

# Android进阶34讲

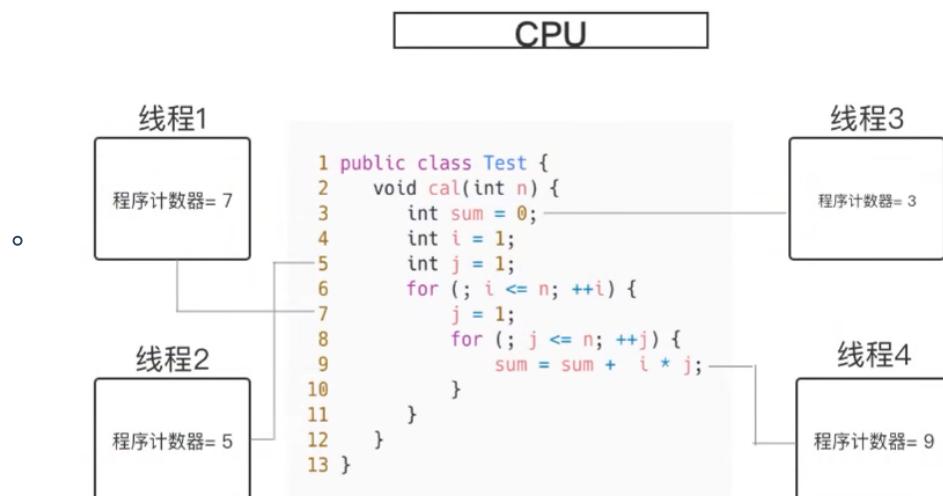
## 1、运行时内存分配

将Java的内存分为堆（heap）内存和栈内存（stack）实际上，Java虚拟机在执行java程序的过程钟，会把它所管理的内存划分为不同的数据区域



### JVM 运行时内存分布

- **程序计数器**: Java程序是多线程的，CPU可以在多个线程中分配执行的时间片段
  - 作用：当某一个线程被CPU挂起时，需要记录代码已经执行的位置，方便CPU重新执行此线程时，知道从哪行指令开始执行
  - 是虚拟机中一块较小的内存空间，主要用于记录当前线程执行的位置



### 关于程序计数器的几点注意：

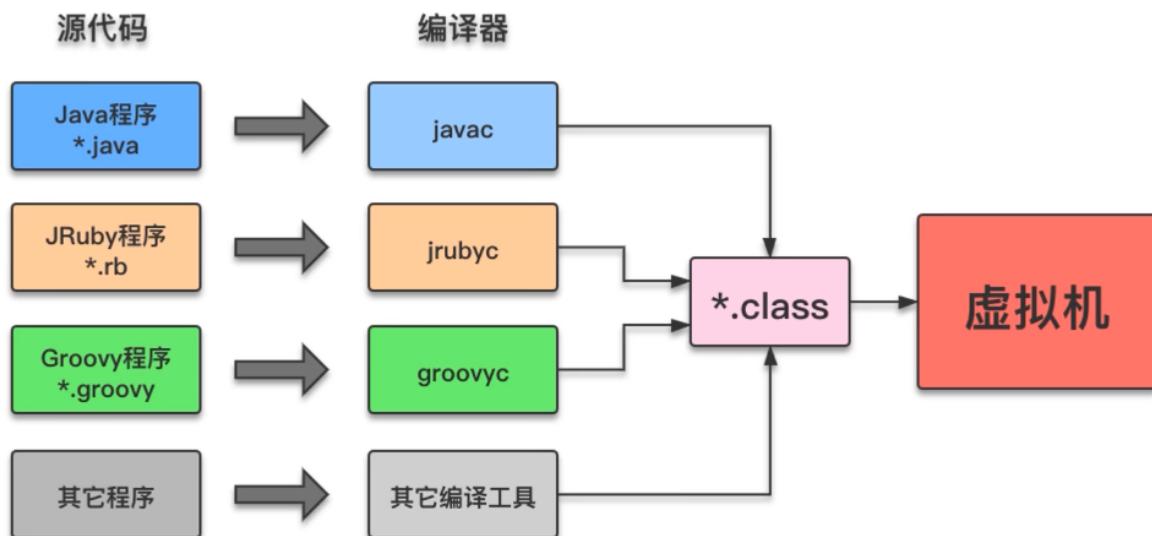
1. 在 Java 虚拟机规范中
    - 对程序计数器这一区域没有规定任何 OutOfMemoryError 情况（或许是感觉没有必要吧）
  2. 程序计数器是线程私有的，每条线程内部都有一个私有程序计数器
    - 它的生命周期随着线程的创建而创建，随着线程的结束而死亡
  3. 当一个线程正在执行一个 Java 方法的时候，这个计数器记录的是正在执行的虚拟机字节码指令的地址
    - 如果正在执行的是 Native 方法，这个计数器值则为空（Undefined）
- 虚拟机栈：是线程私有的，与线程的生命周期同步。在java虚拟机中规范中，对这个去域规定了两种异常情况：
    - StackOverflowError：当线程请求栈深度超过虚拟机栈所允许的深度时抛出
    - OutOfMemoryError：当Java虚拟机动态扩展到无法申请足够内容时抛出

## 2、GC回收机制与分代回收策略

## 3、字节码层面分析cla类文件结构

java提供了一种可以在所有平台上都能使用的一种中间代码-字节码类文件(.class文件)

- 有了字节码，无论是哪种平台只要安装了虚拟机都可以直接运行字节码
- 有了字节码，解除了Java虚拟机和Java语言之间的耦合



从纵观的角度看，class文件里只有两种数据结构：无符号数和表

无符号数：属于基本的数据类型

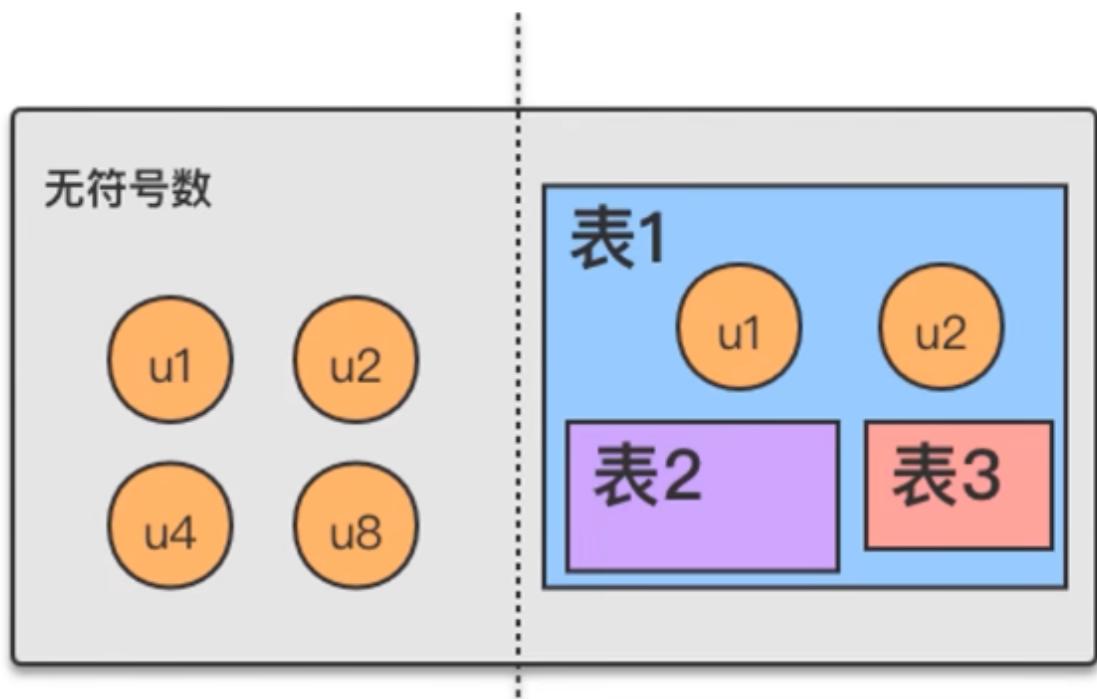
- 以u1、u2、u4、u8来分别代表1个字节、2个字节、4个字节和8个字节的无符号数，无符号数可以用来描述数字、索引引用、数量值或字符串(UTF-8编码)

表：表是多个无符号数或其他表作为数据项构成的复合数据类型

- class文件中所有的表都以“\_info”结尾

整个Class文件本质上就是一张表

# 表和无符号之间的关系



## Android是如何通过Activity进行交互的

### taskAffinity

通过设置不同的启动模式可以实现调度不同的Task，但是taskAffinity在一定程度上也会影响任务栈的调配流程

每一个Activity都有一个Affinity属性，如果不在清单文件中指定，默认为当前应用的包名

注意：

在一个 Android 项目 LagouTaskAffinity 中，创建两个 Activity

First 和 Second

应用名称	Activity 名称	taskAffinity	launchMode
LagouTaskAffinity	First	默认（应用包名）	默认（standard）
LagouTaskAffinity	Second	默认（应用包名）	默认（standard）

点击First中的Button，从First页面跳转到Second页面

adb shell dumpsys activity activities

TaskRecord代表一个任务栈

```

* TaskRecord{eb2cce #95 A=material.danny_jiang.com.lagoutaskaffinity U=0 sz=2}
    affinity=material.danny_jiang.com.lagoutaskaffinity
    intent=(act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10000000 cmp=material.danny_jiang.
realActivity=material.danny_jiang.com.lagoutaskaffinity/.First
Activities=[ActivityRecord{9e38370 u0 material.danny_jiang.com.lagoutaskaffinity/.First t95},
    ActivityRecord{6e29d91 u0 material.danny_jiang.com.lagoutaskaffinity/.Second t95}]
* Hist #1: ActivityRecord{6e29d91 u0 material.danny_jiang.com.lagoutaskaffinity processName=material.danny_jiang.com.lagoutaskaffinity
launchedFromUid=10064 launchedFromPackage=material.danny_jiang.com.lagoutaskaffinity userId=0
app=ProcessRecord{d807ef 25443:material.danny_jiang.com.lagoutaskaffinity/u0a64}
Intent { cmp=material.danny_jiang.com.lagoutaskaffinity/.Second }
frontOfTask=false task=TaskRecord{eb2cce #95 A=material.danny_jiang.com.lagoutaskaffinity U=0 sz=2}
taskAffinity=material.danny_jiang.com.lagoutaskaffinity
realActivity=material.danny_jiang.com.lagoutaskaffinity/.Second
baseDir=/data/app/material.danny_jiang.com.lagoutaskaffinity-1/base.apk
dataDir=/data/user/0/material.danny_jiang.com.lagoutaskaffinity
* Hist #0: ActivityRecord{9e38370 u0 material.danny_jiang.com.lagoutaskaffinity/.First t95}
    packageName=material.danny_jiang.com.lagoutaskaffinity processName=material.danny_jiang.com.lagoutaskaffinity
    app=ProcessRecord{d807ef 25443:material.danny_jiang.com.lagoutaskaffinity/u0a64}
Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10000000 cmp=material.danny_jiang.
frontOfTask=true task=TaskRecord{eb2cce #95 A=material.danny_jiang.com.lagoutaskaffinity U=0 sz=2}
taskAffinity=material.danny_jiang.com.lagoutaskaffinity
realActivity=material.danny_jiang.com.lagoutaskaffinity/.First
baseDir=/data/app/material.danny_jiang.com.lagoutaskaffinity-1/base.apk
dataDir=/data/user/0/material.danny_jiang.com.lagoutaskaffinity

```

### 修改 Second 的 taskAffinity

应用名称	Activity 名称	taskAffinity	launchMode
LagouTaskAffinity	First	默认 (应用包名)	默认 (standard)
LagouTaskAffinity	Second	lagou.affinity	默认 (standard)

### 重新查看任务栈情况

```

* TaskRecord{fabee9f #96 A=material.danny_jiang.com.lagoutaskaffinity U=0 sz=2}
    affinity=material.danny_jiang.com.lagoutaskaffinity
    intent=(act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10000000 cmp=material.danny_jiang.
realActivity=material.danny_jiang.com.lagoutaskaffinity/.First
Activities=[ActivityRecord{24bca66 u0 material.danny_jiang.com.lagoutaskaffinity/.First t96},
    ActivityRecord{24bca66 u0 material.danny_jiang.com.lagoutaskaffinity/.Second t96}]
* Hist #1: ActivityRecord{24bca66 u0 material.danny_jiang.com.lagoutaskaffinity processName=material.danny_jiang.com.lagoutaskaffinity
launchedFromUid=10064 launchedFromPackage=material.danny_jiang.com.lagoutaskaffinity userId=0
app=ProcessRecord{ce33bec 25559:material.danny_jiang.com.lagoutaskaffinity/u0a64}
Intent { cmp=material.danny_jiang.com.lagoutaskaffinity/.Second }
frontOfTask=false task=TaskRecord{fabee9f #96 A=material.danny_jiang.com.lagoutaskaffinity U=0 sz=2}
taskAffinity=lagou.affinity
realActivity=material.danny_jiang.com.lagoutaskaffinity/.Second
baseDir=/data/app/material.danny_jiang.com.lagoutaskaffinity-2/base.apk
dataDir=/data/user/0/material.danny_jiang.com.lagoutaskaffinity
* Hist #0: ActivityRecord{8b8c351 u0 material.danny_jiang.com.lagoutaskaffinity/.First t96}
    packageName=material.danny_jiang.com.lagoutaskaffinity processName=material.danny_jiang.com.lagoutaskaffinity
    app=ProcessRecord{ce33bec 25559:material.danny_jiang.com.lagoutaskaffinity/u0a64}
Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10000000 cmp=material.danny_jiang.
frontOfTask=true task=TaskRecord{fabee9f #96 A=material.danny_jiang.com.lagoutaskaffinity U=0 sz=2}
taskAffinity=material.danny_jiang.com.lagoutaskaffinity
realActivity=material.danny_jiang.com.lagoutaskaffinity/.First
baseDir=/data/app/material.danny_jiang.com.lagoutaskaffinity-2/base.apk
dataDir=/data/user/0/material.danny_jiang.com.lagoutaskaffinity

