# Android事件传递机制

关于Android中的事件传递，在Android系统源代码层级的实现上非常的复杂，而对于应用程序的开发而言，我们倒不必要深究太多的细节，我们只需要掌握事件传递机制所带来的一些结论即可。这篇文章通过三种事件模型来聊聊关于事件传递机制的一些知识点。阅读本文需要大家静下心来耐心阅读，必定会有所收获。

在正式开始之前，大伙可以先记着几个结论，这样可以便于大家理解。

结论1：事件的一定是**先**到达父控件上

结论2：事件简单来说可以分为三种：Down事件、Move事件、Up事件，结合结论1可以得到，Down事件最先到达父控件上，Move事件也是最先到达父控件上，Up事件也是最先到达父控件上的。

结论3：**父控件**和**父类**不是一回事，这两个概念初学者很容易混淆

## 事件模型1：父控件🡪子控件

关于这种事件模型主要涉及到3个概念：

**事件的分发**

**事件的拦截**

**事件的响应**

我们先把基础知识给回顾一下，事件分发是dispatchTouchEvent，事件拦截是onInterceptTouchEvent，事件的响应是onTouchEvent，对于ViewGroup类型的控件来说，它拥有这三种方法，而对于单个View控件来说的话，它只有dispatchTouchEvent和onTouchEvent，因为View不能包含其他的View，所以不需要判断是否要拦截事件。

**一个事件如果到达了某一个View或者ViewGroup，那么一定会最先调用到这个控件的dispatchTouchEvent.**

dispatchTouchEvent这个方法就是含义就是把一个事件给分发下去，那么它具体分发的逻辑是怎样的呢？

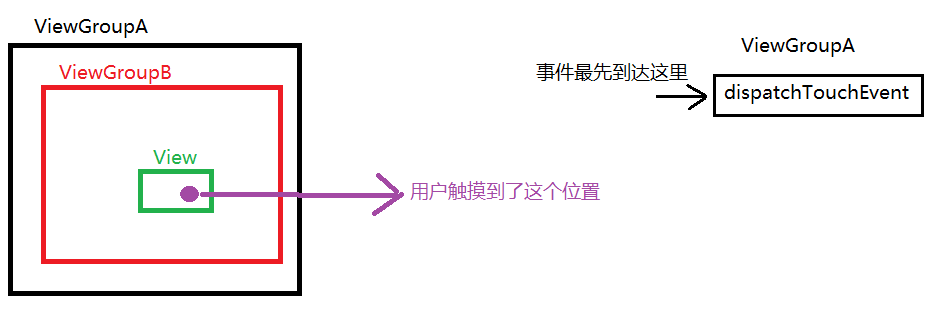


图1

1. 它首先会先调用自身的onInterceptTouchEvent方法，调用此方法的目的是为了先让自己这个控件判断下是否需要把此事件拦截下来，如果拦截下来，那么就代表自己这个控件需要来处理这个事件，所以此时会调用自身了onTouchEvent来对这个事件进行响应。

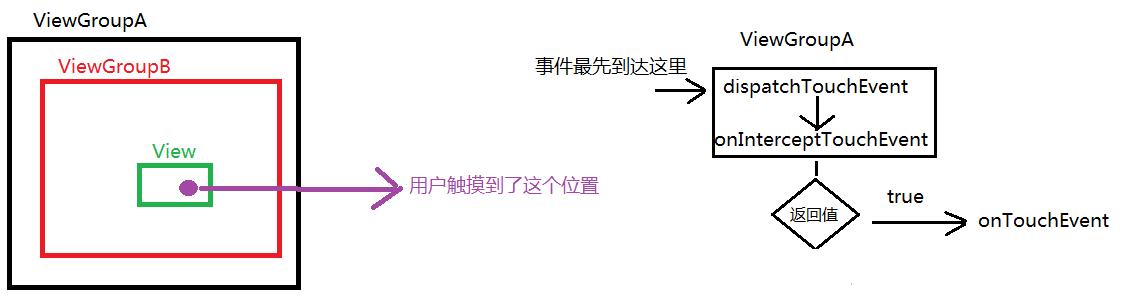


图2

1. 如果不拦截下来，那么才会有后续的事件向下传递的流程。将这个事件传递给子控件。现在子控件接收到了这个事件，按照刚刚的说法，一个事件到达了一个View或者ViewGroup，就会最先调用这个控件的dispatchTouchEvent，所以此时，事件到达了子控件的dispatchTouchEvent方法，如果这个控件仍然是一个ViewGroup的类型，那么事件继续分发的逻辑依然遵循A流程的逻辑。

