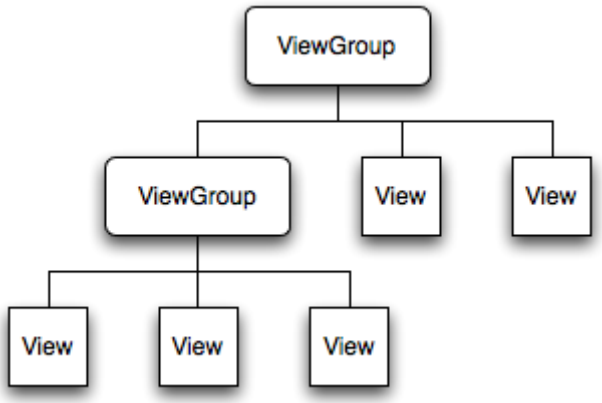
第八章 Android GWES

8.1 View System

View & ViewGroup

Android 的 UI 系统是建立在 View 和 ViewGroup 之上的。View 是组成 UI 基本部件，而 ViewGroup 可以把它看做 Panel，就是容器。可以把 View 和 ViewGroup 放到一个 ViewGroup 里布局，这些 View 和 ViewGroup 就组成了显示 UI 的树形结构（下图）。Activity 创建的 Window 中的主 View 既 DecorView 继承自 FrameLayout，而 FrameLayout 又继承自 ViewGroup，这个 DecorView 就是每个 Activity 最最根部的那个 ViewGroup。



View 在 Android 中不单纯只是负责显示功能，它还封装了 UI Event 的处理功能。在系统中所有要显示的 UI 元素都继承自 View（Surface 除外），用户可以使用 Android 中已实现的一些 View 的子类，这些子类包括 TextView，Button 等，在这里叫他们 widget，这些 widget 都继承自 View，但都他们各自的 UI 表现形式，以及对 UI Event 的不同处理方式。当然如果用户觉得这些 widget 无法满足的话，可以自定义 View，既用户去实现一个继承了 View 的类，也可以继承 widget，并对 View 的相关成员函数进行重写，如 OnDraw。

UI Event

View 中以接口的形式定义一些类 UI Event 处理，其实接口里就是定义了一个回调函数，共使用者去实现，当有消息来时，就会调用到使用者实现的回调函数里。

参考 View.java

```
public void setOnTouchListener(OnTouchListener l) {
    mOnTouchListener = l;
}

public interface OnTouchListener {
    boolean onTouch(View v, MotionEvent event);
}
```

```
public boolean dispatchTouchEvent(MotionEvent event) {
    if (mOnTouchListener != null && (mViewFlags & ENABLED_MASK) == ENABLED &&
        mOnTouchListener.onTouch(this, event)) {
            return true;
    }
    return onTouchEvent(event);
}
```

参考 usr.java

```
usrTouchView. setOnTouchListener(new OnTouchListener{
    onTouch() {
        TODO::
    }
})
```

关于消息是如何传递到 dispatchTouchEvent 的，在输入消息处理那一块会有详细介绍。

Graphics

上面介绍了 View 是如何处理 UI Event，接下来会介绍 View 是如何显示出来，既画出来的。前面提到过 View 的 OnDraw 函数，可以说 View 的所有显示都在 OnDraw 里面，每当 View 需要刷新时，都会调用的 OnDraw，它的函数声明如下：

```
protected void onDraw(Canvas canvas)
```

