

Android事件分发机制详解：史上最全面、最易懂 - 简书

简 jianshu.com/p/38015afcdb58



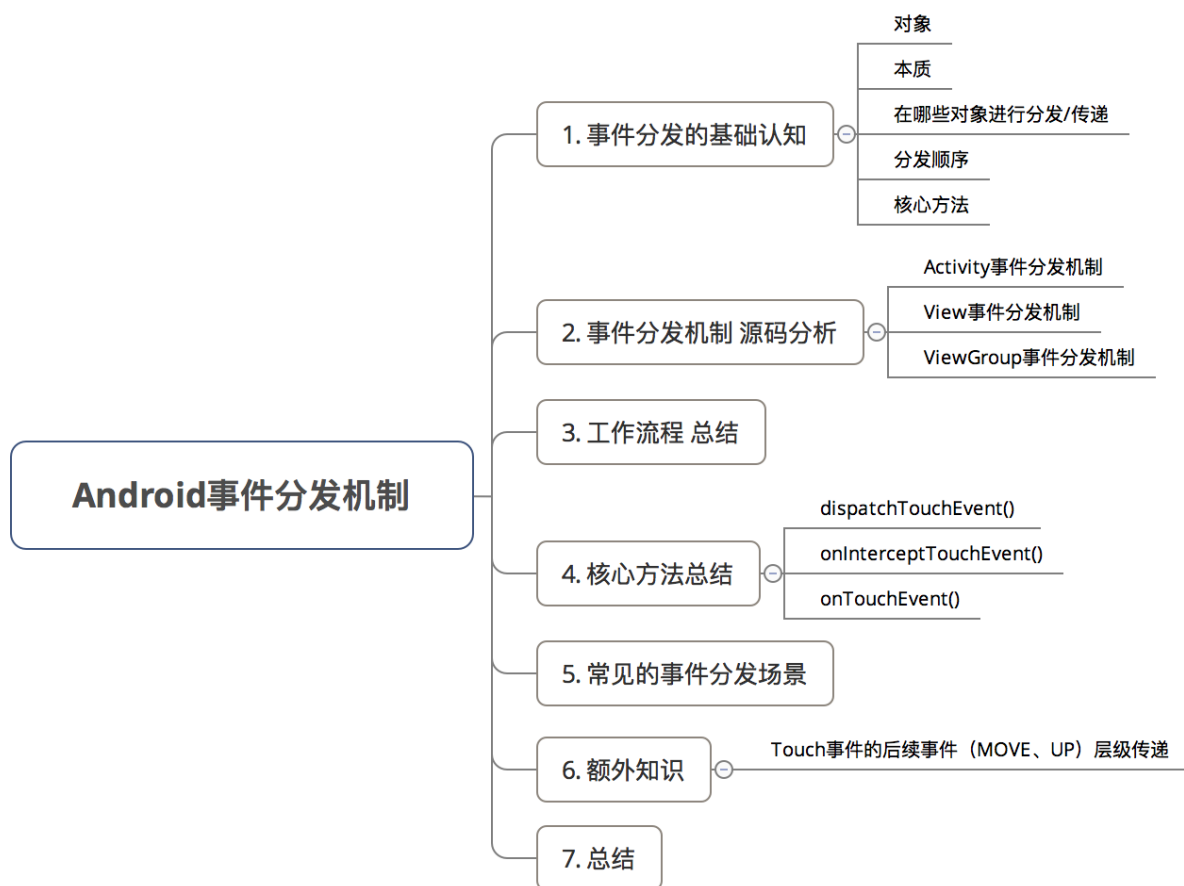
前言

- **Android** 事件分发机制是 **Android** 开发者必须了解的基础
- 网上有大量关于 **Android** 事件分发机制的文章，但存在一些问题：**内容不全、思路不清晰、无源码分析、简单问题复杂化等等**
- 今天，我将全面总结 **Android** 的事件分发机制，我能保证这是**市面上的最全面、最清晰、最易懂的**

1. 本文秉着“结论先行、详细分析在后”的原则，即先让大家感性认识，再通过理性分析从而理解问题；
2. 所以，请各位读者先记住结论，再往下继续看分析；

文章较长，阅读需要较长时间，建议收藏等充足时间再进行阅读

目录



示意图

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

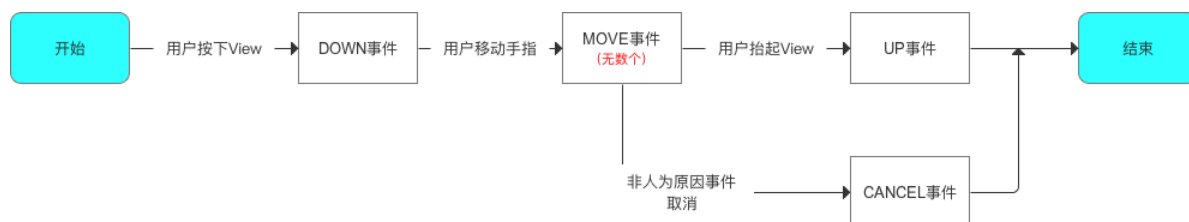
事件类型（4种）

事件类型	具体动作
<code>MotionEvent.ACTION_DOWN</code>	按下View（所有事件的开始）
<code>MotionEvent.ACTION_UP</code>	抬起View（与DOWN对应）
<code>MotionEvent.ACTION_MOVE</code>	滑动View
<code>MotionEvent.ACTION_CANCEL</code>	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

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：



事件列

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

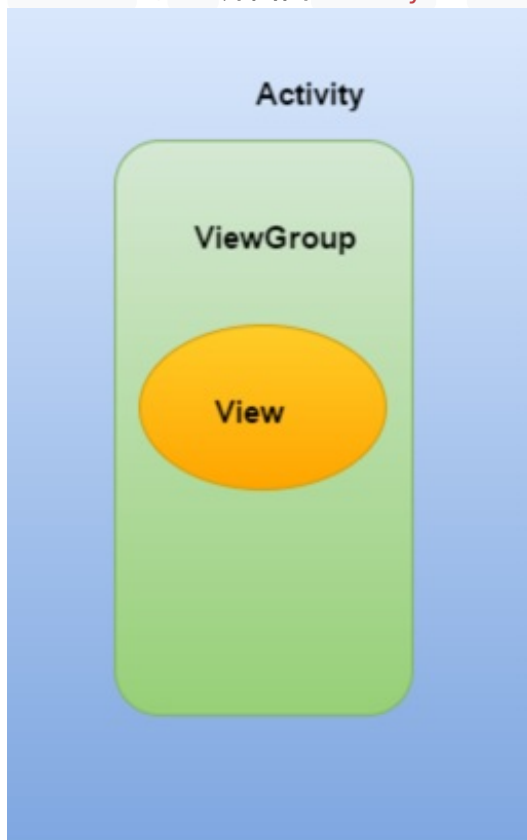
答：将点击事件（**MotionEvent**）传递到某个具体的 **View** & 处理的整个过程

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



UI界面

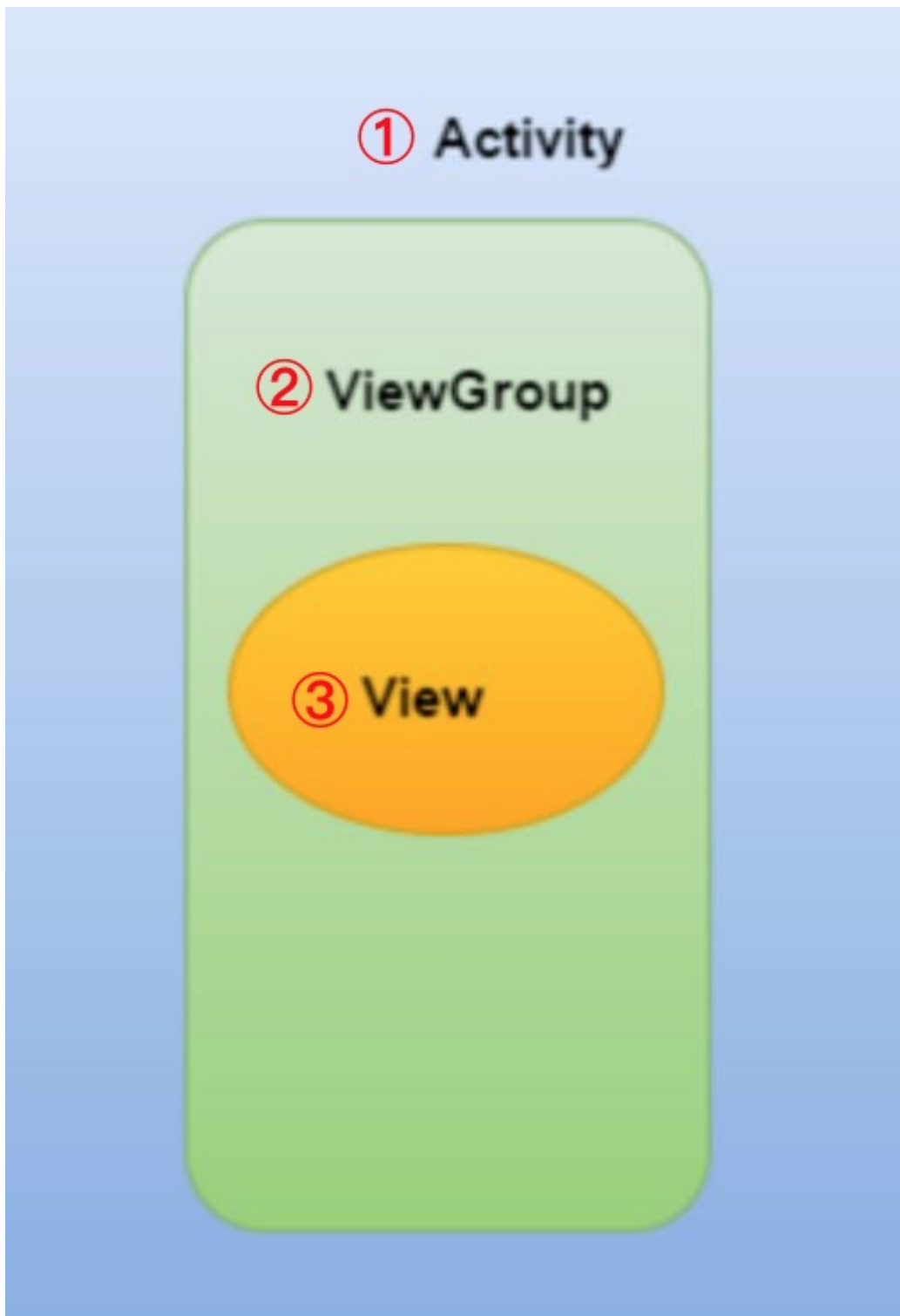
类型	简介	备注
Activity	控制生命周期 & 处理事件 (类似 控制器)	<ul style="list-style-type: none">• 统筹视图的添加 & 显示• 通过其他回调方法与Window、View交互
View	所有UI组件的基类	一般Button、TextView等控件都是继承父类View
ViewGroup	一组View的集合 (含多个子View)	<ul style="list-style-type: none">• 其本身也是View的子类• 是Android所有布局的父类：如LinearLayout等• 区别于普通View：ViewGroup实际上也是1个View，只是多了可包含子View & 定义布局参数的功能

示意图

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



示意图

1.5 事件分发过程由哪些方法协作完成？

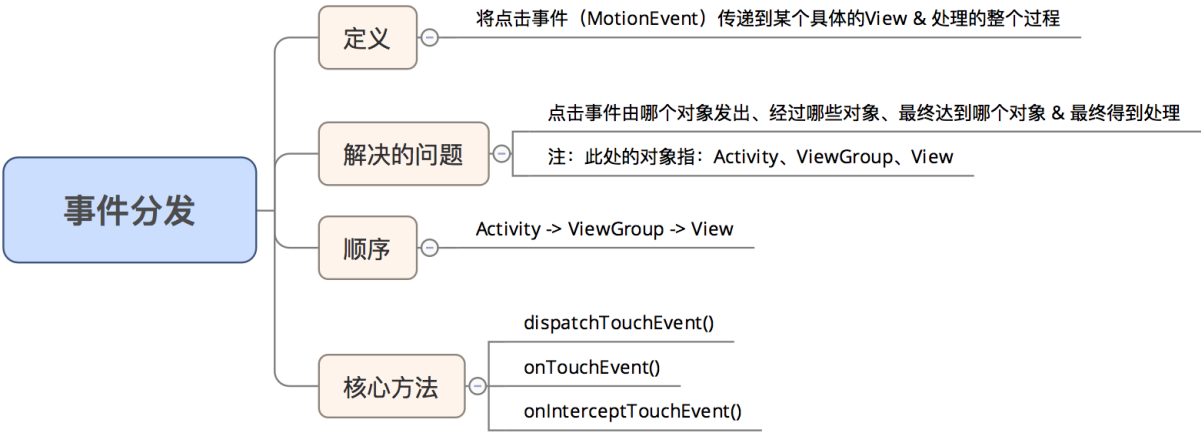
答：dispatchTouchEvent()、onInterceptTouchEvent()和onTouchEvent()

方法	作用	调用时刻
<code>dispatchTouchEvent()</code>	分发（传递） 点击事件	当点击事件能够传递给当前View时，该方法就会被调用
<code>onTouchEvent()</code>	处理点击事件	在 <code>dispatchTouchEvent()</code> 内部调用
<code>onInterceptTouchEvent()</code>	判断是否拦截了某个事件 • 只存在于ViewGroup • 普通的View无该方法	在ViewGroup的 <code>dispatchTouchEvent()</code> 内部调用

示意图

下文会对这3个方法进行详细介绍

1.6 总结



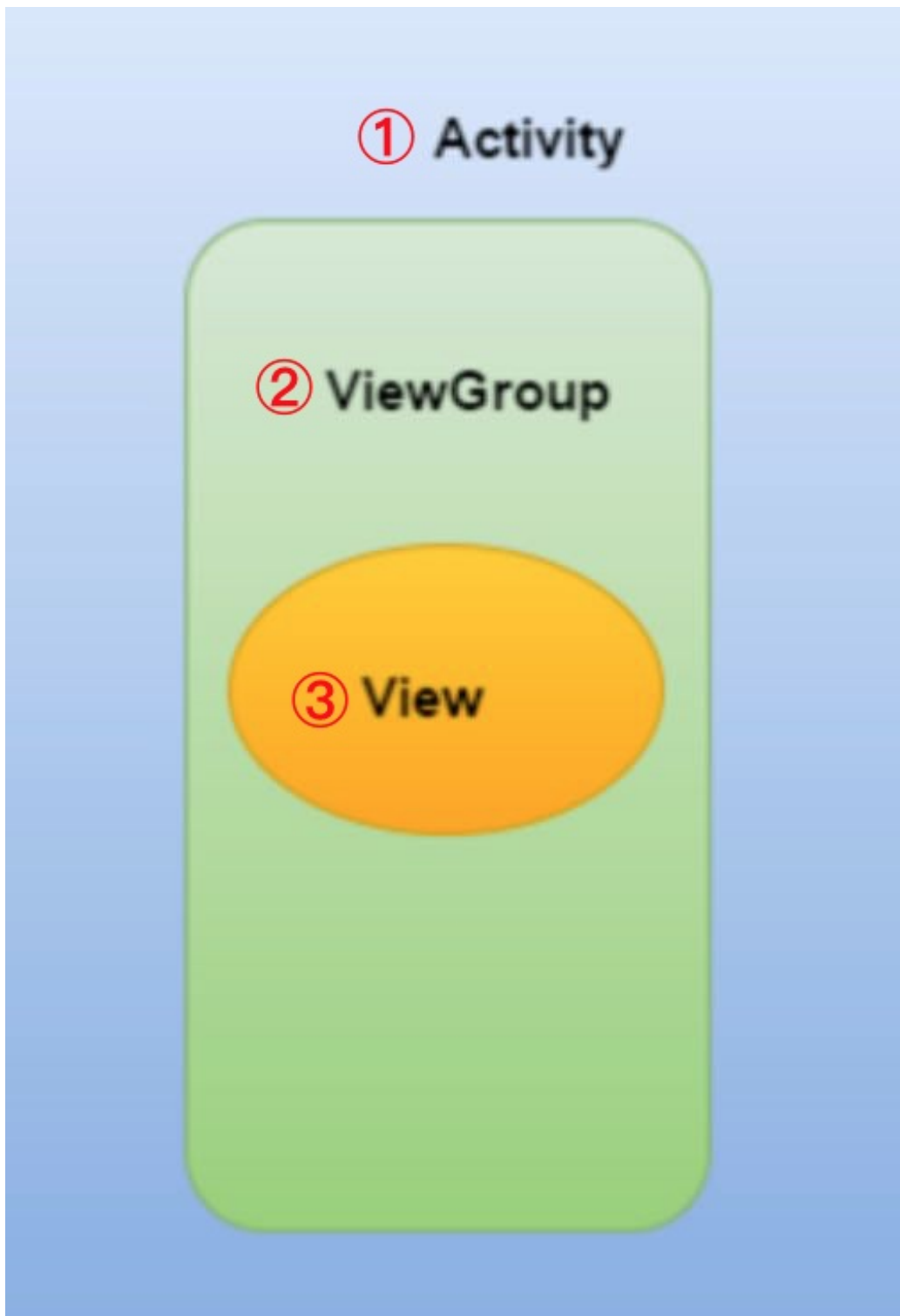
示意图

- 至此，相信大家已经对 **Android** 的事件分发有了感性的认知
- 下面，我将详细介绍 **Android** 事件分发机制

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**



示意图

- 从上可知，要想充分理解Android分发机制，本质上是要理解：
 1. Activity 对点击事件的分发机制
 2. ViewGroup 对点击事件的分发机制
 3. 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

