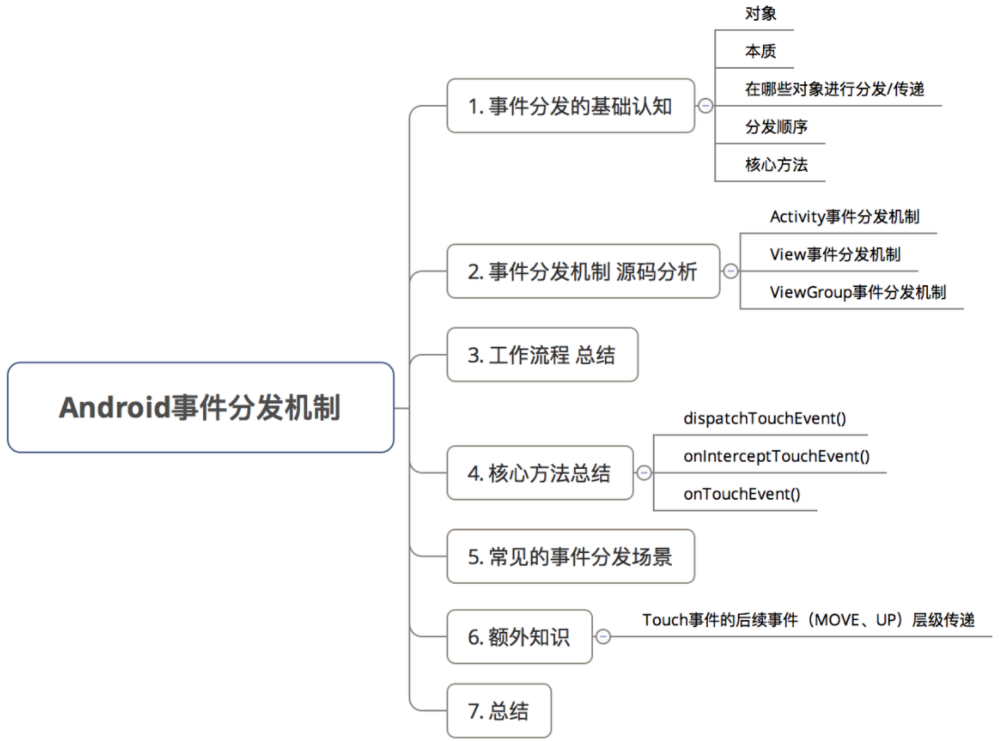
Android事件分发机制

<https://blog.csdn.net/carson_ho/article/details/54136311>



1. 基础认知

1.1 事件分发的对象是谁？

答：点击事件（Touch事件）

定义

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

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）

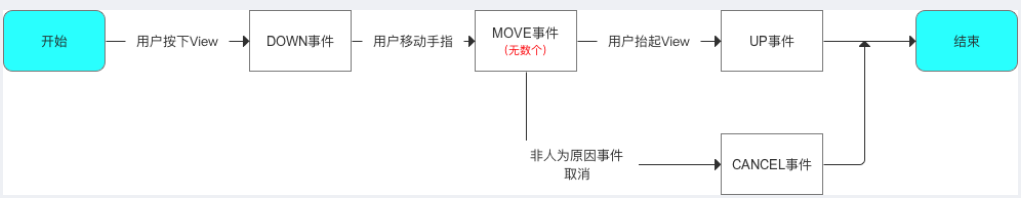
MotionEvent.ACTION\_UP 抬起View（与DOWN对应）

MotionEvent.ACTION\_MOVE 滑动View

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

特别说明：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



即当一个点击事件（MotionEvent ）产生后，系统需把这个事件传递给一个具体的 View 去处理。

1.2 事件分发的本质

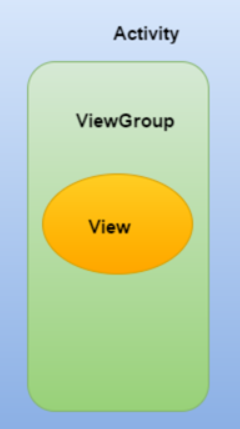
答：将点击事件（MotionEvent）传递到某个具体的View & 处理的整个过程

