关于Android 8.0 Oreo 提醒窗口行为变更调研结果

提醒窗口变更　https://developer.android.com/about/versions/oreo/android-8.0-changes.html#cwt

原文内容：

行为变更专门应用于针对 O 平台或更高平台版本的应用。针对 Android 8.0 或更高平台版本进行编译，或将 targetSdkVersion 设为 Android 8.0 或更高版本的应用开发者必须修改其应用以正确支持这些行为（如果适用）。

> 使用 `SYSTEM_ALERT_WINDOW` 权限的应用无法再使用以下窗口类型来其他应用和系统窗口上方显示提醒窗口： - TYPE*PHONE* - *TYPE*PRIORITY*PHONE* - *TYPE*SYSTEM*ALERT* - *TYPE*SYSTEM*OVERLAY* - *TYPE*SYSTEM_ERROR
>
> 相反，应用必须使用名为 `TYPE_APPLICATION_OVERLAY` 的新窗口类型。
>
> 使用 TYPE*APPLICATION*OVERLAY 窗口类型显示应用的提醒窗口时，请记住新窗口类型的以下特性： - 应用的提醒窗口始终显示在状态栏和输入法等关键系统窗口的下面。 - 系统可以移动使用 TYPE*APPLICATION*OVERLAY 窗口类型的窗口或调整其大小，以改善屏幕显示效果。 - 通过打开通知栏，用户可以访问设置来阻止应用显示使用 TYPE*APPLICATION*OVERLAY 窗口类型显示的提醒窗口。

结论: app的compileSdkVersion 或者　targetSdkVersion　>= 26(Anroid oreo)后，通过 `SYSTEM_ALERT_WINDOW` 权限添加的一些系统类型的提醒窗口将无法显示再其他应用或者系统的窗口上方，简而言之就是之前想要的提醒窗口效果废了。且安卓方想让你改成使用 `TYPE_APPLICATION_OVERLAY` 这个新窗口类型，这样好让Android Oreo系统控制提醒窗口和按照系统意愿调整窗口大小和效果。

问题：那如果我的compileSdkVersion 或者　targetSdkVersion　＜26的话，还能使用只之前的系统类型的提醒窗口么？是不是就可以显示在其他应用或者系统窗口的上面了呢？来,RTFSC。

RTFSC 之 Android Oreo Source Code

类说明(可以跳到该段结论，只需知道结论就好)

提醒窗口其实就是通过 `WindowManager` 展示的一些View,我们给WindowManager type类型和属性，其中Type类型的值决定了View在窗口显示在那一层，原则Type的值越大，会显示在值小的上面，换句话说值越大越在上面，当然系统会对type做检查，如果不符合规定的值及范围就会出现异常，当然这只是最简单的策略，其中还有其他很多条件。

相关类介绍 - `WindowManagerPolicy`：

> 该接口提供了WindowManager的所有UI特定行为。它的实例由WindowManager在启动的时候创建，并允许自定义窗口分层、特殊窗口类型、key分发和布局。
>
> 因为这提供了与系统WindowManager的深层交互，所以接口中的特定方法可以通过被严格限制他们可以做什么的各种Context来调用。这些方法在代码加入了后缀；如果方法上没有后缀，那只可以被Window Manager线程或者其他调用Window Manager线程调用。
>
> 现有的后缀： - Ti : input thread调用，这是一个收集input event并讲他们分发到适当的Window的thread 。它可能被阻塞等待事件被处理，所以input stream会被序列化。 - Tq : low-level的input queue thread调用。 这个线程从原始输入设备读取事件并将他们放入由Ti读取的全局队列中。这个线程除了驱动程序不应该被阻塞很长时间。 - Lw : 主WindowManager锁的持有者调用。由于WindowManager是一个非常低级别的系统服务，所以你可以使用锁持有者调用其他几个系统服务，如,package manager，power manager是可以的，但是activity manager或者其他很多系统服务是不可以的。 - Li : input thread锁持有者调用，这个锁可以通过WindowManager在持有窗口锁时获得，所以它比Lw严格。

- `PhoneWindowManager`

> 用于Android phone UI的 `WindowManagerPolicy` 的实现。它包含一个新的方法后缀'Lp'，用于PhoneWindowManager的内部锁。该Manager用来保护一些内部状态，并且可以获得其他 `Lw` 和 `Li` 锁，所以当持有时受到两者的限制。