```

### 修改Second的启动模式，修改为SingleTask

```

* TaskRecord{8a4cee0 #98 A=lagou.affinity U=0 sz=1}
userId=0 effectiveUserId=u0a64 mCallingUid=u0a64 mCallingPackage=material.danny_jiang.com.lagoutaskaffinity
affinity=lagou.affinity
intent={flg=0x1000000 cmp=material.danny_jiang.com.lagoutaskaffinity/.Second}
realActivity=material.danny_jiang.com.lagoutaskaffinity/.Second
Activities=[ActivityRecord{23eed4 u0 material.danny_jiang.com.lagoutaskaffinity/.Second t98}]
* Hist #0: ActivityRecord{23eed4 u0 material.danny_jiang.com.lagoutaskaffinity/.Second t98}
    packageName=material.danny_jiang.com.lagoutaskaffinity processName=material.danny_jiang.com.lagoutaskaffinity
    launchedFromUid=10064 launchedFromPackage=material.danny_jiang.com.lagoutaskaffinity userId=0
    app=ProcessRecord{6748599 25654:material.danny_jiang.com.lagoutaskaffinity/u0a64}
    intent { flg=0x1000000 cmp=material.danny_jiang.com.lagoutaskaffinity/.Second }
    frontOfTask=true task=TaskRecord{8a4cee0 #98 A=lagou.affinity U=0 sz=1}
    taskAffinity=lagou.affinity
    realActivity=material.danny_jiang.com.lagoutaskaffinity/.Second
    baseDir=/data/app/material.danny_jiang.com.lagoutaskaffinity-1/base.apk
    dataDir=/data/user/0/material.danny_jiang.com.lagoutaskaffinity
* TaskRecord{da7b05e #97 A=material.danny_jiang.com.lagoutaskaffinity U=0 sz=1}
affinity=material.danny_jiang.com.lagoutaskaffinity
intent={act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10000000 cmp=material.danny_jiang.com.lagoutaskaffinity/.First}
realActivity=material.danny_jiang.com.lagoutaskaffinity/.First
Activities=[ActivityRecord{c56f881 u0 material.danny_jiang.com.lagoutaskaffinity/.First t97}]
* Hist #0: ActivityRecord{c56f881 u0 material.danny_jiang.com.lagoutaskaffinity/.First t97}
    packageName=material.danny_jiang.com.lagoutaskaffinity processName=material.danny_jiang.com.lagoutaskaffinity
    app=ProcessRecord{6748599 25654:material.danny_jiang.com.lagoutaskaffinity/u0a64}
    Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10000000 cmp=material.danny_jiang.com.lagoutaskaffinity }
    frontOfTask=true task=TaskRecord{da7b05e #97 A=material.danny_jiang.com.lagoutaskaffinity U=0 sz=1}
    taskAffinity=material.danny_jiang.com.lagoutaskaffinity
    realActivity=material.danny_jiang.com.lagoutaskaffinity/.First
    baseDir=/data/app/material.danny_jiang.com.lagoutaskaffinity-1/base.apk
    dataDir=/data/user/0/material.danny_jiang.com.lagoutaskaffinity

```

单纯使用taskAffinity不能导致Activity被创建在新的任务栈中，需要配合singleTask或singleInstance。

allowTaskReparenting赋予Activity在各个Task中间转移的特性

一个在后台任务中的Activity A当有其他任务前台，并且taskAffinity与A相同，则会自动将A添加到当前启动的任务栈中

## 通过Binder传递数据的限制

Activity 界面跳转时，使用Intent传递数据是最常用的操作，Intent传值也会导致数据崩溃。

方法一：减少通过Intent传递的数据，将非必须字段使用transient关键字修饰

方法二：将对象转化为JSON字符串，减少数据体积（Gson.toJson）

## 事件分发

分析角度：

- touch事件是如何从驱动层传递给Framework层的InputManagerService
- WMS是如何通过ViewRootImpl将事件传递给目标窗口
- touch事件到达DecorView后，是如何一步步传递到内部的子View中

**ViewGroup**：是一组view的组合，在其内部有可能包含多个子view

当手指触摸屏幕上时，手指所在区域既能在viewGroup显示范围内，也可能在其内部view控件上。内部的事件分发的重心是处理 **当前Group和子View之间的逻辑关系**：

- 当前group是否需要拦截touch事件
- 是否需要将touch事件继续分发给子view
- 如何将touch事件分发给子view

**View**：是一个单纯的控件，它的事件分发的重点在于当前的View如何去处理touch事件，并根据相应手势逻辑进行一些列的效果展示（比如滑动、放大G、点击、长按）

- 是否存在touchListener
- 是否自己接收处理touch事件（主要逻辑在onTouchEvent方法中）

## 事件分发核心： dispatchTouchEvent

整个view之间的事件分发，实质是一个大的递归函数，这个递归函数就是 `dispatchTouchEvent` 方法，在这个递归过程中会适时调用 `onInterceptTouchEvent` 来拦截事件或调用 `onTouchEvent` 方法来处理事件。

```
public boolean dispatchTouchEvent() {  
    /**/  
     * 步骤1：检查当前ViewGroup是否需要拦截事件  
     */  
    ...  
    /**/  
     * 步骤2：将事件分发给子View  
     */  
    ...  
    /**/  
     * 步骤3：根据mFirstTouchTarget，再次分发事件  
     */  
    ...  
}
```

```
/**  
 * 1 检查当前ViewGroup是否需要拦截事件  
 */  
final boolean intercepted;  
if (actionMasked == MotionEvent.ACTION_DOWN  
    || mFirstTouchTarget != null) {  
    final boolean disallowIntercept = (mGroupFlags & FLAG_DISALLOW_INTERCEPT) != 0;  
    if (!disallowIntercept) {  
        intercepted = onInterceptTouchEvent(ev);  
        ev.setAction(action); // restore action in case it was changed  
    } else {  
        intercepted = false;  
    }  
} else {  
    // There are no touch targets and this action is not an initial down  
    // so this view group continues to intercept touches.  
    intercepted = true;  
}
```

```

/**
 * 2 将事件分发给子View
 */
if (!canceled && !intercepted) {
    if (actionMasked == MotionEvent.ACTION_DOWN) { ①
        || (split && actionMasked == MotionEvent.ACTION_POINTER_DOWN)
        || actionMasked == MotionEvent.ACTION_HOVER_MOVE) {
            final int childrenCount = mChildrenCount;
            if (newTouchTarget == null && childrenCount != 0) {
                for (int i = childrenCount - 1; i >= 0; i--) { ②
                    final int childIndex = getAndVerifyPreorderedIndex(
                        childrenCount, i, customOrder);
                    final View child = getAndVerifyPreorderedView(
                        reorderedList, children, childIndex);

                    ③ if (!canViewReceivePointerEvents(child)
                        || !isTransformedTouchPointInView(x, y, child, null)) {
                        ev.setTargetAccessibilityFocus(false);
                        continue;
                    }

                    ④ if (dispatchTransformedTouchEvent(ev, false, child, idBitsToAssign)) {
                        // Child wants to receive touch within its bounds.
                        ...
                        newTouchTarget = addTouchTarget(child, idBitsToAssign);
                        alreadyDispatchedToNewTouchTarget = true;
                        break;
                    }
                    ...
                }
            }
        }
    }
}

```

1. 表明事件主动分发的前提是事件为DOWN事件
2. 遍历所有的子view
3. 判断事件坐标是否在子view坐标范围内，并且子view并没有处于动画状态。
4. dispatchTransformedTOuchEvent方法将事件分发给子view，如果子view捕获事件成功，则将mFirstTouchTarget赋值给子view

```

/**
 * 3 根据mFirstTouchTarget, 再次分发事件
 */
if (mFirstTouchTarget == null) { ①
    handled = dispatchTransformedTouchEvent(ev, canceled, null, 传入child为null
        TouchTarget.ALL_POINTER_IDS);
} else {
    ...
    TouchTarget predecessor = null;
    TouchTarget target = mFirstTouchTarget;
    while (target != null) {
        final TouchTarget next = target.next;
        if (alreadyDispatchedToNewTouchTarget && target == newTouchTarget) {
            handled = true;
        } else {
            final boolean cancelChild = resetCancelNextUpFlag(target.child)
                || intercepted;
            ② if (dispatchTransformedTouchEvent(ev, cancelChild,
                target.child, target.pointerIdBits)) {
                handled = true;
            }
            ...
        }
    }
}

```

如果没有子view捕获处理touch事件， ViewGroup会通过自身的onTouchEvent方法进行处理。

## 为什么Down事件特殊

- 所有touch事件都是从Down事件开始的
- Down事件的处理结果会直接影响后续的MOVE、UP事件的逻辑

只有Down事件会传递给子view进行捕获判断，一旦子View捕获成功，后续的MOVE和UP事件是通过遍历mFirstTouchTarget链表查找之前接收ACTION\_DOWN的子View，并将触摸事件分配给子View

后续的MOVE、UP等事件的分发交给谁，取决于他们的起始事件DOWN由谁捕获的

## dispatchTouchEvent事件处理流程机制

- 判断是否需要拦截->主要根据onInterceptTouchEvent方法的返回值来决定是否拦截
- 在Down事件中将Touch事件分发给子View->这一过程如果有子View捕获消费了touch事件，会对mFirstTouchEvent进行赋值
- DOWN、MOVE、UP事件都会根据mFirstTouchTarget是否为null，决定是自己处理touch事件还是再次分发给子view

## 事件分发特殊点

DOWN事件的特殊之处：事件的起点；决定后续事件由谁来消费处理

mFirstTouchEvent作用：记录捕获消费touch事件的view，是一个链表结构

CANCEL事件的触发场景：当父视图先不拦截，然后在MOVE事件中重新拦截，此时子View会接收一个CANCEL事件

# 自定义View

自定义控件有两种方式：

- 继承系统提供的成熟控件（LinearLayout、RelativeLayout、ImageView）

自定义属性：在res/values目录下attrs.xml文件，使用标签自定义属性

```
<declare-styleable name="CustomToolBar">
    <attr name="titleText" format="string|reference" />
    <attr name="myTitleTextColor" format="color|reference" />
    <attr name="titleTextSize" format="dimension|reference" />
    <attr name="leftImageSrc" format="reference" />
    <attr name="rightImageSrc" format="reference" />
</declare-styleable>
```

**<declare-styleable> 标签**代表定义一个自定义属性集合，一般会与自定义控件结合使用

**<attr> 标签**是某一条具体的属性，name是属性名称，format代表属性的格式

获取自定义属性值：

```
TypedArray ta = context.obtainStyledAttributes(attrs, R.styleable.CustomToolBar);
String titleText = ta.getString(R.styleable.CustomToolBar_titleText);
//第二个参数表示默认颜色
int titleTextColor = ta.getColor(R.styleable.CustomToolBar_myTitleTextColor, Color.BLACK);
//已经由sp转为px
float titleTextSize = ta.getDimension(R.styleable.CustomToolBar_titleTextSize, 12);

//读取图片
Drawable leftDrawable = ta.getDrawable(R.styleable.CustomToolBar_leftImageSrc);
Drawable rightDrawable = ta.getDrawable(R.styleable.CustomToolBar_rightImageSrc);
```

- 直接继承系统View或ViewGroup，并自绘制显示内容

注意问题：

- 如何根据相应的属性将UI元素绘制到界面 ---onDraw
- 自定义控件的大小，也就是宽和高分别设置多少 -- onMeasure
- 如果是ViewGroup，如果合理安排其内部子view的摆放位置---onLayout

onDraw:

### 系统提供了一系列 Canvas 操作方法

```
void drawRect(RectF rect, Paint paint) : 绘制矩形区域  
void drawOval(RectF oval, Paint paint) : 绘制椭圆  
void drawCircle(float cx, float cy, float radius, Paint paint) : 绘制圆形  
void drawArc(RectF oval, float startAngle, float sweepAngle, boolean useCenter, Paint paint) : 绘制弧形  
void drawPath(Path path, Paint paint) : 绘制Path路径  
void drawLine(float startX, float startY, float stopX, float stopY, Paint paint) : 绘制连线。  
void drawPoint(float x, float y, Paint paint) : 绘制点。
```

paint的属性，就是一个画笔

```
setStyle(Style style) 设置绘制模式  
setColor(int color) 设置颜色  
setAlpha(int a) : 设置透明度。  
setShader(Shader shader) : 设置Paint的填充效果。  
Paint.setStrokeWidth(float width) 设置线条宽度  
Paint.setTextSize(float textSize) 设置文字大小  
Paint.setAntiAlias(boolean aa) 设置抗锯齿开关  
Paint.setDither(boolean dither) 设置防抖动开关
```

onMeasure

直接在XML布局文件中定义好View的宽高，然后让自定义View在此宽高的区域内显示

Android系统提供了wrap\_content和match\_parent属性来规范空间的 显示规则

分别代表自适应大小和填充父视图的大小，这两个属性没有指定具体的大小，需要在onMeasure方法中过滤这两种情况。

三种测量模式

- EXACTLY:表示在XML布局文件中宽高使用match\_parent或者固定大小的宽高
- AT\_MOST：表示在XML布局文件中宽高使用wrap\_content
- UNSPECIFIED:父容器没有对当前View有任何限制，当前View可以取任意尺寸，比如ListView中item

```

// 宽度测量模式
int widthMode = MeasureSpec.getMode(widthMeasureSpec);
// 宽度测量大小
int widthSize = MeasureSpec.getSize(widthMeasureSpec);
// 高度测量模式
int heightMode = MeasureSpec.getMode(heightMeasureSpec);
// 高度测量大小
int heightSize = MeasureSpec.getSize(heightMeasureSpec);

