Android 按键的焦点分发处理机制


wanderingguy LV.4
2019年08月16日 16:58 · 阅读 1703

关注

> 本文重点针对android TV开发的同学，分析遥控或键盘按键事件后焦点的分发机制。尤其是刚从手机开发转向TV开发的同学，因为在实际开发中总会出现丢焦点或者焦点给到非预期的View或者ViewGroup。但此问题实际开发情况比较复杂，本文仅限从android基本的分发机制出发，对整体流程进行梳理，并提供一些方法改变默认行为以达到特定需求。如果有机会后续还会有更偏向实战的内容更新。

一.前言

本文源码基于android 7.0

二.核心流程

首先我们要知道按键事件和触屏事件一样都是从硬件通过系统驱动传递给android framework层的，当然这也不是我们要关注的重点。事件的入口就是ViewRootImpl的processKeyEvent方法。

java 复制代码

```
private int processKeyEvent(QueuedInputEvent q) {
    final KeyEvent event = (KeyEvent) q.mEvent;
    // ①view树处理事件消费逻辑
    if (mView.dispatchKeyEvent(event)) {
        return FINISH_HANDLED;
    }
    if (shouldDropInputEvent(q)) {
        return FINISH_NOT_HANDLED;
    }
    ....//处理ctrl键 也就是快捷按键相关逻辑

    // 自动寻焦逻辑
    if (event.getAction() == KeyEvent.ACTION_DOWN) {
        int direction = 0;
        ....//根据keycode赋值direction
        if (direction != 0) {
            // ②寻找当前界面的焦点view/viewGroup
```

```
                View focused = mView.findFocus();
                if (focused != null) {
                    // ③根据方向按键寻找合适的focus view
                    View v = focused.focusSearch(direction);
                    if (v != null && v != focused) {
                        ...
                        // ④请求焦点
                        if (v.requestFocus(direction, mTempRect)) {
                            playSoundEffect(SoundEffectConstants
                                    .getContantForFocusDirection(direction));
                            return FINISH_HANDLED;
                        }
                    }

                    // 没有找到焦点 给view最后一次机会处理按键事件
                    if (mView.dispatchUnhandledMove(focused, direction)) {
                        return FINISH_HANDLED;
                    }
                } else {
                    // 如果当前界面没有焦点走这里
                    View v = focusSearch(null, direction);
                    if (v != null && v.requestFocus(direction)) {
                        return FINISH_HANDLED;
                    }
                }
            }
        }
        return FORWARD;
    }
```

如上面的标号，就是寻焦的主要流程。其他的一些判断代码由于篇幅限制就不贴出了。下面分别对上述四个节点一一分析。

如果你去看了ViewRootImpl的源码会发现 其中有三个内部类都有这个方法，分别为：ViewPreImeInputStage，EarlyPostImeInputStage,ViewPostImeInputStage他们都继承InputStage类，上面的代码是ViewPostImeInputStage类中的，从类名可以判断按键的处理跟输入法相关，如果输入法在前台则会将事件先分发给输入法。

2.1 dispatchKeyEvent

想必你已经很熟悉android事件分发机制了(当然这也不是重点)，key事件和touch事件原理都是一样。总得来说就是由根view，这里就是DecorView它是一个FrameLayout它先分发给activity，之后顺序为activity-->PhoneWindow-->DecorView-->View树，view树中根据focusd path分发，也就是从根节点开始直至focused view 为止的树，具体流程可参看Android按键事件

处理流程 -- KeyEvent。 遍历过程中一旦有节点返回true即表示消费此事件，否则会一直传递下去。**之所以说明这些，是想提供一种拦截焦点的思路，如果按键事件传递过程中被消费便不会走寻焦逻辑。**具体的流程后续会分享个大家。 ####2.2 findFocus 此方法的核心就是找到当前持有focus的view。 调用者mView即DecorView是FrameLayout 布局，没复写findFocus方法，所以找到ViewGroup中的findFocus方法。

kotlin 复制代码

```
public View findFocus() {
    if (isFocused()) {
        return this;
    }
    if (mFocused != null) {
        return mFocused.findFocus();
    }
    return null;
}
```

逻辑很简单如果当前view是focused的状态直接返回自己，否则调用内部间接持有focus的子view即mFocused，遍历查找focused view。可见此番查找的路径就是focused tree。

