### Activity的生命周期和启动模式

#### 正常情况下的生命周期

**Activity从启动到退出**

onCreate——onStart——onResume

onPause——onStop——onDestroy

**Activity1启动Activity2**

onPause :Activity1

onCreate :Activity2

onStart :Activity2

onResume :Activity2

onStop :Activity1（若Activity2为透明主题，则此处的onStop不会有！）

**从Activity2返回到Activity1**

onPause :Activity2

onRestart :Activity1

onStart :Activity1

onResume :Activity1

onStop :Activity2

onDestroy :Activity2

#### 异常情况下的生命周期

系统配置发生变化（屏幕尺寸、屏幕方向发生改变），默认情况下，销毁Activity后，会再重新创建。

被销毁时：onSaveInstanceState

重新创建时：onRestoreInstanceState

但是如果配置了android:configChanges则不会被重新创建。

系统配置改变时：onConfigurationChanged

#### Intent-filter匹配规则

如果有多个Intent-filter，只需要匹配其中任意一个即可启动Activity。

但是一个Intent-filter中包含多种类，如<action>、<category>、<data>都有，则这几类都需要同时匹配上，才能启动Activity。

如果一个Intent-filter中的<category>有多个，则给intent添加category时，所有的category都必须在<category>列表中，才能匹配上从而启动Activity。

#### 启动模式

Standard （缺省）

每次启动，创建新的Activity。

singleTop

栈顶复用（如果不在栈顶，则创建新的。如果在栈顶，则不创建，调用onNewIntent。）

singleTask

栈内复用（无论在不在栈顶，都不会创建新的。将其上方的Activity全部出栈，在调用自身的onNewIntent。）

singleInstance

系统为其单独创建一个任务栈，其独享此栈。

故系统中仅存在一个它的实例，除非任务栈被销毁，否则不会创建新的实例。

### **IPC机制**

#### **多进程：”:remote”和”xxx.remote”的区别？**

:remote 其他应用的组件不可以和它跑在同一个进程，而xxx.remote则可以通过shareUserId的方式让其他应用的组件和它跑在同一个进程，从而互相访问对方的私有数据。

#### **Parcelable和Serializable**

Parcelable主要用在内存序列化上，而如果要用于序列化到存储设备，或者是在网络上传输，则建议使用Serializable过程更显方便。

#### **Binder**

Transact() 客户端发请求 。如果是同一进程，则不会有transact()的调用。

onTransact() 服务端调用完成数据处理或服务

AIDL不是Binder的必需品，只是提供代码规范化及开发速度。客户端实现可以将IXXX.java中的Stub单独提成一个类，代码和AIDL的方式完成一致，然后服务端使用相同的Stub来创建对象并在服务端的Service的onBind()中返回即可。

##### 远程服务出现异常，导致客户端连接中断怎们处理？

给binder设置死亡代理，一旦出现binderDied，重新绑定远程Service。涉及方法和代码如下：

mDeathRecipient = new IBinder.DeathRecipient(){

@Override

public void binderDied() {

......

unlinkToDeath(mDeathRecipient , 0);

......

// 这里重新绑定远程Service

......

}

}

mService = IMessageBoxManager.Stub.asInterface(binder);

Binder.linkToDeath(mDeathRecipient, 0);

#### **Android中的IPC方式有哪些？**

##### 使用Bundle

##### **文件共享（进程间通信中不建议使用SharedPreference）**

##### **Messenger（Message + Binder + Handler）**

##### **AIDL**

**这是进程间通信方式的首选！（划重点）**

##### **5、Content Provider**

>>URL

schem://android:authories/table/...

