# 深入理解Android

## 基础框架

## JavaBinder

在java层初创时期，系统提前注册一些JNI函数。建立两界binder关系。

## MessageQueue

2.3.1 MessageQueue的创建

new MessageQueue

nativeInit: c层创建一个消息队列

new NativeMessageQueue

new Looper

2.3.2 提取消息

messageQueue.next

~.nativePollOnce: 等待一个消息

mMessages.next: 链表

messageQueue.enqueueMessage

{添加消息到队尾}

nativeWake：结束poll的等待

# 卷2

## SystemServer

可以称之为系统进程，运行了系统的主要服务。

进程名：system\_server

线程：ServerThread

SystemServer.main

## 4 PKMS

### 4.3 pkms初始化

pkms成员mSettings：保存manifest sharedUserId

三角关系：文本字符串-》linux uid-》多个package

创建

流程

1. 解析xml，保存到内存
2. 解析APK，保存到内存
3. 保存到文件

systemThread.run

PackageManagerService.main：扫描文件夹，解析APK

new PackageManagerService

new Setting

settings.addSharedUserLPw：解析sharedUserId

new SharedUserSetting

sharedUserSetting.addUserIdLPw

pkms.readPermission：解析系统权限

~.readPermissionsFromXml

settings.readLPw：解析半成品文件

installer.dexopt：dex化核心库

file.delete: 删除缓存文件

pkms.scanDirLI: 扫描Package

~.scanPackageLI

new PackageParser

packageparser.parsePackage：解析manifest

~.parsePackage

~.parsePackageName

new Package

~.paserApplication

pkms.scanPacakgeLi

{单独处理系统Activity}

~.fixProcessName

~.getDataPathForPackage

userManager.installPacakgeForAllUsers

{处理so}

~.performDexOptLI

ServiceManager.addService

packageManagerService.isFirstBoot

~.performBootDexOpt

~.systemReady

### 4.4 APK安装

安装会把apk复制到/data/app目录下

pkms.installPacakgeWithVerification

new InstallParams

sendMessage INIT\_COPY

pkms.handler. INIT\_COPY

connectToserice: 连接到DefaultContainerservice

mPendinInstalls.add

sendMessage MCS\_BOUND

pkms.handler. MCS\_BOUND

handlerParams.startCopy

installParams.handleStartCopy

defContainerService.getMinimalPackageInfo:{确认安装路径}

new InstallArgs

installArgs.copyApk: 复制到/data/app目录下

~.handleReturnCode

~.processPendingInstall

pkms.handler.post

run

pkms.installPackage: 重命名；扫描APK，保存到内存

sendMessage POST\_INSTALL

### 4.5 queryIntent

Intent: action, data, category, type, component, extras

IntentFilters: action, category, data(uri/type)

数据预处理

Pkms.mActivites保存了解析的activity、filter

pkms.scanPackageLi

activityIntentResolver.addActivity

~.addFilter

register: 保存各种xxxToFilter

匹配

应用进程

applicationPackageManager.queryIntentActivities

mPM.~

系统进程

pkms.~

{指定comp：组件中查询}

{指定包名：包中查询}

activityIntentResolver .queruIntent:{全局查询}

intentResolver.~

{四轮查询}

## 6 AMS

### 6.2 创建AMS

除了系统进程本来的ServerThread，这里又创建了AThread（AMS工作线程）和ActivityThread（系统交互进程）

系统进程：

ServerThread.run

AMS.main：准备AMS工作场景，准备系统应用运行场景

new AThread

->AThread.run

new AMS

ActivityThread.systemMain

new ActivityThread

activityThread.attach

~.getSystemContext

ContextImpl.createSystemContext

new LoadedApk

context.init

Instrumentation.newApplication

application.onCreate

activityThread.getSystemContext

new ActivityStack

ams.startRunning

AMS.setSystemProcess：安装系统应用；向进程管理注册自己

activityThread.installSystemApplicationInfo

context.init

ams.newProcessRecordLocked

new ProcessRecord

AMS.installSystemProvider：安装SettingsProvider

ams.generateApplicationProviderLocked

pms.queryContentProviders

activityThread.installSystemProviders

~.installContentProviders

~.installProvider

newInstance

ams.publishContentProviders

ams.mProvidersByClass.put

ams.mProvidersByName.put

AMS.systemReady：重启应用进程，启动系统UI

{杀死persist进程}

runnable.run

ams.startSystemUi

{启动persist进程}

activityStack.resumeTopActivityLocked: 启动桌面

ams.startHomeActivityLocked

activityStack.activityIdelInternal

ams.finishBooting

broadcastIntentLocked ACTION\_BOOT\_COMPLETED

AMS和应用进程交互接口

系统进程：客户端：保存在ProcessRecord：ApplicationThreadNative

应用进程：服务端（响应控制命令）：保存在ActivityThread：ActivityThread.ApplicationThread

### 6.3 Activity生命周期

设计理念：一个Task对应一个任务，包含多个Activity

系统支持一个处于前台Task，其他处于后台。

Recent屏幕是Task列表

ActivityStack负责栈管理。全局只有一个ActivityStack管理所有Activity。注意这里实现和设计的差异。Task栈是逻辑上存在的概念，实现比较复杂。

查找方法：

activityStack.topRunningActivityLocked

~.topRunningNonDelayedActivityLocked

~.findActivityLocked

~.findTaskLocked

启动模式：

standard：每次启动增加一个新的Activity，放在同一个task下

singleTop: 顶部不重复

singleTask：一个Task，一个Activity。非排他，Task中可以容纳其他。

singleInstance: 同上，排他。一个Task里只有一个Activity

FLAG\_ACTIVITY\_

NEW\_TASK: 启动新一个新的Task

CLEAR\_TASK: 必须和上面同时使用。去除老的相关Task，创建新的Task放心的Activity

CLEAT\_TOP: 去除栈上面的Activity

//TODO 启动模式再查查资料

暂时禁止App切换

重要Activity处于前台，不允许切到其他APP

ams.stopAppSwitches: 设置标志，设置允许时间，设置超时动作

ams.resumeAppSwitches: 设置允许时间

系统进程：

ams.startActivityAndWait

activityStatck.startActivityMayWait

~.resolveActivity：和pkms交互

~.startActivityLocked

new ActivityRecord

~.checkAppSeitcvhAllowedLocked:{处理禁用App switch，延迟启动}

~.mPendingActivityLaunches.add

~.doPendingActivityLaunchesLocked

/~.startActivityUncheckedLocked: 立即启动，处理Task

{是否设置NEW\_TASK}

{查找ActivityRecord}

{处理是否调用onNewIntent}

/new TaskRecord

activityRecord.setTask

~.startActivtyLocked

~.resumeTopActivityLocked

**~.startPausingLocked: 暂停当前Activity**

~.startSpecificActivityLocked

ams.startProcessLocked

new ProcessRecordLocked

~.startProcessLocked

Process.start

以上启动了一个空的应用进程。

应用进程

ActivityThread.main

activityThread.attach

activityManager.attachApplication

系统进程

ams.attachApplictionLocked

applicationThread.bindApplication : 让应用进程准备安卓运行环境

ams.udpateLruProcessLocked

activityStack.realStartActivtyLocked

app.thread. scheduleLaunchActivity

activtyStack.completeResumeLocked: 设置启动超时等待

ams.realStartServiceLocked

{启动BroadcastReceiver}

ams.updateOomAdjLocked: 如果有新进程启动，则调整内存优先级

应用进程: 应用启动/准备运行环境

activityThread.bindApplication

~.handleBindApplication

loadedApk.makeApplication

/installCPs

mInstrumentation.callApplicationOnCreate

应用进程：启动Activity

activityThread.scheduleLaunchActivity

new ActivityClientRecord

~.handleLaunchActivity

~.performLaunchActivity: onCreate\onStart

~.handleResumeActivty

~.performResumeActivity

Looper.myQueue.addIndleHandler

idleHandler.queueIdle

am.activityIdle

系统进程：启动结束的处理

activityStack.activityIdleInternal

~.processStoppingActivitiesLocked: 查找停止的activity

~.finishCurrentActivityLocked: onDestroy

/~.stopActivityLocked: onStop

app.thread.scheduleStopActivity

应用进程

activityThread.handleStopActivity

~.performStopActivityInner

am.activityStopped

×××××××××××××××××××

**接上面，启动新的Activity前，处理旧的Activity**

系统进程：

activityStack.startPausingLocked

app.thread.schedulePauseActivity: 通知应用进程处理pause

应用进程

activityThread.handlePausingActivity

~.performdUserLeavingActivity

~.performPauseActivity

activityManager.activityPaused: 通知系统进程

系统进程

ams.activityPasued

activityStack.activityPaused

~.completePasueLocked

~.scheduleIdleLocked

~.resumeTopActivityLocked: 启动新的Activity

### 6.4 Broadcast

三种广播处理：并行、串行、sticky

异步广播的处理流程

接收器注册：

IIntentReceviver: 客户端接受广播的接口

应用进程

contextImpl.registerReceiver

contextImpl.registerReceiverInternal

amn.registerReceiver-》转到系统进程

系统进程

ams.registerReceiver: 保存客户端receiver

new ReceiverList：对应单一接收器：保存过滤器、客户端接口

mRegisteredReceivers.put

mReceiverResolver.addFilter: 动态注册的

scheduleBroadcastLocked: 发送sticky广播

BROADCAST\_INTENT\_MSG

6.4.2 sendBroadcast

contextImpl.sendBroadcast

amn.broadcastIntent->转系统进程

ams.broadcastIntentInternal: 内部处理广播；匹配；发送；

new BroadcastRecord

ams.mParallelBroadcasts.add

ams.scheduleBroadcastsLocked

ams.mOrderedBroadcasts.add

ams.scheduleBroadcastsLocked

BROADCAST\_INTENT\_MSG

ams.processNextBroadcast

//动态注册处理

ams.deliverToRegisteredReceiverLocked: 真正的发广播

ams.performReceiveLocked

applicationThread.scheduleRegisteredReceiver： 动态注册广播-》转应用进程

//串行处理

~.mOrderedBroadcasts.get

~.performReceiveLocked

r.receivers.get

~.deliverToRegisteredReceiveLocked: 动态

~.processCurBroadcastLocekd：静态

应用进程：

activityThread. scheduleRegisteredReceiver

LoadedApk.receiverDispatcher.performReceive

handler.post

args.sendFinished

ams.finshReceiver

args.run

broadcastReceiver.onReceive

ams.finshReceiver: 调度下一轮广播

processNextBroadcast:

### 6.5 startService

startService->onStartCommand->stopSelf/stopService

bindService->onBind->unbindService

//TODO service生命周期

### 6.6 进程

#### 6.6.1 Linux进程管理

进程管理分为两方面：CPU资源分配（实际上基于线程），内存资源分配

1 CPU

调度优先级（开发可控）、调度策略（系统）影响了CPU分配。

linux接口

setpriority()

设置进程、进程组、用户的优先级；nicer值约低，优先级越高。

sched\_setScheduler()

设置调度策略：分时策略、批处理策略、空闲策略

举例：MediaScannerService优先级11，默认优先级是0.

2 进程

oom\_adj内存优先级

linux接口：

/proc/进程id/oom\_adj写入设置的值

安卓为linux内核增加了lowmemorykiller

关键工作参数：

minfree：2048，3072

adj：0，1

当剩余内存为2048KB时，lmk将杀死adj>=0的进程

#### 6.6.2 Android进程管理

1 进程分类

2 Process类

注意：linux kernal不区分线程和进程

setThreadGroup

THREAD\_GROUP\_DEFAULT

THREAD\_GROUP\_BG\_NONINTERACTIVE

3 ProcessList ProcessRecord

#### 6.6.3 AMS进程管理函数

//系统进程

ams.attachApplictionLocked：ams新创建一个应用进程，该进程启动后调用AMS的attachAppliction

**~.updateLrcProcessLocked**

~. updateLrcProcessInternalLocked: 调整进程列表，处理lruWeight

{计算lrc权重}

mLruProcess.add

~. updateLrcProcessInternalLocked:调整依赖进程

**~.updateOomAdjLocked：处理curAdj**

~.updateOomAdjLocked

~.computeOomAdjLocked：这里计算了进程内存优先级，决定了进程的生死

Process.setProcessGroup

Process.killProcessQuiet: 某些后台进程和空进程会被杀死

applicationThread.scheduleTrimMemory: 通知应用进程清理内存

很多地方都调用了**updateOomAdjLocked：例如Service、CP启动**

### 6.7 crash

系统给每个进程设置一个异常处理器，处理每个线程的抛出的异常。

RuntimeInit.commonInit

Thread.setDeafultUncaughtExceptionHandler

uncaughtHandler.uncaughtException

amn. handleApplicationCrash

{杀死进程}

系统进程

ams.handleApplicationCrash

~.ddErrorToDropBox

~.crashApplication

~.makeAppCrashingLocked

~.generateProcessError

~.startAppProblemLocked

~.handleAppCrashLocked

~.handler.sendMessage: 界面弹窗

6.7.3 AppDeathRecipient

应用进程杀死后，死亡处理对象被唤醒

appDeathRecipient.binderDied

ams.appDiedLocked：处理内存优先级

~.handleAppDiedLocked

~.cleanUpApplicationRecordLocked

{清理service}

~.removeDyingProviderLocked：清理provider，杀死依赖进程

{清理Broadcast}

## 7 ContentProvider

### 7.2 MediaProvider

应用示例：

ContentResolver cr = context.getContentResolver();

Uri uri = MediaStore.Images.Media.EXTERNAL\_CONTENT\_URI;

Cursor cursor = MediaStore.Images.Media.query(cr, uri, null);

Cursor.moveToFirst();

…

cursor.close();

ContentResolver创建

context.getContentResolver

contextImpl.getContentResolver

contextImpl.init

new ApplicationContentResolver

查询

MediaStore.Image.Media.query

contentResolver.query

~.acquireProvider

applciationContentResolver.acquireProvider

activityThread.acquireProvider

**~.getProvider**

amn.ContentProvider->转系统进程

activityThread.installProvider

ams.ContentProvider

~.getContentProviderImpl

ams.mProvidersByName.get

ams.mProvidersByClass.get

new ContentProviderRecord

~.startProcessLocked

~.incProviderCount

contentProviderRecord.wait

ContentProvider创建

系统进程

ams.attachApplicationLocked

applicationThread.bindApplication

服务端进程

activityThread.handleBindApplication

~.installContentProviders

~.installProvider

new ContentProvider

contentProvider.getIContentProvider

ams.publishContentProviders

查询

ContentProvider.transport.query

contentProvider.query

客户端对服务端有了强依赖。Cursor关闭结束了该依赖。

cursor.close

contentImpl.releaseProvider

activityThread.releaseProvider

~.completeRemoveProvider

ams.removeContentProvider: 删除依赖关系

### 7.3 SQLite

SQLiteOpenHelper: onCreate, onUpdate

数据库创建

mediaProvider.attach

new DatabaseHelper

SQLiteOpenHelper(VERSION)

databaseHelper.getWritableDatabase

contentImpl.openOrCreateDatabase

SQLiteDatabase.openOrCreateDatabase

~.openDatabase

New SQLiteDatabase

sqliteDatabase.open：c层调用sqlite api打开数据库

~.onCreate

/~.onUpgrade

### 7.4 Cursor

客户端进程

contentResolver.query

~.acquireProvider

contentProviderProxy.query

new BulkCursorToCursorAdapter

adaptor.getObserver

mRemote.transact->转到服务端

BulkCursorNative.asInterface

adaptor.initailize

contentResolver.releaseProvider

return new CursorWrapperInner

服务端进程

contentProviderNative.onTransact

IContentObserver.Stub.asInterface

**mediaProvider.query**

databaseHelper.getReadableDatabase

sqliteQueryBuilder.query

~.buildQuery

sqliteDatabase.rawQueryWithFactory

~.getDbConnection

new SQLiteDirectCursorDiver

sqliteCursorDriver.query

new SQLiteQuery

**new SQLiteCursor**

releaseDbConnection

cusor.setNotificationUri

new CursorToBulkCursorAdapter

adaptor.count

sqliteCursor.count

~.fillWindow

~.clearOrCreateLocalWindow: 创建共享内存

new CursorWindow

nativeCreate

sqliteQuery.fillWindow

nativeFillWindow

客户端进程

cursorWrapper.moveToFirst

abstractCursor.moveToFirst

~.moveToPosition

bulkCursorToCursorAdaptor.onMove

bulkCursorProxy.getWindow

服务端

cursorToBulkCursorAdaptor.getWindow

sqliteCursor.getWindow

~.moveToPosition

~.onMove

~.fillWindow

### 7.5 cusor close

客户端

cursorWrapperInner.close

cursorWrapper.close

bulkCursorToCursorAdapter.close

abstarctCursor.close

abstarctWindowedCursor.onDeactivateOrClose

~.closeWindow

cursorWindow.close

**sqliteClosable.releaseReference**

cursorWindow.onAllReferenceReleased

**~.navtiveDispose**

contentResolver.releaseProvider

服务端

cursorToBulkCursorAdapteor.close

~.disposeLocked

sqlite.close

垃圾回收

如果不close，客户端cursor会被回收，而服务端不会被回收

### 7.6 openAssetFileDescriptor

客户端

contentResolver.openAssetFileDescriptor

~.openTypedAssetFileDescriptor

~.acquireProvider

contentProviderProxy.openTypedAssetFile

new ParcelFileDescriptorInner

new AssetFileDescriptor

服务端

contentProvider.openTypedAssetFile

~.openAssetFile

mediaProvider.openFile

contentProvider.openFileHelper

ParcelFileDescriptor.open

New AssetFileDescriptor

7.6.3 跨进程传递文件描述符

parcelFileDescriptor.writeToParcel

parcel.writeFileDescriptor

parcelFileDescriptor.readFileDescriptor: Binder驱动替客户端打开了文件

## ContentService

创建

ServerThread.run

ContentService.main

new ContentService

getSyncManager

new SyncManager

注册

contentResovler.registerContentObserver

contentObserver.getContentObserver

contentService.registerContentObserver

observerNode.addObserverLocked

通知

mediaProvider.update

contentResolver.notifyChange

~.~

contentService.~

contentObserver.onChange

## AccountManagerService

serverThread.run

new AccountManagerService

new AccountAuthenticatorCache

new RegisteredServicesCache

generateServicesMap

parseServiceInfo

new AccountManagerService

## SyncManager

new SyncManager

SyncStorageEngine.init

new SyncStorageEngine

~.getSingleton

new SyncAdaptersCache

registeredServicesCache

new SyncQueue

syncStorageEngine.addStatusChangeListener

accountManager.addOnAccountsUpatedListener

contentService.setSyncAutomatically

syncStorageEngine.~

发起同步请求：

contentResolver.requestSync

contentService.~

syncManager.scheduleSync

new SyncOperation

scheduleSyncOperation

syncHandler.handleMessage

syncManager.maybeStartNextSyncLocked

syncManager.dispatchSyncOperation

activeSyncContext.bindToSyncAdapter

context.bindService

activeSyncContext.onServiceConnected

MESSAGE\_SERVICE\_CONNECTED

syncHandler.handleMessage

syncManager.runBoundToSyncAdapter

syncAdapter.startSync

同步服务进程？

emailSyncAdapterService.onBind

sSyncAdapter.getSyncAdapterBinder

iSyncAdapter.startSync

new SyncThread

iSyncAdapterImpl.run

abstractThreadedSyncAdapter.onPerformdSync

emailSyncAdapterService.~

syncContext.onFinished

# 卷3

## 4 WindowManagerService

### 4.1 wms示例和创建

4.1.2 wms的构成

系统提前创建WindowManager线程、UI线程

1 wms创建

serverThread.run

WMS.main

wmHandler.runWithScissors->new WMS

wms.displayReady

wms.systemReady

wm线程

new WMS

wms.createDisplayContentLocked

mInputManager

initPolicy: 初始化WindowPolicyManager，即PhoneWindowManager

### 4.2 窗口管理

wms.addWindow

wms.mTokenMap.get

/new WindowToken

new WindowState

4.2.1 WindowToken

WindowToken对应一个应用组件，例如一个Activity；创建新窗口时，申明自己的身份。

客户端(AMS等)：WT仅仅是一个Binder对象。

服务端：WT是WindowToken类，其中保存了传过来的Binder。

两边不是一个东西。

Token类型：

|  |  |  |  |
| --- | --- | --- | --- |
|  | 客户端 | 服务端 |  |
| Wallpaper和InputMethod的Token | 普通Binder |  |  |
| Activity的Token | 有通信作用 | AppWindowToken | TYPE\_APPLICATION |

