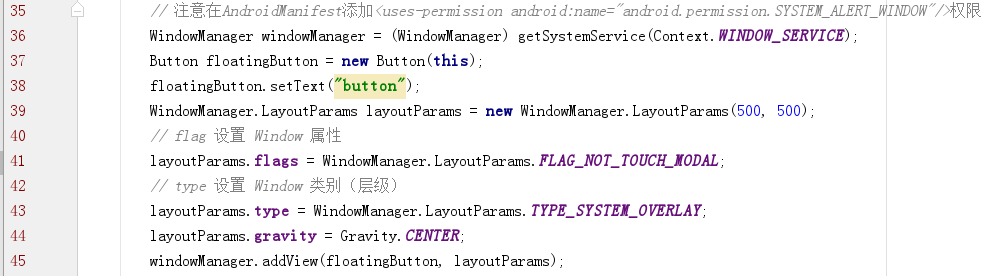
**Window**

参考：<http://www.bijishequ.com/detail/515623?p=>

Window表示一个窗口的概念，在android中常见的Window是Activity、Dialog、Toast或者是桌面的一个悬浮框，他们的视图View都是附加在Window上的，所以Window是它们视图的直接管理者。

**创建**

通过WindowManager创建一个Window：



上图中并没有出现Window类，而是通过获取了一个WindowManager添加了一个控件而创建了一个Window，其实Window是个抽象类。而WindowManager是外界访问Window的入口，WindowManager是一个接口，WindowManager的实现类是WindowManagerImpl。这里有两个参数需要分析一下：

**flags**:控制Window的显示特性。下面展示部分：

FLAG\_NOT\_TOUCH\_MODAL：此模式下，系统会将Window区域以外的单击事件传递给底层的Window，区

域以内的单击事件会则由自己处理。

FLAG\_NOT\_FOCUSABLE：表明Window不需要获取焦点，也不需要接收各种输入事件，此标记会同时启

用FLAG\_NOT\_TOUCH\_MODAL。

FLAG\_SHOW\_WHEN\_LOCKED：开启此模式可以让Window显示在锁屏界面上。

**type：**代表Window的类型。

【应用Window】：对应一个Activity。

【子Window】：不能单独存在必须附加在一个特定的父Window中，例如一个Dialog就是一个子

Window。

【系统Window】:比如：Toast和系统状态栏。但是需要声明权限才能创建的Window

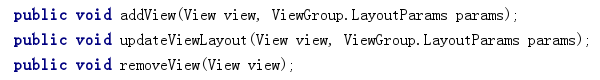
<uses-permission android:name="android.permission.SYSTEM\_ALERT\_WINDOW"/>。

Window是分层级的层级大的会覆盖在层级小的Window上面 ，层级的值对应着WindowManager.LayoutParams.type的参数：

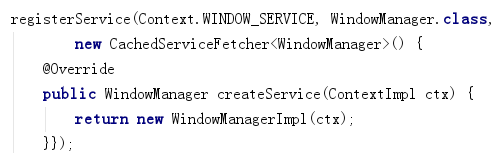
|  |  |
| --- | --- |
| Window类型 | 层级 |
| 应用Window | 1-99 |
| 子Window | 1000-1999 |
| 系统Window | 2000-2999 |

* **WindowManager**

在上面Window创建过程中是通过WindowManager来添加一个View的，但是WindowManager是一个接口它继承了只有三个方法的ViewManager接口：