2.1.1 源码分析

```
public boolean dispatchTouchEvent(MotionEvent ev) {

    if (ev.getAction() == MotionEvent.ACTION_DOWN) {
        onUserInteraction();
    }

    if (getWindow().superDispatchTouchEvent(ev)) {
        return true;
    }

    return onTouchEvent(ev);
}

public void onUserInteraction() {

}

@Override
public boolean superDispatchTouchEvent(MotionEvent event) {

    return mDecor.superDispatchTouchEvent(event);

}

public boolean superDispatchTouchEvent(MotionEvent event) {

    return super.dispatchTouchEvent(event);
}
```



```
}
```

```
public boolean onTouchEvent(MotionEvent event) {
```

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

```
    return false;
```

```
}
```

```
public boolean shouldCloseOnTouch(Context context, MotionEvent event) {
```

```
    if (mCloseOnTouchOutside && event.getAction() == MotionEvent.ACTION_DOWN  
        && isOutOfBounds(context, event) && peekDecorView() != null) {  
        return true;
```

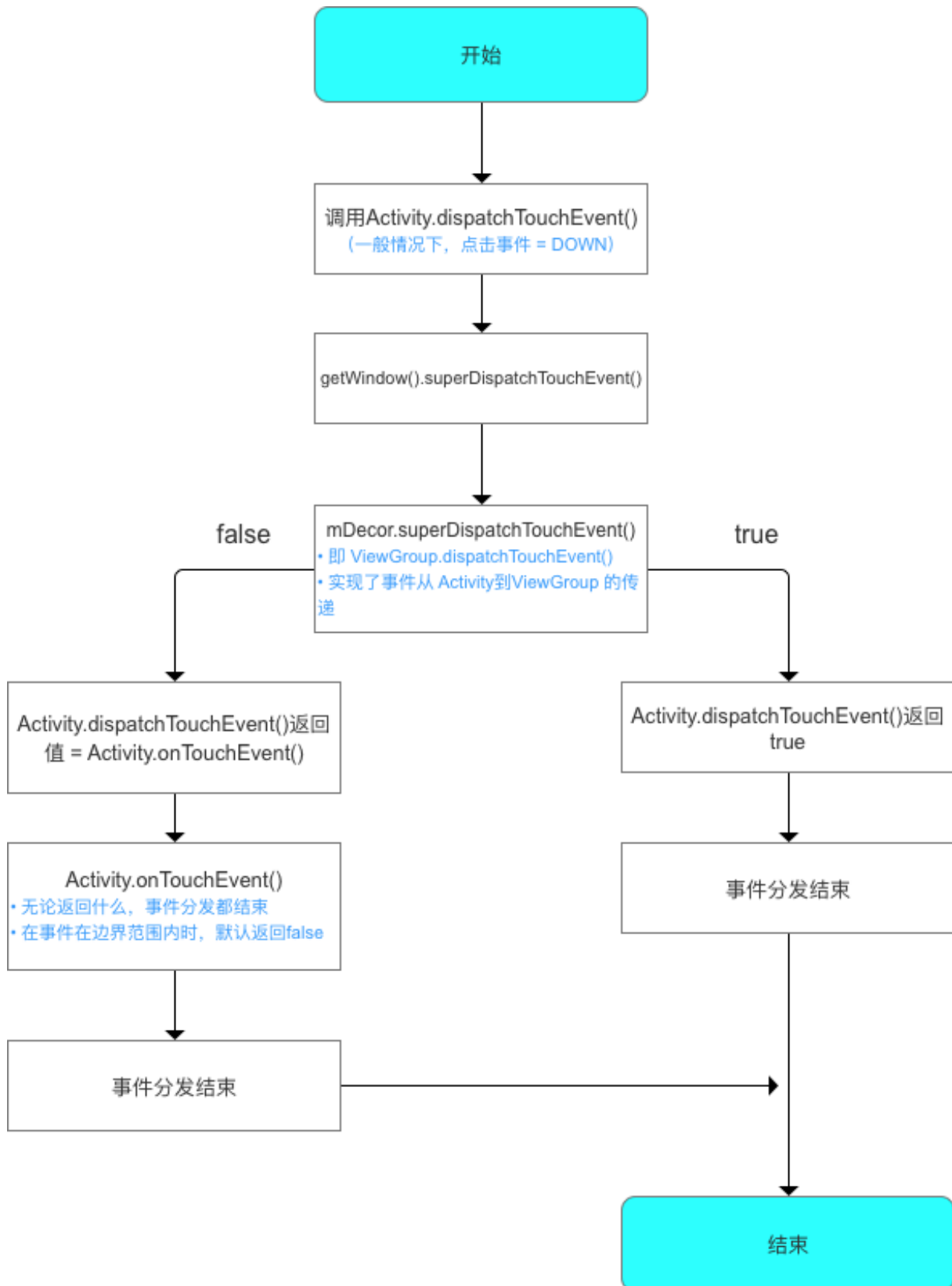
```
    }
```

```
    return false;
```

```
}
```

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()`)



示意图

方法总结

核心方法	调用时刻	返回结果说明			
		返回结果	具体含义	产生该结果的条件	后续动作
dispatchTouchEvent()	用户触碰屏幕产生点击事件时	默认1	调用该方法时会直接调用如右方法	默认、无条件	调用ViewGroup.dispatchTouchEvent()
		默认2	第2处默认调用方法	ViewGroup.dispatchTouchEvent()返回false时	调用Activity.onTouchEvent()
		true	当前事件被消费 (即事件已被View/ViewGroup接收&处理)	满足以下之一即可: • ViewGroup.dispatchTouchEvent()返回true • onTouchEvent() 返回true	• 事件分发结束 • 后续事件会继续分发到该 View
		false	当前事件未被消费 (即事件未被View/ViewGroup接收&处理)	onTouchEvent() 返回true	• 事件分发结束 • 当前View仍然接受此事件的其他事件 (与onTouchEvent()区别)
onTouchEvent()	ViewGroup.dispatchTouchEvent() 返回false后, 默认执行调用	true	判断了点击事件在Window边界外 (即此时事件也算被消费)	点击事件在边界外 (点击事件未被Activity下任何一个View接收/处理)	• 事件分发结束 • 当前View不再接受此事件的其他事件
		false	不处理当前事件	点击事件在边界内 (点击事件未被Activity下任何一个View接收/处理)	

示意图

那么, `ViewGroup` 的 `dispatchTouchEvent()` 什么时候返回 `true` / `false` ? 请继续往下看ViewGroup事件的分发机制

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

1. `Android 5.0` 后, `ViewGroup.dispatchTouchEvent()` 的源码发生了变化 (更加复杂), 但原理相同;
2. 本文为了让读者容易理解, 故采用 `Android 5.0` 前的版本

```
public boolean dispatchTouchEvent(MotionEvent ev) {  
  
    ...  
  
    if (disallowIntercept || !onInterceptTouchEvent(ev)) {  
  
  
  
  
  
  
  
  
  
        ev.setAction(MotionEvent.ACTION_DOWN);  
        final int scrolledXInt = (int) scrolledXFloat;  
        final int scrolledYInt = (int) scrolledYFloat;  
        final View[] children = mChildren;  
        final int count = mChildrenCount;  
  
  
  
        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);

            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;

                if (child.dispatchTouchEvent(ev)) {

                    mMotionTarget = child;
                    return true;

                }
            }
        }
    }
}
boolean isUpOrCancel = (action == MotionEvent.ACTION_UP) ||
    (action == MotionEvent.ACTION_CANCEL);
if (isUpOrCancel) {
    mGroupFlags &= ~FLAG_DISALLOW_INTERCEPT;
}
final View target = mMotionTarget;

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);

}

...

}

```

```

public boolean onInterceptTouchEvent(MotionEvent ev) {

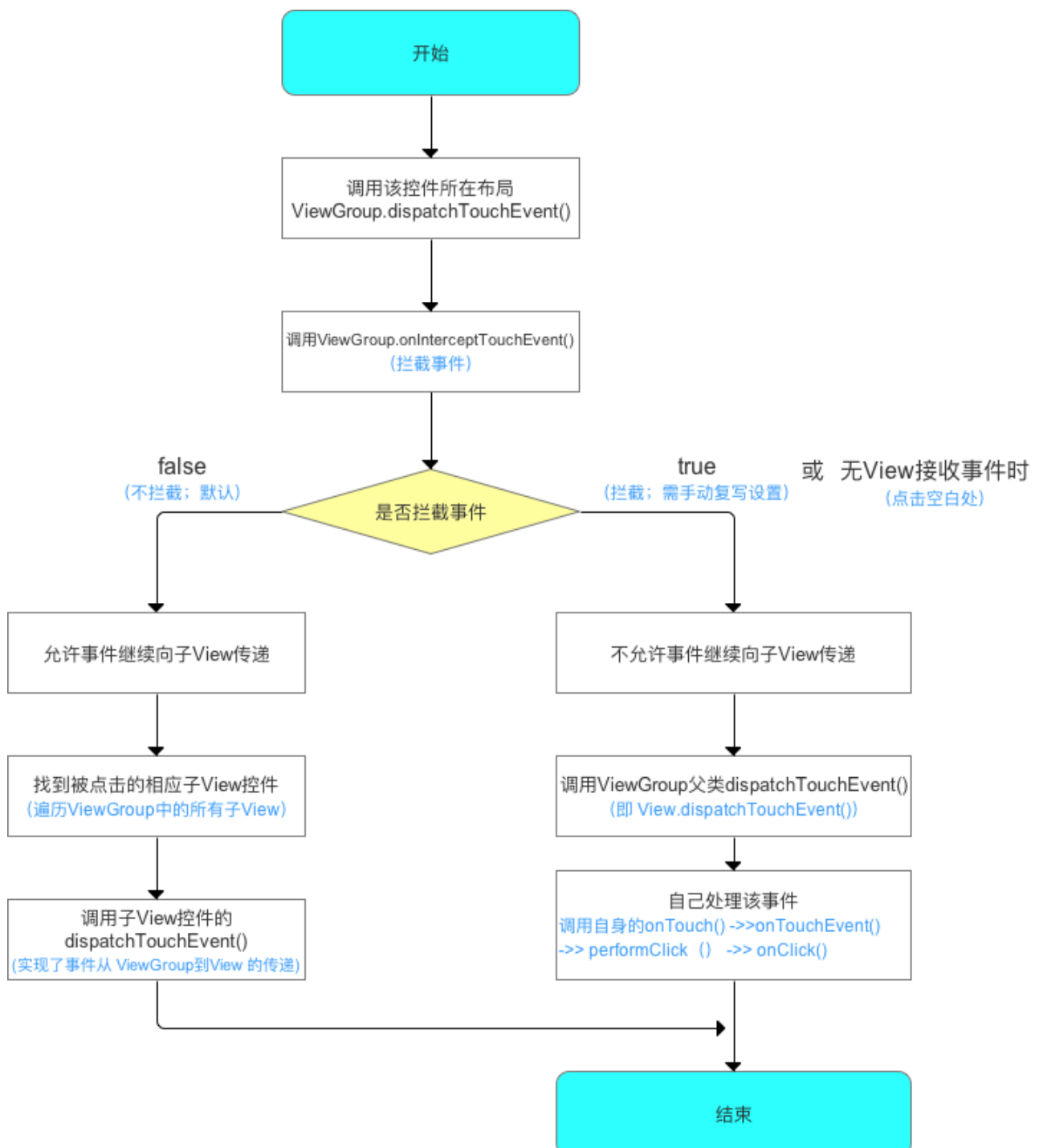
    return false;

}

```

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



示意图

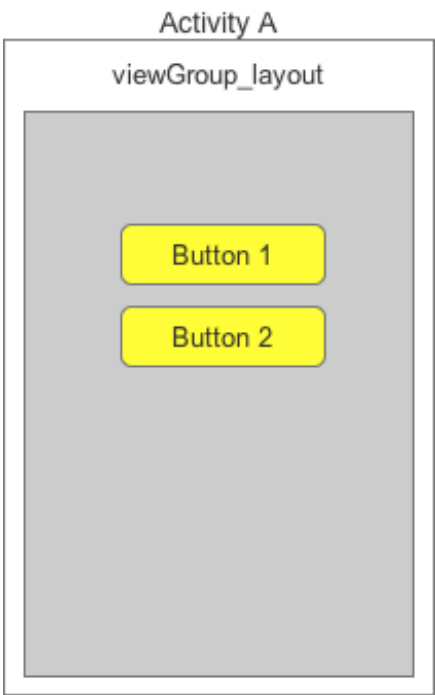
核心方法总结

核心方法	调用时刻	返回结果说明			
		返回结果	具体含义	产生该结果的条件	后续动作
dispatchTouchEvent()	事件从Activity传递过来时	默认	调用该方法时会直接调用如右方法	默认、无条件	调用ViewGroup.onInterceptTouchEvent()
		true	当前事件被消费 (即事件已被View/ViewGroup接收&处理)	满足以下之一即可： • 子View.dispatchTouchEvent()返回true • ViewGroup父类的dispatchTouchEvent(), 即View.dispatchTouchEvent()返回true	• 事件分发结束 • 后续事件会继续分发到该 View
		false	当前事件未被消费 (即事件未被ViewGroup自身接收&处理)	ViewGroup父类的dispatchTouchEvent(), 即View.dispatchTouchEvent()返回false	将事件回传给上层Activity.onTouchEvent()处理
onInterceptTouchEvent()	在ViewGroup的dispatchTouchEvent() 内部调用	true	当前事件被ViewGroup拦截	手动设置：复写onInterceptTouchEvent()	• 事件停止往下传递 • ViewGroup自己处理事件，调用父类super.dispatchTouchEvent(), 最终执行自己的onTouchEvent() ; • 同一个事件列的其他事件都直接交由该View处理；在同一个事件列中该方法不会再次被调用；
		false (default)	当前事件未被ViewGroup拦截	默认设置	• 事件继续往下传递 • 事件传递到子view，即调用View.dispatchTouchEvent() 处理 • 当前View仍然接受此事件的其他事件 (与onTouchEvent()区别)
onTouchEvent()	ViewGroup父类的dispatchTouchEvent(), 即super.dispatchTouchEvent()时	true (处理)	ViewGroup处理了当前事件	通过setOnClickListener () 为ViewGroup注册1个点击事件	• 事件分发结束，逐层返回true结果 • 后续事件序列让其处理；
		false (不处理)	ViewGroup无处理当前事件	无通过setOnClickListener () 为ViewGroup注册1个点击事件	• 将事件向上传给上层Activity的onTouchEvent()处理 • 当前View不再接受此事件的其他事件 (与dispatchTouchEvent () 、onInterceptTouchEvent () 的区别)

示意图

2.2.3 Demo讲解

- 布局如下



布局层次

- 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>
```

核心代码：*MainActivity.java*

```

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);

        myLayout.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Log.d("TAG", "点击了ViewGroup");
            }
        });

        button1.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Log.d("TAG", "点击了button1");
            }
        });

        button2.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Log.d("TAG", "点击了button2");
            }
        });
    }
}

```

结果测试

1. 点击按钮