2.3 focusSearch

根据开篇的核心流程，如果在上一步中找到了focused view，则会执行view的focusSearch(int direction)方法，否则执行focusSearch(View focused, int direction)。这两个方法分别来自于View和ViewGroup，但核心功能是一致的，看代码。

kotlin 复制代码

```
/**
 * Find the nearest view in the specified direction that can take focus.
 * This does not actually give focus to that view.
 *
 * @param direction One of FOCUS_UP, FOCUS_DOWN, FOCUS_LEFT, and FOCUS_RIGHT
 *
 * @return The nearest focusable in the specified direction, or null if none
 *         can be found.
 */
public View focusSearch(@FocusRealDirection int direction) {
    if (mParent != null) {
        return mParent.focusSearch(this, direction);
    } else {
        return null;
    }
}

/**
```

```
     * Find the nearest view in the specified direction that wants to take
     * focus.
     *
     * @param focused The view that currently has focus
     * @param direction One of FOCUS_UP, FOCUS_DOWN, FOCUS_LEFT, and
     *        FOCUS_RIGHT, or 0 for not applicable.
     */
    public View focusSearch(View focused, int direction) {
        if (isRootNamespace()) {
            // root namespace means we should consider ourselves the top of the
            // tree for focus searching; otherwise we could be focus searching
            // into other tabs.  see LocalActivityManager and TabHost for more info
            return FocusFinder.getInstance().findNextFocus(this, focused, direction);
        } else if (mParent != null) {
            return mParent.focusSearch(focused, direction);
        }
        return null;
    }
```

连注释都是惊人的相似有木有，大致意思是将focusSearch事件一直向父View传递，如果这个过程上层一直没有干涉则会遍历到顶层DecorView。回到上面的分水岭，如果findFocus没有找到focused view，即把null 赋值给focused传递，整个流程不受影响。 重点方法是**FocusFinder.getInstance().findNextFocus(this, focused, direction);**来看源码。

java 复制代码

```
    /**
     * Find the next view to take focus in root's descendants, starting from the view
     * that currently is focused.
     * @param root Contains focused. Cannot be null.
     * @param focused Has focus now.
     * @param direction Direction to look.
     * @return The next focusable view, or null if none exists.
     */
    public final View findNextFocus(ViewGroup root, View focused, int direction) {
        return findNextFocus(root, focused, null, direction);
    }

    private View findNextFocus(ViewGroup root, View focused, Rect focusedRect, int direction
        View next = null;
        if (focused != null) {
            next = findNextUserSpecifiedFocus(root, focused, direction);
        }
        if (next != null) {
            return next;
        }
        ArrayList<View> focusables = mTempList;
        try {
            focusables.clear();
```

```
            root.addFocusables(focusables, direction);
            if (!focusables.isEmpty()) {
                next = findNextFocus(root, focused, focusedRect, direction, focusables);
            }
        } finally {
            focusables.clear();
        }
        return next;
    }
```

如果当前焦点不为空，则先去读取上层设置的specifiedFocusId。

kotlin 复制代码

```
    private View findNextUserSpecifiedFocus(ViewGroup root, View focused, int direction) {
        // check for user specified next focus
        View userSetNextFocus = focused.findUserSetNextFocus(root, direction);
        if (userSetNextFocus != null && userSetNextFocus.isFocusable()
                && (!userSetNextFocus.isInTouchMode()
                        || userSetNextFocus.isFocusableInTouchMode())) {
            return userSetNextFocus;
        }
        return null;
    }

    /**
     * If a user manually specified the next view id for a particular direction,
     * use the root to look up the view.
     * @param root The root view of the hierarchy containing this view.
     * @param direction One of FOCUS_UP, FOCUS_DOWN, FOCUS_LEFT, FOCUS_RIGHT, FOCUS_FORWARD,
     * or FOCUS_BACKWARD.
     * @return The user specified next view, or null if there is none.
     */
    View findUserSetNextFocus(View root, @FocusDirection int direction) {
        switch (direction) {
            case FOCUS_LEFT:
                if (mNextFocusLeftId == View.NO_ID) return null;
                return findViewInsideOutShouldExist(root, mNextFocusLeftId);
            case FOCUS_RIGHT:
                if (mNextFocusRightId == View.NO_ID) return null;
                return findViewInsideOutShouldExist(root, mNextFocusRightId);
            case FOCUS_UP:
                if (mNextFocusUpId == View.NO_ID) return null;
                return findViewInsideOutShouldExist(root, mNextFocusUpId);
            case FOCUS_DOWN:
                if (mNextFocusDownId == View.NO_ID) return null;
                return findViewInsideOutShouldExist(root, mNextFocusDownId);
            case FOCUS_FORWARD:
                if (mNextFocusForwardId == View.NO_ID) return null;
                return findViewInsideOutShouldExist(root, mNextFocusForwardId);
```

```
            case FOCUS_BACKWARD: {
                if (mID == View.NO_ID) return null;
                final int id = mID;
                return root.findViewByPredicateInsideOut(this, new Predicate<View>() {
                    @Override
                    public boolean apply(View t) {
                        return t.mNextFocusForwardId == id;
                    }
                });
            }
        }
        return null;
    }
```

