View事件体系

1. 什么是View？

View是Android中所有控件的基类。（例如：TextView、Button、LinearLayout、RelativeLayout都继承自View。）

ViewGroup：从字面来看是，翻译为控件组，言外之意就是它可以包含许多控件，即一组view。但是ViewGroup也继承自View。所以，像LinearLayout、RelativeLayout这种容器类的控件是继承自ViewGroup，而TextView、Button这类的控件是继承自View。

根据这些概念我们知道它是一个树状结构，LinearLayout不但是一个View而且还是一个ViewGroup，而ViewGroup内部是可以有子View的，而这个子View还可以是ViewGroup，以此类推。

1. View的位置参数

首先，这里说的参数都是相对于父容器的参数。

top = getTop() ：左上角纵坐标。 ( left ,top )

left = getLeft()：左上角横坐标。

right = getRight()：右下角横坐标。

bottom = getBottom()：右下角纵坐标。 (right ,bottom )

width = right - left

height = bottom - top

从系统Android 3.0开始，View增加了：x，y，translationX，translationY 也是相对于父 容器的坐标。其中x和y是View左上角的坐标，translationX和translationY是左上角相 对于父容器的偏移量。

x = getLeft() + translationX

Y = getTop() + translationY

注意：当View在平移过程中，top和left 表示的是原始左上角的位置信息，其值并不 会发生变化，此时发生改变的只有x，y，translationX，translationY这四个参数。

3、（1）MotionEvent

当用户触摸屏幕时，就会产生一个MotionEvent对象。MotionEvent对象中包含这个触摸事件的位置和时间以及一些其他的细节的信息。

获取：1）重载Activity中onTouchEvent ( MotionEvent event ) 方法；

2）View对象调用View.setOnTouchListener接口实现

onTouch( View v, MotionEvent event ) 方法；

常用方法说明：

getX() 和getY() 获取的是相对于当前View的左上角的x、y坐标。

getRawX() 和 getRawY 获取的是相对于屏幕左上角的x、y坐标（不管activity 是否有titleBar 或是否全屏）。

这两个都代表点击事件的坐标。

1. TouchSlop 是系统所能识别出的被认为最小的滑动距离，假如滑动距离小于这个 值那么系统不认为这不是一个滑动事件，它是一个常量，跟设备有关，不同设备上这个 值可能不同。我们可以使用这个值来做一些屏幕触摸事件的过滤。

（3）VelocityTracker

速度追踪，用于追踪手指在滑动过程中的速度，包括水平速度和垂直速度。

使用：使用addMovement( MotionEvent event )将MotionEvent添加到 VelocityTracker的实例中；

代码：VelocityTracker velocityTracker = VelocityTracker.obtain();

velocityTracker.addMovement( event );

谨记：1）在获取速度之前一定要调用computeCurrentVelocity( int arge );

velocityTracker.computeCurrentVelocity( 1000 );

int xVelocity = (int) velocityTracker.getXVelocity();

Int yVelocity = (int) velocityTracker.getYVelocity();

computeCurrentVelocity 这个方法的参数是时间单元或者时间间隔，它的 单位是毫秒（ms）。

1. 这里的速度指的是一段时间内手指所滑动过的像素数。

速度 = （终点位置 - 起点位置）/ 时间段

例如，设置时间间隔为velotityTracker.computeCurrentVelocity(1000) ，

手指水平方向从左到右滑过100像素，那么水平速度就是100像素/1000 ms。

假如设置单位时间为100毫秒，velotityTracker.computeCurrentVelocity(100)，

那么水平速度为10像素/ms，（这里假设滑动过程是匀速的）。

另外：当不需要使用的时候，需要调用clear方法来重置并回收内存：

velocityTracker.clear();

velocityTracker.recycle();

（4）GestureDetetor

手势检测，用于辅助用户的单击、滑动、长按、双击等行为。

使用：首先，实现OnGestureListener接口，另外OnDoubleTapListener能够监听 双击行为。

GestureDetetor mGestureDetetor =

new GestureDetetor( context , new OnGestureListener(){...} ) ;

监听双击行为：

mGestureDetetor.setOnDoubleTapListener(

new OnDoubleTapListener(){...});

然后，在接管View的onTouchEvent的方法中添加如下实现：

boolean consume = mGestureDetetor.onTouchEvent( event );

return consume;

注意：在实际开发中可以不使用GestureDetetor，完全可以在View的 onTouchEvent方法中实现所需监听，这个就看个人喜好了。建议：如果 要监听双击效果使用GestureDetetor，其他情况在onTouchEvent中实现。

1. View的滑动
2. 使用scrollTo/scrollBy

首先，这两个方法改变的的是View整体内容的位置（不管是容器View， LinearLayout...或者是控件View，TextView...）。而不是改变的View本身的位置。

View的位置是指View的四个顶点组成。

**public void** scrollTo(**int** x, **int** y) {  
 **if** (**mScrollX** != x || **mScrollY** != y) {  
 **int** oldX = **mScrollX**;  
 **int** oldY = **mScrollY**;  
 **mScrollX** = x;  
 **mScrollY** = y;  
 invalidateParentCaches();  
 onScrollChanged(**mScrollX**, **mScrollY**, oldX, oldY);  
 **if** (!awakenScrollBars()) {  
 postInvalidateOnAnimation();  
 }  
 }  
}