```

11-23 22:07:56.929 6149-6149/scut.carson_ho.viewgroup_test I/System.out: 点击了Button1
11-23 22:07:58.185 6149-6149/scut.carson_ho.viewgroup_test I/System.out: 点击了Button2

```

2. 再点击空白处

```

11-23 22:07:56.929 6149-6149/scut.carson_ho.viewgroup_test I/System.out: 点击了Button1
11-23 22:07:58.185 6149-6149/scut.carson_ho.viewgroup_test I/System.out: 点击了Button2
11-23 22:09:05.197 6149-6149/scut.carson_ho.viewgroup_test I/System.out: 点击了ViewGroup

```

示意图

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

```
public boolean dispatchTouchEvent(MotionEvent event) {  
  
    if (mOnTouchListener != null && (mViewFlags & ENABLED_MASK) == ENABLED &&  
        mOnTouchListener.onTouch(this, event)) {  
        return true;  
    }  
    return onTouchEvent(event);  
}
```

```
public void setOnTouchListener(OnTouchListener l) {  
  
    mOnTouchListener = l;  
  
}
```

```
button.setOnTouchListener(new OnTouchListener() {  
    @Override  
    public boolean onTouch(View v, MotionEvent event) {  
  
        return false;  
    }  
});
```

接下来，我们继续看：`onTouchEvent(event)`的源码分析

1. 详情请看注释
2. **Android 5.0** 后 `View.onTouchEvent()` 源码发生了变化（更加复杂），但原理相同；
3. 本文为了让读者更好理解，所以采用 **Android 5.0** 前的版本

```
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;
        }
    }

    if (((viewFlags & CLICKABLE) == CLICKABLE ||
        (viewFlags & LONG_CLICKABLE) == LONG_CLICKABLE)) {

        switch (event.getAction()) {

            case MotionEvent.ACTION_UP:
                boolean prepressed = (mPrivateFlags & PREPRESSED) != 0;

                ...

                performClick();
                break;

            case MotionEvent.ACTION_DOWN:
                if (mPendingCheckForTap == null) {
                    mPendingCheckForTap = new CheckForTap();
                }
                mPrivateFlags |= PREPRESSED;
                mHasPerformedLongPress = false;
                postDelayed(mPendingCheckForTap, ViewConfiguration.getTapTimeout());
                break;

            case MotionEvent.ACTION_CANCEL:
                mPrivateFlags &= ~PREPRESSED;
                refreshDrawableState();
                removeTapCallback();
            
```

```

        break;

    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)) {

            removeTapCallback();
            if ((mPrivateFlags & PRESSED) != 0) {

                removeLongPressCallback();

                mPrivateFlags &= ~PRESSED;
                refreshDrawableState();
            }
        }
        break;
    }

    return true;
}

return false;
}

public boolean performClick() {

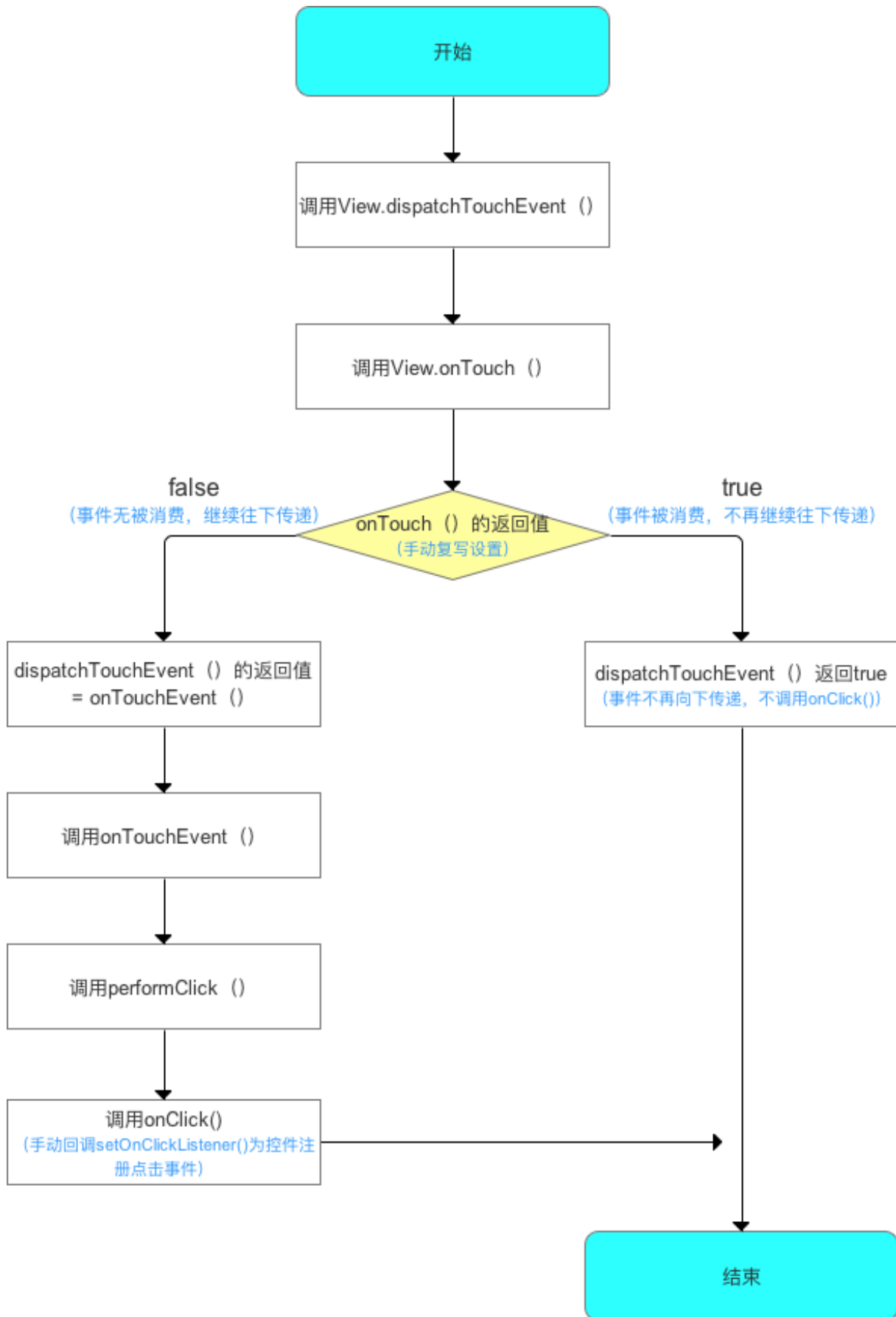
    if (mOnClickListener != null) {
        playSoundEffect(SoundEffectConstants.CLICK);
        mOnClickListener.onClick(this);
        return true;
    }

    return false;
}

```

2.3.2 总结

每当控件被点击时：



示意图

注：`onTouch()` 的执行先于 `onClick()`

核心方法总结

核心方法	调用时刻	返回结果说明			
		返回结果	具体含义	产生该结果的条件	后续动作
dispatchTouchEvent()	事件从ViewGroup传递过来时	默认	调用该方法时会直接调用如右方法	默认、无条件	调用View.onTouchEvent()
		true	当前事件被消费 (即事件已被View/ViewGroup接收&处理)	满足以下之一即可： • View.onTouchEvent()返回true • ViewGroup.onTouchEvent()返回true	• 事件分发结束 • 后续事件会继续分发到该 View
		false	当前事件未被消费 (即事件未被View/ViewGroup接收&处理)	View.onTouchEvent() 返回false	将事件回传给上层ViewGroup.onTouchEvent()处理
onTouchEvent()	View.dispatchTouchEvent()默认调用	true (处理)	View处理了当前事件	通过setOnClickListener () 为View注册1个点击事件	• 事件分发结束，逐层返回true结果 • 后续事件序列让其处理；
		false (不处理)	View无处理当前事件	无通过setOnClickListener () 为View注册1个点击事件	• 将事件向上传给上层ViewGroup的onTouchEvent()处理 • 当前View不再接受此事件的其他事件 (与dispatchTouchEvent () 、onInterceptTouchEvent () 区别)

示意图

2.3.3 Demo讲解

下面我将用 **Demo** 验证上述的结论

```
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;
    }
});
```

```
button.setOnClickListener(new View.OnClickListener() {

    @Override
    public void onClick(View v) {
        System.out.println("执行了onClick()");
    }

});
```

```
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;
    }
});
```

```
button.setOnClickListener(new View.OnClickListener() {

    @Override
    public void onClick(View v) {
        System.out.println("执行了onClick()");
    }

});
```

测试结果

1. onTouch()返回false

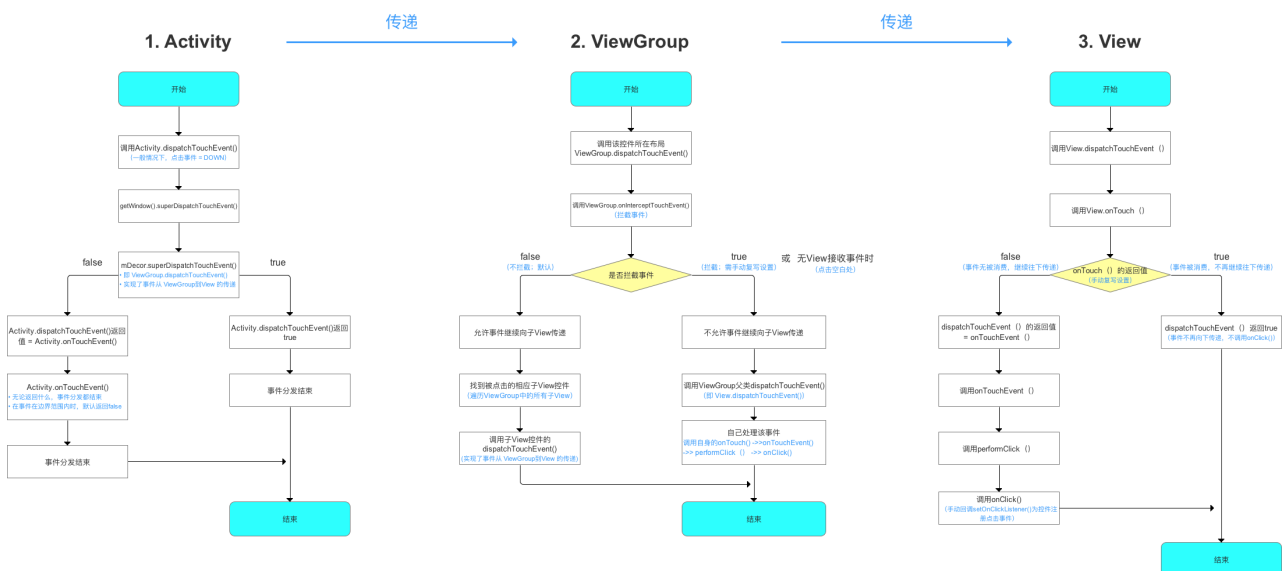
```
04:08:36.514 1703-1703/scut.carson_ho.touch_event I/System.out: 执行了onTouch(), 动作是:0
04:08:36.578 1703-1703/scut.carson_ho.touch_event I/System.out: 执行了onTouch(), 动作是:1
04:08:36.578 1703-1703/scut.carson_ho.touch_event I/System.out: 执行了onClick()
```

2. onTouch()返回true

