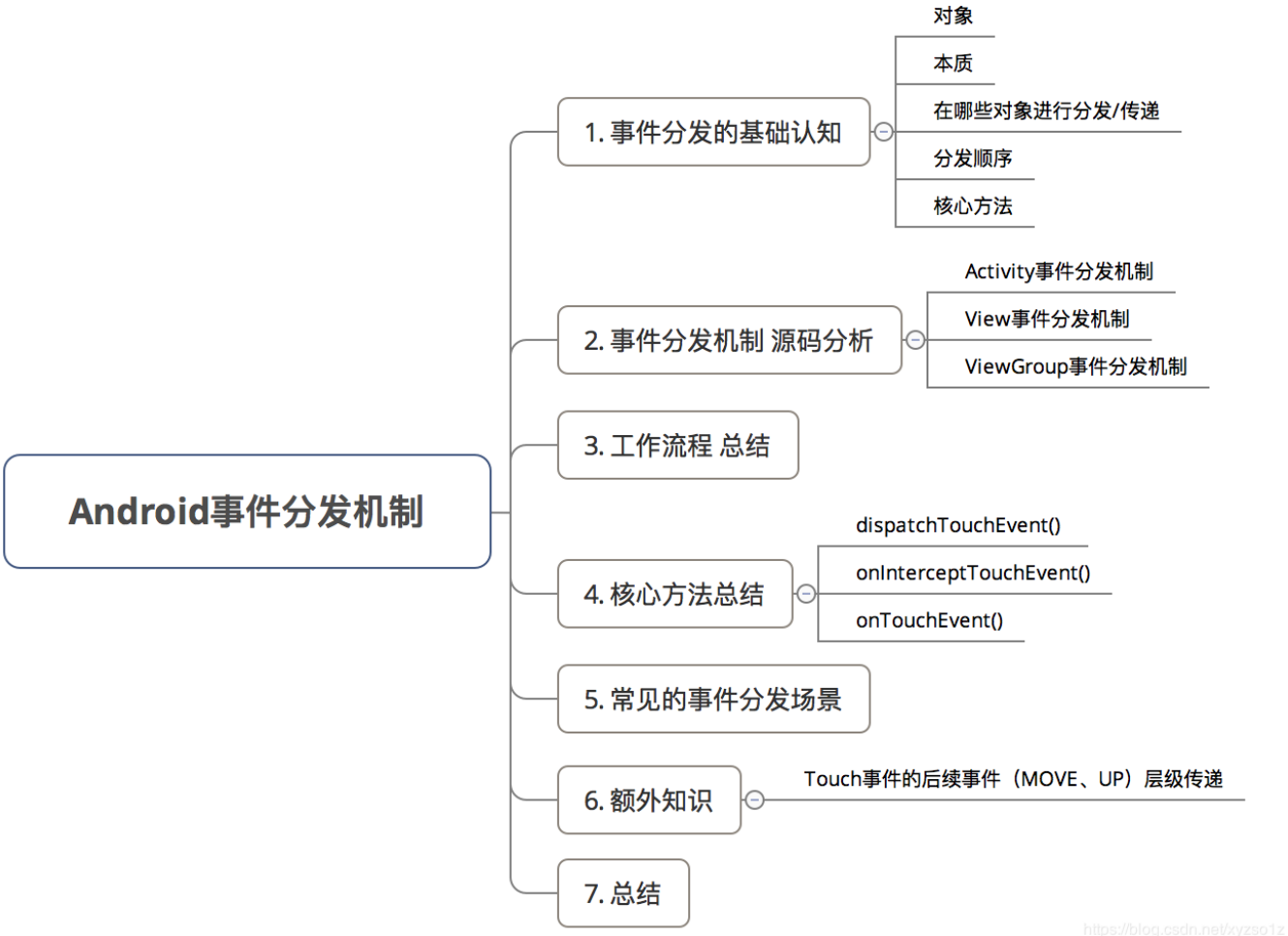
Android事件分发机制详解

原创 xyzso1z 最后发布于2019-05-04 17:11:00 阅读数 49 收藏 1 编辑 展开

查看Android总结专题

以下内容转自Android事件分发机制详解：史上最全面、最易懂，感谢Carson_Ho的分享。

1.目录



2 基础知识

2.1 事件分发的对象是谁

答：点击事件(Touch事件)

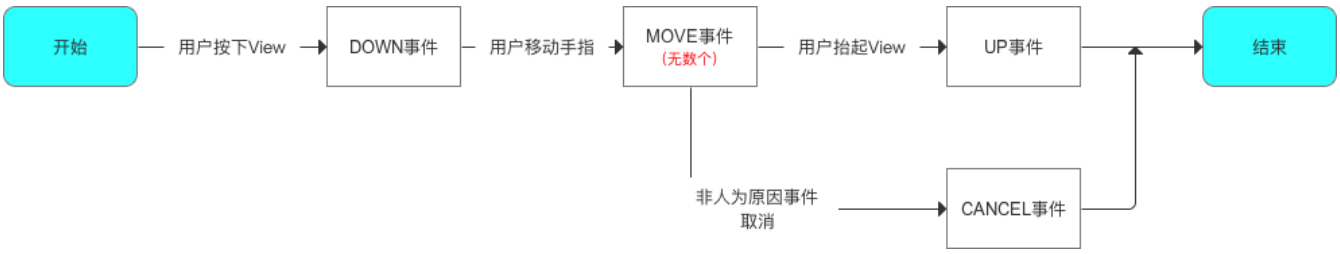
- 定义
  当事件触摸屏幕时（`View`或`ViewGroup`的派生控件）,将产生点击事件（`Touch`事件）

`Touch`事件的相关细节（发生触发的位置、时间等）被封装成`MotionEvent`对象

- 事件类型

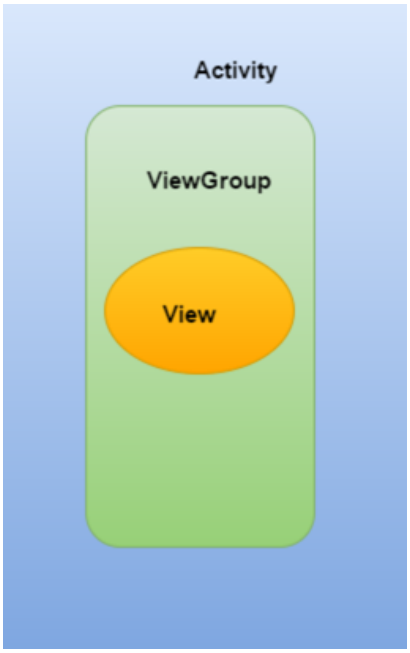
事件类型	具体动作
MotionEvent.ACTION_DOWN	按下View（所有事件的开始）
MotionEvent.ACTION_UP	抬起View（与DOWN对应）
MotionEvent.ACTION_MOVE	滑动View
MotionEvent.ACTION_CANCEL	结束事件（非人为原因）

- 特别说明
  从手指接触屏幕到手指离开屏幕，这个过程产生了一系列的事件
  注：一般情况下，事件都是从`DOWN`事件开始、`UP`事件结束，中间有无数个`MOVE`事件，如图：



即一个点击事件（`MotionEvent`）产生后，系统需要把这个具体的事件传递给一个具体的`View`去处理。
2.2 事件分发的本质
答:将点击事件（`MotionEvent`）传递到某个具体的`View`和处理的整个过程
即事件传递的过程=分发过程
2.3 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



类型	简介	备注

类型	简介	备注
Activity	控制生命周期和处理事件	* 统筹视图的添加和显示; * 通过其它回掉方法与`Window`、`View`交互
View	所有UI组建的基类	一般`Button`、`TextView`等控件都是继承父类`View`
ViewGroup	一组View的集合	* 其本身也是`View`的子类; *是`Android`所有布局的父类; *区别于普通`View`:`ViewGroup`实际上也是一个`View`,只是可包含多个子`View`和定义布局参数

2.4事件分发的顺序

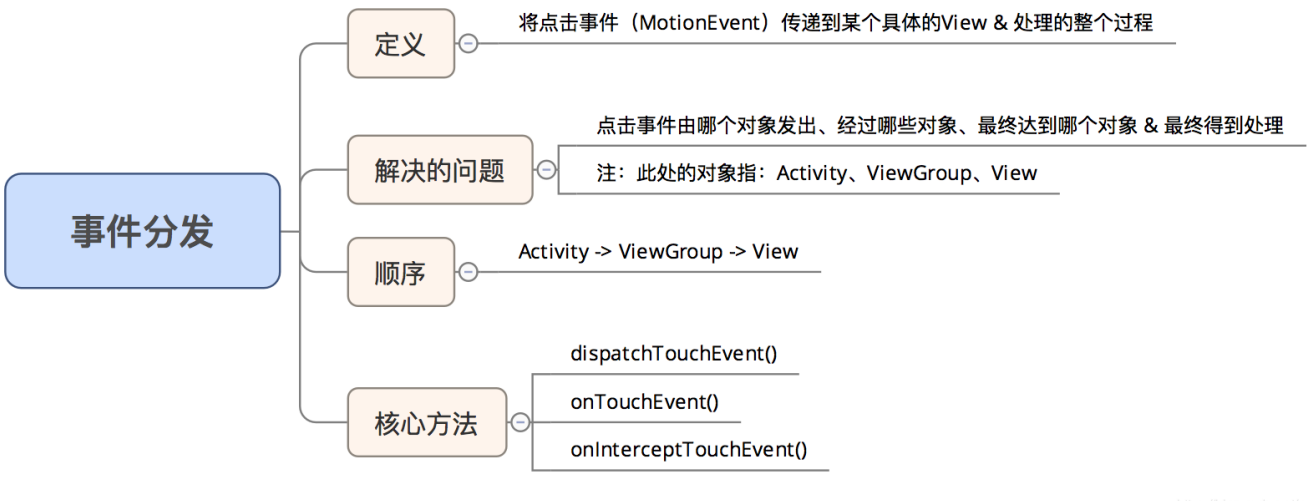
答：事件传递的顺序：`Activity` ——> `ViewGroup` ——> `View`

即：一个点击事件发生后，事件先传到`Activity`、再传到`ViewGroup`、最终再传到`View`

2.5 事件分发过程由哪些方法协作完成

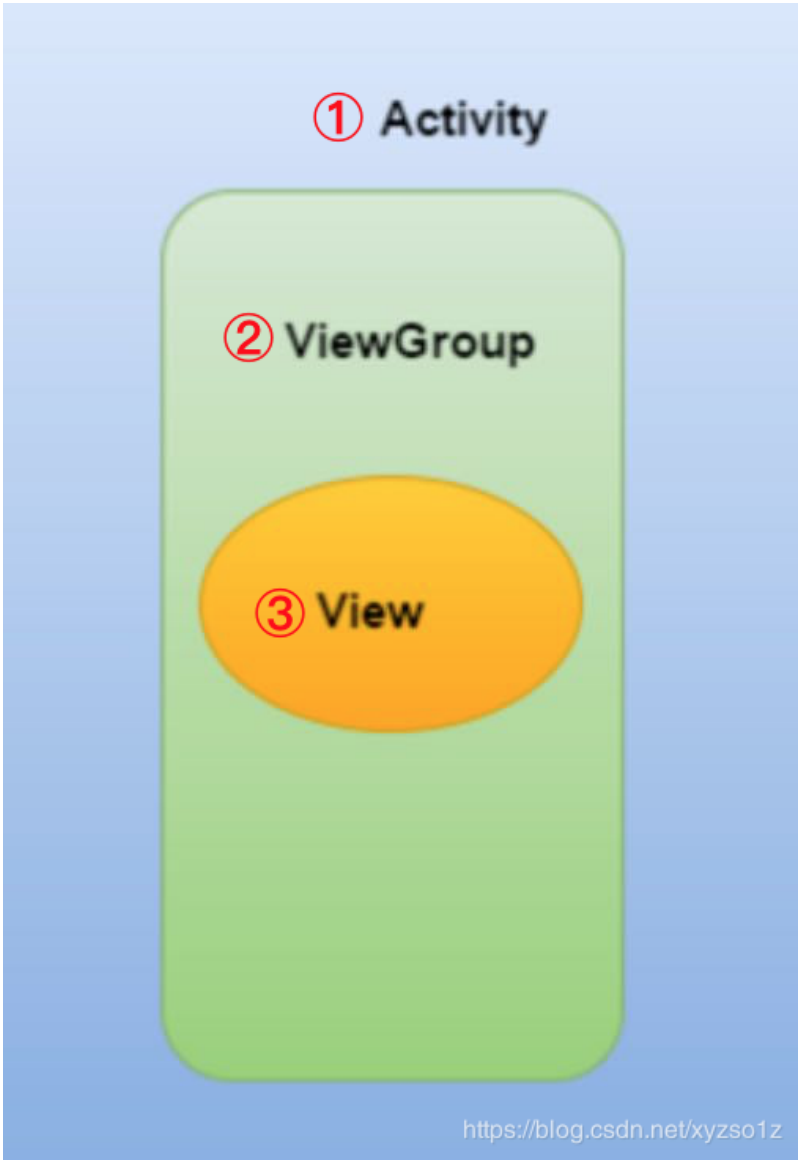
答：`dispatchTouchEvent()`、`onInterceptTouchEvent()`和`onTouchEvent()`

