

Unisoc Confidential For hiar

## Android 10.0 通知及导航栏介绍

文档版本  
发布日期

V1.0  
2020-10-26

**版权所有 © 紫光展锐（上海）科技有限公司。保留一切权利。**

本文件所含数据和信息都属于紫光展锐（上海）科技有限公司（以下简称紫光展锐）所有的机密信息，紫光展锐保留所有相关权利。本文件仅为信息参考之目的提供，不包含任何明示或默示的知识产权许可，也不表示有任何明示或默示的保证，包括但不限于满足任何特殊目的、不侵权或性能。当您接受这份文件时，即表示您同意本文件中内容和信息属于紫光展锐机密信息，且同意在未获得紫光展锐书面同意前，不使用或复制本文件的整体或部分，也不向任何其他方披露本文件内容。紫光展锐有权在未经事先通知的情况下，在任何时候对本文件做任何修改。紫光展锐对本文件所含数据和信息不做任何保证，在任何情况下，紫光展锐均不负责任任何与本文件相关的直接或间接的、任何伤害或损失。

请参照交付物中说明文档对紫光展锐交付物进行使用，任何人对紫光展锐交付物的修改、定制化或违反说明文档的指引对紫光展锐交付物进行使用造成的任何损失由其自行承担。紫光展锐交付物中的性能指标、测试结果和参数等，均为在紫光展锐内部研发和测试系统中获得的，仅供参考，若任何人需要对交付物进行商用或量产，需要结合自身的软硬件测试环境进行全面的测试和调试。

Unisoc Confidential For hiar

**紫光展锐（上海）科技有限公司**



# 前言

## 概述

本文档基于 Android 10.0 介绍系统界面（SystemUI）中的通知（Notification）和导航栏（Navigation Bar），并对常见问题进行分析说明。本文档不涉及系统界面其他部分的介绍。

## 读者对象


本文档主要适用于展锐平台的 SystemUI 应用开发人员，帮助其了解 Android 10.0 系统界面中通知和导航栏的主要流程，以便其开发和维护系统界面模块。

## 缩略语

| 缩略语 | 英文全名                         | 中文解释    |
|-----|------------------------------|---------|
| NMS | Notification Manager Service | 通知管理服务  |
| WMS | Window Manager Service       | 窗口管理服务  |
| RRO | Runtime Resources Overlay    | 动态资源覆盖  |
| FW  | Frameworks                   | 应用程序框架层 |
| App | Application                  | 应用程序    |

## 符号约定

在本文中可能出现下列标志，它所代表的含义如下。

| 符号   | 说明  |
|--|---|
|  说明 | 用于突出重要/关键信息、补充信息和小窍门等。<br>“说明”不是安全警示信息，不涉及人身、设备及环境伤害。 |

## 变更信息

| 文档版本 | 发布日期       | 修改说明     |
|------|------------|----------|
| V1.0 | 2020-10-26 | 第一次正式发布。 |

## 关键字

Notification/通知、NavigationBar/导航栏、SystemUI/系统界面。

Unisoc Confidential For hiar

# 目 录

|                        |    |
|------------------------|----|
| 1 通知.....              | 1  |
| 1.1 流程说明 .....         | 1  |
| 1.1.1 发送流程 .....       | 1  |
| 1.1.2 显示流程 .....       | 2  |
| 1.2 常见问题分析 .....       | 11 |
| 1.2.1 通知 log 及含义 ..... | 11 |
| 1.2.2 问题分析举例 .....     | 12 |
| 2 导航栏.....             | 14 |
| 2.1 流程说明 .....         | 14 |
| 2.1.1 加载流程 .....       | 14 |
| 2.1.2 工作原理 .....       | 18 |
| 2.2 导航栏设置 .....        | 21 |
| 2.2.1 隐藏导航栏 .....      | 21 |
| 2.2.2 设置导航模式 .....     | 21 |
| 2.3 常见问题分析 .....       | 23 |

Unisoc Confidential For hiar

# 1 通知

通知（Notification）是指 Android 在应用的界面之外显示的消息，旨在向用户提供提醒、来自他人的通信信息或应用中的其他实时信息。点击通知可以打开相应的应用，也可以直接在通知中执行某项操作。

通知是一类看不见的程序组件（如 Broadcast Receiver、Service 或不活跃的 Activity），它是用来提示用户有需要注意的事件发生时的最好途径。创建通知有两个步骤：

1. 获取 NotificationManager 实例。
2. 创建 Notification 实例，设置属性，并发送通知正文。

## 1.1 流程说明

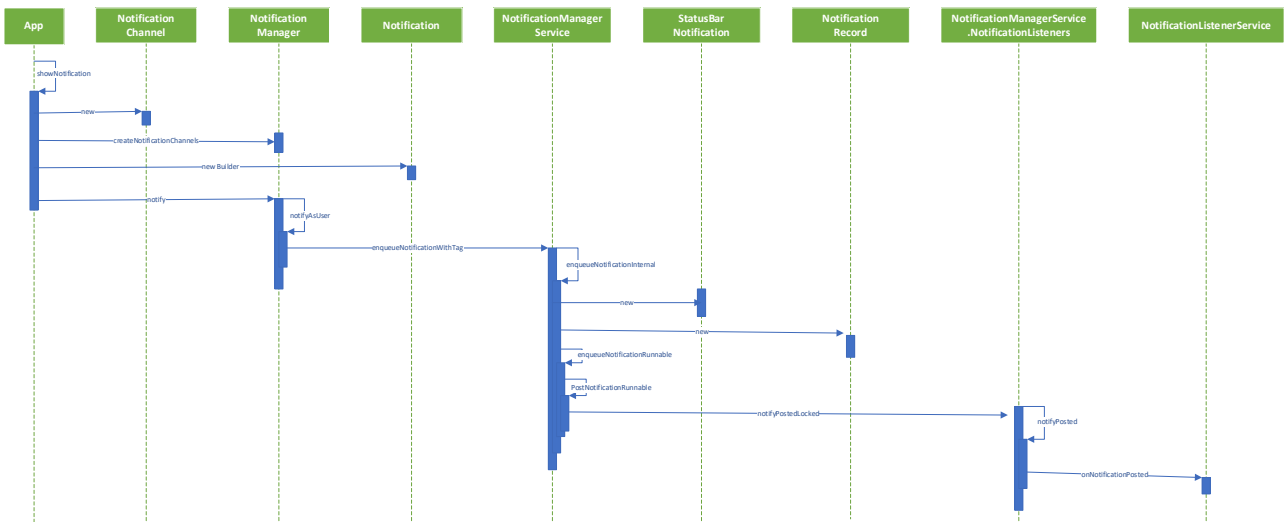
### 1.1.1 发送流程

通知发送者（比如 App）构建一个 Notification 对象，调用 NotificationManager.notify()方法发送通知，其主要代码如下：

```
// 1. 创建一个通知(设置channelId)
String channelId = "ChannelId";
Notification notification = new Notification.Builder(mContext)
    .setChannelId(channelId)
    .setSmallIcon(R.mipmap.icon_xx)
    .setContentTitle("通知标题")
    .setContentText("通知内容")
    .build();
// 2. 获取系统的通知管理器(设置channelId)
NotificationManager notificationManager = (NotificationManager) mContext
    .getSystemService(NOTIFICATION_SERVICE);
NotificationChannel channel = new NotificationChannel(
    channelId,
    "Channel 名称",
    NotificationManager.IMPORTANCE_DEFAULT);
notificationManager.createNotificationChannel(channel);
// 3. 发送通知(Notification与NotificationManager的channelId必须对应)
notificationManager.notify(id, notification);
```

