Android 核心分析 之一--------分析方法论探讨之设计意图

分析方法论探讨之设计意图

为什么要研究 Android，是因为它够庞大，它够复杂，他激起了我作为一个程序员的内心的渴望，渴望理解这种复杂性。我研究的对象是作为手机开发平台的 Android 软件系统部分，而不是 Dalvik 虚拟机本身。

作为一个从其他平台装接过来的程序员，要从事 Andoid 平台系统开发，我的关于手机平台上积累的知识已经不能满足需要了，Android 为我们带来了大量的新名词，Activity，Manifest，

INTENT，Service，Binder，Dalvik 虚拟机，Framework，Linux，Navtive ，JNI.....。通过在源代码，在开发社区，在开发博客，甚至在招聘过程中，我不断的寻求 Android 是什么。经过一定时间的沉淀，我慢慢的理解到 Android 不仅仅是一类手机的总称，不仅仅是一个手机开发平台，不仅仅是一个虚拟 java 操作系统，不仅仅是一个开发社区，一个开发标准，不仅仅是一堆代码，Android 已经成了一个新的潮流。

代码多，系统复杂，纵观社区中 Android 的研究者，一开始从源代码分析 Android 就走向迷途，不断的跋山涉水，向纵深冲刺，最终脑袋堆栈不够用，迷失在开始的旅程，或者挂在半途中，鲜有通达者。我感觉到大部分的研究者总是忘记站在高山上向下望一望设计者的意图，一味的随着代码的控制流走入繁杂的谜团，陷入到复杂性的深渊。

我的研究分析是从设计者的意图出发，从抽象的甚至从哲学的高度，从最简单的系统原型开始，从设计猜想开始，而不是一开始就从代码分析展开。首先理解 Android 大的运行框架，主干流程，系统原型，之后再用源代码分析充实之。当然我这里的设计者意图并不是真正的 Android 设计者意图，而是我以为的 Android 设计者意图。

要理解设计者意图，就需要抽象。我们需要在哲学意义空间中去考虑系统的描述，即系统在本质上要表达什么。在逻辑空间上去考虑系统基本构成和动态结构。从现实到虚拟对象的映射去理解系统对象的组成，在从数据流的角度分析数据的产生者和消费者之间作用关系，从控制流的角度去分析对象之间的交互关系，从函数调用去分析具体的层次关系。

在系统设计上，原型是最能表达哲学空间和逻辑空间中系统本质的东西，原型是事物本质的第一层体现。我以为任何复杂的系统都一个简洁的系统原型，都有它简洁的意义。系统原型是设计者意图的第一体现，所以我们需要从几个方向上去提炼系统原型：

（1）从系统本质和基本原理出发

（2）从分析系统数据流和控制流分析出发。

从设计者意图出发，得出系统原型，提取到大的逻辑结构和系统构成是第一步。之后我们可以从设计者的角度考虑系统猜想系统设计，为什么要这样设计，为什么要有这些构成。这样的基本原型是什么？系统的限制是什么，应用场景有哪些，有些设计的引进还是系统收敛性而为之呢。我们还可以从代码痕迹上去分析，这些概念是如何的得来的？从一定的抽象和高度去理解这些问题，遵循系统原型出发之原则，在深入分析代码的时候，就不容易陷入细节中。我们就可以随时跳出来想，这些代码在整体上载表达一个什么概念，在描绘一个什么逻辑，他要构成一个虚拟层吗？他是在管理这个硬件吗？他在 虚拟这个对象吗？他在构建管理机构？还是在构建一个对象管理？空间管理，为了快速引入了什么样的复杂算法，实际上的原型算法应该是什么样的？

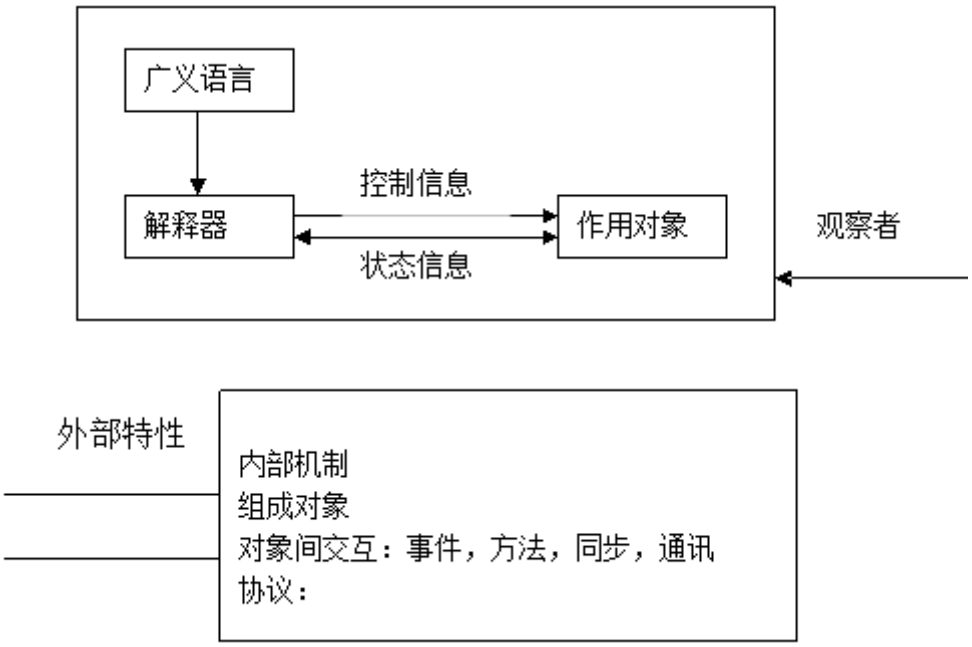
只有深入到这个抽象层次，我们才能很好的把握住系统的每一条线，每一个对象的意义。只

用从原型出发，我们才能把握住这个系统的实质所在，在干什么？他要表达什么？设计者为什么要这样想？最终极的想法是什么？这样，代码分析就变得简单明了，读代码就变成了是在印证猜想，修正方向。

Android 核心分析 之二 -------方法论探讨之概念空间篇

方法论探讨之概念空间篇

我们潜意识就不想用计算机的方式来思考问题，我们有自己的思维描述方式，越是接近我们思维描述方式，我们越容易接受和使用。各种计算机语言，建模工具，不外乎就是建立一个更接近人的思维方式的概念空间，再使用工具从该概念空间向另外一个概念空间映射，我称之为人性思维空间向 01 序列描述空间的一个映射。实现方面来看，系统就是一个翻译器，将机器性更加人性化的一种机制。大学计算机经典课“计算机体系结构”，其他的可以忘记，但是下面这个图不能忘记：



这个就是概念空间最本质的原型体现：作为观测者看到了什么？设计者给了观察者什么？给出的答案是外部特性。

（1）提供给观察者的概念空间是什么？

（2）内部特性的概念空间是什么？

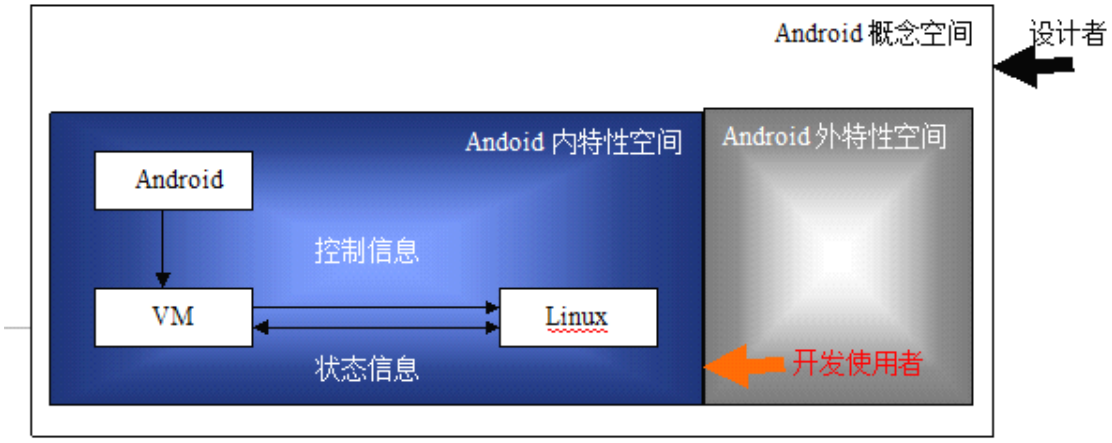
概念空间所表达的东西带有两个方面的缠绕：一面是人性自由，一面是物性制约（实时响应，系统资源的限制）。所以程序实现的概念空间是人性自由与特定计算机系统物性之间有一个折中，并且根据实际系统而采取某种动态的平衡。而这种平衡将会影响到系统架构，以及设计的思想。特别在手机这样的嵌入式系统中，这种矛盾和平衡无处不在，这种折中无处不在。而对系统的选取和采用，也就接受了某个方面的折中或某中即在的，也许是看不见的标准，及这样的标准有隐式和显式的。正因为如此，不管是工具的产生，新的平台的产生，都是计算机的物性向人性靠近的一个小台阶。一个新的思想的形成随即带来的新工具，新系统框架，新的体系结构。

如果设计者站的高度足够高，那么设计者一开始就会考虑到“我该给他们一个什么样的概念空间，甚至一个什么样的理念，让他们这个概念空间去建立自己的产品”，于是设计者就会开始主动的去建立概念空间，这个概念空间要表达的实际意义，概念空间应该有哪些内容构成，考虑概念空间的完备性和封闭性，考虑概念空间的边界，考虑从哪个基础上建立这个概念空间，考虑如何与概念空间外的实体进行交互，考虑系统的资源限制条件，考虑功能性构建的合理性，考虑机器系统与人的平衡问题。

我们在学习新系统时，首先映入眼帘的就是新概念。新名词，就如现在我们面临的 Android 大量的新名词，在程序员的世界都是从代码实践开始的，是从写应用开始去涉及。SDK 给了我们一个概念，我们就在这个概念框架下，使用 SDK 给我提供的函数接口，数据结构，初始化过程等，我们最初的接触到原型就是“HelloWorld”之类的 DEMO 程序，我们在 Hello world 上去使用各种不同的接口函数，对于应用程序员来讲,他说看到的系统就是系统调用接口，及其编程开发流程。实际上只要一使用这些接口,就不得不接受一系列的概念,只有在这种概念系统下,我们才能工作。但是,实际上我们却忽略了这样的概念系统的理解,只是在编程接口的这个狭窄的空间去理解系统.我们理解系统在形成理解概念的空间只是微小的一角，很少有资料来介绍这种概念系统的形成和理解,编程接口只是这个概念空间一个，对外部的一个表征。我们可以抽象起来,以接口,协议和行为,来描述系统的情况。SDK API 的实质向上层提供了一个语义接口，从而在层间实现了一个转义过程，同时又成为一个功能的集合体。但是我们很少这样跳出来看，我们到底是处于一种什么样的概念空间，SDK 除了调用接口外，还给了我们怎样一种整体概念？目标系统的基本构架在本质上的东西就是一个概念系统到另一个概念系统的映射。让我们大脑理解的概念系统映射到计算机能实现的概念域的一个映射。我们假定这个概念域 E,机器能够理解的概念域为 M,我们的软件工程要做的事情实质就是：EàM 领域的一个映射过程。

为什么要在宏观上把握这些概念呢，显然有我的目的，理解概念空间是理解设计者意图的一个重要途径。设计者要想给开发者提供什么，设计者想要提供给最终用户什么。我们需要站在高处看待系统明白设计者意图。

Android 的实质还是一套管理手机硬件系统的软件，这个话讲起来没有多大意义,计算机操作系统本质都是如此，Android 是 Google 云计算计划的一部分，我们修正成：Android 建立的本质就是让计算机成为我的云接入移动智能终端。作为硬件管理软件，Android 提供概念空间内涵实质上泛操作系统内涵，我们的理解可以从泛操作系统概念空间映射到 Android 系统中去。而作为云计算的一部分的内容，我们可以云计算的概念入手去研究 Andoird。

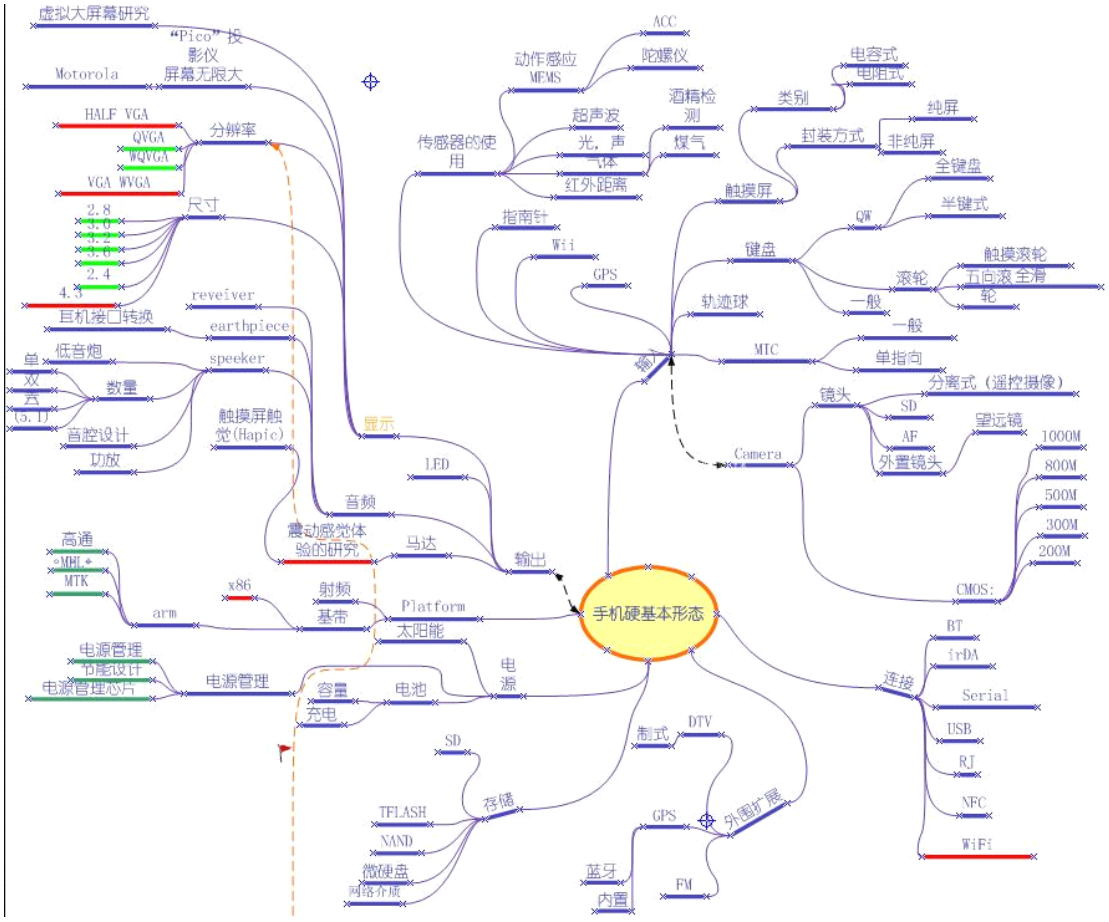


Android 是什么 之三------手机之硬件形态

手机硬件形态

本节可能与 Android 无关，但是 Android 系统现在这个阶段更多的是移动终端形态的开发平台，本节给出了 Android 背后的工作-Android 管理的硬件是什么，Android 的本质就是要管理好这些硬件部分，为用户提供一个体验更好，速度更快的智能移动终端。对手机硬件形态的认识是要让我们对手机硬件组成有个感性的认识，让程序员知道系统中的代码是管理那一部分的，即我们堆砖头的目的是什么，让思维有一个伸展。

为了对手机这类嵌入式系统有一个较为深入的了解，我制作了如下的手机硬件结构思维导图，在这张图上我们可以看到组成手机硬件的有哪些，初步了解到手机管理平台为什么要那么多的管理框架和层次，从最底层理解 Android 设计者的设计意图，这个思维导图其实只是示意图。