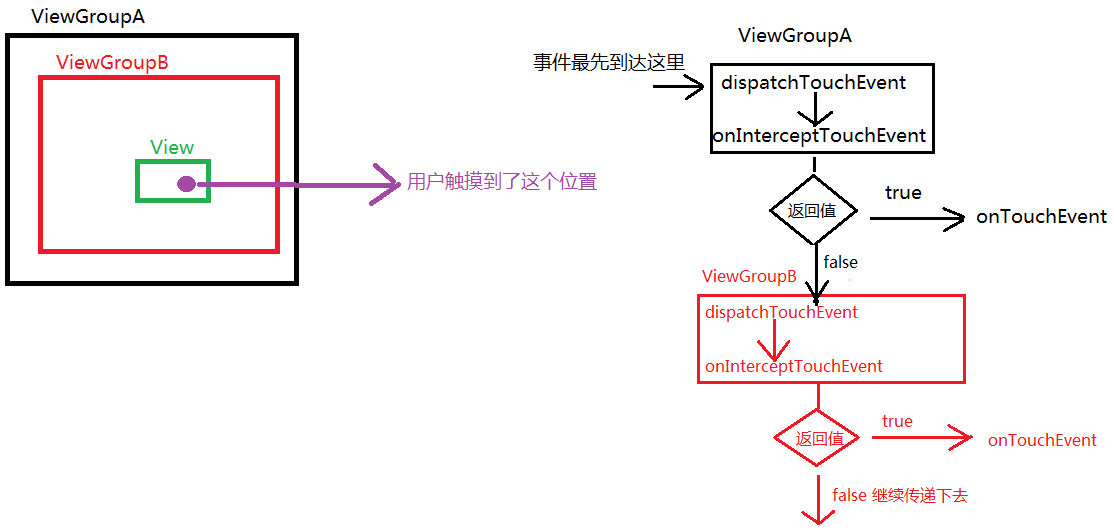


图3

1. 如果这个子控件只是一个View，而不是ViewGroup，那么此时事件分发的逻辑略有不同。由于View是没有onInterceptTouchEvent的方法，所以当一个事件到达这个View的dispatchTouchEvent的时候，这时dispatchTouchEvent就调用不到onInterceptTouchEvent了，**它会直接调用onTouchEvent的方法**，直接让这个View来响应此事件。