4.2.2 WindowState

WMS中描述一个窗口。全部保存在mWindowMap

客户端对应IWindow.Stub：通信

4.2.3 DisplayContent

### 4.3 窗口的显示次序

主序由窗口类型决定（非应用一般只看主序）

子序由子（相对父窗口）窗口类型（全都APPLICATION开头）决定（应用窗口还跟AMS有关系。）

wms.addWindow：mBaseLayer+mSubLayer=>mLayer=>mAnimLayer

~.addWindowToListInOrderLocked：分屏幕管理；按主序和父子关系排序

~.assignLayersLocked：处理主序相同的情况

设置显示次序到Surface：动画系统最终指示SurfaceFlinger

设置时机

session.relayout为新窗口分配surface以后让

设置代码

windowStateAnimator.prepareSufaceLocked

surface.serLayer

### 4.4 窗口布局

1 WMS各种功能：修改窗口、屏幕属性

2 布局子系统：为所有窗口计算布局

performLayoutAndPlaceSurfaceLocked等函数

3 动画子系统：移动窗口到指定位置，附加特效

Chroreographer、WindowAnimator等

wms.relayoutWindow

~.windowForClientLocked: 找到指定的窗口

windowState.mDisplayContent.layoutNeeded = true

~.performLayoutAndPlaceSurfacesLocked:

{循环处理}

~.performLayoutAndPlaceSurfacesLockedLoop

~.performLayoutAndPlaceSurfacesLockedInner

布局前的处理

遍历DisplayContent

遍历该DisplayContent的所有窗口

布局窗口

检查布局结果，是否重新布局

处理DispplayContent

布局后策略处理

~.needsLayout: 检查DisplayContent的标志

~.sendNewConfiguration

布局前处理

布局常用区域

unrestricted: 整个屏幕

restricted: 除底部导航栏，受导航栏可见性影响，可变

stableFullscreen: = unrestricted

stable: 除状态栏，导航栏，不可变

dock: 输入法窗口等

content: 除状态栏、导航栏、输入法

cur：大部分情况下=content

system: 近似dock

ParentFrame: 可能是Restricted\Unrestricted\Dock\Content

DisplayFrame: 一般=Par

ContentFrame: 可能是Content\Dock\Restricted

VisibleFrame: ContentFrame\Cur

一些FLAG

FORCE\_NOT\_FULLSCREEN: 必须同时显示状态栏等

FULLSCREEN：全屏。

SYSTEM\_UI\_FLAG\_FULLSCREEN: = FULLSCREEN

SHOW\_WHEN\_LOCKED：锁屏下显示

DISMISS\_KEYGUARD: 退出锁屏

### 4.5 WMS的动画系统

4.5.1 简介

计算逻辑：

Animation：根据开始、结束状态、时间计算变换。

Transform: 上面的计算结果，包含两个分量：透明度和二维变换矩阵

Iterpolater: 根据标准时间计算插值时间

执行框架：

Choreographer: 根据VSYNC回调。

## 5 输入系统

### 5.1 简介

当输入设备可用时，内核在/dev/input/下穿件名为event0 ~n的设备节点。不可用时，则移除。

用户空间通过ioctl读取设备信息

当操作输入设备时，内核收到硬件中断，加工成原始事件，写入设备。用户空间通过read函数读出事件。

5.1.1 getEvent和sendEvent

5.1.2 Android输入系统

内核->设备节点-》ims-》wms-》Window-》ViewRootImpl-》View

ims：eventhub-》inputReader-》（派发队列）-》inputDispatcher-》（窗口管道）-》Looper->（事件处理者）

5.1.3 InputManagerService启动

ServerThread.run

new IMS

new InputManagerHandler：分发策略实现

nativeInit

new NativeInputManager: 实现了InputeReaderPolicy，InputDispatcherPolicy, 转给Java层IMS

new EventHub

new InputManager

new InputDispatcher

new InputReader

initialize

new InputReaderThread

new InputDispatcherThread

ims.start：启动线程

display.setInputManager

### 5.2 Reader

5.2.1

1 INotify

文件系统变化通知机制，监控文件的变化：创建、删除、读写

需要主动读取

2 Epoll

针对描述符，等待事件发生

5.2.2 InputReader流程

inputReader.loopOnce

eventHub.getEvents

~.processEventsLocked

mQueuedListener.flush

5.2.3 EventHub

监听底层设备节点和原始输入事件

5.2.4 InputReader

5.5.1 ANR的产生

1 窗口可以接收事件的条件

2 重试派发与ANR的引发

## 6 控件系统

### 6.2 WindowManager

6.2.1 WM体系结构

6.2.2 WMGlobal

6.3 ViewRootImpl

# 动画常用类

# 事件

# Webview

# 安卓文档

# 进程和线程

当某个应用组件启动且该应用没有运行其他任何组件时，Android 系统会使用单个执行线程为应用启动新的 Linux 进程。默认情况下，同一应用的所有组件在相同的进程和线程（称为“主”线程）中运行。 如果某个应用组件启动且该应用已存在进程（因为存在该应用的其他组件），则该组件会在此进程内启动并使用相同的执行线程。 但是，您可以安排应用中的其他组件在单独的进程中运行，并为任何进程创建额外的线程。

本文档介绍进程和线程在 Android 应用中的工作方式。

进程

默认情况下，同一应用的所有组件均在相同的进程中运行。如果您发现需要控制某个组件所属的进程，则可在清单文件中执行此操作。

各类组件元素的清单文件条目—[<activity>](https://developer.android.google.cn/guide/topics/manifest/activity-element.html)、[<service>](https://developer.android.google.cn/guide/topics/manifest/service-element.html)、[<receiver>](https://developer.android.google.cn/guide/topics/manifest/receiver-element.html) 和 [<provider>](https://developer.android.google.cn/guide/topics/manifest/provider-element.html)—均支持 android:process 属性，此属性可以指定该组件应在哪个进程运行。您可以设置此属性，使每个组件均在各自的进程中运行，或者使一些组件共享一个进程，而其他组件则不共享。 此外，您还可以设置 android:process，使不同应用的组件在相同的进程中运行，但前提是这些应用共享相同的 Linux 用户 ID 并使用相同的证书进行签署。

此外，[<application>](https://developer.android.google.cn/guide/topics/manifest/application-element.html) 元素还支持 android:process 属性，以设置适用于所有组件的默认值。

进程生命周期

如果内存不足，而其他为用户提供更紧急服务的进程又需要内存时，Android 可能会决定在某一时刻关闭某一进程。在被终止进程中运行的应用组件也会随之销毁。 当这些组件需要再次运行时，系统将为它们重启进程。

决定终止哪个进程时，Android 系统将权衡它们对用户的相对重要程度。例如，相对于托管可见 Activity 的进程而言，它更有可能关闭托管屏幕上不再可见的 Activity 的进程。 因此，是否终止某个进程的决定取决于该进程中所运行组件的状态。 下面，我们介绍决定终止进程所用的规则。

Android 系统将尽量长时间地保持应用进程，但为了新建进程或运行更重要的进程，最终需要移除旧进程来回收内存。 为了确定保留或终止哪些进程，系统会根据进程中正在运行的组件以及这些组件的状态，将每个进程放入“重要性层次结构”中。 必要时，系统会首先消除重要性最低的进程，然后是重要性略逊的进程，依此类推，以回收系统资源。

重要性层次结构一共有 5 级：

1. **前台进程**

用户当前操作所必需的进程。

* + 托管用户正在交互的 [Activity](https://developer.android.google.cn/reference/android/app/Activity.html)（已调用 [Activity](https://developer.android.google.cn/reference/android/app/Activity.html) 的 [onResume()](https://developer.android.google.cn/reference/android/app/Activity.html#onResume()) 方法）
  + 托管某个 [Service](https://developer.android.google.cn/reference/android/app/Service.html)，后者绑定到用户正在交互的 Activity
  + **托管正在“前台”运行的**[**Service**](https://developer.android.google.cn/reference/android/app/Service.html)**（服务已调用**[**startForeground()**](https://developer.android.google.cn/reference/android/app/Service.html#startForeground(int,%20android.app.Notification))**）**
  + 托管正执行一个生命周期回调的 [Service](https://developer.android.google.cn/reference/android/app/Service.html)（[onCreate()](https://developer.android.google.cn/reference/android/app/Service.html#onCreate())、[onStart()](https://developer.android.google.cn/reference/android/app/Service.html#onStart(android.content.Intent,%20int)) 或 [onDestroy()](https://developer.android.google.cn/reference/android/app/Service.html#onDestroy())）
  + 托管正执行其 [onReceive()](https://developer.android.google.cn/reference/android/content/BroadcastReceiver.html#onReceive(android.content.Context,%20android.content.Intent)) 方法的 [BroadcastReceiver](https://developer.android.google.cn/reference/android/content/BroadcastReceiver.html)

通常，在任意给定时间前台进程都为数不多。只有在内存不足以支持它们同时继续运行的情况下，系统才会终止它们。 此时，设备往往已达到内存分页状态，因此需要终止一些前台进程来确保用户界面正常响应。

1. **可见进程**

会影响用户在屏幕上所见内容的进程。

* + 托管不在前台、对用户可见的 [Activity](https://developer.android.google.cn/reference/android/app/Activity.html)（已调用其 [onPause()](https://developer.android.google.cn/reference/android/app/Activity.html#onPause()) 方法）。
  + 托管绑定到可见（或前台）Activity 的 [Service](https://developer.android.google.cn/reference/android/app/Service.html)。

除非为了维持所有前台进程同时运行而必须终止，否则系统不会终止这些进程。

1. **服务进程**

正在运行已使用 [startService()](https://developer.android.google.cn/reference/android/content/Context.html#startService(android.content.Intent)) 方法启动的服务的进程。尽管服务进程与用户所见内容没有直接关联，但是它们通常在执行一些用户关心的操作（例如，在后台播放音乐或从网络下载数据）。

除非内存不足以维持所有前台进程和可见进程同时运行，否则系统会让服务进程保持运行状态。

1. **后台进程**

包含目前对用户不可见的 Activity 的进程（已调用 Activity 的 [onStop()](https://developer.android.google.cn/reference/android/app/Activity.html#onStop()) 方法）。系统可能随时终止它们。 通常会有很多后台进程在运行，它们会保存在 **LRU** （最近最少使用）列表中，用户最近查看的 Activity 的进程最后一个被终止。如果某个 Activity 正确实现了生命周期方法，并保存了其当前状态，当用户导航回该 Activity 时，Activity 会恢复其所有可见状态。

1. **空进程**

不含任何活动应用组件的进程。保留这种进程的的唯一目的是用作缓存，以缩短下次在其中运行组件所需的启动时间。 为使总体系统资源在进程缓存和底层内核缓存之间保持平衡，系统往往会终止这些进程。

根据进程中当前活动组件的重要程度，Android 会将进程评定为它可能达到的最高级别。

此外，一个进程的级别可能会因其他进程对它的依赖而有所提高，即服务于另一进程的进程其级别永远不会低于其所服务的进程。 例如，如果进程 A 中的内容提供程序为进程 B 中的客户端提供服务，或者如果进程 A 中的服务绑定到进程 B 中的组件，则进程 A 始终被视为至少与进程 B 同样重要。

由于运行服务的进程其级别高于托管后台 Activity 的进程，因此启动长时间运行操作的 Activity 最好为该操作启动[服务](https://developer.android.google.cn/guide/components/services.html)，而不是简单地创建工作线程，当操作有可能比 Activity 更加持久时尤要如此。例如，正在将图片上传到网站的 Activity 应该启动服务来执行上传，这样一来，即使用户退出 Activity，仍可在后台继续执行上传操作。广播接收器也应使用服务，而不是简单地将耗时冗长的操作放入线程中。

线程

应用启动时，系统会为应用创建一个名为“主线程”的执行线程。 此线程非常重要，因为它负责将事件分派给相应的用户界面小部件，其中包括绘图事件。 此外，它也是应用与 Android UI 工具包组件（来自 [android.widget](https://developer.android.google.cn/reference/android/widget/package-summary.html) 和 [android.view](https://developer.android.google.cn/reference/android/view/package-summary.html)软件包的组件）进行交互的线程。因此，主线程有时也称为 UI 线程。

系统*不会*为每个组件实例创建单独的线程。运行于同一进程的所有组件均在 UI 线程中实例化，并且对每个组件的系统调用均由该线程进行分派。 因此，响应系统回调的方法（例如，报告用户操作的 [onKeyDown()](https://developer.android.google.cn/reference/android/view/View.html#onKeyDown(int,%20android.view.KeyEvent)) 或生命周期回调方法）始终在进程的 UI 线程中运行。

例如，当用户触摸屏幕上的按钮时，应用的 UI 线程会将触摸事件分派给小部件，而小部件反过来又设置其按下状态，并将失效请求发布到事件队列中。 UI 线程从队列中取消该请求并通知小部件应该重绘自身。

在应用执行繁重的任务以响应用户交互时，除非正确实现应用，否则这种单线程模式可能会导致性能低下。 具体地讲，如果 UI 线程需要处理所有任务，则执行耗时很长的操作（例如，网络访问或数据库查询）将会阻塞整个 UI。 一旦线程被阻塞，将无法分派任何事件，包括绘图事件。 从用户的角度来看，应用显示为挂起。 更糟糕的是，如果 UI 线程被阻塞超过几秒钟时间（目前大约是 5 秒钟），用户就会看到一个让人厌烦的“[应用无响应](http://developer.android.google.cn/guide/practices/responsiveness.html)”(ANR) 对话框。如果引起用户不满，他们可能就会决定退出并卸载此应用。

此外，Android UI 工具包*并非*线程安全工具包。因此，您不得通过工作线程操纵 UI，而只能通过 UI 线程操纵用户界面。 因此，Android 的单线程模式必须遵守两条规则：

1. 不要阻塞 UI 线程
2. 不要在 UI 线程之外访问 Android UI 工具包

工作线程

根据上述单线程模式，要保证应用 UI 的响应能力，关键是不能阻塞 UI 线程。 如果执行的操作不能很快完成，则应确保它们在单独的线程（“后台”或“工作”线程）中运行。

例如，以下代码演示了一个点击侦听器从单独的线程下载图像并将其显示在 [ImageView](https://developer.android.google.cn/reference/android/widget/ImageView.html) 中：

public void onClick(View v) {  
    new Thread(new Runnable() {  
        public void run() {  
            Bitmap b = loadImageFromNetwork("http://example.com/image.png");  
            mImageView.setImageBitmap(b);  
        }  
    }).start();  
}

乍看起来，这段代码似乎运行良好，因为它创建了一个新线程来处理网络操作。 但是，它违反了单线程模式的第二条规则：*不要在 UI 线程之外访问 Android UI 工具包* — 此示例从工作线程（而不是 UI 线程）修改了 [ImageView](https://developer.android.google.cn/reference/android/widget/ImageView.html)。 这可能导致出现不明确、不可预见的行为，但要跟踪此行为困难而又费时。

为解决此问题，Android 提供了几种途径来从其他线程访问 UI 线程。 以下列出了几种有用的方法：

* [Activity.runOnUiThread(Runnable)](https://developer.android.google.cn/reference/android/app/Activity.html#runOnUiThread(java.lang.Runnable))
* [View.post(Runnable)](https://developer.android.google.cn/reference/android/view/View.html#post(java.lang.Runnable))
* [View.postDelayed(Runnable, long)](https://developer.android.google.cn/reference/android/view/View.html#postDelayed(java.lang.Runnable,%20long))

例如，您可以通过使用 [View.post(Runnable)](https://developer.android.google.cn/reference/android/view/View.html#post(java.lang.Runnable)) 方法修复上述代码：

public void onClick(View v) {  
    new Thread(new Runnable() {  
        public void run() {  
            final Bitmap bitmap =  
                    loadImageFromNetwork("http://example.com/image.png");  
            mImageView.post(new Runnable() {  
                public void run() {  
                    mImageView.setImageBitmap(bitmap);  
                }  
            });  
        }  
    }).start();  
}

现在，上述实现属于线程安全型：在单独的线程中完成网络操作，而在 UI 线程中操纵 [ImageView](https://developer.android.google.cn/reference/android/widget/ImageView.html)。

但是，随着操作日趋复杂，这类代码也会变得复杂且难以维护。 要通过工作线程处理更复杂的交互，可以考虑在工作线程中使用 [Handler](https://developer.android.google.cn/reference/android/os/Handler.html) 处理来自 UI 线程的消息。当然，最好的解决方案或许是扩展 [AsyncTask](https://developer.android.google.cn/reference/android/os/AsyncTask.html) 类，此类简化了与 UI 进行交互所需执行的工作线程任务。

使用 AsyncTask

[AsyncTask](https://developer.android.google.cn/reference/android/os/AsyncTask.html) 允许对用户界面执行异步操作。 它会先阻塞工作线程中的操作，然后在 UI 线程中发布结果，而无需您亲自处理线程和/或处理程序。

要使用它，必须创建 [AsyncTask](https://developer.android.google.cn/reference/android/os/AsyncTask.html) 的子类并实现 [doInBackground()](https://developer.android.google.cn/reference/android/os/AsyncTask.html#doInBackground(Params...)) 回调方法，该方法将在后台线程池中运行。 要更新 UI，应该实现 [onPostExecute()](https://developer.android.google.cn/reference/android/os/AsyncTask.html#onPostExecute(Result)) 以传递 [doInBackground()](https://developer.android.google.cn/reference/android/os/AsyncTask.html#doInBackground(Params...)) 返回的结果并在 UI 线程中运行，以便您安全地更新 UI。 稍后，您可以通过从 UI 线程调用 [execute()](https://developer.android.google.cn/reference/android/os/AsyncTask.html#execute(Params...)) 来运行任务。

例如，您可以通过以下方式使用 [AsyncTask](https://developer.android.google.cn/reference/android/os/AsyncTask.html) 来实现上述示例：

public void onClick(View v) {  
    new DownloadImageTask().execute("http://example.com/image.png");  
}  
  
private class DownloadImageTask extends AsyncTask<String, Void, Bitmap> {  
    /\*\* The system calls this to perform work in a worker thread and  
      \* delivers it the parameters given to AsyncTask.execute() \*/  
    protected Bitmap doInBackground(String... urls) {  
        return loadImageFromNetwork(urls[0]);  
    }  
  
    /\*\* The system calls this to perform work in the UI thread and delivers  
      \* the result from doInBackground() \*/  
    protected void onPostExecute(Bitmap result) {  
        mImageView.setImageBitmap(result);  
    }  
}