总结: WindowMangerPolicy接口设定了很多策略，其中通过方法加入后缀来区分限制不同系统调用，而PhoneWindowManager是WindowManagerPolicy策略的具体实现。其中Lw后缀的方法只能WindowManager调用。

实现细节

WindowManagerPolicy.java

```
/**
 * @return whether {@param win} can be hidden by Keyguard
 */
public boolean canBeHiddenByKeyguardLw(WindowState win);
```

> 该方法定义是否被系统键盘锁隐藏,且方法后缀Lw，只能被WindowManager调用

PhoneWindowManager.java

```
@Override
public boolean canBeHiddenByKeyguardLw(WindowState win) {
    switch (win.getAttrs().type) {
        case TYPE_STATUS_BAR:
        case TYPE_NAVIGATION_BAR:
        case TYPE_WALLPAPER:
        case TYPE_DREAM:
            return false;
        default:
            // Hide only windows below the keyguard host window.
            return getWindowLayerLw(win) < getWindowLayerFromTypeLw(TYPE_STATUS_BAR);
    }
}
```

> 除了系统type外，type只要在系统锁之下就会被隐藏，系统锁的type是TYPE*STATUS*BAR。

WindowManagerPolicy.java

```
/**
 * Returns the layer assignment for the window state. Allows you to control how different
 * kinds of windows are ordered on-screen.
 *
 * @param win The window state
 * @return int An arbitrary integer used to order windows, with lower numbers below higher ones.
 */
default int getWindowLayerLw(WindowState win) {
    return getWindowLayerFromTypeLw(win.getBaseType(), win.canAddInternalSystemWindow());
}
```

```
/**
     * Returns the layer assignment for the window type. Allows you to control how different
     * kinds of windows are ordered on-screen.
     *
     * @param type The type of window being assigned.
     * @param canAddInternalSystemWindow If the owner window associated with the type we are
     *        evaluating can add internal system windows. I.e they have
     *        {@link android.Manifest.permission#INTERNAL_SYSTEM_WINDOW}. If true, alert window
     *        types {@link android.view.WindowManager.LayoutParams#isSystemAlertWindowType(int)}
     *        can be assigned layers greater than the layer for
     *        {@link android.view.WindowManager.LayoutParams#TYPE_APPLICATION_OVERLAY} Else, their
     *        layers would be lesser.
     * @return int An arbitrary integer used to order windows, with lower numbers below higher ones.
     */
    default int getWindowLayerFromTypeLw(int type, boolean canAddInternalSystemWindow) {
        if (type >= FIRST_APPLICATION_WINDOW && type <= LAST_APPLICATION_WINDOW) {
            return APPLICATION_LAYER;
```

```
        }

        switch (type) {
            case TYPE_WALLPAPER:
                // wallpaper is at the bottom, though the window manager may move it.
                return  1;
            case TYPE_PRESENTATION:
            case TYPE_PRIVATE_PRESENTATION:
                return  APPLICATION_LAYER;
            case TYPE_DOCK_DIVIDER:
                return  APPLICATION_LAYER;
            case TYPE_QS_DIALOG:
                return  APPLICATION_LAYER;
            case TYPE_PHONE:
                return  3;
            case TYPE_SEARCH_BAR:
            case TYPE_VOICE_INTERACTION_STARTING:
                return  4;
            case TYPE_VOICE_INTERACTION:
                // voice interaction layer is almost immediately above apps.
                return  5;
            case TYPE_INPUT_CONSUMER:
                return  6;
            case TYPE_SYSTEM_DIALOG:
                return  7;
            case TYPE_TOAST:
                // toasts and the plugged-in battery thing
                return  8;
            case TYPE_PRIORITY_PHONE:
                // SIM errors and unlock.  Not sure if this really should be in a high layer.
                return  9;
            case TYPE_SYSTEM_ALERT:
                // like the ANR / app crashed dialogs
                return  canAddInternalSystemWindow ? 11 : 10;
            case TYPE_APPLICATION_OVERLAY:
                return  12;
            case TYPE_DREAM:
                // used for Dreams (screensavers with TYPE_DREAM windows)
                return  13;
            case TYPE_INPUT_METHOD:
                // on-screen keyboards and other such input method user interfaces go here.
                return  14;
            case TYPE_INPUT_METHOD_DIALOG:
                // on-screen keyboards and other such input method user interfaces go here.
                return  15;
            case TYPE_STATUS_BAR_SUB_PANEL:
                return  17;
            case TYPE_STATUS_BAR:
                return  18;
            case TYPE_STATUS_BAR_PANEL:
                return  19;
            case TYPE_KEYGUARD_DIALOG:
                return  20;
            case TYPE_VOLUME_OVERLAY:
                // the on-screen volume indicator and controller shown when the user
                // changes the device volume
                return  21;
            case TYPE_SYSTEM_OVERLAY:
                // the on-screen volume indicator and controller shown when the user
                // changes the device volume
                return  canAddInternalSystemWindow ? 22 : 11;
            case TYPE_NAVIGATION_BAR:
                // the navigation bar, if available, shows atop most things
                return  23;
```

```
            case TYPE_NAVIGATION_BAR_PANEL:
                // some panels (e.g. search) need to show on top of the navigation bar
                return  24;
            case TYPE_SCREENSHOT:
                // screenshot selection layer shouldn't go above system error, but it should cover
                // navigation bars at the very least.
                return  25;
            case TYPE_SYSTEM_ERROR:
                // system-level error dialogs
                return  canAddInternalSystemWindow ? 26 : 10;
            case TYPE_MAGNIFICATION_OVERLAY:
                // used to highlight the magnified portion of a display
                return  27;
            case TYPE_DISPLAY_OVERLAY:
                // used to simulate secondary display devices
                return  28;
            case TYPE_DRAG:
                // the drag layer: input for drag-and-drop is associated with this window,
                // which sits above all other focusable windows
                return  29;
            case TYPE_ACCESSIBILITY_OVERLAY:
                // overlay put by accessibility services to intercept user interaction
                return  30;
            case TYPE_SECURE_SYSTEM_OVERLAY:
                return  31;
            case TYPE_BOOT_PROGRESS:
                return  32;
            case TYPE_POINTER:
                // the (mouse) pointer layer
                return  33;
            default:
                Slog.e("WindowManager", "Unknown window type: " + type);
                return APPLICATION_LAYER;
        }
    }
```