方法	作用	调用时刻
dispatchTouchEvent()	分发（传递）点击事件	当点击事件能够传递给当前View时，i
onTouchEvent()	处理点击事件	在dispatchTouchEvent()内部调用
onInterceptTouchEvent()	判断是否拦截某个事件(只存在于ViewGroup、普通的View无该方法)	在ViewGroup的dispatchTouchEvent(



3.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个点击事件后，事件先传到Activity、再传到ViewGroup、最终传到View。



- 从上可知，要想充分理解Activity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，通过源码，全面解析事件分发机制
  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。

3.1 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

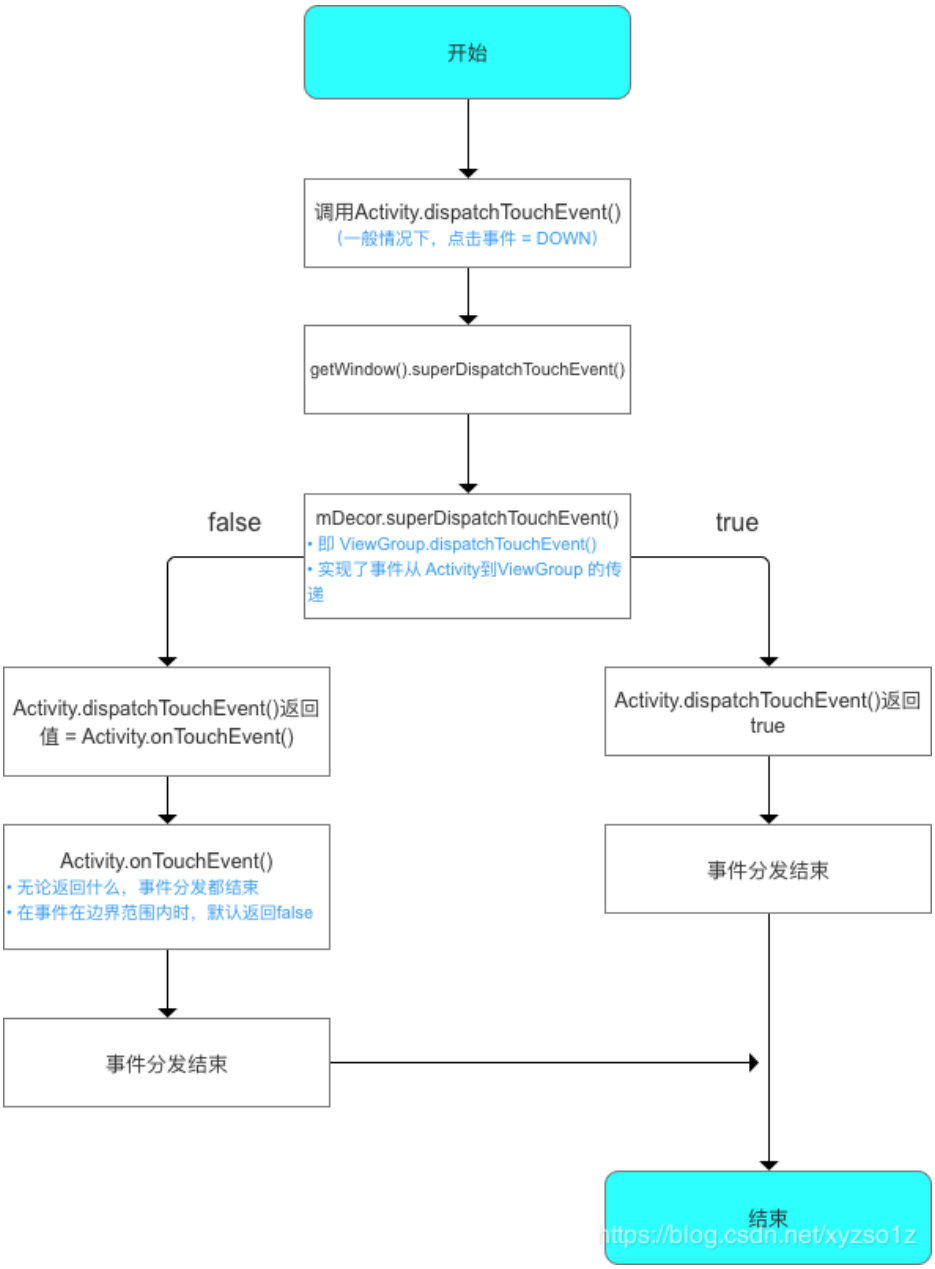
3.1.1源码分析

```java
/**
  * 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
                // 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
```

3.1.2 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

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
dispatchTouchEvevt()	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.onTouchEvevt()
		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.dispatchTouchEvevt()返回true • onTouchEvevt() 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 （即事件无被View/ViewGroup接收&处理）	onTouchEvevt() 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）
onTouchEvevt()	ViewGroup.dispatchTouchEvevt()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）	

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。

3.2 ViewGroup事件的分发机制从

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

3.2.1源码分析

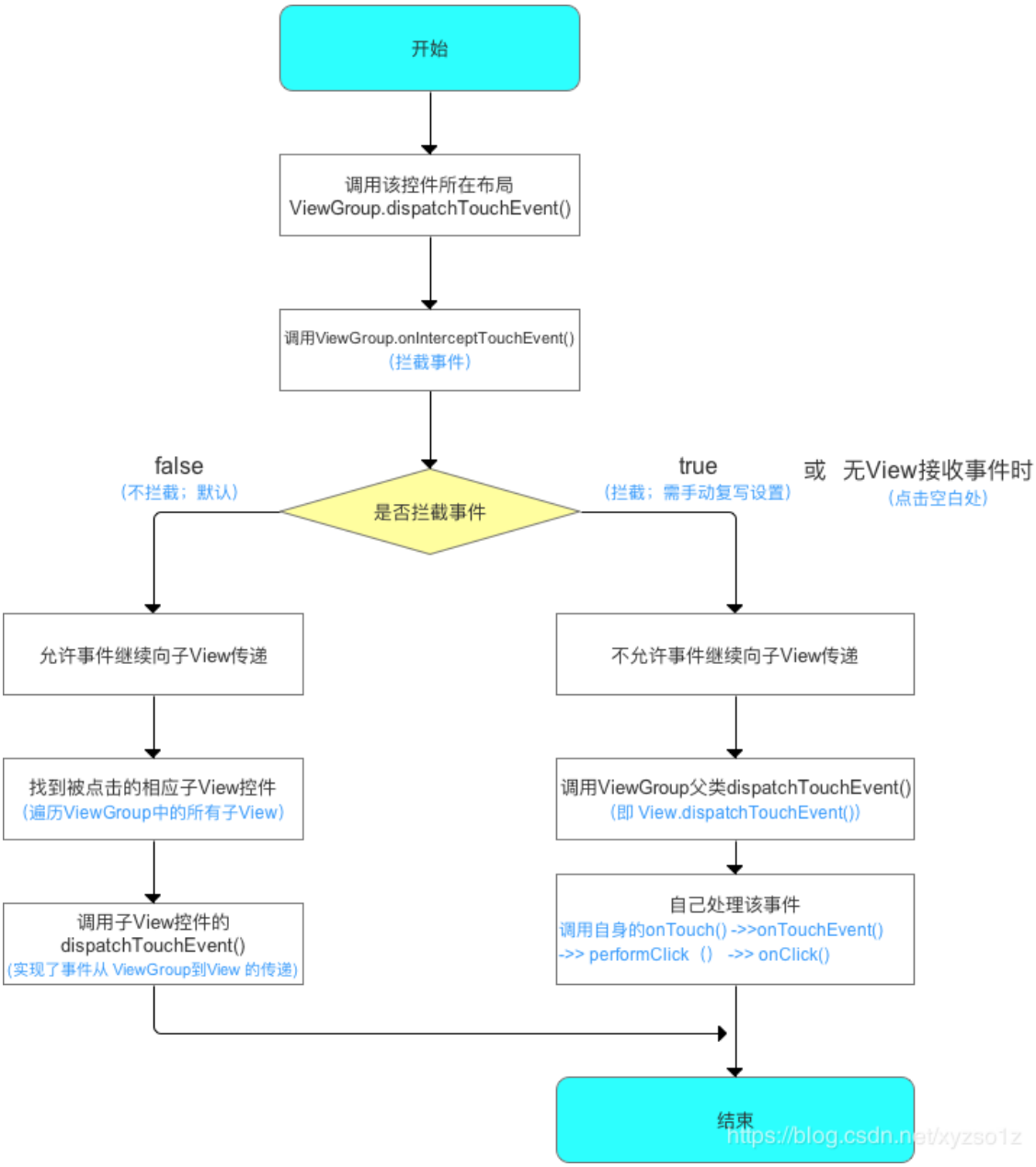
```
/**
  * 源码分析：ViewGroup.dispatchTouchEvent（）
  */
    public boolean dispatchTouchEvent(MotionEvent ev) {

    ... // 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
                    // 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
                    // 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
```

```
                        // 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
        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
```

3.2.2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



*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
dispatchTouchEvevt()	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.dispatchTouchEvevt()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 (即事件无被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)	当前事件无被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）
onTouchEvevt()	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）

3.3 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

3.3.1 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，
    // 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

