（22）

- Intent回传数据（续）

  下一个活动（子）向上一个活动（主）传递数据

  - 主活动启动子活动：new Intent()
  - 主活动发动带唯一请求码的Intent对象：startActivityForResult()回调方法
  - 子活动存储一个数据：Intent的putExtra()方法
  - 子活动发送带结果状态的Intent对象：setResult()方法
  - 主活动接收带结果状态和请求码的Intent对象：onActivityResult()方法

子活动向主活动传递数据

- 主活动不发送数据，只发送请求码，用于接收时判断数据源
- 主活动使用startActivityForResult()可以使在子活动销毁时回调主活动的onActivityResult()方法
- onActivityResult()方法接收子活动返回三个参数：请求码、结果状态、传递数据

```
button1.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        Intent intent = new Intent(FirstActivity.this, SecondActivity.class);
        startActivityForResult(intent, 1);
    }
});
```

主活动

```
public class SecondActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.second_layout);
        Button button2 = (Button) findViewById(R.id.button_2);
        button2.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Intent intent = new Intent();
                intent.putExtra("data_return", "Hello FirstActivity");
                setResult(RESULT_OK, intent);
                finish();
            }
        });
    }
}
```

子活动

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    switch (requestCode) {
        case 1:
            if (resultCode == RESULT_OK) {
                String returnedData = data.getStringExtra("data_return");
                Log.d("FirstActivity", returnedData);
            }
            break;
        default:
    }
}
```

主活动

Android通信桥梁——Intent（23）

```
//原Activity中启动新Activity并请求返回数据
Intent intent = new Intent(this,TagerActivity.class);
startActivityForResult(intent,1);
```

```
//新Activity中设定返回Intent并销毁，销毁后会回调到原 Activity 的 onActivityResult()方法
Intent intent=new Intent();
intent.putExtra("extra_boolean",true);
setResult(RESULT_OK,intent); finish();
```

```
//原 Activity中重写onActivityResult()方法
@Override protected void onActivityResult(int requestCode, int resultCode, Intent data){
    switch (requestCode){
        case 1:
            if (resultCode==RESULT_OK){
                boolean b=data.getBooleanExtra("extra_boolean",false);
            }
            break;
       default:
       }
}
```

Android通信桥梁——Intent（24）

- Intent将多个数据传递给多个页面

```
//页面A发送多个数据给页面B
Intent intent=new Intent(MainActivity.this,BActivity.class);
Intent.putExtra("String","MainActivity中的值");
intent.putExtra("int",11); startActivity(intent);
```

```
//页面B接收多个数据
Intent intent = getIntent();
string = intent.getStringExtra("String");
key = intent.getIntExtra("int",0);
```

```
//页面B继续将多个数据发送给页面C
Intent intent=new Intent(BActivity.this,CActivity.class);
intent.putExtra("String1",string);
intent.putExtra("int1",key);
intent.putExtra("boolean",true);
startActivity(intent);
```

Android通信桥梁——Intent（25）

- Bindle和Intent将多个数据传递给多个页面

```
//页面A发送多个数据给页面B
 Intent intent = new Intent(MainActivity.this, BActivity.class);
Bundle bundle = new Bundle();
bundle.putString("String","MainActivity中的值");
bundle.putInt("int",11);
intent.putExtra("bundle",bundle);
startActivity(intent);
```

```
//页面B接收多个数据
Intent intent = getIntent();
bundle=intent.getBundleExtra("bundle");
```

```
//页面B继续将多个数据发送给页面C
 Intent intent=new Intent(BActivity.this,CActivity.class);
//可以传给CActivity额外的值
bundle.putBoolean("boolean",true);
intent.putExtra("bundle1",bundle);
startActivity(intent);
```

扩展内容

- Intent解析机制Intent解析机制主要是通过查找已注册在AndroidManifest.xml中的所有<intent-filter>及其中定义的Intent，通过PackageManager来查找能处理这个Intent的组件
- 解析中，Android是通过Intent的action、type、category等属性来进行判断，方法如下：
  - ✓ 如果Intent指明action，则目标组件IntentFilter的action列表中就必须包含有这个action，否则不能匹配
  - ✓ 如果Intent没有提供type，系统将从data中得到数据类型。和action一样，目标组件的数据类型列表中必须包含Intent的数据类型，否则不能匹配
  - ✓ 如果Intent中的数据不是content:类型的URI，而且Intent也没有明确指定type，将根据Intent中数据的scheme（比如 http:或者mailto:）进行匹配。同上，Intent 的scheme必须出现在目标组件scheme列表
  - ✓ 如果Intent指定了一个或多个category，这些类别必须全部出现在组建的类别列表中。比如Intent中包含了两个类别：LAUNCHER_CATEGORY和ALTERNATIVE_CATEGORY，解析得到的目标组件必须至少包含这两个类别。

Android——活动（12）

- 常见活动

利用通用资源标志符URI（Universal Resource Identifier）表示数据（图像、字符串、视频、网页等）

- 拨打电话

```
Uri uri = Uri.parse("tel:10086"); // 给移动客服10086拨打电话
Intent intent = new Intent(Intent.ACTION_DIAL, uri);
startActivity(intent);
```

- 发送短信

```
Uri uri = Uri.parse("smsto:10086"); // 给10086发送内容为 “Hello”的短信
Intent intent = new Intent(Intent.ACTION_SENDTO, uri);
intent.putExtra("sms_body", "Hello");
startActivity(intent);
```

Android四大组件——活动（13）

- 常见活动（续）

利用通用资源标志符Uri（Universal Resource Identifier）表示数据（图像、字符串、视频、网页等）

- 打开浏览器

```
Uri uri = Uri.parse(“http://www.baidu.com”); // 打开百度主页
Intent intent = new Intent(Intent.ACTION_VIEW, uri);
startActivity(intent);
```

- 多媒体播放

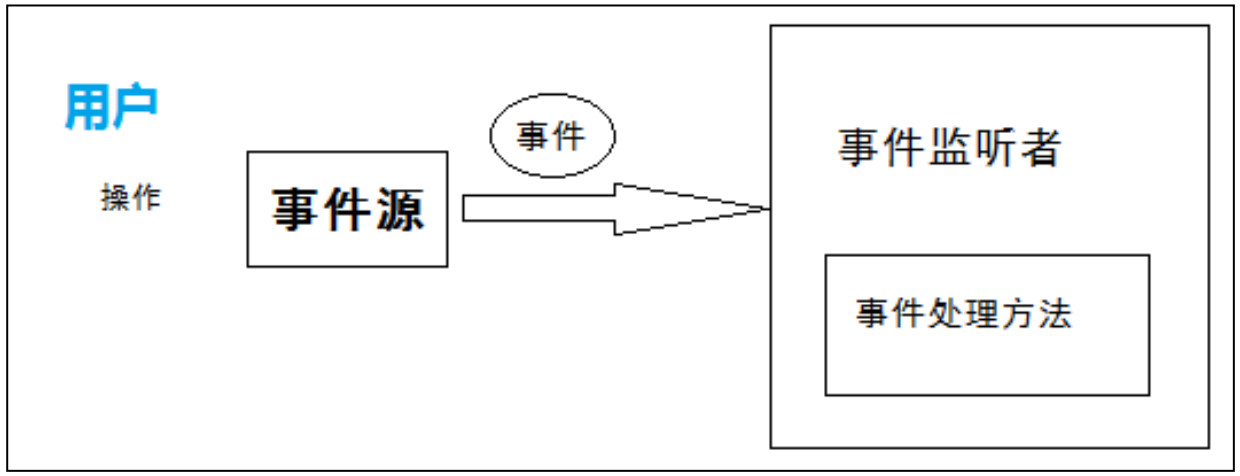
```
Intent intent = new Intent(Intent.ACTION_VIEW);
Uri uri = Uri.parse("file:///sdcard/foo.mp3");
intent.setDataAndType(uri, "audio/mp3");
startActivity(intent);
```

Android事件处理机制

- 事件处理机制

  事件处理机制包括事件源、事件和事件处理三个要素。

  - 事件源：发生事件的组件
  - 事件：封装界面组件发生的特定事件，即可以被识别的操作
  - 事件处理：响应并完成具体的事件



Android事件处理机制

- Android两种事件处理机制
  - 基于监听的事件处理机制：为组件绑定特定的事件监听器
  - 基于回调的事件处理机制：重写组件中特定的响应事件的回调方法

事件监听和事件回调
- 事件回调：处理通用性事件，如按下键盘、弹起键盘、按下组件
- 事件监听：处理特殊性事件，如控制光标四个方向移动

基于监听的事件处理机制

- 基于监听的事件处理机制

  事件监听处理机制包括事件源、事件、事件监听器三个要素

  - 事件源：发生事件的组件
  - 事件：封装界面组件发生的特定事件信息，即可以被识别的操作，由系统自动产生
  - 事件监听器：监听事件源发生的事件，调用相应的事件处理器
    - 事件处理器：响应并完成相应的事件，即事件监听器中的实现方法
    - 监听器用于对特定事件进行监听。一旦监听到特定事件，则由监听器截获该事件，并回调自身的特定方法对事件进行处理。

基于监听的事件类型

- View 组件将截获的事件分为 6 种，对应以下 6 种事件监听器接口
  - OnClickListener 接口此接口处理的是单击事件，例如，在View上进行单击动作，在 View 获得焦点的情况下单击“确定”按钮或者单击轨迹球都会触发该事件。当单击事件发生时，OnClickListener 接口会回调 public void onClick(View v) 方法对事件进行处理。其中参数 v 指的是发生单击事件的 View 组件。
  - OnLongClickListener 接口此接口处理的是长按事件，当长时间按住某个 View 组件时触发该事件。其对应的回调方法为 public boolean onLongClick(View v)，当返回 true 时，表示已经处理完此事件，若事件未处理完，则返回 false，该事件还可以继续被其他监听器捕获并处理。
  - OnFocusChangeListener 接口此接口用于处理 View 组件焦点改变事件。当 View 组件失去或获得焦点时会触发该事件。其对应的回调方法为 public void onFocusChange(View v, Boolean hasFocus)，其中参数 v 表示产生事件的事件源，hasFocus 表示事件源的状态，即是否获得焦点

基于监听的事件类型

- OnKeyListener 接口此接口用于对手机键盘事件进行监听，当View获得焦点并且键盘被敲击时会触发该事件。其对应的回调方法为 public boolean onKey(View v, int keyCode, KeyEvent event)。方法参数说明：keyCode 为键盘码。event 为键盘事件封装类的对象。
- OnTouchListener 接口此接口用来处理手机屏幕事件，当在 View 的范围内有触摸、按下、抬起、滑动等动作时都会触发该事件，并触发该接口中的回调方法。其对应的回调方法为 public boolean onTouch(View v, MotionEvent event)，对应的参数同上。
- OnCreateContextMenuListener 接口此接口用于处理上下文菜单被创建的事件。其对应的回调方法为 public void onCreateContextMenu(ContextMenu menu, View v, ContextMenuInfo info)，其中参数 menu 为事件的上下文菜单，参数 info 是该对象中封装了有关上下文菜单的其他信息。

基于监听的事件处理方法

- 五种响应事件监听的方法

btn = (Button) findViewById(R.id.btnshow);

- **第一种：内部类响应事件**

```
//内部类处理事件响应
btn_login.setOnClickListener(new MyListener());
```

```
//内部类作为事件监听器
class MyListener implements OnClickListener{
    @Overridepublic void onClick(View v){
        ****//响应事件
    }
}
```

基于监听的事件处理方法

- 五种响应事件监听的方法
  - **第二种：匿名类响应事件**

```
//匿名类同时作为事件监听器和处理事件响应
btn.setOnClickListener(new OnClickListener(){
    public void onClick(View v){
        ****//响应事件
}
```

基于监听的事件处理方法

- 五种响应事件监听的方法
  - **第三种：在布局文件中设置UI标签的android:onClick()方法**

```
//在标签上直接绑定事件处理方法
<Buttion
    android:layout_width = “wrap_content”
    android:layout_height = “wrap_content”
    android:onClick = “btnClick”
```

```
//在Activity代码中定义一个事件处理方法
public void btnClick(View view){
    *****//响应事件
}
```

直接将监听绑定到组件的说明

- Android标签基本都onClick方法和onLongClick方法
- 在代码中定义的监听事件处理方法名称必须和android:onClick设置的监听事件处理方法名称一致，并且必须接受一个View类型参数，该方法需要在Activity中实现

基于监听的事件处理方法

- 五种响应事件监听的方法
  - **第四种：外部类响应事件**

```
//预先定义外部类MyExternListener.java
public class MyExternListener implements OnClickListener{
    public MyExternListener(Activity activity, EditText username,EditText passwd{
        this.activity = activity;
        this.username = username;
        this.passwd = passwd;
}
    public void onClick(View v){
         *****//响应事件
}//通过事件监听器的成员变量及重写构造方法，事件源可以传递信息给事件监听器
```

外部类实现监听的优劣？

```
//在Activity代码中调用外部类的事件响应方法
import com.MyExternListener
btn.setOnClickListener(newMyExternListener(MainActivity.this,et_username,et_passwd));
```

基于监听的事件处理方法

- 五种响应事件监听的方法
  - **第五种：Activity自身类响应事件**