现在 UI 是安全的，代码也得到简化，因为任务分解成了两部分：一部分应在工作线程内完成，另一部分应在 UI 线程内完成。

下面简要概述了 AsyncTask 的工作方法，但要全面了解如何使用此类，您应阅读 [AsyncTask](https://developer.android.google.cn/reference/android/os/AsyncTask.html) 参考文档：

* 可以使用泛型指定参数类型、进度值和任务最终值
* 方法 [doInBackground()](https://developer.android.google.cn/reference/android/os/AsyncTask.html#doInBackground(Params...)) 会在**工作线程上执行**
* [onPreExecute()](https://developer.android.google.cn/reference/android/os/AsyncTask.html#onPreExecute())、[onPostExecute()](https://developer.android.google.cn/reference/android/os/AsyncTask.html#onPostExecute(Result)) 和 [onProgressUpdate()](https://developer.android.google.cn/reference/android/os/AsyncTask.html#onProgressUpdate(Progress...)) **均在 UI 线程中调用**
* [doInBackground()](https://developer.android.google.cn/reference/android/os/AsyncTask.html#doInBackground(Params...)) 返回的值将发送到 [onPostExecute()](https://developer.android.google.cn/reference/android/os/AsyncTask.html#onPostExecute(Result))
* 您可以随时在 [doInBackground()](https://developer.android.google.cn/reference/android/os/AsyncTask.html#doInBackground(Params...)) 中调用[publishProgress()](https://developer.android.google.cn/reference/android/os/AsyncTask.html#publishProgress(Progress...))，以在 UI 线程中执行 [onProgressUpdate()](https://developer.android.google.cn/reference/android/os/AsyncTask.html#onProgressUpdate(Progress...))
* 您可以随时取消任何线程中的任务

**注意**：使用工作线程时可能会遇到另一个问题，即：[运行时配置变更](https://developer.android.google.cn/guide/topics/resources/runtime-changes.html)（例如，用户更改了屏幕方向）导致 Activity 意外重启，这可能会销毁工作线程。 要了解如何在这种重启情况下坚持执行任务，以及如何在 Activity 被销毁时正确地取消任务，请参阅[书架](http://code.google.com/p/shelves/)示例应用的源代码。

线程安全方法

在某些情况下，您实现的方法可能会从多个线程调用，因此编写这些方法时必须确保其满足线程安全的要求。

这一点主要适用于可以远程调用的方法，如[绑定服务](https://developer.android.google.cn/guide/components/bound-services.html)中的方法。如果对 [IBinder](https://developer.android.google.cn/reference/android/os/IBinder.html) 中所实现方法的调用源自运行 [IBinder](https://developer.android.google.cn/reference/android/os/IBinder.html) 的同一进程，则该方法在调用方的线程中执行。但是，如果调用源自其他进程，则该方法将在从线程池选择的某个线程中执行（而不是在进程的 UI 线程中执行），线程池由系统在与 [IBinder](https://developer.android.google.cn/reference/android/os/IBinder.html) 相同的进程中维护。 例如，即使服务的 [onBind()](https://developer.android.google.cn/reference/android/app/Service.html#onBind(android.content.Intent)) 方法将从服务进程的 UI 线程调用，在 [onBind()](https://developer.android.google.cn/reference/android/app/Service.html#onBind(android.content.Intent)) 返回的对象中实现的方法（例如，实现 RPC 方法的子类）仍会从线程池中的线程调用。 由于一个服务可以有多个客户端，因此可能会有多个池线程在同一时间使用同一 [IBinder](https://developer.android.google.cn/reference/android/os/IBinder.html) 方法。因此，[IBinder](https://developer.android.google.cn/reference/android/os/IBinder.html) 方法必须实现为线程安全方法。

同样，内容提供程序也可接收来自其他进程的数据请求。尽管 [ContentResolver](https://developer.android.google.cn/reference/android/content/ContentResolver.html) 和 [ContentProvider](https://developer.android.google.cn/reference/android/content/ContentProvider.html) 类隐藏了如何管理进程间通信的细节，但响应这些请求的 [ContentProvider](https://developer.android.google.cn/reference/android/content/ContentProvider.html) 方法（[query()](https://developer.android.google.cn/reference/android/content/ContentProvider.html#query(android.net.Uri,%20java.lang.String[],%20android.os.Bundle,%20android.os.CancellationSignal))、[insert()](https://developer.android.google.cn/reference/android/content/ContentProvider.html#insert(android.net.Uri,%20android.content.ContentValues))、[delete()](https://developer.android.google.cn/reference/android/content/ContentProvider.html#delete(android.net.Uri,%20java.lang.String,%20java.lang.String[]))、[update()](https://developer.android.google.cn/reference/android/content/ContentProvider.html#update(android.net.Uri,%20android.content.ContentValues,%20java.lang.String,%20java.lang.String[])) 和 [getType()](https://developer.android.google.cn/reference/android/content/ContentProvider.html#getType(android.net.Uri)) 方法）将从内容提供程序所在进程的线程池中调用，而不是从进程的 UI 线程调用。 由于这些方法可能会同时从任意数量的线程调用，因此它们也必须实现为线程安全方法。

进程间通信

Android 利用远程过程调用 (RPC) 提供了一种进程间通信 (IPC) 机制，通过这种机制，由 Activity 或其他应用组件调用的方法将（在其他进程中）远程执行，而所有结果将返回给调用方。 这就要求把方法调用及其数据分解至操作系统可以识别的程度，并将其从本地进程和地址空间传输至远程进程和地址空间，然后在远程进程中重新组装并执行该调用。 然后，返回值将沿相反方向传输回来。 Android 提供了执行这些 IPC 事务所需的全部代码，因此您只需集中精力定义和实现 RPC 编程接口即可。

要执行 IPC，必须使用 [bindService()](https://developer.android.google.cn/reference/android/content/Context.html#bindService(android.content.Intent,%20android.content.ServiceConnection,%20int)) 将应用绑定到服务上。

后台执行限制

为了提升用户体验，Android 8.0 对应用在后台运行时可以执行的操作施加了限制。

概览

多个 Android 应用和服务可以同时运行。 例如，用户可以在一个窗口中玩游戏，同时在另一个窗口中浏览网页，并使用第三个应用播放音乐。

同时运行的应用越多，对系统造成的负担越大。 如果还有应用或服务在后台运行，这会对系统造成更大负担，进而可能导致用户体验下降；例如，音乐应用可能会突然关闭。

为了降低发生这些问题的几率，Android 8.0 对应用在用户不与其直接交互时可以执行的操作施加了限制。

应用在两个方面受到限制：

* [后台服务限制](https://developer.android.google.cn/about/versions/oreo/background#services)：处于空闲状态时，应用可以使用的后台服务存在限制。 这些限制不适用于前台服务，因为前台服务更容易引起用户注意。
* [广播限制](https://developer.android.google.cn/about/versions/oreo/background#broadcasts)：除了有限的例外情况，应用无法使用清单注册隐式广播。 它们仍然可以在运行时注册这些广播，并且可以使用清单注册专门针对它们的显式广播。

注：默认情况下，这些限制仅适用于针对 O 的应用。 不过，用户可以从 **Settings** 屏幕为任意应用启用这些限制，即使应用并不是以 O 为目标平台。

在大多数情况下，应用都可以使用 [JobScheduler](https://developer.android.google.cn/reference/android/app/job/JobScheduler.html) 作业克服这些限制。 这种方式让应用安排为在未活跃运行时执行工作，不过仍能够使系统可以在不影响用户体验的情况下安排这些作业。

Android 8.0 提供针对 [JobScheduler](https://developer.android.google.cn/reference/android/app/job/JobScheduler.html) 的多个改进，让您可以更轻松地使用计划作业取代服务和广播接收器；如需了解详细信息，请参阅 [JobScheduler 改进](https://developer.android.google.cn/preview/api-overview.html#jobscheduler)。

后台服务限制

在后台中运行的服务会消耗设备资源，这可能降低用户体验。 为了缓解这一问题，系统对这些服务施加了一些限制。

系统可以区分 *前台* 和 *后台* 应用。 （用于服务限制目的的后台定义与[内存管理](https://developer.android.google.cn/topic/performance/memory-overview.html)使用的定义不同；一个应用按照内存管理的定义可能处于后台，但按照能够启动服务的定义又处于前台。）如果满足以下任意条件，应用将被视为处于前台：

* 具有可见 Activity（不管该 Activity 已启动还是已暂停）。
* 具有前台服务。
* 另一个前台应用已关联到该应用（不管是通过绑定到其中一个服务，还是通过使用其中一个内容提供程序）。 例如，如果另一个应用绑定到该应用的服务，那么该应用处于前台：
  + [IME](https://developer.android.google.cn/guide/topics/text/creating-input-method.html)
  + 壁纸服务
  + 通知侦听器
  + 语音或文本服务

如果以上条件均不满足，应用将被视为处于后台。

绑定服务不受影响

这些规则不会对绑定服务产生任何影响。 如果您的应用定义了绑定服务，则不管应用是否处于前台，其他组件都可以绑定到该服务。

处于前台时，应用可以自由创建和运行前台服务与后台服务。 进入后台时，在一个持续数分钟的时间窗内，应用仍可以创建和使用服务。

在该时间窗结束后，应用将被视为处于 *空闲* 状态。 此时，系统将停止应用的后台服务，就像应用已经调用服务的“[Service.stopSelf()](https://developer.android.google.cn/reference/android/app/Service.html#stopSelf())”方法。

在这些情况下，后台应用将被置于一个临时白名单中并持续数分钟。 位于白名单中时，应用可以无限制地启动服务，并且其后台服务也可以运行。

处理对用户可见的任务时，应用将被置于白名单中，例如：

* 处理一条高优先级 [Firebase 云消息传递 (FCM)](https://firebase.google.cn/docs/cloud-messaging/) 消息。
* 接收广播，例如短信/彩信消息。
* 从通知执行 [PendingIntent](https://developer.android.google.cn/reference/android/app/PendingIntent.html)。

在很多情况下，您的应用都可以使用 [JobScheduler](https://developer.android.google.cn/reference/android/app/job/JobScheduler.html) 作业替换后台服务。 例如，CoolPhotoApp 需要检查用户是否已经从朋友那里收到共享的照片，即使该应用未在前台运行。

之前，应用使用一种会检查其云存储的后台服务。 为了迁移到 Android 8.0，开发者使用一个计划作业替换了这种后台服务，该作业将按一定周期启动，查询服务器，然后退出。

在 Android 8.0 之前，创建前台服务的方式通常是先创建一个后台服务，然后将该服务推到前台。

Android 8.0 有一项复杂功能；系统不允许后台应用创建后台服务。 因此，Android 8.0 引入了一种全新的方法，即 Context.startForegroundService()，以在前台启动新服务。

在系统创建服务后，应用有五秒的时间来调用该服务的 [startForeground()](https://developer.android.google.cn/reference/android/app/Service.html#startForeground(int, android.app.Notification)) 方法以显示新服务的用户可见通知。

如果应用在此时间限制内*未*调用 [startForeground()](https://developer.android.google.cn/reference/android/app/Service.html#startForeground(int, android.app.Notification))，则系统将停止服务并声明此应用为 [ANR](https://developer.android.google.cn/training/articles/perf-anr.html)。

广播限制

如果应用注册为接收广播，则在每次发送广播时，应用的接收器都会消耗资源。 如果多个应用注册为接收基于系统事件的广播，这会引发问题；触发广播的系统事件会导致所有应用快速地连续消耗资源，从而降低用户体验。

为了缓解这一问题，Android 7.0（API 级别 25）对广播施加了一些限制，如[后台优化](https://developer.android.google.cn/topic/performance/background-optimization.html)中所述。

Android 8.0 让这些限制更为严格。

* 针对 Android 8.0 的应用无法继续在其清单中为隐式广播注册广播接收器。 *隐式广播*是一种不专门针对该应用的广播。 例如，[ACTION\_PACKAGE\_REPLACED](https://developer.android.google.cn/reference/android/content/Intent.html#ACTION_PACKAGE_REPLACED) 就是一种隐式广播，因为它将发送到注册的所有侦听器，让后者知道设备上的某些软件包已被替换。

不过，[ACTION\_MY\_PACKAGE\_REPLACED](https://developer.android.google.cn/reference/android/content/Intent.html#ACTION_MY_PACKAGE_REPLACED) 不是隐式广播，因为不管已为该广播注册侦听器的其他应用有多少，它都会只发送到软件包已被替换的应用。

* 应用可以继续在它们的清单中注册显式广播。
* 应用可以在运行时使用 [Context.registerReceiver()](https://developer.android.google.cn/reference/android/content/Context.html#registerReceiver(android.content.BroadcastReceiver, android.content.IntentFilter)) 为任意广播（不管是隐式还是显式）注册接收器。
* 需要[签名权限](https://developer.android.google.cn/guide/topics/manifest/permission-element.html#plevel)的广播不受此限制所限，因为这些广播只会发送到使用相同证书签名的应用，而不是发送到设备上的所有应用。

在许多情况下，之前注册隐式广播的应用使用 [JobScheduler](https://developer.android.google.cn/reference/android/app/job/JobScheduler.html) 作业可以获得类似的功能。

例如，一款社交照片应用可能需要不时地执行数据清理，并且倾向于在设备连接到充电器时执行此操作。

之前，应用已经在清单中为 [ACTION\_POWER\_CONNECTED](https://developer.android.google.cn/reference/android/content/Intent.html#ACTION_POWER_CONNECTED) 注册了一个接收器；当应用接收到该广播时，它会检查清理是否必要。 为了迁移到 Android 8.0，应用将该接收器从其清单中移除。

应用将清理作业安排在设备处于空闲状态和充电时运行。

**注**：很多隐式广播当前均已不受此限制所限。 应用可以继续在其清单中为这些广播注册接收器，不管应用针对哪个 API 级别。 有关已豁免广播的列表，请参阅[隐式广播例外](https://developer.android.google.cn/about/versions/oreo/background-broadcasts.html)。

迁移指南

默认情况下，这些更改仅影响针对 O 的应用。 不过，用户可以从 **Settings** 屏幕为任意应用启用这些限制，即使应用并不是以 O 为目标平台。

您可能需要更新应用，使其符合新限制。

了解您的应用如何使用服务。 如果您的应用依赖某些在它处于空闲时于后台运行的服务，您需要替换这些服务。

可能的解决方法包括：

* 如果处于后台时您的应用需要创建一个前台服务，请使用新的 NotificationManager.startServiceInForeground()

方法，而不是创建一个后台服务，然后尝试将其推到前台。

* 如果服务容易被用户注意，请将其设为前台服务。 例如，播放音频的服务始终应为前台服务。

使用 NotificationManager.startServiceInForeground()

而不是 [startService()](https://developer.android.google.cn/reference/android/content/Context.html#startService(android.content.Intent)) 创建服务。

* 寻找一种使用计划作业实现服务功能的方式。 如果服务未在执行容易立即被用户注意到的操作，一般情况下，您都能够使用计划作业。
* 发生网络事件时，请使用 [FCM](https://firebase.google.cn/docs/cloud-messaging/) 选择性地唤醒您的应用，而不是在后台轮询。
* 在应用正常处于前台之前，请推迟后台工作。

检查在您应用的清单中定义的广播接收器。 如果您的清单为显式广播声明了接收器，您必须予以替换。 可能的解决方法包括：

* 通过调用 [Context.registerReceiver()](https://developer.android.google.cn/reference/android/content/Context.html#registerReceiver(android.content.BroadcastReceiver, android.content.IntentFilter)) 而不是在清单中声明接收器的方式在运行时创建接收器。
* 使用计划作业检查条件是否会触发隐式广播。

后台位置限制

为降低功耗，**无论应用的目标 SDK 版本为何**，Android 8.0 都会对后台应用检索用户当前位置的频率进行限制。

如果您的应用在后台运行时依赖实时提醒或运动检测，这一位置检索行为就显得特别重要，必须紧记。

**重要说明**：作为起点，我们只允许后台应用每小时接收几次位置更新。我们将在整个预览版阶段继续根据系统影响和开发者的反馈优化位置更新间隔。

系统会对前台应用和后台应用进行区分。应用满足以下任一条件即视为前台应用：

* 它具有可见的 Activity，无论 Activity 处于启动还是暂停状态。
* 它具有前台服务。
* 另一个前台应用通过绑定到应用的其中一个服务或使用应用的其中一个内容提供程序与应用相连。

如果以上所有条件均不满足，应用即视为后台应用。

前台应用行为得到保留

如果应用在运行 Android 8.0 的设备上处于前台，其位置更新行为将与 Android 7.1.1（API 级别 25）及更低版本上相同。

**警告**：如果您的应用长时间进行近乎实时的位置更新检索，将大幅度缩短设备的电池寿命。

优化应用的位置行为

考虑在您的应用接收位置更新不频繁的情况下其后台运行用例是否根本无法成功。如果属于这种情况，您可以通过执行下列操作之一提高位置更新的检索频率：

* 将您的应用转至前台。
* 使用应用中的某个[前台服务](https://developer.android.google.cn/guide/components/services.html#Foreground)。激活此服务时，您的应用必须在[通知区](https://developer.android.google.cn/guide/topics/ui/notifiers/notifications.html)显示一个持续性的通知。
* 使用 Geofencing API 的元素（例如 [GeofencingApi](https://developers.google.cn/android/reference/com/google/android/gms/location/GeofencingApi) 接口），这些元素针对最大限度减少耗电进行了专门优化。
* 使用被动位置侦听器，它可以在后台应用加快位置请求频率时提高位置更新的接收频率。

**注**：如果您的应用需要访问的位置历史记录包含时间频繁更新，请使用批处理版本的 Fused Location Provider API 元素，例如 [**FusedLocationProviderApi**](https://developers.google.cn/android/reference/com/google/android/gms/location/FusedLocationProviderApi) 接口。当您的应用运行于后台时，此 API 会以高于非批处理版本 API 的频率接收用户的位置。但切记，您的应用批量接收更新的频率仍仅为每小时几次。

受影响的 API

对后台应用位置检索行为的更改影响下列 API：

[**Fused Location Provider (FLP)**](https://developers.google.cn/android/reference/com/google/android/gms/location/FusedLocationProviderApi)

* 如果您的应用运行在后台，位置系统服务只会根据 Android 8.0 行为变更中定义的间隔，按每小时几次的频率为其计算新位置。即使您的应用请求进行更频繁的位置更新，也仍是如此。
* 如果您的应用运行在前台，与 Android 7.1.1（API 级别 25）相比，在位置采样率上不会有任何变化。

**Geofencing**

* 后台应用可以高于接收 Fused Location Provider 更新的频率接收地理围栏转换事件。
* 地理围栏事件的平均响应时间是大约每两分钟一次。

**GNSS Measurements 和 GNSS Navigation Messages**

* 当您的应用位于后台时，注册用于接收 [GnssMeasurement](https://developer.android.google.cn/reference/android/location/GnssMeasurement.html) 和 [GnssNavigationMessage](https://developer.android.google.cn/reference/android/location/GnssNavigationMessage.html) 输出的回调会停止执行。

**Location Manager**

* 提供给后台应用的位置更新只会根据 Android 8.0 行为变更中定义的间隔，按每小时几次的频率提供。

**注**：如果运行您的应用的设备安装了 Google Play 服务，强烈建议您改用 [Fused Location Provider (FLP)](https://developers.google.cn/android/reference/com/google/android/gms/location/FusedLocationProviderApi)。

