|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 分类号 |  | |  | | | 密 级 | |  |
| U D C |  | |  | | | 编 号 | | 0002 |
|  | | | | | | | | |
| **读 书 笔 记** | | | | | | | | |
|  | | | | | | | | |
| Android开发艺术探索 | | | | | | | | |
|  | |  | |  |  | |  | |
|  | |  |  | |
|  | |  |  | |
|  | |  |  | |
|  | |  |  | |
|  | |  |  | |
|  | | | | | | | | |
| By key.guan  2016.9~ 2017.6 | | | | | | | | |

# Activity的生命周期和启动模式

## 1 Activity的生命周期全面分析

正常周期（用户参与）和异常周期（系统回收、Configuration改变）

**正常周期**



总结一下各种情况下Activity生命周期的调用情况。

**1，正常打开单个Activity，然后退出应用：**

这种情况是最普通的状况，Activity的生命周期会按照上图从上到下的方式走。即：onCreate --> onStart --> onResume --> 运行--> 按返回键结束程序--> onPause-->onStop-->onDestory

**2，打开一个Activity A，然后再打开另一个Activity B**

对于A：

onCreate --> onStart --> onResume --> A运行 --> A发出打开B的Intent --> onPause-->B可见-->onStop

此时，会打开B，B同样会经历一个完整的Activity生命周期。

等B结束，A再度可见的时候，A会经历：onRestart-->onStart-->onResume

注意，B这个Activity是在A的onPause执行后才变成可见状态的，所以为了不影响B的显示，最好不要在onPause里执行一些耗时操作，可以考虑将这些操作放到onStop里，这时B已经可见了。

**3，打开一个Activity A，然后再打开另一个Activity B（透明主题）或者是对话框B**

对于A：

onCreate --> onStart --> onResume --> A运行 --> A发出打开B的Intent --> onPause-->B可见--~~>onStop~~

此时，会打开B，B同样会经历一个完整的Activity生命周期。

等B结束，A再度可见的时候，A会经历：~~onRestart-->onStart-->~~onResume

**3，优先级低的Activity被系统回收**

Activity优先级从高到低大致有三种：

1）刚执行onResume，前台Activity。可见的前台Activity优先级最高

2）刚执行onPause，可见，但非前台的Activity。例如弹出了一个对话框，Activity可见，但是不可与用户交互。此类别优先级低一些。

3）刚执行onStop，后台Activity。这个就是不可见的Activity。

系统资源不足时，会优先回收优先级低的Activity。这个时候被结束的Activity，并不会像正常的Activity一样，在销毁的时候会走onPause，onStop，onDestory这样的生命周期，而是会多一个onSaveInstanceState的方法来保存一些数据。这个方法会在onStop前调用，但是不确定是在onPause之前还是之后。Activity重建的时候，这个方法中所保存的数据会以Bundle的形式作为参数传递给onCreate，所以我们可以用参数来判断这个Activity是新建的还是重建的，如果是重建的，则可以从Bundle中取出数据来恢复界面。

**QA：**

**Q1**: onStart和onResume, onPause和onStop从描述上来看差不多，对我们来说有什么实质的不同呢?

**A1：**从整个生命周期来说，onCreate和onDestroy是配对的，分别标识着Activity的创建和销毁，并且只可能有一次调用。从Activity是否可见来说，onStart和onStop是配对的（**可见性状态**），随着用户的操作或者设备屏幕的点亮和熄灭，这两个方法可能被调用多次;从Activity是否在前台来说，onResume和onPause是配对的（**前后台状态**），随着用户操作或者设备屏幕的点亮和熄灭，这两个方法可能被调用多次。

从实际使用过程来说，onStart和anResume. onPause和onStop看起来的确差不多，甚至我们可以只保留其中一对，比如只保留onStart和onStop 既然如此，那Android系统还要提供看起来重复的接口呢?这两个配对的回调分别表示不同的意义，onStart和onStop是从Activity是否可见这个角度来回调的，而onResume和onPause。是从Activity是否位于前台这个角度来回调的，除了这种区别，在实际使用中没有其他明显区别。

**Q2:**假设当前Activity为A，如果这时用户打开一个新Activity B，那么B的onResume和A的onPause哪个先执行呢?

**A2：**onPause先执行-》onPause和onStop都不能执行耗时的操作，尤其是onPause，应当尽量在onStop中做操作，从而使得新Activity尽快显示出来并切换到前台。

**A3：**了解这些生命周期

**Q3：**自定义与视图有关的工具的时候，也应该有这些生命周期，比如动画工具

**异常周期**

**系统配置**

在默认情况下，如果我们的Activity不做特殊处理，那么当系统配置发生改变后，Activity就会被销毁并重新创建，其生命周期如图1-3所示。系统只在Activity异常终止的时候才会调用onSaveInstanceState和onRestoreInstanceState来存储和恢复数据，其他情况不会触发这个过程。



当系统配置发生改变后，系统会调用onSaveInstanceState来保存当前Activity的状态。这个方法的调用时机是在onStop之前，它和onPause没有既定的时序关系，可前可后。这个方法只会出现在Activity被异常终止的情况下，正常情况下系统不会回调。当Activity被重新创建后，系统会调用onRestorelnstanceSiate，从时序来说，onRestoreInstanceState的调用时机在onStart之后。

Activity被销毁并重新创建后，我们再去获取之前存储的字符串。接收的位置可以选择onRestorelnstanceState或者onCreate，二者的区别onRestorelnstanceState参数非空，但是onCreate可空，onCreate如果是正常启动的话，其参数Bundle saVedlnstanceState为null,

所以必须要额外判断。这两个方法我们选择任意一个都可以进行数据恢复，但官方建议是采用onRestoreInstanceState去恢复数据

系统配置监听，取代onSaveInstanceState和onRestorelnstanceState

android:configChanges="orientation|screenSize"

@Override

public void onConfigurationChanged(Configuration newConfig) {

super.onConfigurationChanged(newConfig);

Log.d(TAG, "onConfigurationChanged, newOrientation:" + newConfig.orientation);

}

情况2:**资源内存**不足导致低优先级的Activity被杀死

这种情况我们不好模拟，但是其数据存储和恢复过程和情况1完全一致。这里我们描述一下Activity的优先级情况口Activity按照优先级从高到低，可以分为如下3种:

(1)前台Activity—正在和用户交互的Activity，优先级最高。

(2)可见但非前台Activity—比如Activity中弹出了一个对话框，导致Activity可见，但是位于后台无法和用户直接交互。

（3)后台Activity—已经被暂停的Activity，比如执行了onStop，优先级最低。

**小结**：如果一个进程中没有四大组件在执行，那么这个进程将很快被系统杀死，因此，一此后台工作不适合脱离四大组件而独自运行在后台中，这样进程很容易被杀死。比较好的方法是将后台工作放入Service中从而保证进程有一定的优先级，这样就不会轻易地被系统杀死。

## 2.Activity的LaunchMode

**引言：**

* 为什么需要启动模式（创建Activity实例的方式）：由于android存在**taskStack特性**和java的**class单例特性**，从而产生了不同的启动模式
* why we need taskStack：因为是android 只能运行单窗口app，startActivity(intent)入栈push，finish()pop
* taskStack（1）<->Activity Insurance（N）

a. standard 。标准模式，1栈N例，系统默认，每次启动会重新创建新的实例，谁启动了这个Activity，这个Activity就在谁的栈里。

正确使用场景：适用于绝大大数情况，ABCDEFC；

错误使用场景，用ApplicationContext（自身没有taskStack），去启动Standard模式的Activity会报错

b. singleTop 单例栈顶模式，1栈top1例。该Activity的onNewIntent方法会被回调，onCreate和onStart并不会被调用。

正确使用场景：A例在top模式下生效，答题；ABC

错误使用场景：B例在非top下回退为standard 模式，ABCB

c. singleTask 单例栈内模式栈内复用模式，1栈1例。只要该Activity在一个栈中存在，都不会重新创建，onNewIntent会被回调。如果不存在，系统会先寻找是否存在需要的栈，如果不存在该栈，就创建一个任务栈，然后把这个Activity放进去；如果存在，就会创建到已经存在的这个栈中。**FLAG\_ACTIVITY\_CLEAR\_TOP，**[**http://www.2cto.com/kf/201410/347854.html**](http://www.2cto.com/kf/201410/347854.html)

|  |
| --- |
| android:taskAffinity="com.other" |

Manifest 配置文件这个属性的意思是 “task 空间”=== “任务空间”，系统默认给设置为当前应用的包名。我们这里设置 的和包名不一样就会新建一个taskStack

<http://www.cnblogs.com/CSU-PL/p/3794280.html>



正确使用场景：1.根据affinity重新为Activity选择宿主task（与allowTaskReparenting属性配合工作）；2.启动一个Activity过程中Intent使用了FLAG\_ACTIVITY\_NEW\_TASK标记，根据affinity查找或创建一个新的具有对应affinity的task。我们会在后面进行详细讲解。览器activity， 整个任务栈只有一个实例，节约内存和cpu的目的

错误使用场景：B例在非top下回退为standard 模式，ABCB

d. singleInstance。单例模式具有此种模式的Activity只能单独存在于一个任务栈。

 应用场景：呼叫来电界面 InCallScreen

**Q&A:什么是Activity所需要的任务栈呢:由TaskAffinity确定，默认为应用包名。TaskAffinity属性主要和singleTask启动模式或者allowTaskReparenting属性配对使用，在其他情况下没有意义。另外，任务栈分为前台任务栈和后台任务栈，后台任务栈中的Activity位于暂停状态，用户可以通过切换将后台任务栈再次调到前台。**

当TaskAffinity和singleTask启动模式配对使用法，它是具有该模式的待启动的Activity会运行在任务栈TaskAffinity中。

当TaskAffinity和**allowTaskReparenting配对使用，**这个属性用来标记一个Activity实例在当前应用退居后台后，是否能从启动它的那个task移动到有共同affinity的task，“true”表示可以移动，“false”表示它必须呆在当前应用的task中，默认值为false。例如在一个应用中要查看一个web页面，在启动系统浏览器Activity后，这个Activity实例和当前应用处于同一个task，当我们的应用退居后台之后用户再次从主选单中启动应用，此时这个Activity实例将会重新宿主到Browser应用的task内，在我们的应用中将不会再看到这个Activity实例，而如果此时启动Browser应用，就会发现，第一个界面就是我们刚才打开的web页面，证明了这个Activity实例确实是宿主到了Browser应用的task内。我们就来结合实例演示一下这个过程：

<http://www.cnblogs.com/CSU-PL/p/3794280.html>

**“拿来主义”standard模式。哪里需要调用我我就去哪里，可以多次实例化，可以几个相同的Activity重叠。一般默认模式**

**“拒绝堆叠”singleTop模式。可以多次实例化，但是不可以多个相同的Activity重叠，当堆栈的顶部为相同的Activity时，会调用onNewIntent函数。可以用于图片浏览**

**“独立门户”singleTask模式。同一个应用中调用该Activity时，如果该Activity没有被实例化，会在本应用程序的Task内实例 化，如果已经实例化，会将Task中其上的Activity销毁后，调用onNewIntent；其它应用程序调用该Activity时，如果该 Activity没有被实例化，会创建新的Task并实例化后入栈，如果已经实例化，会销毁其上的Activity，并调用onNewIntent。一句 话，singleTask就是“独立门户”，在自己的Task里，并且启动时不允许其他Activity凌驾于自己之上。可以用浏览器-**微信

**“孤独寂寞”singleInstance模式。加载该Activity时如果没有实例化，他会创建新的Task后，实例化入栈，如果已经存在，直接调用 onNewIntent，该Activity的Task中不允许启动其它的Activity，任何从该Activity启动的其他Activity都将被 放到其他task中，先检查是否有本应用的task，没有的话就创建。电话拨打界面**

**2.2 activity 的Flags**

**FLAG ACTIVITY NEW TASK**

这个标记位的作用是为Activity指定“singleTask”启动模式，其效果和在XML中指定该启动模式相同。

**FLAG\_ACTIVITY\_CLEAR\_TOP**

    如果设置，并且这个Activity已经在当前的Task中运行，因此，不再是重新启动一个这个Activity的实例，而是在这个Activity上方 的所有Activity都将关闭，然后这个Intent会作为一个新的Intent投递到老的Activity（现在位于顶端）中。

    例如，假设一个Task中包含这些Activity：A，B，C，D。如果D调用了startActivity()，并且包含一个指向Activity B的Intent，那么，C和D都将结束，然后B接收到这个Intent，因此，目前stack的状况是：A，B。

    上例中正在运行的Activity B既可以在onNewIntent()中接收到这个新的Intent，也可以把自己关闭然后重新启动来接收这个Intent。如果它的启动模式声明为 “multiple”(默认值)，并且你没有在这个Intent中设置FLAG\_ACTIVITY\_SINGLE\_TOP标志，那么它将关闭然后重新创 建；对于其它的启动模式，或者在这个Intent中设置FLAG\_ACTIVITY\_SINGLE\_TOP标志，都将把这个Intent投递到当前这个实 例的onNewIntent()中。

    这个启动模式还可以与FLAG\_ACTIVITY\_NEW\_TASK结合起来使用：用于启动一个Task中的根Activity，它会把那个Task中任 何运行的实例带入前台，然后清除它直到根Activity。这非常有用，例如，当从Notification Manager处启动一个Activity。

FLAG ACTIVITY SINGLE TOP

这个标记位的作用是为Activity指定"singleTop”启动模式，其效果和在XML中指定该启动模式相同。

**FLAG\_ACTIVITY\_EXCLUDE\_FROM\_RECENTS**

    如果设置，新的Activity不会在最近启动的Activity的列表中保存。

**3 IntentFilter的匹配规则**

隐式 Intent

Intent intent = new Intent(); intent.setAction("com.wooyun.test"); startActivity(intent);

并不知道也不关心接收者是谁，有利于降低发送者和接收者之间的耦合，它一般用在没有明确指出目标组件名称的前提下，一般是用于在不同应用程序之间，如下：

* a. action匹配规则：要求intent中的action 存在 且 必须和过滤规则中的其中一个相同 区分大小写；
* b. category匹配规则：系统会默认加上一个android.intent.category.DEAFAULT，所以intent中可以不存在category，但如果存在就必须匹配其中一个；
* c. data匹配规则：data由两部分组成，mimeType和URI，要求和action相似。如果没有指定URI，URI但默认值为content和file（schema）

使用案例：

（1）如果我们想要匹配 http 以 “.pdf” 结尾的路径，使得别的程序想要打开网络 pdf 时，用户能够可以选择我们的程序进行下载查看。

我们可以将 scheme 设置为 “http”，pathPattern 设置为 “.\*//.pdf”，整个 intent-filter 设置为：

<intent-filter> <action android:name="android.intent.action.VIEW"></action> <category android:name="android.intent.category.DEFAULT"></category> <data android:scheme="http" android:pathPattern=".\*//.pdf"></data> </intent-filter>

如果你只想处理某个站点的 pdf，那么在 data 标签里增加 android:host=”yoursite.com” 则只会匹配<http://yoursite.com/xxx/xxx.pdf>，但这不会匹配 www.yoursite.com，如果你也想匹配这个站点的话，你就需要再添加一个 data 标签，除了 android:host 改为 “www.yoursite.com” 其他都一样。

（2）如果我们做的是一个IM应用，或是其他类似于微博之类的应用，如何让别人通过 Intent 进行调用出现在选择框里呢？我们只用注册 android.intent.action.SEND 与 mimeType 为 “text/plain” 或 “*/*” 就可以了，整个 intent-filter 设置为：

<intent-filter> <action android:name="android.intent.action.SEND" /> <category android:name="android.intent.category.DEFAULT" /> <data mimeType="\*/\*" /> </intent-filter>

这里设置 category 的原因是，创建的 Intent 的实例默认 category 就包含了 Intent.CATEGORY\_DEFAULT ，google 这样做的原因是为了让这个 Intent 始终有一个 category。



（3）如果我们做的是一个音乐播放软件，当文件浏览器打开某音乐文件的时候，使我们的应用能够出现在选择框里？这类似于文件关联了，其实做起来跟上面一样，也很简单，我们只用注册 android.intent.action.VIEW 与 mimeType 为 “audio/\*” 就可以了，整个 intent-filter 设置为：

<intent-filter> <action android:name="android.intent.action.VIEW" /> <category android:name="android.intent.category.DEFAULT" /> <data android:mimeType="audio/\*" /> </intent-filter>



IntentFilter匹配规则。

# IPC机制

IPC是 Inter-Process Communication的缩写，含义为进程间通信或者跨进程通信，是指两个进程之间进行数据交换的过程。进程间通信的方式有：Bundle、文件共享、AIDL、Messenger、ContentProvider和Socket等，最简单的情况下，一个进程中只可以有一个线程，即主线程，也叫UI线程。IPC不是Android中所独有的，任何一个操作系统都需要相应的IPC机制，比如Windows上可以通过剪贴板等来进行进程间通信。Android是一种基于Linux内核的移动操作系统，它的进程间通信方式并不能完全继承自Linux，相反，它有自己的进程间通信方式。在Android中最有特色的进程间通信方式就是Binder了。

why we need 多进程，可能有很多，比如有些模块由于特殊原因需要运行在单独的进程中，或者为了加 大一个应用可使用的内存所以需要通过多进程来获取多份内存空间。Android对单个应用所使用的最大内存做了16M限制。另一种情况是当前应用需要向其他应用获取数据，由于是两个应用，所以必须采用跨进程的方式来获取所需的数据，甚至我们通过系统提供的ContentPra}ider去查询数据的时候，建议DJI的midware这么搞？

--------------------------------------------------------------------------------

开启多进程模式

在Android中使用多进程只有一种方法：给四大组件(Activity、Service、Receiver、ContentProvider)在AndroidManifest中指定android:process属性，除此之外没有其他办法(除去通过JNI在native层去fork):

● android:process=”:remote”，私有进程，这种方式的标记的进程名为包名:remote，“:“的含义是指要在当前的进程名前附加包名，其次，进程名以“：”开头的进程属于当前应用的私有进程，其他应用的组件不可以和它跑在同一个进程中。

● android:process=”com.example.wpp.remote”，全局进程，这种声明方式是完整的命名方式，不会附件包名信息，属于全局进程，其他应用通过ShareUID方式可以和它跑在同一个进程中

Android系统会为每个应用分配一个唯一的UID，具有相同UID的应用才能共享数据，而两个应用通过ShareUID跑在同一个进程需要这两个应用有相同的ShareUID并且签名相同才可以，在这种情况下，它们可以互相 访问对方的私有数据，比如data目录、组件信息等，不管它们是否泡在同一个进程中。当然如果它们跑在同一个进程中，那么除了能共享data目录、组件信息，还可以共享内存数据

使用多进程会造成如下几方面的问题：

1. 静态成员和单例模式完全失效 （每个进程都分配一个独立的虚拟机，有着不同的虚拟机空间）

2. 线程同步机制完全失效。（不管是锁对象还是锁全局类都无法保证线程同步，因为不同进程锁的对象也不同了）

3. SharedPreferences的可靠性下降

4. Application会多次创建。

Q:多进程如果共享数据？

--------------------------------------------------------------------------------

3 IPC基础概念介绍

基本概念：Serializable接口、Parcelable接口以及Binder

Serializable接口

Parcelable接口

用来保存到本地的类特别好，取代了shareprefer

SerialiaabIe是Java中的序列化, 其使用起来简单但是开销很大，序列化到存储设备中或者将对象序列化后通过网络传输

Parcelable是Android中的序列化方式，使用起来稍微麻烦点，但是它的效率很高，主要用在内存序列化

Parcelable接口是Android提供的新的序列化方式，Parcelable也是一个接口，下面的示例是一个典型的用法。

public class User implements Parcelable{

public int userId;

public String userName;

public boolean isMale;

public Book book;

public User(int userId, String userName, boolean isMale){

this.userId = userId;

this.userName = userName;

this.isMale = isMale;

}

public int describeContents(){

return 0;

}

public void writeToParcel(Parcel out, int flags){

out.writeInt(userId);

out.wirteString(userName);

out.wirteInt(isMale ? 1 : 0);

out.writeParcelable(book, 0);

}

public static final Parcelable.Creator<User> CREATOR = new Parcelable.Creator<User>(){

public User createFromParcel(Parcel in){

return new User(in);

}

public User[] new Array(int size){

return new User[size];

}

};

private User(Parcel in){

userId = in.readInt();

userName = in.readString();

isMale = in.readInt()==1;

book = in.readParcelable(Thread.currentThread().getContextClassLoader());

}

}

Parcelable方法说明



方法功能标记位createFromParcel(Parcel in)从序列化后的对象中创建原对象new Array(int size)创建指定长度的原始对象数组User(Parcel in)从序列化后的对象中创建原始对象writeToParcel(Parcel out, int flag)将当前对象写入序列化结构中，其中flag标识有两种值：0或者1.为1时标识当前对象需要作为返回值返回，不能立即释放资源，几乎所有情况都为0PARCELABLE\_WRITE\_RETURN\_VALUEdescribeContents返回当前对象的内容描述，如果还有文件描述符，返回1，否则返回0，几乎所有情况都返回0

系统已经为我们提供了许多实现了Parcelable接口的类，它们都是可以直接序列化的，比如Intent、Bundle、Bitmap等，同时List和Map也可以序列化，前提是它们里面每个元素都是可序列化的。

Binder

选择AIDL的使用场合

官方文档特别提醒我们何时使用AIDL是必要的：只有你允许客户端从不同的应用程序为了进程间的通信而去访问你的service，以及想在你的service处理多线程。

2.4.2使用文件共享

对数据同步要求不高的进程之间进行遁信，基本走java的路线了

oos = new ObjectOutputStream( new FileOutputStream(file));

oos.writeObject(user);

注:SharePreferences是一个轻量级的存储方案，但是系统对这个的读写存在一定的缓存策略，在多进程高并发的情况下，丢失数据的概率很高，因此进程之间不推荐用这种方式通信(之前和zhongping就是一次失败的情况)

2.4.3使用Messenger

服务端

messenger = new Messenger(new MyHandler());

客户端

messenger = new Messenger(iBinder);

m.replyTo = mGetReplyMessenger;

2.4.6 使用Socket

# View的事件体系

## 基础知识

View是Android中所有控件的基类。

### 位置参数

单位: pixels(像素)，坐标均是相对于父容器的坐标！

基本参数：top、left、right、bottom.左上角为（0，0）

高级参数: x,y,translationX() ,translationY(),mMinWidth、mMinHeight

换算公式：

getX()=mLeft + getTranslationX();

getWidth() = mRight - mLeft;

getSuggestedMinimumWidth()=max(mMinWidth, mBackground.getMinimumWidth ());

onMeasure(int widthMeasureSpec, int heightMeasureSpec) ->{

setMeasuredDimension(getDefaultSize(getSuggestedMinimumWidth(), widthMeasureSpec),…);}

注意:View平移时，top/left不变; x,y,translationX() ,translationY()会变

QA: 1，为什么我有时候在使用getLeft(), getRight(), getTop(), getBottom()它们得到的结果是0？

A：刚启动程序，由于view 是刚开始绘制的，就会是 0. new 一个线程，使用Thread.Sleep(long int) 函数延时一段时间后再 使用get

### MotionEvent和TouehSlop

#### MotionEvent

手指解除屏幕后所产生的一系列事件中，典型的事件类型有如下几种：

1. ACTION\_DOWN-手指刚接触屏幕
2. ACTION\_MOVE-手指在屏幕上移动
3. ACTION-UP-手指从屏幕上松开的一瞬间

正常情况下，一次手指触摸屏幕的行为会触发一些列点击事件，考虑有如下几种:

1.点击屏幕后松开:事件序列为DOWN->UP

2.点击屏幕滑动一会再松开，事件序列DOWN->MOVE->..->UP

方法:getX/getRawX:相对于当前View的坐标，相对于手机的坐标getRawY包含了状态栏的高度~

#### TouchSlop

TouchSlop是系统所能识别出的被认为是滑动的最小距离，小于这个系统就不认执行滑动操作。和设备有关

定义:/frameworks/base/core/res/res/values/config.xml

<dimen name="config\_viewConfigurationTouchSlop">8dp</dimen>

访问方法:

ViewConfiguration.get(getApplicationContext()).getScaledTouchSlop();

### VelocityTracker, GestureDetector和Stroller

#### VelocityTracker

简介: VelocityTracker 是用来在滑动操作中计算速率的工具

一般使用方法如下:

|  |
| --- |
| dispatchTouchEvent(MotionEvent ev) {  switch (ev.getAction()) {  case MotionEvent.ACTION\_DOWN:  if (null == mVelocityTracker) {//1.初始化单例  mVelocityTracker = VelocityTracker.obtain();  } else {  mVelocityTracker.clear();  }  mVelocityTracker.addMovement(ev);  break;  case MotionEvent.ACTION\_MOVE://2.添加记录  mVelocityTracker.addMovement(ev);  break;  case MotionEvent.ACTION\_UP://3.获取v，释放单例  mVelocityTracker.computeCurrentVelocity(5000); //5s  int xV = (int)mVelocityTracker.getXVelocity();  int yV = (int)mVelocityTracker.getYVelocity();  KLogUtil.D(TAG, "dispatchTouchEvent", ": xV="+ xV+",yV="+ yV );  mVelocityTracker.clear();  break; |

简单流程整理

当 action\_down 时初始化单例，然后在之后的touch操作时通过addMovement来添加记录，当 action\_up 是计算速率并根据业务需要获取x y方向的速率的值，最后调用 recycle释放。

#### GestureDetector

手势检测，用于辅助检测用户的点击、滑动、长按、双击行为，方法介绍：





|  |
| --- |
| private void testGestureDetector() {  final GestureDetector mGestureDetector = new GestureDetector(this, new MyListener ());  //解决长按无法拖动的现象  mGestureDetector.setIsLongpressEnabled(false);  mGestureTv.setOnTouchListener(new View.OnTouchListener() {  @Override  public boolean onTouch(View v, MotionEvent event) {  //接管View的消费  boolean consume = mGestureDetector.onTouchEvent(event);  return consume;  }  });  }  private class MyListener extends GestureDetector.SimpleOnGestureListener {…} |

完全可以自己在View的onTouchEvent方法中实现所有功能。建议监听滑动相关用**onTouchEvent，要监听双击用GestureDetetor**

#### Scroller

## View的滑动

实现View滑动的三种方式：

　1 通过View本身的scrollTo/scrollBy方法实现滑动；

　2 通过动画给View施加平移效果实现滑动；

　3 通过改变View的LayoutParams使得View重新布局实现滑动；

### scrollTo/scrollBy

　　其中mScrollX的值总是等于View左边缘和View内容左边缘在水平方向的距离，mScrollY的值总是等于View的上边缘和View内容上边缘在垂直方向上的距离；scrollTo/scrollBy只能改变View内容的位置而不能改变自身在布局中的位置. 也就是说使用scrollTo/scrollBy来实现View的滑动只能将View的内容进行移动，并不能将View本身进行移动。

### 使用动画

使用动画来移动View 主要是操作View的translationX和translationY属性。（传统的View动画和属性动画） View动画是对View的影像操作，它并不能真正改变View的位置参数，包括宽/高。

ObjectAnimator.ofFloat(mScrollTv,"translationX",0,200).setDuration(10000).start();

### 改变布局参数

主要是通过改变View的LayoutParams来实现；

|  |
| --- |
| ViewGroup.MarginLayoutParams params=(ViewGroup.MarginLayoutParams)mScrollTv.getLayoutParams();  params.width+=100;  params.leftMargin+=100;  mScrollTv.requestLayout();//或者mScrollTv.setLayoutParams(params)或者指定一个空的view的宽度间接调整 |

### 各种滑动方式的对比

scrollTo/scrollBy：操作简单，适合对View内容的滑动；

动画：操作简单，主要适用于没有交互的View和实现复杂的动画效果；

改变布局参数：操作稍微复杂，适用于有交互的View。

## 强性滑动（详细笔记开始）

生硬滑动用户体验太差了，因此需要渐进式滑动，思想：将一次打的滑动分成若干次小的滑动。实现方式很多，比如Scroller，Handler#PostDelayed以及Thread#Sleep,我们接下来一一介绍：

#### 使用Scroller???????????

首先分析一下他的源码，从而探索为什么能实现View的弹性滑动：

#### 通过动画

动画本身就是一种渐进的过程，因此通过他来实现滑动天然就具有弹性效果，比如以下代码让一个view在100ms内左移100像素.

ObjectAnimator.ofFloat(mScrollTv,"translationX",0,200).setDuration(10000).start();

此外，我们还可以动画的特性来实现View更新的定时器作用

|  |
| --- |
| //5s內，fraction匀速从0变化到1  final ValueAnimator animator = ValueAnimator.ofInt(0,1).setDuration(5000);  animator.addUpdateListener(new ValueAnimator.AnimatorUpdateListener() {  @Override  public void onAnimationUpdate(ValueAnimator valueAnimator) {  float fraction = animator.getAnimatedFraction();  mScrollTv.scrollTo(0 + (int)(100 \* fraction),0);  }  });  animator.start(); |

我们在动画的每一帧到来时获取动画完成的比例，然后再根据这个比例计算出当前View所要滑动的距离。注意，这里的滑动针对的是**View的内容**而非View本身。这个方法的思想其实和Scroller比较类似，都是通过改变一个百分比配合scrolITo方法来完成View的滑动。

采用这种方法除了能够完成弹性滑动以外，还可以实现其他动画效果，我们完全可以在onAnimationUpdate方法中加上我们想要的其他操作，如更新百分比。

#### 使用延时策略

核心思想是通过发送一系列延时消息从而达到一种渐近式的效果，具体来说可以使用Handler或View的postDelayed方法，也可以使用线程的sleep方法。对于postDelayed方法来说，我们可以通过它来延时发送一个消息，然后在消息中来进行View的滑动，如果接连不断地发送这种延时消息，那么就可以实现弹性滑动的效果。对于sleep方法来说，通过在while循环中不断的滑动View和sleep，就可以实现弹性滑动的效果。

|  |
| --- |
| private void testScrollDelay() {  //method 1  // handler.sendEmptyMessageDelayed(0, 1000);  //method 2  /\* mScrollTv.postDelayed(new Runnable() {  @Override  public void run() {  if (count++ <= FRAME\_COUNT) {  mScrollTv.scrollTo((int) (100 \* count / FRAME\_COUNT), 0);  mScrollTv.postDelayed(this,1000);  }else {  count = 0;  }  }  },1000);\*/  //method 3  /\* new Thread(new Runnable() {  @Override  public void run() {  while (true) {  try {  Thread.sleep(1000);  if (count++ <= FRAME\_COUNT) {  mScrollTv.scrollTo((int) (100 \* count / FRAME\_COUNT), 0);  } else {  break;  }  } catch (InterruptedException e) {  e.printStackTrace();  }  }  }  }).start();\*/  //method 4  CountDownTimer timer = new CountDownTimer(5000,1000) {  @Override  public void onTick(long millisUntilFinished) {  mScrollTv.scrollTo((int) (100 \* millisUntilFinished / 5000), 0);  }  @Override  public void onFinish() {  }  };  timer.start();  }  private static final float FRAME\_COUNT = 30;  private int count = 1;  private Handler handler = new Handler() {  @Override  public void handleMessage(Message msg) {  if (count++ <= FRAME\_COUNT) {  mScrollTv.scrollTo((int) (100 \* count / FRAME\_COUNT), 0);  handler.sendEmptyMessageDelayed(0, 1000);  }else {  count = 0;  }  }  }; |

## View的事件分发

重点/难点，滑动冲突

### 点击事件的传递规则

我们分析的点击事件可不是View.OnClickListener，而是我们MotionEvent，即点击事件，所谓事件分发，其实就是对MotionEvent事件的分发过程，即当一个MotionEvent产生了以后，系统需要把这个事件传递给一个具体的View，而这个传递的过程就是分发过程。点击事件的分发过程由三个很重要的分发来完成.dispatchTouchEvent，onInterceptTouchEvent和onTouchEvent,下面我们先介绍一下这几个方法。

* puhlic boolean dispatchTouchEvent (MotionEvent ev):用来进行事件的分发。如果事件能够传递给当前View，那么此方法一定会被调用，返回结果受当前View的onTouchEvent和下级View的dispatchTouchEvent方法的影响，表示是否消耗当前事件。
* public boolean onIntercept fouchEven(MotionEvent event):在上述方法内部调用，用来判断是否拦截某个事件，如果当前View拦截了某个事件，那么在同一个事件序列当中，此方法不会被再次调用，返回结果表示是否拦截当前事件
* public boolean onTouchEvent(MotionEvent event):在dispatchTouchEvent方法中调用，用来处理点击事件，返回结果表示是否消耗当前事件，如果不消耗，则在同一个事件序列中，当前View无法再次接收到事件。

上述三个方法到底有什么区别呢?它们是什么关系呢？其实它们的关系可以用如下伪代码可以了解：

|  |
| --- |
| @Override  public boolean **dispatchTouchEvent**(MotionEvent ev) {  boolean consume = false;  if(**onInterceptTouchEvent**(ev)){  consume = **onTouchEvent**(ev);  }else {  consume = **child.dispatchTouchEvent**(ev);  }  return consume;  } |

当一个View需要处理事件时，如果它设置了OnTouchListener，那么OnTouchListener中的onTooch方法会被回调。这时事件如何处理还要看onTouch的返回值，如果返回false,那当前的View的方法onTouchEvent会被调用；**如果返回true**(launcher透明点击的原因)，那么onTouchEvent方法将不会被调用。在onTouchEvent方法中，如果当前设置的有OnClickListener，那么它的onClick方法会用。由此可见， OnTouchListener，其优先级比onTouchEvent要高， OnClickListener优先级最低，即处于事尾端。

当一个点击事件产生后，它的传递过程遵循如下顺序：Activity>Window-View，即事件总是先传递给Activity,Activity再传递给Window，最后Window再传递给顶级View顶级View接收到事件后，就会按照事件分发机制去分发事件。考虑一种情况，如果一个view的onTouchEvent返回false，那么它的父容器的onTouchEvent将会被调用，依此类推,如果所有的元素都不处理这个事件，那么这个事件将会最终传递给Activity处理，即Activity的onTouchEvent方法会被调用。

这个过程其实也很好理解，我们可以换一种思路，假如点击事件是一个难题，这个难题最终被上级领导分给了一个程序员去处理（这是事件分发过程），结果这个程序员搞不定(onTouchEvent返回false)，现在该怎么办呢？难题必须要解决，那只能交给水平更高的上级解决（上级的onTouchEvent被调用)，如果上级再搞不定，那只能交给上级的上级去解决，就这样将难题一层层地向上抛，这是公司内部一种很常见的处理问题的过程。从这个角度来看，View的事件传递过程还是很贴近现实的，毕竟程序员也生活在现实中。

关于事件传递的机制，这里给出一些结论，根据这些结论可以更好地理解整个传递机制，如下所示

（1）同一个事件序列是指从手指接触屏幕的那一刻起，到手指离开屏慕的那一刻结束，在这个过程中所产生的一系列事件，这个事件序列以down事件开始，中间含有数量不定的move事件，最后以up结束

（2）正常情况下，一个事件序列只能被一个Visw拦截且消耗。这一条的原因可以参考（3），因为一旦一个元素拦截了某此事件，那么同一个事件序列内的所有事件都会直接交给它处理，因此同一个事件序列中的事件不能分别由两个View同时处理，但是通过特殊手段可以做到，比如一个Vew将本该自己处理的事件通过onTouchEvent强行传递给其他View处理。

（3)某个View一旦决定拦截，那么这一个事件序列都只能由它来处理（如果事件序列能够传递给它的话)，并且它的onInterceprTouchEvent不会再被调用。这条也很好理解，就是说当一个View决定拦截一个事件后，那么系统会把同一个事件序列内的其他方法都直接交给它来处理，因此就不用再调用这个View的onInterceptTouchEvent去询问它是否要拦截了。

(4）某个View一旦开始处理事件，如果它不消耗ACTON\_DOWN事件(onTouchEvent返回了false)，那么同一事件序列中的其他事件都不会再交给它来处理，并且事件将重新交由它的父元素去处理，即父元素的onTouchEvent会被调用。意思就是事件一旦交给一个View处理，那么它就必须消耗掉，否则同一事件序列中剩下的事件就不再交给它来处理了，这就好比上级交给程序员一件事，如果这件事没有处理好，短期内上级就不敢再把事情交给这个程序员做了，二者是类似的道理。

（5）如果View不消耗除ACTION\_DOWN以外的其他事件，那么这个点击事件会消失，此时父元素的onTouchEvent并不会被调用，并且当前View可以持续收到后续的事件，最终这些消失的点击事件会传递给Activity处理。

（6)ViewGroup默认不拦截任何事件。Android源码中ViewGroup的onInterceptTouchEvent方法默认返回false

（7）View没有onInterceptTouchEvent方法，一旦有点击事件传递给它，那么它的onTouchEvent方法就会被调用。

（8）view的onTouchEvent默认都会消耗事件（返回true)，除非它是不可点击的(clickable和longClickable同时为false)，View的longClickable属性默认都为false,clickable属性要分情况，比如Button的clickable属性默认为true，而TextView 的clickable属性默认为false

（9)view 的enable.属性不影响onTouchEvent的默认返回值。哪怕一个View是disable状态的，只要它的clickable或者longclickable有一个为true，那么它的onTouchEvent就返会true。

（10）onclick会发生的前提实际当前的View是可点击的，并且他收到了down和up的事件

(11)事件传递过程是由外到内的，理解就是事件总是先传递给父元素，然后再由父元素分发给子View，通过requestDisallowInterptTouchEvent方法可以再子元素中干预元素的事件分发过程，但是ACTION\_DOWN除外。

