# 小结

## 1.1.1 小结

跟touch事件相关的3个方法：

public boolean dispatchTouchEvent(MotionEvent ev);    //用来分派event

public boolean onInterceptTouchEvent(MotionEvent ev); //用来拦截event

public boolean onTouchEvent(MotionEvent ev);          //用来处理event

dispatchTouchEvent() 用来分派事件。

其中调用了onInterceptTouchEvent()和onTouchEvent()，一般不重写该方法

onInterceptTouchEvent() 用来拦截事件。

ViewGroup类中的源码实现就是{return false;}表示不拦截该事件，事件将向下传递（传递给其子View）；

若手动重写该方法，使其返回true则表示拦截，事件将终止向下传递，事件由当前ViewGroup类来处理，就是调用该类的onTouchEvent()方法。

onTouchEvent() 用来处理事件。

返回true则表示该View能处理该事件，事件将终止向上传递（传递给其父View）；

返回false表示不能处理，则把事件传递给其父View的onTouchEvent()方法来处理。

## 1.1.2 为什么要有事件分发机制？

安卓上面的View是树形结构的，View可能会重叠在一起，当我们点击的地方有多个View都可以响应的时候，这个点击事件应该给谁呢？为了解决这一个问题，就有了事件分发机制。

Activity 作为原始的事件分发者，如果 Activity 拦截了事件会导致整个屏幕都无法响应事件，这肯定不是我们想要的效果。所以，Activity没有onInterceptTouchEvent()。

View最为事件传递的最末端，要么消费掉事件，要么不处理进行回传，根本没必要进行事件拦截。所以，View也没有onInterceptTouchEvent()。

前面我们了解到了我们的View是树形结构的，基于这样的结构，我们的事件可以进行有序的分发。

事件收集之后最先传递给 Activity， 然后依次向下传递，大致如下：

Activity －> PhoneWindow －> DecorView －> ViewGroup －> ... －> View

这样的事件分发机制逻辑非常清晰，可是，你是否注意到一个问题？如果最后分发到View，如果这个View也没有处理事件怎么办，就这样让事件浪费掉？

当然不会啦，如果没有任何View消费掉事件，那么这个事件会按照反方向回传，最终传回给Activity，如果最后 Activity 也没有处理，本次事件才会被抛弃:

Activity <－ PhoneWindow <－ DecorView <－ ViewGroup <－ ... <－ View

## 1.1.3 View的事件调度顺序

dispatchTouchEvent 是事件分发机制中的核心，所有的事件调度都归它管。不过我细看表格， ViewGroup 有 dispatchTouchEvent 也就算了，毕竟人家有一堆 ChildView 需要管理，但为啥 View 也有？这就引出了我们的第一个疑问。

Q: 为什么 View 会有 dispatchTouchEvent ?

A: 我们知道 View 可以注册很多事件监听器，例如：单击事件(onClick)、长按事件(onLongClick)、触摸事件(onTouch)，并且View自身也有 onTouchEvent 方法，那么问题来了，这么多与事件相关的方法应该由谁管理？毋庸置疑就是 dispatchTouchEvent，所以 View 也会有事件分发。

相信看到这里很多小伙伴会产生第二个疑问，View 有这么多事件监听器，到底哪个先执行？

Q: 与 View 事件相关的各个方法调用顺序是怎样的？

A: 如果不去看源码，想一下让自己设计会怎样？

单击事件(onClickListener) 需要两个两个事件(ACTION\_DOWN 和 ACTION\_UP )才能触发，如果先分配给onClick判断，等它判断完，用户手指已经离开屏幕，黄花菜都凉了，定然造成 View 无法响应其他事件，应该最后调用。(最后)

长按事件(onLongClickListener) 同理，也是需要长时间等待才能出结果，肯定不能排到前面，但因为不需要ACTION\_UP，应该排在 onClick 前面。(onLongClickListener > onClickListener)