**public void** scrollBy(**int** x, **int** y) {  
 scrollTo(**mScrollX** + x, **mScrollY** + y);  
}

我们可以看出在View的内部有mScrollX和mScrollY两个变量，可以通过getScrollX和getScrollY获取，这个两个参数的单位为像素。这两个变量记录着View内容的四个顶点的移动距离，假如View内容自View的位置向左移动那么mScrollX为正值，反之为负值。假如View内容自View的位置向上移动mScrollY为正值，反之为负值。假如View内容移动的距离超出了View自身的大小，那么将隐藏View内容超出的部分。

scrollTo ：使View内容基于View内容的【初始位置】移动。

scrollBy ：使View内容基于View内容的【当前位置】移动。

1. 使用动画

使用动画来移动View，主要操作的View的translationX和translationY的属性。

View动画是对View的影像进行操作，他并不是真的能改变View的位置参数，包括（宽/高），使用属性动画就不回存在上述问题，如果使用属性动画的话，为了能兼容3.0以下的版本，需要采用开源动画库nineoldandroids。

1. 改变布局参数

即改变 LayoutParams。

三种欢动的对比：

scrollTo/scrollBy：操作简单适用于View的内容操作。

动画：操作简单，主要对于没有交互的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。

1. 弹性滑动

弹性滑动有一个共同的思想：将一次大的滑动分成若干次小的滑动 ，并在一段时间内完成。

（1）使用Scroller

Scroller是一个专门处理滚动的工具类。当View的scrollTo和scrollBy方法滑动时，其过程都是瞬间完成的没有过渡效果。这时候就需要Scroller来实现过度效果的滑动了。

用法：1）创建一个Scroller对象；

2）调用startScroll()方法初始化滚动数据并刷新界面，紧接着调用 invalidate()才会开始滑动；

3）重写computeScroll方法，并在内部完成平滑滚动的逻辑，最后调 用invalidate()方法；

例子：

Scroller scroller = **new** Scroller(context);

**public void** smoothSctollTo(**int** x, **int** y) {  
 **scroller**.startScroll(10, 10, -100, -200, 1000);  
 invalidate();  
 }

@Override  
 **public void** computeScroll() {  
 **if** (**scroller**.computeScrollOffset()) {  
 scrollTo(**scroller**.getCurrX(), **scroller**.getCurrY());  
 invalidate();  
 }  
 }

当我们调用startScroll()方法时，Scroller的内部什么也没有做，他只是保

存了我们传递的几个参数，第一、二个参数分别代表滑动的起点的x,y坐标，

第三、四个参数代表滑动的距离，最后一个表示整个滑动所需要的时间，注意这里

的滑动指的是View内容的滑动，而并非是View的滑动。

滑动原理：首先我们调用startScroll()方法保存好我们滑动的起始位置、滑

动距离、滑动时间。只有当我们调用invalidate()方法时会导致View重绘，在View

的draw方法中又会调用computeScroll()方法（此方法在View中是一个空实现，我 们需要将滑动的逻辑在这个方法里实现），在computeScroll()方法中我们首先要通

过Scroller中computeScrollOffset()方法判断我们在startScroll()中设置的滑

动距离是否滑动完（同时也重新计算当前mCurrX和mCurrY的值），返回true则 是没有滑动完，这时需要向Scrolle获取当前的mCurrX和mCurrY，传递给View的

scrollTo()方法继续滑动，并且继续调用invalidate()继续重绘，以此类推，直到

startScroll()中设置的滑动距离滑动完。

computeScrollOffset()方法：

**public boolean** computeScrollOffset() {  
 ...  
 **int** timePassed = (**int**)(AnimationUtils.*currentAnimationTimeMillis*() - **mStartTime**);  
 **...**  
 **switch** (**mMode**) {  
 **case *SCROLL\_MODE***:  
 **final float** x = **mInterpolator**.getInterpolation(timePassed \*

**mDurationReciprocal**);  
 **mCurrX** = **mStartX** + Math.*round*(x \* **mDeltaX**);  
 **mCurrY** = **mStartY** + Math.*round*(x \* **mDeltaY**);  
 **break**;

