Android15 SystemUI

重要改变指南

版本号: 1.2
发布日期: 2024.10.31

深圳市宇芯数码技术有限公司yfzy_01

版本历史

版本号	日期	制/修订人	内容描述
1.0	2022.11.20	AWA1741	android13 初始版本文档
1.1	2023.11.15	AWA1741	android14 更新描述
1.2	2024.10.31	AWA2222	android15 更新插图，描述

深圳市宇芯数码技术有限公司yfzy_01

版权所有 © 珠海全志科技股份有限公司。保留一切权利

目　录

1 前言 1
1.1 概述 1
1.2 读者对象 1
1.3 关键字 1

2 简介 2
2.1 主要功能 2
2.2 源码位置 2
2.3 安装位置 2
2.4 整体架构 2
2.5 启动流程 4

3 状态栏 7
3.1 基本功能 7
3.2 图标类型 7
3.3 图标显示 7

4 导航栏 9
4.1 初始化 9
4.2 点击事件 11
4.3 Taskbar 新特性 12

5 下拉菜单 14
5.1 基本介绍 14
5.2 QuickQuickSettings 新特性 18
5.3 快捷设置 19
5.3.1 关键类 19
5.3.2 客制化配置 20
5.3.3 新增快捷设置 20

6 壁纸 21
6.1 锁屏壁纸 21
6.2 桌面壁纸 22
6.3 客制化配置 24

7 锁屏 25
7.1 关键类 25
7.2 Power 键灭屏锁屏 25
7.3 客制化配置 29

版权所有 © 珠海全志科技股份有限公司。保留一切权利

深圳市宇芯数码技术有限公司yfzy_01

插　图

图 2-1 systemui 架构 3
图 2-2 systemui 核心模块关系图 4
图 2-3 systemui 启动 5
图 3-1 SystemUI 显示 Alarm 流程 8
图 4-1 导航栏初始化 9
图 4-2 Taskbar 展示 12
图 5-1 qqs 面板 15
图 5-2 横屏状态下拉菜单显示 18
图 5-3 创建 QSTile 流程 19
图 6-1 锁屏壁纸 22
图 6-2 桌面壁纸 23
图 7-1 Power 键灭屏显示锁屏 framework 层处理 26
图 7-2 Power 键灭屏后显示锁屏 systemui 层处理 28

版权所有 © 珠海全志科技股份有限公司。保留一切权利

1 前言

1.1 概述

本文档主要适用于 Android15，主要用来介绍 Android 15 中对 SystemUI 的重要改变。

1.2 读者对象

本文档适用于 Android 开发人员。

1.3 关键字

状态栏、导航栏、QuickSettings、QuickQuickSettings、下拉菜单、通知栏。

2 简介

SystemUI 是一个系统级的 APK，也在单独的进程中运行，作为 Android 系统的最核心应用之一，它负责反馈系统状态并与用户保持交互。

2.1 主要功能

- 状态栏，比如电池电量，wifi 信号，闹钟等图标的显示。
- 导航栏，比如 Home 键、返回键、最近任务键等。
- 音量 VolumeUI，控制媒体、铃音、通知等音量大小。
- 截屏，长按电源键 + 音量减键或 AWUI 中截屏键截屏。
- 电源管理 PowerUI，负责监控电源的变化和通知，主要处理和 Power 相关的事件。
- 通知栏，比如系统消息，第三方应用消息。
- 壁纸服务，用于显示壁纸。
- 锁屏 Keyguard，解锁设置密码等。

2.2 源码位置

```
frameworks/base/packages/SystemUI
```

2.3 安装位置

```
/system/system_ext/priv-app/SystemUI
```

2.4 整体架构

首先，SystemUI 的整体架构如下图所示：

版权所有 © 珠海全志科技股份有限公司。保留一切权利

SystemUI 数据层: KeyguardSecurityModel, MediaButton, ThumbnailData, DataSaverTile, ...
SystemUI 控制层: ScrimController, BatteryController, NotificationListenerController, ScreenshotController, ...
SystemUI 视图层: PhoneStatusBarView, NavigationBarInflaterView, NotificationStackScrollLayout, ScrimView, ...
SystemUI service 层: TakeScreenshotService, KeyguardService, RecordingService
SystemUI核心模块层: KeyguardViewMediator, PowerUI, StatusBar, VolumeUI, WMShell, SystemActions, ...
SystemUI 启动层: SystemUIService, SystemUIApplication
SystemUI层
SystemUI 对接服务实现跨进程调用: StatusBarManagerService
SystemUI 在框架层关联的系统服务: ActivityManagerService, WindowManagerService, PowerManagerService, NotificationManagerService
framework层