```
04:15:54.578 7334-7334/scut.carson_ho.touch_event I/System.out: 执行了onTouch(), 动作是:0
04:15:54.638 7334-7334/scut.carson_ho.touch_event I/System.out: 执行了onTouch(), 动作是:1
```

示意图

2.4 总结



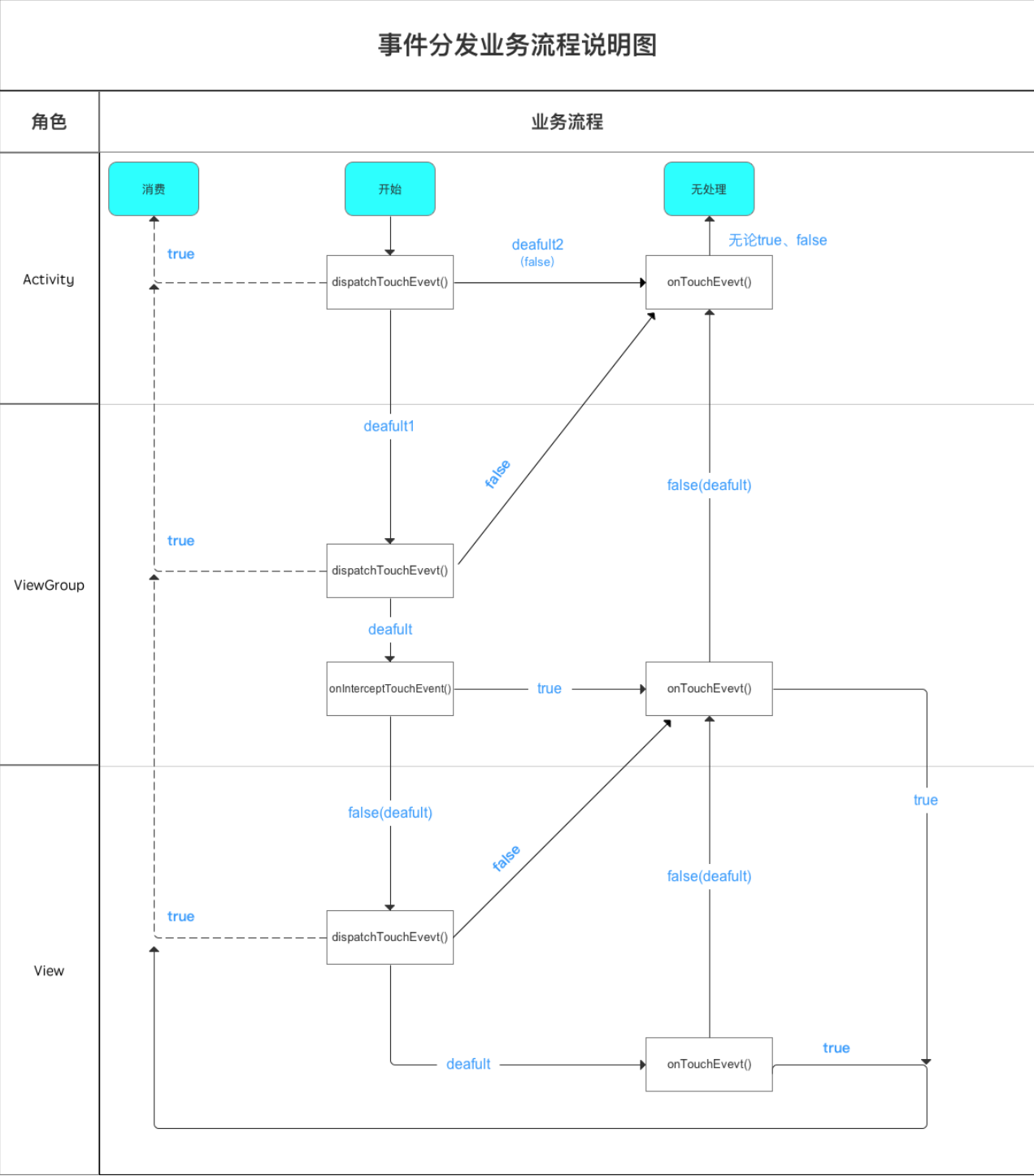
示意图

若您已经看到此处，那么恭喜你，你已经能非常熟悉掌握Android的事件分发机制了

即：Activity、ViewGroup、View 的事件分发机制

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：



示意图

左侧虚线：具备相关性 & 逐层返回

以角色为核心的图解说明

使用对象	核心方法	调用时刻	返回结果说明			
			返回结果	具体含义	产生该结果的条件	后续动作
Activity	dispatchTouchEvent()	用户触碰屏幕产生点击事件时	默认1	调用该方法时会直接调用如右方法	默认、无条件	调用ViewGroup.dispatchTouchEvent()
			默认2	第2处默认调用方法	ViewGroup.dispatchTouchEvent()返回false时	调用Activity.onTouchEvent()
			true	当前事件被消费 (即事件已被View/ViewGroup接收&处理)	满足以下之一即可： • ViewGroup.dispatchTouchEvent()返回true • onTouchEvent() 返回true	• 事件分发结束 • 后续事件会继续分发到该 View
			false	当前事件无被消费 (即事件未被View/ViewGroup接收&处理)	onTouchEvent() 返回false	• 事件分发结束 • 当前View仍然接受此事件的其他事件 (与onTouchEvent()区别)
	onTouchEvent()	ViewGroup.dispatchTouchEvent() 返回false后，默认执行调用	true	判断了点击事件在Window边界外 (即此时事件也算被消费)	点击事件在边界外 (点击事件未被Activity下任何一个View接收/处理)	• 事件分发结束 • 当前View不再接受此事件的其他事件
			false	不处理当前事件	点击事件在边界内 (点击事件未被Activity下任何一个View接收/处理)	
ViewGroup	dispatchTouchEvent()	事件从Activity传递过来时	默认	调用该方法时会直接调用如右方法	默认、无条件	调用ViewGroup.onInterceptTouchEvent()
			true	当前事件被消费 (即事件已被View/ViewGroup接收&处理)	满足以下之一即可： • 子View.dispatchTouchEvent()返回true • ViewGroup父类的dispatchTouchEvent(), 即View.dispatchTouchEvent()返回true	• 事件分发结束 • 后续事件会继续分发到该 View
			false	当前事件无被消费 (即事件未被ViewGroup自身接收&处理)	ViewGroup父类的dispatchTouchEvent(), 即View.dispatchTouchEvent()返回false	将事件回传给上层Activity.onTouchEvent()处理
	onInterceptTouchEvent()	在ViewGroup的dispatchTouchEvent() 内部调用	true	当前事件被ViewGroup拦截	手动设置：重写onInterceptTouchEvent()	• 事件停止往下传递 • ViewGroup自己处理事件，调用父类super.dispatchTouchEvent(), 最终执行自己的onTouchEvent(); • 同一个事件列的其他事件都直接交由该View处理；在同一个事件列中该方法不会再次被调用；
			false (default)	当前事件未被ViewGroup拦截	默认设置	• 事件继续往下传递 • 事件传递到子view，即调用View.dispatchTouchEvent() 处理 • 当前View仍然接受此事件的其他事件 (与onTouchEvent()区别)
	onTouchEvent()	ViewGroup父类的dispatchTouchEvent(), 即super.dispatchTouchEvent()时	true (处理)	ViewGroup处理了当前事件	通过setOnClickListener () 为ViewGroup注册1个点击事件	• 事件分发结束，逐层返回true结果 • 后续事件序列让其处理；
			false (不处理)	ViewGroup无处理当前事件	无通过setOnClickListener () 为ViewGroup注册1个点击事件	• 将事件向上传给上层Activity的onTouchEvent()处理 • 当前View不再接受此事件的其他事件 (与dispatchTouchEvent () 、onInterceptTouchEvent () 的区别)
View	dispatchTouchEvent()	事件从ViewGroup传递过来时	默认	调用该方法时会直接调用如右方法	默认、无条件	调用View.onTouchEvent()
			true	当前事件被消费 (即事件已被View/ViewGroup接收&处理)	满足以下之一即可： • View.onTouchEvent()返回true • ViewGroup.onTouchEvent()返回true	• 事件分发结束 • 后续事件会继续分发到该 View
			false	当前事件无被消费 (即事件未被View/ViewGroup接收&处理)	View.onTouchEvent() 返回false	将事件回传给上层ViewGroup.onTouchEvent()处理
	onTouchEvent()	View.dispatchTouchEvent()默认调用	true (处理)	View处理了当前事件	通过setOnClickListener () 为View注册1个点击事件	• 事件分发结束，逐层返回true结果 • 后续事件序列让其处理；