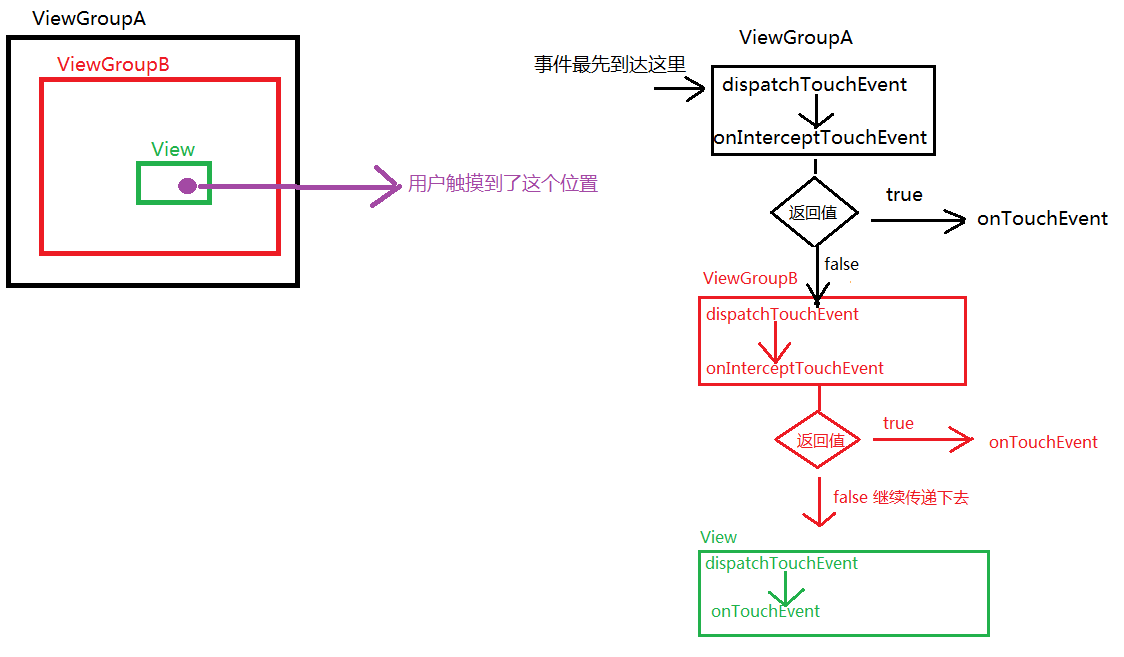


图4

通过ABC三个流程，我们就能够非常清楚的清楚事件传递的逻辑。父控件传递给子控件，当没有控件拦截事件的话，将会一层层的向下传递，直到最后一个子控件。

事件传递的流程说完之后，我们就来说一说事件响应的方向。我们知道onTouchEvent也是有返回值的，要么是true，要么是false，不同的返回值会有不同的表现。

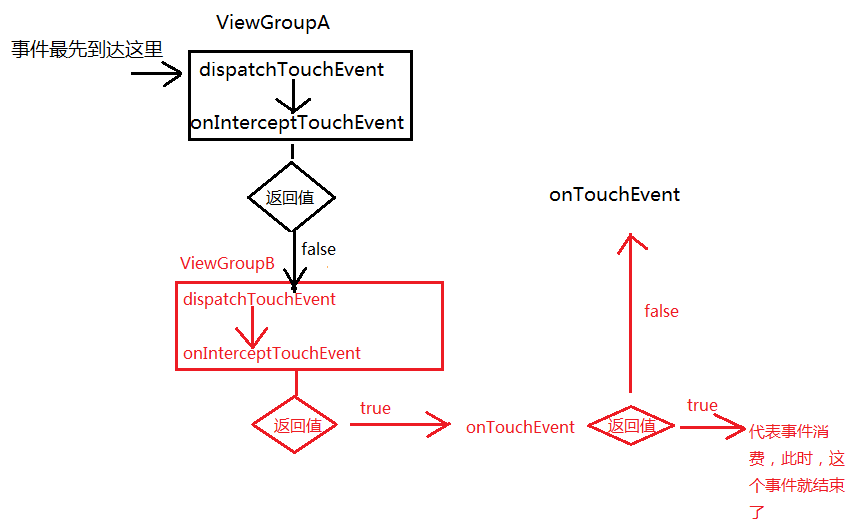


图5

如上图所示，如果ViewGroupB拦截了事件，那么此时事件就会由ViewGroupB来响应，调用ViewGroupB中的onTouchEvent，此时ViewGroupB中的onTouchEvent的返回值有两种可能，一种是true，一种是false，如果返回true，则代表，ViewGroupB消费了此事件，事件此时终止。如果返回的值是false，那么此时这个事件会回传给父控件，调用到父控件的onTouchEvent方法，由父控件来进行响应，那父控件的onTouchEvent也是同样的逻辑。要么消费事件，要么回传给父控件的父控件。