```

## RecyclerView

目的：在有限的屏幕展示大量的内容，复用机制的实现是他的核心部分

使用方法：

`setLayoutManager` :必选项，设置RV的布局管理器，决定RV的显示风格

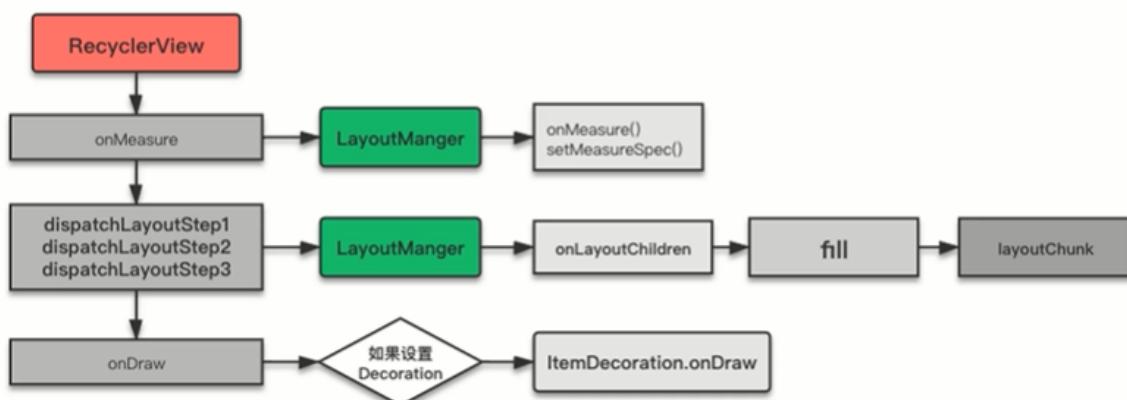
常用的线性布局管理器( `LinearLayoutManager` )、网格布局管理器( `GridLayoutManager` )、瀑布流布局管理器( `StaggeredGridLayoutManager` )

`setAdapter` : 必选项，设置RV的数据适配器

当数据发生改变时，以通知者的身份，通知RV数据改变进行列表刷新操作。

`setItemDecoration` :非必选项，设置RV中item的装饰器，经常用来设置item的分割线

`setItemAnimator` :非必选项，设置RV中Item的动画



缓存复用：

主要实现了ViewHolder的缓存以及复用

```

public final class Recycler {
    final ArrayList<ViewHolder> mAttachedScrap = new ArrayList<>();

    ArrayList<ViewHolder> mChangedScrap = null;

    final ArrayList<ViewHolder> mCachedViews = new ArrayList<ViewHolder>();

    RecycledViewPool mRecyclerPool;

    private ViewCacheExtension mViewCacheExtension;
}

```

缓存根据访问优先级从上到下分为4级

第一级缓存：mAttachedScrap、mChangedScrap

第二级缓存：mCachedViews

第三级缓存：ViewCacheExtension

第四级缓存：RecycledViewPool

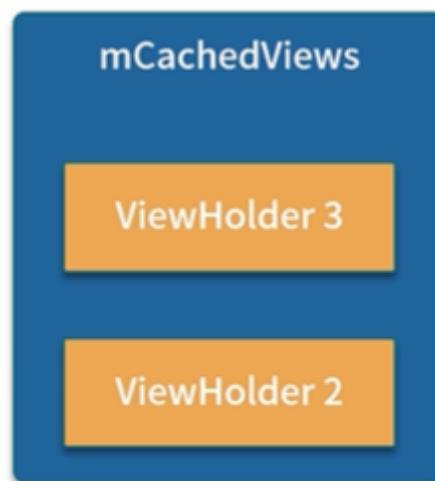
第一级缓存主要用来缓存屏幕内的viewHolder

第一级缓存 mAttachedScrap&mChangedScrap



第二级缓存：

主要保存移除屏幕的viewitem，容量只有2



第三级缓存:继承之后实现自定义缓存

RV 预留给开发人员的一个抽象类，在这个类中只有一个抽象方法

```
1 public abstract static class ViewCacheExtension {  
2     @Nullable  
3         public abstract View getViewForPositionAndType(@NonNull Recycler recycler,  
4                                         int position, int type);  
5 }
```

第四级缓存：

### 第四级缓存 RecycledViewPool

RecycledViewPool 是用来缓存屏幕外的 ViewHolder

当 mCachedViews 中的个数已满（默认为 2）

则从 mCachedViews 中淘汰出来的 ViewHolder 会先缓存到 RecycledViewPool 中

ViewHolder 在被缓存到 RecycledViewPool 时，会将内部的数据清理

**从 RecycledViewPool 中取出来的 ViewHolder 需要重新调用 onBindViewHolder 绑定数据**

可以多个RV共享recycledViewPool，多个RV共享时，确保使用的apater是同一个

核心实现：

- RV如何经过测量、布局、最终绘制到屏幕上，其中大部分工作是通过委托给LayoutManager来实现
- RV的缓存复用机制，主要通过内部类Recycler来实现

## 17、OKhttp

是一套处理Http网络请求的依赖库

Retrofit+OkHttp实现网络请求

流程分析：

```
OkHttpClient client = new OkHttpClient();  
  
Request request = new Request.Builder()  
    .url(url)  
    .build();  
  
client.newCall(request).enqueue(new Callback() {  
    @Override  
    public void onFailure(Call call, IOException e) {}  
    @Override  
    public void onResponse(Call call, Response response) throws IOException {}});
```

## 使用内部工厂类 Builder 来设置 OkHttpClient

```
OkHttpClient.Builder builder = new OkHttpClient.Builder();
builder.connectTimeout(60, TimeUnit.SECONDS) // 设置超时
    .addInterceptor(interceptor) // 添加拦截器
    .proxy(proxy) // 设置请求代理
    .cache(cache); // 设置缓存策略
OkHttpClient client = builder.build();
```

请求操作的起点从 OkHttpClient.newCall().enqueue() 方法开始

### newCall

```
1 @Override public Call newCall(Request request) {
2     return RealCall.newRealCall(this, request, false);
3 }
```

### RealCall.enqueue

```
1 @Override public void enqueue(Callback responseCallback) {
2     synchronized (this) {
3         if (executed) throw new IllegalStateException("Already Executed");
4         executed = true;
5     }
6     captureCallStackTrace();
7     eventListener.callStart(this);
8     client.dispatcher().enqueue(new AsyncCall(responseCallback));
9 }
```

Dispatcher是OkHttpClient的调度器，是一种门户模式，主要用来实现执行、取消异步请求操作。本质上是内部维护了一个线程池去执行异步操作，保证最大并发个数，同一host主机允许执行请求的线程个数。

### Dispatcher 的 enqueue 方法的具体实现

```
1 synchronized void enqueue(AsyncCall call) {
2     if (runningAsyncCalls.size() < maxRequests && runningCallsForHost(call) < maxRequestsPerHost) {
3         runningAsyncCalls.add(call);
4         executorService().execute(call);
5     } else {
6         readyAsyncCalls.add(call);
7     }
8 }
```

```

final class AsyncCall extends NamedRunnable {
    @Override public final void run() {
        .....
        execute(); ①
        .....
    }

    @Override protected void execute() {
        boolean signalledCallback = false;
        try {
            Response response = getResponseWithInterceptorChain(); ②
            if (retryAndFollowUpInterceptor.isCanceled()) {
                signalledCallback = true;
                responseCallback.onFailure(RealCall.this, new IOException("Canceled"));
            } else {
                signalledCallback = true;
                responseCallback.onResponse(RealCall.this, response);
            }
        } catch (IOException e) {
            if (signalledCallback) {
                .....
                responseCallback.onFailure(RealCall.this, e);
            }
        } finally {
            client.dispatcher().finished(this);
        }
    }
}

```

```

1     Response getResponseWithInterceptorChain() throws IOException {
2         // Build a full stack of interceptors.
3         List<Interceptor> interceptors = new ArrayList<>();
4         interceptors.addAll(client.interceptors());
5         interceptors.add(retryAndFollowUpInterceptor);
6         interceptors.add(new BridgeInterceptor(client.cookieJar()));
7         interceptors.add(new CacheInterceptor(client.internalCache()));
8         interceptors.add(new ConnectInterceptor(client));
9         if (!forWebSocket) {
10             interceptors.addAll(client.networkInterceptors());
11         }
12         interceptors.add(new CallServerInterceptor(forWebSocket));
13
14         Interceptor.Chain chain = new RealInterceptorChain(interceptors, null, null, null, 0,
15             originalRequest, this, eventListener, client.connectTimeoutMillis(),
16             client.readTimeoutMillis(), client.writeTimeoutMillis());
17
18         return chain.proceed(originalRequest);
19     }

```

## 每一个拦截器的作用：

- **BridgeInterceptor**: 主要对 Request 中的 Head 设置默认值  
比如 Content-Type、Keep-Alive、Cookie 等
- **CacheInterceptor**: 负责 HTTP 请求的缓存处理
- **ConnectInterceptor**: 负责建立与服务器地址之间的连接，也就是 TCP 链接
- **CallServerInterceptor**: 负责向服务器发送请求，并从服务器拿到远端数据结果

## 18、Bitmap

用来描述一张图片的长、宽、颜色等信息

可以使用BitmapFactory来将某一个路径下的图片解析为Bitmap对象

```

1 public class MainActivity extends AppCompatActivity {
2
3     private static final String TAG = "Bitmap";
4
5     @Override
6     protected void onCreate(Bundle savedInstanceState) {
7         super.onCreate(savedInstanceState);
8         setContentView(R.layout.activity_main);
9
10        Bitmap bitmap = BitmapFactory.decodeResource(getResources(), R.drawable.rodman);
11        Log.i(TAG, "bitmap size is " + bitmap.getAllocationByteCount());
12
13        imageView.setImageBitmap(bitmap);
14    }

```

I/Bitmap ( 5673): bitmap size is 1440000      宽 \* 高 \* 4 = 600 \* 600 \* 4 = 1440000

BitmapFactory在解析图片的过程中，根据当前设备屏幕密度和图片所在的drawable目录来做一个对比，根据这个对比值进行缩放操作

公式：

缩放比例scale=当前设备屏幕密度/图片所在drawable目录对应的屏幕密度

Bitmap实际大小=宽\*scale \* 高 \* scale \* Config对应的存储像素数

在 Android 中，各个 drawable 目录对应的屏幕密度

目录	drawable-mdpi	drawable-hdpi	drawable-xhdpi	drawable-xxhdpi	drawable-xxxhdpi
density	1	1.5	2	3	4
densityDpi	160	240	320	480	640

Android中的图片不仅可以保存在drawable目录中，还可以保存在assets目录下，然后通过AssetManager获取图片的输入流

```

1 try {
2
3     InputStream inputStream = getAssets().open("rodman.png");
4
5     Bitmap bitmap = BitmapFactory.decodeStream(inputStream);
6
7     Log.e(TAG, "bitmap size is " + bitmap.getAllocationByteCount());
8
9 } catch (IOException e) {}

```

I/Bitmap ( 5673): bitmap size is 1440000

Bitmap优化：

- 缩略优化：

- 修改图片加载的config，一个像素占用两个字节

```

1  public class MainActivity extends AppCompatActivity {
2
3      private static final String TAG = "Bitmap";
4
5      @Override
6      protected void onCreate(Bundle savedInstanceState) {
7          super.onCreate(savedInstanceState);
8          setContentView(R.layout.activity_main);
9          ImageView imageView = findViewById(R.id.image);
10
11         BitmapFactory.Options options = new BitmapFactory.Options();
12         options.inPreferredConfig = Bitmap.Config.RGB_565;
13         Bitmap bitmap = BitmapFactory.decodeResource(getResources(), R.drawable.rodman);
14         Log.i(TAG, "bitmap size is " + bitmap.getByteCount());
15
16         imageView.setImageBitmap(bitmap);
17     }
18 }

```

```

1  public class MainActivity extends AppCompatActivity {
2
3      private static final String TAG = "Bitmap";
4
5      @Override
6      protected void onCreate(Bundle savedInstanceState) {
7          super.onCreate(savedInstanceState);
8          setContentView(R.layout.activity_main);
9
10         ImageView imageView = findViewById(R.id.image);
11
12         BitmapFactory.Options options = new BitmapFactory.Options();
13         options.inPreferredConfig = Bitmap.Config.RGB_565;
14         options.inSampleSize = 2; // 宽和高每隔2个像素进行一次采样
15
16         Bitmap bitmap = BitmapFactory.decodeResource(getResources(), R.drawable.rodman, options);
17         Log.i(TAG, "bitmap size is " + bitmap.getByteCount());
18
19         imageView.setImageBitmap(bitmap);
20     }