			false (不处理)	View无处理当前事件	无通过setOnClickListener () 为View注册1个点击事件	• 将事件向上传给上层ViewGroup的onTouchEvent()处理 • 当前View不再接受此事件的其他事件 (与dispatchTouchEvent () 、onInterceptTouchEvent () 区别)

示意图

以方法为核心的图解说明

方法	使用对象	作用	调用时刻	返回结果说明		
				返回结果	具体含义	后续动作
dispatchTouchEvent()	• Activity • ViewGroup • View	分发（传递）点击事件	当点击事件能够传递给当前层时（Activity、ViewGroup、View），该方法就会被调用	默认	调用该方法时会直接调用如右方法	根据当前对象的不同而返回方法不同 • Activity：调用ViewGroup.dispatchTouchEvent() / Activity.onTouchEvent() • ViewGroup：调用自身的onInterceptTouchEvent() • View：调用自身的onTouchEvent（）
				true	当前事件被消费 <small>（即事件已被View/ViewGroup接收&处理）</small>	• 事件停止分发、逐层往上返回（若无上层返回，则结束） • 后续事件会继续分发到该 View
				false	当前事件无被消费 <small>（即事件未被View/ViewGroup接收&处理）</small>	• 将事件回传给上层的onTouchEvent（）处理（若无上层返回，则结束） • 当前View仍然接受此事件的其他事件（与onTouchEvent()区别）
onInterceptTouchEvent()	• ViewGroup	判断是否拦截了某个事件 • 只存在于ViewGroup • 普通的View无该方法	在ViewGroup的dispatchTouchEvent() 内部调用	true	当前事件被ViewGroup拦截	• 事件停止往下传递 • ViewGroup自己处理事件，调用父类super.dispatchTouchEvent()，最终执行自己的onTouchEvent()； • 同一个事件序列的其他事件都直接交由该View处理；在同一个事件序列中该方法不会再次被调用；
				false <small>(default)</small>	当前事件无被ViewGroup拦截	• 事件继续往下传递 • 事件传递到子view，调用View.dispatchTouchEvent() 方法中去处理 • 当前View仍然接受此事件的其他事件（与onTouchEvent()区别）
onTouchEvent()	• Activity • ViewGroup • View	处理点击事件	在dispatchTouchEvent() 内部调用	true <small>(处理)</small>	当前使用对象处理了当前事件 <small>（使用对象指：Activity、View、Group）</small>	• 事件停止分发、逐层往dispatchTouchEvent() 返回 <small>（对于Activity：先返回当前dispatchTouchEvent()；由于无上层，故结束）</small> • 后续事件序列让其处理；
				false <small>(不处理)</small>	当前使用对象无处理当前事件 <small>（使用对象指：Activity、View、Group）</small>	• 将事件向上传给上层的onTouchEvent()处理 <small>（对于Activity：由于无上层，故结束）</small> • 当前View不再接受此事件的其他事件 <small>（与dispatchTouchEvent（）、onInterceptTouchEvent（）的区别）</small>
特别注意	• 注意点1：各层dispatchTouchEvent() 返回true的情况保持一致（图中虚线） • 原因：上层dispatchTouchEvent() 的返回true情况 取决于 下层dispatchTouchEvent() 是否返回sure，如Activity.dispatchTouchEvent() 返回true的情况 = ViewGroup.dispatchTouchEvent() 返回true • 注意点2：各层dispatchTouchEvent() 与 onTouchEvent()的返回情况保持一致 • 原因：最下层View的dispatchTouchEvent()的返回值 取决于 View.onTouchEvent()的返回值；结合注意点1，逐层往上返回，从而保持一致					

示意图

4. 核心方法总结

- 已知事件分发过程的核心方法为：`dispatchTouchEvent()`、`onInterceptTouchEvent()`和 `onTouchEvent()`

方法	使用对象	作用	调用时刻	返回结果说明		
				返回结果	具体含义	后续动作
dispatchTouchEvent()	• Activity • ViewGroup • View	分发（传递）点击事件	当点击事件能够传递给当前层时 (Activity、ViewGroup、View) , 该方法就会被调用	默认	调用该方法时会直接调用如右方法	根据当前对象的不同而返回方法不同 • Activity: 调用ViewGroup.dispatchTouchEvent() / Activity.onTouchEvent() • ViewGroup: 调用自身的onInterceptTouchEvent() • View: 调用自身的onTouchEvent ()
				true	当前事件被消费 (即事件已被View/ViewGroup接收&处理)	• 事件停止分发、逐层往上返回 (若无上层返回, 则结束) • 后续事件会继续分发到该View
				false	当前事件未被消费 (即事件未被View/ViewGroup接收&处理)	• 将事件回传给上层的onTouchEvent () 处理 (若无上层返回, 则结束) • 当前View仍然接受此事件的其他事件 (与onTouchEvent()区别)
onInterceptTouchEvent()	• ViewGroup	判断是否拦截了某个事件 • 只存在于ViewGroup • 普通的View无该方法	在ViewGroup的 dispatchTouchEvent() 内部调用	true	当前事件被ViewGroup拦截	• 事件停止往下传递 • ViewGroup自己处理事件, 调用父类super.dispatchTouchEvent(), 最终执行自己的onTouchEvent(); • 同一个事件列的其他事件都直接交由该View处理; 在同一个事件列中该方法不会再次被调用;