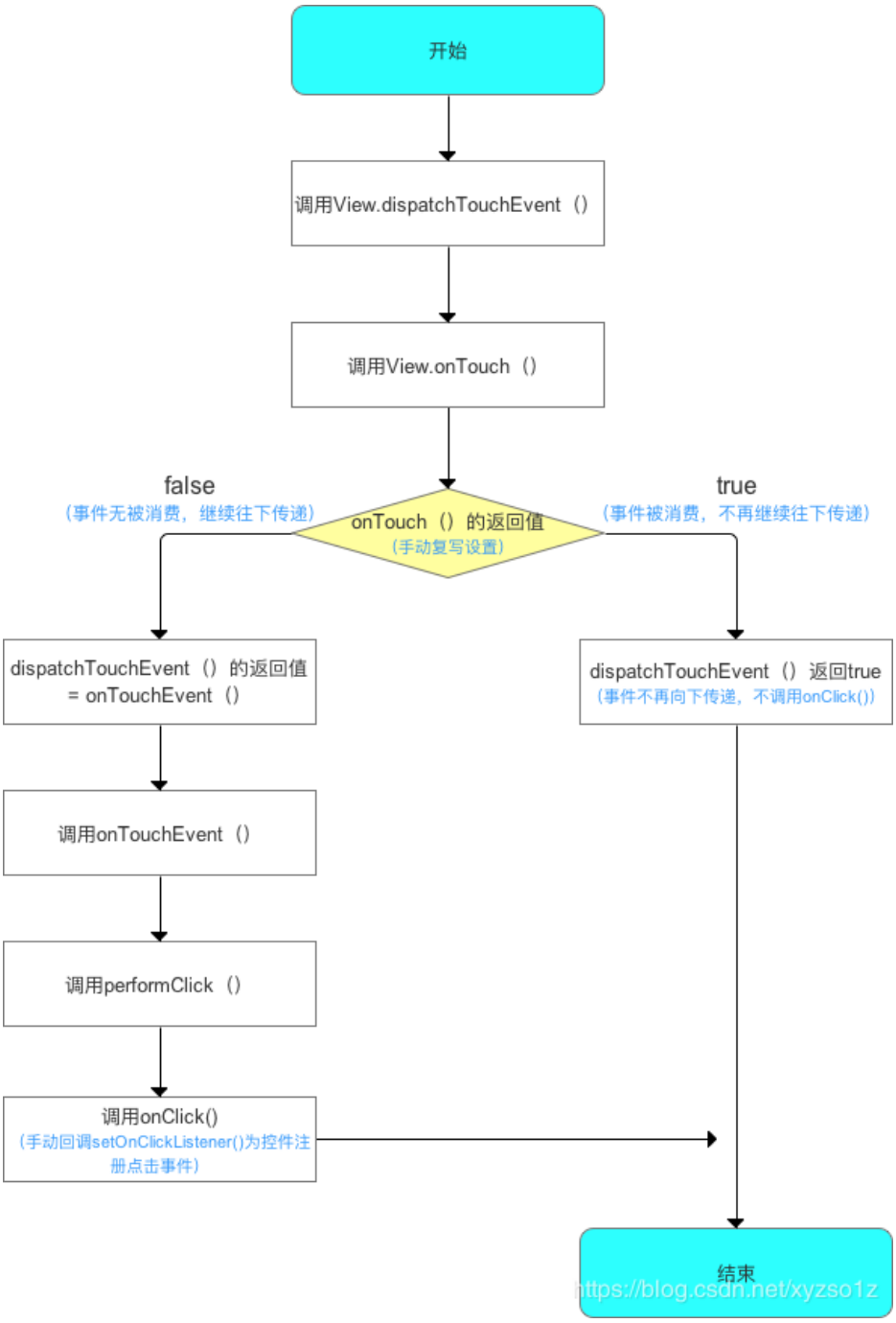
```
/**
  * 源码分析：View.onTouchEvent（）
  */
  public boolean onTouchEvent(MotionEvent event) {
    final int viewFlags = mViewFlags;

    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;
    }
```

3.3.2 总结

- 每当控件被点击时：



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
dispatchTouchEvevt()	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.onTouchEvevt()
		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.onTouchEvevt()返回true • ViewGroup..onTouchEvevt()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 (即事件无被View/ViewGroup接收&处理)	View.onTouchEvevt() 返回false	将事件回传给上层ViewGroup.onTouchEvent()处理
onTouchEvevt()	View.dispatchTouchEvevt()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)

3.3.3 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onCl
    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()");
            }

        });

```

测试结果

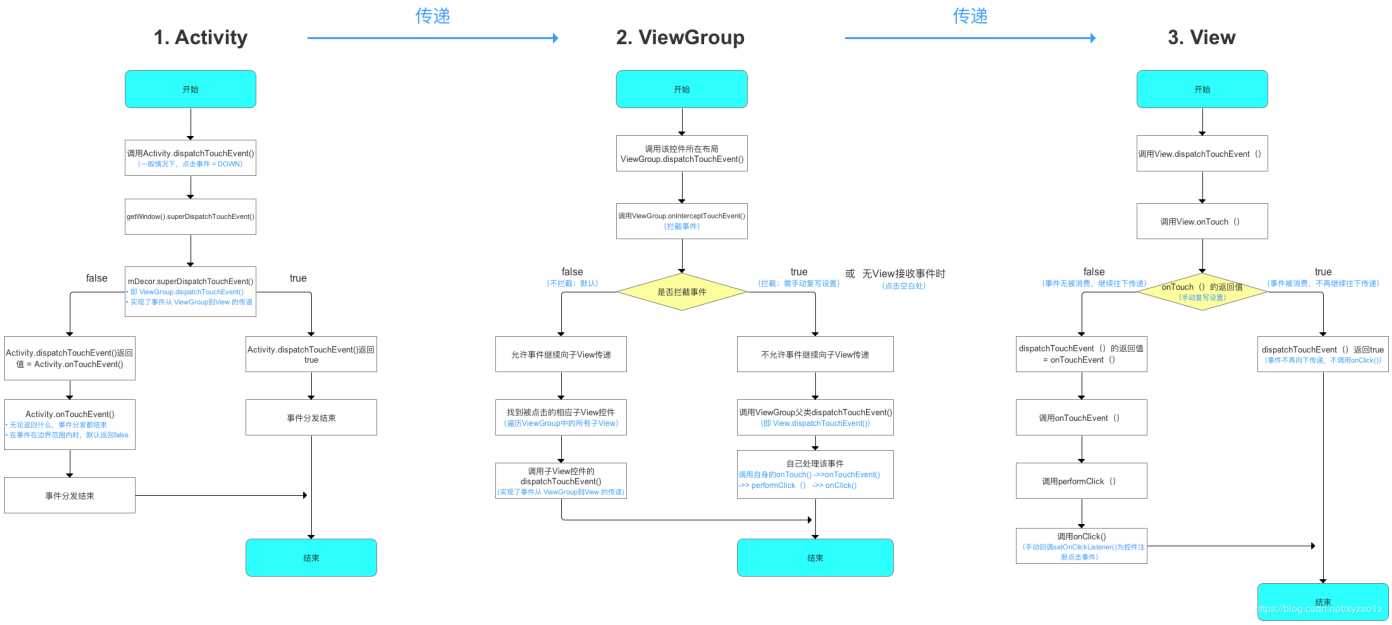
1. onTouch()返回false

```
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

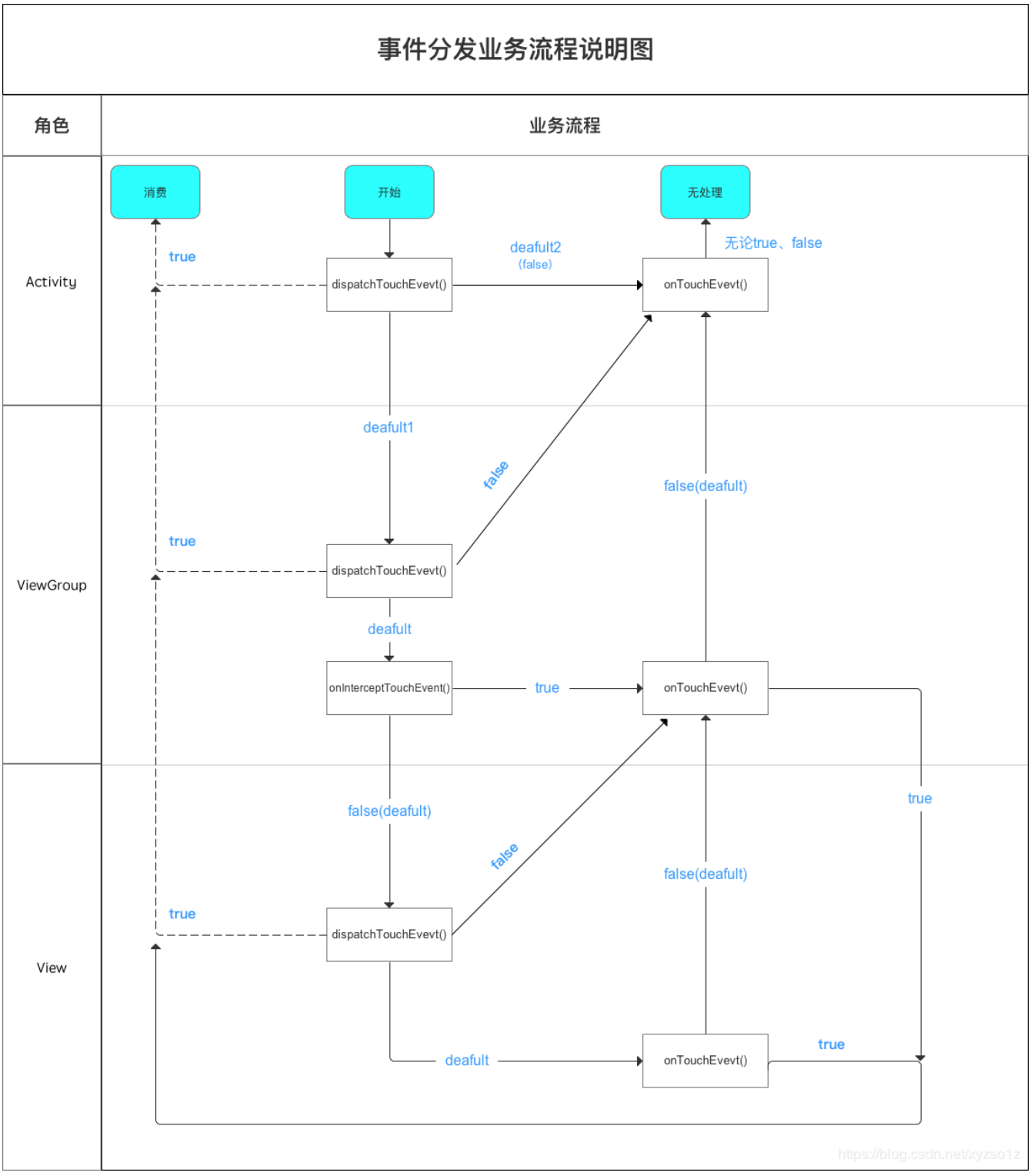
```
04:15:54.578 7334-7334/scut.carson_ho.touch_event I/System.out: 执行了onTouch(), 动作是:0
04:15:54.638 7334-7334/scut.carson_ho.touch_event I/System.out: 执行了onTouch(), 动作是:1
```


3.3.4 总结


1. Activity
传递
2. ViewGroup
传递
3. View
开始
调用Activity.dispatchTouchEvent()
（一般情况下，点击事件 = DOWN）
getWindow().superDispatchTouchEvent()
false
mDecor.superDispatchTouchEvent()
- 即 ViewGroup.dispatchTouchEvent()
- 实现了事件从 ViewGroup到View 的传递
true
Activity.dispatchTouchEvent()返回值 = Activity.onTouchEvent()
Activity.dispatchTouchEvent()返回true
Activity.onTouchEvent()
- 无论返回什么，事件分发都结束
- 在事件在边界范围内时，默认返回false
事件分发结束
事件分发结束
结束
开始
调用该控件所在布局ViewGroup.dispatchTouchEvent()
调用ViewGroup.onInterceptTouchEvent()
（拦截事件）
false
（不拦截，默认）
是否拦截事件
true
（拦截，需手动复写设置）
或 无View接收事件时
（点击空白处）
允许事件继续向子View传递
找到被点击的相应子View控件
（遍历ViewGroup中的所有子View）
调用子View控件的dispatchTouchEvent()
（实现了事件从 ViewGroup到View 的传递）
不允许事件继续向子View传递
调用ViewGroup父类dispatchTouchEvent()
（即 View.dispatchTouchEvent()）
自己处理该事件
调用自身的onTouch() ->>onTouchEvent()
->> performClick（） ->> onClick()
结束
开始
调用View.dispatchTouchEvent（）
调用View.onTouch（）
false
（事件无被消费，继续往下传递）
onTouch（）的返回值
（手动复写设置）
true
（事件被消费，不再继续往下传递）
dispatchTouchEvent（）的返回值 = onTouchEvent（）
调用onTouchEvent（）
调用performClick（）
调用onClick()
（手动回调setOnClickListener()为控件注册点击事件）
dispatchTouchEvent（）返回true
（事件不再向下传递，不调用onClick()）
结束
https://blog.csdn.net/xyzso1z