触摸事件(onTouchListener) 如果用户注册了触摸事件，说明用户要自己处理触摸事件了，这个应该排在最前面。(最前)

View自身处理(onTouchEvent) 提供了一种默认的处理方式，如果用户已经处理好了，也就不需要了，所以应该排在 onTouchListener 后面。(onTouchListener > onTouchEvent)

所以事件的调度顺序应该是 onTouchListener > onTouchEvent > onLongClickListener > onClickListener。

## 1.1.4小结

事件分发原理: 责任链模式，事件层层传递，直到被消费。

View 的 dispatchTouchEvent 主要用于调度自身的监听器和 onTouchEvent。

View的事件的调度顺序是 onTouchListener > onTouchEvent > onLongClickListener > onClickListener 。

不论 View 自身是否注册点击事件，只要 View 是可点击的就会消费事件。

事件是否被消费由返回值决定，true 表示消费，false 表示不消费，与是否使用了事件无关。

ViewGroup 中可能有多个 ChildView 时，将事件分配给包含点击位置的 ChildView。

ViewGroup 和 ChildView 同时注册了事件监听器(onClick等)，由 ChildView 消费。

一次触摸流程中产生事件应被同一 View 消费，全部接收或者全部拒绝。

只要接受 ACTION\_DOWN 就意味着接受所有的事件，拒绝 ACTION\_DOWN 则不会收到后续内容。

如果当前正在处理的事件被上层 View 拦截，会收到一个 ACTION\_CANCEL，后续事件不会再传递过来。

# 1.2 TouchEvent

（1）事件分发的对象是谁？就是点击事件（Touch事件）。

当用户触摸屏幕时（View 或 ViewGroup派生的控件），将产生点击事件（Touch事件）。

Touch事件的相关细节（发生触摸的位置、时间等）被封装成MotionEvent对象。

（2）事件类型（4种）

事件类型 具体动作

MotionEvent.ACTION\_DOWN 按下View（所有事件的开始）

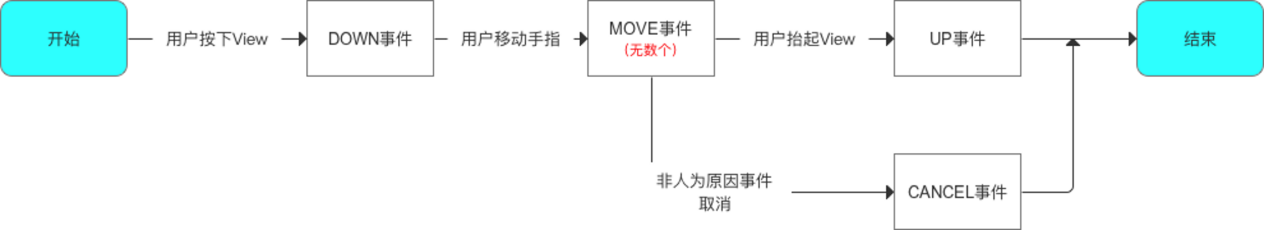
MotionEvent.ACTION\_UP 抬起View（与DOWN对应）

MotionEvent.ACTION\_MOVE 滑动View

MotionEvent.ACTION\_CANCEL 结束事件（非人为原因）

（3）事件列：从手指接触屏幕 至 手指离开屏幕，这个过程产生的一系列事件。

注：一般情况下，事件列都是以DOWN事件开始、UP事件结束，中间有无数的MOVE事件，如下图：



