# 设置控价抖动的动画效果：

Animation translateAnimation = new TranslateAnimation(0, 10, 0, 0);  
//设置一个循环加速器，使用传入的次数就会出现摆动的效果。  
translateAnimation.setInterpolator(new CycleInterpolator(2));  
translateAnimation.setDuration(300);  
etPhone.startAnimation(translateAnimation);

# 自定义控价属性：

1. 在value文件夹下新建atts.xml文件，用于存放自定义的属性定义，如下：
2. <declare-styleable name="MyView" >  
    <attr name="boll\_color" format="color"></attr>  
    <attr name="boll\_size" format="dimension"></attr>  
   </declare-styleable>

其中**<declare-styleable** name=" MyView "**>中的** name为自定义名字，通常与自定义的view名字一致，下面attr只是定义了一些属性，至于这些属性被哪个控件引用，在此处是不知道的。属性的format可以有如下几种：

* reference 参考某一资源ＩＤ
* string
* color
* dimension 尺寸值
* boolean
* integer
* float
* fraction 百分数
* enum
* flag 位或运算

1. 在布局文件中的根布局添加如下声明：

xmlns:zht="http://schemas.android.com/apk/res-auto"

或者：**xmlns:zht ="http://schemas.android.com/apk/res/(包名)**"

其中myname 是自定义属性的空间名。

使用控价：

<com.gaia.member.gaiatt.ui.MyView  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 zht:boll\_color="@android:color/darker\_gray"  
 zht:boll\_size="100px"  
 />

1. 在自定义的控价代码中抽取自定义的属性：MyView extends View

public MyView(Context context, AttributeSet attrs) {  
 super(context, attrs);  
 this.context = context;  
  
 TypedArray typedArray = context.obtainStyledAttributes(attrs, R.styleable.*MyView*);  
 bollColor = typedArray.getColor(R.styleable.*MyView\_boll\_color*, Color.*BLUE*);  
 bollSize = typedArray.getDimension(R.styleable.*MyView\_boll\_size*, 10);  
 typedArray.recycle();

init();  
}

清注意TypedArray对象是一个共享资源，必须被在使用后进行回收。recycle().

# ViewPager禁止滑动：

public class CustomViewPager extends ViewPager {

private boolean isPagingEnabled = true;

public CustomViewPager(Context context) {

super(context);

}

public CustomViewPager(Context context, AttributeSet attrs) {

super(context, attrs);

}

@Override

public boolean onTouchEvent(MotionEvent event) {

return this.isPagingEnabled && super.onTouchEvent(event);

}

@Override

public boolean onInterceptTouchEvent(MotionEvent event) {

return this.isPagingEnabled && super.onInterceptTouchEvent(event);

}

public void setPagingEnabled(boolean b) {

this.isPagingEnabled = b;

}

}

# dp、dip、ppi、px总结

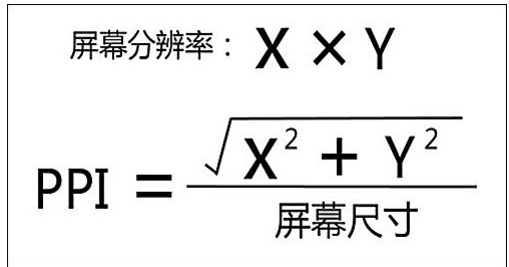
dp也称dip, device independent pixels(设备独立像素)，定义为160dpi的屏幕中1px=1dp, px即pixel,即一个像素点，是衡量屏幕分辨率的最小物理单位。dp用来适配和调整不同分辨率下的图像的显示效果，当分辨率高时，如320dpi, 1dp=2px； 480dpi,1dp=3px。dp也可认为是1px在160dpi的屏幕上的物理长度，即不同分辨率的屏幕上，1dp所显示的物理长度是一样的，尽管高分辨率上1px的物理长度小与低分辨率屏幕上1px的物理长度，但可以让1dp代表更多的px像素点来达到同样的物理长度和显示效果。

1dp = (dpi/160)\*px；其中 density = dpi/160。

ppi：pixel per inch，每inch长度的像素数，如320\*480的屏幕，320代表屏幕宽度方向有320个px,480代表高度方向的px个数或者单位。如果屏幕的宽 w, 长 l， 则ppi = 320/w=480/l；

因为一般屏幕厂家的宽高是不定的，但对角线的长度是固定，所以用对角线的长度来计算ppi，通常所说的屏幕的大小也是指对角线的长度，如5寸的屏幕对角线长度为5\*2.54cm。

用对角线来计算ppi的公式为：



dpi：dot per inch，每inch长度上的点数。dpi=Diagonal pixel/ Screen size，是屏幕像素密度。就是1英寸上像素点的个数。多用于屏幕显示领域。类似的有ppi（Pixels per inch）多用于印刷领域。分辨率越高，则每英寸内包含的像素数越大。通常系统会设定dpi的值，和真实的ppi有一定的差距，系统设定的dpi的值一般为120,160,240,320,480等。

# bitmap和bitmapdrawable

Bitmap，代表一个位图图像，[Android](http://lib.csdn.net/base/15)支持三种格式的位图图像：.png (preferred)，.jpg (acceptable)， .gif (discouraged)。

括号里的说明，代表这三种格式的图片在Android中的支持情况，.png格式图片优先，.jpg格式也可以，但是效果没有.png好，.gif支持最差。

在构建应用的时候，Bitmap文件可能会被appt工具压缩自动优化为无损图像。例如,一个真彩色PNG，不需要超过256的颜色可以被转换成一个8位PNG和调色板。这将导致一个图像质量相同，但这需要更少的内存。所以要意识到，在drawable目录中图像的二进制文件在构建程序时可以改变。如果你打算读一个图像作为字节流并将它转换成一个位图，把你的图片放在在res /raw/文件夹里，在那里他们不会被优化。

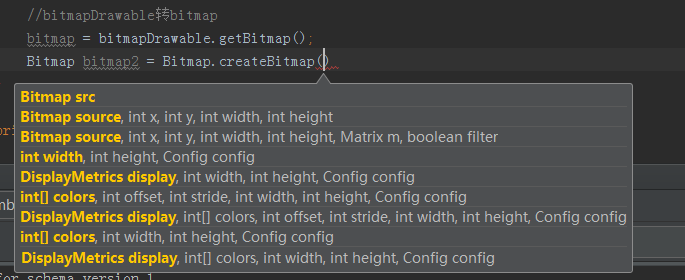
Bitmap - 称作位图，一般位图的文件格式后缀为bmp，当然编码器也有很多如RGB565、RGB888。作为一种逐像素的显示对象执行效率高，但是缺点也很明显存储效率低。我们理解为一种存储对象比较好。

Bitmap:图片文件的封装,可以看做是一张位图  
此类中的静态方法可以通过源Bitmap创建新的Bitmap对象  
此类封装了位图的一些信息

BitmapFactory:一个工具类，用于创建Bitmap对象  
使用此类可以通过文件，资源，byte-array等创建Bitmap对象

Bitmap bitmap = BitmapFactory.decodeXXX

同时bitmap还可以从现有的bitmap中进行创建：



BitmapDrawable:是Drawable类的扩展，Drawable是一个可以绘制到屏幕上的一个对象，Drawable就是一个可画的对象，其可能是一张位图（BitmapDrawable），也可能是一个图形（ShapeDrawable），还有可能是一个图层（LayerDrawable），我们根据画图的需求，创建相应的可画对象。

BitmapDrawable则为可以会知道屏幕上的一个位图对象  
BitmapDrawable对象中封装了Bitmap对象

ImageView:图片控件，用于承载Bitmap绘制在屏幕上  
图片控件,需要显示图片时需要把Bitmap,set给它

一个BitmapDrawable就是封装了一个位图。直接以文件的方式，就是封装了一个原始的位图。以Xml方式，可以对原始的位图进行一系列的处理，比如说抗锯齿，拉伸，对齐等等。

要了解BitmapDrawable的使用，还需要明白Bitmap、BitmapFactory等类。Bitmap代表了一个原始的位图，并且可以对位图进行一系列的变换操作。BitmapFactory提供一系列的方法用于产生一个Bitmap对象。

//从资源文件获取BitmapDrawable  
BitmapDrawable drawable = (BitmapDrawable) getResources().getDrawable(R.drawable.*female*);

//bitmap转bitmapDrawable  
Bitmap bitmap = BitmapFactory.*decodeResource*(getResources(),R.drawable.*female*);  
BitmapDrawable bitmapDrawable = new BitmapDrawable(getResources(), bitmap);  
  
//bitmapDrawable转bitmap  
bitmap = bitmapDrawable.getBitmap();

# Canvas总结：

**Class Overview**

The Canvas class holds the "draw" calls. To draw something, you need 4 basic components: A Bitmap to hold the pixels, a Canvas to host the draw calls (writing into the bitmap), a drawing primitive (e.g. Rect, Path, text, Bitmap), and a paint (to describe the colors and styles for the drawing).

