Material Design控件使用

什么是Material Design

Material Design，中文名：材料设计语言，是由Google推出的全新的设计语言，谷歌表示，这种设计语言旨在为手机、平板电脑、台式机和“其他平台”提供更一致、更广泛的“外观和感觉”。

> Tip: 在Android5.0最引人注意的就是MaterialDesign设计风格 Material Design：谷歌拿出媲美苹果的设计 过去Google的产品线，每一个都相当的独立，在产品的设计上反映得尤为明显，甚至不必看产品设计，只要看一下Google每款产品的LOGO都能发现许多不同风格的设计。这种混乱难以体现出Google的风格，如果Google的风格不是混乱和无序的话。 2011年，拉里·佩奇成为Google CEO之后，他管理公司的政策从过去的自由、放任，变为紧密、整合。根据我们的报道，在Google发展的早期，因为鼓励观点的碰撞，结果发展成一种不留情面的争论氛围，高管之间冲突不断，甚至会拒绝合作。佩奇决心要改变公司的氛围，2013年2月，在纳帕山谷的卡内罗斯客栈酒店里，他对所有Google高管说，Google要实现10倍的发展速度，要用全新的方法来解决问题，因此高管之间要学会合作。从现在开始，Google要对争斗零容忍。

- Material Design不再让像素处于同一个平面，而是让它们按照规则处于空间当中，具备不同的维度
- Material Design还规范了Android的运动元素
- Material Design更加倾向于用色彩来提示

Google 发布的Material Design语言更像是一套界面设计标准

Z轴

在Material Design主题当中给UI元素引入了高度的概念，视图的高度由属性Z来表示，决定了阴影的视觉效果，Z越大，阴影就越大且越柔和。但是Z值并不会影响视图的大小。

视图的Z值由两个分量表示：

1. Elevation：静态的分量
2. Translation：用于动画的动态的分量

> Z值的计算公式为： Z=elevation+translationZ

- 通过在xml布局文件当中给一个视图设置android:elevation属性，来设置视图的高度。当然我们也可以在代码当中使用View.setElevation()来给视图设置高度。
- 还可以在代码当中设置视图的translationZ分量:View.setTranslationZ()。
- 新的ViewPropertyAnimator.z()以及ViewPropertyAnimator.translationZ()方法能够很容易的改变视图的高度。关于这个动画的更多信息，参考ViewPropertyAnimator以及PropertyAnimation相关的API。
- 还可以给视图设置Android:StateListAnimator属性来设置视图的状态改变动画，比如当点击按钮的时候改变其translationZ分量的值。
- Z值的单位是dp

Material Design的一些theme

- Theme.MaterialComponents
- Theme.MaterialComponents.NoActionBar
- Theme.MaterialComponents.Light
- Theme.MaterialComponents.Light.NoActionBar
- Theme.MaterialComponents.Light.DarkActionBar

- Theme.MaterialComponents.DayNight
- Theme.MaterialComponents.DayNight.NoActionBar
- Theme.MaterialComponents.DayNight.DarkActionBar

Toolbar

> https://developer.android.com/reference/com/google/android/material/packages

Toolbar 是在 Android 5.0 开始推出的一个 Material Design 风格的导航控件 ， Google 非常推荐大家使用 **Toolbar** 来作为Android客户端的导航栏，以此来取代之前的 **Actionbar** 。与 **Actionbar** 相比，**Toolbar** 明显要灵活的多。它不像 **Actionbar** 一样，一定要固定在Activity的顶部，而是可以放到界面的任意位置。除此之外，在设计 **Toolbar** 的时候，Google也留给了开发者很多可定制修改的余地，这些可定制修改的属性在API文档中都有详细介绍，如：

- 设置导航栏图标
- 设置App的logo
- 支持设置标题和子标题
- 支持添加一个或多个的自定义控件
- 支持Action Menu

在布局文件中设置

- app:navigationIcon 设置navigation button
- app:logo 设置logo 图标
- app:title 设置标题
- app:titleTextColor 设置标题文字颜色
- app:subtitle 设置副标题
- app:subtitleTextColor 设置副标题文字颜色
- app:popupTheme Reference to a theme that should be used to inflate popups - shown by widgets in the toolbar.
- app:titleTextAppearance 设置title text 相关属性，如：字体,颜色，大小等等
- app:subtitleTextAppearance 设置subtitletext相关属性，如：字体,颜色，大小等等
- app:logoDescription logo 描述
- android:background Toolbar 背景
- android:theme 主题

FloatingActionButton

FloatingActionButton是一个继承ImageView悬浮的动作按钮，经常用在一些比较常用的操作中，一个页面尽量只有一个FloatingActionButton，否则会给用户一种错乱的感觉！FloatingActionButton的大小分为标准型(56dp)和迷你型(40dp)，google是这么要求的，如果你不喜欢可以自己设置其他大小。并且对于图标进行使用materialDesign的图标，大小在24dp为最佳！

- android:src 设置相应图片
- app:backgroundTint 设置背景颜色
- app:borderWidth 设置边界的宽度。如果不设置0dp，那么在4.1的sdk上FAB会显示为正方形，而且在5.0以后的sdk没有阴影效果。
- app:elevation 设置阴影效果
- app:pressedTranslationZ 按下时的阴影效果
- app:fabSize 设置尺寸normal对应56dp，mini对应40dp
- app:layout_anchor 设置锚点，相对于那个控件作为参考
- app:layout_anchorGravity 设置相对锚点的位置 top/bottom之类的参数
- app:rippleColor 设置点击之后的涟漪颜色

Snackbar

Snackbar就是一个类似Toast的快速弹出消息提示的控件，手机上显示在底部，大屏幕设备显示在左侧。但是在显示上比Toast丰富，也提供了于用户交互的接口

BottomAppBar

Material Design的一个重要特征是设计 BottomAppBar。可适应用户不断变化的需求和行为，So，BottomAppBar是一个从标准物质指导的演变。它更注重功能，增加参与度，并可视化地锚定UI

> 要求Activity的主题必须是MaterialComponents的主题

- style="@style/Widget.MaterialComponents.BottomAppBar"
- 可以通过FabAlignmentMode，FabCradleMargin，FabCradleRoundedCornerRadius和FabCradleVerticalOffset来控制FAB的放置；
- （FabAlignmentMode）可以设置为中心或结束。如果FabAttached设置为True，那么Fab将被布置为连接到BottomAppBar；
- FabCradleMargin是设置FAB和BottomAppBar之间的间距，改变这个值会增加或减少FAB和BottomAppBar之间的间距；
- FabCradleRoundedCornerRadius指定切口周围角的圆度；
- FabCradleVerticalOffset指定FAB和BottomAppBar之间的垂直偏移量。如果fabCradleVerticalOffset为0，则FAB的中心将与BottomAppBar的顶部对齐。

NavigationView

NavigationView表示应用程序的标准导航菜单，菜单内容可以由菜单资源文件填充。NavigationView通常放在DrawerLayout中，可以实现侧滑效果的UI。DrawerLayout布局可以有3个子布局，第一个布局必须是主界面且不可以不写，其他2个子布局就是左、右两个侧滑布局，左右两个侧滑布局可以只写其中一个

