Android原生SystenUI启动流程

1.SystemSever中的相关启动

开机时Java层会最先执行SystenSever.java中的程序，SystenSever中有startOtherServices()方法。startOtherServices()中会另起线程来执行mActivityManagerService.systemReady()方法，systemReady()中调用了startSystemUi()。

```
public final class SystemServer {
    ...;
    private void startOtherServices() {
        ...;
        mActivityManagerService.systemReady(() -> {
            ...;
            try {
                startSystemUi(context, windowManagerF);
            }
        },BOOT_TIMINGS_TRACE_LOG);
    }
}
private static void startSystemUi(Context context
                                  ,WindowManagerService windowManager) {
    Intent intent = new Intent();
    intent.setComponent(new ComponentName("com.android.systemui",

 "com.android.systemui.SystemUIService"));
    intent.addFlags(Intent.FLAG_DEBUG_TRIAGED_MISSING);
    //启动SystemUIService
    context.startServiceAsUser(intent, UserHandle.SYSTEM);
    //调用PhoneWindowManager的onSystemUiStarted()方法
    windowManager.onSystemUiStarted();
}
```

这里会有两条线：

- **第一个条通过intent启动SystemUIService**
- **第二条是调用WindowManagerService的onSystemUiStarted方法**

下面先分析第一条线（第2、3部分），再分析第二条线（第4部分）。

2.启动SystemUIService

2.1.SystemUIService.java

位于frameworks/base /packages/SystemUI/src/com/android/systemui/目录下

SystemUIService.java主要有两个内容：

- onCreate()方法中调用了SystemUIApplication的startServicesIfNeeded()方法来启动SystemUI的相关服务，SystemUIApplication是SystemUI这个APP的主要类。
- 重写了Service的dumpServices()方法

2.2.SystemUIApplication中的startServicesIfNeeded()方法

- 首先获得所有需要启动的服务名称，并将这些服务的名称保存在数组services中。
- 通过名称获取这些服务对象，并把它们转为SystemUI类型的，保存在数组mServices中。
- 依次调用mServices[i].start();来启动这些服务。

```
public void startServicesIfNeeded() {
    String[] names = getResources()
        //获取config_systemUIServiceComponents字符串数组的值，是在xml文件中写死的
        .getStringArray(R.array.config_systemUIServiceComponents);
    startServicesIfNeeded("StartServices", names);
}
private void startServicesIfNeeded(String metricsPrefix, String[] services) {
    mServices = new SystemUI[services.length];
    final int N = services.length;
    for (int i = 0; i < N; i++) {
        ...;
        String clsName = services[i];
        Class cls;
        //通过反射获取服务名称对应的class
        cls = Class.forName(clsName);
        Object o = cls.newInstance();
        if (o instanceof SystemUI.Injector) {
            //再通过class获取相应的服务类的实例对象
            o = ((SystemUI.Injector) o).apply(this);
        }
        mServices[i] = (SystemUI) o;
        //设置该实例对象的属性
        mServices[i].mContext = this;
        mServices[i].mComponents = mComponents;
        ...;
        //调用该对象的.start()方法
        mServices[i].start();
    }
    mServicesStarted = true;
}
```

这里可以看出：

- **需要启动的服务都继承自SystemUI**
- **这些服务的名称保存在数组config_systemUIServiceComponents中**

config_systemUIServiceComponents包含的内容：

```
frameworks/base/packages/SystemUI/res/values/config.xml
    <string-array name="config_systemUIServiceComponents" translatable="false">
        <item>com.android.systemui.util.NotificationChannels</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.statusbar.phone.StatusBar</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.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.AuthController</item>
        <item>com.android.systemui.SliceBroadcastRelayHandler</item>

    <item>com.android.systemui.statusbar.notification.InstantAppNotifier</item>
        <item>com.android.systemui.theme.ThemeOverlayController</item>
        <item>com.android.systemui.accessibility.WindowMagnification</item>
        <item>com.android.systemui.accessibility.SystemActions</item>
        <item>com.android.systemui.toast.ToastUI</item>
        <item>com.android.systemui.wmshell.WMShell</item>
    </string-array>
```