所以此时，就可以得出我们通常所说的两个方向：

1. **事件传递的方向：父控件🡪子控件**
2. **事件响应的方向：子控件🡪父控件**

更加清晰的图：

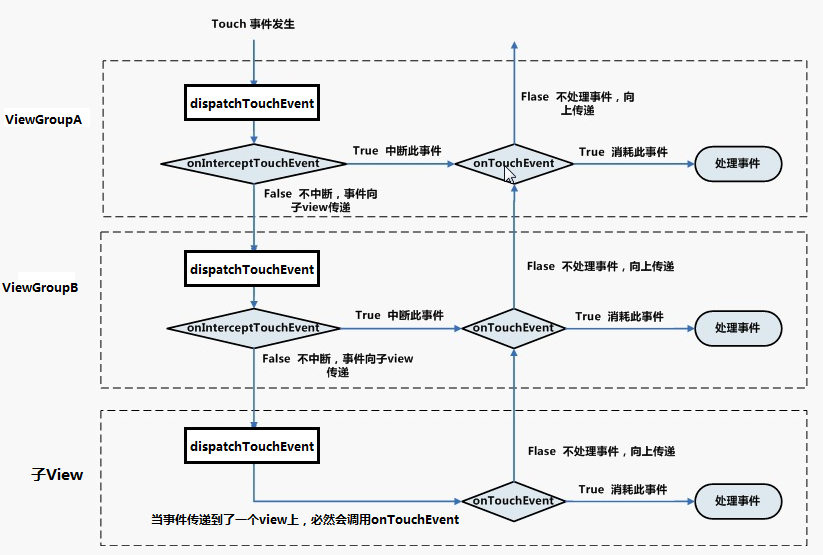


图6

当然仅仅是这个结论是无法满足我们实际开发的需要，我们需要更细致的分析。这里有一个细节上的问题需要注意，**就是事件分为Down事件、Move事件、Up事件**，任何一种事件都遵循事件传递和响应的逻辑原则，很多同学常常会认为 Down-Move-Up 这连在一起才是一个事件的产生，这种想法是非常错误的。

事件的起点是由Down事件开始的，然后产生一系列的Move事件，最后通常以Up事件结束。当Down事件产生的时候，会由父控件传递给子控件，Move事件也是由父控件传递给子控件，Up事件也是由父控件传递给子控件。他们都遵循同样的传递事件的逻辑流程。不过Down事件最终响应的结果，会影响到后续事件的执行。这句话是什么意思呢？

我们看图6，如果Down事件传递到了子View上，但是子View的onTouchEvent对于这个Down事件的处理是return了一个false，这样造成的结果就是会造成父View的onTouchEvent的调用，同时还有另外一个后果，那就是后续的Move事件、Up事件就都传递不到子View上了。所以，**如果一个View要处理滑动事件，也就是Move事件的话，那么它一定不能在onTouchEvent中，对Down事件return false.**

继续看图6，如果Down事件到了父View上，父View需要调用自身的onInterceptTouchEvent判断是否对这个Down事件进行拦截，如果拦截，return了true，那么这个事件就会到父View的onTouchEvent中进行响应。如果此时父View的onTouchEvent也返回了true，那么代表这个父View响应了Down事件。不过这里有一点不太一样的地方是在于，事件传递到父View的onTouchEvent方法是因为自身的onInterceptTouchEvent方法判断拦截导致的，而不是由子View回传回来的，在这种情况下，**当Move事件、Up事件传递到父View的时候，它当然不会传递给子View，并且，也不再调用自身的onInterceptTouchEvent方法**了，而是dispatchTouchEvent🡪onTouchEvent的传递。但是，对于ViewGroupA来说，它依然是dispatchTouch🡪onInterceptTouchEvent的流程。

理解事件传递的基本逻辑，对于工作过程中解决滑动事件冲突是非常有帮助的。比如我们此时有一个父控件ViewPager，这个ViewPager的其中一个Item是ScrollView.此时会发生什么问题呢？当ViewPager滑动到ScrollView这个条目的时候，再左右滑动，发现ViewPager再也左右滑动不了了。这是为什么呢？我们结合图6一起来分析一下。

1. 我们都知道ViewPager是能够横向滑动的控件，而ScrollView是纵向滑动的控件，当Down事件产生的时候，此时会由ViewPager传递给ScrollView，ViewPager没有对Down事件拦截，ScrollView也不会对这个Down事件进行拦截，所以事件就会传递给ScrollView的孩子，也就是类似于图6中的子View，子View如果没有对Down事件响应，那么最后会到ScrollView中的onTouchEvent，而ScrollView的onTouchEvent对于这个Down事件返回了true，代表ScrollView消费了这个Down事件。
2. 接下来开始滑动手指，产生一系列的Move事件。Move事件也是由ViewPager传递给ScrollView。由于Down事件是被ScrollView的onTouchEvent中消费的，所以Move事件就不会传递给ScrollView的子控件了。一系列的Move事件也是在ScrollView的onTouchEvent中被执行。
3. 最后的Up事件也是由ScrollView中的onTouchEvent消费