即 事件传递的过程 = 分发过程。

1.3 事件在哪些对象之间进行传递？

答：Activity、ViewGroup、View

Android的UI界面由Activity、ViewGroup、View 及其派生类组成

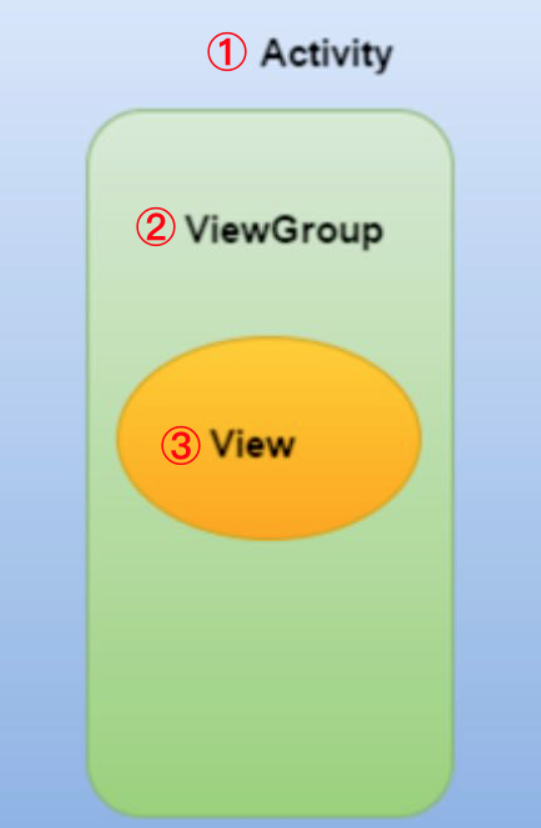




1.4 事件分发的顺序

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

即：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



1.5 事件分发过程由哪些方法协作完成？

答：dispatchTouchEvent() 、onInterceptTouchEvent()和onTouchEvent()



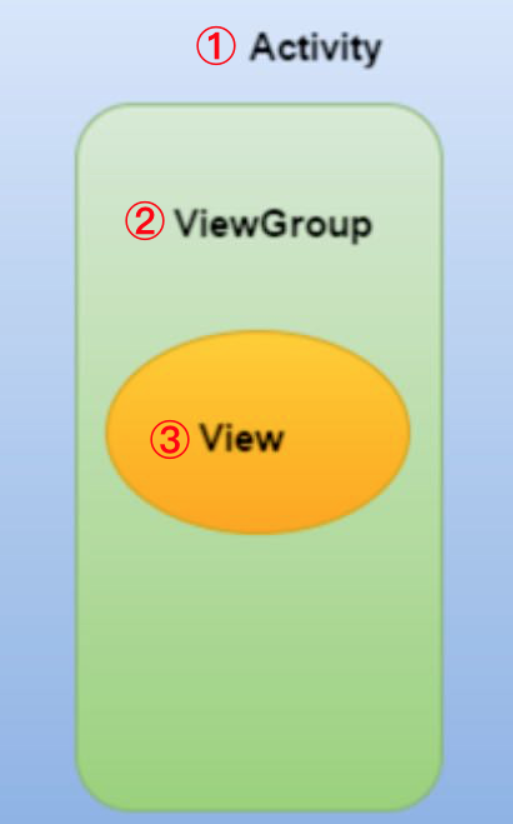
1.6 总结



2. 事件分发机制 源码分析

请谨记：Android事件分发流程 = Activity -> ViewGroup -> View

即：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



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

2.1 Activity的事件分发机制

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();