我们知道手机这种嵌入式系统，硬件架构最简单描述的描述为：

应用处理器+Modem+射频

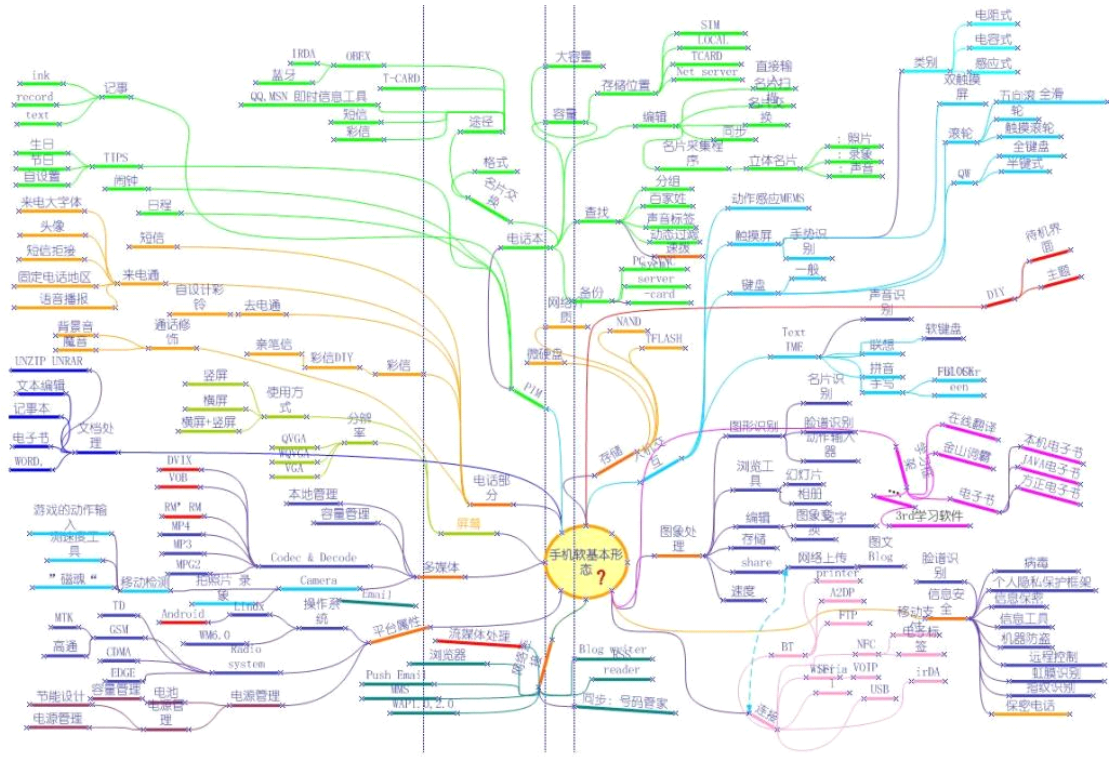
对于应用处理器而言，对设计者最为本质的描述为输入输出，而对于移动终端设备电源管理，连接机制，多媒体又是很重要的考虑环节，而这些环节都会在软件平台上有所体现。

Android 核心分析之四 ---手机的软件形态

手机的软件形态

上节我给出了手机的硬件树，本节将给出手机软件形态树。主要突出手机软件涵盖的内容。通过该思维导图，我们可以看到手机软件所涉及到的方方面面，Android 所涉及到的内容也不会超过下面所示太多，这个也是 Andoid 系统外特性空间所要展示的，这个也是 Android 设计者需要考虑管理的大部分内容，通过

下面的整理，我们可以让我们的思维更加贴近 Android 设计意图，从而更深入的了解 Android 中各种组成的由来，这个就是前面讲到的分析思想之一从退到源头出发，从思考最终极的问题开始。

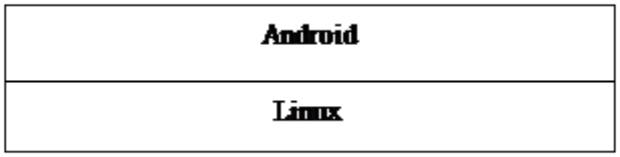


Android 核心分析 之五 -----基本空间划分

基本空间划分

Google 给了我们一张系统架构图，在这张图上我们可以看到 Android 的大体框架组成。

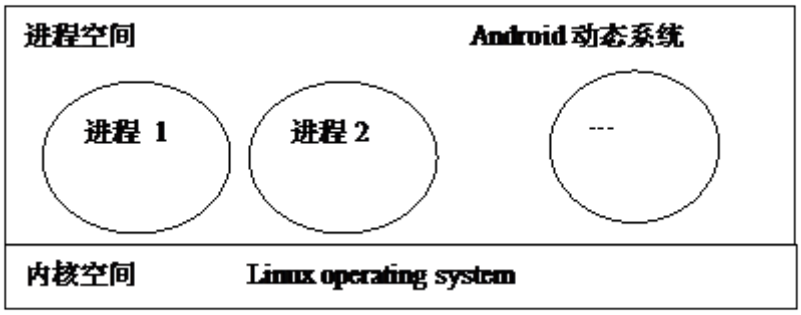
从上图可以看到：Android Applications,Application Framework,Dalvik Virtual Machine,Linux。如果将 Android 泛化，我们可以将系统划分成两部分：



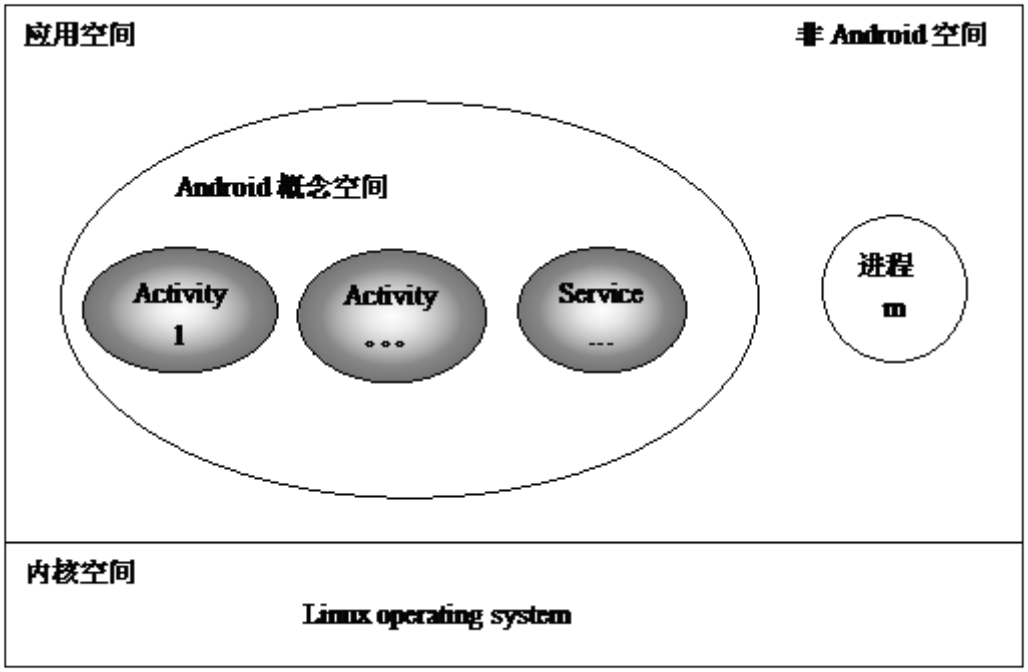
但是为了研究的方便我们先看最为本质的三层，上面是 Android，中间叫 Dalvik 虚拟机，下面叫 Linux。

虽然上两层都包含在 Android 中，但是为了理解的方便或者从实用主义出发，我还是将虚拟机这次给分开出来，因为我研究的对象是 Android 的手机系统相关部分，对于虚拟机我们不做太深入的研究。

e: pre;"> 从上面我们可以看到这个系统静态的划分成这样的三层。但是从动态运行逻辑上不是这样划分的，所以空间的划分是一个有趣的概念。我们从操作系统的角度看，Android 就是一堆 Linux 应用的集合。从 Linux 角度看到的空间划分：进程空间和内核空间。从 Android 的应用对应着 Linux 的一个个进程。

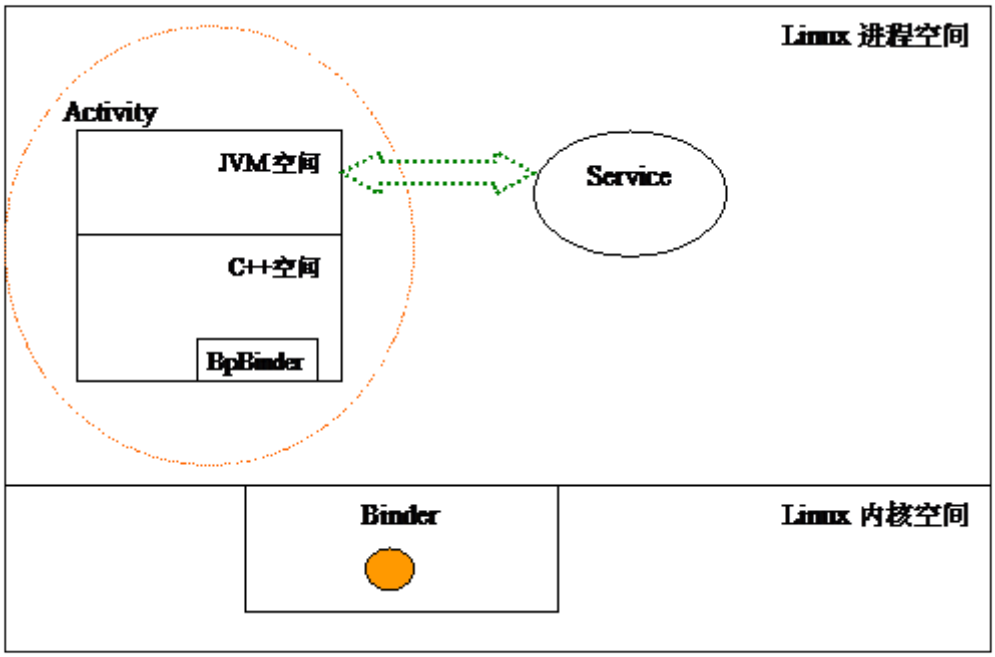


Andoid 中包含一个 Java 虚拟机，虚拟机是运行在 Linux 之上的，Android 构建在 JVM 之上，从 Android 动态运行逻辑上我们需要将 Android 划分成 Android 空间和非 Android 空间。在 Andoid 系统中我们面对的是 Andoid 概念空间，而不是 Linux 进程了，在 Andoid 概念空间中已经没有了 Lliux 进程的概念，而是 Service，proxy，Activity，provider 等。



至于虚拟机 JVM，我们只需要知道 JVM 是 Dalvik VM（虚拟机）这是一个专为嵌入式设备打造的ＪＡＶＡ虚拟机，是一个有着自己的 code-byte 和格式的可以在嵌入式设备上高效运行的 Java 虚拟机。

为了研究的深入，我们还是需要涉及到 JNI Native 部分。在这个分类中我将 JVM 分为 JVM 空间和 C++空间。



Android 应用的开发者是工作在 Android 外特性概念空间的，这里没有了 Linux 的一点气息，Android 构建的外特性空间概念包含了：Activity,Provider,Interface,Events,Provider，Service 等。至于 JVM 空间和 C++空间的划分是为了研究 Android 核心的描述而提出的，我们在做 Android 系统开发时，常常需要修改到 JNI 的 Native 部分。后面我将用较多的篇幅来深入阐述这个部分。

Android 核心分析 之六 -----IPC 框架分析 Binder，Service，Service manager

IPC 框架分析 Binder，Service，Service manager

我首先从宏观的角度观察 Binder,Service,Service Manager，并阐述各自的概念。从 Linux 的概念空间中，Android 的设计 Activity 托管在不同的的进程，Service 也都是托管在不同的进程，不同进程间的 Activity,Service 之间要交换数据属于 IPC。Binder 就是为了 Activity 通讯而设计的一个轻量级的 IPC 框架。

在代码分析中，我发现 Android 中只是把 Binder 理解成进程间通讯的实现，有点狭隘，而是应该站在公共对象请求代理这个高度来理解 Binder，Service 的概念，这样我们就会看到不一样的格局，从这个高度来理解设计意图，我们才会对 Android 中的一些天才想法感到惊奇。从 Android 的外特性概念空间中，我们看不到进程的概念，而是 Activity，Service，AIDL，INTENT。一般的如果我作为设计者，在我们的根深蒂固的想法中，这些都是如下的

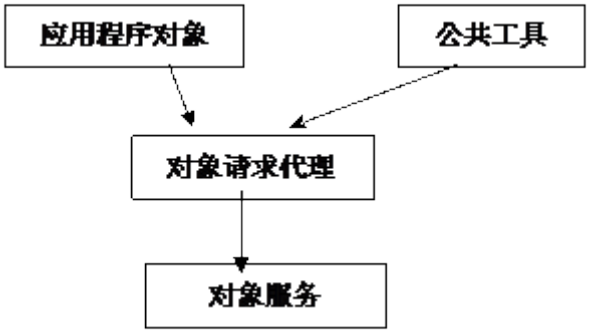
C/S 架构，客户端和服务端直接通过 Binder 交互数据，打开 Binder 写入数据，通过 Binder 读取数据，通讯就可以完成了。

该注意到 Android 的概念中，Binder 是一个很低层的概念，上面一层根本都看不到 Binder，而是 Activity 跟一个 Service 的对象直接通过方法调用，获取服务。

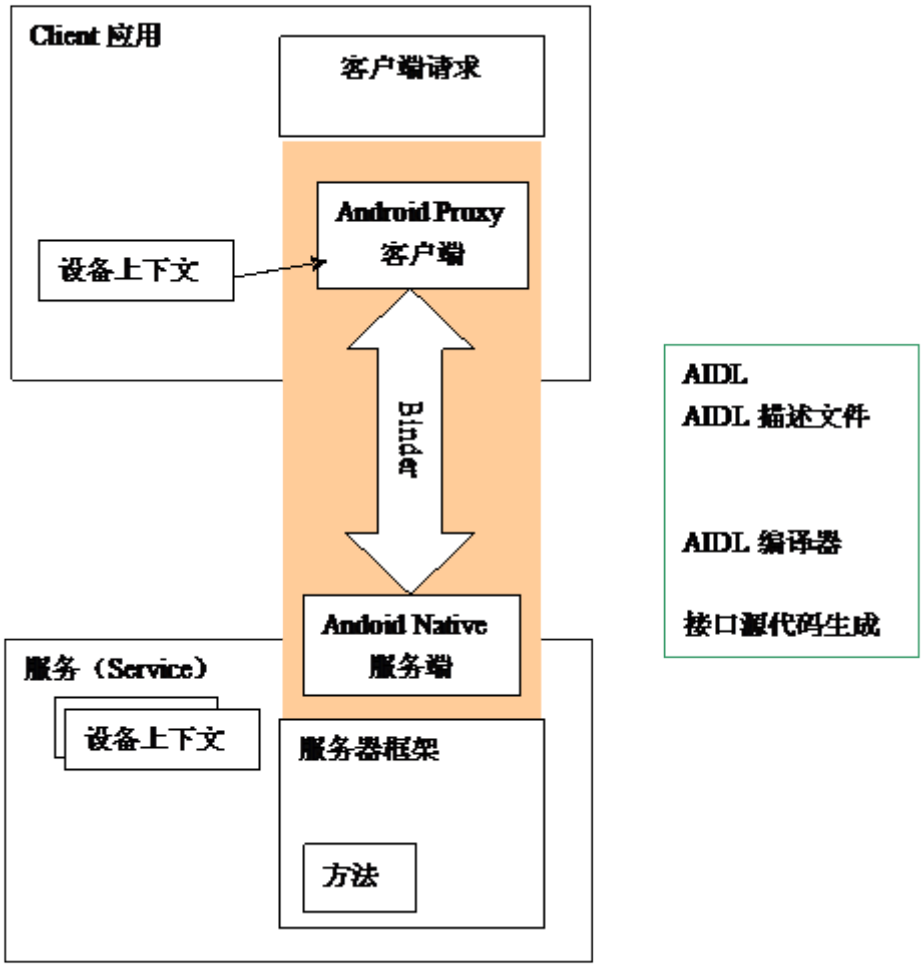
这个就是 Android 提供给我们的外特性：在 Android 中，要完成某个操作，所需要做的就是请求某个有能力的服务对象去完成动作，而无需知道这个通讯是怎样工作的，以及服务在哪里。所以 Andoid 的 IPC 在本质上属于对象请求代理架构，Android 的设计者用 CORBA 的概念将自己包装了一下，实现了一个微型的轻量级 CORBA 架构，这就是 Andoid 的 IPC 设计的意图所在，它并不是仅仅解决通讯，而是给出了一个架构，一种设计理念，这就是 Android 的闪光的地方。Android 的 Binder 更多考虑了数据交换的便捷，并且只是解决本机的进程间的通讯，所以不像 CORBA 那样复杂，所以叫做轻量级。