在 Frameworks 中（主要是 Notification Manager Service）的流程如图 1-1 所示。

图1-1 通知发送流程



## 1.1.2 显示流程

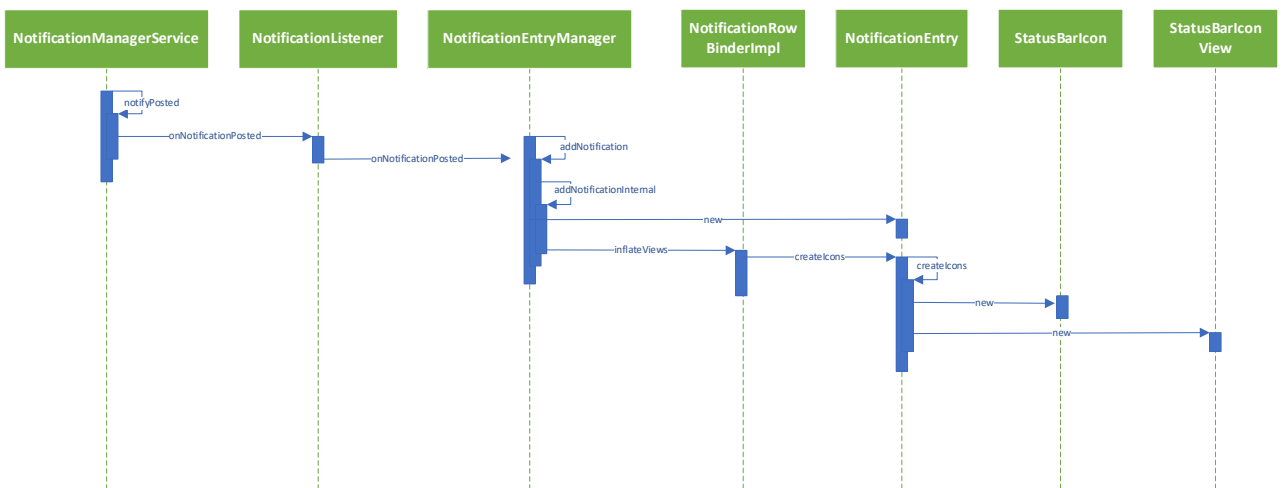
通知发出后，由系统界面（SystemUI）显示通知内容。

根据“图 1-1 通知发送流程”，通知发出后 NMS（Notification Manager Service）通过 NotificationListeners notifyPosted 回调 NotificationListenerService 的 onNotificationPosted 方法，而 SystemUI 正是通过继承 NotificationListenerService 的实现做到接收通知的。

### 1.1.2.1 通知图标

SystemUI 接收通知后，需要创建和显示状态栏（StatusBar）的图标，其主要流程如图 1-2 所示。

图1-2 通知图标加载流程



1. NotificationListener 是 FW NotificationListenerService 的一个子类，扩展实现了 onNotificationPosted、onNotificationRemoved 等方法，所以当 NMS 发出通知发布的信息后，SystemUI 可以接收到对应的 notification。NotificationListener 收到 notification 后通知 NotificationEntryManager。



## 说明

NMS 传递给 Listener 的其实并不是 Notification 对象，而是 StatusBarNotification，它是 Notification 的封装类。

2. NotificationListener 中会调用 onNotificationPosted 方法，当收到通知（StatusBarNotification sbn）后，根据 sbn.getKey() 获得相应的 key 值，状态栏会通过通知的唯一 key 值来判断该通知是更新还是新增的。
  - 如果是新增通知（addNotification），通过 key 获取的 NotificationEntry 是 null，则首先要去构造一个 NotificationEntry，它可被视为 SystemUI 上一个具体的通知，包括状态栏的图标、HeadsUp 通知（悬浮通知）以及通知详情等。
  - 如果是更新通知（updateNotification），通过 key 直接获得对应的 NotificationEntry 实例。

下面流程都以新增为例。关键代码如下：

```
public void onNotificationPosted(final StatusBarNotification sbn,
    final RankingMap rankingMap) {
    if (DEBUG) Log.d(TAG, "onNotificationPosted: " + sbn);
    if (sbn != null && !onPluginNotificationPosted(sbn, rankingMap)) {
        Dependency.get(Dependency.MAIN_HANDLER).post(() -> {
            processForRemoteInput(sbn.getNotification(), mContext);
            String key = sbn.getKey();
            boolean isUpdate =
                mEntryManager.getNotificationData().get(key) != null;
            // In case we don't allow child notifications, we ignore children of
            // notifications that have a summary, since we're not going to show them
            // anyway. This is true also when the summary is canceled,
            // because children are automatically canceled by NoMan in that case.
            if (!ENABLE_CHILD_NOTIFICATIONS
                && mGroupManager.isChildInGroupWithSummary(sbn)) {
                if (DEBUG) {
                    Log.d(TAG, "Ignoring group child due to existing summary: " + sbn);
                }
            }

            // Remove existing notification to avoid stale data.
            if (isUpdate) {
                mEntryManager.removeNotification(key, rankingMap, UNDEFINED_DISMISS_REASON);
            } else {
                mEntryManager.getNotificationData()
                    .updateRanking(rankingMap);
            }
            return;
        });
        if (isUpdate) {
            mEntryManager.updateNotification(sbn, rankingMap);
        } else {
            mEntryManager.addNotification(sbn, rankingMap);
        }
    }
}
```

3. NotificationEntry 对象构造之后，开始构造通知显示的各个 view，通知 Icon 也是其中之一。NotificationRowBinder 是定义通知添加或更新时 inflate views 的接口，NotificationRowBinderImpl 是其具体实现。通过上面的代码看到，构造通知 views 是通过 NotificationRowBinderImpl.inflateViews 实现的，关键代码如下：



```
public void inflateViews(
    ..... NotificationEntry entry,
    ..... Runnable onDismissRunnable)
    ..... throws InflationException {
    ..... ViewGroup parent = mListContainer.getViewParentForNotification(entry);
    ..... PackageManager pmUser = StatusBar.getPackageManagerForUser(mContext,
    ..... entry.notification.getUser().getIdentifier());

    ..... final StatusBarNotification sbn = entry.notification;
    ..... if (entry.rowExists()) {
    ..... entry.updateIcons(mContext, sbn);
    ..... entry.reset();
    ..... updateNotification(entry, pmUser, sbn, entry.getRow());
    ..... } else {
    ..... entry.createIcons(mContext, sbn);
    ..... new RowInflaterTask().inflate(mContext, parent, entry,
    ..... row -> {
    ..... ..... bindRow(entry, pmUser, sbn, row, onDismissRunnable);
    ..... ..... updateNotification(entry, pmUser, sbn, row);
    ..... ..... });
    ..... }
    ..... }
}
```

- a NotificationRowBinderImpl.inflateViews 首先判断 entry row 是否存在。如果存在则更新通知 view。如果不存在，则要构造一个 row。

## 说明

ExpandableNotificationRow, 表示通知项的视图, 可以是单个子通知, 也可以是一组通知摘要 (包含 1 个或多个子通知)。

- b 通过 entry.createIcons 来构造通知图标, 或通过 updateIcons 更新图标。下面只介绍通知图标的构造, 关键代码如下:

Unisoc Confidential For hiar

```
public void createIcons(Context context, StatusBarNotification sbn)
    throws InflationException {
    Notification n = sbn.getNotification();
    final Icon smallIcon = n.getSmallIcon();
    if (smallIcon == null) {
        throw new InflationException("No small icon in notification from "
            + sbn.getPackageName());
    }

    // Construct the icon.
    icon = new StatusBarIconView(context,
        sbn.getPackageName() + "/" + Integer.toHexString(sbn.getId()), sbn);
    icon.setScaleType(ImageView.ScaleType.CENTER_INSIDE);

    // Construct the expanded icon.
    expandedIcon = new StatusBarIconView(context,
        sbn.getPackageName() + "/" + Integer.toHexString(sbn.getId()), sbn);
    expandedIcon.setScaleType(ImageView.ScaleType.CENTER_INSIDE);

    final StatusBarIcon ic = new StatusBarIcon(context,
        icon, expandedIcon);
    if (!icon.set(ic) || !expandedIcon.set(ic)) {
        icon = null;
        expandedIcon = null;
        centeredIcon = null;
        throw new InflationException("Couldn't create icon: " + ic);
    }
    expandedIcon.setVisibility(View.INVISIBLE);
    expandedIcon.setOnVisibilityChangeListener(
        new VisibilityListener() {
            @Override public void onVisibilityChanged() {
                // ...
            }
        });

    // Construct the centered icon
    if (notification.getNotification().isMediaNotification()) {
        // ...
    }
}
```

entry.createIcons 不仅构造状态栏上的通知 Icon，也会同时构造通知在 Shelf（状态栏下滑后的通知栏）以及通知详情中的图标。上面代码中 Icon 即状态栏上的通知图标，它是通过 new StatusBarIconView() 来构造 StatusBarIconView（显示图标的 view）。