// ->>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

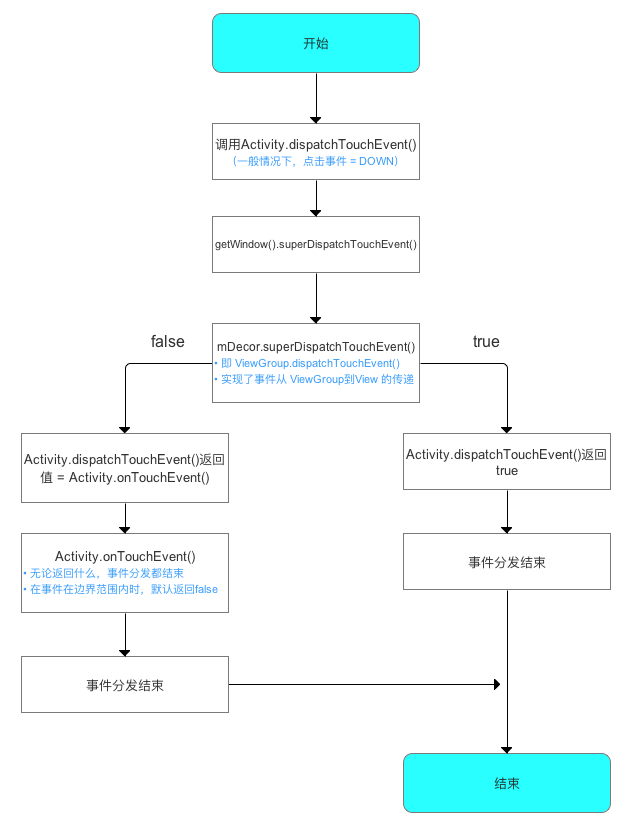
// 返回false：未消费（默认）

}

// 回到分析4调用原处

2.1.2 总结

当一个点击事件发生时，从Activity的事件分发开始（Activity.dispatchTouchEvent()）



方法总结



2.2 ViewGroup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

2.2.1 源码分析

Android 5.0后，ViewGroup.dispatchTouchEvent()的源码发生了变化（更加复杂），但原理相同；

本文为了让读者容易理解，故采用Android 5.0前的版本

/\*\*

\* 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;