				false (default)	当前事件未被ViewGroup拦截	• 事件继续往下传递 • 事件传递到子view, 调用View.dispatchTouchEvent() 方法中去处理 • 当前View仍然接受此事件的其他事件 (与onTouchEvent()区别)
onTouchEvent()	• Activity • ViewGroup • View	处理点击事件	在dispatchTouchEvent() 内部调用	true (处理)	当前使用对象处理了当前事件 (使用对象指: Activity、View、Group)	• 事件停止分发、逐层往上返回 (若无上层返回, 则结束) • 后续事件序列让其处理;
				false (不处理)	当前使用对象无处理当前事件 (使用对象指: Activity、View、Group)	• 将事件向上传给上层的onTouchEvent()处理 • 当前View不再接受此事件的其他事件 (若无上层返回, 则结束) (与dispatchTouchEvent ()、onInterceptTouchEvent () 的区别)
特别注意	• 注意点1: 各层dispatchTouchEvent() 返回true的情况保持一致 (图中虚线) • 原因: 上层dispatchTouchEvent() 的返回true情况 取决于 下层dispatchTouchEvent() 是否返回sure, 如Activity.dispatchTouchEvent() 返回true的情况 = ViewGroup.dispatchTouchEvent() 返回true • 注意点2: 各层dispatchTouchEvent() 与 onTouchEvent()的返回情况保持一致 • 原因: 最下层View的dispatchTouchEvent()的返回值 取决于 View.onTouchEvent()的返回值; 结合注意点1, 逐层往上返回, 从而保持一致					

示意图

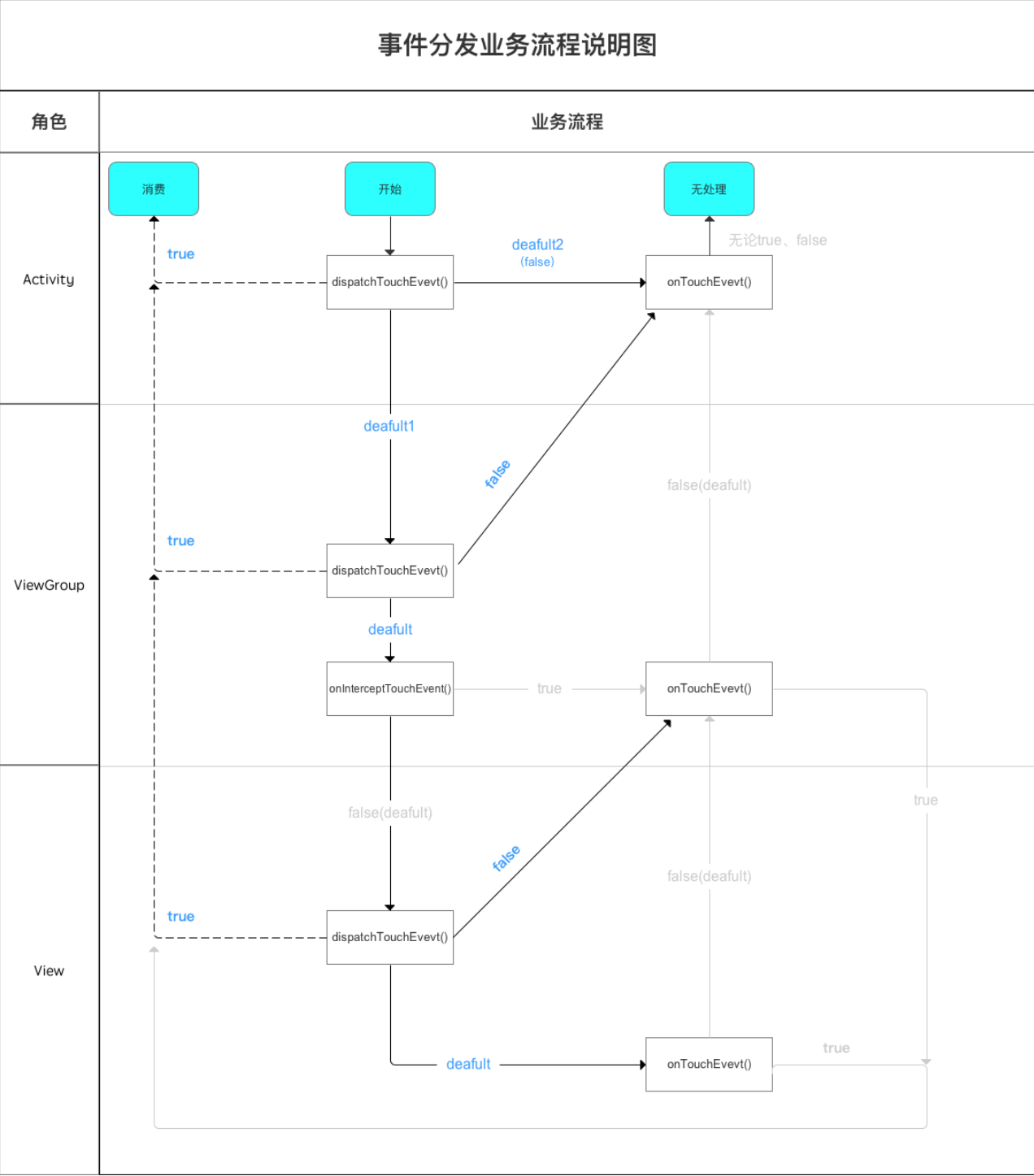
- 下面, 我将结合总结的工作流程, 再次详细讲解该3个方法

4.1 dispatchTouchEvent()

简介

使用对象	作用	调用时刻	返回结果说明		
			返回结果	具体含义	后续动作
<ul style="list-style-type: none">ActivityViewGroupView	分发（传递）点击事件	当点击事件能够传递给当前层时 (Activity、ViewGroup、View)，该方法就会被调用	默认	调用该方法时会直接调用如右方法	根据当前对象的不同而返回方法不同 <ul style="list-style-type: none">Activity: 调用ViewGroup.dispatchTouchEvent() / Activity.onTouchEvent()ViewGroup: 调用自身的onInterceptTouchEvent()View: 调用自身的onTouchEvent ()
			true	当前事件被消费 (即事件已被View/ViewGroup接收&处理)	事件停止分发、逐层往上返回 (若无上层返回, 则结束) 后续事件会继续分发到该 View
			false	当前事件未被消费 (即事件未被View/ViewGroup接收&处理)	将事件回传给上层的onTouchEvent () 处理 (若无上层返回, 则结束) 当前View仍然接受此事件的其他事件 (与onTouchEvent()区别)

示意图



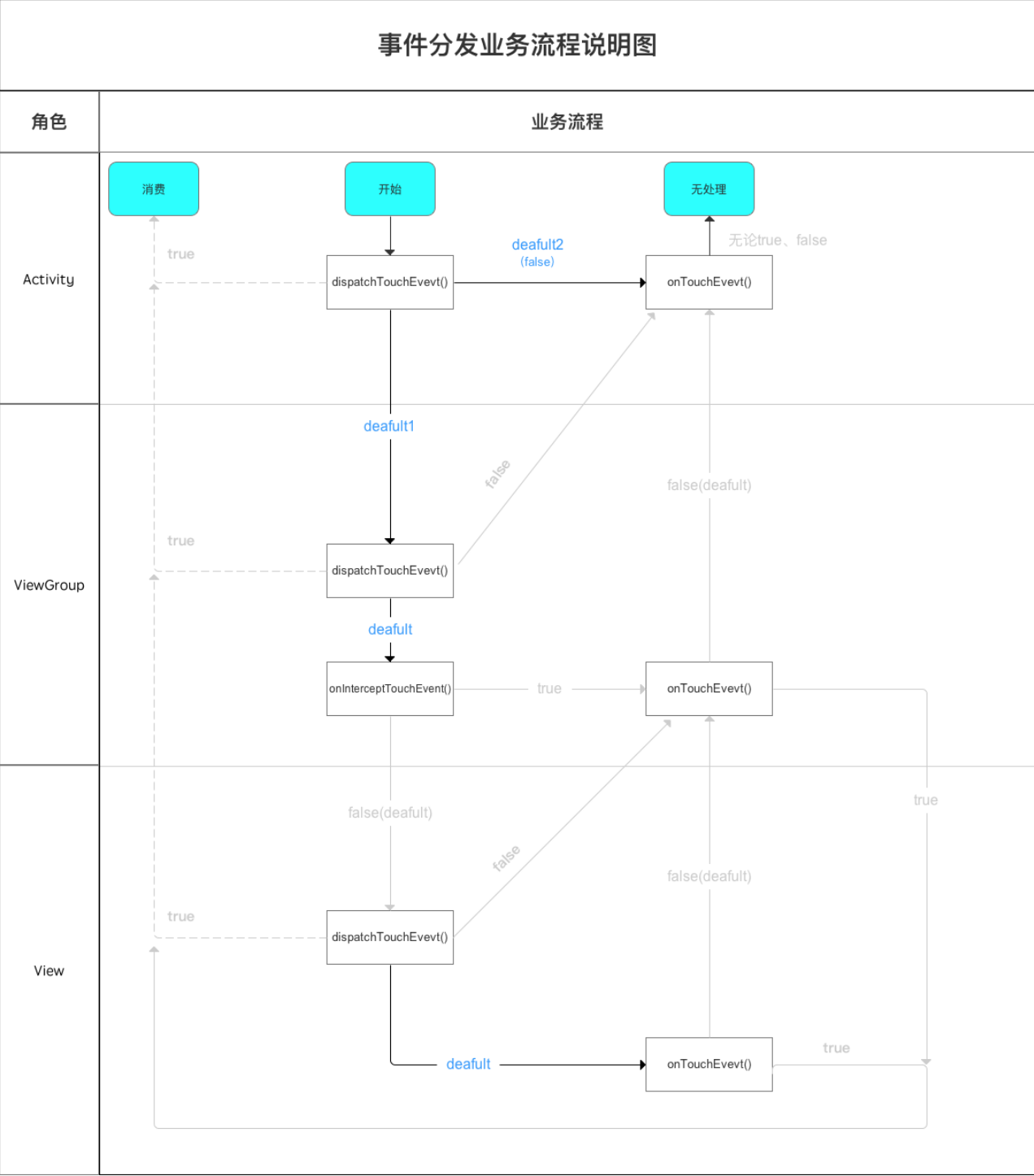
示意图

返回情况说明

情况1：默认

返回结果	具体含义	后续动作
默认	调用该方法时会直接调用如右方法	根据当前对象的不同而返回方法不同 <ul style="list-style-type: none">Activity: 调用ViewGroup.dispatchTouchEvent() / Activity.onTouchEvent()ViewGroup: 调用自身的onInterceptTouchEvent()View: 调用自身的onTouchEvent ()

示意图

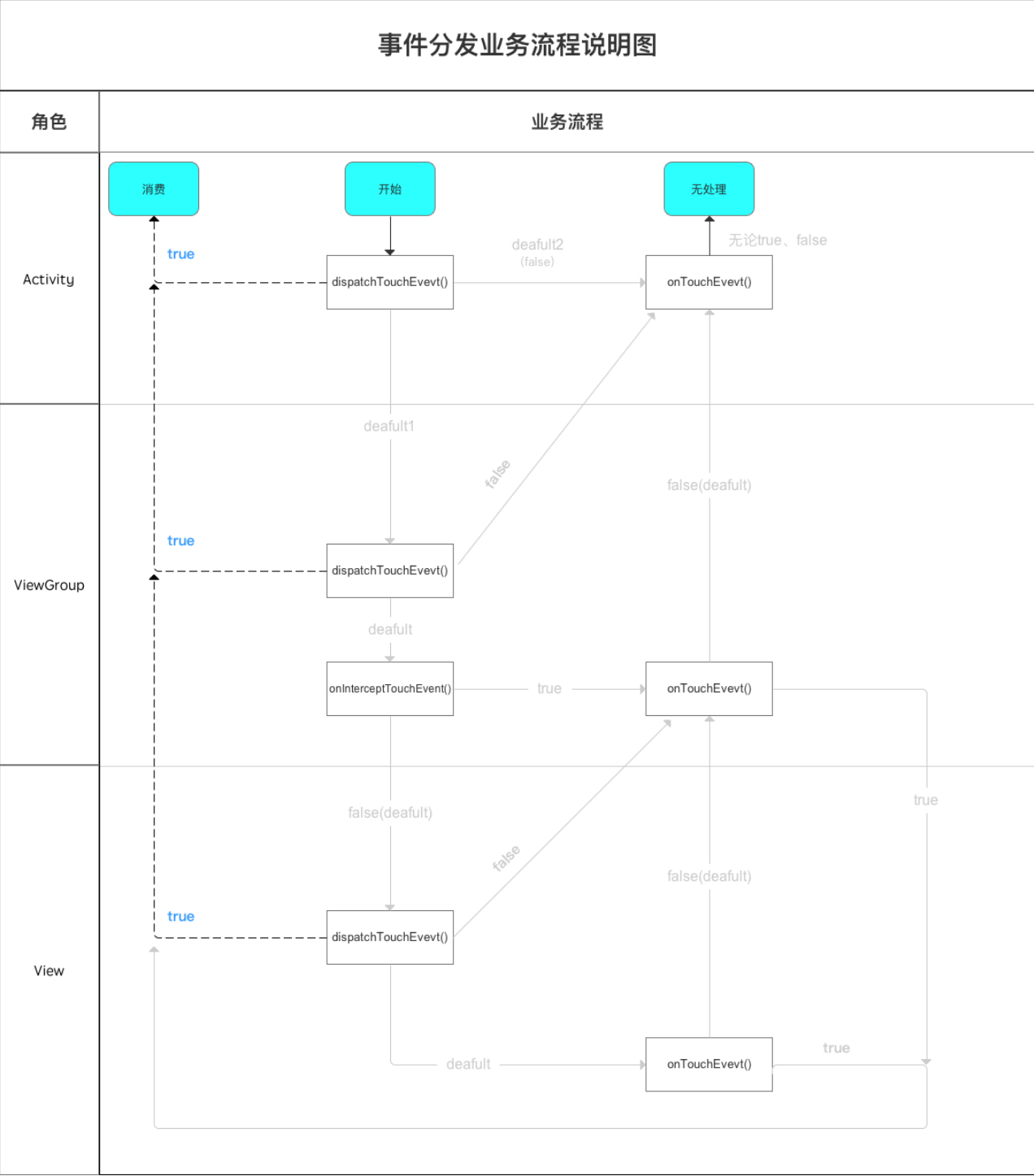


示意图

情况2：返回true

返回结果	具体含义	后续动作
true	当前事件被消费 (即事件已被View/ViewGroup接收&处理)	<ul style="list-style-type: none">事件停止分发、逐层往上返回 (若无上层返回, 则结束)后续事件会继续分发到该 View
特别注意	<ul style="list-style-type: none">注意点1: 各层dispatchTouchEvent() 返回true的情况保持一致 (图中虚线)原因: 上层dispatchTouchEvent() 的返回true情况 取决于 下层dispatchTouchEvent() 是否返回true, 如Activity.dispatchTouchEvent() 返回true的情况 = ViewGroup.dispatchTouchEvent() 返回true注意点2: 各层dispatchTouchEvent() 与 onTouchEvent()的返回情况保持一致原因: 最下层View的dispatchTouchEvent()的返回值 取决于 View.onTouchEvent()的返回值; 结合注意点1, 逐层往上返回, 从而保持一致	

示意图

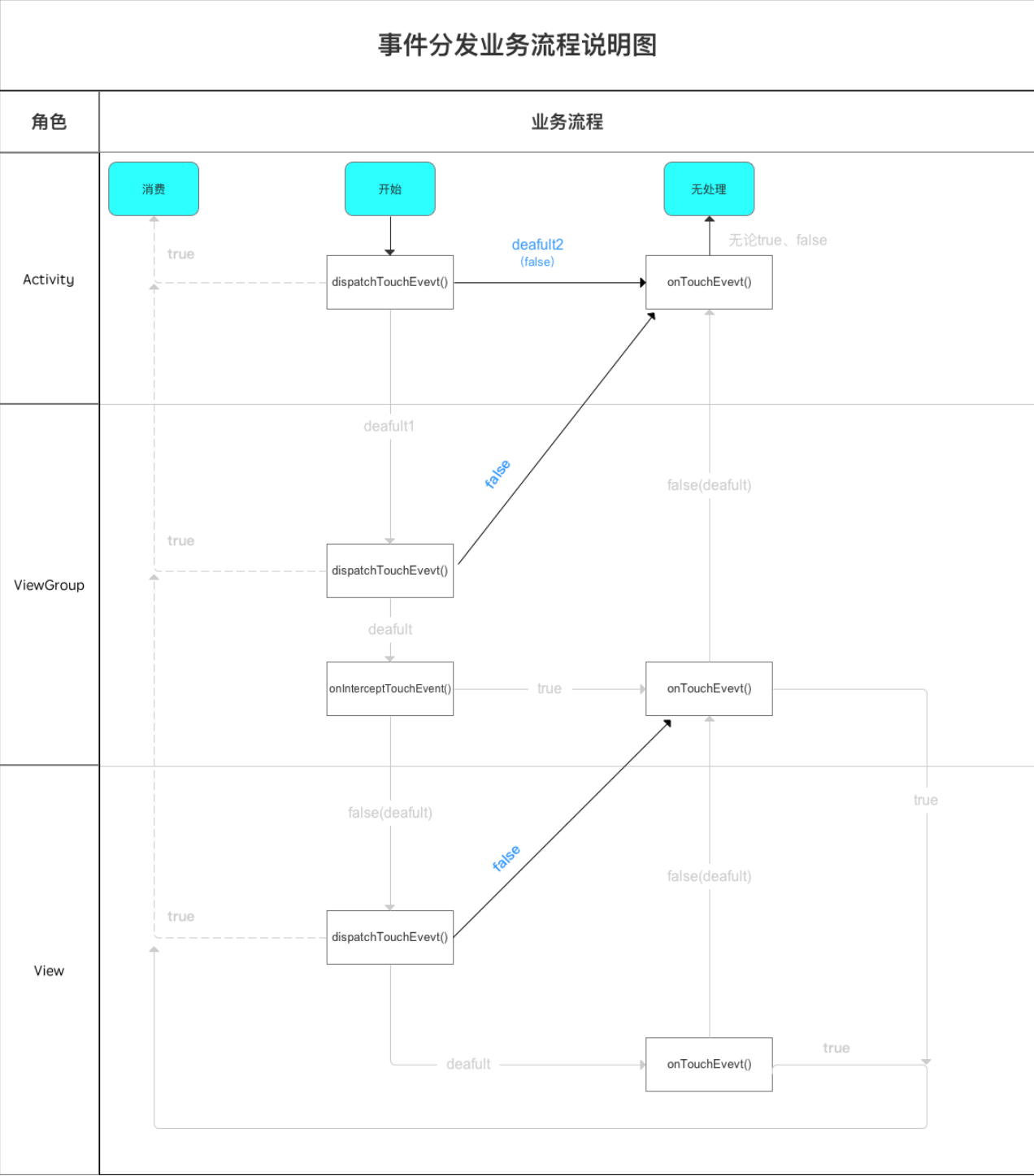


示意图

情况3：返回false

返回结果	具体含义	后续动作
false	当前事件无被消费 (即事件无被View/ViewGroup接收&处理)	<ul style="list-style-type: none">将事件回传给上层的onTouchEvent () 处理(若无上层返回, 则结束: 对于Activity, dispatchTouchEvent() 返回false 即 onTouchEvent () 返回false, 即事件无被任何View接收&处理, 故事件分发结束)当前View仍然接受此事件的其他事件 (与onTouchEvent()区别)