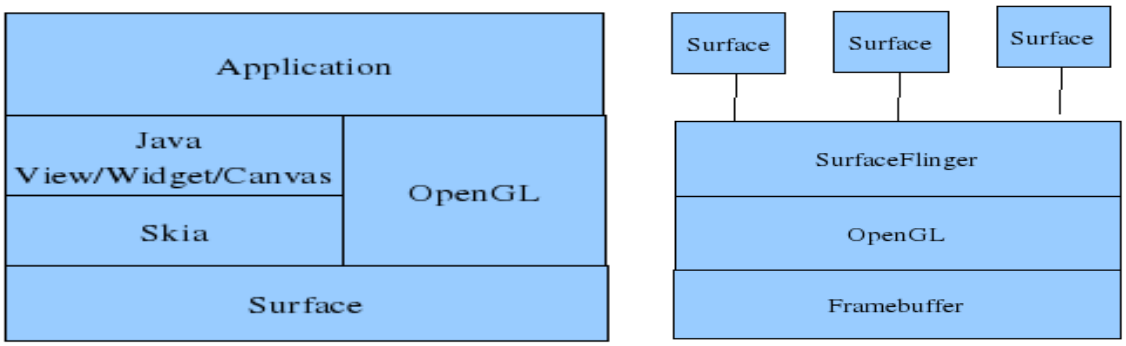
每一个 View 的子类，包括 widget 都对它重写，这个函数的重点就是 Canvas，所有显示的画图动作都是通过该类来实现的。它位于 android.Graphics 的 package 内，Canvas 提供一些类画图的方法，如 drawPath(Path path, Paint paint)。使用 Canvas 画图还需要一些其他资源

- Bitmap：用于承载 pixel，既所画的内容
- Path：Region 提供画的区域
- Paint：描述颜色及样式

参考 Canvas.java

```
public void drawPath(Path path, Paint paint) {
    native_drawPath(mNativeCanvas, path.ni(), paint.mNativePaint);
};
private static native void native_drawPath(int nativeCanvas, int path,  int
paint);
```

从上面的代码片段了解到 Canvs 的这些画图方法，最后都是调用相对应的 native API。这些 native API 到底层的调用流程是 Skia/OpenGL====>SurfaceFlinger=====>framebuffer。