### 1.1.2.2 通知布局

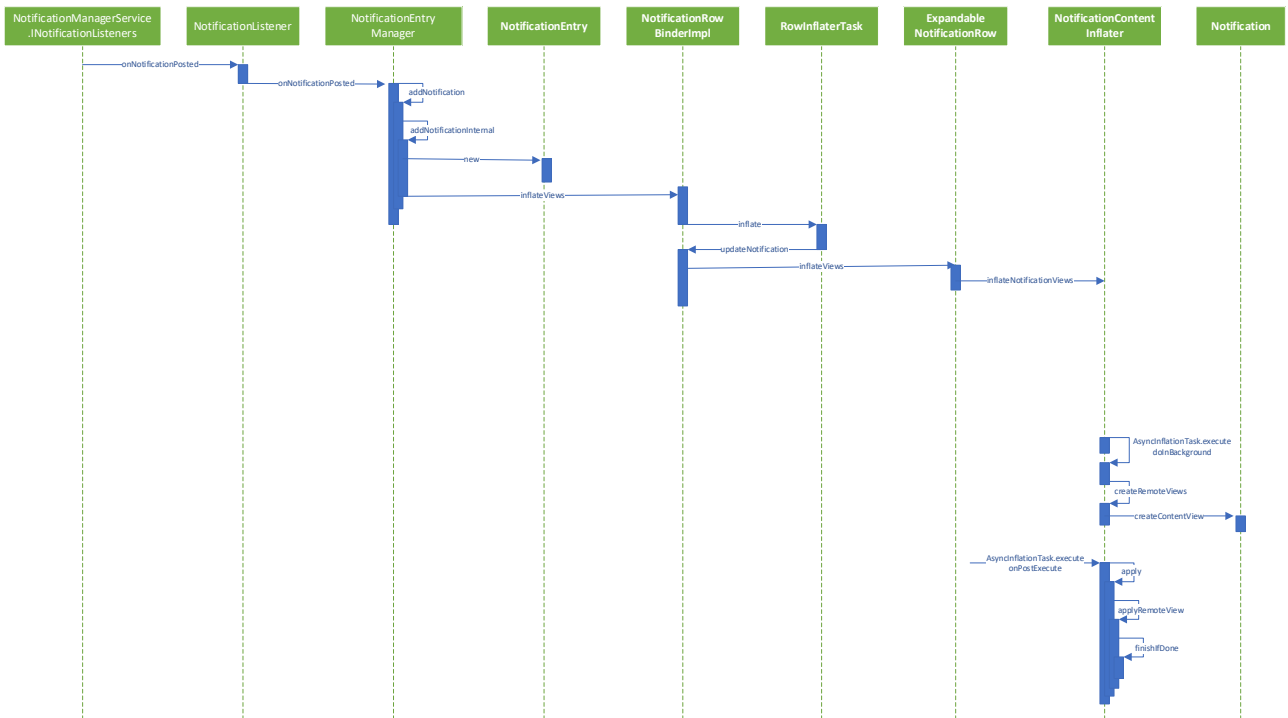
SystemUI 收到通知之后，需要解析通知显示的布局。如果是 HeadsUp 通知会弹出悬浮提示，用户还可以通过下拉状态栏查看通知详情信息。

#### 说明

发送者可以通过 Notification.Builder builder.setCustomContentView(RemoteViews) 的方式自定义通知布局。如果没有自定义布局，Notification 中通过默认的 R.layout.notification\_template\_material\_base 构造 RemoteViews。SystemUI 需要获取该 RemoteViews 以显示通知内容。

通知显示布局整体流程如图 1-3 所示。

图1-3 通知显示布局加载流程



1. SystemUI 监听 NMS post 通知等和前文通知图标构建的流程是同一过程。只是在 NotificationRowBinderImpl.inflateViews 时，构建好通知图标之后就开始 inflate 通知内容的布局了。在 inflateView 中需要通过 RowInflaterTask 来 inflate Row 的布局，RowInflaterTask 是异步 inflate ExpandableNotificationRow 布局的类。关键代码如下：

```
public void inflate(Context context, ViewGroup parent, NotificationEntry entry,
    ... RowInflationFinishedListener listener) {
    ... if (TRACE_ORIGIN) {
    ... mInflateOrigin = new Throwable("inflate requested here");
    ... }
    ... mListener = listener;
    ... AsyncLayoutInflater inflater = new AsyncLayoutInflater(context);
    ... mEntry = entry;
    ... entry.setInflationTask(this);
    ... inflater.inflate(R.layout.status_bar_notification_row, parent, this);
}
```

2. Row 的布局是 R.layout.status\_bar\_notification\_row。Inflate row 布局后，在 onInflationFinished 的回调中构造 ExpandableNotificationRow 对象，此时会调用 updateNotification 去构建显示通知内容的 views：

```
private void updateNotification(
    NotificationEntry entry,
    PackageManager pmUser,
    StatusBarNotification sbn,
    ExpandableNotificationRow row) {
    row.setIsLowPriority(entry.ambient);

    // Extract target SDK version.
    try {
    } catch (PackageManager.NameNotFoundException ex) {
        Log.e(TAG, "Failed looking up ApplicationInfo for " + sbn.getPackageName(), ex);
    }
    row.setLegacy(entry.targetSdk >= Build.VERSION_CODES.GINGERBREAD
        && entry.targetSdk < Build.VERSION_CODES.LOLLIPOP);

    // TODO: should updates to the entry be happening somewhere else?
    entry.setIconTag(R.id.icon_is_pre_L, entry.targetSdk < Build.VERSION_CODES.LOLLIPOP);
    entry.autoRedacted = entry.notification.getNotification().publicVersion == null;

    entry.setRow(row);
    row.setOnActivatedListener(mPresenter);

    boolean useIncreasedCollapsedHeight =
        mMessagingUtil.isImportantMessaging(sbn, entry.importance);
    boolean useIncreasedHeadsUp = useIncreasedCollapsedHeight
        && !mPresenter.isPresenterFullyCollapsed();
    row.setUseIncreasedCollapsedHeight(useIncreasedCollapsedHeight);
    row.setUseIncreasedHeadsUpHeight(useIncreasedHeadsUp);
    row.setEntry(entry);

    if (mNotificationInterruptionStateProvider.shouldHeadsUp(entry)) {
    }
    if (mNotificationInterruptionStateProvider.shouldPulse(entry)) {
    }
    row.setNeedsRedaction(
        Dependency.get(NotificationLockscreenUserManager.class).needsRedaction(entry));
    row.inflateViews();

    // bind the click event to the content area
    checkNotNull(mNotificationClicker).register(row, sbn);
}
```

ExpandableNotificationRow 类会调用 inflateViews 去构建显示通知内容的 views:

```
public void inflateViews() {
    mNotificationInflater.inflateNotificationViews();
}
```

mNotificationInflater 是 NotificationContentInflater 类的实例。

- 接着调用 NotificationContentInflater.inflateNotificationViews 去 inflate 通知内容控件。inflateNotificationViews 中主要是构造 AsyncInflationTask 并执行，关注 AsyncInflationTask 的 doInBackground:

```
protected InflationProgress doInBackground(Void... params) {
    try {
        final Notification.Builder recoveredBuilder
            = Notification.Builder.recoverBuilder(mContext,
            mSbn.getNotification());

        Context packageContext = mSbn.getPackageContext(mContext);
        Notification notification = mSbn.getNotification();
        if (notification.isMediaNotification()) {
            MediaNotificationProcessor processor = new MediaNotificationProcessor(mContext,
            packageContext);
            processor.processNotification(notification, recoveredBuilder);
        }

        InflationProgress inflationProgress = createRemoteViews(mReInflateFlags,
            recoveredBuilder, mIsLowPriority,
            mIsChildInGroup, mUsesIncreasedHeight, mUsesIncreasedHeadsUpHeight,
            mRedactAmbient, packageContext);
        return inflateSmartReplyViews(inflationProgress, mReInflateFlags, mRow.getEntry(),
            mRow.getContext(), mRow.getHeadsUpManager(),
            mRow.getExistingSmartRepliesAndActions());
    } catch (Exception e) {
        mError = e;
        return null;
    }
}
```