```
//Activity直接继承监听器接口
public class MyExternListener extends Activity implements OnClickListener{
    btn.setOnClickListener(this);
    @override
    public void onClick(View v){
        *****//响应事件
}
```

Activity自身响应事件的说明

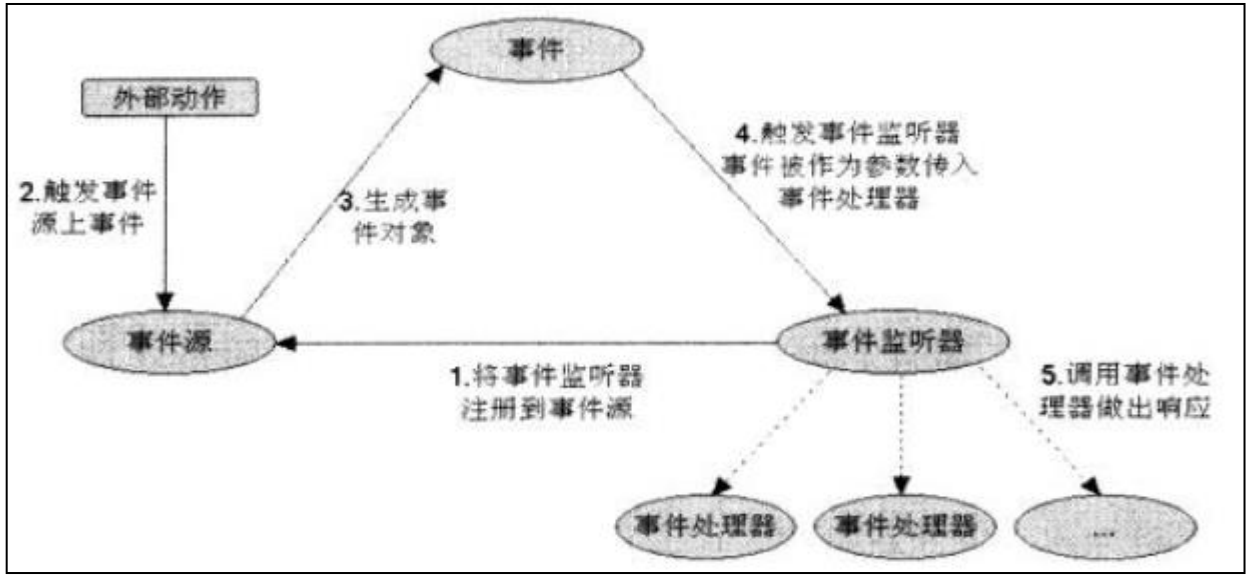
- 监听器设置为this代表的Activity自身类
- 自身类实现事件监听接口
- 自身类重写接口的方法

基于监听的事件处理方法

- 五种响应事件监听的方法
  - 通过设置UI组件的android:onClick属性，然后代码实现方法。这种方法业务逻辑和UI耦合性太高，实际业务一般不用
  - 匿名类：一般只应用特定组件的特定业务响应内部类（只用一次）
  - 外部类：可作为多个UI共同的事件处理，适用多个UI组件复用；当事件源要传递信息给事件监听器时则不太灵活，必须通过成员变量及构造方法
  - Activity自身类：处理比较灵活，实际中用的最多

基于监听的事件处理机制

- 基于监听的事件处理机制（续）



监听的两个顺序

- 一个事件源绑定多个监听器：按事件源(组件)响应的顺序决定监听器的执行顺序
- 一个监听器具有多个事件处理器：按事件中自子事件发生的逻辑顺序决定事件处理器的执行顺序

事件监听机制

- 委派式：事件源事先将整个事件处理委托给事件监听器
- 一个事件可以绑定多个事件监听器
- 一个事件监听器处理同一类型的事件
- 一个事件可以分解为多个子事件，为此一个事件监听器可以添加多个事件处理器响应时，同一事件监听器中各个事件处理器都会被触发，触发顺序（即子事件的执行顺序）是按照事件发生的逻辑顺序由系统确定的
- 事件处理器就是监听器中对某类事件处理的特定方法

基于监听的事件处理机制

- 事件监听机制实现方法

  方法一：一个监听器去处理一类事件（set**Listener）

  方法二：一组监听器去处理一类事件（add**Listener）

```
//一个监听器去处理一类事件
ViewPager.addOnPageChangeListener(new ViewPager.OnPageChangeListener(){
        @Override
        public void onPageScrolled(int position, float positionOffset, int positionOffsetPixels){
                Log.i(TAG, “onPageScrolled:显示时间”);
        }
        @Override
        public void onPageSelected(int position){
                Log.i(TAG, “onPageSelected:记录用户行为”);
        }
        @Override public void onPageScrollStateChanged(int state){
                Log.i(TAG, “onPageScrollStateChanged:显示当前访问人数”);
        }
})
```

- set**Listener只能设置一个监听，之后调用该方法会覆盖之前设置的监听
- add**Listener可设置多个监听，不会被覆盖

```
//一组监听器去处理一类事件
ViewPager.addOnPageChangeListener(new ViewPager.OnPageChangeListener(){
        @Override
        public void onPageScrolled(int position, float positionOffset, int positionOffsetPixels){
                Log.i(TAG, “onPageScrolled:显示时间”);
        }
        @Override
        public void onPageSelected(int position){
                Log.i(TAG, ”onPageSelected:记录用户行为”);
        }
        @Override public void onPageScrollStateChanged(int state){
                Log.i(TAG, “onPageScrollStateChanged:显示当前访问人数”);
        }
})
ViewPager.addOnPageChangeListener(new ViewPager.OnPageChangeListener(){
        @Override
        public void onPageScrolled(int position, float positionOffset, int positionOffsetPixels){
                Log.i(TAG, “onPageScrolled:显示排名”);
        }
        @Override
        public void onPageSelected(int position){
                Log.i(TAG, “onPageSelected:统计分数”);
        }
        @Override public void onPageScrollStateChanged(int state){
                Log.i(TAG, “onPageScrollStateChanged:显示用户名”);
        }
});
```

输出日志是什么?

【例】手指翻页事件的输出日志内容

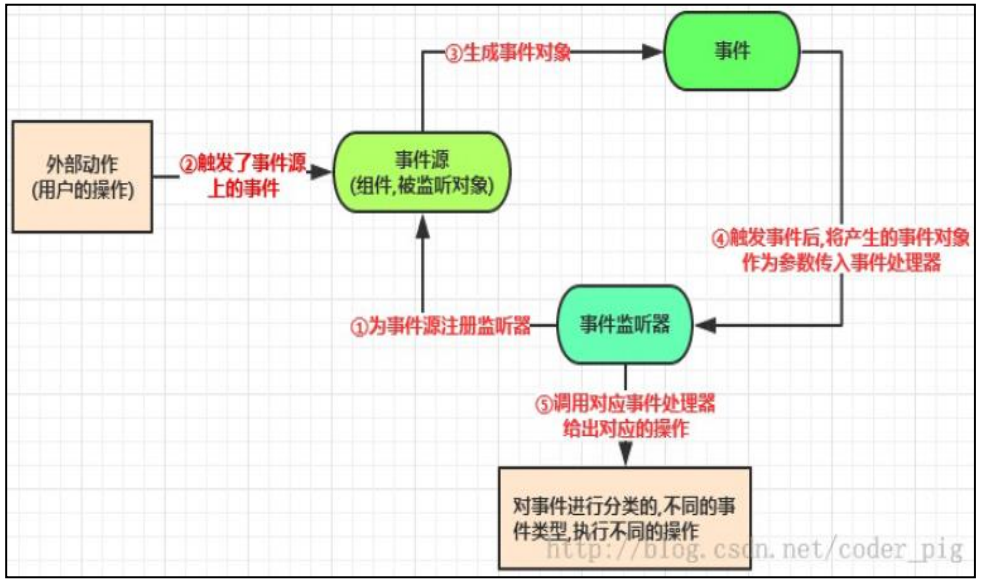
- onPageScrollStateChanged：手指操作屏幕使其发生变化时被调用
- onPageScrolled：屏幕滚动过程中不断被调用
- onPageSelected：手指滑动翻页且翻页距离足够大时被调用

```
//MainActivity.java日志输出
//开始翻页
onPageScrollStateChanged:显示当前访问人数
onPageScrollStateChanged:显示用户名
//连续滑动
onPageScrolled:显示时间
onPageScrolled:显示排名
onPageScrolled:显示时间
onPageScrolled:显示排名
onPageScrolled:显示时间
onPageScrolled:显示排名
//手指放开
onPageScrollStateChanged:显示当前访问人数
onPageScrollStateChanged:显示用户名
onPageSelected:记录用户行为
onPageSelected:统计分数
```

- 同时触发的不同监听器上同类事件处理器的执行顺序是按照组件上绑定监听器的顺序执行
- 同一时间监听器内多个事件处理器的触发顺序是按照事件的执行逻辑顺序

基于监听的事件处理机制

- 基于监听的事件处理流程
  - 第一步：为某个事件源(组件)绑定一个监听器，用于监听用户操作
  - 第二步：用户操作触发了事件源绑定的监听器
  - 第三步：生成对应的事件对象
  - 第四步：将事件源对象作为参数传给事件监听器
  - 第五步：事件监听器对事件源对象进行判断，执行对应的事件处理器(对应事件的处理方法)



基于监听的事件处理机制

```
public class MainActivity extends Activity {
        private Button btnshow;
        @Override
        protected void onCreate(Bundle savedInstanceState){
                super.onCreate(savedInstanceState);
                setContentView(R.layout.activity_main);
                btnshow = (Button) findViewById(R.id.btnshow);
                //直接new一个内部类对象作为参数
                btnshow.setOnClickListener(new BtnClickListener());
        }
        //定义一个内部类,实现View.OnClickListener接口,并重写onClick()方法
        class BtnClickListener implements View.OnClickListener{
            @Override
            public void onClick(View v) {
                Toast.makeText(getApplicationContext(), "按钮被点击了", Toast.LENGTH_
                SHORT).show();
                }
        }
}
```

事件参数去哪里了?

基于监听的事件处理机制

- Android中事件监听机制的优化

事件触发时，Android会将事件信息封装成**Event事件对象。根据实际需求确定是否将事件对象作为参数传递给事件处理器。

- 如果不传递事件对象也能满足设计需求，就不用封装事件对象
- 如果事件处理器中需要使用事件信息，就要封装事件对象并传递事件参数

```
planeView.setOnKeyListener(new OnKeyListener()
{
    @Override
    public boolean onKey(View source, int keyCode, KeyEvent event)
    {
        // 获取由哪个键触发的事件
        switch (event.getKeyCode())
        {
            // 控制飞机下移
            case KeyEvent.KEYCODE_S:
                planeView.currentY += speed;
                break;
            // 控制飞机上移
            case KeyEvent.KEYCODE_W:
                planeView.currentY -= speed;
                break;
            // 控制飞机左移
            case KeyEvent.KEYCODE_A:
                planeView.currentX -= speed;
                break;
            // 控制飞机右移
            case KeyEvent.KEYCODE_D:
                planeView.currentX += speed;
                break;
        }
        // 通知planeView组件重绘
        planeView.invalidate();
        return true;
    }
});
```

基于监听的事件处理机制

思考：

- 事件处理机制和JAVA中方法调用区别是什么
- 谁负责调用事件处理器

基于回调的事件处理

- 回调事件处理机制

  回调事件处理机制包括事件源和事件处理两个要素。

  - 事件源：界面上的组件
  - 事件处理：指定的事件处理方法，该方法继承GUI组件类，并重写类中的事件处理方法

GUI组件View的回调方法

①在该组件上触发屏幕事件: boolean onTouchEvent(MotionEvent event);
②在该组件上按下某个按钮时: boolean onKeyDown(int keyCode,KeyEvent event);
③松开组件上的某个按钮时: boolean onKeyUp(int keyCode,KeyEvent event);
④长按组件某个按钮时: boolean onKeyLongPress(int keyCode,KeyEvent event);
⑤键盘快捷键事件发生: boolean onKeyShortcut(int keyCode,KeyEvent event);
⑥在组件上触发轨迹球屏事件: boolean onTrackballEvent(MotionEvent event);

基于回调的事件处理

例：按钮的回调应用

```
//MyButton.java部分代码
public class MyButton extends Button{
    private static String TAG = "呵呵";
    public MyButton(Context context, AttributeSet attrs) {
        super(context, attrs);
    }
    //重写键盘按下触发的事件
    @Override
    public boolean onKeyDown(int keyCode, KeyEvent event) {
        super.onKeyDown(keyCode,event);
        Log.i(TAG, "onKeyDown方法被调用");
        return true;
    }//重写View的okKeyDown方法
}
```

为什么回调方法的返回值是boolean类型？

```
//布局文件部分代码
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:tools="http://schemas.android.com/tools"
android:layout_width="match_parent"
android:layout_height="match_parent"
tools:context=".MyActivity">
<example.jay.com.mybutton.MyButton
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="按钮"/>
```

在布局文件中使用自定义View时必须用全限定类名

基于回调的事件处理

- 回调方法的返回值

  回调方法的返回值标识该方法是否能完全处理该事件

  - 返回true，表明该方法已经完全处理该事件，事件不会再传播出去
  - 返回false，表明该方法无法完全处理该事件，事件会被再传播出去
    - 传播方向：该组件所在Activity的回调方法

基于回调的事件处理

- 基于监听的事件处理vs基于回调的事件处理
  - 监听模型：事件源和事件监听器相互分离（托管机制），自定义事件
  - 回调模型：事件源和事件监听器融为一体（自管机制），通用型事件
- 监听和回调同时存在的事件处理顺序

基于回调的事件处理

例：按钮的回调事件传播