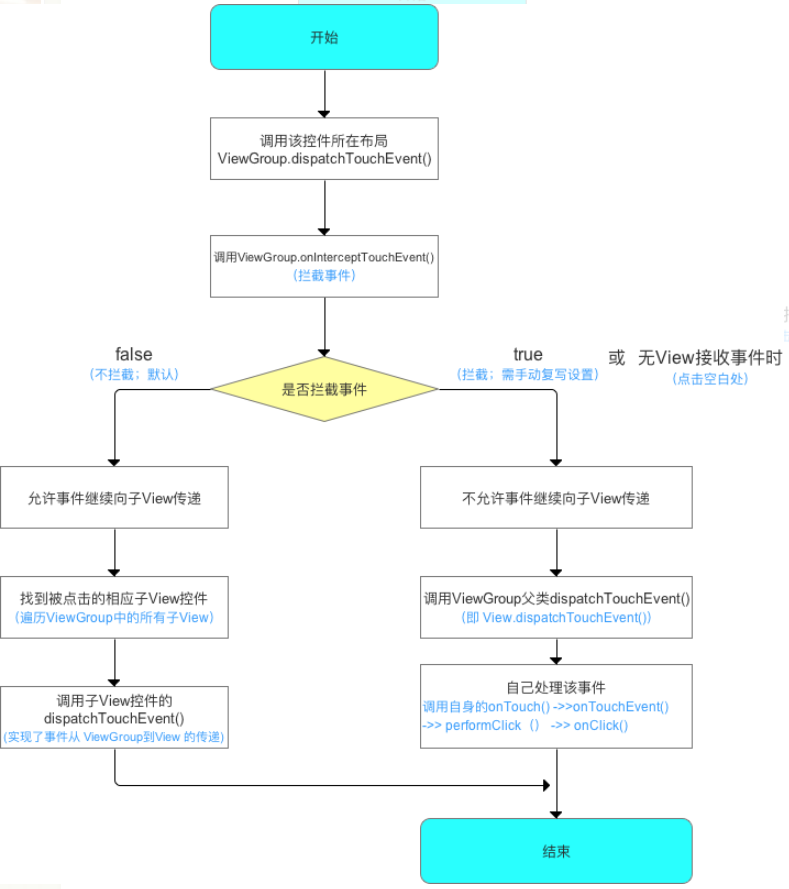
}

// 回到调用原处

2.2.2 总结

结论：Android事件分发总是先传递到ViewGroup、再传递到View

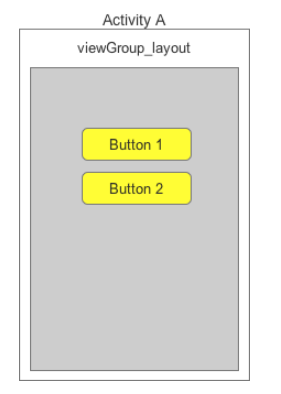
过程：当点击了某个控件时





2.2.3 Demo讲解

布局如下



测试代码

布局文件：activity\_main.xml

<?xml version="1.0" encoding="utf-8"?>

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"

android:id="@+id/my\_layout"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

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

android:focusableInTouchMode="true"

android:orientation="vertical">

<Button

android:id="@+id/button1"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="按钮1" />

<Button

android:id="@+id/button2"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="按钮2" />

</LinearLayout>

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

核心代码：MainActivity.java

/\*\*

\* ViewGroup布局（myLayout）中有2个子View = 2个按钮

\*/

public class MainActivity extends AppCompatActivity {

Button button1,button2;

ViewGroup myLayout;

@Override

public void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

button1 = (Button)findViewById(R.id.button1);

button2 = (Button)findViewById(R.id.button2);

myLayout = (LinearLayout)findViewById(R.id.my\_layout);

// 1.为ViewGroup布局设置监听事件

myLayout.setOnClickListener(new View.OnClickListener() {

@Override

public void onClick(View v) {

Log.d("TAG", "点击了ViewGroup");

}

});

// 2. 为按钮1设置监听事件

button1.setOnClickListener(new View.OnClickListener() {

@Override

public void onClick(View v) {

Log.d("TAG", "点击了button1");

}

});

// 3. 为按钮2设置监听事件

button2.setOnClickListener(new View.OnClickListener() {

@Override

public void onClick(View v) {

Log.d("TAG", "点击了button2");

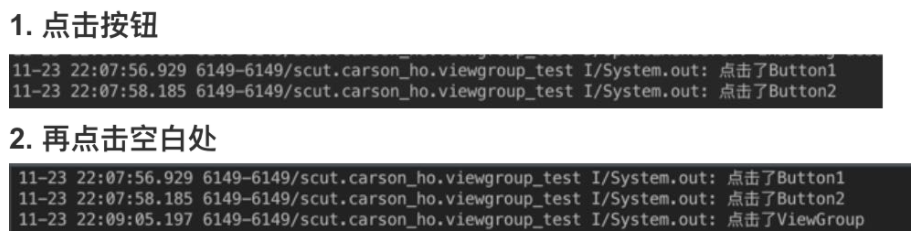
}

});

}

}

结果测试



从上面的测试结果发现：

点击Button时，执行Button.onClick()，但ViewGroupLayout注册的onTouch（）不会执行

只有点击空白区域时，才会执行ViewGroupLayout的onTouch（）

结论：Button的onClick()将事件消费掉了，因此事件不会再继续向下传递。

## 2.3 View事件的分发机制

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

2.3.1 源码分析

/\*\*

\* 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)