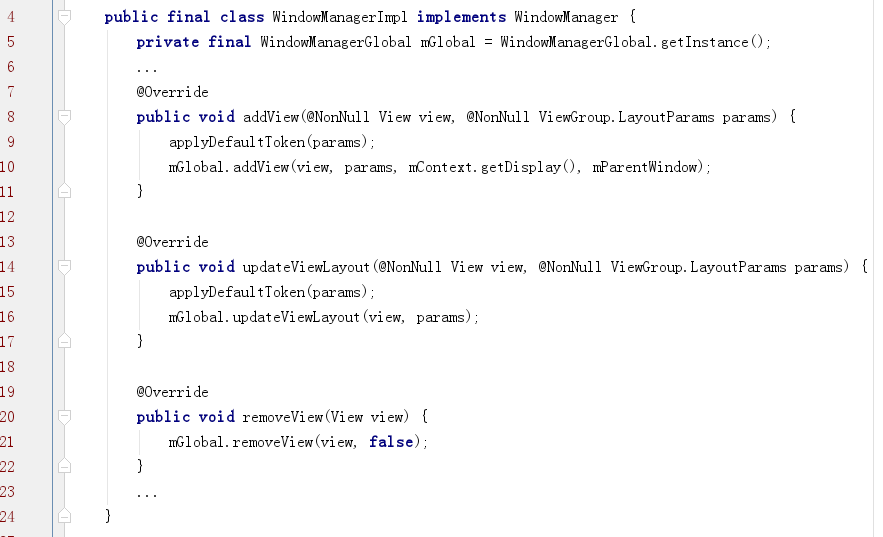


而且WindowManager只对外提供这三个方法：添加View、更新View、删除View。而这三个功能已经足够我们使用了。从这三个方法来看WindowManager只是提供操作Window里面的View的功能。所以通过getSystemService方法返回的一定不是WindowManager对象，那么我们看下getSystemService方法源码返回的是什么（具体见获取系统服务篇）：

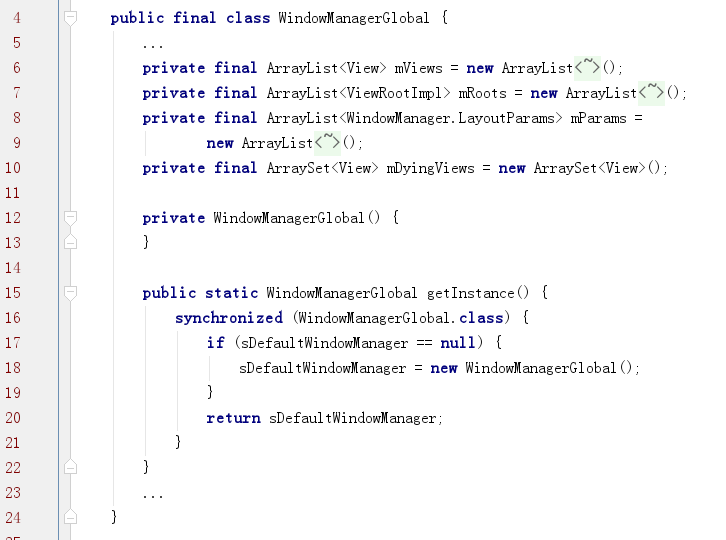


通过源码看出getSystemService方法返回的是一个WindowManagerImpl实例对象，它是WindowManager的实现类。所以WindowManager对Window视图的操作实现在WindowManagerImpl类里：

说明：由上面代码可以看出返回的WindowManagerImpl实例是可以被当前Activity缓存的，意思就是说每个调用getSystemService方法获取WindowManager的Activity都会有一个属于自己的WindowManagerImpl实例，当WindowManagerImpl实例第一次被创建后就会缓存在这个Activity中，下次使用这个Activity调用getSystemService方法获取WindowManager的时候会直接返回缓存的这个WindowManagerImpl实例，所以并不是所有的Activity使用的是同一个WindowManager。



通过上图可以看出WindowManagerImpl实现了WindowManager接口中的方法，但是她又将对视图的操作全部交给了WindowManagerGlobal来实现，L5并不是new了一个WindowManagerGlobal对象，而是通过getInstance方法获取得实例，继续来看WindowManagerGlobal类：



上图可以看出WindowManagerGlobal的实例是个单例模式存在的，也就是说整个app里只有一份WindowManagerGlobal实例，那么问题来了，一个应用的所有的Window中的视图都是由一个WindowManagerGlobal实例来处理吗？答案是肯定的，答案就在上图中L6、L7、L8、L10的几个集合里：

mViews：存储了所有Window所对应的View。

mRoots：存储了所有的Window所对应的ViewRootImpl。

mParams：存储了多有Window所对应的布局参数。

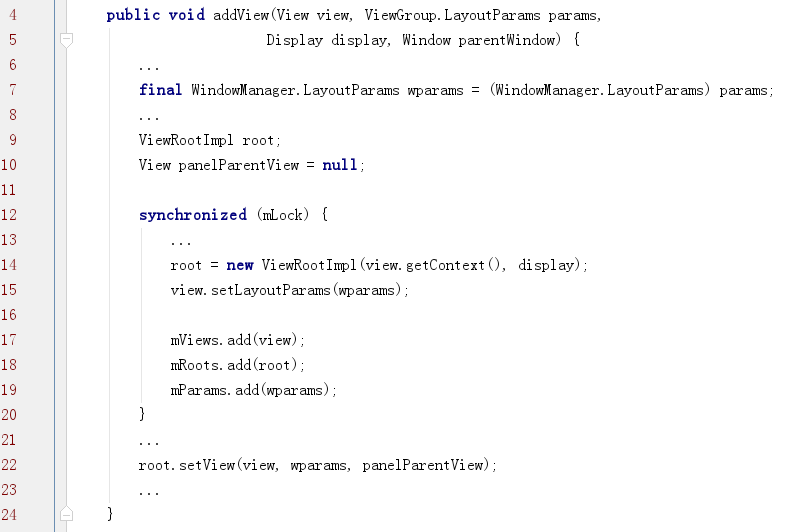
mDyingViews：存储了那些正在被删除的对象，或者说是那些已经调用WindowManager