**WLAN 管理器**

[startScan()](https://developer.android.google.cn/reference/android/net/wifi/WifiManager.html#startScan()) 方法对后台应用执行完整扫描的频率仅为每小时数次。如果不久之后后台应用再次调用此方法， [WifiManager](https://developer.android.google.cn/reference/android/net/wifi/WifiManager.html) 类将提供上次扫描所缓存的结果。

# 网上资料

# 整理

# 系统API

获取可用内存大小

ActivityManager am = *getService*(context, ***ACTIVITY\_SERVICE***);  
**boolean** largeHeap = (context.getApplicationInfo().**flags** & ***FLAG\_LARGE\_HEAP***) != 0;  
**int** memoryClass = largeHeap ? am.getLargeMemoryClass() : am.getMemoryClass();

# Fragment

Fragment的主要类

Fragment与activity的关系

**Fragment的栈管理**

**Fragment回退按键事件传递流程。**

## fragment事务

### 涉及到的类

* [Fragment](https://link.jianshu.com/?t=https://developer.android.com/reference/android/app/Fragment.html)
* [FragmentTransaction](https://link.jianshu.com/?t=https://developer.android.com/reference/android/app/FragmentTransaction.html)
* [FragmentManager](https://link.jianshu.com/?t=https://developer.android.com/reference/android/app/FragmentManager.html)

### 什么是Fragment

Fragment是Android3.0引入的API，号称是为了解决屏幕碎片化和帮助重用代码的构造。中文翻译为碎片、片段，或者理解成子Act，相比Activity更轻量级、更灵活，使用姿势一般有以下2种：

1. 静态写在xml中，就和我们使用一般的widget一样，如下：



写在act的xml布局文件中

1. 在代码里动态生成，添加、删除等等，代码如下：



通过代码添加

参考[fragment开发文档](https://link.jianshu.com/?t=https://developer.android.com/guide/components/fragments.html)。

### 什么是FragmentTransaction

它封装了一系列对fragment的操作，并一次性执行这些操作。看眼它的API就能明白，全是各种add/remove/replace/show/hide等等操作。

### 什么是FragmentManager

它是和某个act相关联的，并不是全局唯一的，而是每个act都有一个自己的FragmentManager，内部有自己的状态mCurState，对应外部act的生命周期状态。它提供和act中fragment交互的API。

### 示例代码

private void addTestFragment() {

FragmentManager fm = getFragmentManager();

Fragment fragment = TestFragment.newInstance();

FragmentTransaction ft = fm.beginTransaction();

*// ft.add(R.id.pf\_base\_fragment\_container, fragment, "test\_fragment");*

ft.replace(R.id.pf\_base\_fragment\_container, fragment);

ft.addToBackStack(null);

ft.commit();

}

之后的源码解析都按照这段示例代码来讲，源码基于Android6.0。

FragmentManager、FragmentTransaction都是abstract的：





### 具体分析

通过FragmentTransaction这样的API，我们可以一次执行多个操作，就像这样：

ft.remove(R.id.fragment\_container, fragment1);

ft.add(R.id.fragment\_container, fragment2);

ft.hide(fragment3);

...等等还可以写很长

它是怎么做到的呢？关键是其内部的双向链表结构：



Op的定义

接下来我们看下在此结构上，FragmentTransaction的一堆操作方法的实现，代码如下：

void addOp(Op op) {

*// 典型的双向链表插入操作*

if (mHead == null) {

mHead = mTail = op;

} else {

op.prev = mTail;

mTail.next = op;

mTail = op;

}

*// 记录这些动画信息*

op.enterAnim = mEnterAnim;

op.exitAnim = mExitAnim;

op.popEnterAnim = mPopEnterAnim;

op.popExitAnim = mPopExitAnim;

mNumOp++;

}

fragmentTransaction.add最终都调用了 doAddOp(0, fragment, tag, OP\_ADD);

private void doAddOp(int containerViewId, Fragment fragment, String tag, int opcmd) {

fragment.mFragmentManager = mManager;

if (tag != null) {

if (fragment.mTag != null && !tag.equals(fragment.mTag)) {

throw new IllegalStateException("Can't change tag of fragment "

+ fragment + ": was " + fragment.mTag

+ " now " + tag);

}

*// 记录这些信息到fragment对象上*

fragment.mTag = tag;

}

if (containerViewId != 0) {

if (fragment.mFragmentId != 0 && fragment.mFragmentId != containerViewId) {

throw new IllegalStateException("Can't change container ID of fragment "

+ fragment + ": was " + fragment.mFragmentId

+ " now " + containerViewId);

}

*// 记录这些信息到fragment对象上*

fragment.mContainerId = fragment.mFragmentId = containerViewId;

}

Op op = new Op();

op.cmd = opcmd;

op.fragment = fragment;

addOp(op);

}

fragmentTransaction.replace 最后调用 doAddOp(containerViewId, fragment, tag, OP\_REPLACE);

public FragmentTransaction remove(Fragment fragment) {

Op op = new Op();

op.cmd = OP\_REMOVE;

op.fragment = fragment;

addOp(op);

return this;

}

public FragmentTransaction hide(Fragment fragment) {

Op op = new Op();

op.cmd = OP\_HIDE;

op.fragment = fragment;

addOp(op);

return this;

}

public FragmentTransaction show(Fragment fragment) {

Op op = new Op();

op.cmd = OP\_SHOW;

op.fragment = fragment;

addOp(op);

return this;

}

public FragmentTransaction detach(Fragment fragment) {

Op op = new Op();

op.cmd = OP\_DETACH;

op.fragment = fragment;

addOp(op);

return this;

}

public FragmentTransaction attach(Fragment fragment) {

Op op = new Op();

op.cmd = OP\_ATTACH;

op.fragment = fragment;

addOp(op);

return this;

}

public FragmentTransaction setCustomAnimations(int enter, int exit) {

return setCustomAnimations(enter, exit, 0, 0);

}

public FragmentTransaction setCustomAnimations(int enter, int exit,

int popEnter, int popExit) {

mEnterAnim = enter;

mExitAnim = exit;

mPopEnterAnim = popEnter;

mPopExitAnim = popExit;

return this;

}

public FragmentTransaction setTransition(int transition) {

mTransition = transition;

return this;

}

@Override

public FragmentTransaction addSharedElement(View sharedElement, String name) {

String transitionName = sharedElement.getTransitionName();

if (transitionName == null) {

throw new IllegalArgumentException("Unique transitionNames are required for all" +

" sharedElements");

}

if (mSharedElementSourceNames == null) {

mSharedElementSourceNames = new ArrayList<String>();

mSharedElementTargetNames = new ArrayList<String>();

}

mSharedElementSourceNames.add(transitionName);

mSharedElementTargetNames.add(name);

return this;

}

public FragmentTransaction setTransitionStyle(int styleRes) {

mTransitionStyle = styleRes;

return this;

}

public FragmentTransaction addToBackStack(String name) {

if (!mAllowAddToBackStack) {

throw new IllegalStateException("This FragmentTransaction is not allowed to be added to the back stack.");

}

mAddToBackStack = true;

mName = name;

return this;

}

从源码我们可以看出，各种不同操作方法内部实际上只是个不同cmd的Op对象，通过双向链表链起来了而已，至于为什么要是双向的呢，单链表不行吗？那是因为Fragment在act层面维护了一个回退栈，即如果你调用了ft.addToBackStack(null);这样的代码，那么当你按下back键的时候不是直接结束当前act，而是先从fragment栈里尝试pop出来栈顶的fragment。它不仅要向前进，还得支持向后退。  
紧接着的是commit的源码，如下：

fragmentTransaction.commitXXX两个方法都调用了commitInternal

int commitInternal(boolean allowStateLoss) {

if (mCommitted) {

throw new IllegalStateException("commit already called");

}

mCommitted = true;

if (mAddToBackStack) {

**mIndex = mManager.allocBackStackIndex(this);**

} else {

mIndex = -1;

}

mManager.enqueueAction(this, allowStateLoss);

return mIndex;

}

*// FragmentManagerImpl#enqueueAction方法*

*/\*\**

*\* Adds an action to the queue of pending actions.*

*\**

*\* @param action the action to add*

*\* @param allowStateLoss whether to allow loss of state information*

*\* @throws IllegalStateException if the activity has been destroyed*

*\*/*

public void enqueueAction(Runnable action, boolean allowStateLoss) {

if (!allowStateLoss) {

checkStateLoss();

}

synchronized (this) {

if (mDestroyed || mHost == null) {

throw new IllegalStateException("Activity has been destroyed");

}

if (mPendingActions == null) {

mPendingActions = new ArrayList<Runnable>();

}

mPendingActions.add(action);

if (mPendingActions.size() == 1) {

mHost.getHandler().removeCallbacks(mExecCommit);

mHost.getHandler().post(mExecCommit);

}

}

}

我们看到commit方法并没有立即执行这个动作，只是入队了action（通过mPendingActions.add(action);），系统会在下次event loop到来时执行它。

FragmentManagerImpl的其他关键方法

Runnable mExecCommit = new Runnable() {

@Override

public void run() {

execPendingActions();

}

};

*/\*\**

*\* Only call from main thread!*

*\*/*

public boolean execPendingActions() {

if (mExecutingActions) {

throw new IllegalStateException("Recursive entry to executePendingTransactions");

}

if (Looper.myLooper() != mHost.getHandler().getLooper()) {

throw new IllegalStateException("Must be called from main thread of process");

}

boolean didSomething = false;

while (true) {

int numActions;

synchronized (this) {

if (mPendingActions == null || mPendingActions.size() == 0) {

break;

}

numActions = mPendingActions.size();

if (mTmpActions == null || mTmpActions.length < numActions) {

mTmpActions = new Runnable[numActions];

}

mPendingActions.toArray(mTmpActions);

mPendingActions.clear();

mHost.getHandler().removeCallbacks(mExecCommit);

}

mExecutingActions = true;

for (int i=0; i<numActions; i++) {

mTmpActions[i].run();

mTmpActions[i] = null;

}

mExecutingActions = false;

didSomething = true;

}

if (mHavePendingDeferredStart) {

boolean loadersRunning = false;

for (int i=0; i<mActive.size(); i++) {

Fragment f = mActive.get(i);

if (f != null && f.mLoaderManager != null) {

loadersRunning |= f.mLoaderManager.hasRunningLoaders();

}

}

if (!loadersRunning) {

mHavePendingDeferredStart = false;

startPendingDeferredFragments();

}

}

return didSomething;

}

具体要执行的action就是BackStackRecord#run方法，因为我们注意到它自己实现了Runnable接口，来看下其实现：

public void run() {

if (mAddToBackStack) {

if (mIndex < 0) {

throw new IllegalStateException("addToBackStack() called after commit()");

}

}

bumpBackStackNesting(1);

SparseArray<Fragment> firstOutFragments = new SparseArray<Fragment>();

SparseArray<Fragment> lastInFragments = new SparseArray<Fragment>();

calculateFragments(firstOutFragments, lastInFragments);

beginTransition(firstOutFragments, lastInFragments, false);

Op op = mHead;

while (op != null) {

switch (op.cmd) {

case OP\_ADD: {

Fragment f = op.fragment;

f.mNextAnim = op.enterAnim;

mManager.addFragment(f, false);

}

break;

case OP\_REPLACE: {

Fragment f = op.fragment;

int containerId = f.mContainerId;

if (mManager.mAdded != null) {

for (int i = 0; i < mManager.mAdded.size(); i++) {

Fragment old = mManager.mAdded.get(i);

*// 注意这个判断，必须是同一个containerId里面的，换句话说你如果是在2个不同的container里面做fragment的replace操作，互相是不会影响的*

if (old.mContainerId == containerId) {

if (old == f) {

op.fragment = f = null;

} else {

if (op.removed == null) {

op.removed = new ArrayList<Fragment>();

}

op.removed.add(old);

old.mNextAnim = op.exitAnim;

if (mAddToBackStack) {

**old.mBackStackNesting += 1;**

}

mManager.removeFragment(old, mTransition, mTransitionStyle);

}

}

}

}

*// 注意这个非空判断，因为replace自己的时候f会被置为null，相当于什么也不会做的*

if (f != null) {

f.mNextAnim = op.enterAnim;

mManager.addFragment(f, false);

}

}

break;

case OP\_REMOVE: {

Fragment f = op.fragment;

f.mNextAnim = op.exitAnim;

mManager.removeFragment(f, mTransition, mTransitionStyle);

}

break;

case OP\_HIDE: {

Fragment f = op.fragment;

f.mNextAnim = op.exitAnim;

mManager.hideFragment(f, mTransition, mTransitionStyle);

}

break;

case OP\_SHOW: {

Fragment f = op.fragment;

f.mNextAnim = op.enterAnim;

mManager.showFragment(f, mTransition, mTransitionStyle);

}

break;

case OP\_DETACH: {

Fragment f = op.fragment;

f.mNextAnim = op.exitAnim;

mManager.detachFragment(f, mTransition, mTransitionStyle);

}

break;

case OP\_ATTACH: {

Fragment f = op.fragment;

f.mNextAnim = op.enterAnim;

mManager.attachFragment(f, mTransition, mTransitionStyle);

}

break;

default: {

throw new IllegalArgumentException("Unknown cmd: " + op.cmd);

}

}

op = op.next;

}

*// 不带fragment参数的状态转移*

mManager.moveToState(mManager.mCurState, mTransition,

mTransitionStyle, true);

if (mAddToBackStack) {

**mManager.addBackStackState(this);**

}

}

这个方法主要做的事情是向前遍历Op双向链表，根据cmd的不同调用FragmentManagerImpl的以下方法，特别注意下这里的OP\_REPLACE cmd，看代码我们知道它其实相当于add/remove的组合，即如果有added会把所以已经added了的先remove掉，再将新的add进去，效果上是这样。

public void addFragment(Fragment fragment, boolean moveToStateNow) {

if (mAdded == null) {

mAdded = new ArrayList<Fragment>();

}

**makeActive(fragment);**

if (!fragment.mDetached) {

if (mAdded.contains(fragment)) {

throw new IllegalStateException("Fragment already added: " + fragment);

}

mAdded.add(fragment);

fragment.mAdded = true;

fragment.mRemoving = false;

if (fragment.mHasMenu && fragment.mMenuVisible) {

mNeedMenuInvalidate = true;

}

if (moveToStateNow) {

moveToState(fragment);

}

}

}

public void removeFragment(Fragment fragment, int transition, int transitionStyle) {

*// 加到回退栈中和不加的区别*

final boolean inactive = !fragment.isInBackStack();

if (!fragment.mDetached || inactive) {

if (false) {

*// Would be nice to catch a bad remove here, but we need*

*// time to test this to make sure we aren't crashes cases*

*// where it is not a problem.*

if (!mAdded.contains(fragment)) {

throw new IllegalStateException("Fragment not added: " + fragment);

}

}

if (mAdded != null) {

mAdded.remove(fragment);

}

if (fragment.mHasMenu && fragment.mMenuVisible) {

mNeedMenuInvalidate = true;

}

fragment.mAdded = false;

fragment.mRemoving = true;

*// 没有加到回退栈中则状态退的更彻底，到状态0而不是状态1，区别就是最后的几个destory相关的callback会不会被调用。*

moveToState(fragment, inactive ? Fragment.INITIALIZING : Fragment.CREATED,

transition, transitionStyle, false);

}

}

public void hideFragment(Fragment fragment, int transition, int transitionStyle) {

if (!fragment.mHidden) {

fragment.mHidden = true;

if (fragment.mView != null) {

Animator anim = loadAnimator(fragment, transition, false,

transitionStyle);

if (anim != null) {

anim.setTarget(fragment.mView);

*// Delay the actual hide operation until the animation finishes, otherwise*

*// the fragment will just immediately disappear*

final Fragment finalFragment = fragment;

anim.addListener(new AnimatorListenerAdapter() {

@Override

public void onAnimationEnd(Animator animation) {

if (finalFragment.mView != null) {

finalFragment.mView.setVisibility(View.GONE);

}

}

});

setHWLayerAnimListenerIfAlpha(finalFragment.mView, anim);

anim.start();

} else {

fragment.mView.setVisibility(View.GONE);

}

}

if (fragment.mAdded && fragment.mHasMenu && fragment.mMenuVisible) {

mNeedMenuInvalidate = true;

}

fragment.onHiddenChanged(true);

}

}

public void showFragment(Fragment fragment, int transition, int transitionStyle) {

if (fragment.mHidden) {

fragment.mHidden = false;

if (fragment.mView != null) {

Animator anim = loadAnimator(fragment, transition, true,

transitionStyle);

if (anim != null) {

anim.setTarget(fragment.mView);

setHWLayerAnimListenerIfAlpha(fragment.mView, anim);

anim.start();

}

fragment.mView.setVisibility(View.VISIBLE);

}

if (fragment.mAdded && fragment.mHasMenu && fragment.mMenuVisible) {

mNeedMenuInvalidate = true;

}

fragment.onHiddenChanged(false);

}

}

public void detachFragment(Fragment fragment, int transition, int transitionStyle) {

if (!fragment.mDetached) {

fragment.mDetached = true;

if (fragment.mAdded) {

*// We are not already in back stack, so need to remove the fragment.*

if (mAdded != null) {

mAdded.remove(fragment);

}

if (fragment.mHasMenu && fragment.mMenuVisible) {

mNeedMenuInvalidate = true;

}

fragment.mAdded = false;

moveToState(fragment, Fragment.CREATED, transition, transitionStyle, false);

}

}

}

public void attachFragment(Fragment fragment, int transition, int transitionStyle) {

if (fragment.mDetached) {

fragment.mDetached = false;

if (!fragment.mAdded) {

if (mAdded == null) {

mAdded = new ArrayList<Fragment>();

}

if (mAdded.contains(fragment)) {

throw new IllegalStateException("Fragment already added: " + fragment);

}

mAdded.add(fragment);

fragment.mAdded = true;

if (fragment.mHasMenu && fragment.mMenuVisible) {

mNeedMenuInvalidate = true;

}

moveToState(fragment, mCurState, transition, transitionStyle, false);

}

}

}

上面的run方法在对这一系列Op处理完后，调用了下面的方法：  
mManager.moveToState(mManager.mCurState, mTransition, mTransitionStyle, true);，对应的源码如下：

void moveToState(Fragment f) {

moveToState(f, mCurState, 0, 0, false);

}

void moveToState(int newState, boolean always) {

moveToState(newState, 0, 0, always);

}

*// 这里调用的是这个方法*

void moveToState(int newState, int transit, int transitStyle, boolean always) {

if (mHost == null && newState != Fragment.INITIALIZING) {

throw new IllegalStateException("No activity");

}

if (!always && mCurState == newState) {

return;

}

mCurState = newState;

if (mActive != null) {

boolean loadersRunning = false;

for (int i=0; i<mActive.size(); i++) {

Fragment f = mActive.get(i);

if (f != null) {

*// fragment状态改变的方法，其各种生命周期回调也在这里调用，是最重要的一个方法！！！*

moveToState(f, newState, transit, transitStyle, false);

if (f.mLoaderManager != null) {

loadersRunning |= f.mLoaderManager.hasRunningLoaders();

}

}

}

if (!loadersRunning) {

startPendingDeferredFragments();

}

if (mNeedMenuInvalidate && mHost != null && mCurState == Fragment.RESUMED) {

mHost.onInvalidateOptionsMenu();

mNeedMenuInvalidate = false;

}

}

}

这里我们需要对mManager.mCurState这个值的变化细说下，其初始值是这样的：

int mCurState = Fragment.INITIALIZING;