这里最重要的是 createRemoteViews:

```
private static InflationProgress createRemoteViews(@InflationFlag int reInflateFlags,
    Notification.Builder builder, boolean isLowPriority, boolean isChildInGroup,
    boolean usesIncreasedHeight, boolean usesIncreasedHeadsUpHeight, boolean redactAmbient,
    Context packageContext) {
    InflationProgress result = new InflationProgress();
    isLowPriority = isLowPriority && !isChildInGroup;

    if ((reInflateFlags & FLAG_CONTENT_VIEW_CONTRACTED) != 0) {
        result.newContentView = createContentView(builder, isLowPriority, usesIncreasedHeight);
    }

    if ((reInflateFlags & FLAG_CONTENT_VIEW_EXPANDED) != 0) {
        result.newExpandedView = createExpandedView(builder, isLowPriority);
    }

    if ((reInflateFlags & FLAG_CONTENT_VIEW_HEADS_UP) != 0) {
        result.newHeadsUpView = builder.createHeadsUpContentView(usesIncreasedHeadsUpHeight);
    }

    if ((reInflateFlags & FLAG_CONTENT_VIEW_PUBLIC) != 0) {
        result.newPublicView = builder.makePublicContentView();
    }

    if ((reInflateFlags & FLAG_CONTENT_VIEW_AMBIENT) != 0) {
        result.newAmbientView = redactAmbient ? builder.makePublicAmbientNotification()
            : builder.makeAmbientNotification();
    }

    result.packageContext = packageContext;
    result.headsUpStatusBarText = builder.getHeadsUpStatusBarText(false /* showingPublic */);
    result.headsUpStatusBarTextPublic = builder.getHeadsUpStatusBarText(
        true /* showingPublic */);
    return result;
}
```

可以看到，在 createRemoteViews 中分别去创建通知 content、expanded、heads-up 的 RemoteView，以 expanded 为例：

```
private static RemoteViews createExpandedView(Notification.Builder builder,
... boolean isLowPriority) {
... RemoteViews bigContentView = builder.createBigContentView();
... if (bigContentView != null) {
...     return bigContentView;
... }
... if (isLowPriority) {
...     RemoteViews contentView = builder.createContentView();
...     Notification.Builder.makeHeaderExpanded(contentView);
...     return contentView;
... }
... return null;
}

private static RemoteViews createContentView(Notification.Builder builder,
... boolean isLowPriority, boolean useLarge) {
... if (isLowPriority) {
...     return builder.makeLowPriorityContentView(false /*useRegularSubtext*/);
... }
... return builder.createContentView(useLarge);
}
```

createExpandedView 中调用 Notification.Builder.createContentView 获取 RemoteViews，而这个返回的 RemoteViews 正是 Notification 默认布局，或者发送者设置的（比如 setCustomContentView）布局。

4. AsyncTask 的 doInBackground 执行完成后，会回到主线程执行 onPostExecute，执行 apply，关键代码如下：

Unisoc Confidential For hiar

```
public static CancellationSignal apply(
    boolean inflateSynchronously,
    InflationProgress result,
    @InflationFlag int reInflateFlags,
    ArrayMap<Integer, RemoteViews> cachedContentViews,
    ExpandableNotificationRow row,
    boolean redactAmbient,
    RemoteViews.OnClickHandler remoteViewClickHandler,
    @Nullable InflationCallback callback) {
    NotificationContentView privateLayout = row.getPrivateLayout();
    NotificationContentView publicLayout = row.getPublicLayout();
    final HashMap<Integer, CancellationSignal> runningInflations = new HashMap<>();

    int flag = FLAG_CONTENT_VIEW_CONTRACTED;
    if ((reInflateFlags & flag) != 0) { ...
    }

    flag = FLAG_CONTENT_VIEW_EXPANDED;
    if ((reInflateFlags & flag) != 0) { ...
    }

    flag = FLAG_CONTENT_VIEW_HEADS_UP;
    if ((reInflateFlags & flag) != 0) { ...
    }

    flag = FLAG_CONTENT_VIEW_PUBLIC;
    if ((reInflateFlags & flag) != 0) { ...
    }

    flag = FLAG_CONTENT_VIEW_AMBIENT;
    if ((reInflateFlags & flag) != 0) { ...
    }

    // Let's try to finish, maybe nobody is even inflating anything
    finishIfDone(result, reInflateFlags, cachedContentViews, runningInflations, callback, row,
        redactAmbient);
    CancellationSignal cancellationSignal = new CancellationSignal();
    cancellationSignal.setOnCancelListener(
        () -> runningInflations.values().forEach(CancellationSignal::cancel));
    return cancellationSignal;
}
```

上面代码以 Content 的内容为例：

- 先构造 ApplyCallback，用以 apply 完成之后回调获得 view。
- applyRemoteView 则主要是执行 RemoteViews apply。

5. 显示通知内容的 view 构建好之后，需将该 view 设置给 ExpandableNotificationRow：



```
private static boolean finishIfDone(InflationProgress result,
    @InflationFlag int reInflateFlags, ArrayMap<Integer, RemoteViews> cachedContentViews,
    HashMap<Integer, CancellationSignal> runningInflations,
    @Nullable InflationCallback endListener, ExpandableNotificationRow row,
    boolean redactAmbient) {
    Assert.isMainThread();
    NotificationEntry entry = row.getEntry();
    NotificationContentView privateLayout = row.getPrivateLayout();
    NotificationContentView publicLayout = row.getPublicLayout();
    if (runningInflations.isEmpty()) {
        if ((reInflateFlags & FLAG_CONTENT_VIEW_CONTRACTED) != 0) {
        }

        if ((reInflateFlags & FLAG_CONTENT_VIEW_EXPANDED) != 0) {
        }

        if ((reInflateFlags & FLAG_CONTENT_VIEW_HEADS_UP) != 0) {
        }

        if ((reInflateFlags & FLAG_CONTENT_VIEW_PUBLIC) != 0) {
        }

        if ((reInflateFlags & FLAG_CONTENT_VIEW_AMBIENT) != 0) {
        }
        entry.headsUpStatusBarText = result.headsUpStatusBarText;
        entry.headsUpStatusBarTextPublic = result.headsUpStatusBarTextPublic;
        if (endListener != null) {
            endListener.onAsyncInflationFinished(row.getEntry(), reInflateFlags);
        }
        return true;
    }
    return false;
}
```

## 1.2 常见问题分析

### 1.2.1 通知 log 及含义

当应用或者系统发送一条通知时，一般会有以下 log 打印，了解这些 log 的含义有助于分析问题。下面以发送一条短信通知为例对几种主要的通知 log 进行说明。

- notification\_enqueue:  
[10086,4335,com.google.android.apps.messaging,0,com.google.android.apps.messaging:sms:1,0,Notification(channel=bugle\_default\_channel pri=2 contentView=null vibrate=null...)]  
表示通知进入消息队列。
- notification\_canceled:  
[0|com.google.android.apps.messaging|0|com.google.android.apps.messaging:sms:1|10086,1,76920,76920,14016,0,3,NULL]  
表示通知取消。其中上面标红色的“1”所在位置代表通知取消的原因，“1”表示在状态栏点击该通知条取消通知。具体每个数字代表的原因，参见：  
/frameworks/base/core/java/android/service/notification/NotificationListenerService.java 以下代码部分：

```
// Notification cancellation reasons

/** Notification was canceled by the status bar reporting a notification click. */
public static final int REASON_CLICK = 1;
/** Notification was canceled by the status bar reporting a user dismissal. */
public static final int REASON_CANCEL = 2;
/** Notification was canceled by the status bar reporting a user dismiss all. */
public static final int REASON_CANCEL_ALL = 3;
/** Notification was canceled by the status bar reporting an inflation error. */
public static final int REASON_ERROR = 4;
/** Notification was canceled by the package manager modifying the package. */
public static final int REASON_PACKAGE_CHANGED = 5;
.....
/** Notification was snoozed. */
public static final int REASON_SNOOZED = 18;
/** Notification was canceled due to timeout */
public static final int REASON_TIMEOUT = 19;
```

