Android面试题之Window和WindowManager，包括Window的添加、删除、更新Dialog源码、Toast源码等。

> 本文是我一点点归纳总结的干货，但是难免有疏忽和遗漏，希望不吝赐教。
> 转载请注明链接：https://blog.csdn.net/feather_wch/article/details/81437056

有帮助的话请点个赞！万分感谢！

Android面试题-Window和WindowManager(26题)

版本：2018/8/5-1(23:0)

1、Window是什么?

> 1. 表示一个窗口的概念，是所有 `View` 的直接管理者，任何视图都通过 `Window` 呈现(点击事件由Window->DecorView->View; Activity的 `setContentView` 底层通过 `Window` 完成)
> 2. `Window` 是一个抽象类，具体实现是 `PhoneWindow`
> 3. 创建 `Window` 需要通过 `WindowManager` 创建
> 4. `WindowManager` 是外界访问 `Window` 的入口
> 5. `Window` 具体实现位于 `WindowManagerService` 中
> 6. `WindowManager` 和 `WindowManagerService` 的交互是通过 `IPC` 完成

2、如何通过 `WindowManager` 添加 `Window` (代码实现)?

```
//1. 控件
Button button = new Button(this);
button.setText("Window Button");
//2. 布局参数
WindowManager.LayoutParams layoutParams = new WindowManager.LayoutParams(WindowManager.LayoutParams.W
        WindowManager.LayoutParams.WRAP_CONTENT, 0, 0, PixelFormat.TRANSPARENT);
layoutParams.flags = WindowManager.LayoutParams.FLAG_NOT_TOUCH_MODAL
        | WindowManager.LayoutParams.FLAG_NOT_FOCUSABLE
        | WindowManager.LayoutParams.FLAG_SHOW_WHEN_LOCKED;
layoutParams.gravity = Gravity.LEFT | Gravity.TOP;
layoutParams.x = 100;
layoutParams.y = 300;
// 必须要有type不然会异常: the specified window type 0 is not valid
layoutParams.type = WindowManager.LayoutParams.TYPE_SYSTEM_ERROR;
//3. 获取WindowManager并添加控件到Window中
WindowManager windowManager = getWindowManager();
windowManager.addView(button, layoutParams);
```

1. 注意一定要指定布局类型 `layoutParams.type`
2. 需要动态申请 `Draw over other apps` 权限：http://blog.csdn.net/feather_wch/article/details/79185045

3、LayoutParams的 `flags` 属性

Flags	解释
FLAG_NOT_FOCUSABLE	表示 `Window` 不需要焦点，会同时启用 `FLAG_NOT_TOUCH_MODAL`， 最终事件会直接传递到下层具有焦点的 `Window`
FLAG_NOT_TOUCH_MODEL	将当前 `Window` 区域以外的单击事件传递给底层 `Window`， 当前区域内的单击事件自己处理(如果不开启， 其他 `Window` 会无法收到单击事件)
FLAG_SHOW_WHEN_LOCKED	可以让 `Window` 显示在锁屏的界面上

4、LayoutParams的 `type` 属性

Window类型	含义	Window层级	Type参数
应用Window	对应着一个Activity	1~99(视图最下层)	
子Window	不能单独存在， 需要附属在特定的父 `Window` 之中 (如Dialog就是子Window)	1000~1999	
系统Window	需要声明权限才能创建的 `Window`， 比如 `Toast` 和 系统状态栏	2000~2999(视图最上层)	TYPE_SYSTEM_OVERLAY / TYPE_SYSTEM_ERROR

需要在AndroidManifest中声明权限： `SYSTEM_ALERT_WINDOW`

5、WindowManager的三个主要功能：添加、更新、删除View

```
public interface ViewManager
{
    public void addView(View view, ViewGroup.LayoutParams params); //添加View
    public void updateViewLayout(View view, ViewGroup.LayoutParams params); //更新View
    public void removeView(View view); //删除View
}
```