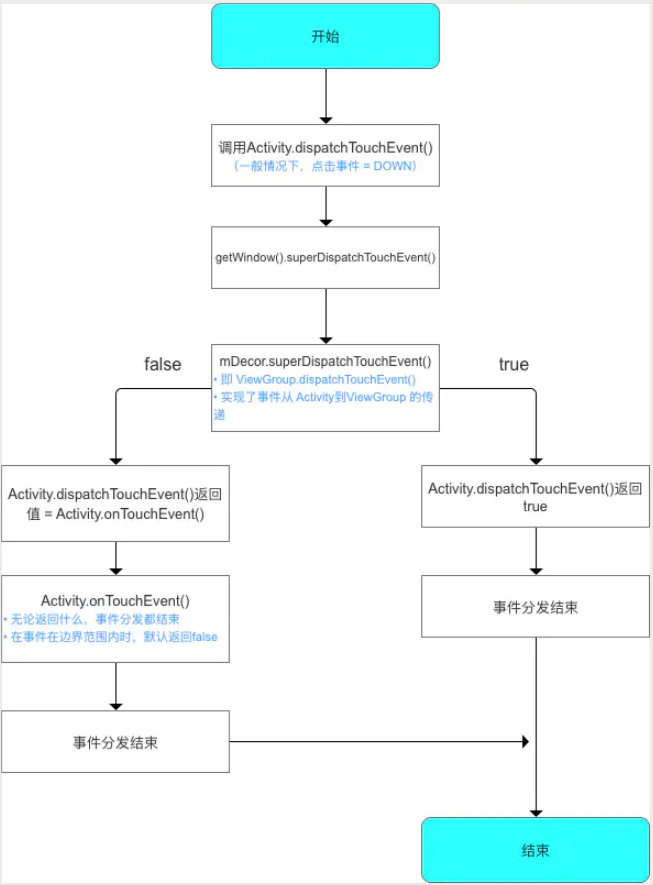
（4）事件在哪些对象之间进行传递？

Activity、ViewGroup、View及派生类之间传递。

# 1.3 Activity的事件分发

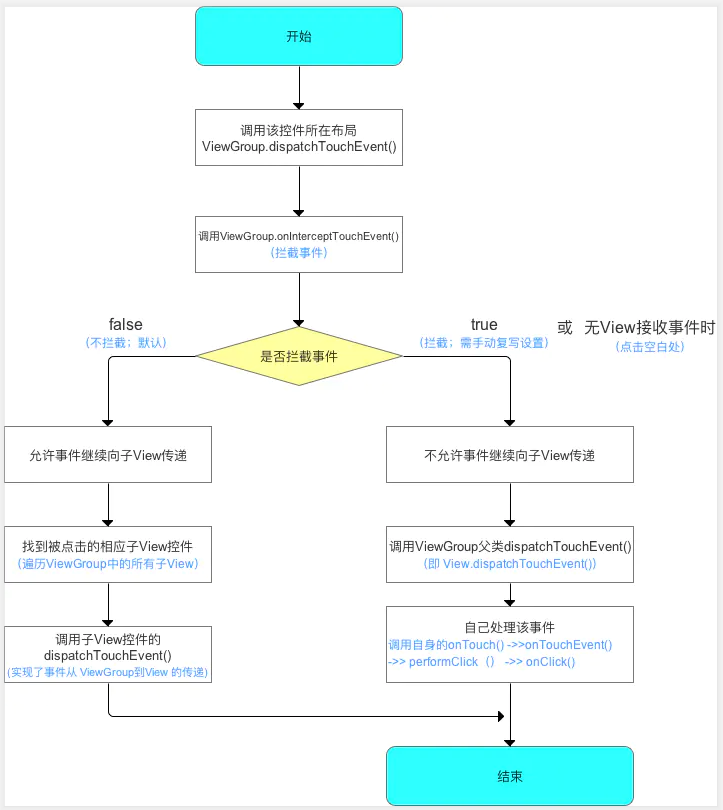
当一个点击事件发生时，事件最先传到Activity的dispatchTouchEvent()进行事件分发。

|  |
| --- |
| /\*\*  \* 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();  // ->>分析1  }  // ->>分析2  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  }  // ->>分析4  return onTouchEvent(ev);  }  /\*\*  \* 分析1：onUserInteraction()  \* 作用：实现屏保功能  \* 注：  \* 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() {  }  // 回到最初的调用原处  /\*\*  \* 分析2：getWindow().superDispatchTouchEvent(ev)  \* 说明：  \* 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类的superDispatchTouchEvent() = 1个抽象方法，由子类PhoneWindow类实现  \*/  @Override  public boolean superDispatchTouchEvent(MotionEvent event) {  return mDecor.superDispatchTouchEvent(event);  // mDecor = 顶层View（DecorView）的实例对象  // ->> 分析3  }  /\*\*  \* 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  }  // 回到最初的调用原处  /\*\*  \* 分析4：Activity.onTouchEvent（）  \* 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 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  // ->> 分析5  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）  }  // 回到分析4调用原处 |