4.工作流程总结

- 事件分发的工作流程总结：



*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	dispatchTouchEvevt()	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.onTouchEvevt()
			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.dispatchTouchEvevt()返回true • onTouchEvevt() 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 (即事件无被View/ViewGroup接收&处理)	onTouchEvevt() 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）
	onTouchEvevt()	ViewGroup.dispatchTouchEvevt()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	dispatchTouchEvevt()	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.dispatchTouchEvevt()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 (即事件无被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)	当前事件无被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）
	onTouchEvevt()	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	dispatchTouchEvevt()	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.onTouchEvevt()
			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.onTouchEvevt()返回true • ViewGroup..onTouchEvevt()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 (即事件无被View/ViewGroup接收&处理)	View.onTouchEvevt() 返回false	将事件回传给上层ViewGroup.onTouchEvent()处理
	onTouchEvevt()	View.dispatchTouchEvevt()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)

- 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 （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	当前事件无被消费 （即事件无被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 （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；
				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 （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	• 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					

5.核心方法总结

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

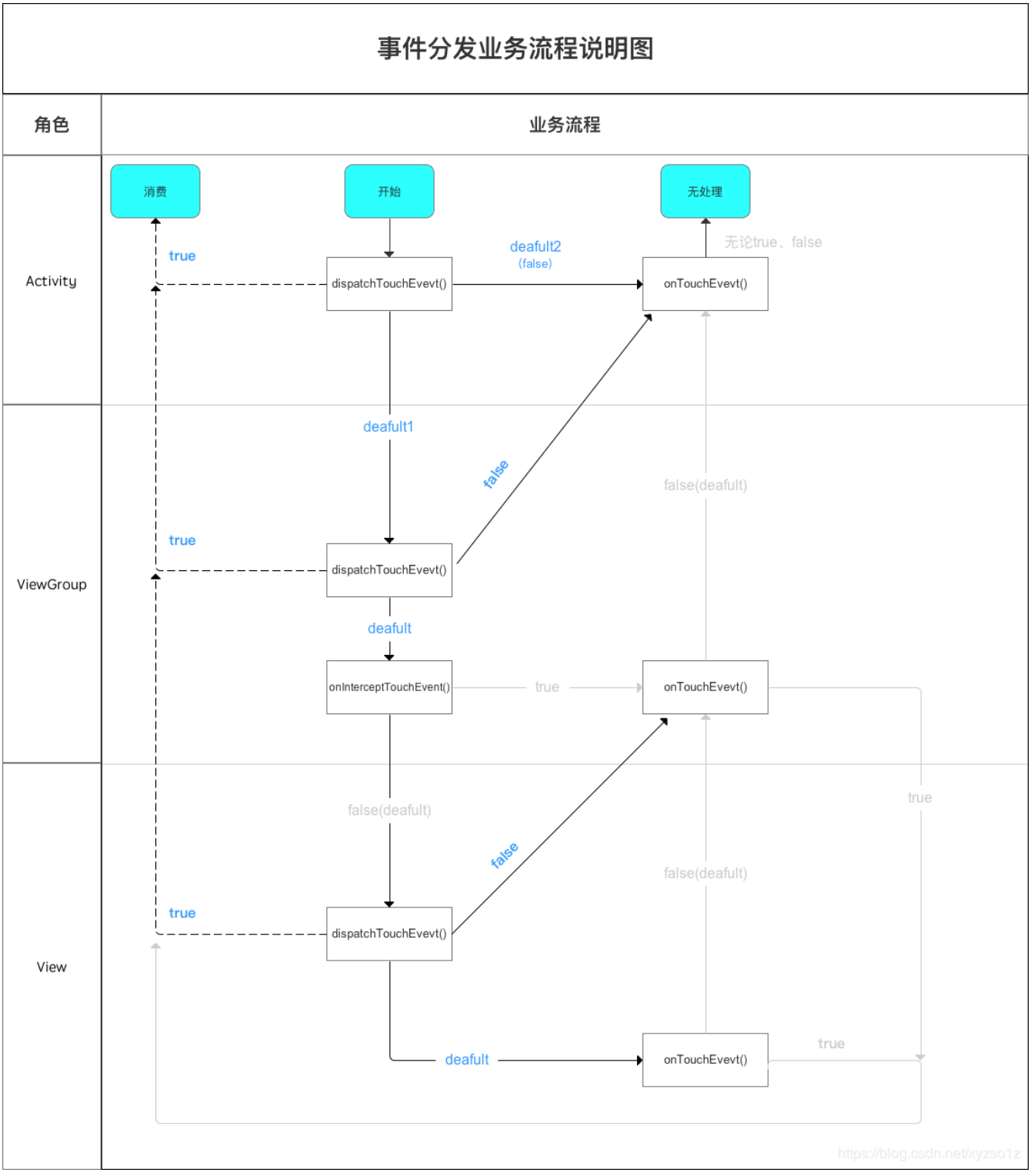
方法	使用对象	作用	调用时刻	返回结果说明		
				返回结果	具体含义	后续动作
dispatchTouchEvent()	• Activity • ViewGroup • View	分发（传递）点击事件	当点击事件能够传递给当前层时（Activity、ViewGroup、View），该方法就会被调用	默认	调用该方法时会直接调用如右方法	根据当前对象的不同而返回方法不同 • Activity：调用ViewGroup.dispatchTouchEvent() / Activity.onTouchEvent() • ViewGroup：调用自身的onInterceptTouchEvent() • 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	当前事件无被消费 （即事件无被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 （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					

- 下面，详细讲解该3个方法

4.1 dispatchTouchEvent()

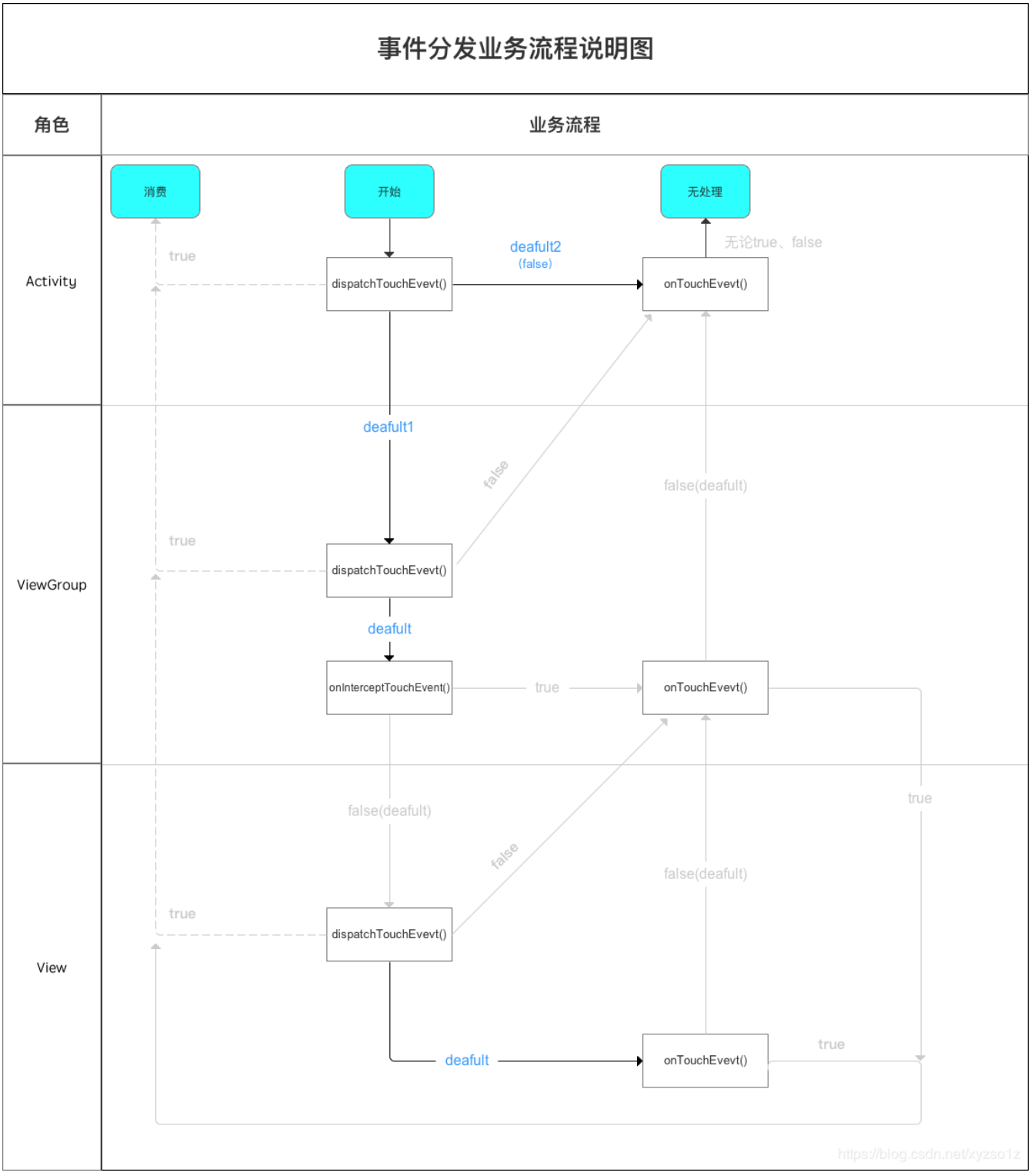
- 简介

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
• 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	当前事件无被消费 （即事件无被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）