8.2 Android 窗口管理

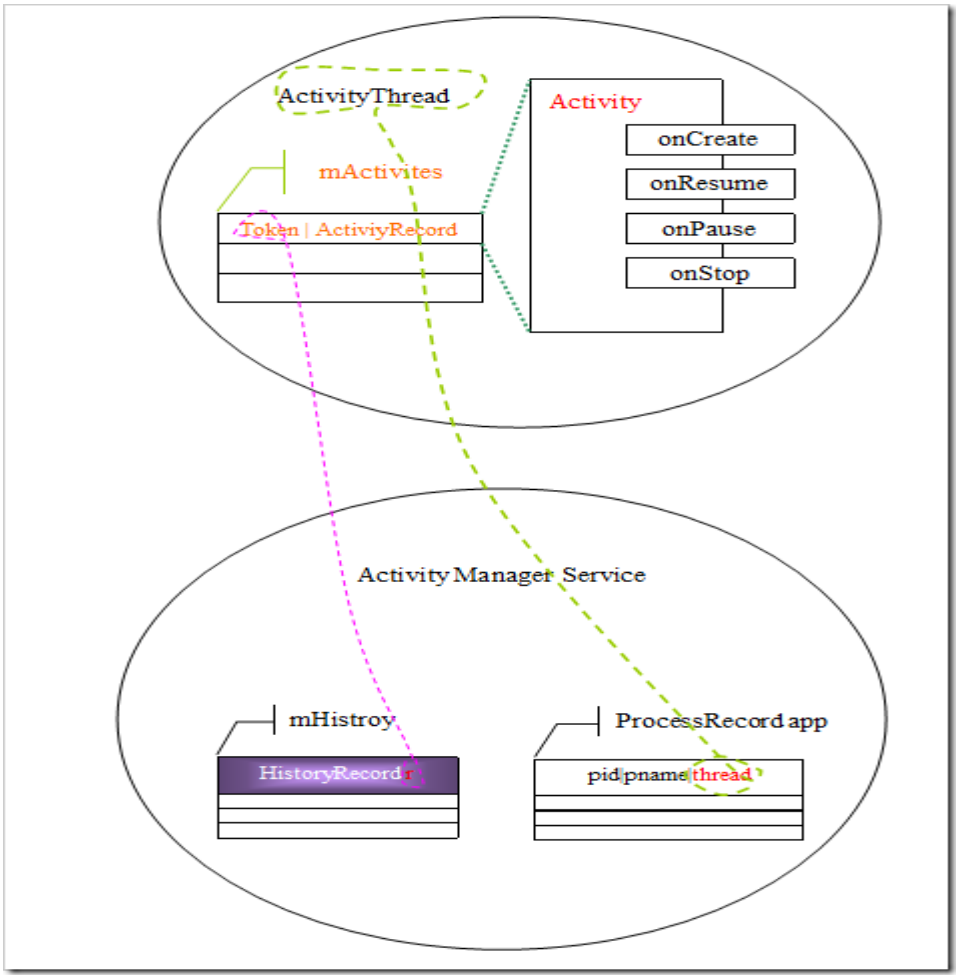
Activity 如何启动的？

系统初始化时会启动一系列服务，这其中包括 Activity Manager Service 和 Window Manager service，这些 Service 属于常驻程序，应用的进程可获得 Service 的本地代理来与 Service 进行通信，一般通过 Binder 或 AIDL 接口。

Activity 启动首先要程序的进程跑起来，而一个应用程序的主线程的启动时通过 AMS（Activity Manager Service 的缩写）创建 Activity Thread 对象，并由该对象来创建进程的主线程，通过 AMS 的 request 调度该进程的 Activity。并且会创建消息循环 Looper 来实现消息的分发。

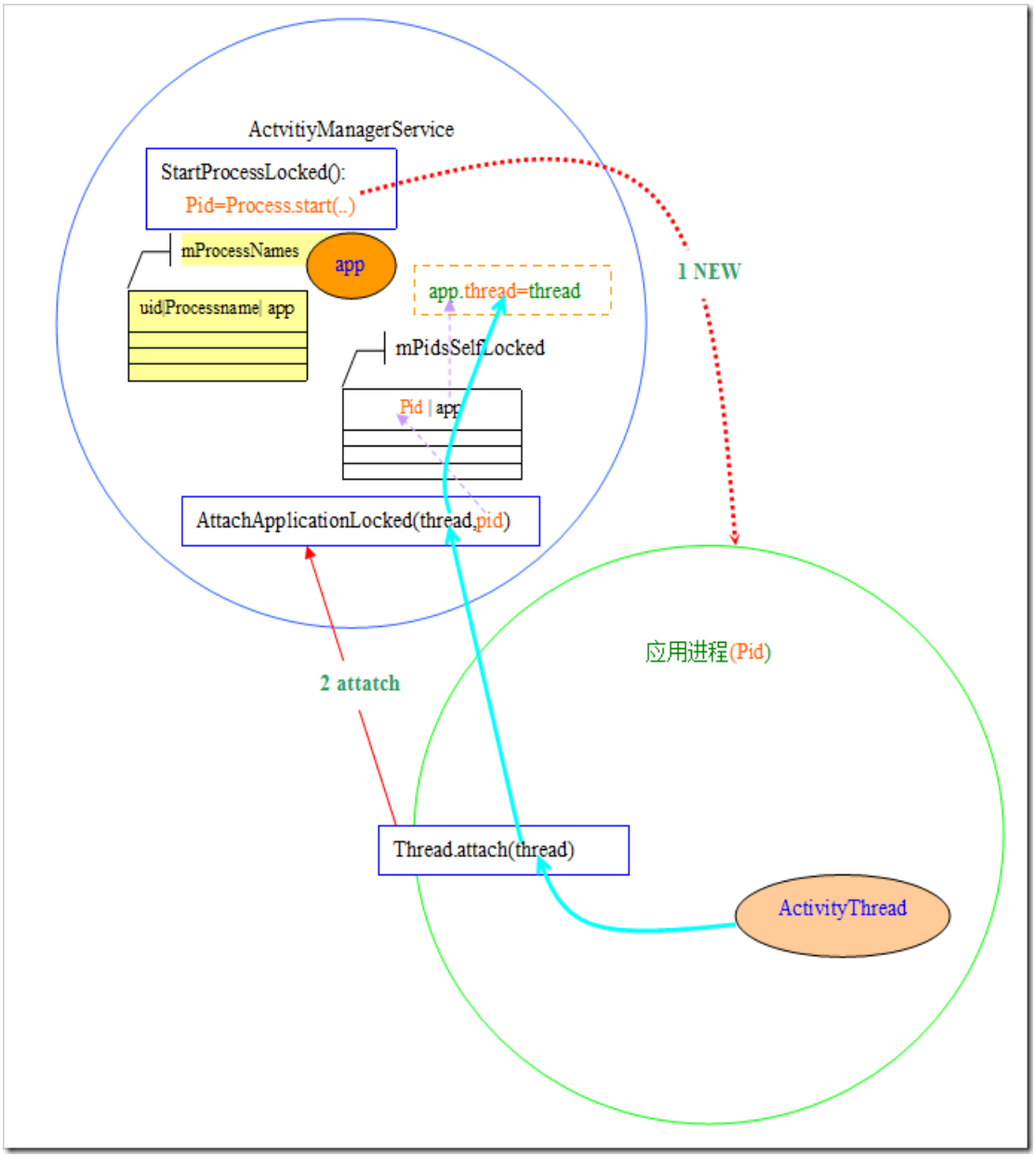
> ActivityThread 的成员变量 mActivities 是个数组，用来保存该进程中的所有的 Activity，并以 ActivityRecord 形式存在，而在 AMS 中也同样有个 mHistory 数组来管理所有的 Activity。这个 mHistory 保存的是 HistoryRecord 类，HistoryRecord 中有 token，taskid。Taskid 标示是哪个 task， token 标示哪个 Activity。