6、通过WindowManager实现拖动View的效果

1. 给 `View` 设置 `onTouchListener` 监听器
2. 在 `onTouch` 方法中根据当前坐标，来更新View `updateViewLayout`

```
mButton.setOnTouchListener(new View.OnTouchListener() {
    @Override
    public boolean onTouch(View v, MotionEvent event) {
        int rawX = (int) event.getRawX();
        int rawY = (int) event.getRawY();
        switch (event.getAction()){
            case MotionEvent.ACTION_MOVE:
                mLayoutParams.x = rawX;
                mLayoutParams.y = rawY;
                mWindowManager.updateViewLayout(mButton, mLayoutParams);
                break;
        }
        return false;
    }
});
```

Window的内部机制

7、Window概念解析

> 1. Window和View通过 `ViewRootImpl` 建立联系
> 2. `Window` 并不是实际存在的，而是以 `View` 的形式存在
> 3. `WindowManager` 的三个接口方法也是针对 `View` 的
> 4. 实际使用中无法直接访问Window，必须通过 `WindowManager`
> 5. View是视图的呈现方式，但是不能单独存在，必须依附在 `Window` 这个抽象的概念上
> 6. `WMS` 把所有的用户消息发给View/ViewGroup，但是在View/ViewGroup处理消息的过程中，有一些操作是公共的, Window把这些公共行为抽象出来, 这就是Window。

8、WindowSession的创建过程

```
//ViewRootImpl.java--通过WindowManager去获取WindowSession
public ViewRootImpl(Context context, Display display) {
    mContext = context;
    mWindowSession = WindowManagerGlobal.getWindowSession();
    ...
}
//WindowManagerGlobal.java--IPC过程
public static IWindowSession getWindowSession() {
    ...
    // 1.获取WindowManagerService
    IWindowManager windowManager = getWindowManagerService();
    // 2. 创建WindowSession(Session是具体实现)
    sWindowSession = windowManager.openSession(...);
}
```

> 1. 在WindowManager的addView中会创建ViewRootImpl，内部会通过WMS去获取WindowSession
> 2. WindowSession的类型是IWindowSession，本身是Binder对象，真正实现类是Session

9、WindowSession的作用?

> 1. 表示一个Active Client Session
> 2. 每个进程一般都有一个Session对象
> 3. 用于WindowManager交互

10、Token的使用场景?

1. `Popupwindow` 的showAtLocation第一个参数需要传入View，这个View就是用来获取Token的。
2. Android 5.0新增空间 `SnackBar` 同理也需要一个View来获取Token

11、Token是什么?

1. 类型为IBinder，是一个Binder对象。
2. 主要分两种Token：
    1. 指向Window的token: 主要是实现WmS和应用所在进程通信。
    2. 指向ActivityRecord的token: 主要是实现WmS和AmS通信的。