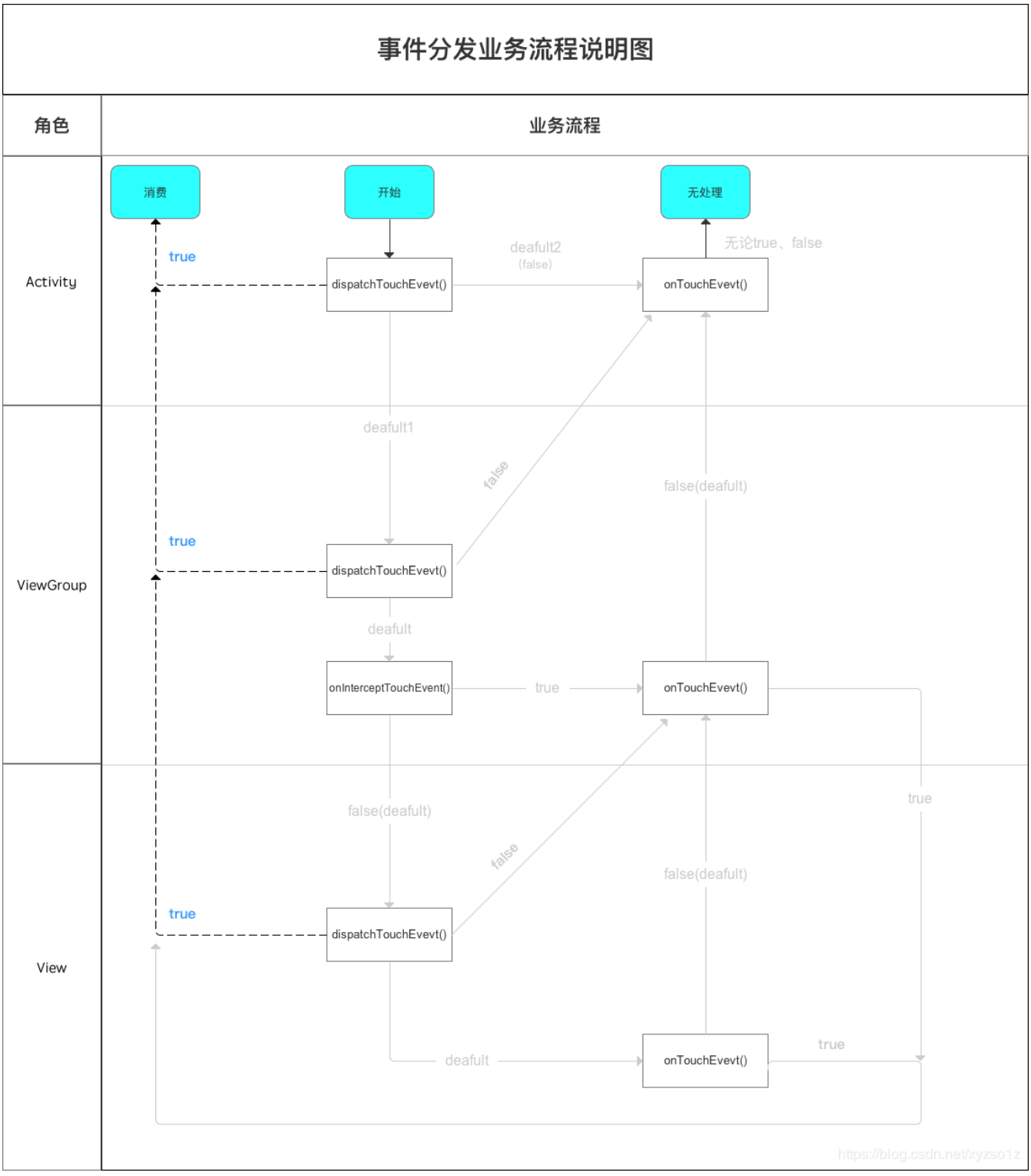
- 返回情况说明
  情况1：默认

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 • Activity：调用ViewGroup.dispatchTouchEvent() / Activity.onTouchEvent() • ViewGroup：调用自身的onInterceptTouchEvent() • View：调用自身的onTouchEvent （）



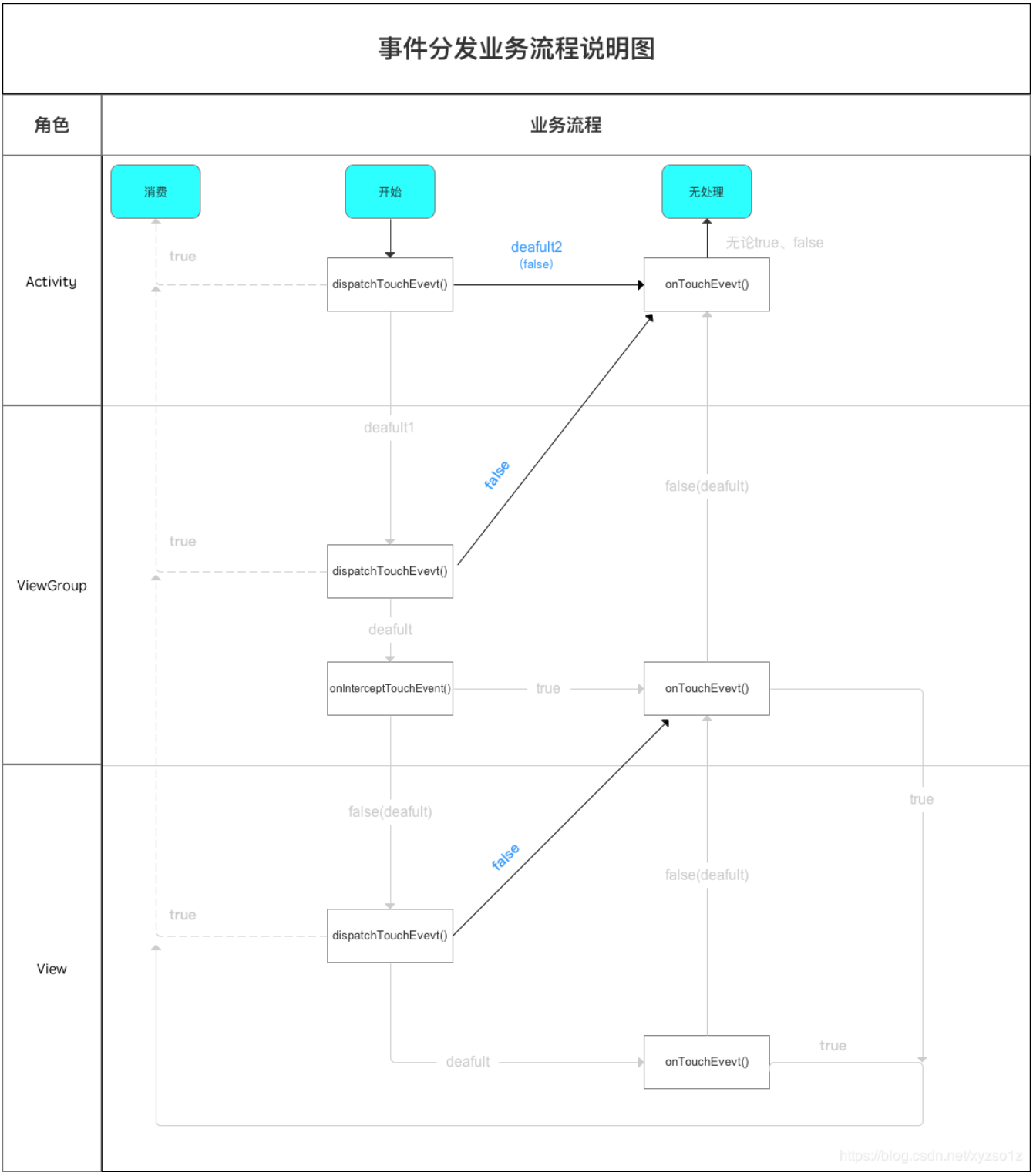
情况2：返回true

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）	• 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	• 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	



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）	• 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	• 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	

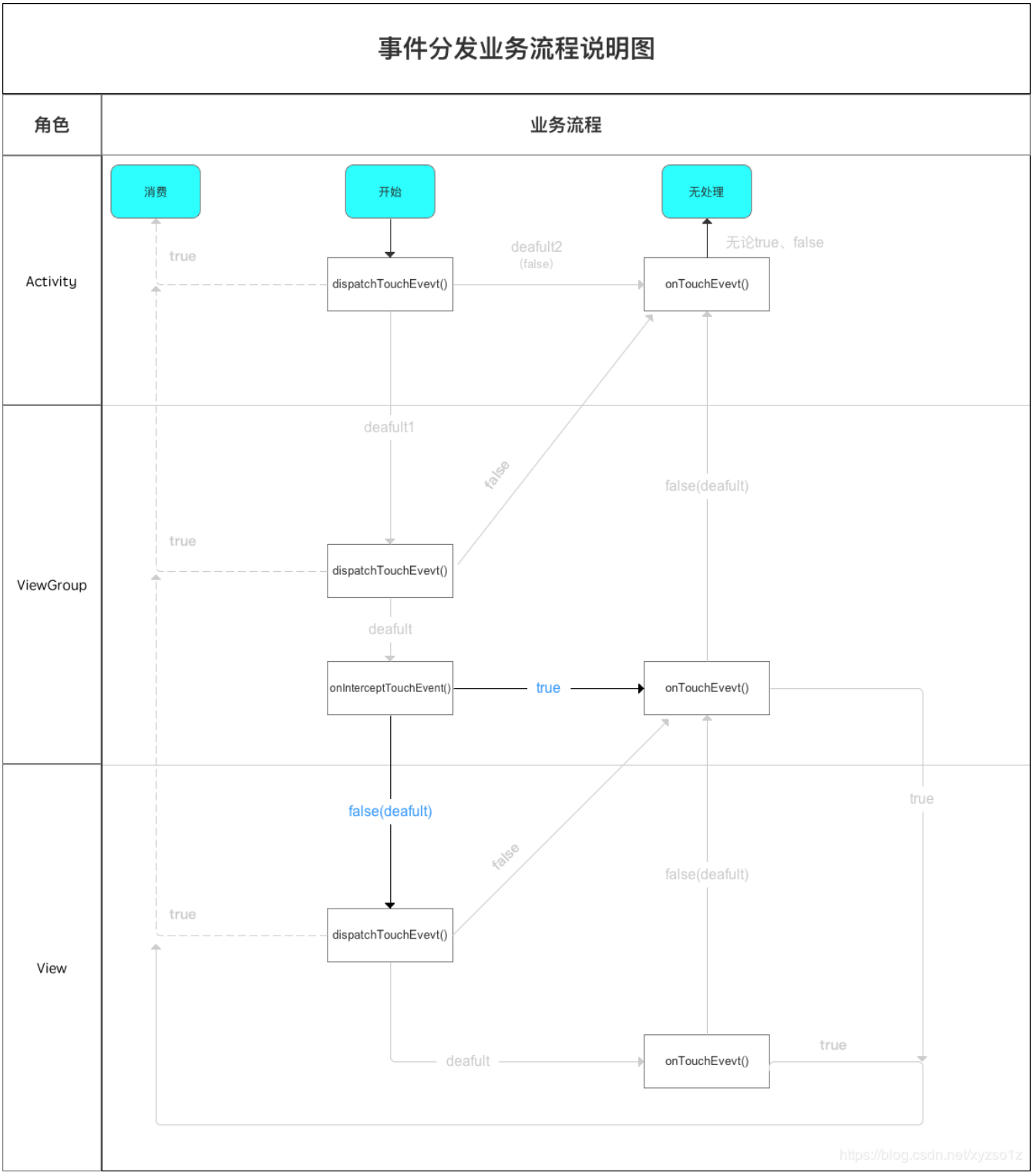


5.2 onInterceptTouchEvent()