```
//MyButton.java部分代码
public class MyButton extends Button{
    private static String TAG = "呵呵";
    public MyButton(Context context, AttributeSet attrs){
        super(context, attrs);
    }
    //重写键盘按下触发的事件
    @Override
    public boolean onKeyDown(int keyCode, KeyEvent event){
        super.onKeyDown(keyCode,event);
        Log.i(TAG, "自定义按钮的onKeyDown方法被调用");
        return false;
    }
}
```

```java
//MainActivity.java部分代码
public class MyActivity extends ActionBarActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState){
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_my);
        Button btn = (Button)findViewById(R.id.btn_my);
        btn.setOnKeyListener(new View.OnKeyListener(){
            @Override
            public boolean onKey(View v, int keyCode, KeyEvent event){
                if(event.getAction() == KeyEvent.ACTION_DOWN){
                    Log.i("呵呵","监听器的onKeyDown方法被调用");
                }
                return false;
                }
        });
    }
    @Override
    public boolean onKeyDown(int keyCode, KeyEvent event){
        super.onKeyDown(keyCode, event);
        Log.i("呵呵","Activity的onKeyDown方法被调用");
        return false;
    }
}
```

```
//程序执行结果
"呵呵","监听器的onKeyDown方法被调用”
"自定义按钮的onKeyDown方法被调用“
"呵呵","Activity的onKeyDown方法被调用"
```

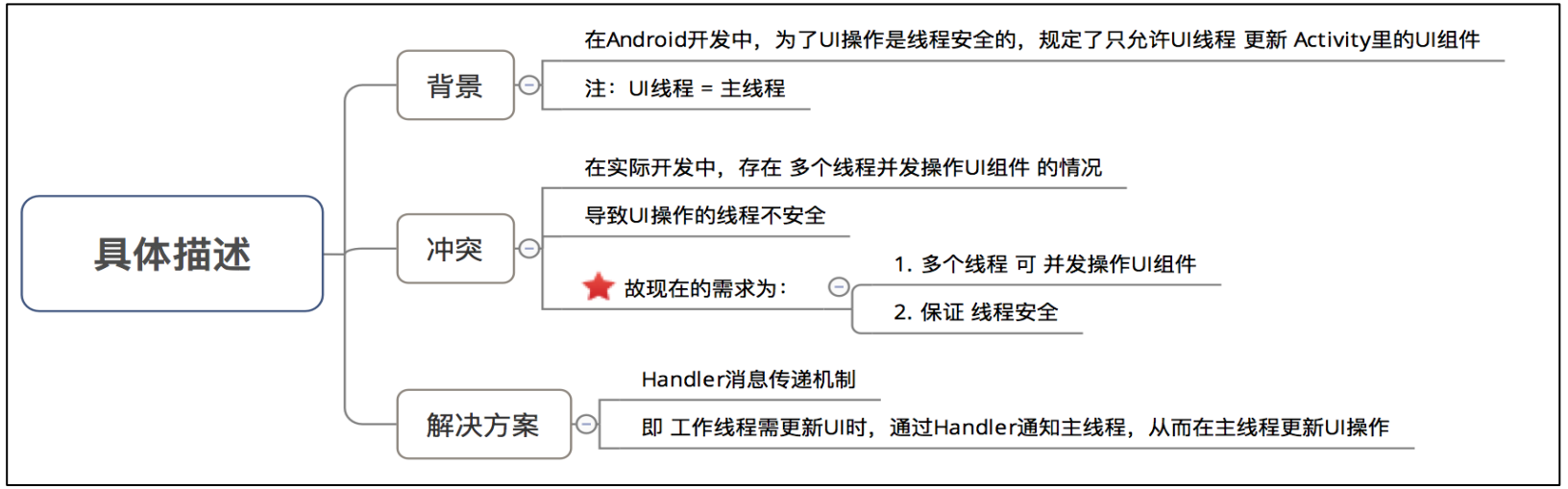
基于Handler的消息传递机制

Handler

- 概念：在多线程的应用场景中，将工作线程中需更新UI的操作信息传递到 UI主线程，从而实现工作线程对UI的更新处理，最终实现异步消息的处理。
- 作用：多个线程并发更新UI的同时保证线程安全。

基于Handler的消息传递机制

- 为什么要使用Handler



Handler将工作线程需操作UI的消息传递到主线程，使得主线程可根据工作线程的需求更新UI，从而避免线程操作不安全的问题

基于Handler的消息传递机制

- Handler相关概念

概念	定义	作用	备注
主线程 (UI线程、Main Thread)	当应用程序第1次启动时，会同时自动开启1条主线程	处理与 UI 相关的事件 (如更新、操作等)	主线程 与 子线程 通信媒介= Handler
子线程 (工作线程)	人为手动开启的线程	执行耗时操作 (如 网络请求、数据加载等)	
消息 (Message)	线程间通讯的数据单元 (即 Handler接受 & 处理的消息对象)	存储需操作的通信信息	/
消息队列 (Message Queue)	一种 数据结构 (存储特点：先进先出)	存储 Handler发送过来的消息（Message）	/
处理者 (Handler)	• 主线程 与 子线程 的通信媒介 • 线程消息的主要处理者	• 添加 消息（Message） 到消息队列（Message Queue） • 处理循环器（Looper）分派过来的消息（Message）	/
循环器 (Looper)	消息队列（Message Queue） 与 处理者（Handler） 的通信媒介	消息循环，即 • 消息获取：循环取出消息队列（Message Queue）的消息（Message） • 消息分发：将取出的消息（Message） 发送给 对应的处理者（Handler）	• 每个线程中只能拥有1个Looper • 1个Looper可 绑定 多个线程的Handler • 即 多个线程可往1个Looper所持有的MessageQueue中发送消息，提供了线程间通信的可能

基于Handler的消息传递机制

- 线程间传递消息

Handler是Android一套消息处理机制，可以让子线程更新主线程UI界面，实现线程间通通信和消息处理。

- 核心概念：
  - ✓ UI 线程：主线程 ActivityThread，系统创建UI线程时，会初始化一个Looper对象、并创建一个与其关联的MessageQueue（一般一个应用程序有一个UI线程，即主线程）。
  - ✓ Message：Handler 发送和处理的消息，由 MessageQueue 管理。
  - ✓ MessageQueue：消息队列，用来存放通过 Handler 发送的消息，按照先进先出执行，内部使用的是单链表的结构。
  - ✓ Handler：负责发送消息和处理消息。
  - ✓ Looper：负责消息循环，循环取出 MessageQueue 里面的 Message，并交给相应的 Handler 进行处理（一个线程只对应一个Looper，一个Looper可以对应多个Handler）。

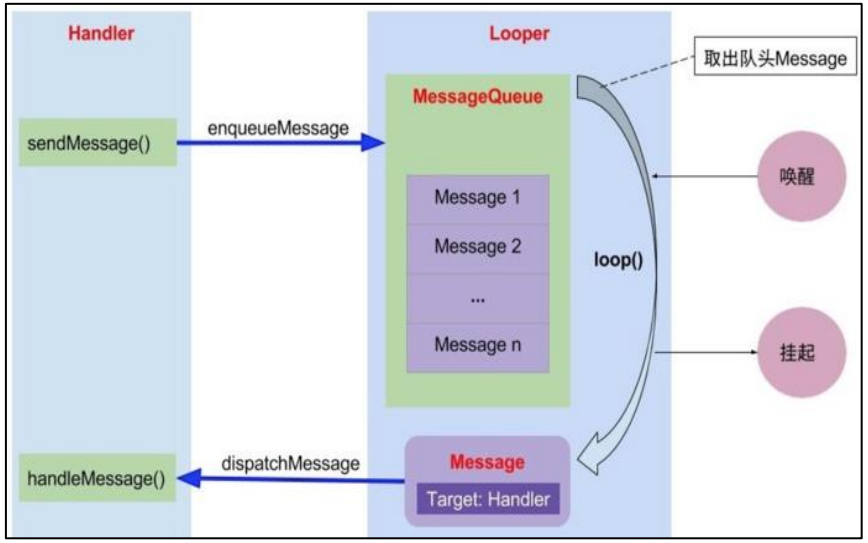
基于Handler的消息传递机制

- Handler工作原理

Handler机制的工作流程主要包括4个步骤，分别是异步通信准备、消息发送、消息循环、消息处理。

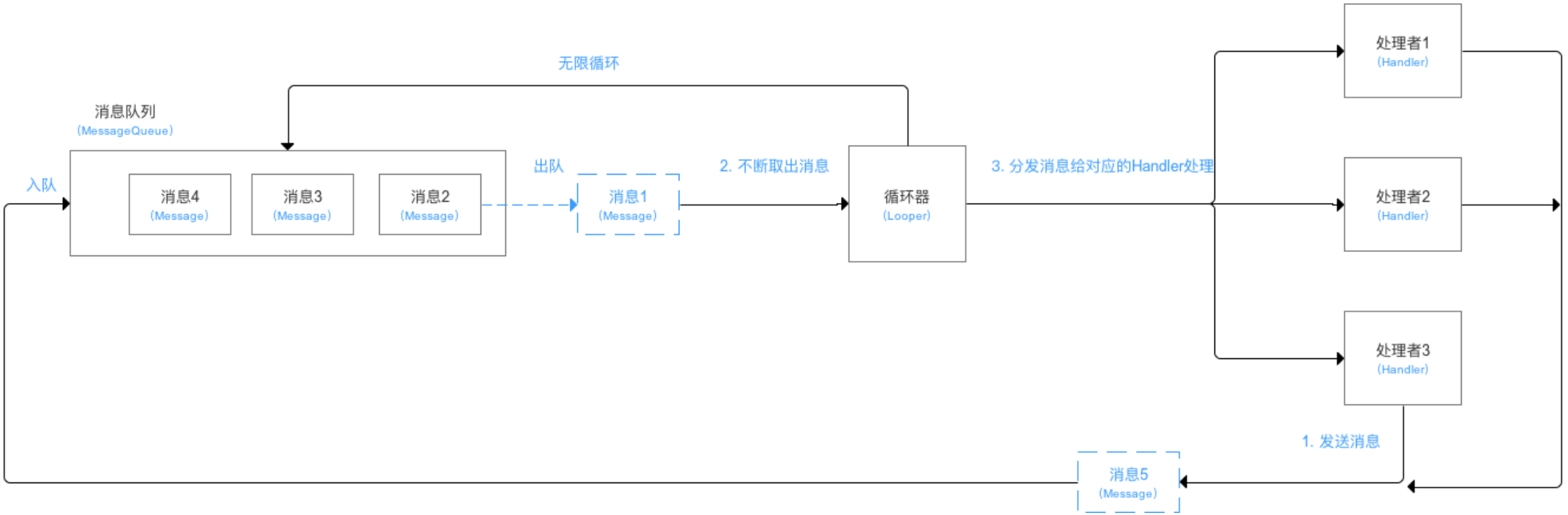
步骤	具体描述	备注
1. 异步通信准备	在主线程中创建： • 处理器 对象（Looper） • 消息队列 对象（Message Queue） • Handler 对象	• Looper、Message Queue均属于主线程 • 创建Message Queue后，Looper则自动进入消息循环 • 此时，Handler自动绑定了主线程的Looper、Message Queue
2. 消息入队	工作线程 通过 Handler 发送消息（Message）到消息队列（Message Queue）中	该消息内容 = 工作线程对UI的操作
3. 消息循环	• 消息出队：Looper循环取出 消息队列（Message Queue） 中的的消息（Message） • 消息分发：Looper将取出的消息（Message） 发送给 创建该消息的处理者(Handler)	在消息循环过程中，若消息队列为空，则线程阻塞
4. 消息处理	• 处理者(Handler）接收 处理器（Looper）发送过来的消息（Message） • 处理者(Handler）根据消息（Message）进行UI操作	

基于Handler的消息传递机制

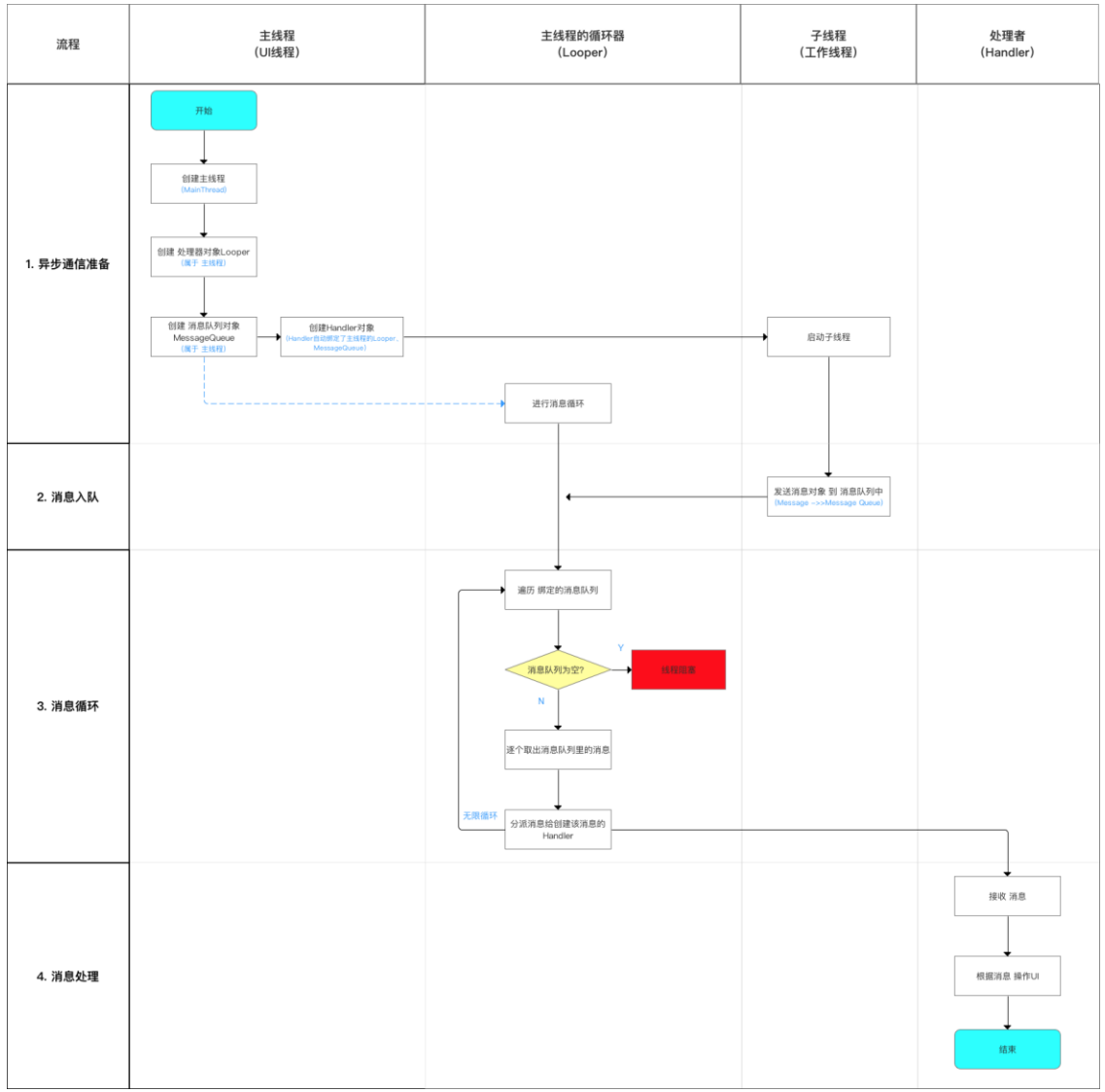


- Handler工作原理
  - 四个对象：Handler, Message, MessageQueue, Looper
  - 两个状态：唤醒和挂起
  - 三个过程：子线程发送消息、消息队列取出消息、主线程接收和处理消息
    - 发送消息：子线程的Handler利用sendMessage方法发送一个消息，系统将该消息加入子线程自己的消息队列MessageQueue中
    - 取出消息：子线程消息队列的Looper遍历消息队列并将队列中的消息分发给主线程的Handler处理
    - 主线程Handler利用handleMessage方法处理该消息，完成了一个消息的发送和处理过程。

- Handler工作示意图



Handler工作流程图



- Handler使用步骤

使用 Handler.sendMessage()实现，继承了Handler类并且创建了子类

方法1：新建Handler子类（内部类）

方法2：匿名 Handler子类