下图是介绍 ActivityThread 中的 ActivityRecord 与 AMS 中 HistoryRecord 联系。



Activity 启动流程

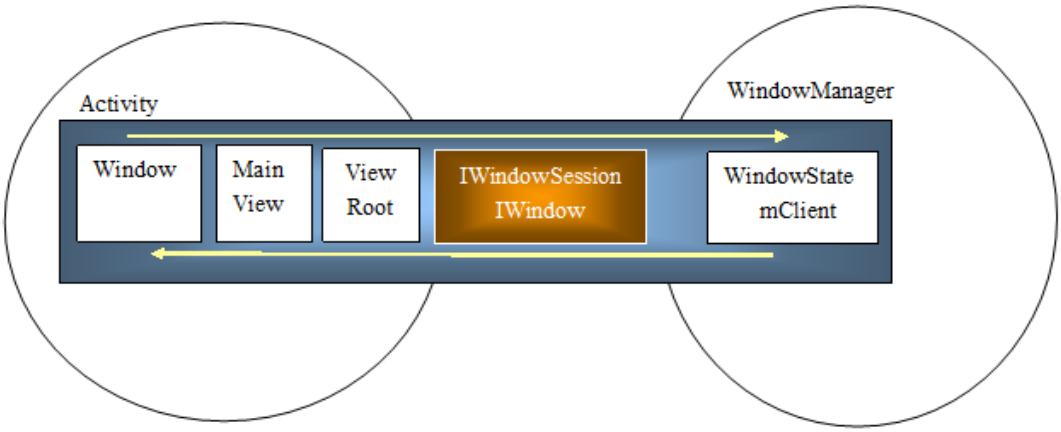
1. 发起请求 startActivity(intent)
2. Activity Service Manager 接收到请求执行 StartActivity 函数。
3. 通过 app.thread.scheduleLaunchActvity 在 App 应用中建立新的 ActivityRecord。
4. 建立新的 Activity 对象并放入到 ActivityRecord 中。
5. 将 ActivityRecord 加入到 mActivites@ActivityThread
6. 发起 Activity.onCreate(..),,该 onCreate 就是在你的应用程序 XXXActivity 中的 onCreate。