所以要理解 Android 的 IPC 架构，就需要了解 CORBA 的架构。而 CORBA 的架构在本质上可以使用下面图来表示：

在服务端，多了一个代理器，更为抽象一点我们可以下图来表示。



分析和 CORBA 的大体理论架构，我给出下面的 Android 的对象代理结构。



在结构图中，我们可以较为清楚的把握 Android 的ＩＰＣ包含了如下的概念：

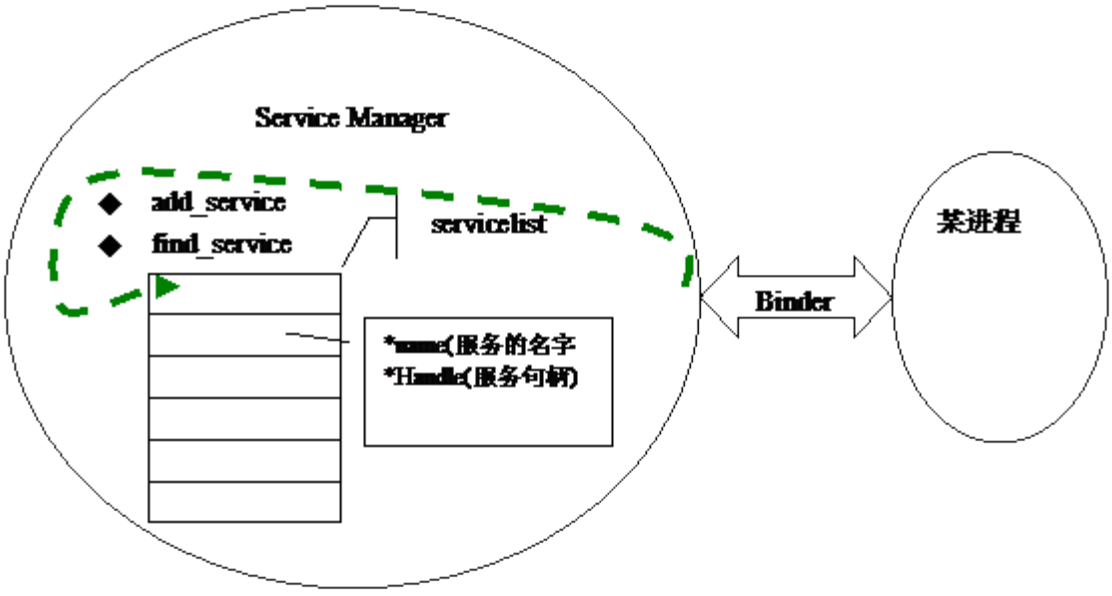
设备上下文什（ContextObject）

设备上下文包含关于客服端，环境或者请求中没有作为参数传递个操作的上下文信息，应用程序开发者用 ContextObject 接口上定义的操作来创建和操作上下文。

Android 代理：这个是指代理对象

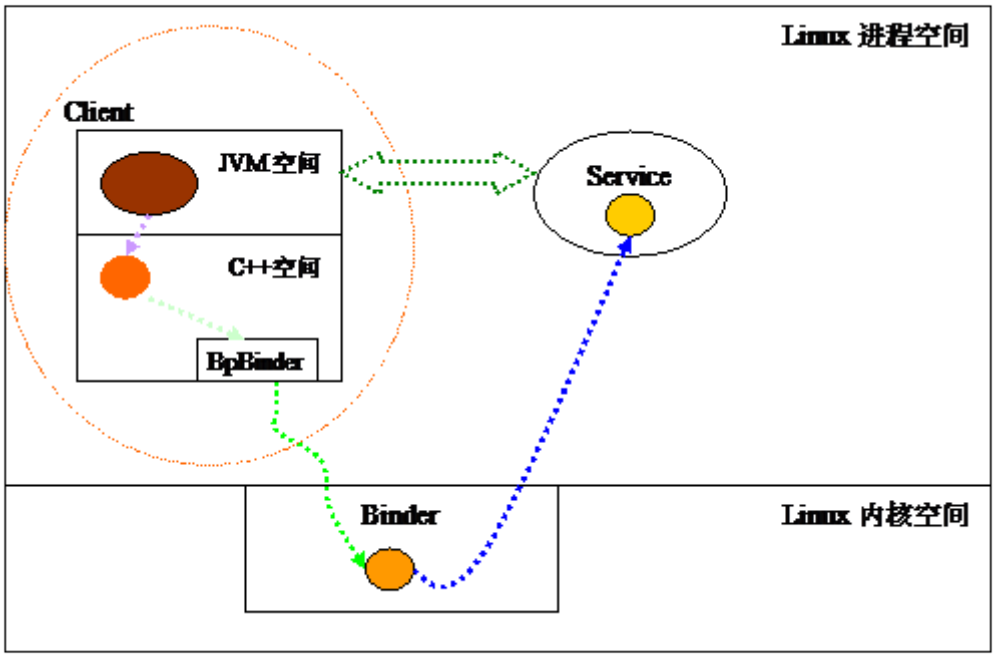
Binder Linux 内核提供的 Binder 通讯机制

Android 的外特性空间是不需要知道服务在那里，只要通过代理对象完成请求，但是我们要探究 Android 是如何实现这个架构，首先要问的是在 Client 端要完成云服务端的通讯，首先应该知道服务在哪里？我们首先来看看 Service Manger 管理了那些数据。Service Manager 提供了 add service,check service 两个重要的方法，并且维护了一个服务列表记录登记的服务名称和句柄。



Service manager service 使用 0 来标识自己。并且在初始化的时候，通过 binder 设备使用 BINDER_SET_CONTEXT_MGR ioctl 将自己变成了 CONTEXT_MGR。Svclist 中存储了服务的名字和 Handle，这个 Handle 作为 Client 端发起请求时的目标地址。服务通过 add_service 方法将自己的名字和 Binder 标识 handle 登记在 svclist 中。而服务请求者，通过 check_service 方法，通过服务名字在 service list 中获取到 service 相关联的 Binder 的标识 handle,通过这个 Handle 作为请求包的目标地址发起请求。

我们理解了 Service Manager 的工作就是登记功能，现在再回到ＩＰＣ上，客服端如何建立连接的？我们首先回到通讯的本质：IPC。从一般的概念来讲，Android 设计者在 Linux 内核中设计了一个叫做 Binder 的设备文件，专门用来进行 Android 的数据交换。所有从数据流来看 Java 对象从 Java 的 VM 空间进入到 C++空间进行了一次转换，并利用 C++空间的函数将转换过的对象通过 driver\binder 设备传递到服务进程，从而完成进程间的 IPC。这个过程可以用下图来表示。



这里数据流有几层转换过程。

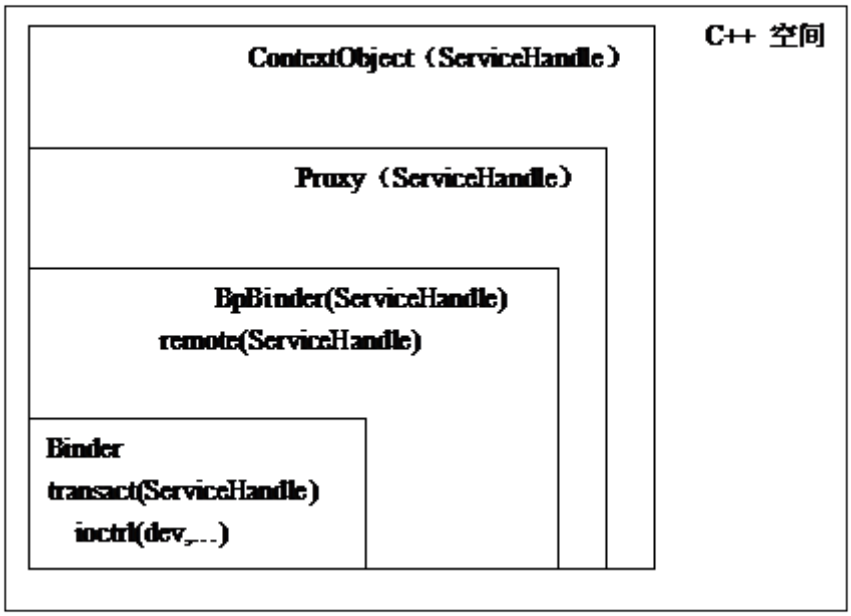
（1） 从JVM空间传到c++空间，这个是靠JNI使用ENV来完成对象的映射过程。

（2） 从c++空间传入内核Binder设备，使用ProcessState类完成工作。

（3） Service从内核中Binder设备读取数据。

Ａｎｄｒｏｉｄ设计者需要利用面向对象的技术设计一个框架来屏蔽掉这个过程。要让上层概念空间中没有这些细节。Android 设计者是怎样做的呢？我们通过 c++空间代码分析，看

到有如下空间概念包装(ProcessState@(ProcessState.cpp)



在 ProcessState 类中包含了通讯细节，利用 open_binder 打开 Linux 设备 dev\binder, 通过 ioctrl 建立的基本的通讯框架。利用上层传递下来的 servicehandle 来确定请求发送到那个 Service。通过分析我终于明白了 Bnbinder，BpBinder 的命名含义，Bn-代表 Native，而 Bp 代表 Proxy。一旦理解到这个层次，ProcessState 就容易弄明白了。

下面我们看 JVM 概念空间中对这些概念的包装。为了通篇理解设备上下文，我们需要将 Android VM 概念空间中的设备上下文和 C++空间总的设备上下文连接起来进行研究。

为了在上层使用统一的接口，在 JVM 层面有两个东西。在 Android 中，为了简化管理框架，引入了 ServiceManger 这个服务。所有的服务都是从 ServiceManager 开始的，只用通过 Service Manager 获取到某个特定的服务标识构建代理 IBinder。在 Android 的设计中利用 Service Manager 是默认的 Handle 为 0，只要设置请求包的目标句柄为 0，就是发给 Service Manager 这个 Service 的。在做服务请求时，Android 建立一个新的 Service Manager Proxy。Service Manager Proxy 使用 ContexObject 作为 Binder 和 Service Manager Service（服务端）进行通讯。

我们看到 Android 代码一般的获取 Service 建立本地代理的用法如下：

IXXX　mIxxx=IXXXInterface.Stub.asInterface(ServiceManager.getService("xxx"));

例如：使用输入法服务：

IInputMethodManager mImm=

IInputMethodManager.Stub.asInterface(ServiceManager.getService("input_method"));

这些服务代理获取过程分解如下：

（1） 通过调用 GetContextObject 调用获取设备上下对象。注意在 AndroidJVM 概念空间的 ContextObject 只是　　与 Service Manger Service 通讯的代理 Binder 有对应关系。这个跟 c++概念空间的 GetContextObject 意义是不一样的。

注意看看关键的代码

BinderInternal.getContextObject（）　　@BinderInteral.java

NATIVE JNI:getContextObject()　　@android_util_Binder.cpp

Android_util_getConextObject　　@android_util_Binder.cpp

ProcessState::self()->getCotextObject(0)　@processState.cpp

getStrongProxyForHandle(0)　@

NEW BpBinder(0)

注意 ProcessState::self()->getCotextObject(0) @processstate.cpp，就是该函数在进程空间建立　了 ProcessState 对象，打开了 Binder 设备 dev\binder,并且传递了参数 0，这个 0 代表了与 Service Manager 这个服务绑定。

（2）通过调用 ServiceManager.asInterface（ContextObject）建立一个代理 ServiceManger。

mRemote= ContextObject(Binder)

这样就建立起来 ServiceManagerProxy 通讯框架。

(3)客户端通过调用 ServiceManager 的 getService 的方法建立一个相关的代理 Binder。

ServiceMangerProxy.remote.transact(GET_SERVICE)

IBinder=ret.ReadStrongBinder() -》这个就是 JVM 空间的代理 Binder

JNI Navite: android_os_Parcel_readStrongBinder() @android_util_binder.cpp

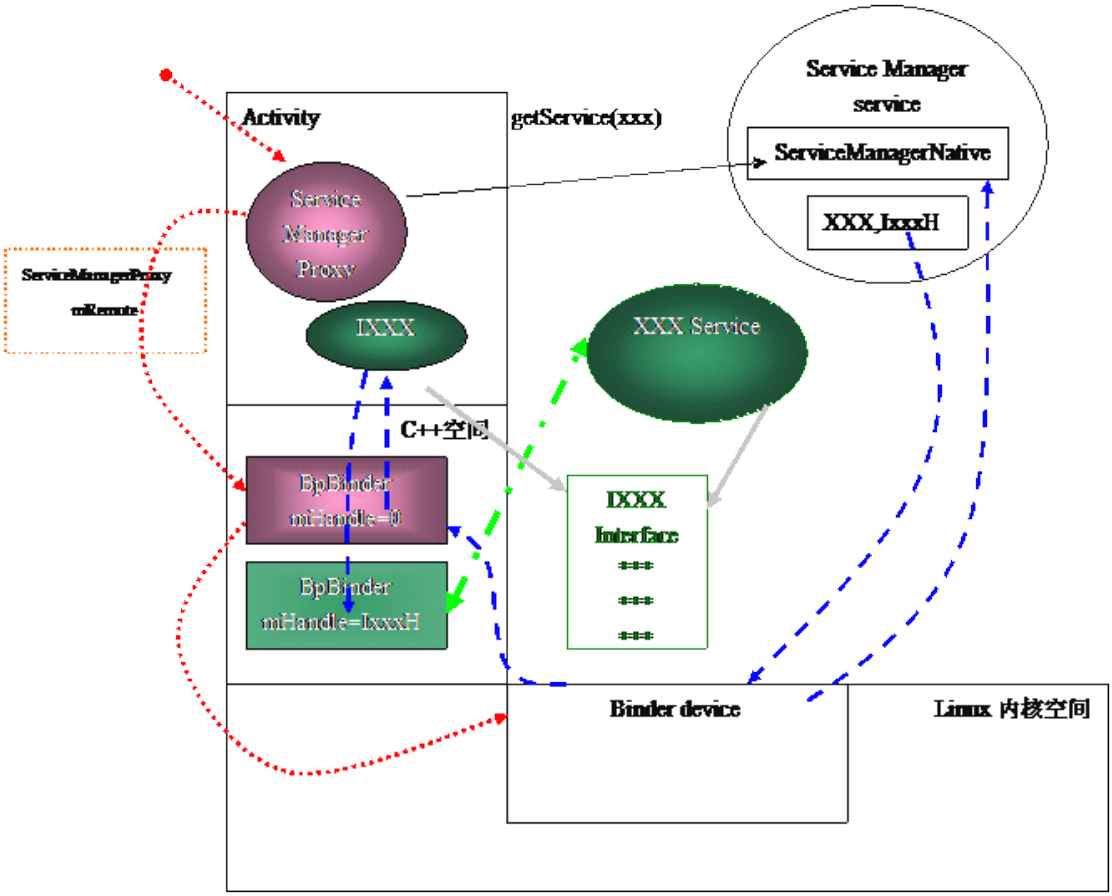
Parcel->readStrongBinder() @pacel.cpp

unflatten_binder @pacel.cpp

getStrongProxyForHandle(flat_handle)

NEW BpBinder(flat_handle)-》这个就是底层 c++空间新建的代理 Binder。

整个建立过程可以使用如下的示意图来表示：



Activity 为了建立一个 IPC，需要建立两个连接：访问 Servicemanager Service 的连接，IXXX 具体 XXX Service 的代理对象与 XXXService 的连接。这两个连接对应 c++空间 ProcessState 中 BpBinder。对 IXXX 的操作最后就是对 BpBinder 的操作。由于我们在写一个 Service 时，在一个 Package 中写了 Service Native 部分和 Service Proxy 部分，而 Native 和 Proxy 都实现相同的接口：IXXX Interface,但是一个在服务端，一个在客服端。客户端调用的方式是使用 remote->transact 方法向 Service 发出请求，而在服务端的 OnTransact 中则是处理这些请求。所以在 Android Client 空间就看到这个效果：只需要调用代理对象方法就达到了对远程服务的调用目的，实际上这个调用路径好长好长。

我们其实还一部分没有研究，就是同一个进程之间的对象传递与远程传递是区别的。同一个进程间专递服务地和对象，就没有代理 BpBinder 产生，而只是对象的直接应用了。应用程序并不知道数据是在同一进程间传递还是不同进程间传递，这个只有内核中的 Binder 知道，所以内核 Binder 驱动可以将 Binder 对象数据类型从 BINDER_TYPE_BINDER 修改为 BINDER_TYPE_HANDLE 或者 BINDER_TYPE_WEAK_HANDLE 作为引用传递。

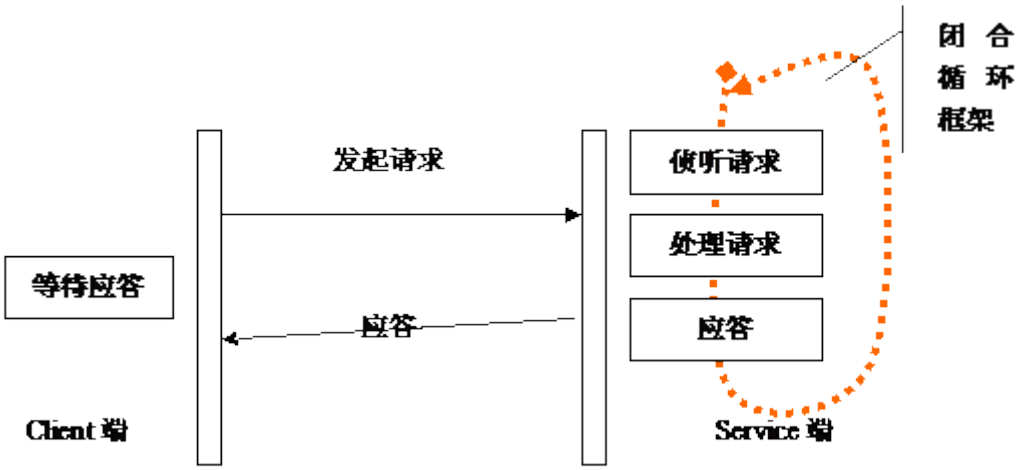
Android 核心分析 之七------Service 深入分析

Service 深入分析
上一章我们分析了 Android IPC 架构,知道了 Android 服务构建的一些基本理念和原理，本章我们将深入分析 Android 的服务。Android 体系架构中三种意义上服务：