- notification\_alert:  
[0|com.google.android.apps.messaging|0|com.google.android.apps.messaging:sms:1|10086,1,1,1]  
表示发出通知的形式：响铃、震动或者指示灯等。
- sysui\_fullscreen\_notification:  
0|com.android.dialer|1|null|10006  
表示通知最后以全屏方式显示，比如在锁屏时，电话或者闹钟是以全屏方式显示的。
- notification\_clicked:  
[0|com.google.android.apps.messaging|0|com.google.android.apps.messaging:sms:1|10086,76824,76824,13920,0,3]  
表示点击该通知条。
- notification\_action\_clicked:  
[0|com.google.android.apps.messaging|0|com.google.android.apps.messaging:sms:1|10086,0,25412,25412,1323,0,5]  
表示点击通知的 action 按钮，比如短信通知的“标记为已读”按钮等。

## 说明

Log 中每个参数代表的具体意义请参见/frameworks/base/services/core/java/com/android/server/EventLogTags.logtags

比如：notification\_enqueue(uid|1|5),(pid|1|5),(pkg|3),(id|1|5),(tag|3),(userid|1|5),(notification|3),(status|1)，根据上面的 log，我们就可以知道执行通知操作时的具体时间点以及相应操作是否正确执行。

## 1.2.2 问题分析举例

结合上面具体的通知流程和 log 含义可以进行简单的问题分析。

### 通知栏未弹出提示

**问题描述：**打开 WiFi，附近有可连接的开放 WLAN 网络，但通知栏未弹出提示。

#### Log 分析：

```
13:56:47.995 800 800 I notification_enqueue: [1000,800,android,17303299,NULL,0,Notification(channel=NETWORK_AVAILABLE ...
13:56:50.908 800 800 I notification_canceled: [0|android|17303299|null|1000,18,3000,3000,1717,-1,-1,NULL]
```

针对此类通知没有显示或者显示后很快消失的问题：

1. 查找 log 中的 notification\_enqueue 是否有该通知队列或者该通知是否被取消。

2. 通过上面的 log 可以看到，通知已经进入到消息队列，但是很快就被取消掉了，而取消的原因是 18，对应之前的 REASON\_SNOOZED，即通知被设置为了 snoozed 状态，所以不显示该通知。
3. 在非 go 版本，这个是 GMS 应用中的 Android 设置向导造成的，通过“设置->应用和通知->特殊应用权限->通知使用权限”关闭“Android 设置向导”选项，然后重启就可以看到 WiFi 的通知。所以不显示通知是由于第三方应用拥有通知控制权限，Android 设置向导关闭了该权限。

## 来电没有提示

**问题描述：**来电没有提示，但在通话记录中有未接来电。

**Log 分析：**

```
13:59:20.652 810 810 I notification_enqueue: [10006,2309,com.android.dialer,1,NULL,0,Notification(channel=phone_incoming_call
13:59:20.655 810 810 D ZenLog : intercepted: 0|com.android.dialer|1|null|10006,!audienceMatches (debug版本打开)
13:59:20.528 810 1328 I Telecom : Ringer: startRinging: skipping because ringer would not be audible.
isVolumeOverZero=false, shouldRingForContact=false, isRingtonePresent=true: (...->CS.crCo->H.CS.crCo->H.CS.crCo.pICR)
->CSW.hCCC->ABCF.oPE->ICF.oCFC->CAMSM.pM_2002@E-E-AJI
```

1. 在 log 中搜索 notification\_enqueue，可以很快找到对应时间点，说明通知消息已经进入队列，但没有看到有 notification\_canceled 的 log 打印，说明通知不是被取消掉的。
2. 在 log 中看到，在 notification\_enqueue 的 log 打印之后，紧接着就打印了 zenlog，提示该通知被拦截了（该 log 在 debug 版本是默认打开的），说明用户手动打开了勿扰模式。在 tele 的 log 中 shouldRingForContact=false 也可以说明是打开了勿扰模式。

## 说明

勿扰模式主要针对来电，短信，闹钟以及事件提醒等通知。如果打开勿扰模式，默认同一联系人在 15 分钟内第二次来电才会有提示。可通过状态栏的通知图标判断是否打开了勿扰模式。User 版本默认没有打开勿扰模式。

3. 如果以上情况均排除，在确认通知已经正确发送的前提下，可根据通知显示流程图具体分析，定位有问题的流程。

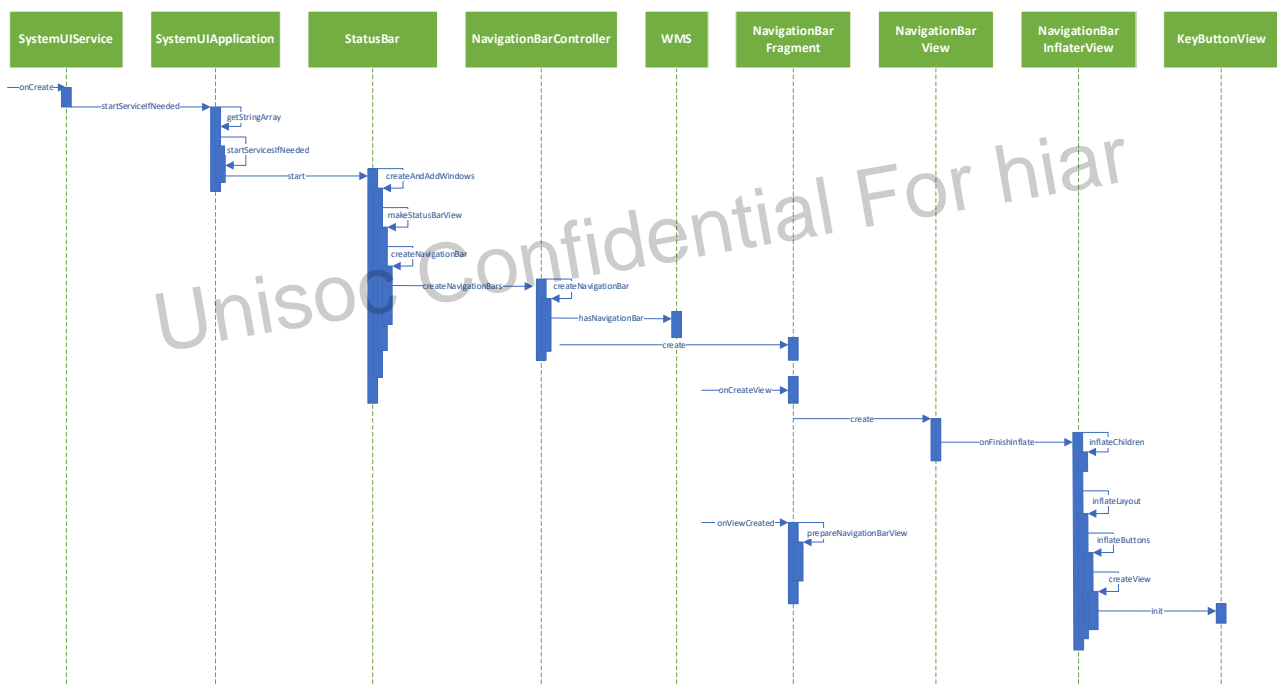
# 2 导航栏

导航栏（NavigationBar）属于系统界面（SystemUI）的一部分，是在屏幕底端或右端容纳了一排虚拟按键的一个窗口，可用来代替物理按键，最常使用的是 BACK、HOME 和 RECENT 键。可通过“设置->系统->手势->系统导航”中改变导航栏模式，有手势导航、三按钮导航可以选择。

## 2.1 流程说明

### 2.1.1 加载流程

图2-1 导航栏加载流程



#### 1. 启动 SystemUIService。

SystemService 负责启动 SystemUIService。

#### 说明

SystemUI 从某种意义上可以看作是 Android 系统的一个系统服务，也就是 SystemUIService。

#### 2. 初始化 SystemUI。

SystemUIApplication 是 Application 的子类，负责 SystemUI 的初始化。SystemUIApplication 首先通过 getResources().getStringArray 获取所有需要启动的 SystemUI service 组件。

