**第1章 Activity的生命周期和启动模式**

1.1 Activity的生命周期分析

Activity的生命周期：

1. onCreate：Activity正在被创建，整个Activity生命周期的第一个调用。
2. onRestart：Activity正在重新启动，一般是在Activity从不可见重新变为可见状态。
3. onStart：Activity正在启动，此时Activity已经可见，但是没有出现在前台，还不可以和用户交互。
4. onResume：Activity已经可见，并且在前台活跃。
5. onPause：Activity正在停止，一般紧接着会调用onStop。
6. onStop：Activity即将停止，可以做一些回收动作，但不能太耗时。
7. onDestroy：Activity即将被销毁，Activity生命周期的最后一个调用。

Activity启动

onCreate

用户返回原Activity

onResume

onStart

Activity运行

onRestart

应用被杀死

用户返回原Activity

高优先级应用需要内存

onPause

新Activity启动

用户返回原Activity

Activity已经不可见

onStop

Activity正在停止或者即将被销毁

onDestroy

Activity销毁

对一个Activity

首次启动，调用onCreate -> onStart -> onResume。

打开新的Activity或者切换回桌面时，调用onPause -> onStop。

再次回到原Activity时，调用onRestart -> onStart -> onResume。

当按back键返回launcher时，调用onPause -> onStop -> onDestroy。

**异常情况时的生命周期**

**资源相关的系统配置发生改变**

比如手机旋转屏幕，一般情况下Activity会被销毁并重新创建。

这种情况下，在onDestory之前系统会调用onSaveInstanceState，在重新创建Activity时，完成onCreate之后调用onRestoreInstanceState。

**内存不足时低优先级的Activity会被杀死**

系统按照：后台Activity、可见但非前台Activity、前台Activity的顺序杀死Activity的进程，杀死进程时，会调用onSaveInstanceState来保存状态。

1.2 Activity的启动模式

**Activity的LaunchMode**

1. standard：标准模式。这种模式下的Acvitivy可以有多个实例存在于不同的任务栈中。这个模式时，尝试使用ApplicationContext去启动就会出错，因为ApplicationContext并没有自己的任务栈。
2. singleTop：栈顶复用模式。如果该Activity已经处于栈顶，则不会重复创建，仅仅调用onNewIntent。例如栈中ABCD，此时再次启动D，singleTop模式则仍然是ABCD，如果是standard模式，则变成ABCDD。
3. singleTask：栈内复用模式。单实例模式，只要栈中出现过该Activity，都不会重复创建，仅仅调用onNewIntent。
4. singleInstance：单例模式，加强的singleTask模式。

The "standard" and "singleTop" modes differ from each other in just one respect: Every time there's a new intent for a "standard" activity, a new instance of the class is created to respond to that intent. Each instance handles a single intent. Similarly, a new instance of a "singleTop" activity may also be created to handle a new intent. However, if the target task already has an existing instance of the activity at the top of its stack, that instance will receive the new intent (in an [onNewIntent()](http://developer.android.com/reference/android/app/Activity.html#onNewIntent%28android.content.Intent%29) call); a new instance is not created. In other circumstances — for example, if an existing instance of the "singleTop" activity is in the target task, but not at the top of the stack, or if it's at the top of a stack, but not in the target task — a new instance would be created and pushed on the stack.

In contrast, "singleTask" and "singleInstance" activities can only begin a task. They are always at the root of the activity stack. Moreover, the device can hold only one instance of the activity at a time — only one such task.

The "singleTask" and "singleInstance" modes also differ from each other in only one respect: A "singleTask" activity allows other activities to be part of its task. It's always at the root of its task, but other activities (necessarily "standard" and "singleTop" activities) can be launched into that task. A "singleInstance" activity, on the other hand, permits no other activities to be part of its task. It's the only activity in the task. If it starts another activity, that activity is assigned to a different task — as if FLAG\_ACTIVITY\_NEW\_TASK was in the intent.

|  |  |  |  |
| --- | --- | --- | --- |
| **Use Cases** | **Launch Mode** | **Multiple Instances?** | **Comments** |
| Normal launches for most activities | "standard" | Yes | Default. The system always creates a new instance of the activity in the target task and routes the intent to it. |
| "singleTop" | Conditionally | If an instance of the activity already exists at the top of the target task, the system routes the intent to that instance through a call to its [onNewIntent()](http://developer.android.com/reference/android/app/Activity.html#onNewIntent%28android.content.Intent%29) method, rather than creating a new instance of the activity. |
| Specialized launches *(not recommended for general use)* | "singleTask" | No | The system creates the activity at the root of a new task and routes the intent to it. However, if an instance of the activity already exists, the system routes the intent to existing instance through a call to its [onNewIntent()](http://developer.android.com/reference/android/app/Activity.html#onNewIntent%28android.content.Intent%29) method, rather than creating a new one. |
| "singleInstance" | No | Same as "singleTask", except that the system doesn't launch any other activities into the task holding the instance. The activity is always the single and only member of its task. |

**Activity的Flags**

FLAG\_ACTIVITY\_NEW\_TASK：和singleTask一样。

FLAG\_ACTIVITY\_SINGLE\_TOP：和singleTop一样。

FLAG\_ACTIVITY\_CLEAR\_TOP：一般和singleTask一起使用，清掉栈上的所有其它Activity。

FLAG\_ACTIVITY\_EXCLUDE\_FROM\_RECENTS：使得Activity不出现在历史列表中一样。

1.3 IntentFilter的匹配规则

启动Activity分为两种，显示调用和隐式调用。隐式调用主要依靠Intent能够匹配目标组件的IntentFilter，IntentFilter中的过滤信息有action, category, data。