- android:layout_gravity 值为start则是从左侧滑出，值为end则是从右侧滑出
- app:menu NavigationView是通过菜单形式在布局中放置元素的，值为自己创建的菜单文件
- app:headerLayout 给NavigationView设置头文件
- app:itemTextColor 设置菜单文字的颜色
- app:itemIconTint 设置菜单图标的颜色
- app:itemBackground 设置菜单背景的颜色

BottomNavigationView

BottomNavigationView创建底部导航栏，用户只需轻点一下即可轻松浏览和切换顶级内容视图，当项目有3到5个顶层（底部）目的地导航到时，可以使用此模式。

1. 创建一个菜单资源，最多5个导航目标（BottomNavigationView不支持超过5个项目）；
2. 在内容下面放置BottomNavigationView；
3. 将BottomNavigationView上的app：menu属性设置为菜单资源；
4. 设置选择监听事件setOnNavigationItemSelectedListener(...)
5. 通过app:itemIconTint和app:itemTextColor设置响应用户点击状态。包括Icon以及字体颜色

```
<!-- 定义bottom_navigation_colors-->
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:color="@color/colorAccent" android:state_checked="true" />
    <item android:color="@android:color/white" android:state_checked="false" />
</selector>
<!-- 添加引用
app:itemIconTint="@drawable/bottom_navigation_colors"
app:itemTextColor="@drawable/bottom_navigation_colors"
-->
```

- 设置style

> - style="@style/Widget.Design.BottomNavigationView" 默认的材质BottomNavigationView样式由更新的颜色，文字大小和行为组成。默认的BottomNavigationView具有白色背景以及带有颜色的图标和文本colorPrimary
> - style="@style/Widget.MaterialComponents.BottomNavigationView.Colored" 此样式继承默认样式，但将颜色设置为不同的属性。 使用彩色样式获取带有colorPrimary背景的底部导航栏，并为图标和文本颜色添加白色阴影。
> - style="@style/Widget.Design.BottomNavigationView" 如果希望底部导航栏具有旧行为，可以在BottomNavigationView上设置此样式。 但是，还是建议尽可能使用更新后的Material style

DrawerLayout

DrawerLayout是V4 Library包中实现了侧滑菜单效果的控件，可以说drawerLayout是因为第三方控件如MenuDrawer等的出现之后，google借鉴而出现的产物。drawerLayout分为侧边菜单和主内容区两部分，侧边菜单可以根据手势展开与隐藏（drawerLayout自身特性），主内容区的内容可以随着菜单的点击而变化

1. 抽屉式导航栏是显示应用主导航菜单的界面面板。当用户触摸应用栏中的抽屉式导航栏图标 或用户从屏幕的左边缘滑动手指时，就会显示抽屉式导航栏
2. 抽屉式导航栏图标会显示在使用 DrawerLayout 的所有顶级目的地上。顶级目的地是应用的根级目的地。它们不会在应用栏中显示向上按钮。
3. 要添加抽屉式导航栏，请先声明 DrawerLayout 为根视图。在 DrawerLayout 内，为主界面内容以及包含抽屉式导航栏内容的其他视图添加布局。
4. 例如，以下布局使用含有两个子视图的 DrawerLayout：包含主内容的 NavHostFragment 和适用于抽屉式导航栏内容的 NavigationView

```
<?xml version="1.0" encoding="utf-8"?>
    <!-- Use DrawerLayout as root container for activity -->
    <android.support.v4.widget.DrawerLayout
xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:app="http://schemas.android.com/apk/res-auto"
        android:id="@+id/drawer_layout"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:fitsSystemWindows="true">

<!-- Layout to contain contents of main body of screen (drawer will slide over
this) -->
        <fragment
            android:name="androidx.navigation.fragment.NavHostFragment"
            android:id="@+id/nav_host_fragment"
            android:layout_width="match_parent"
```

```
            android:layout_height="match_parent"
            app:defaultNavHost="true"
            app:navGraph="@navigation/nav_graph" />

<!-- Container for contents of drawer - use NavigationView to make configuration
easier -->
        <android.support.design.widget.NavigationView
            android:id="@+id/nav_view"
            android:layout_width="wrap_content"
            android:layout_height="match_parent"
            android:layout_gravity="start"
            android:fitsSystemWindows="true" />


</android.support.v4.widget.DrawerLayout>
```

CardView

Material Design中有一种很个性的设计概念：卡片式设计（Cards），Cards拥有自己独特的UI特征,CardView 继承自FrameLayout类，并且可以设置圆角和阴影，使得控件具有立体性，也可以包含其他的布局容器和控件

- card_view:cardCornerRadius 设置角半径
- CardView.setRadius 要在代码中设置角半径，请使用
- card_view:cardBackgroundColor 设置卡片的背景颜色
- card_view:cardElevation 设置Z轴阴影高度
- card_view:cardMaxElevation 设置Z轴阴影最大高度
- card_view:cardUseCompatPadding 设置内边距
- card_view:cardPreventCornerOverlap 在v20和之前的版本中添加内边距，这个属性是为了防止卡片内容和边角的重叠

Chips

Chip代表一个小块中的复杂实体，如联系人。 它是一个圆形按钮，由一个标签，一个可选的芯片图标和一个可选的关闭图标组成。 如果Chip可检查，则可以点击或切换Chip。

Chip可以被放置在ChipGroup中，其可以被配置为将其Chip布置在单个水平线中或者重新排列在多个线上。如果一个ChipGroup包含可检查的Chip，那么它也可以控制其一组Chip是否单选。

也可以在需要Drawable的上下文中直接使用独立的ChipDrawable

主题设置

- tyle="@style/Widget.MaterialComponents.Chip.Entry"： 默认在末尾展示删除按钮；点击后前面展示选中图标，有选中状态 通常可以作为 chipDrawable 使用，比如在填选邮件收件人时可以使用
- style="@style/Widget.MaterialComponents.Chip.Filter"： 使用 style="@style/Widget.MaterialComponents.Chip.Filter" 初始状态下， 不展示前后图标 点击之后会展示前面的选中图标，并且具有选中状态 通常应用在 ChipGroup 中
- style="@style/Widget.MaterialComponents.Chip.Choice"： 默认不展示前后的图标，但点击后有选中状态 通常用在 ChipGroup 中 , 通过 ChipGroup 的 singleSelection=true/false 属性可以实现单选或多选
- style="@style/Widget.MaterialComponents.Chip.Action"
- 不设置style时，默认使用上述style 默认前后图标都不展示，点击后没有选中状态
- Chip 的属性