图 2-1: systemui 架构

其次，SystemUI 分几个大的核心模块，如下图所示：

深圳市宇芯数码技术有限公司yfzy_01

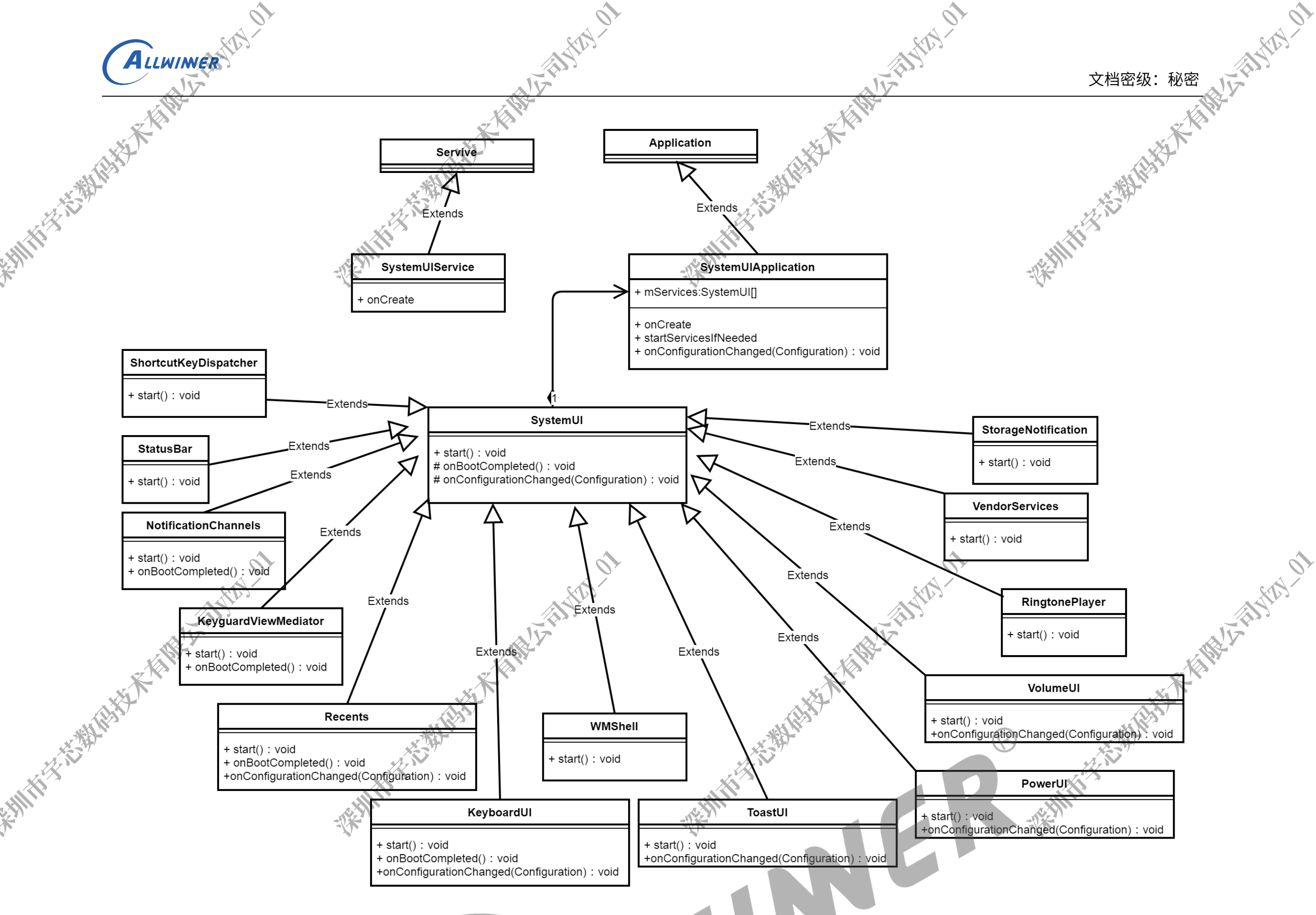

图 2-2: systemui 核心模块关系图

2.5 启动流程

SystemUI 的启动是在 SystemServer 进程里开始，大致流程如下：

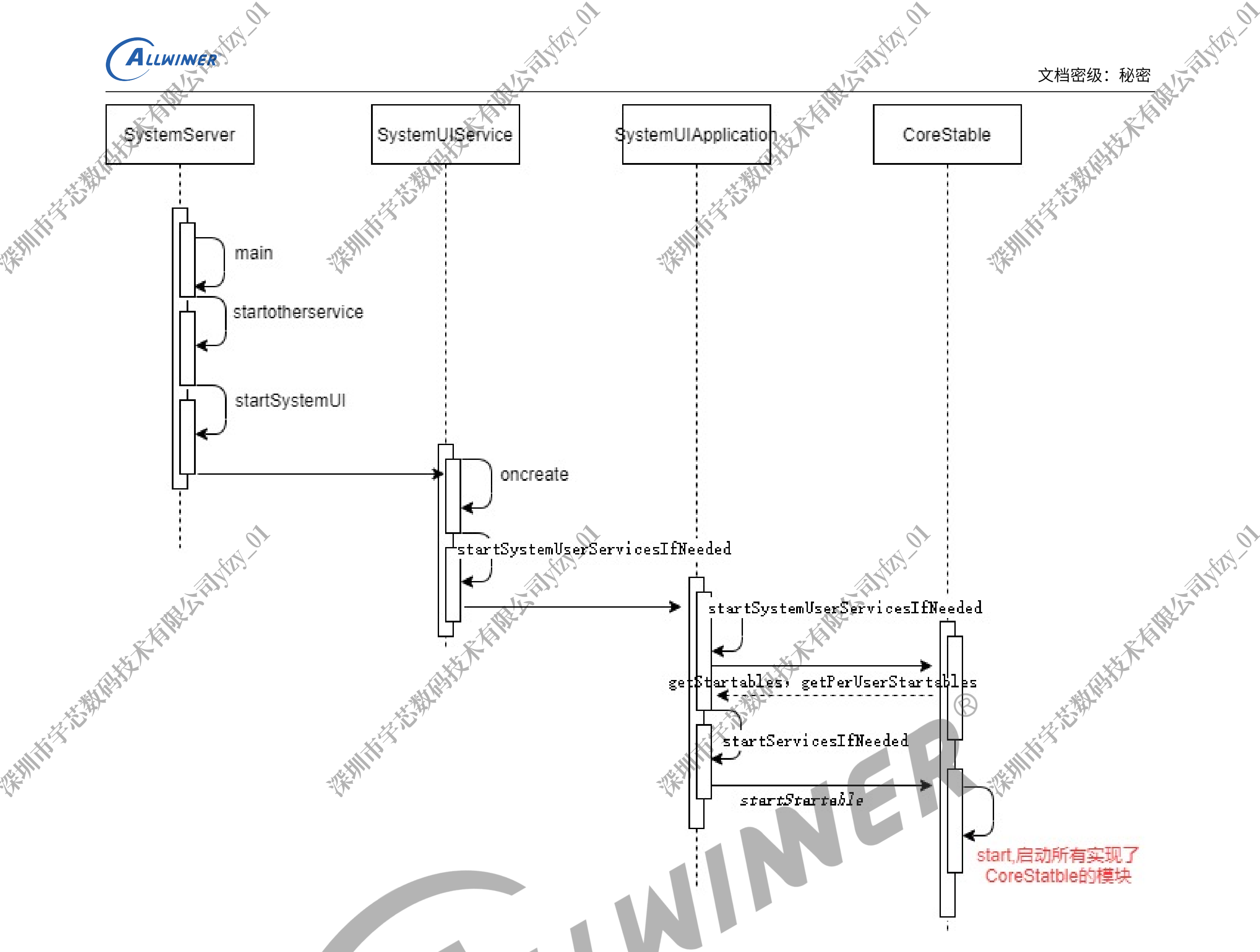

图 2-3: systemui 启动

在 Android15 的版本中，SystemUI 的核心服务的启动方式发生了较大的变化。之前的版本是通过反射的方式将各个服务进行实例化，并执行服务的 start 方法，而在 Android 15 中，不再使用反射的方式进行实例化，而是将所有服务都实现 CoreStartable.java 接口，并通过 Dagger 进行对象的创建与赋值，摒弃了 config_systemUIServiceComponents 的配置属性，核心代码如下：

```
final String vendorComponent = mInitializer.getVendorComponent(getResources());

        // Sort the startables so that we get a deterministic ordering.
        // TODO: make #start idempotent and require users of CoreStartable to call it.
        Map<Class<?>, Provider<CoreStartable>> sortedStartables = new TreeMap<>(
                Comparator.comparing(Class::getName));
        sortedStartables.putAll(mSysUIComponent.getStartables());//获取需要跟application一起启动的CoreStartable
        sortedStartables.putAll(mSysUIComponent.getPerUserStartables());//需要为每个用户都启动的CoreStartable

        startServicesIfNeeded(
                sortedStartables, "StartServices", vendorComponent);
```

对于 CoreStartable 的可以这样理解：需要跟随 SystemUI 一起启动的代码都会实现 CoreStartable 接口，这样这些相关类将在 systemui 进程创建时一起初始化。

主要涉及的文件如下：

版权所有 © 珠海全志科技股份有限公司。保留一切权利

1.frameworks/base/packages/SystemUI/src/com/android/systemui/SystemUIApplication.java (SystemUI应用入口，负责启动服务)
2.frameworks/base/packages/SystemUI/src/com/android/systemui/CoreStartable.java (所有跟随systemui启动的模块所的实现的接口)

深圳市宇芯数码技术有限公司yfzy_01

版权所有 © 珠海全志科技股份有限公司。保留一切权利

3 状态栏

3.1 基本功能

状态栏是 SystemUI 中的重要功能之一，用于显示功能图标和手机基本信息。在 Android 15 版本中，其主要包含：

- 功能图标：通知图标、状态栏图标、WiFi 图标等。
- 基本信息：电池信息、时间、日期等。

状态栏也分三种：

- 桌面状态栏，其 UI 视图在 **status_bar.xml** 。
- 锁屏状态栏，其 UI 视图在 **keyguard_status_bar.xml** 。
- 下拉状态栏，其 UI 视图在 **quick_status_bar_expanded_header.xml**。

3.2 图标类型

Android 15 版本里的状态栏图标 View 控件分为以下三种：

- **StatusBarWifiView**：**wifi** 图标
- **StatusBarMobileView**：信号图标
- **StatusBarIconView**：其他图标，如蓝牙、热点、闹钟等

状态栏图标在某些功能发现变化时才会显示或者更新状态栏图标。其控制类是 **StatusBarSignalPolicy.java**（主要控制信号类状态栏图标 **wifi/vpn/mobile**）和 **PhoneStatusBarPolicy.java**（主要控制状态类状态栏图标 **volume/alarm/bluetooth**）中。

3.3 图标显示

状态栏图标的显示流程基本类似，只是功能控制类不同。都是通过各自的 Controller 类监听功能状态，再通过对应的 Policy 类来注册回调 Callback 给 Controller，最后通过 Controller 来刷新图标 View 视图。以闹钟 Alarm 响起显示在状态栏为例，其调用流程如下：