}

...  
 **return true**;  
 }

这个方法有两个作用：1）根据时间流逝的百分比来计算出mCurrX和mCurrY的值。

2）判断是否完成滚动，返回值true则没有完成，返回false则完成。

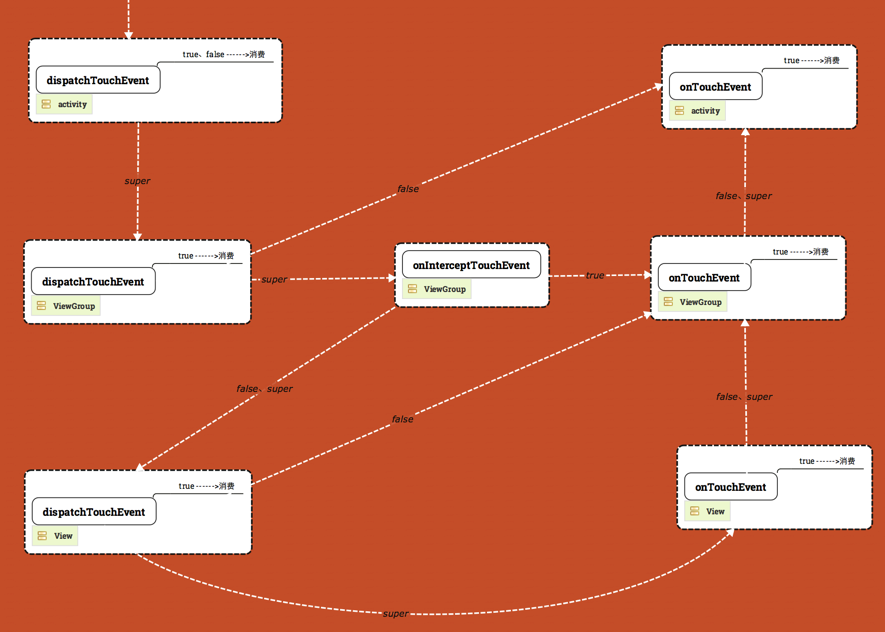
（2）使用动画 略。。。

1. View的事件分发

View事件的分发主要是MotionEvent对象的分发，包括从手指触摸屏幕（ACTION\_DOWN）开始到手指离开屏幕（ACTION\_UP）为止的一系列事件从Activity——>ViewGroup——>View的顺序的分发。

**第一部分 针对ACTION\_DOWN**

事件分发走向总图：



说明：

1. 事件产生后首先从Activity的dispatchTouchEvent()开始做分发（这里需要说明，假如Activity的dispatchTouchEvent()方法执行的是super.XXX()父类的实现，那么最终调用的是Activity的DecorView的对象的dispatchTouchEvent()方法，即Activity的根View继承自FrameLayout）。
2. dispatchTouchEvent()和onTouchEvent()的框里有一个【true---->消费】的字，表示如果方法返回true，代表事件就此被消费，不会往别的地方传递，事件终止，相应的方法可以根据具体的事件作出相应的操作。（假如事件没有终止，继续传递，那么只要事件经过的方法，相应的方法也能作出相应的操作，这是理解事件被消费的一个角度。）

**由上图总结的结论**：

1. 如果我们没有对控件的方法更改返回值或者进行重写，而直接使用super调用父类的默认实现，那么整个事件（从ACTION\_DOWN到ACTION\_UP的一系列事件）的流向会是从Activity——>ViewGroup——>View从上到下调用dispatchTouchEvent()，其中ViewGroup层对调用一下自己的拦截方法onInterceptTouchEvent()，一直到叶子节点（View）的时候，再由View——>ViewGroup——>Activity从下往上调用onTouchEvent方法，这是一个完整的路径。
2. ViewGroup和View的dispatchTouchEvent()方法是做事件分发的。
3. ViewGroup和View的onTouchEvent()是做事件处理。
4. ViewGroup的onInterceptTouchEvent()方法：返回true则把事件传递给自己的 onTouchEvent()方法，否则向下传递。