从上述的1、2、3个步骤我们看出来无论是Down事件、Move事件还是Up事件，最后全部都是被ScrollView所消费。从头到尾ViewPager的onTouchEvent都没有得到执行。而ViewPager之所以能够左右滑动，正是因为ViewPager的onTouchEvent里面的代码逻辑产生的效果。ViewPager的onTouchEvent没有执行，这个ViewPager当然就不能够左右滑动了。所以解决上述问题，就是在于如何让ViewPager中的onTouchEvent方法执行。

我们可以自定义一个MyViewPager继承ViewPager，重写onInterceptTouchEvent方法，如果我们在onInterceptTouchEvent方法中直接野蛮的return 一个true，此时就是代表无论是Down事件，还是Move事件，还是Up事件，全部都拦截下来了，拦截在MyViewPager中，我们可以认为是图6中的ViewGroupB，既然拦截下来了所有事件，那么所有事件就会传递到MyViewPager的onTouchEvent，所以此时，这个MyViewPager一定可以左右滑动。

但是，由此会引发另外一个问题，就是这个ScrollView不能上下滑动了。这又是为什么呢？因为ScrollView能够上下滑动的代码逻辑在ScrollView中的onTouchEvent方法内，而此时事件由全部被MyViewPager拦截下来了，ScrollView完全得不到事件，onTouchEvent方法得不到执行，自然不能上下滑动。所以我们需要修改MyViewPager中的onInterceptTouchEvent的逻辑。

ViewPager只对左右滑动感兴趣，而ScrollView对上下滑动这个动作感兴趣，所以我们只需要在MyViewPager的onInterceptTouchEvent中，根据多个Move事件，判断是左右滑动还是上下滑动，左右滑动的话，return true将事件拦截下来，上下滑动的话，return false将事件传递给ScrollView，这样就能解决问题了。

所以，对于Down事件，我们一般都不进行拦截，判断是否拦截得根据一些列的Move事件才能得出具体的条件是否成立。

**Cancel事件的产生：**

刚才我们说了事件一般有三个，Down、Move、Up，这三个事件比较好理解。其实还有一种事件就是Cancel事件。它代表什么含义呢？

还是回到图6，如果一个Down事件产生了，这个Down事件从ViewGroupA传递到ViewGroupB最终到达子View，被子View的onTouchEvent消费，return了true，那么此时Down事件就终止了。接下来后续的Move事件也会从ViewGroupA传递给ViewGroupB，也就是说ViewGroupA和ViewGroupB会比子View更先拿到Move事件，那既然ViewGroupA和ViewGroupB比子View更先拿到Move事件，那么他们当中的任何一个都有可能在某一个Move事件中，把这个Move事件给拦截下来，一旦Move事件被拦截下来了，子View肯定就拿不到这个Move事件了，不过，此时子View会产生一个新的事件，就是Cancel事件。

所以一个正常的事件序列是 Down🡪Move🡪Up,这样才被认为是一个正常的事件序列。如果一个View响应的Down事件，可是却被没有正常结尾，Move事件或者Up事件被拦截了，此时非正常结尾的情况就会给子View产生一个新的事件Cancel

**子控件可以影响父控件是否拦截的行为**

子控件是可以干预父控件是否拦截事件的结果的。通过在子View中dispatchTouchEvent中增加一行代码即可。getParent().requestDisallowInterceptTouchEvent(true);这行代码就可以请求父控件不要拦截事件。

很多人可能不太明白这句话的意思，既然事件一定是先到达父控件的，然后才到达子View的，那也就是getParent().requestDisallowInterceptTouchEvent(true);这句话是在父控件是否拦截判断结束之后，才调用，怎么能改变父控件是否拦截的结果呢，这里存在一个执行先后顺序的疑惑。

其实是这样的，getParent().requestDisallowInterceptTouchEvent(true);达到的效果不是修改父控件对本次事件是否拦截的结果。而影响的是后续事件。比如子View在Down事件中调用了getParent().requestDisallowInterceptTouchEvent(true);这行代码，那么在后续Move事件、Up事件产生到达父控件的时候，父控件就不会再拦截了。所以getParent().requestDisallowInterceptTouchEvent(true);只会影响Move事件和Up事件，是影响不到Down事件的。

## 事件模型2：同一个View中

在这种事件模型中，主要是要理清楚点击事件和触摸事件的逻辑关系。

我们先来回顾一下我们怎么设置点击事件和触摸事件的。点击事件是响应了onClick方法，而触摸事件是响应了onTouch方法，那他们两者是处于一种怎么样的关系呢？通过事件模型1的结论我们知道，如果事件到达了一个View上，那么最先调用这个View的dispatchTouchEvent，我们就来看看View的dispatchTouchEvent

1 public boolean dispatchTouchEvent(MotionEvent event) {

2 //...省略代码...