有木有很熟悉findUserSetNextFocus的实现，如果上层给View/ViewGroup设置了setNextDownId/setNextLeftId/...，则android系统会从root view树中查找此id对应的view并返回，此分支寻焦逻辑结束。**可见为View/ViewGroup设置了nextDownId,nextLeftId等属性可定向分配焦点。** 若没有设置上面的属性，走下面的流程

ini 复制代码

```
    ArrayList<View> focusables = mTempList;
    try {
        focusables.clear();
        root.addFocusables(focusables, direction);
        if (!focusables.isEmpty()) {
            next = findNextFocus(root, focused, focusedRect, direction, focusables);
        }
    } finally {
        focusables.clear();
    }
    return next;
```

ini 复制代码

```
@Override
public void addFocusables(ArrayList<View> views, int direction, int focusableMode) {
    final int focusableCount = views.size();

    final int descendantFocusability = getDescendantFocusability();

    if (descendantFocusability != FOCUS_BLOCK_DESCENDANTS) {
        if (shouldBlockFocusForTouchscreen()) {
            focusableMode |= FOCUSABLES_TOUCH_MODE;
        }

        final int count = mChildrenCount;
        final View[] children = mChildren;
```

```
            for (int i = 0; i < count; i++) {
                final View child = children[i];
                if ((child.mViewFlags & VISIBILITY_MASK) == VISIBLE) {
                    child.addFocusables(views, direction, focusableMode);
                }
            }
        }
        ...
    }
```

逻辑也出来了，用一个集合存储那些focusable并且可见的view，注意到addFocusables方法调用者是root，也就是整个view树都会进行遍历。FOCUS_BLOCK_DESCENDANTS这个属性也很熟悉，如果为ViewGroup设置该属性则其子view都不会统计到focusable范围中。 最终findNextFocus方法:

csharp 复制代码

```
private View findNextFocus(ViewGroup root, View focused, Rect focusedRect,
        int direction, ArrayList<View> focusables) {
    if (focused != null) {
        if (focusedRect == null) {
            focusedRect = mFocusedRect;
        }
        // fill in interesting rect from focused
        focused.getFocusedRect(focusedRect);
        root.offsetDescendantRectToMyCoords(focused, focusedRect);
    } else {
      ...
    }

    switch (direction) {
        case View.FOCUS_FORWARD:
        case View.FOCUS_BACKWARD:
            return findNextFocusInRelativeDirection(focusables, root, focused, focusedRect,
                    direction);
        case View.FOCUS_UP:
        case View.FOCUS_DOWN:
        case View.FOCUS_LEFT:
        case View.FOCUS_RIGHT:
            return findNextFocusInAbsoluteDirection(focusables, root, focused,
                    focusedRect, direction);
        default:
            throw new IllegalArgumentException("Unknown direction: " + direction);
    }
}
```

正常流程到这里focused不为空，focusedRect为空，键值一般为上下左右方向按键，因此走绝对方向。

scss 复制代码

```
View findNextFocusInAbsoluteDirection(ArrayList<View> focusables, ViewGroup root, View focus
        Rect focusedRect, int direction) {
    // initialize the best candidate to something impossible
    // (so the first plausible view will become the best choice)
    mBestCandidateRect.set(focusedRect);
    switch(direction) {
        case View.FOCUS_LEFT:
            mBestCandidateRect.offset(focusedRect.width() + 1, 0);
            break;
        case View.FOCUS_RIGHT:
            mBestCandidateRect.offset(-(focusedRect.width() + 1), 0);
            break;
        case View.FOCUS_UP:
            mBestCandidateRect.offset(0, focusedRect.height() + 1);
            break;
        case View.FOCUS_DOWN:
            mBestCandidateRect.offset(0, -(focusedRect.height() + 1));
    }

    View closest = null;

    int numFocusables = focusables.size();
    for (int i = 0; i < numFocusables; i++) {
        View focusable = focusables.get(i);

        // only interested in other non-root views
        if (focusable == focused || focusable == root) continue;

        // get focus bounds of other view in same coordinate system
        focusable.getFocusedRect(mOtherRect);
        root.offsetDescendantRectToMyCoords(focusable, mOtherRect);

        if (isBetterCandidate(direction, focusedRect, mOtherRect, mBestCandidateRect)) {
            mBestCandidateRect.set(mOtherRect);
            closest = focusable;
        }
    }
    return closest;
}
```