这个类相当于一个画布，你可以在里面画很多东西；

我们可以把这个Canvas理解成系统提供给我们的一块内存区域(但实际上它只是一套画图的API，真正的内存是下面的Bitmap)，而且它还提供了一整套对这个内存区域进行操作的方法，所有的这些操作都是画图API。也就是说在这种方式下我们已经能一笔一划或者使用Graphic来画我们所需要的东西了，要画什么要显示什么都由我们自己控制。

这种方式根据环境还分为两种：一种就是使用普通View的canvas画图，还有一种就是使用专门的SurfaceView的canvas来画图。两种的主要是区别就是可以在SurfaceView中定义一个专门的线程来完成画图工作，应用程序不需要等待View的刷图，提高性能。前面一种适合处理量比较小，帧率比较小的动画，比如说象棋游戏之类的；而后一种主要用在游戏，高品质动画方面的画图。

1  Bitmap，可以来自资源/文件，也可以在程序中创建，实际上的功能相当于图片的存储空间；

2  Canvas，紧密与Bitmap联系，把Bitmap比喻内容的话，那么Canvas就是提供了众多方法操作Bitamp的平台；

3  Paint，与Canvas紧密联系，是"画板"上的笔刷工具，也用于设置View控件上的样式；

4  Drawable，如果说前三者是看不见地在内存中画图(虚拟的)，那么Drawable就是把前三者绘图结果表现出来的接口(真实的)。 Drawable多个子类，例如：位图(BitmapDrawable)、图形(ShapeDrawable)、图层(LayerDrawable)等。

Canvas就是一套画图的API接口方法，使用它必须要有一个bitmap，bitmap才是真正绘制图像的画布。还需要一个paint。

paint类：

**Class Overview**

The Paint class holds the style and color information about how to draw geometries, text and bitmaps.

paint类拥有风格和颜色信息如何绘制几何学,文本和位图。

Canvas可以自己新建，也可以在View类的onDraw方法中获取，自定义的canvas需要给它设定一个bitmap。

Bitmap bitmap = Bitmap.*createBitmap*(500,500, Bitmap.Config.*ARGB\_8888*);  
Canvas cacheCanvas = new Canvas();  
cacheCanvas.setBitmap(bitmap);  
//或者  
Canvas canvas1 = new Canvas(bitmap);

如果是从View的onDraw方法中获取的

@Override  
protected void onDraw(Canvas canvas) {...... }

则canvas会持有View中的默认的bitmap，此时不能对canvas设定其他的bitmap，否则程序会crash。

不过canvas可以将其他的bitmap显示出来，使用函数：

canvas.drawBitmap(bitmap,0,0, new Paint());

这种方法可用来双缓冲绘图，即先使用一个cacheCanvas在一个bitmap上绘图，画完以后再使用持有View本身bitmap的canvas将之前的bitmap绘制出来即可。

canvas还提供了很多其他的绘图方法，画点、线、矩形等等。

几个Canvas中常用的方法：

* void drawBitmap(Bitmap bitmap,float left,float top,Paint paint)：在指定坐标绘制位图。
* void drawLine(float startX,float startY,float stopX,float stopY,Paint paint)：根据给定的起始点和结束点之间绘制连线。
* void drawPath(Path path,Paint paint)：根据给定的path，绘制连线。
* void drawPoint(float x,float y,Paint paint)：根据给定的坐标，绘制点。
* void drawText(String text,int start,int end,Paint paint)：根据给定的坐标，绘制文字。
* int getHeight()：得到Canvas的高度。
* int getWidth()：得到Canvas的宽度。

SurfaceView是View类的子类，可以直接从内存或者DMA等硬件接口取得图像数据，是个非常重要的绘图视图。它的特性是：可以在主线程之外的线程中向屏幕绘图上。这样可以避免画图任务繁重的时候造成主线程阻塞，从而提高了程序的反应速度。在游戏开发中多用到SurfaceView，游戏中的背景、人物、动画等等尽量在画布canvas中画出。http://www.cnblogs.com/devinzhang/archive/2012/02/03/2337559.html

# 2种自定义控价的方法

第一种：

1. 先把自定义控价的布局文件custom\_layout写好。
2. 新建一个类，继承LinearLayout或者RelativeLayout，重写它的构造函数，主要是2个参数的那个(Context context, AttributeSet attrs)，系统调用的就是它。其中context就是加载该布局文件的activity（暂时不确定）的context，可以使用该context做一些相应的操作，比如activity的退出，在定义界面的头布局的左上角的返回按键时用得上。
3. 在构造函数里inflate布局文件custom\_layout，获取其中的子控价，并进行相应的设定。
4. 在其他的布局中引入自定义控价，和系统控价的引入方式一样，只是自定义控价要把路径写完整：

如<com.example.zht.testandroid.TitleView  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content">

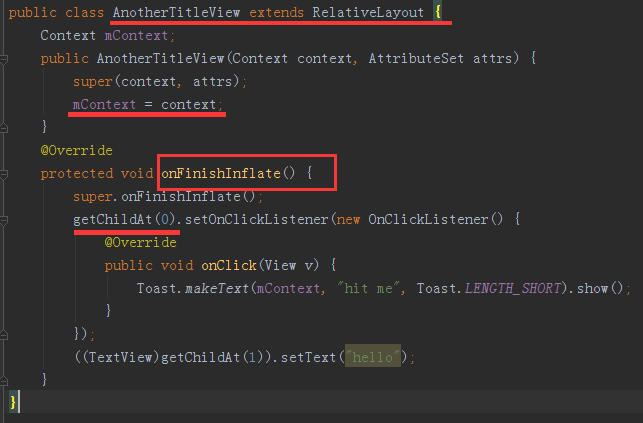
第二种方法：

这种方法是这样引入自定义好的控价：

<include layout="@layout/title\_layout" />

其中在title\_layout中将其中的一个Layout换成自定义的类。然后在类中对这个Layout下的子控价进行相应的操作。





自定义的类继承本应在布局文件中使用的Layout类型，然后重写构造方法，主要是取得context。对其子控件的操作一定要在onFinishInflate函数中操作，否则会报空指针异常。可以使用getChildAt(index)获取子控价或者：

((TextView)findViewById(R.id.*tv\_title*)).setText("MM");

# 自定义View总结，以及SurfaceView

一般自定义View的时候，需要重新onDraw方法，在里面使用canvas绘制新的图像，用时也需要重写onTouchEvent方法，做一些触摸事件的判断和相应。

重新绘图后，调用[invalidate()](http://developer.android.com/reference/android/view/View.html#invalidate()) and [requestLayout()](http://developer.android.com/reference/android/view/View.html#requestLayout()). 这两个调用是确保稳定运行的关键。当view的某些内容发生变化的时候，需要调用invalidate来通知系统对这个view进行redraw，当某些元素变化会引起组件大小变化时，需要调用requestLayout方法。调用时若忘了这两个方法，将会导致hard-to-find bugs。

<http://threezj.com/2015/12/17/Android%20View%E8%AF%A6%E8%A7%A3/>

<http://hukai.me/android-training-course-in-chinese/ui/custom-view/create-view.html>