的removeView方法但是删除操作还未完成的Window对象。

* **Window内部机制**

既然知道了WindowManager对Window视图的操作最终是在WindowManagerGlobal类中实现的，接下来就分析WindowManager接口中的三个方法，也就明白了Window的内部机制。

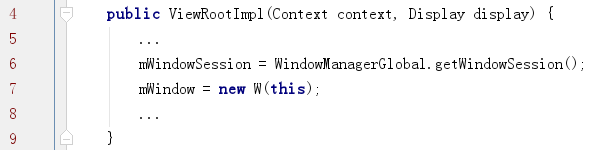
* **addView**



在addView方法中，L14会创建一个ViewRootImpl,然后将view、root、params依次添加进app进程中WindowManagerGlobal实例的集合中维护。关键看L22这里将传进来的View添加进ViewRootImpl实例root中，**ViewRootImpl可以看做也是Window和View之间的通信的纽带**，比如将View添加到WMS、处理WMS传入的触摸事件、通知WMS更新窗口大小等、同时View的绘制过程是由ViewRootImpl完成的。接下来看setView方法：



View添加进ViewRootImpl后首先会在L16开始进行View的绘制(具体看上图注释)，一个View的绘制是由此开始的，requestLayout方法里面会调用scheduleTraversals方法，这个方法就是View绘制的入口。接下来看关键点L22通过mWindowSession来完成mWindow的添加。这里mWindowSession和mWindow对象是在ViewRootImpl初始化的时候实例化的：



mWindowSession它是通过WindowManagerGlobal.getWindowSession获得的一个Binder服务代理，是App端向WMS发送消息的通道。相对的，**mWindow是一个W extends IWindow.Stub Binder服务对象，其实可以看做是App端的窗口对象**，传递给WMS，并作为WMS向APP端发送消息的通道，在Android系统中存在大量的这种互为C\S的场景。接着看mWindowSession获取的具体操作是：首先通过getWindowManagerService 获取WMS的代理，之后通过WMS的代理在服务端open一个Session，并在APP端获取该Session的代理：



Session内部会通过WindowManagerService来实现Window的添加，Window的添加过程是一次IPC调用，这里就不深究了。WindowManagerGlobal、Window、ViewRootImpl、WindowManagerService（WMS）、IWindowSession他们大体的关系：

首先WindowManagerGlobal实例是个单例模式的存在。它里面有又存储了WindowManagerService（WMS）、IWindowSession对象也是单例模式的存在，这里的单例模式是指在整个App进程内只存在一份，IWindowSession是由WMS创建的一个seeeion对象，主要用于App端向WMS发送消息。继续在WindowManagerGlobal的addView方法中创建ViewRootImpl对象，然后ViewRootImpl的构造方法中会创建一个IWindow.Stub对象（可以看做是一个Window对象）和获取IWindowSession对象。mWindow对象被创建后会传给WMS，主要用于WMS向app端发送消息。接着在ViewRootImpl的setView方法中调用requestLayout绘制根View，然后通过IWindowSession的addToDisplay方法将mWindow对象添加到WMS中。所以说ViewRootImpl是Window和View之间的通信的纽带。