12、Activity中的Token

1. ActivityRecord是AmS中用来保存一个Activity信息的辅助类。
2. AMS中需要根据Token去找到对应的ActivityRecord。

13、Window的addView源码分析

```
//WindowManagerGlobal.java
public void addView(View view, ViewGroup.LayoutParams params, Display display, Window parentWindow) {
    //1. 检查参数是否合法
    if (view == null) { throw new IllegalArgumentException("view must not be null");}
    if (display == null) { throw new IllegalArgumentException("display must not be null");}
    if (!(params instanceof WindowManager.LayoutParams)) {throw new IllegalArgumentException("Params must
    //2. 如果是子Window会调整布局参数
    final WindowManager.LayoutParams wparams = (WindowManager.LayoutParams) params;
    if (parentWindow != null) {
        parentWindow.adjustLayoutParamsForSubWindow(wparams);
    } else {
        //3. 如果没有父Window则根据该app的硬件加速设置，设置给该View
        ...
    }
    synchronized (mLock) {
        //...省略....
        //4. 创建Window所对应的ViewRootImpl
        ViewRootImpl root = new ViewRootImpl(view.getContext(), display);
        //5. 将Window对应的ViewRootImpl、View、布局参数添加到列表中
        view.setLayoutParams(wparams);
        mViews.add(view);//将Window对应的View添加到列表中
        mRoots.add(root);//将ViewRootImpl添加到列表中
        mParams.add(wparams);//将Window对应的布局参数添加到列表中
        //6. 通过ViewRootImpl完成View的绘制过程，以及Window的添加过程
        try {
            root.setView(view, wparams, panelParentView);
        } catch (RuntimeException e) {
            //...
        }
    }
}

//ViewRootImpl.java
public void setView(View view, WindowManager.LayoutParams attrs, View panelParentView) {
    ...
    // 1. 进行View绘制三大流程的接口
    requestLayout();
    // 2. 会通过`WindowSession`完成Window的添加过程(一次IPC调用)
    res = mWindowSession.addToDisplay(mWindow, ...);
    ...
}

//ViewRootImpl.java
public void requestLayout() {
    if (!mHandlingLayoutInLayoutRequest) {
        checkThread();
        mLayoutRequested = true;
        //1. 实际是View绘制的入口(测量、布局、重绘)--内部通过mChoreographer去监听下一帧的刷新信号
        scheduleTraversals();
    }
}

//Session.java
@Override
public int addToDisplay(IWindow window, ...) {
    return mService.addWindow(this, window, ...);
}

//WindowManagerService.java
public int addWindow(Session session, IWindow client, ...) {
    /**================================
     * 1. 检查工作
```

```
         *    1-权限检查和设置相关
         *    2-检查是否重复添加---Binder对象会存储在HashMap<IBinder, WindowState>中
         *==================================*/
        ...
        if (mWindowMap.containsKey(client.asBinder())) {
            ...
        }
        // 2. 返回WmS中存在的对应父窗口，若不存在则返回null
        WindowState attachedWindow = windowForClientLocked(null, attrs.token, false);
        ...
        /**==========================================
         * 3. 从WmS中寻找对应的WindowToken，并且处理合法性
         *   1-例如：对于子窗口来说，WmS中必须有对应的Token才能添加
         *   2-例如：token不为null且是应用窗口是必须要有个windowToken
         *==========================================*/
        WindowToken token = mTokenMap.get(attrs.token);
        /**
         * ...检查Token合法性...
         */
        // 4. 创建窗口
        WindowState win = new WindowState(session, client, token, attachedWindow, attrs, viewVisibility);
        /**==========================================
         * 5. 额外的操作、信息保存等
         *      1. 如果是Toast，则此窗口不能够接收input事件
         *      2. Token添加到WmS中
         *      3. 窗口信息添加到WmS中
         *      4. 将窗口(WindowState)添加到Session中
         *      5. 对于应用启动时显示的窗口，设置token
         *==========================================*/
        // 1、如果是Toast，则此窗口不能够接收input事件
        mPolicy.adjustWindowParamsLw(win.mAttrs);
        // 2、从上述代码中得出是否要添加Token，若是则添加Token添加到WmS中
        if (addToken) {
            mTokenMap.put(attrs.token, token);
            mTokenList.add(token);
        }
        // 3、窗口信息添加到WmS中
        mWindowMap.put(client.asBinder(), win);
        // 4、将窗口添加到Session中
        win.attach();
        // 5、对于应用启动时显示的窗口，设置token
        token.appWindowToken.startingWindow = win;
        ...
    }
```