## 说明

这些 service 代表各系统界面，比如 VolumeUI（与音量信息相关，主要有铃声音量，通知音量，媒体音量，闹钟音量）、PowerUI（与电池信息相关，比如电池图标及电量的显示、充电状态下电池图标的动态显示、低电量情况下的报警提示以及电池温度过高情况下的报警提示等）、长按 power 键的界面，Toast 的界面，以及 StatusBar。StatusBar 是状态栏、导航栏通知栏等启动的地方。这些 SystemUI service 都是抽象类 SystemUI 的子类，配置如下：

```
<!-- SystemUI Services: The classes of the stuff to start. -->
<string-array name="config_systemUIServiceComponents" translatable="false">
    <item>com.android.systemui.Dependency$DependencyCreator</item>
    <item>com.android.systemui.util.NotificationChannels</item>
    <item>com.android.systemui.statusbar.CommandQueue$CommandQueueStart</item>
    <item>com.android.systemui.keyguard.KeyguardViewMediator</item>
    <item>com.android.systemui.recents.Recents</item>
    <item>com.android.systemui.volume.VolumeUI</item>
    <item>com.android.systemui.stackdivider.Divider</item>
    <item>com.android.systemui.SystemBars</item>
    <item>com.android.systemui.usb.StorageNotification</item>
    <item>com.android.systemui.power.PowerUI</item>
    <item>com.android.systemui.media.RingtonePlayer</item>
    <item>com.android.systemui.keyboard.KeyboardUI</item>
    <item>com.android.systemui.pip.PipUI</item>
    <item>com.android.systemui.shortcut.ShortcutKeyDispatcher</item>
    <item>@string/config_systemUIVendorServiceComponent</item>
    <item>com.android.systemui.util.leak.GarbageMonitor$Service</item>
    <item>com.android.systemui.LatencyTester</item>
    <item>com.android.systemui.globalactions.GlobalActionsComponent</item>
    <item>com.android.systemui.ScreenDecorations</item>
    <item>com.android.systemui.biometrics.BiometricDialogImpl</item>
    <item>com.android.systemui.SliceBroadcastRelayHandler</item>
    <item>com.android.systemui.SizeCompatModeActivityController</item>
    <item>com.android.systemui.statusbar.notification.InstantAppNotifier</item>
    <item>com.android.systemui.theme.ThemeOverlayController</item>
    <item>com.android.systemui.DisplayServices</item>
</string-array>
```

3. StatusBar 通过 NavigationBarController 创建导航栏。NavigationBarController 是导航栏的控制类，它主要处理导航栏（主要是 NavigationBarFragment）的显示、隐藏等功能。

- a NavigationBarController 首先通过 WMS 接口内获取 hasNavigationBar 的值判断系统是否有导航栏。

## 说明

该接口最终是通过 frameworks\base\core\res\res\values\config.xml 的 config\_showNavigationBar 配置以及 system/build.prop 中 qemu.hw.mainkeys 值判断系统是否有导航栏。

- b 如果有导航栏，NavigationBarController 会去构造 NavigationBarFragment，它是显示导航栏的 Fragment，包含导航栏按键事件的处理。在 NavigationBarFragment 的 onCreateView 中 inflate 导航栏布局：

```
@Override
public View onCreateView(LayoutInflater inflater, @Nullable ViewGroup container,
    Bundle savedInstanceState) {
    return inflater.inflate(R.layout.navigation_bar, container, false);
}
```

R.layout.navigation\_bar 代码如下：

```
<com.android.systemui.statusbar.phone.NavigationBarView
... xmlns:android="http://schemas.android.com/apk/res/android"
... xmlns:systemui="http://schemas.android.com/apk/res-auto"
... android:layout_height="match_parent"
... android:layout_width="match_parent"
... android:background="@drawable/system_bar_background">

... <com.android.systemui.statusbar.phone.NavigationBarInflaterView
... |... android:id="@+id/navigation_inflater"
... |... android:layout_width="match_parent"
... |... android:layout_height="match_parent" />

</com.android.systemui.statusbar.phone.NavigationBarView>
```

可以看到导航栏的根布局是 NavigationBarView（继承自 FrameLayout），其中包含一个子控件 NavigationBarInflaterView（继承自 FrameLayout），在 NavigationBarInflaterView 被加载后（onFinishInflate 被调用），同时解析两套布局：R.layout.navigation\_layout（水平布局）和 R.layout.navigation\_layout\_vertical（垂直布局）：

```
@Override
protected void onFinishInflate() {
... super.onFinishInflate();
... inflateChildren();
... clearViews();
... inflateLayout(getDefaultLayout());
}

private void inflateChildren() {
... removeAllViews();
... mHorizontal = (FrameLayout) mLayoutInflater.inflate(R.layout.navigation_layout,
... |... this /*root*/, false /*attachToRoot*/);
... addView(mHorizontal);
... mVertical = (FrameLayout) mLayoutInflater.inflate(R.layout.navigation_layout_vertical,
... |... this /*root*/, false /*attachToRoot*/);
... addView(mVertical);
... updateAlternativeOrder();
}
```

c 在 navigation\_layout 被 inflate 后，需要去添加各按键的布局了，主要代码如下：



```
private View onCreateView(String buttonSpec, ViewGroup parent, LayoutInflater inflater) {
    View v = null;
    String button = extractButton(buttonSpec);
    if (LEFT.equals(button)) { ...
    } else if (RIGHT.equals(button)) { ...
    }
    if (HOME.equals(button)) {
        v = inflater.inflate(R.layout.home, parent, false);
    } else if (BACK.equals(button)) {
        v = inflater.inflate(R.layout.back, parent, false);
    } else if (RECENT.equals(button)) {
        v = inflater.inflate(R.layout.recent_apps, parent, false);
    } else if (MENU_IME_ROTATE.equals(button)) {
        v = inflater.inflate(R.layout.menu_ime, parent, false);
    } else if (NAVSPACE.equals(button)) {
        v = inflater.inflate(R.layout.nav_key_space, parent, false);
    } else if (CLIPBOARD.equals(button)) {
        v = inflater.inflate(R.layout.clipboard, parent, false);
    } else if (CONTEXTUAL.equals(button)) {
        v = inflater.inflate(R.layout.contextual, parent, false);
    } else if (HOME_HANDLE.equals(button)) {
        v = inflater.inflate(R.layout.home_handle, parent, false);
    } else if (IME_SWITCHER.equals(button)) {
        v = inflater.inflate(R.layout.ime_switcher, parent, false);
    } else if (button.startsWith(KEY)) { ...
    }
    /* UNISOC: Bug 1072090 new feature of dynamic navigationbar @ */
    else if (HIDE.equals(button)) {
        v = inflater.inflate(R.layout.hide, parent, false);
    } else if (PULL.equals(button)) {
        v = inflater.inflate(R.layout.pull, parent, false);
    } /* UNISOC: Add for bug 902309 1146896 @ */
    } else if (SPACE_PLACE.equals(button)) {
        v = inflater.inflate(R.layout.space, parent, false);
    } /* } @ */
    } else {
        return null;
    }
    /* @ */
    return v;
}
```

- d 在 onCreateView 方法中通过 inflate 设置按钮的布局，之后将这些按钮子控件添加到父控件 (navigation layout) 上。这里的 Hide 和 Pull 按钮是展锐添加用以隐藏导航栏和下拉通知栏的功能。以 RECENT 键为例，查看其布局 R.layout.recent\_apps:

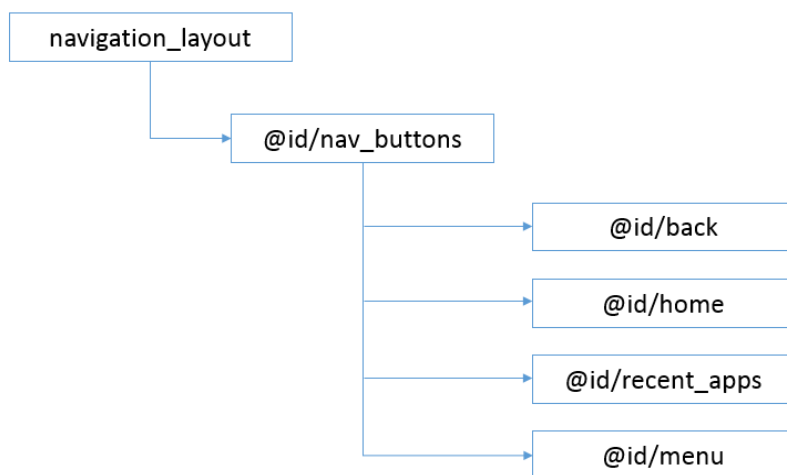
```
<com.android.systemui.statusbar.policy.KeyButtonView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:systemui="http://schemas.android.com/apk/res-auto"
    android:id="@+id/recent_apps"
    android:layout_width="@dimen/navigation_key_width"
    android:layout_height="match_parent"
    android:layout_weight="0"
    android:scaleType="center"
    android:contentDescription="@string/accessibility_recent"
    android:paddingStart="@dimen/navigation_key_padding"
    android:paddingEnd="@dimen/navigation_key_padding"
/>
```



仅有一个控件 `KeyButtonView`，它是 `ImageView` 的子类，主要工作是将触摸事件转换为按键事件。有关触摸事件处理的详细信息，参见“2.1.2 工作原理”。

至此 `NavigationBar` 的布局都加载完成了，`NavigationBarFragment` 的 `onViewCreated`、`prepareNavigationBarView` 被调用，以调整各按钮的显示。整体上导航栏的布局结构如图 2-2 所示。

图2-2 导航栏整体布局



## 2.1.2 工作原理

作为物理按键的替代，维护虚拟按键是导航栏最主要的工作。

1. `InputDispatcher` 类提供的 `inject InputEvent()` 方法是虚拟按键的实现基础，它会将调用者自制的一个输入事件加入 `InputDispatcher` 的派发队列中并派发。
2. `InputDispatcher::injectInputEvent()` 方法经由 `InputManager.injectInputEvent()` 方法向 java 层的使用者提供调用接口，而导航栏中的 `KeyButtonView` 就是这一接口的使用者之一。
  - `KeyButtonView` 继承自 `ImageView`，它对 `ImageView` 最主要的扩展就是将派发给它的触摸事件转化为按键事件，并通过 `InputManager.injectInputEvent()` 方法将按键事件注入 `InputDispatcher` 的派发队列。
  - `KeyButtonView` 中最重要的字段是 `mCode`，用于指示其生成的按键事件的键值。如果没有为 `KeyButtonView` 指定 `mCode`，那么它的行为就与一般的 `ImageView` 没什么实质的区别。在 `KeyButtonView` 中另外一个重要的字段是 `mSupportsLongPress`，它的取值决定了用户长按 `KeyButtonView` 时是否产生按键的长按事件。这两个字段可以在 `layout` 中通过 `systemui:keyCode` 以及 `systemui:keyRepeat` 进行设置。
  - 导航栏中有 4 个 `KeyButtonView`，分别是 `@id/back`、`@id/home`、`@id/recent_apps` 以及 `@id/menu`。
    - 除了 `@id/recent_apps` 以外，其他三个 `KeyButtonView` 都设置了相应的 `mCode`，用于产生 `KEY_BACK`、`KEY_HOME` 以及 `KEY_MENU` 三种按键事件。
    - `@id/recent_apps` 并不会产生按键事件，在其上的点击动作会显示或隐藏 `RecentPanel`。
3. `KeyButtonView` 将触摸事件转换为按键事件，其核心工作位于 `onTouchEvent()` 方法中，该方法主要代码如下：

```
public boolean onTouchEvent(MotionEvent ev) {
    final int action = ev.getAction();
    int x, y;

    switch (action) {
        case MotionEvent.ACTION_DOWN:
            //Log.d("KeyButtonView", "press");
            mDownTime = SystemClock.uptimeMillis();
            setPressed(true);
            if (mCode != 0) {
                sendEvent(KeyEvent.ACTION_DOWN, 0, mDownTime);
            } else {
                // Provide the same haptic feedback that the system offers
                // for virtual keys.
                performHapticFeedback(HapticFeedbackConstants.VIRTUAL_KEY);
            }
            if (mSupportsLongpress) {
                removeCallbacks(mCheckLongPress);
                postDelayed(mCheckLongPress, ViewConfiguration.getLongPressTimeout());
            }
            break;
        case MotionEvent.ACTION_MOVE:
            x = (int) ev.getX();
            y = (int) ev.getY();
            setPressed(x >= -mTouchSlop
                && x < getWidth() + mTouchSlop
                && y >= -mTouchSlop
                && y < getHeight() + mTouchSlop);
            break;
        case MotionEvent.ACTION_CANCEL:
            setPressed(false);
            if (mCode != 0) {
                sendEvent(KeyEvent.ACTION_UP, KeyEvent.FLAG_CANCELED);
            }
            if (mSupportsLongpress) {
                removeCallbacks(mCheckLongPress);
            }
            break;
        case MotionEvent.ACTION_UP:
    }

    return true;
}
```

- 如果 KeyButtonView 设置了 mCode，则创建并发送给 InputDispatcher 一个 ACTION\_DOWN 的按键事件。
- 如果 KeyButtonView 设置为支持长按事件，则发送一个名为 mCheckLongPress 的 Runnable，并在延迟一段时间后执行它。这个 Runnable 会重新发送一个带有 LONG\_PRESS 标记的 ACTION\_DOWN 事件。
- 当触摸事件取消，则发送一个带有 FLAG\_CANCELED 标记的 ACTION\_UP 按键事件，此时由于触摸事件被取消，故不需要发送长按事件，需要调用 removeCallbacks 方法取消尚未执行的 mCheckLongPress。
- 如果 KeyButtonView 中没有设置 mCode，则在此时触发 OnClickListener，当用户抬起按在 KeyButtonView 上的手指，就不再需要发送长按事件，调用 removeCallbacks 取消 mCheckLogPress。

简单来说，当 KeyButtonView 被设置了一个有效键值的情况下，将按照如下方式完成触摸事件到按键事件的映射：

- 触摸事件的 ACTION\_DOWN 对应按键事件的 ACTION\_DOWN。

- 触摸事件的 ACTION\_UP 对应按键事件的 ACTION\_UP。
- 触摸事件的 ACTION\_CANCEL 对应按键事件的 ACTION\_UP+FLAG\_CANCELED。
- 当用户长按 KeyButtonView 时对应按键事件的 ACTION\_DOWN+FLAG\_LONG\_PRESSED。

导航栏中的@id/back、@id/home 以及@id/menu 三个 KeyButtonView 采用了上述工作方式。

另外，如果 KeyButtonView 没有设置一个有效的键值，那么当用户点击它时并不会产生任何按键事件，而是触发监听器 OnClickListener，导航栏中的@id/recent\_apps 便采用了这种工作方式。

4. KeyButtonView 的 onTouchEvent()方法完成了从触摸事件到按键事件的映射，而 KeyButtonView 的 sendEvent()方法则完成按键事件的创建与发送。该方法代码如下所示：

```
public void sendEvent(int action, int flags) {
    sendEvent(action, flags, SystemClock.uptimeMillis());
}

private void sendEvent(int action, int flags, long when) {
    mMetricsLogger.write(new LogMaker(MetricsEvent.ACTION_NAV_BUTTON_EVENT)
        .setType(MetricsEvent.TYPE_ACTION)
        .setSubtype(mCode)
        .addTaggedData(MetricsEvent.FIELD_NAV_ACTION, action)
        .addTaggedData(MetricsEvent.FIELD_FLAGS, flags));
    // TODO(b/122195391): Added logs to make sure sysui is sending back button events
    if (mCode == KeyEvent.KEYCODE_BACK && flags != KeyEvent.FLAG_LONG_PRESS) {
        Log.i(TAG, "Back button event: " + KeyEvent.actionToString(action));
        if (action == MotionEvent.ACTION_UP) {
            mOverviewProxyService.notifyBackAction((flags & KeyEvent.FLAG_CANCELED) == 0,
                -1, -1, true /* isButton */, false /* gestureSwipeLeft */);
        }
    }
    final int repeatCount = (flags & KeyEvent.FLAG_LONG_PRESS) != 0 ? 1 : 0;
    final KeyEvent ev = new KeyEvent(mDownTime, when, action, mCode, repeatCount,
        0, KeyCharacterMap.VIRTUAL_KEYBOARD, 0,
        flags | KeyEvent.FLAG_FROM_SYSTEM | KeyEvent.FLAG_VIRTUAL_HARD_KEY,
        InputDevice.SOURCE_KEYBOARD);

    int displayId = INVALID_DISPLAY;

    // Make KeyEvent work on multi-display environment
    if (getDisplay() != null) {
        displayId = getDisplay().getDisplayId();
    }
    // Bubble controller will give us a valid display id if it should get the back event
    BubbleController bubbleController = Dependency.get(BubbleController.class);
    int bubbleDisplayId = bubbleController.getExpandedDisplayId(mContext);
    if (mCode == KeyEvent.KEYCODE_BACK && bubbleDisplayId != INVALID_DISPLAY) {
        displayId = bubbleDisplayId;
    }
    if (displayId != INVALID_DISPLAY) {
        ev.setDisplayId(displayId);
    }
    mInputManager.injectInputEvent(ev, InputManager.INJECT_INPUT_EVENT_MODE_ASYNC);
}
```