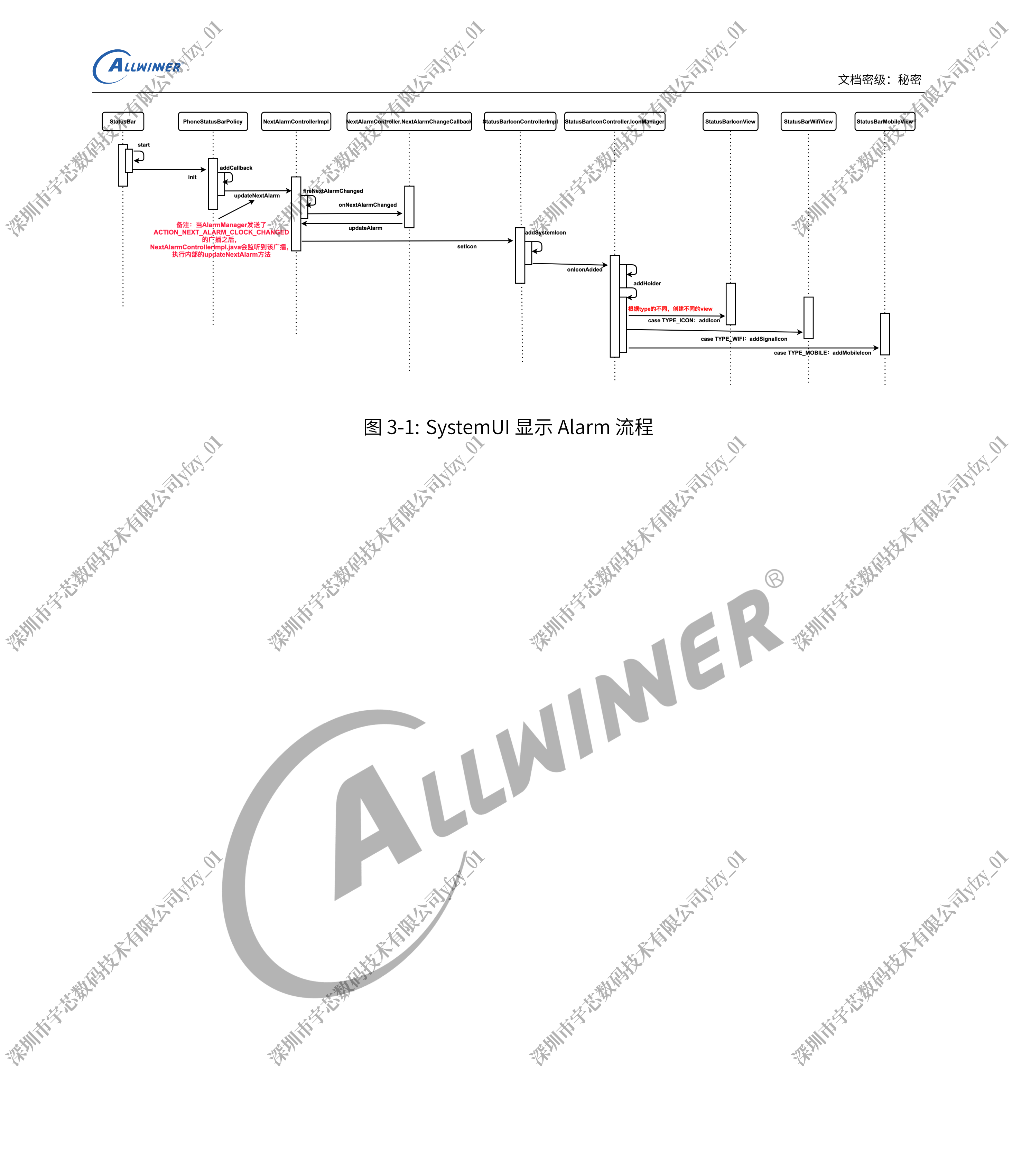

图 3-1: SystemUI 显示 Alarm 流程

4 导航栏

导航栏也是 SystemUI 中的重要功能之一，方便用户操作系统界面。在 Android 15 版本中，其主要包含：

- 返回 Back 键（原生）
- 桌面 Home 键（原生）
- 最近任务 Recent 键（原生）
- 音量 VolumeUp 键（全志 AWUI 2.0 定制化）
- 音量 VolumeDown 键（全志 AWUI 2.0 定制化）
- 隐藏 Hide 键（全志 AWUI 2.0 定制化）
- 截图 Screenshot 键（全志 AWUI 2.0 定制化）

4.1 初始化

在 Android 15 中，导航栏的初始化是在 CentralSurfacesImpl.java 里面的 createAndAddWindows 方法。在 Android 15 中其流程大致如下图所示：

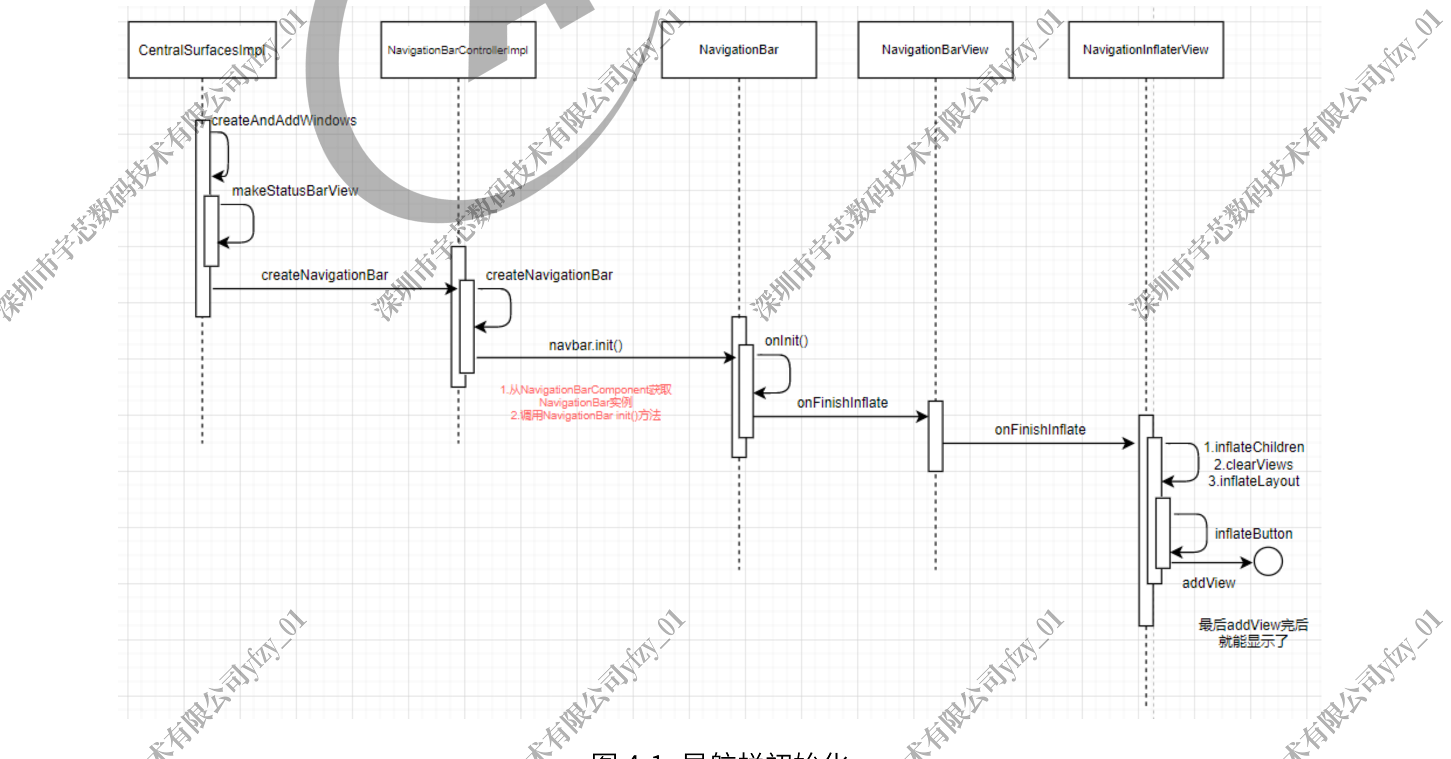

图 4-1: 导航栏初始化

版权所有 © 珠海全志科技股份有限公司。保留一切权利

导航栏的 UI 视图是 **navigation_bar.xml**，源码如下：

```
<com.android.systemui.navigationbar.NavigationBarView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/navigation_bar_view"
    android:layout_height="match_parent"
    android:layout_width="match_parent"
    android:clipChildren="false"
    android:clipToPadding="false"
    android:background="@drawable/system_bar_background">

    <com.android.systemui.navigationbar.NavigationBarInflaterView
        android:id="@+id/navigation_inflater"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:clipChildren="false"
        android:clipToPadding="false" />

</com.android.systemui.navigationbar.NavigationBarView>
```

其中，NavigationBarInflaterView 负责导航栏视图的绘制更新，导航栏的按钮不是直接写在 navigation_bar.xml 文件里面的，而是在 NavigationBarInflaterView 里面通过 createView 方法创建一个 View 之后通过 View.addView 的方式来实现的。createView 方法源码如下：

```
View createView(String buttonSpec, ViewGroup parent, LayoutInflater inflater) {
    //省略...
    if (HOME.equals(button)) {
      v = inflater.inflate(R.layout.home, parent, false);//home按钮布局
    } else if (BACK.equals(button)) {
      v = inflater.inflate(R.layout.back, parent, false);//返回按钮布局
    } else if (RECENT.equals(button)) {
      v = inflater.inflate(R.layout.recent_apps, parent, false);//最近任务按钮布局
    } else if (HIDE.equals(button)) {
      v = inflater.inflate(R.layout.hide, parent, false);//隐藏按钮布局
    } else if (VOLUMEUP.equals(button)) {
      v = inflater.inflate(R.layout.volumeup, parent, false);//音量加按钮布局
    } else if (VOLUMEDOWN.equals(button)) {
      v = inflater.inflate(R.layout.volumedown, parent, false);//音量减按钮布局
    } else if (SCREENSHOT.equals(button)) {
      v = inflater.inflate(R.layout.screenshot, parent, false);//截图按钮布局
    }
    //省略...
}
```

我们以 Home 键的 home.xml 为例，源码如下：

```
<com.android.systemui.navigationbar.buttons.KeyButtonView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:systemui="http://schemas.android.com/apk/res-auto"
    android:id="@+id/home"
    android:layout_width="@dimen/navigation_key_width"
    android:layout_height="match_parent"
    android:layout_weight="0"
    systemui:keyCode="3"
    android:scaleType="center"
    android:contentDescription="@string/accessibility_home"
    android:paddingStart="@dimen/navigation_key_padding"
    android:paddingEnd="@dimen/navigation_key_padding"
```

```
/>
```