1. `WindowManager` 是一个接口，真正实现类是 `WindowManagerImpl` ，并最终以代理模式交给 `WindowManagerGlobal` 实现。
2. addView: 1-创建ViewRootImpl；2-将ViewRoor、DecorView、布局参数保存到WM的内部列表中；3-ViewRoot.setView()建立ViewRoot和DecorView的联系。
3. setView：1-进行View绘制三大流程；2-会通过 `WindowSession` 完成Window的添加过程(一次IPC调用)
4. requestLayout：内部调用scheduleTraversals(), 底层通过mChoreographer去监听下一帧的刷新信号。
5. mWindowSession.addToDisplay: 执行 `WindowManangerService` 的addWindow
6. addWindow: 检查参数等设置;检查Token;将Token、Window保存到WMS中;将WindowState保存到Session中。

14、Window的remove源码与解析

```
//WindowManagerGlobal.java
public void removeView(View view, boolean immediate) {
    ...
    synchronized (mLock) {
        //1. 查询待删除的View的索引
        int index = findViewLocked(view, true);
        View curView = mRoots.get(index).getView();
        //2. 进行进一步删除
        removeViewLocked(index, immediate);
        ...
    }
}
//WindowManagerGlobal.java
private void removeViewLocked(int index, boolean immediate) {
    ViewRootImpl root = mRoots.get(index);
    View view = root.getView();
    ...
    //1. ViewRoot调用die，发送删除请求后立即返回，此时View并没有完成删除
    boolean deferred = root.die(immediate);
    if (view != null) {
        view.assignParent(null);
        //2. 将View添加到等待删除的列表中
        if (deferred) {
            mDyingViews.add(view);
        }
    }
}
//ViewRootImpl.java
boolean die(boolean immediate) {
    // 1. 同步删除时直接调用
    if (immediate && !mIsInTraversal) {
        doDie();
        return false;
    }
    // 2. 异步删除会发送MSG
    mHandler.sendEmptyMessage(MSG_DIE);
    return true;
}

//ViewRootImpl.java内部类: ViewRootHandler
final class ViewRootHandler extends Handler {
    @Override
    public void handleMessage(Message msg) {
        switch (msg.what) {
            // 1. 执行doDie()
            case MSG_DIE:
                doDie();
                break;
            ...
        }
    }
}

//ViewRootImpl.java
void doDie() {
    checkThread();
    ...
    // 1. 真正删除View
    dispatchDetachedFromWindow();
    // 2. 刷新数据，将Window对应的所有对象从列表中删除
    WindowManagerGlobal.getInstance().doRemoveView(this);
}
```

```
//ViewRootImpl.java
void dispatchDetachedFromWindow() {
    if (mView != null && mView.mAttachInfo != null) {
        mAttachInfo.mTreeObserver.dispatchOnWindowAttachedChange(false);
        /**=============================
         * 1. 回调onDetachedFromWindow: 在View被移除时调用，可以进行一些终止动画、线程之类的操作
         * 2. 回调onDetachedFromWindowInternal
         *=====================================*/
        mView.dispatchDetachedFromWindow();
    }
    // 3. 垃圾回收相关操作
    mView = null;
    mAttachInfo.mRootView = null;
    mSurface.release();
    ...
    // 4. 通过Session的remove()删除Window---会调用WMS的removeWindow方法
    mWindowSession.remove(mWindow);
    // 5. 通过Choreographer移除监听器
    unscheduleTraversals();
}
```

