# 事件分发总结

## Activity

Activity只需要关注dispatchTouchEvent()和onTouchEvent()方法：

public boolean dispatchTouchEvent(MotionEvent ev) {

if (ev.getAction() == MotionEvent.*ACTION\_DOWN*) {  
 onUserInteraction();  
 }  
 if (getWindow().superDispatchTouchEvent(ev)) {  
 return true;  
 }  
 return onTouchEvent(ev);  
}

public boolean onTouchEvent(MotionEvent event) {

if (mWindow.shouldCloseOnTouch(this, event)) {  
 finish();  
 return true;  
 }  
  
 return false;  
}

在dispatchTouchEvent方法中，getWindow().superDispatchTouchEvent(ev)最终会调用到ViewGroup的dispatchTouchEvent()方法。

## ViewGroup

public boolean dispatchTouchEvent(MotionEvent ev) {

// onInterceptTouchEvent（）默认返回false disallowIntercept默认是false，所以交由子控件处理事件  
 if (disallowIntercept || !onInterceptTouchEvent(ev)) {  
 */\*\*   
 \* 1. 通过for循环，遍历了当前ViewGroup下的所有子View  
 \* 2. 判断当前遍历的View是不是正在点击的View，从而找到当前被点击的View  
 \* 3. 条件判断的内部调用了该View的dispatchTouchEvent()  
 \*/* if (child.dispatchTouchEvent(ev)) {  
 mMotionTarget = child;  
 return true;  
 }  
 if(mMotionTarget == null)  
 return super.dispatchTouchEvent(ev);  
 }  
}

ViewGroup的onTouchEvent调用的是View类的。

## View

public boolean dispatchTouchEvent(MotionEvent event) {

if (mOnTouchListener != null && (mViewFlags & ENABLED\_MASK) == ENABLED &&  
 mOnTouchListener.onTouch(this, event)) {  
 return true;  
 }  
 return onTouchEvent(event);  
}

public boolean onTouchEvent(MotionEvent event) {

// 判断改控件是否可点击，或者可以长按，如果是，则进入swtich判断ACTION\_DOWN,ACTION\_MOVE,ACTION\_UP  
 if(可以点击或可长按) {  
 switch（event.getAction()）{  
 case MotionEvent.*ACTION\_UP*:  
 // 如果设置了点击事件（setOnClickListener），就调用点击事件的onClick()方法，并且返回true  
 performClick();  
 break;  
 }  
 // 若该控件可点击，就一定返回true  
 return true;  
 }  
 // 若该控件不可点击，就一定返回false  
 return false;  
}

总结：

1. 调用顺序：

Activity: dispatchTouchEvent -> onTouchEvent

ViewGroup:

dispatchTouchEvent -> onInterceptTouchEvent -> onTouch -> onTouchEvent

View: dispatchTouchEvent -> onTouch -> onTouchEvent

1. ViewGroup dispatchTouchEvent返回true，它会影响Activity的

dispatchTouchEvent方法直接返回true，不继续调用onTouchEvent

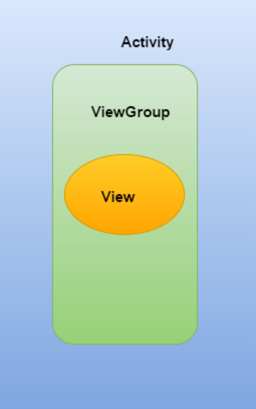
1. View的dispatchTouchEvent返回true，它会影响ViewGroup的

dispatchTouchEvent直接返回true，不在调用自己的dispatchTouchEvent方法。

# 事件传递

事件在哪些对象之间进行传递？

Activity、ViewGroup、View



# 事件分发的顺序

即 事件传递的顺序：Activity -> ViewGroup -> View

1个点击事件发生后，事件先传到Activity、再传到ViewGroup、最终再传到 View

# 事件分发过程由哪些方法协作完成

dispatchTouchEvent() 、onInterceptTouchEvent()和onTouchEvent()

从上可知，要想充分理解Android分发机制，本质上是要理解：

Activity对点击事件的分发机制

ViewGroup对点击事件的分发机制

View对点击事件的分发机制

下面，我将通过源码，全面解析 事件分发机制

即按顺序讲解：Activity事件分发机制、ViewGroup事件分发机制、View事件分发机制

# Activity对点击事件的分发机制

当一个点击事件发生时，事件最先传到Activity的dispatchTouchEvent()进行事件分发

*/\*\**

*\* 源码分析：Activity.dispatchTouchEvent（）  
 \*/*public boolean dispatchTouchEvent(MotionEvent ev) {  
 // 一般事件列开始都是DOWN事件 = 按下事件，故此处基本是true  
 if (ev.getAction() == MotionEvent.*ACTION\_DOWN*) {  
 onUserInteraction();  
 }  
 if (getWindow().superDispatchTouchEvent(ev)) {  
 return true;  
 // 若getWindow().superDispatchTouchEvent(ev)的返回true  
 // 则Activity.dispatchTouchEvent（）就返回true，则方法结束。即 ：该点击事件停止往下传递 & 事件传递过程结束  
 // 否则：继续往下调用Activity.onTouchEvent  
 }  
 return onTouchEvent(ev);  
}  
  