窗口的基本框架

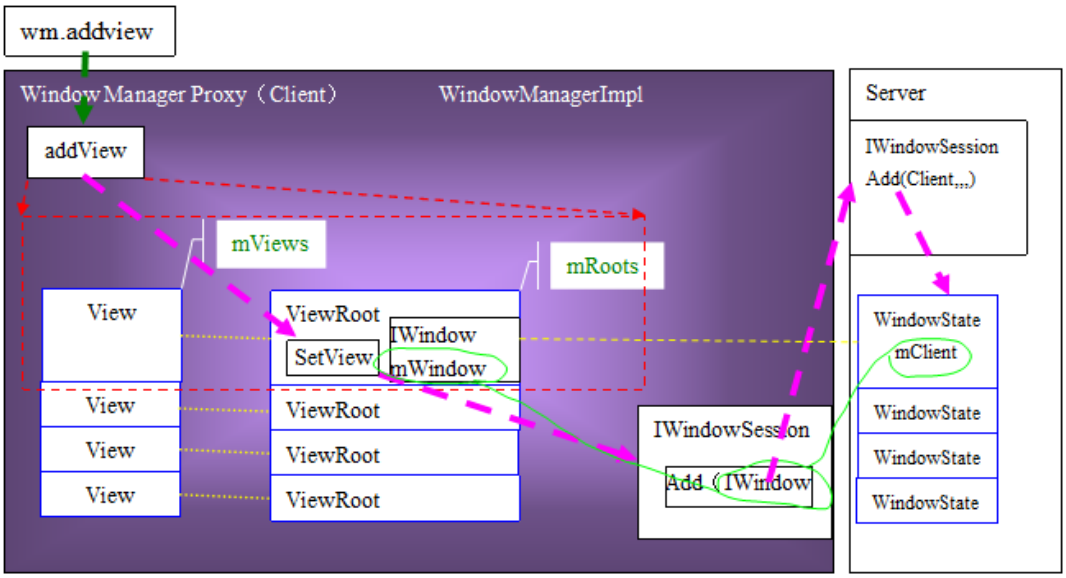
Activity 建立一个主窗口后，在将主窗口添加到 WindowManager 时，首先要建立

WindowManager 代理对象，并打开一个会话（实现 IWindowSession AIDL 接口），并维持该会话。Activity 将通过该会话与 WindowManager 建立联系，这个 Session 是 C/S 体系的基础，Client 通过 WindowSession 将 window 加入到 Window Manager 中。客户端的 Activity 通过 Session 会话与 WindowManager 建立对话，而 WindowManager 则通过 IWindow 接口访问 Client，将消息传递到 Client 端，通过消息分发渠道，将消息传递到处理函数 OnXXX。