自定义移动的小球：

@Override  
protected void onDraw(Canvas canvas) {  
 canvas.drawColor(Color.*BLACK*);  
  
 widthCanvas = canvas.getWidth();  
 heightCanvas = canvas.getHeight();  
 canvas.drawCircle(curX,curY,bollRadius,paintBoll);  
}  
  
public void start() {  
 time = new Timer();  
 time.schedule(new TimerTask() {  
 @Override  
 public void run() {  
  
 curX += xspeed;  
 curY += yspeed;  
  
 if (curX <= bollRadius || curX >= widthCanvas - bollRadius) {  
 xspeed = -xspeed;  
 }  
  
 if (curY <= bollRadius || curY >= heightCanvas - bollRadius) {  
 yspeed = -yspeed;  
 }  
 postInvalidate();  
 }  
 }, 0, 20);  
}  
  
public void stop() {  
 time.cancel();  
}

自定义画板： 采用双缓存机制

@Override  
 protected void onDraw(Canvas canvas) {  
 canvas.drawBitmap(cacheBitmap, 0, 0, null);  
 canvas.drawPath(path, paint);  
 }  
  
 @Override  
 public boolean onTouchEvent(MotionEvent event) {  
  
 curX = event.getX();  
 curY = event.getY();  
  
 switch (event.getAction()) {  
 case MotionEvent.*ACTION\_DOWN*:  
 path.moveTo(curX, curY);  
 preX = curX;  
 preY = curY;  
 break;  
 case MotionEvent.*ACTION\_MOVE*:  
// path.quadTo(preX, preY, curX, curY);  
 path.lineTo(curX, curY);  
 preX = curX;  
 preY = curY;  
 break;  
 case MotionEvent.*ACTION\_UP*:  
 cacheCanvas.drawPath(path, paint);  
 path.reset();  
 break;  
 }  
 invalidate();  
 return true;  
 }

SurfaceView是View类的子类，可以直接从内存或者DMA等硬件接口取得图像数据，是个非常重要的绘图视图。它的特性是：可以在主线程之外的线程中向屏幕绘图上。这样可以避免画图任务繁重的时候造成主线程阻塞，从而提高了程序的反应速度。在游戏开发中多用到SurfaceView，游戏中的背景、人物、动画等等尽量在画布canvas中画出。

2.实现方法

1）实现步骤

    a．继承SurfaceView

    b．实现SurfaceHolder.Callback接口

2）需要重写的方法

（1）public void surfaceChanged(SurfaceHolder holder,int format,int width,int height){}　　//在surface的大小发生改变时激发  
  
（2）public void surfaceCreated(SurfaceHolder holder){}　　//在创建时激发，一般在这里调用画图的线程。  
  
（3）public void surfaceDestroyed(SurfaceHolder holder) {}　　//销毁时激发，一般在这里将画图的线程停止、释放。

3）SurfaceHolder

　　SurfaceHolder,surface的控制器，用来操纵surface。处理它的Canvas上画的效果和动画，控制表面，大小，像素等。  
几个需要注意的方法：

[复制代码](javascript:void(0);)

(1)、abstract void addCallback(SurfaceHolder.Callback callback);  
// 给SurfaceView当前的持有者一个回调对象。  
(2)、abstract Canvas lockCanvas();  
// 锁定画布，一般在锁定后就可以通过其返回的画布对象Canvas，在其上面画图等操作了。  
(3)、abstract Canvas lockCanvas(Rect dirty);  
// 锁定画布的某个区域进行画图等..因为画完图后，会调用下面的unlockCanvasAndPost来改变显示内容。  
// 相对部分内存要求比较高的游戏来说，可以不用重画dirty外的其它区域的像素，可以提高速度。  
(4)、abstract void unlockCanvasAndPost(Canvas canvas);  
// 结束锁定画图，并提交改变。

[复制代码](javascript:void(0);)

4）总结整个过程

　　继承SurfaceView并实现SurfaceHolder.Callback接口 ----> SurfaceView.getHolder()获得SurfaceHolder对象 ---->SurfaceHolder.addCallback(callback)添加回调函数---->SurfaceHolder.lockCanvas()获得Canvas对象并锁定画布----> Canvas绘画 ---->SurfaceHolder.unlockCanvasAndPost(Canvas canvas)结束锁定画图，并提交改变，将图形显示。

@Override  
protected void onCreate(Bundle savedInstanceState) {  
 super.onCreate(savedInstanceState);  
 setContentView(new MySurfaceView(this));  
  
}  
  
class MySurfaceView extends SurfaceView {  
  
 private SurfaceHolder holder;  
 private MyThread myThread;  
 public MySurfaceView(Context context) {  
 super(context);  
 holder = getHolder();  
 myThread = new MyThread(holder);  
 holder.addCallback(new SurfaceHolder.Callback() {  
 @Override  
 public void surfaceCreated(SurfaceHolder holder) {  
 myThread.runFlag = true;  
 myThread.start();  
 }  
 @Override  
 public void surfaceChanged(SurfaceHolder holder, int format, int width, int height) {  
 }  
 @Override  
 public void surfaceDestroyed(SurfaceHolder holder) {  
 myThread.runFlag = false;  
 }  
 });  
 }  
}  
  
class MyThread extends Thread {  
 SurfaceHolder holder;  
 boolean runFlag = false;  
 MyThread (SurfaceHolder holder) {  
 this.holder = holder;  
 }  
  
 @Override  
 public void run() {  
 Canvas canvas = null;  
 Paint paint = new Paint();  
 paint.setColor(Color.*WHITE*);  
 paint.setTextSize(30);  
 paint.setAntiAlias(true);  
 int times = 0;  
 while (runFlag) {  
 synchronized (holder) {  
 canvas = holder.lockCanvas();  
 canvas.drawColor(Color.*BLUE*);  
 canvas.drawRect(100, 100, 300, 300, new Paint());  
 canvas.drawText("第 "+times+" 秒", 150,150,paint);  
 }  
 times++;  
 SystemClock.*sleep*(300);  
 holder.unlockCanvasAndPost(canvas);  
 }  
 }  
}

# 自定义ViewPager

1. 定义类
2. */\*\*  
    \* Created by zhangHaiTao on 2016/5/1.  
    \*/*public class MyViewPager extends ViewGroup{  
    private static final String *TAG* = "ZHT";  
    Context mContext;  
    float xDown,xMove,xLastMove;  
    int leftBorder,rightBorder;  
    int mTouchSlop;//判断为拖动的最小像素  
    Scroller mScroller;  
    public MyViewPager(Context context, AttributeSet attrs) {  
    super(context, attrs);  
    mContext = context;  
    mScroller = new Scroller(context);  
    ViewConfiguration configuration = ViewConfiguration.*get*(context);  
    mTouchSlop = configuration.getScaledPagingTouchSlop();  
    }  
     
    @Override  
    protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {  
    super.onMeasure(widthMeasureSpec, heightMeasureSpec);  
    for (int i=0; i<getChildCount(); i++) {  
    View view = getChildAt(i);  
    measureChild(view, widthMeasureSpec, heightMeasureSpec);  
    }  
    }  
     
    @Override  
    protected void onLayout(boolean changed, int l, int t, int r, int b) {  
    if (changed) {  
    for (int i=0; i<getChildCount(); i++) {  
    View child = getChildAt(i);  
    child.layout(i\*getMeasuredWidth(), 0, (i+1)\*getMeasuredWidth(), getMeasuredHeight());  
    }  
     
    leftBorder = getChildAt(0).getLeft();  
    rightBorder = getChildAt(getChildCount()-1).getRight();  
    }  
    }  
     
    @Override  
    public boolean onInterceptTouchEvent(MotionEvent ev) {  
    switch (ev.getAction()) {  
    case MotionEvent.*ACTION\_DOWN* :  
    xDown = ev.getRawX();  
    xLastMove = xDown;  
    break;  
    case MotionEvent.*ACTION\_MOVE* :  
    xMove = ev.getRawX();  
    float dif = xMove - xDown;  
    xLastMove = xMove;  
    //判断是否滑动 是则拦截事件，进入onTouchEvent  
    //也可以直接在此函数中使用 return true,拦截所有事件  
    if (Math.*abs*(dif) > mTouchSlop) {  
    return true;  
    }  
    break;  
    }  
    return super.onInterceptTouchEvent(ev);  
    }  
     