Native 服务
Android 服务
Init 空间的服务，主要是属性设置，这个ＩＰＣ是利用Ｓｏｃｋｅｔ来完成的，这个我将在另外一章来讨论。
Navite 服务，实际上就是指完全在 C++空间完成的服务，主要是指系统一开始初始化，通过 Init.rc 脚本起来的服务，例如 Service Manger service,Zygote service,Media service , ril_demon service 等。

Android 服务是指在 JVM 空间完成的服务，虽然也要使用 Navite 上的框架，但是服务主体存在于 Android 空间。Android 是二阶段初始（Init2）初始化时建立的服务。

1 Service 本质结构
我们还是从 Service 的根本意义分析入手，服务的本质就是响应客户端请求。要提供服务，就必须建立接收请求，处理请求，应答客服端的框架。我想在 Android Service 设计者也会无时不刻把这个服务本质框图挂在脑海中。从程序的角度，服务一定要存在一个闭合循环框架和请求处理框架



分析清楚服务框就必须弄清楚以下的机制及其构成。

（1）闭合循环结构放置在哪里？

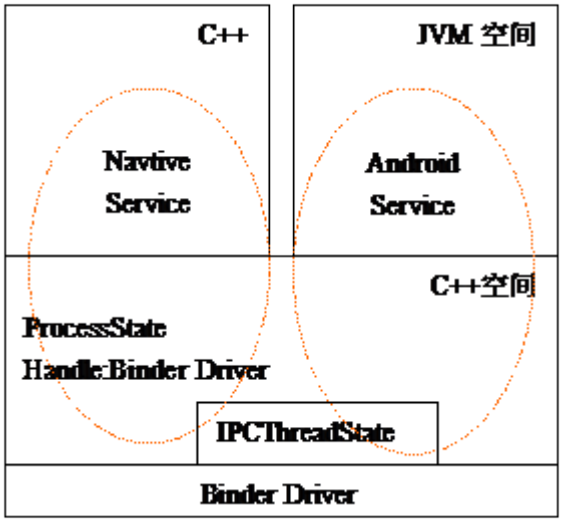
（2）处理请求是如何分发和管理？

（3）处理框架是如何建立的？

（4）概念框架是如何建立的？

2 Service 基本框架分析

Android 设计中，Native Service 和 Android Service 采用了同一个闭合循环框架。这个闭合循环框架放置在 Native 的 C++ 空间中，,ProcessState@ProcessState.cpp 和 IPCThreadState@IPCThreadState.cpp 两个类完成了全部工作。

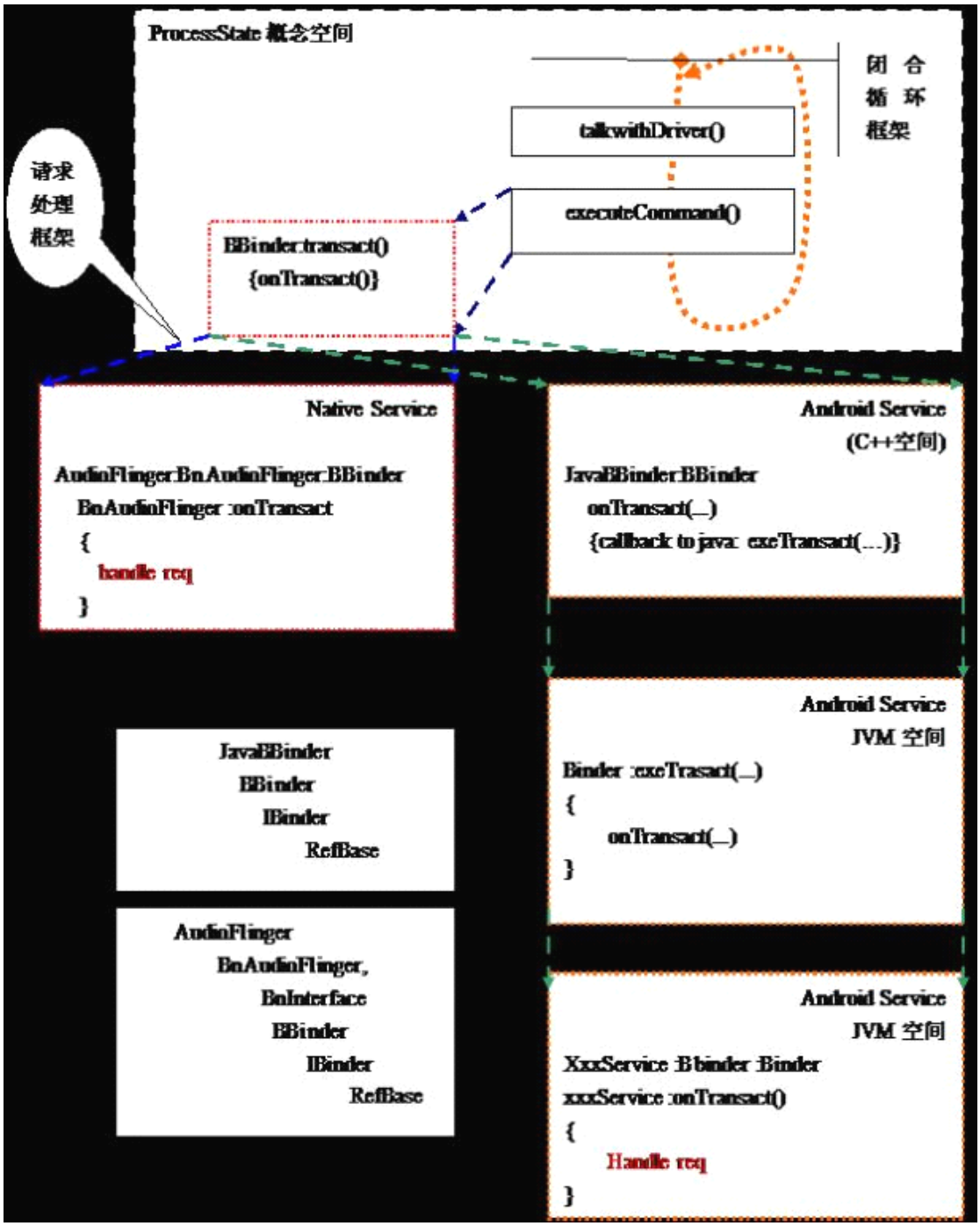


在服务框架中，ProcessState 是公用的部分，这个公用部分最主要的框架就是闭合循环框架和接收到从 Binder 来的请求后的处理框架。我们将服务框架用 ProcessSate 来表示,简言之：

（1） addservice

（2） 建立闭合循环处理框架。

```
int main(int argc, char** argv)

{

sp<ProcessState> proc(ProcessState::self());

addService(String16("xxx0"), new xxx0Service());

addService(String16("xxx1"), new xxx1Service());

…

ProcessState::self()->startThreadPool();

IPCThreadState::self()->joinThreadPool();//闭合循环框架

}
```



2.1 Native Service

Native Service 是在系统 Init 阶段通过 Init.rc 脚本建立的服务。

首先来看看一个例子 mediaserver@main_mediaserver.cpp 的建立过程。

```
int main(int argc, char** argv)
{
    sp<ProcessState> proc(ProcessState::self());
    sp<IServiceManager> sm = defaultServiceManager();
    LOGI("ServiceManager: %p", sm.get());
    AudioFlinger::instantiate();
    MediaPlayerService::instantiate();
    CameraService::instantiate();
    AudioPolicyService::instantiate();
    ProcessState::self()->startThreadPool();
    IPCThreadState::self()->joinThreadPool();
}
```

我们将代码向下展开了一层，更能看到事物的本质。

```
int main(int argc, char** argv)
{
sp<ProcessState> proc(ProcessState::self());
sp<IServiceManager> sm = defaultServiceManager();
defaultServiceManager()->addService(String16("media.audio_flinger"), new AudioFlinger());
```

…

ProcessState::self()->startThreadPool();

IPCThreadState::self()->joinThreadPool();

}

（1）服务进程建立了 ProcessState 对象，并将给对象登记在进程的上下文中。

（2）建立一个新 AudioFlinger 对象，并将对象登记 Service Manager Service 中。

（3）开始就收请求，处理请求，应答这个循环闭合框架。

2.2 Android Service

Androids service 是系统二阶段（Init2）初始化时建立的服务。

Android 的所有服务循环框架都是建立 SystemServer@(SystemServer.java) 上。在 SystemServer.java 中看不到循环结构，只是可以看到建立了 init2 的实现函数，建立了一大堆服务，并 AddService 到 service Manager。

main() @ com/android/server/SystemServer

{

init1();

}

Init1()是在 Native 空间实现的（com_andoird_server_systemServer.cpp）。我们一看这个函数就知道了，原来这个闭合循环处理框架在这里：

init1->system_init() @System_init.cpp

在 system_init()我们看到了这个久违的循环闭合管理框架。

{

Call "com/android/server/SystemServer", "init2"

…..

ProcessState::self()->startThreadPool();

IPCThreadState::self()->joinThreadPool();

}

Init2()@SystemServer.java 中建立了 Android 中所有要用到的服务：

Entropy Service

Power Manager

Activity Manager

Telephony Registry

Package Manager

Account Manager

Content Manager

System Content Providers

Battery Service

Hardware Service

Alarm Manager

Init Watchdog

Sensor Service

Window Manager

Bluetooth Service

statusbar

Clipboard Service

Input Method Service

NetStat Service

Connectivity Service

Accessibility Manager

Notification Manager

Mount Service

Device Storage Monitor

Location Manager

Search Service

Checkin Service

Wallpaper Service

Audio Service

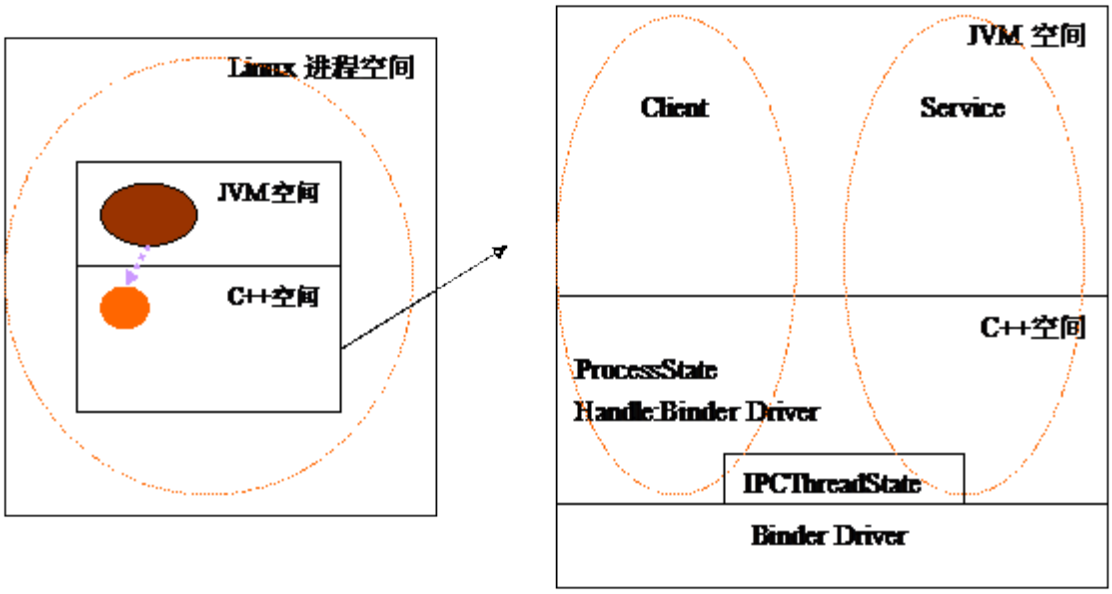
Headset Observer

Backup Service

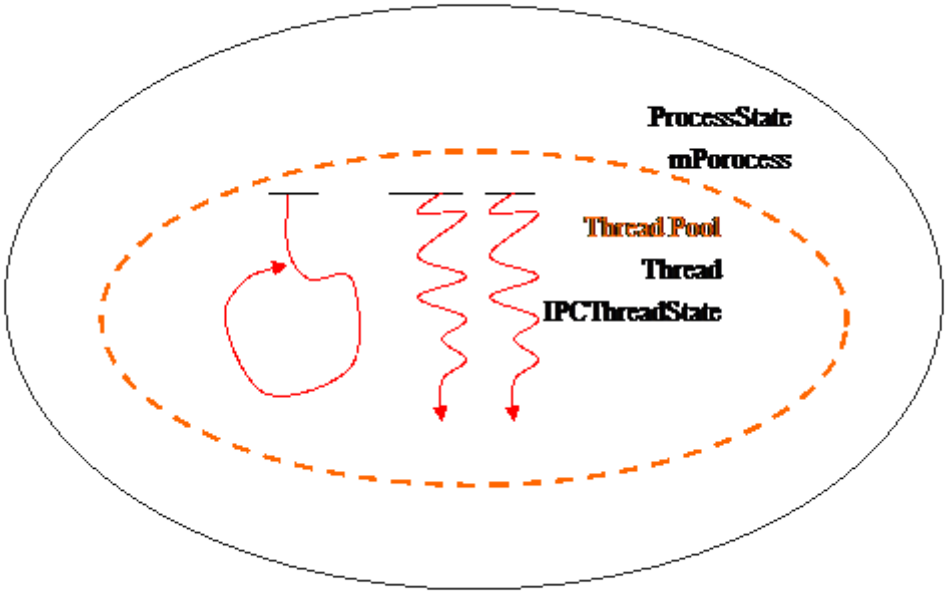
AppWidget Service

3 ProcessState 和 IPCThreadState

从宏观来讲，PocessState 及其 IPCThreadState 处于 IPC 与内核打交道包装层。前面的章节已经提到，下面我将更详细的分析。有关 IPC 的 c++空间的实现都是从 ProcessState 这个对象完成的。



我们可以得出如下的结论：不管 JVM 的 Binder 做了多么复杂的操作，最终还是需要利用 ProcessState 这个 c++空间的对象把数据传递给 Binder Driver，接收数据也是通过 ProcessState 这个对象完成，ProcessState 是所有 Binder IPC 必经的通道。



ProcessState 放置在全局变量 gProcess 中，每个进程只有一个 ProcessState 对象，负责打开 Binder 设备驱动，建立线程池等。而 IPCThreadState 每个线程有一个，IPCThreadState 实例登记在 Linux 线程程的上下文附属数据中，主要负责 Binder 数据读取，写入和请求处理框架。IPCThreadSate 在构造的时候，获取进程的 ProcessSate 并记录在自己的成员变量 mProcess 中,通过 mProcess 可以获取到 Binder 的句柄。

3.1 ProcessState 的生命周期

既然 ProcessState 是 Binder 通讯的基础，那么 Process 必须在 Binder 通讯之前建立。客户端，服务端都必须建立。由于现在重点讨论服务端，所以重心放置在服务端。在 Android 体系中有 c++空间的服务，JVM 空间的服务，这两类服务在本质上相同的，只是形式上不同，由于他们都是建立在ＰｒｏｃｅｓｓＳｔａｔｅ这个基础上，所以在形式上不同就仅仅表现在对 OnTransact 的回调处理的不同。

Native Service

我们直接可以看到使用 sp<ProcessState> proc(ProcessState::self())，建立建立 ProcessState，一旦调用 ProcessState 就建立了，并且这个 self 将 ProcessSate 登记在全局变量中。

Android Service

建立 Android Service 服务 system_init @System_init.cpp 中我们可以看到相同的结构。有一点不同的是所有的 Android Service 都运行在一个进程中：systemsever 进程。

3.2 Binder Driver 包装 @IPCThreadState

ProcessSate 构造的时候，使用 open_binder 打开/driver/binder，并将句柄记录在 mDriverFD，在 ProcessState 中并不使用这个句柄，真正使用这个 Binder 设备句柄的是 IPCThreadState，所有关于 Binder 的操作放置在 IPCThreadState 中：

(1) 读取\写入： talkWithDriver （） @IPCThreadState 对 ioctl(mProcess->mDriverFD, BINDER_WRITE_READ, &bwr)进行包装。

(2)请求处理：executeCommand(...)@ IPCThreadState

(3)循环结构：joinThreadPool()

joinThreadPool()

{

While(1){

talkWithDriver(...)

...

executeCommand(...)