4.2 点击事件

根据上面源码分析，Home 键的视图 View 是 KeyButtonView。查看其点击事件，主要跟踪其 onTouchEvent 方法，源码如下：

```
public boolean onTouchEvent(MotionEvent ev) {
  switch (action) {
    case MotionEvent.ACTION_UP:
        //省略...
        //1. mCode 就是获取xml文件里面systemui:keyCode
        if (mCode != KEYCODE_UNKNOWN) {
            if (doIt) {
                sendEvent(KeyEvent.ACTION_UP, 0);
                sendAccessibilityEvent(AccessibilityEvent.TYPE_VIEW_CLICKED);
            } else {
                sendEvent(KeyEvent.ACTION_UP, KeyEvent.FLAG_CANCELED);
            }
        } else {
            // 没有systemui:keyCode，则执行onClick方法，比如最近任务按钮
            if (doIt && mOnClickListener != null) {
                mOnClickListener.onClick(this);
                sendAccessibilityEvent(AccessibilityEvent.TYPE_VIEW_CLICKED);
            }
        }
        //省略...
        break;
  }
}

private void sendEvent(int action, int flags, long when) {
    //省略...
    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);

    //省略...
    //3.构建出一个对应的keyCode的 KeyEvent，然后调用InputManager的injectInputEvent模拟发送来实现跟物理按键一
      样的功能
    mInputManager.injectInputEvent(ev, InputManager.INJECT_INPUT_EVENT_MODE_ASYNC);
}
```

在上面分析到有的虚拟按钮时没有 systemui:keyCode 的属性，那么它们的点击事件在哪调用呢？我们以最近任务按钮举例，其点击事件是在 NavigationBar.java 里面注册，源码如下：

```
@Override
public void onViewAttachedToWindow(View v) {
    //省略...
    //1.调用prepareNavigationBarView
    prepareNavigationBarView();
    //省略...
}
```

版权所有 © 珠海全志科技股份有限公司。保留一切权利

```
private void prepareNavigationBarView() {
    //省略...
    ButtonDispatcher recentsButton = mNavigationBarView.getRecentsButton();
    //2.注册点击事件
    recentsButton.setOnClickListener(this::onRecentsClick);
    recentsButton.setOnTouchListener(this::onRecentsTouch);
     //省略...
}

//3.最近任务的点击事件
private void onRecentsClick(View v) {
    if (LatencyTracker.isEnabled(mContext)) {
      LatencyTracker.getInstance(mContext).onActionStart(
        LatencyTracker.ACTION_TOGGLE_RECENTS);
    }
    mStatusBarOptionalLazy.get().ifPresent(StatusBar::awakenDreams);
    mCommandQueue.toggleRecentApps();
}
```

4.3 Taskbar 新特性

在 Android 15 中，Taskbar 模块现在由 Launcher 来负责。Taskbar 也实现了导航栏的功能，UI 效果如下图所示：

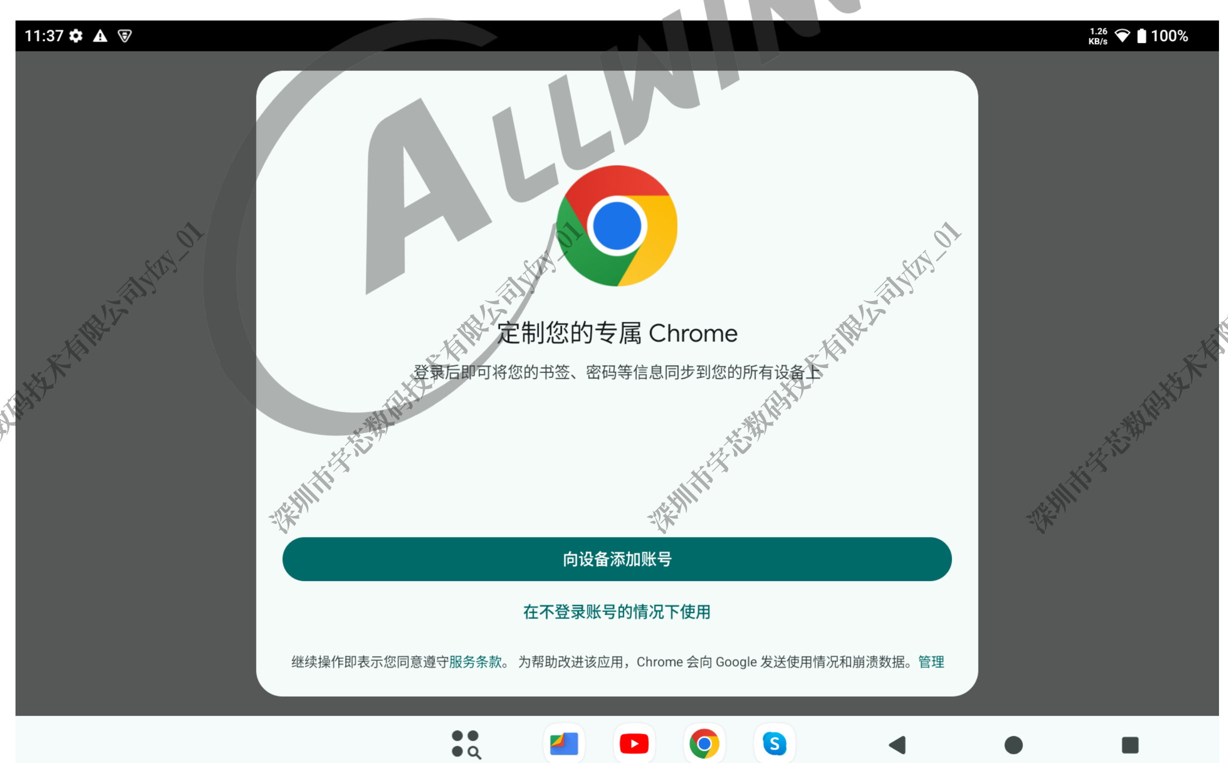

图 4-2: Taskbar 展示

Taskbar 取代了原本由 SystemUI 负责实现导航栏的功能。我们可以从源码可以窥探一二，在 SystemUI 中创建导航栏的入口在 NavigationBarController.java 的 createNavigationBars 方法里面，

入口如下：

```
public void createNavigationBars(final boolean includeDefaultDisplay,
        RegisterStatusBarResult result) {
    updateAccessibilityButtonModeIfNeeded();

    // Don't need to create nav bar on the default display if we initialize TaskBar.
    final boolean shouldCreateDefaultNavbar = includeDefaultDisplay
            && !initializeTaskbarIfNecessary();
    Display[] displays = mDisplayTracker.getAllDisplays();
    for (Display display : displays) {
        if (shouldCreateDefaultNavbar
                || display.getDisplayId() != mDisplayTracker.getDefaultDisplayId()) {
            createNavigationBar(display, null /* savedState */, result);
        }
    }
}
```

跟踪 initializeTaskbarIfNecessary 方法，如下：

```
private boolean initializeTaskbarIfNecessary() {
    // Enable for tablet or (phone AND flag is set); assuming phone = !mIsTablet
    //boolean taskbarEnabled = isTaskbarEnabled(mContext)
    //              || ((mIsLargeScreen
    //                  || mFeatureFlags.isEnabled(Flags.HIDE_NAVBAR_WINDOW))
    //              && shouldCreateNavBarAndTaskBar(mContext.getDisplayId()));
    boolean taskbarEnabled = isTaskbarEnabled(mContext)
                || (mFeatureFlags.isEnabled(Flags.HIDE_NAVBAR_WINDOW)
                && shouldCreateNavBarAndTaskBar(mContext.getDisplayId()));
    if (taskbarEnabled) {
        Trace.beginSection("NavigationBarController#initializeTaskbarIfNecessary");
        final int displayId = mContext.getDisplayId();
        // Hint to NavBarHelper if we are replacing an existing bar to skip extra work
        mNavBarHelper.setTogglingNavbarTaskbar(mNavigationBars.contains(displayId));
        // Remove navigation bar when taskbar is showing
        removeNavigationBar(displayId);
        mTaskbarDelegate.init(displayId);
        mNavBarHelper.setTogglingNavbarTaskbar(false);
        Trace.endSection();
    } else {
        mTaskbarDelegate.destroy();
    }
    return taskbarEnabled;
}
```

从源码可见，taskbarEnabled 的真假会影响 taskbar 和 navbar 的创建与否。这里可以添加定制。

提示：smallest width 可以在 Settings 里面的开发者模式下设置。

主要涉及的文件如下：

```
1.frameworks/base/packages/SystemUI/src/com/android/systemui/statusbar/phone/CentralSurfacesImpl.java (核心服务类，与Android14之前版本的StatusBar.java功能一样，换了名字)
2.frameworks/base/packages/SystemUI/src/com/android/systemui/navigationbar/NavigationBarController.java (负责创建导航栏的类)
```

5 下拉菜单

5.1 基本介绍

首先，下拉状态面板分为两种状态：

- **Quick Quick Settings (QQS)**：即初级展开面板，是一次下拉后看到的简版 UI，包含少量的开关。其显示效果如下图所示：，如下图显示：