    @Override  
    public boolean onTouchEvent(MotionEvent event) {  
    switch (event.getAction()) {  
    case MotionEvent.*ACTION\_DOWN* :  
    xDown = event.getRawX();  
    xLastMove = xDown;  
    break;  
    case MotionEvent.*ACTION\_MOVE* :  
    xMove = event.getRawX();  
    float dx = xLastMove - xMove;  
    Log.*w*(*TAG*, "xMove"+xMove+" l:"+leftBorder);  
    Log.*w*(*TAG*, "getScrollX"+getScrollX()+" r:"+rightBorder);  
    if (getScrollX()+dx < leftBorder) {  
    scrollTo(leftBorder,0);  
    } else if (getScrollX()+getMeasuredWidth()+dx > rightBorder) {  
    scrollTo(rightBorder-getMeasuredWidth(),0);  
    } else {  
    scrollBy((int)dx, 0);  
    xLastMove = xMove;  
    }  
    break;  
    case MotionEvent.*ACTION\_UP* :  
    int index = getScrollX() / getMeasuredWidth();  
    index += (getScrollX() % getMeasuredWidth() > (getMeasuredWidth()/2) ? 1 : 0);  
   // scrollTo(index\*getMeasuredWidth(),0);//瞬时移动  
    int scrollerDx = index\*getMeasuredWidth() - getScrollX();  
    mScroller.startScroll(getScrollX(), 0, scrollerDx, 0);  
    invalidate();  
    break;  
    }  
    return true;  
    }  
     
     
    @Override  
    public void computeScroll() {  
    if (mScroller.computeScrollOffset()) {  
    scrollTo(mScroller.getCurrX(), mScroller.getCurrY());  
    invalidate();  
    }  
    }  
   }
3. 定义布局文件
4. <com.zhhtao.testad.MyViewPager xmlns:android="http://schemas.android.com/apk/res/android"  
    android:orientation="vertical" android:layout\_width="match\_parent"  
    android:layout\_height="match\_parent">  
     
    <Button  
    android:layout\_width="match\_parent"  
    android:layout\_height="match\_parent"  
    android:text="first"/>  
    <Button  
    android:layout\_width="match\_parent"  
    android:layout\_height="match\_parent"  
    android:text="second"/>  
    <Button  
    android:layout\_width="match\_parent"  
    android:layout\_height="match\_parent"  
    android:text="thirds"/>  
   </com.zhhtao.testad.MyViewPager>

# onMeasure的用法

一般自定义控价时候，可以通过重写onMeasure方法来重写设定控价的大小。先看源码的方法：

protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {  
 setMeasuredDimension(*getDefaultSize*(getSuggestedMinimumWidth(), widthMeasureSpec),  
 *getDefaultSize*(getSuggestedMinimumHeight(), heightMeasureSpec));  
}

即必须调用setMeasuredDimension来设定测量好的宽高值，否则会报错。

当然可以通过setMeasuredDimension(200, 200);

来设置具体的固定值，单位为px.

如果自定义的CustomView采用默认的onMeasure函数，行为如下：

（1） CustomView设置为 match\_parent 或者 wrap\_content 没有任何区别，其显示大小由父控件决定，它会填充满整个父控件的空间。

（2） CustomView设置为固定的值，则其显示大小为该设定的值。

onMeasure(int widthMeasureSpec, int heightMeasureSpec)中，两个参数的作用：widthMeasureSpec和heightMeasureSpec这两个int类型的参数，看名字应该知道是跟宽和高有关系，但它们其实不是宽和高，而是由宽、高和各自方向上对应的模式来合成的一个值：其中，在int类型的32位二进制位中，31-30这两位表示模式，0~29这三十位表示宽和高的实际值.其中模式一共有三种，被定义在Android中的View类的一个内部类中：View.MeasureSpec：

①UNSPECIFIED：表示默认值，父控件没有给子view任何限制。------二进制表示：00

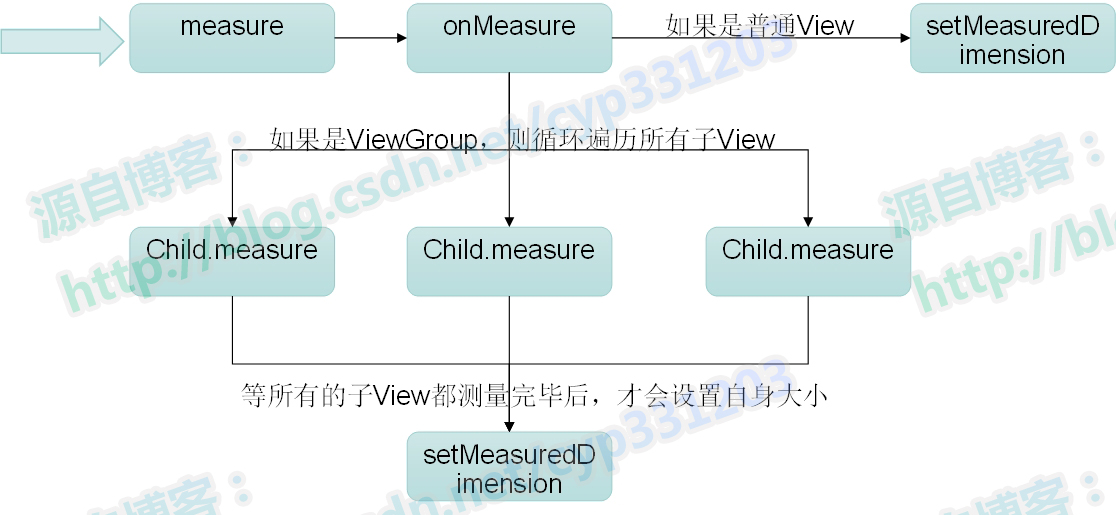
②EXACTLY：表示父控件给子view一个具体的值，子view要设置成这些值的大小。------二进制表示：01

③AT\_MOST：表示父控件个子view一个最大的特定值，而子view不能超过这个值的大小。------二进制表示：10

这2个参数是从xml的layout\_width和layout\_height的属性中提取出来的，并且如果layout的值为具体的值(如layout\_width=200dp)或者matchparent/fillparent，对应模式中的MeasureSpec.EXACTLY。如果为wrap\_content则模式为包裹内容(width=wrapcontent)则对应模式中的MeasureSpec.AT\_MOST。

一个view的宽高尺寸，只有在测量之后才能得到，也就是measure方法被调用之后。大家都应该使用过View.getWidth()和View.getHeight()方法，这两个方法可以返回view的宽和高，但是它们也不是在一开始就可以得到的，比如oncreate方法中，因为这时候measure方法还没有被执行，测量还没有完成，我们可以来作一个简单的实验：自定义一个MyView，继承View类，然后在OnCreate方法中，将其new出来，通过addview方法，添加到现在的布局中。然后调用MyView对象的getWidth()和getHeight()方法，会发现得到的都是0。

控件又分为view和viewGroup两种情况，那么它们测量的流程是怎样的呢，看一下下面这个图你就明白了：



# 焦点总结

当用户在使用方向键或轨迹球浏览用户界面时，有必要给于一个焦点在可操作的组件上(如一个Button)，使用户可以看到它将接受输入命令。如果设备有触摸功能，那么，当用户与界面的交互就不再需要有一个高亮在组件上，或一个焦点在view上，因此，模式的互动名为"触摸模式"。对于一个触摸设备，一旦有用户接触屏幕时，该设备将进入触摸模式.在点触某个View后，只有的它的方法isFocusableInTouchMode()返回为真时，才会有聚集焦点，如文本编辑工具。其他的界面只可以点触，但不会聚集焦点（高亮），如button 被点触时就不会聚集焦点，当它被按下时只会调用on-click监听器的回调方法。  
任何时候用户接触方向键或者滚动轨迹球时，该设备将退出触摸模式，并聚集焦点，用户可以恢复与用户界面的键盘交互，而不必在屏幕上。触摸模式的状态是由整个系统来维持的(all windows and activities),要查询目前所处的状态,你可以调用isInTouchMode()方法来获得，看看设备目前是否处于触摸模式。