特别注意	<ul style="list-style-type: none">注意点1: 各层dispatchTouchEvent() 返回true的情况保持一致 (图中虚线)原因: 上层dispatchTouchEvent() 的返回true情况 取决于 下层dispatchTouchEvent() 是否返回true, 如Activity.dispatchTouchEvent() 返回true的情况 = ViewGroup.dispatchTouchEvent() 返回true注意点2: 各层dispatchTouchEvent() 与 onTouchEvent()的返回情况保持一致原因: 最下层View的dispatchTouchEvent()的返回值 取决于 View.onTouchEvent()的返回值; 结合注意点1, 逐层往上返回, 从而保持一致	

示意图



示意图

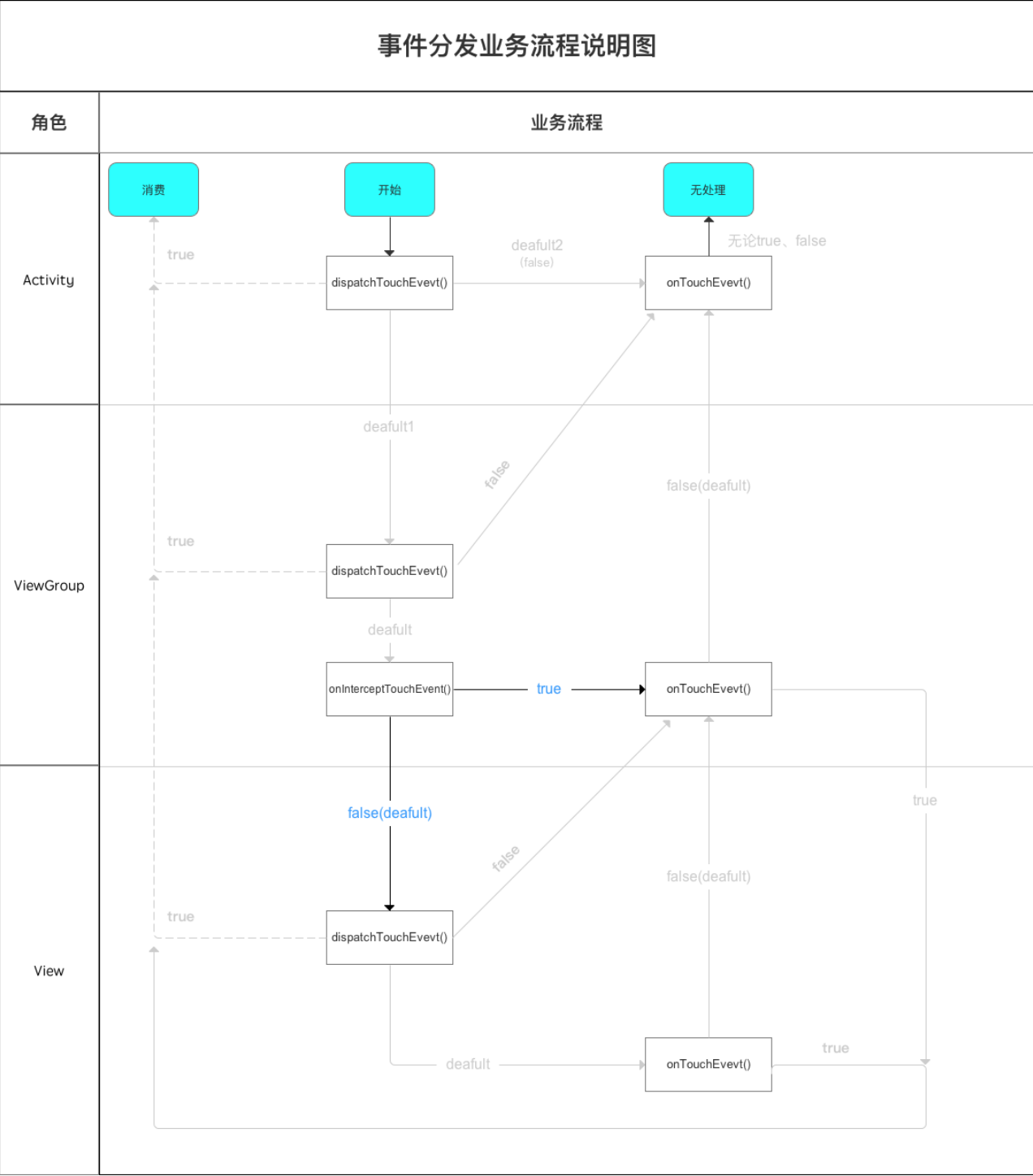
4.2 onInterceptTouchEvent()

简介

使用对象	作用	调用时刻	返回结果说明		
			返回结果	具体含义	后续动作
ViewGroup	判断是否拦截了某个事件 • 只存在于ViewGroup • 普通的View无该方法	在ViewGroup的dispatchTouchEvent()内部调用	true	当前事件被ViewGroup拦截	• 事件停止往下传递 • ViewGroup自己处理事件，调用父类super.dispatchTouchEvent()，最终执行自己的onTouchEvent()； • 同一个事件列的其他事件都直接交由该View处理；在同一个事件列中该方法不会再次被调用；
			false (default)	当前事件未被ViewGroup拦截	• 事件继续往下传递 • 事件传递到子view，调用View.dispatchTouchEvent()方法中去处理 • 当前View仍然接受此事件的其他事件 (与onTouchEvent()区别)

示意图

注：Activity、View 都无该方法



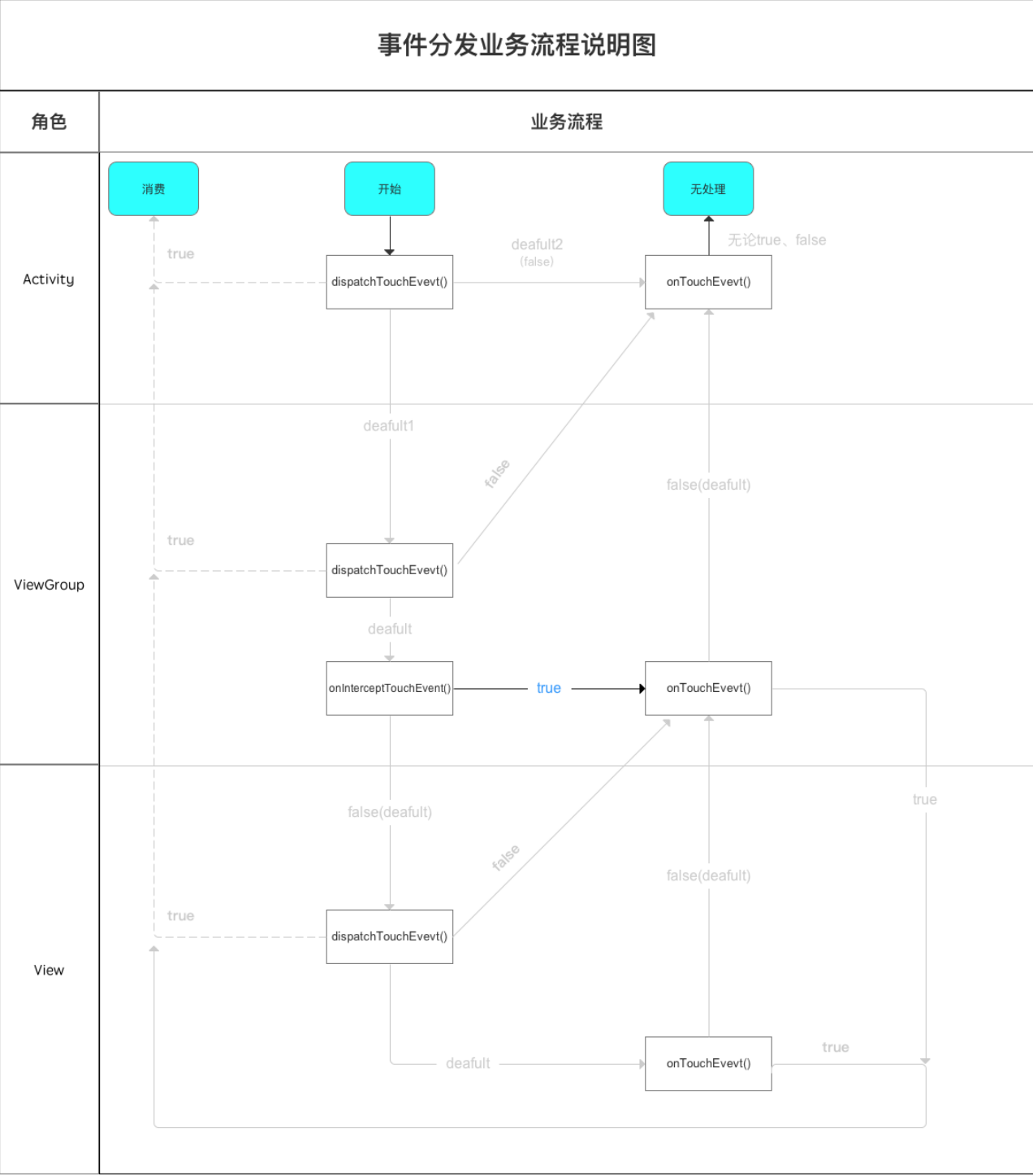
示意图

返回情况说明

情况1：true

返回结果	具体含义	后续动作
true	当前事件被ViewGroup拦截	<ul style="list-style-type: none">事件停止往下传递ViewGroup自己处理事件，调用父类super.dispatchTouchEvent()，最终执行自己的onTouchEvent()；同一个事件列的其他事件都直接交由该View处理；在同一个事件列中该方法不会再次被调用；

示意图

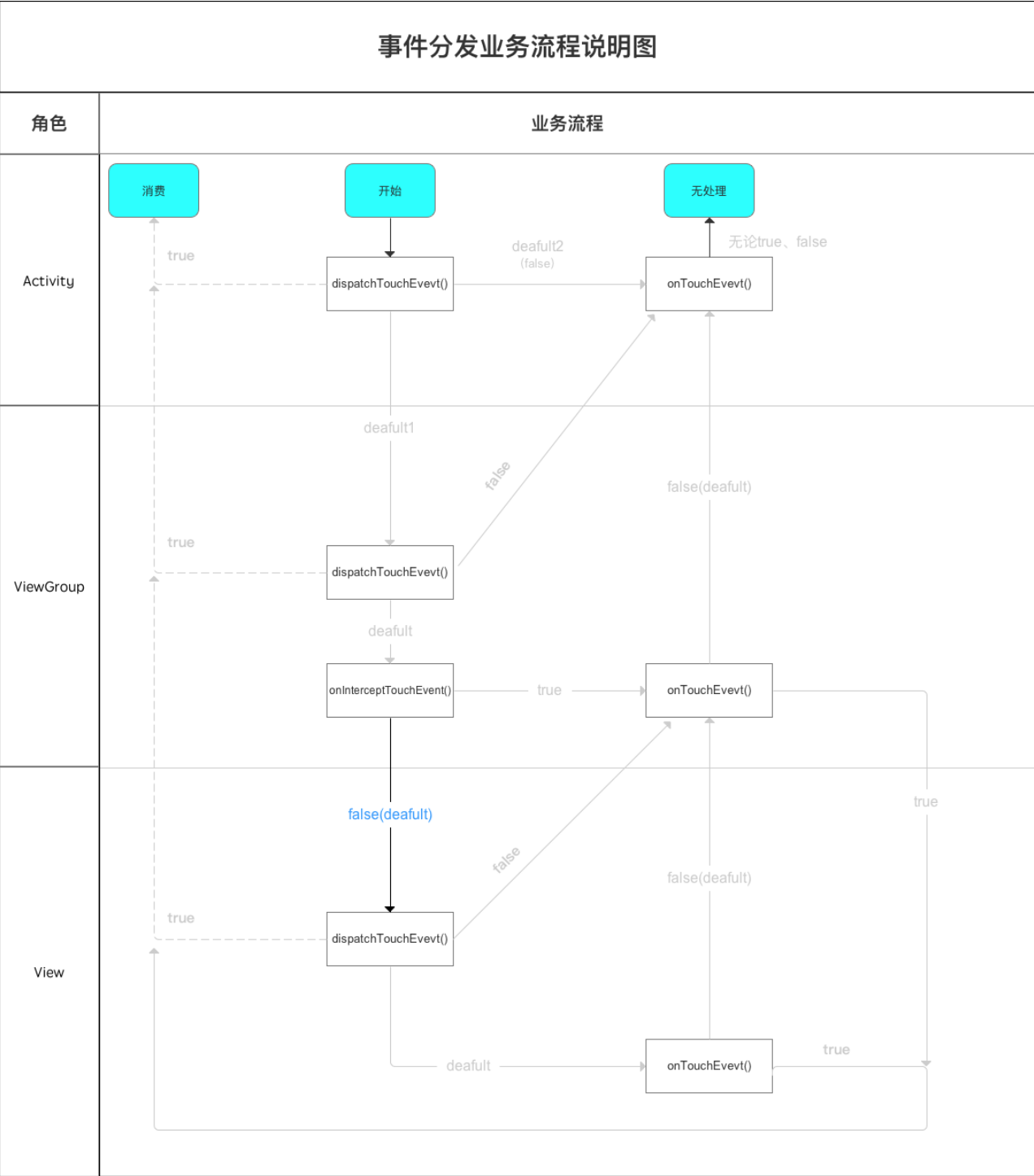


示意图

情况2：false（默认）

返回结果	具体含义	后续动作
false (default)	当前事件无被ViewGroup拦截	<ul style="list-style-type: none">事件继续往下传递事件传递到子view，调用View.dispatchTouchEvent() 方法中去处理当前View仍然接受此事件的其他事件（与onTouchEvent()区别）

示意图



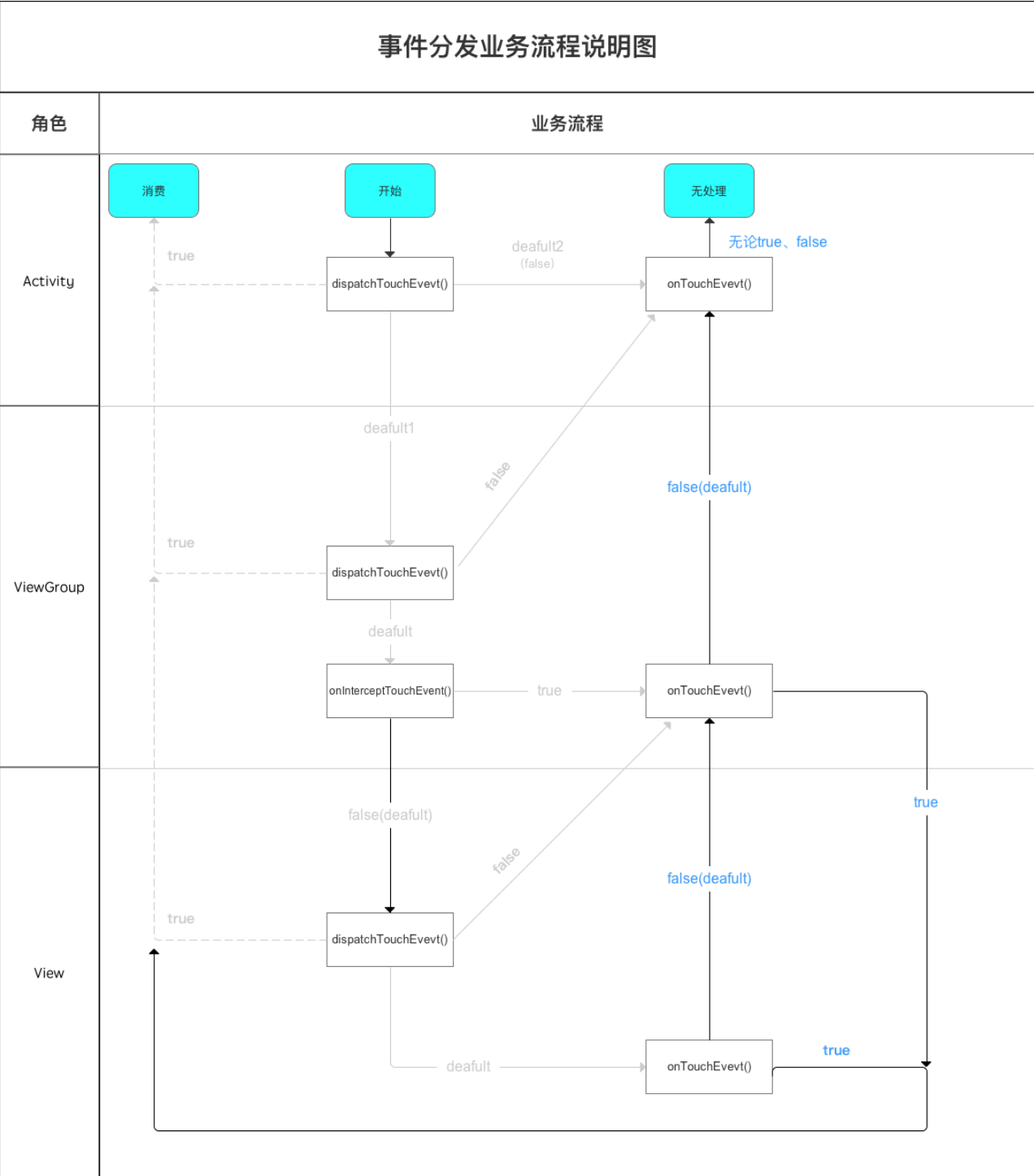
示意图

4.3 onTouchEvent()

简介

使用对象	作用	调用时刻	返回结果说明		
			返回结果	具体含义	后续动作
• Activity • ViewGroup • View	处理点击事件	在dispatchTouchEvent() 内部调用	true (处理)	当前使用对象处理了当前事件 (使用对象指: Activity、View、Group)	• 事件停止分发、逐层往上返回 (若无上层返回, 则结束) • 后续事件序列让其处理;
			false (不处理)	当前使用对象无处理当前事件 (使用对象指: Activity、View、Group)	• 将事件向上传给上层的onTouchEvent()处理 (若无上层返回, 则结束) • 当前View不再接受此事件的其他事件 (与dispatchTouchEvent () 、onInterceptTouchEvent () 的区别)

示意图



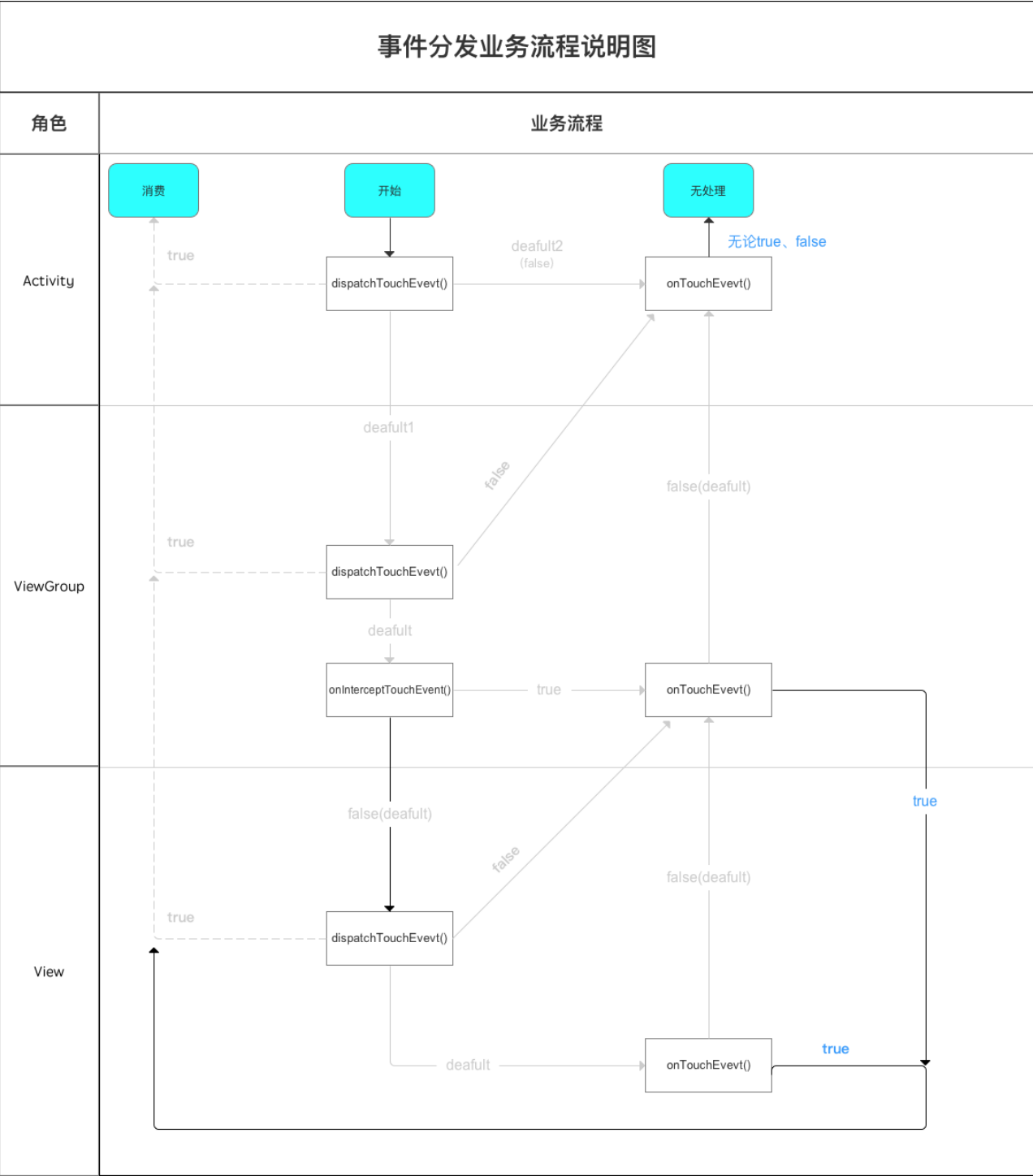
示意图

返回情况说明

情况1：返回true

返回结果	具体含义	后续动作
true (处理)	当前使用对象处理了当前事件 (使用对象指: Activity、View、Group)	<ul style="list-style-type: none">事件停止分发、逐层往dispatchTouchEvent() 返回 (对于Activity: 先返回当前dispatchTouchEvent() ; 由于无上层, 故结束)后续事件序列让其处理;
特别注意	<ul style="list-style-type: none">注意点1: 各层dispatchTouchEvent() 与 onTouchEvent()的返回情况保持一致原因: 最下层View的dispatchTouchEvent()的返回值 取决于 View.onTouchEvent()的返回值; 逐层往上返回, 保持一致	

示意图

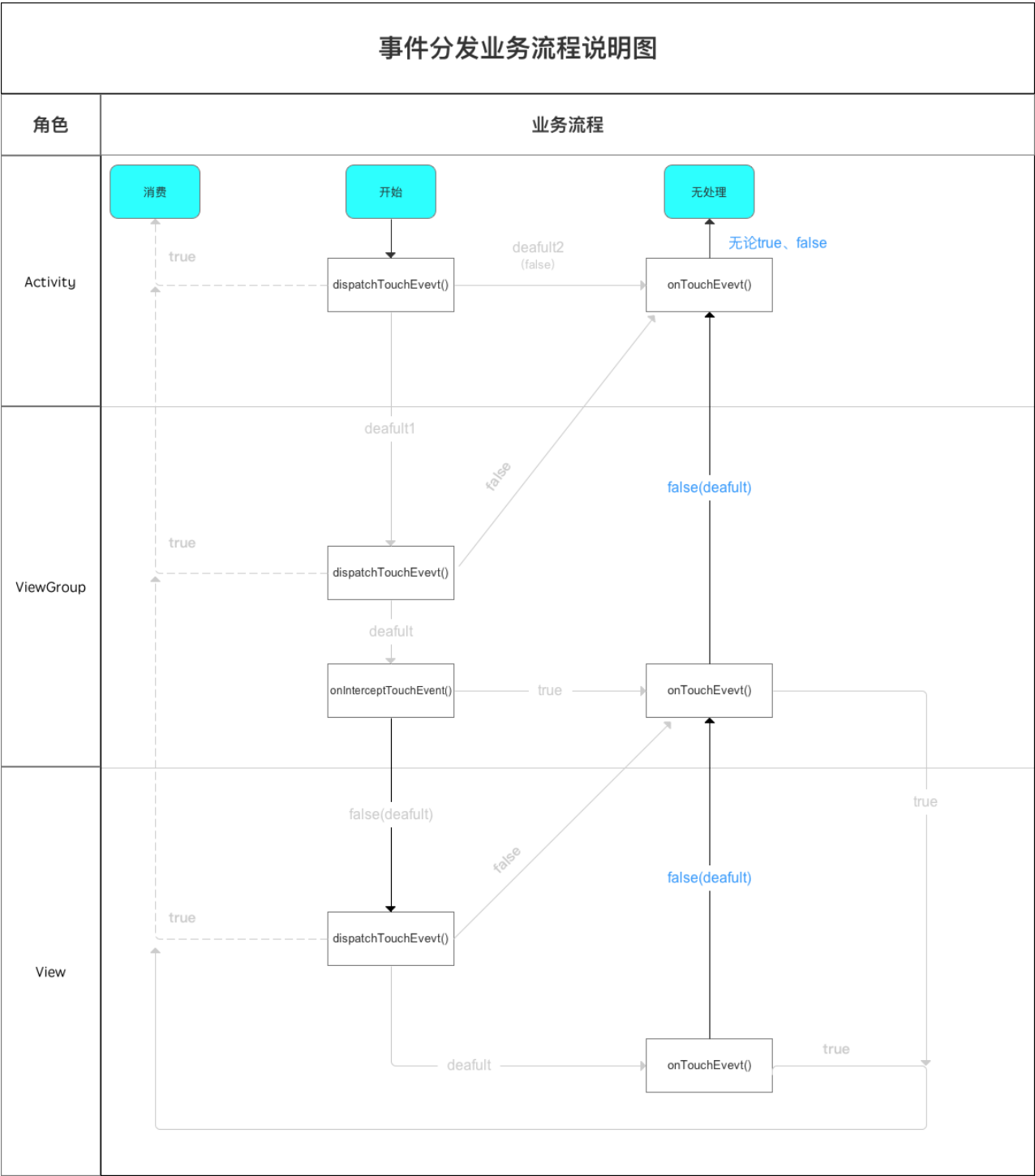


示意图

情况2：返回false（default）

返回结果	具体含义	后续动作
false (不处理)	当前使用对象无处理当前事件 (使用对象指：Activity、View、Group)	<ul style="list-style-type: none">将事件向上传给上层的onTouchEvent()处理 (对于Activity：由于无上层，故结束)当前View不再接受此事件的其他事件 (与dispatchTouchEvent () 、onInterceptTouchEvent () 的区别)
特别注意	<ul style="list-style-type: none">注意点1：各层dispatchTouchEvent() 与 onTouchEvent()的返回情况保持一致原因：最下层View的dispatchTouchEvent()的返回值 取决于 View.onTouchEvent()的返回值；逐层往上返回，保持一致	

示意图



示意图

4.4 三者关系

下面，我用一段伪代码来阐述上述3个方法的关系 & 事件传递规则

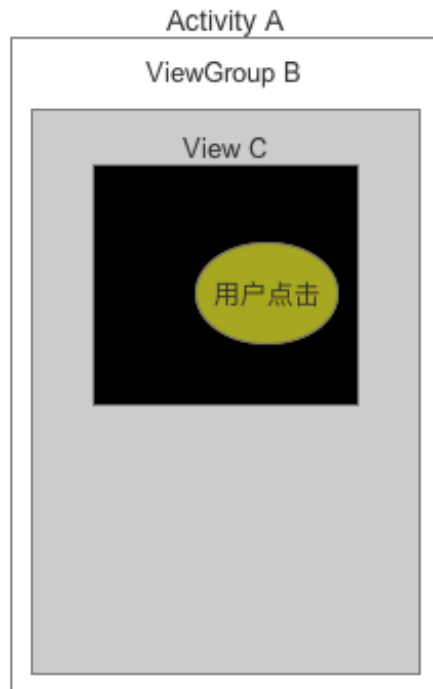
```
public boolean dispatchTouchEvent(MotionEvent ev) {  
  