清单文件中指定android:authorites，唯一标识此Provider。

指定权限

指定为单独的进程

##### **6、Socket**

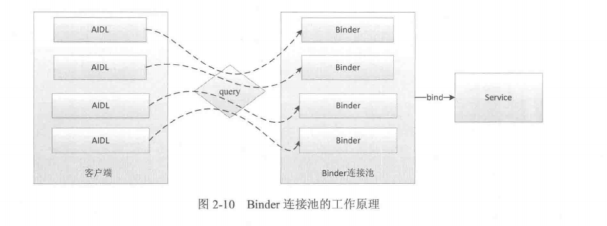
TCP套接字 UDP套接字

客户端 服务端

字节流传输

输入流 输出流

##### **Binder连接池**



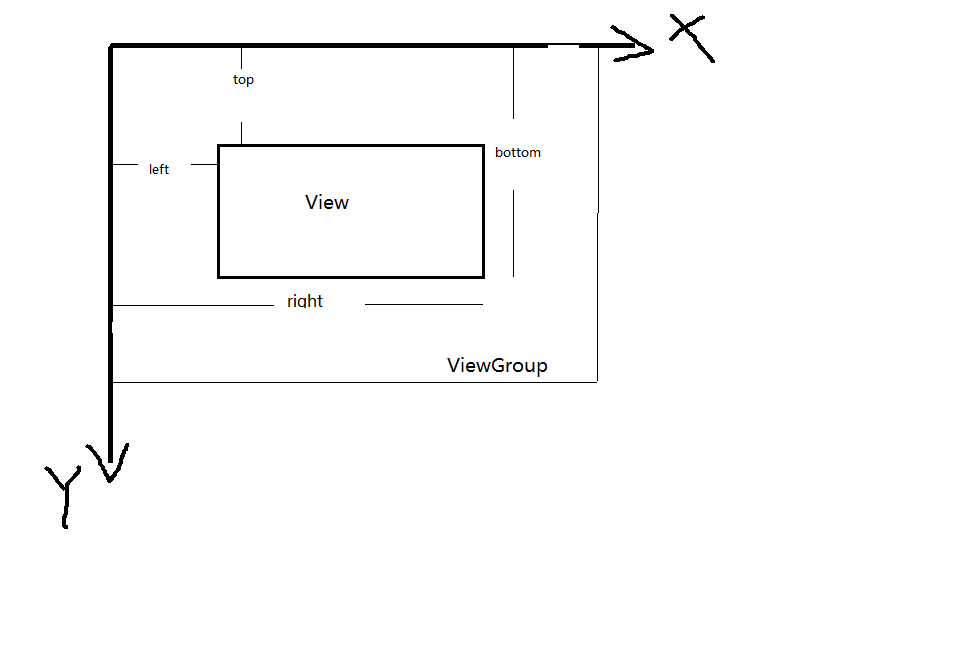
服务端提供一个queryBinder接口

在一个应用内，多个AIDL拥有同一个BinderPool的单例。只需要和服务端连接一次（发生在构造时，当然异常断开除外，有重连机制），连接后获得一个BinderPoolImpl（相当于是过渡），通过这个BinderPoolImpl要去获取对应某个业务模块的Binder，然后再通过这个Binder去调用远程服务的相关业务方法。

**这就实现了一个Service满足了多个AIDL的需求！**

### View的事件体系

##### **1、View的基础知识**



**x = left + translationX**

**y = top + translationY**

以上几个都是View的属性

left和top是不会发生改变的初始的值，x和y是随translationX和translationY变化而变化的。

##### **View的滑动**

滑动1、scrollTo和scrollBy

scrollTo(int x, int y) 基于给定参数的绝对滑动

scrollBy(int x, int y) 基于当前位置的相对滑动

**scrollTo和scrollBy这两个方法只能将View的内容移动，而不能将View本身移动，谨记！！！**

滑动2、动画

**View动画、属性动画可以实现View本身的移动。**

但两者是有区别的：

View动画只是对View的影像做操作，View本身并没有改变。（如新位置上会有事件无响应等问题！）

而属性动画则是对View本身的更改。

滑动3、改变View的布局参数

获取View的LayoutParams，设置相关的布局属性，在请求父容器重新布局。

参考下面的代码：

LayoutParams params = view.getLayoutParams();

params.width += 100;

params.leftMargin += 100;

view.requestLayout(); //或者view.setLayoutParams(params);

##### **弹性滑动（Smooth Scroll）**

**使用Scroller（View内容的滑动）**

Scroller本身并不能实现滑动，而是配合自定义View的computeScroll()这个方法，通过不断让View重绘而重复调用scrollTo()来完成滑动的工作。