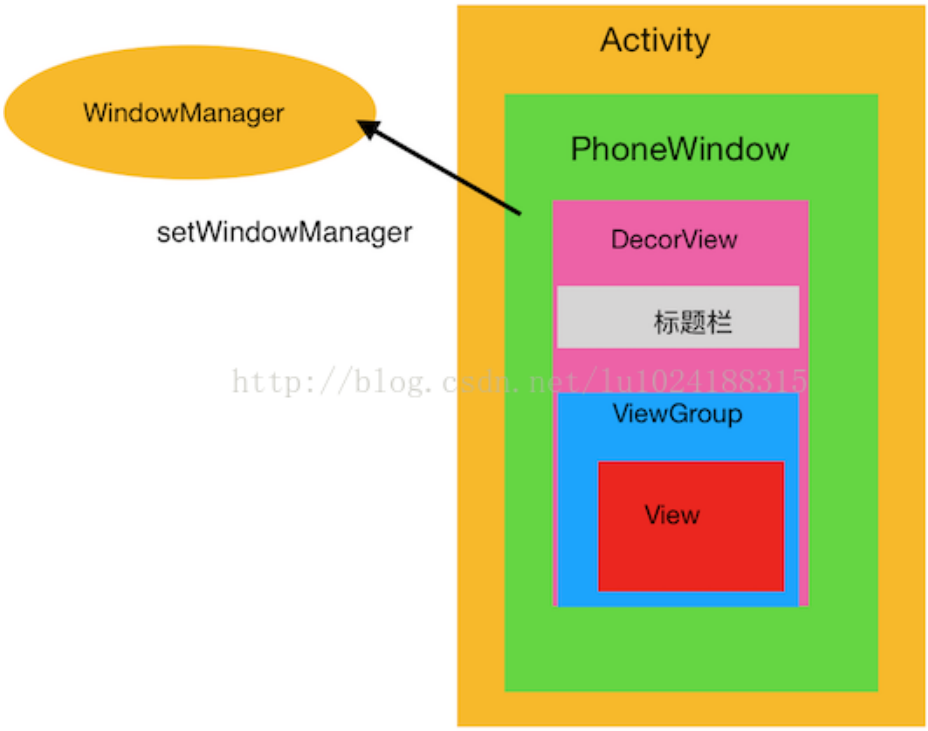
1. `WindowManager` 中提供了两种删除接口： `removeView` 异步删除、 `removeViewImmediate` 同步删除(不建议使用)
2. 调用WMGlobal的removeView
3. 调用到WMGlobal的removeViewLocked进行真正的移除
4. 执行ViewRoot的die方法(): 1-同步方法直接调用doDie 2-异步方法直接发送Message
5. doDie(): 调用 `dispatchDetachedFromWindow()` 和 `WindowManagerGlobal.getInstance().doRemoveView(this)`
6. dispatchDetachedFromWindow: 1-回调onDetachedFromeWindow；2-垃圾回收相关操作；3-通过Session的remove()在WMS中删除Window；4-通过Choreographer移除监听器

15、Window的更新过程 `updateViewLayout`

```
public void updateViewLayout(View view, ViewGroup.LayoutParams params) {
    ...
    final WindowManager.LayoutParams wparams = (WindowManager.LayoutParams)params;
    //1. 给View设置新布局
    view.setLayoutParams(wparams);
    //2. 按照index从列表中删除旧布局，并添加新布局
    synchronized (mLock) {
        int index = findViewLocked(view, true);
        ViewRootImpl root = mRoots.get(index);
        mParams.remove(index);
        mParams.add(index, wparams);
        //3. 更新ViewRootImpl中的布局(会对View进行测量、布局、重绘，)
        root.setLayoutParams(wparams, false);
    }
}
```

1. 和添加删除类似，最终调用 `WindowManagerGlobal` 的 `updateViewLayout` 方法
2. `root.setLayoutParams` 会对View进行重新布局——测量、布局、重绘
3. `root.setLayoutParams` 还会通过 `WindowSession` 更新 `Window` 的视图——最终通过 `WindowManagerService` 的 `relayoutWindow()` 实现(IPC)

Window的创建过程



16、Activity的启动过程

> 1. 最终会由 `ActivityThread` 中的 `performLauchActivity` 来完成整个启动过程
> 2. `performLauchActivity` 内部会通过 类加载器 创建Activity的实例对象
> 3. 并为Activity的实例对象调用 `attach` 方法，为其关联运行过程中所以来的上下文环境变量
> 4. `attch` 方法中，系统会创建Activity所属的 `Window对象` ，并为其设置回调接口
> 5. `Window` 对象的创建是通过 `PolicyManager` 的 `makeNewWindow` 方法实现。
> 6. `Activity` 实现了 `window` 的 `callback接口` ，因此外界状态改变时会回调Activity的方法 (onAttachedToWindow、dispatchTouchEvent等等)