# 1.4 ViewGroup事件的分发

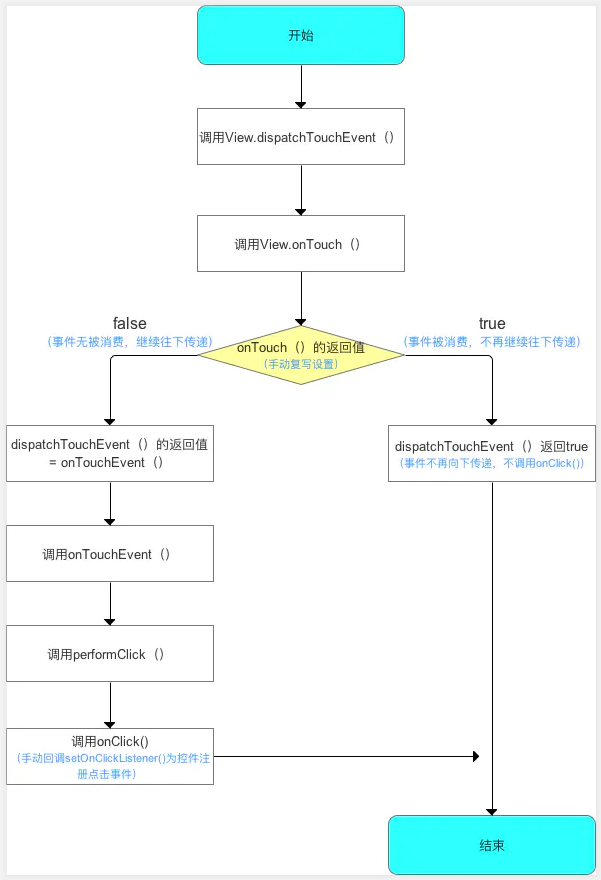
|  |
| --- |
| /\*\*  \* 源码分析：ViewGroup.dispatchTouchEvent（）  \*/  public boolean dispatchTouchEvent(MotionEvent ev) {  ... // 仅贴出关键代码  // 重点分析1：ViewGroup每次事件分发时，都需调用onInterceptTouchEvent()询问是否拦截事件  if (disallowIntercept || !onInterceptTouchEvent(ev)) {  // 判断值1：disallowIntercept = 是否禁用事件拦截的功能(默认是false)，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  \* 说明：  \* 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 |



# 1.5 View的 事件分发

从上面ViewGroup事件分发机制知道，View事件分发机制从dispatchTouchEvent()开始。

|  |
| --- |
| /\*\*  \* 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 |
| /\*\*  \* 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;  } |