所以个人认为，Scroller只是为执行滑动而提供的计算工具。

**通过动画**

用值动画（ValueAnimator）来控制fraction的变化，进而控制每次调用scrollTo(x, y)时参数值的变化，达到滑动效果。

参考下面的代码：

Final int startX = 0;

Final int deltaX = 100;

ValueAnimator animator = ValueAnimator.ofInt(0, 1).setDuration(1000);

Animator.addUpdateListener(new AnimatorUpdateListener(){

@Override

Public void onAnimationUpdate(ValueAnimator animator) {

Float fraction = animator.getAnimatedFraction();

mButton1.scrollTo(startx+deltaX\*fraction, 0);

}

});

**使用Handler#postDelayed或者View#postDelayed的方式**

比如1000ms内，分成30帧来执行scrollTo完成滑动，就是每33ms会发送一次消息来重复调用。

##### **View的事件分发机制**

MotionEvent是指的点击事件。

Activity——Window——DecorView——rootView（ViewGroup）——...

dispatchTouchEvent()

onTouchEvent()

**三类事件：**

ACTION\_DOWN

ACTION\_MOVE

ACTION\_UP

**事件传递机制的11个结论：**

1. 同一个时间序列是指从手指接触屏幕的那一刻起，到手指离开屏幕的那一刻结束，在这个过程中所产生的一系列事件（DOWN/MOVE/UP），这个事件序列以DOWN事件开始，中间含有数量不定的MOVE事件，最终以UP事件结束。
2. 正常情况下，一个事件序列只能被一个View拦截。这一条的原因可以参考（3），因为一旦一个元素拦截了某次事件，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View将本该自己处理的事件通过onTouchEvent强行传递给其他View处理。

（关于拦截的）

1. 某个View一旦决定拦截，那么这一个事件序列都只能有他来处理（如果事件序列能够传递给他的话），并且它的onInterceptTouchEvent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他来处理，因此就不需要拦截其他的事件了，也就不用再调用这个View的onInterceptTouchEvent去询问他是否要拦截了。

（关于拦截的）

1. 某个View一旦开始处理事件，如果它不消耗ACTION\_DOWN事件（onTouchEvent返回了false），那么同一事件序列中的其他事件就都不会再交给它来处理，并且事件将重新交给它的父元素去处理，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。

（关于处理的）

1. 如果View不消耗除ACTION\_DOWN以外的其他事件（只消耗DOWN事件不消耗其他事件也不行），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。

（关于处理的）

1. 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。（事件传递过来，没法拦截那只能处理喽！）

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就返回true。

10、onClick会发生的前提是当前View是可点击的，并且它收到了DOWN和UP的事件。

1. 事件传递过程是由外向内的，即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requestDisallowInterceptTouchEvent方法可以在子元素中干预父元素的事件分发过程，但是ACTION\_DOWN事件除外。（干预不了父元素的ACTION\_DOWN事件！）