### 事件分发的源码解析

#### Activity对点击事件的分发过程

点击事件用MotionEvent来表示，当一个点击操作发生的时候，事件最先传递给Activity，由Activity的dispatchTouchEvent来进行事件的派发，具体的工作是由Activity内部的window来完成的，window会将事件传递给**decorview,decorview一般都是当前界面的底层容器（setContentView所设置的父容器）**，通过Activity.getWindow.getDecorView()获得，我们可用先从dispatchTouchEvent的源码看起：

|  |
| --- |
| public boolean dispatchTouchEvent(MotionEvent ev) {  if (ev.getAction() == MotionEvent.ACTION\_DOWN) {  onUserInteraction();  }//首先事件交给Activity所依附的window，如果true那就结束了  if (getWindow().superDispatchTouchEvent(ev)) {  return true;  }  return onTouchEvent(ev);// 所有View的onTouchEvent都返回false，那么Activity的onTouch就会被调用  } |

接下来我们看下window是如何将事件传递给ViewGroup的, window的实现类phonewindow(控制顶级的View的外观和行为机制).

|  |
| --- |
| public boolean superDispatchTouchEvent(MotionEvent ev){  return **mDecor**.superDispatchTouchEvent(ev);// phoneWindow传递给了DecorView  } |

|  |
| --- |
| public class DecorView extends FrameLayout implements RootViewSurfaceTaker {  private DecorView mDecor;  @Override  public final View getDecorView(){  if(mDecor == null){  installDesor():  }  return mDecor;  }  } |

我们知道:获取Activity所设置的View，这个mDecor显然就是个子View。

//R.layout.act\_ch3main =((ViewGroup)

getWindow().getDecorView().findViewById(android.R.id.content)).getChildAt(0);

目前事件传递到了Decorview这里，由于DecorView继承自FrameLayout且是父View，所以最终事件会传递给View。从这里开始，事件已经传递到顶级View了，就是在Activity中通过setContentview所设置的View，另外顶级View也叫根View，顶级View一般来说都是VewGroup。

#### 顶级View对事件的分发过程

关于点击事件如何在View中进行分发，上一节已经做了详细的介绍，这里再大致回顾一下。点击事件达到顶级View（一般是一个ViewGroup)以后，会调用ViewGiroup的的dispatchTouchEvent方法，然后的逻辑是这样的：如果顶级ViewGroup拦截事件即 onIntercepTouchEvent返回true，则事件由ViewGroup处理，这时如果ViewGroup的mOnTouchListener被设置，则onTouch会被调用，否则onTouchEvent会被调用。也就是说如果都提供的话，onTouch会屏蔽掉onTouchEvent。在onTouchEvent中，如果设置了 mOnClickListener,则onClick会被调用。如果顶级ViewGroup不拦截事件，则事件会传递给它所在的点击事件链上的子View，这时子View的dispatchTouchEvent会被调用。到此为止，事件已经从顶级View传递给了下一层View，接下来的传递过程和顶级View是一致的， 如此循环，完成整个事件的分发。

首先看ViewGroup对点击事件的分发过程，其主要实现在ViewGroup的dispatchTouchEvent方法中，这个方法比较长，这里分段说明。先看下面一段，很显然，它描述的是当View是否拦截点击事情这个逻辑。

|  |
| --- |
| // Check for interception.  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;  } |

从上面代码我们可以看出，ViewGroup在如下两种情况下会判断是否要拦截当前事件：事件类型为ACTION\_DOWN或者mFirstTouchTarget!=null。

ACTION\_DOWN事件好理解，那么mFirstTouchTargetl=null是什么意思呢？这个从后面的代码逻辑可以看出来，当事件由ViewGroup的子元素成功处理时，mFirstTouchTarget会被赋值并指向子元素，换种方式来说，当ViewGroup不拦截事件并将事件交由子元素处理时mFirstTouchTarget != null。反过来，一旦事件由当前ViewGroup拦截时，mFirstTouchTarget !=null就不成立。那么当ACTION\_MOVE和ACTION\_UP事件到来时，由于actionMasked == MotionEvent.ACTION\_DOWN|| mFirstTouchTarget != null这个条件为false，将导致ViewGroup 的onInterceptTouchEvent不会再被调用，并且同一序列中的其他事件都会默认交给它处理。

当然，这里有一种特殊情况，那就是FLAG\_DISALLOW\_INTERCEPT标记位，这个标记位是通过requestDisallowInterceptTouchEvent方法来设置的，一般用于子View中。FLAG\_DISALLOW\_INTERCEPT一旦设置后，ViewGroup将无法拦截除了ACTION\_DOWN以外的其他点击事件。**为什么说是除了ACTION\_DOWN以外的其他事件呢?**这是因为ViewGroup在分发事件时，如果是ACTION\_DOWN就会重置FLAG\_DISALLOW\_INTERCEPT这个标记位，将导致子View中设置的这个标记位无效。因此，当面对ACTION\_DOWN事件时，ViewGroup总是会调用自己的onInterceptTouchEvent方法来询问自己是否要拦截事件，这一点从源码中也可以看出来,在下面的代码中，ViewGroup会在ACTION\_DOWN事件到来时做重置的作用。而在requsstTouchState方法中会对FLAG\_DISALLOW\_INTERCEPT进行重置，因此子View调用requestDisallowInterceptTouchEvent方法并不能影响ViewGroup对ACTION\_DOWN事件的处理：

|  |
| --- |
| // Handle an initial down.  if (actionMasked == MotionEvent.ACTION\_DOWN) {  // Throw away all previous state when starting a new touch gesture.  // The framework may have dropped the up or cancel event for the previous gesture  // due to an app switch, ANR, or some other state change.  cancelAndClearTouchTargets(ev);  resetTouchState();  } |

从上面的源码分析，我们可用得出结论，当ViewGroup决定拦截事件之后，那么后续的点击事件，将会默认交给他处理并且不再调用他的onInterceptTouchEvent方法（**mFirstTouchTarget == null**)），这就证实了我们刚才所说的标记位无效的理论，当然前提是ViewGroup不拦截ACTION\_DOWN事件。

接下来我们再看下ViewGroup不拦截事件的时候，事件会向下分发由他的子View进行处理：

|  |
| --- |
| final View[] children = mChildren;  for (int i = childrenCount - 1; i >= 0; i--) {  final int childIndex = getAndVerifyPreorderedIndex(  childrenCount, i, customOrder);  final View child = getAndVerifyPreorderedView(  preorderedList, children, childIndex);  // If there is a view that has accessibility focus we want it  // to get the event first and if not handled we will perform a  // normal dispatch. We may do a double iteration but this is  // safer given the timeframe.  if (childWithAccessibilityFocus != null) {  if (childWithAccessibilityFocus != child) {  continue;  }  childWithAccessibilityFocus = null;  i = childrenCount - 1;  }  if (!canViewReceivePointerEvents(child)  || !isTransformedTouchPointInView(x, y, child, null)) {  ev.setTargetAccessibilityFocus(false);  continue;  }  newTouchTarget = getTouchTarget(child);  if (newTouchTarget != null) {  // Child is already receiving touch within its bounds.  // Give it the new pointer in addition to the ones it is handling.  newTouchTarget.pointerIdBits |= idBitsToAssign;  break;  }  resetCancelNextUpFlag(child);  if (dispatchTransformedTouchEvent(ev, false, child, idBitsToAssign)) {  // Child wants to receive touch within its bounds.  mLastTouchDownTime = ev.getDownTime();  if (preorderedList != null) {  // childIndex points into presorted list, find original index  for (int j = 0; j < childrenCount; j++) {  if (children[childIndex] == mChildren[j]) {  mLastTouchDownIndex = j;  break;  }  }  } else {  mLastTouchDownIndex = childIndex;  }  mLastTouchDownX = ev.getX();  mLastTouchDownY = ev.getY();  newTouchTarget = addTouchTarget(child, idBitsToAssign);  alreadyDispatchedToNewTouchTarget = true;  break;  } |

上面的代码逻辑还是比较清晰的，首先遍历的是ViewGroup的所有子元素，然后判断子元素是否能接受这个点击事件主要是两点来衡量，子元素是否**在播动画和点击是按的坐标是否落在子元素的区域内**，如果某子元素满足这两个条件，那么事件就会传递给他处理，可以看到，dispatchTransformedTouchEvent实际上调用的就是子元素的dispatchTouchEvent方法，在他的内部有下面的这段话，而在上面的代码中child传递的不是null，因此他会直接调用子元素的dispatchTouchEvent方法，这样事件就交由子元素处理处理，这就从而完成这一轮事件分发。

|  |
| --- |
| if (child == null) {  handled = super.dispatchTouchEvent(event);  } else {  handled = child.dispatchTouchEvent(event);  } |

如果子元素的dispatchTouchEvent返回true，这时我们暂时不考虑事件在子元素的怎么分发的，那么mFirstTouchTarget就会被赋值同时跳出for循环：

|  |
| --- |
| newTouchTarget = addTouchTarget(child, idBitsToAssign);  alreadyDispatchedToNewTouchTarget = true;  break; |

这几行代码就完成了mFirstTouchTarget的赋值并且并终止对子元素的遍历，如果子元素的dispatchTouchEvent返回false，ViewGroup就会把事件分给下一个子元素

其实mFirstTouchTarget真正的赋值过程是在addTouchTarget内部完成的，从下面的addTouchTarget的内部结构就可以看出，mFirstTouchTarget其实是一种单链表的结构，mFirstTouchTarget是否被赋值，将直接影响到ViewGroup对事件的拦截机制，如果mFirstTouchTarget为null，那么ViewGroup的默认拦截下来统一序列中所有的点击事件。

|  |
| --- |
| private TouchTarget addTouchTarget(@NonNull View child, int pointerIdBits) {  final TouchTarget target = TouchTarget.obtain(child, pointerIdBits);  target.next = mFirstTouchTarget;  mFirstTouchTarget = target;  return target;  } |

如果遍历所有的子元素后事件都没有被合适的处理，这包含两种情况，第一是Viewgroup没有子元素，第二是子元素处理了点击事件，但是在dispatchTouchEvent中返回false，这一版是因为子元素在onTouchEvent返回了false，这两种情况下，ViewGroup会自己处理点击事件，看代码：

|  |
| --- |
| // Dispatch to touch targets.  if (mFirstTouchTarget == null) {  // No touch targets so treat this as an ordinary view.  handled = dispatchTransformedTouchEvent(ev, canceled, null,  TouchTarget.ALL\_POINTER\_IDS);  } |

如果遍历所有的子元素后事件都没有被合适的处理，这包含两种情况，第一是Viewgroup没有子元素，第二是子元素处理了点击事件，但是在dispatchTouchEvent中返回false，这一版是因为子元素在onTouchEvent返回了false，这两种情况下，ViewGroup会自己处理点击事件，看代码：

|  |
| --- |
| // Dispatch to touch targets.  if (mFirstTouchTarget == null) {  // No touch targets so treat this as an ordinary view.  handled = dispatchTransformedTouchEvent(ev, canceled, null,  TouchTarget.ALL\_POINTER\_IDS);  } |

注意上面这段话，这里的第三个参数child为null，从上面的分析我们可用知道，他会调用supe.dispatchTouchEvent(event)，很显然，这里就转到了View的dispatchTouchEvent方法，就是点击事件开始给View處理了。

#### View对点击事件的处理

View对点击事件的处理稍微有点简单， 这里注意，这里的View不包含ViewGroup，先看他的dispatchTouchEvent方法

|  |
| --- |
| public boolean dispatchTouchEvent(MotionEvent event) {  // If the event should be handled by accessibility focus first.  if (event.isTargetAccessibilityFocus()) {  // We don't have focus or no virtual descendant has it, do not handle the event.  if (!isAccessibilityFocusedViewOrHost()) {  return false;  }  // We have focus and got the event, then use normal event dispatch.  event.setTargetAccessibilityFocus(false);  }  boolean result = false;  if (mInputEventConsistencyVerifier != null) {  mInputEventConsistencyVerifier.onTouchEvent(event, 0);  }  final int actionMasked = event.getActionMasked();  if (actionMasked == MotionEvent.ACTION\_DOWN) {  // Defensive cleanup for new gesture  stopNestedScroll();  }  if (onFilterTouchEventForSecurity(event)) {  if ((mViewFlags & ENABLED\_MASK) == ENABLED && handleScrollBarDragging(event)) {  result = true;  }  //noinspection SimplifiableIfStatement  ListenerInfo li = mListenerInfo;  if (li != null && li.**mOnTouchListener** != null  && (mViewFlags & ENABLED\_MASK) == ENABLED  && li.mOnTouchListener.**onTouch**(this, event)) {  result = true;  }  if (!result && **onTouchEvent**(event)) {  result = true;  }  }  if (!result && mInputEventConsistencyVerifier != null) {  mInputEventConsistencyVerifier.onUnhandledEvent(event, 0);  }  // Clean up after nested scrolls if this is the end of a gesture;  // also cancel it if we tried an ACTION\_DOWN but we didn't want the rest  // of the gesture.  if (actionMasked == MotionEvent.ACTION\_UP ||  actionMasked == MotionEvent.ACTION\_CANCEL ||  (actionMasked == MotionEvent.ACTION\_DOWN && !result)) {  stopNestedScroll();  }  return result;  } |

View点击事件的处理，因为他只是一个View，他没有子元素所以无法向下传递，所以只能自己处理点击事件，从上面的源码可以看出View对点击事件的处理过程，首选会判断你有没有设置onTouchListener，如果onTouchListener中的onTouch为true，那么onTouchEvent就不会被调用，可见onTouchListener的优先级高于onTouchEvent，这样做到好处就是方便在外界处理点击事件（监听onTouchListener）;

接着我们再来分析下onTouchEvent的实现，先看当View处于不可用的状态下点击事件的处理过程，如下，很显然，不可用状态下的View照样会**消耗点击事件**，尽管他看起来不可用：

|  |
| --- |
| if ((viewFlags & ENABLED\_MASK) == DISABLED) {  if (action == MotionEvent.ACTION\_UP && (mPrivateFlags & PFLAG\_PRESSED) != 0) {  setPressed(false);  }  // A disabled view that is clickable still consumes the touch  // events, it just doesn't respond to them.  return (((viewFlags & CLICKABLE) == CLICKABLE  || (viewFlags & LONG\_CLICKABLE) == LONG\_CLICKABLE)  || (viewFlags & CONTEXT\_CLICKABLE) == CONTEXT\_CLICKABLE);  } |

接着，如果View设置有代理，那么onTouchEvent还会执行TouchDelegate的onTouchEvent方法，这个onTouchEvent的工作机制看起来和onTouchListener类似，这里就不深究了

|  |
| --- |
| if (mTouchDelegate != null) {  if (mTouchDelegate.onTouchEvent(event)) {  return true;  }  } |

下面再看一下onTouchEvent中点击事件的具体处理，如下所示：

|  |
| --- |
| if (((viewFlags & CLICKABLE) == CLICKABLE ||  (viewFlags & LONG\_CLICKABLE) == LONG\_CLICKABLE) ||  (viewFlags & CONTEXT\_CLICKABLE) == CONTEXT\_CLICKABLE) {  switch (action) {  case MotionEvent.**ACTION\_UP:**  boolean prepressed = (mPrivateFlags & PFLAG\_PREPRESSED) != 0;  if ((mPrivateFlags & PFLAG\_PRESSED) != 0 || prepressed) {  // take focus if we don't have it already and we should in  // touch mode.  boolean focusTaken = false;  if (isFocusable() && isFocusableInTouchMode() && !isFocused()) {  focusTaken = requestFocus();  }  if (prepressed) {  // The button is being released before we actually  // showed it as pressed. Make it show the pressed  // state now (before scheduling the click) to ensure  // the user sees it.  setPressed(true, x, y);  }  if (!mHasPerformedLongPress && !mIgnoreNextUpEvent) {  // This is a tap, so remove the longpress check  removeLongPressCallback();  // Only perform take click actions if we were in the pressed state  if (!focusTaken) {  // Use a Runnable and post this rather than calling  // performClick directly. This lets other visual state  // of the view update before click actions start.  if (mPerformClick == null) {  mPerformClick = new PerformClick();  }  if (!post(mPerformClick)) {  **performClick**();  }  }  }  black;  }  ....  return true;  } |

从上面的代码来看，只要View的CLICKABLE和LONG\_CLICKABLE有一个为true，那么他就会消耗这个事件，即onTouchEvent返回true，不管他是不是DISABLE状态，这就证实了上面的第8,9,10结论，然后就是当ACTION\_UP事件发生之后，会触发performClick方法，如果View设置了onClickListener，那么performClick方法内部就会调用他的onClick方法

|  |
| --- |
| public boolean performClick() {  final boolean result;  final ListenerInfo li = mListenerInfo;  if (li != null && li.mOnClickListener != null) {  playSoundEffect(SoundEffectConstants.CLICK);  li.mOnClickListener.**onClick**(this);  result = true;  } else {  result = false;  }  sendAccessibilityEvent(AccessibilityEvent.TYPE\_VIEW\_CLICKED);  return result;  } |

View的LONG\_CLICKABLE属性默认为false，而CLICKABLE属性是否为false和具体的View有关，确切的说是可点击的View其CLICKABLE为true，不可点击的为false，比如button是可点击的，textview是不可点击的，通过setonclik或者longclik都是可以改变状态的，这点我们看源码：

|  |
| --- |
| public void setOnClickListener(@Nullable OnClickListener l) {  if (!isClickable()) {  setClickable(true);  }  getListenerInfo().mOnClickListener = l;  }  public void setOnLongClickListener(@Nullable OnLongClickListener l) {  if (!isLongClickable()) {  setLongClickable(true);  }  getListenerInfo().mOnLongClickListener = l;  } |

到这里，点击事件的分发机制源码就分析完成了，结合滑动事件可能理解的更好一点

## View的滑动冲突

这节要说的是滑动冲突，也是很多人碰到过的，也比较麻烦的，滑动冲突时怎样产生的？其实在界面中只要内外两层同时可以滑动，这个时候就会产生滑动冲突，如何解决滑动冲突？这既是一键困难的事情又是一件简单的事情，说困难时因为许多开发者面对滑动冲突都会显得束手无策，说简单是因为滑动冲突的解决办法有固定的套路，只要知道这个套路就好了。

### 常见的滑动冲突场景

常见的滑动一般有三个方面

外部滑动方向和内部滑动方向不一致

外部滑动方向和内部滑动方向一致

上面两种情况的嵌套



先说场景1，主要是将ViewPager和Fragment配合使用所组成的页面滑动效果，主流应用几乎都会使用这个效果。在这种效果中，可以通过左右滑动来切换页面，而每个页面内部往往又是一个Listview。本来这种情况下是有滑动冲突的，但是viewPager内部处理了这种滑动冲突，因此采用ViewPager时我们无须关注这个问题，如果我们采用的不是ViewPager而是ScrollView等，那就必须手动处理滑动冲突了，

否则造成的后果就是内外两层只能有一层能够滑动，这是因为两者之间的滑动事件有冲突。除了这种典型情况外，还存在其他情况，比如外部上下滑动、内部左右滑动等，但是它们属于同一类滑动冲突。

再说场景2，这种情况就稍微复杂一些，当内外两层都在同一个方向可以滑动的时候，显然存在逻辑问题。因为当手指开始滑动的时候，系统无法知道用户到底是想让哪一层滑动，所以当手指滑动的时候就会出现问题，要么只有一层能滑动，要么就是内外两层都滑动得很卡顿。在实际的开发中，这种场景主要是指内外两层同时能上下滑动或者内外两层，同时能左右滑动。

最后说下场景3，场景3是场景1和场景2两种情况的嵌套，因此场景3的滑动冲突看起来就更加复杂了。比如在许多应用中会有这么一个效果：内层有一个场景1中的滑动效果，然后外层又有一个场景2中的滑动效果。具体说就是，外部有一个SlidingMenu效果，然后内部有一个ViewPager,ViewPager的每一个页面中又是一个Listview。虽然说场景3的滑动冲突看起来更复杂，但是它是几个单一的滑动冲突的叠加，因此只需要分别处理内层和中层、中层和外层之间的滑动冲突即可，而具体的处理方法其实是和场景1、场景2相同的。

从本质上来说，这三种滑动冲突场景的复杂度其实是相同的，因为它们的区别仅仅是滑动策略的不同，至于解决滑动冲突的方法，它们几个是通用的

### 滑动冲突的处理规则

一般来说，不管滑动冲突多么复杂，它都有既定的规则，根据这些规则我们就可以选择合适的方法去处理。

如上图所示，对于场景1，它的处理规则是：当用户左右滑动时，需要让外部的View拦截点击事件，当用户上下滑动时，需要让内部View拦截点击事件。这个时候我们就可以根据它们的特征来解决滑动冲突，具体来说是：根据滑动是水平滑动还是竖直滑动来判断到底由谁来拦截事件，如下图所示，根据滑动过程中两个点之间的坐标就可以得出到到底由谁来拦截事行；如何根据坐标来获取滑动的方向呢？这个很简单，有很多可以参考，比如可以依据滑动路径和水平方向做形成的夹角，也可以依据水平方向和竖直方向上的距离差来判断，某些特殊时候还可以依据水平和竖直方向的速度差来做判断。这里我们可以通过水平和竖直方向的距离差来判断，比如竖直方向滑动的距离大就判断为竖直滑动。否则判断为水平滑动。根据这个规则就可以进行下一步的解决方法制定了。

对于场景2来说，比较特殊，它无法根据滑动的角度、距离差以及速度差来做判断，但是这个时候一般都能在业务上找到突破点，比如业务上有规定：当处于某种状态时需要外部View响应用户的滑动，而处于另外一种状态时则需要内部View来响应View的滑动,根据这种业务上的需求我们也能得出相应的处理规则，有了处理规则同样可以进行下一步处理。这种场景通过文字描述可能比较抽象，在下一节会通过实际的例子来演示这种情况的解决方案，那时就容易理解了，这里先有这个概念即可



对于场景3来说，它的滑动规则就更复杂了，和场景2一样，它也无法直接根据滑动的角度、距离差以及速度差来做判断，同样还是只能从业务上找到突破点，具体方法和场景2一样，都是从业务的需求上得出相应的处理规则，在下一节将会通过实际的例子来说明这种情况的解决方案。

### 滑动冲突的解决方式

在上节中描述了三种典型的滑动冲突场景，在本节将会一一分析各种场景并给出其中解决方法。首先我们要分析第一种滑动冲突场景，这也是最简单、最典型的一种滑动冲突，因为它的滑动规则比较简单，不管多复杂的滑动冲突，它们之间的区别仅仅是滑动的规则不同而已。抛开滑动规则不说，我们需要找到一种不依赖具体的滑动规则的解决办法，，在这里，我们就根据场景1的情况来得出通用的解决方案，然后场景2和场景3我们只需要修改有关滑动规则的逻辑即可。

上面说过，针对场景1中的滑动，我们可以根据滑动的距离差来进行判断，这个距离差就是我们的的滑动规则。如果用ViewPager去实现场景中的效果，我们不需要手动处理滑动冲突，因为ViewPager已经帮我们做了，但是我们这里为了更好的演示滑动冲突的解决思想，我们并没有采用ViewPager，其中在滑动过程中得到滑动的角度是很简单的，但是到底要怎么做才能将点击事件交给合适的View去做呢？这里可以用到事件分发，给出两个解决办法

#### 外部拦截法

所谓的外部拦截费是指点击事件都先经过父容器的拦截处理，如果父容器需要这个事件就给给他，这里我们还得重写我们的onIterceptTouchEvent，如下：

|  |
| --- |
| @Override  public boolean onInterceptTouchEvent(MotionEvent ev) {  boolean intercepted = false;  int x = (int) ev.getX();  int y = (int) ev.getY();  switch (ev.getAction()){  case MotionEvent.ACTION\_DOWN:  intercepted = false;  break;  case MotionEvent.ACTION\_MOVE:  if("父容器的点击事件"){  **intercepted = true;**  }else {  intercepted = false;  }  break;  case MotionEvent.ACTION\_UP:  intercepted = false;  break;  }  mLastXIntercept = x;  mLastYIntercept = x;  return intercepted;  } |

上述代码是外部拦截法的典型逻辑，针对不同的滑动冲突，只需要修改父容器需要当前点击事件这个条件即可，其他均不需做修改并且也不能修改。这里对上述代码再描述一下，在onInterceptTouchEvent方法中，首先是ACTION\_DOWN这个事件，父容器必须返回false，即不拦截ACTION\_DOWN事件，**这是因为一旦父容器拦截了ACTION\_DOWN，那么后续的ACTION\_MOVE和ACTION\_UP事件都会直接交由父容器处理**，这个时候事件没法再传递给子元素了；其次是ACTION\_MOVE事件，这个事件可以**根据需要来决定是否拦截，**如果父容器需要拦截就返回true，否则返回false；最后是ACTION\_UP事件，这里必须要返回false，因为ACTION\_UP事件本身没有太多意义,考虑一种情况，假设事件交由子元素处理，如果父容器在ACTION\_UP时返回了true，会导致子元素无法接收到ACTION\_UP事件，这个时候子元素中的onClick事件就无法触发，但是父容器比较特殊，一旦它开始拦截任何一个事件，那么后续的事件都会交给它处理，而ACTION\_UP作为最后一个事件也必定可以传递给父容器，即便父容器的onInterceptTouchEvent方法在ACTION\_UP时返回了false。

#### 内部拦截法

内部拦截法是指**父容器不拦截任何事件**，所有的事件都传递给子元素，如果子元素要消耗此事件就直接消耗掉，否则就交由父容器进行处理，这种方法和Android中的事件分发机制不一致，需要配合requestDisallowInterceptTouchEvent方法才能正常工作，使用起来较外部拦截法稍显复杂。它的伪代码如下，我们需要重写子元素的dispatchTouchEvent方法：

|  |
| --- |
| @Override  public boolean dispatchTouchEvent(MotionEvent event) {  int x = (int) event.getX();  int y = (int) event.getY();  switch (event.getAction()){  case MotionEvent.ACTION\_DOWN:  getParent().requestDisallowInterceptTouchEvent(true);  break;  case MotionEvent.ACTION\_MOVE:  int deltaX = x - mLastX;  int deltaY = x - mLastY;  if("父容器的点击事件"){  getParent().requestDisallowInterceptTouchEvent(false);  }  break;  case MotionEvent.ACTION\_UP:  break;  }  mLastX = x;  mLastY = y;  return super.dispatchTouchEvent(event);  } |

上述代码就是内部拦截法的典型代码，当面对不同的滑动策略只需要修改里面的条件即可，其他不需要做改动，除了子元素需要处理之外，父元素默认也要**拦截除ACTION\_DOWN之外的其他事件**，这样当子元素调用getParent().requestDisallowInterceptTouchEvent(true)方法时，父元素才能继续拦截所需要的事件

为什么父容器不能拦截ACTION\_DOWN事件呢？那是因为ACTION\_DOWN事件并接受FLAG\_DISALLOW\_DOWN这个标记位的控制，所以一旦父容器拦截，那么所有的事件都无法传递到子元素中，这样额你不拦截就无法起作用了，父元素要做的如下修改

|  |
| --- |
| @Override  public boolean onInterceptTouchEvent(MotionEvent ev) {  int action = ev.getAction();  if(action == MotionEvent.ACTION\_DOWN){  return false;  }else {  return true;  }  } |

#### 冲突解决

下面通过一个实例来分别接受这两种用法，我们来实现一个类似于ViewPgaer中嵌套ListView的效果，为了制造滑动冲突，我们写一个类似ViewPager的控件即可，名字叫做HorizontalScrollViewEx，这个控件的具体实现思想会在第四章进行详细介绍，这里我只说滑动冲突部分

为了实现ViewPager的效果，我们定义一个类似于水平的线性布局，只不过他可以水平滑动，初始化时，我们可以再他的内部添加诺干个ListView，这样一来，由于ListView是可以竖直滑动的，就造成了典型的滑动冲突，先看下Activity的实现：

|  |
| --- |
| public class MainActivity extends AppCompatActivity {  public static final String TAG = "MainActivity";  private HorizontalScrollViewEx mListContainer;  private int w,h;  @Override  protected void onCreate(Bundle savedInstanceState) {  super.onCreate(savedInstanceState);  setContentView(R.layout.activity\_main);  Log.i(TAG,"onCreate");  initView();  }  private void initView() {  LayoutInflater inflater = getLayoutInflater();  mListContainer = findViewById(R.id.container);  //屏幕宽高  WindowManager wm = (WindowManager) getSystemService(WINDOW\_SERVICE);  w = wm.getDefaultDisplay().getWidth();  h = wm.getDefaultDisplay().getHeight();  for (int i = 0; i < 3; i++) {  ViewGroup layout = inflater.inflate(R.layout.content\_layout,mListContainer,false);  layout.getLayoutParams().width = w;  TextView textview = (TextView) layout.findViewById(R.id.title);  textview.setText("page" + (i+1));  layout.setBackgroundColor(Color.rgb(255/(i+1),255/(i+1),0));  createList(layout);  mListContainer.addView(layout);  }  }  private void createList(ViewGroup layout) {  ListView listview = (ListView) layout.findViewById(R.id.list);  ArrayList<String>datas= new ArrayList<>();  for (int i = 0; i < 50; i++) {  datas.add("names" + i);  }  ArrayAdapter<String>adapter = new ArrayAdapter<String>(this,R.layout.content\_list\_item,R.id.name,datas);  listview.setAdapter(adapter);  }  } |

##### 场景1外部拦截法解决

首先我们采用外部拦截法来解决这个问题，按照之前的分析，我们只需要修改父容器需要拦截的条件即可，对于本例来说，父容器的拦截条件就是滑动过程中水平距离差比竖直距离差要大，在这种情况下，父容器就拦截当前点击事件，根据这一个条件进行修改，我们先来看他的拦截事件：

|  |
| --- |
| @Override  public boolean onInterceptTouchEvent(MotionEvent event) {  boolean intercepted = false;  int x = (int) event.getX();  int y = (int) event.getY();  switch (event.getAction()) {  case MotionEvent.ACTION\_DOWN: {  intercepted = false;  if (!mScroller.isFinished()) {  mScroller.abortAnimation();  intercepted = true;  }  break;  }  case MotionEvent.ACTION\_MOVE: {  int deltaX = x - mLastXIntercept;  int deltaY = y - mLastYIntercept;  if (Math.abs(deltaX) > Math.abs(deltaY)) {  intercepted = true;  } else {  intercepted = false;  }  break;  }  case MotionEvent.ACTION\_UP: {  intercepted = false;  break;  }  default:  break;  }  Log.d(TAG, "intercepted=" + intercepted);  mLastX = x;  mLastY = y;  mLastXIntercept = x;  mLastYIntercept = y;  return intercepted;  } |

上面的代码可以看出，他和外部拦截的伪代码差别很小，只是把父容器的拦截条件换成了实际的逻辑，在滑动过程中，当水平方向的距离大就判断水平滑动，为了能够水平滑动所以让父容器拦截事件，而竖直距离大于就不拦截，于是事件传递给了ListView，所以ListView能上下滑动，这就解决了冲突了

考虑到一种情况，如果此时用户正在水平滑动，但是水平滑动停止之前如果用户再迅速的进行竖直滑动，就会导致界面在水平滑动无法滑动到终点，而处于一种中间状态，为了避免这种不友好的体验，我们水平正在滑动的时候，下一个序列的点击仍然交给父容器，具体我们看代码：

|  |
| --- |
| case MotionEvent.ACTION\_DOWN:  intercepted = true;  if (!mScroller.isFinished()) {  mScroller.abortAnimation();  intercepted = true;  }  break; |

##### 场景1内部拦截法

如果采用内部拦截法也是可以的，按照前面对内部拦截法的分析，我们只需要修改ListView的dispatchTouchEvent方法中的父容器的拦截逻辑，同时让父拦截MOVE和Up事件即可。为了重写Listview的dispatchTouchfvent方法，我们必须自定义一个ListView，称为ListViewEx，然后对内部拦截法的模板代码进行修改，根据需要，ListViewEx的实现如下所示：

|  |
| --- |
| public class KListViewIn extends ListView {  public boolean dispatchTouchEvent(MotionEvent ev){  int x = (int)ev.getX();  int y = (int)ev.getY();  switch (ev.getAction()){  case MotionEvent.ACTION\_DOWN:  getParent().requestDisallowInterceptTouchEvent(true);  break;  case MotionEvent.ACTION\_MOVE:  if (Math.abs(mLastX - x) > Math.abs(mLastY - y)){  getParent().requestDisallowInterceptTouchEvent(false);  }  default:  break;  }  mLastX = x;  mLastY = y;  return super.dispatchTouchEvent(ev);  }  } |

除了对ListView的修改，我们还需要修改HorizontalScrollViewEx的事件分发

|  |
| --- |
| @Override  public boolean onInterceptTouchEvent(MotionEvent ev) {  if(ev.getAction() == MotionEvent.ACTION\_DOWN) return false;  return true;  } |

从实现上来看，内部拦截法的操作要稍微复杂一些，因此推荐采用外部拦截法来解决常见的滑动冲突。

下面通过一个实际的例子来分析场景2，首先我们提供一个可以上下滑动的父容器，这里就叫 StickyLayout，它看起来就像是可以上下滑动的竖直的LinearLayout，然后在它的内部中的滑动冲突了。当然这个StickyLayout是有滑动规则的：当Header显示时或者ListView

滑动到顶部时，由StickyLayout拦截事件：当Header隐藏时，这要分情况，如果Listview已经滑动到顶部并且当前手势是向下滑动的话，这个时候还是StickyLayout拦截事件，其他情况则由ListView拦截事件。这种滑动规则看起来有点复杂，为了解决它们之间的滑动冲突，我们还是需要重写父容器StickyLayout的onintercepTouchEvent方法，至于ListView则不用做任何修改，我们来看一下StickyLayout的具体实现，滑动冲突相关的主要代码：

##### 场景2？？？

## 参考

http://www.cnblogs.com/shiwei-bai/p/5206157.html

不同层级的View-》滑动冲突

# View的工作原理

在Android中，View是一个很重要的角色，简单来说，View是Android中视觉的呈现，在界面上Android提供了一套完整的GUI库，里面有很多控件，但是有时候往往并不能满足于需求，所以只有自定义View了，我们会简单的说下流程，然后再去实践

除了View的三大流程之外，View常见的回调方法也是必须掌握的，比如构造方法，onAttach,onVisibilityChanged，onDetach，另外对于一些有滑动效果的自定义View，还要处理滑动事件和滑动冲突，总的来说，自定义View有几种固定的类型，View或者ViewGroup，有的直接重写原生控件，这个就要看需求了，好的，我们直接开始吧！

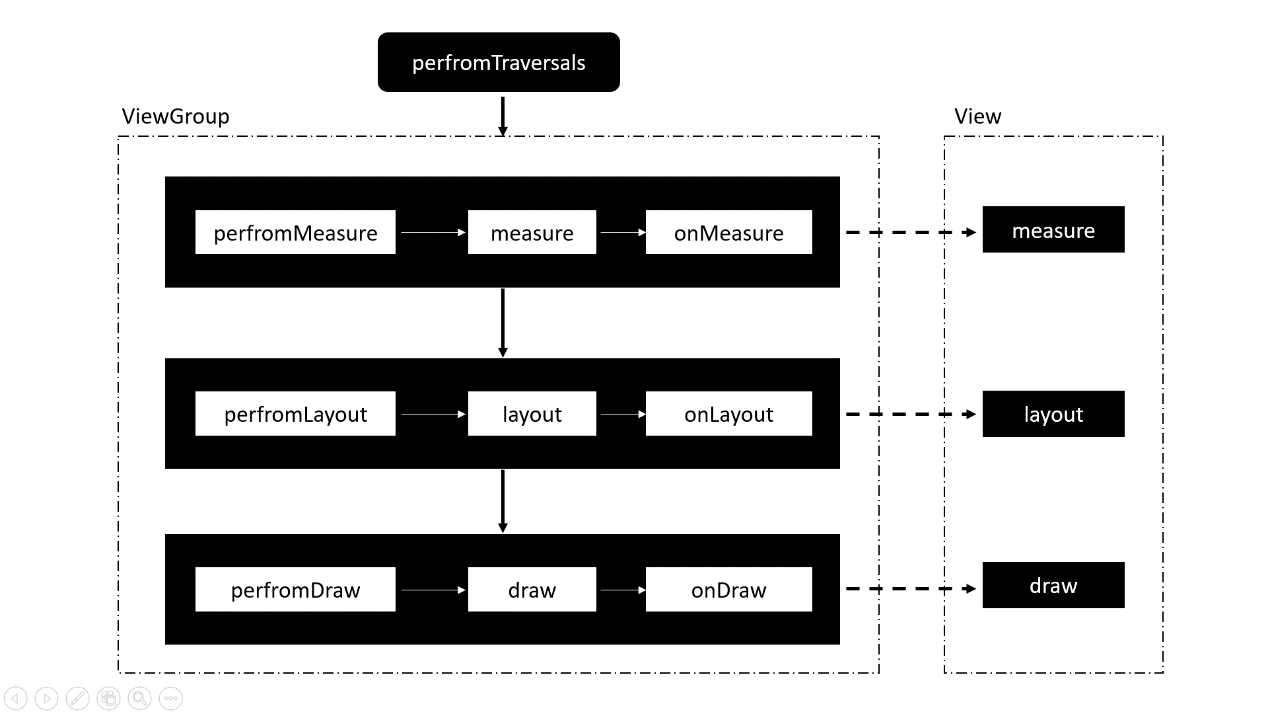
## 初识ViewRoot和DecorView

在正式介绍View的三大流程之前，我们还是要了解一些基本的概念，所以本章会说下ViewRoot和DecorView。

ViewRoot对应于ViewRootImpl类，他是连接WindowManager和DecorView的纽带，View的三大流程都是通过ViewRoot来完成的，在ActivityThread中，当Activity被创建完毕后，会将DecorView添加到Window，同时会创建ViewRootImpl对象，并将ViewRootImpl对象和DecorView建立联系，这个可以参照官网：

|  |
| --- |
| root = new ViewRootImpl(view.getContext(),display);  root.setView(view,wparams,panelParentView); |