系统框架将处理日常的焦点移动来响应用户的输入，它包刮改变焦点（当界面是被移除，隐藏，或者作为一个新的View变为可用状态），通过isFocusable()这个方法我们可以知道view是否具有接受焦点的资格，也可以通过setFocusable().来设置view接受焦点的资格,对应在触摸模式下，你可以调用isFocusableInTouchMode().来获知是否有焦点来响应点触，也可以通过setFocusableInTouchMode().来设置是否有焦点来响应点触的资格.  
系统框架控制焦点移动到另一个组件的算法是在某一方向上邻近的组件，在极个别情况下，默认的算法可能不符合开发者的预想要求，在这种情况下，你可以覆写下列XML属性的布局文件： nextFocusDown ， nextFocusLeft ， nextFocusRight ，和nextFocusUp 设置他们的值来明确。

通常如果你想宣布用户界面具有焦点的资格 (如果这个界面在传统上是没有的)，可以在xml布局里去加上的android:focusable的属性，并设置它的值，您也可以宣布在触摸模式下具有焦点的资格，同样也只在xml里添android:focusableInTouchMode.的属性，并设置它的值. 当用户请求在某个界面聚集焦点时，会调用requestFocus().这个方法。监听到焦点活动(获得焦点或失去焦点都会被通知)，会调用onFocusChange(),这个方法。

例子：使用requestFocus()方法使焦点在各个控件之间切换



弹出框是个PopupWindow，需要依次输入4个密码，为了方便快捷，当上一个文本框输入值之后，焦点自动切换到下一个文本框，当输入到最后一个文本框后，PopupWindow自动关闭，并返回4个文本框中的值，放在String[]数组中。

不让文本输入框默认获得焦点，弹出键盘占据我们的屏幕的方法：有时会遇到一种情况，在真机运行一个页面，焦点会默认在EditText上，然后自动弹出键盘来占据大半个屏幕。处理方法很简单。在配置文件中加一句:

<activity ...  [Android](http://lib.csdn.net/base/15):windowSoftInputMode="stateAlwaysHidden|adjustResize"  />

当有个可滚动的空间嵌套时，比如ScrollView 中放置了 ListView或者GridView时，有时会出现ScrollView初始滚动到ListView处，是因为GridView在初始化时候会获取焦点，导致ScrollView重新绘制，自动滚动到当前获得焦点的View。解决问题的办法很简单，将GridView. setFocusable(false)即可。

focusAble属性不会影响View的点击事件。editText的focusAble和focusableInTouchMode

为false时，edittext将无法输入内容。

# 事件分发总结

事件的处理是在dispatchTouchEvent方法，只要你触摸到了任何一个控件，就一定会调用该控件的dispatchTouchEvent方法。

dispatchTouchEvent方法的原理代码如下：

**[java]** [view plain](http://blog.csdn.net/guolin_blog/article/details/9097463) [copy](http://blog.csdn.net/guolin_blog/article/details/9097463)

1. **public** **boolean** dispatchTouchEvent(MotionEvent event) {
2. **if** (mOnTouchListener != **null** && (mViewFlags & ENABLED\_MASK) == ENABLED &&
3. mOnTouchListener.onTouch(**this**, event)) {
4. **return** **true**;
5. }
6. **return** onTouchEvent(event);
7. }

可见控件注册的OnTouchListener的onTouch是在于控件自身的onTouchEvent方法之前执行的，并且如果onTouch方法返回了true，即代表onTouch消费了事件，则dispatchTouchEvent直接返回true，onTouchEvent得不到执行。如果控件没有注册OnTouchListener监听，或者其onTouch方法返回了false，则进入onTouchEvent方法，并且dispatchTouchEvent方法的返回值和onTouchEvent的返回值是一致的。

onTouchEvent的代码示意执行如下：

public boolean onTouchEvent(MotionEvent event) {  
 if ((viewFlags & ENABLED\_MASK) == DISABLED) {  
 // 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));  
 }  
  
 if (((viewFlags & CLICKABLE) == CLICKABLE ||  
 (viewFlags & LONG\_CLICKABLE) == LONG\_CLICKABLE)) {  
 switch (event.getAction()) {  
 case MotionEvent.*ACTION\_UP* :  
 //其他代码  
 performClick();  
 break;  
   
 case MotionEvent.*ACTION\_DOWN* :  
 //....  
 break;  
 case MotionEvent.*ACTION\_MOVE* :  
 //....  
 break;  
   
 }  
 return true;  
 }  
  
 return false;  
}

首先判断View是否可用，如果不可用，判断其是否可点击，可点击返回true，不可点击返回false。

如果View可用，则判断其是否可点击，如果不可点击返回false，可点击的话，进行各种事件的判断的处理，并最终返回true。这里关键是up事件中有一个performClick()方法，即用来响应onClickListener监听器的onClick事件。其代码如下：

**[java]** [view plain](http://blog.csdn.net/guolin_blog/article/details/9097463) [copy](http://blog.csdn.net/guolin_blog/article/details/9097463)

1. **public** **boolean** performClick() {
2. sendAccessibilityEvent(AccessibilityEvent.TYPE\_VIEW\_CLICKED);
3. **if** (mOnClickListener != **null**) {
4. playSoundEffect(SoundEffectConstants.CLICK);
5. mOnClickListener.onClick(**this**);
6. **return** **true**;
7. }
8. **return** **false**;
9. }

另外在设置View的监听器时，会使其Clickable使能打开。

1. **public** **void** setOnClickListener(OnClickListener l) {
2. **if** (!isClickable()) {
3. setClickable(**true**);
4. }
5. mOnClickListener = l;
6. }

ViewGroup中的dispatchTouchEvent的实现机制和View中不一样，ViewGroup中有一个onInterceptTouchEvent方法，可以用来拦截子view的事件，该方法返回true则表示拦截，返回false不拦截。

ViewGroup的onInterceptTouchEvent方法示意：

@Override  
public boolean dispatchTouchEvent(MotionEvent ev) {  
 if (action == MotionEvent.*ACTION\_DOWN*) {  
 ………………  
 if (disallowIntercept || !onInterceptTouchEvent(ev)) {  
 ……………………没有拦截，遍历其child  
 final View[] children = mChildren;  
 final int count = mChildrenCount;  
 for (int i = count - 1; i >= 0; i--) {  
 final View child = children[i];  
 //判断child 可见 且在点击区域  
 if ((child.mViewFlags & VISIBILITY\_MASK) == *VISIBLE* || child.getAnimation() != null………………) {  
 //如果child消费了本次down事件则直接返回true  
 if (child.dispatchTouchEvent(ev)) {  
 mMotionTarget = child;  
 return true;  
 }  
 }  
 }  
 }  
 }  
 }  
  