}

}

Android 核心分析 之八-----Android 启动过程详解

Android 启动过程详解

Android 从 Linux 系统启动有 4 个步骤；

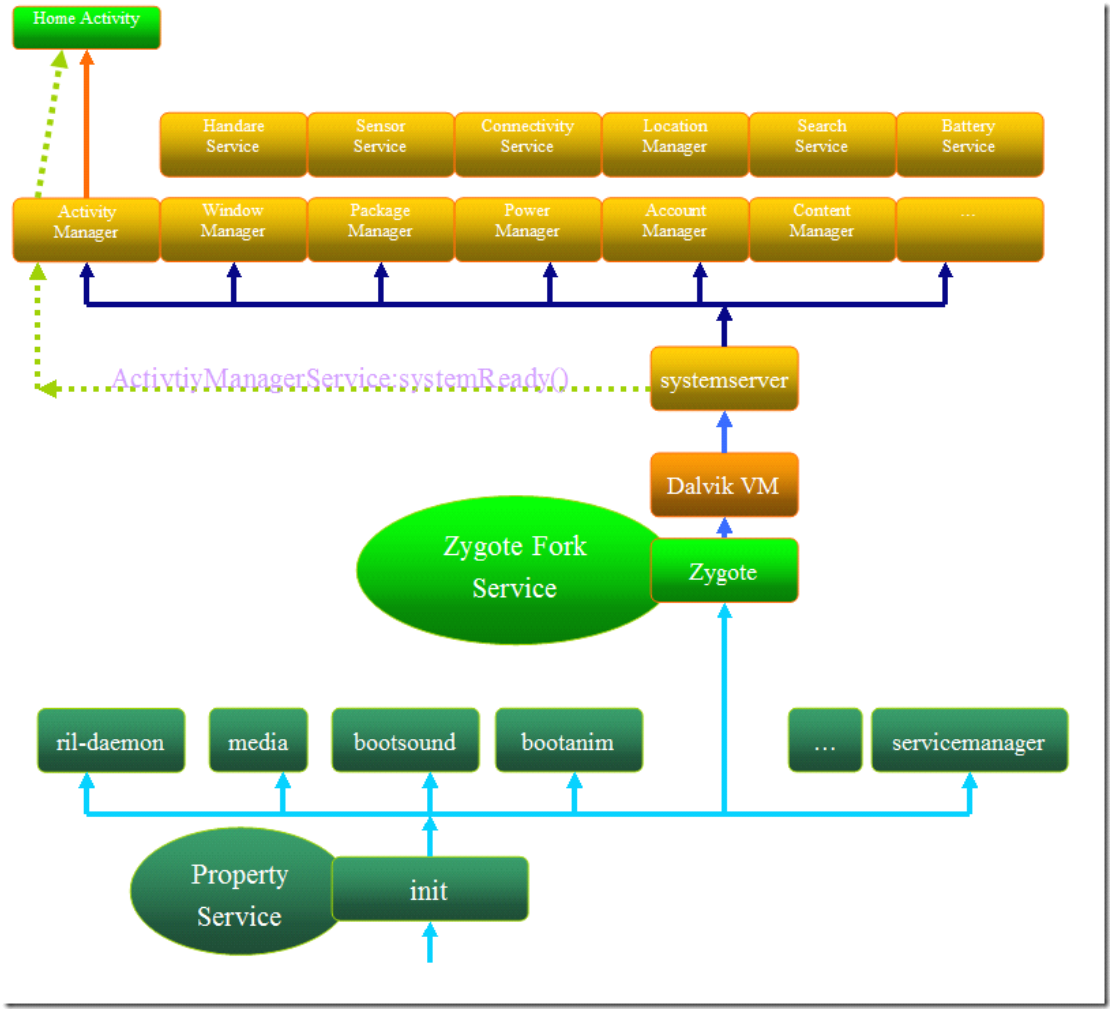
(1) init 进程启动

(2) Native 服务启动

(3) System Server，Android 服务启动

(4) Home 启动

总体启动框架图如：

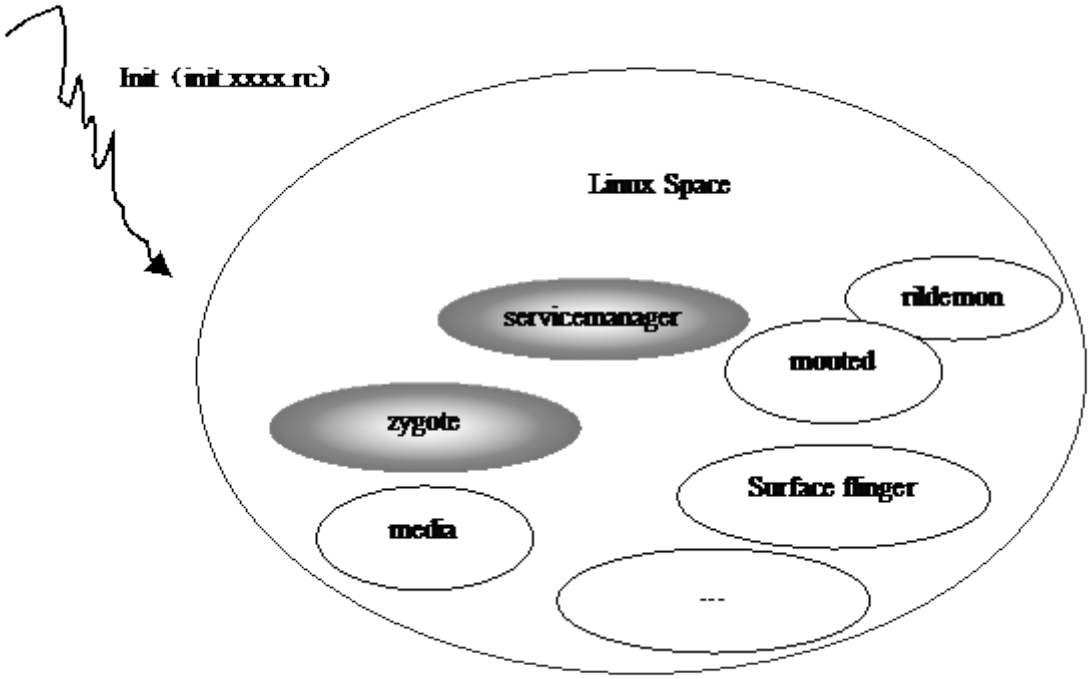


第一步：initial 进程(system\core\init)

init 进程，它是一个由内核启动的用户级进程。内核自行启动（已经被载入内存，开始运行，并已初始化所有的设备驱动程序和数据结构等）之后，就通过启动一个用户级程序 init 的方式，完成引导进程。init 始终是第一个进程.

Init.rc

Init.marvell.rc



Init 进程一起来就根据 init.rc 和 init.xxx.rc 脚本文件建立了几个基本的服务：

servicemanamger
zygote
。。。

最后 Init 并不退出，而是担当起 property service 的功能。

1.1 脚本文件
init@System/Core/Init

Init.c： parse_config_file(Init.rc)

@parse_config_file(Init.marvel.rc)

解析脚本文件：Init.rc 和 Init.xxxx.rc(硬件平台相关)

Init.rc 是 Android 自己规定的初始化脚本(Android Init Language, System/Core/Init/readme.txt)

该脚本包含四个类型的声明：

Actions
Commands
Services
Options.
1.2 服务启动机制

我们来看看 Init 是这样解析.rc 文件开启服务的。

（1）打开.rc 文件，解析文件内容@ system\core\init\init.c

将 service 信息放置到 service_list 中。@ system\core\init parser.c

（2）restart_service()@ system\core\init\init.c

service_start

execve(…).建立 service 进程。

第二步 Zygote

Servicemanager 和 zygote 进程就奠定了 Android 的基础。Zygote 这个进程起来才会建立起真正的 Android 运行空间，初始化建立的 Service 都是 Navtive service.在.rc 脚本文件中 zygote 的描述：

service zygote /system/bin/app_process -Xzygote /system/bin --zygote --start-system-server

所以 Zygote 从 main(…)@frameworks\base\cmds\app_main.cpp 开始。

(1) main(…)@frameworks\base\cmds\app_main.cpp

建立 Java Runtime
runtime.start("com.android.internal.os.ZygoteInit", startSystemServer);

(2) runtime.start@AndroidRuntime.cpp

建立虚拟机
运行：com.android.internal.os.ZygoteInit：main 函数。

（3）main()@com.android.internal.os.ZygoteInit//正真的 Zygote。

registerZygoteSocket();//登记 Listen 端口
startSystemServer();
进入 Zygote 服务框架。

经过这几个步骤，Zygote 就建立好了，利用 Socket 通讯，接收 ActivityManangerService 的请求，Fork 应用程序。

第三步 System Server

startSystemServer@com.android.internal.os.ZygoteInit 在 Zygote 上 fork 了一个进程：com.android.server.SystemServer.于是 SystemServer@(SystemServer.java）就建立了。Android 的所有服务循环框架都是建立 SystemServer@(SystemServer.java）上。在 SystemServer.java 中看不到循环结构，只是可以看到建立了 init2 的实现函数，建立了一大堆服务，并 AddService 到 service Manager。

main() @ com/android/server/SystemServer

```
{
  init1();
}
```

Init1()是在 Native 空间实现的（com_andoird_server_systemServer.cpp）。我们一看这个函数就知道了， init1->system_init() @System_init.cpp

在 system_init()我们看到了循环闭合管理框架。

```
{
  Call "com/android/server/SystemServer", "init2"
  …..
  ProcessState::self()->startThreadPool();
        IPCThreadState::self()->joinThreadPool();
}
```

init2()@SystemServer.java 中建立了 Android 中所有要用到的服务。

这个 init2（）建立了一个线程，来 New Service 和 AddService 来建立服务

第三步 Home 启动
在 ServerThread@SystemServer.java 后半段，我们可以看到系统在启动完所有的 Android 服务后，做了这样一些动作：

（1） 使用 xxx.systemReady()通知各个服务，系统已经就绪。

(2) 特别对于 ActivityManagerService.systemReady(回调)

Widget.wallpaper,imm(输入法)等 ready 通知。

Home 就是在 ActivityManagerService.systemReady() 通知的过程中建立的。下面是 ActivityManagerService.systemReady()的伪代码：

systemReady()@ActivityManagerService.java

resumeTopActivityLocked()

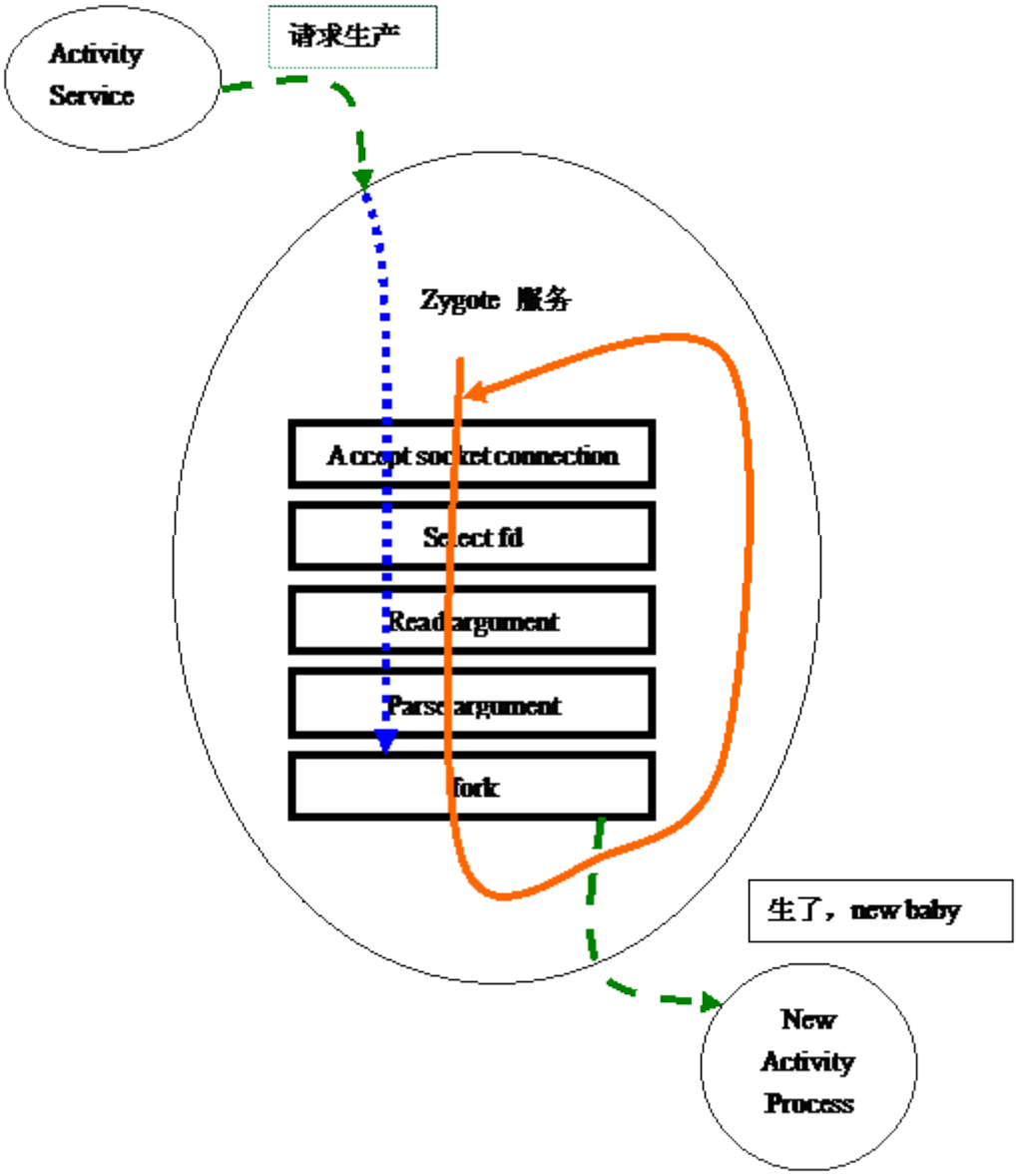
startHomeActivityLocked();//如果是第一个则启动 HomeActivity。

startActivityLocked（。。。）CATEGORY_HOME

Android 核心分析 之九-------Zygote Service

Zygote Service
在本章我们会接触到这两个单词：

Zygote [生物] 受精卵, 接合子, 接合体
Spawn：产卵
通过这两个单词，我们就可以大体知道 Zygote 是干什么的了，就是叫老母鸡下蛋。通过“Zygote”产出不同的子“Zygote”。从大的架构上讲，Zygote 是一个简单的典型 C/S 结构。其他进程作为一个客服端向 Zygote 发出”孵化”请求，Zygote 接收到命令就“孵化”出一个 Activity 进程来。



Zygote 系统代码组成及其调用结构：

Zygote.java

提供访问 Dalvik “zygote”的接口。主要是包装 Linux 系统的 Fork，以建立一个新的 VM 实例进程。

ZygoteConnection.java

Zygote 的套接口连接管理及其参数解析。其他 Actvitiy 建立进程请求是通过套接口发送命令参数给 Zygote。

ZygoteInit.java

Zygote 的 main 函数入口。

Zygote 系统代码层次调用

main()

startSystemServer()…

runSelectLoopMode()

Accept socket connection

Conntecion.RunOnce()

Read argument

folkAndSpecialize

folkAndSpecialize 使用 Native 函数 Dalvik_dalvik_system_Zygote_forkAndSpecialize

//native 的获取

dalvik\vm\native

//dalvik_system_Zygote.c

const DalvikNativeMethod dvm_dalvik_system_Zygote[] = {

{ "fork", "()I",

```
        Dalvik_dalvik_system_Zygote_fork },
    { "forkAndSpecialize",              "(II[II[[I)I",
        Dalvik_dalvik_system_Zygote_forkAndSpecialize },
    { "forkSystemServer",               "(II[II[[I)I",
        Dalvik_dalvik_system_Zygote_forkSystemServer },
    { NULL, NULL, NULL },
};
```

在这里我们就有了 Zygote 服务的全貌理解，也在 Code 中印证了。【应 yk_hu0621 修正】{由于 Android 中没有具体应用程序的入口，都是通过启动 Actvity 来启动相关的 Android 应用，而这个 Android 应用则对应着 Linux 进程，Activity 便 Host 在这个应用程序上。}