图 5-1: qqs 面板

- **Quick Settings (QS)**： 完整 QS 面板，是二次下拉后看到的完整版 UI，其包含更多的开关。如下图显示：

版权所有 © 珠海全志科技股份有限公司。保留一切权利

4:58 10月30日周三
100%
互联网
有可用的网络
蓝牙
已关闭
勿扰
已关闭
闹钟
未设置闹钟
飞行模式
已关闭
电脑模式
关闭
设备控制器
不可用
钱包
不可用
自动屏幕旋转
已关闭

5.2 QuickQuickSettings 新特性

在 Android15 中，下拉菜单有个很大的变化，主要在横屏状态下由原先的 QuickQuickSettings 也就是下拉后看到只有少量开关的面板，改成了分屏展示。如下图所示：

图 5-2: 横屏状态下拉菜单显示

这部分的代码逻辑主要由一个 config_use_split_notification_shade 的属性去控制是否需要分屏效果。在横屏状态下，该值为 true。竖屏则为 false。config_use_split_notification_shade 属性可以通过 overlay 的方式来实现定制化的功能，该属性在如下文件里：

```
1.frameworks/base/packages/SystemUI/res/values/config.xml
```

下拉菜单的 UI 部分是在 status_bar_expanded.xml 视图里，而 QuickSettings 和通知栏部分的根节点是 NotificationsQuickSettingsContainer，对应的控制器是 NotificationsQSContainerController。分屏效果的逻辑主要在 NotificationsQSContainerController 里，该控制器是在 Android 14 上新增的。主要涉及的文件如下：

```
1.frameworks/base/packages/SystemUI/res/layout/status_bar_expanded.xml (下拉菜单的主视图)
2.frameworks/base/packages/SystemUI/src/com/android/systemui/statusbar/phone/
    NotificationsQuickSettingsContainer.java (下拉菜单中QuickSettings和通知栏部分的根节点的View)
3.frameworks/base/packages/SystemUI/src/com/android/systemui/statusbar/phone/
    NotificationsQSContainerController.kt (下拉菜单中QuickSettings和通知栏部分的根节点的Controller)
```

版权所有 © 珠海全志科技股份有限公司。保留一切权利

5.3 快捷设置

下拉面板中的 QS 状态和 QQS 状态下都可以进行一些快捷设置的功能，比如 wifi、蓝牙、录屏等功能。

5.3.1 关键类

- **QS**：最顶层容器，主要处理 QS 面板的展开和收起的逻辑。其子类有 **QSFragment**。
- **QSPanel**：承载整个快捷设置（包括 QQS 和 QS 状态下）的容器（继承 LinearLayout）。其子类有 **QuickQSPanel** （QQS 状态下的部分快捷开关）。其内部的 **QSTileLayout** 接口定义了很多方法，如设置显示多少行和多少列等，滑动页 **PagedTileLayout** 是实现了这个接口 QSTile-Layout 。
- **QSTileView**：单个快捷设置的视图展示的容器（继承 LinearLayout），其子类有 **QSTile-ViewImpl** 和 **CustomizeTileView**（编辑页面）。
- **QSFooterView**：QS 状态下底部的容器（继承 FrameLayout）。
- **QSTile**：负责快捷开关的逻辑处理，其子类有 **QSTileImpl** 。**CustomTile**（编辑）、**WifiTile**（wifi）、**BluetoothTile** 都是继承 **QSTileImpl** ，负责对应模块的逻辑。
- **QSHost**：主要负责快捷开关 **QSTileView** 和 **QSTile** 的构建，提供获取 QSTile 集合的接口，提供了展开和收起下拉面板的接口，是快捷开关对外沟通的桥梁。其实现类有 **QSTileHost**，QSTileHost 内部有 **QSFactory** （负责创建 QSTile 和 QSTileView）。

在 Android 15 中，快捷设置的初始化流程与 Android 13 基本一致，流程如下：

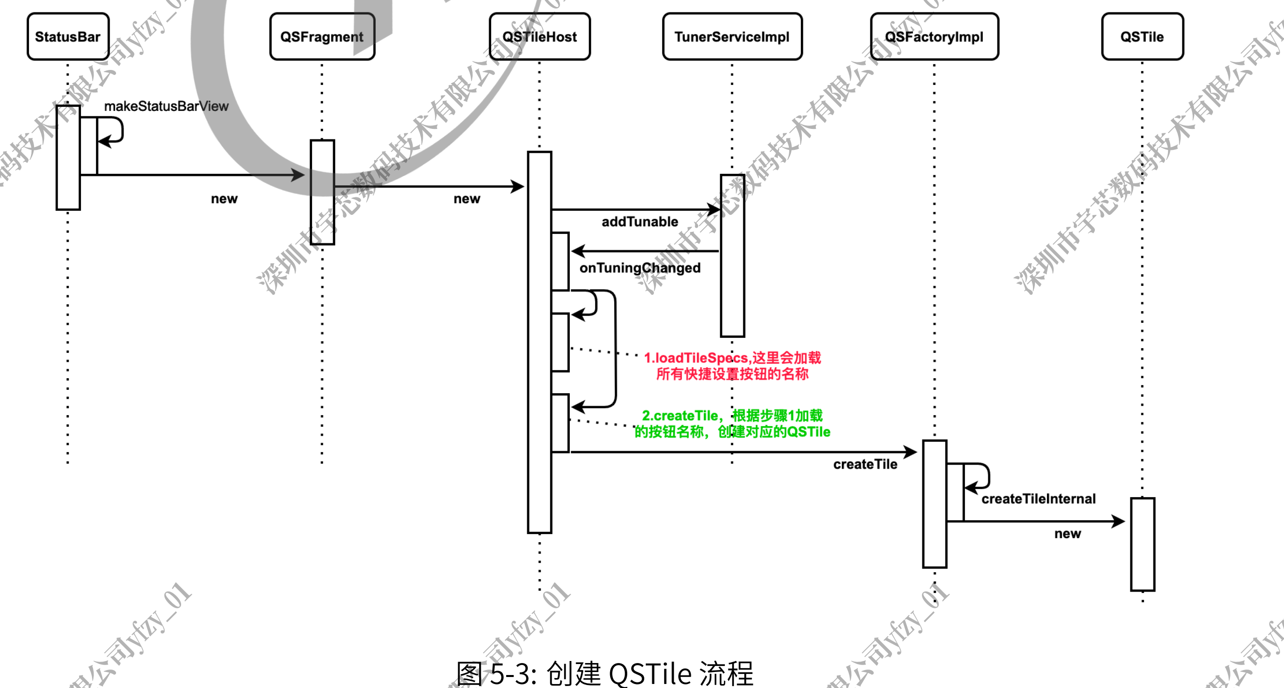

图 5-3: 创建 QSTile 流程

版权所有 © 珠海全志科技股份有限公司。保留一切权利

5.3.2 客制化配置

1. 设置快捷设置中 QQS 状态下的快捷按钮个数。overlay systemui 内部配置 "quick_qs_panel_max_tiles"
2. 默认QS状态所有快捷设置按钮的名称。 overlay systemui 内部配置 "quick_settings_tiles_default"
3. 默认SystemUI 所有快捷设置按钮的名称，包括编辑界面下可拖动的按钮。overlay systemui 内部配置 "quick_settings_tiles_stock"
4. 快捷设置界面显示的列数。overlay systemui 内部配置 "quick_qs_panel_max_columnsquick_settings_num_columns"
5. 快捷设置界面显示最大的行数。overlay systemui 内部配置 "quick_settings_max_rows"
6. QS状态下按钮最少要显示的个数。overlay systemui 内部配置 "quick_settings_min_num_tiles"

5.3.3 新增快捷设置

在 SystemUI 中新增一个快捷设置的步骤如下：

1. 增加 xxxTile. java、xxxController.java、xxxControllerImpl.java 以及相应的资源文件。xxxTile.java 继承 QSTileImpl<TState>、xxxController.java 继承 CallbackController<Callback>、xxxControllerImpl.java 继承 xxxController.java。
2. 在 QSFactoryImpl.java 的 createTile 方法中添加 new xxxTile()。
3. 在 frameworks/base/package/SystemUI/res/values/config.xml 里面对 "quick_settings_tiles_default" 添加对应功能的 xxx。

版权所有 © 珠海全志科技股份有限公司。保留一切权利

6 壁纸

在 Android 15 上，点击壁纸应用的设置壁纸按钮后，最终是调用到 WallpaperManagerService 的 setWallpaper 方法，而壁纸的最终显示放在 SystemUI 中。根据源码可知，WallpaperManagerService 里面有个 WallpaperObserver（继承原生的 FileObserver，就是监听文件的变化）用来监听壁纸文件的更新，在其 onEvent 方法中会有锁屏壁纸的通知回调以及桌面壁纸的处理方法。其中，壁纸分锁屏壁纸和桌面壁纸。下面做重点介绍。