17、PolicyManager是什么

> 1. 是一个策略类
> 2. Activity的 `Window` 就是通过 `PolicyManager` 的一个工厂方法创建
> 3. `PolicyManager` 实现的工厂方法全部在策略接口 `IPolicy` 中声明
> 4. `PolicyManager` 的实现类是 `Policy` 类

```
//Window的实现类就是`PhoneWindow`
public Window makeNewWindow(Context context){
  return new PhoneWindow(context);
}
```

18、Activity的视图加载的源码分析

```
// Activity.java
public void setContentView(@LayoutRes int layoutResID) {
    // 1. 交给Window(PhoneWindow)处理
    getWindow().setContentView(layoutResID);
    // 2. 创建ActionBar
    initWindowDecorActionBar();
}

//PhoneWindow.java
public void setContentView(int layoutResID) {
    // 1. 如果不存在ContentView,就创建
    if (mContentParent == null) {
        installDecor();
    }
    // 2. 加载setContentView参数所指定的布局，加载到ContentView中
    mLayoutInflater.inflate(layoutResID, mContentParent);
    // 3. 回调Activity的onContentChanged()接口---该接口是空实现，用于通知Activity视图已经发生改变.
    final Window.Callback cb = getCallback();
    if (cb != null && !isDestroyed()) {
        cb.onContentChanged();
    }
}

//PhoneWindow.java
private void installDecor() {
    // 1. 不存在DecorView就创建
    if (mDecor == null) {
        mDecor = generateDecor(-1);
    }
    // 2. 不存在ContentView就创建，布局为android.R.id.content
    if (mContentParent == null) {
        mContentParent = generateLayout(mDecor);
    }
}
```

1. Activity.setContentView(): 1-交给PhoneWindow处理 2-设置ActionBar
2. PhoneWindow.setContentView(): 1-创建ContentView(mContentParent就是布局为android.R.id.content的ContentView) 2-加载布局到ContentView中 3-回调Activity的onContentChanged()接口
3. onContentChanged()：用于通知Activity视图已经发生改变

19、 `DecorView` 何时才被 `WindowManager` 真正添加到 `Window` 中?

1. 即使Activity的布局已经成功添加到 `DecorView` 中， `DecorView` 此时还没有添加到 `Window` 中
2. `ActivityThread` 的 `handleResumeActivity` 方法中，首先会调用 `Activity` 的 `onResume` 方法，接着调用 `Activity` 的 `makeVisible()` 方法
3. `makeVisible()` 中完成了 `DecorView` 的添加和显示两个过程

```
void makeVisible() {
  //1. 将`DecorView`添加到`Window`中(通过WindowManager)
    if (!mWindowAdded) {
        ViewManager wm = getWindowManager();
        wm.addView(mDecor, getWindow().getAttributes());
        mWindowAdded = true;
    }
  //2. 将DecorView显示出来
    mDecor.setVisibility(View.VISIBLE);
}
```

20、DecorView是什么?

> 1. `DecorView` 是一个FrameLayout
> 2. `DecorView` 是Activity中的顶级View，内不包含标题栏(根据主题可以没有)和内部栏(id是android.R.id.content)

Dialog

21、 Dialog的Window创建过程

> 1. 创建Window——同样是通过 `PolicyManager` 的 `makeNewWindow` 方法完成，与Activity创建过程一致
> 2. 初始化 `DecorView` 并将 `Dialog` 的视图添加到 `DecorView` 中——和Activity一致(setContentView)
> 3. 将 `DecorView` 添加到 `Window` 中并显示——在 `Dialog` 的 `show` 方法中，通过 `WindowManager` 将 `DecorView` 添加到 `Window` 中(mWindowManager.addView(mDecor, 1))
> 4. `Dialog` 关闭时会通过 `WindowManager` 来移除 `DecorView` ： `mWindowManager.removeViewImmediate(mDecor)`
> 5. `Dialog` 必须采用 `Activity` 的 `Context` ， 因为有应用 `token` (`Application` 的 `Context` 没有应用token)，也可以将 `Dialog` 的 `Window` 通过 `type` 设置为系统Window就不再需要token。

22、为什么Dialog不能用Application的Context?