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

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

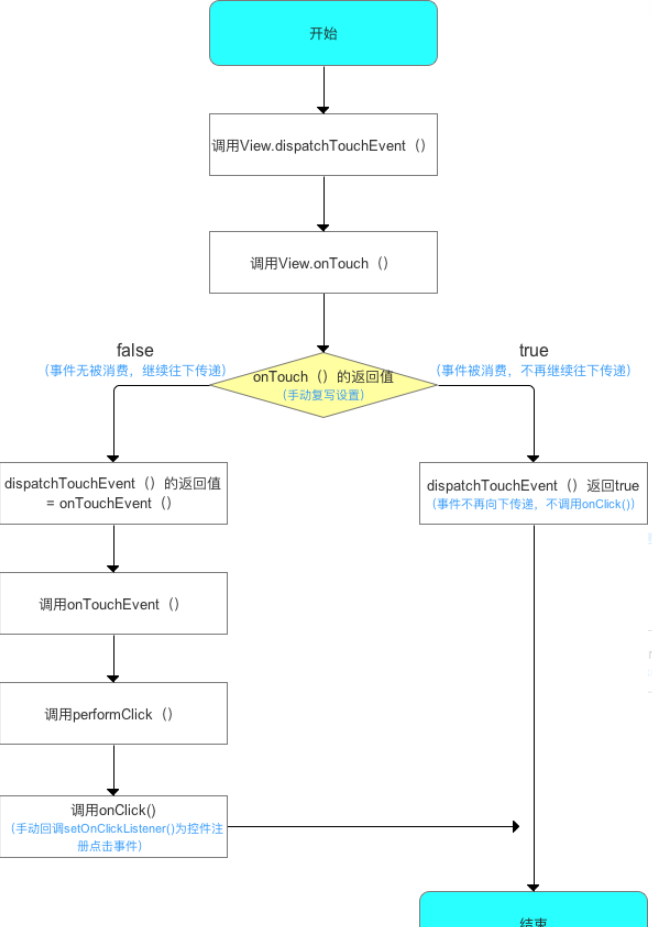
}

return false;

}

2.3.2 总结

每当控件被点击时：



核心方法总结



2.3.3 Demo讲解

下面我将用Demo验证上述的结论

/\*\*

\* 结论验证1：在回调onTouch()里返回false

\*/

// 1. 通过OnTouchListener()复写onTouch()，从而手动设置返回false

button.setOnTouchListener(new View.OnTouchListener() {

@Override

public boolean onTouch(View v, MotionEvent event) {

System.out.println("执行了onTouch(), 动作是:" + event.getAction());

return false;

}

});

// 2. 通过 OnClickListener（）为控件设置点击事件，为mOnClickListener变量赋值（即不为空），从而往下回调onClick（）

button.setOnClickListener(new View.OnClickListener() {

@Override

public void onClick(View v) {

System.out.println("执行了onClick()");

}

});

/\*\*

\* 结论验证2：在回调onTouch()里返回true

\*/

// 1. 通过OnTouchListener()复写onTouch()，从而手动设置返回true

button.setOnTouchListener(new View.OnTouchListener() {

@Override

public boolean onTouch(View v, MotionEvent event) {

System.out.println("执行了onTouch(), 动作是:" + event.getAction());

return true;

}

});