6.1 锁屏壁纸

在 WallpaperObserver 的 onEvent 事件里面会有锁屏壁纸的通知回调，其方法是 notifyLockWallpaperChanged，该方法源码如下：

```
private void notifyLockWallpaperChanged() {
    final IWallpaperManagerCallback cb = mKeyguardListener;
    if (cb != null) {
        try {
            // 调用onWallpaperChanged方法进行回调
            cb.onWallpaperChanged();
        } catch (RemoteException e) {
            // Oh well it went away; no big deal
        }
    }
}
```

我们可以看到 IWallpaperManagerCallback 这个类，它其实是 IWallpaperManagerCallback.aidl，用来实现 WallpaperManagerService 与 SystemUI 锁屏服务对接的接口，源码如下：

```
oneway interface IWallpaperManagerCallback {
    /**
     * Called when the wallpaper has changed
     */
    void onWallpaperChanged();

    /**
     * Called when wallpaper colors change
     */
    void onWallpaperColorsChanged(in WallpaperColors colors, int which, int userId);

}
```

实现 IWallpaperManagerCallback Binder 服务端的是 SystemUI 内部的 LockscreenWallpaper.java 类，那么就可以直接看锁屏壁纸的显示流程了，如下图所示：

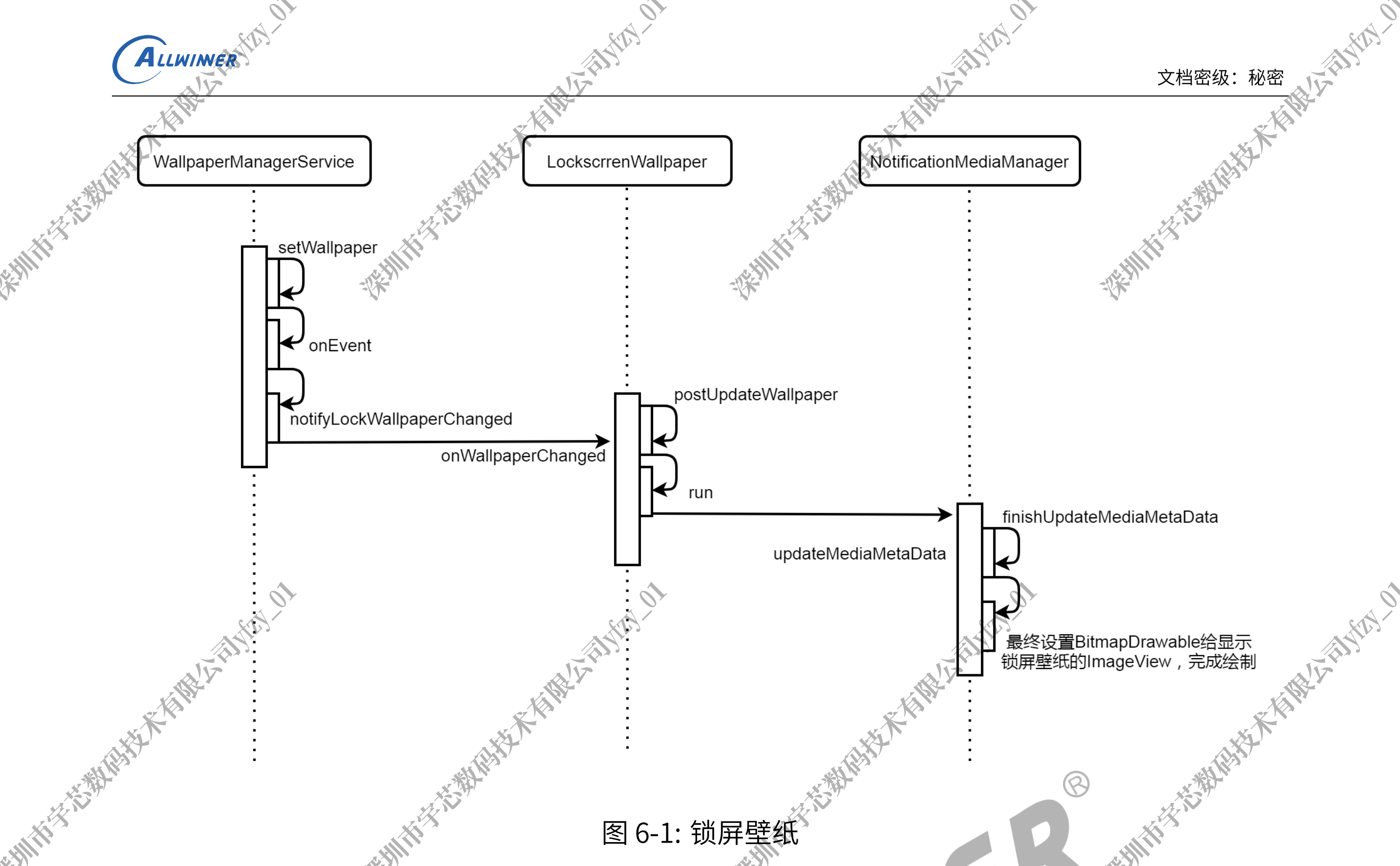

图 6-1: 锁屏壁纸

6.2 桌面壁纸

在 WallpaperManagerService 的 WallpaperObserver 的 onEvent 事件里面会有桌面壁纸的处理方法，其方法是 bindWallpaperComponentLocked，源码如下：

```
boolean bindWallpaperComponentLocked(ComponentName componentName, boolean force,
            boolean fromUser, WallpaperData wallpaper, IRemoteCallback reply) {
    //省略...
    Intent intent = new Intent(WallpaperService.SERVICE_INTERFACE);
    //省略...
    //1.创建ServiceConnection
    WallpaperConnection newConn = new WallpaperConnection(wi, wallpaper, componentUid);

    //2.componentName是传入进来的R.string.image_wallpaper_component，也就是SystemUI里面的ImageWallpaper
      组件。
    if (DEBUG) Slog.v(TAG, "Binding to:" + componentName);
    final int componentUid = mIPackageManager.getPackageUid(componentName.getPackageName(),
            MATCH_DIRECT_BOOT_AUTO, wallpaper.userId);
    WallpaperConnection newConn = new WallpaperConnection(wi, wallpaper, componentUid);
    intent.setComponent(componentName);
    intent.putExtra(Intent.EXTRA_CLIENT_LABEL,
            com.android.internal.R.string.wallpaper_binding_label);
    intent.putExtra(Intent.EXTRA_CLIENT_INTENT, clientIntent);
    int bindFlags = Context.BIND_AUTO_CREATE | Context.BIND_SHOWING_UI
                        | Context.BIND_FOREGROUND_SERVICE_WHILE_AWAKE
                        | Context.BIND_INCLUDE_CAPABILITIES;

     //3.通过bindService的方式启动ImageWallpaper服务。
    if (mContext.getResources().getBoolean(
          com.android.internal.R.bool.config_wallpaperTopApp)) {
```

```
                bindFlags |= Context.BIND_SCHEDULE_LIKE_TOP_APP;
            }
            boolean bindSuccess = mContext.bindServiceAsUser(intent, newConn, bindFlags,
            getHandlerForBindingWallpaperLocked(), new UserHandle(serviceUserId));
        //省略...
        }
        //省略...
}
```

我们可以看到，WallpaperManagerService 是通过 bindService 的方法来启动 SystemUI 里面的 ImageWallpaper 服务的，那么它们是通过 IWallpaperService.aidl 来实现通信的，源码如下：

```
oneway interface IWallpaperService {
        void attach(IWallpaperConnection connection,
                IBinder windowToken, int windowType, boolean isPreview,
                int  reqWidth, int reqHeight, in Rect padding, int displayId);
        void detach();
}
```

而实现 IWallpaperService Binder 服务端的是 WallpaperService.IWallpaperServiceWrapper 内部类。ImageWallpaper 主要是使用 OpenGL 的方式来实现渲染绘制的。详细的绘制流程如图所示：