核心算法就在这里了，遍历focusables集合，拿出每个view的rect属性和当前focused view的rect进行“距离”的比较，最终得到“距离”最近的候选者并返回。至此，整个寻焦逻辑结束。感兴趣的同学可研究内部比较的算法。

在整个寻焦过程中，我们发现focusSearch方法是public的，因此可在view树的某个节点复写此方法并返回期望view从而达到“拦截”默认寻焦的流程。同理，addFocusables方法也是public的，复写此方法可缩小比较view的范围，提高效率。

2.4 requestFocus

最后一步是请求焦点，根据代码条件会出现两个分支，一个是调用两个参数的requestFocus(int direction, Rect previouslyFocusedRect)，此方法来自View但是ViewGroup有override；另一个是一个参数的requestFocus(int direction)，来自View且声明为final。所以就要分上一步寻找到的focus目标是View还是ViewGroup两种情况进行分析。

如果是View，来看View的requestFocus源码

java 复制代码

```
public final boolean requestFocus(int direction) {
    return requestFocus(direction, null);
}

public boolean requestFocus(int direction, Rect previouslyFocusedRect) {
    return requestFocusNoSearch(direction, previouslyFocusedRect);
}

private boolean requestFocusNoSearch(int direction, Rect previouslyFocusedRect) {
    // need to be focusable
    if ((mViewFlags & FOCUSABLE_MASK) != FOCUSABLE ||
            (mViewFlags & VISIBILITY_MASK) != VISIBLE) {
        return false;
    }

    // need to be focusable in touch mode if in touch mode
    if (isInTouchMode() &&
        (FOCUSABLE_IN_TOUCH_MODE != (mViewFlags & FOCUSABLE_IN_TOUCH_MODE))) {
           return false;
    }

    // need to not have any parents blocking us
    if (hasAncestorThatBlocksDescendantFocus()) {
        return false;
    }

    handleFocusGainInternal(direction, previouslyFocusedRect);
    return true;
}
```

看来最终都会走到requestFocusNoSearch方法，而且其中的核心方法一看就知道是handleFocusGainInternal。

scss 复制代码

```
void handleFocusGainInternal(@FocusRealDirection int direction, Rect previouslyFocusedRect)
    if (DBG) {
        System.out.println(this + " requestFocus()");
    }

    if ((mPrivateFlags & PFLAG_FOCUSED) == 0) {
        mPrivateFlags |= PFLAG_FOCUSED;

        View oldFocus = (mAttachInfo != null) ? getRootView().findFocus() : null;

        if (mParent != null) {
            mParent.requestChildFocus(this, this);
        }

        if (mAttachInfo != null) {
            mAttachInfo.mTreeObserver.dispatchOnGlobalFocusChange(oldFocus, this);
        }

        onFocusChanged(true, direction, previouslyFocusedRect);
        refreshDrawableState();
    }
}
```

大致也分为几步：

1. requestChildFocus 将焦点通过递归传递给父View，父View更新mFocused属性，属性值就是其中包含focused view的子View/ViewGroup，这样focused view tree就更新了。
2. 各种回调focus状态给各个监听器，我们常用的OnFocusChangedListener就是其中一种。
3. refreshDrawableState更新drawable状态。