View的绘制流程从ViewRoot的perfromTraversals方法开始，他经过measure，layout和draw三个过程才能将View画出来，，其中measure测量，layout确定view在容器的位置，draw开始绘制在屏幕上，针对perfromTraversals的大致流程，可以看图



图中的perfromTraversals会依次调用perfromMeasure，perfromLayout，perfromDraw，他们分别完成顶级View的measure，layout和draw这三大流程，其中在perfromMeasure中会调用measure方法，在measure方法中又调用onMeasure,这个时候measure流程就从父容器传递到子元素了，这样就完成了一次measure过程，接着子元素会重复父容器的measure过程，如此反复的完成了整个View树的遍历，同理，其他两个也是如此，唯一有点区别的是perfromDraw的传递过程是在draw反复中通过dispatchDraw来实现的，不过这并没有什么本质的区别

measure過程决定了View的宽高，Measure完成之后可以通过**getMeasureWidth**和getMeasureHeight来获取View测量后的高宽，在所有的情况下塔几乎都是等于最终的宽高，但是**特殊情况**除外，这点以后说，layout过程决定了view的四个顶点的坐标和实际View的宽高，完成之后，通过getTop，getLeft，getRight,getBottom获得，，Draw决定了View的显示，只有draw方法完成了之后，view才会显示在屏幕上

如下图，顶级View DecorView，一般情况下他内部会包含一个竖直方向的LinearLayout，这里面有上下两部分，上面是标题栏，下面是内容，在Activity中，我们可用通过setContentView设置布局文件就是放在内容里，而内容栏的id为content,因此我們可以理解为实际上是在setView，那如何得到content呢？你可以ViewGroup content = findviewbyid(android.R.id.content),如何得到我们设置的View呢：content.getChildAt(0),同时，通过源码我们可用知道，DeaorView其实就是一个FrameLayout，View层事件都先经过DecorView，然后传递给View



## 理解MeasureSpec

为了更好的理解View的测量过程，我们还需要理解MeasureSpec，从名字上看，MeasureSpec看起来像“测量规格”或者“测量说明书”，不管怎么翻译，他看起来就好像是或多或少的决定了View的测量过程，通过源码可以发现，MeasureSpec的确参与了View的测量过程，读者可能有疑问，MeasureSpec是干什么的呢？MeasureSpec在很大程度上决定了一个View的尺寸规格，之所以说很大程度上是因为这个过程还收到了父容器的影像，因为父容器影像MeasureSpec的创建过程，在测量过程中，系统会将View的LayoutParams根据父容器所施加的规则转换成对应的MeasureSpec，然后再根据这个measureSpec来测量出View的宽高，MeasureSpec看起来有点复杂，其实他的实现很简单，我们来详细分解一下

### MeasureSpec

MeasureSpec代表一个32位int值，高两位代表SpecMode，低30位代表SpecSize，SpecMode是指测量模式，而SpecSize是指在某个测量模式下的规格大小，下面先看一下，MeasureSpec内部的一些常量定义，通过这些就不难理解MeasureSpec的工作原理了

|  |
| --- |
| private static final int MODE\_SHIFT = 30;  private static final int MODE\_MASK = 0x3 << MODE\_SHIFT;  public static final int UNSPECIFIED = 0 << MODE\_SHIFT;  public static final int EXACTLY = 1 << MODE\_SHIFT;  public static final int AT\_MOST = 2 << MODE\_SHIFT;    @MeasureSpecMode  public static int getMode(int measureSpec) {  //noinspection ResourceType  return (measureSpec & MODE\_MASK);  }  public static int getSize(int measureSpec) {  return (measureSpec & ~MODE\_MASK);  } |

MeasureSpec通过将SpecMode和SpecSize打包成一个int值来避免过多的对象内存分配，为了方便操作，其提供了打包和解包的作用，SpecMode和specSize也是一个int值，一直SpecMode和specSize可以打包成一个MeasureSpec，一个MeasureSpec可以通过解包的形式来得出其原始的SpecMode和SpecSize，需要注意的是这里提到的MeasureSpec是指MeasureSpec所代表的int值，而非MeasureSpec本身。

SpecMode有三类，每一类都有特殊的含义

* UNSPECIFIED：父容器不对View有任何的限制，要多大给多大，这种情况一般用于系统内部，表示一种测量的状态
* EXACTLY：父容器已经检测出View所需要的精度大小，这个时候View的最终大小就是SpecSize所指定的值，它对应于LayoutParams中的match\_parent,和具体的数值这两种模式
* AT\_MOST:父容器指定了一个可用大小，即SpecSize，view的大小不能大于这个值，具体是什么值要看不同view的具体实现，它对应于LayoutParams中wrap\_content

### MeasureSpec 和 LayoutParams 的对应关系

系统内部是通过MeasureSpec来进行View的测量，正常情况下我们使用View指定MeasureSpec，但是尽管如此，我们也可以给View设置layoutparams，在view测量的时候，系统会将layoutparams在父容器的约束下转换成对应的MeasureSpec，然后再根据这个MeasureSpec来确定view测量后的宽高.需要注意的是，MeasureSpec不是唯一由layoutparams决定的，layoutparams需要和父容器一起决定view的MeasureSpec从而进一步决定view的宽高.

对于顶级view（DecorView）和普通的view来说，MeasureSpec的转换过程有些不同，对于decorview由窗口属性和自身layoutparams；普通的view其MeasureSpec由父容器的MeasureSpec和自身的layoutparams来决定。

MeasureSpec一旦确定后，MeasureSpec就可以去为view测量了.对于DecorView来说，在ViewRootImpl中的measureHierarchy方法中有这么一段代码。他展示了DecorViwew的MeasureSpec创建过程，其中desiredWindowWidth和desiredWindowHeight是屏幕的尺寸。

#### DecorView

对于DecorView来说，在ViewRootImpl中的measureHierarchy方法中有这么一段代码。他展示了DecorViwew的MeasureSpec创建过程，其中desiredWindowWidth和desiredWindowHeight是屏幕的尺寸

|  |
| --- |
| childWidthMeasureSpec = getRootMeasureSpec(baseSize, lp.width);  childHeightMeasureSpec = getRootMeasureSpec(desiredWindowHeight, lp.height);  performMeasure(childWidthMeasureSpec, childHeightMeasureSpec); |

接下来看下getRootMeasureSpec方法的实现：

|  |
| --- |
| private static int getRootMeasureSpec(int windowSize, int rootDimension) {  int measureSpec;  switch (rootDimension) {  case ViewGroup.LayoutParams.MATCH\_PARENT:  // Window can't resize. Force root view to be windowSize.  measureSpec = MeasureSpec.makeMeasureSpec(windowSize, MeasureSpec.EXACTLY);  break;  case ViewGroup.LayoutParams.WRAP\_CONTENT:  // Window can resize. Set max size for root view.  measureSpec = MeasureSpec.makeMeasureSpec(windowSize, MeasureSpec.AT\_MOST);  break;  default:  // Window wants to be an exact size. Force root view to be that size.  measureSpec = MeasureSpec.makeMeasureSpec(rootDimension, MeasureSpec.EXACTLY);  break;  }  return measureSpec;  } |

通过上述代码，DecorView的MesourSpec的产生过程就很明确了，具体来说其遵守了如下格式，根据layoutparams的宽高的参数来划分

* + LayouParams.MATCH\_PARENT:精确模式，大小就是窗口的大小
  + LayouParams.WRAP\_CONTENT:最大模式，大小不定，但是不能超出屏幕的大小
  + 固定大小（比如100dp）:精确模式，大小为LayoutParams中指定的大小

#### 普通的View

对于普通的View来说，这里是指我们布局中的View,View的measure过程由ViewGroup传递而来，先看下ViewGroup的measureChildWithMargis方法：



上述的方法会对子元素进行measure，在调用子元素的measure方法之前会通过getChildMeasureSpec方法得到子元素的MesureSpec，从代码上看，很显然，子元素的MesureSpec的创建和父容器的MesureSpec和子元素的LayoutParams有关，此外，还和view的margin有关，具体可以看下ViewGroup的getChildMeasureSpec方法







上述方法不难理解，他的主要作用是根据父容器的MeasureSpec同时结合view本身来layoutparams来确定子元素的MesureSpec，参数中的pading是指父容器中已占有的控件大小，因此子元素可以用的大小为父容器的尺寸减去pading，具体代码

|  |
| --- |
| int specSize = MesureSpec.getSize(spec);  int size = Math.max(0,specSize - pading); |

getChildMeasureSpec清楚的展示了普通View的MeasureSpec同时结合View本身的LayoutParams来确定子元素的MeaureSpec的创建规则，更加清晰的理解getChildMeasureSpec的逻辑，这里提供一个表，表中对getChildMeasureSpec的工作原理进行了梳理，表中的parentSize是指父容器中目前可使用的大小：



针对这张表，这里再做一下说明。前面已经提到，对于普通View，其MeasureSpec 由父容器的MeasureSpec和自身的LayoutParams来共同决定，那么针对不同的父容器和Viev本身不同的LayoutParams,View就可以有多种MeasureSpec。这里简单说一下，当View采用固定宽/高的时候，不管父容器的MeasureSpec是什么，View 的MeasureSpee都是精确模式，那么View也是精准模式并且其大小是父容器的剩余空间；如果父容器是最大模式，那么View也是最大模式并且其大小不会超过父容器的剩余空间。当View的宽/高是wrap\_content时，不管父容器的模式是精准还是最大化，View的模式总是最大化,并且大小不能超过父容器的剩余空间，可能读者会发现，在我们的分析中漏掉了UNSPECIFIED模式，那是因为这个模式主要用于系统内部多次Measure的情形，一般来说，我们不需要关注此模式。

通过这张表可以看出，只要提供父容器的MeasureSpec和子元素的LayoutParams，就可以快速地确定出子元素的MeasureSpec了，有了 MeasureSpec就可以进一步确定出子元亲测量后的大小了。需要说明的是，表中并非是什么经验总结，它只是getcchildMeasureSpec这个方法以表格的方式呈现出来而已

## View的工作流程

View的工作流程主要是指measure、layout、draw这三大流程，即测量、布局和绘制，其中measure确定View的测量宽/高，layout确定View的最终宽/高和四个顶点的位置，而draww则将View绘制到屏幕上。

### measure过程

measure过程要分情况来看，如果只是一个原始的View，那么通过measure方法就可以完成了其测量过程，如果是一个ViewGroup，除了完成自己的测量过程外，还会遍历去调用所有子元素的measure方法，各个子元素再递归去执行这个流程，下面针对这两种情况分别讨论.

#### View的measure过程

View 的 measure过程由其measure方法来完成，measure方法是一个final类型的方法，这就意味着子类不能重写此方法，在View的measure方法中去调用View的onMesure方法，因此只需要看onMeasure的实现即可，View的onMesure方法如下所示：

|  |
| --- |
| @Override  protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {  setMeasuredDimension(  getDefaultSize(getSuggestedMinimumWidth(), widthMeasureSpec),  getDefaultSize(getSuggestedMinimumHeight(), heightMeasureSpec));  } |

上面的代码很简介，但是简洁不代表简单，setMeasuredDimension会设置View宽/高的测量值，因此我们只需要getDefaultSize方法即可。

|  |
| --- |
| public static int getDefaultSize(int size, int measureSpec) {  int result = size;  int specMode = MeasureSpec.getMode(measureSpec);  int specSize = MeasureSpec.getSize(measureSpec);  switch (specMode) {  case MeasureSpec.UNSPECIFIED:  result = size;  break;  case MeasureSpec.AT\_MOST:  case MeasureSpec.EXACTLY:  result = specSize;  break;  }  return result;  } |

可以看出，getDefaultSize这个逻辑很简单，对于我们来说，我们只需要看AT\_MOST和EXACTLY这两种情况，简单的理解，其实getDefaultSize返回的大小就是mesourSpec中的specSize，而这个specSize就是view的大小，这里多次提到测量后的大小，是因为View最终的大小，是在layout阶段的，所以这里必须要加以区分，但是几乎所有情况下的View的测量大小和最终大小是相等的.

至于UNSPECIFIED这种情况，一般用于系统内部的测量过程，在这种情况下，View的大小为getDefaultSize的第一个参数是size，即宽高分别为getSuggestedMinimumWidth和getSuggestedMinimumHeight()这两个方法的返回值：



从 getSuggestedMinimumWidth的代码可以看出，如果View没有设置背景，View的宽度为mMinwidth，而mMinwidth对应于android:minwidth这个属性所指定的值，因此View的宽度即为android:minwidth属性所指定的值。这个属性如果不指定，那么MinWidth则默认为0；如果View指定了背景，则View的宽度为max(mMinwidth

mbackground().getMininumwidth)，mMinwidthh的含义我们已经知道了，那么mBackground.getMinimumWidth()是什么呢?我们看一下Drawable的 getMinimumWidth方法，如下所示：



可以看出，getMinimumWidth返回的就是Drawable的原始宽度，前提是这个Drawable有原始宽度，否则就返回0。那么Drawable在什么情况下有原始宽度呢？这里先举个例子说明一下，ShapeDrawable无原始宽/高，而BitmapDrawable有原始宽/高(图片的尺寸)，详细内容会在第6章进行介绍。

这里再总结一下getSuggestedMinimumWidth的逻辑：如果View没有设置背景，那么返回android:minwidth这个属性所指定的值，这个值可以为0：如果View设置了背景，则返回 android:minwidth和背景的最小宽度这两者中的最大值，getSuggestedMinimumWidth和getSuggestedMinimumHeight的返回值就是View 在UNSPECIFIED情况下的测量宽/高。

从getDefaulSize方法的实现来看，**View的宽/高由specSize决定**，所以我们可以得出如下结论：**直接继承View的自定义控件需要重写onMeasure**方法并设置wrap\_content时的自身大小，**否则在布局中使用wrap\_content就相当于使用matchparent**。为什么呢?这个原因需要结合上述代码和之前的表才能更好地理解。从上述代码中我们知道，如果View在布局中使用wrapcontent，那么它的specMode是AT\_MOST模式，在这种模式下，它的宽/高等于 specSize；查表4-1可知，这种情况下View的specSize是parentSize,而parentSize是父容器中目前可以使用的大小，也就是父容器当前剩余的空间大小。很显然，View的宽/高就等于父容器当前剩余的空间大小，这种效果和在布局中使用match\_parent完全一致。如何解决这个问题呢?也很简单，代码如下所示。

|  |
| --- |
| @Override  protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {  super.onMeasure(widthMeasureSpec, heightMeasureSpec);  int widthSpecMode = MeasureSpec.getMode(widthMeasureSpec);  int widthSpecSize = MeasureSpec.getMode(widthMeasureSpec);  int heightSpecMode = MeasureSpec.getMode(heightMeasureSpec);  int heightSpecSize = MeasureSpec.getMode(heightMeasureSpec);  if (widthSpecMode == MeasureSpec.AT\_MOST && heightSpecMode == MeasureSpec.AT\_MOST) {  setMeasuredDimension(mWidth, mHeight);  } else if (widthSpecMode == MeasureSpec.AT\_MOST) {  setMeasuredDimension(mWidth, heightSpecSize);  } else if (eightSpecMode == MeasureSpec.AT\_MOST) {  setMeasuredDimension(widthSpecSize, mHeight);  }  } |

在上面的代码中，我们只需要给View指定一个默认的内部宽/高(mWidth和mHeight)),并在wrapcontent时设置此宽/高即可。对于非wrapcontent情形，我们沿用系统的测量值即可，至于这个默认的内部宽/高的大小如何指定，这个没有固定的依据，根据需要灵活指定即可。如果查看TextView、Imageview等的源码就可以知道，针对 wrapcontent情形，它们的onMeasure方法均做了特殊处理，读者可以自行查看它们的源码

#### ViewGroup的measure过程

对于ViewGroup来说，除了完成自己的measure过程以外，还会遍历去调用所有子元素的measure方法，各个子元素再通归去执行这个过程。和View不同的是，ViewGroup是一个抽象类，因此它没有重写View的onMeasure方法，但是它提供了一个叫measureChildren:

|  |
| --- |
| protected void measureChildren(int widthMeasureSpec, int heightMeasureSpec) {  final int size = mChildrenCount;  final View[] children = mChildren;  for (int i = 0; i < size; ++i) {  final View child = children[i];  if ((child.mViewFlags & VISIBILITY\_MASK) != GONE) {  measureChild(child, widthMeasureSpec, heightMeasureSpec);  }  }  } |

从上述代码中看到，在ViewGroup的measure时，会对每一个子元素进行测量，那么这个方法就很好理解了

|  |
| --- |
| protected void measureChild(View child, int parentWidthMeasureSpec,  int parentHeightMeasureSpec) {  final LayoutParams lp = child.getLayoutParams();  final int childWidthMeasureSpec = getChildMeasureSpec(parentWidthMeasureSpec,  mPaddingLeft + mPaddingRight, lp.width);  final int childHeightMeasureSpec = getChildMeasureSpec(parentHeightMeasureSpec,  mPaddingTop + mPaddingBottom, lp.height);  child.measure(childWidthMeasureSpec, childHeightMeasureSpec);  } |

很显然，measurechild的思想就是取出子元素的LayoutParams，然后再通过getChidMeasureSpec来创建子元素的MeasureSpec，接着将MeasureSpec直接传递给View的measure方法来进行测量。getchildMeasureSpec的工作过程已经在上面进行了详细分析。

我们知道，ViewGroup并没有定义其测量的具体过程，这是因为ViewGroup是一个抽象类，其测量过程的onMeasure方法需要各个子类去具体实现，比如LinearLayout，RelativeLayout等，为什么ViewGroup不像View一样对其onMeasure方法做统一的实现呢？那是因为不同的ViewGroup子类有不同的布局特性，这导致它们的测量细节各不相同，比如Lineartayout和RelativeL.ayout这两者的布局特性显然不同，因此ViewGroup无法做统一实现。下面就通过LinearLayout的onMeasure方法来分析ViewGroup的 measure过程，其他Layout类型读者可以自行分析。

首先，我们来看一下LinearLayout的onMeasure方法

|  |
| --- |
| @Override  protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {  if (mOrientation == VERTICAL) {  measureVertical(widthMeasureSpec, heightMeasureSpec);  } else {  measureHorizontal(widthMeasureSpec, heightMeasureSpec);  }  } |

上述的代码很简单我们选择一个来看下，比如选中竖直方向的LinearLayout测量过程，即measureVertical，他的源码还比较长，我们看:

从上面的代码可以看出，系统会遍历子元素并对每一个子元素执行measureChildBeforeLayout方法，这个方法内部会调用子元素的measure方法，这样各个子元素就开始依次进入measure过程，并且系统通过mTotalLength这个变量来存储LinearLayout在竖直方向上的初步高度，没测量一个子元素，mTotalLength就会增加，增加的部分主要包括子元素的高度以及竖直方向上的margin等，当子元素测量完毕之后，LinearLayout会测量自己的大小，看源码:

这里对上述代码进行说明，当子元素测量完毕之后，LinearLayout会根据子元素的情况来测量自己的大小，针对竖直的LinearLayout而言，他的水平方向的测量过程遵循View的测量过程，在竖直方向的测量过程和View有些不同，具体来说，是指，如果他的布局中高度采用的是match\_parent或者具体值，那么他的绘制过程和View一致，即高度为specSize，如果他的布局中高度采用warp\_content，那么她的高度是所有的子元素所占用的高度综合，但是仍然不能超过他的父容器剩余空间，但是他的最终高度还是需要考虑其他的竖直方向上的pading，这个过程进一步参看源码:

View的onMeasure是三大流程中最复杂的一个，measure完成以后，通过getMeasureWidth/Height就可以正确地获取到View的测量宽/高。需要注意的是，在某些极端情况下measure才能确定最终的测量宽/高，在这种情形下，系统可能要多次调用measure方法进行测量，在这种情况下，载onMeasure方法中拿到的测量值很可能是不准确的。一个比较好的习惯是在**onLayout**方法中去获取View的测量宽/高或者最终宽/高。

#### 获取某个View的宽/高

上面已经对Viaw的measure过程进行了详细的分析，现在考虑一种情况，比如我们想在Activity已启动的时候就做一件任务，但是这一件任务需要获取某个View的宽/高，读者可能会说，这很简单啊，在onCreate或者onResume里面去获取这个View的宽/高就行了，读者可以自行试一下，实际上在onCreate、onStart、onResume中均无法正确得View的宽/高信息，这是因为View的measure过程和Activity的生命周期方法不是同步执行的，因此无法保证Activiy执行了onCreate、onStart、onResume时某个Vicw已经完毕了，如果View还没有测量完毕，那么获得的宽/高就是0。有没有什么方法能解决问题呢？答案是有的，这里给出四种方法来解决这个问题：

##### Activity/View#onWindowFocusChanged

onWindowFocusChanged这个方法的含义是：View已经初始化完毕了，宽/高已经准备好了，这个时候去获取宽/高是没问题的。需要注意的是，onWindowFocusChanged会被调用多次，当Activity的窗口得到焦点和失去焦点时均会被调用一次。具体来说，当Activity继续执行和暂停执行时，onWindowFocusChanged均会被调用，如果频繁地进行onResume和onPause，那么onWindowFocusChanged也会被频繁地调用。典型代码如下：

|  |
| --- |
| public void onWindowFocusChanged(boolean hasFocus) {  if (hasFocus){  KLogUtil.D(TAG,"ok1:onWindowFocusChanged,h="+ mKv.getMeasuredHeight()+",w="+ mKv.getMeasuredWidth());  }  } |

##### view.post(runnable)

通过post可以将一个runnable投递到消息队列，然后等到Lopper调用runnable的时候，View也就初始化好了，典型代码如下:

|  |
| --- |
| mKv.post(new Runnable() {  @Override  public void run() {  KLogUtil.D(TAG,"ok2:Runnable,h="+ mKv.getMeasuredHeight()+",w="+ mKv.getMeasuredWidth());  }  }); |

##### ViewTreeObserver

使用ViewTreeObserver的众多回调可以完成这个功能，比如使用OnGlobalLayoutListener这个接口，当View树的状态发生改变或者View树内部的View的可见性发生改变，onGlobalLayout方法就会回调，因此这是获取View的宽高一个很好的例子，需要注意的是，伴随着View树状态的改变，这个方法也会被调用多次，典型代码如下

|  |
| --- |
| ViewTreeObserver observer = mTextView.getViewTreeObserver();  observer.addOnGlobalLayoutListener(new ViewTreeObserver.OnGlobalLayoutListener() {  @Override  public void onGlobalLayout() {  mTextView.getViewTreeObserver().removeOnGlobalLayoutListener(this);  int width = mTextView.getMeasuredWidth();  int height = mTextView.getMeasuredHeight();  }  }); |

##### ~~view.measure(int wMeaSpec , int hMeaSpec)~~

通过手动测量View的宽高，这种方法比较复杂，这里要分情况来处理，根据View的LayoutParams来处理.

match\_parent:直接放弃，无法测量出具体的宽高，根据View的测量过程，构造这种measureSpec需要知道parentSize，即父容器的剩下空间，而这个时候我们无法知道parentSize的大小，所以理论上我们不可能测量出View的大小

具体的数值: 比如宽高都是100dp，那我们可以这样:

|  |
| --- |
| int widthMeasureSpec = View.MeasureSpec.makeMeasureSpec(100, View.MeasureSpec.EXACTLY);  int heightMeasureSpec = View.MeasureSpec.makeMeasureSpec(100, View.MeasureSpec.EXACTLY);  mTextView.measure(widthMeasureSpec,heightMeasureSpec); |

warap\_content:

|  |
| --- |
| int widthMeasureSpec = View.MeasureSpec.makeMeasureSpec((1<<30)-1, View.MeasureSpec.AT\_MOST);  int heightMeasureSpec = View.MeasureSpec.makeMeasureSpec((1<<30)-1, View.MeasureSpec.AT\_MOST);  mTextView.measure(widthMeasureSpec,heightMeasureSpec); |

注意到(1<<30)-1， 通过分析MeasureSpec的实现可以知道，View的尺寸三十位的二进制表示，也就是说最大是30个1（2^30-1）,也就是（1<30-1）,在最大的模式下，我们用View理论上能支持最大值去构造MwasureSpec是合理的

关于View的measure，网络上有两个错误的用法，为什么说是错误的，首先其违背了系统的内部实现规范（因为无法通过错误的MeasureSpec去得出合理的SpecMode，从而导致measure过程出错，其次不能保证mwasure出正确的结果）

第一种错误的方法:

|  |
| --- |
| int widthMeasureSpec = View.MeasureSpec.makeMeasureSpec(-1, View.MeasureSpec.UNSPECIFIED);  int heightMeasureSpec = View.MeasureSpec.makeMeasureSpec(-1, View.MeasureSpec.UNSPECIFIED);  mTextView.measure(widthMeasureSpec,heightMeasureSpec); |

第二种错误的用法

mTextView.measure(LayoutParams.WRAP\_CONTENT,LayoutParams.WRAP\_CONTENT);

### layout过程

Layout的作用是ViewGroup用来确定子元素的作用的，当ViewGroup的位置被确认之后，他的layout就会去遍历所有子元素并且调用onLayout方法，在layout方法中onLayou又被调用，layout的过程和measure过程相比就要简单很多了，layout方法确定了View本身的位置，而onLayout方法则会确定所有子元素的位置，先看View的layout方法



layout的方法的大致流程如下，首先会通过一个setFrame方法来设定View的四个顶点的位置，即初始化mLeft,mTop,mRight,mBottom这四个值，View的四个顶点一旦确定，那么View在父容器的位置也就确定了，接下来会调用onLayout方法，这个方法的用途是调用父容器确定子元素的位置，和onMeasure类似，onLayout的具体位置实现同样和具体布局有关，所有View和ViewGroup均没有真正的实现onLayout方法，我们来看一下LinearLayout的onLayout，layoutVertical(l, t, r, b);

|  |
| --- |
| void layoutVertical(int left, int top, int right, int bottom) {  final int paddingLeft = mPaddingLeft;  int childTop;  int childLeft;  // Where right end of child should go  final int width = right - left;  int childRight = width - mPaddingRight;  // Space available for child  int childSpace = width - paddingLeft - mPaddingRight;  final int count = getVirtualChildCount();  final int majorGravity = mGravity & Gravity.VERTICAL\_GRAVITY\_MASK;  final int minorGravity = mGravity & Gravity.RELATIVE\_HORIZONTAL\_GRAVITY\_MASK;  switch (majorGravity) {  case Gravity.BOTTOM:  // mTotalLength contains the padding already  childTop = mPaddingTop + bottom - top - mTotalLength;  break;  // mTotalLength contains the padding already  case Gravity.CENTER\_VERTICAL:  childTop = mPaddingTop + (bottom - top - mTotalLength) / 2;  break;  case Gravity.TOP:  default:  childTop = mPaddingTop;  break;  }  for (int i = 0; i < count; i++) {  final View child = getVirtualChildAt(i);  if (child == null) {  childTop += measureNullChild(i);  } else if (child.getVisibility() != GONE) {  final int childWidth = child.getMeasuredWidth();  final int childHeight = child.getMeasuredHeight();  final LinearLayout.LayoutParams lp =  (LinearLayout.LayoutParams) child.getLayoutParams();  int gravity = lp.gravity;  if (gravity < 0) {  gravity = minorGravity;  }  final int layoutDirection = getLayoutDirection();  final int absoluteGravity = Gravity.getAbsoluteGravity(gravity, layoutDirection);  switch (absoluteGravity & Gravity.HORIZONTAL\_GRAVITY\_MASK) {  case Gravity.CENTER\_HORIZONTAL:  childLeft = paddingLeft + ((childSpace - childWidth) / 2)  + lp.leftMargin - lp.rightMargin;  break;  case Gravity.RIGHT:  childLeft = childRight - childWidth - lp.rightMargin;  break;  case Gravity.LEFT:  default:  childLeft = paddingLeft + lp.leftMargin;  break;  }  if (hasDividerBeforeChildAt(i)) {  childTop += mDividerHeight;  }  childTop += lp.topMargin;  setChildFrame(child, childLeft, childTop + getLocationOffset(child),  childWidth, childHeight);  childTop += childHeight + lp.bottomMargin + getNextLocationOffset(child);  i += getChildrenSkipCount(child, i);  }  }  } |

里分析一下layoutVertical的代码逻辑，可以看到，此方法会遍历所有子元素并调用setChildFrame方法来为子元素指定对应的位置，其中childTop会逐渐变大，这就意味着后面的子元素会被放置在靠下的位置，这刚好符合树立方向的线性布局，至于setChildFrame，他仅仅是调用元素的layout方法而已，这样的父容器在layout方法中完成自己的定位以后，就通过onLayout方法去调用，子元素又会通过自己的layout方法来确定自己的位置，这样一层一层传递下去完成整个View树的layout过程，setChildFrame方法可以看:

|  |
| --- |
| private void setChildFrame(View child, int left, int top, int width, int height) {  child.layout(left, top, left + width, top + height);  } |

我们注意到setChildFrame中的width和height实际上就是子元素测量宽高，从下面的代码可以看出

|  |
| --- |
| final int childHeightMeasureSpec = MeasureSpec.makeMeasureSpec(Math.max(0, childHeight), MeasureSpec.EXACTLY);  final int childWidthMeasureSpec = getChildMeasureSpec(widthMeasureSpec,mPaddingLeft + mPaddingRight + lp.leftMargin + lp.rightMargin,lp.width);  child.measure(childWidthMeasureSpec, childHeightMeasureSpec); |

而在Layout方法中通过setFrame去设置子元素的四个顶点位置，方法中有这么几句:

|  |
| --- |
| mLeft = left;  mTop = top;  mRight = right;  mBottom = bottom; |

下面我们来回到之前的问题，View的测量宽高和最终宽高有什么区别，这个问题现在可以具体回答了，View的getMeasureWidth和getWidth这两个方法有什么区别？至于getMeasureHeight和getHeight是完全一样的，为了回答这个问题我们首先来看下getWidth和getHeight具体实现

|  |
| --- |
| public final int getWidth() {  return mRight - mLeft;  }  public final int getHeight() {  return mBottom - mTop;  } |

从getWidth和getHeight的源码在结合这四个变量的赋值来看，getWidth返回的刚好是View测量的值，而getHeight也是一样的，所以我们可用回答上面的问题了：在View的默认实现中，View的测量宽高和最终的是一样的，只不过一个是measure过程，一个是layout过程，而最终形成的是layout过程，即两者的赋值时机不同，测量宽高的赋值时机，稍微早一些，因此，在日常开发中，我们可用认为他们是相等的，但是还是有些不相同的，我们可用重写View的layout方法

|  |
| --- |
| public void layout(int l,int t,int r, int b){  super.layout(l,t,r+100,b+100);  } |

上述代码会导致在任何平台下View的最终宽高总是比测量大于100px,虽然这样这样会导致View显示不正常和没什么意义，但是这证明了测量 不等于 最终，**另一种情况是在某种情况下，View需要多次measure才能确定自己的测量宽高，在前几次的测量过程中，其得出的测量宽高是不一致的但最终是一致的**

### draw过程

Drawable是比較简单的，他的作用是将View绘制到屏幕上面，View的绘制过程由如下几个步骤：

1.绘制背景

2.绘制自己

3.绘制children

4.绘制装饰

这一点我们看他的源码就知道了

从setwilINotDraw这个方法的注释中可以看出，如果一个View不需要绘制任何内容,那么设置这个标记位为true以后，系统会进行相应的优化。默认情况下，View没有启用这个校化标记位，但是ViewGroup会默认启用这个优化标记位。这个标记位对实际开发的意义是。当我们的自定义控件继承于ViewGroup并且本身不具备绘制功能时，就可以开启这个标记位从而便于系统进行后续的优化。当然，当明确知道一个ViewGroup需要通过onDraw来绘制内容时，我们需要显式地关闭WILL\_NOT\_DRAW这个标记位。

## 自定义View

### 自定义View的分类

从特性大致的分为4类

1. 继承View重写onDraw方法: 重写了绘制，一般就是想自己实现某些图形了，因为原生控件已经满足不了你了，很显然这需要绘制的方式来完成，采用这个方式需要自身支=warp\_content,并且pading也要自己处理，比较考验你的功底了
2. 继承ViewGroup派生出来的Layout: 这个相当于重写容器了，当某些效果看起来像是View的组合的时候，就是他上场的时候了，不过这个很复杂，需要合理的使用测量和布局这两个过程，还要兼顾子元素的这两个过程
3. 继承特定的View: 比如TextView，就是重写原生的View嘛，比如你想让TextView默认有颜色之类的，有一些小改动，这个就可以用它的，他相对来说比较简单，这个就不需要自己支持包裹内容和pading了
4. 继承特定的ViewGroup: 这个和上述一样，只不过是重写容器而已，这个也比较常见，事件分发的时候用的也多

### 自定义View的须知

1. 让View支持warp\_content: 这个在之前将测量的时候说过，如果你不特殊处理一下是达不到满意的效果的，这里就不重复了
2. 如果有有必要，让你的View支持padding: 这是因为如果你不处理下的话，那么该属性是不会生效的，在ViewGroup也是一样
3. 尽量不要在View中使用Handler: 为什么不能用，是因为没有必要，View本身就有一系列的post方法，当然，你想用也没人拦着你，我倒是觉得handler写起来代码简洁很多
4. View中如果有线程或者动画，需要及时停止，参考View#onDetachedFromWindow: 你要是不停止这个线程或者动画，容易导致内存溢出的，所以你要在一个合适的机会销毁这些资源，在Activity有生命周期，而在View中，当View被remove的时候，onDetachedFromWindow会被调用，，和此方法对应的是onAttachedToWindow
5. View带有滑动嵌套时，需要处理好滑动冲突: 滑动冲突之前就BB过，这里就不讲了

### 自定义View的实例

#### 继承View重写onDraw方法

我们来实现一个很简单的图形：圆。尽管如此，还是有很多细节需要注意的，实现的过程中需要考虑warp\_content和padding，OK，自行分析代码的注释，我们先来看代码:

|  |
| --- |
| public class KCircleView extends View {  public static final String TAG = "KView";  private int mColor = Color.RED;  private Paint mPaint = new Paint(Paint.ANTI\_ALIAS\_FLAG);  public KCircleView(Context context) {  this(context, null, 0);  init();  }  public KCircleView(Context context, AttributeSet attrs) {  this(context, attrs, 0);  init();  }  public KCircleView(Context context, AttributeSet attrs, int defStyle) {  super(context, attrs, defStyle);  //自定义的属性，  // 在values目录下面创建自定义属性的xml，比如attrs.xml,也可以其他名字，  // <?xml version="1.0" encoding="utf-8"?>  /\* <resources>  <declare-styleable name="KCircleView">  <attr name="circle\_color" format="color" />  </declare-styleable>  </resources>\*/  final TypedArray a = context.obtainStyledAttributes(  attrs, R.styleable.KCircleView, defStyle, 0);  mColor = a.getColor(R.styleable.KCircleView\_k\_color, Color.RED);  init();  }  private void init() {  mPaint.setColor(mColor);  }  @Override  public void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {  super.onMeasure(widthMeasureSpec, heightMeasureSpec);  int widthMode = MeasureSpec.getMode(widthMeasureSpec);  int heightMode = MeasureSpec.getMode(heightMeasureSpec);  int w = widthMode == MeasureSpec.AT\_MOST ? MeasureSpec.getSize(widthMeasureSpec) : 50;  int h = heightMode == MeasureSpec.AT\_MOST ? MeasureSpec.getSize(heightMeasureSpec) : 100;  setMeasuredDimension(w, h);  }  @Override  public void onDraw(Canvas canvas) {  //处理android:padding  int w = getWidth() - getPaddingLeft() - getPaddingRight();  int h = getHeight() - getPaddingTop() - getPaddingEnd();  canvas.drawCircle(getPaddingLeft() + w / 2, getPaddingTop() + getHeight() / 2, Math.min(w / 2, h / 2), mPaint);  super.onDraw(canvas);  }  } |

#### 继承ViewGroup派生出来的Layout

这个同等于自定义布局，在之前介绍滑动的时候，有过类似的例子，主席就偷懒的搬上来了，当时分析滑动冲突的两种自定义View：HorizontalScrollViewEx和StickyLayout,其中HorizontalScrollViewEx就是通过继承ViewGroup来实现的，我们再次来分析他的测量和布局过程

这里BB一句，要规范的写View，需要一定的代价，这个，需要去看线性布局去了解了，他们的实现都很复杂，对于HorizontalScrollViewEx来说，就不这么精细了