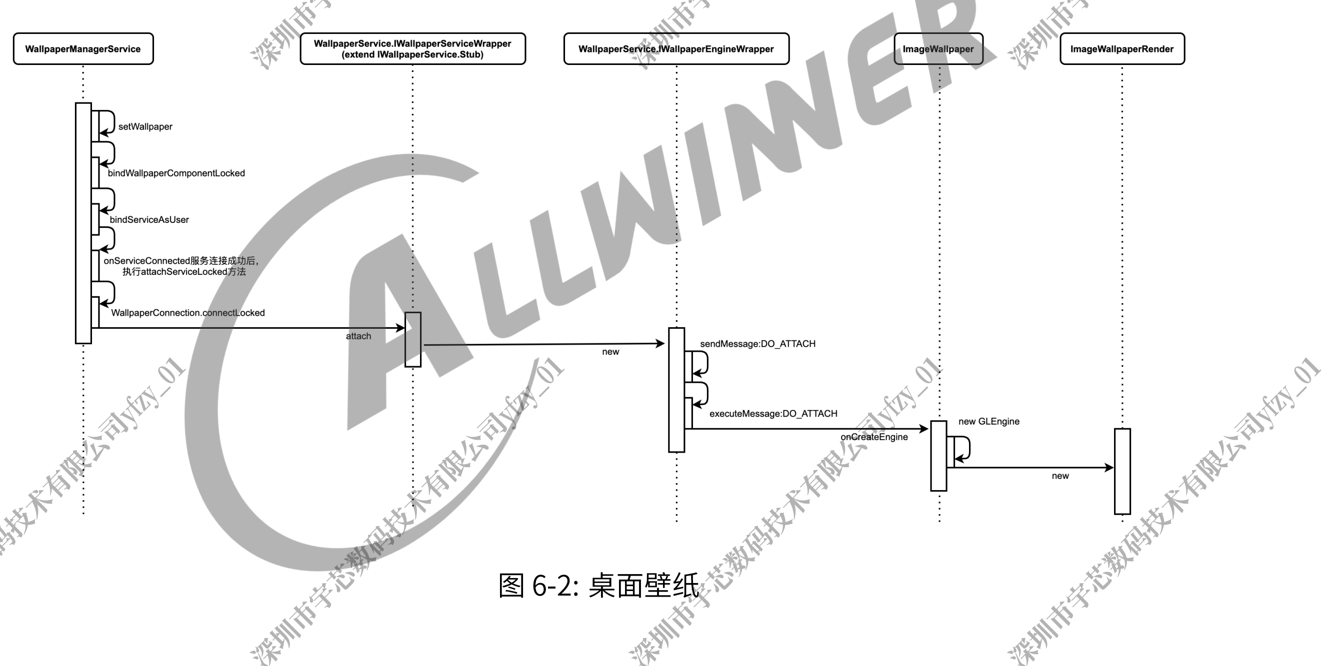

图 6-2: 桌面壁纸

之所以这样设计，是谷歌为了让开发者能实现自己的动态壁纸服务，直接继承 WallpaperService ，其中必须实现的抽象方法 onCreateEngine 方法创建一个 Engine 对象，由它来完成动态壁纸的渲染绘制。以及重写其 drawFrame 绘制方法（不断渲染）。SystemUI 内部这个 ImageWallpaper 是只展示一张静态壁纸的功能。（这里开发可以实现动态壁纸的定制化需求）

版权所有 © 珠海全志科技股份有限公司。保留一切权利

6.3 客制化配置

1.修改默认壁纸：

overlay: frameworks/base/core/res/res/xxx(分辨率)/default_wallpaper.png

深圳市宇芯数码技术有限公司yfzy_01

7 锁屏

锁屏 keyguard 是 SystemUI 的核心模块之一，包括了按 Power 键休眠唤醒亮灭屏、密码锁认证、通知等功能交叉在一起。

7.1 关键类

- **KeyguardService**：framework 层处理锁屏相关的服务 Service。
- **KeyguardViewMediator**：SystemUI 核心服务之一，负责处理 SystemUI 的所有和锁屏界面有关的操作，该类主要被 KeyguardService 调用。其有一个启动锁屏的关键函数 doKeyguard-Locked , 该函数内会检测是否开启了锁屏功能，如果没有开启则直接 return。
- **StatusBarKeyguardViewManager**：负责处理锁屏的所有操作，但该类也会负责显示解锁 View(PIN、Pattern、Fingerprint)。其中 doKeyguardLocked 方法负责进行锁屏，而这些操作最终依赖 KeyguardBouncer。该类主要被 KeyguardViewMediator 调用。
- **KeyguardUpdateMonitor**：负责监控各类事件，电量指纹等事件外，该类还整合 Keyguard-ViewMediator 传来的事件整合为一个事件全集。如：电量、壁纸、指纹解锁等等所有跟锁屏有关的事件均可以在此找到。所有希望监听事件的都需要需要通过 registerCallback 方法进行注册。
- **KeyguardBouncer**：负责具体执行安全锁屏的操作，包括锁屏 UI 的加载，解锁等。
- **KeyguardHostView**：为锁屏界面的视图类。布局文件是：R.id.keyguard_host_view。

7.2 Power 键灭屏锁屏

通常我们点击 Power 键会灭屏，灭屏时就会加载锁屏，以便用户能在下次亮屏时第一时间看到锁屏，我们就来看看点击 Power 键灭屏锁屏的流程，分为两个流程，一是 framework 层的处理，二是 SystemUI 的处理。先看一，其流程如下图所示：

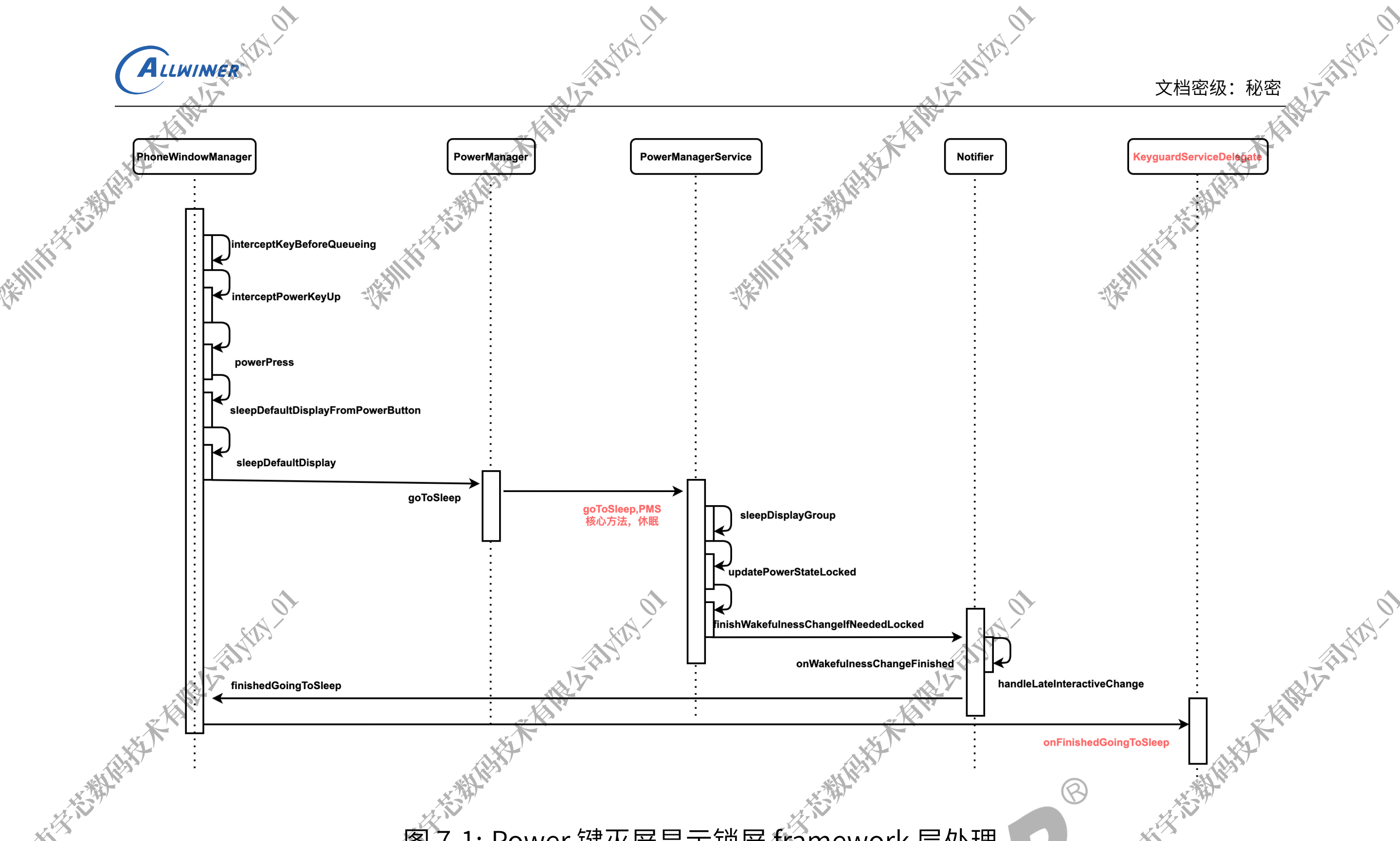

图 7-1: Power 键灭屏显示锁屏 framework 层处理

紧接着，framework 层与 SystemUI 锁屏的交互是通过 IKeyguardService.aidl 来实现，IKeyguardService.aidl 源码如下：

```
oneway interface IKeyguardService {
    //省略...
    void setOccluded(boolean isOccluded, boolean animate);

    void addStateMonitorCallback(IKeyguardStateCallback callback);
    void verifyUnlock(IKeyguardExitCallback callback);
    void dismiss(IKeyguardDismissCallback callback, CharSequence message);
    void onDreamingStarted();
    void onDreamingStopped();

    void onStartedGoingToSleep(int pmSleepReason);

    void onFinishedGoingToSleep(int pmSleepReason, boolean cameraGestureTriggered);

    void onStartedWakingUp(int pmWakeReason,  boolean cameraGestureTriggered);

    void onFinishedWakingUp();

    void onScreenTurningOn(IKeyguardDrawnCallback callback);

    void onScreenTurnedOn();

    void onScreenTurningOff();

    void onScreenTurnedOff();

    @UnsupportedAppUsage
    void setKeyguardEnabled(boolean enabled);
    void onSystemReady();
```

```
    @UnsupportedAppUsage
    void doKeyguardTimeout(in Bundle options);
    void setSwitchingUser(boolean switching);
    void setCurrentUser(int userId);
    void onBootCompleted();

    void startKeyguardExitAnimation(long startTime, long fadeoutDuration);

    void onShortPowerPressedGoHome();

    void dismissKeyguardToLaunch(in Intent intentToLaunch);

    void onSystemKeyPressed(int keycode);

    void showDismissibleKeyguard();
    //省略...
}
```