其中当type类型是 `TYPE_SYSTEM_ALERT` ， `TYPE_SYSTEM_OVERLAY` ， `TYPE_SYSTEM_ERROR` 这几个类型系统会做一次限制修正，那就是如果参数中不是 `canAddInternalSystemWindow = false` 时，它们的值会被修正成10或者11，该值是小于Android Oreo推荐的类型 `TYPE_APPLICATION_OVERLAY = 12` 的，且它们都比 `TYPE_STATUS_BAR = 18` 要小，也就是这些类型如果 `canAddInternalSystemWindow = false` 时，在系统Android Oreo上被系统锁屏隐藏住。

对比Android 7.0及以下是没有 `canAddInternalSystemWindow` 的检测的，也就是 `TYPE_SYSTEM_ERROR` 是可以盖在系统锁屏之上的。

问题: canAddInternalSystemWindow如何成为true?

根据上面 `getWindowLayerFromTypeLw` 方法注释可知，canAddInternalSystemWindow用上述几个类型的Window必须具有 `android.Manifest.permission.INTERNAL_SYSTEM_WINDOW` 权限，才可以。

```
<!-- @SystemApi Allows an application to open windows that are for use by parts
     of the system user interface.
     <p>Not for use by third-party applications.
     @hide
-->
<permission android:name="android.permission.INTERNAL_SYSTEM_WINDOW"
    android:protectionLevel="signature" />
```

从上面该权限的定义可以看到， `protectionLevel = signature`

> "signature" : 只有在请求应用程序与已声明权限的应用程序具有相同证书时才允许系统授予此权限。如果证书匹配，系统将自动授予权限，而不通知用户或请求用户的明确批准。 该权限的定义是在Android系统，也就是只有与系统签名相同，才会授权该权限，此

> 时`canAddInternalSystemWindow = true`

问题：`android.permission.INTERNAL_SYSTEM_WINDOW`是在哪里检测的?