 //如果child没有消费down事件或者没有child，则使用自身的onTouchEvent处理事件  
}

总结就是：

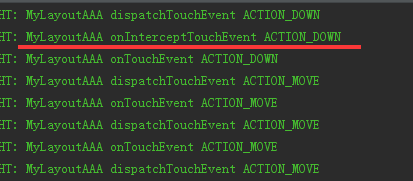
* 所有Touch事件都被封装成MotionEvent对象，包括Touch的位置、历史记录、第几个手指等.
* 事件类型分ACTION\_DOWN,ACTION\_UP,ACTION\_MOVE,ACTION\_POINTER\_DOWN,ACTION\_POINTER\_UP,ACTION\_CANCEL, 每个 一个完整的事件以ACTION\_DOWN开始ACTION\_UP结束，并且ACTION\_CANCEL只能由代码引起.一般对于CANCEL的处理和UP的相同。 CANCEL的一个简单例子：手指在移动的过程中突然移动到了边界外，那么这时ACTION\_UP事件了，所以这是的CANCEL和UP的处理是一致的。
* 事件的处理分别为dispatchTouchEveent()分发事件、onInterceptTouchEvent()拦截事件(ViewGroup中拦截事件)、onTouchEvent()消费事件.
* 事件从Activity.dispatchTouchEveent()开始传递，只要没有停止拦截，就会从最上层(ViewGroup)开始一直往下传递，子View通过onTouchEvent()消费事件。(隧道式向下分发).
* 如果时间从上往下一直传递到最底层的子View，但是该View没有消费该事件，那么该事件会反序网上传递(从该View传递给自己的ViewGroup，然后再传给更上层的ViewGroup直至传递给Activity.onTouchEvent()). (冒泡式向上处理).
* 如果View没有消费ACTION\_DOWN事件，之后其他的MOVE、UP等事件都不会传递过来.
* 事件由父View(ViewGroup)传递给子View,ViewGroup可以通过onInterceptTouchEvent()方法对事件进行拦截，停止其往下传递，如果拦截(返回true)后该事件 会直接走到该ViewGroup中的onTouchEvent()中，不会再往下传递给子View.如果从DOWN开始，之后的MOVE、UP都会直接在该ViewGroup.onTouchEvent()中进行处理。 如果子View之前在处理某个事件，但是后续被ViewGroup拦截，那么子View会接收到ACTION\_CANCEL.
* OnTouchListener优先于onTouchEvent()对事件进行消费。
* TouchTarget是保存手指点击区域属性的一个类，手指的所有移动过程都会被它记录下来, 包含被touch的View。

View的事件dispathTouchEvent或者onTouchEvent处理过程中，如果在ACTION\_DOWN时候返回了false,则后续的事件如move,up便不再被传递进来。如果是再move的时候返回了false，则后续up事件依然会继续传递进来。

在从ViewGroup到View的事件传递中，如果View在Down事件的时候返回了false，则该事件返回到ViewGroup的onTouchEvent中进行处理。如果View在Down事件返回true，即消费了Down事件，那么不管后续的MOVE和UP事件返回false还是true,该事件都不会再返回到ViewGroup中的onTouchEvent了。

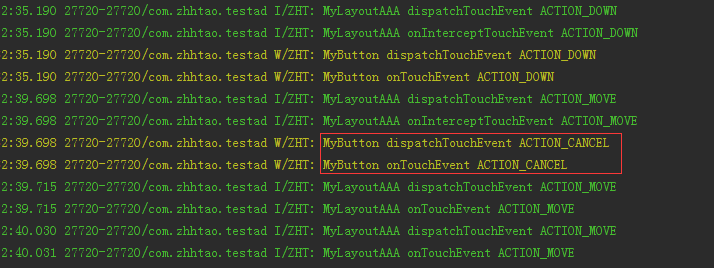
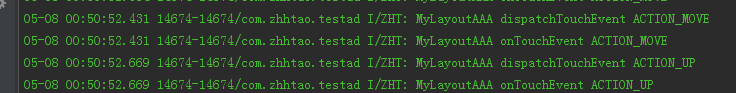
在从ViewGroup到View的事件传递中，如果ViewGroup在onInterceptTouchEvent中在Down事件时返回了true,即拦截了本次事件，那么后续的move和up事件将不再进入onInterceptTouchEvent方法中，而是直接交由ViewGroup的onTouchEvent处理。

@Override  
 public boolean onInterceptTouchEvent(MotionEvent ev) {  
 LogUtil.*i*(this.getClass().getSimpleName()+" onInterceptTouchEvent " + getAction(ev));  
// if (ev.getAction() == MotionEvent.ACTION\_MOVE) return true;  
 if (ev.getAction() == MotionEvent.*ACTION\_DOWN*) return true;  
 return super.onInterceptTouchEvent(ev);  
 }



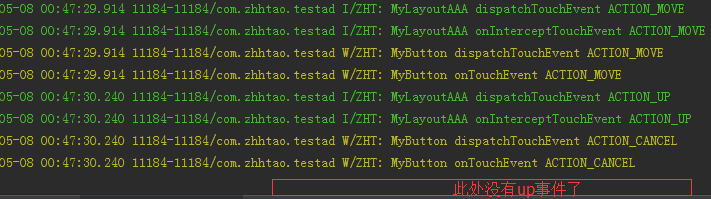
如果ViewGroup在onInterceptTouchEvent中拦截了move事件，即返回了true；如果之前子View已经消费了Down事件，那么此时子View会收到ACTION\_CANCLE的事件。后续的move和up事件将直接交由ViewGroup的onTouchEvent处理。

@Override  
 public boolean onInterceptTouchEvent(MotionEvent ev) {  
 LogUtil.*i*(this.getClass().getSimpleName()+" onInterceptTouchEvent " + getAction(ev));  
 if (ev.getAction() == MotionEvent.*ACTION\_MOVE*) return true;  
// if (ev.getAction() == MotionEvent.ACTION\_DOWN) return true;  
 return super.onInterceptTouchEvent(ev);  
 }