下面依次介绍这些服务对应的UI界面

3.启动SystemUIService相关服务

3.1.NotificationChannels

- 对应通知，主要用来创建并初始化各类型的通知，以便systemui组件调用。

3.2.KeyguardViewMediator

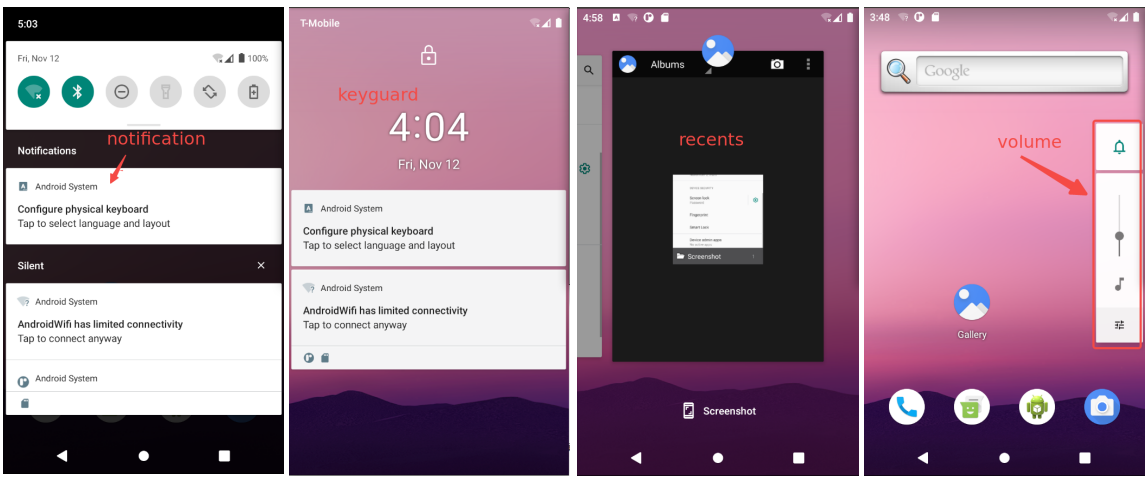
- 对应锁屏界面，初始化并设置锁屏的各种属性：锁屏广播、锁屏延迟、时间、锁屏声音、解锁声音等等。

3.3.Recents

- 对应近期任务界面

3.4.VolumeUI

- 对应音量调节对话框



3.5.StatusBar

- 对应屏幕上方的状态栏；在start()方法中初始化了屏幕生命周期的观察者、WindowManager、DreamManager、display、controller等。