```

## Bitmap复用

- 使用Options.inBitmap优化

```

1  public class BitmapPoolActivity extends AppCompatActivity {
2
3      private static final String TAG = BitmapPoolActivity.class.getSimpleName();
4
5      ImageView poolImage;
6      Bitmap reuseBitmap; ①
7      int resIndex;
8      int[] resIds = {R.drawable.rodman, R.drawable.rodman2};
9
10     @Override
11     protected void onCreate(@Nullable Bundle savedInstanceState) {
12         super.onCreate(savedInstanceState);
13         setContentView(R.layout.activity_pool);
14         poolImage = findViewById(R.id.poolImage);
15         final BitmapFactory.Options options = new BitmapFactory.Options();
16         options.inMutable = true;
17         reuseBitmap = BitmapFactory.decodeResource(getResources(), resIds[0], options);
18     }
19
20     public void switchImage(View view) {
21         poolImage.setImageBitmap(getBitmap());
22     }
23
24     private Bitmap getBitmap() {
25         final BitmapFactory.Options options = new BitmapFactory.Options();
26         options.inJustDecodeBounds = true;
27         BitmapFactory.decodeResource(getResources(), resIds[resIndex % 2], options);
28         if (canUseForInBitmap(reuseBitmap, options)) {
29             options.inMutable = true;
30             options.inBitmap = reuseBitmap; ②
31         }
32         options.inJustDecodeBounds = false;
33         return BitmapFactory.decodeResource(getResources(), resIds[resIndex++ % 2], options);
34     }
35 }

```

**注意：**在上述 `getBitmap` 方法中，复用 `inBitmap` 之前

需要调用 `canUseForInBitmap` 方法来判断 `reuseBitmap` 是否可以被复用

Bitmap 的复用有一定的限制：

- 在 Android 4.4 版本之前，只能重用相同大小的 Bitmap 内存区域
- 4.4 之后可以重用任何 Bitmap 的内存区域，只要这块内存比将要分配内存的 bitmap 大就可以

## 20、Window、Activity、View

Activity的setContentView

```
1 private Window mWindow;
2
3 public void setContentView(@LayoutRes int layoutResID) {
4     getWindow().setContentView(layoutResID);
5     initWindowDecorActionBar();
6 }
7
8 public Window getWindow() {
9     return mWindow;
10 }
```

屏幕绘制在Window中绘制，通过getWindow传入

startActivity 的过程中，最终代码会调用到 ActivityThread 中的 performLaunchActivity 方法  
通过反射创建 Activity 对象，并执行其 attach 方法

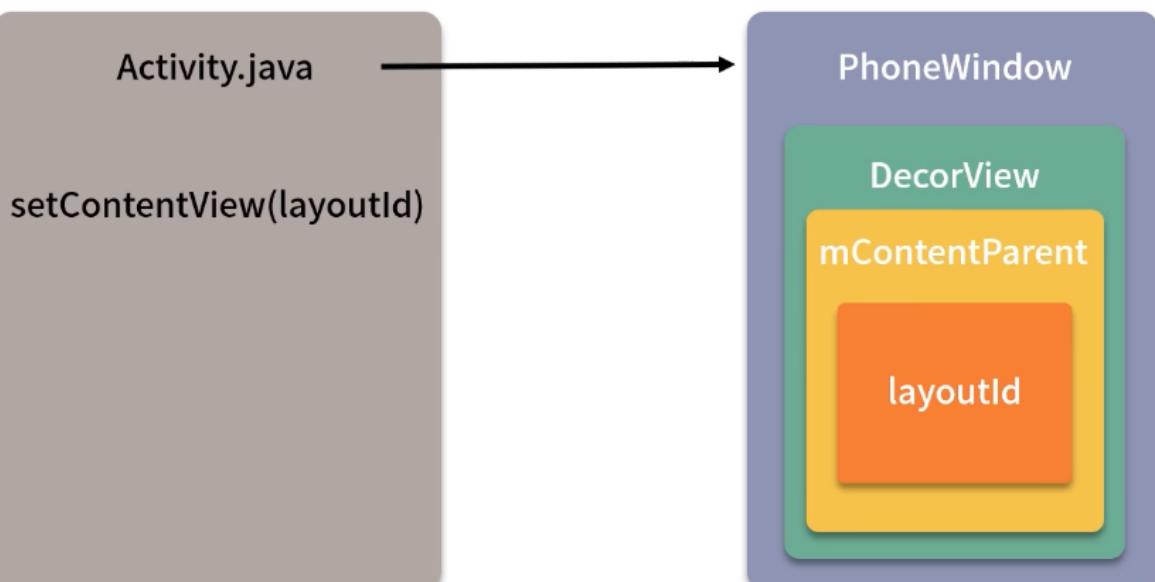
```
1 final void attach(Context context, ActivityThread aThread,
2                   Instrumentation instr, IBinder token, int ident,
3                   Application application, Intent intent, ActivityInfo info,
4                   CharSequence title, Activity parent, String id,
5                   NonConfigurationInstances lastNonConfigurationInstances,
6                   Configuration config, String referrer, IVoiceInteractor voiceInteractor,
7                   Window window, ActivityConfigCallback activityConfigCallback) {
8 ...
9
10    mWindow = new PhoneWindow(this, window, activityConfigCallback);
11    mWindow.setWindowControllerCallback(this);
12    mWindow.setCallback(this);
13    mWindow.setOnWindowDismissedCallback(this);
14    mWindow.getLayoutInflater().setPrivateFactory(this);
15    ...
16    mWindow.set WindowManager(
17        (WindowManager)context.getSystemService(Context.WINDOW_SERVICE),
18        mToken, mComponent.flattenToString(),
19        (info.flags & ActivityInfo.FLAG_HARDWARE_ACCELERATED) != 0);
20 }
```

调用 setWindowManager 方法，将系统 WindowManager 传给 PhoneWindow

```
1 private WindowManager mWindowManager;
2
3 public void setWindowManager(WindowManager wm, IBinder appToken, String appName,
4                               boolean hardwareAccelerated) {
5     mAppToken = appToken;
6     mName = appName;
7     mHardwareAccelerated = hardwareAccelerated
8         || SystemProperties.getBoolean(PROPERTY_HARDWARE_UI, false);
9     if (wm == null) {
10         wm = (WindowManager)mContext.getSystemService(Context.WINDOW_SERVICE);
11     }
12     mWindowManager = ((WindowManagerImpl)wm).createLocal WindowManager(this);
13 }
14
15 public WindowManagerImpl createLocal WindowManager(Window parentWindow) {
16     return new WindowManagerImpl(mContext, parentWindow);
17 }
```

具体的 setContentView

```
1    @Override
2    public void setContentView(int layoutResID) {
3        if (mContentParent == null) {
4            installDecor(); ①
5        } else if (!hasFeature(FEATURE_CONTENT_TRANSITIONS)) {
6            mContentParent.removeAllViews();
7        }
8
9        if (hasFeature(FEATURE_CONTENT_TRANSITIONS)) {
10            final Scene newScene = Scene.getSceneForLayout(mContentParent, layoutResID,
11                getContext());
12            transitionTo(newScene);
13        } else {
14            mLayoutInflater.inflate(layoutResID, mContentParent); ②
15        }
16        ...
17    }
18
19    private void installDecor() {
20        // 初始化DecorView
21        if (mDecor == null) {
22            mDecor = generateDecor(-1);
23        }
24        // 初始化mContentParent
25        if (mContentParent == null) {
26            mContentParent = generateLayout(mDecor);
27        }
28    }
}
```



通过setContentView调用window的PhoneWindow, DecorView是一个FragmentLayout, mContentParent是一个viewgroup, layoutId自定义布局

DecorView什么时候被绘制?

OnCreate阶段 只是初始化了Activity需要显示的内容

OnResume阶段 会将PhoneWindow中的DecorView真正绘制到屏幕上。

```
1 final void handleResumeActivity(IBinder token,
2         boolean clearHide, boolean isForward, boolean reallyResume, int seq, String reason) {
3     ViewManager wm = a.getWindowManager();
4     ...
5     if (a.mVisibleFromClient) {
6         if (!a.mWindowAdded) {
7             a.mWindowAdded = true;
8             wm.addView(decor, l);
9         } else {
10            a.onWindowAttributesChanged(l);
11        }
12    }
13 }
```

WindowManager的addview效果:

- decorView被渲染 绘制到屏幕上显示
- Decorview可以接收屏幕触摸事件

把一个 View 作为窗口添加到 WMS 的过程是由 **WindowManager** 来完成的  
WindowManager 是接口类型, 它真正的实现者是 WindowManagerImpl 类

```
1 // WindowManagerImpl.java
2 @Override
3 public void addView(@NonNull View view, @NonNull ViewGroup.LayoutParams params) {
4     applyDefaultToken(params);
5     mGlobal.addView(view, params, mContext.getDisplay(), mParentWindow);
6 }
7
8 // WindowManagerGlobal.java
9 public void addView(View view, ViewGroup.LayoutParams params,
10                     Display display, Window parentWindow) {
11     ...
12     ViewRootImpl root;
13     synchronized (mLock) {
14         ...
15         root = new ViewRootImpl(view.getContext(), display);
16         view.setLayoutParams(params);
17
18         mViews.add(view);
19         mRoots.add(root);
20         mParams.add(params);
21
22         try {
23             root.setView(view, params, panelParentView);
24         } catch (RuntimeException e) {
25             ...
26     }
}
```

```

1  public void setView(View view, WindowManager.LayoutParams attrs, View panelParentView) {
2      int res;
3
4      requestLayout(); //View的绘制流程 ①
5
6      if ((mWindowAttributes.inputFeatures
7          & WindowManager.LayoutParams.INPUT_FEATURE_NO_INPUT_CHANNEL) == 0) {
8          //创建InputChannel
9          mInputChannel = new InputChannel();
10     }
11
12     try {
13         ...
14         res = mWindowSession.addToDisplay(mWindow, mSeq, mWindowAttributes,
15             getHostVisibility(), mDisplay.getDisplayId(),
16             mAttachInfo.mContentInsets, mAttachInfo.mStableInsets,
17             mAttachInfo.mOutsets, mInputChannel); ②
18     }
19
20     ...
21 }

```

```

1  // WindowManagerGlobal.java
2  private static IWindowSession sWindowSession;
3
4  public static IWindowSession getWindowSession() {
5      synchronized (WindowManagerGlobal.class) {
6          if (sWindowSession == null) {
7              try {
8                  InputMethodManager imm = InputMethodManager.getInstance();
9                  IWindowManager windowManager = getWindowManagerService();
10                 sWindowSession = windowManager.openSession(
11                     new IWindowSessionCallback.Stub() {
12                         @Override
13                         public void onAnimatorScaleChanged(float scale) {
14                             ValueAnimator.setDurationScale(scale);
15                         }
16                     },
17                     imm.getClient(), imm.getInputContext());
18             } catch (RemoteException e) {
19                 throw e.rethrowFromSystemServer();
20             }
21         }
22         return sWindowSession;
23     }
24 }

```

```

1  @Override
2  public int addToDisplay(IWindow window, int seq, WindowManager.LayoutParams attrs,
3      int viewVisibility, int displayId, Rect outFrame, Rect outContentInsets,
4      Rect outStableInsets, Rect outOutsets,
5      DisplayCutout.ParcelableWrapper outDisplayCutout, InputChannel outInputChannel) {
6      return mService.addWindow(this, window, seq, attrs, viewVisibility, displayId, outFrame,
7          outContentInsets, outStableInsets, outOutsets, outDisplayCutout, outInputChannel);
8  }

```

图中的 mService 就是 WMS

addView成功的一个标志就是能够接受触屏事件

通过堆setContentView流程的分析，可以看出：

添加View的操作实质上是PhoneWindow在全盘操作，背后负责人是WMS

当触屏事件发生后，Touch事件首先被传入到Activity中，然后被下发到布局的ViewGroup或View中。

## Touch事件如何传递到Activity上？

ViewRootImpl 中的 setView 方法中，有一项重要的操作就是设置输入事件的处理

```
1 public void setView(View view, WindowManager.LayoutParams attrs, View panelParentView) {
2     ...
3     res = mWindowSession.addToDisplay(mWindow, mSeq, mWindowAttributes,
4             getHostVisibility(), mDisplay.getDisplayId(), mWinFrame,
5             mAttachInfo.mContentInsets, mAttachInfo.mStableInsets,
6             mAttachInfo.mOutsets, mAttachInfo.mDisplayCutout, mInputChannel);
7     ...
8     // Set up the input pipeline.
9     CharSequence counterSuffix = attrs.getTitle();
10    mSyntheticInputStage = new SyntheticInputStage();
11    InputStage viewPostImeStage = new ViewPostImeInputStage(mSyntheticInputStage);
12    InputStage nativePostImeStage = new NativePostImeInputStage(viewPostImeStage,
13            "aq:native-post-ime:" + counterSuffix);
14    InputStage earlyPostImeStage = new EarlyPostImeInputStage(nativePostImeStage);
15    InputStage imeStage = new ImeInputStage(earlyPostImeStage,
16            "aq:ime:" + counterSuffix);
17    InputStage viewPreImeStage = new ViewPreImeInputStage(imeStage);
18    InputStage nativePreImeStage = new NativePreImeInputStage(viewPreImeStage,
19            "aq:native-pre-ime:" + counterSuffix);
20 }
```

整个流程需要注意：

1. 一个 Activity 中有一个 window，也就是 PhoneWindow 对象

在 PhoneWindow 中有一个 DecorView，在 setContentView 中会将 layout 填充到此 DecorView 中

2. 一个应用进程中只有一个 WindowManagerGlobal 对象，因为在 ViewRootImpl 中它是 static 静态类型

3. 每一个 PhoneWindow 对应一个 ViewRootImpl 对象

4. WindowManagerGlobal 通过调用 ViewRootImpl 的 setView 方法，完成 window 的添加过程

5. ViewRootImpl 的 setView 方法中主要完成两件事情：View 渲染（requestLayout）以及接收触屏事件

## 22、如何通过View进行渲染？

**ViewRootImpl** 在整个流程中，起着承上启下的作用

- ViewRootImpl 中通过 Binder 通信机制，远程调用 WindowSession 将 View 添加到 Window 中
- ViewRootImpl 在添加 View 之前，需要调用 requestLayout 方法，执行完成的 View 树的渲染操作。