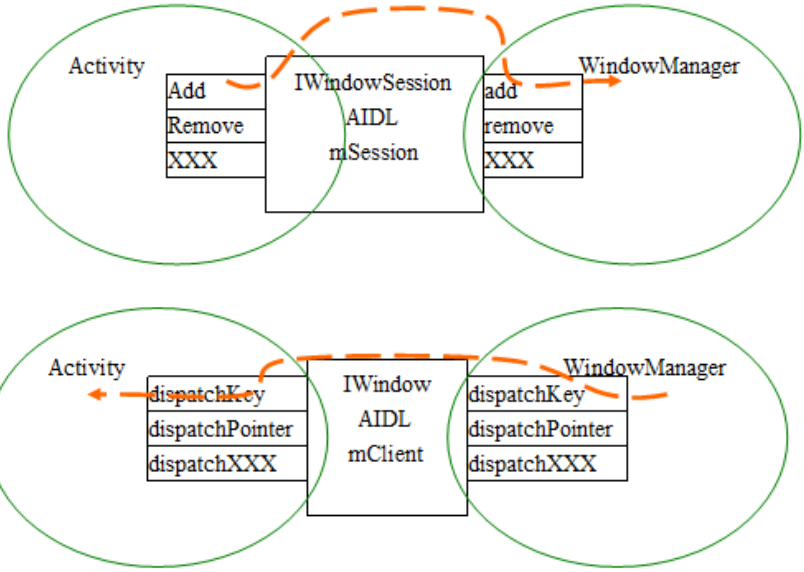
在 ActivityThread 调用 performLaunchActivity 时，会使用 Activity.attach（）建立一个 PhoneWindow 主窗口。这个主窗口的建立并不是一个重点。当 ActivityThread 调用 handleResumeActivity 去真正启动一个 Activity 时候，将主窗口加入到 WindowManager，当然并不是将主窗口本身，而是将主窗口的 DecorView 加入到 WindowManager 中。DecorView 实际上是一个 ViewGroup，DecorView 是 Top-Level View。

下图描述的是如何将窗口的主 View 既 DecorView 添加到 window manager



图中在添加 DecorView 到 WM 中时，会创建一个 ViewRoot，每个窗口(DecorView)都会对应一个 ViewRoot，他是整个 View 树形结构的根，ViewRoot 实际是一个 Handler，View Tree 消息的发送和处理都是从他开始。ViewRoot 是建立主 View 与 WindowsManger 通讯的桥梁。ViewRoot 与 Window Manager 之间的联系是 IWindowSession 和 IWindow。ViewRoot 通过 IWindowSession 添加窗口到 Window Manager。而 IWindow 这是 Window Manager 分发消息

给 Client ViewRoot 的渠道。IWindowSession 和 IWindow 都是 AIDL 接口，窗口与 Window Manager Service 就是利用这两个 AIDL 接口进行进程间通信。