类别	属性名称	具体作用
Shape	app:chipCornerRadius	圆角半径
Size	app:chipMinHeight	最小高度
Background	app:chipBackgroundColor	背景颜色
Border	app:chipStrokeColor	边线颜色
Border	app:chipStrokeWidth	边线宽度
Ripple	app:rippleColor	水波纹效果的颜色
Label	android:text	文本内容
Label	android:textColor	修改文本颜色
Label	android:textAppearance	字体样式
Chip Icon	app:chipIconVisible	前面的图标是否展示
Chip Icon	app:chipIcon	chip中文字前面的图标
Chip Icon	app:chipIconTint	文字前面的图标着色
Chip Icon	app:chipIconSize	chip中文字前面的图标
Close Icon	app:closeIconVisible	chip中文字后面的关闭按钮是否可见
Close Icon	app:closeIcon	chip中文字后面的关闭图标
Close Icon	app:closeIconSize	文字后面的关闭图标的大小
Close Icon	app:closeIconTint	文字后面的着色
Checkable	app:checkable	是否可以被选中
Checked Icon	app:checkedIconVisible	选中状态的图标是否可见
Checked Icon	app:checkedIcon	选中状态的图标
Motion	app:showMotionSpec	动效?
Motion	app:hideMotionSpec	动效?
Paddings	app:chipStartPadding	chip左边距
Paddings	app:chipEndPadding	chip右边距
Paddings	app:iconStartPadding	chipIcon的左边距
Paddings	app:iconEndPadding	chipIcon的右边距
Paddings	app:textStartPadding	文本左边距
Paddings	app:textEndPadding	文本右边距
Paddings	app:closeIconStartPadding	关闭按钮的做左边距
Paddings	app:closeIconEndPadding	关闭按钮的右边距

Material Button

Material Button是具有更新视觉样式的可自定义按钮组件，且内部内置了多种样式。

TextInputLayout&TextInputEditText

- android:hint 提示文字
- app:counterEnabled 是否添加计数功能，默认是false
- app:counterMaxLength 最大的输入数量（如果计数显示的话，影响显示）
- app:errorEnabled 是否有错误提示
- app:errorTextAppearance 设置错误提示的文字样式
- app:hintAnimationEnabled 是否设置提示文字的动画
- app:hintEnabled 是否启动浮动标签功能，如果不启用的话，所有提示性信息都将在Edittext中显示
- app:hintTextAppearance 设置提示性文字的样式
- app:passwordToggleContentDescription 该功能是为Talkback或其他无障碍功能提供
- app:passwordToggleEnabled 是否显示后面的提示图片
- app:passwordToggleDrawable 替换后面的提示图片
- app:passwordToggleTint 给后面的提示图片设置颜色
- app:passwordToggleTintMode 控制密码可见开关图标的背景颜色混合模式
- app:counterOverflowTextAppearance 设置计算器越位后的文字颜色和大小(通过style进行设置的)
- app:counterTextAppearance 设置正常情况下的计数器文字颜色和大小(通过style进行设置的)

TabLayout

- app:tabBackground　设置TableLayout的背景色
- app:tabTextColor　设置未被选中时文字的颜色
- app:tabSelectedTextColor　设置选中时文字的颜色
- app:tabIndicatorColor　设置滑动条的颜色
- app:tabTextAppearance="@android:style/TextAppearance.Large" 设置TableLayout的文本主题，无法通过textSize来设置文字大小，只能通过主题来设定
- app:tabMode="scrollable" 设置TableLayout可滑动，当页数较多时，一个界面无法呈现所有的导航标签，此时就必须要用。

Feature	Relevant attributes
Size	app:tabMinWidth app:tabMaxWidth
Scroll	app:tabMode
Centered	app:tabGravity
Background	app:tabBackground
Icon	app:tabIconTint app:tabIconTintMode
Label	app:tabInlineLabel app:tabTextAppearance app:tabTextColor app:tabSelectedTextColor
Indicator	app:tabIndicatorColor app:tabIndicatorHeight app:tabIndicator app:tabIndicatorGravity app:tabIndicatorFullWidth
Position	app:tabContentStart app:tabPaddingStart app:tabPaddingTop app:tabPaddingEnd app:tabPaddingBottom app:tabPadding
Ripple	app:tabRippleColor

Bottom Sheet

Bottom Sheet是Design Support Library23.2 版本引入的一个类似于对话框的控件，可以暂且叫做底部弹出框吧。 Bottom Sheet中的内容默认是隐藏起来的，只显示很小一部分，可以通过在代码中设置其状态或者手势操作将其完全展开，或者完全隐藏，或者部分隐藏

有两种类型的Bottom Sheet：

1. Persistent bottom sheet :- 通常用于显示主界面之外的额外信息，它是主界面的一部分，只不过默认被隐藏了，其深度（elevation）跟主界面处于同一级别；还有一个重要特点是在Persistent bottom sheet打开的时候，主界面仍然是可以操作的，其实Persistent bottom sheet不能算是一个控件，因为它只是一个普通的布局在CoordinatorLayout这个布局之下所表现出来的特殊行为。所以其使用方式跟普通的控件也很不一样，它必须在CoordinatorLayout中，并且是CoordinatorLayout的直接子view
    - **app:layout_behavior="@string/bottom_sheet_behavior"，定义了这个属性就相当于告诉了CoordinatorLayout这个布局是一个bottom sheet，它的显示和交互都和普通的view不同。@string/bottom_sheet_behavior是一个定义在支持库中的字符串，等效于android.support.design.widget.BottomSheetBehavior**

Bottom Sheets具有五种状态：

- STATE_COLLAPSED： Bottom Sheets是可见的，但只显示可视（部分）高度。此状态通常是底部工作表的“静止位置”。可视高度由开发人员选择，应足以表明有额外的内容，允许用户触发某个动作或扩展Bottom Sheets；

- STATE_EXPANDED： Bottom Sheets是可见的并且它的最大高度并且不是拖拽或沉降；
- STATE_DRAGGING：用户主动向上或向下拖动Bottom Sheets；
- STATE_SETTLING： 拖动/轻扫手势后，Bottom Sheets将调整到特定高度。这将是可视高度，展开高度或0，以防用户操作导致底部表单隐藏；
- STATE_HIDDEN： Bottom Sheets隐藏。

如果已经在Activity使用CoordinatorLayout，添加底部表单很简单：

1. 将任何视图添加为CoordinatorLayout的直接子视图。
2. 通过添加以下xml属性来应用该行为
   app:layout_behavior="com.google.android.material.bottomsheet.BottomSheetBehavior"
3. 设置所需的行为标志
   - app:behavior_hideable：是否可以通过拖拽隐藏底部表单。
   - app:behavior_peekHeight：折叠状态的窥视高度。
   - app:behavior_skipCollapsed：如果底部表单可隐藏，并且设置为true，则表单不会处于折叠状态

bottom sheet的状态是通过BottomSheetBehavior来设置的，因此需要先得到BottomSheetBehavior对象，然后调用BottomSheetBehavior.setState()来设置状态，比如设置为折叠状态：

BottomSheetBehavior.setState(BottomSheetBehavior.STATE_COLLAPSED)； 我们还可以通过BottomSheetBehavior.getState() 来获得状态。

要监听bottom sheet的状态变化则使用setBottomSheetCallback方法，之所以需要监听是因为bottom sheet的状态还可以通过手势来改变