回顾下HorizontalScrollViewEx的功能，他类似于ViewPager，或者说水平方向的线性布局，它内部的View可以竖直滑动，解决他的冲突的代码就不提了，我们主要还是看下他的测量

|  |
| --- |
| @Override  protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {  super.onMeasure(widthMeasureSpec, heightMeasureSpec);  int measureWidth = 0;  int measureHeight = 0;  final int childCount = getChildCount();  measureChildren(widthMeasureSpec,heightMeasureSpec);  int widthSpecMode = MeasureSpec.getMode(widthMeasureSpec);  int widthSpecSize = MeasureSpec.getMode(widthMeasureSpec);  int heightSpecMode = MeasureSpec.getMode(heightMeasureSpec);  int heightSpecSize = MeasureSpec.getMode(heightMeasureSpec);  if(childCount ==0){  setMeasuredDimension(0, 0);  }else if (widthSpecMode == MeasureSpec.AT\_MOST && heightSpecMode == MeasureSpec.AT\_MOST) {  final View childView = getChildAt(0);  measureWidth = childView.getMeasuredWidth() \* childCount;  measureHeight = childView.getMeasuredHeight();  setMeasuredDimension(measureWidth, measureHeight);  } else if (widthSpecMode == MeasureSpec.AT\_MOST) {  final View childView = getChildAt(0);  measureWidth = childView.getMeasuredWidth() \* childCount;  setMeasuredDimension(measureWidth, heightSpecSize);  } else if (heightSpecMode == MeasureSpec.AT\_MOST) {  final View childView = getChildAt(0);  measureHeight = childView.getMeasuredHeight();  setMeasuredDimension(widthSpecSize, measureHeight);  }  } |

这里发现一点小bug，不过不碍事，这里的逻辑呢，可以这样理理，首先有没有子元素，没有就全部都是0，有的话再去判断是否是warp\_content,,如果是包裹内容，那这个控件的宽度就是所以的总和了，如果高度采用包裹内容，那这个控件就是第一个子元素的高度，这样说应该好理解一点

再回来说说规范性，上面的代码可以说有两点吧，首先，是不应该直接设置为0，还有就是测量的时候没有考虑到padding和子元素的maggin，好的我们继续来看下onLayout

|  |
| --- |
| @Override  protected void onLayout(boolean changed, int l, int t, int r, int b) {  int childLeft = 0;  final int childCount = getChildCount();  mChildSize = childCount;  for (int i = 0; i < childCount; i++) {  final View childView = getChildAt(i);  final int childWidth = childView.getMeasuredWidth();  mChildWidth = childWidth;  childView.layout(childLeft, 0, childLeft + childWidth, childView.getMeasuredHeight());  childLeft += childWidth;  }  } |

# 理解RemoteViews

RemoteViews可以理解为一种远程的View,其实他和远程的Service是一样的，RemoteViews表示的是一种View的结构，他可以在其他的进程中显示，最常用的就是通知栏和桌面小组件了

## RemoteViews的应用

RemoteViews在实际的开发中，就是通知栏和桌面小组件了，这个大家应该都不陌生，主要是通过NotificationManager的notify方法去实现通知栏，也有通过AppwidthProvider来实现桌面小部件的，其实小部件本质上就是一个广播，他们两个的更新都无法像Activity中直接更新View，这是因为两者都运行在其他大家进程中，确切来说是SystemService中，为了跨进程更新UI，RemoteViews提供了一系列的set方法，我们接下来就是来实际的演示了。

### RemoteViews在通知栏上的应用

首先来看下通知栏，我们先了解一下系统默认的样式

|  |
| --- |
| //自定义通知栏  private void viewStyleNotirfication() {  Notification notification = new Notification();  notification.icon = R.mipmap.ic\_launcher;  notification.tickerText = "Hello K";  notification.when = System.currentTimeMillis();  notification.flags = Notification.FLAG\_AUTO\_CANCEL;  Intent intent = new Intent(this, Ch5MainActivity.class);  PendingIntent pendingIntent = PendingIntent.getActivity(this, 0, intent, PendingIntent.FLAG\_UPDATE\_CURRENT);  //RemoteViews的使用也是非常的简单，只要提供当前应用的包名和布局文件的资源ID就可以创建  // 一个RemoteViews了，如何更新呢？这一点和View还是有很大的不同，  // RemoteViews更新的时候，无法直接访问里面的View,必须通过RemoteViews所提供的一系列方法来更新，  // 比如设置TextView,那就需要  RemoteViews remoteViews = new RemoteViews(getPackageName(), R.layout.notification\_item);  remoteViews.setTextViewText(R.id.tv\_title, "Title");  remoteViews.setTextViewText(R.id.tv\_content, "Content");  remoteViews.setImageViewResource(R.id.iv\_img, R.mipmap.ic\_launcher);  remoteViews.setOnClickPendingIntent(R.id.ll\_open, pendingIntent);//待定的意图  notification.contentView = remoteViews;  NotificationManager notificationManager = (NotificationManager) getSystemService(NOTIFICATION\_SERVICE);  notificationManager.notify(2, notification);  } |

### RemoteViews 在桌面小部件的应用

AppWidgetProvider是Android提供给我们的用于实现桌面小部件的类，其本质也就是一个广播，，所以实际使用中把他看成一个广播即可，我们来看下怎么去具体的实现一个小部件

#### 定义小部件的界面

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?>  <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  android:layout\_width="match\_parent"  android:layout\_height="match\_parent"  android:orientation="vertical">  <ImageView  android:id="@+id/iv1"  android:layout\_width="wrap\_content"  android:layout\_height="wrap\_content"  android:src="@mipmap/ic\_launcher" />  <ImageView  android:id="@+id/iv2"  android:layout\_width="wrap\_content"  android:layout\_height="wrap\_content"  android:src="@mipmap/ic\_launcher" />  <TextView  android:id="@+id/tv\_test"  android:layout\_width="wrap\_content"  android:layout\_height="wrap\_content"  android:text="Test"  android:textSize="20dp" />  </LinearLayout> |

#### 定义小部件配置信息

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?>  <appwidget-provider xmlns:android="http://schemas.android.com/apk/res/android"  android:initialLayout="@layout/widget"  android:minHeight="84dp"  android:minWidth="84dp"  android:updatePeriodMillis="86400000">  </appwidget-provider> |

上面的几个参数的含义很明确，android:initialLayout就是加载布局其他两个就是最小的高宽，而updatePeriodMillis就是更新小组件的时间周期

#### 定义小组件的实现类

这个类需要继承AppWidgetProvider

|  |
| --- |
| public class KAppWidgetImpl extends AppWidgetProvider {  public static final String TAG = "KAppWidgetImpl";  public static final String ACTION\_CLICK = "k.art.ch5.click\_intent";  private Bitmap mBitmap;  private RemoteViews mRv;  public void onReceive(final Context context, Intent intent) {  **super.onReceive(context, intent);//必须父类处理**  String action = intent.getAction();  KLogUtil.D(TAG, "onReceive,action = " + action);  if (ACTION\_CLICK.equals(action)) {  new Thread(new Runnable() {  @Override  public void run() {  AppWidgetManager appWidgetManage = AppWidgetManager.getInstance(context);  for (int i = 0; i < 37; i++) {  RemoteViews rv = getRemoteViews(context);  rv.setImageViewBitmap(R.id.iv1, rotateBitmap((i \* 10) % 360));  appWidgetManage.updateAppWidget(new ComponentName(context, KAppWidgetImpl.class), rv);  SystemClock.sleep(30);  }  }  }).start();  }  }  /\*\*  \* 安装apk才调用  \*/  @Override  public void onUpdate(Context context, AppWidgetManager appWidgetManager, int[] appWidgetIds) {  KLogUtil.D(TAG, "onUpdate,num = " + appWidgetIds.length);  for (int it : appWidgetIds) {  appWidgetManager.updateAppWidget(it, getRemoteViews(context));  }  }  private RemoteViews getRemoteViews(Context cxt) {  if (mRv == null){  mRv = new RemoteViews(cxt.getPackageName(), R.layout.widget);  PendingIntent pi = PendingIntent.getBroadcast(cxt, 0, new Intent(ACTION\_CLICK), 0);  mRv.setOnClickPendingIntent(R.id.iv1, pi);  mBitmap = BitmapFactory.decodeResource(cxt.getResources(), R.mipmap.ic\_launcher);  }  return mRv;  }  //动画  private Bitmap rotateBitmap(int degree) {  Matrix matrix = new Matrix();  matrix.reset();  matrix.setRotate(degree);  Bitmap temBitmap = Bitmap.createBitmap(this.mBitmap, 0, 0, this.mBitmap.getWidth(), this.mBitmap.getHeight(), matrix, true);  return temBitmap;  }  } |

上述的代码实现了一个简单的桌面小部件，这里加了个动画，其他到没什么。我们需要注册才能使用

#### 在清单文件中声明小部件

|  |
| --- |
| <receiver android:name=".ch5Remote.KAppWidgetImpl">  <meta-data  android:name="android.appwidget.provider"  android:resource="@xml/appwidget\_info" />  <intent-filter>  <action android:name="k.art.ch5.click\_intent" />  <action android:name="android.appwidget.action.APPWIDGET\_UPDATE" />  </intent-filter>  </receiver> |

上面的代码有两个Action，其中第一个是识别小部件的动作，第二个就是他的标识，必须存在，**这是系统的规范，手动在桌面添加就可以看到自己添加的小部件了。**

AppWidgetProvider 除了最常用的onUpdate方法，还有其他的方法，onEnable，onDisabled,onDeleted以及onReceiver，这些方法都会被onReceiver在适当的时候调用，所以含义如下

onEnable:当该窗口小部件第一次添加到桌面的时候调用该方法，可添加多次但是只有第一次调用

onUpdate:小部件被添加或者第一次更新的时候都会调用一次该方法，小部件的更新机制由updatePeriodMillis来指定

onDeleted:每删除一次小部件都会调用

onDisabled:当最后一个该类型的小部件会删除时调用

onReceiver:广播内置的方法

关于AppWidgetProvider 的onReceiver方法的具体分发过程，可以看源码中的实现，如下图，通过下面的代码可以看出，onReceiver会根据不同的Action来分别调用这几个方法

上面描述了一个开发桌面小部件的完整过程 ，例子比较简单，实际开发过程中会稍微的复杂一些，但是开发流程都是一样的， 可以发现，桌面小程序在界面上的操作都是通过RemoteViews，不管小部件的界面初始化还是界面的更新都需要依赖他。

## PendingIntent概述

我们在前面多次提到了PendingIntent，归根结底，他和Intent有什么区别呢我们就来说下：顾名思义，PendingIntent表示处于一种等待的意图，即特定，等待的意思，就是说你要在某种条件下才会触发，所以我们很容易的就联想到RemoteViews了，

PendingIntent有三种待定的意图，就是Activity,Service，广播，使用方法如下：

PendingIntent.getActivity (Context, **requestCode**, **intent**, flags)=Context.startActivity(intent);

PendingIntent. getService(Context, requestCode, intent, flags)=Context.startService (intent);

PendingIntent.getBroadcast(Context, requestCode, intent, flags)=Context.sendBroadcast (intent);

这三个方法的参数都是一样的，主要理解的是第二个参数requstCode和第四个参数flags,code代表的是发送码，多数情况下为0，而且code会影响到flag，flag常见的有几种我们下面会说，其实最主要是理解匹配规则。

PendingIntent的匹配规则为：如果两个PendingIntent他们内部的**Intent相同并且requstCode**也相同的话，那么PendingIntent就是相同的，code比较好理解,那什么情况下Intent相同呢，Intent的匹配规则是：如果两个Intent的ComponentName的匹配过程，只要Intent之间的**ComponentName和intent-filter**相同,那么这两个intent就相同，需要注意的是Extras不参与匹配过程，只要intent之间的name和intent-filter相同就行，我们再来说下flags的参数含义。

FLAG\_ONE\_SHOT：当前描述的PendingIntent只能被使用一次,然后他就会被cancel,如果后续还有相同的PendingIntent，那么他的send方法就会失败，对于通知栏的消息来说，如果采用此标记位，那么同类的通知只能使用一次，后续将无法打开

FLAG\_NO\_CREATE：当前描述的PendingIntent不会主动去创建，如果当前PendingIntent之前不存在，那么getActivity等方法都会直接返回null,即获取PendingIntent失败，这个标记位很少见，他无法单独使用，因此在日常开发当中，并没有太多的意义，这里就不过多的介绍了

FLAG\_CANCEL\_CURRENT：当前描述的PendingIntent如果已经存在，那么就会被cancel,然后系统创建一个新的PendingIntent，对于通知栏来说，那些被cancel的消息将无法被打开

**FLAG\_UPDATE\_CURRENT**：当前描述的PendingIntent如果已经存在的话，那么他们就会被更新，他们的intent中的extras会被替换成新的。

从上面的分析看还是不太好理解这四个标记位，下面结合实际的项目来，这里分析两种情况，如下代码 **notificationManager.notify(id, notification);**

**如果notify方法的id是常量**，那么不管PendingIntent是否匹配，后面的通知会替换前面的通知，这个很好理解

**如果id不同&&PendingIntent不匹配**，在这种情况下不管采用了何种标记位，这些通知之间不互相干扰。

如果**id不同&&PendingIntent匹配**：如果采用FLAG\_ONE\_SHOT标记位，那么后续通知中，PendingIntent会和第一条通知保持一致，包括其中的Extras,单击任何一条通知，剩余的都无法打开当所有的通知被清除后，会重复这个过程，如果采用FLAG\_CANCEL\_CURRENT标记位，那么只有最新的通知可以打开，之前弹出的所有通知均无法打开，如果采用**FLAG\_UPDATE\_CURRENT**（升级就应该这么搞！！）标记位。那么之前弹出的通知PendingIntent会被更新，最终他们和最新的一条通知保持一致，包括其中的Extras,那么这些通知都可以被打开。

## RemoteViews内部机制

RemoteViews的作用在其他进程中显示并且更新View的界面，为了更好的理解他的内部机制，我们来看一下他的主要功能，首先我们看一下他的构造方法，这里介绍一个最常用的构造方法：

|  |
| --- |
| public RemoteViews(String packageName, int layoutId) {  this(getApplicationInfo(packageName, UserHandle.myUserId()), layoutId);  } |

他接受两个参数，第一个表示当前的包名，第二个是加载的布局，这个很好理解，RemoteViews目前并不能支持所有的View类型，我们看下他支持哪些：

Layout:FrameLayout LinearLayout RelativeLayout GridLayout

View:AnalogClock,Button,Chronometer,ImageButton,ImageView,ProgressBar,TextView,ViewFlipper ListView,GridView,stackView,AdapterViewFlipper,ViewStub

不支持：**~~EditView！~~**

上面的描述是RemoteViews所支持的View类，RemoteViews不支持他们的子类以及其他View的类型，也就是说RemoteViews中不能使用除了上述列表中以外的View,那么通知栏消息无法将弹出并且会抛出异常。

RemoteViews也没有提供findViewById方法，因此无法直接访问里面的View元素，而必须通过**RemoteViews所提供的一系列set方法来完成**，当然这是因为RemoteViews在远程进程中显示，所以没办法直接findViewById，关于set方法，可以看下这表



从这张表可以看出，原本可以直接调用View的方法，现在都需要通过set来完成，而从这些方法的声明来看，很像是通过反射来完成的，事实上也是如此。

### 通信原理

下面我们继续来说下RemoteViews的内部机制，由于RemoteViews主要用于通知栏和通知栏和桌面小部件，这里说一下他的工作过程，我们知道小部件分别由NotificationManager和AppWidgetProvider管理，而NotificationManager和AppWidgetProvider分别通过**Binder**为SystemService进程中的NotificationManagerService和AppWidgetService通信，由此可见，通知栏和小部件的布局文件实际上是这两个中加载出来的，而他们运行在系统的Server中，这就是我们的进程构成了跨进程通信的原理。

首先RemoteViews会通过Binder传递到SystemServer进程，这是因为RemoteViews实现了Parcelable接口，因此它可以跨进程传输，系统会根据RemoteViews中的包名等信息去得到该应用的资源。然后会通过Layoutinflater去加载RemoteViews中的布局文件，在SystemServer进程中加载后的布局文件是一个普通的View，只不过相对于我们的进程他是一个RemoteViews而已。接着系统会对View执行一系列界面更新任务，这些任务就是之前我们通过set方法来提交的。set方法对View所做的更新并不是立刻执行的，在RemoteViews内部会记录所有的更新操作，具体的执行时机要等到RemoteViews被加载以后才能执行，这样RemoteViews就可以在SystemServer进程中显示了，这就是我们所看到的通知栏消息或者桌面小部件。当需要更新RemoteViews时，我们需要调用一系列set法并通过NotificationManager和AppWidgetManager来提交更新任务，具体的更新操作也是在SystemServer进程中完成的。

从理论上来说，系统完全可以通过Binder去支持所有的View和View操作，但是这样做的话代价太大，因为View的方法太多了，另外就是**大量的IPC操作会影响效率**。为了解决这个问题，系统并没有通过Binder去直接支持View的跨进程访问，而是提供了一个Action，Action代表一个View操作，Action同样实现了Parcelable接口。系统首先将Vew操作封装装到Action对象并将这些对象跨进程传输到远程进程，接着在远程进程中执行Action对象中对象，当我们通过NotificationManager和AppWidgetManager来提交我们的更新时，这些Action对象就会传输到远程进程并在远程进程中依次执行，这个过程可以参看下面的图片，远程进程通过RemoteViews的apply方法来进行View的更新操作，RemoteViews的apply方法内部则回去遍历所有的Action对象并调用它们的apply方法，具体的View更新操作是由Acton对apply方法来完成的。上述做法的好处是显而易见的，首先不需要定义大量的Binder,其次通过在远程进程中批量执行RemoteViews的修改操作从而避免了大量的IPC视作这就提高了程序的性能，由此可见，Android系统在这方面的设计的确很精妙。



### 源码分析

上面从理论上分析了RemoteViews的内部机制，接下来我们从源码的角度分析下Remoteviews的工作流程。它的构造方法就不用多说了。

#### View更新接口-apply

这里我们首先看他一系列的set方法，比如setTextViewText。

|  |
| --- |
| public void setTextViewText(int viewId, CharSequence text) {  setCharSequence(viewId, "setText", text);  } |

上述ID中，viewId是被操作的View的id，setText是一个方法名，text是给TextView要设置的文本，可以联想一下，是不是清晰了很多，我们再来看下setCharSequence方法。

|  |
| --- |
| public void setCharSequence(int viewId, String methodName, CharSequence value) {  addAction(new ReflectionAction(viewId, methodName, ReflectionAction.CHAR\_SEQUENCE, value));  } |

从setCharSequence的实现来看，他的内部并没有对View进行直接的处理，然是添加了一个ReflectionAction独享，从名字来看，这应该是一个反射的动作，再看addAction的实现



从上述代码可以得知RemoteViews内部有一个mActions 成员，他是一个ArrayList，外界每一次调用一个set方法，他都能保存下来并未对view进行实际的操作，这一点从上面的理论分析就可以知道，我们继续看，我们现在看一下apply的实现以及Action类的实现



从上面的代码可以看出，首先会通过LayoutInflater 去加载RemoteViews中的布局文件，RemoteViews中的布局文件可以通过getLayoutId这个方法获得，加载完布局之后会通过performApply去执行一些更新操作，代码如下



performApply的实现是比较好理解的，他的作用就是遍历mActions 这个列表并执行每一个Action对象的apply方法，还记得mAction吗？每一次的set操作都会对应他里面的一个Action对象，因此也可以断定，Action对象的apply方法就是真正操作View的地方，实际上也是如此。

#### Apply调用时机

RemoteViews在通知栏和桌面小部件中的工作过程和上面描述的过程是一致的，当我们调用RemoteViews的set方法时，并不会立刻更新它们的界面，而必须要通过NotificationManager的**notify**方法以及AppWidgetManager的**updateAppWidget**才能更新它们的界面。实际上在AppWidgetManager的 updateAppWidget的内部实现中，它们的确是通过RemoteViews的apply以及reapply方法来加载或者更新界面的，apply和reApply的区别在于：**apply会加载布局并更新界面**，而reApply则只会更新界面。通知栏和桌面小插件在初始化界面时会调用apply方法，而在后续的更新界面时则会调用reapply方法。这里先看一下 BaseStatusBar的updateNotificationViews方法中，如下所示。

|  |
| --- |
| private void updateNotificationViews(NotificationData.Entry entry,  StatusBarNotification notification, boolean isHeadsUp) {  final RemoteViews contentView = notification.getNotification().contentView;  final RemoteViews bigContentView = isHeadsUp  ? notification.getNotification().headsUpContentView  : notification.getNotification().bigContentView;  final Notification publicVersion = notification.getNotification().publicVersion;  final RemoteViews publicContentView = publicVersion != null ? publicVersion.contentView  : null;  // Reapply the RemoteViews  contentView.reapply(mContext, entry.expanded, mOnClickHandler);  if (bigContentView != null && entry.getBigContentView() != null) {  bigContentView.**reapply**(mContext, entry.getBigContentView(),  mOnClickHandler);  }  if (publicContentView != null && entry.getPublicContentView() != null) {  publicContentView.**reapply**(mContext, entry.getPublicContentView(), mOnClickHandler);  }  // update the contentIntent  final PendingIntent contentIntent = notification.getNotification().contentIntent;  if (contentIntent != null) {  final View.OnClickListener listener = makeClicker(contentIntent, notification.getKey(),  isHeadsUp);  entry.row.setOnClickListener(listener);  } else {  entry.row.setOnClickListener(null);  }  entry.row.setStatusBarNotification(notification);  entry.row.notifyContentUpdated();  entry.row.resetHeight();  } |

很显然，上述代码表示当通知栏需要更新的时候，会通过RemoteViews的reapply方法来更新，接下来我们看一下AppWidgetHostView的updateAppWidget方法

|  |
| --- |
| public void updateAppWidget(int[] appWidgetIds, RemoteViews views) {  if (mService == null) {  return;  }  try {  mService.updateAppWidgetIds(mPackageName, appWidgetIds, views);  }  catch (RemoteException e) {  throw new RuntimeException("system server dead?", e);  }  } |

从上面的代码，我们是通过IAppWidgetService的接口去实现的，实际上最红也是通过RemoteViews的reapply方法来更新。

#### ReflectionAction实现

了解了apply和reapply的作用之后我们继续看Action的子类实现，首先看下ReflectionAction的具体实现，他的核心源码如下：



通过上述的代码可以看出，ReflectionAction 表示的是一个反射的动作，他通过对View的操作会反射的方式调用，其中getMethod就是根据方法名来反射所需要的方法，其中ReflectionAction 也有很多的set方法，除了ReflectionAction 还有Action，等，这里我们拿TextViewSizeAction来分析，具体实现如下：



他的实现是比较简单的，他之所以不用反射来实现，是因为setTextView这个方法有2个参数，因此无法复用ReflectionAction，因为ReflectionAction 的反射调用只能一个参数，这里就不一一分析了。

关于单击事件，RemoteView中只支持PendingIntent，是不支持其他模式的，我们这里需要注意的就是setOnClickPendingIntent，setPendingIntentTemplate，以及setonClickFillinIntent的区别，首先setOnClickPendingIntent用于给普通view设置单击事件，但是不能给ListView之类的View设置，如果需要，就用后两者

## RemoteViews的意义&实战

这一章，我们来真正意义上的理解他的意义，通过打造一个模拟的通知栏效果并且实现跨进程的UI更新。

### 同一app不同进程UI更新

首先我们两个Activity分别运行在了两个不同的进程，一个是A，一个是B，其中A扮演的是通知栏的角色，而B则可以不停地发送通知栏消息，当然这是模拟的消息。为了模拟通知栏的效果，我们修改A的process属性使其运行在单独的进程中，这样A和B就构成了多进程通信的情形。我们在B中创建Remoteviews对象，然后通知A显示这个RemoteViews对象。如何通知A显示B中的RemoteViews呢?我们可以像系统一样采用Binder来实现，但是这里为了简单起见就采用了广播。B每发送一次模拟通知，就会发送一个特定的广播，然后A接收到广播后就开始显示B中定义的RemoteViews对象，这个过程和系统的通知栏消息的显示过程几乎一致，或者说这里就是复制了通知栏的显示过程而已。

首先看B的实现，B只要构造RemoteViews对象并将其传输给A即可，这一过程通知栏是采用Binder实现的，但是本例中采用广播来实现，RemoteViews对象通过Intent传输A中，代码如下所示。A的代码比较简单只是接收一个广播就行。

发广播方（Ch5AActivity，对应书本B）

|  |
| --- |
| public class Ch5AActivity extends Activity {  private final String TAG = "Ch5AActivity";  public static final String ACTION = "K-ch5-send";  public static final String RV = "rv";  @Override  protected void onCreate(Bundle savedInstanceState) {  super.onCreate(savedInstanceState);  setContentView(R.layout.act\_ch5a);  findViewById(R.id.start\_brocast).setOnClickListener(new View.OnClickListener() {  @Override  public void onClick(View v) {  sendRemoteViewsBrocast();  }  });  findViewById(R.id.open\_b).setOnClickListener(new View.OnClickListener() {  @Override  public void onClick(View v) {  startActivity(new Intent(Ch5AActivity.this, Ch5BActivity.class));  }  });  }  private void sendRemoteViewsBrocast() {  RemoteViews rv = new RemoteViews(getPackageName(), R.layout.ch5\_a\_item);  rv.setTextViewText(R.id.tv\_title, "K，端午安康"+"msg from process:" + Process.myPid());  PendingIntent pi = PendingIntent.getActivity(this, 0, new Intent(this,Ch4MainActivity.class), PendingIntent.FLAG\_UPDATE\_CURRENT);  rv.setOnClickPendingIntent(R.id.iv\_img, pi);  Intent intent = new Intent(ACTION);  intent.putExtra(RV,rv);  sendBroadcast(intent);  } |

|  |
| --- |
| public class Ch5BActivity extends Activity {  private LinearLayout mRootView;  @Override  protected void onCreate(Bundle savedInstanceState) {  super.onCreate(savedInstanceState);  setContentView(R.layout.act\_ch5b);  registerReceiver(receiver,new IntentFilter(Ch5AActivity.ACTION));  mRootView = (LinearLayout) findViewById(R.id.act\_ch5b\_root);  }  public void onDestroy(){  super.onDestroy();  unregisterReceiver(receiver);  }  BroadcastReceiver receiver = new BroadcastReceiver() {  @Override  public void onReceive(Context context, Intent intent) {  KLogUtil.D("onReceive","intent="+intent);  RemoteViews rv = intent.getParcelableExtra(Ch5AActivity.RV);  View v = rv.apply(context, mRootView);  mRootView.addView(v);  }  };  } |

因为要模拟多进程运行：需要manifest中配置，运行时候，先通过点击按钮运行B，然后到home页，再次点击app图标，打开A，点击广播发送，然后在点击按钮进入B就能看到新的界面了。

|  |
| --- |
| <activity android:name=".ch5Remote.Ch5AActivity">  <intent-filter>  <action android:name="android.intent.action.MAIN" />  <category android:name="android.intent.category.LAUNCHER" />  </intent-filter>  </activity>  <activity android:name=".ch5Remote.Ch5BActivity"  **android:launchMode="singleInstance"**  **android:process=":remote"**>  </activity> |



上述代码很简单，除了注册和解除广播以外，最主要的逻辑其实就是updateUI,当A收到广播后，会从Intent中取出RemoteViews对象，然后通过apply方法加载布局并且执行更新操作，最后将得到的View添加到A的布局中即可。可以发现这个过程很简单，但是通知栏的底层是如何实现的呢？

本节这个例子是可以在实际中使用的，比如现在有两个应用，一个应用需要能够事更新另一个应用中的某个界面，这个时候我们当然可以选择AIDL去实现，但是如果对界面的更新比较频繁，这个时候就会有效率问题，同时AIDL接口就有可能会变得很复杂。这个时间如果采用RemoteViews来实现就没有这个问题了，当然RemoteViews也有缺点，那就是他只支持一些常见的View，对于自定义View它是不支持的。面对这种问题，到底是采用AIDL还是采用RemoteViews，这个要看具体情况，如果界面中的View都是一些简单的且被RemoteViews支持的View，那么可以考虑采用RemoteViews，否则就不适合用RemoteViews 了。

### 两个应用之间的界面更新

如果打算采用RemoteViews来实现两个应用之间的界面更新，那么这里还有一个问题，那就是布局文件的加载问题。在上面的代码中，我们直接通过RemoteViews的的apply方法来加载并更新界面，如下所示。

|  |
| --- |
| View view = remoteViews.apply(this, mLinearLayout);  mLinearLayout.addView(view); |

这种写法在同一个应用的多进程情形下是适用的，但是如果A和B属于不同应用，那么B中的布局文件的资源id传输到A中以后很有可能是无效的，因为A中的这个布局文件的资源id不可能刚好和B中的资源id一样，面对这种情况，我们就要适当的修改Remoteviews的显示过程的代码了。这里给出一种方法，既然资源不相同，那我们就通过**资源名称**来加载布局文件。首先两个应用要提前约定好RemoteViews中的布局的文件名称，比如“layout simulated notification”，然后在A中根据名称找到并加载，接着再调用Remoteviews 的的reapply方法即可将B中对View所做的一系列更新操作加载到View上了，关于apply 和reapply方法的差别在前面说了，这样历程就OK了

# Android的Drawable

这本书的涉及面真的很抓重点，这章说的是Drawable,Drawable表示的是一种可以在Canvas上进行绘制的抽象概念，Drawable可不光是指图片，其实他的种类也特别繁多，在实际开发当中，他使用简单，而且对大小，效率都有不错的效果，所以是我们不可或缺的一个知识点，我们接下来就来聊聊Drawable给我们带来了哪些好处！

## Drawable的简介

在Android中，Drawable可不是单单的图片那么简单，他说直观点可以理解为一种图片的抽象概念，通过颜色也可以定义出各式各样的图片，在实际开发中，Drawable常常被作为一个View的背景，一般分XML和图片两种，当然，我们也可以用图片来实现，不过这种方式就比较复杂了，Drawable作为一个抽象类，他衍生出了很多的子类，我们可以看下他的结构图.



基本每个标签都对应于一个真正的实体类，关系如下



Drawable的内部宽高参数很重要，通过getIntrinsicWidth和getIntrinsicHeight这两个方法可以获取到他们，但是比并不是所有的Drawable都有内部宽高，比如一张图片所形成的Drawable，他就有，但是如果你是颜色所形成的的，那就自然是没有的，而且要注意的是，内部宽高不等于他的大小，因为当View是背景，会被拉伸至View的等同大小。

## Drawable的分类

Drawable的种类是比较多的，比如我们之前常用的ShapeDrawable，BitmapDrawable，LayerDrawable，StateListDrawable等，注意的是，StateListDrawable是DrawableContainer的子类，接下来，我们继续看下这些种类繁多的Drawable到底有什么神奇的地方。

### BitmapDrawable

BitmapDrawable是比较简单的一张图片的Drawable，我们在实际开发中，可以直接设置为View的背景，也可以通过XML的形式来描述BitmapDrawable展示更多的效果，如下所示：

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?>  <bitmap xmlns:android="http://schemas.android.com/apk/res/android"  android:antialias="true"  android:dither="true"  android:filter="true"  android:gravity="center"  android:mipMap="true"  android:src="@color/colorPrimaryDark"  android:tileMode="clamp">  </bitmap> |

下面我们来看下他们的含义都是什么

android:src 这个是资源，可以是图片也可以是颜色

android:antialias是否开启图片**抗锯齿**，开启后图片会变得平滑一点，同时也会在一定程度上降低清晰度，不过这个降低我们完全可以无视，所以这个可以开启

android:dither是否开启**抖动**效果，当图片的像素配置和手机不一致的时候，开启这个选项可以让高质量的图片在低分辨率的屏幕上保持比较好的显示效果，比如图片的色彩模式ARGB8888，但是设备只支持RGB555，这个时候开启抖动模式可以让图片不会过于失真，在Android中创建的Bitmap一般会使用ARGB8888这个模式，即ARGB四个通道各占8位，在这个色彩下，一个像素所占为4个字节，一个像素的位数综合越高，图片越逼真，抖动也应该开启

android:filter是否开启**过滤**效果，当图片尺寸被拉伸或者压缩时，开启过滤效果会保持比较好的显示效果，所以这个也可开启

android:gravity这个就不用说，位置方向，上下左右，可以用“|”符号来实现左上，右下等效果，可以具体看下他的属性，如图



android:mipMap这是一种图片相关的处理技术，也叫纹理映射，比较抽象，默认为false，不常用

android:tileMode平铺模式，这个选项有几个值： disabled | clamp | repeat | mirror 其中disabled 是关闭平铺模式，这个也是默认值，开启后，gravity属性会无效，先说下其余三个属性的区别，三种都表示平铺模式

repeat表示简单的水平和竖直方向上平铺效果

mirror表示一种在水平和竖直方向上的镜面投影效果

clamp表示四周像素扩散效果

接下来说一下NinePatchDrawable,他说白了就是用代码实现的.9图片而已，我们看一下代码

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?>  <nine-patch xmlns:android="http://schemas.android.com/apk/res/android"  android:dither="true"  android:src="@drawable/ic\_launcher">  </nine-patch> |

这个就不过多介绍了，基本属性和BitmapDrawable一样，不过现在已经很少有人会用代码去实现了，这样没有图片使用的好。

### ShapeDrawable

ShapeDrawable是一种很常见的Drawable,可以理解为色彩构造的图片，他即是纯色的图形，也可以具有渐变的图形，不过语法要多很多，而且繁杂，我们来看代码的实现：

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?>  <shape xmlns:android="http://schemas.android.com/apk/res/android"  android:shape="line">  <corners  android:bottomLeftRadius="10dp"  android:bottomRightRadius="10dp"  android:radius="10dp"  android:topLeftRadius="10dp"  android:topRightRadius="10dp" />  <gradient  android:angle="10"  android:centerColor="@color/colorPrimary"  android:centerX="10"  android:centerY="10"  android:endColor="@color/colorPrimary"  android:gradientRadius="10dp"  android:startColor="@color/colorAccent"  android:type="linear"  android:useLevel="true" />  <padding  android:bottom="10dp"  android:left="10dp"  android:right="10dp"  android:top="10dp" />  <size  android:width="100dp"  android:height="100dp" />  <solid android:color="@color/colorAccent" />  <stroke  android:width="100dp"  android:color="@color/colorPrimaryDark"  android:dashGap="@dimen/activity\_horizontal\_margin"  android:dashWidth="10dp" />  </shape> |

需要注意的是，shape标签创建的Drawable,实际上是GradientDrawable，我们还是先来分析下他们的属性区别。

**android:shape**表示图片的形状，有四个选项，line（横线），oval(椭圆)，rectangle(矩形)，ring（圆环），他的默认值是矩形，而且line（横线）和ring（圆环）都必须通过stroke标签来指定宽高，颜色等信息，否则无法达到预期的效果

针对ring的形状，有五个特殊的属性，android:radius，android:tickness,android:innerRadiusRatio,android:ticknessRatio和android:userLevel，我们看图



**< corners>：**表示shape的四个角度，它只是用于矩形的shape，这里的角度是指圆角的成都，用px来表示，他有五个属性

android:radius：为四个角同时设置相同的角度，优先级较低，会被其他四个覆盖

android:topLeftRadius：设置最上角的角度

android:topRightRadius：设置右上角的角度

android:bottomLeftRadius：设置最下角的角度

android:bottomRightRadius：设置右下角的角度

**< gradient>:**他与solid标签是互相排斥的，因为solid表示纯色，而他表示渐变，他的属性如下

android:angle：渐变的角度，默认为0，其值必须为45的倍数，0表示从左往右，90表示从上到下，具体的效果自己体验

android:centerColor：渐变的中心颜色

android:centerX：渐变中心点的横坐标

android:centerY：渐变中心点的纵坐标

android:endColor：渐变的结束颜色

android:gradientRadius：渐变半径，只有当type = radial 时才有效

android:startColor：渐变的开始颜色

android:type：渐变的类型，

android:useLevel：false

**< solid>:**这个标签表示纯色填充，通过android:color来表示填充颜色

**< stroke>**shape的描边

android:width:宽度

android:color:颜色

android:dashwidth:虚线线段的宽度

android:dashGap:组成虚线的线段之间的间隔，间隔越大虚线看起来空隙越大

注意，如果android:dashGap和android:dashwidth有任何一个为0的话，那么虚线就不能生效了，我们来看下效果

**< pading>:**这个表示空白，但是他表示的不是shape的空白，而是包含他view的空白，而且有四个属性，左上右下

**< size>**shape的大小，有两个属性，width/height ，分别表示的是shape的宽高，也可以理解为shape的股友大小，但是一般来说，他并不是最终的大小，这个有点抽象，但是我们要明白，对于shape来说并没有宽高这个概念，作为view的背景他会适应view的宽高，size标签虽然是设置股友大小，但是还是会被拉伸。通过getIntrinsicWidth和getIntrinsicHeight能得到这个size的值。

### LayerDrawable

LayerDrawable对应的xml是< layer-list>,他可以理解为图层，通过不同的view达到叠加的效果

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?>  <layer-list xmlns:android="http://schemas.android.com/apk/res/android">  <item  android:id="@+id/test1"  android:bottom="5dp"  android:drawable="@drawable/ic\_launcher"  android:left="5dp"  android:right="5dp"  android:top="5dp" />  <item  android:id="@+id/test"  android:bottom="@dimen/activity\_horizontal\_margin"  android:drawable="@drawable/ic\_launcher"  android:left="@dimen/activity\_horizontal\_margin"  android:right="@dimen/activity\_horizontal\_margin"  android:top="@dimen/activity\_horizontal\_margin" />  </layer-list> |