{原文：Activity 在本质上是个什么东西，就是一个 Linux 进程}

从分析中我们可以看到，Android 使用了 Linux 的 fork 机制。在 Linux 中 Fork 是很高效的。

一个 Android 的应用实际上一个 Linux 进程，所谓进程具备下面几个要素，

a.要有一段程序供该进程运行，程序是可以被多个进程共享的。

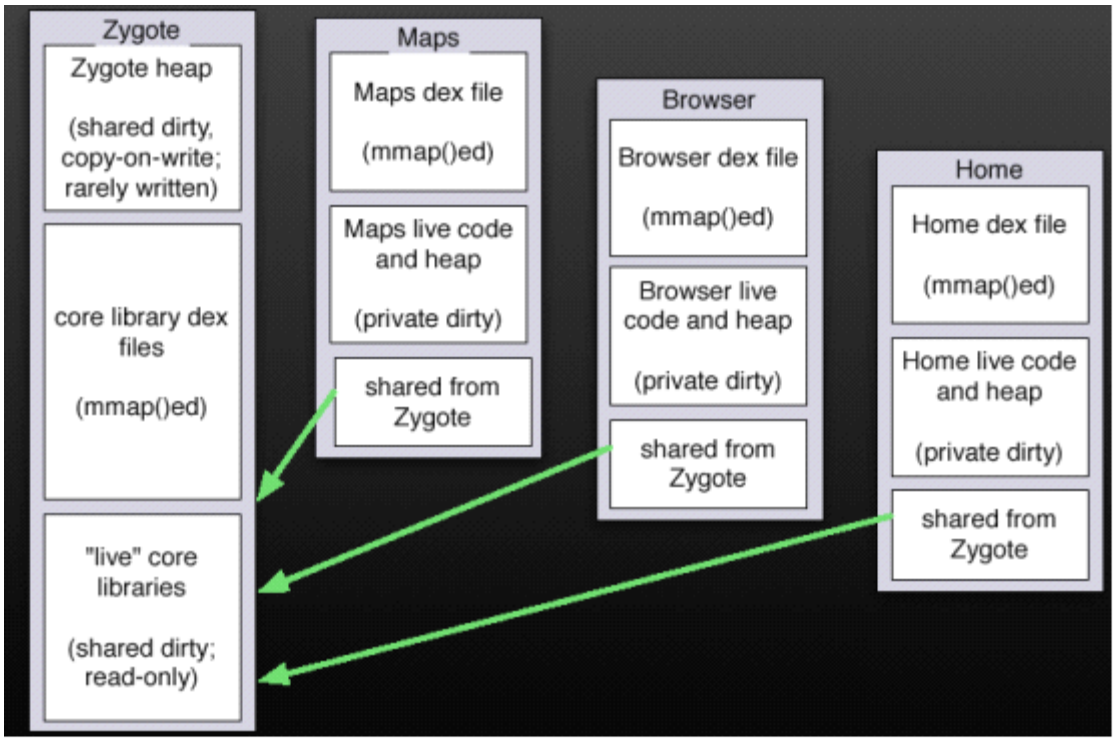
b..进程专用的系统堆栈空间。

c.进程控制块，在 linux 中具体实现是 task_struct

d.有独立的存储空间。

fork 创造的子进程复制了父亲进程的资源，包括内存的内容 task_struct 内容，在复制过程中，子进程复制了父进程的 task_struct，系统堆栈空间和页面表，而当子进程改变了父进程的变量时候，会通过 copy_on_write 的手段为所涉及的页面建立一个新的副本。所以只有子进程有改变变量时，子进程才新建了一个页面复制原来页面的内容，基本资源的复制是必须的，整体看上去就像是父进程的独立存储空间也复制了一遍。

再看看下面 Google 在讲解 Dalvik 虚拟机的图片，我们就大体有了 Android 系统中 Actvitiy 的实际映射状态有了基本的认识。



Android 核心分析 之十------Android GWES 之基本原理篇

Android GWES
基本框架篇

我这里的 GWES 这个术语实际上从 Microsoft 的 Window 上移植过来的，用 GWES 来表示 Android 的窗口事件系统不是那么准确，在 Android 中 Window 是个弱化了的概念，更多的表现在 View 这个概念上。在很大程度上，Android 的 View 的概念可以代替 Microsoft Window 这个概念，有点和 Microsof 暗中较劲的意味，你用过的概念我就偏不用，这个也是我以为的设计者意图。

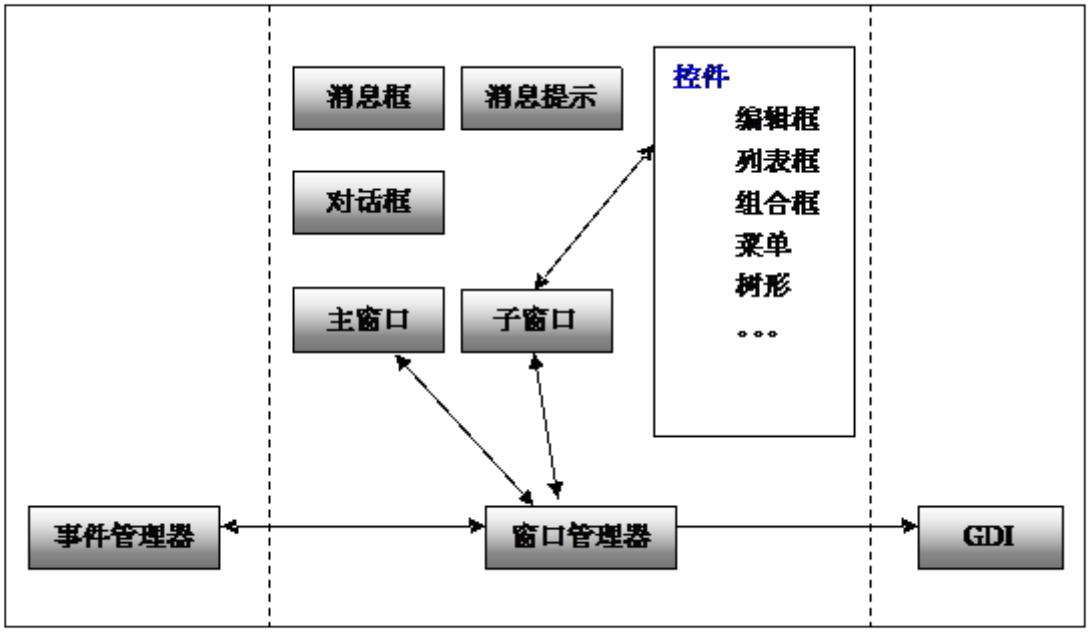
原始 GUI 基本框架

首先我们从 Android 的 SDK 外特性空间开始，在编写 Actvitiy 时，我们都是面对的处理函数：OnXXXX（），例如有按键按下就是 OnKeyDown 等，在这个过程中系统做了怎样的处理？要详细的理解这个过程，我们就需要理解 Andoid 的 View 管理，窗口系统，消息系统和输入系统。我们还是从最本质的地方开始，Android 作为一种嵌入式的图形用户界面系统，它的基本原理与一般 GUI 的原理是相同的，同时也是遵循 GWES(图形窗口事件系统)的一般规律，总体上 Android 就是管理用户输入和系统屏幕输出的一个系统。其实 GWES 这个名称更能体现 GUI 的基本实质要素：图形、窗口、事件。

1． 一般 GUI 的基本组成

GUI 的实现就是对上面提到的三个基本要素的管理，根据这这三个要素的特性及其涉及的范围，GUI 在总体上可以分为三部分：

事件管理器
窗口管理器
GDI(绘制与 GDI 逻辑对象管理)



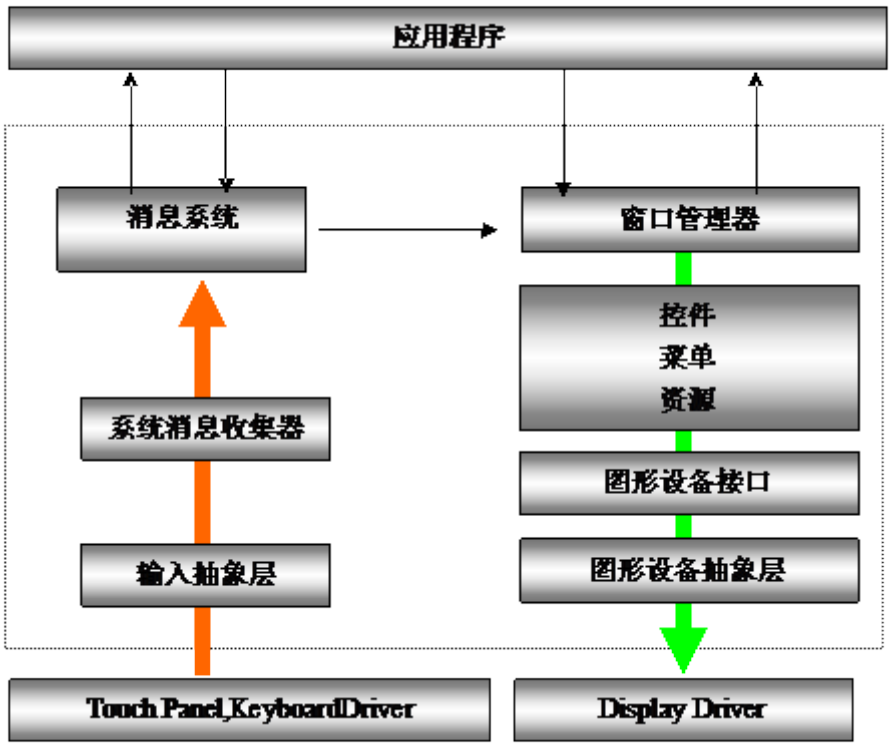
（1） 事件管理器

收集系统消息，转换并分发系统消息和用户消息给各个窗口对象。
消息队列管理
（2）窗口管理器：

管理窗口的创建，销毁
窗口的绘制
活动窗口，输入焦点的切换
窗口间关系的管理
控件，菜单实现
（3）GDI

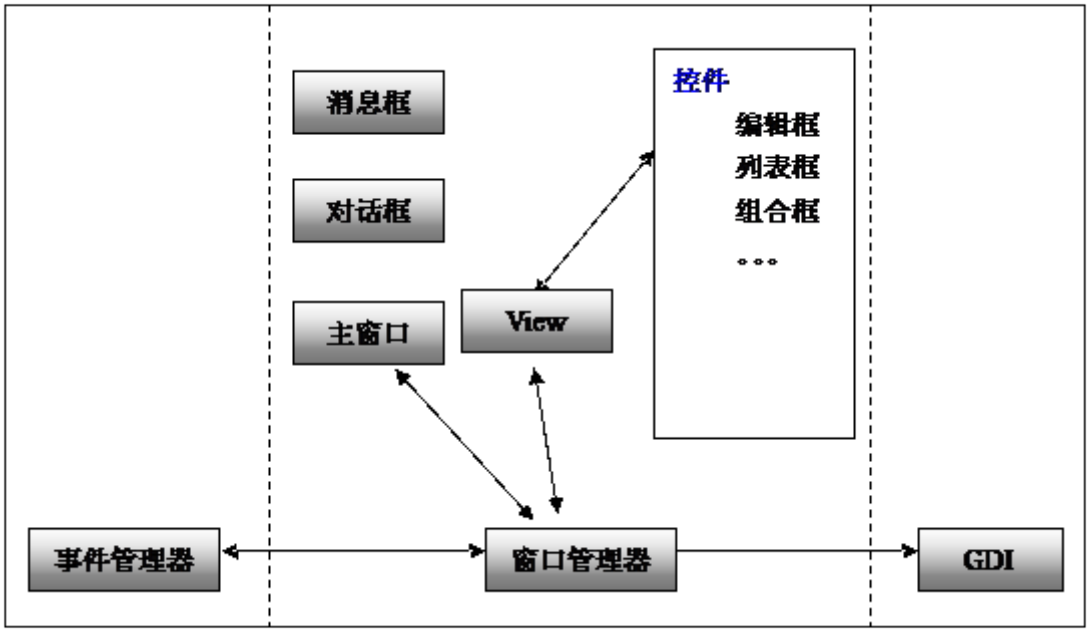
上下文设备管理
上下文设备对象管理：字体，画笔等
图形绘制:点、线，填充等
图象操作:位传送、位图操作

2 系统体系构架及其数据流的大体走向



在本质上 GUI 就是管理用户输入和屏幕输出，我们从上面的体系结构可以看到 GUI 的这两大数据流的基本流向，这也决定了 Android GWES 设计的最基本的着眼点。

Android 弱化了窗口的概念，着重使用 View 的概念。所以 Android 的基本组成可以从上面的图修改成如下的组成：



Android 核心分析 之十一------Android GWES 之消息系统

Android GWES 之 Android 消息系统
Looper，Handler，View

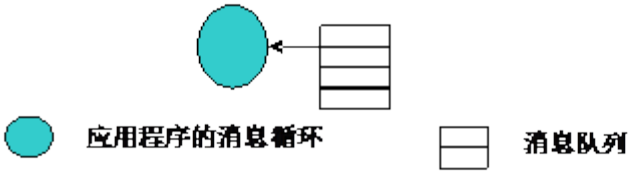
我们要理解 Android 的消息系统，Looper，Handle，View 等概念还是需要从消息系统的基本原理及其构造这个源头开始。从这个源头，我们才能很清楚的看到 Android 设计者设计消息系统之意图及其设计的技术路线。

消息系统的基本原理

从一般的系统设计来讲，一个消息循环系统的建立需要有以下几个要素：

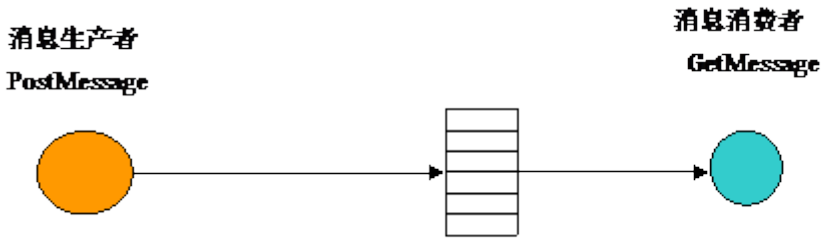
消息队列
发送消息
消息读取
消息分发
消息循环线程

首先来研究一下消息驱动的基本模型，我使用如下的图形来表示一个消息系统最基本构成：

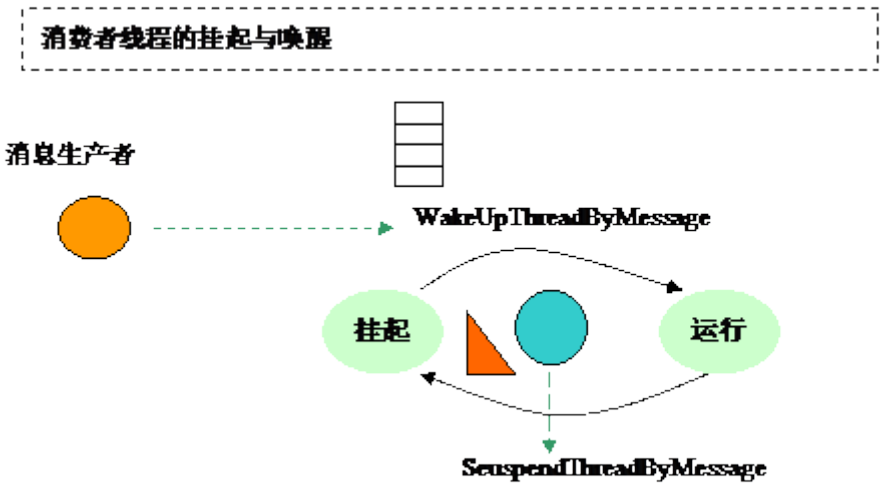


上面的模型代表应用程序一直查询自己的消息队列,如果有有消息进来,应用消息处理函数中根据消息类型及其参数来作相应的处理。

消息系统要运作起来,必定有消息的产生和消费。我们可以从下图看到消息生产和消费的一个基本的链条，这是一个最基本的，最简单的消息系统。



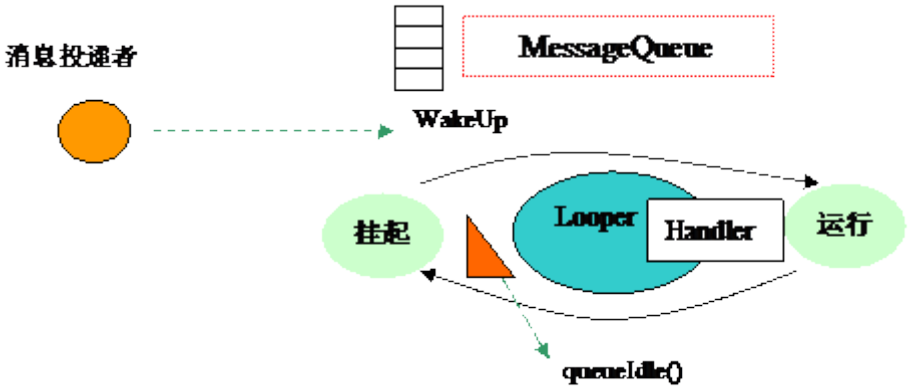
生产线程将消息发送到消息队列，消息消费者线程从消息队列取出消息进行相应的处理。但是这样简单的模型对实际运行的系统来说是不够的，例如对系统资源的消耗等不能很好的处理，我们就需要一个有旗语的消息系统模型，在上面的消息系统模型中加入了一个旗语，让消息消费者线程在没有消息队列为空时，等待旗语，进入到挂起状态，而有消息到达时，才被唤醒继续运行。当然生产者同时也可以是消费者。