- 简介

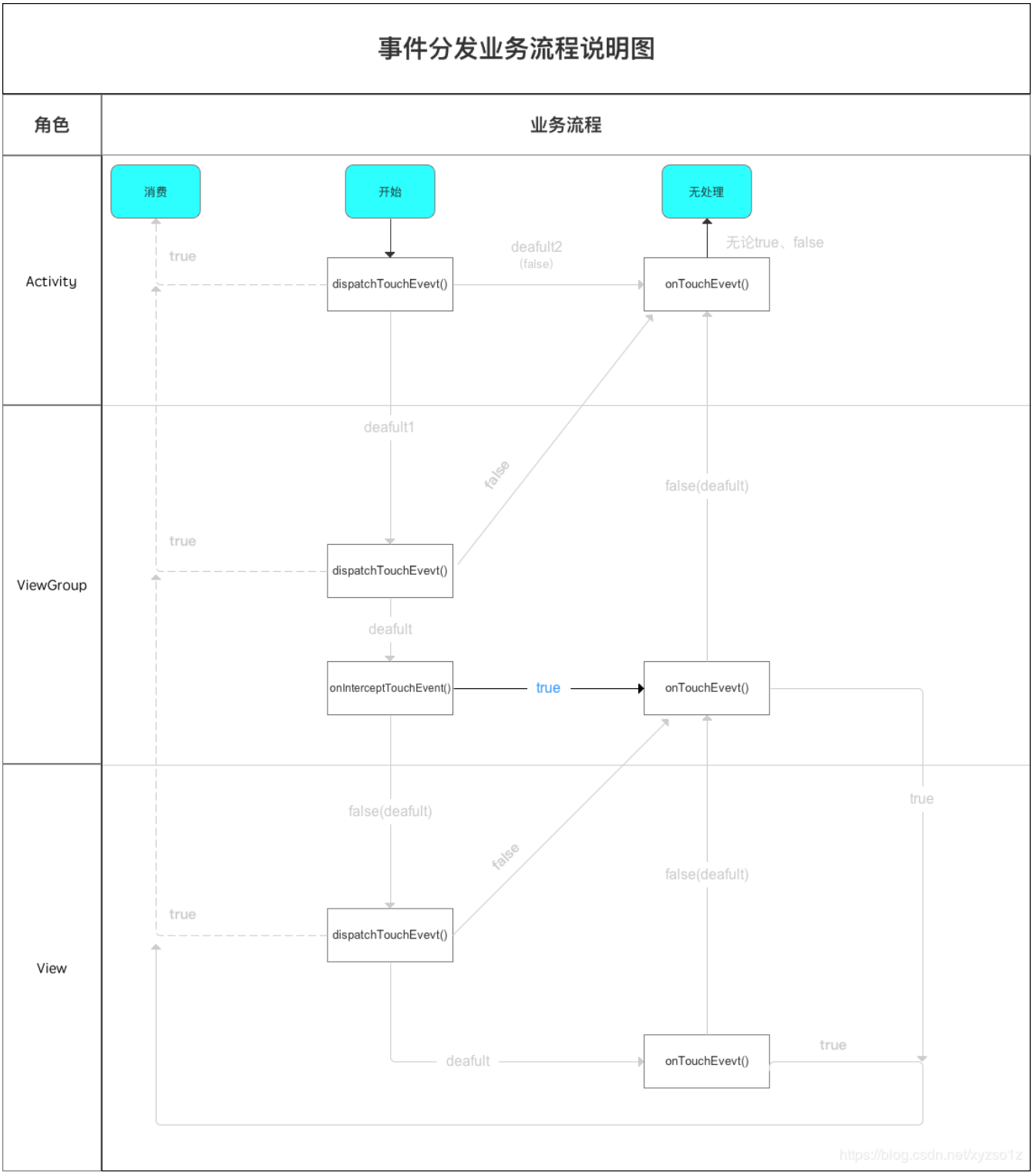
使用对象	作用	调用时刻	返回结果说明		
			返回结果	具体含义	后续动作
ViewGroup	判断是否拦截了某个事件 • 只存在于ViewGroup • 普通的View无该方法	在ViewGroup的dispatchTouchEvent() 内部调用	true	当前事件被ViewGroup拦截	• 事件停止往下传递 • ViewGroup自己处理事件，调用父类super.dispatchTouchEvent()，最终执行自己的onTouchEvent()； • 同一个事件列的其他事件都直接交由该View处理；在同一个事件列中该方法不会再次被调用；
			false (default)	当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

注：在`Activity`、`View`中无该方法



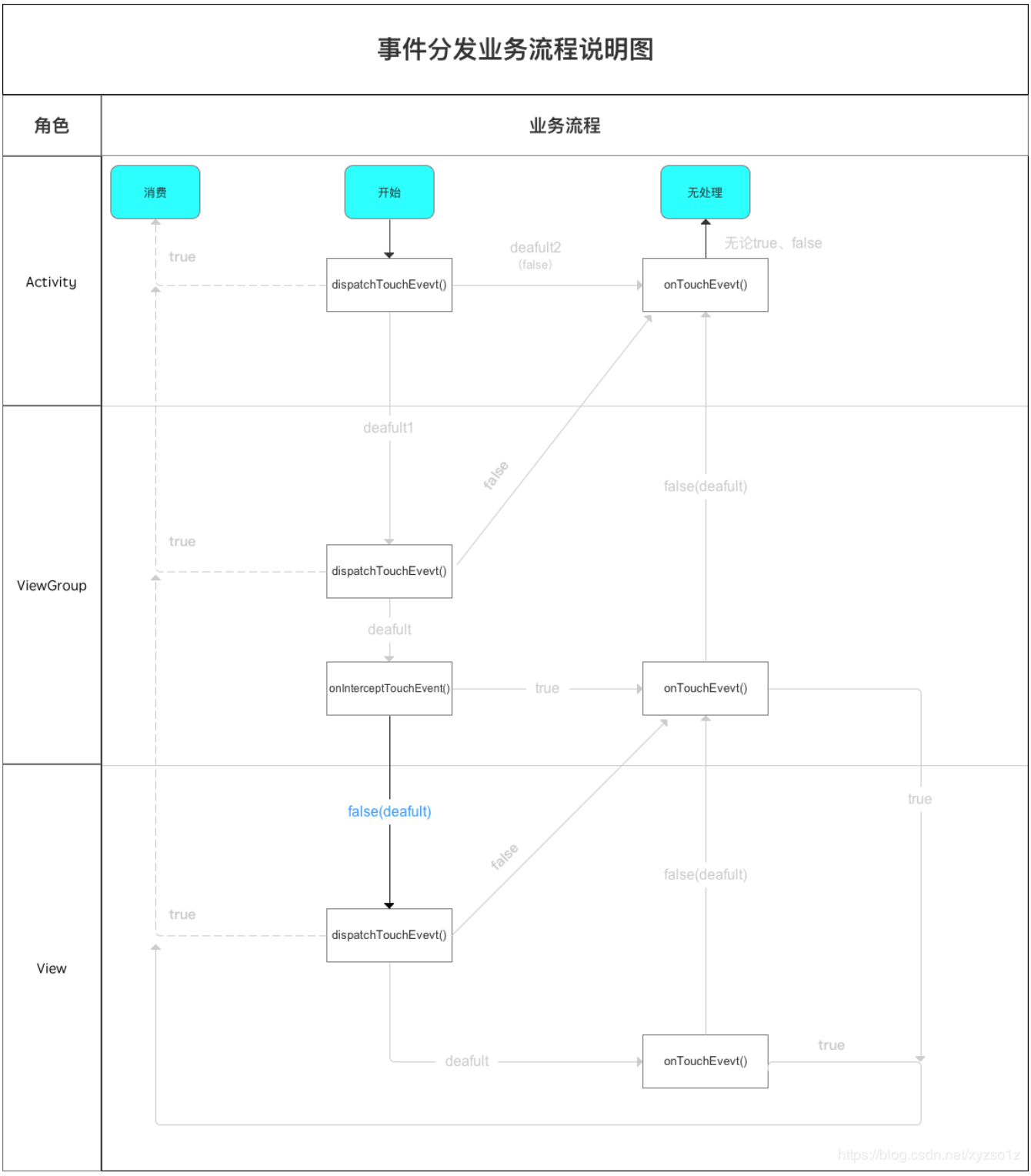
返回情况说明
情况1：true

返回结果	具体含义	后续动作
true	当前事件被ViewGroup拦截	• 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；


事件分发业务流程说明图
角色
业务流程
Activity
ViewGroup
View
消费
开始
无处理
true
dispatchTouchEvevt()
deafult2
(false)
onTouchEvevt()
无论true、false
deafult1
false
false(deafult)
true
dispatchTouchEvevt()
deafult
onInterceptTouchEvent()
true
onTouchEvevt()
false(deafult)
false
false(deafult)
true
dispatchTouchEvevt()
deafult
onTouchEvevt()
true
https://blog.csdn.net/xyzso1z

情况2：false(默认)

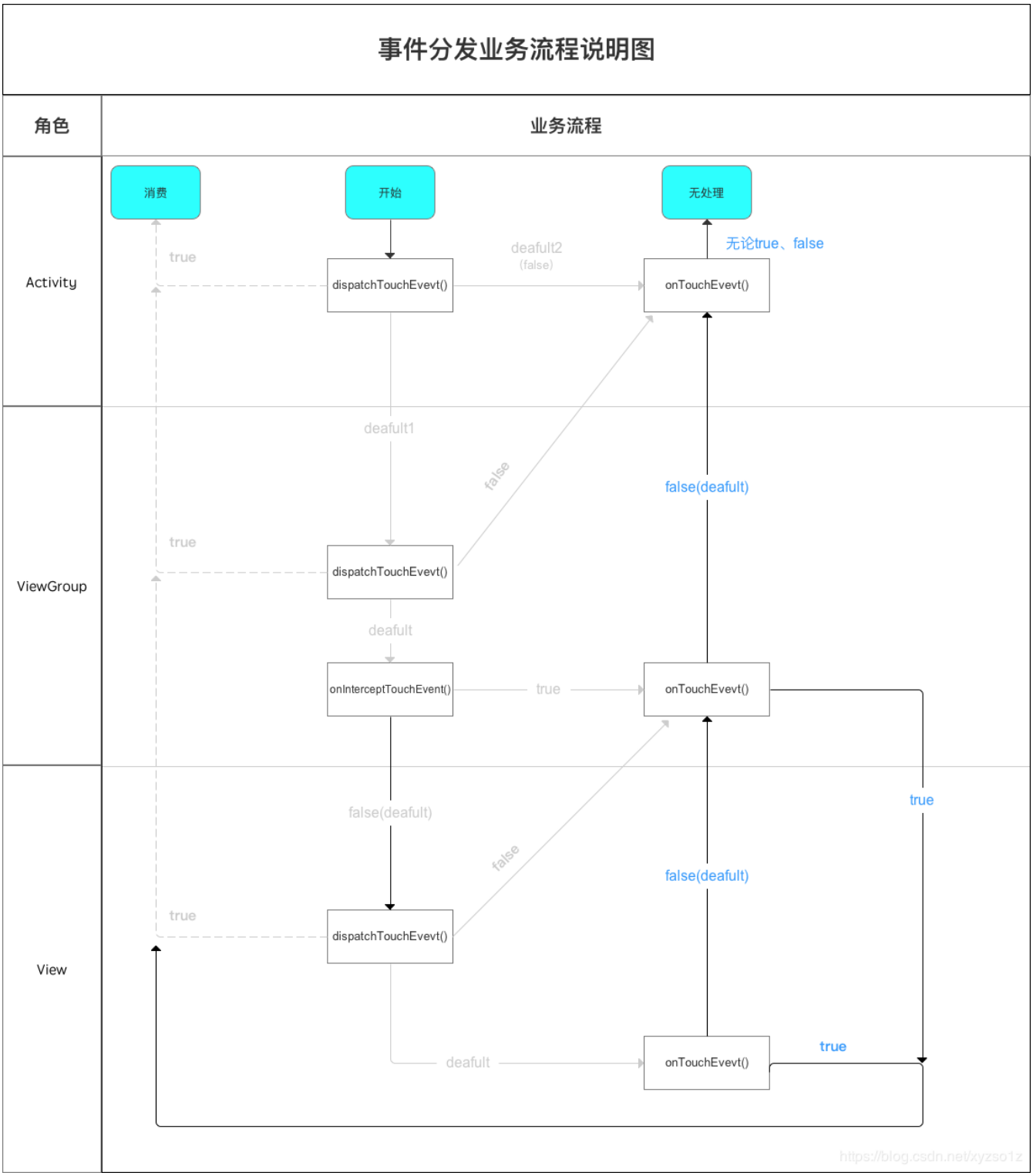
返回结果	具体含义	后续动作
false (default)	当前事件无被ViewGroup拦截	• 事件继续往下传递 • 事件传递到子view，调用View.dispatchTouchEvent() 方法中去处理 • 当前View仍然接受此事件的其他事件（与onTouchEvent()区别）



