

Activity详解

启动模式

在Android中Activity有4种启动模式，配置时通常在AndroidManifest.xml文件中，配置Activity的属性`android:launchMode="`“即可

standard: 默认的启动模式，不论是打开一个Activity还是intent消息，系统都会每次创建新实例，每个task都可以有，每个task都可以有多个实例，缺点太耗费系统资源

singleTop: 如果当前实例在task栈顶，就不再去创建实例，系统会通过`onNewIntent()`去寻找实例，如不再栈顶，则需创建实例，该模式解决栈顶多个重复相同的Activity的问题（在栈顶可以复用）

singleTask: 新建一个task,如果已经有其他的task并且包含该实例，那就直接调用那个task的实例并把该Activity以上的Activity实例都pop掉，也即该实例存在于该Task的栈底，手机浏览器就使用该模式（只有一个task中会有）

singleInstance: 新建一个task且在task中只有他的唯一实例，该模式一般用于加载较慢的，比较耗性能且不需要每次都重新创建的Activity（该实例是task的唯一成员）

启动过程

1. 使用代理模式启动到ActivityManagerService中执行；
 - 2 创建ActivityRecord到mHistory记录中；
 - 3 通过socket通信到Zygote相关类创建process；
 - 4 通过ApplicatonThread与ActivityManagerService建立通信；
 - 5 ActivityManagerService通知ActiveThread启动Activity的创建；
 - 6 ActivityThread创建Activity加入到mActivities中并开始调度Activity执行；
-

首先了解一些相关的类

ActivityManagerServices :简称AMS，服务端对象，负责系统中所有Activity的生命周期

ActivityThread :App的真正入口。当应用启动后会调用main()开始运行，开启消息循环队列，这就是所谓的UI线程，与ActivityManagerServices配合，一起完成Activity的管理工作。每个App都有一个ActivityThread来表示应用程序的主进程，而每一个ActivityThread都包含有一个ApplicationThread实例，它是一个Binder对象，负责和其它进程进行通信

ApplicationThread :用来实现ActivityManagerService与ActivityThread之间的交互。在ActivityManagerService需要管理相关Application中的Activity的生命周期时，通过ApplicationThread的代理对象与ActivityThread通讯。

ApplicationThreadProxy :是ApplicationThread在服务器端的代理，负责和客户端的ApplicationThread通讯。AMS就是通过该代理与ActivityThread进行通信的。

Instrumentation :每一个应用程序只有一个Instrumentation对象，每个Activity内都有一个对该对象的引用。Instrumentation可以理解为应用进程的管家，ActivityThread要创建或暂停某个Activity时，都需要通过Instrumentation来进行具体的操作。

ActivityStack :Activity在AMS的栈管理，用来记录已经启动的Activity的先后关系，状态信息等。通过ActivityStack决定是否需要启动新的进程。

ActivityRecord :ActivityStack的管理对象，每个Activity在AMS对应一个ActivityRecord，来记录Activity的状态以及其他的管理信息。其实就是服务器端的Activity对象的映像。

TaskRecord :AMS抽象出来的一个“任务”的概念，是记录ActivityRecord的栈，一个“Task”包含若干个ActivityRecord。AMS用TaskRecord确保Activity启动和退出的顺序。

初始化ActivityManagerServices

当zygote开启时调用main()进行初始化

```
Zygote.main(){
    private static boolean startSystemServer(){
        Zygote.forkSystemServer();//fork出一个SystemServer
    }
}
```

在SystemServer进程开启时初始化ActivityManagerService

```
public final class SystemServer{
    mActivityManagerService =
    mSystemServiceManager.startService();
}
```

主线程与Instrumentation的关系

```

public void startActivity(Intent intent,...){
    startActivityForResult(intent){
        mInstrumentation.execStartActivity(){
            //有很多关于Application和Activity初始化和生命周期的方法
            callActivityOnCreate(Activity activity,Bundle bundle){
                final void performCreate(Bundle bundle){
                    onCreate(bundle); //Activity的入口函数
                }
            }
        }
    }
}

```

AMS和ActivityThread之间的Binder通信

客户端利用Binder对象，调用transact()将参数封装成Parcel对象，向AMS发送数据进行通信

客户端：ActivityManagerProxy → Binder驱动 → ActivityManagerService：服务器

由于Binder通信是单向的，要想AMS发送到ActivityThread

客户端：ApplicationThread ← Binder驱动 ← ApplicationThreadProxy：服务器

AMS接收到客户端的请求后启动Activity的调用链

```

startActivity(){ //某一事件触发跳转
    startActivityAsUser(){
        //通过ActivityStackSupervisor实现对ActivityStack的部分操
        mActivityStackSupervisor.startActivityMayWait(){
            startActivityLocked(){
                startActivityUncheckedLocked(){ //检验为正当启动请求
                    mActivityStack.startActivityLocked(){
                        mActivityStackSupervisor.resumeTopActivitiesLocked(){
                            mActivityStack.resumeTopActivityLocked(){
                                resumeTopActivityInnerLocked(){
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}