2. 模态bottom sheet :- 顾名思义，模态的bottom sheet在打开的时候会阻止和主界面的交互，并且在视觉上会在bottom sheet背后加一层半透明的阴影，使得看上去深度（elevation）更深

CoordinatorLayout

CoordinatorLayout(协调者布局)是在 Google IO/15 大会发布的，遵循Material 风格，包含在 support Library中，结合AppbarLayout, CollapsingToolbarLayout等 可 产生各种炫酷的效果

CoordinatorLayout是用来协调其子view并以触摸影响布局的形式产生动画效果的一个super-powered FrameLayout，其典型的子View包括：FloatingActionButton，SnackBar。注意：CoordinatorLayout是一个顶级父View

AppBarLayout

AppBarLayout是LinearLayout的子类，必须在它的子view上设置app:layout_scrollFlags属性或者是在代码中调用setScrollFlags()设置这个属性。

AppBarLayout的子布局有5种滚动标识：

- scroll：所有想滚动出屏幕的view都需要设置这个flag， 没有设置这个flag的view将被固定在屏幕顶部。
- enterAlways：这个flag让任意向下的滚动都会导致该view变为可见，启用快速“返回模式”。
- enterAlwaysCollapsed：假设你定义了一个最小高度（minHeight）同时enterAlways也定义了，那么view将在到达这个最小高度的时候开始显示，并且从这个时候开始慢慢展开，当滚动到顶部的时候展开完。
- exitUntilCollapsed：当你定义了一个minHeight，此布局将在滚动到达这个最小高度的时候折叠。
- snap：当一个滚动事件结束，如果视图是部分可见的，那么它将被滚动到收缩或展开。例如，如果视图只有底部25%显示，它将折叠。相反，如果它的底部75%可见，那么它将完全展开。

CollapsingToolbarLayout

CollapsingToolbarLayout作用是提供了一个可以折叠的Toolbar，它继承自FrameLayout，给它设置layout_scrollFlags，它可以控制包含在**CollapsingToolbarLayout**中的控件(如：ImageView、Toolbar)在响应**layout_behavior**事件时作出相应的**scrollFlags**滚动事件(移除屏幕或固定在屏幕顶端)。CollapsingToolbarLayout可以通过**app:contentScrim**设置折叠时工具栏布局的颜色，通过**app:statusBarScrim**设置折叠时状态栏的颜色。默认contentScrim是colorPrimary的色值，statusBarScrim是colorPrimaryDark的色值。

CollapsingToolbarLayout的子布局有3种折叠模式（Toolbar中设置的app:layout_collapseMode）

- off：默认属性，布局将正常显示，无折叠行为。
- pin：CollapsingToolbarLayout折叠后，此布局将固定在顶部。
- parallax：CollapsingToolbarLayout折叠时，此布局也会有视差折叠效果。

> 当CollapsingToolbarLayout的子布局设置了parallax模式时，我们还可以通过app:layout_collapseParallaxMultiplier设置视差滚动因子，值为：0~1。

NestedScrollView

在新版的support-v4兼容包里面有一个NestedScrollView控件，这个控件其实和普通的ScrollView并没有多大的区别，这个控件其实是Meterial Design中设计的一个控件，目的是跟MD中的其他控件兼容。应该说在MD中，RecyclerView代替了ListView，而NestedScrollView代替了ScrollView，他们两个都可以用来跟ToolBar交互，实现上拉下滑中ToolBar的变化。在NestedScrollView的名字中其实就可以看出他的作用了，Nested是嵌套的意思，而ToolBar基本需要嵌套使用。

FloatingActionButton

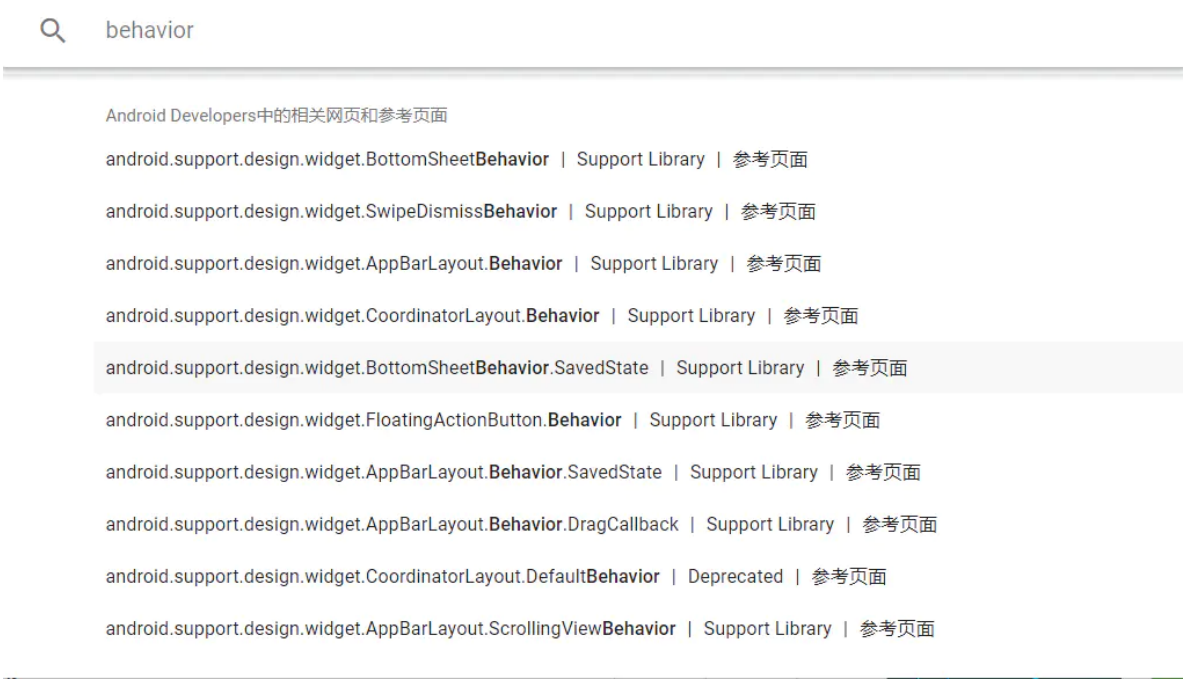
FloatingActionButton就是一个漂亮的按钮，其本质是一个ImageVeiw。有一点要注意，Meterial Design引入了Z轴的概念，就是所有的view都有了高度，他们一层一层贴在手机屏幕上，而FloatingActionButton的Z轴高度最高，它贴在所有view的最上面，没有view能覆盖它。

Behavior(注意暂时了解概念就行)

> Interaction behavior plugin for child views of CoordinatorLayout. A Behavior implements one or more interactions that a user can take on a child view. These interactions may include drags, swipes, flings, or any other gestures.