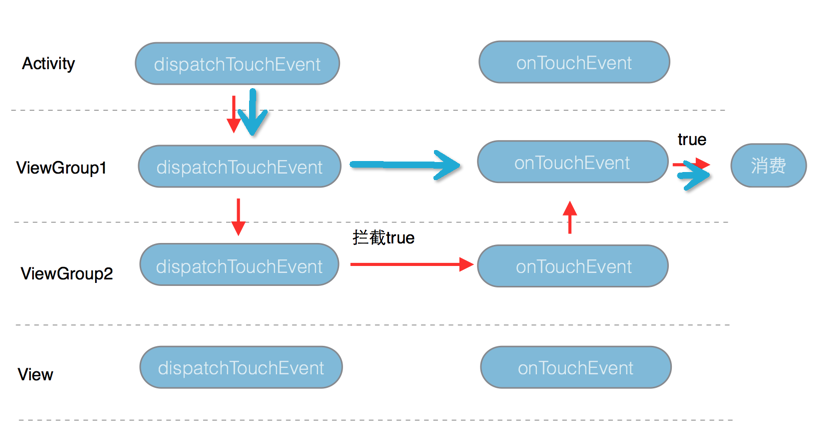
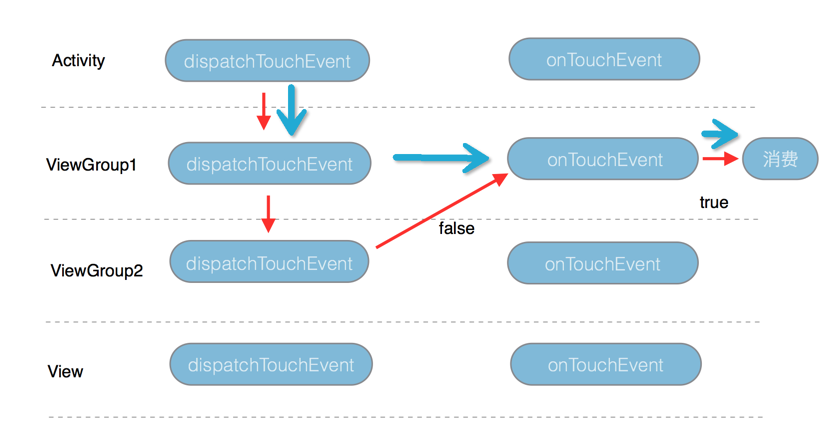
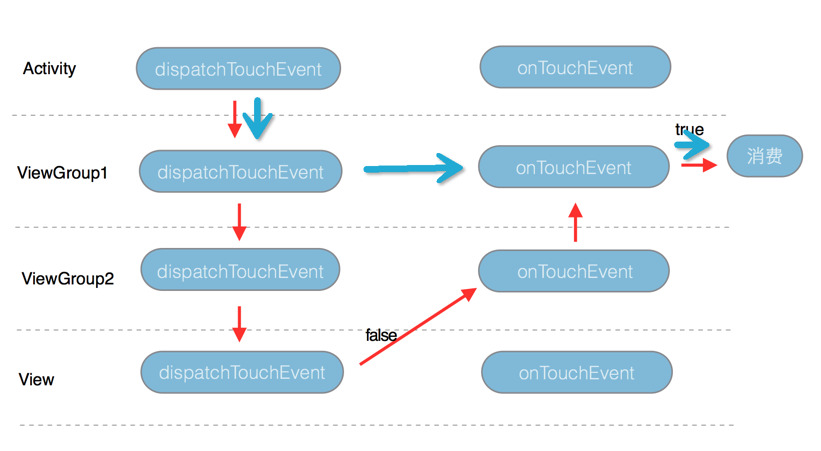
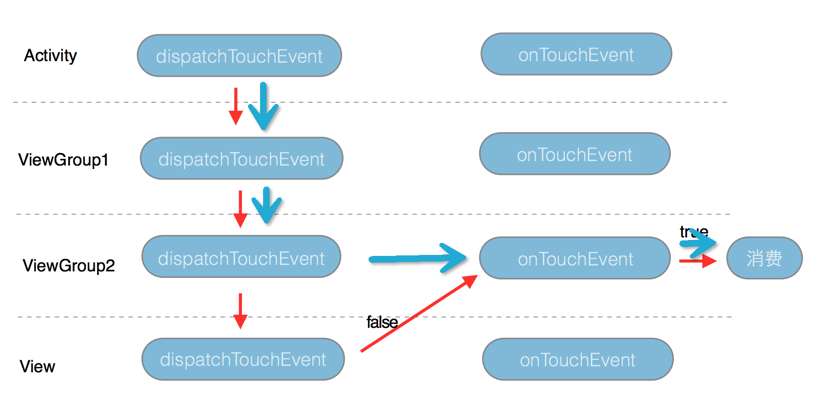
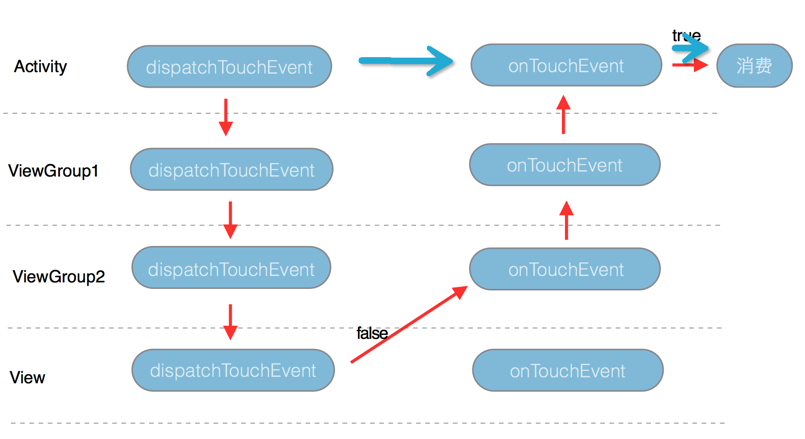
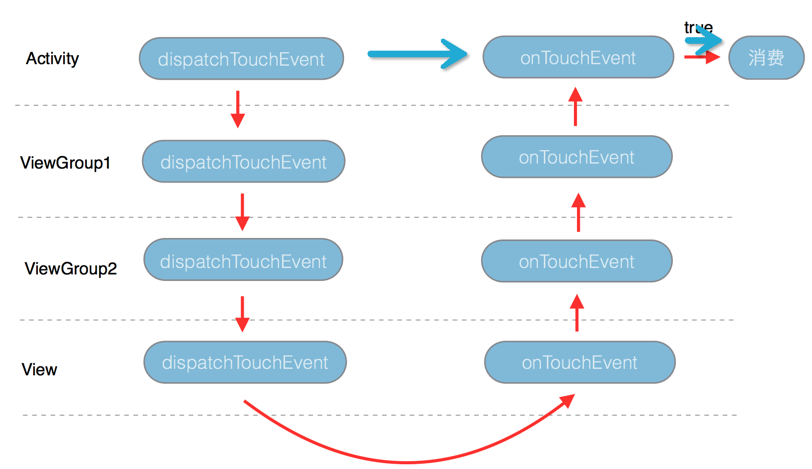