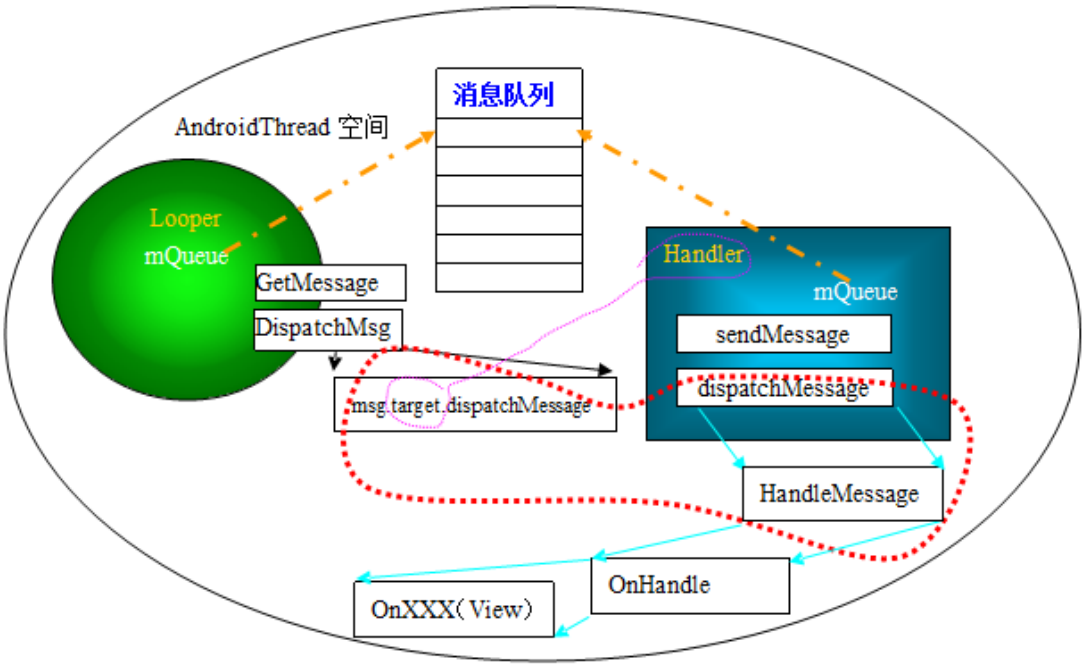


消息机制 Looper，Handle，MessageQueue

Looper 只是产生一个消息循环框架，首先 Looper 创建了 MessageQueue 并把它挂接在 Linux 的线程上下文中，进入到取消息，并分发消息的循环当中。

Handle 对象在同一个线程上下文中取得 MessageQueue，最主要的就是 SendMessage 和担当起 dispatchMessage 这个实际工作。