Fragment的所有可能状态：

static final int INVALID\_STATE = -1; *// Invalid state used as a null value.*

static final int INITIALIZING = 0; *// Not yet created.*

static final int CREATED = 1; *// Created.*

static final int ACTIVITY\_CREATED = 2; *// The activity has finished its creation.*

static final int STOPPED = 3; *// Fully created, not started.*

static final int STARTED = 4; *// Created and started, not resumed.*

static final int RESUMED = 5; *// Created started and resumed.*

FragmentManager中有一堆public void dispatchxxx这样的方法，如下：

public void dispatchCreate() {

mStateSaved = false;

moveToState(Fragment.CREATED, false);

}

public void dispatchActivityCreated() {

mStateSaved = false;

moveToState(Fragment.ACTIVITY\_CREATED, false);

}

public void dispatchStart() {

mStateSaved = false;

moveToState(Fragment.STARTED, false);

}

public void dispatchResume() {

mStateSaved = false;

moveToState(Fragment.RESUMED, false);

}

public void dispatchPause() {

moveToState(Fragment.STARTED, false);

}

public void dispatchStop() {

moveToState(Fragment.STOPPED, false);

}

public void dispatchDestroyView() {

moveToState(Fragment.CREATED, false);

}

public void dispatchDestroy() {

mDestroyed = true;

execPendingActions();

moveToState(Fragment.INITIALIZING, false);

mHost = null;

mContainer = null;

mParent = null;

}

而这些正好是和act的生命周期对应起来，也就是说这些方法是随着act进入到不同的生命周期而被调用的，即mCurState的值是被这些方法触发设置的。比如act进入到了Resume状态，那么FragmentManagerImpl.mCurState也就等于Fragment.RESUMED。

最后，我们分析下fragment状态转移最重要的一个方法，如下：

void moveToState(Fragment f, int newState, int transit, int transitionStyle,

boolean keepActive) {

*// Fragments that are not currently added will sit in the onCreate() state.*

if ((!f.mAdded || f.mDetached) && newState > Fragment.CREATED) {

newState = Fragment.CREATED;

}

if (f.mRemoving && newState > f.mState) {

*// While removing a fragment, we can't change it to a higher state.*

newState = f.mState;

}

*// Defer start if requested; don't allow it to move to STARTED or higher*

*// if it's not already started.*

if (f.mDeferStart && f.mState < Fragment.STARTED && newState > Fragment.STOPPED) {

newState = Fragment.STOPPED;

}

if (f.mState < newState) {

*// For fragments that are created from a layout, when restoring from*

*// state we don't want to allow them to be created until they are*

*// being reloaded from the layout.*

if (f.mFromLayout && !f.mInLayout) {

return;

}

if (f.mAnimatingAway != null) {

*// The fragment is currently being animated... but! Now we*

*// want to move our state back up. Give up on waiting for the*

*// animation, move to whatever the final state should be once*

*// the animation is done, and then we can proceed from there.*

f.mAnimatingAway = null;

moveToState(f, f.mStateAfterAnimating, 0, 0, true);

}

switch (f.mState) {

case Fragment.INITIALIZING:

if (f.mSavedFragmentState != null) {

f.mSavedViewState = f.mSavedFragmentState.getSparseParcelableArray(

FragmentManagerImpl.VIEW\_STATE\_TAG);

f.mTarget = getFragment(f.mSavedFragmentState,

FragmentManagerImpl.TARGET\_STATE\_TAG);

if (f.mTarget != null) {

f.mTargetRequestCode = f.mSavedFragmentState.getInt(

FragmentManagerImpl.TARGET\_REQUEST\_CODE\_STATE\_TAG, 0);

}

f.mUserVisibleHint = f.mSavedFragmentState.getBoolean(

FragmentManagerImpl.USER\_VISIBLE\_HINT\_TAG, true);

if (!f.mUserVisibleHint) {

f.mDeferStart = true;

if (newState > Fragment.STOPPED) {

newState = Fragment.STOPPED;

}

}

}

f.mHost = mHost;

f.mParentFragment = mParent;

f.mFragmentManager = mParent != null

? mParent.mChildFragmentManager : mHost.getFragmentManagerImpl();

f.mCalled = false;

f.onAttach(mHost.getContext());

if (!f.mCalled) {

throw new SuperNotCalledException("Fragment " + f

+ " did not call through to super.onAttach()");

}

if (f.mParentFragment == null) {

mHost.onAttachFragment(f);

}

if (!f.mRetaining) {

f.performCreate(f.mSavedFragmentState);

}

f.mRetaining = false;

if (f.mFromLayout) {

*// For fragments that are part of the content view*

*// layout, we need to instantiate the view immediately*

*// and the inflater will take care of adding it.*

f.mView = f.performCreateView(f.getLayoutInflater(

f.mSavedFragmentState), null, f.mSavedFragmentState);

if (f.mView != null) {

f.mView.setSaveFromParentEnabled(false);

if (f.mHidden) f.mView.setVisibility(View.GONE);

f.onViewCreated(f.mView, f.mSavedFragmentState);

}

}

case Fragment.CREATED:

if (newState > Fragment.CREATED) {

if (!f.mFromLayout) {

ViewGroup container = null;

if (f.mContainerId != 0) {

container = (ViewGroup)mContainer.onFindViewById(f.mContainerId);

if (container == null && !f.mRestored) {

throwException(new IllegalArgumentException(

"No view found for id 0x"

+ Integer.toHexString(f.mContainerId) + " ("

+ f.getResources().getResourceName(f.mContainerId)

+ ") for fragment " + f));

}

}

f.mContainer = container;

f.mView = f.performCreateView(f.getLayoutInflater(

f.mSavedFragmentState), container, f.mSavedFragmentState);

if (f.mView != null) {

f.mView.setSaveFromParentEnabled(false);

if (container != null) {

Animator anim = loadAnimator(f, transit, true,

transitionStyle);

if (anim != null) {

anim.setTarget(f.mView);

setHWLayerAnimListenerIfAlpha(f.mView, anim);

anim.start();

}

container.addView(f.mView);

}

if (f.mHidden) f.mView.setVisibility(View.GONE);

f.onViewCreated(f.mView, f.mSavedFragmentState);

}

}

f.performActivityCreated(f.mSavedFragmentState);

if (f.mView != null) {

f.restoreViewState(f.mSavedFragmentState);

}

f.mSavedFragmentState = null;

}

case Fragment.ACTIVITY\_CREATED:

case Fragment.STOPPED:

if (newState > Fragment.STOPPED) {

f.performStart();

}

case Fragment.STARTED:

if (newState > Fragment.STARTED) {

f.mResumed = true;

f.performResume();

*// Get rid of this in case we saved it and never needed it.*

f.mSavedFragmentState = null;

f.mSavedViewState = null;

}

}

} else if (f.mState > newState) {

switch (f.mState) {

case Fragment.RESUMED:

if (newState < Fragment.RESUMED) {

f.performPause();

f.mResumed = false;

}

case Fragment.STARTED:

if (newState < Fragment.STARTED) {

if (DEBUG) Log.v(TAG, "movefrom STARTED: " + f);

f.performStop();

}

case Fragment.STOPPED:

case Fragment.ACTIVITY\_CREATED:

if (newState < Fragment.ACTIVITY\_CREATED) {

if (DEBUG) Log.v(TAG, "movefrom ACTIVITY\_CREATED: " + f);

if (f.mView != null) {

*// Need to save the current view state if not*

*// done already.*

if (mHost.onShouldSaveFragmentState(f) && f.mSavedViewState == null) {

saveFragmentViewState(f);

}

}

f.performDestroyView();

if (f.mView != null && f.mContainer != null) {

Animator anim = null;

if (mCurState > Fragment.INITIALIZING && !mDestroyed) {

anim = loadAnimator(f, transit, false,

transitionStyle);

}

if (anim != null) {

final ViewGroup container = f.mContainer;

final View view = f.mView;

final Fragment fragment = f;

container.startViewTransition(view);

f.mAnimatingAway = anim;

f.mStateAfterAnimating = newState;

anim.addListener(new AnimatorListenerAdapter() {

@Override

public void onAnimationEnd(Animator anim) {

container.endViewTransition(view);

if (fragment.mAnimatingAway != null) {

fragment.mAnimatingAway = null;

moveToState(fragment, fragment.mStateAfterAnimating,

0, 0, false);

}

}

});

anim.setTarget(f.mView);

setHWLayerAnimListenerIfAlpha(f.mView, anim);

anim.start();

}

f.mContainer.removeView(f.mView);

}

f.mContainer = null;

f.mView = null;

}

case Fragment.CREATED:

if (newState < Fragment.CREATED) {

if (mDestroyed) {

if (f.mAnimatingAway != null) {

*// The fragment's containing activity is*

*// being destroyed, but this fragment is*

*// currently animating away. Stop the*

*// animation right now -- it is not needed,*

*// and we can't wait any more on destroying*

*// the fragment.*

Animator anim = f.mAnimatingAway;

f.mAnimatingAway = null;

anim.cancel();

}

}

if (f.mAnimatingAway != null) {

*// We are waiting for the fragment's view to finish*

*// animating away. Just make a note of the state*

*// the fragment now should move to once the animation*

*// is done.*

f.mStateAfterAnimating = newState;

newState = Fragment.CREATED;

} else {

if (DEBUG) Log.v(TAG, "movefrom CREATED: " + f);

if (!f.mRetaining) {

f.performDestroy();

}

f.mCalled = false;

f.onDetach();

if (!f.mCalled) {

throw new SuperNotCalledException("Fragment " + f

+ " did not call through to super.onDetach()");

}

if (!keepActive) {

if (!f.mRetaining) {

makeInactive(f);

} else {

f.mHost = null;

f.mParentFragment = null;

f.mFragmentManager = null;

f.mChildFragmentManager = null;

}

}

}

}

}

}

f.mState = newState;

}

这个方法最终会将FragmentManager的状态赋值给fragment，另外这个方法会根据不同的state调用各种onAttach, Fragment.performXXX，进而调到用户自己override的fragment的各种生命周期方法，比如onCreate、onCreateView等等。

### 总结

1. show/hideFragment只是改变fragment根View的visibility，最多带上个动画效果，另外只有本身是hidden的fragment，调用show才起作用，否则没用的，fragment.onHiddenChanged会被触发；其次不会有生命周期callback触发，当然了这些操作的前提是已经被add了的fragment；
2. addFragment的时候，不管加不加入回退栈都一样，经历的生命周期如下：onAttach、onCreate、onCreateView、onActivityCreate、onStart、onResume；
3. removeFragment的时候，经历的生命周期如下：onPause、onStop、onDestroyView，如果不加回退栈还会继续走  
   onDestroy、onDetach；remove的时候不仅从mAdded中移除fragment，也从mActive中移除了；



FragmentManagerImpl中关键字段申明

1. attach/detachFragment的前提都是已经add了的fragment，其生命周期回调不受回退栈影响。attach的时候onCreateView、onActivityCreate、onStart、onResume会被调用；detach的时候onPause、onStop、onDestroyView会被调用，onDestroy、onDetach不会被调用；对应的fragment只是从mAdded中移除了；
2. remove、detachFragment的时候，当FragmentManagerImpl.makeInactive()被调用的话，fragment就变成了一个空壳，里面绝大部分字段都会被置空，注意只是系统内部自己管理的字段，假如你在自己的fragment子类中引入了新的字段，当你重用这些类的对象时要自己处理这种情况（即系统不会reset你自己造的字段），代码如下：

void makeInactive(Fragment f) {

if (f.mIndex < 0) {

return;

}

mActive.set(f.mIndex, null);

if (mAvailIndices == null) {

mAvailIndices = new ArrayList<Integer>();

}

mAvailIndices.add(f.mIndex);

mHost.inactivateFragment(f.mWho);

f.initState();

}

*// Fragment.initState方法*

*/\*\**

*\* Called by the fragment manager once this fragment has been removed,*

*\* so that we don't have any left-over state if the application decides*

*\* to re-use the instance. This only clears state that the framework*

*\* internally manages, not things the application sets.*

*\*/*

void initState() {

mIndex = -1;

mWho = null;

mAdded = false;

mRemoving = false;

mResumed = false;

mFromLayout = false;

mInLayout = false;

mRestored = false;

mBackStackNesting = 0;

mFragmentManager = null;

mChildFragmentManager = null;

mHost = null;

mFragmentId = 0;

mContainerId = 0;

mTag = null;

mHidden = false;

mDetached = false;

mRetaining = false;

mLoaderManager = null;

mLoadersStarted = false;

mCheckedForLoaderManager = false;

}

## 回退事件

下面是Fragment的回退栈使用方法的代码：

Activity

1. **protected** **void** onCreate(Bundle savedInstanceState) {
2. **super**.onCreate(savedInstanceState);
3. setContentView(R.layout.main);
5. findViewById(R.id.addBackStack).setOnClickListener(**new** OnClickListener() {
6. @Override
7. **public** **void** onClick(View v) {
8. //添加Fragment到id为root的ViewGroup。并将其加入回退栈
9. getSupportFragmentManager().beginTransaction()
10. .addToBackStack(**null**)
11. .add(R.id.root, **new** TestFragment()).commit();
12. }
13. });
15. findViewById(R.id.popBackStack).setOnClickListener(**new** OnClickListener() {
16. @Override
17. **public** **void** onClick(View v) {
18. //从回退栈里弹出最新加入的Fragment
19. getSupportFragmentManager().**popBackStackImmediate**();
20. }
21. });
22. }

在界面加入两个按钮。一个点击后将TestFragment加入回退栈。一个点击后将新加入的TestFragment从Fragment的回退栈中弹出。

要让添加的Fragment加入回退栈。需要记住，调用addToBackStack(String)方法必须在commit()方法之前。让Fragment从回退栈中弹栈的方法。有两种。一种就是上面用的popBackStackImmediate()方法。另一种是popBackStack()。

首先说让Fragment的入栈操作。需要在commit()方法前调用addToBackStack()方法。下面我们来看此方法。此方法定义在BackStackRecord类中。它是FragmentTransaction的子类。

1. **public** FragmentTransaction addToBackStack(String name) {
2. **if** (!mAllowAddToBackStack) {
3. **throw** **new** IllegalStateException(
4. "This FragmentTransaction is not allowed to be added to the back stack.");
5. }
6. mAddToBackStack = **true**;
7. mName = name;
8. **return** **this**;
9. }

在这个方法中。我们可以看到。它只是简单的将mAddToBackStack 标志位设为true。再保存当前Fragment的tag名。现在我们注意到。在最外层有一个mAllowAddToBackStack标志位。代表是否允许加入回退栈。若你不想它加入回退栈。可以调用 disallowAddToBackStack()方法，接着在我们提交此次添加操作时。调用了commitInternal()方法。它的调用层次为commit()--->commitInternal()：

1. **int** commitInternal(**boolean** allowStateLoss) {
2. **if** (mCommitted) **throw** **new** IllegalStateException("commit already called");
3. **if** (FragmentManagerImpl.DEBUG) {
4. Log.v(TAG, "Commit: " + **this**);
5. LogWriter logw = **new** LogWriter(TAG);
6. PrintWriter pw = **new** PrintWriter(logw);
7. dump("  ", **null**, pw, **null**);
8. }
9. mCommitted = **true**;
10. **if** (mAddToBackStack) {
11. mIndex = mManager.allocBackStackIndex(**this**);
12. } **else** {
13. mIndex = -1;
14. }
15. mManager.enqueueAction(**this**, allowStateLoss);
16. **return** mIndex;
17. }

在第11行。当此Fragment要求入栈的时候。会由FargmentManager为它分配一个索引下标值。代表此次提交会将此组Fragment放于栈中的第几层。最下层的索引下标为0。接着往下。到BackStackRecord类中的run方法中：

1. **public** **void** run() {
2. ......
3. **if** (mAddToBackStack) {
4. mManager.addBackStackState(**this**);
5. }
6. }

FragmentManager

1. **void** addBackStackState(BackStackRecord state) {
2. **if** (mBackStack == **null**) {
3. mBackStack = **new** ArrayList<BackStackRecord>();
4. }
5. mBackStack.add(state);
6. reportBackStackChanged();
7. }

这里就是入栈的操作。入栈后调用reportBackStackChanged()方法。

**[java]** [view plain](https://blog.csdn.net/liu470368500/article/details/38145601) [copy](https://blog.csdn.net/liu470368500/article/details/38145601)

1. **void** reportBackStackChanged() {
2. **if** (mBackStackChangeListeners != **null**) {
3. **for** (**int** i=0; i<mBackStackChangeListeners.size(); i++) {
4. mBackStackChangeListeners.get(i).onBackStackChanged();
5. }
6. }
7. }

设置回退栈的监听器在addOnBackStackChangedListener()中。

**[java]** [view plain](https://blog.csdn.net/liu470368500/article/details/38145601) [copy](https://blog.csdn.net/liu470368500/article/details/38145601)

1. **public** **void** addOnBackStackChangedListener(OnBackStackChangedListener listener) {
2. **if** (mBackStackChangeListeners == **null**) {
3. mBackStackChangeListeners = **new** ArrayList<OnBackStackChangedListener>();
4. }
5. mBackStackChangeListeners.add(listener);
6. }

到此。Fragment的入栈操作完毕。现在开始弹栈分析：

如果你不想自己写弹栈的操作。可以直接按手机的返回按钮。也可以达到弹栈的效果。点击返回按键。会走Activity的onBackPress()方法：

1. **public** **void** onBackPressed() {
2. **if** (!mFragments.popBackStackImmediate()) {
3. finish();
4. }
5. }

可以发现。在你按返回键的时候。它其实也是调用的popBackStackImmediate()方法。上面我们说到弹栈有两种方法。下面分列出来：

1. **public** **void** popBackStack() {
2. enqueueAction(**new** Runnable() {
3. @Override **public** **void** run() {
4. popBackStackState(mActivity.mHandler, **null**, -1, 0);
5. }
6. }, **false**);
7. }
9. **public** **boolean** popBackStackImmediate() {
10. checkStateLoss();
11. executePendingTransactions();
12. **return** popBackStackState(mActivity.mHandler, **null**, -1, 0);
13. }

上面的popBackStack()方法。它调用enqueueAction方法。将弹栈操作放入一个Runnable对象中。最后将此Runnable对象放入主线程的消息队列尾部。使主线程在处理完其他操作后再处理弹栈操作。而第二种。就直接开始弹栈了。

FragmentManagerImpl类中：

1. **boolean** popBackStackState(Handler handler, String name, **int** id, **int** flags) {
2. **if** (mBackStack == **null**) {
3. **return** **false**;
4. }
5. **if** (name == **null** && id < 0 && (flags&POP\_BACK\_STACK\_INCLUSIVE) == 0) {
6. **int** last = mBackStack.size()-1;
7. **if** (last < 0) {
8. **return** **false**;
9. }
10. **final BackStackRecord bss = mBackStack.remove(last);**
11. **bss.popFromBackStack(true);**
12. reportBackStackChanged();
13. } **else** {
14. **int** index = -1;
15. **if** (name != **null** || id >= 0) {
16. // If a name or ID is specified, look for that place in  the stack.
17. index = mBackStack.size()-1;
18. **while** (index >= 0) {
19. **BackStackRecord bss = mBackStack.get(index);**
20. **if** (name != **null** && name.equals(bss.getName())) {
21. **break**;
22. }
23. **if** (id >= 0 && id == bss.mIndex) {
24. **break**;
25. }
26. index--;
27. }
28. **if** (index < 0) {
29. **return** **false**;
30. }
31. **if** ((flags&POP\_BACK\_STACK\_INCLUSIVE) != 0) {
32. index--;
33. // Consume all following entries that match.
34. **while** (index >= 0) {
35. **BackStackRecord bss = mBackStack.get(index);**
36. **if** ((name != **null** && name.equals(bss.getName()))
37. || (id >= 0 && id == bss.mIndex)) {
38. index--;
39. **continue**;
40. }
41. **break**;
42. }
43. }
44. }
45. **if** (index == mBackStack.size()-1) {
46. **return** **false**;
47. }
48. **final** ArrayList<BackStackRecord> states  = **new** ArrayList<BackStackRecord>();
49. **for** (**int** i=mBackStack.size()-1; i>index; i--) {
50. states.add(mBackStack.remove(i));
51. }
52. **final** **int** LAST = states.size()-1;
53. **for** (**int** i=0; i<=LAST; i++) {
54. states.get(i).popFromBackStack(i == LAST);
55. }
56. reportBackStackChanged();
57. }
58. **return** **true**;
59. }