```
//方法1：新建Handler子类（内部类）

    // 步骤1：自定义Handler子类（继承Handler类） & 复写handleMessage（）方法
    class mHandler extends Handler {

        // 通过复写handlerMessage() 从而确定更新UI的操作
        @Override
        public void handleMessage(Message msg) {
         ...// 需执行的UI操作

        }
    }
```

基于Handler的消息传递机制

- 在主活动中创建Handler

```
//MainActivity.java部分代码，主活动创建Handler
//引自《Android基础入门教程》第三章
public class MainActivity extends Activity {
    //定义切换的图片的数组id
    int imgids[] = new int[]{
        R.drawable.s_1, R.drawable.s_2,R.drawable.s_3,
        R.drawable.s_4,R.drawable.s_5,R.drawable.s_6,
        R.drawable.s_7,R.drawable.s_8
    };
    int imgstart = 0;
    final Handler myHandler = new Handler(){
        @Override//重写handleMessage方法,根据msg中what的值判断是否执行后续操作
        public void handleMessage(Message msg) {
            if(msg.what == 0x123){//message.what是用户定义的消息代码，以便收件人识别
                imgchange.setImageResource(imgids[imgstart++ % 8]);
            }
        }
    };
    @Override
    protected void onCreate(Bundle savedInstanceState) {
```

- Handler使用步骤

```
// 方式2：匿名内部类

  // 步骤1：在主线程中 通过匿名内部类 创建Handler类对象
          private Handler mhandler = new  Handler(){
              // 通过复写handlerMessage()从而确定更新UI的操作
              @Override
              public void handleMessage(Message msg) {
                      ...// 需执行的UI操作
                  }
          };

  // 步骤2：创建消息对象
    Message msg = Message.obtain(); // 实例化消息对象
    msg.what = 1; // 消息标识
    msg.obj = "AA"; // 消息内容存放

  // 步骤3：在工作线程中 通过Handler发送消息到消息队列中
  // 多线程可采用AsyncTask、继承Thread类、实现Runnable
   mHandler.sendMessage(msg);
```

- Handler实现实例（更新UI操作）
- 布局代码：*activity_main.xml*

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:gravity="center"

tools:context="com.example.carson_ho.handler_learning.MainActivity">

    <TextView
        android:id="@+id/show"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="" />
</RelativeLayout>
```

- 方法1：新建Handler子类（内部类）

```
public class MainActivity extends AppCompatActivity {

    public TextView mTextView;
    public Handler mHandler;

    // 步骤1：（自定义）新创建Handler子类(继承Handler类) & 复写handleMessage（）方法
    class Mhandler extends Handler {

        // 通过复写handlerMessage() 从而确定更新UI的操作
        @Override
        public void handleMessage(Message msg) {
            // 根据不同线程发送过来的消息，执行不同的UI操作
            // 根据 Message对象的what属性 标识不同的消息
            switch (msg.what) {
                case 1:
                    mTextView.setText("执行了线程1的UI操作");
                    break;
                case 2:
                    mTextView.setText("执行了线程2的UI操作");
                    break;
```

- 方式2：匿名内部类

```
public class MainActivity extends AppCompatActivity {

    public TextView mTextView;
    public Handler mHandler;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        mTextView = (TextView) findViewById(R.id.show);

        // 步骤1：在主线程中 通过匿名内部类 创建Handler类对象
        mHandler = new Handler(){
            // 通过复写handlerMessage()从而确定更新UI的操作
            @Override
            public void handleMessage(Message msg) {
                // 根据不同线程发送过来的消息，执行不同的UI操作
                switch (msg.what) {
                    case 1:
                        mTextView.setText("执行了线程1的UI操作");
```

- 在子活动中创建Handler
  - S1：直接调用Looper.prepare()方法为当前子线程创建Looper对象，它的构造器会自动创建配套的MessageQueue;
  - S2：创建Handler对象，重写handleMessage()方法，用于处理来自其他线程信息
  - S3：调用Looper.loop()方法启动Looper

```
public class MyThread extends Thread {
    private Handler handler;
    @Override
    public void run() {
        super.run();
        // 创建Looper对象并关联到Thread
        Looper.prepare();
        handler = new Handler(Looper.myLooper(), new Handler.Callback() {
            @Override
            public boolean handleMessage(Message msg) {
                return false;
            }
        });
        Looper.loop();
    }
}
```

Looper常用方法

Looper.prepare(); // Looper初始化

Looper.myLooper(); // 获取当前线程中ThreadLocal缓存的Looper对象

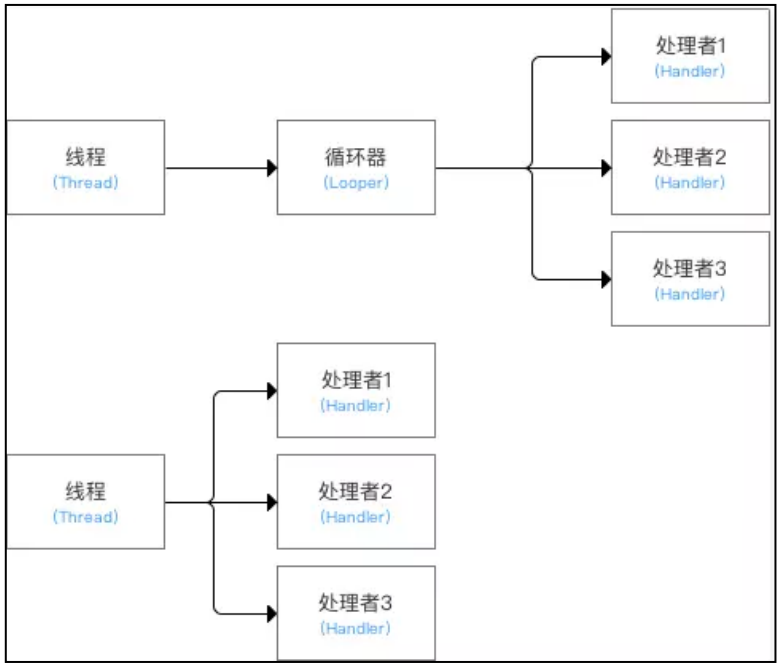
Looper.loop(); // 使当前线程进入循环，从而启动Handler机制

getLooper(); // 获取Handler缓存中的Looper对象

myHandler.getLooper().quit(); // 终止Looper.loop()的死循环

基于Handler的消息传递机制

- 线程(Thread)、循环器(Looper)、处理者(Handler)三者关系
  - 1个线程(Thread)只能绑定1个循环器(Looper)，但可以有多个处理者(Handler)
  - 1个循环器(Looper)可绑定多个处理者(Handler)
  - 1个处理者(Handler)只能绑定1个循环器(Looper)

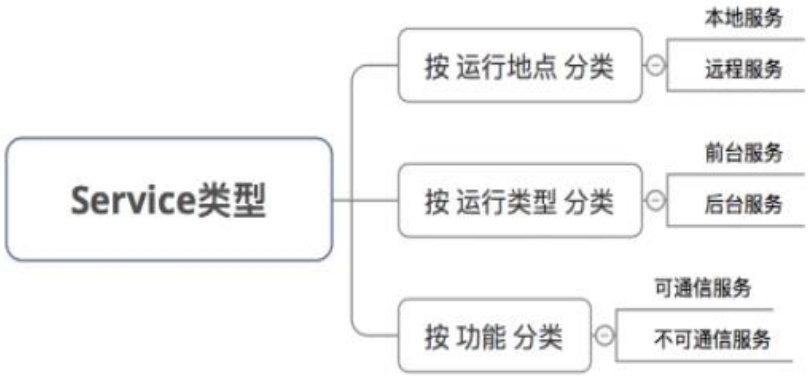


Android四大组件——Service

- 服务组件（Service）

  长期运行在后台、没有用户界面、甚至程序关闭仍可运行。

- 定义：服务，是Android四大组件之一， 属于计算型组件
- 作用：在后台执行一系列计算任务，耗时的后台计算建议在单独的线程中执行
- 特点：无用户界面、在后台运行、生命周期长



注：一般会将 Service 和 Thread联合着用，即在Service中再创建一个子线程（工作线程）去处理耗时操作逻辑

类别	特点	优点	缺点	应用场景
本地服务	• 运行在主线程 • 主进程被终止，服务也被终止	• 节约资源 • 通信方便，在相同进程中通信，不需要IPC和AIDL	限制大：主进程终止后，服务也终止	需要依附某个进程的服务，如音乐播放
远程服务	• 运行在独立进程 • 服务常驻后台，不受其他activity影响	灵活：服务常驻后台，不受其他activity影响	• 消耗资源：单独进程 • 通信不便：使用AIDL进行IPC	系统级别服务（如电源管理、网络管理）

类别	特点	应用场景
前台服务	在通知栏显示通知（用户可见）	• 服务使用时，需要让用户知道并且运行相关操作，如音乐播放服务。 • 服务终止时，通知栏的通知也会消失
后台服务	处于后台的服务（用户不可见）	• 服务使用时，不需要让用户知道并且运行相关操作，如天气更新、位置同步 • 服务被终止时，用户无法知道

类别	特点	应用场景
不可通信的后台服务	• 用startService()启动 • 调用者退出后，service仍存在	服务不需要与activity或者其他service通信
可通信的后台服务	• 调用bindService()启动 • 调用者退出后，service被销毁	服务需要与activity或者其他service通信，需要控制服务的启动时刻 • 节约系统资源：第一次bindService()时才会创建服务的实例并且运行，特别当是远程服务时，该效果更明显 • 服务只是公开一个远程接口，供客服端远程调用服务的执行方法
	• 依次调用StartService()和bindService()启动 • 调用者退出后，service被销毁	服务需要与activity或者其他service通信，不需要控制服务的启动时刻（一开始，服务随activity进程运行而运行）

Android四大组件——Service

- 服务组件（Service）
  - Service不运行在独立进程，而是默认运行在主线程，即它依赖于创建它的应用程序进程
  - 当应用程序进程被杀掉时，所有依赖于该进程的服务都会停止运行
  - Service不能自动开启线程，需要在服务内部手动创建子线程，并在其中执行具体任务。否则，主线程会被阻塞
  - Service分为两种形式：本地服务和远程服务
    - 本地服务：非独立进程，依附于主进程，服务和启动服务的组件在相同进程，节省资源
    - 远程服务：独立进程，不依附于主进程，服务和启动服务的组件在不同进程，为多进程提供服务