## 2.2 导航栏设置

### 2.2.1 隐藏导航栏

展锐添加了隐藏导航栏的功能，可以通过“设置->系统->导航栏”进行设置，如图 2-3 所示。

图2-3 导航栏的显示与隐藏



Google 原生通过将 `config_showNavigationBar` 配置为 `true`，直接显示导航栏。展锐在原生的基础上通过将属性 `qemu.hw.mainkeys` 值配置为 0 实现动态导航栏功能：

- 点击导航栏左下角的隐藏按钮，向数据库写入字段 `show_navigationbar` 值为 0，`WindowManager` 监听数据库字段值发生变化后隐藏导航栏。
- 从设备底部上滑则重新显示导航栏。对应的代码在 `DisplayPolicy.java` 的 `onSwipeFromBottom` 中。

### 2.2.2 设置导航模式

Android 原生为导航栏设置了不同的模式，用户可以在“设置->系统->手势->系统导航”中选择，如图 2-4 所示。

图2-4 系统导航模式设置



系统默认的导航栏是三按钮形式。导航模式是通过一系列 res RRO 进行设置，在 Settings 中的设置代码如下：

```
@VisibleForTesting
static void setCurrentSystemNavigationMode(Context context, IOverlayManager overlayManager,
    String key) {
    switch (key) {
        case KEY_SYSTEM_NAV_GESTURAL:
            int sensitivity = getBackSensitivity(context, overlayManager);
            setNavBarInteractionMode(overlayManager, BACK_GESTURE_INSET_OVERLAYS[sensitivity]);
            break;
        case KEY_SYSTEM_NAV_2BUTTONS:
            setNavBarInteractionMode(overlayManager, NAV_BAR_MODE_2BUTTON_OVERLAY);
            break;
        case KEY_SYSTEM_NAV_3BUTTONS:
            setNavBarInteractionMode(overlayManager, NAV_BAR_MODE_3BUTTON_OVERLAY);
            break;
    }
}

private static void setNavBarInteractionMode(IOverlayManager overlayManager,
    String overlayPackage) {
    try {
        overlayManager.setEnabledExclusiveInCategory(overlayPackage, USER_CURRENT);
    } catch (RemoteException e) {
        throw e.rethrowFromSystemServer();
    }
}
```

通过 `overlayManager.setEnabledExclusiveInCategory` 设置当前 enable 的 RRO package，而其他 overlay 相同 target package 的 RRO 都会被禁用。

比如设置导航模式的 RRO package 如下：

```
// Associated overlays for each nav bar mode
String NAV_BAR_MODE_3BUTTON_OVERLAY = "com.android.internal.systemui.navbar.threebutton";
String NAV_BAR_MODE_2BUTTON_OVERLAY = "com.android.internal.systemui.navbar.twobutton";
String NAV_BAR_MODE_GESTURAL_OVERLAY = "com.android.internal.systemui.navbar.gestural";
```

其中手势导航的 RRO：

xref: /sprdroid10\_trunk\_19c/frameworks/base/packages/overlays/NavigationBarModeGesturalOverlay/

Home | History | Annotate  Search ☐ current directory

| Name                |       | Date        | Size    | #Lines | LOC |
|---------------------|-------|-------------|---------|--------|-----|
| ..                  |       | 25-Feb-2020 | -       |        |     |
| res/values/         | H     | 25-Feb-2020 | -       |        |     |
| Android.mk          | H A D | 25-Feb-2020 | 904     | 30     | 8   |
| AndroidManifest.xml | H A D | 25-Feb-2020 | 1.1 KiB | 27     | 9   |

## 2.3 常见问题分析

### GSI 版本开机没有虚拟按键

**问题分析：**GSI 版本使用 Google 的 system.img，如需显示虚拟按键，只能通过 overlay 的方式将属性 `config_showNavigationBar` 的值配置为 true，示例如下。

**代码路径：**device/sprd/xxxxx/moverlay/device/base/frameworks/base/core/res/res/values/config.xml

#### 说明

代码路径中的“xxxxx”代表具体的项目。

**代码修改：**

```
<!--unisoc bug:1151685,must add SRO config for navbar, because of gsi -->
<bool name="config_showNavigationBar">true</bool>
```

### Go 版本打开动态导航栏功能

**问题分析：**打开动态导航栏功能，需要将属性 `qemu.hw.mainkeys` 配置为 0。但因为内存原因，go 版本未添加此功能，因为在低内存情况下打开此功能会导致 SystemUI 占用内存增加，从而影响整体性能。请慎重评估是否开启并将强测试。如打开此功能，go 版本还要再添加如下修改。

**代码路径：**device/sprd/mpool/module/app/systemui/mversion/go/go.mk

**代码修改：**



```
#Add for Go DynamicNavigationBar & Enabled Gesture Nav for GO.
PRODUCT_PACKAGES += \
    NavigationBarDynamicSupport \
    Launcher3QuickStepGoConfigOverlay
```

## 将手势导航设置为默认导航模式

### 错误的做法：

通过 FW res config 可直接设置导航模式是，代码如下：

```
<!-- Controls the navigation bar interaction mode:
    0: 3 button mode (back, home, overview buttons)
    1: 2 button mode (back, home buttons + swipe up for overview)
    2: gestures only for back, home and overview -->
<integer name="config_navBarInteractionMode">0</integer>
```

但通过直接修改这个 config 为 2（gestures）实现会带来一系列问题。如“2.2.2 设置导航模式”所述，导航模式是通过一系列 res RRO 进行设置的。当然可以将 FW res 中对应的默认值全部改为手势导航的值，但这并不是个合适的方法。

### 正确的做法：

正确的做法是将 NavigationBarModeGesturalOverlay 配置成默认的 RRO。Google 也提供了这种方法，通过属性 ro.boot.vendor.overlay.theme 配置 RRO 包名来达到目的，多个 package 用“;”分隔。比如这里配置手势导航可以在 board 中做如下配置：

```
PRODUCT_PROPERTY_OVERRIDES += \
    qemu.hw.mainkeys=0 \
    ro.boot.vendor.overlay.theme=com.android.internal.systemui.navbar.gestural
```

## 在应用中隐藏 home 键和 recent 键，只保留 back 键

可以在应用中先获取 StatusBar 的 service，然后设置对应的属性，代码示例如下：

```
mStatusBarManager = (StatusBarManager) getSystemService(Context.STATUS_BAR_SERVICE);
mStatusBarManager.disable(StatusBarManager.DISABLE_HOME | StatusBarManager.DISABLE_RECENT);
退出时将属性设回即可
mStatusBarManager.disable(StatusBarManager.DISABLE_NONE);
```

## 隐藏应用界面导航栏

隐藏应用界面导航栏需要应用设置相应的 flag，示例如下：

```
View decorView = this.getWindow().getDecorView();
decorView.setSystemUiVisibility(
    View.SYSTEM_UI_FLAG_HIDE_NAVIGATION
    | View.SYSTEM_UI_FLAG_FULLSCREEN
    | View.SYSTEM_UI_FLAG_IMMERSIVE_STICKY);
```

### 说明

此种方式设置的导航栏是沉浸式的，从底部上滑即可向导航栏显示出来。