> 1. Dialog本身的Token为null，在初始化时如果是使用Application或者Service的Context，在获取到WindowManager时，获取到的token依然是null。
> 2. Dialog如果采用Activity的Context，获取到的WindowManager是在activity.attach()方法中创建，token指向了activity的token。
> 3. 因为通过Application和Service的Context将无法获取到Token从而导致失败。

Toast

23、Toast的内部机制介绍

> 1. `Toast` 也是基于 `Window` 来实现的
> 2. `Toast` 具有定时取消功能，系统采用 `Handler` 实现
> 3. `Toast` 内部有两类IPC过程：1-Toast访问NotificationManagerService；2-NotificationManagerService回调 `Toast` 的 `TN` 接口

24、Toast的show()方法原理分析

```
/**
 * Toast.java中显示和隐藏都是IPC过程
 */
public void show() {
    if (mNextView == null) {
        throw new RuntimeException("setView must have been called");
    }
    //1. 获取NotificaitonMnagerService
    INotificationManager service = getService();
    String pkg = mContext.getOpPackageName();
    //2. TN是Binder类，NotificationManagerService处理Toast显示时，会跨进程回调TN中的方法
    //3. TN运行在Binder线程池中，需要通过Hanlder将其切换到(发送Toast请求的)线程中
    //4. 使用Hanlder意味着Toast无法在没有Looper的线程中弹出
    TN tn = mTN;
    tn.mNextView = mNextView;
    try {
        //5. NotificationManagerService进行toast
        service.enqueueToast(pkg, //当前应用包名
                tn,
                mDuration);
    } catch (RemoteException e) {
        // Empty
    }
}

//NotificationManagerService
public void enqueueToast(String pkg, ITransientNotification callback, int duration)
{
    ...参数合法性判断...
    ...省略...
    synchronized (mToastQueue) {
        .......
        try {
            //1. 将Toast请求封装为`ToastRecord`
            ToastRecord record;
            int index = indexOfToastLocked(pkg, callback);
            //2. 如果ToastRecord已经在队列中(mToastQueue)，则直接更新
            if (index >= 0) {
                record = mToastQueue.get(index);
                record.update(duration);
            } else {
                /**================================================================
                 *3. 限制非系统应用，队列最多能存50个(MAX_PACKAGE_NOTIFICATIONS)。
                 *   防止DOS(Denail of Service)攻击：
                 *    如果一个应用大量连续弹出Toast，会导致其他应用没有机会弹出Toast
                 *================================================================*/
                if (!isSystemToast) {
                    int count = 0;
                    final int N = mToastQueue.size();
                    for (int i=0; i<N; i++) {
                        final ToastRecord r = mToastQueue.get(i);
                        if (r.pkg.equals(pkg)) { //同一个包
                            count++;
                            if (count >= MAX_PACKAGE_NOTIFICATIONS) {
                                Slog.e(TAG, "Package has already posted " + count
                                        + " toasts. Not showing more. Package=" + pkg);
                                return;
                            }
                        }
                    }
                }
                //4. 将Toast存放到队列中
```

```
                record = new ToastRecord(callingPid, pkg, callback, duration);
                mToastQueue.add(record);
                ......
            }
            /**============================================================
             * 5. 用于显示当前Toast
             *  Toast显示由ToastRecord的callback(Toast的TN对象的远程Binder)完成
             *  通过CallBack调用了TN中的方法，该方法运行在Toast请求方的Binder线程池中
             *============================================================*/
            //5. 用于显示当前Toast。
            if (index == 0) {
                showNextToastLocked(); //Toast显示由ToastRecord的callback(Toast的TN对象的远程Binder)完成
            }
        } finally {
            Binder.restoreCallingIdentity(callingId);
        }
    }
}

//NotificationManagerService.java
void showNextToastLocked() {
    ToastRecord record = mToastQueue.get(0);
    while (record != null) {
         ...省略...
        try {
            //1. 跨进程调用Toast中TN内部的show()方法进行展示
            record.callback.show();
            /**============================================================
             * 2. 会发送一个延时消息(取决于Toast的时长)。
             *  1-Message最终会由NotificationMaService中的WorkerHandler处理
             *  2-调用handleTimeout->cancelToastLocked->`record.callback.hide();
             *  3-最终调用TN中的hide方法并且将ToastRecord从队列中移除
             *============================================================*/
            scheduleTimeoutLocked(record);
            return;
        } catch (RemoteException e) {
            ...省略...
        }
    }
}
```