服务不能自己启动自己，需要借助其他组件

- 创建服务
  - 创建服务：右击项目，选择“New->Service-> Service”，新建*.java文件
  - 创建活动：在主活动.java文件中添加组件
  - 启动/停止服务：在组件中调用启动服务方法和停止服务方法
  - 注册服务：在manifest.xml文件添加<service>标签

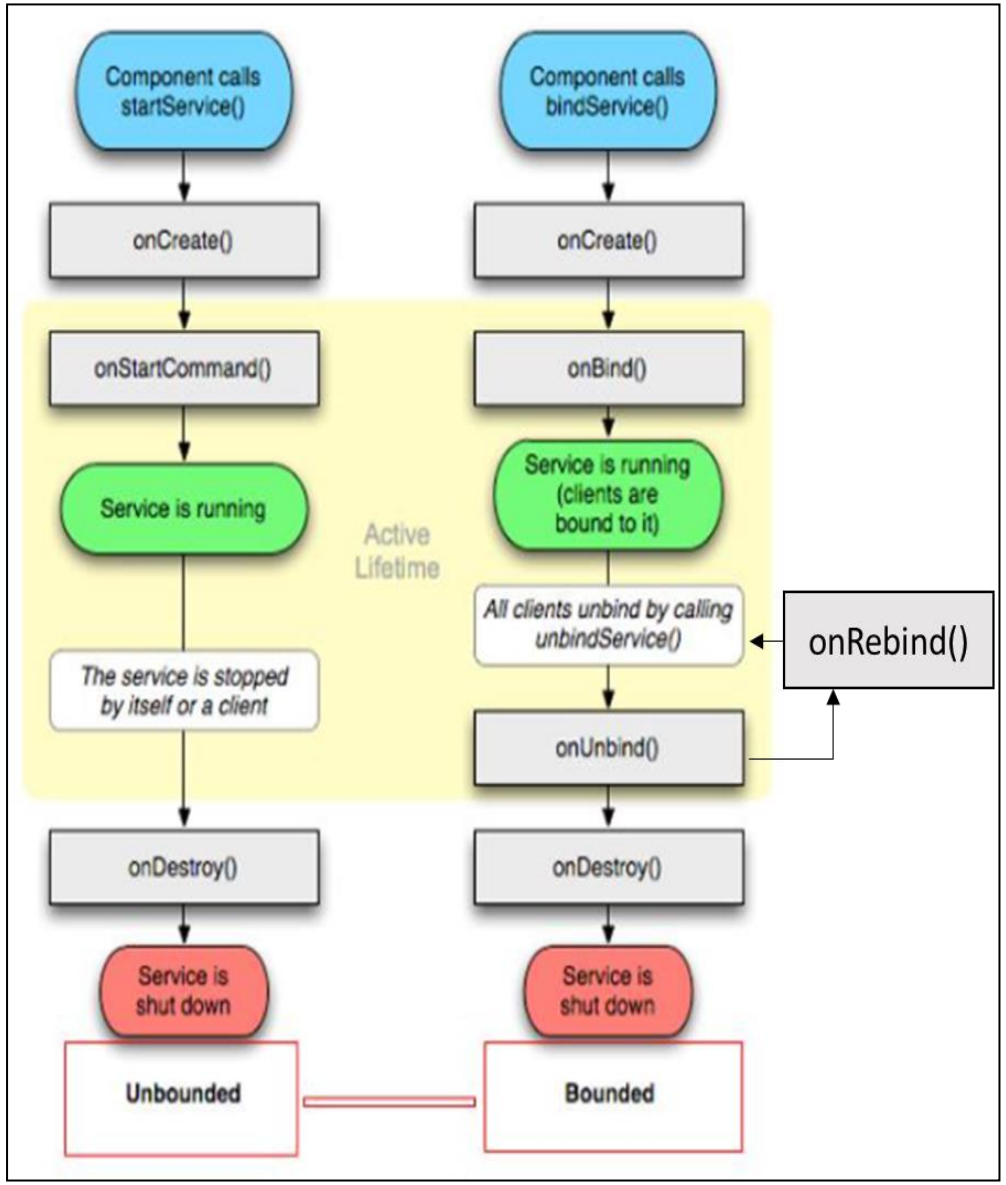
```
//创建服务
public class HelloService extends Service {
    /*当服务被创建时调用*/
    @Override
    public void onCreate(){
    }
    /*调用startService()启动服务时回调*/
    @Override public int onStartCommand(Intent intent,
int flags, int startId){
        return mStartMode;
    }
    /*通过bindService()绑定到服务的客户端*/
    @Override public IBinder onBind(Intent intent){
```

```
//注册服务
<application
    <service
        android:name="com.example.service.MyService"
        android:enabled="true"
        android:exported="true">
    </service>
</application>
```

- 启动服务——两种类型
  - StartService()模式
    - 启动：调用StartService()方法
    - 停止：调用stopService ()方法
    - 状态：服务处于stated状态
    - 特点：当启动服务的组件被销毁，服务仍然运行

      服务和组件在一个进程
  - bindService()模式
    - 启动：绑定bindService()方法
    - 停止：解绑unbindService()方法
    - 状态：服务处于bound状态
    - 特点：绑定服务的组件解绑后，服务被销毁

      多个组件可以绑定一个服务



- 启动服务——所需方法

手动调用方法	作用
startService()	启动服务
stopService()	关闭服务
bindService()	绑定服务
unbindService()	解绑服务

内部自动的回调方法	作用
onCreat()	创建服务
onStartCommand()	开始服务
onDestroy()	销毁服务
onBind()	绑定服务
onUnbind()	解绑服务

注意：

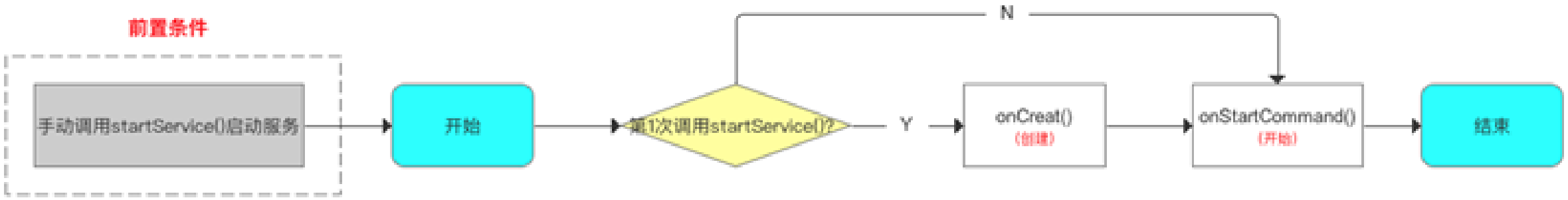
- startService()和stopService()只能开启和关闭服务，但是无法操作服务
- bindService()和unbindService()不仅能绑定和解绑服务，还能操作服务
- startService()开启的服务，调用者退出后，服务仍然存在
- bindService()开启的服务，调用者退出后，服务也被自动消耗

- 服务中的回调方法

回调	描述
onStartCommand()	组件(如活动)通过调用startService()来请求启动服务时，系统回调该方法。一旦执行此方法，服务即会启动并可在后台无限期运行。如果采用该方法启动服务，必须通过stopSelf()或者stopService()方法来停止服务。
onBind()	当组件通过bindService()绑定服务时，系统回调该方法。如果采用该方法启动服务，需要通过返回IBinder对象提供一个接口，实现客户端与服务通信。无论是启动状态还是绑定状态，此方法必须重写，但在启动状态下直接返回 null。
onUnbind()	当客户中断所有服务发布的特殊接口时，系统回调该方法。
onRebind()	当新的客户端与服务连接，且此前它已经通过onUnbind(Intent)通知断开连接时，系统回调用该方法。
onCreate()	首次创建服务时，系统回调方法来执行一次性设置程序（在回调onStartCommand() 或onBind() 之前）。如果服务已在运行，则不会调用此方法，该方法只调用一次
onDestroy()	当服务不再有用或者被销毁时，系统回调该方法。需要销毁的服务通过实现该方法来清理任何资源，如线程、已注册的监听器、接收器等。（执行一次）

- 服务执行过程

方法	作用	回调方法	说明
startService()	启动服务	依次回调onCreat()和onStartCommand()	• 如果一个service被startService()启动多次，那么onCreate()只执行一次 • 整个生命周期中，只有onStartCommand()可以被多次调用，其他回调方法都只能调用1次 • onStartCommand()的调用次数 = startService()的调用次数 • onStartCommand()必须返回一个整数，用于描述系统在销毁服务后应该如何继续运行(START_NOT_STICKY，START_STICKY，START_REDELIVER_ITENT，START_STICKY_COMPATIBILIT)



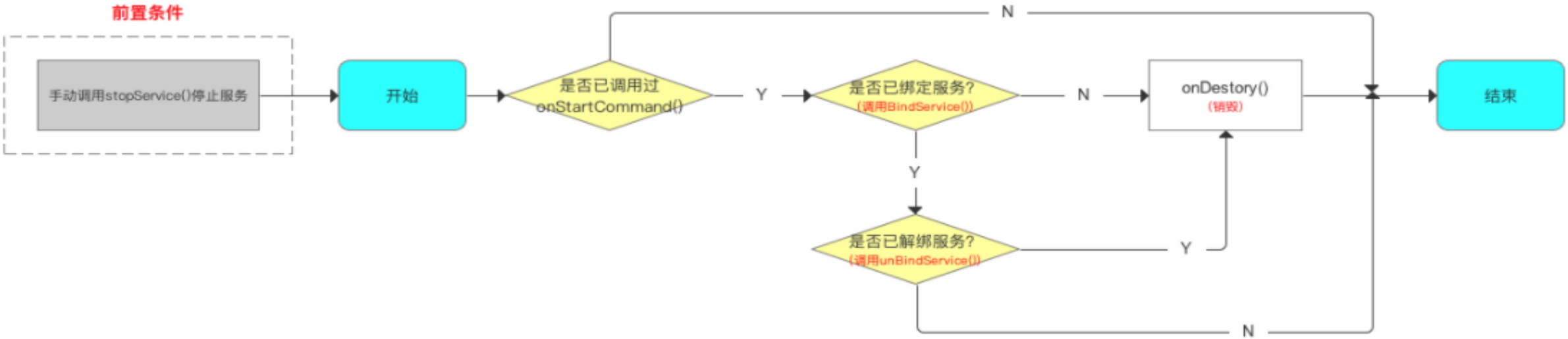
课外知识

onStartCommand()详解

- START_STICKY：如果service进程被kill掉，保留service的状态为开始状态，但不保留递送的intent对象。随后系统会尝试重新创建service，由于服务状态为开始状态，所以创建服务后一定会调用onStartCommand(Intent,int,int)方法。如果在此期间没有任何启动命令被传递到service，那么参数Intent将为null。
- START_NOT_STICKY：“非粘性的”。使用这个返回值时，如果在执行完onStartCommand后，服务被异常kill掉，系统不会自动重启该服务。
- START_REDELIVER_INTENT：重传Intent。使用这个返回值时，如果在执行完
- onStartCommand后，服务被异常kill掉，系统会自动重启该服务，并将Intent的值传入。START_STICKY_COMPATIBILITY：START_STICKY的兼容版本，但不保证服务被kill后一定能重启。

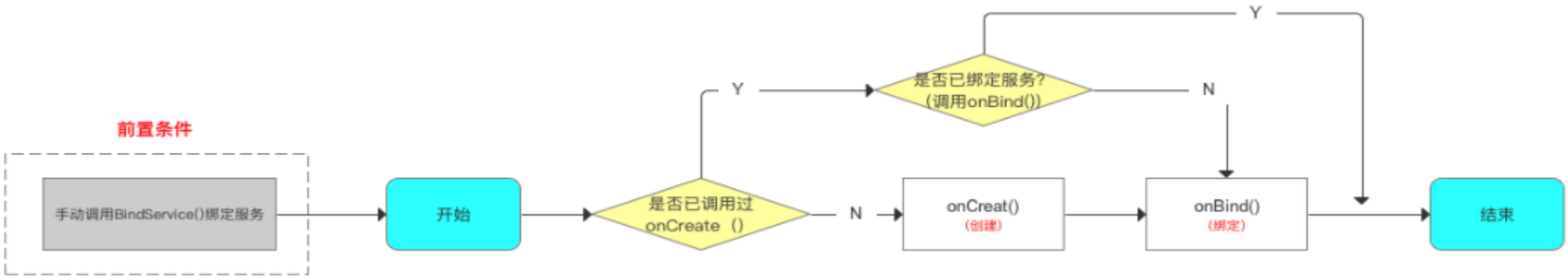
- 服务执行过程

方法	作用	回调方法	说明
stopService()	关闭服务	onDestroy()	• 启动并且绑定一个service实例 • 如果在没有解绑前调用stopService()，无法关闭服务。