IKeyguardService 的 Binder 服务端在 KeyguardService.java 里面，它是一个真正的 service，在上面分析最后的 KeyguardServiceDelegate.java 类里面启动的，源码如下：

```
    public void bindService(Context context) {
        Intent intent = new Intent();
        final Resources resources = context.getApplicationContext().getResources();

        final ComponentName keyguardComponent = ComponentName.unflattenFromString(
                resources.getString(com.android.internal.R.string.config_keyguardComponent));
        intent.addFlags(Intent.FLAG_DEBUG_TRIAGED_MISSING);
        intent.setComponent(keyguardComponent);
        // 启动 KeyguardService
        if (!context.bindServiceAsUser(intent, mKeyguardConnection,
                Context.BIND_AUTO_CREATE, mHandler, UserHandle.SYSTEM)) {
            mKeyguardState.showing = false;
            mKeyguardState.secure = false;
            synchronized (mKeyguardState) {
                mKeyguardState.deviceHasKeyguard = false;
            }
        } else {
            if (DEBUG) Log.v(TAG, "*** Keyguard started");
        }

        final DreamManagerInternal dreamManager =
                LocalServices.getService(DreamManagerInternal.class);

        dreamManager.registerDreamManagerStateListener(mDreamManagerStateListener);
    }

    private final ServiceConnection mKeyguardConnection = new ServiceConnection() {

        //启动 KeyguardService 服务成功
        @Override
        public void onServiceConnected(ComponentName name, IBinder service) {
            mKeyguardService = new KeyguardServiceWrapper(mContext,
                    IKeyguardService.Stub.asInterface(service), mCallback);
            if (mKeyguardState.systemIsReady) {
                //执行 KeyguardService 相应的方法
                mKeyguardService.onSystemReady();
                if (mKeyguardState.currentUser != UserHandle.USER_NULL) {
                    mKeyguardService.setCurrentUser(mKeyguardState.currentUser);
```

```
            }
            if (mKeyguardState.interactiveState == INTERACTIVE_STATE_AWAKE
                    || mKeyguardState.interactiveState == INTERACTIVE_STATE_WAKING) {
                mKeyguardService.onStartedWakingUp(PowerManager.WAKE_REASON_UNKNOWN,
                        false /* cameraGestureTriggered */);
            }
            if (mKeyguardState.interactiveState == INTERACTIVE_STATE_AWAKE) {
                mKeyguardService.onFinishedWakingUp();
            }
            if (mKeyguardState.screenState == SCREEN_STATE_ON
                    || mKeyguardState.screenState == SCREEN_STATE_TURNING_ON) {
                mKeyguardService.onScreenTurningOn(
                        new KeyguardShowDelegate(mDrawnListenerWhenConnect));
            }
            if (mKeyguardState.screenState == SCREEN_STATE_ON) {
                mKeyguardService.onScreenTurnedOn();
            }
            mDrawnListenerWhenConnect = null;
        }
        if (mKeyguardState.bootCompleted) {
            mKeyguardService.onBootCompleted();
        }
        if (mKeyguardState.occluded) {
            mKeyguardService.setOccluded(mKeyguardState.occluded, false /* animate */);
        }
        if (!mKeyguardState.enabled) {
            mKeyguardService.setKeyguardEnabled(mKeyguardState.enabled);
        }
    }
    //省略...
};
```

步骤二，分析 SystemUI 的过程，如下图所示：

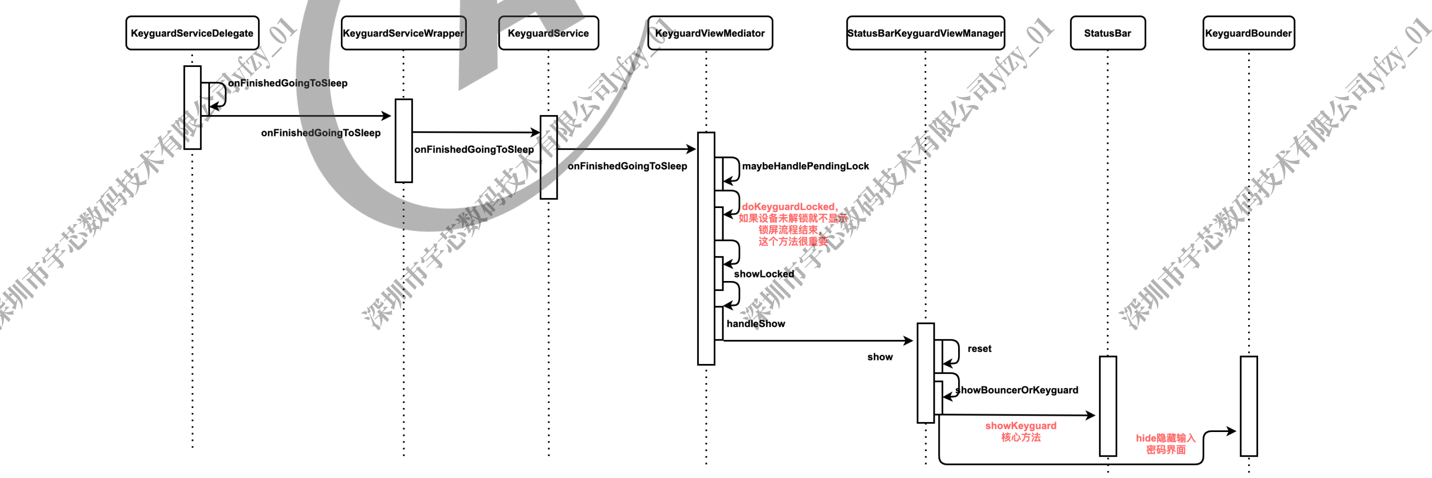

图 7-2: Power 键灭屏后显示锁屏 systemui 层处理

7.3 客制化配置

1. 设置锁屏与否：
   overlay: frameworks/base/core/res/res/values/config.xml 中的 "config_disableLockscreenByDefault" 属性

版权所有 © 珠海全志科技股份有限公司。保留一切权利

著作权声明

版权所有 ©2024 珠海全志科技股份有限公司。保留一切权利。

本文档及内容受著作权法保护，其著作权由珠海全志科技股份有限公司（“全志”）拥有并保留一切权利。

本文档是全志的原创作品和版权财产，未经全志书面许可，任何单位和个人不得擅自摘抄、复制、修改、发表或传播本文档内容的部分或全部，且不得以任何形式传播。

商标声明

ALLWINNER、ALLWINNER 全志科技、全志科技、ALLWINNER®（不完全列举）均为珠海全志科技股份有限公司的商标或者注册商标。在本文档描述的产品中出现的其它商标，产品名称，和服务名称，均由其各自所有人拥有。

免责声明

您购买的产品、服务或特性应受您与珠海全志科技股份有限公司（“全志”）之间签署的商业合同和条款的约束。本文档中描述的全部或部分产品、服务或特性可能不在您所购买或使用的范围内。使用前请认真阅读合同条款和相关说明，并严格遵循本文档的使用说明。您将自行承担任何不当使用行为（包括但不限于如超压，超频，超温使用）造成的不利后果，全志概不负责。

本文档作为使用指导仅供参考。由于产品版本升级或其他原因，本文档内容有可能修改，如有变更，恕不另行通知。全志尽全力在本文档中提供准确的信息，但并不确保内容完全没有错误，因使用本文档而发生损害（包括但不限于间接的、偶然的、特殊的损失）或发生侵犯第三方权利事件，全志概不负责。本文档中的所有陈述、信息和建议并不构成任何明示或暗示的保证或承诺。

本文档未以明示或暗示或其他方式授予全志的任何专利或知识产权。在您实施方案或使用产品的过程中，可能需要获得第三方的权利许可。请您自行向第三方权利人获取相关的许可。全志不承担也不代为支付任何关于获取第三方许可的许可费或版税（专利税）。全志不对您所使用的第三方许可技术做出任何保证、赔偿或承担其他义务。