ViewRootImpl requestLayout 流程

请求布局操作

```
1 public void setView(View view, WindowManager.LayoutParams attrs, View panelParentView) {
2     synchronized (this) {
3         ...
4         // Schedule the first layout -before- adding to the window
5         // manager, to make sure we do the relayout before receiving
6         // any other events from the system.
7         requestLayout();
8         ...
9         res = mWindowSession.addToDisplay(mWindow, mSeq, mWindowAttributes,
10                 getHostVisibility(), mDisplay.getDisplayId(), mWinFrame,
11                 mAttachInfo.mContentInsets, mAttachInfo.mStableInsets,
12                 mAttachInfo.mOutsets, mAttachInfo.mDisplayCutout, mInputChannel);
13     ...
14 }
```

```
1  @Override
2  public void requestLayout() {
3      if (!mHandlingLayoutInLayoutRequest) {
4          checkThread(); // 1
5          mLayoutRequested = true; // 2
6          scheduleTraversals();
7      }
8  }
```

1: 检测是否是合法进程。一般是主线程

2:

```
1  void scheduleTraversals() {
2      if (!mTraversalScheduled) {
3          mTraversalScheduled = true;
4          mTraversalBarrier = mHandler.getLooper().getQueue().postSyncBarrier(); // 1
5          mChoreographer.postCallback(
6              Choreographer.CALLBACK_TRAVERSAL, mTraversalRunnable, null); // 2
7          ...
8      }
9  }
```

```
1  public void postCallback(int callbackType, Runnable action, Object token) {
2      postCallbackDelayed(callbackType, action, token, 0);
3  }
4
5  public void postCallbackDelayed(int callbackType,
6      Runnable action, Object token, long delayMillis) {
7      if (action == null) {
8          throw new IllegalArgumentException("action must not be null");
9      }
10     ...
11     postCallbackDelayedInternal(callbackType, action, token, delayMillis);
12 }
13
14 private void postCallbackDelayedInternal(int callbackType,
15     Object action, Object token, long delayMillis) {
16     synchronized (mLock) {
17         final long now = SystemClock.uptimeMillis();
18         final long dueTime = now + delayMillis;
19         mCallbackQueues[callbackType].addCallbackLocked(dueTime, action, token);
20
21         if (dueTime <= now) {
22             scheduleFrameLocked(now);
23         } else {
24             Message msg = mHandler.obtainMessage(MSG_DO_SCHEDULE_CALLBACK, action);
25             msg.arg1 = callbackType;
26             msg.setAsynchronous(true);
27             mHandler.sendMessageAtTime(msg, dueTime);
28         }
29     }
30 }
```

```

1 final class TraversalRunnable implements Runnable {
2     @Override
3     public void run() {
4         doTraversal();
5     }
6 }
7
8 void doTraversal() {
9     if (mTraversalScheduled) {
10         mTraversalScheduled = false;
11         mHandler.getLooper().getQueue().removeSyncBarrier(mTraversalBarrier);
12
13         if (mProfile) {
14             Debug.startMethodTracing("ViewAncestor");
15         }
16
17         performTraversals(); // Perform traversals
18
19         if (mProfile) {
20             Debug.stopMethodTracing();
21             mProfile = false;
22         }
23     }
24 }

```

performTraversals()方法开始View绘制： measure->layout->draw

```

1 private void performTraversals() {
2     ...
3     // 调用performMeasure进行测量工作
4     measureHierarchy(host, lp, res, desiredWindowWidth, desiredWindowHeight);
5     // 执行onLayout操作
6     performLayout(lp, mWidth, mHeight);
7     // 执行onDraw操作
8     performDraw();
9     ...
10 }

```

```

1 private boolean measureHierarchy(final View host, final WindowManager.LayoutParams lp,
2                                 final Resources res, final int desiredWindowWidth, final int desiredWindowHeight) {
3
4     int childWidthMeasureSpec;
5     int childHeightMeasureSpec;
6     boolean windowSizeMayChange = false;
7
8     ...
9
10    if (!goodMeasure) {
11        childWidthMeasureSpec = getRootMeasureSpec(desiredWindowWidth, lp.width);
12        childHeightMeasureSpec = getRootMeasureSpec(desiredWindowHeight, lp.height); // Measure
13
14        performMeasure(childWidthMeasureSpec, childHeightMeasureSpec);
15
16        if (mWidth != host.getMeasuredWidth() || mHeight != host.getMeasuredHeight()) {
17            windowSizeMayChange = true;
18        }
19    }
20    return windowSizeMayChange;

```

```

1 private void performMeasure(int childWidthMeasureSpec, int childHeightMeasureSpec) {
2     if (mView == null) {
3         return;
4     }
5     Trace.traceBegin(Trace.TRACE_TAG_VIEW, "measure");
6     try {
7         mView.measure(childWidthMeasureSpec, childHeightMeasureSpec);
8     } finally {
9         Trace.traceEnd(Trace.TRACE_TAG_VIEW);
10    }
11 }

```

```
1 private void performDraw() {
2     ...
3     try {
4         draw(fullRedrawNeeded);
5     }
6 }
7
8 private void draw(boolean fullRedrawNeeded) {
9     Surface surface = mSurface;
10    ...
11    if (!dirty.isEmpty() || mIsAnimating || accessibilityFocusDirty) {
12        if (mAttachInfo.mHardwareRenderer != null && mAttachInfo.mHardwareRenderer.isEnabled()) {
13            mAttachInfo.mHardwareRenderer.draw(mView, mAttachInfo, this); // 1
14        } else {
15            if (!drawSoftware(surface, mAttachInfo, xOffset, yOffset, scalingRequired, dirty)) { // 2
16                return;
17            }
18        }
19    }
20 }
```

//注释1

ViewRootImpl中有一个非常重要的对象Surface

ViewRootImpl的一个核心功能是负责UI渲染

在ViewRootImpl中会将在draw方法中绘制的UI元素，绑定到这个Surface上，Surface中的内容最终会被传递给底层的SurfaceFliger。

最终将Surface中的内容进行合成并显示到屏幕上。

//注释2

```
1 private boolean drawSoftware(Surface surface, AttachInfo attachInfo, int xoff, int yoff,
2                             boolean scalingRequired, Rect dirty) {
3
4     final Canvas canvas;
5
6     canvas = mSurface.lockCanvas(dirty);
7
8     try {
9
10         mView.draw(canvas); // 图形DecorView 1
11     } finally {
12         surface.unlockCanvasAndPost(canvas); 2
13     }
14     return true;
15 }
```

默认情况下，软件绘制没有采用GPU渲染的方式，drawSoftware工作完全由CPU来完成。

```
1 public void draw(Canvas canvas) {
2     final int privateFlags = mPrivateFlags;
3
4     final boolean dirtyOpaque = (privateFlags & PFLAG_DIRTY_MASK) == PFLAG_DIRTY_OPAQUE &&
5             (mAttachInfo == null || !mAttachInfo.mIgnoreDirtyState);
6     mPrivateFlags = (privateFlags & ~PFLAG_DIRTY_MASK) | PFLAG_DRAWN;
7
8     int saveCount;
9     if (!dirtyOpaque) {
10         drawBackground(canvas); ①
11     }
12
13     final int viewFlags = mViewFlags;
14     boolean horizontalEdges = (viewFlags & FADING_EDGE_HORIZONTAL) != 0;
15     boolean verticalEdges = (viewFlags & FADING_EDGE_VERTICAL) != 0;
16
17     if (!verticalEdges && !horizontalEdges) {
18         if (!dirtyOpaque) onDraw(canvas); ②
19
20         dispatchDraw(canvas); ③
21
22         // Overlay is part of the content and draws beneath Foreground
23         if (mOverlay != null && !mOverlay.isEmpty()) {
24             mOverlay.getOverlayView().dispatchDraw(canvas);
25         }
26
27         onDrawForeground(canvas);
28
29         // we're done...
30         return;
31     }
32
33     .....
34 }
```

启动硬件加速：

在 ViewRootImpl 的 draw 方法中，通过如下方法判断是否启用硬件加速

```
1 if (mAttachInfo.mThreadedRenderer != null && mAttachInfo.mThreadedRenderer.isEnabled()) {
2     // 执行硬件加速绘制
3 }
```

在 AndroidManifest 清单文件中，指定 Application 或者某一个 Activity 支持硬件加速

```
1 <application android:hardwareAccelerated="true">
2     <activity ... />
3     <activity android:hardwareAccelerated="false" />
4 </application>
```

可以进行粒度更小的硬件加速设置，比如设置某个 View 支持硬件加速

```
1 // Window级别
2 getWindow().setFlags(
3     WindowManager.LayoutParams.FLAG_HARDWARE_ACCELERATED,
4     WindowManager.LayoutParams.FLAG_HARDWARE_ACCELERATED);
5
6 // View级别
7 View.setLayerType(View.LAYER_TYPE_SOFTWARE, null);
```

## Invalidate 轻量刷新

通过 invalidate 来刷新 View，与 requestLayout 的区别在于

它不一定会触发 View 的 measure 和 layout 的操作，多数情况下只会执行 draw 操作

```
1 public final void measure(int widthMeasureSpec, int heightMeasureSpec) {  
2     ...  
3     final boolean forceLayout = (mPrivateFlags & PFLAG_FORCE_LAYOUT)  
4             == PFLAG_FORCE_LAYOUT;  
5     ...  
6     if (forceLayout || needsLayout) {  
7         onMeasure(widthMeasureSpec, heightMeasureSpec);  
8         mPrivateFlags |= PFLAG_LAYOUT_REQUIRED;  
9     }  
10 }
```

invalidate 与 postInvalidate 两者之间的区别

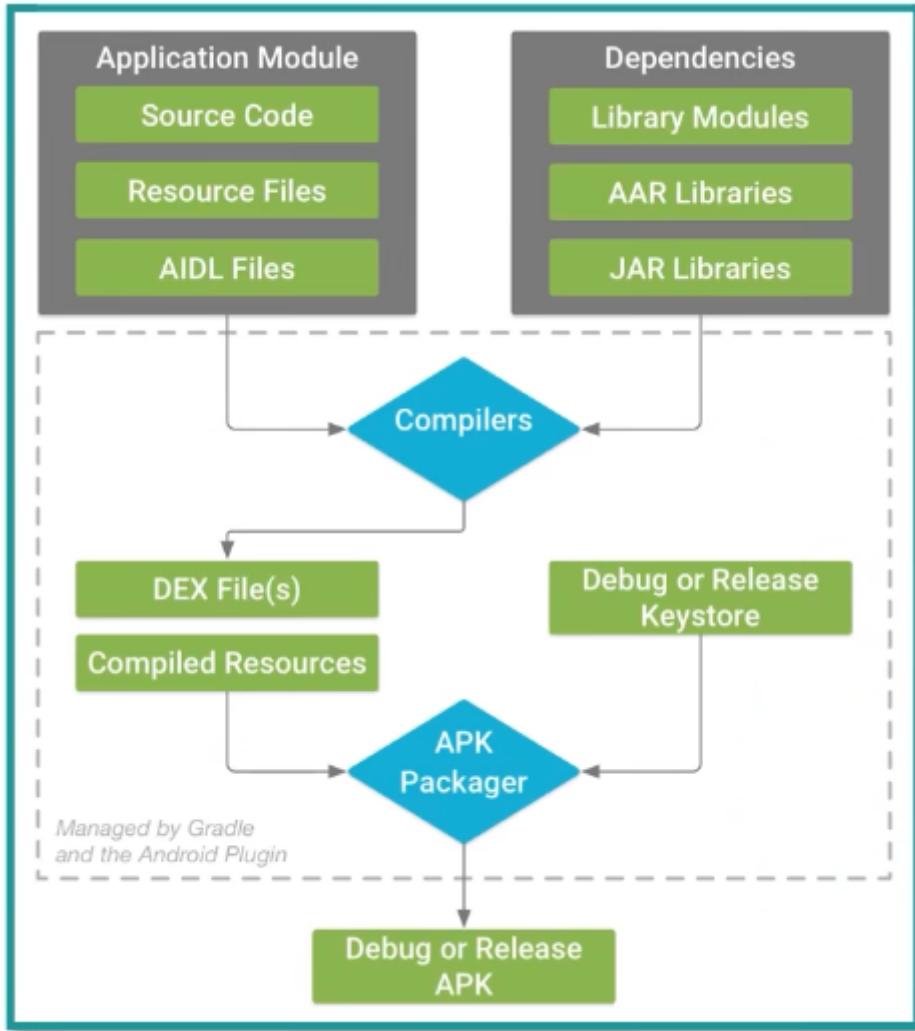
invalidate 是在UI线程调用， postInvalidate 是在非UI线程调用

```
1 // View.java  
2 public void postInvalidate() {  
3     postInvalidateDelayed(0);  
4 }  
5  
6 public void postInvalidateDelayed(long delayMilliseconds) {  
7     final AttachInfo attachInfo = mAttachInfo;  
8     if (attachInfo != null) {  
9         attachInfo.mViewRootImpl.dispatchInvalidateDelayed(this, delayMilliseconds);  
10    }  
11 }
```

- 主要介绍了 ViewRootImpl 是如何执行 View 的渲染操作的  
其中核心方法在 performTraversals 方法中会按顺序执行 measure -> layout -> draw 操作
- 介绍了软件绘制和硬件加速的区别
- 介绍了 View 刷新的两种方式 Invalidate 和 postInvalidate

## 22、APP安装

PackageManagerService (PMS)