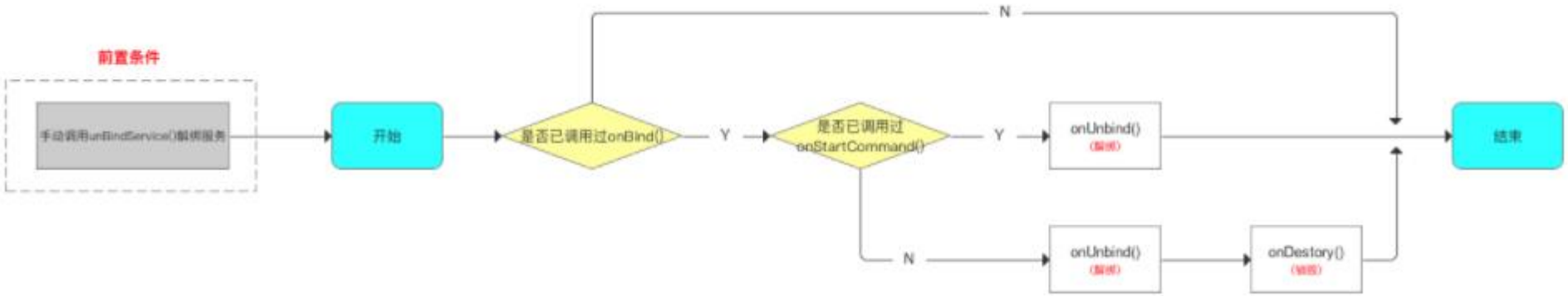
- 服务执行过程

方法	作用	回调方法	说明
bindService()	绑定服务	依次回调onCreat()和onBind()	• 如果已经创建，不再调用onCreate() • 如果已经绑定，不再调用onBind()



- 服务执行过程

方法	作用	回调方法	说明
unbindService()	解绑服务	依次回调onUnbind()和onDestroy()	• 如果通过activity调用onStartCommand()启动的服务，解绑后不被销毁（不回调onDestroy()）



服务常见场景的生命周期

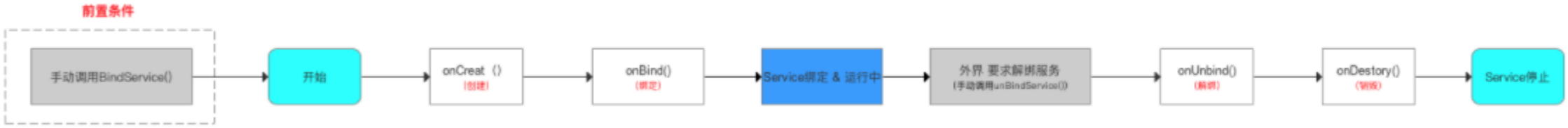
（1）启动服务

- 只能用startService()启动
- 如果一个service被startService()启动多次，那么只回调一次onCreate()，仅生成一个service实例
- 整个生命周期中，只有onStartCommand()可以被多次回调，其他回调方法都只能调用1次
- onStartCommand()的调用次数 = startService()的调用次数

服务常见场景的生命周期

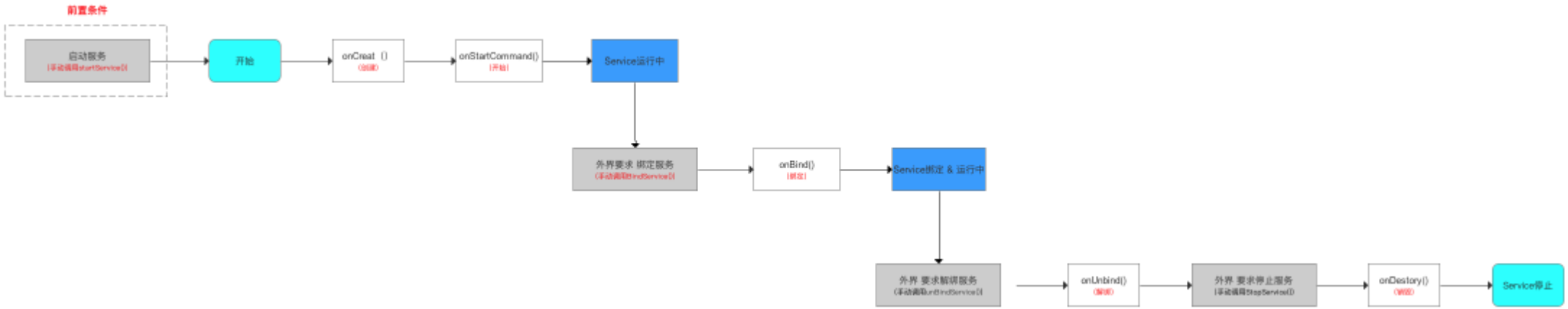
（2）绑定服务

- 只能用BindService()绑定
- 客户端通过一个Ibinder接口与服务器进行通信
- 如果一个service被BindService()启动多次，那么只回调一次onCreate()和onBind()，仅生成一个service实例
- 多个客户端可以被绑定到同一个服务上，当所有客服端都解绑后，该服务被系统自动销毁（不需要手动终止）



服务常见场景的生命周期

（3）先启动服务，再绑定服务

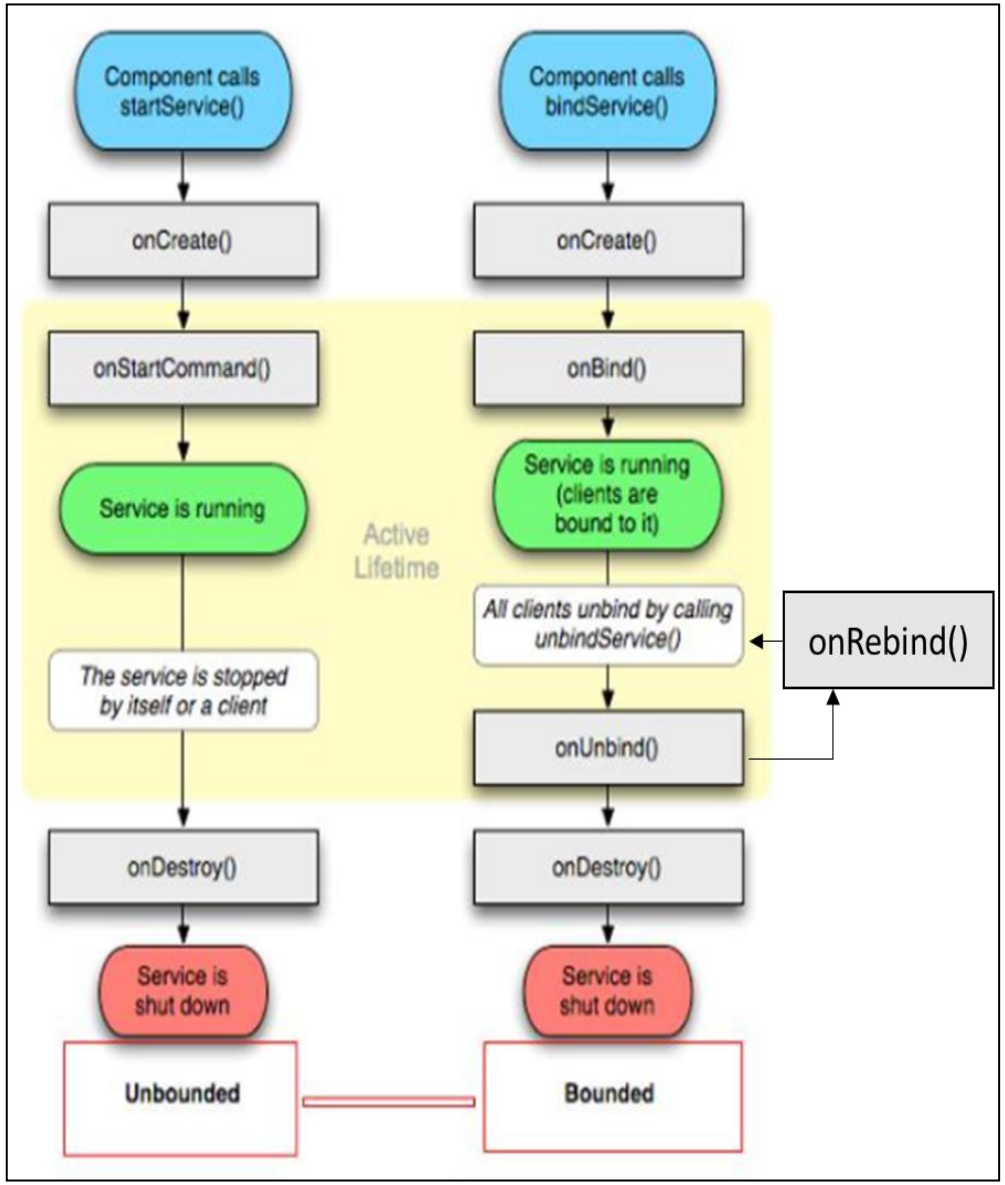


- 两种服务启动模式的区别
  - startService() 模式：
    - 一旦启动，服务即可在后台无限期运行
    - 即使启动服务的组件已被销毁也不受影响，除非手动调用才能停止服务
    - 已启动的服务通常是执行单一操作，而且不会将结果返回给调用方（无通信）
  - bindService() 模式：
    - 被绑定服务提供了一个客户端-服务器接口，允许组件与服务进行交互、发送请求、获取结果
    - 利用进程间通信 (IPC，Inter-Process Communicate) 执行跨进程操作。 仅当与另一个应用组件（如活动）绑定时，绑定服务才会运行。 多个组件可以同时绑定到该服务，但全部取消绑定后，该服务即会被销毁。

- 服务的两个状态

状态	形式	含义
started	本地服务	组件通过调用startService()启动了服务，则服务是started状态。一旦启动，服务可以在后台无限期运行，即使启动它的组件已经被销毁。
bound	本地服务/远程服务	组件通过调用bindService()绑定了服务，则服务是Bound状态。Bound状态的服务提供了一个客户服务器接口来允许组件与服务进行交互，如发送请求，获取结果，甚至通过IPC来进行跨进程通信。

- 两种启动服务方式的执行过程
  - startService()方式：
    - 启动：调用startService()方法，依次回调–>onCreate()–>onStartCommand()
    - 停止：调用stopService()方法，依次回调–>onDestroy()
    - 多次启动：多次调用startService()方法，只执行执行一次onCreate()，自动执行多次onStartCommand()
  - bindService()方式
    - 启动：调用bindService()方法，依次回调–>onCreate() –>onBind()
    - 停止：调用unbindService()方法，依次回调–>onUnbind() –>onDestroy()
    - 多次启动：回调一次onCreate()，回调一次onBind()



【例】startService()方法：连续单击三次开始服务，一次停止服务，logcat输出什么，依次调用了哪些方法

单击一次开始按钮
- 回调onCreate()方法
- 回调onStartCommand()方法

再连续单击两次开始按钮
- 回调onStartCommand()方法
- 回调onStartCommand()方法

单击一次停止按钮
- 回调onDestory()方法

本地服务案例

- 步骤1：新建子类继承Service类，需重写父类的onCreate()、onStartCommand()、onDestroy()和onBind()

```
public class MyService extends Service {
//启动Service之后，就可以在onCreate()或onStartCommand()方法里去执行一些具体的逻辑
//由于这里作Demo用，所以只打印一些语句
    @Override
    public void onCreate() {
        super.onCreate();
        System.out.println("执行了onCreat()");
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        System.out.println("执行了onStartCommand()");
        return super.onStartCommand(intent, flags, startId);
    }

    @Override
    public void onDestroy() {
```

- 步骤2：在主布局文件设置两个Button分别用于启动和停止Service

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context="scut.carson_ho.demo_service.MainActivity">

    <Button
        android:layout_centerInParent="true"
        android:id="@+id/startService"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="启动服务" />

    <Button
        android:layout_centerInParent="true"
        android:layout_below="@+id/startService"
```

- 步骤3：构建Intent对象，并调用startService()启动Service、stopService停止服务

```
public class MainActivity extends AppCompatActivity implements View.OnClickListener
{
    private Button startService;
    private Button stopService;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        startService = (Button) findViewById(R.id.startService);
        stopService = (Button) findViewById(R.id.stopService);

        startService.setOnClickListener(this);
        startService.setOnClickListener(this);
    }

    @Override
    public void onClick(View v) {
        switch (v.getId()) {
```

思考：

- 两个组件间如何传递数据?
- 为什么要用intent间接传递服务组件?

- 步骤4：在AndroidManifest.xml里注册Service

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="scut.carson_ho.demo_service">

    <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>

        //注册Service服务
        <service android:name=".MyService">
```

扩展知识

Service常见属性

表 Androidmanifest里Service的常见属性说明

属性	说明	备注
android:name	Service的类名	
android:label	Service的名字	若不设置，默认为Service类名
android:icon	Service的图标	
android:permission	申明此Service的权限	有提供了该权限的应用才能控制或连接此服务
android:process	表示该服务是否在另一个进程中运行（远程服务）	不设置默认为本地服务；remote则设置成远程服务
android:enabled	系统默认启动	true：Service 将会默认被系统启动；不设置则默认为false
android:exported	该服务是否能够被其他应用程序所控制或连接	不设置默认此项为 false

- StartService()方法小结
- 涉及方法：回调方法包括onCreate()、onStartCommand()、onDestory()
  普通方法包括startService()、stopService()
- 执行过程：1）首次调用StartService()方法，创建Service对象，依次回调onCreate()和onStartCommand()方法；
  2）后续再调用StartService()方法，不创建Service对象，只回调onStartCommand()方法
  3）建立的Service对象与调用组件无关，关闭调用组件Service仍运行
  4）调用StopService()方法，关闭Service对象，同时回调onDestory()方法

回调方法和普通方法的区别：系统自动调用和手动调用

- Service与Activity之间的通信
  - 通信类型：本地通信、远程通信
  - 通信前提：必须保证服务是以绑定的形式开启的，否则无法进行通信和数据交换
  - 通信载体：IBinder对象
  - 通信方法：调用bindService()方法和依次回调onBind()方法、onServiceConnected()方法

bindService(Intent service, ServiceConnection conn, int flags)