**action的匹配规则**

可以使用系统预定义的action，也可以使用应用定义的。action字符串必须完全匹配。

**category的匹配规则**

Intent如果有category，必须每个category都能和IntentFilter中的某一个category匹配，不能有一个不匹配。每个Intent都有隐式的category android.intent.category.DEFAULT。

**data的匹配规则**

更多的是看MIMETYPE，就像文件打开对话框一样。

**第2章 IPC机制**

2.1 Android IPC简介

2.2 Android中的多进程模式

通过为四大组件（Activity, Service, Receiver, ContentProvider）指定android:process属性，可以开启多进程模式。

2.3 IPC基础概念介绍

**Serializable接口**

**Parcelable接口**

一个类实现Parcelabel接口，需要实现：

int describeContents()，描述是否含有fd。

void writeToParcel(Parcel out, int flags)，实现如何将对象写入Parcel对象。

public static final Parcelable.Creator<XXX> CREATOR，一个CREATOR的实现。

CREATOR中包含了：

XXX createFromParcel(Parcel in)，实现如何从Parcel对象读出并生成XXX。

XXX[] newArray(int size)，实现如何产生包含size个XXX的数组。

Parcelable比较适合在Android中传输。

**Binder**

Binder是Android系统中的一种IPC机制，它使用了/dev/binder设备来实现，在Android Framework中，Binder是ServiceManager连接各种Manager和Service之间的桥梁，在Android应用中，Binder连接了客户端和服务端，bindService通过Binder向客户端提供服务端的业务调用的Binder对象，可以由AIDL辅助实现。

Binder的工作机制

Client

Binder

远程请求

挂起

返回数据 唤醒client

data

写入参数

Transact

Service

onTransact

写入结果

reply

当客户端发起远程请求，当前线程会被挂起直到服务端进程返回数据。服务端的Binder方法运行在Binder线程池中，所以无论如何，Binder的处理方法都要用同步方法实现，不要自己开线程了。

2.4 Android中的IPC方式

**使用Bundle**

Android中的三大组件（Activity, Service, Receiver）都支持在Intent中传递Bundle。Bundle实现了Parcelable接口。

**使用文件共享**

直接使用Java Serializable接口。

**使用Messenger**

Messenger是一种轻量级的IPC方案，底层实现就是AIDL。

实现Messenger需要服务端和客户端。

服务端中创建一个Service来处理客户端连接请求，同时创建一个Handler来创建Messenger对象，在Service的onBind中返回这个Messenger的Binder。

客户端中首先绑定服务端的Service，绑定后就能通过服务端返回的IBinder创建Messenger，通过Messenger发送Message对象。

Messenger

Messenger Binder

Messenger

replyTo Messenger Binder

Client

Handler

Service

Handler

Messenger的工作原理

**使用AIDL**

使用AIDL进行进程间通信，分为服务端和客户端两个方面。

**服务端**

服务端需要创建一个Service来监听客户端的连接请求，然后创建AIDL文件，将暴露给客户端的接口在AIDL文件中声明，最终在Service中实现这些AIDL接口。

**客户端**

客户端首先绑定服务端的Service，然后将服务端的Binder转换成AIDL接口所属的类型，调用AIDL接口即可。

**使用ContentProvider**

ContentProvider是Android中专门用于不同应用间数据共享的方式，它的底层实现也是Binder。

ContentProvider主要以表格形式来组织数据，并且可以包含多个表，每个表都有行列，行对应一条记录，列对应一个字段。

**使用Socket**

Socket也可以用于通信，需要网络权限。

2.5 Binder连接池

AIDL和Binder的实现中，Android系统使用了连接池，可以使一些Binder并发处理，加快Binder的处理速度。

**第3章 View的事件体系**

3.1 View基础知识

View是界面层控件的抽象。

View的位置参数属性是top, left, right, bottom。

3.2 View的滑动

使用scrollTo和scrollBy，scrollTo是绝对值的scroll，scrollBy是相对当前位置的scroll。

3.3 弹性滑动

3.4 View的事件分发机制

3.5 View的滑动冲突

常见的滑动冲突场景

* 外部滑动方向和内部滑动方向不一致
* 外部滑动方向和内部滑动方向一致
* 上面两种情况嵌套

**第4章 View的工作原理**

4.1 初试ViewRoot和DecorView

ViewRoot对应于ViewRootImpl类，它是连接WindowManager和DecorView的纽带。View的三大流程均是通过ViewRoot来完成。

View的绘制流程是从ViewRoot的performTraversals方法开始的，它经过measure, layout和draw三个过程最终将View绘制出来。

DecorView作为Activity中的顶级View，在上面显示TitleBar，在下面显示我们的View内容。

4.2 理解MeasureSpec

4.3 View的工作流程

**第5章 理解RemoteView**

**第6章 Android的Drawable**

6.1 Drawable简介

Drawable是Android中可绘制对象的抽象基类。

**第7章 Android动画深入分析**

**第8章 理解Window和WindowManager**

**第9章 四大组件的工作过程**

**第10章 Android的消息机制**

**第11章 Android的线程和线程池**

**第12章 Bitmap的加载和Cache**