5.3 onTouchEvent()

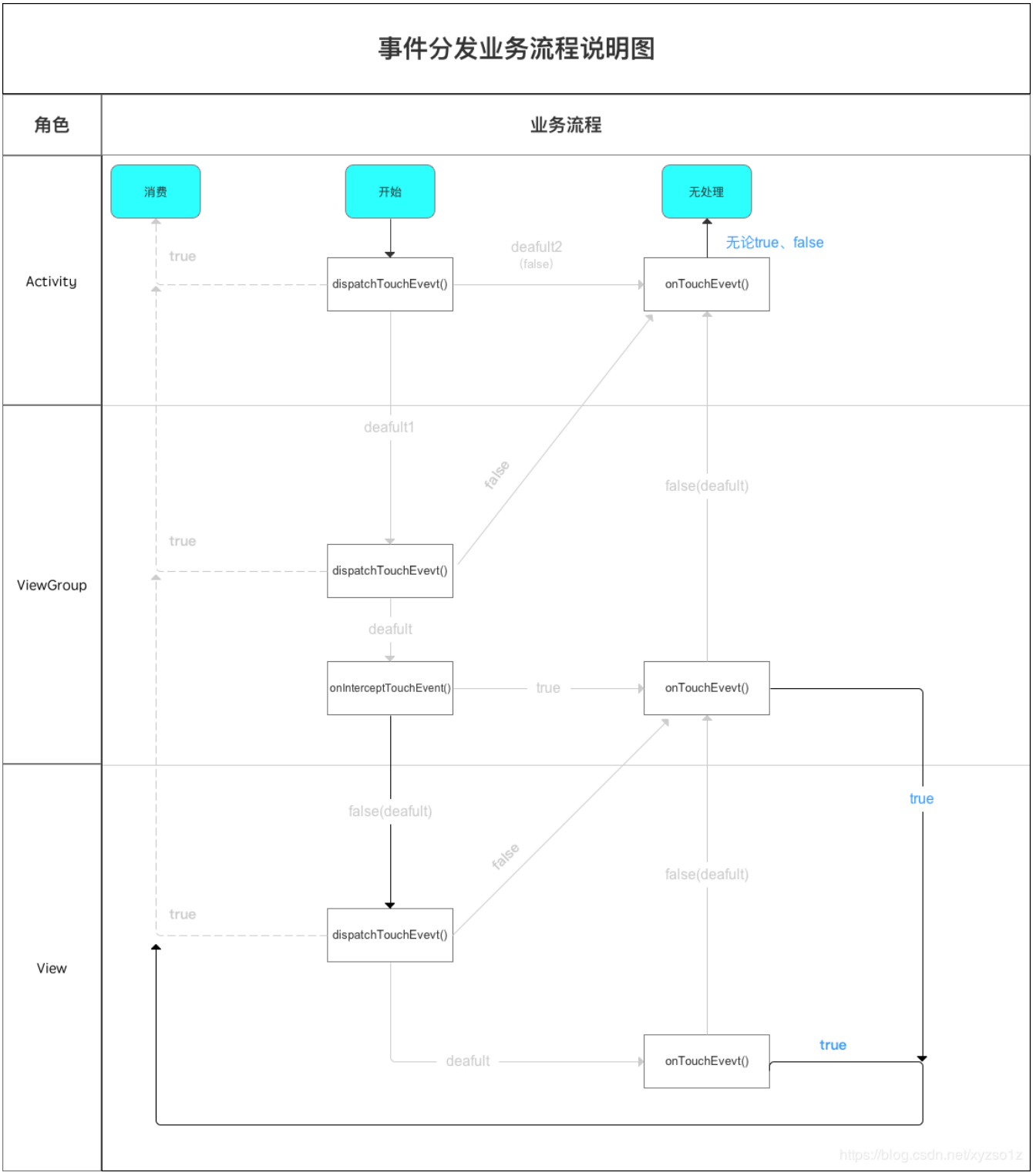
- 简介

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）



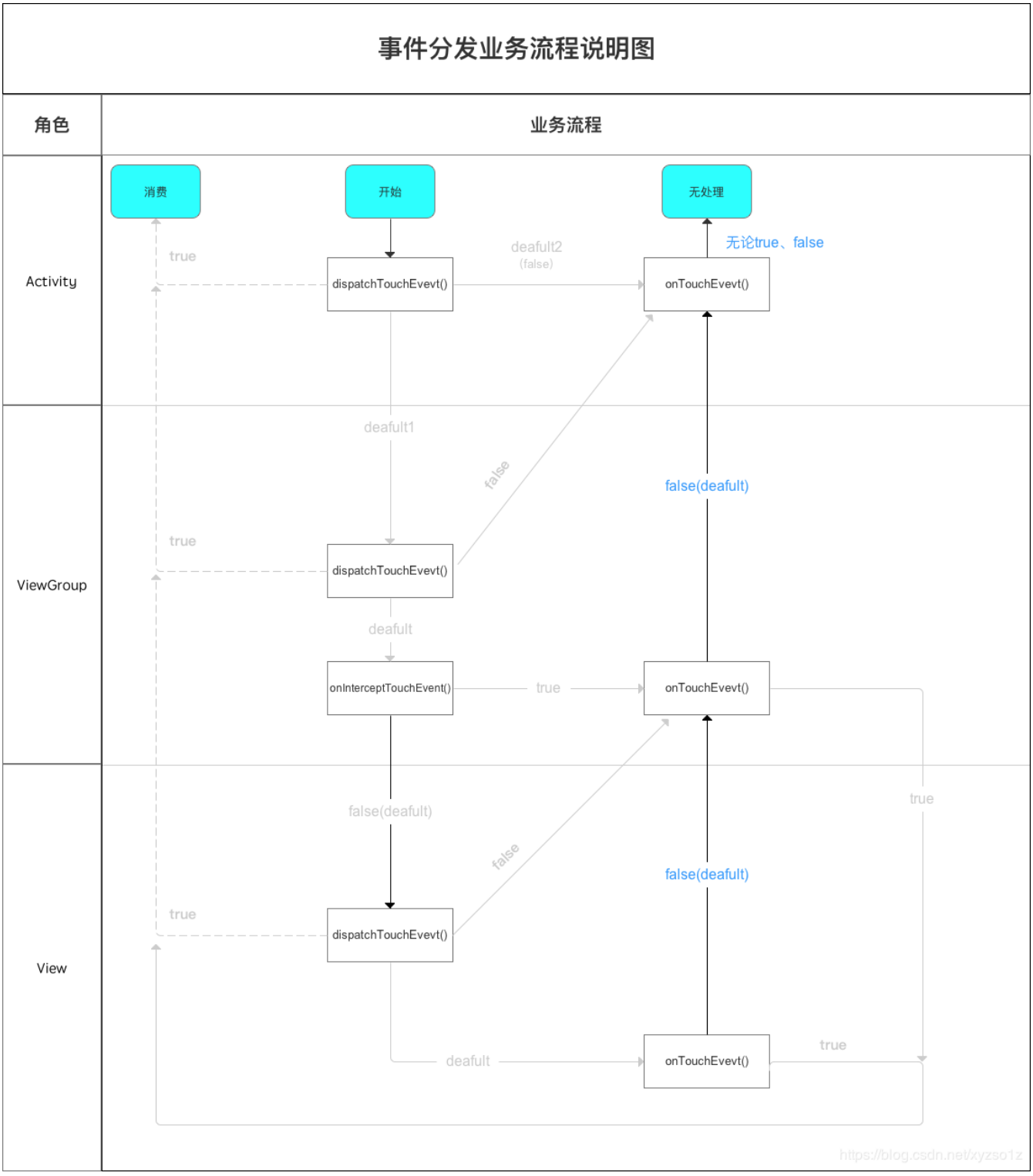
返回情况说明
情况1：返回true

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）	• 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	• 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	



情况2：返回false(默认)

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）	• 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	• 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	



5.4 三者关系

下面，用一段伪代码来阐述上述3个方法的关系和时间传递规则

```
/**
 * 点击事件产生后
 */
// 步骤1：调用dispatchTouchEvent（）
public boolean dispatchTouchEvent(MotionEvent ev) {
```

```
6
7       boolean consume = false; //代表 是否会消费事件
8
9       // 步骤2：判断是否拦截事件
10      if (onInterceptTouchEvent(ev)) {
11        // a. 若拦截，则将该事件交给当前View进行处理
12        // 即调用onTouchEvent ()方法去处理点击事件
13          consume = onTouchEvent (ev) ;
14
15      } else {
16
17        // b. 若不拦截，则将该事件传递到下层
18        // 即 下层元素的dispatchTouchEvent ()就会被调用，重复上述过程
19        // 直到点击事件被最终处理为止
20        consume = child.dispatchTouchEvent (ev) ;
21      }
22
23      // 步骤3：最终返回通知 该事件是否被消费（接收 & 处理）
24      return consume;
25
26    }
```

6.常见的事件分发场景

下面，将通过实例说明常见的事件传递情况和流程

6.1 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

- 情景
1.用户先触摸到屏幕上`View c`上的某个点（图中黄区）

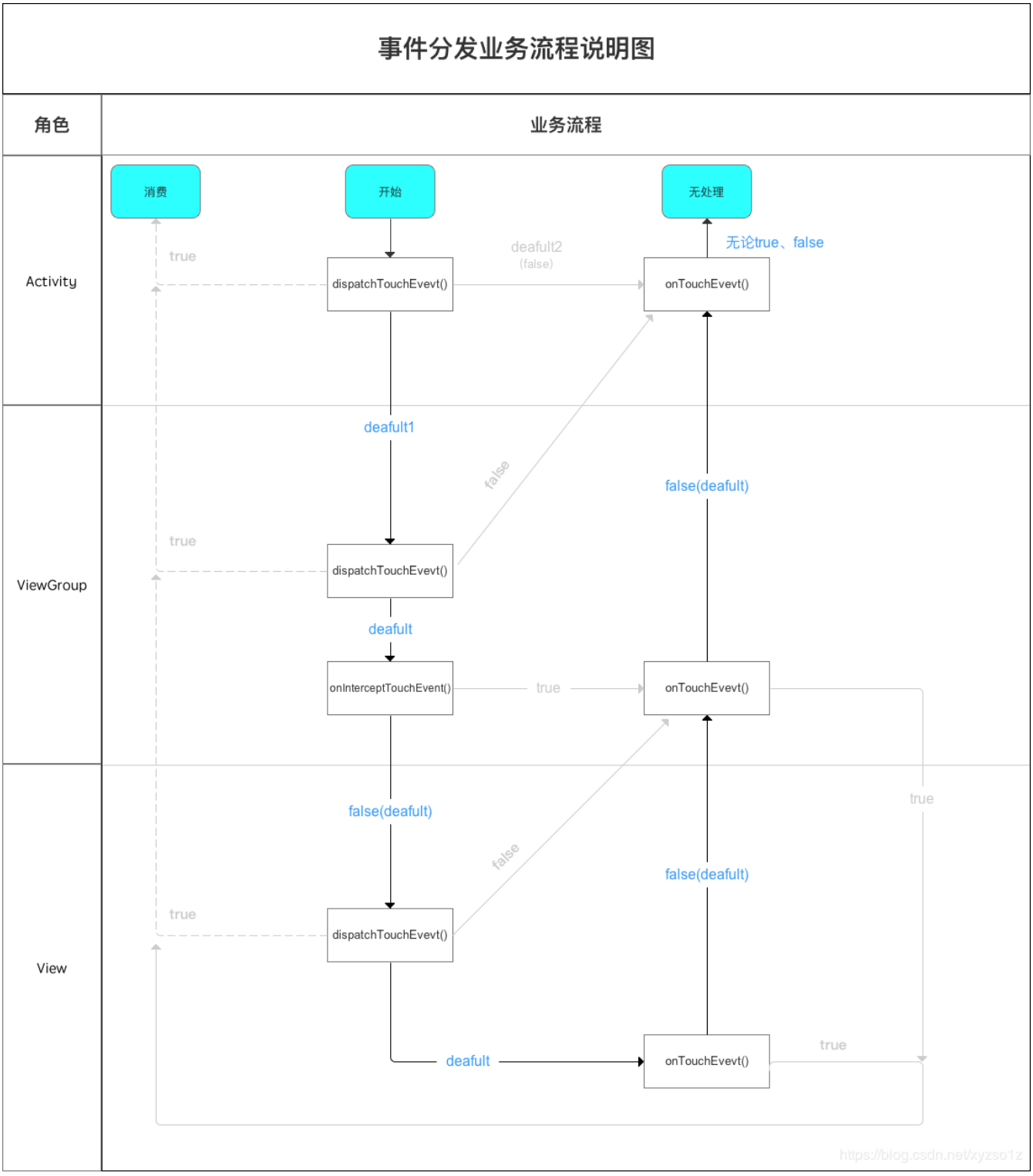
`Action_DOWN`事件在此处产生
2.用户移动手
3.最后离开屏幕

6.2 一般的 事件传递情况

- 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
- 那么调用的试着3个方法的默认实现；调用下层的方法的&逐层返回
- 事件传递情况：
  1.从上往下调用`dispatchTouchEvent()`
  ActivityA——>ViewGroupB——>View C
  2.从下往上调用onTouchEvent()
  View C——>ViewGroup B——>Activity A