CoordinatorLayout中子View的交互行为，可以在CoordinatorLayout的子类中实现一个或多个交互，这些交互可能是拖动，滑动，闪动或任何其他手势。其实就是实现CoordinatorLayout内部控件的交互行为，可以在非侵入的方式实现相应的交互

Behavior只有是CoordinatorLayout的直接子View才有意义。只要将Behavior绑定到CoordinatorLayout的直接子元素上，就能对触摸事件（touch events）、window insets、measurement、layout以及嵌套滚动（nested scrolling）等动作进行拦截。Design Library的大多功能都是借助Behavior的大量运用来实现的。当然，Behavior无法独立完成工作，必须与实际调用的CoordinatorLayout子视图相绑定。具体有三种方式：通过代码绑定、在XML中绑定或者通过注释实现自动绑定。上面NestedScrollView中app:layout_behavior=”@string/appbar_scrolling_view_behavior”的Behavior是系统默认的，我们也可以根据自己的需求来自定义Behavior。



- Behavior里面回调的说明

> behavior的嵌套滚动都是依照一个相应的参考物，所以在自定义的时候一定要区分哪个是依照的View哪个是被观察的View，只有区分了这些才能更好的理解下面的内容，下面出现的所有child都是被观察的View，也就是xml中定义behavior的View

- **layoutDependsOn(CoordinatorLayout parent, View child, View dependency)** 表示是否给应用了Behavior 的View 指定一个依赖的布局
  - 参数1：coordinatorlayout对象
  - 参数2：child 被观察的View
  - 参数3：依赖变化的View（被观察的View）
- **onDependentViewChanged(CoordinatorLayout parent, View child, View dependency)** 当依赖的View发生变化的时候hi掉的方法
- **onStartNestedScroll(@NonNull CoordinatorLayout coordinatorLayout, @NonNull View child, @NonNull View directTargetChild, @NonNull View target, int axes, int type)** 当用户手指按下的时候，你是否要处理这次操作。当你确定要处理这次操作的时候，返回true；如果返回false的时候，就不会去响应后面的回调事件了。你想怎么滑就怎么话，我都不做处理。这里的(axes)滚动方向很重要，可以通过此参数判断滚动方向！
  - 参数3：直接目标，相当于能滑动的控件
  - 参数4：观察的View
  - 参数5：这个可以简单理解为滚动方向
    - ViewCompat#SCROLL_AXIS_HORIZONTAL 水平方向
    - ViewCompat#SCROLL_AXIS_VERTICAL 竖直方向
  - 参数6：这个参数是之后有的，如果你输入的类型不是TYPE_TOUCH那么就不会相应这个滚动
- **onNestedScrollAccepted(@NonNull CoordinatorLayout coordinatorLayout, @NonNull V child, @NonNull View directTargetChild, @NonNull View target, @ScrollAxis int axes, @NestedScrollType int type)** 当onStartNestedScroll准备处理这次滑动的时候(返回true的时候)，回调这个方法。可以在这个方法中做一些响应的准备工作！
- **onNestedPreScroll(@NonNull CoordinatorLayout coordinatorLayout, @NonNull View child, @NonNull View target, int dx, int dy, @NonNull int[] consumed, @NestedScrollType int type)** 当滚动开始执行的时候回调这个方法。

    - 参数4/参数5：用户x/y轴滚动的距离(注意这里是每一次都回调的啊！！！)
    - 参数6：处理滚动的距离的参数，内部维护着输出距离，假设用户滑动了100px,child 做了90px的位移，你需要把consumed [1] 的值改成90，这样coordinatorLayout就能知道只处理剩下的10px的滚动。其中consumed[0]代表x轴、consumed[1]代表y轴。可能你不理解这个问题，换个形象点的比喻，比如你开发某一个功能，但是你只会其中的90%那么怎么办呢？不能就不管了。好你找到了你的同事或者老大，让他去完成剩下的10%。这样问题就完美的解决了，是一个概念的！
- **onNestedScroll(@NonNull CoordinatorLayout coordinatorLayout, @NonNull View child, @NonNull View target, int dxConsumed, int dyConsumed, int dxUnconsumed, int dyUnconsumed, int type)** 上面这个方法结束的时候，coordinatorLayout处理剩下的距离，比如还剩10px。但是coordinatorLayout发现滚动2px的时候就已经到头了。那么结束其滚动，调用该方法，并将coordinatorLayout处理剩下的像素数作为参`(dxUnconsumed、dyUnconsumed)`传过来，这里传过来的就是 8px。参数中还会有coordinatorLayout处理过的像素数（dxConsumed、dyConsumed）。老大开始处理剩下的距离了！这个方法主要处理一些越界后的滚动。还是不懂对吧！还拿你们老大做比喻：比如上面还剩 10%的工作，这时老大处理了2%后发现已经可以上线了，于是老大结束了工作，并将处理剩下的内容（dxUnconsumed、dyUnconsumed）纪录下来，告诉你。老大处理了的内容（dxConsumed、dyConsumed）也告诉了你。
    - 参数4/参数5：当没有滚动到顶部或者底部的时候，x/y轴的滚动距离
    - 参数6/参数7：当滚动到顶部或者底部的时候，x/y轴的滚动距离
- onNestedPreFling(@NonNull CoordinatorLayout coordinatorLayout, @NonNull View child, @NonNull View target, float velocityX, float velocityY) 当手指松开发生惯性动作之前调用，这里提供了响应的速度，你可以根据速度判断是否需要进行折叠等一系列的操作，你要确定响应这个方法的话，返回true。
    - 参数4/参数5：代表相应的速度
- onStopNestedScroll(@NonNull CoordinatorLayout coordinatorLayout, @NonNull View child, @NonNull View target, int type) 停止滚动的时候回调的方法。当你不去响应Fling的时候会直接回调这个方法。在这里可以做一些清理工作。或者其他的内容。。。
- onLayoutChild(CoordinatorLayout parent, View child, int layoutDirection) 确定子View位置的方法，这个方法可以重新定义子View的位置(这里明确是设置behavior的那个View哦),例如下面这样
    - ViewCompat#LAYOUT_DIRECTION_LTR 视图方向从左到右
    - ViewCompat#LAYOUT_DIRECTION_RTL 视图方向从优到左