```

启动过程

1. Launcher中发送startActivity请求 Instrumentation.execStartActivity()
2. AMS接收客户端startActivity请求 ActivityStack.startActivityLocked()
3. 创建新的Task ActivityStack.startActivityUncheckedLocked()&startActivityLoacked()
4. 运行mHistory中最后一个ActivityRecord ActivityStack. resumeTopActivityLocked()
5. 暂停当前运行Activity 调用ActivityStack.startPausingLocked()暂停当前Activity
6. AMS处理暂停Activity事情 在ActivityStack.completePauseLocked()中完成暂停
7. 正式启动目标Activity ActivityStack.resumeTopActivityLocked、
ActivityStack.startSpecificActivityLocked
8. fork一个新的进程 调用ActivityThread.attach()开始新的应用程序
9. AMS准备执行目标Activity ActivityManagerService.attachApplication()
10. 客户进程启动指定Activity 调用ActivityThread.performLaunchActivity()完成Activity的加载

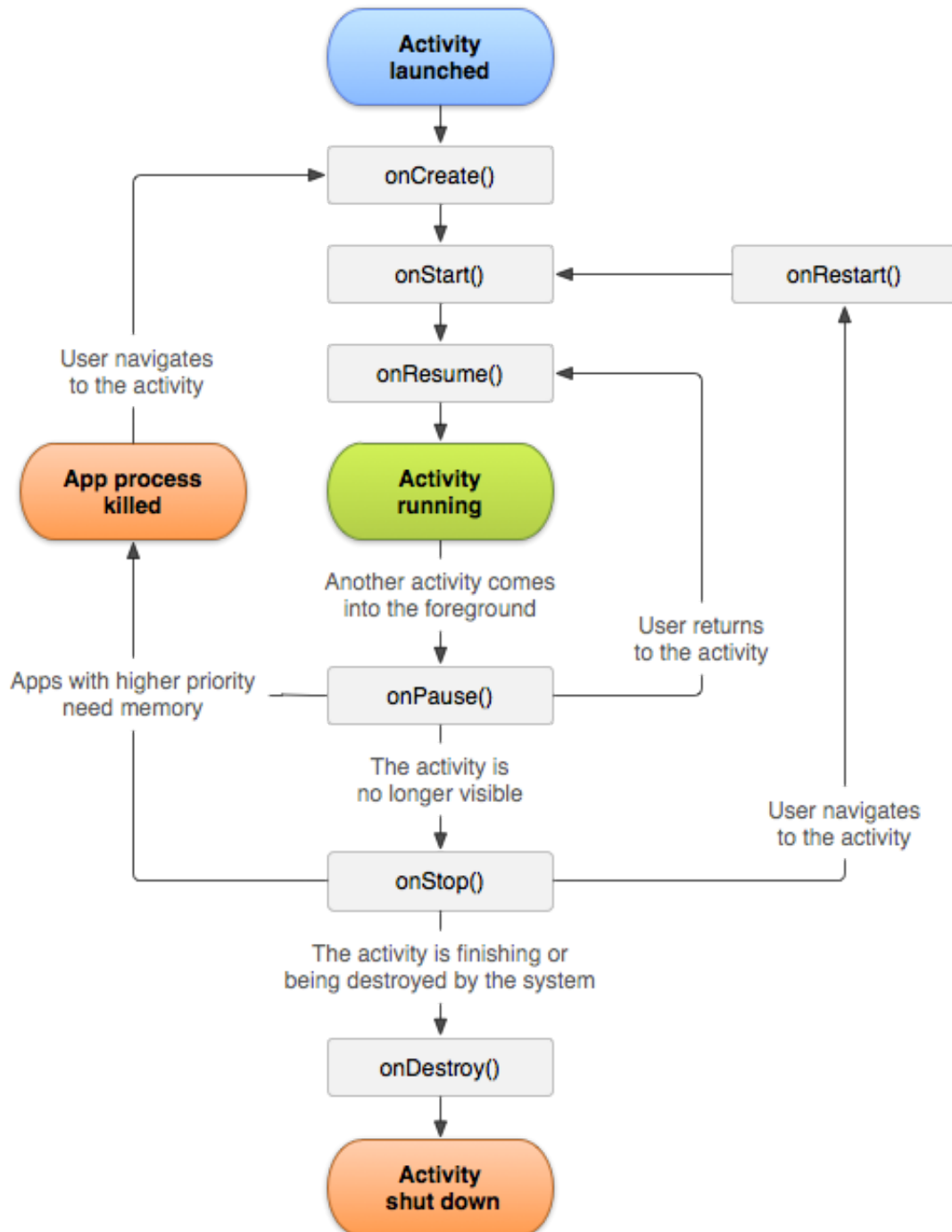
生命周期

Activity的三种状态

Resumed:显示在屏幕并获取焦点

Paused:依然显示在屏幕失去焦点，当内存低时有可能被系统杀死

Stope:完全不可见



onCreate():当activity是被创建时候执行该方法。该方法做一些初始化动作，比如创建views，设置数据到list等等，该方法提供了一个Bundle类型的变量，该变量中有这个activity以前的状态信息，前提是以前存过这些信息。