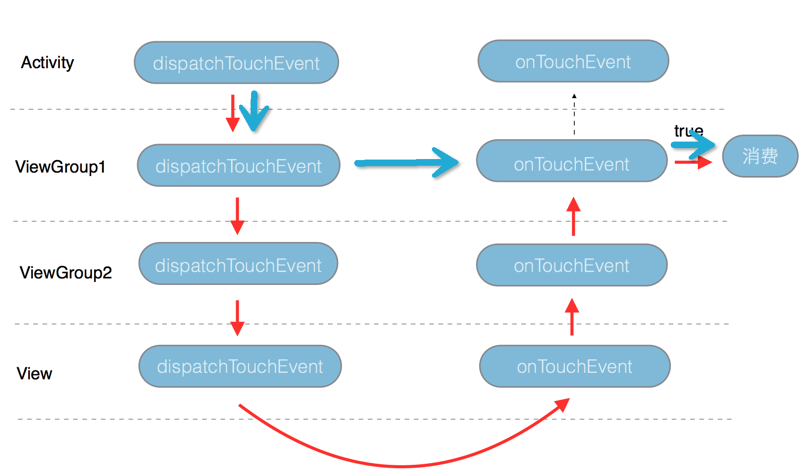
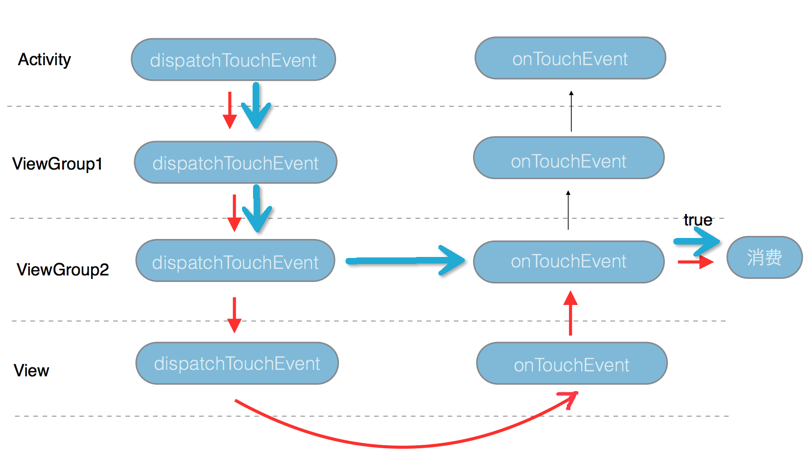
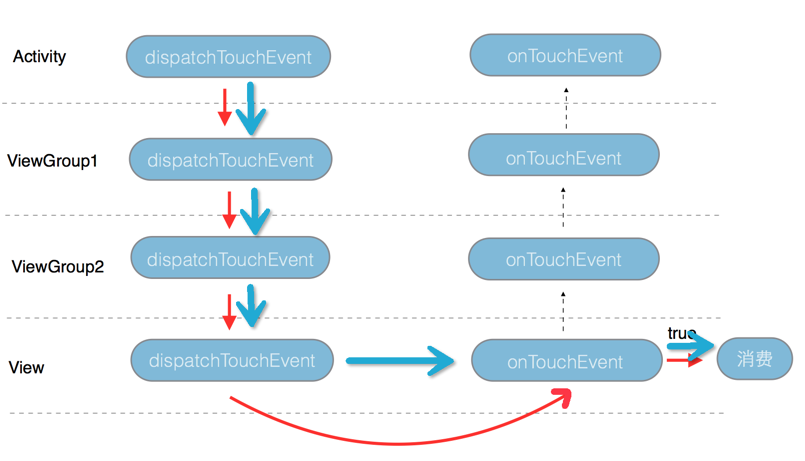
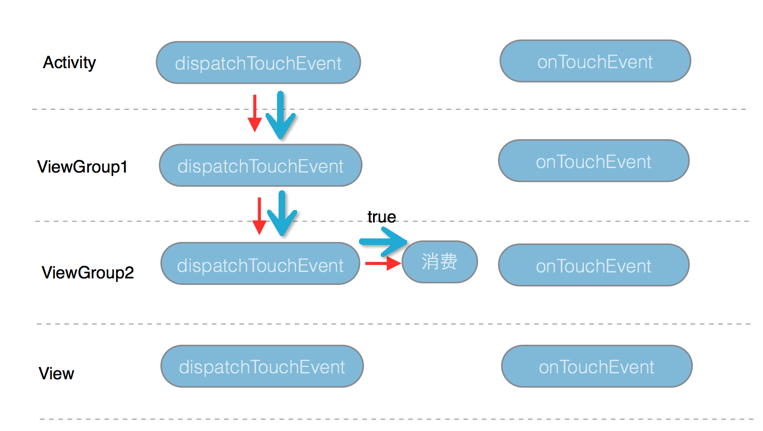
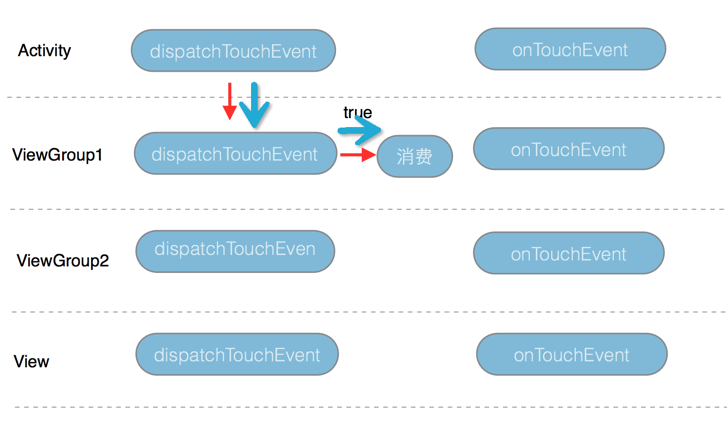
**第二部分 针对ACTION\_MOVE和ACTION\_UP**