*/\*\*  
 \* 分析1：onUserInteraction()  
 \* 作用：实现屏保功能  
 \* a. 该方法为空方法  
 \* b. 当此activity在栈顶时，触屏点击按home，back，menu键等都会触发此方法  
 \*/*public void onUserInteraction() {  
  
}  
  
*/\*\*  
 \* 分析2：getWindow().superDispatchTouchEvent(ev)  
 \* a. getWindow() = 获取Window类的对象  
 \* b. Window类是抽象类，其唯一实现类 = PhoneWindow类；即此处的Window类对象 = PhoneWindow类对象  
 \* c. Window类superDispatchTouchEvent() = 1个抽象方法，由子类PhoneWindow实现  
 \*/*@Override  
public boolean superDispatchTouchEvent(MotionEvent event) {  
  
 return mDecor.superDispatchTouchEvent(event);  
 // mDecor = 顶层View（DecorView）的实例对象  
}  
  
*/\*\*  
 \* 分析3：mDecor.superDispatchTouchEvent(event)  
 \* 定义：属于顶层View（DecorView）  
 \* 说明：  
 \* a. DecorView类是PhoneWindow类的一个内部类  
 \* b. DecorView继承自FrameLayout，是所有界面的父类  
 \* c. FrameLayout是ViewGroup的子类，故DecorView的间接父类 = ViewGroup  
 \*/*public boolean superDispatchTouchEvent(MotionEvent event) {  
 return super.dispatchTouchEvent(event);  
 // 调用父类的方法 = ViewGroup的dispatchTouchEvent()  
 // 即 将事件传递到ViewGroup去处理，详细请看ViewGroup的事件分发机制  
}  
  
*/\*\*  
 \* 分析4：Activity.onTouchEvent（）  
 \* 定义：属于顶层View（DecorView）  
 \* a. DecorView类是PhoneWindow类的一个内部类  
 \* b. DecorView继承自FrameLayout，是所有界面的父类  
 \* c. FrameLayout是ViewGroup的子类，故DecorView的间接父类 = ViewGroup  
 \*/*public boolean onTouchEvent(MotionEvent event) {  
 // 当一个点击事件未被Activity下任何一个View接收 / 处理时  
 // 应用场景：处理发生在Window边界外的触摸事件  
 if (mWindow.shouldCloseOnTouch(this, event)) {  
 finish();  
 return true;  
 }  
  
 return false;  
 // 即 只有在点击事件在Window边界外才会返回true，一般情况都返回false，分析完毕  
}  
  
*/\*\*  
 \* 分析5：mWindow.shouldCloseOnTouch(this, event)  
 \*/*public boolean shouldCloseOnTouch(Context context, MotionEvent event) {  
 // 主要是对于处理边界外点击事件的判断：是否是DOWN事件，event的坐标是否在边界内等  
 if (mCloseOnTouchOutside && event.getAction() == MotionEvent.*ACTION\_DOWN* && isOutOfBounds(context, event) && peekDecorView() != null) {  
 return true;  
 }  
 return false;  
 // 返回true：说明事件在边界外，即 消费事件  
 // 返回false：未消费（默认）  
 return false;  
}

# ViewGroup对点击事件的分发机制

从上面Activity事件分发机制可知，ViewGroup事件分发机制从dispatchTouchEvent()开始

*/\*\**

*\* 源码分析：ViewGroup.dispatchTouchEvent（）  
 \*/*public boolean dispatchTouchEvent(MotionEvent ev) {   
... // 仅贴出关键代码  
 // 重点分析1：ViewGroup每次事件分发时，都需调用onInterceptTouchEvent()询问是否拦截事件  
 if (disallowIntercept || !onInterceptTouchEvent(ev)) {  
 // 判断值1：disallowIntercept = 是否禁用事件拦截的功能(默认是false)，可通过调用requestDisallowInterceptTouchEvent（）修改  
 // 判断值2： !onInterceptTouchEvent(ev) = 对onInterceptTouchEvent()返回值取反  
 // a. 若在onInterceptTouchEvent()中返回false（即不拦截事件），就会让第二个值为true，从而进入到条件判断的内部  
 // b. 若在onInterceptTouchEvent()中返回true（即拦截事件），就会让第二个值为false，从而跳出了这个条件判断  
 // c. 关于onInterceptTouchEvent() ->>分析1  
 ev.setAction(MotionEvent.*ACTION\_DOWN*);  
 final int scrolledXInt = (int) scrolledXFloat;  
 final int scrolledYInt = (int) scrolledYFloat;  
 final View[] children = mChildren;  
 final int count = mChildrenCount;  
  