25、Toast的cancel()方法原理分析

> 1. 内部调用 `NotificationManagerService` 的cancelToast方法()
> 2. cancelToast方法()->cancelToastLocked()->record.callback.hide();
> 3. record.callback.hide()通过IPC调用TN中的hide方法
> 4. 最后将ToastRecord移除队列

```
//Toast.java
public void cancel() {
    mTN.hide();

    try {
        getService().cancelToast(mContext.getPackageName(), mTN);
    } catch (RemoteException e) {
        // Empty
    }
}
```

26、Toast的TN(Binder)内部机制

```
//Toast.java TN
private static class TN extends ITransientNotification.Stub {
    ...省略...
    WindowManager mWM;
    TN() { ...省略... }
    //1. 开启了mShow的Runnable
    public void show() {
        mHandler.post(mShow);
    }
    //2. 开启了mHide的Runnable
    public void hide() {
        if (localLOGV) Log.v(TAG, "HIDE: " + this);
        mHandler.post(mHide);
    }
    //3. 调用handleShow()进行显示
    final Runnable mShow = new Runnable() {
        public void run() {
            handleShow();
        }
    };
    //4. 调用handleHide()进行显示
    final Runnable mHide = new Runnable() {
        public void run() {
            handleHide();
            mNextView = null;
        }
    };
    //5. 真正完成显示功能
    public void handleShow() {
        ......
        //7. 将Toast视图添加到Window中
        mWM = (WindowManager)context.getSystemService(Context.WINDOW_SERVICE);
        ......
        mWM.addView(mView, mParams);
        ......
    }
    //6. 真正完成隐藏功能
    public void handleHide() {
        if (mView != null) {
            //8. 将Toast视图从Window中移除
            if (mView.getParent() != null) {
                mWM.removeView(mView);
            }
            mView = null;
        }
    }
    private void trySendAccessibilityEvent() {
        ......
    }
}
```

进阶题

1、UI线程一定是主线程吗?

> 1. WindowManager的addView方法中会创建ViewRootImpl。 ViewRootImpl在创建时会将所在线程指定为UI线程。
> 2. 也就是说调用WM的addView方法的线程就是UI线程。
> 3. UI操作会在 `ViewRootImpl` 的checkThread中进行线程检测。

> 4. 对于Activity的启动：UI线程 = 主线程。
> 5. 对于WM的addView： 在哪个线程调用，哪个线程就是新建的Window的UI线程。

2、子线程操作UI一定会崩溃吗?

> 1. 在onCreate()里面创建子线程进行UI操作(线程中不能有延迟)，就不会崩溃。因为在handleResumeActivity内部执行WM.addView时才创建了ViewRoot并进行UI线程检测。
> 2. 在荣耀Play(Android 8.1 API27)的手机上，通过子线程操作UI不会有问题。

参考资料

1. Dialog为什么不能用Application的context
2. WMS之Token