这里有两个分支。if 里面的语句块。代表弹出当前栈中最上层的Fragment。else里的语句块。代表将栈中tag值为name的fragment**及以上**的全都从栈中弹出。由于不管是第一种还是第二种。实际上都是弹栈。在此处我们就只分析弹最上层的Fragment。为了方便阅读。将里面的代码块移出来：

1. **int** last = mBackStack.size()-1;
2. **if** (last < 0) {
3. **return** **false**;
4. }
5. **final** BackStackRecord bss = mBackStack.remove(last);
6. bss.popFromBackStack(**true**);
7. reportBackStackChanged();

第5行拿到栈中最上层的事务管理实例。提交时的Fragment都是放在此类中的Op类中的。第6行就是弹栈操作了：

1. **public** **void** popFromBackStack(**boolean** doStateMove) {
2. Op op = mTail;
3. **while** (op != **null**) {
4. **switch** (op.cmd) {
5. **case** OP\_ADD: {
6. Fragment f = op.fragment;
7. f.mNextAnim = op.popExitAnim;
8. mManager.removeFragment(f,
9. FragmentManagerImpl.reverseTransit(mTransition),
10. mTransitionStyle);
11. } **break**;
12. **case** OP\_REPLACE: {
13. Fragment f = op.fragment;
14. **if** (f != **null**) {
15. f.mNextAnim = op.popExitAnim;
16. mManager.removeFragment(f,
17. FragmentManagerImpl.reverseTransit(mTransition),
18. mTransitionStyle);
19. }
20. **if** (op.removed != **null**) {
21. **for** (**int** i=0; i<op.removed.size(); i++) {
22. Fragment old = op.removed.get(i);
23. old.mNextAnim = op.popEnterAnim;
24. mManager.addFragment(old, **false**);
25. }
26. }
27. } **break**;
28. **case** OP\_REMOVE: {
29. Fragment f = op.fragment;
30. f.mNextAnim = op.popEnterAnim;
31. mManager.addFragment(f, **false**);
32. } **break**;
33. **case** OP\_HIDE: {
34. Fragment f = op.fragment;
35. f.mNextAnim = op.popEnterAnim;
36. mManager.showFragment(f,
37. FragmentManagerImpl.reverseTransit(mTransition), mTransitionStyle);
38. } **break**;
39. **case** OP\_SHOW: {
40. Fragment f = op.fragment;
41. f.mNextAnim = op.popExitAnim;
42. mManager.hideFragment(f,
43. FragmentManagerImpl.reverseTransit(mTransition), mTransitionStyle);
44. } **break**;
45. **case** OP\_DETACH: {
46. Fragment f = op.fragment;
47. f.mNextAnim = op.popEnterAnim;
48. mManager.attachFragment(f,
49. FragmentManagerImpl.reverseTransit(mTransition), mTransitionStyle);
50. } **break**;
51. **case** OP\_ATTACH: {
52. Fragment f = op.fragment;
53. f.mNextAnim = op.popEnterAnim;
54. mManager.detachFragment(f,
55. FragmentManagerImpl.reverseTransit(mTransition), mTransitionStyle);
56. } **break**;
57. **default**: {
58. **throw** **new** IllegalArgumentException("Unknown cmd: " + op.cmd);
59. }
60. }
62. op = op.prev;
63. }
64. **if** (doStateMove) {
65. mManager.moveToState(mManager.mCurState,
66. FragmentManagerImpl.reverseTransit(mTransition), mTransitionStyle, **true**);
67. }
69. **if** (mIndex >= 0) {
70. mManager.freeBackStackIndex(mIndex);
71. mIndex = -1;
72. }
73. }

在里面的switch语句块中。不管你入栈的时候是以add还是以replace方式入的。都会调用FragmentManagerImpl中的removeFragment方法。

会将需要移除的fragment的状态值重新设为INITIALIZING，这时原inactive变量为true：直接跟进：

1. **void** moveToState(Fragment f, **int** newState, **int** transit, **int** transitionStyle,

这次由于是要做弹栈操作。所以我们直接看f.mState > newState的语句块。此时的f.mState状态值为RESUMED，要到状态INITIALIZING。所以会依次走RESUMED、STARTED、STOPPED、ACTIVITY\_CREATED、CREATED。在这里面。程序会走完剩下的生命周期：

在第9行f.performStop()中会执行Fragment的onStop()方法。第32行f.performDestroyView()执行onDestroyView()方法。第92行f.performDestroy()执行onDestroy()方法。95行f.onDetach()执行onDetach()方法与Activity解除绑定。至此。生命周期走完全程。最后在102行的 makeInactive(f)中对当前的fragment重新初始化各种状态值。

真正将此Fragment中的视图从界面移除的地方在第61行。f.mContainer.removeView(f.mView)，从上篇博客我们知道。f.mContainer就是入栈时add参数中第一个的布局ID对应的ViewGroup的实例。f.mView就是Fragment中在onCreateView()中返回的填充出来的View视图。

最后回到popFromBackStack中。在removeFragment后。FragmentManager会要求上一次加入栈中的Fragment将其状态移动到RESUMED状态。为了方便分析。我们将代码移下来。

1. **public** **void** popFromBackStack(**boolean** doStateMove) {
2. ......
3. **if** (doStateMove) {
4. mManager.moveToState(mManager.mCurState,
5. FragmentManagerImpl.reverseTransit(mTransition), mTransitionStyle, **true**);
6. }
8. **if** (mIndex >= 0) {
9. mManager.freeBackStackIndex(mIndex);
10. mIndex = -1;
11. }
12. }

这里又再次调用了FragmentManagerImpl的moveToState方法。在这里。如果之前移除的Fragment的入栈方式是以replace方式入栈的。则此时栈顶层的Fragment的状态值为INITIALIZING，也就是此时在moveToState中又会重新将此Fragment的创建一次。重新走一遍创建时的生命周期。如果是以add方式入栈的。则此Fragment的状态仍是RESUMED。则moveToState里面就将不进行操作。

当状态移动完毕。会接着调用freeBackStackIndex释放回退栈中顶层的索引下标。便于下次入栈时得到正确的下标值：

1. **public** **void** freeBackStackIndex(**int** index) {
2. **synchronized** (**this**) {
3. mBackStackIndices.set(index, **null**);
4. **if** (mAvailBackStackIndices == **null**) {
5. mAvailBackStackIndices = **new** ArrayList<Integer>();
6. }
7. **if** (DEBUG) Log.v(TAG, "Freeing back stack index " + index);
8. mAvailBackStackIndices.add(index);
9. }
10. }

# Picasso

Picasso可以根据imageview的大小调整bitmap大小

**一、Picasso 加载图片流程图**



**二、重要的类介绍**

（0）**Picasso**: 图片加载、转换、缓存的管理类。单列模式 ，通过with方法获取实例，也是加载图片的入口。  
（1）**RequestCreator**: Request构建类，Builder 模式，采用链式设置该Request的属性(如占位图、缓存策略、裁剪规则、显示大小、优先级等等)。最后调用build()方法生成一个请求（Request）。  
（2）**DeferredRequestCreator:**RequestCreator的包装类,当创建请求的时候还不能获取ImageView的宽和高的时候，则创建一个DeferredRequestCreator，DeferredRequestCreator里对 target 设置监听，直到可以获取到宽和高的时候重新执行请求创建。  
（3） **Action**: 请求包装类，存储了该请求和RequestCreator设置的这些属性，最终提交给线程执行下载。  
（4）**Dispatcher**:分发器，分发执行各种请求、分发结果等等。  
（5）**PicassoExecutorService**:Picasso使用的线程池，默认池大小为3。  
（6）**LruCache**:一个使用最近最少使用策略的内存缓存。  
（7）**BitmapHunter**:**这是Picasso的一个核心的类，开启线程执行下载，获取结果后解码成Bitmap,然后做一些转换操作如图片旋转、裁剪等，如果请求设置了转换器Transformation,也会在BitmapHunter里执行这些转换操作。**  
（8）**NetworkRequestHandler**:网络请求处理器，如果图片需要从网络下载，则用这个处理器处理。  
（9）**FileRequestHandler**:文件请求处理器，如果请求的是一张存在文件中的图片，则用这个处理器处理。  
（10）**AssetRequestHandler**: Asset 资源图片处理器，如果是加载asset目录下的图片，则用这个处理器处理。  
（11）**ResourceRequestHandler**:Resource资源图片处理器，如果是加载res下的图片，则用这个处理器处理。  
（12）**ContentStreamRequestHandler**: ContentProvider 处理器，如果是ContentProvider提供的图片，则用这个处理器处理  
（13）**MediaStoreRequestHandler**: MediaStore 请求处理器，如果图片是存在MediaStore上的则用这个处理器处理。  
（14）**ContactsPhotoRequestHandler**:ContactsPhoto 请求处理器，如果加载com.android.contacts/ 下的图片用这个处理器处理。如：

*// e.g. content://com.android.contacts/contacts/38)*

*//匹配的路径如下：*

static {

matcher = new UriMatcher(UriMatcher.NO\_MATCH);

matcher.addURI(ContactsContract.AUTHORITY, "contacts/lookup/\*/#", ID\_LOOKUP);

matcher.addURI(ContactsContract.AUTHORITY, "contacts/lookup/\*", ID\_LOOKUP);

matcher.addURI(ContactsContract.AUTHORITY, "contacts/#/photo", ID\_THUMBNAIL);

matcher.addURI(ContactsContract.AUTHORITY, "contacts/#", ID\_CONTACT);

matcher.addURI(ContactsContract.AUTHORITY, "display\_photo/#", ID\_DISPLAY\_PHOTO);

}

上面8-14 是默认的提供的几个处理器，分别处理不同来源的请求。

（15）**Response**: 返回的结果信息，Stream流或者Bitmap。  
（16）**Request**: 请求实体类，存储了应用在图片上的信息。  
（17）**Target**:图片加载的监听器接口，有3个回调方法，onPrepareLoad 在请求提交前回调，onBitmapLoaded 请求成功回调，并返回Bitmap,onBitmapFailed请求失败回调。  
（18）**PicassoDrawable**:继承BitmapDrawable，实现了过渡动画和图片来源的标识（就是图片来源的指示器，要调用 setIndicatorsEnabled(true)方法才生效），请求成功后都会包装成BitmapDrawable显示到ImageView 上。  
（19）**OkHttpDownloader**:用OkHttp实现的图片下载器，默认就是用的这个下载器。  
（20）**UrlConnectionDownloader**:使用HttpURLConnection 实现的下载器。  
（21）**MemoryPolicy**: 内存缓存策略，一个枚举类型。  
（22）**NetworkPolicy**: 磁盘缓存策略，一个枚举类型。  
(23) **Stats**: 这个类相当于日志记录，会记录如：内存缓存的命中次数，丢失次数，下载次数，转换次数等等，我们可以通过StatsSnapshot类将日志打印出来，看一下整个项目的图片加载情况。  
（24）**StatsSnapshot** :状态快照，和上面的Stats对应，打印Stats纪录的信息。

以上就是Picasso 的一些关键的类的介绍(还有一些简单的没有列举)。

## 三、流程分析

以加载网络图片为例，分析Picasso加载图片的整个流程

Picasso.with(this).load(URL)

.placeholder(R.drawable.default\_bg)

.error(R.drawable.error\_iamge)

.into(mBlurImage);

**1, 获取Picasso instance**

首先要获取一个Picasso对象，采用的单例模式

*//单例模式获取Picasso 对象*

public static Picasso with(Context context) {

if (singleton == null) {

synchronized (Picasso.class) {

if (singleton == null) {

singleton = new Builder(context).build();

}

}

}

return singleton;

}

*// 真正new 的地方在build()方法里*

public Picasso build() {

Context context = this.context;

if (downloader == null) {

*//配置默认的下载器，首先通过反射获取OkhttpClient,如果获取到了，就使用OkHttpDwownloader作为默认下载器*

*//如果获取不到就使用UrlConnectionDownloader作为默认下载器*

downloader = Utils.createDefaultDownloader(context);

}

if (cache == null) {

*// 配置内存缓存，****大小为手机内存的15%***

cache = new LruCache(context);

}

if (service == null) {

*// 配置Picaso 线程池，核心池大小为3*

service = new PicassoExecutorService();

}

if (transformer == null) {

*// 配置请求转换器，默认的请求转换器没有做任何事，直接返回原请求*

transformer = RequestTransformer.IDENTITY;

}

Stats stats = new Stats(cache);

*//分发器*

Dispatcher dispatcher = new Dispatcher(context, service, HANDLER, downloader, cache, stats);

return new Picasso(context, dispatcher, cache, listener, transformer, requestHandlers, stats,

defaultBitmapConfig, indicatorsEnabled, loggingEnabled);

}

**2, 通过load方法生成一个RequestCreator**

通过load方法生成一个RequestCreator，用链式api 来构建一个图片下载请求

*//load有几个重载方法，参数为string和File 的重载最重都会包装成一个Uri 调用这个方法*

public RequestCreator load(Uri uri) {

return new RequestCreator(this, uri, 0);

}

*// 如果是加载资源id 的图片会调用这个方法*

public RequestCreator load(int resourceId) {

if (resourceId == 0) {

throw new IllegalArgumentException("Resource ID must not be zero.");

}

return new RequestCreator(this, null, resourceId);

}

RequestCreator提供了很多的API 来构建请求，如展位图、大小、转换器、裁剪等等,这些API其实是为对应的属性赋值，最终会在into方法中构建请求。

*// 配置占位图，在加载图片的时候显示*

public RequestCreator placeholder(int placeholderResId) {

if (!setPlaceholder) {

throw new IllegalStateException("Already explicitly declared as no placeholder.");

}

if (placeholderResId == 0) {

throw new IllegalArgumentException("Placeholder image resource invalid.");

}

if (placeholderDrawable != null) {

throw new IllegalStateException("Placeholder image already set.");

}

this.placeholderResId = placeholderResId;

return this;

}

*// 配置真正显示的大小*

public RequestCreator resize(int targetWidth, int targetHeight) {

data.resize(targetWidth, targetHeight);

return this;

}

**3,into 添加显示的View，并且提交下载请求**

into方法里面干了3件事情：

1, 判断是否设置了fit 属性，如果设置了，再看是否能够获取ImageView 的宽高，如果获取不到，生成一个DeferredRequestCreator（延迟的请求管理器），然后直接return,在DeferredRequestCreator中当监听到可以获取ImageView 的宽高的时候，再执行into方法。

2, 判断是否从内存缓存获取图片，如果没有设置NO\_CACHE,则从内存获取，命中直接回调CallBack 并且显示图片。

3, 如果缓存未命中，则生成一个Action，并提交Action。

public void into(ImageView target, Callback callback) {

long started = System.nanoTime();

*// 检查是否在主线程*

checkMain();

if (target == null) {

throw new IllegalArgumentException("Target must not be null.");

}

*//如果没有url或者resourceId 则取消请求*

if (!data.hasImage()) {

picasso.cancelRequest(target);

if (setPlaceholder) {

setPlaceholder(target, getPlaceholderDrawable());

}

return;

}

*//判断是否设置了fit属性*

if (deferred) {

if (data.hasSize()) {

throw new IllegalStateException("Fit cannot be used with resize.");

}

int width = target.getWidth();

int height = target.getHeight();

if (width == 0 || height == 0) {

if (setPlaceholder) {

setPlaceholder(target, getPlaceholderDrawable());

}

*//如果获取不到宽高，生成一个DeferredRequestCreator（延迟的请求管理器），然后直接return,*

*//在DeferredRequestCreator中当监听到可以获取ImageView 的宽高的时候，再执行into方法。*

picasso.defer(target, new DeferredRequestCreator(this, target, callback));

return;

}

data.resize(width, height);

}

Request request = createRequest(started);

String requestKey = createKey(request);

*//是否从内存缓存中获取*

if (shouldReadFromMemoryCache(memoryPolicy)) {

Bitmap bitmap = picasso.quickMemoryCacheCheck(requestKey);

if (bitmap != null) {

*//缓存命中，取消请求，并显示图片*

picasso.cancelRequest(target);

setBitmap(target, picasso.context, bitmap, MEMORY, noFade, picasso.indicatorsEnabled);

if (callback != null) {

callback.onSuccess();

}

return;

}

}

if (setPlaceholder) {

setPlaceholder(target, getPlaceholderDrawable());

}

*//内存缓存未命中或者设置了不从内存缓存获取，则生成一个Action ，提交执行。*

Action action =

new ImageViewAction(picasso, target, request, memoryPolicy, networkPolicy, errorResId,

errorDrawable, requestKey, tag, callback, noFade);

picasso.enqueueAndSubmit(action);*// 提交请求*

}

**4, 提交、分发、执行请求。**

会经过下面这一系列的操作，最重将Action 交给BitmapHunter 执行。  
enqueueAndSubmit －> submit －> dispatchSubmit －> performSubmit：

*//将action 保存到了一个Map 中，目标View作为key*

void enqueueAndSubmit(Action action) {

Object target = action.getTarget();

if (target != null && targetToAction.get(target) != action) {

*// This will also check we are on the main thread.*

cancelExistingRequest(target);

targetToAction.put(target, action);

}

submit(action);

}

*// 交给分发器分发提交请求*

void submit(Action action) {

dispatcher.dispatchSubmit(action);

}

*//执行请求提交*

*//1, 先查看保存暂停tag表里面没有包含Action的tag,如果包含，则将Action 存到暂停Action表里*

*//2，从BitmapHunter表里查找有没有对应action的hunter,如果有直接attach*

*//3, 为这个请求生成一个BitmapHunter，提交给线程池执行*

void performSubmit(Action action, boolean dismissFailed) {

*// 先查看保存暂停tag表里面没有包含Action的tag,如果包含，则将Action 存到暂停Action表里*

if (pausedTags.contains(action.getTag())) {

pausedActions.put(action.getTarget(), action);

return;

}

BitmapHunter hunter = hunterMap.get(action.getKey());

if (hunter != null) {

hunter.attach(action);

return;

}

*// 如果线程池北shutDown,直接return*

if (service.isShutdown()) {

return;

}

*// 为请求生成一个BitmapHunter*

hunter = forRequest(action.getPicasso(), this, cache, stats, action);

hunter.future = service.submit(hunter);*//提交执行*

hunterMap.put(action.getKey(), hunter);

if (dismissFailed) {

failedActions.remove(action.getTarget());

}

}

**5,指定对应的处理器（RequestHandler）**

在上面执行的请求的performSubmit 方法里，**调用了forRequest 方法为对应的Action 生成一个BitmapHunter,里面有一个重要的步骤，指定请求处理器（在上面一节介绍Picasso有7种请求处理器**,看一下对应的代码：

static BitmapHunter forRequest(Picasso picasso, Dispatcher dispatcher, Cache cache, Stats stats,

Action action) {

Request request = action.getRequest();

List<RequestHandler> requestHandlers = picasso.getRequestHandlers();

*// Index-based loop to avoid allocating an iterator.*

*//noinspection ForLoopReplaceableByForEach*

for (int i = 0, count = requestHandlers.size(); i < count; i++) {

RequestHandler requestHandler = requestHandlers.get(i);

*// 循环请求处理器列表，如果找到有能处理这个请求的请求处理器*

*// 则生成BitmapHunter*

if (requestHandler.**canHandleRequest**(request)) {

return new BitmapHunter(picasso, dispatcher, cache, stats, action, requestHandler);

}

}

return new BitmapHunter(picasso, dispatcher, cache, stats, action, ERRORING\_HANDLER);

}

从Picasso里获取一个处理器列表，然后循环列表，看是否有能处理该请求的处理器，如果有，则生成BitmapHunter,那么这个请求处理器的列表在哪儿初始化的呢？请看源码：

*// 1,首先调用了getRequestHandlers*

List<RequestHandler> getRequestHandlers() {

return requestHandlers;

}

*// 2 requestHandlers 列表是在Picasso 构造函数里初始化的*

Picasso(Context context, Dispatcher dispatcher, Cache cache, Listener listener,

RequestTransformer requestTransformer, List<RequestHandler> extraRequestHandlers, Stats stats,

Bitmap.Config defaultBitmapConfig, boolean indicatorsEnabled, boolean loggingEnabled) {

....