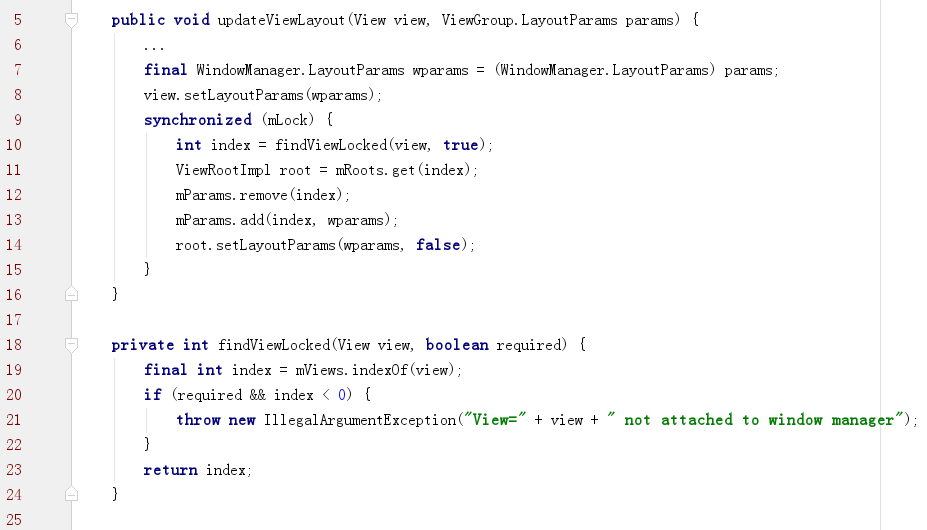
通过mWindow向App端发送消息

通过IWindowSession代理发送消息

WMS

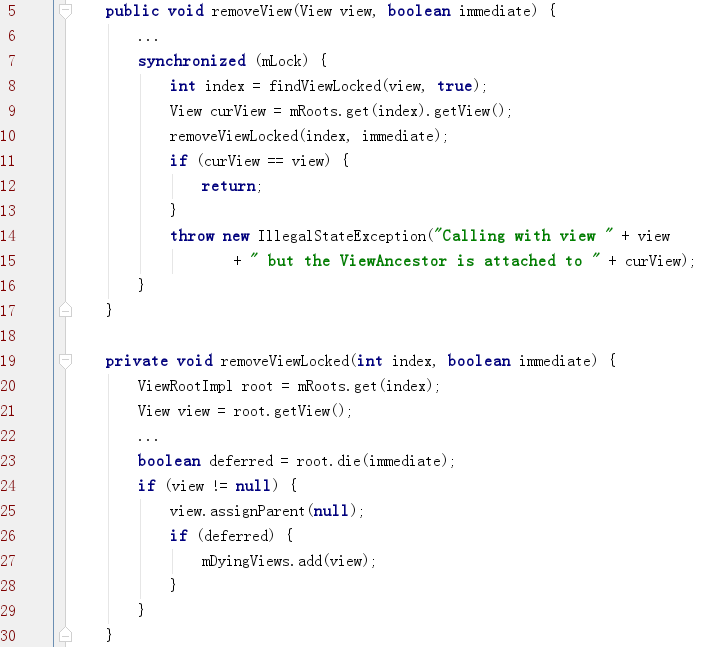
App端

* **updateViewLayout**

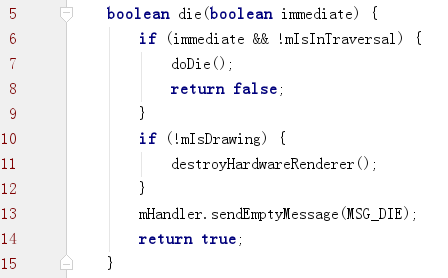


更新View的LayoutParams参数，然后替换掉mParams集合中的旧的LayoutParams，最后调用ViewRootImpl的setLayoutParams方法替换掉ViewRootImpl中的LayoutParams，而在setLayoutParams方法中会调用scheduleTraversals()方法对View进行重新测量、布局、重绘。

* **removeView**



删除View有removeView和removeViewImmediate两个方法，它们分别表示异步删除和同步删除，为了避免发生意外错误一般使用异步删除。删除过程主要在L23调用ViewRootImpl的die方法中。在die中主要使用mHandler发送了一个删除View的消息就立刻返回了，这个时候View并没有完成删除操作，所以在L27将view添加进mDyingViews集合中表示待删除的View。下面是ViewRootImpl的die方法：



L6做了判断，如果是异步删除则调用L13发送消息，Handler接收到消息后会调用doDie方法，如果是同步删除则L7直接调用doDie方法删除View，这就是两种删除的区别,下面看doDie方法：



真正删除View的逻辑在L13调用destroyHardwareRenderer方法中，这里就不贴代码了，其中主要做了四件事:

1. 垃圾回收的相关工作, 比如清除数据和消息,移除回调。
2. 通过Session的remove方法删除Window: mWindowSession.remove(mWindow), 这同样是一个IPC过程, 最终会调用WMS的removeWindow()方法.
3. 调用View的dispatchDetachedFromWindow()方法, 内部会调用View的onDetachedFromWindow()以及onDetachedFromWindowInternal(). 而对于onDetachedFromWindow()就是在View从Window中移除时, 这个方法就会被调用, 可以在这个方法内部做一些资源回收的工作. 比如停止动画,停止线程
4. 调用WindowManagerGlobal#doRemoveView方法刷新数据, 包括mRoots, mParams, mDyingViews, 需要将当前Window所关联的这三类对象从列表中删除.

删除完了之后最后在L37调用WindowManagerGlobal的方法跟新集合中数据：