3 ListenerInfo li = mListenerInfo;

4 if (li != null && li.mOnTouchListener != null

5 && (mViewFlags & ENABLED\_MASK) == ENABLED

6 && li.mOnTouchListener.onTouch(this, event)) {

7 return true;

8 }

9 if (onTouchEvent(event)) {

10 return true;

11 }

12 //...省略代码

核心代码如上所示。通过上述代码第9行我们看出了在View中的dispatchTouchEvent中直接调用了onTouchEvent方法，但是注意，在onTouchEvent方法调用之前，还做了一件事情，那就是代码第4行一直到第7行的逻辑。

第4行的意思是判断mListenerInfo是否为空和mListenerInfo中的mOnTouchEventListener是否为空。只要你给这个View设置了setOnTouchListener(onTouchListener)，那么这两个值都不会为空。

第5行的意思判断一下这个View是否是Enable的状态。

第6行的意思是调用mOnTouchListener中的onTouch方法，而这个mOnTouchListener就是你给这个View设置的触摸监听对象，如果这个onTouch方法返回了true，那么所有的if条件成立，进入到第7行的代码逻辑，直接return 了true，后续的代码全部都不走了，也就是onTouchEvent方法不走了。

这里有必要理清楚两个方法，一个是onTouch方法，一个是onTouchEvent方法，很多初学者总是被这两个方法弄的头晕。onTouchEvent这个方法是这个View的方法，而onTouch方法是设置给这个View的触摸监听对象中的方法，这两个方法是完全不同的两个东西。

通过上述代码我们可以得出一个结论，触摸监听对象的onTouch方法比View的onTouchEvent执行的早，并且，如果触摸监听对象的onTouch方法的返回值为true的话，这个View的onTouchEvent将得不到执行。

而我们经常说的点击事件onClick的执行的代码逻辑其实全部都是在View的onTouchEvent中，我们可以看下View中onTouchEvent的核心代码：

1 public boolean onTouchEvent(MotionEvent event) {

2 //...省略代码

3 if (((viewFlags & CLICKABLE) == CLICKABLE ||

4 (viewFlags & LONG\_CLICKABLE) == LONG\_CLICKABLE)) {

5 switch (event.getAction()) {

6 case MotionEvent.ACTION\_UP:

7 //...省略代码

8 performClick();

9 //...省略代码

10 break;

看代码第8行，调用了一个方法performClick,而performClick的代码如下：

1 public boolean performClick() {

2 ListenerInfo li = mListenerInfo;

3 if (li != null && li.mOnClickListener != null) {

4 playSoundEffect(SoundEffectConstants.CLICK);

5 li.mOnClickListener.onClick(this);

6 return true;

7 }

8 return false;

}

看代码的第5行，这里做的事情就是调用点击监听对象中的onClick方法，所以我们经常给一个View.setOnClickListener设置点击事件的监听，监听对象的onClick方法是在这个View的Up事件到来的时候进行调用的。

所以正常的调用顺序是：

dispatchTouchEvent🡪触摸监听对象的onTouch🡪onTouchEvent🡪点击监听对象的onClick，

所以当你需要响应这个View的点击事件的时候，切记不要在这个View的触摸监听对象的onTouch方法中return true

还有一种很容易犯的错误会造成点击事件得不到响应，比如我自定义一个View叫做MyView，重写它的onTouchEvent方法，在onTouchEvent中return true，如下：

public class MyView extends View {

public MyView(Context context) {

super(context);

}

public MyView(Context context, AttributeSet attrs) {

super(context, attrs);

}

public MyView(Context context, AttributeSet attrs, int defStyle) {

super(context, attrs, defStyle);

}

@Override

public boolean onTouchEvent(MotionEvent event) {

return true;