onRestart():把activity从onStop状态唤醒时，会用onRestart方法，该方法优先于再次运行的onStart，运行完onRestart之后运行onStart。

onStart():当activity对用户可见时会调用onStart，当activity在前台显示时，会运行onResume

onResume():当activity开始与用户交互时，会调用onResume，并且为了用户操作此时该activity位于activity栈的顶部。

onPause():当一个activity运行到onResume方法后，不管是这个activity要销毁还是要暂停或停止，都会调用该方法。这个方法之后有可能是onResume也有可能是onStop，若是在这个activity-A中打开一个不完全覆盖这个activity-A的新activity-B，那么activity-A就会是onPause状态，当activity-B退出时，activity-A就直接运行onResume，所以不建议在这个方法中执行CPU密集的操作)。若是需要退出activity-A，那么下一个就会执行onStop。onPause()用于提交未保存发生变化了的持久化数据。

onStop():当这个activity完全看不见的时候，会调用onStop方法，因为另一个activity会调用onResume并且覆盖这个activity。以下三种情况都会使这个activity调用onStop()方法，第一种是一个新的activity被执行，第二种是一个已经存在的activity被切换到最前端，第三种是这个activity要被销毁。如果通过用户召回这个activity，那么会调用onRestart方法;若这个activity要被销毁，则调用onDestroy方法

onDestroy():当activity销毁前会调用该方法，比如发生如下情况：activity调用了finish()方法来结束这个activity，或者因为系统为了节省空间而临时销毁这个activity，这两个情况可以通过isFinishing()方法判断

1.启动Activity

执行onCreate () -> onStart () -> onResume(), 执行完后Activity就显示在屏幕上

2.销毁Activity

执行finish()者被系统强制杀死时, activity会被销毁。则activity内部会执行 onPause()->onStop()->onDestory()

3.暂停和继续

当界面被部分挡住时, 会进入暂停状态。此时会执行onPause ()

界面重新完全显示后又回到继续状态 (Resumed), 会执行onResume ()

在即将pause时, 我应应该在onPause中执行一些释放操作, 比如停止正在进行的动画, 一些用户的状态 (确定用户会保存的, 比如邮件草稿), 释放系统支援 (如广播接收者, 传感器), 以及其他会消耗电池并且在paused时用户不需要用到的。同时这些释放或者保存的, 我们在onResume时候需要恢复。

4.停止和重启

当进去其他界面之后, 如接听电话, 或者打开其他activity, 我们的界面会停止, 进入stoped状态。此时会执行onPause () ->onStop(), 重启时会执行onRestart () ->onStart()->onResume()

在stop的时候, 我们需要执行一些比在onPause中更加消耗CPU的更大的任务, 比如写数据库。同时建议, 在onStart () 中恢复, 而不是在onRestart () 中恢复。

5.重新创建Activity

当我们的进程被destory (不是用户主动调用finish), 可以返回。返回的时候会重新创建。执行过程和创建activity一样。

当activity被系统kill之前, 会调用onSaveInstanceState () 去保存UI状态, 如果我们有信息需要保存, 也可以去重写这个方法去做。同时我们可以重写onRestoreInstanceState () 去恢复状态。不重写, 系统会恢复系统保存的那一部分UI。或者我们可以在onCreate中恢复, onCreate的参数savedInstanceState就是我们保存的信息, 可以判断该参数是否为空, 来恢复界面。

横竖屏切换时Activity的生命周期

1.不设置Android : configChanges时, 且屏会重新调用各个生命周期, 切横屏调用一次, 切竖屏调用两次。

2.设置Android : configChanges="rientation"时, 切屏会调用各个生命周期, 切横竖屏只会调用一次

3.设置Android : configChanges="rientation|keyboardHidden"时, 切屏不会重新调用各个生命周期, 只会执行onConfigurationChanged方法。

小结

- 当用户自己退出程序的时候, 建议在onStop方法中保存数据;
- 当用户打开一个新的activity的时候, 建议通过onSaveInstanceState来保存数据;也有人应该说应该放在onPause里保存, 其实我觉得在打开新的一个activity的时候, 或者将程序至于后台的时候, 都会默认调用onSaveInstanceState方法, 而且在这种暂停的状态下, Android的内存管理机制也不太会杀死这种状态的activity。而用onPause保存的时候, 若是下一个执行onResume的方法的话, 会影响速度, 当然数据量小的话也感

觉不出来

参考