每一个module中的内容可以分为2部分：

- resources资源文件
- Java或者Kotlin源代码

### 编译阶段--Resources资源文件

资源文件包括项目中res目录下的各种xml文件、动画、drawable图片、音视频等

**AAPT工具** 负责编译项目中的这些资源文件，所有资源文件会被编译处理

XML文件(Drawable图片除外)会被编译成二进制文件

assets和raw目录下的资源并不会被编译，会被原封不动的打包到apk压缩包中。

资源文件编译之后的产物包括两部分：

- resources.arsc文件（保存的是一个资源索引表）
- R.java文件（定义了各个资源ID差两个常量）

resources.arsc相当于一个资源索引表，也可以理解为一个map映射表，map的 **key** 是R.java中的资源ID，而value是其对应的资源所在路径。

### 编译阶段--源码部分

项目中源代码首先会通过javac编译成.class字节码文件

然后这些.class文件连同依赖的三方库中的.class文件一同被dx工具优化为.dex文件

如果由分包，那么也可能会生成多个.dex文件

源代码文件也包括 AIDL接口文件编译之后生成的.java文件

Android项目中如果包含.aidl接口文件，这些.aidl文件会被编译成.java文件。

## 打包阶段

使用工具APK Builder将经过编译之后的resource和.dex文件一起打包到apk中，实际上被打包到apk中的还有一些其他资源，比如AndroidManifest.xml清单列表和三方库中使用的动态库.so文件。

apk创建好之后，需要使用工具jarsigner对其进行签名，签名之后会生成META-INF文件夹，次文件夹中保存着跟签名相关的各个文件

- CETF.SF:生成每个文件相对的密钥
- MANIFEST.MF：数字签名信息
- xxx.SF:JAR文件的签名文件
- xxx.DSA：对输出文件的签名和公钥

实际打包过程还会多一步APK优化操作

使用工具zipalign对APK中的未压缩资源(图片、视频等)进行对其操作，让资源按照4字节的边界进行对其：**主要为了加快资源的访问速度**。如果每个资源的开始位置都是上一个资源之后的 $4 * n$ 字节，那么访问下一个资源就不用编译，直接跳到 $4 * n$ 字节处判断是不是一个新的资源即可

## 安装过程

当点击某一个App安装包进行安装时，首先会弹出一个系统界面指示我们进行安装操作

这个界面是Android Framework中预置的一个Activity-PackageInstallerActivity.java，当点击安装后，PackageInstallerActivity 最终会将所安装的apk信息通过PackageInstallerSession 传给PMS

```
1 // PackageInstallerSession.java
2 private void commitLocked()
3     throws PackageManagerException {
4     ...
5     mPm.installStage(mPackageName, stageDir, localObserver, params,
6                       mInstallerPackageName, mInstallerUid, user, mSigningDetails);
7 }
```

整个apk的安装过程可以分为两大步

- 拷贝安装包
- 装载代码

```

1 void installStage(String packageName, File stagedDir,
2                     IPackageInstallObserver2 observer, PackageInstaller.SessionParams sessionParams,
3                     String installerPackageName, int installerUid, UserHandle user,
4                     PackageParser.SigningDetails signingDetails) {
5     if (DEBUG_INSTANT) {
6         if ((activeInstallSession.getSessionParams().installFlags
7             & PackageManager.INSTALL_INSTANT_APP) != 0) {
8             Slog.d(TAG, "Ephemeral install of " + activeInstallSession.getPackageName());
9         }
10    }
11
12    1 final Message msg = mHandler.obtainMessage(INIT_COPY);
13
14    2 final InstallParams params = new InstallParams(origin, null, observer,
15                     sessionParams.installFlags, installerPackageName, sessionParams.volumeUuid,
16                     verificationInfo, user, sessionParams.abiOverride,
17                     sessionParams.grantedRuntimePermissions, signingDetails, installReason);
18    params.setTraceMethod("installStage").setTraceCookie(System.identityHashCode(params));
19    msg.obj = params;
20
21    Trace.asyncTraceBegin(TRACE_TAG_PACKAGE_MANAGER, "installStage",
22                          System.identityHashCode(msg.obj));
23    Trace.asyncTraceBegin(TRACE_TAG_PACKAGE_MANAGER, "queueInstall",
24                          System.identityHashCode(msg.obj));
25
26    mHandler.sendMessage(msg);
27 }

```

```

1 class PackageHandler extends Handler {
2     public void handleMessage(Message msg) {
3         try {
4             doHandleMessage(msg);
5         } finally {
6             Process.setThreadPriority(Process.THREAD_PRIORITY_BACKGROUND);
7         }
8     }
9
10    void doHandleMessage(Message msg) {
11        switch (msg.what) {
12            case INIT_COPY: {
13                HandlerParams params = (HandlerParams) msg.obj; 1
14                ...
15                if (!connectToService()) { 2
16                    ...
17                } else {
18                    mPendingInstalls.add(idx, params);
19                }
20            }
21        }
22    }

```

```

1     private boolean connectToService() {
2         if (DEBUG_INSTALL) Log.i(TAG, "Trying to bind to DefaultContainerService");
3         Intent service = new Intent().setComponent(DEFAULT_CONTAINER_COMPONENT); 3
4         Process.setThreadPriority(Process.THREAD_PRIORITY_DEFAULT);
5         if (mContext.bindServiceAsUser(service, mDefContainerConn,
6             Context.BIND_AUTO_CREATE, UserHandle.SYSTEM)) {
7             Process.setThreadPriority(Process.THREAD_PRIORITY_BACKGROUND);
8             mBound = true;
9             return true;
10        }
11        Process.setThreadPriority(Process.THREAD_PRIORITY_BACKGROUND);
12        return false;
13    }

```

```
1 class DefaultContainerConnection implements ServiceConnection {
2     public void onServiceConnected(ComponentName name, IBinder service) {
3         if (DEBUG_SD_INSTALL) Log.i(TAG, "onServiceConnected");
4         IMediaContainerService imcs =
5             IMediaContainerService.Stub.asInterface(service);
6         mHandler.sendMessage(mHandler.obtainMessage(MCS_BOUND, imcs));
7     }
8
9     public void onServiceDisconnected(ComponentName name) {
10        if (DEBUG_SD_INSTALL) Log.i(TAG, "onServiceDisconnected");
11    }
12 }
```

```
1     void doHandleMessage(Message msg) {
2         switch (msg.what) {
3             ...
4             case MCS_BOUND: {
5                 HandlerParams params = mPendingInstalls.get(0);
6                 if (params != null) {
7                     if (params.startCopy()) {
8                         ...
9                         if (mPendingInstalls.size() > 0) {
10                             mPendingInstalls.remove(0);
11                         }
12                     }
13                     ...
14                 }
15                 break;
16             }
17         }
18     }
19 }
```

```
1 class InstallParams extends HandlerParams {
2     public void handleStartCopy() throws RemoteException {
3         ...
4         // If we're already staged, we've firmly committed to an install location
5         if (origin.staged) {
6             if (origin.file != null) {
7                 installFlags |= PackageManager.INSTALL_INTERNAL;
8                 installFlags &= ~PackageManager.INSTALL_EXTERNAL;
9             } else if (origin.cid != null) {
10                 installFlags |= PackageManager.INSTALL_EXTERNAL;
11                 installFlags &= ~PackageManager.INSTALL_INTERNAL;
12             }
13         }
14
15         ① final boolean onSd = (installFlags & PackageManager.INSTALL_EXTERNAL) != 0;
16         final boolean onInt = (installFlags & PackageManager.INSTALL_INTERNAL) != 0;
17         final boolean ephemeral = (installFlags & PackageManager.INSTALL_INSTANT_APP) != 0;
18
19         ② final InstallArgs args = createInstallArgs(this);
20
21         ...
22
23         ③ ret = args.copyApk(mContainerService, true);
24     }
25 }
```

```

1 int copyApk(IMediaContainerService imcs, boolean temp) throws RemoteException {
2     ...
3     return doCopyApk(imcs, temp);
4 }
5
6 private int doCopyApk(IMediaContainerService imcs, boolean temp) throws RemoteException {
7     ...
8     final File tempDir = mInstallerService.allocateStageDirLegacy(
9             volumeUuid, isEphemeral); 1
10    ...
11
12    final IParcelFileDescriptorFactory target = new IParcelFileDescriptorFactory.Stub() {
13        @Override
14        public ParcelFileDescriptor open(String name, int mode) throws RemoteException {
15            if (!FileUtils.isValidExtFilename(name)) {
16                ...
17            }
18        };
19        int ret = PackageManager.INSTALL_SUCCEEDED;
20
21        ret = imcs.copyPackage(origin.file.getAbsolutePath(), target); 2
22
23        ...
24
25        ret = NativeLibraryHelper.copyNativeBinariesWithOverride(handle, libraryRoot,
26                           abiOverride); 3
27    }
}

```

```

1 // DefaultContainerService.java
2 @Override
3 public int copyPackage(String packagePath, IParcelFileDescriptorFactory target) {
4     ...
5     PackageLite pkg = null;
6     final File packageFile = new File(packagePath);
7     pkg = PackageParser.parsePackageLite(packageFile, 0);
8     return copyPackageInner(pkg, target);
9 }
10
11 private int copyPackageInner(PackageLite pkg, IParcelFileDescriptorFactory target)
12     throws IOException, RemoteException {
13
14     copyFile(pkg.baseCodePath, target, "base.apk"); 4
15
16     if (!ArrayUtils.isEmpty(pkg.splitNames)) {
17         for (int i = 0; i < pkg.splitNames.length; i++) {
18             copyFile(pkg.splitCodePaths[i], target, "split_" + pkg.splitNames[i] + ".apk");
19         }
20     }
21
22     return PackageManager.INSTALL_SUCCEEDED;
23 }
24 private void copyFile(String sourcePath, IParcelFileDescriptorFactory target, String targetName)
25     throws IOException, RemoteException {
26     InputStream in = null;
27     OutputStream out = null;
28     try {
29         in = new FileInputStream(sourcePath);
30         out = new ParcelFileDescriptor.AutoCloseOutputStream(
31                 target.open(targetName, ParcelFileDescriptor.MODE_READ_WRITE));
32         FileUtils.copy(in, out);
33     } finally {
34         IoUtils.closeQuietly(out);
35         IoUtils.closeQuietly(in);
36     }
37 }

```

装载代码：

```

1  final boolean startCopy() {
2      ...
3      handleStartCopy();
4      ...
5
6      handleReturnCode();
7
8      ...
9  }
10
11 @Override
12 void handleReturnCode() {
13     if (mArgs != null) {
14         processPendingInstall(mArgs, mRet);
15     }
16 }
```



```

1  private void processPendingInstall(final InstallArgs args, final int currentStatus) {
2      // Queue up an async operation since the package installation may take a little while.
3      mHandler.post(new Runnable() {
4          public void run() {
5              PackageInstalledInfo res = new PackageInstalledInfo();
6              if (res.returnCode == PackageManager.INSTALL_SUCCEEDED) {
7                  args.doPreInstall(res.returnCode); 1
8                  synchronized (mInstallLock) {
9                      installPackageLI(args, res); 2
10                 }
11             }
12         });
13     });
14 }
```

## 安装核心

```

1  private void installPackageLI(InstallArgs args, PackageInstalledInfo res) {
2      ...
3      1 final PackageParser.Package pkg;
4      pkg = pp.parsePackage(tmpPackageFile, parseFlags);
5      ...
6      2 pp.collectCertificates(pkg, parseFlags);
7      pp.collectManifestDigest(pkg);
8      ...
9      int result = mPackageDexOptimizer
10         .performDexOpt(pkg, null /* instruction sets */, false /* forceDex */,
11                         false /* defer */, false /* inclDependencies */,
12                         true /* boot complete */);
13     ...
14     4 installNewPackageLI(pkg, parseFlags, scanFlags | SCAN_DELETE_DATA_ON_FAILURES,
15                           args.user, installerPackageName, volumeUuid, res);
16   }
17 }
```

```

1 private void installNewPackageLI(PackageParser.Package pkg, int parseFlags, int scanFlags,
2         UserHandle user, String installerPackageName, String volumeUuid,
3         PackageInstalledInfo res) {
4
5     PackageParser.Package newPackage = scanPackageLI(pkg, parseFlags, scanFlags,
6             System.currentTimeMillis(), user);
7
8     updateSettingsLI(newPackage, installerPackageName, volumeUuid, null, null, res, user);
9     // delete the partially installed application. the data directory will have to be
10    // restored if it was already existing
11
12    if (res.returnCode != PackageManager.INSTALL_SUCCEEDED) {
13        // remove package from internal structures. Note that we want deletePackageX to
14        // delete the package data and cache directories that it created in
15        // scanPackageLocked, unless those directories existed before we even tried to
16        // install.
17        deletePackageLI(pkgName, UserHandle.ALL, false, null, null,
18                         dataDirExists ? PackageManager.DELETE_KEEP_DATA : 0,
19                         res.removedInfo, true);
20    }
21 }

```

安装成功之后，还会发送一个App安装成功的广播ACTION\_PACKAGE\_ADDED手机桌面应用注册了这个广播，当接收到应用安装成功之后就将apk的启动icon显示在桌面上

## 23、Handler

主要场景是子线程完成耗时操作的过程中通过Handler向主线程发送消息Message，用来刷新UI界面

从new Handler()开始

```

1 // Handler.java
2
3 final Looper mLooper;
4 final MessageQueue mQueue;
5
6 public Handler() {
7     this(null, false);
8 }
9
10 public Handler(Callback callback, boolean async) {
11     if (FIND_POTENTIAL_LEAKS) {
12         final Class<? extends Handler> klass = getClass();
13         if ((klass.isAnonymousClass() || klass.isMemberClass() || klass.isLocalClass()) &&
14             (klass.getModifiers() & Modifier.STATIC) == 0) {
15             Log.w(TAG, "The following Handler class should be static or leaks might occur: " +
16                   klass.getCanonicalName());
17         }
18     }
19
20     mLooper = Looper.myLooper(); 1
21     if (mLooper == null) {
22         throw new RuntimeException(
23             "Can't create handler inside thread " + Thread.currentThread()
24             + " that has not called Looper.prepare()");
25     }
26     mQueue = mLooper.mQueue; 2
27     mCallback = callback;
28     mAsynchronous = async;
29 }