##### **View的滑动冲突**

界面中只要内外两层同时可以滑动，就会产生滑动冲突。

不难，有固定套路可以解决这个问题！

场景1 外部和内部滑动方向不一致

场景2 内部和外部滑动方向一致

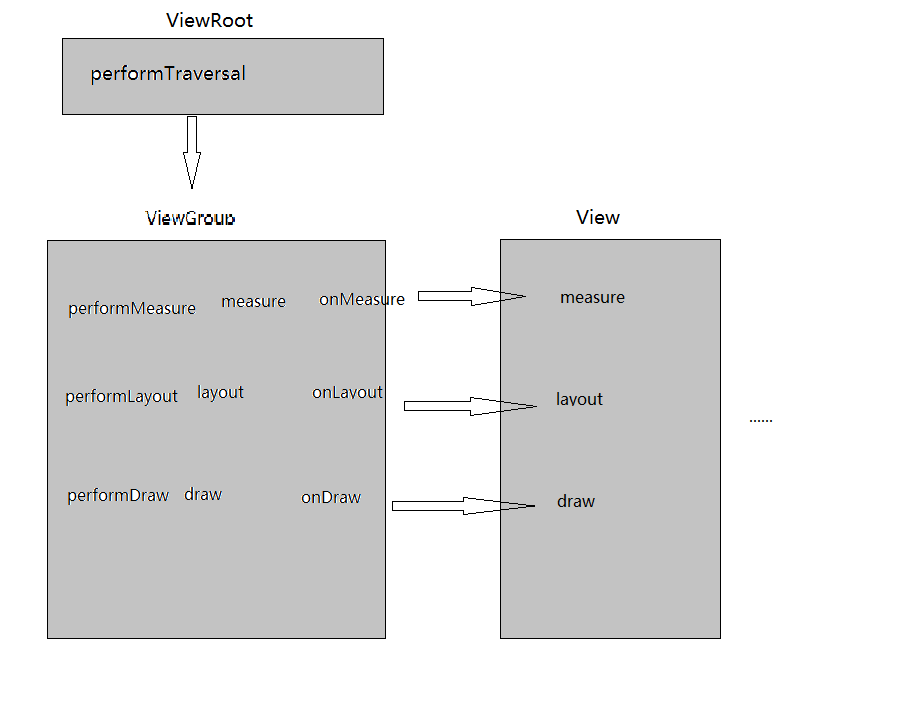
场景3 场景1和场景2两种的嵌套

外部拦截法

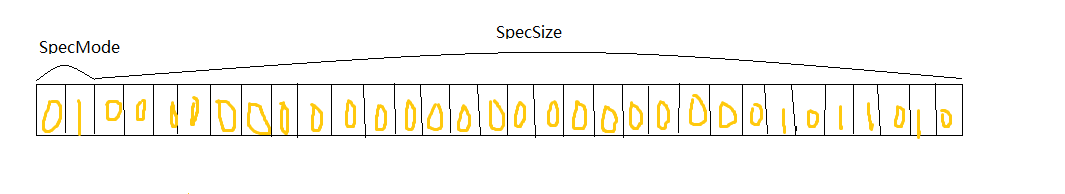
内部拦截法

### View的工作原理

##### **1、ViewRoot & DecorView**



##### **理解MeasureSpec**



MeasureSpec = SpecMode + SpecSize （由SpecMode和SpecSize两部分组成）

SpecMode分三类：UNSPECIFIED | EXACTLY | AT\_MOST

**MeasureSpec和LayoutParams的对应关系：**

View的MeasureSpec由父容器的MeasureSpec和View自身的LayoutParams一同来决定。

参看ViewGroup#measureChildWithMargins()和getChildMeasureSpec()

##### **View的工作流程**

**》measure过程**

measure完成后，可以通过getMeasuredWidth/getMeasuredHeight正确获取View的测量宽/高。**但是比较好的习惯是在onLayout中去获取。**

正确获取宽高的四种方法：

（1）Activity/View#onWindowFocusChanged

（2）view.post(runnable)

（3）ViewTreeObserver

（4）view.measure() 不推荐

**》layout过程**

测量宽/高：View#getMeasuredWidth()/View#getMeasuredHeight()。

最终宽/高：View#getWidth()/View#getHeight()。

测量宽高的赋值时机要早于最终宽高的赋值。前者是在View的measure过程，后者则是形成于View的layout过程。

在View的默认实现中，View的测量宽/高 == View的最终宽/高。

特殊情况下的实现会导致两者不相等：重写View#onLayout

**》draw过程**

（1）绘制背景background.draw(canvas)

（2）绘制自（onDraw）

（3）绘制children（dispatchDraw）

（4）绘制装饰（onDrawScrollBars）

》setWillNotDraw(boolean willNotDraw)

如果继承ViewGroup的自定义View不需要绘制任何内容，则启动该标记位，置为true。如果需要绘制内容，则应该关闭此标记位，而默认就是关闭的。

##### **自定义View**

**》自定义View的分类**

继承View （需要自己支持wrap\_content，padding也需要自己处理）

继承ViewGroup （需要分别处理自身及子元素的测量和布局两个过程）

继承特定View（如TextView）

继承特定ViewGroup（如LinearLayout）

**》自定义View须知**

（1）处理wrap\_content

（2）最好处理padding

（3）不要Handler

（4）需要停止线程或动画，考虑View#onDetachedFromWindow

（5）有嵌套滑动时，处理好滑动冲突。

**》自定义View的示例**

**》自定义View的思想**

### 理解RemoteViews

##### **RemoteViews的应用**

**》RemoteViews在通知栏上的应用**

**》RemoteViews在桌面小部件上的应用**

**》PendingIntent概述**

PendingIntent：即将发生的意图，在将来的某个不确定时刻发生。Intent则是即刻发生的意图。

PendingIntent的典型使用场景：

给RemoteViews添加单击事件。（setOnClickPendingIntent(PendingIntent)）

PendingIntent支持三种待定意图，对应其三个静态方法：

启动Activity：static getActivity(context, requestCode, intent, flags)