这两个事件跟ACTION\_DOWN的传递路径不同，并不是能接收到ACTION\_DOWN的就一定能接收到ACTION\_MOVE和ACTION\_UP这两个事件。

总结：ACTION\_DOWN在哪个控件被消费（dispatchTouchEvent() 或者onTouchEvent() return true），ACTION\_MOVE和ACTION\_UP通过dispatchTouchEvent()分发，就会从上到下传递到这个控件的相应方法，不会向下传递，在这个控件相应方法被消费。 如果ACTION\_DOWN在dispatchTouchEvent()方法被消费，那么ACTION\_MOVE和ACTION\_UP也就在这个控件中的dispatchTouchEvent()方法被消费。

如果ACTION\_DOWN在onTouchEvent()方法被消费，那么ACTION\_MOVE和ACTION\_UP通过dispatchTouchEvent()分发，不会向别的控件传递，就在这个控件中的onTouchEvent()方法被消费。

如下示例：红色代表ACTION\_DOWN；蓝色代表ACTION\_MOVE和ACTION\_UP



**事件分发原理探析**

上文中是十分精简的总结，请务必记住。本篇将补充上文中的知识点，并且根据源码分析事件的分发和消费的原理。

知识要点：

1. 如果一个View设置了OnTouchListener，那么当这个View将收到的事件分发后，

OnTouchListener中的onTouch()方法会被调用，这时，如果onTouch()方法返回

true，那么事件序列在onTouch()方法中被消费，这个View的onTouchEvent()方法 不会被调用。反之，事件会继续走到onTouchEvent()中。

当事件走到onTouchEvent()方法后，如果当前View设置了OnClickListener，那么它

的onClick()会被调用（这里注意：是当onTouchEvent()方法收到

MotionEvent.ACTION\_UP之后才会去调用OnClickListener的onClick()方法）。

可见优先级：OnTouchListener > onTouchEvent > OnClickListener

1. 某一个View一旦决定拦截事件，那么当这一些列的事件传递到这个View时，

ACTION\_MOVE和ACTION\_UP都会交给这个View处理，不会再流向其他的View

了（详细见**第二部分 针对ACTION\_MOVE和ACTION\_UP** 总结）并且不会再次

去调用onInterceptTouchEvent()方法去询问它是否要拦截了。

3、View的onTouchEvent()方法默认返回true，除非他是不可点击的（clickable和 longClickable同时为false）。longClickable默认为false，而clickable视情况而定，

如：Button的clickable默认为 true，而TextView的clickable默认为false。但是当

设置View的OnClickListener时会首先设置View的clickable为true。

4、enable属性不影响onTouchEvent()的默认返回值。哪怕enable为false，只要

clickable和longClickable有一个为true那么它的onTouchEvent就返回true。

5、事件总是先传递给父元素，然后再由父元素分发给子元素，通过

requestDisallowInterceptTouchEvent()方法可以在子元素中干扰父元素的事件分发 过程，但ACTION\_DOWN事件除外。

**事件分发的源码解析**

* **Activity对点击事件的分发**

当一个点击操作发生时事件最先传递给当前Activity ，由Activity的dispatchTouchEvent()方法来分发，在此方法内具体是由Activity内部的Window来处理的，调用的是getWindow().superDispatchTouchEvent()方法，我们进入到Window的实现类PhoneWindow类里面，发现superDispatchTouchEvent()方法最终调用的是Activity的根view，即DecorView（setContentView所设置的View的父容器，继承自FrameLayout）的dispatchTouchEvent()方法。

这样一个点击事件就从Activity传递到了顶级View。

* **顶级ViewGroup对事件的分发过程**

当View的dispatchTouchEvent()收到事件后（View和ViewGroup对于dispatchTouchEvent()的实现是不一样的，虽然ViewGroup继承自View但是ViewGroup完全重写了dispatchTouchEvent()方法，但是下面的逻辑是一样的。），逻辑是这样的：如果ViewGroup拦截事件，即InterceptTouchEvent()返回true，（当然非ViewGoup不会走这一步，但是接下来的执行是一样）这时如果它的mOnTouchListener被设置且enable为true，则onTouch()会被调用，如果onTouch()返回false（返回true则事件被消费，不会往下传递了），那么onTouchEvent()会被调用，如果设置了mOnClickListener那么onClick()会被调用。这是分析了mOnTouchListener和mOnClickListener都被设置的情况，加入都没有设置那么InterceptTouchEvent()返回true后会立即调用onTouchEvent()。

虽然View与ViewGroup的dispatchTouchEvent()方法实现不同，但是View的实现与上面描述一致，只是ViewGroup的实现比View稍微复杂一点，下面我们选取重点处理逻辑分析一下ViewGroup的dispatchTouchEvent()方法的源码：