*//前面代码省略*

*// 添加了7个内置的请求处理器*

*// 如果你自己通过Builder添了额外的处理器，也会添加在这个列表里面*

int builtInHandlers = 7; *// Adjust this as internal handlers are added or removed.*

int extraCount = (extraRequestHandlers != null ? extraRequestHandlers.size() : 0);

List<RequestHandler> allRequestHandlers =

new ArrayList<RequestHandler>(builtInHandlers + extraCount);

*// ResourceRequestHandler needs to be the first in the list to avoid*

*// forcing other RequestHandlers to perform null checks on request.uri*

*// to cover the (request.resourceId != 0) case.*

allRequestHandlers.add(new ResourceRequestHandler(context));

if (extraRequestHandlers != null) {

allRequestHandlers.addAll(extraRequestHandlers);

}

allRequestHandlers.add(new ContactsPhotoRequestHandler(context));

allRequestHandlers.add(new MediaStoreRequestHandler(context));

allRequestHandlers.add(new ContentStreamRequestHandler(context));

allRequestHandlers.add(new AssetRequestHandler(context));

allRequestHandlers.add(new **FileRequestHandler**(context));

allRequestHandlers.add(new NetworkRequestHandler(dispatcher.downloader, stats));

requestHandlers = Collections.unmodifiableList(allRequestHandlers);

*//后面代码省略*

...

}

小结： 在Picasso 的构造函数里 初始化了内置的7中请求处理器，然后在生成BitmapHunter的时候，循环列表，找到可以处理对应请求的处理器。

### 6, 重点：BitmapHunter (图片捕获器)

BitmapHunter继承Runnable，其实就是开启一个线程执行最终的下载。看一下源码：  
**1, run() 方法**

public void run() {

try {

updateThreadName(data);

*// 调用hunt() 方法获取最终结果*

result = hunt();

if (result == null) {

dispatcher.dispatchFailed(this);*//如果为null，分发失败的消息*

} else {

dispatcher.dispatchComplete(this);*//如果不为null，分发成功的消息*

}

} catch (Downloader.ResponseException e) {

if (!e.localCacheOnly || e.responseCode != 504) {

exception = e;

}

dispatcher.dispatchFailed(this);

} catch (NetworkRequestHandler.ContentLengthException e) {

exception = e;

dispatcher.dispatchRetry(this);

} catch (IOException e) {

exception = e;

dispatcher.dispatchRetry(this);

} catch (OutOfMemoryError e) {

StringWriter writer = new StringWriter();

stats.createSnapshot().dump(new PrintWriter(writer));

exception = new RuntimeException(writer.toString(), e);

dispatcher.dispatchFailed(this);

} catch (Exception e) {

exception = e;

dispatcher.dispatchFailed(this);

} finally {

Thread.currentThread().setName(Utils.THREAD\_IDLE\_NAME);

}

}

**当将一个bitmapHunter submit 给一个线程池执行的时候，就会执行run() 方法，run里面调用的是hunt方法来获取结果**，看一下hunt方法：

Bitmap hunt() throws IOException {

Bitmap bitmap = null;

*// 是否从内存缓存获取Bitmap*

if (shouldReadFromMemoryCache(memoryPolicy)) {

bitmap = cache.get(key);

if (bitmap != null) {

stats.dispatchCacheHit();

loadedFrom = MEMORY;

return bitmap;

}

}

data.networkPolicy = retryCount == 0 ? NetworkPolicy.OFFLINE.index : networkPolicy;

*// 请求处理器处理请求，获取结果，Result里可能是Bitmap，可能是Stream*

RequestHandler.Result result = requestHandler.load(data, networkPolicy);

if (result != null) {

loadedFrom = result.getLoadedFrom();

exifRotation = result.getExifOrientation();

bitmap = result.getBitmap();

*// If there was no Bitmap then we need to decode it from the stream.*

if (bitmap == null) {

InputStream is = result.getStream();

try {

bitmap = decodeStream(is, data);

} finally {

Utils.closeQuietly(is);

}

}

}

if (bitmap != null) {

stats.dispatchBitmapDecoded(bitmap);

if (data.needsTransformation() || exifRotation != 0) {

synchronized (DECODE\_LOCK) {

if (data.needsMatrixTransform() || exifRotation != 0) {

*//如果需要做转换，则在这里做转换处理，如角度旋转，裁剪等。*

bitmap = transformResult(data, bitmap, exifRotation);

}

if (data.hasCustomTransformations()) {

*// 如果配置了自定义转换器，则在这里做转换处理。*

bitmap = applyCustomTransformations(data.transformations, bitmap);

}

}

if (bitmap != null) {

stats.dispatchBitmapTransformed(bitmap);

}

}

}

return bitmap;

}

**7,Downloader 下载器下载图片**

上面的hunt方法获取结果的时候，最终调用的是配置的处理器的load方法，如下：

RequestHandler.Result result = requestHandler.load(data, networkPolicy);

加载网络图片用的是NetworkRequestHandler，匹配处理器，有个canHandleRequest 方法：

@Override public boolean canHandleRequest(Request data) {

String scheme = data.uri.getScheme();

return (SCHEME\_HTTP.equals(scheme) || SCHEME\_HTTPS.equals(scheme));

}

**判断的条件是，Uri带有"http://" 或者 https:// 前缀则可以处理**

我们接下来看一下NetworkRequestHandler的load方法：

public Result load(Request request, int networkPolicy) throws IOException {

*//最终调用downloader的load方法获取结果*

Response response = downloader.load(request.uri, request.networkPolicy);

if (response == null) {

return null;

}

Picasso.LoadedFrom loadedFrom = response.cached ? DISK : NETWORK;

Bitmap bitmap = response.getBitmap();

if (bitmap != null) {

return new Result(bitmap, loadedFrom);

}

InputStream is = response.getInputStream();

if (is == null) {

return null;

}

*// Sometimes response content length is zero when requests are being replayed. Haven't found*

*// root cause to this but retrying the request seems safe to do so.*

if (loadedFrom == DISK && response.getContentLength() == 0) {

Utils.closeQuietly(is);

throw new ContentLengthException("Received response with 0 content-length header.");

}

if (loadedFrom == NETWORK && response.getContentLength() > 0) {

stats.dispatchDownloadFinished(response.getContentLength());

}

return new Result(is, loadedFrom);

}

NetworkRequestHandler最终是调用的downloader 的load方法下载图片。内置了2个Downloader，OkhttpDownloader和UrlConnectionDownloader 。我们以UrlConnectionDownloader为例，来看一下load方法：

public Response load(Uri uri, int networkPolicy) throws IOException {

*//用的是HttpResponseCache（缓存http或者https的response到文件系统）*

installCacheIfNeeded(context);

HttpURLConnection connection = openConnection(uri);

*//设置使用缓存*

connection.setUseCaches(true);

if (networkPolicy != 0) {

String headerValue;

*// 下面一段代码是设置缓存策略*

if (NetworkPolicy.isOfflineOnly(networkPolicy)) {

headerValue = FORCE\_CACHE;

} else {

StringBuilder builder = CACHE\_HEADER\_BUILDER.get();

builder.setLength(0);

if (!NetworkPolicy.shouldReadFromDiskCache(networkPolicy)) {

builder.append("no-cache");

}

if (!NetworkPolicy.shouldWriteToDiskCache(networkPolicy)) {

if (builder.length() > 0) {

builder.append(',');

}

builder.append("no-store");

}

headerValue = builder.toString();

}

connection.setRequestProperty("Cache-Control", headerValue);

}

int responseCode = connection.getResponseCode();

if (responseCode >= 300) {

connection.disconnect();

throw new ResponseException(responseCode + " " + connection.getResponseMessage(),

networkPolicy, responseCode);

}

long contentLength = connection.getHeaderFieldInt("Content-Length", -1);

boolean fromCache = parseResponseSourceHeader(connection.getHeaderField(RESPONSE\_SOURCE));

*// 最后获取InputStream流包装成Response返回*

return new Response(connection.getInputStream(), fromCache, contentLength);

}

小结：梳理一下调用链, BitmapHunter －> NetworkRequestHandler －> UrlConnectionDownloader(也有可能是OkHttpDownloader),经过这一系列的调用，最后在BitmapHunter 的run 方法中就可以获取到我们最终要的Bitmap。

**8,返回结果并显示在Target上**

在BitmapHunter获取结果后，分发器分发结果,通过Hander处理后，执行performComplete方法：

*//1,*

void performComplete(BitmapHunter hunter) {

*// 这里将结果缓存到内存*

if (shouldWriteToMemoryCache(hunter.getMemoryPolicy())) {

cache.set(hunter.getKey(), hunter.getResult());

}

hunterMap.remove(hunter.getKey());*// 请求完毕，将hunter从表中移除*

batch(hunter);

}

*// 2,然后将BitmapHunter添加到一个批处理列表，通过Hander发送一个批处理消息*

private void batch(BitmapHunter hunter) {

if (hunter.isCancelled()) {

return;

}

batch.add(hunter);

if (!handler.hasMessages(HUNTER\_DELAY\_NEXT\_BATCH)) {

handler.sendEmptyMessageDelayed(HUNTER\_DELAY\_NEXT\_BATCH, BATCH\_DELAY);

}

}

*// 3，最后执行performBatchComplete 方法，通过主线程的Handler送处理完成的消息*

void performBatchComplete() {

List<BitmapHunter> copy = new ArrayList<BitmapHunter>(batch);

batch.clear();

mainThreadHandler.sendMessage(mainThreadHandler.obtainMessage(HUNTER\_BATCH\_COMPLETE, copy));

logBatch(copy);

}

*// 4,最后在Picasso 中handleMessage,显示图片*

static final Handler HANDLER = new Handler(Looper.getMainLooper()) {

@Override public void handleMessage(Message msg) {

switch (msg.what) {

case HUNTER\_BATCH\_COMPLETE: {

@SuppressWarnings("unchecked") List<BitmapHunter> batch = (List<BitmapHunter>) msg.obj;

*//noinspection ForLoopReplaceableByForEach*

for (int i = 0, n = batch.size(); i < n; i++) {

BitmapHunter hunter = batch.get(i);

hunter.picasso.complete(hunter);

}

break;

}

*//后面代码省略*

...

};

*// 5,最后回调到ImageViewAction 的complete方法显示图片*

@Override public void complete(Bitmap result, Picasso.LoadedFrom from) {

if (result == null) {

throw new AssertionError(

String.format("Attempted to complete action with no result!\n%s", this));

}

ImageView target = this.target.get();

if (target == null) {

return;

}

Context context = picasso.context;

boolean indicatorsEnabled = picasso.indicatorsEnabled;

*//将结果包装成一个PicassoDrawable 并显示*

PicassoDrawable.setBitmap(target, context, result, from, noFade, indicatorsEnabled);

if (callback != null) {

callback.onSuccess(); 回调callback

}

}

小结：通过上面一系列的方法调用， performComplete －> batch —> performBatchComplete －> handleMessage －> complete 把BitmapHunter中获取到的结果回调到主线程，并且显示在Target上。

**通过以上的8个步骤，就把图片从加载到显示的整个过程分析完了。**

**四，缓存特别说明**

内存缓存很简单，用的是LRUCache，大小为 手机内存的15% ，

这里重点说一下Disk Cahce。Picasso内存了2个默认的下载器，UrlConnectionDownloader和OkHttpDownloader,它们的磁盘缓存实现还是有一些差异的，看一下代码：

public OkHttpDownloader(final File cacheDir, final long maxSize) {

this(defaultOkHttpClient());

try {

client.setCache(new com.squareup.okhttp.Cache(cacheDir, maxSize));

} catch (IOException ignored) {

}

}

**在OkHttpDownloader 的构造方法里设置了磁盘缓存，使用的okHttp 的 DiskLruCache 实现的。**

然后看一下UrlConnectionDownloader的磁盘缓存实现，代码：

private static void installCacheIfNeeded(Context context) {

*// DCL + volatile.*

if (cache == null) {

try {

synchronized (lock) {

if (cache == null) {

cache = ResponseCacheIcs.install(context);

}

}

} catch (IOException ignored) {

}

}

}

private static class ResponseCacheIcs {

static Object install(Context context) throws IOException {

File cacheDir = Utils.createDefaultCacheDir(context);

HttpResponseCache cache = HttpResponseCache.getInstalled();

if (cache == null) {

long maxSize = Utils.calculateDiskCacheSize(cacheDir);

cache = HttpResponseCache.install(cacheDir, maxSize);

}

return cache;

}

static void close(Object cache) {

try {

((HttpResponseCache) cache).close();

} catch (IOException ignored) {

}

}

}

\*\* UrlConnectionDownloader 的磁盘缓存是用HttpResponseCache实现的\*\*

尽管2种磁盘缓存实现的方式不一样，但是它们的最后结果都是一样的：

**1,磁盘缓存的地址：** 磁盘缓存的地址在：data/data/your package name/cache/picasso-cache /  
**2,磁盘缓存的大小：磁盘缓存的大小为 手机磁盘大小的2%** ，不超过50M不小于5M。  
**3, 缓存的控制方式一样：**都是在请求的header设置**Cache-Control**的值来控制是否缓存。

**缓存清除：**  
**1, 清除内存缓存：调用invalidate方法,如：**

Picasso.with(this).invalidate("http://ww3.sinaimg.cn/large/610dc034jw1fasakfvqe1j20u00mhgn2.jpg");

清除指定url 的内存缓存。

但是Picasso没有提供清除全部内存缓存的方法，那就没有办法了吗？办法还是有的,LRUCahce 提供了clear方法的,只是Picasso没有向外部提供这个接口，因此可以通过反射获取到Picasso的cache字段，然后调用clear方法清除。

**2, 清除磁盘缓存**  
很遗憾Picasso没有提供清除磁盘缓存的方法。它没有提供方法我们就自己想办法呗。

思路：很简单，既然我们知道磁盘缓存是存在：data/data/your package name/cache/picasso-cache 这个路径下的，那我们把这个文件夹下面的所有文件清除不就行了。

实现：

private void clearDiskCache(){

File cache = new File(this.getApplicationContext().getCacheDir(), "picasso-cache");

deleteFileOrDirectory(cache.getPath());

}

# OkHttp

**以 Okhttp3源码 为例 ------ 图解 缓存机制 的原理和实现（下）**

2017年07月04日 09:17:29

阅读数：1206

之前写的一篇是介绍缓存机制的流程和原理，并讲解了HTTP协议中缓存相关的字段，彻底了解了缓存机制原理后实践了Okhttp3框架的缓存实现，即第一篇的内容已经打下了基础，此篇就从源码的角度来解析Okhttp3框架的缓存机制的实现。

（未看过第一篇，建议先了解缓存机制原理及HTTP协议中的重要字段，链接如下：）   
<http://blog.csdn.net/itermeng/article/details/52297077>

**一. OKhttp3 缓存机制整体流程**

在上篇博客中已列出一个例子来实践Okhttp3框架的缓存机制，这是一个举出的例子，使用Okhttp3进行网络请求，并设置缓存属性，代码如下：

public class CacheHttp {

public static void main(String args[]) throws IOException {

int maxCacheSize = 10 \* 1024 \* 1024;

Cache cache = new Cache(new File("/Users/nate/source"), maxCacheSize);

OkHttpClient client = new OkHttpClient.Builder().cache(cache).build();

Request request = new Request.Builder().url("http://www.qq.com/").

cacheControl(new CacheControl.Builder().maxStale(365, TimeUnit.DAYS).build()). build();

Response response = client.newCall(request).execute();

String body1 = response.body().string();

System.out.println("network response " + response.networkResponse());

System.out.println("cache response " + response.cacheResponse());

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

Response response1 = client.newCall(request).execute();

String body2 = response1.body().string();

System.out.println("network response " + response1.networkResponse());

System.out.println("cache response " + response1.cacheResponse());

}

}

以上代码已经使用了OKhttp3框架中的缓存机制，接下来从源码角度了解其实现。首先查看请求最终调用方法 —— client.newCall(request).execute() ，其实execute() 是**Call**接口中的方法，查看其实现类：**RealCall**

**1. 入口 —— RealCall 类**

**（1）execute() 方法**

【RealCall 类】

@Override public Response execute() throws IOException {

synchronized (this) {

if (executed) throw new IllegalStateException("Already Executed");

executed = true;

}

try {

client.dispatcher().executed(this);

Response result = getResponseWithInterceptorChain();

if (result == null) throw new IOException("Canceled");

return result;

} finally {

client.dispatcher().finished(this);

}

}

首先看try方法块，通过getResponseWithInterceptorChain() 获取一个Response响应对象，此方法应该是处理一些请求相关的信息，来具体查看实现：

**（2）getResponseWithInterceptorChain()方法**

private Response getResponseWithInterceptorChain() throws IOException {

// Build a full stack of interceptors.

List<Interceptor> interceptors = new ArrayList<>();

interceptors.addAll(client.interceptors());

interceptors.add(retryAndFollowUpInterceptor);

interceptors.add(new BridgeInterceptor(client.cookieJar()));

//☆☆☆☆☆缓存拦截器

**interceptors.add(new CacheInterceptor(client.internalCache()));**

interceptors.add(new ConnectInterceptor(client));

if (!retryAndFollowUpInterceptor.isForWebSocket()) {

interceptors.addAll(client.networkInterceptors());

}

interceptors.add(new CallServerInterceptor(

retryAndFollowUpInterceptor.isForWebSocket()));

Interceptor.Chain chain = new RealInterceptorChain(

interceptors, null, null, null, 0, originalRequest);

return chain.proceed(originalRequest);

}

}

从此方法可以看出Okhttp3 采用了一种拦截机制，在添加拦截器时发现一个类为 **CacheInterceptor** ，猜测与缓存相关的处理在此拦截器中进行。注意在创建此类时需要往构造函数中传参 —— client.interceptors() ，实际上它是Okhttp3 内部的一个缓存Cache，参看缓存拦截器**CacheInterceptor**其具体实现：

**2. 核心 —— 缓存拦截器CacheInterceptor类的 intercept方法**

此类主要缓存逻辑处理是在intercept 方法中进行，以下将方法分成两部分来进行讲解。

**（1）intercept 方法上半部分**

【CacheInterceptor 类】

public Response intercept(Chain chain) throws IOException {

**Response cacheCandidate = cache != null? cache.get(chain.request()) : null;**

long now = System.currentTimeMillis();

CacheStrategy strategy = new CacheStrategy.Factory(now, chain.request(), cacheCandidate).get();

Request networkRequest = strategy.networkRequest;

**Response cacheResponse = strategy.cacheResponse;**

if (cache != null) {

cache.trackResponse(strategy);

}

if (cacheCandidate != null && cacheResponse == null) {

closeQuietly(cacheCandidate.body()); // The cache candidate wasn't applicable. Close it.