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

> 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）

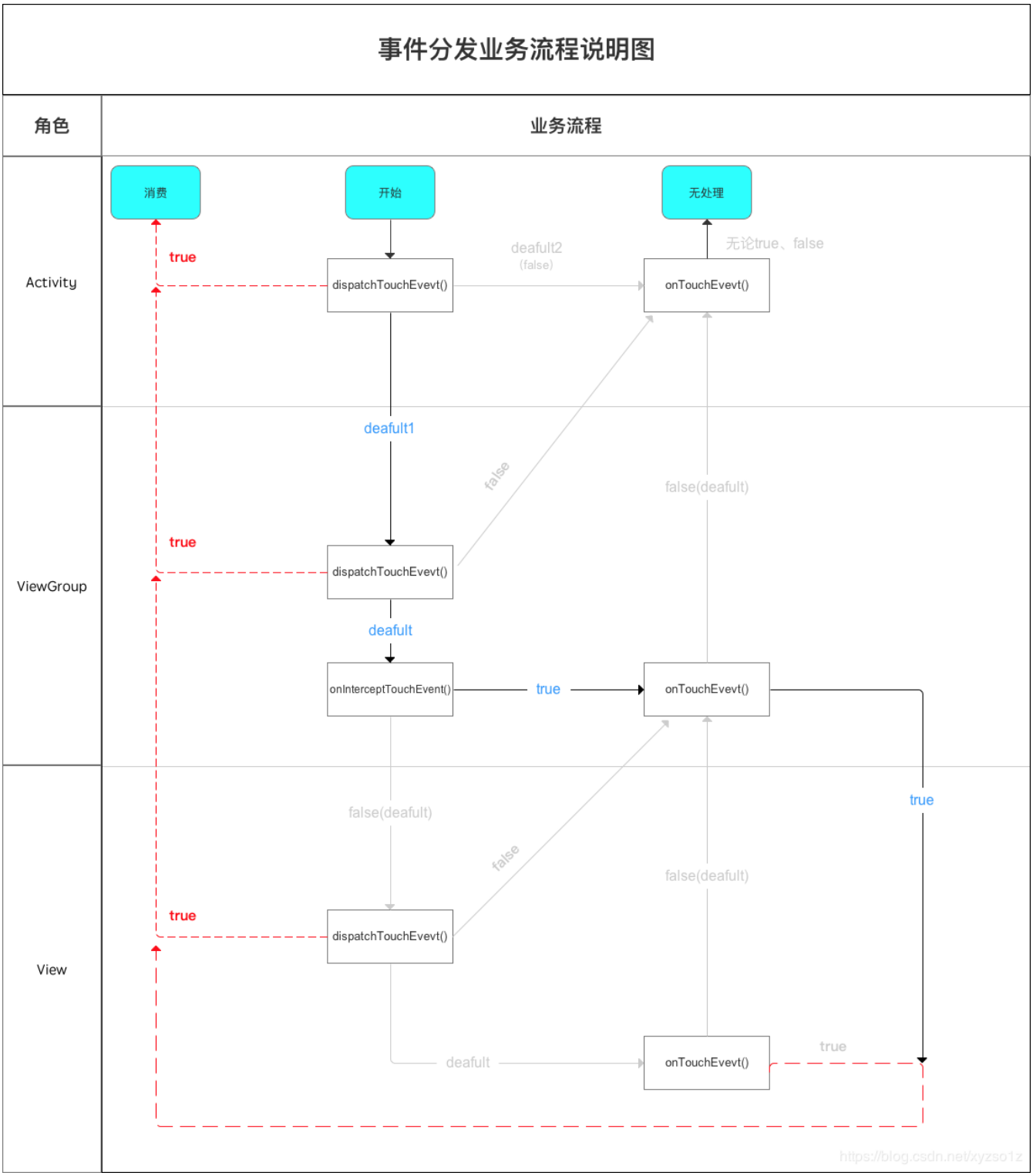
- `DOWN` 事件被传递给C的 `onTouchEvent()` 方法，该方法返回 `true` ，表示处理该事件

- 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()；![在这里插入图片描述](https://img-blog.csdnimg.cn/20190429001454778.png?x-oss-process=image/watermark,type_ZmFuZ3poZW5naGVpdGk,shadow_10,text_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L3h5enNvMXo=,size_16,color_FFFFFF,t_70) 会逐层往dispathchtouchEvent()``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onInterceptTouchEvent()返回true，onTouchEvent()返回true``
> 事件传递情况：

- 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(),因为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



场景4：拦截DOWN的后续事件
结论

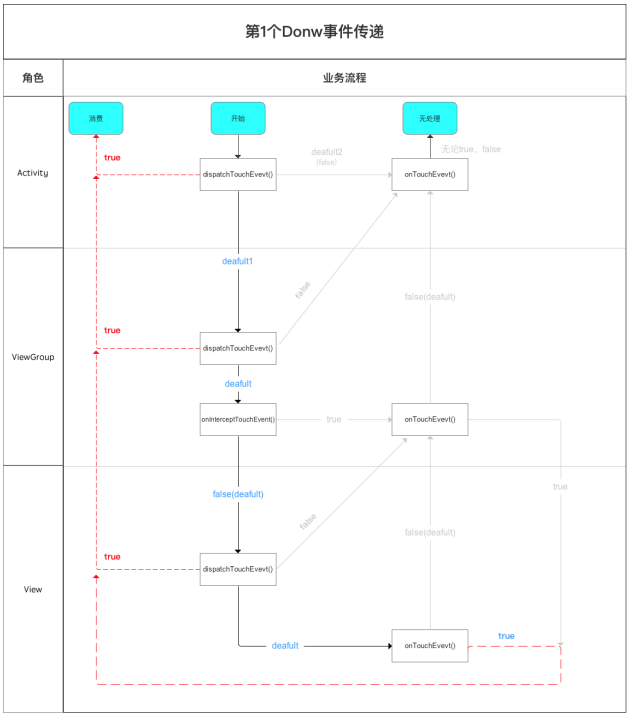
- 若 ViewGroup 拦截了一个半路的事件(如 Move)，该事件将会被系统变成一个 CANCELL 事件&传递给之前处理该事件的子 View ;
- 该事件不会再传递给 ViewGroup 的 onTouchEvent()
- 只有再到来的事件才会传递到 ViewGroup 的 onTouchEvent()
场景描述

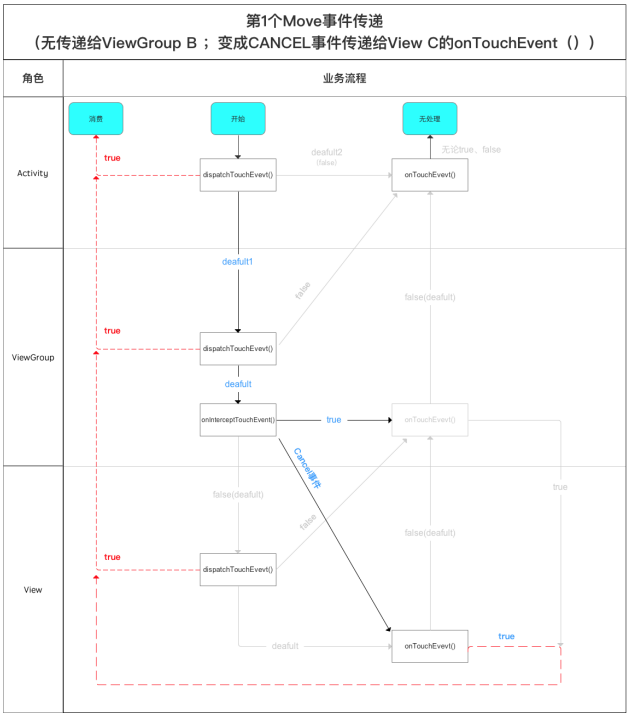
`ViewGroup B`无法拦截`DOWN`事件（还是`View C`来处理`Down`事件），但它拦截了接下来的`Move`事件
即`Down`事件传递到`View C`的`onTouchEvent()`，返回`true`
实例讲解

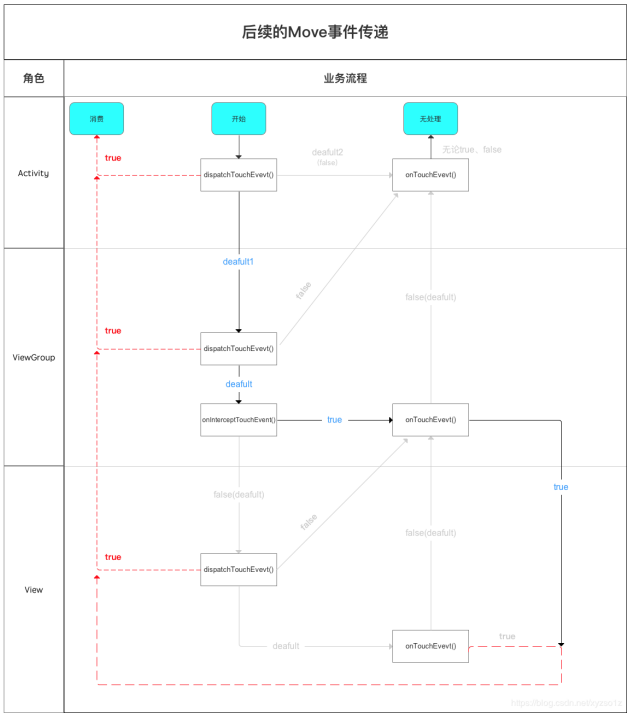
- 在后续到来的`MOVE`事件，`ViewGroup B`的`onInterceptTouchEvent()`返回`true`拦截该`MOVE`事件，但该事件并没有传递给`ViewGroup B`;这个`MOVE`事件将会被系统变成一个`CANCEL`事件传递给`View C`的`onTouchEvent()`;
- 后续又来了一个`MOVE`事件，该`MOVE`事件才会直接传递给`ViewGroup B`的`onTouchEvent()`

> 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







7.额外知识

7.1 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，它相当于是一个分岔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
- 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，结束本次事件传递的过程。


xyzso1z
原创文章 80　获赞 40　访问量 2万+