}

}

这样子之后，你就算给这个View设置了点击事件也永远无效。为什么呢？按照刚才所说的逻辑，点击事件的响应是UP事件来到时在这个View的onTouchEvent方法中做的响应。那么这一部分的逻辑是在MyView的父类，也就是View的onTouchEvent中，**注意，此时说的是父类，不是父控件。**所以只有在MyView的onTouchEvent中调用一下super.onTouchEvent才有可能让点击事件得到响应吧！

引申一个问题，如果我在MyView的onTouchEvent中直接return false，会怎么样呢？

这个就要牵扯到事件模型1中的知识点了，如果一个View在onTouchEvent对Down事件的响应是return false，那么后续的Move、Up事件，父控件都不会再传给这个View了。

Up事件都传递不到这个View上，那这个View肯定也响应不了点击事件了。

## 事件模型3：同一层级View

这种事件模型比较简单，我们简单讲讲即可。一个父控件包含了三个子View，这三个子View有重叠的部分。如下图所示

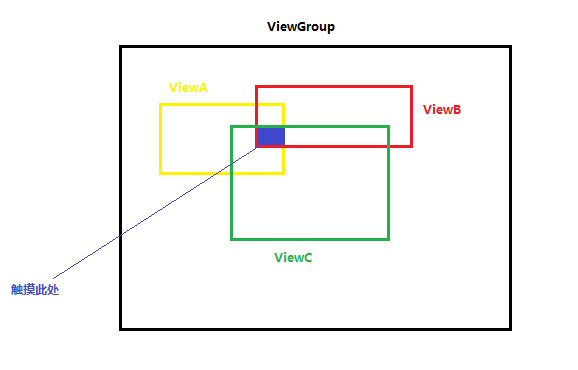


图7

如果手指触摸到蓝色重叠区域，那么ViewA和ViewB和ViewC谁先拿到事件呢？

究竟是哪个View先拿到事件，得看布局文件中摆放这三个View的顺序。

<ViewGroup>

<ViewA/>

<ViewB/>

<ViewC/>

</ViewGroup>

如果是这种布局类型的话，那么很明显ViewC会先拿到事件，如果ViewC不响应事件，ViewB才会拿到事件，如果ViewB也不响应事件，那么将是ViewA拿到事件。这个其实很好理解，ViewC在布局文件的最后面，相当于位于整个布局的最顶层，压在了ViewB和ViewA上，按照正常的思维，肯定是ViewC先拿到事件。注意，此时事件顺序ViewC—ViewB—ViewA和事件模型1没有关系，事件模型1是父控件与子控件的传递。

如果想通过源码来分析这个结论的话，其实也比较简单。当系统在加载这个布局文件的时候，其实会做类似的代码逻辑：

ViewGroup.addView(ViewA);

ViewGroup.addView(ViewB);

ViewGroup.addView(ViewC);

ViewC是最后添加的。所以当我们调用ViewGroup.getChildAt(0)的时候，此时获得到的是ViewA，ViewGroup.getChildAt(1)🡪ViewB,ViewGroup.getChildAt(2)🡪ViewC

接下来我们来看看ViewGroup是怎么分发事件的。找到ViewGroup的dispatchTouchEvent方法。如下：

@Override

1 public boolean dispatchTouchEvent(MotionEvent ev) {

2 //...省略代码

3 //先判断一下自己是否拦截事件

4 intercepted = onInterceptTouchEvent(ev);

5 //..省略代码

6 if (!canceled && !intercepted) {//自己不拦截，才把事件传递给子控件

7 //...省略代码

8 // Find a child that can receive the event.

9 // Scan children from front to back.

10 final View[] children = mChildren;

11 //...省略代码

12 for (int i = childrenCount - 1; i >= 0; i--) { //遍历循环查找能响应的子控件

13 final View child = children[childIndex];

14 if (!canViewReceivePointerEvents(child)

15 || !isTransformedTouchPointInView(x, y, child, null)) {

16 continue;

17 }

这段代码的大体意思是对事件进行分发。先判断一下自己是否需要拦截事件，如果自己不拦截事件再把事件分发给子控件。它会把自己所有的子控件照出来，做一次循环遍历。通过代码第12行到17行我们可以得出结论，查找能够响应这个事件的子控件的顺序是从后往前找，因为i是从最大值开始循环的。

这个文档主要介绍了事件的3种模型，不同模型之间设计到不同的知识点，模型与模型之间的知识点也存在很多关联性。关于这一部分的知识需要同学们细细体会。