MessageQueue 就是消息队列，用来存放消息。

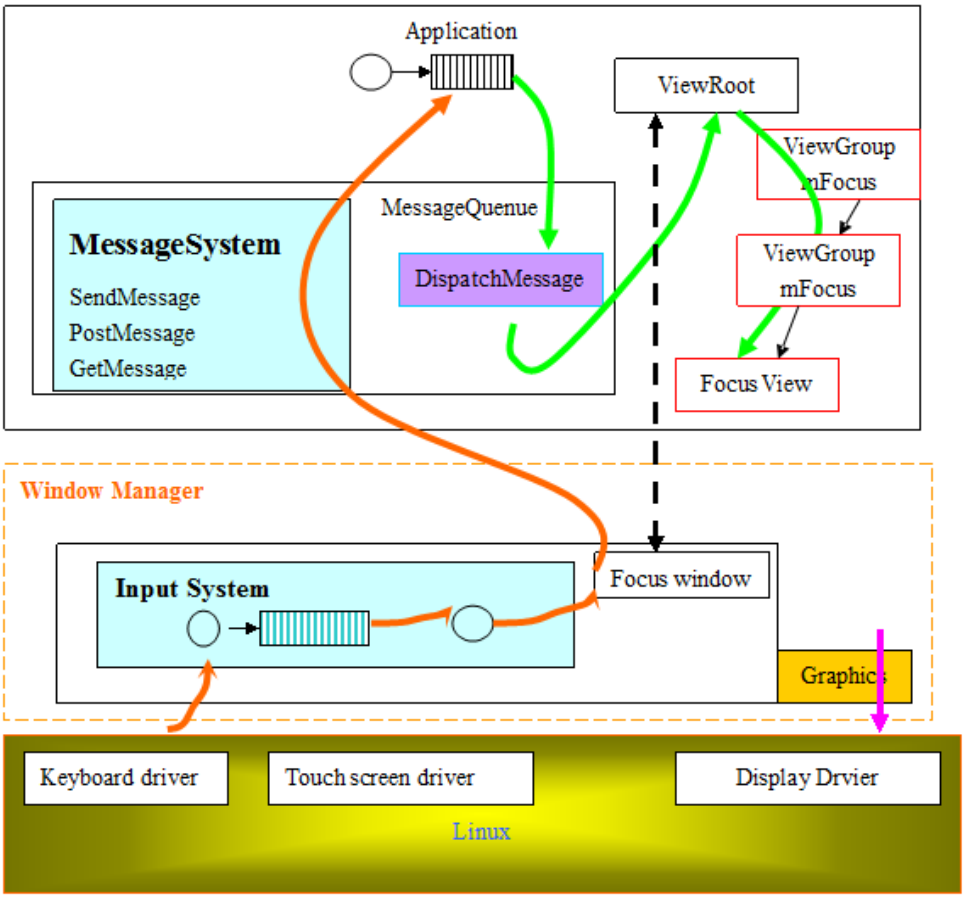


消息的产生及处理流程（见上图），首先在应用程序启动时会创建 ActivityThread 对象来启动主线程，并管理当前应用所有的 Activity，保存到 mActivities 数组中（详细介绍见 Activity 启动介绍），在 ActivityThread 里会创建一个 Looper 对象，既该线程的所有 Activity 都共用一个消息循环，Looper 里会创建 MessageQueue 来存放 Msg，Looper 对象创建成功后，会调用 Looper 的静态成员函数 loop()开始消息循环，该静态函数中会获得当前线程的 Looper 对象，并从该 Looper 的 MessageQueue 中 GetMessage，获得的 msg 对象有个 target 的成员变量，该 target 就是要处理该 msg 的 Handle 对象，如果 target 为 null，则代表终止该消息循环，否则调用该 handle 的 dispatchmessage，该函数中会调用 handlemessage 回调函数，该函数会对消息进行处理，每个创建的 handle 对象要对 handlemessage 进行重写。如 ViewRoot 的 handlemessage 就是分发到 Focus View 的 OnXXX()。

但是消息是如何放到 MessageQueue 中的呢？下面介绍一下 Handle 类，就是刚才提到的 msg 中的 target 域，该类中包括 handlemessage，dispatchmessage 以及一系列 sendmessage 函数，前两个函数的应用场景前面已经讲过了。这一系列 SendMessage 就是最后都会调用 sendMessageAtTime()，这个函数就是把发送的消息放到当前消息循环的 MessageQueue 中。

对输入事件如何响应？

系统中与用户交互的程序，都是用户使用输入设备(鼠标，键盘，触控板)，然后由这些设备的驱动获得输入，并生成 Event，放到 Event 队列中。上层的 Window Manager Service 会通过 Native 函数来读取这些 Event，并分发到具体的 Focus View 上。



1、 输入事件的收集

WindowMangerService 启动时会创建一个 KeyQ 对象 mQueue，这是一个消息队列，继承自抽象类 KeyInputQ，在 KeyInputQ 中建立一个独立的线程 InputDeviceReader，使用 Native 函数 readEvent 来读取 Linux Driver 的数据构建 Raw Event，放到 KeyQ 消息队列中。

2、 消息分发到 Focus Window

WMS（Window Manager Service）有 InputDispatcherThread 的线程，该线程循环的从 mQueue 中获得 Raw Events，并根据 Event Type 分别调用不通 dispatch 函数(dispatchKey, dispatchPointer, dispatchTrackball)，在 WMS 的 dispatch 找到中的 Focus Window，下面的代码片段中 WindowState，是 Client 端 Window 在 WMS 端的对应对象，通过 WindowState 记录的 mClient(IWindow)接口，将 Events 专递到 Client 端。

```
//InputDispatcherThread
DispatchKey()
{
Object focusObj = mKeyWaiter.waitForNextEventTarget(event, null,
             null, false, false, pid, uid);
。。。
WindowState focus = (WindowState)focusObj;
。。。
Focusobj.mClient.dispatchKey(event)
。。。
}
```

3、 应用消息队列分发

通过上面窗口的框架介绍，我们了解到 WindowState 中 mClient 既 IWindow 接口，是 WMS 同 Activity 的 Window 沟通的接口，通过该接口把 event 发到 Focus Window 的 ViewRoot 对象，ViewRoot 继承自 Handle，ViewRoot 中的 dispatchKey 会调用 SendMessage，把该 Event 发到当前线程的消息队列里，最后会调用到 ViewRoot 重写 handlemessage。

4、 通过 FocusPath，发到 FocusView

ViewRoot 的 handlemessage 函数找到他对应 DecorView 并，按下图所示的 Focus Path 会最终传递到要处理该事件的 view 上，并调用相应的回调处理函数，如 OnClick()。