- service：通过Intent指定要启动的服务
- coon：监听访问者与服务的连接情况，如果连接成功，则回调onServiceConnected()方法；如果连接断开，则回调onServiceDisconnected()方法
- flags：指定绑定时是否自动创建服务(前提是该服务还未创建)。如果是0，则不自动创建；如果是BIND_AUTO_CREATE，则自动创建

onServiceConnected(ComponentName name, IBinder service)

- ComponentName：已连接服务的具体组件名称
- IBinder：通信接口，实现与绑定服务间的通信

Binder和IBinder的关系

- IBinder是接口，必须自己写代码逻辑实现功能
- Binder是IBinder的具体实现类，继承Binder的类就是IBinder对象

通信服务案例

- 步骤1：在新建子类继承Service类，并新建一个子类继承自Binder类、写入与Activity关联需要的方法、创建实例

```
public class MyService extends Service {

    private MyBinder mBinder = new MyBinder();

    @Override
    public void onCreate() {
        super.onCreate();
        System.out.println("执行了onCreat()");
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        System.out.println("执行了onStartCommand()");
        return super.onStartCommand(intent, flags, startId);
    }
```

- 步骤2：在主布局文件再设置两个Button分别用于绑定和解绑Service

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context="scut.carson_ho.demo_service.MainActivity">

    <Button
        android:layout_centerInParent="true"
        android:id="@+id/startService"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="启动服务" />

    <Button
        android:layout_centerInParent="true"
        android:layout_below="@+id/startService"
```

- 步骤3：在Activity中调用MyBinder类中的public方法，实现Activity与Service的联系

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

    private Button startService;
    private Button stopService;
    private Button bindService;
    private Button unbindService;

    private MyService.MyBinder myBinder;

    //创建ServiceConnection的匿名类
    private ServiceConnection connection = new ServiceConnection() {
        //重写onServiceConnected()方法和onServiceDisconnected()方法
        //在Activity与Service建立关联和解除关联的时候调用
        @Override
        public void onServiceDisconnected(ComponentName name) {
        }

        //在Activity与Service解除关联的时候调用
        @Override
```

```java
//在activity.java中实现
private BindService.MyBinder binder;
//回调ServiceConnection()方法
private ServiceConnection connection = new ServiceConnection(){
    @Override
    public void onServiceConnected(ComponentName name, IBinder service){
        System.out.println("连接服务")
        binder=(BindService.MyBinder)service;
        binder.startComm();//调用服务中的自定义方法
    }
    @Override
    public void onServiceDisconnected(ComponentName name){
        System.out.println("断开服务")
    }
};
//绑定服务
Intent bindIntent=new Intent(this,MyService.class);
bindService(bindIntent,connection,BIND_AUTO_CREATE);
//解绑服务
unbindService(connection);
```

连接成功，返回Ibinder对象

```java
//自定义BindService.java
public class BindService extends Service {
    private MyBinder mBinder = new MyBinder();
    private final String TAG = "TestService";
    //通过继承Binder类实现Ibinder对象
    public class MyBinder extends Binder{
        public void startComm(){
            Log.d(TAG, "startComm方法被调用");
        }
    }
    //必须要实现的方法
    @Override
    public IBinder onBind(Intent intent) {
        Log.i(TAG, "onBind方法被调用!");
        return mBinder;
    }
    //Service被创建时调用
    @Override
    public void onCreate() {
        Log.i(TAG, "onCreate方法被调用!");
        super.onCreate();
    }
    //Service被启动时调用
    @Override
    public int onStartCommand(Intent intent, in
        Log.i(TAG, "onStartCommand方法被调用!"
        return super.onStartCommand(intent, fla
```

- BindService()方法实现服务通信
  - 本地服务通信：客户端通过Ibinder对象与服务器端service通信
    - 在Service的*.java文件中，自定义方法提供给客户端使用
    - 在Activity的*.java文件中，调用服务器端Service中自定义的各种方法
      - 创建Binder子类，让客户端能够通过Binder类实现对服务器端service的调用
        - ✓ 客户端调用bindService()方法，回调服务器端服务的onCreate()方法和onBind()方法，返回IBinder对象
      - 在客户端的onServiceConnected()方法中接收返回的Ibinder对象
        - ✓ 如果服务连接成功，则回调onServiceConnected()方法，获取服务返回的IBinder对象
      - 客户端利用IBinder对象调用服务中自定义的各方法

- startService()方法与BindService()方法
  - 利用intent和service组件或其他组件间的传递
    - startService(intent)
    - bindService(intent, conn, Service.BIND_AUTO_CREATE);
  - startService()中的Service组件和其他组件无关
  - BindService()中的Service组件和其他组件相关，并且支持他们之间的通信

前台服务案例

只需在原有的Service类对onCreate()方法进行稍微修改即可

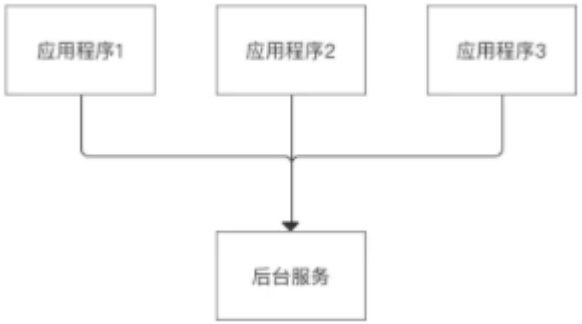
```
@Override
    public void onCreate() {
        super.onCreate();
        System.out.println("执行了onCreat()");

        //添加下列代码将后台Service变成前台Service
        //构建"点击通知后打开MainActivity"的Intent对象
        Intent notificationIntent = new Intent(this,MainActivity.class);
        PendingIntent pendingIntent = PendingIntent.getActivity(this,0,notificationIntent,0);

        //新建Builer对象
        Notification.Builder builer = new Notification.Builder(this);
        builer.setContentTitle("前台服务通知的标题");//设置通知的标题
        builer.setContentText("前台服务通知的内容");//设置通知的内容
        builer.setSmallIcon(R.mipmap.ic_launcher);//设置通知的图标
        builer.setContentIntent(pendingIntent);//设置点击通知后的操作

        Notification notification = builer.getNotification();//将Builder对象转变成普通的notification
        startForeground(1, notification);//让Service变成前台Service,并在系统的状态栏显示出来

    }
```

前台服务在下拉通知栏有显示通知，但后台服务没有

- 远程通信

远程服务通信时通过AIDL（Android Interface Definition Language）实现接口定义语言。

- 产生原因：Android系统中每个应用程序都运行在自己的进程中。如果想要完成不同进程（不同应用应用程序）间的通信，需要利用远程服务通信。
- 应用场景：多个应用程序共享同一个后台服务



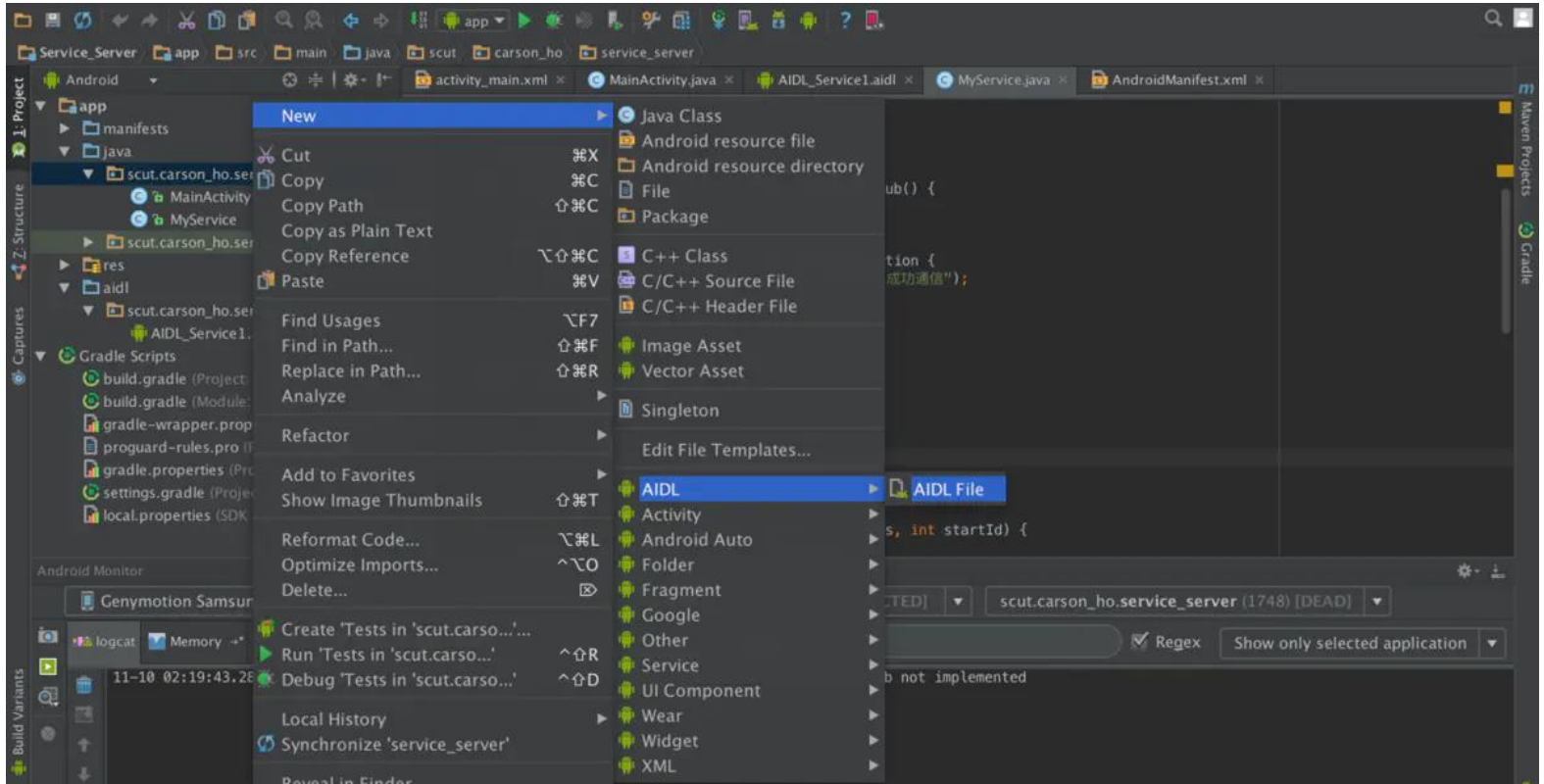
- AIDL使用方法
  - 满足要求：
    - ✓ AIDL定义接口的源代码必须以.aidl结尾。
    - ✓ AIDL接口中用到的数据类型，除了基本数据类型及String、List、Map、CharSequence之外，其他类型全部都需要导入到包，即使它们在同一个包中。
    - ✓ Service服务端和client客户端都需要使用Android SDK安装目录下的platform-tools子目录下的aidl.exe工具为AIDL接口提供实现（ADT工具会自动为AIDL接口生成实现）

定义好上面的 AIDL 接口之后，ADT 工具会自动在 gen/org/crazyit/service 目录下生成一个 ICat.java 接口，在该接口里包含一个 Stub 内部类，该内部类实现了 IBinder、ICat 两个接口，这个 Stub 类将会作为远程 Service 的回调类——它实现了 IBinder 接口，因此可作为 Service 的 onBind()方法的返回值。

- 服务器端的实现

新建一个服务器端的工程： Service-server

- 步骤1. 新建一个AIDL文件（文件名为AIDL_Service1.aidl” ）



参考资料：
https://www.jianshu.com/p/34326751b2c6

- 步骤2. 在新建AIDL文件里定义Service需要与Activity进行通信方法（类似JAVA接口定义），并进行编译

```
//在新建的AIDL_Service1.aidl里声明需要与Activity进行通信的方法
package scut.carson_ho.demo_service;
interface AIDL_Service1 {
    void AIDL_Service();
}
```

- 步骤3：在Service子类中实现AIDL中定义的接口方法，并定义生命周期的方法（onCreat、onBind()等）

```
//在MyService.java中实现
public class MyService extends Service {
    // 实例化AIDL的Stub类(Binder的子类)
    AIDL_Service1.Stub mBinder = new AIDL_Service1.Stub() {
        //重写接口里定义的方法
        @Override
        public void AIDL_Service() throws RemoteException {
            System.out.println("客户端通过AIDL与远程后台成功通信");
        }
    };
    //重写与Service生命周期的相关方法
    @Override
    public void onCreate() {
        super.onCreate();
        System.out.println("执行了onCreat()");
    }
    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
```

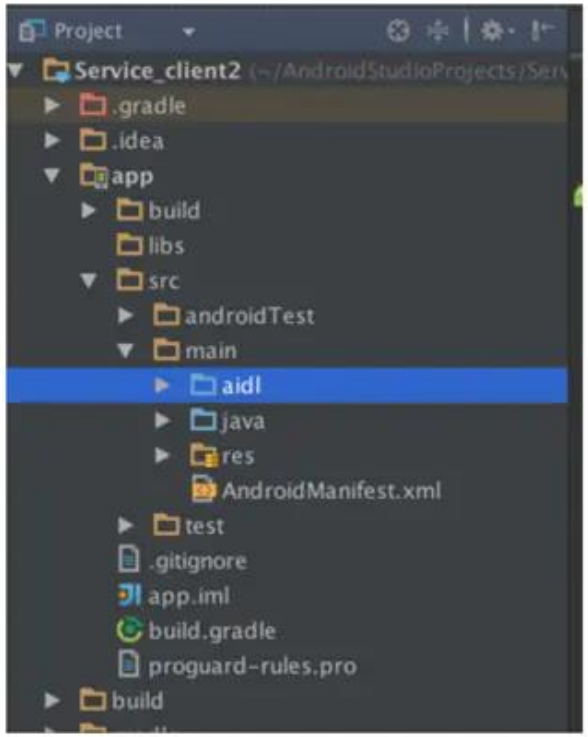
- 步骤4：在AndroidMainfest.xml中注册服务，同时声明为远程服务
  - 如有要在服务中注册Intent的action，则必须写成“服务器端包名.aidl文件名“