3.6.StorageNotification

- 对应存储设备的通知，比如SD卡拔插。

3.7.PowerUI

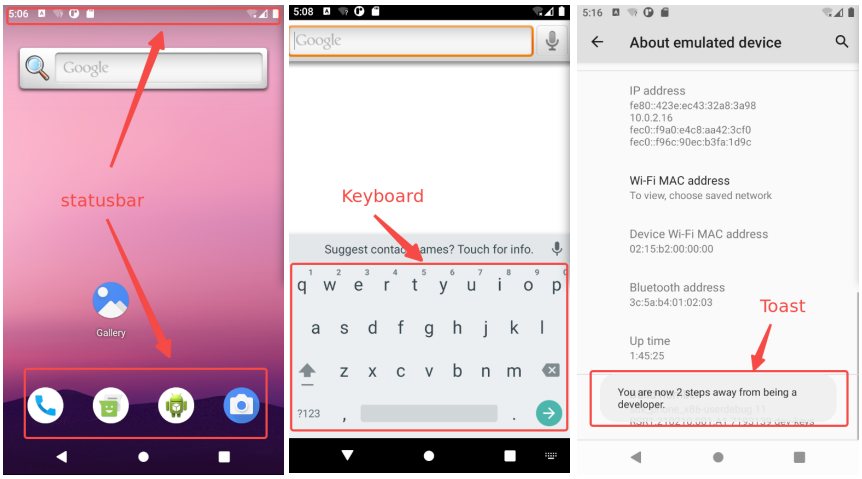
- 对应电源显示界面，比如充电、低电量等

3.8.RingtonePlayer

- 对应手机铃声，比如闹钟、来电铃声等

3.9.KeyboardUI

- 对应键盘



3.10.ShortcutKeyDispatcher

- 对应快捷方式

3.11.@string/config_systemUIVendorServiceComponent

- 自定义systemUI服务，会链接到com.android.systemui.VendorServices，开发者可以在这里定义自己的SystemUI相关服务

3.12.GarbageMonitor$Service

- 负责SystemUI的维护检查和垃圾清理

3.13.LatencyTester

- 测试类

3.14.ScreenDecorations

- 对应屏幕装饰，比如全面屏适配刘海

3.15.InstantAppNotifier

- 对应即时通知的显示

3.16.ThemeOverlayController

- 对应系统主题的替换

3.17.accessibility/

- SystemUI与framework的通信、SystemUI与WindowsManager的通信

3.18.ToastUI

- 对应toast提示

4.PhoneWindowManager的启动

在这条线中，主要内容是将SystenUI下的KeyguardService绑定到PhoneWindowManager的context中。

4.1.在SystemServer中的获取流程

- 在调用startSystemUi()方法时，会传入两个参数，一个是context，另一个是WindowManagerService类型的windowManagerF；windowManagerF被定义在startOtherService()中并被赋值为wm，wm也是定义在startOtherService()中，通过调用WindowManagerService.main()来赋值。

```
private void startOtherServices(@NonNull TimingsTraceAndSlog t) {
    WindowManagerService wm = null;
    wm = WindowManagerService.main(context, inputManager, !mFirstBoot,
mOnlyCore,
                                   new PhoneWindowManager(),
mActivityManagerService.mActivityTaskManager);
    ...;
    final WindowManagerService windowManagerF = wm;
    ...;
    startSystemUi(context, windowManagerF);
}
```

- 再来看WindowManagerService的main()方法，在WindowManagerService中，调用main()会返回WindowManagerService的单例对象sInstance，而传入的对象new PhoneWindowManager()则会被添加到LocalServices中作为WindowManager的对象，其他地方通过反射调用WindowManagerPolicy.class对应的对象是就会获得这个对象。

```
public static WindowManagerService main(...,WindowManagerPolicy policy,...) {
    sInstance = new WindowManagerService(..., policy,...);
    return sInstance;
}
private WindowManagerService(..., WindowManagerPolicy policy,...) {
    ...;
    mPolicy = policy;
    ...;
    LocalServices.addService(WindowManagerPolicy.class, mPolicy);
}
```

4.2.在WindowManagerService中的调用

- 在SystemServre中的startSystemUi()方法中，会调用windowManager的onSystemUiStarted()方法，而在WindowManagerService中，onSystemUiStarted()会调用对应WindowManager的onSystemUiStarted()方法。也就是说会调用PhoneWindowManager的onSystemUiStarted()方法。

```
//SystemServer.java
private static void startSystemUi(Context context, WindowManagerService
windowManager) {
    ...;
    windowManager.onSystemUiStarted();
}
//WindowManagerService.java
public void onSystemUiStarted() {
    mPolicy.onSystemUiStarted();
}
```