属性很简单，一个layer-list包含多个item，形成叠加的图层效果，我们可以看一个例子



可以画一些复杂的图

### StateListDrawable

StateListDrawable对应的是< selector>标签，他会根据view的状态来选择出现的drawable，语法如下

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?>  <selector  xmlns:android="http://schemas.android.com/apk/res/android"  android:constantSize="true"  android:dither="true"  android:variablePadding="true">  <item  android:drawable="@drawable/ic\_launcher"  android:state\_pressed="true"  android:state\_focused="true"  android:state\_hovered="true"  android:state\_selected="true"  android:state\_checkable="true"  android:state\_checked="true"  android:state\_enabled="true"  android:state\_activated="true"  android:state\_window\_focused="true" />  </selector> |

我们来解释一下上面的属性的意思

**android:constantSize**：StateListDrawable的固有大小是不随其状态的改变发生改变的，因为状态的改变会导致他切换不同的drawable,而不同的drawable具有不同的drawable,true表示StateListDrawable的固有大小不变，这时他的固有大小就是内部所有drawable的固有大小的**最大值**，false则是跟随状态改变，默认false

**android:dither**是否开启抖动效果，这个在之前就已经提过，开启此选项可以让图片在低质量的屏幕上显示较好的效果，默认为true

**android:variablePadding** StateListDrawable的pading是跟随其状态发生改变的而改变，fasle为最大值，跟constantSize类似，**不建议开启，默认false**

**< item>**标签标示的是一个具体的Drawable,他的结构也比较简单，其中drawable标示已经资源的id，剩下的就是各种状态了，我们可以看下下面的图。



系统会根据View的状态从selector中选择对应的item，每一个对应着一个drawable,系统会按照从上到下来查。

### LevelListDrawable

LevelListDrawable对应着标签，同样表示一个drawable的集合。集合中的每一个Drawable都有一个等级，根据不同的等级切换不同的Item

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?>  <level-list xmlns:android="http://schemas.android.com/apk/res/android">  <item  android:drawable="@drawable/ic\_launcher"  android:maxLevel="10"  android:minLevel="1" />  </level-list> |

上面的语法中，每一个item代表一个drawable,并且对应着等级范围，由min和max来决定，我们来看一个实际的例子，，当他作为view的背景时，可以通过setLevenl来设置不同的drawable,等级的范围是0-10000

### TransitionDrawable

TransitionDrawable对应的是用于实现两个Drawable的淡入淡出，给出一个实例的例子

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?>  <transition xmlns:android="http://schemas.android.com/apk/res/android">  <item android:drawable="@color/colorAccent" />  <item android:drawable="@color/colorPrimary" />  </transition> |

接着将上面的设置为View的背景，最后通过startTransition和resetTransition来操作。

|  |
| --- |
| TransitionDrawable rawable = (TransitionDrawable) iv.getBackground();  drawable.startTransition(1000); |

### InsetDrawable

InsetDrawable对应的是,他可以将其他的Drawable内嵌到自己当中，并且可以在四周留下一定的距离，当一个View想他的背景比实际的距离小的时候就可以用，我们先来看下他的语法，看一个实例

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?>  <inset xmlns:android="http://schemas.android.com/apk/res/android"  android:insetBottom="10dp"  android:insetLeft="40dp"  android:insetRight="70dp"  android:insetTop="90dp">  <shape>  <solid android:color="@color/colorAccent" />  </shape>  </inset> |



### ScaleDrawable

ScaleDrawable对应的是,他可以指定自己的等级将指定的Drawable缩放带一定的比例，语法如下。放大镜这类的应用可以考虑？

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?>  <scale xmlns:android="http://schemas.android.com/apk/res/android"  android:drawable="@drawable/ic\_launcher"  android:scaleGravity="center"  android:scaleHeight="50%"  android:scaleWidth="50%">  </scale> |

在上面的属性中，scaleGravity的含义等同于shape中的gravity,而这个宽高就是缩放比例，以百分比的形式

ScaleDrawable有点费解，要理解他，我们就要先理解他的等级，0为不可见，要想可见，需要不为0，这个在源码中可以看出来，我们看他的draw方法，为0都不会绘制，我们再看一下他的onBoundsChange。

@Override

public void draw(Canvas canvas) {

final Drawable d = getDrawable();

if (d != null && d.getLevel() != 0) {

d.draw(canvas);

}

}



在他的onBoundsChange方法中，我们可以看出他的等级和大小，这里拿宽度来说：final int iw = min ? d.getIntrinsicWidth() : 0;

w -= (int) ((w - iw) \* (MAX\_LEVEL - level) \* mState.mScaleWidth / MAX\_LEVEL);

由于iw一般为0，所以上面的代码其实可以再简化一下 w -=(int)(w\*(10000-level)\*mState.mScaleWidth /10000)，可以发现，级别越大，规律越大，那么内部的Drawable就看起来越大了，我们看一个例子，缩小30%

### ClipDrawable

ClipDrawable对应的是，他可以根据自己的等级裁剪另一个Drawable,裁剪的方法可以通过android:clipOrientation,先看下语法

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?>  <clip xmlns:android="http://schemas.android.com/apk/res/android"  android:clipOrientation="vertical"  android:drawable="@drawable/ic\_launcher"  android:gravity="center">  </clip> |

其中clipOrientation标示裁剪方向，有水平和竖直，gravity比较复杂，需要和clipOrientation才有作用，我们来看下这张图表



网络图片加载，下面我们举个例子来实现一张图片从上往下进行裁剪的效果，那么我们定义一个ClipDrawable

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?>  <clip xmlns:android="http://schemas.android.com/apk/res/android"  android:clipOrientation="vertical"  android:drawable="@mipmap/ic\_launcher"  android:gravity="bottom">  </clip> |

clipDrawable = (ClipDrawable) iv.getBackground();

clipDrawable.setLevel(5000);

## 自定义Drawable

Drawbaled的使用范围很单一，要么作为ImageView在图像的演示，还有就是View的背景，核心就是draw方法，通常我们没有必要去自定义的，因为你无法在xml中使用，下面我们来演示自定义的过程。

|  |
| --- |
| public class KDrawable extends Drawable {  private Paint mPaint;  public KDrawable(int color) {  mPaint = new Paint(Paint.ANTI\_ALIAS\_FLAG);  mPaint.setColor(color);  }  @Override  public void draw(Canvas canvas) {  final Rect f = getBounds();  float cx = f.exactCenterX();  float cy = f.exactCenterY();  canvas.drawCircle(cx, cy, Math.min(cx, cy), mPaint);  }  @Override  public void setAlpha(int alpha) {  mPaint.setAlpha(alpha);  invalidateSelf();  }  @Override  public void setColorFilter(ColorFilter colorFilter) {  mPaint.setColorFilter(colorFilter);  invalidateSelf();  }  @Override  public int getOpacity() {  return PixelFormat.TRANSLUCENT;  }  } |

在上面的代码中，重写了需要实现的几个方法，其中draw是主要的方法，这个方法和View的draw方法类似。而其他三个就比较简单了，这个参靠下shapDrawable的实现就好了，所以说，看源码是很有必要的

# Android动画深入分析

Android的动画可以分成三种，view动画，帧动画，还有属性动画，其实帧动画也是属于view动画的一种，，只不过他和传统的平移之类的动画不太一样的是表现形式上有点不一样，view动画是通过对场景的不断图像交换而产生的动画效果，而帧动画就是播放一大段图片，很显然，图片多了会OOM，属性动画通过动态的改变对象的属性达到动画效果，也是api11的新特性，在低版本无法使用属性动画，但是我们依旧有一些兼容库，OK，我们还是继续来看下详细的类别

## View动画

View动画的作用是view，他支持四种动画，平移，缩放，旋转和透明，帧动画算是特殊的第五种了，我们接下来就一个个的去分析

### View动画的种类

数学含义应该是几何变化、图形变化：位置和像素值的变化

view动画的变换效果对应着Animation的四个子类，分别是TranslateAnimation,ScaleAnimation,RotateAnimation,AlphaAnimation,这四种动画可以通过xml来定义，也可以代码来实现，如图



要使用view动画，首先要创建XML文件，这个文件的路径为res/anim/filename.xml，他的描述语法是固定的，我们来看下:

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?>  <set xmlns:android="http://schemas.android.com/apk/res/android"  android:shareInterpolator="true">  <alpha  android:fromAlpha="10.0"  android:toAlpha="10.0" />  <scale  android:fromXScale="10dp"  android:fromYScale="10dp"  android:pivotX="10"  android:pivotY="10"  android:toXScale="10dp"  android:toYScale="10dp" />  <translate  android:fromXDelta="10"  android:fromYDelta="10"  android:toXDelta="10"  android:toYDelta="10" />  <rotate  android:fromDegrees="10"  android:pivotX="10"  android:pivotY="10"  android:toDegrees="10" />  </set> |

从上面的语法可以看出，view的动画既可以是单一的动画，也可以组合在一起，而set标签就是组合动画，对应着AnimationSet,他包含很多个类。我们来看下他的两个属性

**android:interpolator**表示动画集合所使用的插值器，插值器影响动画的速度，比如非匀速动画就需要插值器来制作动画的过程，这个属性可以不指定，默认为加速减速插值器，这个会在后面详细描述

**android:shareInterpolator**表示集合中的动画是否和集合共享同一个插值器，如果集合不指定插值器，那么子动画就需要单独的去指定所需要的插值器了

**< translate>**标签表示平移动画，对应的TranslateAnimation类，他可以使用一个view在水平和竖直方向完成平移，我们来看下他的属性

android:fromXDelta表示x的起始值，比如0

android:fromYDelta表示y的结束值，比如100

android:toXDelta表示x的起始值

android:toYDelta表示y的结束值

**< scale>**标签表示的是缩放动画，对应的ScaleAnimation，他可以使view具有放大，缩小的动画效果，他的一系列属性的含义如下

android:fromXScale水平方向缩放的起始值，比如0.5

android:fromYScale竖直方向缩放的起始值

android:pivotX缩放轴点的x坐标，它会影响缩放的效果

android:pivotY缩放轴点的y坐标，它会影响缩放的效果

android:toXScale水平方向缩放的结束值，比如1.2

android:toYScale竖直方向缩放的起始值

在< scale>中提到轴点的这个概念，我举个例子，默认情况下轴点事view的中心点，这个时候水平缩放的话会导致view向左右两个方向进行缩放，但是如果把轴点设置在view的右边界，那么view只会向左缩放，反之，xxx

**rotate**标签是旋转标签，对应RotateAnimation，他可以让view旋转，他的属性如下

android:fromDegrees旋转开始的角度，比如0

android:toDegrees旋转结束的角度，比如180

android:pivotX旋转轴点的x

android:pivotY旋转轴点的y

在旋转中也有轴的概念，他也会影响到旋转的效果，轴点扮演者旋转轴的角色，view围绕着轴点进行旋转，默认情况下在view的中心，考虑到一种情况，view围绕自己的中心，和围绕左上角进行90度显然是不同的轨迹

**< alpha>**表示透明动画。对应的AlphaAnimation，我们看下属性

android:fromAlpha表示透明度的起始值，比如0.1

android:toAlpha表示透明度的结束值，比如1

上面都只是很简单的介绍了XML格式，具体的使用方法还是看文档，我们还有一些常用的属性如下

**android:duration**动画的时间

**android:fillAfter**动画结束之后是否停留在结束的位置

下面我们再来一个实际的例子

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?>  <set xmlns:android="http://schemas.android.com/apk/res/android"  android:fillAfter="true"  android:zAdjustment="normal">  <translate  android:duration="2000"  android:fromXDelta="0"  android:fromYDelta="0"  android:toXDelta="100"  android:toYDelta="100" />  <rotate  android:duration="4000"  android:fromDegrees="0"  android:toDegrees="90" />  </set> |

|  |
| --- |
| private void test1() {  Animation animation = AnimationUtils.loadAnimation(this, R.anim.test1);  mTestAn1View.startAnimation(animation);  } |

除了在XML中定义之外，我们还可以通过代码来使用，如下

|  |
| --- |
| private void test2() {  AlphaAnimation animation = new AlphaAnimation(0,1);  animation.setDuration(4\*1000);  mTestAn1View.startAnimation(animation);  } |

我们还可以对动画进行监听，来看下代码

|  |
| --- |
| alpha.setAnimationListener(new Animation.AnimationListener() {  @Override  public void onAnimationStart(Animation animation) {  Log.i("TAG", "动画开始");  }  @Override  public void onAnimationEnd(Animation animation) {  Log.i("TAG", "动画结束");  }  @Override  public void onAnimationRepeat(Animation animation) {  Log.i("TAG", "重复动画");  }  }); |

### 自定义View动画

除了系统提供的四种动画以外，我们还可以自定义动画，自定义是一种简单又复杂的工作，说他简单，是因为派生出来只需要继承animation这个抽象类，这里也不打算去讲讲，直接看APIdemo里面3D效果的源码来分析:

|  |
| --- |
| public class Rotate3dAnimation extends Animation {  private final float mFromDegrees;  private final float mToDegrees;  private final float mCenterX;  private final float mCenterY;  private final float mDepthZ;  private final boolean mReverse;  private Camera mCamera;  /\*\*  \* Creates a new 3D rotation on the Y axis. The rotation is defined by its  \* start angle and its end angle. Both angles are in degrees. The rotation  \* is performed around a center point on the 2D space, definied by a pair  \* of X and Y coordinates, called centerX and centerY. When the animation  \* starts, a translation on the Z axis (depth) is performed. The length  \* of the translation can be specified, as well as whether the translation  \* should be reversed in time.  \*  \* @param fromDegrees the start angle of the 3D rotation  \* @param toDegrees the end angle of the 3D rotation  \* @param centerX the X center of the 3D rotation  \* @param centerY the Y center of the 3D rotation  \* @param reverse true if the translation should be reversed, false otherwise  \*/  public Rotate3dAnimation(float fromDegrees, float toDegrees,  float centerX, float centerY, float depthZ, boolean reverse) {  mFromDegrees = fromDegrees;  mToDegrees = toDegrees;  mCenterX = centerX;  mCenterY = centerY;  mDepthZ = depthZ;  mReverse = reverse;  }  @Override  public void initialize(int width, int height, int parentWidth, int parentHeight) {  super.initialize(width, height, parentWidth, parentHeight);  mCamera = new Camera();  }  @Override  protected void applyTransformation(float interpolatedTime, Transformation t) {  final float fromDegrees = mFromDegrees;  float degrees = fromDegrees + ((mToDegrees - fromDegrees) \* interpolatedTime);  final float centerX = mCenterX;  final float centerY = mCenterY;  final Camera camera = mCamera;  final Matrix matrix = t.getMatrix();  camera.save();  if (mReverse) {  camera.translate(0.0f, 0.0f, mDepthZ \* interpolatedTime);  } else {  camera.translate(0.0f, 0.0f, mDepthZ \* (1.0f - interpolatedTime));  }  camera.rotateY(degrees);  camera.getMatrix(matrix);  camera.restore();  matrix.preTranslate(-centerX, -centerY);  matrix.postTranslate(centerX, centerY);  }  } |

### 帧动画

帧动画就是顺序的播放一组图片，系统提供了一个AnimationDrawable来实现帧动画，帧动画的使用比较简单，我们看下在xml中如何定义

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?>  <animation-list xmlns:android="http://schemas.android.com/apk/res/android"  android:oneshot="false">  <item  android:drawable="@drawable/ic\_dashboard\_black\_24dp"  android:duration="200" />  <item  android:drawable="@drawable/ic\_dashboard\_black\_24dp"  android:duration="200" />  <item  android:drawable="@drawable/ic\_dashboard\_black\_24dp"  android:duration="200" />  </animation-list> |

然后将上述的XML作为view的背景并通过drawable来播放就可以了

|  |
| --- |
| iv\_icon.setBackgroundResource(R.drawable.animation\_test2);  AnimationDrawable ad = (AnimationDrawable) iv\_icon.getBackground();  ad.start(); |

帧动画比较简单，但是容易OOM，这个要注意

## View动画的特殊使用场景

动画的应用场景很多，我们来看下，比如activity的切换动画，又比如viewgroup的子view切换动画等，一起来看下吧

### LayoutAnimation

LayoutAnimation作用于**ViewGroup**，为ViewGroup指定一个动画，这样他的子元素出场的时候就会具有这种动画了，这种效果常常在listview上，我们时常会看到一种特殊的listview，他的每一个item都有一个动画，其实这并不是什么高深的技术，它使用的就是LayoutAnimation，我们来看下具体的实现。

#### 定义LayoutAnimation

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?>  <layoutAnimation xmlns:android="http://schemas.android.com/apk/res/android"  android:animation="@anim/animation"  android:animationOrder="normal"  android:delay="0.5">  </layoutAnimation> |

他们的含义分别是

**android:delay**表示子元素开始动画的延迟，假设子元素入场动画的周期为300ms，那么0.5表示每一个子元素都需要延迟150ms才能播放入场动画，总体来说，第一个子元素延迟150ms，第二个子元素300ms以此类推

**android:animationOrder**表示子元素动画的顺序，有三种模式，normal，random,reverse,其中第一个表示顺序执行，第二个表示随机，第三个表示倒叙执行

**android:animation**为子元素指定动画，如下

#### 为子元素指定的入场动画

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?>  <set xmlns:android="http://schemas.android.com/apk/res/android"  android:duration="300"  android:shareInterpolator="true">  <alpha  android:fromAlpha="0.1"  android:toAlpha="1.0" />  <translate  android:fromXDelta="500"  android:fromYDelta="0" />  </set> |

#### 为ViewGroup指定layoutanimation属性

对于listview来说，这样item就具有出场动画了

|  |
| --- |
| <ListView  android:id="@+id/mListView"  android:layout\_width="match\_parent"  android:layout\_height="match\_parent"  android:layoutAnimation="@anim/anim\_layout" /> |

除了在XML中实现，我们还可以通过LayoutAnimationController来实现

|  |
| --- |
| Animation animation = AnimationUtils.loadAnimation(this,R.anim.anim\_layout);  LayoutAnimationController controller = new LayoutAnimationController(animation);  controller.setDelay(0.5f);  controller.setOrder(LayoutAnimationController.ORDER\_NORMAL);  mListView.setLayoutAnimation(controller); |

### Activity的切换效果

activity有默认的切换效果，但是这个效果我们是可以自定义的，主要是利用overridePendingTransition(int enterAnim, int exitAnim)这个方法，这个方法必须在startactivity或者finish之后调用才是有效的，里面的两个参数也很简单，就是进出的动画，让我们启动一个activity的时候，我们可以这样做

|  |
| --- |
| startActivity(new Intent(MainActivity.this,OneActivity.class));  //这是activity的跳转动画  overridePendingTransition(R.anim.ch7\_animation\_enter,R.anim.ch7\_animation\_back); |

当activity退出的时候我们也可以为他指定一个切换效果

|  |
| --- |
| @Override  public void onBackPressed() {  super.onBackPressed();  overridePendingTransition(R.anim.ch7\_animation\_enter,R.anim.ch7\_animation\_back);  } |

使用它只需要在后面调用，这个也是注意的地方，其他地方调用无效.

在fragment中也是可以的，使用的方法是通过FragmentTransaction中的setCustomAnimations去使用.

## 属性动画

属性动画是API11加入的，和view动画不同的是，他对作用对象进行了扩展，属性动画可以对任何对象做动画，甚至没有对象也是可以的，除了作用对象进行了扩展以外，属性动画的效果也得到了加强，不再像view动画一样只支持四种，属性动画有ValueAnimator,ObjectAnimator,AnimatorSet;

### 使用属性动画

属性动画可以对任意**对象的属性**进行动画而不仅仅是view，动画默认的时间间隔是300ms，默认帧率是10，可以达到的效果是： 在一个时间间隔内完成对象从一个属性值到另一个属性值的改变。

为何叫属性动画，这里有个公式化的解释：ObjectAnimator.of$1(mView, "$2",value…); mView对mView的set$2($1 arg)方法执行动画，这里的方法改变的View的属性值。比如ObjectAnimator.ofFloat(iv\_icon, "translationY",50)，对iv\_icon的setTranslationY（float arg）方法执行平移动画，平移量为50。不用担心属性值填错，ide会自动纠错。属性动画要求对象的属性有set get方法。

但是属性动画是从API11才开始有的，所以这个比较有限制性。当然，网上还是有很多的就兼容库，这个我们就不多说，我们举几个例子：

(1)改变一个对象的translationY属性，让其沿着Y轴向上平移一个时间，该动画在默认的时间完成，我们来看下怎么去用,[demo](http://img.blog.csdn.net/20170422184616225?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvcXFfMjY3ODcxMTU=/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast):

ObjectAnimator.ofFloat(mOV,"translationY",50).start();

(2)改变一个对象的背景颜色值，典型的就是改变view的背景，下面的动画是让view的背景从0xffff8080到0xff8080ff,动画会无限循环和反转,[demo](http://img.blog.csdn.net/20170422184728352?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQv/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast)

|  |
| --- |
| ValueAnimator valueAnimator = ObjectAnimator.ofInt(mVV,"backgroundColor",0xFFFF8080, 0xFF8080FF);  valueAnimator.setDuration(3\*1000);  valueAnimator.setEvaluator(new ArgbEvaluator());  valueAnimator.setRepeatCount(ValueAnimator.INFINITE);  valueAnimator.setRepeatMode(ValueAnimator.REVERSE);  valueAnimator.start(); |

(3)动画集合，10s内对view的旋转，平移，缩放和透明度进行改变，注意此处是AnimatorSet,不是AnimationSet ,[demo](http://img.blog.csdn.net/20170422184811248?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvcXFfMjY3ODcxMTU=/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast)

|  |
| --- |
| AnimatorSet set = new AnimatorSet();  set.playTogether(  ObjectAnimator.ofFloat(mSV,"rotationX",0,360)  ,ObjectAnimator.ofFloat(mSV,"rotationY",0,90)  ,ObjectAnimator.ofFloat(mSV,"rotation",0,-90)  ,ObjectAnimator.ofFloat(mSV,"translationX",90)  ,ObjectAnimator.ofFloat(mSV,"scaleX",0,5f)  ,ObjectAnimator.ofFloat(mSV,"alpha",0,0.25f,1));  set.setDuration(10\*1000);  set.start(); |

属性动画还可以用XML来表示的，在res/animator目录下

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?>  <set xmlns:android="http://schemas.android.com/apk/res/android">  <objectAnimator  android:duration="1000"  android:propertyName="@string/app\_name"  android:repeatCount="infinite"  android:repeatMode="restart"  android:startOffset="15"  android:valueFrom="0.1dp"  android:valueTo="1.0"  android:valueType="colorType" />  <animator  android:duration="1000"  android:repeatCount="infinite"  android:repeatMode="restart"  android:startOffset="15"  android:valueFrom="0.5dp"  android:valueTo="1.0dp"  android:valueType="colorType" />  </set> |

属性动画的各个参数是比较好理解的，我们简单来说下他们之间的含义

android:propertyName:表示属性动画作用对象的属性的名称

android:duration：表示动画的时长

android:valueFrom：表示属性的起始值

android:valueTo：表示属性的结束值

android:startOffset：表示动画的延迟时间，当动画开始后，需要延迟多少毫秒才会真正的播放

android:repeatCount：表示动画的重复次数

android:repeatMode：表示动画的重复模式

android:valueType:表示propertyName有两个属性有int和float两个可选项，分别表示属性的类型，和浮点型，另外，如果所制定的是颜色类型，那么就不需要指定propertyName，系统会自动对颜色类型进行处理

对于一个动画来说，有两个属性这里要特别注意一下，一个是count另一个就是mode

下面我们给出一个具体的事例来分析，我们通过XML定义属性动画并且作用在view上

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?>  <set xmlns:android="http://schemas.android.com/apk/res/android">  <objectAnimator  android:duration="300"  android:propertyName="x"  android:valueTo="200"  android:valueType="intType" />  <objectAnimator  android:duration="300"  android:propertyName="y"  android:valueTo="300"  android:valueType="intType" />  </set> |

那我们该如何使用呢？

|  |
| --- |
| AnimatorSet sets = (AnimatorSet) AnimatorInflater.loadAnimator(MainActivity.this, R.animator.property\_animator);  sets.setTarget(iv\_icon);  sets.start(); |

在实际开发当中我还是建议使用**代码来实现属性动画**，这是因为用代码会比较简单，比如一个view需要从左边移动到右边，但是如果是XML，是不知道屏幕的宽高的

### 理解插值器和估值器

TimeInterpolator中文翻译是时间插值器的意思，他的作用是根据时间流逝的百分比来计算出当前属性值改变的百分比，系统预设的是LinearInterpolator(线性加速器，匀速加速器)，加速和减速插值器，TypeEvaluator的中文翻译是类型估值算法，也叫估值器，他的作用是根据当前属性变化的百分比来计算变化后的属性值，系统也预设了针对整型属性，浮点型，和color颜色值，定义的其实上是一个时刻-属性值的函数映射关系。

属性动画中的插值器和估值器很重要，他们实现非匀速动画的重要手段，可能说的有点苦涩，我们去通过实际的例子就能很好的理解了。如图，他表示的是一个匀速动画，采用了线性插值器和整形估值算法，在40ms内，x从0-40的变换。



由于动画的默认刷新率为10ms/帧，所有该动画将分5帧进行，我们来考虑一下第三帧，当时间为20ms的时候，百分比为0.5，意味着时间过去了一半，那x改变了多少？其实x也是0.5，为什么因为他使用了线性的插值器也就是匀速动画。

很显然，线性插值器的返回值和输入值是一样的，因为都是0.5，这个估算我们可以看他的源码



IntEvaluator.java中，对于我们的例子而言分别是0.5 0 40 ，所有返回给我们的是20，这就是（x= 20 t = 20ms）的由来。



插值器和估值器除了系统提供给我们的外，我们还可自定义，实现起来也很简单，因为他们都只是一个接口，而且内部只有一个方法。

### 属性动画的监听

属性动画提供了监听器用于监听动画的播放过程，主要有两个接口AnimationUpdateListener和AnimationListener

AnimationListener监听了开始，结束，取消已经重复。

AnimatorListenerAdapter监听整个动画过程。

### 对任意属性做动画

这里最一个需求，就是给buttion设置一个动画，让他的宽度从当前的变成500px，这个可以用view动画来搞定，但是你仔细想想，view不能对宽高变化，所有我么可以使用属性动画，我们来试试

ObjectAnimator.ofInt(mBtn,"width",300).setDuration(10\*1000).start();

不work，为何呢？我们 先看原理。

#### 属性动画的使用条件

属性动画要求动画作用在对象提供的get/set方法，属性动画根据外界传递的该属性的初始值和最终值，以动画效果多次去set，每次传递的set方法的值都不一样，确切来说是随着时间的时间推移，所传递的值越来越接近最终值，总结一下，我们对object的属性abc做动画，如果想让动画生效，要同时满足两个条件：

(1)object必须要提供set方法，如果动画的时候没有传递初始值，那么我们还要提供get方法，因为系统要去取abc的属性(如果这条不满意，程序直接Crash)

(2)object的set方法对abc所做的改变必须通过某种方法反应，比如带来UI的改变(如果这条不满足，动画无效果但是不会Crash)

以上条件缺一不可，那么为什么我们对button的width属性做动画没有效果，这是因为button内部虽然提供了get/set方法，但是这个set方法并不是改变视图大小，他是textview新添加的方法，view是没有这个setWidth方法的，由于button继承了textview，所有button也就有了set方法，下面看一下这个get/set的源码





从上述的源码可以看出，get的确是获取view的宽度，而set是textview的专属方法，他的作用不是设置view的宽度，而是设置textview的最大宽度和最小宽度，这个和textview的宽度不死一个东西，具体来说，textview的宽度对应XML中的android:layout\_width，而textview还有一个属性android:width，这个就对应了setwidth，总之textview和button的set/get干的不是同一件事，通过set无法改变控件的宽度，所以对width做属性动画没有效果，对于属性动画的两个条件来说，本例中的动画只满足了第一个条件，我们有三个解决办法：

* 给你的对象增加set/get方法，前提是你有权限的话
* 用这个类来包装原始对象，间接提供get/set方法
* 采用ValueAnimator，监听动画过程自己去实现

#### 解决办法

我们来具体的实现下这三个解决办法

**1.给你的对象增加set/get方法，前提是你有权限的话**

这个的意思很好理解，如果你有权限的话，加个set/get方法就搞定了，但是很多时候我们没有权限去这么做，比如本文开头所提到的问题，你无法给button加上一个合乎要求的setwidth方法，因为这个是Android SDK内部实现的，这个方法很简单，但是往往是不可行的，这里就不对其进行更多的分析了

**2.用这个类来包装原始对象，间接提供get/set方法**

这是一个很有用的解决方法，是笔者最喜欢用的，因为用起来很方便，也好理解，下面我将一个具体的实现来介绍，为了达到这个效果我们写了一个包装类去提供方法，这样也就完美的实现了

|  |
| --- |
| public class AnimViewWraper {  private View view;  public AnimViewWraper(View view) {  this.view = view;  }  public int getWidth() {  return view.getLayoutParams().width;  }  public void setWidth(int width) {  view.getLayoutParams().width = width;  view.requestLayout();  }  } |

ObjectAnimator.ofInt(new AnimViewWraper(mBtn),"width",300).setDuration(10\*1000).start();

**3.采用ValueAnimator，监听动画过程自己去实现**

首先说下什么是ValueAnimator，ValueAnimator本身不作用于任何对象，也就是说直接使用它没有任何的效果，他可以对一个值做动画，然后我们监听这个过程，在过程中修改我们对象的属性值，这样就相当于我们的对象做了动画，下面我们用例子来说明：

|  |
| --- |
| private void performAnimator(final View target, final int start, final int end) {  ValueAnimator valueAnimator = ValueAnimator.ofInt(1, 100);  valueAnimator.addUpdateListener(new ValueAnimator.AnimatorUpdateListener() {  //持有一个IntEvaluator对象，方便下面估值的时候使用  private IntEvaluator mEvaluator = new IntEvaluator();  @Override  public void onAnimationUpdate(ValueAnimator animation) {  //获得当前动画的进度值，整形1-100之间  int currentValue = (int) animation.getAnimatedValue();  //获得当前进度占整个动画之间的比例，浮点0-1之间  float fraction = animation.getAnimatedFraction();  //直接使用整形估值器，通过比例计算宽度，然后再设置给按钮  target.getLayoutParams().width = mEvaluator.evaluate(fraction, start, end);  target.requestLayout();  }  });  valueAnimator.setDuration(5000).start();  } |

上面的代码的效果和刚才的viewwrapper是一样的，关于ValueAnimator还要再说一下，拿上来的例子来说，他会在5s内将一个数1变成100，然后动画的每一帧会回调的每一帧onAnimationUpdate方法，在这个方法里，我们可以获取当前的值和占用的比例我们可以计算出宽度是多少，比如时间过去了一半，当前值是50，比例是0.5，假设起始值为100，最终是500px，那么500-100=400，所有这个时候乘以0.5=200，这些都是内部实现，我们不用自己写，直接用。

### 属性动画的源码分析

属性动画他要求作用的对象必须有set方法根据传递的最终值去不断的更改然后set，而且每次的值都不一样，就这样根据时间的推移形成动画，那我们具体来看下源码是怎么操作的，首先我们找一个入口，可以看他的start方法。  属性动画需要运行在Lopper线程中，上述代码终会调用AnimationHandler的start，这个并不是真正的handler，他是一个Runnable，看一下他的代码，通过代码我们发现，很快的调用了JNI层，不过JNI层最终还是调回来的，他的run方法被调用，这个Runnable涉及和底层的调用，我们来看下他的doAnimationFrame方法 }

当动画的下一帧到来的时候，PropertyValuesHolder中的setAnimatedValue方法将新的属性值设置给对象，调用其set方法，下面是源码，通过反射调用

在初始化的时候，如果属性的初始值没有提供，则get方法就会调用，请看PropertyValuesHolder中的setupValue



## 使用动画的注意事项

通过动画可以实现一些绚丽的效果，但是在使用过程中也发生了一些问题

1.OOM问题

这个问题主要还是帧动画中，当图片过多的时候就OOM了，这个在实际的开发中尤其注意，尽量避免使用帧动画

2.内存泄漏

在属性动画中有一类无限循环的动画，如果你在activity退出后不停止的话，可能就会存在这个问题了

3.兼容性问题

动画在3.0以下的系统上有缺陷，最好做好适配工作

4.View动画的问题

view动画死对view的影像做动画，并不是真正的改变view的状态，因此有时候会出现完成后view无法隐藏的现象，即setVisibility(View.GONE),这个时候只要调用clearAnimation清除动画即可

5.不要使用PX

在进行动画的过程，要尽量使用dp，使用px会导致适配问题

6.动画元素的交互

将view移动后，在3.0以前的系统，不管是view动画还是属性动画，新位置都无法调用单机事件，同时老位置却可以，从3.0之后，但是事件就是懂后的，但是view动画仍然在原位置。

7.硬件加速

使用动画的过程，建议开启硬件加速，这样会提高动画的流畅性

# 理解Window和WindowManager

对于重要的系统服务在手机开机的时候进行初始化, 然后保存在了系统进程中. 当我们应用启动的时候会触发ContextImpl类的加载, 在类加载的时候通过静态代码块对各个系统服务进行进行了注册,并保存到一个静态map容器中, 之后就可以通过getSystemService(serviceName)的形式获取不同的系统服务.

Window表示的是一个窗口的概念，在日常生活中使用的并不是很多，但是某些特殊的需求还是需要的，比如悬浮窗之类的，他的具体实现是PhoneWindow,创建一个Window很简单，只需要WindowManager去实现，WindowManager是外界访问Window的入口，Window的具体实现是在WindowManagerService中，他们两个的交互是一个IPC的过程，Android中的所有视图都是通过Windowl来实现的，无论是Activity,Dialog还是Toast,他们的视图都是直接附加在Window上的，因此Window是View的直接管理者，在之前的事件分发中我们说到，View的事件是通过WIndow传递给DecorView，然后DecorView传递给我们的View，就连Activity的setContentView,都是由Window传递的。

## Window和WindowManager

为了了解Window的工作机制，我们首先来看下如何通过WindowManager来创建一个Window.

|  |
| --- |
| Button btn = new Button(this);  btn.setText("我是窗口");  WindowManager wm = (WindowManager) getSystemService(WINDOW\_SERVICE);  WindowManager.LayoutParams layout = new WindowManager.LayoutParams(WindowManager.LayoutParams.WRAP\_CONTENT  , WindowManager.LayoutParams.WRAP\_CONTENT, 0, 0, PixelFormat.TRANSLUCENT);  layout.flags = WindowManager.LayoutParams.FLAG\_NOT\_TOUCH\_MODAL  | WindowManager.LayoutParams.FLAG\_NOT\_FOCUSABLE  | WindowManager.LayoutParams.FLAG\_SHOW\_WHEN\_LOCKED;  layout.gravity = Gravity.CENTER;  layout.type = WindowManager.LayoutParams.TYPE\_PHONE;  layout.x = 300;  layout.y = 100;  wm.addView(btn, layout); |

上述的代码，其中type和flag是比较重要的，我们来看下

**Flag参数表示window的属性**，他有很多选项，我们挑几个重点的

* FLAG\_NOT\_FOCUSABLE: 表示窗口不需要获取焦点，也不需要接收各种事件，这属性会同时启动FLAG\_NOT\_TOUCH\_MODAL，最终的事件会传递给下层的具体焦点的window.
* FLAG\_NOT\_TOUCH\_MODAL:在此模式下，系统会将当前window区域以外的单击事件传递给底层的Window，此前的Window区域以内的单机事件自己处理，这个标记很重要，一般来说都需要开启，否则其他window将无法获取单击事件
* FLAG\_SHOW\_WHEN\_LOCKED: 开启这个属性可以让window显示在锁屏上.

**Type参数表示window的类型**，window有三种类型，分别是应用，子，系统，应用window对应一个Activity,子Window不能单独存在，需要依赖一个父Window，比如常见的Dialog都是子Window,系统window需要声明权限，比如系统的状态栏.Window是分层的，每个Window对应着z-ordered,层级大的会覆盖在层级小的Window上面，这和HTML中的z-index的概念是一致的，在这三类中.

* 应用是层级范围是1-99.
* 子window的层级是1000-1999
* 系统的层级是2000-2999。

这些范围对应着type参数，如果想要window在最顶层，那么层级范围设置大一点就好了，很显然系统的值要大一些，系统的值很多，我们一般会选择TYPE\_SYSTEM\_OVERLAY和TYPE\_SYSTEM\_ERROR，记得要设置权限.

**<uses-permission android:name="android.permission.SYSTEM\_ALERT\_WINDOW"/>**

WindowManager所提供的功能很简单，常用的有三个方法，添加View,更新View,删除View,这三个方法定义在ViewManager中，而WindowManager继承自ViewManager:

|  |
| --- |
| public interface ViewManager {  public void addView(View view, ViewGroup.LayoutParams params);  public void updateViewLayout(View view, ViewGroup.LayoutParams params);  public void removeView(View view);  } |

虽然只有三个功能，但是这三个功能足够我们使用了，我们常见的可以推动的View，其实也很好实现，就是不断的更改他xy的位置.

|  |
| --- |
| button.setOnTouchListener(new View.OnTouchListener() {  @Override  public boolean onTouch(View v, MotionEvent event) {  int rawX = (int) event.getRawX();  int rawY = (int) event.getRawY();  switch (event.getAction()) {  case MotionEvent.ACTION\_MOVE:  layout.x = rawX;  layout.y = rawY;  wm.updateViewLayout(button, layout);  break;  }  return false;  }  }); |

## Window的内部机制

Window是一个抽象的概念，每一个Window对应着一个View和一个ViewRootImpl,Window和View通过ViewRootImpl建立关系，因此Window并不是实际存在的，这点从WindowManager定义的接口都是针对View，这说明View才是window的实体，在实际使用当中我们并不能直接访问Window的添加过程.