```

```
1 // Handler.java
2
3 final MessageQueue mQueue;
4
5 public static @Nullable Looper myLooper() {
6     return sThreadLocal.get();
7 }
```

## Looper介绍

Looper内部维护一个无限循环，保证App进程持续进行

```
1 // ActivityThread.java
2
3 public static void main(String[] args) {
4     ...
5
6     Looper.prepareMainLooper(); ①
7
8     ...
9
10    if (sMainThreadHandler == null) {
11        sMainThreadHandler = thread.getHandler();
12    }
13
14    Looper.loop(); ②
15
16    throw new RuntimeException("Main thread loop unexpectedly exited");
17 }
```

## Looper初始化

```
1 public static void prepareMainLooper() {
2     ① prepare(false);
3     synchronized (Looper.class) {
4         if (sMainLooper != null) {
5             throw new IllegalStateException("The main Looper has already been prepared.");
6         }
7         ③ sMainLooper = myLooper();
8     }
9 }
10
11 private static void prepare(boolean quitAllowed) {
12     ② if (sThreadLocal.get() != null) {
13         throw new RuntimeException("Only one Looper may be created per thread");
14     }
15     sThreadLocal.set(new Looper(quitAllowed));
16 }
```

1. 创建Looper对象
2. 判断是否绑定过Looper对象，将looper对象设置到本地线程，与线程绑定  
2处，确保在一个线程中Looper.prepare()方法只能被调用1次。
3. 取出Looper对象

## Looper 的构造方法

```
1 private Looper(boolean quitAllowed) {  
2     mQueue = new MessageQueue(quitAllowed);  
3     mThread = Thread.currentThread();  
4 }
```

```
1 public static @Nullable Looper myLooper() {  
2     return sThreadLocal.get();  
3 }
```

在MainActivity 所在进程被创建时，Looper的prepare方法已经在main方法中调用了1遍这会直接导致一个非常重要的结果：是

- prepare方法在一个线程中只能被调用1次；
- Looper的构造方法在一个线程中只能被调用1次
- 最终导致MessageQueue在一个线程中只会被初始化1次

### Looper负责内容

不断从MessageQueue中取出Message，然后处理Message中指定的任务

```
1 public static void loop() {  
2     final Looper me = myLooper();  
3     if (me == null) {  
4         throw new RuntimeException("No Looper; Looper.prepare() wasn't called on this thread.");  
5     }  
6     final MessageQueue queue = me.mQueue;  
7     ...  
8     for (;;) {  
9         1 Message msg = queue.next(); // might block  
10        if (msg == null) {  
11            // No message indicates that the message queue is quitting.  
12            return;  
13        }  
14        // 如果取出的Message不为null，则进行后续处理  
15        try {  
16            2 msg.target.dispatchMessage(msg);  
17            dispatchEnd = needEndTime ? SystemClock.uptimeMillis() : 0;  
18        } finally {  
19            if (traceTag != 0) {  
20                Trace.traceEnd(traceTag);  
21            }  
22        }  
23        ...  
24    }  
25 }
```

loop方法，执行一个死循环，不断的获取message

```
1 public final class Message implements Parcelable {  
2     public int what;  
3     public int arg1;  
4     public int arg2;  
5     ...  
6  
7     Handler target; // Boxed  
8  
9     ...  
10 }
```

```
1 // Handler.java  
2  
3 public void dispatchMessage(Message msg) {  
4     if (msg.callback != null) {  
5         handleCallback(msg);  
6     } else {  
7         if (mCallback != null) {  
8             if (mCallback.handleMessage(msg)) {  
9                 return;  
10            }  
11        }  
12        handleMessage(msg);  
13    }  
14 }  
15  
16 public void handleMessage(Message msg) {  
17 }
```

### Handler的sendMessage方法

```
1 public final boolean sendMessage(Message msg) {  
2     return sendMessageDelayed(msg, 0);  
3 }  
4  
5 public final boolean sendMessageDelayed(Message msg, long delayMillis) {  
6     if (delayMillis < 0) {  
7         delayMillis = 0;  
8     }  
9     return sendMessageAtTime(msg, SystemClock.uptimeMillis() + delayMillis);  
10 }  
11  
12 public boolean sendMessageAtTime(Message msg, long uptimeMillis) {  
13     MessageQueue queue = mQueue;  
14     if (queue == null) {  
15         RuntimeException e = new RuntimeException(  
16             this + " sendMessageAtTime() called with no mQueue");  
17         Log.w("Looper", e.getMessage(), e);  
18         return false;  
19     }  
20     return enqueueMessage(queue, msg, uptimeMillis);  
21 }
```

## Handler的enqueueMessage方法

```
1 // Handler.java
2 private boolean enqueueMessage(MessageQueue queue, Message msg, long uptimeMillis) {
3     msg.target = this; ①
4     if (mAsynchronous) {
5         msg.setAsynchronous(true);
6     }
7     return queue.enqueueMessage(msg, uptimeMillis);
8 }
9
10 // MessageQueue.java
11 boolean enqueueMessage(Message msg, long when) {
12     if (msg.target == null) {
13         throw new IllegalArgumentException("Message must have a target."); ②
14     }
15     synchronized (this) {
16         msg.markInUse();
17         msg.when = when;
18         Message p = mMessages;
19         ...
20         if (p == null || when == 0 || when < p.when) {
21             msg.next = p;
22             mMessages = msg;
23             needWake = mBlocked;
24         } else {
25             Message prev;
26             for (;;) {
27                 prev = p;
28                 p = p.next;
29                 if (p == null || when < p.when) {
30                     break;
31                 }
32             }
33             msg.next = p; // invariant: p == prev.next
34             prev.next = msg;
35         }
36     }
37     return true;
38 }
```

## Handler的post(Runnable)与sendMessage有什么区别

```
1 public final boolean post(Runnable r) {
2     return sendMessageDelayed(getPostMessage(r), 0);
3 }
4
5 private static Message getPostMessage(Runnable r) {
6     Message m = Message.obtain();
7     m.callback = r;
8     return m;
9 }
```

Looper从MessageQueue中取出message之后，会调用dispatchMessage方法进行处理

```

1  public void dispatchMessage(Message msg) {
2      if (msg.callback != null) {
3          handleCallback(msg);
4      } else {
5          if (mCallback != null) {
6              if (mCallback.handleMessage(msg)) {
7                  return;
8              }
9          }
10         handleMessage(msg);
11     }
12 }
13
14 private static void handleCallback(Message message) {
15     message.callback.run();
16 }
```



如果msg.callback为空，则为sendMessage处理

### Looper.loop () 方法为什么不回阻塞主线程

Looper中的loop方法实际上是一个死循环，但是我们的ui线程却并没有被阻塞，反而可以进行各种手势操作？

```

1  Message next() {
2      int pendingIdleHandlerCount = -1; // -1 only during first iteration
3      int nextPollTimeoutMillis = 0;
4      for (;;) {
5          if (nextPollTimeoutMillis != 0) {
6              Binder.flushPendingCommands();
7          }
8          nativePollOnce(ptr, nextPollTimeoutMillis);
9      }
10     ...
11 }
12
13 private native void nativePollOnce(long ptr, int timeoutMillis);
```



nativePollOnce 方法是一个native方法，当调用此native 方法时主线程会释放CPU资源进入休眠状态，直到下条消息到达或者有事务发生通过往 pipe 管道写端写入数据来唤醒主线程工作。

### Handler的sendMessageDelayed或者postDelayed是如何实现的

在向MessageQueue队列中插入Message时，会根据Message的执行时间顺序，而消息的延迟处理的核心是在获取Message阶段。

```

1  Message next() {
2      nativePollOnce(ptr, nextPollTimeoutMillis);
3      synchronized (this) {
4          if (msg != null) {
5              if (now < msg.when) {
6                  // Next message is not ready. Set a timeout to wake up when it is ready.
7                  nextPollTimeoutMillis = (int) Math.min(msg.when - now, Integer.MAX_VALUE);
8              } else {
9                  // Got a message.
10                 mBlocked = false;
11                 if (prevMsg != null) {
12                     prevMsg.next = msg.next;
13                 } else {
14                     mMessages = msg.next;
15                 }
16                 msg.next = null;
17                 msg.markInUse();
18                 return msg;
19             }
20         } else {
21             // No more messages.
22             nextPollTimeoutMillis = -1;
23         }
24     }

```

如果当前系统时间大于等于message.when，那么会返回Message给Looper.loop()，但是这个逻辑只能保证在when之前消息不被处理，不能够保证一定在when时被处理

## 总结

- 应用启动是从ActivityThread的main开始的先是执行了Looper.prepare ()，该方法先是new了一个Looper对象在私有的构造方法中又创建了MessageQueue作为此Looper对象的成员变量  
Looper 对象通过ThreadLocal绑定MainThread中
- 当创建 Handler子类对象时，在构造方法中通过ThreadLocal 获取绑定的Looper对象并获取此 Looper对象的成员变量MessageQueue作为该Handler对象的成员变量
- 在子线程中调用上一步创建的Handler子类对象的sendMesage (msg) 方法时  
在该方法中将msg的target属性设置为自己本身  
同时调用成员变量MessageQueue对象的enqueueMessag () 方法将msg放入MessageQueue中
- 主线程创建好之后，会执行Looper.loop () 方法，该方法中获取与线程绑定的Looper对象继而  
获取该Looper对象的成员变量MessageQueue对象并开启一个会阻塞（不占用资源）的死循环，  
只要MessageQueue中有msg，就会获取该msg并执行msg.target.dispatchMessage (msg) 方法  
(msg.target即上一步引用的handler对象) 此方法中调用了第二步创建 handler 子类对象时覆  
写的handleMessage () 方法

## 24、APK如何包体积优化

### 1. 安装包监控

#### a) Android Analyzer

Android Studio的APK Analyser是Android Studio提供的一个APK检测工具通过它可以查看一个APK文件内部各项内容所占的大小，并且按照大小排序显示可以很容易观察到APK中哪一部分内容占用了最大空间

实际上APK Analyzer的作用不光是查看APK大小，也能用来分析APK因此可以使用它来分析一些优秀APK的目录结构、代码规范甚至是使用了哪些动态库技术等

#### b) Matrix中的ApkChecker

ApkChecker 是腾讯开源框架Matrix的一部分

主要是用来对Android安装包进行分析检测，并输出较为详细的检测结果报告。正常情况下需要下载Matrix源码，并单独编译matrix-apk-canary部分。如果想快速使用ApkChecker，可以在网上下载其ApkChecker.jar文件，然后创建一个配置文件.json。

```
1 {
2   "--apk": "xxx.apk",
3   "--mappingTxt": "mapping.txt",
4   "--output": "输出路径",
5   "--format": "xml,html,mm,json",
6   "--formatConfig":
7   [
8     ...
9   ],
10  "options": [
11    {
12      "name": "-duplicatedFile"
13    },
14    {
15      "name": "-unusedResources",
16      "-rTxt": "/Users/Downloads/matrix/origin/R.txt",
17      "-ignoreResources"
18      [
19        "R.raw.*",
20        "R.styleable.*",
21        "R.attr.*",
22        "R.id.*",
23        "R.string.ignore_"
24      ],
25      ...
26    },
27    {
28      "name": "-unusedAssets",
29      "-ignoreAssets": ["*.so"]
30    },
31  ]
32 }
```

配置文件有几个地方需要替换：

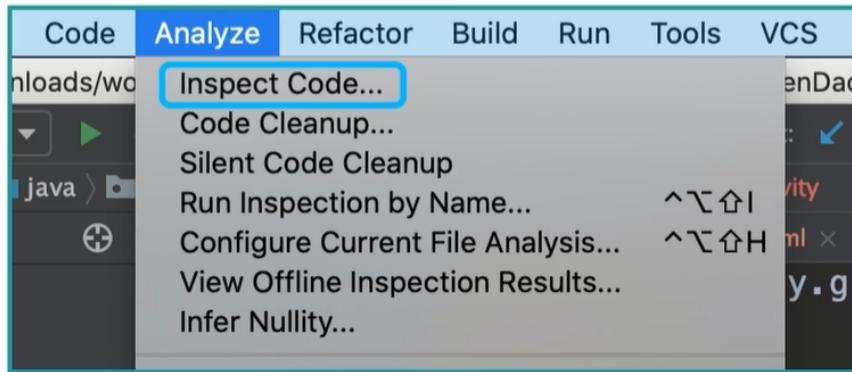
- **apk**: 需要分析的 APK 文件的路径
- **mappingTxt**: 指定混淆 mapping 文件的路径
- **output**: 分析报告的输出目录
- **rTx**: APK 文件生成时，对应的 R 文件目录

## 2. 安装包大小优化

### a) 删除无用文件

使用Lint查看未引用资源

Lint 是一个**静态扫描工具**，它可以识别出项目中没有被任何代码所引用到的资源文件



下面两个选项可以在项目编译时期减少被打包到APK中的文件：

- 使用shrinkResources能够在项目编译阶段，删除所有在项目中未被使用到的资源文件但是需要将minifyEnabled选项设置为true
- 使用resConfig 限定国际化资源文件