}

// If we're forbidden from using the network and the cache is insufficient, fail.

if (networkRequest == null && cacheResponse == null) {

return new Response.Builder()

.request(chain.request())

.protocol(Protocol.HTTP\_1\_1)

.code(504)

.message("Unsatisfiable Request (only-if-cached)")

.body(EMPTY\_BODY)

.sentRequestAtMillis(-1L)

.receivedResponseAtMillis(System.currentTimeMillis())

.build();

}

// If we don't need the network, we're done.

if (networkRequest == null) {

return cacheResponse.newBuilder()

.cacheResponse(stripBody(cacheResponse))

.build();

}

......

//下一节讲解

}

注意此方法最终返回结果是一个Response响应变量，首先来了解此方法中至关重要的局部变量：

* **Request networkRequest**   
  网络请求，若此请求不需要使用到网络（即直接从缓存获取），则为null。
* **Response cacheResponse**   
  缓存响应，若此次请求不使用缓存（即缓存失效的情况下需请求网络资源），则为null

了解以上两个变量后，得知它是在创建**CacheStrategy**类后获取的其成员变量。需注意此类是缓存机制的重点，代表为Okhttp3框架中的缓存拦截**策略**，直接决定此方法的返回值Response响应。它的主要逻辑就是判断当前本地缓存是否有效，若缓存依然有效，则填充**cacheResponse**缓存响应对象；若无效，则填充**networkRequest**网络请求对象，而intercept方法会根据这两个值做下一步的判断。（CacheStrategy类源码具体实现后续会介绍，这里先将流程走通。）

根据判断这两个对象是否为null值，来决定返回结果，流程图如下：



可以发现，**CacheInterceptor**类的 intercept方法主要判断流程是根据 网络请求对象**networkRequest** 和 缓存响应对象**cacheResponse**决定的，它们存在的可能情况有以下三种：

* 网络请求对象、缓存响应对象皆为null
* 网络请求对象为null，缓存响应对象不为null
* 网络请求对象不为null，（缓存对象为null或不为null）

intercept方法的上半部分已经判断了以上两种情况，接下来最后一种情况在下半部分逻辑。注意最后一种情况，当网络请求对象不为null时，表明**CacheStrategy**缓存策略类判断后需要进行网络请求，也意味着本地缓存对象已失效，所以此时的缓存对象为null与否并非决定性关键了。

**（2）intercept 方法下半部分**

【CacheInterceptor 类】

public Response intercept(Chain chain) throws IOException {

......

//接上一节

Response networkResponse = null;

try {

networkResponse = chain.proceed(networkRequest);

} finally {

// If we're crashing on I/O or otherwise, don't leak the cache body.

if (networkResponse == null && cacheCandidate != null) {

closeQuietly(cacheCandidate.body());

}

}

// If we have a cache response too, then we're doing a conditional get.

if (cacheResponse != null) {

if (validate(cacheResponse, networkResponse)) {

Response response = cacheResponse.newBuilder()

.headers(combine(cacheResponse.headers(), networkResponse.headers()))

.cacheResponse(stripBody(cacheResponse))

.networkResponse(stripBody(networkResponse))

.build();

networkResponse.body().close();

// Update the cache after combining headers but before stripping the

// Content-Encoding header (as performed by initContentStream()).

cache.trackConditionalCacheHit();

cache.update(cacheResponse, response);

return response;

} else {

closeQuietly(cacheResponse.body());

}

}

Response response = networkResponse.newBuilder()

.cacheResponse(stripBody(cacheResponse))

.networkResponse(stripBody(networkResponse))

.build();

if (HttpHeaders.hasBody(response)) {

CacheRequest cacheRequest = maybeCache(response, networkResponse.request(), cache);

response = cacheWritingResponse(cacheRequest, response);

}

return response;

}

这下半部分的前提判断条件是：网络请求对象不为null，缓存对象为null或不为null。首先了解此方法中至关重要的局部变量：

* **Response networkResponse**   
  请求网络后的响应结果对象
* **Response response**   
  最终返回对象

它的主要逻辑是执行新的网络请求，最终返回response响应对象，至于第一个networkResponse响应对象则是用于执行网络请求获取资源数据。查看其源码，以流程图来解释逻辑：



**3. CacheInterceptor类的 intercept方法的总流程图：**



**二. 细节解析 —— CacheStrategy 缓存策略类**

**1. 原理图解**



**2. CacheStrategy 缓存策略类的创建 —– Factory()**

此类的调用是在**CacheInterceptor**类的 intercept方法里：

**CacheStrategy** 策略缓存类的构造是在其静态方法Factory 中，传递了三个参数：

- **now ：**是当前请求资源时间，将用于校验缓存是否过期。   
- **chain.request() ：** 一个**Request**对象，当前请求头的参数。   
- **cacheCandidate ：** 一个**Response**对象，如果当前的缓存目录（这里的缓存目录是cache，类型为InternalCache）不等于null，则从缓存目录中取出对应请求的**Response**数据。

【CacheStrategy 类 】

public Factory(long nowMillis, Request request, Response cacheResponse) {

this.nowMillis = nowMillis;

this.request = request;

this.cacheResponse = cacheResponse;

if (cacheResponse != null) {

this.sentRequestMillis = cacheResponse.sentRequestAtMillis();

this.receivedResponseMillis = cacheResponse.receivedResponseAtMillis();

Headers headers = cacheResponse.headers();

for (int i = 0, size = headers.size(); i < size; i++) {

String fieldName = headers.name(i);

String value = headers.value(i);

if ("Date".equalsIgnoreCase(fieldName)) {

servedDate = HttpDate.parse(value);

servedDateString = value;

} else if ("Expires".equalsIgnoreCase(fieldName)) {

expires = HttpDate.parse(value);

} else if ("Last-Modified".equalsIgnoreCase(fieldName)) {

lastModified = HttpDate.parse(value);

lastModifiedString = value;

} else if ("ETag".equalsIgnoreCase(fieldName)) {

etag = value;

} else if ("Age".equalsIgnoreCase(fieldName)) {

ageSeconds = HttpHeaders.parseSeconds(value, -1);

}

}

}

}

这里首要的就是判断从缓存目录中读取的数据**cacheResponse** 是否为空，只有在不为空的情况下才可以取出响应头中的数据，在if语句块中采用循环查找对应的几个标识符进行赋值。这几个标识符分别为”Date”、”Expires”、Last-Modified”、”ETag”、”Age”，注意并非每个标识符都存在于缓存数据中，例如上篇博客中的测试缓存腾讯网，只包含”Date”、”Expires”标识符。

循环中的赋值主要是为了填充**CacheStrategy** 类的两个重要局部变量：**cacheResponse**缓存响应对象和**networkRequest**网络请求对象。它们决定着Okhttp3框架缓存机制的核心方法 —— **CacheInterceptor**类的 intercept方法的核心逻辑。

**3. CacheStrategy 类中局部变量的获取 —— get()**

【CacheInterceptor 类】

CacheStrategy strategy = new CacheStrategy.Factory(now, chain.request(), cacheCandidate).get();

**CacheStrategy** 类的创建方法分析完后还没有结束，以上只是填充了两个局部变量，但是别忘了在**CacheInterceptor**类的 **intercept**方法中获取**CacheStrategy** 类最终调用是 get 方法，来查看其具体实现：

【CacheStrategy 类】

public CacheStrategy get() {

//getCandidate()才是真正逻辑判断实现 ☆☆☆☆☆

CacheStrategy candidate = getCandidate();

if (candidate.networkRequest != null && request.cacheControl().onlyIfCached()) {

// We're forbidden from using the network and the cache is insufficient.

return new CacheStrategy(null, null);

}

return candidate;

}

主要的就是调用getCandidate() 方法获取缓存策略类并返回，所以此方法才是真正处理缓存是否失效的核心逻辑方法，查看其实现（也将该方法分成上下两部分来分别解析）：

**（1）getCandidate() 的上半部分代码解析**

【CacheStrategy 类】

private CacheStrategy getCandidate() {

// No cached response.

if (cacheResponse == null) {

return new CacheStrategy(request, null);

}

if (request.isHttps() && cacheResponse.handshake() == null) {

return new CacheStrategy(request, null);

}

if (!isCacheable(cacheResponse, request)) {

return new CacheStrategy(request, null);

}

CacheControl requestCaching = request.cacheControl();

if (requestCaching.noCache() || hasConditions(request)) {

return new CacheStrategy(request, null);

}

//----------上部分为返回缓存响应为null的情况-------------

//----------下部分为判断缓存时间是否失效逻辑-------------

......

}

getCandidate() 方法的上半部分主要是返回无缓存响应Response对象的缓存策略类**CacheStrategy** ，即必定请求网络资源的情况，这里通过四种不同情况去判断，以下流程图分析其过程：



**（2）getCandidate() 的下半部分代码解析 ☆☆☆☆☆**

private CacheStrategy getCandidate() {

......

//----------上部分为返回缓存响应为null的情况-------------

//----------下部分为判断缓存时间是否失效逻辑-------------

long ageMillis = cacheResponseAge();

long freshMillis = computeFreshnessLifetime();

if (requestCaching.maxAgeSeconds() != -1) {

freshMillis = Math.min(freshMillis, SECONDS.toMillis(requestCaching.maxAgeSeconds()));

}

long minFreshMillis = 0;

if (requestCaching.minFreshSeconds() != -1) {

minFreshMillis = SECONDS.toMillis(requestCaching.minFreshSeconds());

}

long maxStaleMillis = 0;

CacheControl responseCaching = cacheResponse.cacheControl();

if (!responseCaching.mustRevalidate() && requestCaching.maxStaleSeconds() != -1) {

maxStaleMillis = SECONDS.toMillis(requestCaching.maxStaleSeconds());

}

//缓存未过期的情况

if (!responseCaching.noCache() && ageMillis + minFreshMillis < freshMillis + maxStaleMillis) {

Response.Builder builder = cacheResponse.newBuilder();

if (ageMillis + minFreshMillis >= freshMillis) {

builder.addHeader("Warning", "110 HttpURLConnection \"Response is stale\"");

}

long oneDayMillis = 24 \* 60 \* 60 \* 1000L;

if (ageMillis > oneDayMillis && isFreshnessLifetimeHeuristic()) {

builder.addHeader("Warning", "113 HttpURLConnection \"Heuristic expiration\"");

}

return new CacheStrategy(null, builder.build());

}

//缓存过期的情况

String conditionName;

String conditionValue;

if (etag != null) {

conditionName = "If-None-Match";

conditionValue = etag;

} else if (lastModified != null) {

conditionName = "If-Modified-Since";

conditionValue = lastModifiedString;

} else if (servedDate != null) {

conditionName = "If-Modified-Since";

conditionValue = servedDateString;

} else {

return new CacheStrategy(request, null); // No condition! Make a regular request.

}

Headers.Builder conditionalRequestHeaders = request.headers().newBuilder();

Internal.instance.addLenient(conditionalRequestHeaders, conditionName, conditionValue);

Request conditionalRequest = request.newBuilder()

.headers(conditionalRequestHeaders.build())

.build();

return new CacheStrategy(conditionalRequest, cacheResponse);

}

此方法的下半部分主要在判断缓存响应**Response**，这部分的处理逻辑与第二大点开头讲解的图片缓存机制原理相对应，可谓是缓存机制的精髓！同样将其分成两种不同情况进行处理（我注释的两个if语句块）：

**缓存未过期的情况**

return new CacheStrategy(null, builder.build());

查看其返回结果可知，若缓存未过期，在创建**CacheStrategy**类时，网络请求对象为null，将其缓存数据传入，即代表此请求无需请求网络，重复利用缓存数据即可。

**缓存已过期的情况（逻辑步骤）**

* **（a）**首先判断**Etag**字段是否为null，若不为null，则携带参数**If-None-Match**，此参数需与服务器校验本地过期的缓存是否需要更新，而服务器通过比较客户端与自身的这些字段标识来判断是否需要通知客户端进行缓存更新。
* **（b）** 若**Etag**字段为null，则继而判断**lastModified**字段是否为null，若不为null，则携带参数**If-Modified-Since**，理由同上。
* **（c）** 若**lastModified**字段为null，则继而判断**servedDate**字段是否为null，若不为null，则携带参数**If-Modified-Since**，这里缓存机制提供了三种字段去进行判断，因为并不是每个响应资源的字段中包含这些数据。
* **（d）** 若缓存中以上字段皆为null，则意味着此缓存资源过期且无使用价值，还是需要请求网络资源获取最新资源，则创建**CacheStrategy**类时，缓存响应对象为null。

return new CacheStrategy(request, null);

* **（e）**若以上三种字段有其一不为null，则将网络请求头重新进行封装，即在现有的字段上增加了新的携带参数，最终创建**CacheStrategy**类，传入新封装好的网络请求参数request、现有的缓存响应response。

**4. 小结**

以上就是缓存策略类的详细讲解，其中缓存机制的精髓就在其类的逻辑判断方法中 —— **getCandidate()**，重点放在缓存机制的原理图解和此方法的解析！

**三. 细节补充 —— Cache 和 DiskLruCache**

**1. Cache类初识**

首先思绪还是回到最初拦截器的创建代码，如下：

【RealCall 类】

interceptors.add(new CacheInterceptor(client.internalCache()));

这里传入**CacheInterceptor**类构造方法的参数是：

【OkHttpClient 类】

InternalCache internalCache() {

return cache != null ? cache.internalCache : internalCache;

}

主要逻辑是判断cache（**此变量代表当前程序指定的cache**）是否为空，不为空则返回cache的内部缓存，为空则返回**OkHttpClient** 的内部缓存。而这个变量cache是**Cache**类：

**2. Cache类介绍**

【Cache 类】

public final class Cache implements Closeable, Flushable {

private static final int VERSION = 201105;

private static final int ENTRY\_METADATA = 0;

private static final int ENTRY\_BODY = 1;

private static final int ENTRY\_COUNT = 2;

final InternalCache internalCache = new InternalCache() {

@Override public Response get(Request request) throws IOException {

return Cache.this.get(request);

}

@Override public CacheRequest put(Response response) throws IOException {

return Cache.this.put(response);

}

@Override public void remove(Request request) throws IOException {

Cache.this.remove(request);

}

@Override public void update(Response cached, Response network) {

Cache.this.update(cached, network);

}

......

private final DiskLruCache cache;

}

查看其源码可见cache.internalCache 是**Cache** 类的内部类，但是需要注意的是Cache最终调用的是**DiskLruCache** 类，**Cache** 类只不过是它的封装，调用**Cache** 类来对本地缓存数据进行修改、删除等操作。

**3. 操作本地缓存**

继续来探究操作本地缓存的问题，当我们调用Okhttp3框架api往本地缓存中添加数据时，**CacheInterceptor**类的maybeCache 方法，源码如下：

private CacheRequest maybeCache(Response userResponse, Request networkRequest,

InternalCache responseCache) throws IOException {

if (responseCache == null) return null;

// Should we cache this response for this request?

if (!CacheStrategy.isCacheable(userResponse, networkRequest)) {

if (HttpMethod.invalidatesCache(networkRequest.method())) {

try {

responseCache.remove(networkRequest);

} catch (IOException ignored) {

// The cache cannot be written.

}

}

return null;

}

// Offer this request to the cache.

return responseCache.put(userResponse);

}

【InternalCache 接口】

public interface InternalCache {

Response get(Request request) throws IOException;

CacheRequest put(Response response) throws IOException;

......

}

这里操作本地缓存数据最终调用的是**responseCache**对象的put 方法，查看其方法可知是**InternalCache**接口中的一个方法，而它具体实现对象便是**Cache**类，查看**Cache**类的put 方法：

【Cache 类】

private CacheRequest put(Response response) {

String requestMethod = response.request().method();

if (HttpMethod.invalidatesCache(response.request().method())) {

try {

remove(response.request());

} catch (IOException ignored) {

}

return null;

}

**if (!requestMethod.equals("GET")) {**

**return null;**

**}**

if (HttpHeaders.hasVaryAll(response)) {

return null;

}

Entry entry = new Entry(response);

DiskLruCache.Editor editor = null;

try {

editor = cache.edit(urlToKey(response.request()));

if (editor == null) {

return null;

}

entry.writeTo(editor);

return new CacheRequestImpl(editor);

} catch (IOException e) {

abortQuietly(editor);

return null;

}

}

**Cache**类的put 方法最终调用的还是**DiskLruCache**类，即它才是操作本地缓存的底层对象，此方法就是往本地文件中写入缓存数据，首先if判断当前的请求方式是否支持，如果支持的话先移除对应请求的数据，避免重复；然后通过**DiskLruCache**类获取可书写**Editor** 对象，然后会根据当前请求的URL做一个MD5的加密，生成的本地缓存名字就是一串字符，如下图所示，获取此文件后，调用Entry的writeTo 方法写入数据即可结束。此方法源码如下：

【Entry 类】

public void writeTo(DiskLruCache.Editor editor) throws IOException {

BufferedSink sink = Okio.buffer(editor.newSink(ENTRY\_METADATA));

sink.writeUtf8(url)

.writeByte('\n');

sink.writeUtf8(requestMethod)

.writeByte('\n');

sink.writeDecimalLong(varyHeaders.size())

.writeByte('\n');

for (int i = 0, size = varyHeaders.size(); i < size; i++) {

sink.writeUtf8(varyHeaders.name(i))

.writeUtf8(": ")

.writeUtf8(varyHeaders.value(i))

.writeByte('\n');

}

sink.writeUtf8(new StatusLine(protocol, code, message).toString())

.writeByte('\n');

sink.writeDecimalLong(responseHeaders.size() + 2)

.writeByte('\n');

for (int i = 0, size = respon

......

}

**结果图：**   
**7f4c79817fabaeaa0e909754cfe655e7.0 文件**



以上本地缓存文件是测试访问腾讯网后生成的文件，很明显文件名经过了md5加密，证实了以上说法。根据对比以上源码及本地缓存内容可证实其操作本地缓存数据流程。

至于最后的**DiskLruCache**类，它才是真正管理本地缓存目录文件操作，相关方法如下，这里就不再坐一一介绍了，此篇博客的重点还是在第一、二大点。



**四. 总结**

总体而言，Okhttp3源码中有关缓存机制重点的类为以下四个，通过这四个类来总结缓存机制的核心流程：

* **CacheIntercepter**
* **CacheStrategy**
* **Cache**
* **DiskLruCache**

**核心流程**

通过**CacheIntercepter**类去拦截请求处理缓存相关逻辑，其中使用缓存策略器**CacheStrategy**类来取出与缓存相关的数据，若缓存中有相应数据则取出，若缓存中数据不存在或过期则重新向服务器发出请求。当前本地有指定缓存时，重新请求的资源数据会被同步到当前本地中，涉及到的同步类是**Cache**、**DiskLruCache**类，以上是OKhttp3缓存机制的主要流程概括。

# End

## 