### Window的添加过程

Window的添加过程是通过WindowManager的addView去实现的，而真正实现的是一个接口，也就是WindowManagerImpl.

|  |
| --- |
| Activity.attach() {  mWindow = new PhoneWindow(this, window);  mWindow.setWindowManager(  (WindowManager)context.getSystemService(Context.WINDOW\_SERVICE));  **mWindowManager = mWindow.getWindowManager();**  }  Window.setWindowManager() {  mWindowManager = ((**WindowManagerImpl**)wm).createLocalWindowManager(this);  } |

|  |
| --- |
| @Override  public void addView(View view, ViewGroup.LayoutParams params) {  **mGlobal**.addView(view, params, mDisplay, mParentWindow);  }  @Override  public void updateViewLayout(View view, ViewGroup.LayoutParams params) {  mGlobal.updateViewLayout(view, params);  }  @Override  public void removeView(View view) {  mGlobal.removeView(view, false);  } |

可以发现，WindowManagerImpl并没有直接去实现一个Window的三大操作，而是全部交给了WindowManagerGlobal来处理，WindowManagerGlobal是一个工厂的性质提供自己的实现，在WindowManagerGlobal中有一段如下的代码：

private final WindowManagerGlobal mGlobal = WindowManagerGlobal.getInstance();

WindowManagerImpl这种工作模式就是典型的桥接模式，将所有的操作全部委托给WindowManagerGlobal（**@hide**这是系统隐藏类，普通app不可见！）去实现，WindowManagerGlobal的addView方法主要分如下几步：

|  |
| --- |
| public final class WindowManagerGlobal {  private final ArrayList<View> mViews = new ArrayList<View>();  private final ArrayList<ViewRootImpl> mRoots = new ArrayList<ViewRootImpl>();  private final ArrayList<WindowManager.LayoutParams> mParams =  new ArrayList<WindowManager.LayoutParams>();  public void addView(View view, ViewGroup.LayoutParams params,  Display display, Window parentWindow) {  ......  ViewRootImpl root;  View panelParentView = null;  synchronized (mLock) {  ......  **//1.创建ViewRootImpl**  root = new ViewRootImpl(view.getContext(), display);  view.setLayoutParams(wparams);  **//2.添加到列表中**  mViews.add(view);  mRoots.add(root);  mParams.add(wparams);  }  **//3. 最后执行这个，将View显示在手机屏幕上**  try {  root.setView(view, wparams, panelParentView);  } catch (RuntimeException e) {  ......  }  }  } |

#### 创建ViewRootImpl

**ViewRootImpl的作用**可以总结为: Android Framework与WMS之间的通信也是通过Binder机制进行的. 这里就建立了与WMS的通信. 最后通过openSession()函数来与WMS建立一个通信会话, 相当于建立了一个长期的处理中心, 双方有什么需要够可以通过这个Session来交换信息.

WMS是运行在Native层的, 而现在只是在Framewrok层. ViewRootImpl是通信的桥梁

上面的方法中先是new了一个ViewRootImpl，然后调用他的setView来显示布局。

Android中的View都是通过ViewRootImpl来完成绘制的。

|  |
| --- |
| ublic final class ViewRootImpl implements ViewParent,  View.AttachInfo.Callbacks, ThreadedRenderer.HardwareDrawCallbacks {  public ViewRootImpl(Context context, Display display) {  mContext = context;  //获取IWindowSession，与WindowManagerService建立连接  mWindowSession = **WindowManagerGlobal.getWindowSession();**  //这里保存当前线程  mThread = Thread.currentThread();  }  } |

继续追踪WindowManagerGlobal.getWindowSession()：

|  |
| --- |
| public static IWindowSession **getWindowSession**() {  synchronized (WindowManagerGlobal.class) {  if (sWindowSession == null) {  try {  InputMethodManager imm = InputMethodManager.getInstance();  //获取WindowManagerService  IWindowManager windowManager = **getWindowManagerService**();  //与系统的WindowManagerService建立一个IWindowSession  sWindowSession = windowManager.openSession(  new IWindowSessionCallback.Stub() {  @Override  public void onAnimatorScaleChanged(float scale) {  ValueAnimator.setDurationScale(scale);  }  },  imm.getClient(), imm.getInputContext());  } catch (RemoteException e) {  throw e.rethrowFromSystemServer();  }  }  return sWindowSession;  }  }  public static IWindowManager getWindowManagerService() {  synchronized (WindowManagerGlobal.class) {  if (sWindowManagerService == null) {  //这里返回的是IBinder对象，进行IPC通信  **sWindowManagerService = IWindowManager.Stub.asInterface(**  **ServiceManager.getService("window"));**  ......  }  return sWindowManagerService;  }  } |

继续看 sWindowManagerService = IWindowManager.Stub.asInterface(

ServiceManager.getService("window"));这行：

|  |
| --- |
| public final class ServiceManager {  public static IBinder getService(String name) {  try {  IBinder service = sCache.get(name);  if (service != null) {  return service;  } else {  return getIServiceManager().getService(name);  }  } catch (RemoteException e) {  Log.e(TAG, "error in getService", e);  }  return null;  }  } |

所以IWindowManager.Stub.asInterface(ServiceManager.getService("window"));得到的是一个IBinder对象。到这里，在ServiceManager的getService方法中通过getIServiceManager().getService("window")获取到一个IBinder，与WMS建立初步连接。然后通过IWindowManager.Stub.asInterface方法将IBinder转换成IWindowManager对象。通过这个对象调用openSession打开一个Session，实现通话。但是，WMS只负责管理View的z-order，也就是管理当前那个View在最上层显示，并不管理绘制。

#### ViewRootImpl.setView

绘制View的任务就在ViewRootImpl的setView方法中。

|  |
| --- |
| public void setView(View view, WindowManager.LayoutParams attrs, View panelParentView) {  synchronized (this) {  ......  //请求绘制View  requestLayout();  ......  int res; /\* = WindowManagerImpl.ADD\_OKAY; \*/  try {  ......  //请求WindowManagerService，让WMS实现Window的添加。  res = mWindowSession.addToDisplay(mWindow, mSeq, mWindowAttributes,  getHostVisibility(), mDisplay.getDisplayId(),  mAttachInfo.mContentInsets, mAttachInfo.mStableInsets,  mAttachInfo.mOutsets, mInputChannel);  }  ......  }  } |

setView主要做了两件事：

（1）requestLayout();

（2）向WMS请求添加Window；

#### requestLayout();

@Override

public void requestLayout() {

if (!mHandlingLayoutInLayoutRequest) {

checkThread();

mLayoutRequested = true;

scheduleTraversals();

}

}

**这里我另外注意一下checkThread()这个方法。**

void checkThread() {

if (mThread != Thread.currentThread()) {

throw new CalledFromWrongThreadException(

"Only the original thread that created a view hierarchy can touch its views.");

}

}

是不是看到了熟悉的异常提示？这个方法要判断当前更新UI 的线程是不是创建ViewRootImpl时的线程，只有在创建ViewRootImpl的线程中更新对应的UI才不会报错。所以不能在子线程中更新UI也是这个原因。

setView->requestLayout->scheduleTraversals—>mTraversalRunnable->doTraversal->performTraversals来通过一个handler把这个任务发送出去：完成异步刷新请求

看performTraversals();这个方法里将近一千行代码,大致是四个过程：

private void performTraversals(){

// 1 获取Surface对象，用于图形绘制

//2 测量整个视图树中各个View的大小，用performMeasure方法

//3 布局整个视图树，用performLayout方法

//4 绘制整个视图树，用performDraw方法

}

performDraw->ViewRootImpl.draw()方法->.ViewRootImpl.drawSoftware方法，调用GPU绘图。

draw()中获取到绘制表面Surface，里面最后调用ViewRootImpl的drawSoftware方法，调用GPU绘图。

private boolean drawSoftware(Surface surface, AttachInfo attachInfo, int xoff, int yoff,

boolean scalingRequired, Rect dirty) {

final Canvas canvas;

try {

//获取canvas

canvas = mSurface.lockCanvas(dirty);

// TODO: Do this in native

canvas.setDensity(mDensity);

} catch (Surface.OutOfResourcesException e) {

}

try {

if (!canvas.isOpaque() || yoff != 0 || xoff != 0) {

canvas.drawColor(0, PorterDuff.Mode.CLEAR);

}

try {

......

//从这里开始绘制整个视图树，从DecorView开始

mView.draw(canvas);

} finally {

......

}

} finally {

try {

//解锁canvas，并通知SurfaceFlinger更新这块区域。

surface.unlockCanvasAndPost(canvas);

} catch (IllegalArgumentException e) {

......

}

}

return true;

}

综上所述，视图树的绘制主要有以下步骤：

（1）判断使用CPU还是GPU绘制

（2）获取绘制表面Surface对象

（3）通过Surface对象获取并锁住绘图对象

（4）从DecorView开始绘制整个视图树

（5）解锁Canvas，并通知SurfaceFlinger更新这块区域。

**2.创建ViewRootImpl并将View添加到列表中**

在WindowManagerGlobal有如下几个列表是比较重要的.

|  |
| --- |
| private final ArrayList<View> mViews = new ArrayList<View>();  private final ArrayList<ViewRootImpl> mRoots = new ArrayList<ViewRootImpl>();  private final ArrayList<WindowManager.LayoutParams> mParams =  new ArrayList<WindowManager.LayoutParams>();  private final ArraySet<View> mDyingViews = new ArraySet<View>(); |

在上面的声明中，mViews存储所有window所对应的View，mRoots存储是所有window所对应的ViewRootImpl，mParams存储是所对应的布局参数，而mDyingViews则存储那些正在被删除的对象，在addView中通过如下方式将Window的一系列对象添加到列表中.

|  |
| --- |
| root = new ViewRootImpl(view.getContext(), display);  view.setLayoutParams(wparams);  mViews.add(view);  mRoots.add(root);  mParams.add(wparams); |

如此一来,Window的添加请求就交给WindowManagerService去处理了，在WindowManagerService内部为每一个应用添加了一个单独的Session,具体WIndow在WindowManagerService中怎么添加的，我们就不去分析了，读者可以自己去熟悉下.

### Window的删除过程

Window的删除过程和添加过程一样，都是通过Impl再通过Global来实现了，下面是Global的removeView.

|  |
| --- |
| public void removeView(View view) {  synchronized (this) {  mViews.remove(view);  for (int i = mViewRects.size(); i-- > 0; ) {  AttachInfo.InvalidateInfo info = mViewRects.get(i);  if (info.target == view) {  mViewRects.remove(i);  info.recycle();  }  }  if (mPosted && mViews.isEmpty() && mViewRects.isEmpty()) {  mChoreographer.removeCallbacks(Choreographer.CALLBACK\_ANIMATION, this, null);  mPosted = false;  }  } |

removeView的代码很清晰，首先通过findViewLocked来查找待删除的索引，这个超找过程就是建立的数组遍历，然后进一步的删除。

removeViewLocked是通过ViewRootImpl来完成删除操作的，在windowmanager中提供了两种接口removeView和removeViewImmediate，他们分别表示异步删除和同步删除，其中removeViewImmediate，使用起来要格外注意，一般来说不需要使用此方法来删除window以免发生意外的错误，这里主要是异步删除的问题，具体的删除操作是ViewImple的die方法来完成的，在异步删除的情况下，die只是发生一个删除的请求后就返回了，这个时候View并没有完成删除的操作，所有最后会将其添加到mDyingViews中，mDyingViews表示待删除的View列表，ViewRootImpe的die方法如下：

在die方法内部只是做了简单的判断，那么就发送了一个MSG\_DIE的消息，ViewRootImpl中的mHandler会处理此消息并且并调用doDie方法，如果是同步删除就会直接调用doDie方法，在doDie方法内部会操作dispatchDetachedFromWindow，真正删除window就是在这里面实现的，他主要做了四件事。

1.垃圾回收相关的工作，比如清除数据和消息，移除回调

2.通过Session的remove方法来删除window,这同样是一个IPC的过程，最终会调用wms的removeWindow方法

3.调用view的dispatchDetachedFromWindow方法，在内不会调用onDetachedFromWindow,他做了一些回收资源或者停止动画的一些操作

4.调用WindowManagerGlobal的doRemoveView方法刷新数据

### Window的更新过程

到这里，window的删除就接收完了，在说下更新，需要看WindowManagerGlobal的updateViewLayout方法。

|  |
| --- |
| public void updateViewLayout(View view, ViewGroup.LayoutParams params) {  final WindowManager.LayoutParams wparams = (WindowManager.LayoutParams)params;  view.setLayoutParams(wparams);  synchronized (mLock) {  int index = findViewLocked(view, true);  ViewRootImpl root = mRoots.get(index);  **mParams.remove(index);**  **mParams.add(index, wparams);**  root.setLayoutParams(wparams, false);  }  } |

updateViewLayout做的方法比较简单，首先他更新View的LayoutParams替换老的，接着再更新下ViewRootimpl中的LayoutParams，这一步是通过viewrootimpl的setLayoutParams来做的，在ViewRootImpl中会通过scheduleTraversals方法来对View，测量，布局，重绘等等，除了view本身的重绘之外，ViewRootImpl还会通过WindowSession来更新Window的视图，这个过程最终是WindowManagerService的relayoutWindow来实现的，具体也是一个IPC的过程

## Window的创建过程

通过上面的分析，我们知道，view是android中视图的呈现方式，但是view不能单独存在，他必须依附在window这个抽象类中，因此有视图的地方就有window，activity，toast都是，我们继续来分析window的创建过程。

### Activity的Window创建过程

要分析Activity的Window创建过程就需要去了解activity的启动过程，详细的会在后面说，这里简单概括，activity的启动过程很复杂，最终会由ActivityThread中的perfromLaunchActivity()来完成整个启动过程。

在Activity的attach方法中，系统会创建activity所属的window对象PhoneWindow并为其设置回调接口，window对象的创建过程是由PolicyManager的akeNewWindow方法实现的，由于activity实现了window的callback方法接口，因此当window接受到外界的状态改变的时候就会去调用activity的方法，callback接口中的方法很多，但是有几个确实我们非常熟悉的，你如onAttachedToWindow等。

下面分析activity的视图是怎么依附在window上的由于activity的视图是由setContentView开始的，所有我们先看下这个方法：

|  |
| --- |
| public void setContentView(@LayoutRes int layoutResID) {  getWindow().setContentView(layoutResID);  initWindowDecorActionBar();  } |

1.如果没有DecorView就去创建他

DecorView是一个FrameLayout,在之前就已经说过了，这里说一下。DecorView是activity中的顶级View,一般来说他的内部包含标题栏和内部栏，但是这个会随着主题的变化而发生改变的，不管怎么样，内容是一定要存在的，并且内容有固定的id，那就是content,完整的就是android.R.id.content，DecorView的创建是由installDecor方法来完成的，在方法内部会通过generateDecor方法来完成创建DecorView,这个时候他就是一个空白的FrameLayout.

|  |
| --- |
| protected DecorView generateDecor(){  return new DecorView(getContext(),-1);  } |

为了初始化DecorView的结构，PhoneWindow还需要通过generateLayout方法来加载具体的布局文件到DecorView中，这个跟主题有关.

|  |
| --- |
| View in = mLayoutInflater.inflate(layoutResource, null);  decor.addView(in, new ViewGroup.LayoutParams(MATCH\_PARENT, MATCH\_PARENT));  mContentRoot = (ViewGroup) in;  ViewGroup contentParent = (ViewGroup)findViewById(ID\_ANDROID\_CONTENT); |

其中ID\_ANDROID\_CONTENT的定义如下，这个id对应的就是ViewGroup的mContentParent

public static final int ID\_ANDROID\_CONTENT = com.android.internal.R.id.content

2.将View添加到DecorView的mContentParent中

这个过程比较简单，由于在第一步的时候已经初始化了DecorView，因此这一部就直接将activity的视图添加到DecorView的mContentParent中既可，mLayoutInflater.inflate(layoutResID, mContentParent);到此为止，由此可以理解activity的setcontentview的来历了，也许有读者会怀疑，为什么不叫setview来，他明明是给activity设置视图啊，从这里来看，他的确不适合叫做setview,因为activity的布局文件只是添加到了DecorView的mContentParent中，因此交setContentView更加准确。

3.回调Activity的onCreateChanged方法来通知Activity视图已经发生改变

这个过程很简单，由于window实现了Callback接口，这里表示布局已经被添加到DecorView的mContentParent中了，于是通知activity。使其可以做相应的处理Activity的onCreateChanged是一个空实现，我们可以在子activity处理这个回调

|  |
| --- |
| final callback cb = getCallback();  if(cb != null && !isDestroyed()){  cb.onContentChanged();  } |

经过了上面的三个步骤，到这里为止DecorView已经被创建并且初始化完毕了activity的布局文件也已经添加到了DecorView的内容中，但是这个时候DecorView还没有被windowmanager添加到window中，这里需要正确的理解window的概念，window更多的是表示一种抽象的功能集合，虽然说早在activity的attch中window就已经被创建了，但是这个时候由于DecorView还没有被windowmanager识别，所有还不能提供具体的功能，因为他还无法接收外界的输入，在activityThread的makeVisible中，才能被视图看到：

|  |
| --- |
| void makeVisible() {  if (!mWindowAdded) {  ViewManager wm = getWindowManager();  wm.addView(mDecor, getWindow().getAttributes());  mWindowAdded = true;  }  mDecor.setVisibility(View.VISIBLE);  } |

到这里，window的创建过程就已经分析完了

### Dialog的Window创建过程

#### 创建Window

dialog的window创建过程和activity的类似，我们来看下

|  |
| --- |
| Dialog(Context context, int theme, boolean createContextThemeWrapper) {  if (createContextThemeWrapper) {  if (theme == 0) {  TypedValue outValue = new TypedValue();  context.getTheme().resolveAttribute(com.android.internal.R.attr.dialogTheme,  outValue, true);  theme = outValue.resourceId;  }  mContext = new ContextThemeWrapper(context, theme);  } else {  mContext = context;  }  mWindowManager = (WindowManager)context.getSystemService(Context.WINDOW\_SERVICE);  Window w = PolicyManager.makeNewWindow(mContext);  mWindow = w;  w.setCallback(this);  w.setOnWindowDismissedCallback(this);  w.setWindowManager(mWindowManager, null, null);  w.setGravity(Gravity.CENTER);  mListenersHandler = new ListenersHandler(this);  } |

#### 初始化DecorView并将Dialog的师徒添加到DecorView

这个过程也和activity的类似，都是通过window去指定加载的布局文件:

|  |
| --- |
| public void setContentView(int layoutResID) {  mWindow.setContentView(layoutResID);  } |

#### 将DecorView添加到window并且显示

在Dialog的show方法中，会通过windowmanager将DecorView添加到window中

|  |
| --- |
| mWindowManager.addView(mDecor, l);  mShowing = true; |

从上述的三个步骤可以发现，dialog的创建过程和activity很类似

普通的dialog有一个特殊的地方，那就是必须用activity的context，否则会报错

Dialog dialog = new Dialog(this);

dialog.setTitle("Hello");

dialog.show();



上面的信息非常的明确，是没有应用token导致的，而应用token一般只有activity持有，所有这里需要activity的context.

另外系统比较特殊，他可以不需要token. 使用过程：在AndroidManifest.xml中加上权限android.permission.SYSTEM\_ALERT\_WINDOW; 调用dialog.show()之前加alertDialog.getWindow().setType(WindowManager.LayoutParams.TYPE\_SYSTEM\_ERROR); 系统的type有很多都可以用，TYPE\_SYSTEM\_ERROR，TYPE\_SYSTEM\_OVERLAY都可以。

但是Android6.0的则需要通过代码申请权限，使用要特别注意，实在嫌麻烦，最好自己new一个activity当dlg。

### Toast的Window创建过程

Toast和dialog不同，他的工作过稍微复杂一点，首先Toast也是基于Window来实现的，但是由于Toast具有定时取消的功能，所以系统采用了handler，在toast内部有两类IPC的过程，第一类Toast访问NotificationManagerService，第二类是NotificationManagerService回调toast的TN接口

Toast具有系统的window,他内部试下的师徒有两种方式制定，一个是系统设定，还可以setview指定

|  |
| --- |
| public void show() {  if (mNextView == null) {  throw new RuntimeException("setView must have been called");  }  INotificationManager service = getService();  String pkg = mContext.getOpPackageName();  TN tn = mTN;  tn.mNextView = mNextView;  try {  service.enqueueToast(pkg, tn, mDuration);  } catch (RemoteException e) {  // Empty  }  }  /\*\*  \* Close the view if it's showing, or don't show it if it isn't showing yet.  \* You do not normally have to call this. Normally view will disappear on its own  \* after the appropriate duration.  \*/  public void cancel() {  mTN.hide();  try {  getService().cancelToast(mContext.getPackageName(), mTN);  } catch (RemoteException e) {  // Empty  }  } |

从上面的代码可以看出，显示和隐藏是通过NMS来实现的，由于NMS运行在系统，所以只能通过远程调用，这里就跨进程实现了IPC，在这个时候NMS是运行在binder线程池中，所以需要handler切换到主线程中，所以这就意味着Toast无法**在没有Lopper的线程中弹出.**

我们首先来看下Toast的显示过程

|  |
| --- |
| INotificationManager service = getService();  String pkg = mContext.getOpPackageName();  TN tn = mTN;  tn.mNextView = mNextView;  try {  service.enqueueToast(pkg, tn, mDuration);  } catch (RemoteException e) {  // Empty  } |

NMS的enqueueToast方法的第一个参数就是当前应用的包名，第二个参数表示远程回调，第三个是时长，enqueueToast首先将Toast请求封装为一个ToastRecord对象将其添加到一个队列中，其实这本书就是一个arraylist，对于非系统应用来说，他只能存50个，这样是为了防止dos,如果不这样做，试想一下，其他应用还能弹出东西来吗？

代码丢失

正常情况下，一个应用不可能达到上限，当ToastRecord被添加到mToastQueue中后，NMS就会通过showNextToastLocked方法来显示Toast，下面的代码很好理解，需要注意的是，Toast的显示是由ToastRecord的callback来完成的，这个callback实际上就是TN对象的远程Binder，通过callback来访问TN中的方法是需要跨进程来完成的，最终被调用的对象是TN中方法发起在Binder线程池中。

|  |
| --- |
| void showNextToastLocked() {  ToastRecord record = mToastQueue.get(0);  while (record != null) {  if (DBG) Slog.d(TAG, "Show pkg=" + record.pkg + " callback=" + record.callback);  try {  record.callback.show(record.token);  scheduleTimeoutLocked(record);  return;  } catch (RemoteException e) {  Slog.w(TAG, "Object died trying to show notification " + record.callback  + " in package " + record.pkg);  // remove it from the list and let the process die  int index = mToastQueue.indexOf(record);  if (index >= 0) {  mToastQueue.remove(index);  }  keepProcessAliveIfNeededLocked(record.pid);  if (mToastQueue.size() > 0) {  record = mToastQueue.get(0);  } else {  record = null;  }  }  }  } |

Toast显示出来之后，NMS会通过scheduleTimeoutLocked来发送一个延时消息

|  |
| --- |
| private void scheduleTimeoutLocked(ToastRecord r)  {  mHandler.removeCallbacksAndMessages(r);  Message m = Message.obtain(mHandler, MESSAGE\_TIMEOUT, r);  long delay = r.duration == Toast.LENGTH\_LONG ? LONG\_DELAY : SHORT\_DELAY;  mHandler.sendMessageDelayed(m, delay);  } |

上面的代码，LONG\_DELAY是3.5s，SHORT\_DELAY是2s,NMS会通过cancelToastLocked来隐藏toast并且清楚队列。

|  |
| --- |
| ToastRecord record = mToastQueue.get(index);  try {  record.callback.hide();  } catch (RemoteException e) {  Slog.w(TAG, "Object died trying to hide notification " + record.callback  + " in package " + record.pkg);  // don't worry about this, we're about to remove it from  // the list anyway  } |

通过上面的分析，大家知道toast的显示隐藏实际上是toast的TN这个类来实现的，分别对应的show/hide，由于这两个方法都是NMS以跨进程的方式调用的，因此他运行在Binder池中，为了切换，在他们内部使用了handler

|  |
| --- |
| @Override  public void show(IBinder windowToken) {  if (localLOGV) Log.v(TAG, "SHOW: " + this);  mHandler.obtainMessage(0, windowToken).sendToTarget();  }  /\*\*  \* schedule handleHide into the right thread  \*/  @Override  public void hide() {  if (localLOGV) Log.v(TAG, "HIDE: " + this);  mHandler.post(mHide);  } |

上述的代码，mShow和mHide是两个Runnable，他们的内部实现分别调用了具体方法，由此可见，handlershow才是真正的方法，

|  |
| --- |
| public void handleShow(IBinder windowToken) {  if (localLOGV) Log.v(TAG, "HANDLE SHOW: " + this + " mView=" + mView  + " mNextView=" + mNextView);  if (mView != mNextView) {  // remove the old view if necessary  handleHide();  mView = mNextView;  Context context = mView.getContext().getApplicationContext();  String packageName = mView.getContext().getOpPackageName();  if (context == null) {  context = mView.getContext();  }  mWM = (WindowManager)context.getSystemService(Context.WINDOW\_SERVICE);  // We can resolve the Gravity here by using the Locale for getting  // the layout direction  final Configuration config = mView.getContext().getResources().getConfiguration();  final int gravity = Gravity.getAbsoluteGravity(mGravity, config.getLayoutDirection());  mParams.gravity = gravity;  if ((gravity & Gravity.HORIZONTAL\_GRAVITY\_MASK) == Gravity.FILL\_HORIZONTAL) {  mParams.horizontalWeight = 1.0f;  }  if ((gravity & Gravity.VERTICAL\_GRAVITY\_MASK) == Gravity.FILL\_VERTICAL) {  mParams.verticalWeight = 1.0f;  }  mParams.x = mX;  mParams.y = mY;  mParams.verticalMargin = mVerticalMargin;  mParams.horizontalMargin = mHorizontalMargin;  mParams.packageName = packageName;  mParams.hideTimeoutMilliseconds = mDuration ==  Toast.LENGTH\_LONG ? LONG\_DURATION\_TIMEOUT : SHORT\_DURATION\_TIMEOUT;  mParams.token = windowToken;  if (mView.getParent() != null) {  if (localLOGV) Log.v(TAG, "REMOVE! " + mView + " in " + this);  mWM.removeView(mView);  }  if (localLOGV) Log.v(TAG, "ADD! " + mView + " in " + this);  mWM.addView(mView, mParams);  trySendAccessibilityEvent();  }  } |

移除

|  |
| --- |
| public void handleHide() {  if (localLOGV) Log.v(TAG, "HANDLE HIDE: " + this + " mView=" + mView);  if (mView != null) {  // note: checking parent() just to make sure the view has  // been added... i have seen cases where we get here when  // the view isn't yet added, so let's try not to crash.  if (mView.getParent() != null) {  if (localLOGV) Log.v(TAG, "REMOVE! " + mView + " in " + this);  mWM.removeViewImmediate(mView);  }  mView = null;  }  }  } |

到这里，我们的window就全部分析完成了，相信大家对window有了一些新的见解了。

## 常见使用注意

1. 用Application作为dlg的context：
2. 在子线程更新ui
3. 实现系统转屏功能：

**ViewRootImpl**

**mWm** = WindowManagerGlobal.*getWindowManagerService*();

**mWm** = IWindowManager.Stub.*asInterface*(ServiceManager.*getService*(**"window"**));

**mWm**.freezeRotation(2);//180度

## REF

[Android设计模式（二） 续：WindowManager](http://www.jianshu.com/p/3e9f068ed82d)

# 四大组件工作过程

## 四大组件概述

Activity，是一种展示型组件，用于向用户展示UI。它只有一种运行模式：处于启动状态。

Service，是一种计算型组件，用于在后台执行一系列计算任务。它有两种状态：启动状态和绑定状态。Service是运行在主线程的，因此耗时的任务需要在工作线程去完成。Service处于绑定状态时，它内部同样可以进行后台计算。

BroadCastReceiver，是一种消息行组件，用于在不同的组件甚至不同应用间传递消息。静态注册在应用安装时会被系统解析，不需要应用启动就能收到相应的广播；动态注册必须要应用启动后才能收到广播。

ContentProvider，是一种数据共享型组件，用于向其他组件甚至其他应用共享数据。它对数据集合的具体实现没有要求，可以是数据库，List、Map甚至文件等。它内部的insert、delete、update和query方法需要处理好线程同步，因为这几个方法是运行在Binder线程池中的。ContentProvider不需要手动停止。

1、Activity（展示型组件）

Activity的主要作用是展示一个界面并和用户交互，它扮演的是一种前台界面的角色。

（1）需要在AndroidManifest中注册。

（2）需要借助Intent启动。有显示Intent和隐式Intent。隐式Intent指向一个或多个目标Activity组件，当然也可能没有任何一个Activity组件可以处理这个隐式Intent。

（3）对用户而言是可见的。

（4）可以具有特定的启动模式，比如singleTop、singleTask等。

（5）通过Activity的finish方法来结束一个Activity组件的运行。

2、Service（计算型组件）

Service用于在后台执行一系列计算任务。

（1）需要在AndroidManifest中注册。

（2）需要借助Intent启动。

（3）用户无法感知。

（4）两种状态：启动状态和绑定状态。

（5）启动状态：做后台计算，不需要和外界有直接的交互。尽管Service组件用于执行后台计算，但它本身是运行在主线程中的，因此耗时的后台计算仍然需要在单独的线程中去完成。

（6）绑定状态：这个时候Service内部同样可以进行后台计算，但是处于这种状态时外界可以很方便的和Service组件进行通信。IPC啊~

（7）灵活采用stopService和unBindService这两个方法才能完全停止一个Service组件。

3、BroadcastReceiver（消息型组件）

BroadcastReceiver用于在不同的组件乃至不同的应用之间传递消息。

（1）可以在AndroidManifest中静态注册，这种注册方式不需要应用启动就可以收到相应的广播；也可以在代码中动态注册，Context.registerReceiver（）和 Context.unRegisterReceiver（），必须要启动应用才能注册并接收广播。

（2）需要借助Intent启动。

（3）用户无法感知。

（4）在实际开发中通过Context的一系列send方法来发送广播，被发送的广播会被系统发送给感兴趣的广播接收者，发送和接收过程的匹配是通过广播接收者的<intent-filter>来描述的。所以呢，BroadcastReceiver组件可以用来实现低耦合的观察者模式，观察者和被观察者之间可以没有任何耦合。

（5）不适合用来执行耗时操作。

（6）BroadcastReceiver组件一般来说不需要停止，它也没有停止的概念。

4、ContentProvider（数据共享型组件）

ContentProvider用于向其他组件乃至其他应用共享数据。

（1）需要在AndroidManifest中注册。

（2）无需借助Intent启动。

（3）用户无法感知。

（4）它的内部需要实现增删查改这四种操作，在它的内部维持着一份数据集合，这个数据集合既可以通过数据库来实现，也可以采用其他任何类型来实现，比如List和Map，ContentProvider对数据集合的具体实现并没有任何要求。

（5）注意点！！！ContentProvider内部的insert、deleted、update和query方法需要处理好线程同步，因为这几个方法是在Binder线程池中被调用的。

（6）ContentProvider无需手动停止。

## Activity的工作过程



performLaunchActivity这个方法主要完成如下几件事：  
1、从ActivityClientRecord中获取待启动的Activity的组件信息

ActivityInfo aInfo = r.activityInfo;

if (r.packageInfo == null) {

r.packageInfo = getPackageInfo(aInfo.applicationInfo, r.compatInfo,

Context.CONTEXT\_INCLUDE\_CODE);

}

ComponentName component = r.intent.getComponent();

if (component == null) {

component = r.intent.resolveActivity(

mInitialApplication.getPackageManager());

r.intent.setComponent(component);

}

if (r.activityInfo.targetActivity != null) {

component = new ComponentName(r.activityInfo.packageName,

r.activityInfo.targetActivity);

}

2、通过Instrumention的newActivity方法使用类加载器创建Activity对象

Activity activity = null;

try {

java.lang.ClassLoader cl = r.packageInfo.getClassLoader();

activity = mInstrumentation.newActivity(

cl, component.getClassName(), r.intent);

StrictMode.incrementExpectedActivityCount(activity.getClass());

r.intent.setExtrasClassLoader(cl);

r.intent.prepareToEnterProcess();

if (r.state != null) {

r.state.setClassLoader(cl);

}

} catch (Exception e) {

if (!mInstrumentation.onException(activity, e)) {

throw new RuntimeException(

"Unable to instantiate activity " + component

+ ": " + e.toString(), e);

}

}

3、通过LoadedApk的makeApplication方法来尝试创建Application对象  
LoadedApk.calss被标记为@hide类，源码在sdk/sources/android-22/android/app目录下

public Application makeApplication(boolean forceDefaultAppClass,

Instrumentation instrumentation) {

if (mApplication != null) {

return mApplication;

}

Application app = null;

String appClass = mApplicationInfo.className;

if (forceDefaultAppClass || (appClass == null)) {

appClass = "android.app.Application";

}

try {

java.lang.ClassLoader cl = getClassLoader();

if (!mPackageName.equals("android")) {

initializeJavaContextClassLoader();

}

ContextImpl appContext = ContextImpl.createAppContext(mActivityThread, this);

app = mActivityThread.mInstrumentation.newApplication(

cl, appClass, appContext);

appContext.setOuterContext(app);

} catch (Exception e) {

if (!mActivityThread.mInstrumentation.onException(app, e)) {

throw new RuntimeException(

"Unable to instantiate application " + appClass

+ ": " + e.toString(), e);

}

}

mActivityThread.mAllApplications.add(app);

mApplication = app;

if (instrumentation != null) {

try {

instrumentation.callApplicationOnCreate(app);

} catch (Exception e) {

if (!instrumentation.onException(app, e)) {

throw new RuntimeException(

"Unable to create application " + app.getClass().getName()

+ ": " + e.toString(), e);

}

}

}

// Rewrite the R 'constants' for all library apks.