 // 重点分析2  
 // 通过for循环，遍历了当前ViewGroup下的所有子View  
 for (int i = count - 1; i >= 0; i--) {  
 final View child = children[i];  
 if ((child.mViewFlags & VISIBILITY\_MASK) == VISIBLE  
 || child.getAnimation() != null) {  
 child.getHitRect(frame);  
  
 // 判断当前遍历的View是不是正在点击的View，从而找到当前被点击的View  
 // 若是，则进入条件判断内部  
 if (frame.contains(scrolledXInt, scrolledYInt)) {  
 final float xc = scrolledXFloat - child.mLeft;  
 final float yc = scrolledYFloat - child.mTop;  
 ev.setLocation(xc, yc);  
 child.mPrivateFlags &= ~CANCEL\_NEXT\_UP\_EVENT;  
  
 // 条件判断的内部调用了该View的dispatchTouchEvent()  
 // 即 实现了点击事件从ViewGroup到子View的传递（具体请看下面的View事件分发机制）  
 if (child.dispatchTouchEvent(ev)) {  
  
 mMotionTarget = child;  
 return true;  
 // 调用子View的dispatchTouchEvent后是有返回值的  
 // 若该控件可点击，那么点击时，dispatchTouchEvent的返回值必定是true，因此会导致条件判断成立  
 // 于是给ViewGroup的dispatchTouchEvent（）直接返回了true，即直接跳出  
 // 即把ViewGroup的点击事件拦截掉  
 }  
 }  
 }  
 }  
 boolean isUpOrCancel = (action == MotionEvent.*ACTION\_UP*) || (action == MotionEvent.*ACTION\_CANCEL*);  
 if (isUpOrCancel) {  
 mGroupFlags &= ~FLAG\_DISALLOW\_INTERCEPT;  
 }  
 final View target = mMotionTarget;  
 // 重点分析3  
 // 若点击的是空白处（即无任何View接收事件） / 拦截事件（手动复写onInterceptTouchEvent（），从而让其返回true）  
 if (target == null) {  
 ev.setLocation(xf, yf);  
 if ((mPrivateFlags & CANCEL\_NEXT\_UP\_EVENT) != 0) {  
 ev.setAction(MotionEvent.*ACTION\_CANCEL*);  
 mPrivateFlags &= ~CANCEL\_NEXT\_UP\_EVENT;  
 }  
 return super.dispatchTouchEvent(ev);  
 // 调用ViewGroup父类的dispatchTouchEvent()，即View.dispatchTouchEvent()  
 // 因此会执行ViewGroup的onTouch() ->> onTouchEvent() ->> performClick（） ->> onClick()，即自己处理该事件，事件不会往下传递（具体请参考View事件的分发机制中的View.dispatchTouchEvent（））  
 // 此处需与上面区别：子View的dispatchTouchEvent（）  
 }  
 ...  
 }  
}  
*/\*\*  
\* 分析1：ViewGroup.onInterceptTouchEvent()  
\* 作用：是否拦截事件  
\* a. 返回true = 拦截，即事件停止往下传递（需手动设置，即复写onInterceptTouchEvent（），从而让其返回true）  
\* b. 返回false = 不拦截（默认）  
\*/*public boolean onInterceptTouchEvent(MotionEvent ev) {  
 return false;  
}  
// 回到调用原处

# View对点击事件的分发机制

*/\*\**

*\* 源码分析：View.dispatchTouchEvent（）  
 \*/*public boolean dispatchTouchEvent(MotionEvent event) {  
 if (mOnTouchListener != null && (mViewFlags & ENABLED\_MASK) == ENABLED &&  
 mOnTouchListener.onTouch(this, event)) {  
 return true;  
 }  
 return onTouchEvent(event);  
}  
// 说明：只有以下3个条件都为真，dispatchTouchEvent()才返回true；否则执行onTouchEvent()  
// 1. mOnTouchListener != null  
// 2. (mViewFlags & ENABLED\_MASK) == ENABLED  
// 3. mOnTouchListener.onTouch(this, event)  
// 下面对这3个条件逐个分析  
  
  
*/\*\*  
 \* 条件1：mOnTouchListener != null  
 \* 说明：mOnTouchListener变量在View.setOnTouchListener（）方法里赋值  
 \*/*public void setOnTouchListener(OnTouchListener l) {  
 mOnTouchListener = l;  
 // 即只要我们给控件注册了Touch事件，mOnTouchListener就一定被赋值（不为空）  
}  
  