    boolean consume = false;  
  
    if (onInterceptTouchEvent(ev)) {  
  
        consume = onTouchEvent (ev) ;  
  
    } else {  
  
  
  
        consume = child.dispatchTouchEvent (ev) ;  
    }  
  
    return consume;  
  
}
```

5. 常见的事件分发场景

下面，我将通过实例说明**常见的事件传递情况 & 流程**

5.1 背景描述

讨论的布局如下：



最外层：Activiy A，包含两个子View：ViewGroup B、View C
中间层：ViewGroup B，包含一个子View：View C
最内层：View C
示意图

情景

用户先触摸到屏幕上 **View C** 上的某个点（图中黄区）

| **Action_DOWN** 事件在此处产生

2. 用户移动手指
3. 最后离开屏幕

5.2 一般的事件传递情况

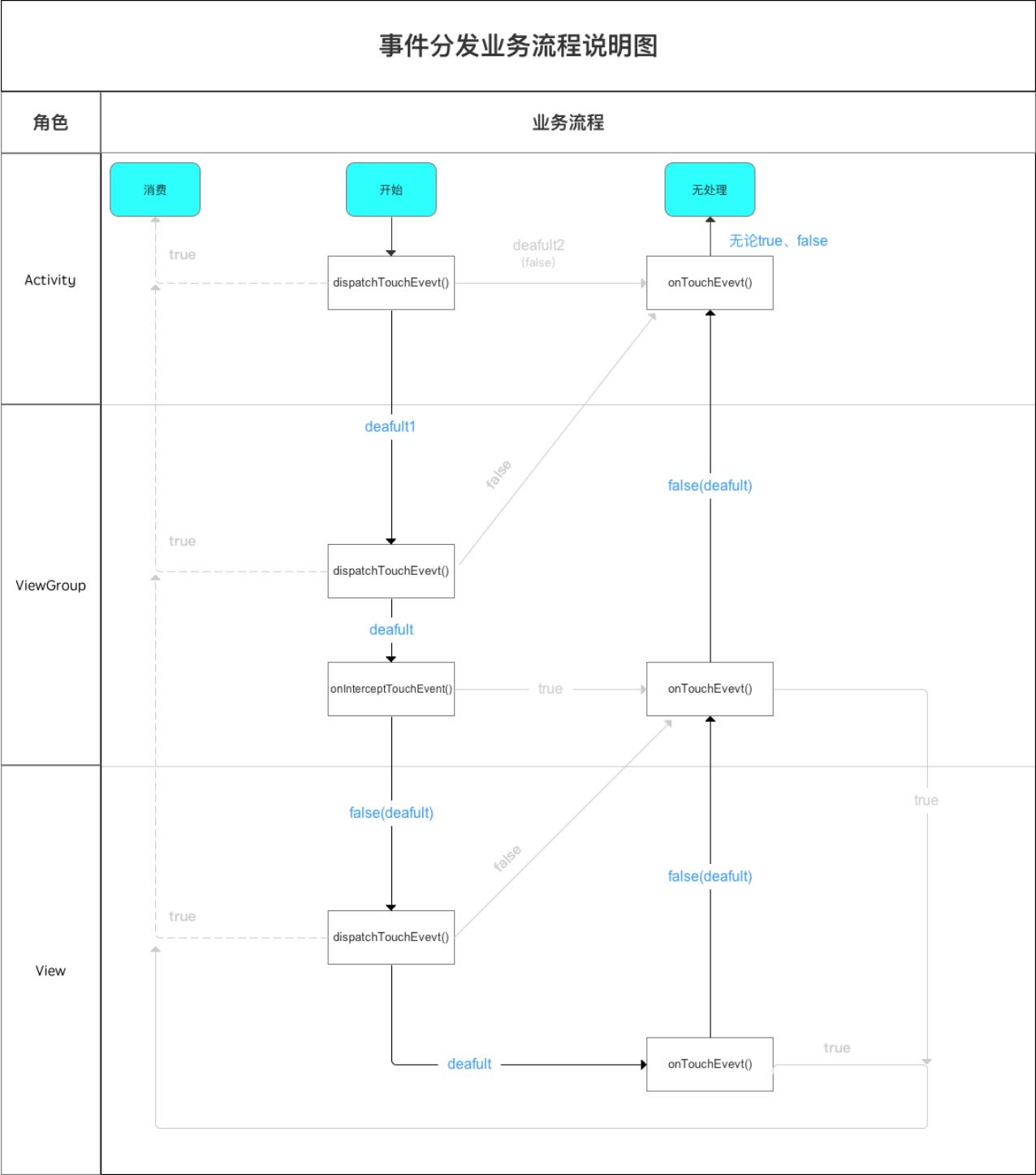
一般的事件传递场景有：

- 默认情况
- 处理事件
- 拦截 **DOWN** 事件
- 拦截后续事件（**MOVE**、**UP**）

场景1：默认

- 即不对控件里的方法（**dispatchTouchEvent()**、**onTouchEvent()**、**onInterceptTouchEvent()**）进行重写或更改返回值
- 那么调用的是这3个方法的默认实现：调用下层的方法 & 逐层返回

- 事件传递情况：（呈 U 型）
 - 从上往下调用dispatchTouchEvent()
 - Activity A ->> ViewGroup B ->> View C
 - 从下往上调用onTouchEvent()
 - View C ->> ViewGroup B ->> Activity A



示意图

注：虽然 `ViewGroup B` 的 `onInterceptTouchEvent()` 对 `DOWN` 事件返回了 `false`，但后续的事件（`MOVE、UP`）依然会传递给它。
这一点与 `onTouchEvent()` 的行为是不一样的：不再传递 & 接收该事件列的其他事件

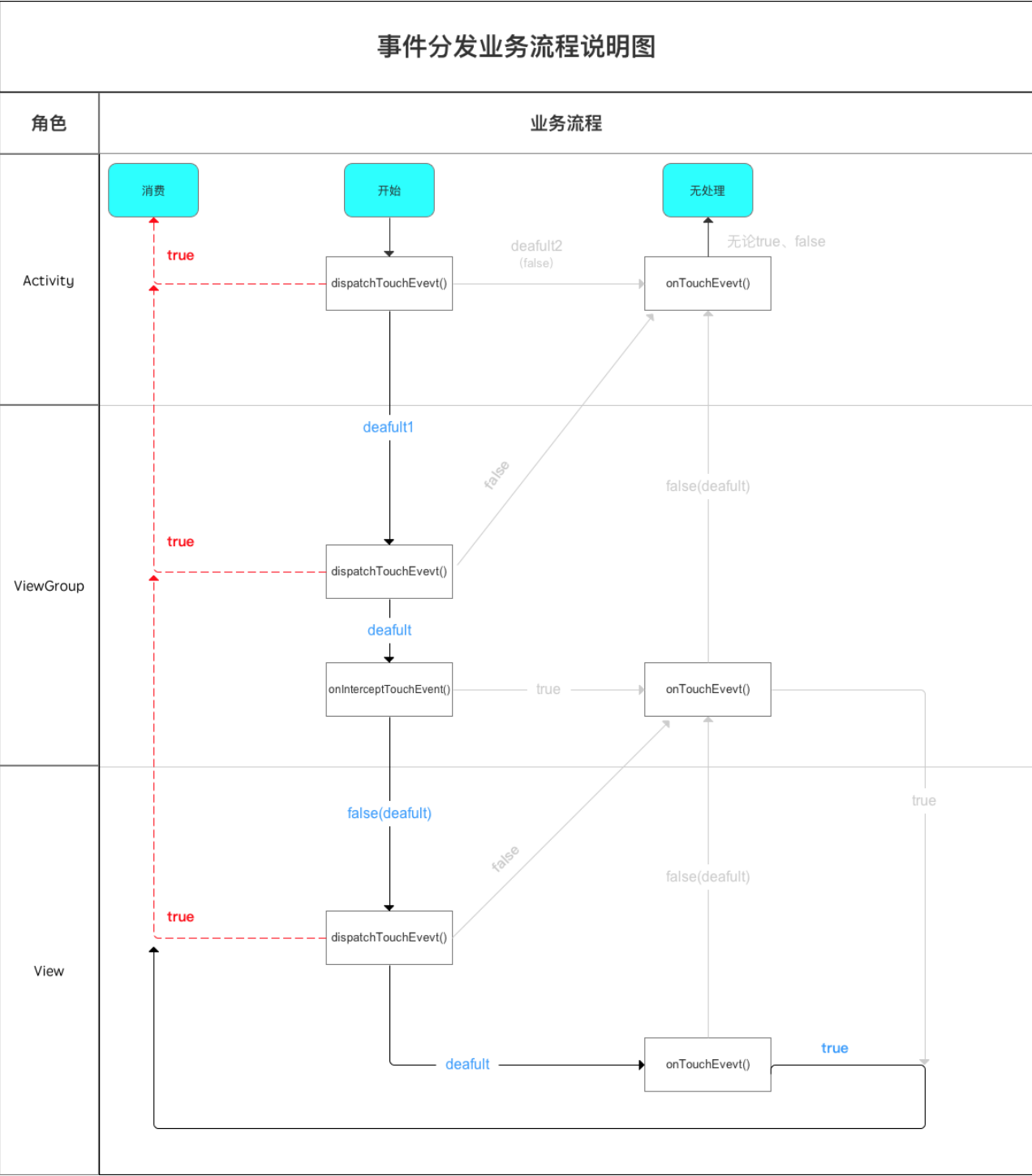
场景2：处理事件

设 `View C` 希望处理该点击事件，即：设置 `View C` 为可点击的（`Clickable`）或复写其 `onTouchEvent()` 返回 `true`

最常见的：设置 `Button` 按钮来响应点击事件

事件传递情况：（如下图）

- `DOWN` 事件被传递给C的 `onTouchEvent` 方法，该方法返回 `true`，表示处理该事件
- 因为 `View C` 正在处理该事件，那么 `DOWN` 事件将不再往上传递给 `ViewGroup B` 和 `Activity A` 的 `onTouchEvent()`；
- 该事件列的其他事件（`Move、Up`）也将传递给 `View C` 的 `onTouchEvent()`



示意图

会逐层往 `dispatchTouchEvent()` 返回，最终事件分发结束

场景3：拦截DOWN事件

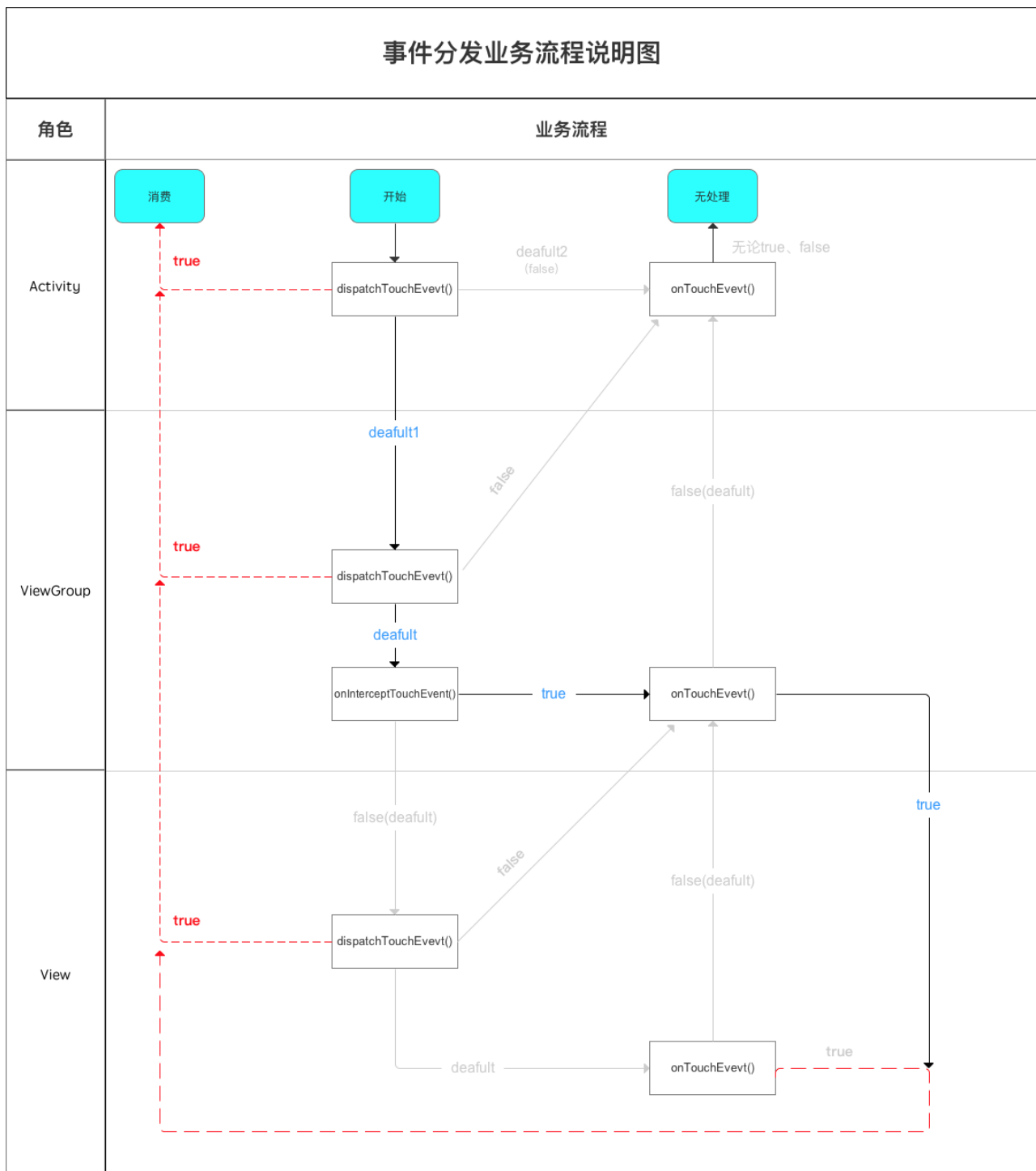
假设 `ViewGroup B` 希望处理该点击事件，即 `ViewGroup B` 复写了 `onInterceptTouchEvent()` 返回 `true`、`onTouchEvent()` 返回 `true`
事件传递情况：（如下图）

- `DOWN` 事件被传递给 `ViewGroup B` 的 `onInterceptTouchEvent()`，该方法返回 `true`，表示拦截该事件，即自己处理该事件（事件不再往下传递）

- 调用自身的 `onTouchEvent()` 处理事件（`DOWN` 事件将不再往上传递给 `Activity A` 的 `onTouchEvent()`）
- 该事件列的其他事件（`Move、Up`）将直接传递给 `ViewGroup B` 的 `onTouchEvent()`

注：

1. 该事件列的其他事件（`Move、Up`）将不会再传递给 `ViewGroup B` 的 `onInterceptTouchEvent()`（）；因：该方法一旦返回一次 `true`，就再也不会被调用
2. 逐层往 `dispatchTouchEvent()` 返回，最终事件分发结束



示意图

场景4：拦截DOWN的后续事件

结论

- 若 `ViewGroup` 拦截了一个半途的事件（如 `MOVE` ），该事件将会被系统变成一个 `CANCEL` 事件 & 传递给之前处理该事件的子 `View` ；
- 该事件不会再传递给 `ViewGroup` 的 `onTouchEvent()`
- 只有再到来的事件才会传递到 `ViewGroup` 的 `onTouchEvent()`

场景描述

`ViewGroup B` 无拦截 `DOWN` 事件（还是 `View C` 来处理 `DOWN` 事件），但它拦截了接下来的 `MOVE` 事件

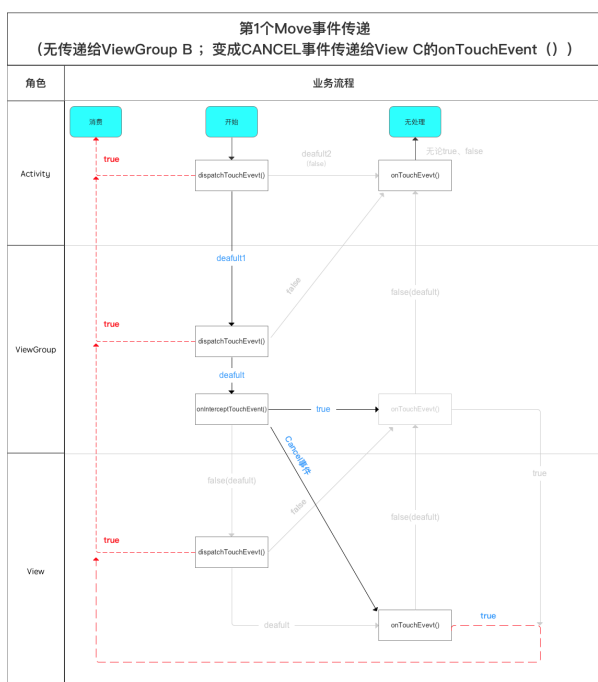
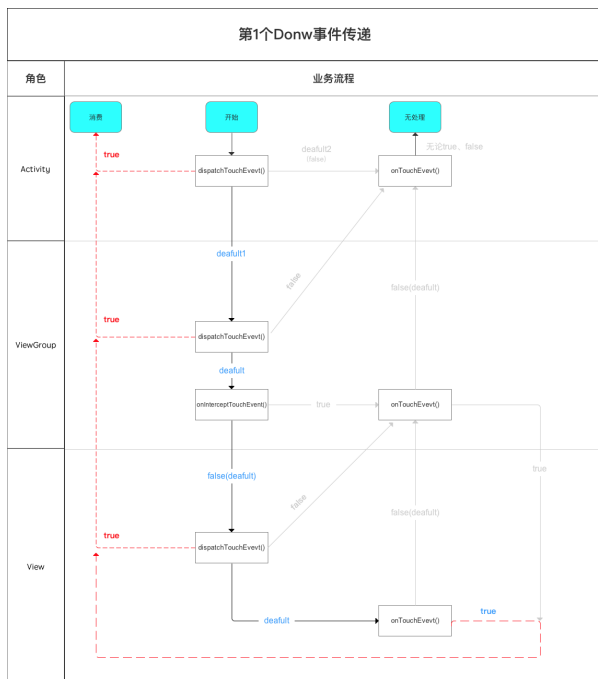
即 `DOWN` 事件传递到 `View C` 的 `onTouchEvent ()` ，返回了 `true`

实例讲解

- 在后续到来的`MOVE`事件，`ViewGroup B` 的 `onInterceptTouchEvent ()` 返回 `true` 拦截该 `MOVE` 事件，但该事件并没有传递给 `ViewGroup B` ；这个 `MOVE` 事件将会被系统变成一个 `CANCEL` 事件传递给 `View C` 的 `onTouchEvent ()`
- 后续又来了一个 `MOVE` 事件，该 `MOVE` 事件才会直接传递给 `ViewGroup B` 的 `onTouchEvent()`

后续事件将直接传递给 `ViewGroup B` 的 `onTouchEvent()` 处理，而不会再传递给 `ViewGroup B` 的 `onInterceptTouchEvent ()` ，因该方法一旦返回一次`true`，就再也不会被调用了。

`View C` 再也不会收到该事件列产生的后续事件



示意图

至此，关于 **Android** 常见的事件传递情况 & 流程已经讲解完毕。

6. 额外知识

6.1 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

从上面对事件分发机制分析知：

- dispatchTouchEvent()、onTouchEvent() 消费事件、终结事件传递（返回true）
- 而onInterceptTouchEvent 并不能消费事件，它相当于是一个分叉口起到分流导流的作用，对后续的ACTION_MOVE和ACTION_UP事件接收起到非常大的作用

请记住：接收了ACTION_DOWN事件的函数不一定能收到后续事件（ACTION_MOVE、ACTION_UP）

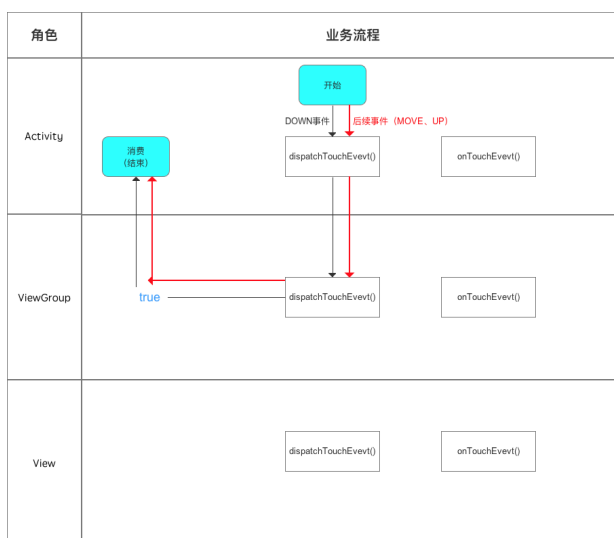
这里给出ACTION_MOVE和ACTION_UP事件的传递结论：

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



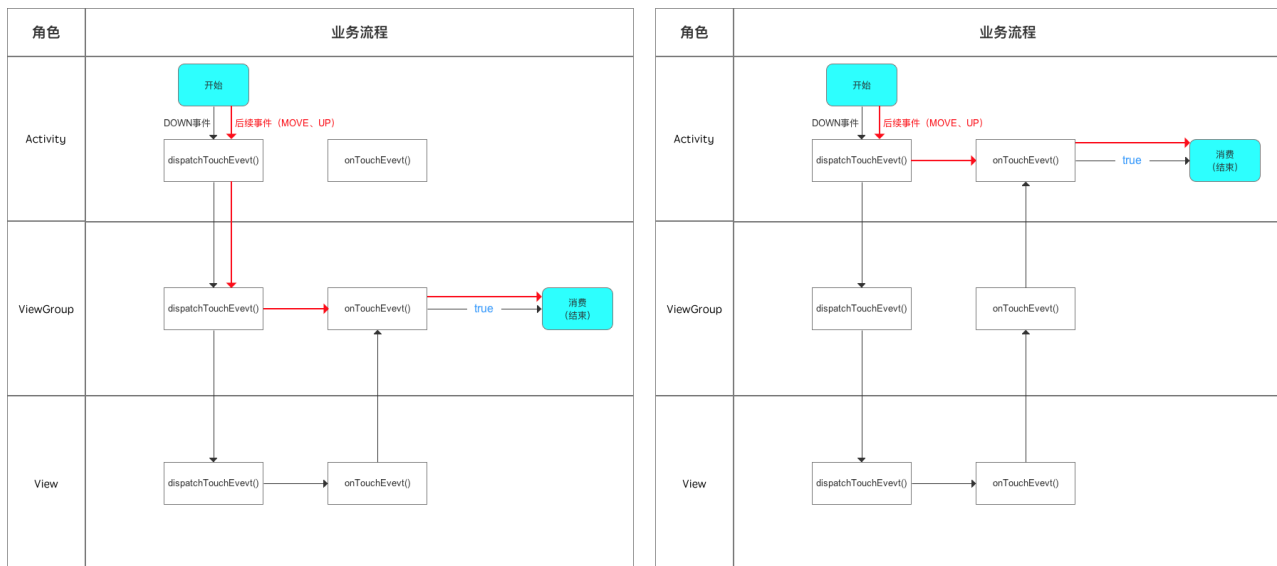
流程讲解

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



流程讲解

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

```
mOnTouchListener != null && (mViewFlags & ENABLED_MASK) == ENABLED &&
    mOnTouchListener.onTouch(this, event)
```

7. 总结

- 通过阅读本文，相信您已经可以全面了解 `Android` 事件分发机制
- 与 `Android` 事件分发最相关的知识：**自定义View**系列文章
 - (1) 自定义View基础 - 最易懂的自定义View原理系列
 - (2) 自定义View Measure过程 - 最易懂的自定义View原理系列
 - (3) 自定义View Layout过程 - 最易懂的自定义View原理系列
 - (4) 自定义View Draw过程 - 最易懂的自定义View原理系列
- 接下来我将继续介绍与 `Android` 事件分发最相关的知识：**自定义View**，感兴趣的同学可以继续关注本人运营的 `Wechat Public Account`：
- 我想给你们介绍一个与众不同的Android微信公众号（福利回赠）

- [我想邀请您和我一起写Android \(福利回赠\)](#)
-

请点赞！因为你们的赞同/鼓励是我写作的最大动力！

相关文章阅读

[Android开发：最全面、最易懂的Android屏幕适配解决方案](#)

[Android开发：史上最全的Android消息推送解决方案](#)

[Android开发：最全面、最易懂的WebView详解](#)


[Android开发：JSON简介及最全面解析方法!](#)

[Android四大组件：Service服务史上最全面解析](#)

[Android四大组件：BroadcastReceiver史上最全面解析](#)

欢迎关注[Carson_Ho](#)的简书！

不定期分享关于**安卓开发**的干货，追求**短、平、快**，但**却不缺深度**。

The background of the slide features a blurred image of a workspace. It includes a silver laptop with an Apple logo, a white keyboard, and a tablet. The tablet displays the text "DRAG YOUR SCREEN HERE" and a yellow Android robot icon. The overall aesthetic is clean and modern, with a soft, out-of-focus effect.

安卓开发 知识总结

Carson-Ho