SparseArray<String> packageIdentifiers = getAssets(mActivityThread)

.getAssignedPackageIdentifiers();

final int N = packageIdentifiers.size();

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

final int id = packageIdentifiers.keyAt(i);

if (id == 0x01 || id == 0x7f) {

continue;

}

rewriteRValues(getClassLoader(), packageIdentifiers.valueAt(i), id);

}

return app;

}

Application创建完毕后，系统会通过Instrumention的callApplicationOncreate来调用Application的onCreate方法

4、创建ContextImpl对象并通过Activity发attach方法来完成一些重要数据的初始化  
5、调用Activity的onCreate方法

## Service的工作过程

### Service的启动过程

一张图说明Service的启动流程：



handleCreateService方法主要完成如下几件事：

1、通过类加载器创建Service的实例

2、创建Application对象并调用其onCreate方法

3、创建ContextImpl对象并通过Service发attach方法建立二者之间的关系

4、调用Service的onCreate方法并将Service对象存储到ActivityThread中的一个列表中。ActivityThread通过handleServiceArgs方法调用Service的onStarCommand方法

* + 1. **Service的绑定过程**

一张图说明Service的绑定过程：



这里写图片描述

handleBindService方法主要完成如下几件事：  
1、根据Service的token取出Service对象，调用Service的onBind方法  
2、通过ActivityManagerService调用publishService方法通知客户端已经成功连接Service

## BroadcastReceiver的工作过程

Activity、Service、ContentProvider和BroadcastReceiver的静态注册都是在应用安装时由PackageManagerService解析注册的。

**BroadcastReceiver的注册过程**



这里写图片描述

**BroadcastReceiver的发送和接收过程**

自Android3.1开始，系统为Intent新增了两个标志，并为所有广播默认添加 FLAG\_EXCLUDE\_STOPPED\_PACKAGES 标志，防止无意间或者在不必要的时候唤醒已经停止的APP，这个特性同样会影响开机广播

* FLAG\_INCLUDE\_STOPPED\_PACKAGES  
  表示包含已经停止的APP，这个时候广播会发送到已经停止的APP
* FLAG\_EXCLUDE\_STOPPED\_PACKAGES  
  表示不包含已经停止的APP，这个时候广播不会发送给已经停止的APP

当两个标志共存时，以 FLAG\_INCLUDE\_STOPPED\_PACKAGES 为准



## ContentProvider的工作过程

当ContentProvider所在的进程启动时，ContentProvider会同时启动并被发布到ActivityManagerService中，***这个时候ContentProvider的onCreate要先于Application的onCreate执行***。



这里写图片描述

ActivityThread的handleBindApplication方法最终完成Application和ContentProvider的创建，步骤如下：

* 创建ContextImpl和Instrumention
* 创建Application对象
* 启动当前进程的ContentProvider并调用其onCreate方法
* 调用Application的onCreate方法

# Android的消息机制

本章所要讲述的内容是Android的消息机制。提到消息机制读者应该都不陌生，在日常开发中不可避免地要涉及这方面的内容。从开发的角度来说，Handler是Android消息机制的上层接口，这使得在开发过程中只需要和Handler交互即可。Handler的使用过程很简单，通过它可以轻松地将一个任务切换到Handler所在的线程中去执行。很多人认为Handler的作用是更新UI，这的确没错，但是更新UI仅仅是Handler的一个特殊的使用场景.具体来说是这样的：有时候需要在子线程中进行耗时的I/O操作，可能是读取文件或者访问网络等，当耗时操作完成以后可能需要在UI上做一些改变，由于Android开发规范的限制，我们并不能在子线程中访问UI控件，否则就会触发程序异常，**这个时候通过Handler就可以将更新UI的操作切换到主线程中执行**。因此，本质上来说，Handler并不是专门用于更新UI的，它只是常被开发者用来更新UI。

Android的消息机制主要是指Handler的运行机制，Handler的运行需要底层的MessageQueue和Looper的支撑。MessageQueue的中文翻译是消息队列，顾名思义，它的内部存储了一组消息，以队列的形式对外提供插入和删除的工作。虽然叫消息队列，但是它的内部存储结构并不是真正的队列，而是采用**单链表**的数据结构来存储消息列表。Looper的中文翻译为循环，在这里可以理解为消息循环。由于MessageQueue只是一个消息的存储单元，它不能去处理消息，而Looper就填补了这个功能，Looper会以无限循环的形式去查找是否有新消息，如果有的话就处理消息，否则就一直等待着。Looper中还有一个特殊的概念，那就是ThreadLocal，ThreadLocal并不是线程，它的作用是可以在每个线程中存储数据。我们知道，Handler创建的时候会采用当前线程的Looper来构造消息循环系统，那么Handler内部如何获取到当前线程的Looper呢？这就要使用ThreadLocal了，ThreadLocal可以在不同的线程中互不干扰地存储并提供数据，通过ThreadLocal可以轻松获取每个线程的Looper。当然需要注意的是，线程是默认没有Looper的，如果需要使用Handler就必须为线程创建Looper。我们经常提到的主线程，也叫UI线程，它就是ActivityThread，ActivityThread被创建时就会初始化Looper，这也是在主线程中默认可以使用Handler的原因。

## Android的消息机制概述

前面提到，Android的消息机制主要是指Handler的运行机制以及Handler所附带的MessageQueue和Looper的工作过程，这三者实际上是一个整体，只不过我们在开发过程中比较多地接触到Handler而已。**Handler的主要作用是将一个任务切换到某个指定的线程中去执行**，**那么Android为什么要提供这个功能呢？或者说Android为什么需要提供在某个具体的线程中执行任务这种功能呢？**这是因为Android规定访问UI只能在主线程中进行，如果在子线程中访问UI，那么程序就会抛出异常。 ViewRootImpl对UI操作做了验证，这个验证工作是由ViewRootImpl的checkThread方法来完成的，如下所示。



针对checkThread方法中抛出的异常信息，相信读者在开发中都曾经遇到过。由于这一点的限制，导致必须在主线程中访问UI，但是Android又建议不要在主线程中进行耗时操作，否则会导致程序无法响应即ANR。考虑一种情况，假如我们需要从服务端拉取一些信息并将其显示在UI上，这个时候必须在子线程中进行拉取工作，拉取完毕后又不能在子线程中直接访问UI，如果没有Handler，那么我们的确没有办法将访问UI的工作切换到主线程中去执行。因此，**系统之所以提供Handler，主要原因就是为了解决在子线程中无法访问UI的矛盾**。

这里再延伸一点，**系统为什么不允许在子线程中访问UI呢？这是因为Android的UI控件不是线程安全的**，如果在多线程中并发访问可能会导致UI控件处于不可预期的状态，那为什么系统不对UI控件的访问加上锁机制呢？缺点有两个：首先加上锁机制会让UI访问的逻辑变得复杂；其次锁机制会降低UI访问的效率，因为锁机制会阻塞某些线程的执行。鉴于这两个缺点，最简单且高效的方法就是采用**单线程模型来处理UI操作**，对于开发者来说也不是很麻烦，只是需要通过Handler切换一下UI访问的执行线程即可。

Handler的使用方法这里就不做介绍了，这里描述一下Handler的工作原理。Handler创建时会采用当前线程的Looper来构建内部的消息循环系统，如果当前线程没有Looper，那么就会报错，如下所示。

代码如下：

|  |
| --- |
| **new** Thread(**new** Runnable() {  @Override  **public void** run() {*//Can't create handler inside thread that has not called Looper.prepare()* Toast.*makeText*(KApplication.*getIns*(),**"toast in sub Thread"**,Toast.***LENGTH\_LONG***).show();  } }).start(); |

报错如下：



如果解决上述问题呢？其实很简单，只需要为当前线程创建Looper即可，或者在一个有Looper的线程中创建Handler也行，具体会在10.2.3节中进行介绍。

或者直接用主线程的Looper



Handler创建完毕后，这个时候其内部的Looper以及MessageQueue就可以和Handler一起协同工作了，然后通过Handler的post方法将一个Runnable投递到Handler内部的Looper中去处理，也可以通过Handler的send方法发送一个消息，这个消息同样会在Looper中去处理。其实post方法最终也是通过**send**方法来完成的，接下来主要来看uop下send方法的工作过程。当Handler的send方法被调用时，它会调用MessageQueue的enqueueMessage方法将这个消息放入消息队列中，然后Looper发现有新消息到来时，就会处理这个消息，最终消息中的Runnable或者Handler的 handleMessage方法就会被调用。注意Looper是运行在创建Handler所在的线程中的，这样一来Handler中的业务逻辑就被切换到创建Handler所在的线程中去执行了，这个过程可以用图10-1来表示。



图10-1  Handler的工作过程

## Android的消息机制分析

在10.1节中对Android的消息机制已经做了一个概括性的描述，通过图10-1也能够比较好地理解Handler的工作过程。本节将对Android消息机制的实现原理做一个全面的分析。由于Android的消息机制实际上就是Handler的运行机制，因此本节主要围绕着Handler的工作过程来分析Android的消息机制，主要包括Handler、MessageQueue和Looper。同时为了更好地理解Looper的工作原理，本节还会介绍ThreadLocal，通过本节的介绍可以让读者对Android的消息机制有一个深入的理解。

### ThreadLocal的工作原理

ThreadLocal是一个线程内部的数据存储类，通过它可以在指定的线程中存储数据，数据存储以后，只有在指定线程中可以获取到存储的数据，对于他线程来说则无法获取到数据。在日常开发中用到ThreadLocal的地方较少，但是在某些特殊的场景下，通过ThreadLocal可以轻松地实现一些看起来很复杂的功能，这一点在Android的源码中也有所体现，比如Looper、ActivityThread以及AMS中都用到了ThreadLocal。具体到ThreadLocal的使用场景，这个不好统一来描述，一般来说，当某些数据是以线程为作用域并且不同线程具有不同的数据副本的时候，就可以考虑采用ThreadLocal。比如对于Handler来说，它需要获取当前线程的Looper，很显然Looper的作用域就是线程并且不同线程具有不同的Looper，这个时候通过ThreadLocal就可以轻松实现Looper在线程中的存取。如果不采用ThreadLocal，那么系统就必须提供一个全局的哈希表供Handler查找指定线程的Looper，这样一来就必须提供一个类似于LooperManager的类了，但是系统并没有这么做而是选择了ThreadLocal，这就是ThreadLocal的好处。

ThreadLocal另一个使用场景是复杂逻辑下的对象传递，比如监听器的传递，有些时候一个线程中的任务过于复杂，这可能表现为函数调用栈比较深以及代码入口的多样性，在这种情况下，我们又需要监听器能够贯穿整个线程的执行过程，这个时候可以怎么做呢？其实这时就可以采用ThreadLocal，采用ThreadLocal可以让监听器作为线程内的全局对象而存在，**在线程内部只要通过get方法就可以获取到监听器**。如果不采用ThreadLocal，那么我们能想到的可能是如下两种方法：第一种方法是将监听器通过参数的形式在函数调用栈中进行传递，第二种方法就是将监听器作为静态变量供线程访问。上述这两种方法都是有局限性的。第一种方法的问题是当函数调用栈很深的时候，通过函数参数来传递监听器对象这几乎是不可接受的，这会让程序的设计看起来很糟糕。**第二种方法是可以接受的，但是这种状态是不具有可扩充性的**，比如同时有两个线程在执行，那么就需要提供两个静态的监听器对象，如果有10个线程在并发执行呢？提供10个静态的监听器对象？这显然是不可思议的，而采用ThreadLocal，每个监听器对象都在自己的线程内部存储，根据就不会有方法2的这种问题。

介绍了那么多ThreadLocal的知识，可能还是有点抽象，下面通过实际的例子来演示ThreadLocal的真正含义。首先定义一个ThreadLocal对象，这里选择Boolean类型的，如下所示。代码如下，结果看注释部分：



在上面的代码中，在主线程中设置mBooleanThreadLocal的值为true，在子线程1中设置mBooleanThreadLocal的值为false，在子线程2中不设置mBooleanThreadLocal的值。然后分别在3个线程中通过get方法去mBooleanThreadLocal的值，根据前面对ThreadLocal的描述，这个时候，主线程中应该是true，子线程1中应该是false，而子线程2中由于没有设置值，所以应该是null。

从上面日志可以看出，虽然在不同线程中访问的是同一个ThreadLocal对象，但是它们通过ThreadLocal获取到的值却是不一样的，这就是ThreadLocal的奇妙之处。结合这这个例子然后再看一遍前面对ThreadLocal的两个使用场景的理论分析，我们应该就能比较好地理解ThreadLocal的使用方法了。ThreadLocal之所以有这么奇妙的效果，是因为不同线程访问同一个ThreadLocal的get方法，ThreadLocal内部会从各自的线程中取出一个数组，然后再从数组中根据当前ThreadLocal的索引去查找出对应的value值，很显然，不同线程中的数组是不同的，这就是为什么通过ThreadLocal可以在不同的线程中维护一套数据的副本并且彼此互不干扰。

对ThreadLocal的使用方法和工作过程做了介绍后，下面分析ThreadLocal的内部实现， ThreadLocal是一个泛型类，它的定义为public class ThreadLocal<T>，只要弄清楚ThreadLocal的get和set方法就可以明白它的工作原理。

首先看ThreadLocal的set方法，如下所示。

Get和set的源码如下，api24和其他版本的可能不一样，其实是内部维护了一个HashMap！！！



此处和书中分析的源码不同，不做展开。

### 消息队列的工作原理

消息队列在Android中指的是MessageQueue，MessageQueue主要包含两个操作：插入和读取。读取操作本身会伴随着删除操作，插入和读取对应的方法分别为enqueueMessage和next，其中enqueueMessage的作用是往消息队列中插入一条消息，而next的作用是从消息队列中取出一条消息并将其从消息队列中移除。尽管MessageQueue叫消息队列，但是它的内部实现并不是用的队列，实际上它是通过一个单链表的数据结构为维护消息列表，单链表在插入和删除上比较有优势。下面主要看一下它的enqueueMessage和next方法的实现，enqueueMessage的源码的伪代码如下所示。

|  |
| --- |
| boolean enqueueMessage(Message msg, long when) {  msg.when = when;  Message p = mMessages;  if (p == null || when == 0 || when < p.when) {  // New head, wake up the event queue if blocked.  msg.next = p;  mMessages = msg;  } else {  Message prev;  for (;;) {  prev = p;  p = p.next;  if (p == null || when < p.when) {  break;  }  }  msg.next = p; // invariant: p == prev.next  prev.next = msg;  }  } |

就是一个按照时间排序的单链表的插入操作。下面看一下next方法的实现，next的主要逻辑如下所示。

|  |
| --- |
| **if** (msg != **null** && msg.target == **null**) {  *// Stalled by a barrier. Find the next asynchronous message in the queue.* **do** {  prevMsg = msg;  msg = msg.next;  } **while** (msg != **null** && !msg.isAsynchronous()); } |

可以发现next方法是一个无限循环的方法，如果消息队列中没有消息，那么next方法会一直阻塞在这里。当有新消息到来时，next方法会返回这条消息并将其从单链表中移除。

### Looper的工作原理

在10.2.2节中介绍了消息队列的主要实现，本节将分析Looper的具体实现。Looper在Android的消息机制中扮演着消息循环的角色，具体来说就是它会不停地从MessageQueue中去查看是否有新消息，如果有新消息就会立刻处理，否则就一直阻塞在那里。首先看一下它的构造方法，在构造方法中它会创建一个MessageQueue即消息队列，然后将当前线程的对象保存起来，如下所示。



我们知道，Handler的工作需要Looper，没有Looper的线程就会报错，那么如何为一个线程创建Looper呢？其实很简单，通过Looper.prepare()即可为当前线程创建一个Looper，接着通过Looper.loop()来开启消息循环，如下所示。



Looper除了prepare方法外，还提供了prepareMainLooper方法，这个方法主要是给主线程也就是ActivityThread创建Looper所使用的，其本质也是通过prepare方法来实现的。由于主线程的Looper比较特殊，所以Looper提供了一个getMainLooper方法，通过它可以在任何地方获取到主线程的Looper。Looper也是可以退出的，Looper提供了quit和quitSafely来退出一个Looper，二者的区别是：quit会直接退出Looper，而quitSafely只是设定一个退出标记，然后把消息队列中的已有消息处理完毕后才安全地退出。Looper退出后，通过Handler发送的消息会失败，这个时候Handler的send方法会返回false。在子线程中，如果手动为其创建了Looper，那么在所有的事情完成以后应该调用quit方法来终止消息循环，否则这个子线程就会一直处于等待的状态，而如果退出Looper以后，这个线程就会立刻终止，因此建议不需要的时候终止Looper。

Looper最重要的一个方法是loop方法，只有调用了loop后，消息循环系统才会真正地起作用，它的实现如下所示。

|  |
| --- |
| public static void loop() {  for (;;) {  Message msg = queue.next(); // might block  if (msg == null) {  // No message indicates that the message queue is quitting.  return;  }  }  } |

Looper的loop方法的工作过程也比较好理解，loop方法是一个死循环，唯一跳出循环的方式是MessageQueue的next方法返回了null。当Looper的quit方法被调用时，Looper就会调用MessageQueue的quit或者quitSafely方法来通知消息队列退出，当消息队列被标记为退出状态时，它的next方法就会返回null。也就是说，Looper必须退出，否则loop方法就会无限循环下去。loop方法会调用MessageQueue的next方法来获取新消息，**而next是一个阻塞操作，当没有消息时，next方法会一直阻塞在那里**，这也导致loop方法一直阻塞在那里。如果MessageQueue的next方法返回了新消息，Looper就会处理这条消息：msg.target.dispatchMessage(msg)，这里的msg.target是发送这条消息的Handler对象，这样Handler发送的消息最终又交给它的dispatchMessage方法来处理了。但是这里不同的是，Handler的dispatchMessage方法是在创建Handler时所使用的Looper中执行的，这样就成功地将代码逻辑切换到指定的线程中去执行了。

### Handler的工作原理

Handler的工作主要包含消息的发送和接收过程。消息的发送可以通过post的一系列方法以及send的一系列方法，post的一系列方法最终是通过send的一系列方法来实现的。发送一条消息的典型过程如下所示。sendMessage，sendMessageDelayed，sendMessageAtTime，enqueueMessage

可以发现，Handler发送消息的过程仅仅是向消息队列中插入了一条消息，MessageQueue的next方法就会返回这条消息给Looper，Looper收到消息后就开始处理了，最终消息由Looper交由Handler处理，即Handler的dispatchMessage方法会被调用，这时Handler就进入了处理消息的阶段。dispatchMessage的实现如下所示：

|  |
| --- |
| **public void** dispatchMessage(Message msg) {  **if** (msg.callback != **null**) {  *handleCallback*(msg);  } **else** {  **if** (**mCallback** != **null**) {  **if** (**mCallback**.handleMessage(msg)) {  **return**;  }  }  handleMessage(msg);  } } |

Handler处理消息的过程如下：

首先检查Message的callback是否为null，不为null就通过handleCallback来处理消息。**Message的callback是一个Runnable对象**，**实际上就是Handler的post方法所传递的Runnable参数**。handleCallback就是执行message.callback.run();

其次检查mCallback是否为null，不为null就调用mCallback的handleMessage方法来处理消息。Callback是个仅含handleMessage的接口。通过Callback可以采用如下方式来创建Handler对象：Handler handler = new Handler(callback)。那么Callback的意义是什么呢？源码里面的注释已经做了说明：可以用来创建一个Handler的实例但并不需要派生Handler的子类。在日常开发中，创建Handler最常见的方式就是派生一个Handler的子类并重写其handleMessage方法来处理具体的消息，而Callback给我们提供了另外一种使用Handler的方式，当我们不想派生子类时，就可以通过Callback来实现。 简称匿名线程

|  |
| --- |
| **new** Thread(**new** Runnable() {  @Override  **public void** run() {*//  /\* \*/* } }).start(); |

最后，调用Handler的handleMessage方法来处理消息。Handler处理消息的过程可以归纳为一个流程图，如图10-2所示。



Handler还有一个特殊的构造方法，那就是通过一个特定的Looper来构造Handler，它的实现如下所示。通过这个构造方法可以实现一些特殊的功能，然后调用重载的多参数方法。

|  |
| --- |
| **public** Handler(Looper looper, Callback callback, **boolean** async) {  **mLooper** = looper;  **mQueue** = looper.mQueue;  **mCallback** = callback;  **mAsynchronous** = async; } |

下面看一下Handler的一个默认构造方法public Handler()，这个构造方法会调用下面的构造方法。很明显，如果当前线程没有Looper的话，就会抛出“Can't create handler inside thread that has not called Looper.prepare()”这个异常，这也解释了在没有Looper的子线程中创建Handler会引发程序异常的原因。

|  |
| --- |
| **public** Handler(Callback callback, **boolean** async) {  **mLooper** = Looper.*myLooper*();  **if** (**mLooper** == **null**) {  **throw new** RuntimeException(  **"Can't create handler inside thread that has not called Looper.prepare()"**);  }  **mQueue** = **mLooper**.mQueue;  **mCallback** = callback;  **mAsynchronous** = async; } |

## 主线程的消息循环

Android的主线程就是ActivityThread，主线程的入口方法为main，在main方法中系统会通过Looper.prepareMainLooper()来创建主线程的Looper以及MessageQueue，并通过Looper.loop()来开启主线程的消息循环，这个过程如下所示。



主线程的消息循环开始了以后，ActivityThread还需要一个Handler来和消息队列进行交互，这个Handler就是ActivityThread.H，它内部定义了一组消息类型，主要包含了四大组件的启动和停止等过程，如下所示。

|  |
| --- |
| **private class** H **extends** Handler {  **public static final int** LAUNCH\_ACTIVITY = 100;  **public static final int** PAUSE\_ACTIVITY = 101;  **public static final int** PAUSE\_ACTIVITY\_FINISHING = 102;  **public static final int** STOP\_ACTIVITY\_SHOW = 103;  **public static final int** STOP\_ACTIVITY\_HIDE = 104;  **public static final int** SHOW\_WINDOW = 105;  **public static final int** HIDE\_WINDOW = 106;  **public static final int** RESUME\_ACTIVITY = 107;  **public static final int** SEND\_RESULT = 108;  **public static final int** DESTROY\_ACTIVITY = 109;  **public static final int** BIND\_APPLICATION = 110;  **public static final int** EXIT\_APPLICATION = 111;  **public static final int** NEW\_INTENT = 112;  **public static final int** RECEIVER = 113;  **public static final int** CREATE\_SERVICE = 114;  **public static final int** SERVICE\_ARGS = 115;  **public static final int** STOP\_SERVICE = 116; |

ActivityThread通过ApplicationThread和AMS进行进程间通信，AMS以进程间通信的方式完成ActivityThread的请求后会回调ApplicationThread中的Binder方法，然后ApplicationThread会向H发送消息，H收到消息后会将ApplicationThread中的逻辑切换到ActivityThread中去执行，即切换到主线程中去执行，这个过程就是**主线程的消息循环模型。**

# Android的线程和线程池

除了最常使用的Thread之外, 在Android之中可以扮演线程角色的还有很多: 如AsyncTask和IntentService, 同时HandlerThread也是一种特殊的线程. 虽然这些线程的表现形式有别于基础线程. 但是本质上还是传统的线程. 例如AsyncTask它的底层使用了线程池. 而对于IntentService和HandlerThread来说, 他们的底层则直接使用了线程.

**根据不同的特性来实现不同的场景**

* AsyncTask: 封装了线程池和Handler, 它主要是为了方便开发者在子线程中更新UI.
* HandlerThread: 是一种具有消息循环的线程, 在它的内部可以使用Handler.
* IntentService: 是一个服务, 系统对其进行了封装使其可以更方便地执行后台任务, IntentService内部采用了HandlerThread来执行任务, 当任务执行完毕后IntentService会自动退出. 从执行任务的角度来看, 更像一个后台的线程. 但是因为其本身是一种服务, 所以导致不容易被系统杀死从而保证任务的执行. 而如果是一个后台线程, 由于这个时候进行中没有活动的四大组件, 那么这个进程的优先级会很低, 很容易被系统杀死, 这就是IntentService的优点.

**线程的简单概述**

在操作系统中, 线程是操作系统调度的最小单元, 同时线程又是一种受限的系统资源, 即线程不可能无限制的产生, 并且线程的创建和销毁都会有相应的开销. 当系统中存在大量的线程时, 系统会通过时间片转轮的方式调度每个线程, 因此线程不可能做到绝对的并行, 除非线程数量小于等于cpu的核心数. 但这种情况太少了, 所以线程池的概念就由此出现. 通过线程池就可以避免因为频繁创建和销毁线程所带来的系统开销。

## 主线程和子线程

主线程是指进程所拥有的线程, 在Java中默认情况下一个进程只有一个线程, 这个线程就是主线程. 主线程主要处理界面交互相关的逻辑, 因为用户随时会和界面发生交互, **因此主线程在任何时候都必须有较高的响应速度**, 否则就会产生一种界面卡顿的感觉. 为了保持较高的响应速度, 这就要求主线程中不能执行耗时的任务, 这个时候子线程就派上用场. 子线程也叫作工作线程, 除了主线程以外的线程都叫做子线程

Android沿用了Java的线程模型, 其中的线程也分为主线程和子线程, 其中主线程也叫UI线程. **主线程的作用是运行四大组件以及处理它们和用户的交互**. 而子线程的作用则是执行耗时任务, 比如网络请求, I/O操作等. 从Android 3.0 开始系统要求网络访问必须在子线程中进行, 否则网络访问将会失败并抛出NetworkOnMainThreadException这个异常, 这样做是为了避免主线程由于耗时操作所阻塞从而出现ANR现象.

## Android中的线程形态

### AsyncTask

*AsyncTask是一种轻量级的异步任务类, 他可以在线程池中执行后台任务, 然后把执行的进度和最终的结果传递给主线程并在主线程更新UI. 从实现上来说. AsyncTask封装了Thread和Handler, 通过AsyncTask可以更加方便地执行后台任务, 对于特别耗时的任务来说, 不建议用AsyncTask，建议使用线程池.*

public abstract class AsyncTask<Params, Progress, Result>

AsyncTask就是一个抽象的泛型类. 这三个泛型的意义.

* Params: 表示参数的类型
* Progress: 表示后台任务的执行进度的类型
* Result: 则表示后台任务的返回结果的类型

如果不需要传递具体的参数, 那么这三个泛型参数可以用Void来代替.

**AsyncTask提供了4个核心方法, 含义如下**

1. onPreExecute(): **在主线程执行**, 在异步任务执行之前, 此方法会被调用, 一般可以用于做一些准备工作
2. doInBackground(): **在线程池中执行**, 此方法用于执行异步任务, 参数params表示异步任务的输入参数. 在此方法中可以通过publishProgress()方法来更新任务的进度, publishProgress()方法会调用onProgressUpdate()方法. 另外此方法需要返回计算结果给onPostExecute()
3. onProgressUpdate(): **在主线程执行**,在异步任务执行之后, 此方法会被调用, 其中result参数是后台任务的返回值, 即doInBackground()的返回值.
4. onPostExecute(): **在主线程执行**, 在异步任务执行之后, 此方法会被调用, 其中result参数是后台任务的返回值, 即doInBackground的返回值.

除了上述的四种方法,还有onCancelled(), 它同样在主线程执行, 当异步任务被取消时, onCancelled()方法会被调用, 这个时候onPostExecute()则不会被调用.

**AsyncTask在使用过程中有一些条件限制**

1. AsyncTask的类必须在主线程被加载, 这就意味着第一次访问AsyncTask必须发生在主线程, 这个问题不是绝对, 因为在Android 4.1及以上的版本已经被系统自动完成. 在Android 5.0的源码中, 可以看到ActivityThread#main()会调用AsyncTask#init()方法.
2. AsyncTask的对象必须在主线程中创建.
3. execute方法必须在UI线程调用.
4. 不要在程序中直接调用onPreExecute(), onPostExecute(), doInBackground和onProgressUpdate()
5. 一个AsyncTask对象只能执行一次, 即只能调用一次execute()方法, 否则会报运行时异常.
6. 在Android 1.6之前, AsyncTask是串行执行任务的; Android 1.6的时候AsyncTask开始采用线程池里处理并行任务; 但是Android 3.0开始, 为了避免AsyncTask带来的并发错误, AsyncTask又采用了一个线程来串行的执行任务. 尽管如此在3.0以后, 仍然可以通过AsyncTask#executeOnExecutor()方法来并行执行任务.

### AsyncTask的工作原理

这里以源码5.0来分析, 不同的版本源码具体实现是不同的.

为了分析AsyncTask的工作原理, 可以从它的execute()方法开始分析, execute()方法又会调用executeOnExecutor()方法. 实际上这里调用进来后是一个串行的线程池, 一个进程中所有的AsyncTask全都在这个串行的线程池中排队执行, 然后会先调用AsyncTask#onPreExecute()方法, 然后线程池开始执行. 看看SerialExecutor()的源码实现:

|  |
| --- |
| public static final Executor SERIAL\_EXECUTOR = new SerialExecutor();  private static volatile Executor sDefaultExecutor = SERIAL\_EXECUTOR;  private static class SerialExecutor implements Executor {  final ArrayDeque<Runnable> mTasks = new ArrayDeque<Runnable>();  Runnable mActive;  public synchronized void execute(final Runnable r) {  mTasks.offer(new Runnable() {  public void run() {  try {  r.run();  } finally {  scheduleNext();  }  }  });  if (mActive == null) {  scheduleNext();  }  }  protected synchronized void scheduleNext() {  if ((mActive = mTasks.poll()) != null) {  THREAD\_POOL\_EXECUTOR.execute(mActive);  }  }  } |

在SerialExecutor的实现可以分析AsyncTask的排队执行的过程. 首先系统会把AsyncTask#Params参数封装成FutureTask对象, FutureTask是一个并发类, 在这里充当了Runnable的作用. 接着这个FutureTask会交给SerialExecutor#execute()方法去处理. 这个方法首先会把FutureTask对象插入到任务队列mTasks中, 如果这个时候没有正在活动AsyncTask任务, 那么就会调用SerialExecutor#scheduleNext()方法来执行下一个AsyncTask任务. 同时当一个AsyncTask任务执行完后, AsyncTask会继续执行其他任务直到所有的任务都执行完毕为止, 从这一点可以看出, 在默认情况下, **AsyncTask是串行执行的**

AsyncTask中有两个线程池(SerialExecutor和THREAD\_POOL\_EXECUTOR)和一个Handler(InternalHandler), 其中线程池SerialExecutor用于任务的排列, 而线程池THREAD\_POOL\_EXECUTOR用于真正的执行任务, 而InternalHandler用于将执行环境从线程切换到主线程, 其本质仍然是线程的调用过程.

在AsyncTask的构造方法中有如下这段代码, 由于FutureTask#run()方法会调用mWorker.call()方法, 因此mWorker的call方法最终会在线程池中执行.

|  |
| --- |
| mWorker = new WorkerRunnable<Params, Result>() {  public Result call() throws Exception {  mTaskInvoked.set(true);  Process.setThreadPriority(Process.THREAD\_PRIORITY\_BACKGROUND);  *//noinspection unchecked*  return postResult(doInBackground(mParams));  }  }; |

在mWorker.call()方法中, 首先将mTaskInvoked设为了true. 表示当前任务已经被调用过了. 然后执行AsyncTask#doInBackground()方法, 接着将其返回值传递给postResult(), 这个方法的实现:

|  |
| --- |
| private Result postResult(Result result) {  @SuppressWarnings("unchecked")  Message message = sHandler.obtainMessage(MESSAGE\_POST\_RESULT,  new AsyncTaskResult<Result>(this, result));  message.sendToTarget();  return result;  } |

这里, postResult()会通过sHandler发送一个MESSAGE\_POST\_RESULT消息, 而sHandler的定义如下:

|  |
| --- |
| private static final InternalHandler sHandler = new InternalHandler();  private static class InternalHandler extends Handler {  @SuppressWarnings({"unchecked", "RawUseOfParameterizedType"})  @Override  public void handleMessage(Message msg) {  AsyncTaskResult result = (AsyncTaskResult) msg.obj;  switch (msg.what) {  case MESSAGE\_POST\_RESULT:  *// There is only one result*  result.mTask.finish(result.mData[0]);  break;  case MESSAGE\_POST\_PROGRESS:  result.mTask.onProgressUpdate(result.mData);  break;  }  }  } |

可以看出sHandler是一个静态的Handler对象, 为了能够将执行环境切换到主线程, 这就要求sHandler这个对象必须在主线程中创建. 由于静态成员会在加载类的时候进行初始化, 因此这就变相要求AsyncTask的类必须在主线程中加载, 否则同一个进程中的AsyncTask都无法正常工作. sHandler收到了消息后会调用AsyncTask#finish()方法, 如下:

|  |
| --- |
| private void finish(Result result) {  if (isCancelled()) {  onCancelled(result);  } else {  onPostExecute(result);  }  mStatus = Status.FINISHED;  } |

这个finish()方法很简单, 如果AsyncTask被取消了, 那么就调用onCancelled()方法, 否则就会调用onPostExecute()方法, 可以看到doInBackground的反馈结果会传递给onPostExecute()方法.

通过源码分析, 可以确定从3.0开始, 默认情况下AsyncTask就是串行的. 通过一段代码测试.

|  |
| --- |
| @Override  public void onClick(View v) {  switch (v.getId()){  case R.id.btn\_main:  new MyAsync("任务\_1").execute("");  new MyAsync("任务\_2").execute("");  new MyAsync("任务\_3").execute("");  new MyAsync("任务\_4").execute("");  new MyAsync("任务\_5").execute("");  break;  }  }  private static class MyAsync extends AsyncTask<String, Integer, String>{  private static final String TAG = MyAsync.class.getSimpleName();  private final String mTaskName;  public MyAsync(String taskName){  mTaskName = taskName;  }    @Override  protected String doInBackground(String... params) {  SystemClock.sleep(3000);  return mTaskName;  }  @Override  protected void onPostExecute(String s) {  super.onPostExecute(s);  SimpleDateFormat df = new SimpleDateFormat("HH:mm:ss");  Log.e(TAG, s+" onPostExecute finish time: " +df.format(new Date()));  }  } |

给按钮添加一个点击触发, 点击的时候会触发五个AsyncTask的创建和执行 , 看一下结果



在5.0机器上测试所有的任务是串行执行的. 总共耗费了10秒. 而在2.x的版本所有的结束时间却都是一样的.

如果使用executeOnExecutor()那么结果看一下:

*/\*\**

\* 在版本3.0以上使用并行的方式开启

\*/

private void checkConcurrent() {

new MyAsync("任务\_1").executeOnExecutor(AsyncTask.THREAD\_POOL\_EXECUTOR,"");

new MyAsync("任务\_2").executeOnExecutor(AsyncTask.THREAD\_POOL\_EXECUTOR,"");

new MyAsync("任务\_3").executeOnExecutor(AsyncTask.THREAD\_POOL\_EXECUTOR,"");

new MyAsync("任务\_4").executeOnExecutor(AsyncTask.THREAD\_POOL\_EXECUTOR,"");

new MyAsync("任务\_5").executeOnExecutor(AsyncTask.THREAD\_POOL\_EXECUTOR,"");

}

ok, 按照你的需求可以选择使用哪种实现方式.

### HandlerThread

HandlerThread继承了Thread, 它是一种可以使用Handler的Thread, 它的实现也很简单, 就是run方法中通过Looper.prepare()来创建消息队列, 并通过Looper.loop()来开启消息循环, 这样在实际的使用中就允许在HandlerThread中创建Handler, 看一下run()方法.

|  |
| --- |
| @Override  public void run() {  mTid = Process.myTid();  Looper.prepare();  synchronized (this) {  mLooper = Looper.myLooper();  notifyAll();  }  Process.setThreadPriority(mPriority);  onLooperPrepared();  Looper.loop();  mTid = -1;  } |

从HandlerThread的实现来看, 它和普通的Thread有显著的不同之处. 普通的Thread主要用于在run方法中执行一个耗时任务; 而HandlerThread在内部创建了消息队列, 外界需要通过Handler的消息方式来通知HandlerThread执行一个具体的任务. HandlerThread是一个很有用的类, 在Android中一个具体使用场景就是IntentService.

由于HandlerThread#run()是一个无线循环方法, 因此当明确不需要再使用HandlerThread时, 最好通过quit()或者quitSafely()方法来终止线程的执行.

### IntentService

IntentService是一种特殊的Service, 它继承了Service并且它是一个抽象类, 因此需要创建子类才能使用. IntentService可以用于执行后台耗时任务, 当任务执行后会自动停止, 同时由于本质是服务的原因, 这导致了它的优先级比单纯的线程要高很多, 所以IntentService比较适合执行一些高优先级的后台任务.

IntentService封装了Handler和HandlerThread. 这是在onCreate()来实现的.

|  |
| --- |
| @Override  public void onCreate() {  *// TODO: It would be nice to have an option to hold a partial wakelock*  *// during processing, and to have a static startService(Context, Intent)*  *// method that would launch the service & hand off a wakelock.*  super.onCreate();  HandlerThread thread = new HandlerThread("IntentService[" + mName + "]");  thread.start();  mServiceLooper = thread.getLooper();  mServiceHandler = new ServiceHandler(mServiceLooper);  } |

当第一次启动的时候, onCreate()就会被调用, 内部就会创建一个HandlerThread, 然后使用它的Looper来构造一个Handler对象mServiceHandler. 这样通过 mServiceHandler发送的消息最终都会在HandlerThread中执行, 从这个角度看IntentService也可以用于执行后台任务.

每次启动IntentService的时候, 它的onStartCommand()方法就会被调用一次, IntentService在onStartCommand()中处理每个后台任务的Intent. 看看源码是如何处理外界的Intent的.

首先onStartCommand()调用了onStart()

|  |
| --- |
| @Override **public void** onStart(@Nullable Intent intent, **int** startId) {  Message msg = **mServiceHandler**.obtainMessage();  msg.**arg1** = startId;  msg.**obj** = intent;  **mServiceHandler**.sendMessage(msg); } |

这个方法只是通过mServiceHandler发送了一个消息, 所以这个消息会在HandlerThread中被处理. 消息收到后, 会将Intent对象传递给onHandlerIntent()方法去处理. 注意这个Intent对象和外界startService()参数传递内容是一样的. 通过Intent的参数就可以区分具体的后台任务, 这样在onHandlerIntent()方法中就可以对不同的后台任务做处理了.

当onHandlerIntent()方法执行完毕后, IntentService会通过stopSelf()方法来停止服务. 这里使用的方法是有参数的, 不使用无参数的stopSelf()是因为无参函数会立刻停止服务, 可能会导致还有没执行完的任务失效.

有参的stopSelf(int startId)在尝试 停止服务之前会判断最近启动的服务次数是否和startId这个参数值相等, 如果相等就立刻停止服务, 否则反之. 这个策略可以从AMS#stopServiceToken()方法的实现中找到依据.

IntentService#onHandleIntent()方法是一个抽象方法, 他需要我们在子类中实现, 它的作用是从Intent参数中区分具体的任务并执行这些任务. 如果目前只存在一个后台任务, 那么onHandleIntent()方法执行完成后, stopSelf(int startId)会立即停止服务; 如果目前存在多个后台任务, 那么当onHandleIntent()方法执行完最后一个任务时, stopSelf(int startId)才会停止任务. 另外由于每次执行一个后台任务都必须启动一次IntentService, 而其内部则通过消息的方式向HandlerThread请求执行任务, Handler中的Looper是顺序处理消息的, 这就意味着IntentService也是顺序执行后台任务的, 当有多个后台任务同时存在时, 这些后台任务会按照外界发起的顺序排队执行.

用代码来演示一遍:

可以看出, 三个后台任务是串行执行的, 他们的执行顺序就是发起请求的顺序. 当任务3完成了之后.KIntentService才真正的停止, 因为这是时候执行了onDestroy().

注意要在**manifest**声明该服务

|  |
| --- |
| **public class** KIntentService **extends** IntentService {  **private static final** String ***TAG*** = KIntentService.**class**.getSimpleName();  **public** KIntentService() {  **super**(***TAG***);  }  @Override  **protected void** onHandleIntent(@Nullable Intent intent) {  String task = intent.getStringExtra(**"task"**);  Log.*d*(***TAG***, **"receiver task :"**+task);  SystemClock.*sleep*(2000);  }  @Override  **public void** onDestroy() {  Log.*w*(***TAG***, **"onDestroy: 准备关闭"** );  **super**.onDestroy();  } }  *// 开启3个服务* Intent intent = **new** Intent(**this**, KIntentService.**class**); intent.putExtra(**"task"**, **"hi, 我是数据1"**); startService(intent); intent.putExtra(**"task"**, **"hi, 我是数据2"**); startService(intent); intent.putExtra(**"task"**, **"hi, 我是数据3"**); startService(intent); |

## Android中的线程池

概括一下线程池的优点:

1. 重用线程池中的线程, 避免因为线程的创建和销毁所带来的性能开销.
2. 能有效控制线程的最大并大数, 避免大量的线程之间因互相抢占系统资源而导致的阻塞现象.
3. 能够对线程进行简单的管理, 并提供定时执行以及指定间隔循环执行等能力.