// 2. 通过 OnClickListener（）为控件设置点击事件，为mOnClickListener变量赋值（即不为空）

// 但由于dispatchTouchEvent（）返回true，即事件不再向下传递，故不调用onClick()）

button.setOnClickListener(new View.OnClickListener() {

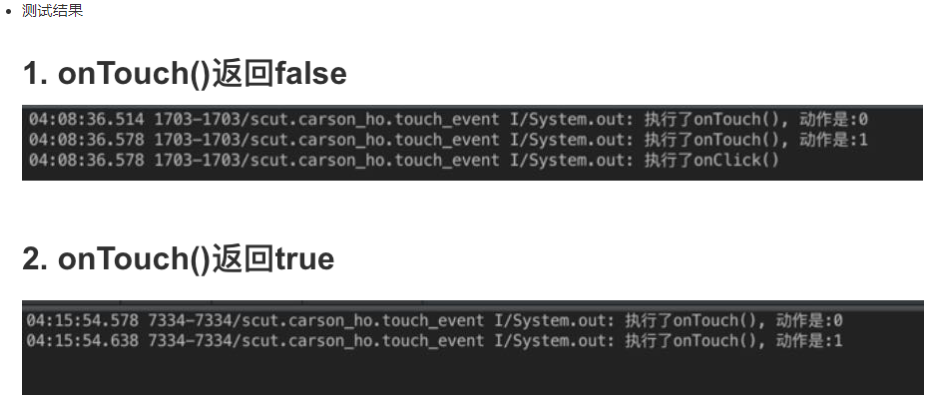
@Override

public void onClick(View v) {

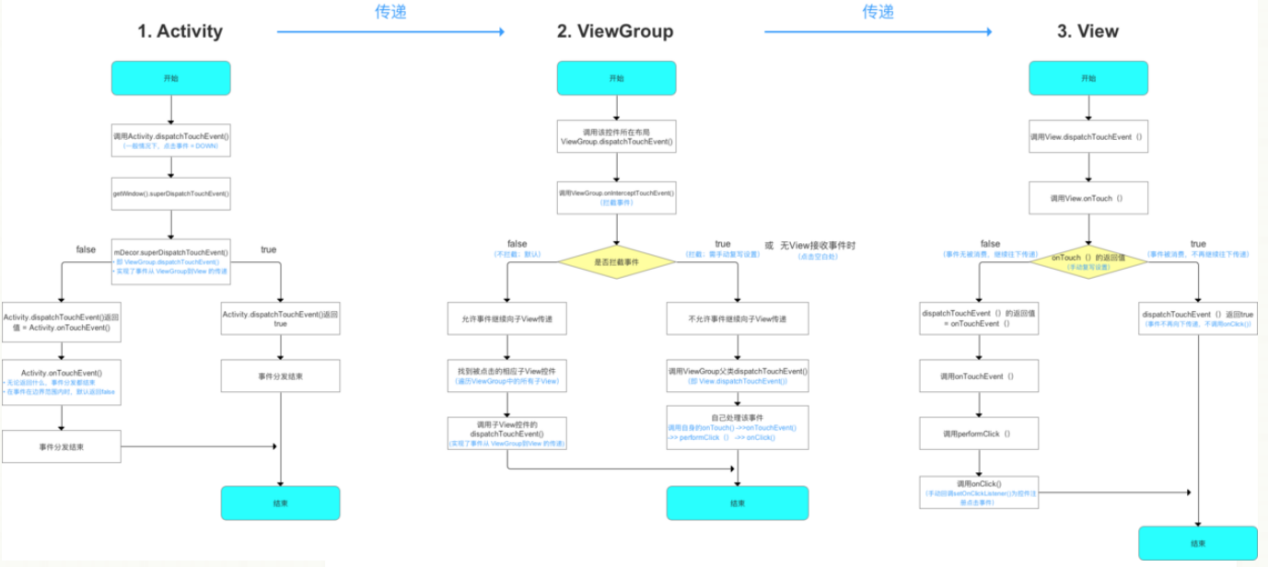
System.out.println("执行了onClick()");

}

});