NestedScrollingChild

```
public interface NestedScrollingChild {
    /**
    * 启用或禁用嵌套滚动的方法，设置为true，并且当前界面的View的层次结构是支持嵌套滚动的
    * (也就是需要NestedScrollingParent嵌套NestedScrollingChild)，才会触发嵌套滚动。
    * 一般这个方法内部都是直接代理给NestedScrollingChildHelper的同名方法即可
    */
    void setNestedScrollingEnabled(boolean enabled);

    /**
    * 判断当前View是否支持嵌套滑动。一般也是直接代理给NestedScrollingChildHelper的同名方法即可
    */
    boolean isNestedScrollingEnabled();

    /**
    * 表示view开始滚动了,一般是在ACTION_DOWN中调用，如果返回true则表示父布局支持嵌套滚动。
```

```
    * 一般也是直接代理给NestedScrollingChildHelper的同名方法即可。这个时候正常情况会触发
Parent的onStartNestedScroll()方法
    */
    boolean startNestedScroll(@ScrollAxis int axes);

    /**
    * 一般是在事件结束比如ACTION_UP或者ACTION_CANCLE中调用,告诉父布局滚动结束。一般也是直
接代理给NestedScrollingChildHelper的同名方法即可
    */
    void stopNestedScroll();

    /**
    * 判断当前View是否有嵌套滑动的Parent。一般也是直接代理给NestedScrollingChildHelper的
同名方法即可
    */
    boolean hasNestedScrollingParent();

    /**
    * 在当前View消费滚动距离之后。通过调用该方法，把剩下的滚动距离传给父布局。如果当前没有发生
嵌套滚动，或者不支持嵌套滚动，调用该方法也没啥用。
    * 内部一般也是直接代理给NestedScrollingChildHelper的同名方法即可
    * dxConsumed：被当前View消费了的水平方向滑动距离
    * dyConsumed：被当前View消费了的垂直方向滑动距离
    * dxUnconsumed：未被消费的水平滑动距离
    * dyUnconsumed：未被消费的垂直滑动距离
    * offsetInWindow：输出可选参数。如果不是null，该方法完成返回时，
    * 会将该视图从该操作之前到该操作完成之后的本地视图坐标中的偏移量封装进该参数中，
offsetInWindow[0]水平方向，offsetInWindow[1]垂直方向
    * @return true：表示滚动事件分发成功,fasle: 分发失败
    */
    boolean dispatchNestedScroll(int dxConsumed, int dyConsumed,
            int dxUnconsumed, int dyUnconsumed, @Nullable int[] offsetInWindow);

    /**
    * 在当前View消费滚动距离之前把滑动距离传给父布局。相当于把优先处理权交给Parent
    * 内部一般也是直接代理给NestedScrollingChildHelper的同名方法即可。
    * dx：当前水平方向滑动的距离
    * dy：当前垂直方向滑动的距离
    * consumed：输出参数，会将Parent消费掉的距离封装进该参数consumed[0]代表水平方向，
consumed[1]代表垂直方向
    * @return true：代表Parent消费了滚动距离
    */
    boolean dispatchNestedPreScroll(int dx, int dy, @Nullable int[] consumed,
            @Nullable int[] offsetInWindow);

    /**
    *将惯性滑动的速度分发给Parent。内部一般也是直接代理给NestedScrollingChildHelper的同名
方法即可
    * velocityX：表示水平滑动速度
    * velocityY：垂直滑动速度
    * consumed：true：表示当前View消费了滑动事件，否则传入false
    * @return true：表示Parent处理了滑动事件
    */
    boolean dispatchNestedFling(float velocityX, float velocityY, boolean
consumed);

    /**
```

```
    * 在当前View自己处理惯性滑动前，先将滑动事件分发给Parent,一般来说如果想自己处理惯性的滑动事件，
    * 就不应该调用该方法给Parent处理。如果给了Parent并且返回true，那表示Parent已经处理了，自己就不应该再做处理。
    * 返回false，代表Parent没有处理，但是不代表Parent后面就不用处理了
    * @return true：表示Parent处理了滑动事件
    */
    boolean dispatchNestedPreFling(float velocityX, float velocityY);
}
```

```
public interface NestedScrollingChild2 extends NestedScrollingChild {

    boolean startNestedScroll(@ScrollAxis int axes, @NestedScrollType int type);

    void stopNestedScroll(@NestedScrollType int type);

    boolean hasNestedScrollingParent(@NestedScrollType int type);

    boolean dispatchNestedScroll(int dxConsumed, int dyConsumed,
            int dxUnconsumed, int dyUnconsumed, @Nullable int[] offsetInWindow,
            @NestedScrollType int type);

    boolean dispatchNestedPreScroll(int dx, int dy, @Nullable int[] consumed,
            @Nullable int[] offsetInWindow, @NestedScrollType int type);
}
```

```
public interface NestedScrollingChild3 extends NestedScrollingChild2 {

    void dispatchNestedScroll(int dxConsumed, int dyConsumed, int dxUnconsumed,
int dyUnconsumed,
            @Nullable int[] offsetInWindow, @ViewCompat.NestedScrollType int
type,
            @NonNull int[] consumed);
}
```

NestedScrollingParent

```
public interface NestedScrollingParent {
    /**
    * 当NestedScrollingChild调用方法startNestedScroll()时,会调用该方法。主要就是通过返回值告诉系统是否需要对后续的滚动进行处理
    * child：该ViewParen的包含NestedScrollingChild的直接子View，如果只有一层嵌套，和target是同一个View
    * target：本次嵌套滚动的NestedScrollingChild
    * nestedScrollAxes：滚动方向
    * @return
    * true:表示我需要进行处理，后续的滚动会触发相应的回到
    * false: 我不需要处理，后面也就不会进行相应的回调了
    */
    //child和target的区别，如果是嵌套两层如:Parent包含一个LinearLayout，LinearLayout里面才是NestedScrollingChild类型的View。这个时候，
    //child指向LinearLayout，target指向NestedScrollingChild；如果Parent直接就包含了NestedScrollingChild，
```

```
    //这个时候target和child都指向NestedScrollingChild
    boolean onStartNestedScroll(@NonNull View child, @NonNull View target,
@ScrollAxis int axes);

    /**
    * 如果onStartNestedScroll()方法返回的是true的话,那么紧接着就会调用该方法.它是让嵌套滚
动在开始滚动之前,
    * 让布局容器(viewGroup)或者它的父类执行一些配置的初始化的
    */
    void onNestedScrollAccepted(@NonNull View child, @NonNull View target,
@ScrollAxis int axes);

    /**
    * 停止滚动了,当子view调用stopNestedScroll()时会调用该方法
    */
    void onStopNestedScroll(@NonNull View target);

    /**
    * 当子view调用dispatchNestedScroll()方法时,会调用该方法。也就是开始分发处理嵌套滑动了
    * dxConsumed：已经被target消费掉的水平方向的滑动距离
    * dyConsumed：已经被target消费掉的垂直方向的滑动距离
    * dxUnconsumed：未被tagert消费掉的水平方向的滑动距离
    * dyUnconsumed：未被tagert消费掉的垂直方向的滑动距离
    */
    void onNestedScroll(@NonNull View target, int dxConsumed, int dyConsumed,
            int dxUnconsumed, int dyUnconsumed);

    /**
    * 当子view调用dispatchNestedPreScroll()方法是,会调用该方法。也就是在
NestedScrollingChild在处理滑动之前,
    * 会先将机会给Parent处理。如果Parent想先消费部分滚动距离，将消费的距离放入consumed
    * dx：水平滑动距离
    * dy：处置滑动距离
    * consumed：表示Parent要消费的滚动距离,consumed[0]和consumed[1]分别表示父布局在x和y
方向上消费的距离.
    */
    void onNestedPreScroll(@NonNull View target, int dx, int dy, @NonNull int[]
consumed);

    /**
    * 你可以捕获对内部NestedScrollingChild的fling事件
    * velocityX：水平方向的滑动速度
    * velocityY：垂直方向的滑动速度
    * consumed：是否被child消费了
    * @return
    * true:则表示消费了滑动事件
    */
    boolean onNestedFling(@NonNull View target, float velocityX, float
velocityY, boolean consumed);

    /**
    * 在惯性滑动距离处理之前，会调用该方法，同onNestedPreScroll()一样，也是给Parent优先处
理的权利
    * target：本次嵌套滚动的NestedScrollingChild
    * velocityX：水平方向的滑动速度
    * velocityY：垂直方向的滑动速度
    * @return
    * true：表示Parent要处理本次滑动事件，Child就不要处理了
```

```
    */
    boolean onNestedPreFling(@NonNull View target, float velocityX, float
velocityY);

    /**
    * 返回当前滑动的方向，一般直接通过
NestedScrollingParentHelper.getNestedScrollAxes()返回即可
    */
    @ScrollAxis
    int getNestedScrollAxes();
}
```

```
public interface NestedScrollingParent2 extends NestedScrollingParent {
    boolean onStartNestedScroll(@NonNull View child, @NonNull View target,
@ScrollAxis int axes,
            @NestedScrollType int type);

    void onNestedScrollAccepted(@NonNull View child, @NonNull View target,
@ScrollAxis int axes,
            @NestedScrollType int type);

    void onStopNestedScroll(@NonNull View target, @NestedScrollType int type);

    void onNestedScroll(@NonNull View target, int dxConsumed, int dyConsumed,
            int dxUnconsumed, int dyUnconsumed, @NestedScrollType int type);

    void onNestedPreScroll(@NonNull View target, int dx, int dy, @NonNull int[]
consumed,
            @NestedScrollType int type);

}
```

```
public interface NestedScrollingParent3 extends NestedScrollingParent2 {

    void onNestedScroll(@NonNull View target, int dxConsumed, int dyConsumed,
int dxUnconsumed,
            int dyUnconsumed, @ViewCompat.NestedScrollType int type, @NonNull
int[] consumed);

}
```