*/\*\*  
 \* 条件2：(mViewFlags & ENABLED\_MASK) == ENABLED  
 \* 说明：  
 \* a. 该条件是判断当前点击的控件是否enable  
 \* b. 由于很多View默认enable，故该条件恒定为true  
 \*/  
  
/\*\*  
 \* 条件3：mOnTouchListener.onTouch(this, event)  
 \* 说明：即 回调控件注册Touch事件时的onTouch（）；需手动复写设置，具体如下（以按钮Button为例）  
 \*/*button.setOnTouchListener(new OnTouchListener() {  
 @Override  
 public boolean onTouch(View v, MotionEvent event) {  
  
 return false;  
 }  
});  
// 若在onTouch（）返回true，就会让上述三个条件全部成立，从而使得View.dispatchTouchEvent（）直接返回true，事件分发结束  
// 若在onTouch（）返回false，就会使得上述三个条件不全部成立，从而使得View.dispatchTouchEvent（）中跳出If，执行onTouchEvent(event)  
//接下来，我们继续看：onTouchEvent(event)的源码分析  
//详情请看注释  
//Android 5.0后 View.onTouchEvent()源码发生了变化（更加复杂），但原理相同；  
//本文为了让读者更好理解，所以采用Android 5.0前的版本  
*/\*\*  
 \* 源码分析：View.onTouchEvent（）  
 \*/*public boolean onTouchEvent(MotionEvent event) {  
 final int viewFlags = mViewFlags;  
  
 if ((viewFlags & ENABLED\_MASK) == DISABLED) {  
 return (((viewFlags & CLICKABLE) == CLICKABLE ||  
 (viewFlags & LONG\_CLICKABLE) == LONG\_CLICKABLE));  
 }  
 if (mTouchDelegate != null) {  
 if (mTouchDelegate.onTouchEvent(event)) {  
 return true;  
 }  
 }  
  
 // 若该控件可点击，则进入switch判断中  
 if (((viewFlags & CLICKABLE) == CLICKABLE || (viewFlags & LONG\_CLICKABLE) == LONG\_CLICKABLE)) {  
 switch (event.getAction()) {  
 // a. 若当前的事件 = 抬起View（主要分析）  
 case MotionEvent.ACTION\_UP:  
 boolean prepressed = (mPrivateFlags & PREPRESSED) != 0;  
 ...// 经过种种判断，此处省略  
 // 执行performClick() ->>分析1  
 performClick();  
 break;  
  
 // b. 若当前的事件 = 按下View  
 case MotionEvent.ACTION\_DOWN:  
 if (mPendingCheckForTap == null) {  
 mPendingCheckForTap = new CheckForTap();  
 }  
 mPrivateFlags |= PREPRESSED;  
 mHasPerformedLongPress = false;  
 postDelayed(mPendingCheckForTap, ViewConfiguration.getTapTimeout());  
 break;  
  
 // c. 若当前的事件 = 结束事件（非人为原因）  
 case MotionEvent.ACTION\_CANCEL:  
 mPrivateFlags &= ~PRESSED;  
 refreshDrawableState();  
 removeTapCallback();  
 break;  
  
 // d. 若当前的事件 = 滑动View  
 case MotionEvent.ACTION\_MOVE:  
 final int x = (int) event.getX();  
 final int y = (int) event.getY();  
  
 int slop = mTouchSlop;  
 if ((x < 0 - slop) || (x >= getWidth() + slop) || (y < 0 - slop) || (y >= getHeight() + slop)) {  
 // Outside button   
 removeTapCallback();  
 if ((mPrivateFlags & PRESSED) != 0) {  
 // Remove any future long press/tap checks   
 removeLongPressCallback();  
 // Need to switch from pressed to not pressed   
 mPrivateFlags &= ~PRESSED;  
 refreshDrawableState();  
 }  
 }  
 break;  
 }  
 // 若该控件可点击，就一定返回true  
 return true;  
 }  
 // 若该控件不可点击，就一定返回false  
 return false;  
 }  
  
*/\*\*  
 \* 分析1：performClick（）  
 \*/*public boolean performClick() {  
 if (mOnClickListener != null) {  
 playSoundEffect(SoundEffectConstants.CLICK);  
 mOnClickListener.onClick(this);  
 return true;  
 // 只要我们通过setOnClickListener（）为控件View注册1个点击事件  
 // 那么就会给mOnClickListener变量赋值（即不为空）  
 // 则会往下回调onClick（） & performClick（）返回true  
 }  
 return false;  
}

# 目录1

## 目录2

### 目录3

#### 目录4

##### 目录5