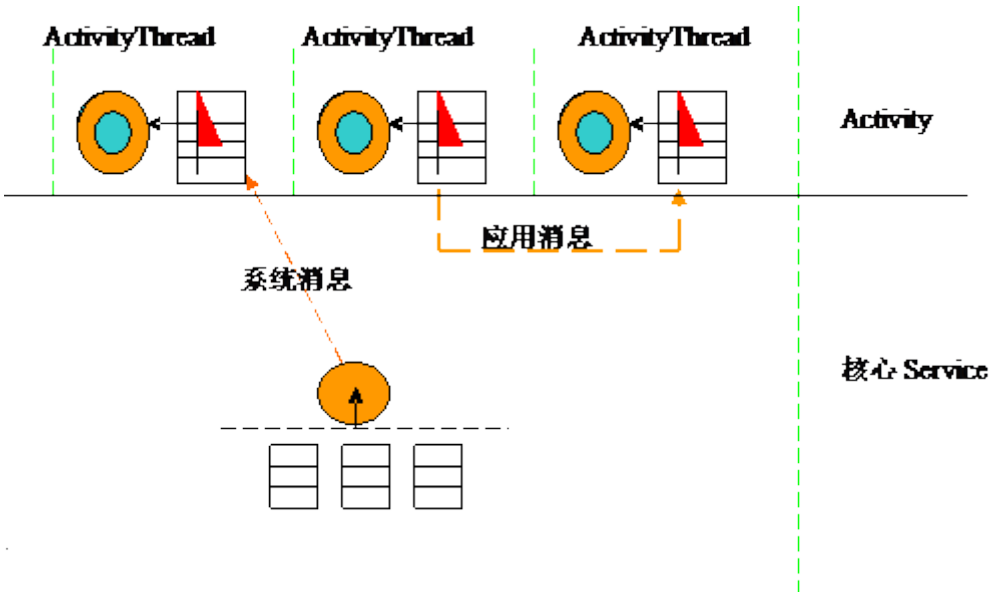
2 Android 的消息模型

Android 要建立一个消息系统使用了 Looper，MessageQueue,Handler 等概念，从上节的原理我们可以知道这些都是概念包装,本质的东西就是消息队列中消息的分发路径的和消息分发处理方式的设计。Android 巧妙的利用了对象抽象技术抽象出了 Looper 和 Handler 的概念。在 Looper 和 Handler 两个概念的基础上,通过 View 的处理函数框架，Android 十分完美的达到消息分发的目的。

参照基本消息系统描述模型，我给出了 Android 消息系统整体框架,表示如下：

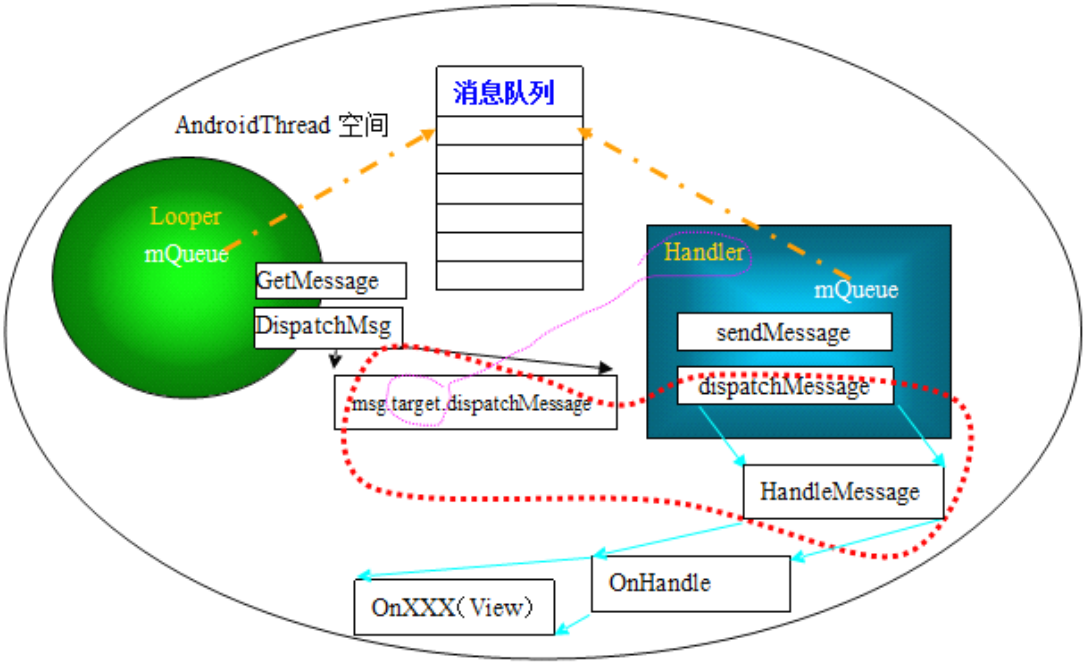


Android 消息系统消息分发框架



3 Looper,Handler 详解

Looper 只是产生一个消息循环框架，首先 Looper 创建了消息队列并把它挂接在 Linux 的线程上下文中，进入到取消息，并分发消息的循环当中。Handler 对象在同一个线程上下文中取得消息队列，对消息队列进行封装操作，最主要的就是 SendMessage 和担当起 dispatchMessage 这个实际工作。外部系统需要向某个 Android 线程发送消息，必须通过属于该 AndroidThread 的 Handler 这个对象进行。



Handler 属于某个线程，取决 Handlerd 对象在哪个线程中建立。Handler 在构建时做了如下的默认动作：

l 从线程上下文取得 Looper。
l 通过 Looper 获取到消息队列并记录在自己的成员 mQueue 变量中

Handler 使用消息队列进行对象封装，提供如下的成员函数：

l 通过 post(Runnable r)发送。Runnable 是消息处理的回调函数，通过该消息的发送，引起 Runable 的回调运行，Post 消息放置消息队列的前面。Message.callback=Runable.
l 通过 sendMessage 发送。放置在所有的 Post 消息之后，sendMessage 发送消息.
l dispatchMessage 分发消息。消息带有回调函数，则执行消息回调函数，如何没有则使用默认处理函数：handleMessage。而 handleMessage 往往被重载成某个继承 Handler 对象的新的特定的 handleMessage。
几乎所有的 Message 发送时，都指定了 target。Message.target=(this).

Looper 运行在 Activity 何处？我们现在可以从代码堆栈中纵观一下 Looper 的位置。

NaiveStart.main()

ZygoteInit.main

ZygoteInit$MethodAndArgsCall.run

Method.Invoke

method.invokeNative

ActivityThread.main()

Looper.loop()

ViewRoot$RootHandler().dispatch()

handleMessage

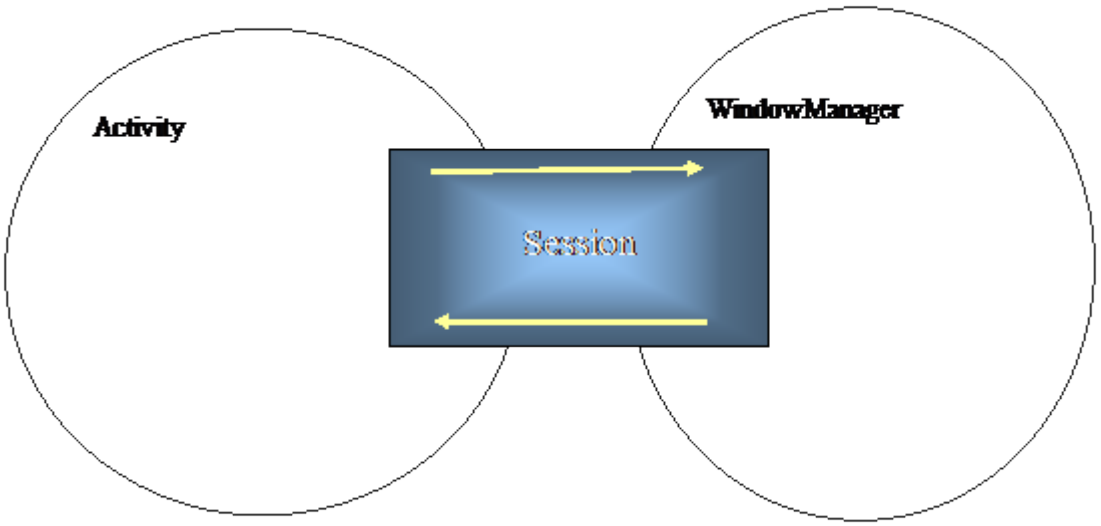
....

这样我们就更清楚的了解到 Looper 的运行位置。

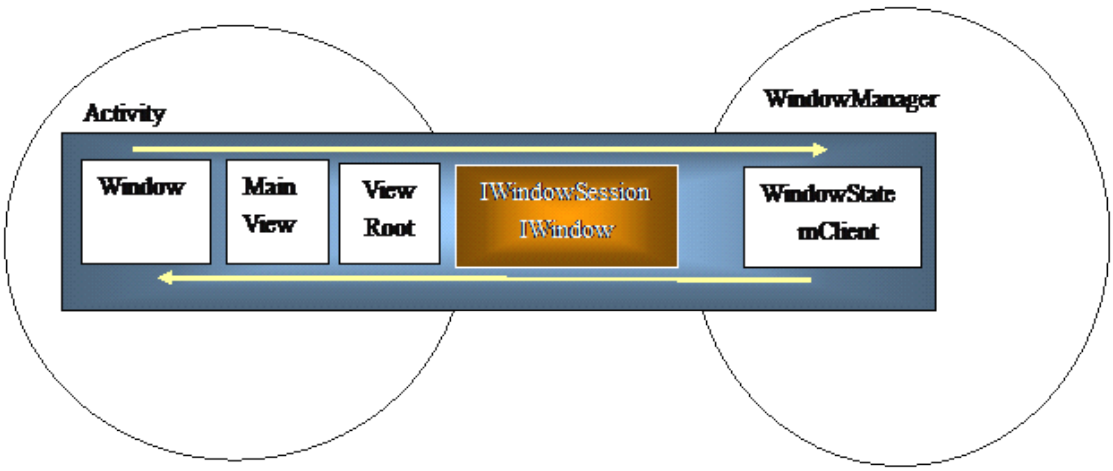
Android 核心分析(12) -----Android GEWS 窗口管理之基本架构原理

Android GWES 之窗口管理之基本构架原理

Android 的窗口管理是 C/S 模式的。Android 中的 Window 是表示 Top Level 等顶级窗口的概念。DecorView 是 Window 的 Top-Level View，这个 View 我称之为主 View，DecorView 会缺省的 attach 到 Activity 的主窗口中。主 View 被加入到 WindowManager 中，WM 使用 WindowState 与这个主 View 对应。



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。一个完整的窗口概念横跨了 View， ViewRoot， WindowManager Service。Window， DecorView， View，IWindow ,ISession，WindowState 之间的关系如下：



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。

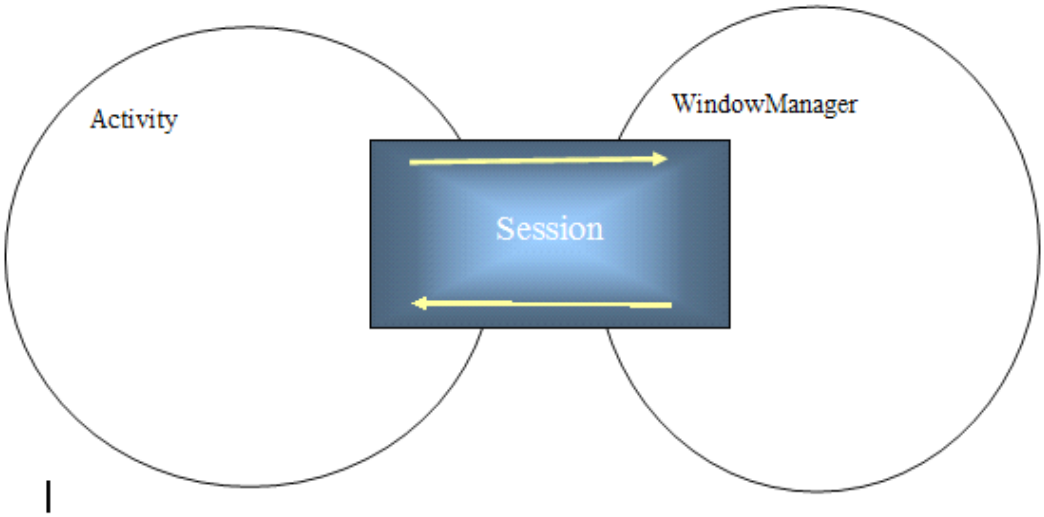
后面我们将通过 Client，WM Service 分别加以分析。

Android 核心分析(13) ----Android GWES 之 Android 窗口管理

Android GWES 之 Android 窗口管理

1 基本构架原理

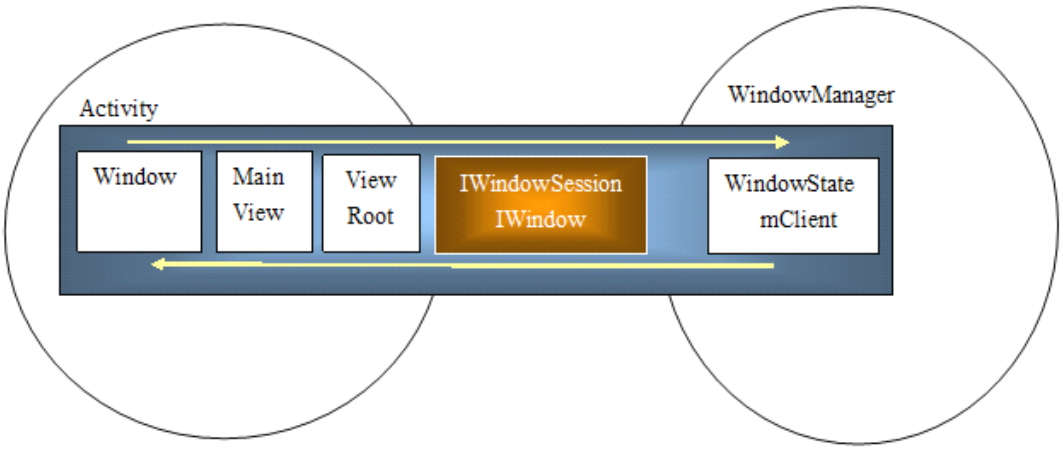
Android 的窗口管理是 C/S 模式的。Android 中的 Window 是表示 Top Level 等顶级窗口的概念。DecorView 是 Window 的 Top-Level View，这个 View 我称之为主 View，DecorView 会缺省的 attach 到 Activity 的主窗口中。主 View 被加入到 WindowManager 中，WM 使用 WindowState 与这个主 View 对应。



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。

一个完整的窗口概念横跨了 View，ViewRoot，WindowManager Service。Window，DecorView，

View，IWindow ,ISession，WindowState 之间的关系如下



Client 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。

后面我们将通过 Client，WM Service 分别加以分析。

2 Client 端

我一致认为在 Android 中 Window 的概念并不是个很重要的概念。他的 Window 类，只是在 PhoneWindow 和 MidWindow 中使用。而 PhoneWindow 只是做了一个具体跟手机功能相关的公用事件的处理，所以在 Android 中 PhoneWindow 并不是一个抽象的纯正概念，而是一个跟手机系统相关的一个特别窗口概念，例如按键的默认动作处理，按键音的发出等等。

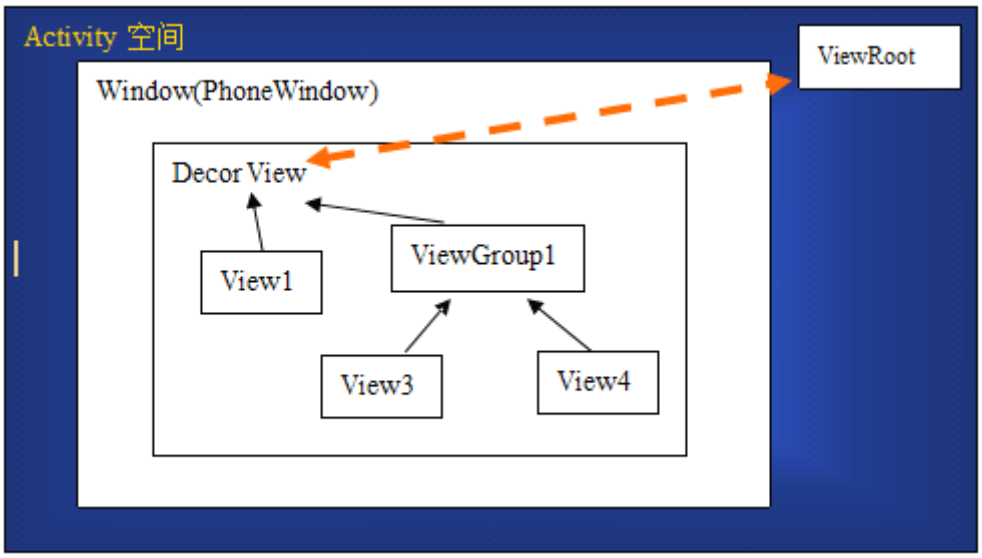
2.1 View

在 Activity 中真正重要的概念是 View，以下是 Google 官方对 View 的定义：

This class represents the basic building block for user interface components. A View occupies a rectangular area on the screen and is responsible for drawing and event handling. View is the base class for widgets, which are used to create interactive UI components (buttons, text fields, etc.). The {@link android.view.ViewGroup} subclass is the base class for layouts, which are invisible containers that hold other Views (or other ViewGroups) and define their layout properties.