4.3.PhoneWindowManager的onSystemUiStarted()方法

- PhoneWindowManager重写了WindowManagerPolicy的onSystemUiStarted()方法，并在其中调用了bindKeyguard()方法，而bindKeyguard()方法调用了KeyguardServiceDelegate中的bindService()方法。

```
mKeyguardDelegate = new KeyguardServiceDelegate(mContext,new StateCallback() {
    //创建KeyguardServiceDelegate对象，传入的参数包括一个匿名StateCallback对象，需要重
写它的onTrustedChanged()和onShowingChanged()方法。
    @Override
    public void onTrustedChanged() {
        mWindowManagerFuncs.notifyKeyguardTrustedChanged();
    }
    @Override
    public void onShowingChanged() {
        mWindowManagerFuncs.onKeyguardShowingAndNotOccludedChanged();
    }
});
private void bindKeyguard() {
    ...;
    //在调用bindService时会传入mContext作为参数
    mKeyguardDelegate.bindService(mContext);
}
@Override
public void onSystemUiStarted() {
    bindKeyguard();
}
```

4.4.KeyguardServiceDelegate中的bindService()方法

- KeyguardServiceDelegate是用来缓存Keyguard状态的，其中包含一个内部类KeyguardState，KeyguardState中有记录各种记录keyguard状态的值，并且在每次初始化时会重置。
- KeyguardServiceDelegate的bindService()方法会绑定位于SystemUI下的KeyguardService。

```
public void bindService(Context context) {
    Intent intent = new Intent();
    //字符config_keyguardComponent的内容是
com.android.systemui/com.android.systemui.keyguard.KeyguardService,
    //也就是说这里创建了systemui下的KeyguardService对应的Component，并绑定了这个服务。
    final ComponentName keyguardComponent = ComponentName.unflattenFromString(

 resources.getString(com.android.internal.R.string.config_keyguardComponent));
    ...;
    //绑定服务，这里会初始化mKeyguardConnection这个值，mKeyguardConnection中有对
KeyguardServiceWrapper的调用
    if (!context.bindServiceAsUser(intent,
mKeyguardConnection,Context.BIND_AUTO_CREATE, mHandler, UserHandle.SYSTEM)) {
        //如果无法绑定，需要设置mKeyguardState中对应的值
        mKeyguardState.showing = false;
        ...;
    }}
```

4.5.绑定KeyguardService的具体实现

- 在KeyguardServiceWrapper的构造方法中，要求传入一个KeyguardService类型的参数，并把它赋值给mService
- 在每个生命周期，都会调用mService的相应生命周期的方法。

```
//KeyguardServiceDelegate.java
private final ServiceConnection mKeyguardConnection = new ServiceConnection() {
    @Override
    public void onServiceConnected(ComponentName name, IBinder service) {
        //获得一个KeyguardServiceWrapper的对象
        mKeyguardService = new
KeyguardServiceWrapper(mContext,IKeyguardService.Stub.asInterface(service),
mCallback);
        if (mKeyguardState.systemIsReady) {
            //在各个生命状态都会调用这个对象的对应方法，让它们两个的生命周期保持一致
            mKeyguardService.onSystemReady();
            if (mKeyguardState.currentUser != UserHandle.USER_NULL) {
                mKeyguardService.setCurrentUser(mKeyguardState.currentUser);
            }
            ...;}
        ...;}};
//KeyguardServiceWrapper.java
public KeyguardServiceWrapper(..., IKeyguardService service,...) {
    //构造方法中会将传入的KeyguardService的对象
    mService = service;
}
...;
@Override
public void onFinishedWakingUp() {
    //调用KeyguardService对象的对应方法
    mService.onFinishedWakingUp();
}
...;
```

4.6.KeyguardService的探究

- 在KeyguardService的绑定方法中，不同的生命周期都会调用KeyguardViewMediator的相应方法。至此完成对Keyguard的绑定，与第一条线合并。

```
@Override
public void addStateMonitorCallback(IKeyguardStateCallback callback) {
    mKeyguardViewMediator.addStateMonitorCallback(callback);
}
@Override
public void verifyUnlock(IKeyguardExitCallback callback) {
    ...;
    mKeyguardViewMediator.verifyUnlock(callback);
}
```