NestedScrollingParentHelper

```
public class NestedScrollingParentHelper {
//兼容NestedScrollingParent2的NestedScrollType
    private int mNestedScrollAxesTouch;
    private int mNestedScrollAxesNonTouch;


    public NestedScrollingParentHelper(@NonNull ViewGroup viewGroup) {
    }

    public void onNestedScrollAccepted(@NonNull View child, @NonNull View
target,
            @ScrollAxis int axes) {
```

```
            //默认就是TYPE_TOUCH
        onNestedScrollAccepted(child, target, axes, ViewCompat.TYPE_TOUCH);
    }


    public void onNestedScrollAccepted(@NonNull View child, @NonNull View
target,
            @ScrollAxis int axes, @NestedScrollType int type) {
        if (type == ViewCompat.TYPE_NON_TOUCH) {
            mNestedScrollAxesNonTouch = axes;
        } else {
            mNestedScrollAxesTouch = axes;
        }
    }


    @ScrollAxis
    public int getNestedScrollAxes() {
        return mNestedScrollAxesTouch | mNestedScrollAxesNonTouch;
    }

    public void onStopNestedScroll(@NonNull View target) {
        onStopNestedScroll(target, ViewCompat.TYPE_TOUCH);
    }


    public void onStopNestedScroll(@NonNull View target, @NestedScrollType int
type) {
        if (type == ViewCompat.TYPE_NON_TOUCH) {
            mNestedScrollAxesNonTouch = ViewGroup.SCROLL_AXIS_NONE;
        } else {
            mNestedScrollAxesTouch = ViewGroup.SCROLL_AXIS_NONE;
        }
    }
}
```