如果子View之前没有消费Down事件，则不会收到ACTION\_CANCLE的事件。

注意，如果ViewGroup在onInterceptTouchEvent中拦截了up事件，则ViewGroup的onTouchEvent是不会收到up事件的。



最后几张图：

1. View不处理事件流程图（View没有消费事件）
2. 2.View处理事件
3. 3.ViewGroup拦截事件

Android事件传递流程：

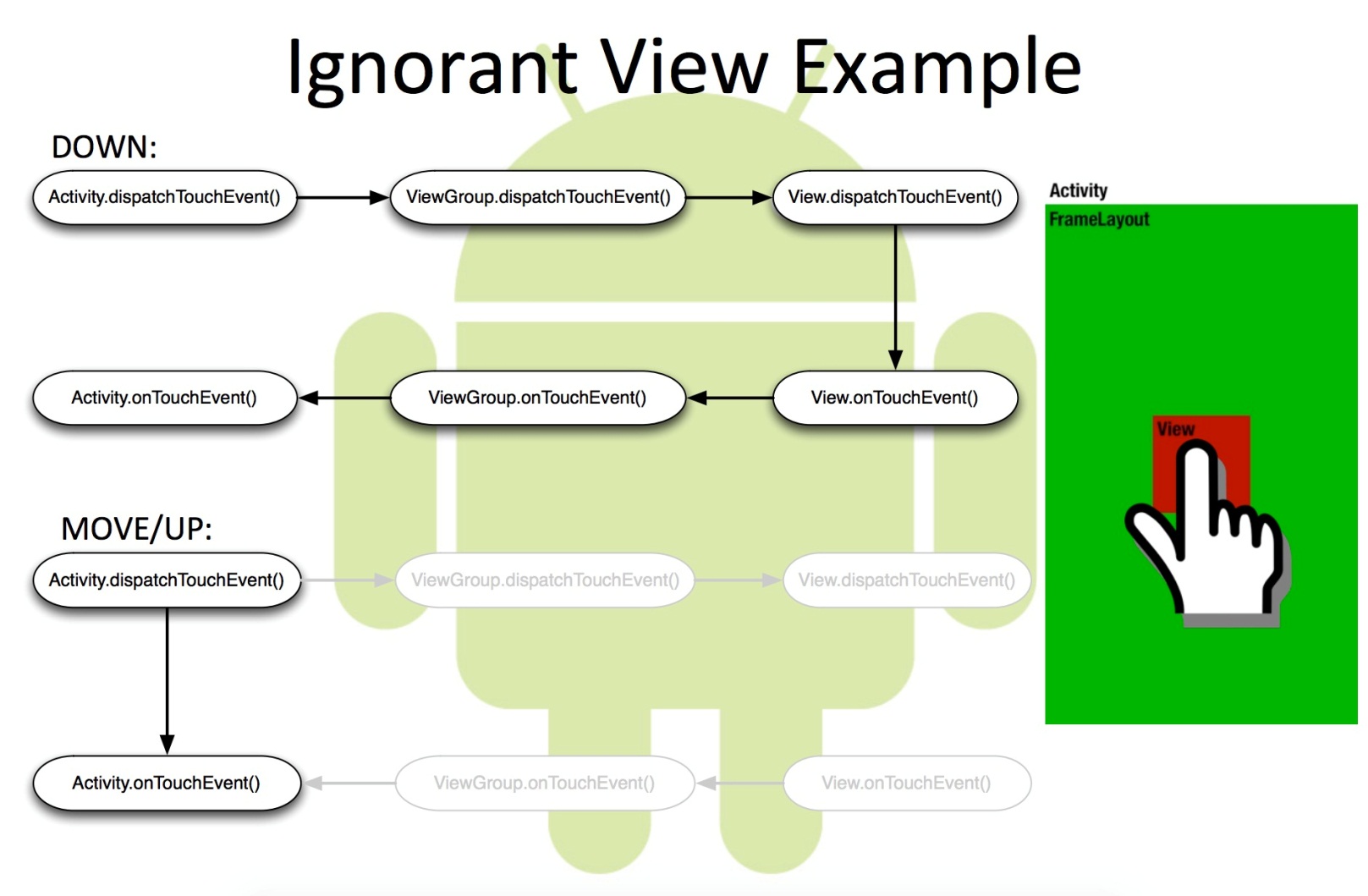
事件都是从Activity.dispatchTouchEvent()开始传递

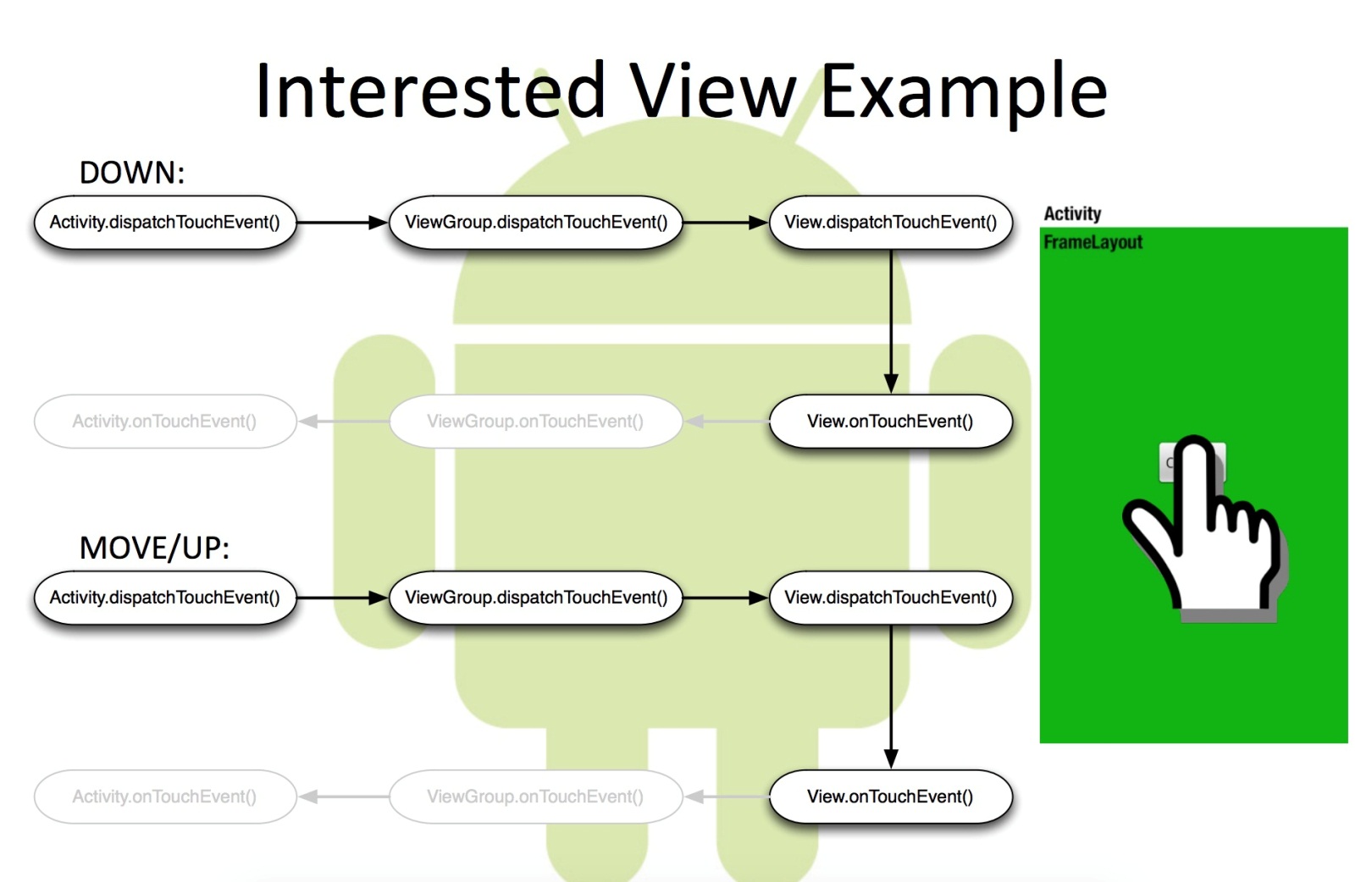
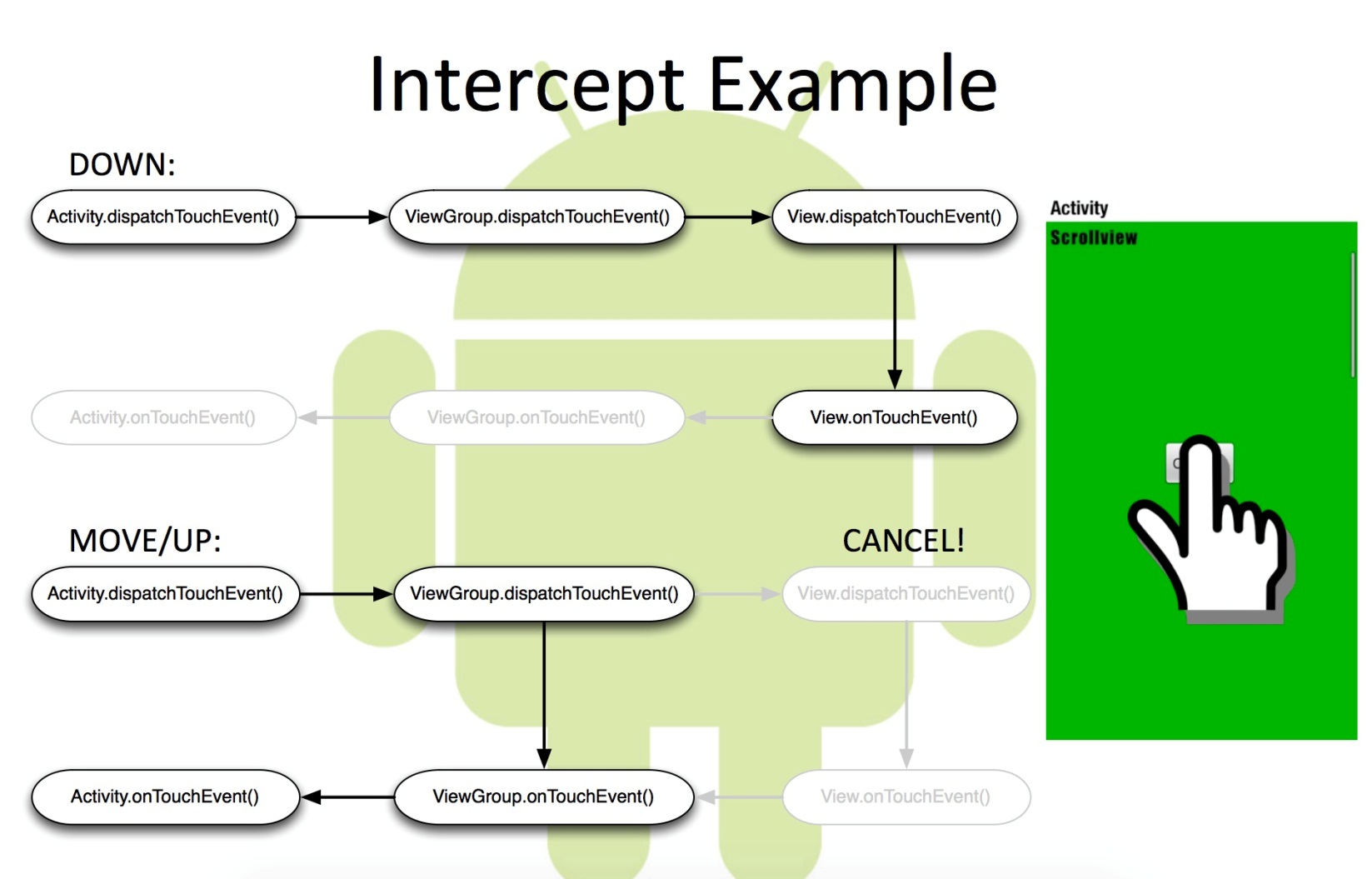
事件由父View传递给子View，ViewGroup可以通过onInterceptTouchEvent()方法对事件拦截，停止其向子view传递