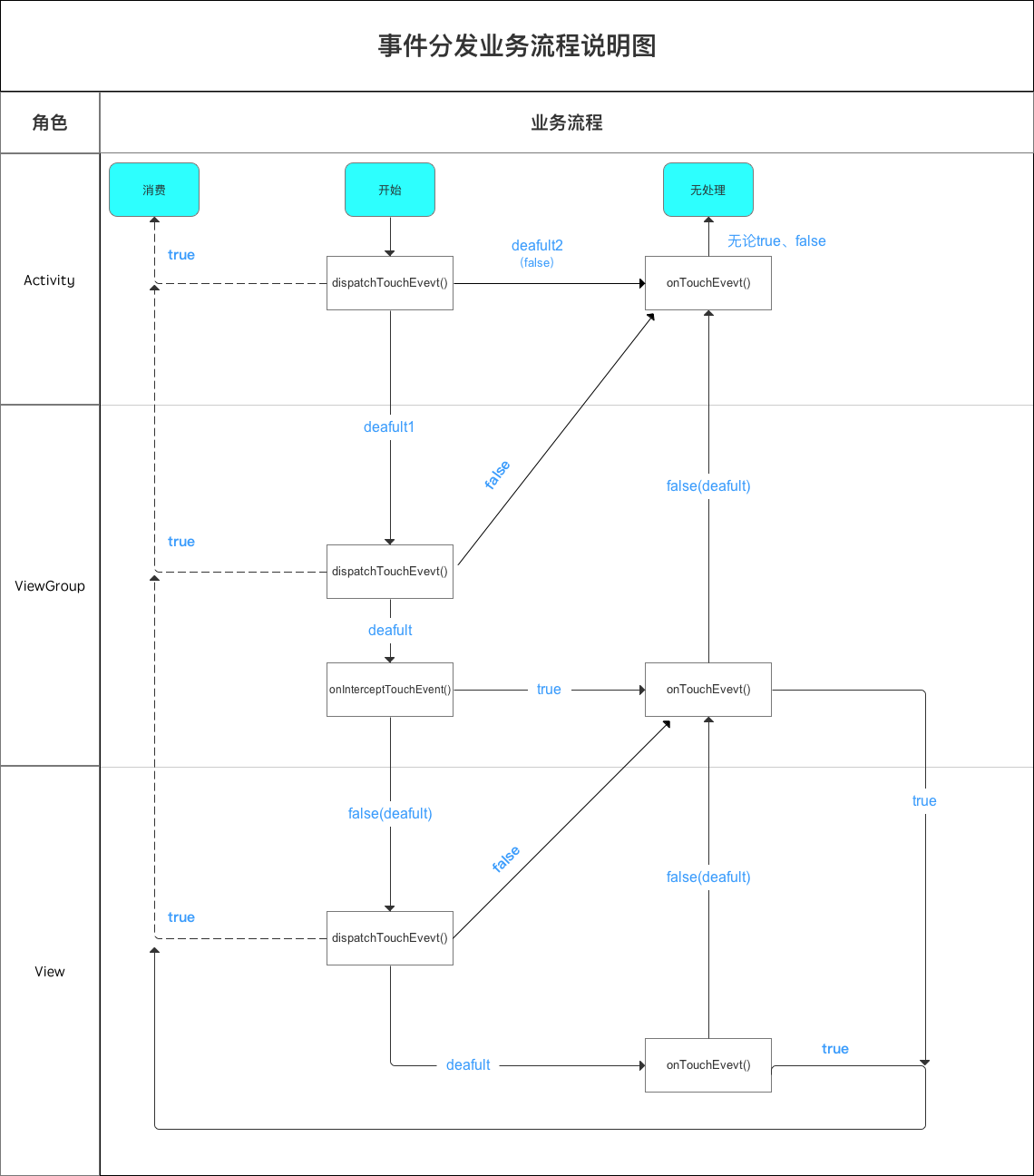


2.4 总结



3. 工作流程 总结

在本节中，我将结合源码，梳理出1个事件分发的工作流程总结，具体如下：



4.4 三者关系

/\*\*

\* 点击事件产生后

\*/

// 步骤1：调用dispatchTouchEvent（）

public boolean dispatchTouchEvent(MotionEvent ev) {

boolean consume = false; //代表 是否会消费事件

// 步骤2：判断是否拦截事件

if (onInterceptTouchEvent(ev)) {

// a. 若拦截，则将该事件交给当前View进行处理

// 即调用onTouchEvent (）方法去处理点击事件

consume = onTouchEvent (ev) ;

} else {

// b. 若不拦截，则将该事件传递到下层

// 即 下层元素的dispatchTouchEvent（）就会被调用，重复上述过程

// 直到点击事件被最终处理为止

consume = child.dispatchTouchEvent (ev) ;

}

// 步骤3：最终返回通知 该事件是否被消费（接收 & 处理）

return consume;

}

6.2 onTouch()和onTouchEvent()的区别

该2个方法都是在View.dispatchTouchEvent（）中调用

但onTouch（）优先于onTouchEvent执行；若手动复写在onTouch（）中返回true（即 将事件消费掉），将不会再执行onTouchEvent（）

注：若1个控件不可点击（即非enable），那么给它注册onTouch事件将永远得不到执行，具体原因看如下代码

// &&为短路与，即如果前面条件为false，将不再往下执行

// 故：onTouch（）能够得到执行需2个前提条件：

// 1. mOnTouchListener的值不能为空

// 2. 当前点击的控件必须是enable的

mOnTouchListener != null && (mViewFlags & ENABLED\_MASK) == ENABLED &&

mOnTouchListener.onTouch(this, event)

// 对于该类控件，若需监听它的touch事件，就必须通过在该控件中重写onTouchEvent（）来实现