# 1.6 Touch事件的后续事件（MOVE、UP）层级传递

若给控件注册了Touch事件，每次点击都会触发一系列action事件（ACTION\_DOWN，ACTION\_MOVE，ACTION\_UP等）。

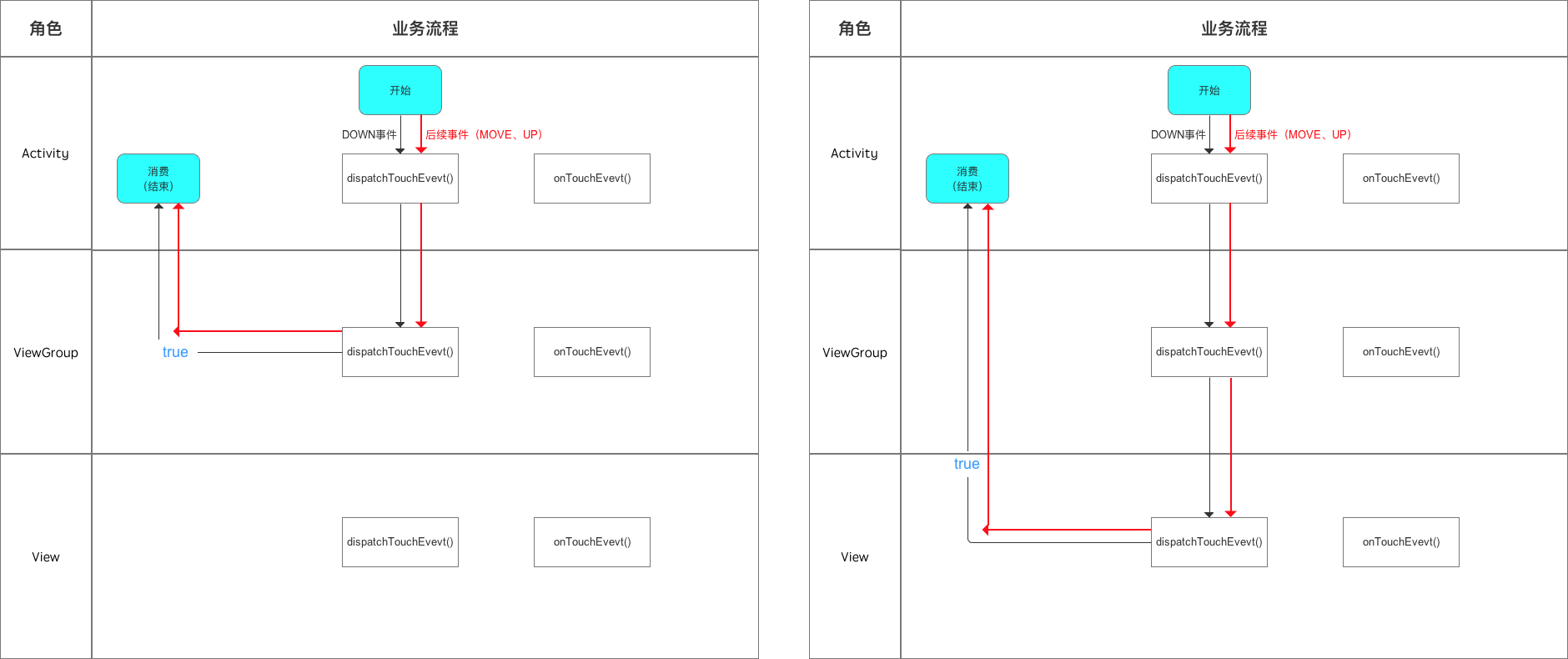
当dispatchTouchEvent（）事件分发时，只有前一个事件（如ACTION\_DOWN）返回true，才会收到后一个事件（ACTION\_MOVE和ACTION\_UP）。即如果在执行ACTION\_DOWN时返回false，后面一系列的ACTION\_MOVE、ACTION\_UP事件都不会执行。

**结论1**

若对象（Activity、ViewGroup、View）的dispatchTouchEvent()分发事件后消费了事件（返回true），那么收到ACTION\_DOWN的函数也能收到ACTION\_MOVE和ACTION\_UP。

黑线：ACTION\_DOWN事件传递方向。

红线：ACTION\_MOVE 、 ACTION\_UP事件传递方向。



**结论2**

若对象（Activity、ViewGroup、View）的onTouchEvent()处理了事件（返回true），那么ACTION\_MOVE、ACTION\_UP的事件从上往下传到该View后就不再往下传递，而是直接传给自己的onTouchEvent()& 结束本次事件传递过程。

黑线：ACTION\_DOWN事件传递方向。

红线：ACTION\_MOVE、ACTION\_UP事件传递方向。