首先：

*//* *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();  
}

ViewGroup分发一次完整的事件是从ACTION\_DOWN开始，到ACTION\_UP结束，中间会有不定数个ACTION\_MOVE。上面这个代码是dispatchTouchEvent()方法分发之前最先做的事情，当dispatchTouchEvent()收到的是一个ACTION\_DOWN的事件时，它会认为这是一个新的触摸事件，所以重置先前所有的触摸状态（包括mFirstTouchTarget和FLAG\_DISALLOW\_INTERCEPT标志位）。

接下来：

*// 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**;  
}

介绍两个变量：

**mFirstTouchTarget ：**当ViewGroup的子元素成功处理事件时，**mFirstTouchTarget** 会指向子元素。

***FLAG\_DISALLOW\_INTERCEPT***标志位：禁止VeiwGroup来拦截ACTION\_DOWN以外的事件，一般是通过子View的requestDisallowInterceptTouchEvent()来设置。

我们通过判断条件看出，只要是一个ACTION\_DOWN事件，那么他一定会执行到onInterceptTouchEvent()方法，好了，接下来会有ACTION\_MOVE和ACTION\_UP事件到来，假如ACTION\_DOWN事件被子View消耗，那么**mFirstTouchTarget** 将不为空这个判断会继续通过，这时候就要看子View有没有设置***FLAG\_DISALLOW\_INTERCEPT***标志位，假如没有设置，那么ViewGroup通过onInterceptTouchEvent()继续拦截ACTION\_MOVE和ACTION\_UP事件，假如设置了那么就是不拦截；假如ACTION\_DOWN事件不被被子View消耗，那么**mFirstTouchTarget**为空，接下来会有ACTION\_MOVE和ACTION\_UP事件到来，就不会在调用onInterceptTouchEvent()方法了，这里也说明onInterceptTouchEvent()并不是被一直调用。

接下来，我们假设ViewGroup没有拦截事件，即onInterceptTouchEvent()返回false则会执行下面的代码：

**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**;  
 }  
  
 *// The accessibility focus didn't handle the event, so clear  
 // the flag and do a normal dispatch to all children.* ev.setTargetAccessibilityFocus(**false**);  
}

*// 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***);  
} **else** {  
 *// Dispatch to touch targets, excluding the new touch target if we already  
 // dispatched to it. Cancel touch targets if necessary.* TouchTarget predecessor = **null**;  
 TouchTarget target = **mFirstTouchTarget**;  
 **while** (target != **null**) {  
 **final** TouchTarget next = target.**next**;  
 **if** (alreadyDispatchedToNewTouchTarget && target == newTouchTarget) {  
 handled = **true**;  
 } **else** {  
 **final boolean** cancelChild = *resetCancelNextUpFlag*(target.**child**)  
 || intercepted;  
 **if** (dispatchTransformedTouchEvent(ev, cancelChild,  
 target.**child**, target.**pointerIdBits**)) {  
 handled = **true**;  
 }  
 **if** (cancelChild) {  
 **if** (predecessor == **null**) {  
 **mFirstTouchTarget** = next;  
 } **else** {  
 predecessor.**next** = next;  
 }  
 target.recycle();  
 target = next;  
 **continue**;  
 }  
 }  
 predecessor = target;  
 target = next;  
 }  
}

接下来ViewGroup会遍历它的子View，然后判断子View是否能接收到这个的点击事件，能否接收到点击事件满足以下两个条件即可：子View是否在播放动画和点击事件的左边是否落在子View的区域内，如果满足则点击事件交由这个子View来处理。

在for循环内主要的判断：

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

*canViewReceivePointerEvents()*方法判断是否有动画或者是否可见；

isTransformedTouchPointInView()方法判断点击事件是否落在子View区域内;

假如没有continue;说明有一个子View能接收到事件，那么就会继续执行到一个重要的判断：

**if** (dispatchTransformedTouchEvent(ev, **false**, child, idBitsToAssign))

其中dispatchTransformedTouchEvent(ev, **false**, child, idBitsToAssign)方法我们暂时不看源码，但是它里面执行的内容一定如下：

1. 假如child不为空那么它一定会调用child.dispachtTouchEvent()方法，这时候这个判断返回值=child.dispachtTouchEvent()；
2. 假如child为空，那么ViewGroup一定会调用super.dispachtTouchEvent()方法，即Veiw的dispachtTouchEvent()方法，这时候这个判断的返回值=super.dispachtTouchEvent()；

根据上面（1）和（2）的结论我们继续分析：

* 假设我们子View的dispachtTouchEvent()返回的值没有被重置，那么会按照View的事件分发逻辑去执行。如果子View的dispachtTouchEvent()返回true说明子View消耗了事件，那么mFirstTouchTarget会被赋值（具体在addTouchTarget(child, idBitsToAssign)方法内被赋值），并且跳出整个子View的循环遍历。代码继续执行mFirstTouchTarget为空的判断不会通过。如果返回false，说明子View没有消耗事件（我们这里假设子View是一个view，而不是一个ViewGroup），那么mFirstTouchTarget不会被赋值，代码继续往下走会判断mFirstTouchTarget是否为空，若为空则调用ViewGroup的super.dispachtTouchEvent()；将事件传递给自己的OnTouchListener或者onTouchEvent方法或者OnClickListener。
* 我们手动设置子View的dispachtTouchEvent()返回值为true：