启动Service：static getService(context, requestCode, intent, flags)

发送广播：static getBroadcast(context, requestCode, intent, flags)

flags常见的类型有：

FLAG\_ONE\_SHOT

FLAG\_NO\_CREATE

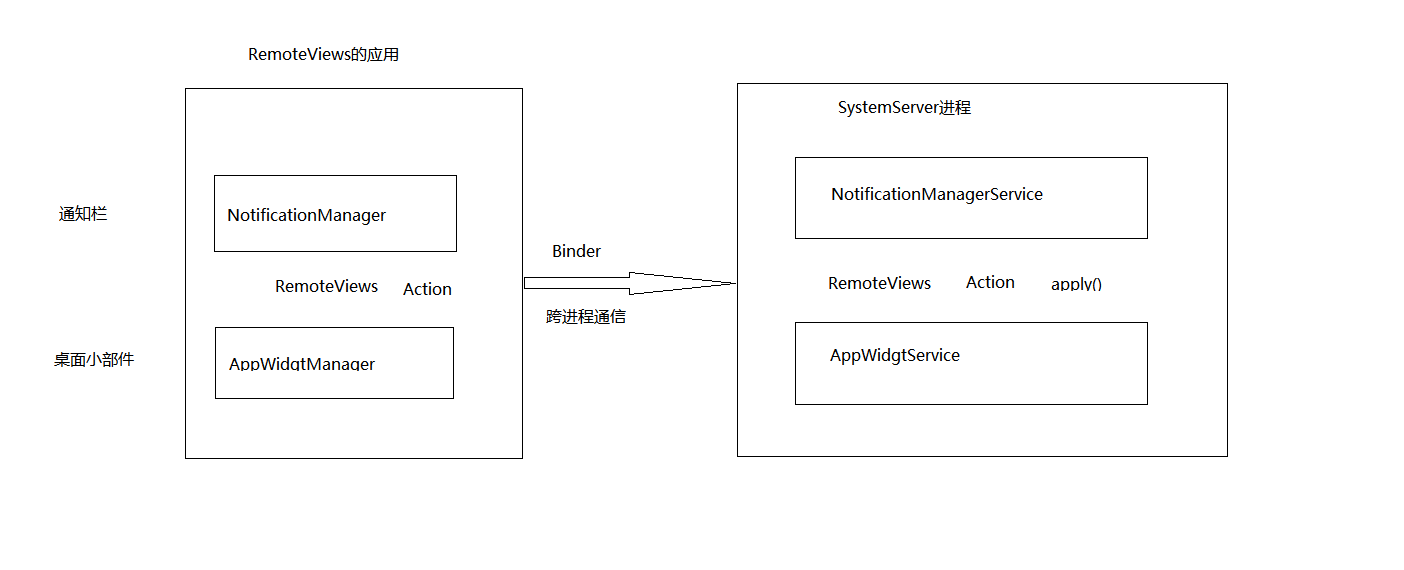
FLAG\_CACEL\_CURRENT

FLAG\_UPDATE\_CURRENT

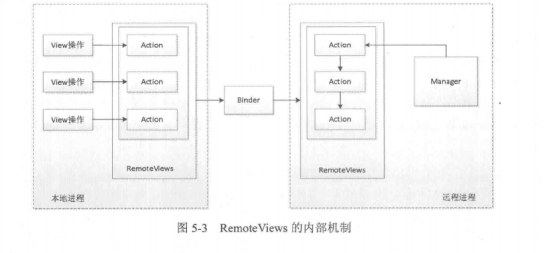
##### **RemoteView的内部机制**

RemoteViews主要用于通知栏和桌面小部件中。

工作流程：



内部机制：



##### **RemoteViews的意义**

可以实现跨进程的UI更新

》和AIDL方式的对比选型

（1）不用频繁更新UI、接口简单，可使用AIDL来跨进程更新UI。

（2）需要频繁更新UI，且布局文件中的View都是RemoteView所支持的，则考虑使用RemoteViews来实现，可提高效率。但是如果有定义RemoteViews不支持的View类型，就不适合使用RemoteViews了。