我们从WindowManager添加一个View开始trace代码：

WindowManager添加View代码过程：

- WindowManger.addView(...);
- WindowManagerGlobal.addView(...);
- ViewRootImpl.setView(...);

```
/**
* We have one child
*/
public void setView(View view, WindowManager.LayoutParams attrs, View panelParentView) {
    ...
    res = mWindowSession.addToDisplay(mWindow, mSeq, mWindowAttributes,
                            getHostVisibility(), mDisplay.getDisplayId(),
                            mAttachInfo.mContentInsets, mAttachInfo.mStableInsets,
                            mAttachInfo.mOutsets, mInputChannel);
    // 根据res判断是否正确添加到Window中，如果有问题，抛异常
    ...
 }
```

- Session.addToDisplay(...);

```
 @Override
public int addToDisplay(IWindow window, int seq, WindowManager.LayoutParams attrs,
        int viewVisibility, int displayId, Rect outContentInsets, Rect outStableInsets,
        Rect outOutsets, InputChannel outInputChannel) {
    return mService.addWindow(this, window, seq, attrs, viewVisibility, displayId,
            outContentInsets, outStableInsets, outOutsets, outInputChannel);
}
```

- WindowManagerService.java

```
public int addWindow(Session session, IWindow client, int seq,
            WindowManager.LayoutParams attrs, int viewVisibility, int displayId,
            Rect outContentInsets, Rect outStableInsets, Rect outOutsets,
            InputChannel outInputChannel) {
        ...
        int res = mPolicy.checkAddPermission(attrs, appOp);
        ...
        // 中间根据res以及其他条件，来做相应的处理，比如抛异常，没有权限等操作
        ...
        return res;
    }
```

- PhoneWindowManager.java(Policy具体实现)

```java
/** {@inheritDoc} */
@Override
public int checkAddPermission(WindowManager.LayoutParams attrs, int[] outAppOp) {
    int type = attrs.type;
    ...
    if (!((type >= FIRST_APPLICATION_WINDOW && type <= LAST_APPLICATION_WINDOW)
            || (type >= FIRST_SUB_WINDOW && type <= LAST_SUB_WINDOW)
            || (type >= FIRST_SYSTEM_WINDOW && type <= LAST_SYSTEM_WINDOW))) {
        return WindowManagerGlobal.ADD_INVALID_TYPE;
    }

    if (type < FIRST_SYSTEM_WINDOW || type > LAST_SYSTEM_WINDOW) {
        // Window manager will make sure these are okay.
        return ADD_OKAY;
    }

    if (!isSystemAlertWindowType(type)) {
        switch (type) {
            case TYPE_TOAST:
                // Only apps that target older than O SDK can add window without a token, after
                // that we require a token so apps cannot add toasts directly as the token is
                // added by the notification system.
                // Window manager does the checking for this.
                outAppOp[0] = OP_TOAST_WINDOW;
                return ADD_OKAY;
            case TYPE_DREAM:
            case TYPE_INPUT_METHOD:
            case TYPE_WALLPAPER:
            case TYPE_PRESENTATION:
            case TYPE_PRIVATE_PRESENTATION:
            case TYPE_VOICE_INTERACTION:
            case TYPE_ACCESSIBILITY_OVERLAY:
            case TYPE_QS_DIALOG:
                // The window manager will check these.
                return ADD_OKAY;
        }
        return mContext.checkCallingOrSelfPermission(INTERNAL_SYSTEM_WINDOW)
                == PERMISSION_GRANTED ? ADD_OKAY : ADD_PERMISSION_DENIED;
    }
    ...
}
```

在该方法的最后会判断 `checkCallingOrSelfPermission(INTERNAL_SYSTEM_WINDOW) == PERMISSION_GRANTED ? ADD_OKAY : ADD_PERMISSION_DENIED` 也就是是否有INTERNAL*SYSTEM*WINDOW权限。

- ContextImpl.java 权限检测实现

```java
@Override
public int checkCallingOrSelfPermission(String permission) {
    if (permission == null) {
        throw new IllegalArgumentException("permission is null");
    }
    return checkPermission(permission, Binder.getCallingPid(),
            Binder.getCallingUid());
}
```

```
Override
public int checkPermission(String permission, int pid, int uid) {
    if (permission == null) {
        throw new IllegalArgumentException("permission is null");
    }

    final IActivityManager am = ActivityManager.getService();
    ...
    try {
        return am.checkPermission(permission, pid, uid);
    } catch (RemoteException e) {
        throw e.rethrowFromSystemServer();
    }
}
```

结论，目前分析整个权限以及Window Type检测过程来看，Android Oreo中对于滥用系统类型做了强硬限制。管理更加严格或者说开始进行规范提醒窗口的使用。