如果事件从上往下传递过程中一直没有被停止，且最底层子View没有消费事件，事件会反向往上传递，这时父View(ViewGroup)可以进行消费，如果还是没有被消费的话，最后会到Activity的onTouchEvent()函数。

如果View没有对ACTION\_DOWN进行消费，之后的其他事件不会传递过来，也就是说ACTION\_DOWN必须返回true，之后的事件才会传递进来。

OnTouchListener优先于onTouchEvent()对事件进行消费





# 自定义点击计数器

*/\*\*  
 \* Created by zhangHaiTao on 2016/5/8.  
 \*/*public class CounterView extends View {  
 Context context;  
 Paint mPaint;  
 Rect mBounds;  
 int mCount = 0;  
 public CounterView(Context context, AttributeSet attrs) {  
 super(context, attrs);  
 this.context = context;  
 mPaint = new Paint(Paint.*ANTI\_ALIAS\_FLAG*);  
 mBounds = new Rect();  
  
 setOnClickListener(new OnClickListener() {  
 @Override  
 public void onClick(View v) {  
 mCount++;  
 invalidate();  
 }  
 });  
 }  
 @Override  
 protected void onDraw(Canvas canvas) {  
 mPaint.setColor(Color.*BLUE*);  
// canvas.drawColor(Color.BLUE); //绘制背景色  
 canvas.drawRect(0,0,getWidth(),getHeight(),mPaint);  
  
 //绘制计数值  
 String text = String.*valueOf*(mCount);  
 mPaint.setColor(Color.*YELLOW*);  
 mPaint.setTextSize(DisplayUtil.*sp2px*(context, 20));  
 mPaint.getTextBounds(text, 0, text.length(), mBounds);  
 int textWidth = mBounds.width();  
 int textHeight = mBounds.height();  
  
 //绘制文字，设置其开始绘制的左下角坐标  
 canvas.drawText(text, getWidth()/2-textWidth/2,  
 getHeight()/2+textHeight/2, mPaint);  
 }  
}

# 自定义滑动添加删除按钮的ListView

关键的自定义ListView代码

*/\*\*  
 \* Created by zhangHaiTao on 2016/5/8.  
 \*/*public class DeletListView extends ListView {  
  
 GestureDetector mGestureDetector;  
 boolean isShowDeleteBtn;  
 Context mContext;  
 int selectedItemIndex;  
 int xDown,yDown;  
  
 ViewGroup selectedItemView;  
 View deleteButton;  
  
 public DeletListView(Context context, AttributeSet attrs) {  
 super(context, attrs);  
 mContext = context;  
  
 mGestureDetector = new GestureDetector(context, new MyGestureDetector());  
 setOnTouchListener(new OnTouchListener() {  
 @Override  
 public boolean onTouch(View v, MotionEvent event) {  
 mGestureDetector.onTouchEvent(event);  
 return false;  
 }  
 });  
 }  
  
 OnDeleteListener mOnDeleteListener;  
 public interface OnDeleteListener {  
 void onDelete(int position);  
 }  
 public void setOnDeleteListener(OnDeleteListener l) {  
 mOnDeleteListener = l;  
 }  
  
   
 class MyGestureDetector extends GestureDetector.SimpleOnGestureListener {  
  
 @Override  
 public boolean onScroll(MotionEvent e1, MotionEvent e2, float distanceX, float distanceY) {  
 LogUtil.*i*("distanceX:"+distanceX);  
 LogUtil.*i*("e1X:"+e1.getX()+" e2 x:"+e2.getX());  
   
 if (e1.getX()-e2.getX() > 100) {//向右滑动100px  
 if (isShowDeleteBtn == false) {

//此处child的只能保存当前屏幕上显示的条目  
 selectedItemView = (ViewGroup)  
 getChildAt(selectedItemIndex -getFirstVisiblePosition());  
  
 RelativeLayout.LayoutParams layoutParams = new RelativeLayout.LayoutParams(  
 ViewGroup.LayoutParams.*WRAP\_CONTENT*,  
 ViewGroup.LayoutParams.*MATCH\_PARENT*);  
 layoutParams.addRule(RelativeLayout.*ALIGN\_PARENT\_RIGHT*);  
 layoutParams.addRule(RelativeLayout.*CENTER\_VERTICAL*);  
  
 deleteButton = View.*inflate*(mContext, R.layout.*delete\_button*, null);  
  
 deleteButton.setOnClickListener(new OnClickListener() {  
 @Override  
 public void onClick(View v) {  
 int position = selectedItemIndex;  
 mOnDeleteListener.onDelete(position);  
 selectedItemView.removeView(deleteButton);  
 deleteButton = null;  
 isShowDeleteBtn = false;  
 }  
 });  
  
 selectedItemView.addView(deleteButton, layoutParams);  
  
 isShowDeleteBtn = true;  
 }  
 }  
 return true;  
 }  
  
 @Override  
 public boolean onFling(MotionEvent e1, MotionEvent e2, float velocityX, float velocityY) {  
  
 LogUtil.*i*("velocityX:"+velocityX+" velocityY:"+velocityY);  
 if (Math.*abs*(velocityX) > Math.*abs*(velocityY)) {  
 //之前的按钮添加逻辑在这里，对滑动事件响应效果不理想  
 }  
 return false;  
 }  
  
 @Override  
 public boolean onDown(MotionEvent e) {  
  
 if (isShowDeleteBtn) {  
 selectedItemView.removeView(deleteButton);  
 deleteButton = null;  
 isShowDeleteBtn = false;  
 }  
  
 xDown = (int) e.getX();  
 yDown = (int) e.getY();  
 selectedItemIndex = pointToPosition(xDown, yDown);  
 LogUtil.*w*("child counts " + getChildCount());  
 LogUtil.*w*("first child " + getFirstVisiblePosition());  
 return false;  
 }  
  
  
 }  
  
  
}

添加的删除button的布局文件：

<?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"  
 >  
 *<!--此时button会把LinearLayout的点击事件消费掉，从而使得LinearLayout无法响应点击-->* *<!--应禁止button点击，这样button的onTouchEvent返回false,点击事件被回送给LinearLayout-->*

*<!—layout\_height的属性只有在存在父类布局时才会生效 -->*<Button xmlns:android="http://schemas.android.com/apk/res/android"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="60dp"  
 android:clickable="false"  
 android:text="delete" />  
</LinearLayout>

Activity中添加自定义的删除监听器：

lvDelete.setAdapter(adapter);  
lvDelete.setOnDeleteListener(new DeletListView.OnDeleteListener() {  
 @Override  
 public void onDelete(int position) {  
 LogUtil.*i*("delete "+position);  
 list.remove(position);  
 adapter.notifyDataSetChanged();  
 }  
});