注意：

RemoteViews是本地进程中定义实现的！远程进程只是接收并使用而已！！

如果本地进程A和远程进程B分别在两个不同应用中，则从本地进程A传到远程进程B的布局文件的资源id不太可能一样，所以需要约定好布局文件名称，在远程进程B中使用如下代码来获取并加载本地进程A中的布局文件。

Int layoutId = getResources().getIdentifier(“布局名称”, “layout”, getPackageName());

View layoutView = getLayoutInflater.infalte(layoutId, mRemoteViewsContent, false);

### Android中的Drawable

##### Drawable是图像的抽象概念，包括颜色、图片都可以是Drawable。

##### **》学习意义**

##### 可以做一些特殊的UI效果，这个UI开发中尤其重要。

（1）使用非图片类型的Drawble，可以帮助减小APK的大小。

（2）学习成本低，比使用自定义View实现图形更简单。

##### **Drawable简介**

可使用xml来描述Drawable

<bitmap

android:src ---资源id

android:antialias ---抗锯齿（启用）

android:dither ---抖动（启用）

android:filter ---过滤（启用）

android:gravity ---对齐方式（决定图像在容器中的位置定位、缩放方式）

android:mipMap ---纹理映射（不常用）

android:titleMode ---平铺模式

/>

关注其中属性对于图像和容器之间关系的影响。

##### **Drawable的分类**

1. BitmapDrawable

它表示的就是一张图片。

<bitmap />如上1中

1. ShapeDrawable

可以理解为通过颜色来构造的图形。

<shape>

<corners/>

<gradient/>

<padding/>

<size/>

<solid/>

<stroke/>

</shape>

1. LayerDrawable

它表示一种层次化的Drawable集合。可以实现复杂的叠加效果。

<layer-list>

<item/>

<item/>

......

</layer-list>

每个item表示一个Drawable，下面的item覆盖上面的item。

1. StateListDrawble

对应的就是我们最熟悉的<selector>，也是一个Drawble集合。每个Drawble对应着View的一种状态。

<selector>

<item/> //如按下时

<item/> //如获得焦点时

......

<item/> 默认的item都应该放在最后一条

</selector>

1. LevelListDrawble

<level-list>，集合中每个Drawable都有一个等级（Level）的概念。

根据不同的等级，LevelListDrawble会切换不同的Drawable。

1. TransitionDrawable

<transition> 用于实现两个Drawable之间的淡入淡出效果。

1. InsetDrawable

<inset> 可以将其他Drawable内嵌到自己当中。

当View需要自己的背景比自己的实际区域小的时候，可以用InsetDrawable。

1. ScaleDrawable

<scale> 可以根据自己的等级将指定的Drawable缩放到一定的比例。

1. ClipDrawable

<clip> 根据自己的等级来裁剪另一个Drawable，裁剪方向有android:clipOrientation和android:gravity两个属性共同控制。