### b) 文件优化

- 静态图片优化
  - 有限使用drawable
  - WebP格式
- 动态图片优化

### c) 引入第三方库

- 在引入之前最好权衡一下是否需要将其代码全部引入，造成不必要的代码或者资源也被打包到APK中

## 总结：

- 安装包的监控  
主要介绍了几个可以用来分析安装包大小以及详细内容的工具：Apk Analyzer和ApkChecker在开发过程中，良好的编程习惯和严格的code review也是非常重要的
- 安装包优化实践  
主要思路是删减无用资源或者代码，并对资源文件进行相应的压缩优化对于代码部分也可以更进一步的优化，比如使用Proguard，或者直接使用R8编译方式极力推荐阅读Jake Wharton的个人博客：jakewharton中的相关介绍

### • 安装包的监控

主要介绍了几个可以用来分析安装包大小以及详细内容的工具：Apk Analyzer 和 ApkChecker

在开发过程中，良好的编程习惯和严格的 code review 也是非常重要的

### • 安装包优化实践

主要思路是删减无用资源或者代码，并对资源文件进行相应的压缩优化

对于代码部分也可以更进一步的优化，比如使用 Proguard，或者直接使用 R8 编译方式

极力推荐阅读 Jake Wharton 的个人博客：jakewharton 中的相关介绍

## 25、android 崩溃日志

crash日志分类：

- **JVM异常(Exception)**堆栈信息
  - 检测异常 checked exception
    - 检查异常是在代码编译时期，Android Studio 就会提示代码有错误，无法通过编译比如IOException。如果没有在代码中将这些异常catch，而是直接抛出，最终也有可能导致程序崩溃
  - 非检查异常 unchecked Exception
    - 非检查异常包括error 和运行时异常（RuntimeException）  
AS并不会在编译时期提示这些异常信息，而是在程序运行时期因为代码错误而直接导致程序崩溃比如OOM或者空指针异常（NPE）
- **native代码崩溃日志**
  - 当程序中的native代码发生崩溃时  
系统会在/data/tombstones/目录下保存一份详细的崩溃日志信息  
如果一个native crash是必现的，不妨在模拟器上重现bug

```
1  @FunctionalInterface
2      public interface UncaughtExceptionHandler {
3          /**
4             * Method invoked when the given thread terminates due to the
5             * given uncaught exception.
6             * <p>Any exception thrown by this method will be ignored by the
7             * Java Virtual Machine.
8             * @param t the thread
9             * @param e the exception
10            */
11      void uncaughtException(Thread t, Throwable e);
12 }
```

并将/data/tombstones中的崩溃日志拉到本地电脑中加以分析

- 需要一种机制，将native crash现场的日志信息保存到可以访问的手机目录中目前比较成熟，使用也比较广泛的是谷歌的BreakPad Breakpad 是一个跨平台的开源库，也可以在其Breakpad Github上下载自己编译并通过JNI的方式引入到项目中

## 26、内存泄漏

### Activity内存泄漏预防

因为Activity承担了与用户交互的职责，因此内部需要持有大量的资源引用以及与系统交互的Context  
这会导致一个Activity对象的retained size特别大

造成Activity内存泄漏的场景

- 将Context或View设置为static

```
1 public class ActivityB extends AppCompatActivity {  
2  
3     private static ImageView imageView;  
4  
5     @Override  
6  
7     protected void onCreate(@Nullable Bundle savedInstanceState) {  
8         super.onCreate(savedInstanceState);  
9         setContentView(R.layout.activity_b);  
10  
11         imageView = findViewById(R.id.iv);  
12         imageView.setImageResource(R.drawable.lagou);  
13     }  
14 }
```

- 未解注册各种Listener

```
1 public class ActivityC extends AppCompatActivity {  
2     BroadcastReceiver receiver = new BroadcastReceiver() {  
3         @Override  
4         public void onReceive(Context context, Intent intent) {  
5             }  
6         };  
7  
8         @Override  
9         protected void onResume() {  
10             super.onResume();  
11             IntentFilter filter = new IntentFilter();  
12             registerReceiver(receiver, filter);  
13         }  
14     }
```

- 非静态Handler导致Activity泄漏

```

1  public class ActivityD extends AppCompatActivity {
2
3      private Handler handler = new Handler(){
4          @Override
5          public void handleMessage(Message msg) {
6              super.handleMessage(msg);
7          }
8      };
9
10     @Override
11     protected void onCreate(@Nullable Bundle savedInstanceState) {
12         super.onCreate(savedInstanceState);
13         setContentView(R.layout.activity_d);
14
15         handler.postDelayed(new Runnable() {
16             @Override
17             public void run() {
18                 try {
19                     Thread.sleep(10000);    // 模拟10s的耗时任务
20                 } catch (InterruptedException e) {
21                 }
22             }
23         }, 3000);
24     }
25 }

```

正确用法：

```

1  private static class MyHandler extends Handler {
2      private final WeakReference<ActivityD> mActivity;
3
4      public MyHandler(SampleActivity activity) {
5          mActivity = new WeakReference<SampleActivity>(activity);
6      }
7
8      @Override
9      public void handleMessage(Message msg) {
10         ActivityD activity = mActivity.get();
11         if (activity != null) {
12             // ...
13         }
14     }
15 }

```

#### 4. 三方库使用Context

```

1  public class ActivityE extends AppCompatActivity {
2      @Override
3      protected void onCreate(@Nullable Bundle savedInstanceState) {
4          super.onCreate(savedInstanceState);
5          setContentView(R.layout.activity_e);
6          // Memory Leak!!
7          ThirdParty.init(this);
8      }
9  }
10
11 public class ThirdParty {
12     private static Context staticContext;
13
14     public static void init(Context context) {
15         if (staticContext == null)
16             staticContext = context;
17     }
18 }

```



正确使用

```

1  public static void init(Context var0) {
2      Logger.dd("JPushInterface", "action:init - sdkVersion:3.3.6, buildId:41");
3      checkContext(var0);
4      boolean var1 = cn.jpush.android.j.a.g(var0);
5      if (var1) {
6          cn.jpush.android.a.d(var0);
7          JCOREHelper.runActionWithService(var0, "JPUSH", "init", (Bundle)null);
8          c.a().b(var0);
9      }
10 }
11
12 private static void checkContext(Context var0) {
13     if (null == var0) {
14         throw new IllegalArgumentException("NULL context");
15     } else {
16         cn.jpush.android.a.a = var0.getApplicationContext();
17     }
18 }

```

在开发阶段可以直接使用 `Android Studio` 来查看Activity是否存在内存泄漏

并结合MAT来查看发生内存泄漏的具体对象

详细使用过程可以参考：Android Studio和MAT结合使用来分析内存问题

## 内存泄漏检测

`LeakCanary` 是Square公司的一个开源库，通过它可以在App运行过程中检测内存泄漏，当内存泄漏发生时会生成发生泄漏对象的引用链，并通知程序开发人员

LeakCanary 主要分2大核心部分：

1. 如何检测内存泄漏
2. 分析内存泄漏对象的引用链

## 如果检测内存泄漏--JVM理论知识

```

1  public class WeakRefDemo {
2      public static void main(String[] args) throws InterruptedException {
3          WeakReference<BigObject> reference = new WeakReference<>(new BigObject());
4
5          System.out.println("before gc, reference.get is " + reference.get());
6
7          System.gc();
8          Thread.sleep(1000);
9
10         System.out.println("after gc, reference.get is " + reference.get());
11     }
12
13     static class BigObject {
14     }
15 }

```

实现思路：

LeakCanary 中对内存泄漏检测的核心原理就是 基于 `WeakReference` 和 `ReferenceQueue` 实现的

1. 当一个Activity需要被回收时，就将其包装到一个WeakReference中并且在WeakReference的构造器中传入自定义的ReferenceQueue
2. 给包装后的WeakReference做一个标记Key，并且在一个强引用Set中添加相应的Key记录
3. 主动触发GC，遍历自定义ReferenceQueue中所有的记录并根据获取的Reference对象将Set中的记录也删除

还保留在Set中的是：应当被GC回收，但是实际还保留在内存中的对象，也就是发生泄漏了的对象  
源码分析

一个可回收对象在 `System.gc()` 之后就应该被GC回收在AndroidApp中，我们并不清楚何时系统会回收Activity按照正常流程，当Activity调用 `onDestroy` 方法时就说明这个Activity就已经处于无用状态因此需要 监听到每一个Activity的 `onDestroy` 方法的调用

总结：

这节课主要介绍了Android内存泄漏优化的相关知识

·内存泄漏预防

这需要了解JVM发生内存泄漏的原因，并在平时开发阶段养成良好的编码规范针对编码规范

Android Studio可以安装一个阿里代码规范的插件，能够起到一定的代码检查效果

·内存泄漏检测

内存泄漏检测工具有很多Android Studio自带的Profiler，以及MAT都是不错的选择使用这些工具排查内存泄漏门槛稍高，并且全部是手动操作，略显麻烦

## 27、UI卡顿

systrace工具：通过Android提供的脚本systrace.py，可以设置数据采集并收集相关程序运行数据，最终生成一个网页文件提供程序开发者分析程序性能问题。

在Android SDK中提供了运行Systrace的脚本

具体路径在 android-sdk/platform-tools/systrace/

```
python systrace.py --time=10 -o my_systrace.html
```

## 29、MVP中presenter生命周期管理

presenter层经常做一些耗时操作，根据请求后的结果刷新View。

如果按返回结束Activity，而Presenter依然在执行耗时操作，就有可能造成内存泄漏，甚至程序崩溃。

将Activity的某些生命周期方法与Presenter保持一致

Lifecycle绑定Presenter生命周期

Activity通过继承AppCompatActivity会自动继承来自父类ComponentActivity的方法getLifecycle

```
1 public class LoginActivity extends AppCompatActivity {
2
3     @Override
4     protected void onCreate(Bundle savedInstanceState) {
5         super.onCreate(savedInstanceState);
6         setContentView(R.layout.activity_login);
7         getLifecycle().addObserver(new LifecycleEventObserver() {
8             @Override
9             public void onStateChanged(@NonNull LifecycleOwner source,
10                                     @NonNull Lifecycle.Event event) {
11                 Log.e("TAG", "event is " + event.name());
12             }
13         });
14     }
15 }
```

```
1 public interface IPresenter extends LifecycleObserver {
2
3     @OnLifecycleEvent(Lifecycle.Event.ON_CREATE)
4     void onCreate(@NonNull LifecycleOwner owner);
5
6     @OnLifecycleEvent(Lifecycle.Event.ON_RESUME)
7     void onResume(@NonNull LifecycleOwner owner);
8
9     @OnLifecycleEvent(Lifecycle.Event.ON_PAUSE)
10    void onPause(@NonNull LifecycleOwner owner);
11
12    @OnLifecycleEvent(Lifecycle.Event.ON_STOP)
13    void onStop(@NonNull LifecycleOwner owner);
14
15    @OnLifecycleEvent(Lifecycle.Event.ON_DESTROY)
16    void onDestroy(@NonNull LifecycleOwner owner);
17 }
```

需要在自定义类中实现这些方法

```
1 public class BasePresenter implements IPresenter {
2     private static final String TAG = BasePresenter.class.getSimpleName();
3
4     @Override
5     public void onCreate(@NonNull LifecycleOwner owner) {
6         Log.e(TAG, "BasePresenter--onCreate");
7     }
8
9     @Override
10    public void onResume(@NonNull LifecycleOwner owner) {
11        Log.e(TAG, "BasePresenter--onResume");
12    }
13
14    @Override
15    public void onPause(@NonNull LifecycleOwner owner) {
16        Log.e(TAG, "BasePresenter--onPause");
17    }
18
19    @Override
20    public void onStop(@NonNull LifecycleOwner owner) {
21        Log.e(TAG, "BasePresenter--onStop");
22    }
23
24    @Override
25    public void onDestroy(@NonNull LifecycleOwner owner) {
26        Log.e(TAG, "BasePresenter--onDestroy");
27    }
28 }
```

修改Activity，将BasePresenter注册到LifeCycle中

```
1 public class LoginActivity extends AppCompatActivity {
2
3     private BasePresenter presenter;
4
5     @Override
6     protected void onCreate(Bundle savedInstanceState) {
7         super.onCreate(savedInstanceState);
8         setContentView(R.layout.activity_login);
9
10        presenter = new BasePresenter();
11
12        getLifecycle().addObserver(presenter);
13    }
14 }
```

## 30、LogUtil

Xlog

**XLog** 是比较常用的打印日志开源库，GitHub 地址参考

[https://github.com/elvishew/XLog/blob/master/README\\_ZH.md](https://github.com/elvishew/XLog/blob/master/README_ZH.md)

XLog 基本囊括了上文介绍的所有功能：

- XML 和 JSON 格式化输出
- 线程信息（线程名等，可自定义）
- 调用栈信息（可配置的调用栈深度，调用栈信息包括类名、方法名、文件名和行号）
- 支持日志拦截器
- 保存日志文件（文件名和自动备份策略可灵活配置）

先调用 init 方法进行初始化，最好是在 Application 中

```
1 public class MyApp extends Application {
2
3     @Override
4     public void onCreate() {
5         super.onCreate();
6
7         LogConfiguration config = new LogConfiguration.Builder()
8             .tag("MY_TAG")           // 指定 TAG, 默认为 "X-LOG"
9             .t()                    // 允许打印线程信息, 默认禁止
10            .st(2)                 // 允许打印深度为2的调用栈信息, 默认禁止
11            .b()                   // 允许打印日志边框, 默认禁止
12            .build();
13         XLog.init(config);
14     }
15 }
```

## 31、屏幕适配

---

1. ConstrainLayout(约束布局) (前身是percentLayout百分百布局) +dimens
2. 控件适配：TextView使用wrap\_content(自适应), ImageView使用固定大小dp
3. 手机和平板或折叠屏，布局是否需要拆分，控件布局方式或大小