那么dispatchTransformedTouchEvent()方法执行的结果是true。

继续执行，mFirstTouchTarget会被赋值，并且跳出整个子View的循环遍历。

继续执行，因为mFirstTouchTarget不为空，所以不会执行到ViewGroup的super.dispachtTouchEvent()；

* 我们手动设置子View的dispachtTouchEvent()返回值为false：

那么dispatchTransformedTouchEvent()方法执行的结果是false。判断不会通过，那么mFirstTouchTarget也就不会被赋值。

继续执行mFirstTouchTarget为空，则会调用ViewGroup的super.dispachtTouchEvent()；

然后事件会被传递给ViewGroup的OnTouchListener或者onTouchEvent方法或者OnClickListener。

这里需要注意：只有ViewGroup执行到super.dispachtTouchEvent()方法时（其实执行的就是View的dispachtTouchEvent()方法）才会去执行OnTouchListener或者onTouchEvent方法或者OnClickListener。

上面的源码将ViewGroup的dispachtTouchEvent()方法分析完了，它很有力的说明了 **第一部分 针对ACTION\_DOWN** 的流程图。ViewGroup的onTouchEvent()方法其实调用的是View的onTouchEvent()方法，见下面；

* **View对事件的分发过程**

View的dispachtTouchEvent()方法很简单，贴出源码不多废话：

**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**;  
 }  
}

其中result 就是dispachtTouchEvent()方法的返回值，默认为false；

接着分析onTouchEvent()方法：

**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***);  
 }

看这个判断，注释：当View设置了setEnabled(**false**);它仍然可以消费事件，只是不响应它们。

接下来看：

**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();  
 }  
 }  
 }  
  
 **if** (**mUnsetPressedState** == **null**) {  
 **mUnsetPressedState** = **new** UnsetPressedState();  
 }  
  
 **if** (prepressed) {  
 postDelayed(**mUnsetPressedState**,  
 ViewConfiguration.*getPressedStateDuration*());  
 } **else if** (!post(**mUnsetPressedState**)) {  
 *// If the post failed, unpress right now* **mUnsetPressedState**.run();  
 }  
  
 removeTapCallback();  
 }  
 **mIgnoreNextUpEvent** = **false**;  
 **break**;

。。。

}

**return true**;

}

根据判断看出只要***CLICKABLE*** 或者 ***LONG\_CLICKABLE*** 有一个为true那么onTouchEvent()就会消耗这个事件，其中View的***LONG\_CLICKABLE***默认为true，而***CLICKABLE*** 是否为true与具体的View有关，可点击的View默认为true，不可点击的的View默认为false，例如：Button的默认为true,而TextView的默认为false。

可以通过下面的方式设置这两个属性为true：

***CLICKABLE ：***setClickable(**true**) 或 setOnClickListener()；

***LONG\_CLICKABLE：***setLongClickable(**true**) 或 setOnLongClickListener();

继续看，对ACTION\_UP事件的处理，因为当接收到这个事件时说明这一系列事件都处理完了，在这个事件处理过程中会调用perFormClick();方法，在这个方法里面会判断假如设置了OnClickListerner那么就会调用它的onClick()方法。

下面特殊说明一下OnLongClickListener继续看onTouchEvent()的源码：

**case** MotionEvent.***ACTION\_DOWN***:  
 **mHasPerformedLongPress** = **false**;  
  
 **if** (performButtonActionOnTouchDown(event)) {  
 **break**;  
 }  
  
 *// Walk up the hierarchy to determine if we're inside a scrolling container.* **boolean** isInScrollingContainer = isInScrollingContainer();  
  
 *// For views inside a scrolling container, delay the pressed feedback for  
 // a short period in case this is a scroll.* **if** (isInScrollingContainer) {  
 **mPrivateFlags** |= ***PFLAG\_PREPRESSED***;  
 **if** (**mPendingCheckForTap** == **null**) {  
 **mPendingCheckForTap** = **new** CheckForTap();  
 }  
 **mPendingCheckForTap**.**x** = event.getX();  
 **mPendingCheckForTap**.**y** = event.getY();  
 postDelayed(**mPendingCheckForTap**, ViewConfiguration.*getTapTimeout*());  
 } **else** {  
 *// Not inside a scrolling container, so show the feedback right away* setPressed(**true**, x, y);  
 checkForLongClick(0, x, y);  
 }  
 **break**;

当View接受到ACTION\_DOWN事件时，会通过checkForLongClick()方法，检测是否设置了OnLongClickListener，若设置了，则会调用它的onLongClick()方法，同时mHasPerformedLongPress记录它的返回值，当onTouchEvent()处理ACTION\_UP事件时，**if** (!**mHasPerformedLongPress** && !**mIgnoreNextUpEvent**) 会判断mHasPerformedLongPress的值，那么最终的结果是，假如onLongClick()返回了true，说明这个事件由OnLongClickListener处理了不会再往别的方法传递了，如果返回false,那么onLongClick()执行完了之后，假如设置OnClickListener那么会继续执行onClick()方法。