##### **自定义Drawable**

（1）自定义Drawable的使用范围很单一，一个是作为ImageView的图像来显示，一个是作为View的背景。

（2）在xml中不允许使用自定义View。

### Android动画深入分析

##### **View动画**

1. View动画的种类

TranslateAnimation

ScaleAnimation

RotateAnimation

AlphaAniamation

1. 自定义View动画

不常用

1. 帧动画

图片多图片大时，容易造成OOM问题。

##### **2、View动画的特殊使用场景**

（1）LayoutAnimation

（2）Activity的切换效果

startActivity()和finish()之后，再调用overridePendingTransiTion()才能生效！！！

##### **3、属性动画**

ObjectAnimator（可以是View，也可以是其他任何有set方法的类对象）

ValueAnimation 值动画（作用于值，和具体的View无关）

AnimatorSet 属性动画集合

插值器（时间插值器TimeInterpolator）时间流逝的百分比float——》属性值变化的百分比float

类型估值算法（TypeEvaluator）属性值变化的百分比float——》最终的属性值

##### **使用动画的注意事项**

1. OOM

尽量避免使用帧动画，特别是图片较多且较大的时候。

1. 内存泄漏

属性动画时，无限循环的动画切记要在退出时停止，否则导致Activity无法释放导致内存泄漏。View动画则无此问题。

（3）兼容性问题

做好3.0以下版本的适配工作。

1. View动画的问题

view#clearAnimation

1. 不要使用px

使用dp

1. 动画元素的交互（View的事件响应，如单击事件）

3.0以后，使用属性动画View事件触发的位置在移动后的位置，而使用View动画则还是原位置触发。

1. 硬件加速

建议开启硬件加速，可提升动画的流畅度。

### 理解Window和WindowManager

##### **Window和WindowManager**

Window有三种类型：应用的Window、子Window、系统的Window。

Window是分层级的：每个Window都有对应的z-ordered，层级大的Window覆盖在层级小的Window上面。

##### **Window的内部机制**

**IPC实现机制**

（1）Window的添加过程

（2）Window的删除过程

（3）Window的更新过程

##### **Window的创建过程**

1. Activity的Window创建过程
2. Dialog的Window创建过程
3. Toast的Window创建过程

### 四大组件的工作过程

ActivityManagerNative#getDefault

ActivityManagerService（AMS）

ActivtyThread#attach

ActivityClientRecord——》Activity.class.newInstance

Activity#onCreate

Activity#onResume

##### **四大组件的运行状态**

##### **Activity的工作过程**

ActivtyThread#attach

ActivityClientRecord——》Activity.class.newInstance

Activity#onCreate

Activity#onResume

##### **Service的工作过程**

ActivtyThread#attach

ServiceClientRecord——》Service.class.newInstance

Service#onCreate

##### **BroadcastReceiver的工作过程**

##### **ContentProvider的工作过程**

ContentProvider#onCreate

Application#onCreate

### Android的消息机制

ThreadLocal的工作原理

消息队列MessageQueue的工作原理

Looper的工作原理

Handler的工作原理

主线程的消息循环

### Android的线程和线程池

##### **主线程和子线程**

##### **Android中的线程形态**

1. AyncTask
2. HandlerThread
3. IntentService

##### **Android中的线程池**

（1）ThreadPoolExecutor

（2）线程池的分类

### Bitmap的加载和cache

##### **Bitmap的高效加载**

依据ImageView按需加载相应尺寸大小的图片。

##### **Android中的缓存策略**

LruCahe（内存缓存）

DiskLruCache（磁盘缓存）

ImageLoader的实现

##### **ImageLoader的使用**

照片墙

列表的卡顿优化（**开启硬件加速、ViewHolder、getView不执行耗时操作、快速滑动时控制任务的执行**）

### 综合技术

##### **CrashHandler获取应用的Crash信息**

存本地sdcard（运维可以获取到运异常信息）

上传服务器（后台随时可查看，利于开发人员升级版本时分析解决发布）

##### **Mutidex解决方法数越界（65536）**

三种方案给应用加上mutidex的功能：

Manifest中指定com.android.support.MutiDexApplication

Application继承MutiDexApplication

实现attachBaseContext()方法，执行MutiDex.install(this)。

可以配置主dex需要打包的类，如在maindexlist.txt文件中配置。

但是，mutidex也有些许问题，如应用启动速率会降低，dex很大时甚至会出现ANR。（所以避免生成较大的dex文件）

**综合来说mutidex还是一个非常好的解决方法数越界的方案。（OK！）**

##### **Android的动态加载技术**

插件化技术

##### **反编译初步**

apktool: <http://ibotpeaches.github.io/Apktool/>

dex2jar: <https://github.com/pxb1988/dex2jar>

jd-gui: <http://jd.benow.ca/>

Android逆向工程-破解 哈皮妹-萝莉

<https://blog.csdn.net/singwhatiwanna/article/details/18797493>