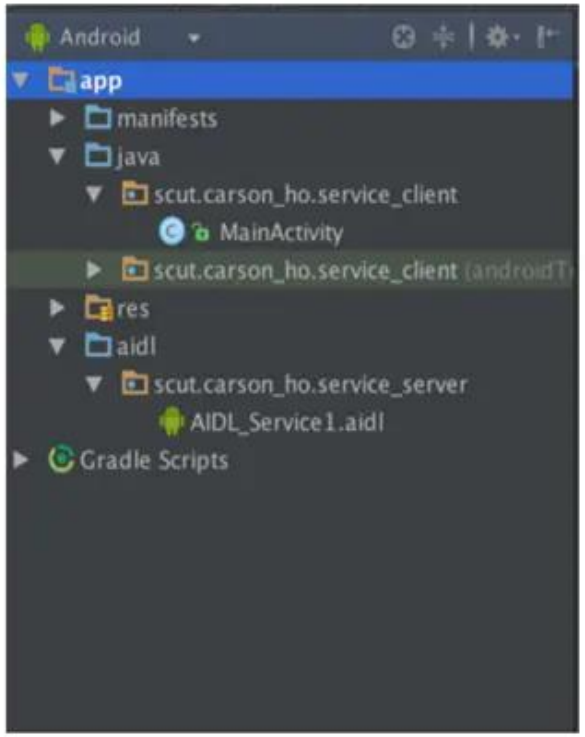
```
//在AndroidMainfest.xml中实现
<service android:name=".MyService"
    android:process=":remote" //将本地服务设置成远程服务
    android:exported="true" //设置可被其他进程调用
    //该Service可以响应带有scut.carson_ho.service_server.AIDL_Service1这个action的Intent。
    //此处Intent的action必须写成“服务器端包名.aidl文件名”
    <intent-filter>
        <action android:name="scut.carson_ho.service_server.AIDL_Service1"/>
    </intent-filter>
</service>
```

- 客户器端的实现

  新建一个客户端的工程：Service-Client

- 步骤1：将服务端的AIDL文件所在的包复制到客户端目录下（Project/app/src/main），并进行编译（Make Project）





- 步骤2：在主布局文件定义“绑定服务”的按钮

```
//在MainActivity.xml中实现
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context="scut.carson_ho.service_client.MainActivity">

    <Button>
        android:layout_centerInParent="true" android:id="@+id/bind_service"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="绑定服务"
    <Buttion/>
</RelativeLayout>
```

- 步骤3：在MainActivity.java中实现如下内容
  - ✓ 使用Stub.asInterface接口获取服务器的Binder；
  - ✓ 通过Intent指定服务端的服务名称和所在包，进行Service绑定；
  - ✓ 根据需要调用服务提供的接口方法。

```
public class MainActivity extends AppCompatActivity {
    private Button bindService;
    private AIDL_Service1 mAIDL_Service; //定义aidl接口变量
    //创建ServiceConnection的匿名类
    private ServiceConnection connection = new ServiceConnection() {
        //重写onServiceConnected()方法和onServiceDisconnected()方法
        //在Activity与Service建立关联和解除关联的时候调用
        //在Activity与Service建立关联时调用
        @Override
        public void onServiceConnected(ComponentName name, IBinder service) {
            //使用AIDLService1.Stub.asInterface()方法获取服务器端返回的IBinder对象
            //将IBinder对象传换成了mAIDL_Service接口对象
            mAIDL_Service = AIDL_Service1.Stub.asInterface(service);
```

- 测试结果

```
11-10 03:51:57.410 1414-1414/scut.carson_ho.service_client I/System.out: 点击了[绑定服务]按钮
11-10 03:51:57.470 1695-1695/scut.carson_ho.service_server:remote I/System.out: 执行了onCreat()
11-10 03:51:57.470 1695-1695/scut.carson_ho.service_server:remote I/System.out: 执行了onBind()
11-10 03:51:57.470 1695-1706/scut.carson_ho.service_server:remote I/System.out: 客户端通过AIDL与远程后台成功通信
```

- 跨App和同App之间的区别
- ①跨App开启服务是提供服务的App需要设置intent-filter过滤器，控制服务的App需要通过setAction和setPackage方法进行设置action和包名，才能开启服务。而同App只需要指定启动的service就可。
- ②跨App的MyBinder实例要通过AIDL获取，两个应用定义同样的接口的方法，通过对应的AIDL名称.Stub.asInterface方法得到binder实例，然后就和同App的myBinder使用么有区别了。
- ③跨App的MyBinder对象的使用必须捕获异常，而同App不需要。

- Stub

Stub类继承了Binder，因此Stub自己是一个Binder本地对象，然后实现了IMyAidlInterface接口，IMyAidlInterface本身是一个IInterface，因此他携带某种客户端需要的能力（这里是方法add)。此类有一个内部类Proxy，也就是Binder代理对象

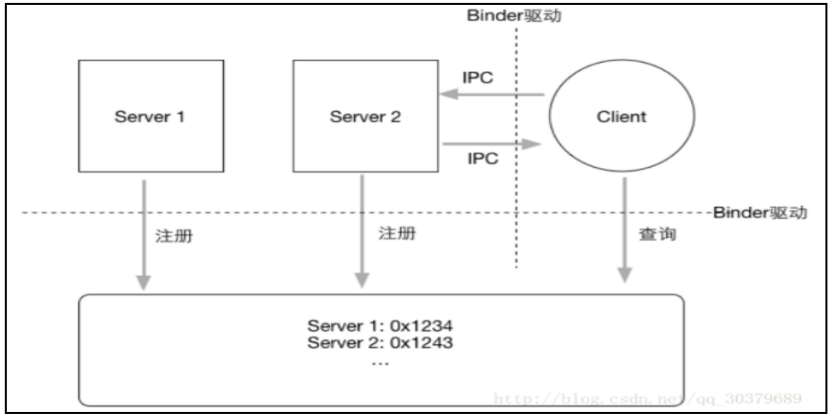
https://blog.csdn.net/qq_30379689/article/details/79451596

类别	特点	优点	缺点	应用场景
本地服务（LocalService）	1. 运行在主线程 2. 主进程被终止后，服务也会终止。	1. 节约资源； 2. 通信方便：由于在同一进程因此不需要IPC和AIDL。	限制性大：主进程被终止后，服务也会终止。	需要依附某个进程的服务，如音乐播放
远程服务（RemoteService）	1. 运行在独立进程； 2. 服务常驻在后台，不受其他Activity影响	灵活：服务常驻在后台，不受其他Activity影响	1. 消耗资源：单独进程 2. 使用AIDL进行IPC复杂	系统级别服务

- Binder通信模式

Binder通信采用的是client-server通信结构，client与server的通信由Binder驱动程序和Service Manager组件协助

- Client进程：跨进程通讯的客户端（运行在某个进程）
- Server进程：跨进程通讯的服务端（运行在某个进程）
- Binder驱动：跨进程通讯的介质
- ServiceManager：跨进程通讯中提供服务的注册和查询（运行在System进程）



Binder模型通信流程

- Server端通过Binder驱动在ServiceManager中注册
- Client端通过Binder驱动获取ServiceManager中注册的Server端
- Client端通过Binder驱动和Server端进行通讯

- Binder通信过程

1.client通过获得一个server的代理接口，对server进行直接调用

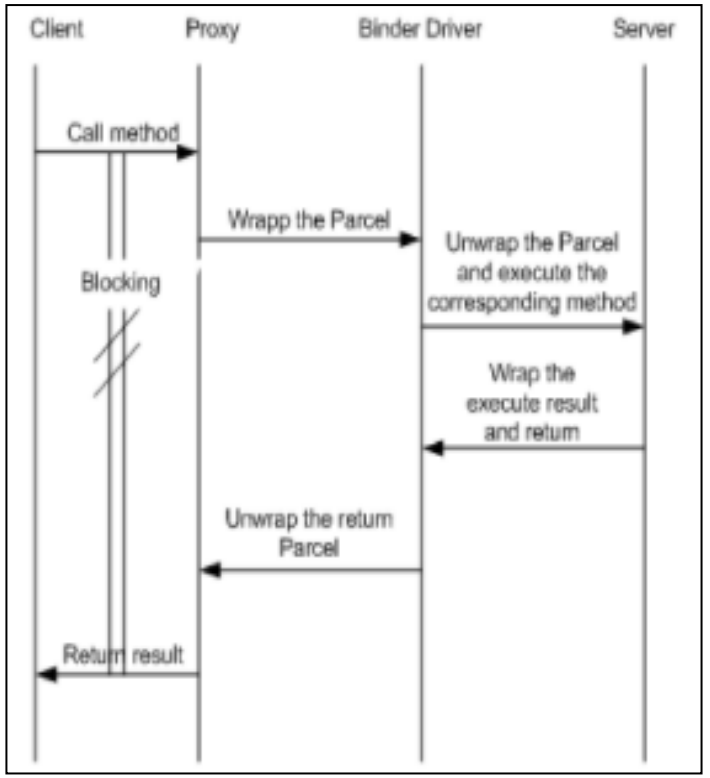
2.代理接口中定义的方法与server中定义的方法一一对应

3.client调用某个代理接口中的方法时，代理接口的方法会将client传递的参数打包成为Parcel对象

4.代理接口将该Parcel发送给内核中的binder driver

5.server会读取binder driver中的请求数据，如果是发送给自己的，解包Parcel对象，处理并返回结果

6.Binder调用过程是同步过程，当server处理时，client会被block



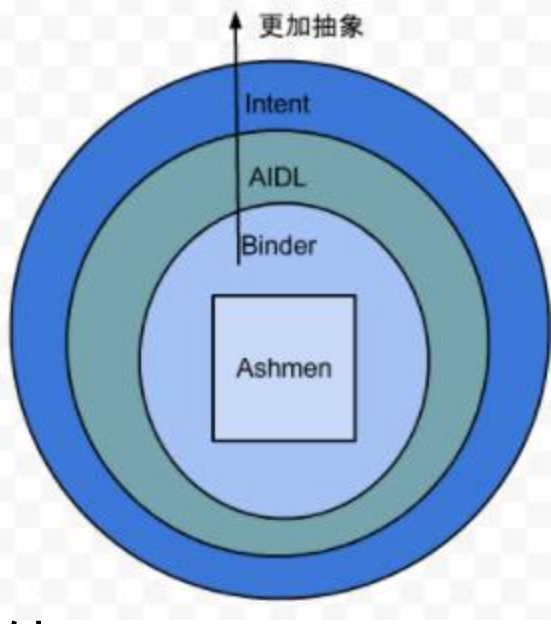
- AIDL使用方法

  AIDL是为了编程方便而设计的，使得开发者尽可能关注代码实现。

  - 只要按照规范写一个.aidl文件，AIDL插件会自动创建一个与interface同名的.java文件(.java文件提供标准化模板，编程者只需在自动生成的框架中填写必要实现)
  - .aidl文件的写法和JAVA接口类似
  - 自动生成的.java中有一个重要的内部类Stub，它继承了Binder类，需要实现预先定义的interface

```
//.aidl文件
interface MyAidl{
    int getId();
    void setId(int id);

  }
```

```
public static abstract class Stub extends android.os.Binder implements com.xxx.MyAi
     public static com.xxx.MyAidl asInterface(android.os.IBinder obj);
     public boolean onTransact(int code, android.os.Parcel data,android.os.Parcel r

     private static class Proxy implements com.hsae.recorder.encoder.Mp4EncoderAidl
             ......
      }
  }
```



- Binder通信概念
  - IPC是一种概念，即进程间通信；
  - Binder是对IPC的具体实行，是IPC的一种具体实现；
  - AIDL是Binder机制向外提供的接口，目的是为了方便对Binder的使用；
  - Intent是最高层级的封装，实质是封装了对Binder的使用，当然Intent也常常在同一进程中调用，只是把两种方式封装在一起了。

参考资料：https://blog.csdn.net/jinxinliu1/article/details/70174591

- 为什么用Binder实现跨进程通信

Linux采用管道、消息队列、信号量、内存共享、套接字等实现跨进程通信，Android采用Binder实现跨进程通信。

- 传输性好
  - ✓ Socket：是一个通用接口，导致其传输效率低，开销大
  - ✓ 共享内存：虽然在传输时不需要拷贝数据，但其控制机制复杂
  - ✓ Binder：复杂数据类型传递可以复用内存
  - ✓ 管道和消息队列：采用存储转发方式，效率低
- 安全性高
  - ✓ 传统的进程：通信方式对于通信双方的身份并没有做出严格的验证，只有在上层协议上进行架设
  - ✓ Binder机制：从协议本身就支持对通信双方做身份校检，因而大大提升了安全性

参考资料：https://blog.csdn.net/qq_30379689/article/details/79451596