NestedScrollingChildHelper

```
/**
* 一个标准的嵌套滚动的框架策略,即如何控制了嵌套滚动的事件分发和一些逻辑处理
* 就是在当前Child的所有的祖辈ViewParent中勋在一个实现了NestedScroolingParent接口，并且支
持嵌套滚动(onStartNestedScroll()返回true)的。找到之后，在对应的分发方法中，将相关参数分发到
ViewParent中与之对应的处理方法中。而且为了兼容性，都是通过ViewParentCompat进行转发操作的
*/
public class NestedScrollingChildHelper {
    private ViewParent mNestedScrollingParentTouch;
    private ViewParent mNestedScrollingParentNonTouch;
    private final View mView;
    private boolean mIsNestedScrollingEnabled;
    private int[] mTempNestedScrollConsumed;


    //就是传一个View进来，该View就是实现了NestedScrollingChild接口的View类型
    public NestedScrollingChildHelper(@NonNull View view) {
        mView = view;
    }
```

```
    public void setNestedScrollingEnabled(boolean enabled) {
    //主要用于是给变量mIsNestedScrollingEnabled进行赋值，
    //记录否可以支持嵌套滚动的方式。可以看到，如果之前是支持嵌套滚动的话，
    //会先调用ViewCompat.stopNestedScroll(mView)停止当前滚动，然后进行赋值操作
        if (mIsNestedScrollingEnabled) {
            ViewCompat.stopNestedScroll(mView);
        }
        mIsNestedScrollingEnabled = enabled;
    }

    public boolean isNestedScrollingEnabled() {
    //判断当前View是否支持嵌套滚动
        return mIsNestedScrollingEnabled;
    }


    public boolean hasNestedScrollingParent() {
        return hasNestedScrollingParent(TYPE_TOUCH);
    }

    public boolean hasNestedScrollingParent(@NestedScrollType int type) {
    //获取嵌套滚动的Parent，也就是实现了NestedScrollingParent的ViewGroup
        return getNestedScrollingParentForType(type) != null;
    }

    public boolean startNestedScroll(@ScrollAxis int axes) {
        return startNestedScroll(axes, TYPE_TOUCH);
    }

    public boolean startNestedScroll(@ScrollAxis int axes, @NestedScrollType int
type) {
        if (hasNestedScrollingParent(type)) {
            // 先判断是否已经在嵌套滑动中，是的话，不处理
            return true;
        }
        if (isNestedScrollingEnabled()) {// 判断是否支持嵌套滑动
            ViewParent p = mView.getParent();
            View child = mView;
// 这里就是利用循环一层一层的网上取出ParentView，直到该ParentView是支持嵌套滑动或者为null的
时候
            while (p != null) {
//就是判断当前ViewParent是否支持嵌套滑动。同时如果ViewParent是NestedScrollingParent的子
类的话，会调用onStartNestedScroll()判断当前ViewParent是否需要嵌套滑动
                if (ViewParentCompat.onStartNestedScroll(p, child, mView, axes,
type)) {//1
// 给mNestedScrollingParentTouch赋值，后面就可以直接获取有效ViewParent
                    setNestedScrollingParentForType(type, p);
                    //注释1 返回true，注释@里面就会调用NestedScrollingParent的
onNestedScrollAccepted()方法,也就是说如果要处理嵌套滑动，onStartNestedScroll必须返回
true
                    ViewParentCompat.onNestedScrollAccepted(p, child, mView,
axes, type);
                    return true;
                }
                if (p instanceof View) {
                    child = (View) p;
                }
                p = p.getParent();
```

```
            }
        }
        return false;
    }

    public void stopNestedScroll() {
        stopNestedScroll(TYPE_TOUCH);
    }

    public void stopNestedScroll(@NestedScrollType int type) {
        ViewParent parent = getNestedScrollingParentForType(type);
        if (parent != null) {
        // 这里面就会调用ViewParent的onStopNestedScroll(target, type)方法
            ViewParentCompat.onStopNestedScroll(parent, mView, type);
            // 该次滑动结束 将给mNestedScrollingParentTouch置空
            setNestedScrollingParentForType(type, null);
        }
    }

    public boolean dispatchNestedScroll(int dxConsumed, int dyConsumed,
            int dxUnconsumed, int dyUnconsumed, @Nullable int[] offsetInWindow)
{
        return dispatchNestedScrollInternal(dxConsumed, dyConsumed,
dxUnconsumed, dyUnconsumed,
                offsetInWindow, TYPE_TOUCH, null);
    }


    public boolean dispatchNestedScroll(int dxConsumed, int dyConsumed, int
dxUnconsumed,
            int dyUnconsumed, @Nullable int[] offsetInWindow, @NestedScrollType
int type) {
        return dispatchNestedScrollInternal(dxConsumed, dyConsumed,
dxUnconsumed, dyUnconsumed,
                offsetInWindow, type, null);
    }


    public void dispatchNestedScroll(int dxConsumed, int dyConsumed, int
dxUnconsumed,
            int dyUnconsumed, @Nullable int[] offsetInWindow, @NestedScrollType
int type,
            @Nullable int[] consumed) {
        dispatchNestedScrollInternal(dxConsumed, dyConsumed, dxUnconsumed,
dyUnconsumed,
                offsetInWindow, type, consumed);
    }

    private boolean dispatchNestedScrollInternal(int dxConsumed, int dyConsumed,
            int dxUnconsumed, int dyUnconsumed, @Nullable int[] offsetInWindow,
            @NestedScrollType int type, @Nullable int[] consumed) {
        if (isNestedScrollingEnabled()) {
        // 在startNestedScroll()中进行了赋值操作，所以这里可以直接获取ViewParent了
            final ViewParent parent = getNestedScrollingParentForType(type);
            if (parent == null) {
                return false;
            }
 // 判断是否是有效的嵌套滑动
```

```
            if (dxConsumed != 0 || dyConsumed != 0 || dxUnconsumed != 0 ||
dyUnconsumed != 0) {
                int startX = 0;
                int startY = 0;
                if (offsetInWindow != null) {
                    mView.getLocationInWindow(offsetInWindow);
                    startX = offsetInWindow[0];
                    startY = offsetInWindow[1];
                }

                if (consumed == null) {
                    consumed = getTempNestedScrollConsumed();
                    consumed[0] = 0;
                    consumed[1] = 0;
                }
 // 这里就会调用ViewParent的onNestedScroll()方法
                ViewParentCompat.onNestedScroll(parent, mView,
                        dxConsumed, dyConsumed, dxUnconsumed, dyUnconsumed,
type, consumed);

                if (offsetInWindow != null) {
                    mView.getLocationInWindow(offsetInWindow);
                    offsetInWindow[0] -= startX;
                    offsetInWindow[1] -= startY;
                }
                return true;
            } else if (offsetInWindow != null) {
                offsetInWindow[0] = 0;
                offsetInWindow[1] = 0;
            }
        }
        return false;
    }


    public boolean dispatchNestedPreScroll(int dx, int dy, @Nullable int[]
consumed,
            @Nullable int[] offsetInWindow) {
        return dispatchNestedPreScroll(dx, dy, consumed, offsetInWindow,
TYPE_TOUCH);
    }

    public boolean dispatchNestedPreScroll(int dx, int dy, @Nullable int[]
consumed,
            @Nullable int[] offsetInWindow, @NestedScrollType int type) {
        if (isNestedScrollingEnabled()) {
            final ViewParent parent = getNestedScrollingParentForType(type);
            if (parent == null) {
                return false;
            }

            if (dx != 0 || dy != 0) {
                int startX = 0;
                int startY = 0;
                if (offsetInWindow != null) {
                    mView.getLocationInWindow(offsetInWindow);
                    startX = offsetInWindow[0];
                    startY = offsetInWindow[1];
```

```
                }

                if (consumed == null) {
                    consumed = getTempNestedScrollConsumed();
                }
                consumed[0] = 0;
                consumed[1] = 0;
                // 这里会调用ViewParent的onNestedPreScroll()方法 Parent消费的数据会缝
在consumed变量中
                ViewParentCompat.onNestedPreScroll(parent, mView, dx, dy,
consumed, type);

                if (offsetInWindow != null) {
                    mView.getLocationInWindow(offsetInWindow);
                    offsetInWindow[0] -= startX;
                    offsetInWindow[1] -= startY;
                }
                return consumed[0] != 0 || consumed[1] != 0;
            } else if (offsetInWindow != null) {
                offsetInWindow[0] = 0;
                offsetInWindow[1] = 0;
            }
        }
        return false;
    }

    public boolean dispatchNestedFling(float velocityX, float velocityY, boolean
consumed) {
        if (isNestedScrollingEnabled()) {
            ViewParent parent = getNestedScrollingParentForType(TYPE_TOUCH);
            if (parent != null) {
            // 这里会调用ViewParent的onNestedFling()方法
                return ViewParentCompat.onNestedFling(parent, mView, velocityX,
                        velocityY, consumed);
            }
        }
        return false;
    }

    public boolean dispatchNestedPreFling(float velocityX, float velocityY) {
        if (isNestedScrollingEnabled()) {
            ViewParent parent = getNestedScrollingParentForType(TYPE_TOUCH);
            if (parent != null) {
                return ViewParentCompat.onNestedPreFling(parent, mView,
velocityX,
                        velocityY);
            }
        }
        return false;
    }

    public void onDetachedFromWindow() {
        ViewCompat.stopNestedScroll(mView);
    }

    public void onStopNestedScroll(@NonNull View child) {
        ViewCompat.stopNestedScroll(mView);
    }
```

```
    private ViewParent getNestedScrollingParentForType(@NestedScrollType int
type) {
        switch (type) {
            case TYPE_TOUCH:
                return mNestedScrollingParentTouch;
            case TYPE_NON_TOUCH:
                return mNestedScrollingParentNonTouch;
        }
        return null;
    }

    private void setNestedScrollingParentForType(@NestedScrollType int type,
ViewParent p) {
        switch (type) {
            case TYPE_TOUCH:
                mNestedScrollingParentTouch = p;
                break;
            case TYPE_NON_TOUCH:
                mNestedScrollingParentNonTouch = p;
                break;
        }
    }

    private int[] getTempNestedScrollConsumed() {
        if (mTempNestedScrollConsumed == null) {
            mTempNestedScrollConsumed = new int[2];
        }
        return mTempNestedScrollConsumed;
    }
}
```

NestedScrollView

NestedScrollView和ScrollView类似，是一个支持滚动的控件。此外，它还同时支持作NestedScrollingParent或者NestedScrollingChild进行嵌套滚动操作。默认是启用嵌套滚动的

```
public class NestedScrollView extends FrameLayout implements
NestedScrollingParent3,
        NestedScrollingChild3, ScrollingView {}
```