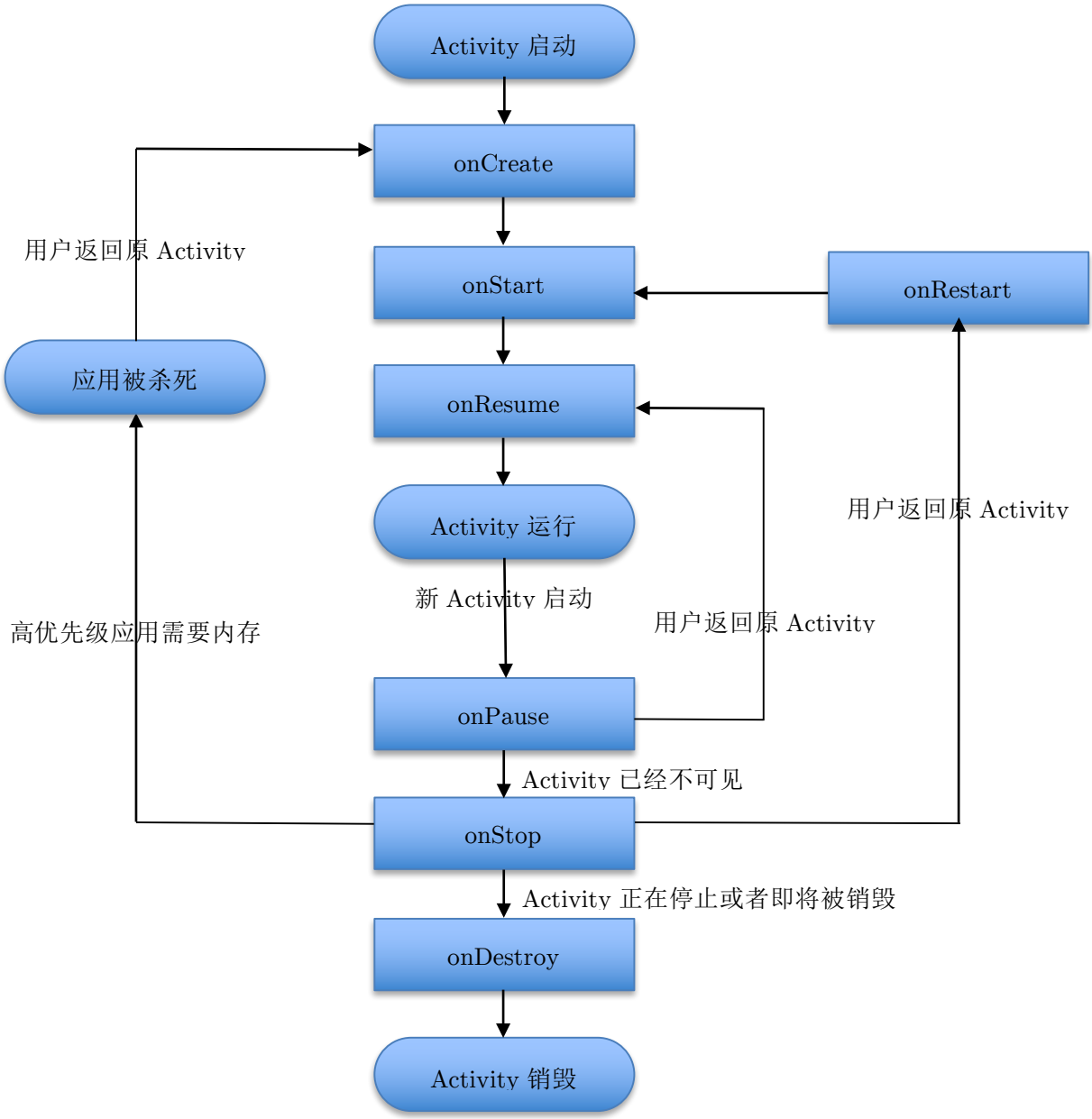
第 1 章 Activity 的生命周期和启动模式

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。



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() 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() 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() 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`。

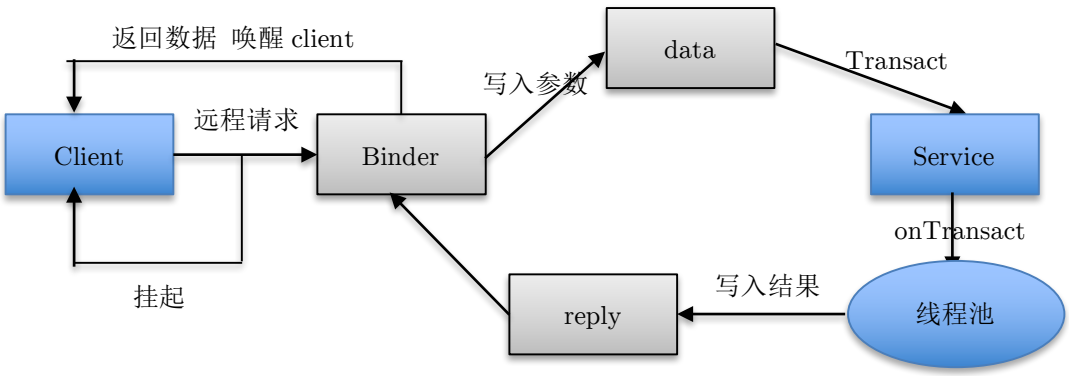
`XXX[] newArray(int size)`，实现如何产生包含 `size` 个 `XXX` 的数组。

`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



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

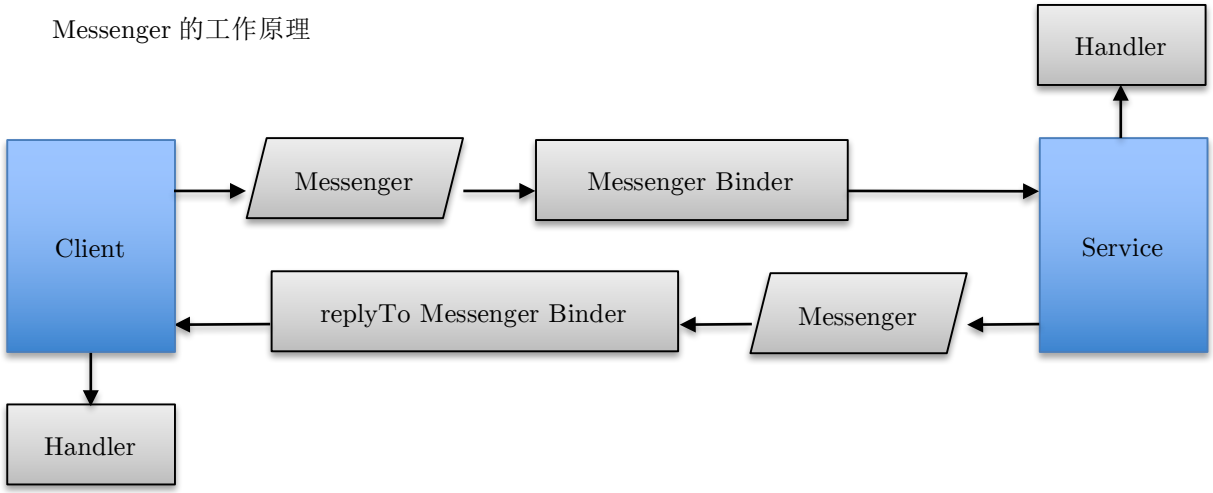
Messenger 是一种轻量级的 IPC 方案，底层实现就是 AIDL。

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 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