Android中的线程池的概念来源于Java中的Executor, Executor是一个接口, 真正的线程池的实现为ThreadPoolExecutor. ThreadPoolExecutor提供了一系列参数来配制线程池, 通过不同的参数可以创建不同的线程池. 而从功能的特性来分的话可以分成四类. 下面说明.

### ThreadPoolExecutor

ThreadPoolExecutor是线程池的真正实现, 它的构造方法提供了一系列参数来配置线程池, 下面对构造方法中参数进行一下说明, 这些参数将会直接影响到线程池的功能特性.

|  |
| --- |
| public ThreadPoolExecutor(int corePoolSize,  int maximumPoolSize,  long keepAliveTime,  TimeUnit unit,  BlockingQueue<Runnable> workQueue) {  this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,  Executors.defaultThreadFactory(), defaultHandler);  } |

* corePoolSize: 线程池的核心线程数, 默认情况下, 核心线程会在线程池中一直存活, 即使都处于闲置状态. 如果将ThreadPoolExecutor#allowCoreThreadTimeOut属性设置为true, 那么闲置的核心线程在等待新任务到来时会有超时的策略, 这个时间间隔由keepAliveTime属性来决定. 当等待时间超过了keepAliveTime设定的值那么核心线程将会终止.
* maximumPoolSize: 线程池所能容纳的最大线程数, 当活动线程数达到这个数值之后, 后续的任务将会被阻塞.
* keepAliveTime: 非核心线程闲置的超时时长, 超过这个时长, 非核心线程就会被回收.allowCoreThreadTimeOut这个属性为true的时候, 这个属性同样会作用于核心线程.
* unit: 用于指定keepAliveTime参数的时间单位, 这是一个枚举, 常用的有TimeUtil.MILLISECONDS(毫秒), TimeUtil.SECONDS(秒)以及TimeUtil.MINUTES(分)
* workQueue: 线程池中的任务队列, 通过线程池的execute方法提交的Runnable对象会存储在这个参数中.
* threadFactory: 线程工厂, 为线程池提供创建新线程的功能. ThreadFactory是一个接口.

**线程池执行任务时大致遵循如下规则:**

1. 如果线程池中的线程数量未达到核心线程的数量, 那么会直接启动一个核心线程来执行任务.
2. 如果线程池中的线程数量已经达到或者超过核心线程的数量, 那么任务会被插入到任务队列中排队等待执行.
3. 如果在步骤2中无法将任务插入到任务队列中, 这通常是因为任务队列已满, 这个时候如果线程数量未达到线程池的规定的最大值, 那么会立刻启动一个非核心线程来执行任务.
4. 如果步骤3中的线程数量已经达到最大值的时候, 那么会拒绝执行此任务, ThreadPoolExecutor会调用RejectedExecution方法来通知调用者.

ThreadPoolExecutor的参数配置在AsyncTask中有明显的体现, 下面是其配置情况



这个配置后的规格是这样的

* 核心线程数等于CPU核心数 + 1;
* 线程池的最大线程数为CPU核心数的2倍 + 1;
* 核心线程无超时机制, 非核心线程在闲置时有超时机制,超时时间为1秒.
* 任务队列的容量为128

### 线程池的分类

**1. FixedThreadPool**

**通过Executors#newFixedThreadPool()方法来创建.** 它是一种线程数量固定的线程池, 当线程处于空闲状态时, 它们并不会被回收, 除非线程池关闭了. 当所有的线程都处于活动状态时, 新任务都会处于等待状态, 直到有线程空闲出来. 由于FixedThreadPool只有核心线程并且这些核心线程不会被回收, **这意味着它能够更加快速地响应外界的请求.**

**2. CachedThreadPool**  
**通过Executors#newCachedThreadPool()方法来创建.** 它是一种线程数量不定的线程池, 它只有非核心线程, 并且其最大值线程数为Integer.MAX\_VALUE. 这就可以认为这个最大线程数为任意大了. 当线程池中的线程都处于活动的时候, 线程池会创建新的线程来处理新任务, 否则就会利用空闲的线程来处理新任务. 线程池中的空闲线程都有超时机制, 这个超时时长为60S, 超过这个时间那么空闲线程就会被回收.

和FixedThreadPool不同的是, CachedThreadPool的任务队列其实相当于一个空集合, 这将导致任何任务都会立即被执行, 因为在这种场景下SynchronousQueue是无法插入任务的. SynchronousQueue是一个非常特殊的队列, 在很多情况下可以把它简单理解为一个无法存储元素的队列. **在实际使用中很少使用.这类线程比较适合执行大量的耗时较少的任务**

**3. ScheduledThreadPool**

**通过Executors#newScheduledThreadPool()方法来创建.** 它的核心线程数量是固定的, 而非核心线程数是没有限制的, 并且当非核心线程闲置时会立刻被回收掉. **这类线程池用于执行定时任务和具有固定周期的重复任务**

**4. SingleThreadExecutor**

**通过Executors#newSingleThreadPool()方法来创建.** 这类线程池内部只有一个核心线程, 它确保所有的任务都在同一个线程中按顺序执行. **这类线程池意义在于统一所有的外界任务到一个线程中, 这使得在这些任务之间不需要处理线程同步的问题**

# Bitmap的加载和Cache

由于Bitmap的特殊性以及Android对单个应用所施加的内存限制，比如16M，这导致加载Bitmap的时候很容易出现内存溢出。比如以下场景：

java.lang.OutofMemoryError:bitmap size exceeds VM budget

Android中常用的缓存策略也是很有意思，缓存策略一个通用的思想，可以用到很多场景中，比如在实际开发中经常需要用到Bitmap做缓存。通过缓存策略，我们不需要每次都从网络上请求图片或者从存储设备中加载图片，这样就极大地提高了图片的加载效率以及产品的用户体验。目前比较常用的缓存策略是LruCache和DiskLruCache，其中LruCache常被用做内存缓存，而DiskLruCache用做存储缓存。Lru是Least Recently Used的缩写，即最近最少使用算法，这种算法的核心思想：当缓存快满时，会淘汰近期最少使用的缓存目标，很显然Lru算法的思想是很容易被接受的。

## Bitmap的高效加载

先来简单介绍一下如何加载一个Bitmap, Bitmap在android中指的是一张图片, 可以是png格式也可以是jpg等其他常见的图片格式.

**那么如何加载一个图片?**首先BitmapFactory类提供了四种方法: decodeFile(), decodeResource(),decodeStream(), decodeByteArray(). 分别用于从文件系统, 资源文件, 输入流以及字节数组加载出一个Bitmap对象. 其中decodeFile和decodeResource又间接调用了decodeStream()方法, 这四类方法最终是在Android的底层实现的, 对应着BitmapFactory类的几个native方法.

**高效加载的Bitmap的核心思想:**采用BitmapFactory.Options来加载所需尺寸的图片. 比如说一个ImageView控件的大小为300\*300. 而图片的大小为800\*800. 这个时候如果直接加载那么就比较浪费资源, 需要更多的内存空间来加载图片, 这不是很必要的. 这里我们就可以先把图片按一定的采样率来缩小图片在进行加载. 不仅降低了内存占用,还在一定程度上避免了OOM异常. 也提高了加载bitmap时的性能.

**而通过Options参数来缩放图片**: 主要是用到了inSampleSize参数, 即采样率.

* 如果是**inSampleSize=1**那么和原图大小一样,
* 如果是**inSampleSize=2**那么宽高都为原图**1/2**, 而像素为原图的**1/4**, 占用的内存大小也为原图的**1/4**
* 如果是**inSampleSize=3**那么宽高都为原图**1/3**, 而像素为原图的**1/9**, 占用的内存大小也为原图的**1/9**
* 以此类推…..

**要知道Android中加载图片具体在内存中的占有的大小是根据图片的像素决定的, 而与图片的实际占用空间大小没有关系.而且如果要加载mipmap下的图片, 还会根据不同的分辨率下的文件夹进行不同的放大缩小.**

列举现在有一张图片像素为:**1024\*1024**, 如果采用ARGB8888(四个颜色通道每个占有一个字节,相当于1点像素占用4个字节的空间)的格式来存储.(这里不考虑不同的资源文件下情况分析) 那么图片的占有大小就是1024\*1024\*4那现在这张图片在内存中占用**4MB**.

如果针对刚才的图片进行inSampleSize=2, 那么最后占用内存大小为512\*512\*4, 也就是**1MB**

采样率的数值必须是大于1的整数是才会有缩放效果, 并且采样率同时作用于宽/高, 这将导致缩放后的图片以这个**采样率的2次方递减, 即内存占用缩放大小为1/(inSampleSize的二次方).** 如果小于1那么相当于=1的时候. 在官方文档中指出, inSampleSize的取值应该总是为2的指数, 比如1,2,4,8,16,32…如果外界传递inSampleSize不为2的指数, 那么系统会向下取整并选择一个最接近的2的指数来代替. 比如如果inSampleSize=3,那么系统会选择2来代替. **但是这条规则并不作用于所有的android版本, 所以可以当成一个开发建议**

整理一下开发中代码流程:

1. 将BitmapFactory.Options的inJustDecodeBounds参数设置为true并加载图片.
2. 从BitmapFactory.Options取出图片的原始宽高信息, 他们对应于outWidth和outHeight参数
3. 根据采样率的规则并结合目标View的所需大小计算出采样率inSampleSize
4. 将BitmapFactory.Options的inJustDecodeBounds参数设为false, 然后重新加载.

第一个步骤设置了一个参数, 这个参数的作用就是在加载图片的时候是否只是加载图片宽高信息而不把图片全部加载到内存. 所以这个操作是个**轻量级的**.

通过这些步骤就可以整理出以下的工具加载图片类调用decodeFixedSizeForResource()即可.

|  |
| --- |
| **public static** Bitmap decode(Resources res, **int** resId, **int** reqW, **int** reqH) {  **final** BitmapFactory.Options options = **new** BitmapFactory.Options();  options.**inJustDecodeBounds** = **true**;  BitmapFactory.*decodeResource*(res,resId ,options);  options.**inSampleSize** = *getSampleSize*(options, reqW, reqH);  options.**inJustDecodeBounds** = **false**;  **return** BitmapFactory.*decodeResource*(res,resId ,options); } **private static int** getSampleSize(BitmapFactory.Options options,**int** reqW, **int** reqH){  **if** (reqW == 0 || reqH == 0) {  **return** 1;  }  **final int** h = options.**outHeight**;  **final int** w = options.**outWidth**;  *LOG\_D*(**"origin, w= "** + w + **" h="** + h);  **int** inSampleSize = 1;  **while** ( h/inSampleSize >= reqH && w/inSampleSize >= reqW)  inSampleSize = inSampleSize<< 1;  **return** Math.*max*(1,inSampleSize/2); } |

当给ImageView设置的时候传入控件的大小, 就会自动转换返回. 可以看一下下面的两张图, 当加载一个分辨率很大的图片如果不使用此方法那么就出**程序崩溃。**

**了解一下一个原始图片到手机显示的最终占用内存大小=加载后的图片H\*加载后的图片W\*decode出来的通道模式（默认为4）**

**加载后的图片H = 原始H/图片所在的dpi文件夹 \* 真实设备的dpi值**

[请看计算实例](http://www.jianshu.com/p/56d7ca26b3b6)

## Android中的缓存策略

目前常用的一种缓存算法是**LRU(Least Recently Used)**, 最近最少使用算法. 核心思想: 当缓存存满时, 会优先淘汰那些近期最少使用的缓存对象. 采用LRU算法的缓存有两种: **LruCache**和**DiskLruCache**,**LruCache**用于实现内存缓存, **DiskLruCache**则充当了存储设备缓存, 当组合使用后就可以实现一个类似ImageLoader这样的类库.

### LruCache

**LruCache**是Android 3.1所提供的一个缓存类, 通过support-v4兼容包可以兼容到早期的Android版本

**LruCache**是一个泛型类, 它内部采用了一个LinkedHashMap以强引用的方式存储外界的缓存对象, 其提供了get和put方法来完成缓存的获取和添加的操作. 当缓存满了时, **LruCache**会移除较早使用的缓存对象, 然后在添加新的缓存对象. 普及一下各种引用的区别:

* 强引用: 直接的对象引用
* 软引用: 当一个对象只有软引用存在时, 系统内存不足时此对象会被gc回收
* 弱引用: 当一个对象只有弱引用存在时, 对象会随下一次gc时被回收

创建的缓存对象大小通过可用内存的1/8来进行分配, 并需要重写sizeOf()方法, sizeOf()是计算缓存对象的大小, 如果有需要还可以重写entryRemoved(),在**Lru**移除旧缓存的时候回调用此方法.

获取, 添加, 删除, 分别对应get, put, remove

### DiskLruCache

DiskLruCache用于实现存储设备缓存, 即磁盘缓存. 它通过将缓存对象写入文件系统从而实现缓存的效果.

#### DiskLruCache的创建

DiskLruCache并不能通过构造方法来创建, 他提供了open()方法用于创建自身, 如下所示

public static DiskLruCache open(File directory, int appVersion, int valueCount, long maxSize)

这个方法有四个参数:

1. File directory: 表示磁盘缓存在文件系统中的存储路径. 可以选择SD卡上的缓存目录, 具体是指/sdcard/Andriod/data/package\_name/cache目录, package\_name表示当前应用的包名, 当应用被卸载后, 此目录会一并删除掉. 也可以选择data目录下. 或者其他地方. **这里给出的建议:**如果应用卸载后就希望删除缓存文件的话 , 那么就选择SD卡上的缓存目录, 如果希望保留缓存数据那就应该选择SD卡上的其他目录.
2. int appVersion: 表示应用的版本号, 一般设为1即可. 当版本号发生改变的时候DiskLruCache会清空之前所有的缓存文件, 在实际开发中这个实用性不大.
3. int valueCount: 表示单个节点所对应的数据的个数, 一般设为1.
4. long maxSize: 表示缓存的总大小, 比如50MB, 当缓存大小超出这个设定值后, DiskLruCache会清除一些缓存而保证总大小不大于这个设定值

#### **DiskLruCache的缓存添加**

DiskLruCache的缓存添加的操作是通过Editor完成的, Editor表示一个缓存对象的编辑对象.

如果还是缓存图片为例子, 每一张图片都通过图片的url为key, 这里由于url可能会有特殊字符所以采用**url的md5值**作为key. 根据这个key就可以通过edit()来获取Editor对象, 如果这个缓存对象正在被编辑, 那么edit()就会返回null. 即DiskLruCache不允许同时编辑一个缓存对象.

当用.edit(key)获得了Editor对象之后. 通过editor.newOutputStream(0)就可以得到一个文件输出流. 由于之前open()方法设置了一个节点只能有一个数据. 所以在获得输出流的时候传入常量0即可.

有了文件输出流, 可以当网络下载图片时, 图片就可以通过这个文件输出流写入到文件系统上. 别忘了使用Bufferedxxxxx写完之后, 要通过Editor中commit()来提交写操作, 如果下载中发生异常, 那么使用Editor中abort()来回退整个操作.

#### **DiskLruCache的缓存查找**

和缓存的添加过程类似, 缓存查找过程也需要将url转换成key, 然后通过DiskLruCache#get()方法可以得到一个Snapshot对象, 接着在通过Snapshot对象即可得到缓存的文件输入流, 有了文件输入流, 自然就可以得到Bitmap对象. 为了避免加载图片出现OOM所以采用压缩的方式. 在前面对BitmapFactory.Options的使用说明了. **但是这中方法对FileInputStream的缩放存在问题**. 原因是FileInputStream是一种有序的文件流, 而两次decodeStream调用会影响文件的位置属性, 这样在第二次decodeStream的时候得到的会是null. 针对这一个问题, 可以通过文件流来得到它所对应的**文件描述符**, 然后通过BitmapFactory.decodeFileDescription()来加载一张缩放后的图片.

## ImageLoader的实现

一个好的ImageLoader应该具备以下几点:

* 图片的压缩
* 网络拉取
* 内存缓存
* 磁盘缓存
* 图片的同步加载
* 图片的异步加载

也可以利用picasso和glide这种图片加载类库, 使用**构建者**模式提供使用. 本文练习demo实现了链式调用设置控件,并制定url然后自动去获取. 相关联的类在load包中

图片的同步加载是指能够以同步的方式向调用者提供所加载的图片，这个图片可能是从内存缓存读取的，也可能是从磁盘缓存中读取的，还可能是从网络拉取的。

图片的异步加载是一个很有用的功能，很多时候调用者不想再单独的线程中以同步的方式来获取图片，这个时候ImageLoader内部需要自己在线程中加载图片并将图片设置所需的ImageView。图片压缩的作用更需要了，这是降低OOM概率的有效手段，ImageLoader必须合适地处理图片的压缩问题。

内存缓存和磁盘缓存是ImageLoader的核心，也是ImageLoader的意义所在，通过这两级缓存极大地提高了程序的效率并且有效地降低了对用户所造成的流量消耗，只有当这两级缓存都不可用时才需要从网络中拉取图片。

1. 图片压缩功能
2. 内存缓存和磁盘缓存
3. 同步加载和异步加载的接口设计 详细请看[随书源码](https://github.com/singwhatiwanna/android-art-res/blob/master/Chapter_12/src/com/ryg/chapter_12/loader/ImageLoader.java)

## ImageLoader的使用

### 照片墙效果

这里定义一个自定义的ImageView作为GridView的item布局控件, 之所以这样是想让图片的控件自动去使用设备达到**宽高相等的效果比较好看些，**这个实现很简单, 利用动态的宽度同时也当做高度即可实现预期的效果.

|  |
| --- |
| @Override **protected void** onMeasure(**int** widthMeasureSpec, **int** heightMeasureSpec) {  **super**.onMeasure(widthMeasureSpec, widthMeasureSpec); } |

# 综合技术- DL动态加载框架技术

归类到 性能优化

## Ref

[DL动态加载框架技术](http://blog.csdn.net/singwhatiwanna/article/details/40283117)

[罗列几个Android插件化开发框架](http://www.cnblogs.com/ligreat/p/6437564.html)

[知乎：怎么将 Android 程序做成插件化的形式？](https://www.zhihu.com/question/19981105)

[Android插件化从入门到放弃-最强合集](http://www.jianshu.com/p/353514d315a7)

[Android插件化：从入门到放弃](http://www.infoq.com/cn/articles/android-plug-ins-from-entry-to-give-up)

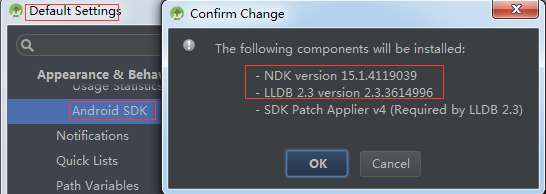
# JNI和NDK编程

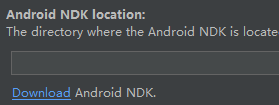
Java JNI本意为Java Native Interface(java本地接口), 是为方便java调用C或者C++等本地代码所封装的一层接口. 由于Java的跨平台性导致本地交互能力的不好, 一些和操作系统相关的特性Java无法完成, 于是Java提供了JNI专门用于和本地代码交互.

NDK是android所提供的一个工具合集, 通过NDK可以在Android中更加方便地通过JNI来访问本地代码. NDK还提供了交叉编译工具, 开发人员只需要简单的修改mk文件就可以生成特定的CPU平台的动态库. 好处如下:

* + 提供代码的安全性. so库反编译比较困难
  + 可以很方便地使用目前已有的C/C++开源库
  + 提高程序在某些特定情形下得执行效率, 但是不能明显提高Android程序的的性能.

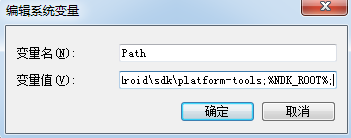
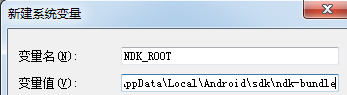
安装非常简单，可以用as的窗口下载，有两个入口





如果下载不了，访问不了官网，[否则自己网上去手动下载](https://developer.android.google.cn/ndk/downloads/index.html)

安装好的NDk一般位于你的sdk文件夹下的ndk-bundle。可以看到里面有ndk-build文件，然后将该路径配置到你系统变量的path里面去，将NDK\_ROOT追加到Path环境变量下;%NDK\_ROOT%



## JNI的开发流程

大体步骤就是在java中声明native方法, 然后调用c/c++实现的native的方法. 下面开始具体步骤

### Java中声明native方法

创建一个java类

|  |
| --- |
| public class HelloJni {  static {  System.*loadLibrary*("HelloJni");  }  public native String get();  public native void set(String str);  // 主函数  public static void main(String arg[]) {  HelloJni jniTest = new HelloJni();  System.*out*.println(jniTest.get());  jniTest.set("java写入数据");  } } |

首先是通过静态代码块先加载so库, 然后声明两个native方法提供使用. 这两个方法需要在JNI中实现. 加载so库名称填入的虽然是HelloJni, 但是so库全名称应该是libHelloJni.so,这是加载so库的规范.

### Java->class->JNI头文件

在Studio中进行终端命令行工具.执行如下命令，将在操作的路径下生成c语言头文件k\_art\_ch14Ndk\_HelloJni.h。并把头文件移动到app/src/main/jni下。



|  |
| --- |
| /\* DO NOT EDIT THIS FILE - it is machine generated \*/ #include <jni.h> /\* Header for class k\_art\_ch14Ndk\_HelloJni \*/  #ifndef \_Included\_k\_art\_ch14Ndk\_HelloJni #define \_Included\_k\_art\_ch14Ndk\_HelloJni /\*这个宏定义是必须的, 作用是指定extern”C”内部的函数采用C语言的命名风格来编译.  如果设定那么当JNI采用C++来实现时, 由于C/C++编译过程对函数的命名风格不同,  这将导致JNI在链接时无法根据函数名找到具体的函数, 那么JNI调用肯定会失效.\*/ #ifdef \_\_cplusplus **extern** "C" { #endif /\*  \* Class: k\_art\_ch14Ndk\_HelloJni  \* Method: get  \* Signature: ()Ljava/lang/String;  \*/  //函数名:格式遵循:Java\_包名\_类名\_方法名包名之间的.分割全部替换成\_分割.  /\* 参数: jstring是代表String类型参数. 具体的类型关系后面会说明.  JNIEnv \*: 表示一个指向JNI环境的指针, 可以通过它来访问JNI提供的方法.  jobject: 表示java对象中的this.  JNIEXPORT和JNICALL: 这是JNI种所定义的宏, 可以在jni.h这个头文件查到\*/ JNIEXPORT jstring JNICALL Java\_k\_art\_ch14Ndk\_HelloJni\_get  (JNIEnv \*, jobject);  /\*  \* Class: k\_art\_ch14Ndk\_HelloJni  \* Method: set  \* Signature: (Ljava/lang/String;)V  \*/ JNIEXPORT **void** JNICALL Java\_k\_art\_ch14Ndk\_HelloJni\_set  (JNIEnv \*, jobject, jstring);  #ifdef \_\_cplusplus } #endif #endif |

### 实现JNI.cpp

JNI方法是指的Java中声明的native方法, 这里可以选择c++和c来实现. 过程都是类似的. 只有少量的区别, 这里两种都实现一下

在工程的主目录创建一个子目录, 名称任意, 然后将之前通过javah命令生成的.h头文件复制到创建的目录下, 接着创建k\_art\_ch14Ndk\_HelloJni.cpp实现如下:

|  |
| --- |
| #include "k\_art\_ch14Ndk\_HelloJni.h" JNIEXPORT jstring JNICALL Java\_k\_art\_ch14Ndk\_HelloJni\_get(JNIEnv \*env, jobject jObj){  // return env->NewStringUTF("I`m Str from jni libs!");  //return (\*env)->NewStringUTF(env, "Hello world from jni)");//C语言格式，文件名应为xxx.c  **return** env->NewStringUTF((**char** \*)"Hello JIN");//C++格式，文件名应为xxx.cpp  } |

### 编译配置，编译C/C++文件为.so库。

在jni目录下新建Android.mk(必须是这个名称Android.mk)文件，如下图所示：

Android.mk指定.so文件名称：

|  |
| --- |
| LOCAL\_PATH := $(call my-dir)  include $(CLEAR\_VARS)  LOCAL\_MODULE := HelloJni //要生成的so库名，但实际为libHelloJni.so  LOCAL\_SRC\_FILES := k\_art\_ch14Ndk\_HelloJni.cpp //指定编译的文件  include $(BUILD\_SHARED\_LIBRARY) |

Application.mk指定生成不同平台的.so库：

# 注释掉了，不写会生成全部支持的平台。目前支持：

# APP\_ABI := armeabi arm64-v8a armeabi-v7a mips mips64 x86 x86\_64

三种常用平台，不写则全部生成（armeabi arm64-v8a armeabi-v7a mips mips64 x86 x86\_64）。架构不同有的不兼容。不写则全部生成。

也可以在.\app\build.gradle指定：

android.defaultConfig.ndk {

moduleName "HelloJni"

ldLibs "log", "z", "m"

// abiFilters "armeabi", "armeabi-v7a", "x86"

abiFilters "armeabi-v7a"

}

[Android Studio上进行NDK编程之Hello world](http://blog.csdn.net/mingyueyixi/article/details/65449723)

# 性能优化

归类到 性能优化一书

# 参考

[Android开发艺术探索详细笔记](http://blog.csdn.net/column/details/13457.html)1-5

[Android开发艺术探索读书笔记（二）](http://blog.csdn.net/amurocrash/article/details/48729313)

[原书源码](http://download.csdn.net/download/gdutxiaoxu/9329545)

[第四章](http://blog.csdn.net/qq_26787115/article/details/53385645)View的工作原理

[第6章](http://blog.csdn.net/qq_26787115/article/details/58681776)

[第七章](http://blog.csdn.net/qq_26787115/article/details/70473861)

[第8章](http://blog.csdn.net/qq_26787115/article/details/71915772?locationNum=6&fps=1)

[第9章简书](http://www.jianshu.com/p/e702540367c7)不够详细，[另一篇的第一部分](http://blog.csdn.net/zizidemenghanxiao/article/details/50639025)

[第10 章 Android 的消息机制](https://wenku.baidu.com/view/a221e2bc31b765ce050814f0.html)

11-12-[Android的线程和线程池抄书系列](http://szysky.com/2016/08/22/%E3%80%8AAndroid-%E5%BC%80%E5%8F%91%E8%89%BA%E6%9C%AF%E6%8E%A2%E7%B4%A2%E3%80%8B-11-Android%E7%9A%84%E7%BA%BF%E7%A8%8B%E5%92%8C%E7%BA%BF%E7%A8%8B%E6%B1%A0/)

[notebook: Android开发艺术探索 第12章 Bitmap的加载和Cache](http://www.jianshu.com/p/765640fe474a)

[第14章JNI和NDK编程](http://szysky.com/2016/08/26/%E3%80%8AAndroid-%E5%BC%80%E5%8F%91%E8%89%BA%E6%9C%AF%E6%8E%A2%E7%B4%A2%E3%80%8B-14-JNI%E5%92%8CNDK%E7%BC%96%E7%A8%8B/)

# QA

## 1. scroller = new Scroller(getContext());

private void smoothScrollTo(int destX,int destY){

int scrollX = getScrollX();

int delta = destX - scrollX;

//1000ms内滑向destX,效果就是慢慢的滑动

scroller.startScroll(scrollX,0,delta,0,1000);

invalidate();

}

@Override

public void computeScroll() {

if(scroller.computeScrollOffset()){

scrollTo(scroller.getCurrX(),scroller.getCurrY());

postInvalidate();

}

}

## android的设计模式借鉴，

mask；代码效率怎么提高，命名规范

/\*\* @hide \*/方法怎么使用

## android.R.id.content)如何隐藏高度

1. onMeasure，onLayout工作原理
2. @MeasureSpecMode int mode
3. DecorView，phonewindow源码看不到？
4. **直接继承View的自定义控件需要重写onMeasure**
5. **自动强制添加桌面小部件？**
6. 定义小组件的实现类，不work？？？？？？
7. remoteView性能如何
8. UserHandle.myUserId()有啥用啊，~~RemoteViews支持的类型为啥很有限~~，setPendingIntentTemplate，以及setonClickFillinIntent
9. super.onDestroy();  
   unregisterReceiver(receiver);

位置关系

1. 适配专题：Drawable
2. 帧动画比较简单，但是容易OOM，如何解决追加吧

## Ch10

new Thread为何没有抛出异常

Looper如何用ThreadLocal，为什么可以取代全局的哈希表供Handler查找指定线程的Looper

什么时候执行自己定义的代码，看不到呢》？

Ch

12

options.**inJustDecodeBounds 为什么要可以控制图片decode，执行两次不累么？**

* 一些和操作系统相关的特性Java无法完成？比如说？指定CPU核心数目
* NDK为何没有代码提示呢？

# Task

ThreadLocal有道更新到这个上面来

为当前线程创建Looper来fix，Can't create handler inside thread that has not called Looper.prepare()

prepareMainLooper为什么在任何地方都可以获取主线程的looper

什么时候推出looper

为什么徐亚提出looper，thread和looper是什么关系

**public** Handler(Looper looper)

实现什么特殊功能

把java编程思想的笔记整理出来吧，用gitbook？

AsyncTask重新学习？把有道云的笔记整理取来

## Ch11

核心线程无超时机制, 非核心线程区别是啥？

11.3.2. 线程池的分类具体要试用一下。

# 启发

1. 指南针360度是否可以用属性动画？
2. 属性动画采用的是反射，为何效率还比view动画高？
3. 由于Android开发规范的限制，我们并不能在子线程中访问UI控件，否则就会触发程序异常：子线程耗时，可能10s更新ui，然而用户第5s就点击了ui，难以控制。
4. **DiskLruCache可以作为日志清理！**

# 目录拷贝

**图书目录：**  
第1章 Activity的生命周期和启动模式 / 1  
1.1 Activity的生命周期全面分析 / 1  
1.1.1 典型情况下的生命周期分析 / 2  
1.1.2 异常情况下的生命周期分析 / 8  
1.2 Activity的启动模式 / 16  
1.2.1 Activity的LaunchMode / 16  
1.2.2 Activity的Flags / 27  
1.3 IntentFilter的匹配规则 / 28  
第2章 IPC机制 / 35  
2.1 Android IPC简介 / 35  
2.2 Android中的多进程模式 / 36  
2.2.1 开启多进程模式 / 36  
2.2.2 多进程模式的运行机制 / 39  
2.3 IPC基础概念介绍 / 42  
2.3.1 Serializable接口 / 42  
2.3.2 Parcelable接口 / 45  
2.3.3 Binder / 47  
2.4 Android中的IPC方式 / 61  
2.4.1 使用Bundle / 61  
2.4.2 使用文件共享 / 62  
2.4.3 使用Messenger / 65  
2.4.4 使用AIDL / 71  
2.4.5 使用ContentProvider / 91  
2.4.6 使用Socket / 103  
2.5 Binder连接池 / 112  
2.6 选用合适的IPC方式 / 121  
第3章 View的事件体系 / 122  
3.1 View基础知识 / 122  
3.1.1 什么是View / 123  
3.1.2 View的位置参数 / 123  
3.1.3 MotionEvent和TouchSlop / 125  
3.1.4 VelocityTracker、GestureDetector和Scroller / 126  
3.2 View的滑动 / 129  
3.2.1 使用scrollTo/scrollBy / 129  
3.2.2 使用动画 / 131  
3.2.3 改变布局参数 / 133  
3.2.4 各种滑动方式的对比 / 133  
3.3 弹性滑动 / 135  
3.3.1 使用Scroller / 136  
3.3.2 通过动画 / 138  
3.3.3 使用延时策略 / 139  
3.4 View的事件分发机制 / 140  
3.4.1 点击事件的传递规则 / 140  
3.4.2 事件分发的源码解析 / 144  
3.5 View的滑动冲突 / 154  
3.5.1 常见的滑动冲突场景 / 155  
3.5.2 滑动冲突的处理规则 / 156  
3.5.3 滑动冲突的解决方式 / 157  
第4章 View的工作原理 / 174  
4.1 初识ViewRoot和DecorView / 174  
4.2 理解MeasureSpec / 177  
4.2.1 MeasureSpec / 177  
4.2.2 MeasureSpec和LayoutParams的对应关系 / 178  
4.3 View的工作流程 / 183  
4.3.1 measure过程 / 183  
4.3.2 layout过程 / 193  
4.3.3 draw过程 / 197  
4.4 自定义View / 199  
4.4.1 自定义View的分类 / 200  
4.4.2 自定义View须知 / 201  
4.4.3 自定义View示例 / 202  
4.4.4 自定义View的思想 / 217  
第5章 理解RemoteViews / 218  
5.1 RemoteViews的应用 / 218  
5.1.1 RemoteViews在通知栏上的应用 / 219  
5.1.2 RemoteViews在桌面小部件上的应用 / 221  
5.1.3 PendingIntent概述 / 228  
5.2 RemoteViews的内部机制 / 230  
5.3 RemoteViews的意义 / 239  
第6章 Android的Drawable / 243  
6.1 Drawable简介 / 243  
6.2 Drawable的分类 / 244  
6.2.1 BitmapDrawable / 244  
6.2.2 ShapeDrawable / 247  
6.2.3 LayerDrawable / 251  
6.2.4 StateListDrawable / 253  
6.2.5 LevelListDrawable / 255  
6.2.6 TransitionDrawable / 256  
6.2.7 InsetDrawable / 257  
6.2.8 ScaleDrawable / 258  
6.2.9 ClipDrawable / 260  
6.3 自定义Drawable / 262  
第7章 Android动画深入分析 / 265  
7.1 View动画 / 265  
7.1.1 View动画的种类 / 265  
7.1.2 自定义View动画 / 270  
7.1.3 帧动画 / 272  
7.2 View动画的特殊使用场景 / 273  
7.2.1 LayoutAnimation / 273  
7.2.2 Activity的切换效果 / 275  
7.3 属性动画 / 276  
7.3.1 使用属性动画 / 276  
7.3.2 理解插值器和估值器 / 280  
7.3.3 属性动画的监听器 / 282  
7.3.4 对任意属性做动画 / 282  
7.3.5 属性动画的工作原理 / 288  
7.4 使用动画的注意事项 / 292  
第8章 理解Window和WindowManager / 294  
8.1 Window和WindowManager / 294  
8.2 Window的内部机制 / 297  
8.2.1 Window的添加过程 / 298  
8.2.2 Window的删除过程 / 301  
8.2.3 Window的更新过程 / 303  
8.3 Window的创建过程 / 304  
8.3.1 Activity的Window创建过程 / 304  
8.3.2 Dialog的Window创建过程 / 308  
8.3.3 Toast的Window创建过程 / 311  
第9章 四大组件的工作过程 / 316  
9.1 四大组件的运行状态 / 316  
9.2 Activity的工作过程 / 318  
9.3 Service的工作过程 / 336  
9.3.1 Service的启动过程 / 336  
9.3.2 Service的绑定过程 / 344  
9.4 BroadcastReceiver的工作过程 / 352  
9.4.1 广播的注册过程 / 353  
9.4.2 广播的发送和接收过程 / 356  
9.5 ContentProvider的工作过程 / 362  
第10章 Android的消息机制 / 372  
10.1 Android的消息机制概述 / 373  
10.2 Android的消息机制分析 / 375  
10.2.1 ThreadLocal的工作原理 / 375  
10.2.2 消息队列的工作原理 / 380  
10.2.3 Looper的工作原理 / 383  
10.2.4 Handler的工作原理 / 385  
10.3 主线程的消息循环 / 389  
第11章 Android的线程和线程池 / 391  
11.1 主线程和子线程 / 392  
11.2 Android中的线程形态 / 392  
11.2.1 AsyncTask / 392  
11.2.2 AsyncTask的工作原理 / 395  
11.2.3 HandlerThread / 402  
11.2.4 IntentService / 403  
11.3 Android中的线程池 / 406  
11.3.1 ThreadPoolExecutor / 407  
11.3.2 线程池的分类 / 410  
第12章 Bitmap的加载和Cache / 413  
12.1 Bitmap的高效加载 / 414  
12.2 Android中的缓存策略 / 417  
12.2.1 LruCache / 418  
12.2.2 DiskLruCache / 419  
12.2.3 ImageLoader的实现 / 424  
12.3 ImageLoader的使用 / 441  
12.3.1 照片墙效果 / 441  
12.3.2 优化列表的卡顿现象 / 446  
第13章 综合技术 / 448  
13.1 使用CrashHandler来获取应用的crash信息 / 449  
13.2 使用multidex来解决方法数越界 / 455  
13.3 Android的动态加载技术 / 463  
13.4 反编译初步 / 469  
13.4.1 使用dex2jar和jd-gui反编译apk / 470  
13.4.2 使用apktool对apk进行二次打包 / 470  
第14章 JNI和NDK编程 / 473  
14.1 JNI的开发流程 / 474  
14.2 NDK的开发流程 / 478  
14.3 JNI的数据类型和类型签名 / 484  
14.4 JNI调用Java方法的流程 / 486  
第15章 Android性能优化 / 489  
15.1 Android的性能优化方法 / 490  
15.1.1 布局优化 / 490  
15.1.2 绘制优化 / 493  
15.1.3 内存泄露优化 / 493  
15.1.4 响应速度优化和ANR日志分析 / 496  
15.1.5 ListView和Bitmap优化 / 501