再来看如果是ViewGroup，ViewGroup的requestFocus源码如下：

java 复制代码

```
/**
 * {@inheritDoc}
 *
 * Looks for a view to give focus to respecting the setting specified by
 * {@link #getDescendantFocusability()}.
 *
 * Uses {@link #onRequestFocusInDescendants(int, android.graphics.Rect)} to
 * find focus within the children of this group when appropriate.
 *
```

```
     * @see #FOCUS_BEFORE_DESCENDANTS
     * @see #FOCUS_AFTER_DESCENDANTS
     * @see #FOCUS_BLOCK_DESCENDANTS
     * @see #onRequestFocusInDescendants(int, android.graphics.Rect)
     */
    @Override
    public boolean requestFocus(int direction, Rect previouslyFocusedRect) {
        if (DBG) {
            System.out.println(this + " ViewGroup.requestFocus direction="
                    + direction);
        }
        int descendantFocusability = getDescendantFocusability();

        switch (descendantFocusability) {
            case FOCUS_BLOCK_DESCENDANTS:
                return super.requestFocus(direction, previouslyFocusedRect);
            case FOCUS_BEFORE_DESCENDANTS: {
                final boolean took = super.requestFocus(direction, previouslyFocusedRect);
                return took ? took : onRequestFocusInDescendants(direction, previouslyFocusedRe
            }
            case FOCUS_AFTER_DESCENDANTS: {
                final boolean took = onRequestFocusInDescendants(direction, previouslyFocusedRe
                return took ? took : super.requestFocus(direction, previouslyFocusedRect);
            }
            default:
                throw new IllegalStateException("descendant focusability must be "
                        + "one of FOCUS_BEFORE_DESCENDANTS, FOCUS_AFTER_DESCENDANTS, FOCUS_BLOCK
                        + "but is " + descendantFocusability);
        }
    }
```

这里必须要清楚descendantFocusability属性值 看注释结合代码逻辑可知，此属性决定requestFocus事件的传递顺序。

- FOCUS_BLOCK_DESCENDANTS：子view不处理，本view直接处理，这样就走到了上述View的requestFocus逻辑。
- FOCUS_BEFORE_DESCENDANTS：本view先处理，如果消费了事件，子View不再处理，反之再交给子View处理。
- FOCUS_AFTER_DESCENDANTS：同上，只不过顺序变为子View先处理。

那这个值的默认值是什么呢？其实在ViewGroup的构造方法中调用了initViewGroup方法，在这个方法中默认设置了descendantFocusability的属性为FOCUS_BEFORE_DESCENDANTS，也就是本View先处理。 最后看下onRequestFocusInDescendants的源码:

ini 复制代码

```
/**
 * Look for a descendant to call {@link View#requestFocus} on.
 * Called by {@link ViewGroup#requestFocus(int, android.graphics.Rect)}
 * when it wants to request focus within its children.  Override this to
 * customize how your {@link ViewGroup} requests focus within its children.
 * @param direction One of FOCUS_UP, FOCUS_DOWN, FOCUS_LEFT, and FOCUS_RIGHT
 * @param previouslyFocusedRect The rectangle (in this View's coordinate system)
 *        to give a finer grained hint about where focus is coming from.  May be null
 *        if there is no hint.
 * @return Whether focus was taken.
 */
@SuppressWarnings({"ConstantConditions"})
protected boolean onRequestFocusInDescendants(int direction,
        Rect previouslyFocusedRect) {
    int index;
    int increment;
    int end;
    int count = mChildrenCount;
    if ((direction & FOCUS_FORWARD) != 0) {
        index = 0;
        increment = 1;
        end = count;
    } else {
        index = count - 1;
        increment = -1;
        end = -1;
    }
    final View[] children = mChildren;
    for (int i = index; i != end; i += increment) {
        View child = children[i];
        if ((child.mViewFlags & VISIBILITY_MASK) == VISIBLE) {
            if (child.requestFocus(direction, previouslyFocusedRect)) {
                return true;
            }
        }
    }
    return false;
}
```

由此可知，根据方向键决定遍历顺序，遍历过程只要有一个子View处理了焦点事件便立即返回，整个流程结束。很多常用的类都复写过此方法，比如 RecyclerView，ViewPager等等。

三.总结

整篇文章源码分析挺多的，主要是为了找到可对寻焦逻辑有影响的关键节点，实际上也是Android系统为上层开的"口子"，方便根据实际需求改变默认行为。

- 消费按键事件，事件在传递过程中如果被消费便不会走寻焦逻辑，这是一种拦截焦点的思路。
- focusSearch，上层可复写此方法返回特定view，来直接中断寻焦流程。RecyclerView就复写了这个方法，并且为LayoutManager留了一个onInterceptFocusSearch回调，将拦截事件转发给LayoutManager来实现特定的拦截焦点逻辑，比如常用的列表边界拦截。
- 为View/ViewGroup设置了nextDownId,nextLeftId等属性可定向分配焦点。
- addFocusables，复写此方法可缩小/扩大比较view的候选者，间接影响焦点的分配。

分类： Android 标签： 源码