我对 View 不做翻译，翻译成视图好像不太佳，View 在 Android 中，View 比视图具有广的外延。View 包含了用户交互，包含了显示，视图在中文中仅仅表示了静态的显示。对于 View 的理解应该从最容易的理解开始。我们使用过编辑器，在 Android 中这个编辑器就是一个 View，这个编辑器需要显示文字，需要接收用户的键盘输入和鼠标选择，但是一个屏幕上有多个编辑器，如何管理，如何切换焦点编辑器，这些都是需要管理的。

客户端的组成：（Window，View，ViewRoot，WindowManager Proxy）

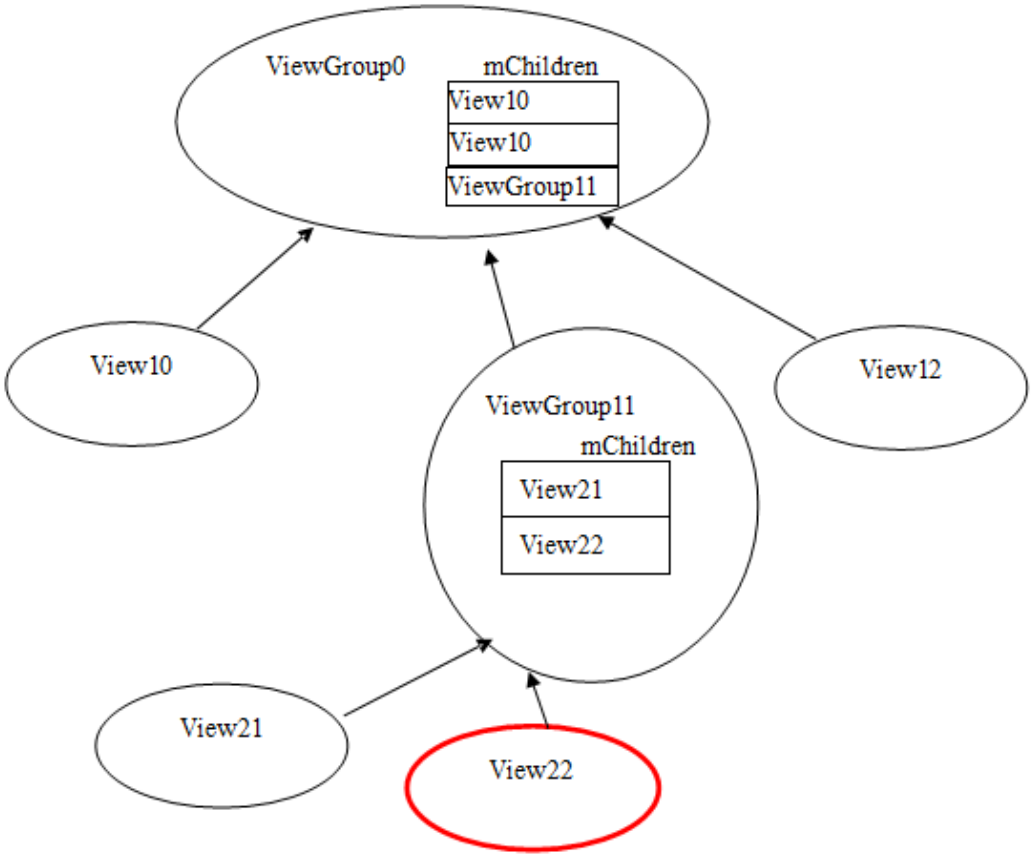


在 Activity 在 performLaunchActivity 时，会使用 Activity.attach()建立一个 PhoneWindow 主窗口。这个主窗口的建立并不是一个重点。handleResumeActivity 真正要启动一个 Activity 时候，将主窗口加入到 WindowManager，当然并不是将主窗口本身，而是将主窗口的 DecorView 加入到 WindowManager 中。

真正 Window 核心的抽象概念存在于 View，ViewRoot，WindowManger 中的 WindowState。为了描述概念的方便性，我特别提出主 View 这个概念，这个主 View 就是 Top-Level View of the window. 主 View 与 View 想对，突出主 View 是 attatch 到主窗口上的。而一般的 View 则是存在于主 View 中的。主窗口这个概念，我讲的主窗口实际上就是 Android 提到的 Top Level Window。

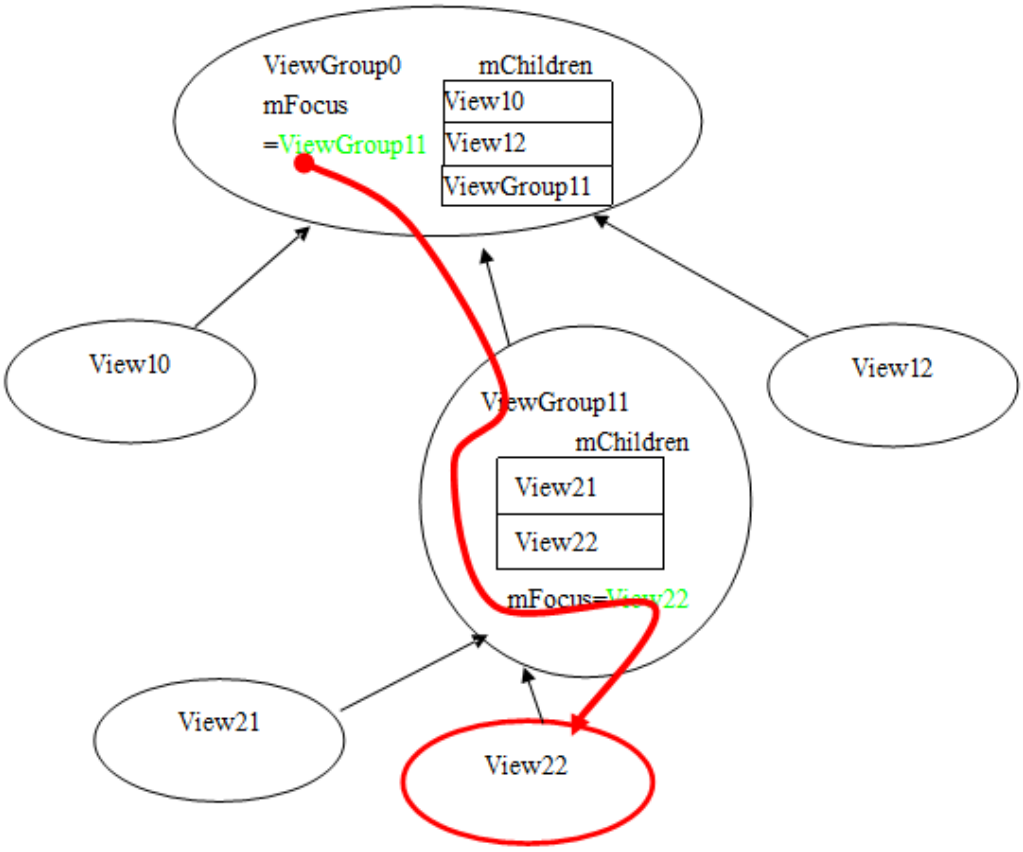
我们所提到的概念：View，GroupView，DecorView，ViewRoot 都是存在于 Client 端，只有 WindowState 这个概念存在于 Window Manager Service 端。

DecorView 实际上是一个 ViewGroup。在依存关系上来讲，对看个主窗口来讲，DecorView 是 Top-Level View.View 并不是关注的重点，重要的是我们如何需要知道分发路径是建立在什么关系上的。View 的成员变量 mParent 用来管理 View 上级关系的。而 ViewGroup 顾名思义就是一组 View 的管理，于是在 ViewGroup 构建了焦点管理和子 View 节点数组。这样通过 View 的 mParent 和 ViewGroup 的 mChildren 构建了 Android 中 View 直接的关系网。



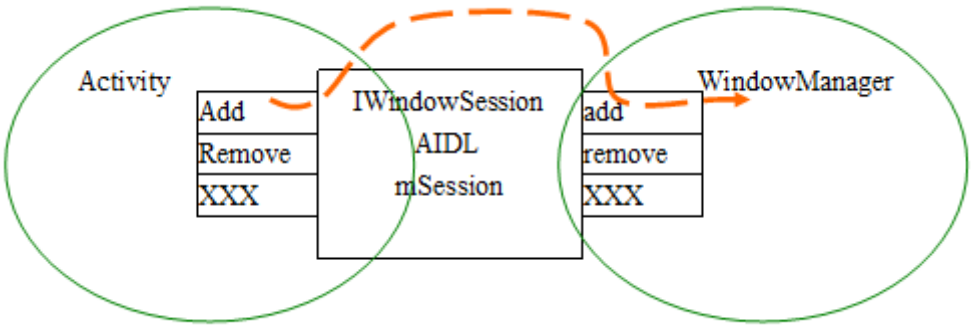
2.2 Focus Path

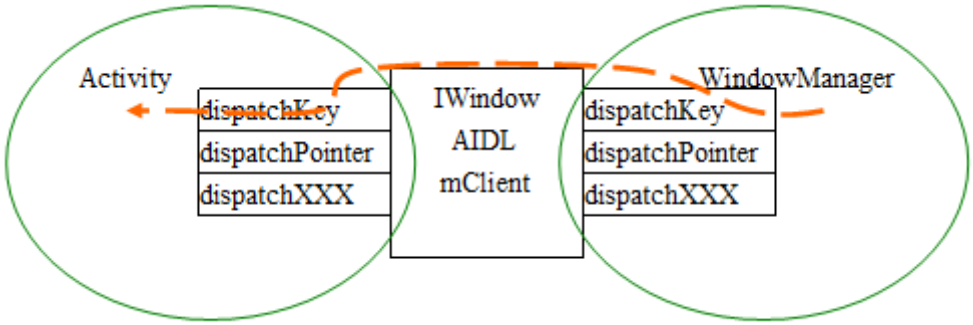
所谓的 Foucs Path 就是我们的 KeyEvent 传递的路线。一般的我们的 KeyEvent 在主循环中主 View 通过 View 的焦点记录关系传递到焦点 View 上。例如下图，View22 是焦点，我们从最顶层的 View 通过 mFcous 的关系链找到最后所形成的路径就是 Focus Path。



2.3 ViewRoot，Window Manager Proxy

ViewRoot 与 Window Manager 的核心是 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。利用 AIDL 接口进行进程间通信。



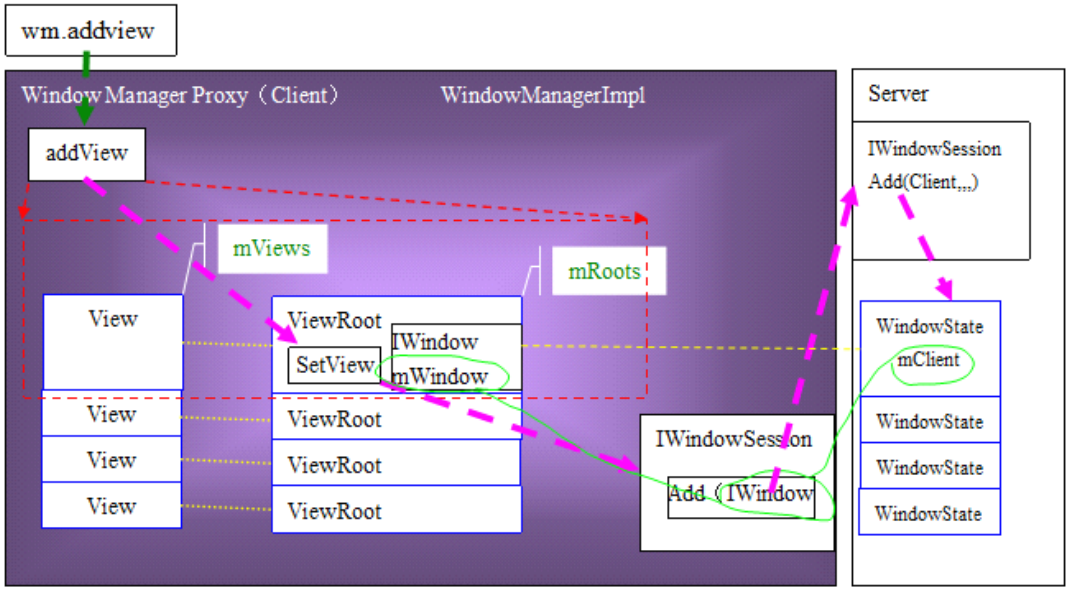


ViewRoot 实际是一个 Handler，ViewRoot 建立主 View 与 WindowsManger 通讯的桥梁。ViewRoot 在本质上一个 Handler。我们知道 Handler 的基本功能就是处理回调，发送消息。

Activity 在使用 getSystemService 获取 WindowManagerImpl ，建立了一个 WindowManagerImpl 实例，即 Window Manager 服务的代理：

wm=(WindowManagerImpl)context.getSystemService(Context.WINDOW_SERVICE); 并调用 wm.addview 添加窗口到 WMService 中。

这个过程在客户端建立了什么样的管理框架，并如何这个会话？在 Window Manager Proxy 中建立了 View，Layout ,ViewRoot 三者的对应关系表。构造一个 ViewRoot 就会打开一个 session,并利用 IWindowSession 建立会话上下文。



4 Window Manager Service

本次对于 Window Manager Service 的研究仅限于 FocusWindow,消息系统。其他的部分将在后面的专门章节讨论。

Window Manager 管理的窗口是应用程序的 Top-level 窗口，我这里参照 Window 的概念叫主窗口。主窗口为什么要放在在 Service 这边来管理呢？为什么不放在 Client 那边？主窗口放置在一起管理是为了计算 Z-order 序列，根据应用程序的状态来显隐应用程序的窗口。我想 Android 设计者在考虑设计窗口系统的时候，一定首先考虑：

窗口 z-order 序的管理
活动窗口的计算，及其变化通知
窗口归属（属于哪个应用）
输入法管理
Window Service 大体上实现了如下的功能：，

（1）Z-ordered 的维护函数

（2）输入法管理

（3）AddWindow/RemoveWindow

（4）Layerout

（5）Token 管理，AppToken

（6）活动窗口管理（FocusWindow）

（7）活动应用管理（FocusAPP）

（8）转场动画

（9）系统消息收集线程

（11）系统消息分发线程

在服务端的窗口对象叫做 WindowState。在 Service 维护了一个 mWindow 数组，这个 mWindow 就是 Window 的 Z-order 序数组。mWindowMap 用于记录<Client：Binder,WindowState 对象>。

WindowState 有一个叫做 mClient 成员变量来记录客户端 IWindow 实例，通过 IWindow 接口实例，Service 可以访问客户端的信息，说以 IWindow 是 Service 连接 View 桥梁。

（1） FocusWindow 活动窗口如何计算?

基本原理就是查找前景应用（FousActivity），并同 Z-Order 序中找出属于该 FousActivity（AppToken）的主窗口，这个窗口就是计算出来的 Focus Window。

（2）为什么要提出 Token 这个概念呢？

一个应用程序要管理自己的窗口，那么如何来标识该窗口是属于某个 Activity，Andoid 设计者提出了 AppToken 这个概念。AppToken 在本质上的描述：<Token：IBinder，allWindows>，通过 Token 找到属于该 Token 的 allWindows。使用 Token 开始完成该应用程序的所有窗口的显示和隐藏。

（3）系统消息收集与处理

我们下面重点研究Ｓｅｒｖｉｃｅ中的系统消息收集模式及其分发模式。Ｓｅｒｖｉｃｅ使用Ｋｅｙ Ｑ作为专门的消息队列。

ＫeyEvent
TouchEvent
TrackballEvent
系统有两个线程：

KeyQ 线程，通过 Navite 函数 readEvent 轮询设备,将读取的结果放置在 KeyQ 队列中。

系统 dispatcher 等待在 KeyQ 消息队列上，一旦从消息队列中获取到消息，就通过分发函数通过 mClient 传递到 Client 端。

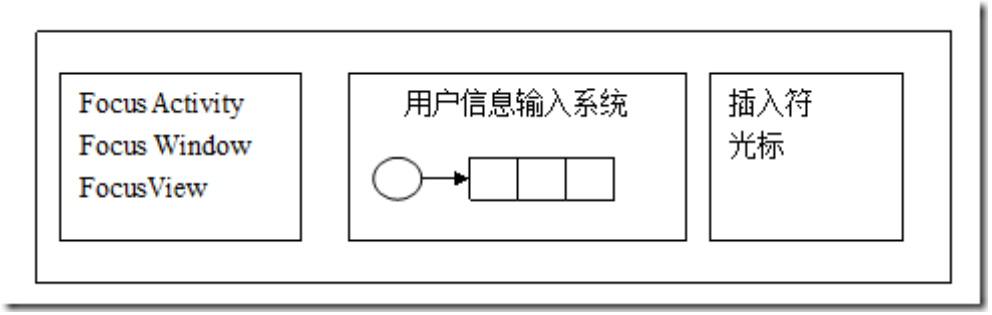
Android 核心分析（14）------ Android GWES 之输入系统

Android 输入系统

依照惯例，在研究 Android 输入系统之前给出输入系统的本质描述：从哲学的观点来看，输入系统就是解决从哪里来又将到哪里去问题。输入的本质上的工作就是收集用户输入信息并放置到目标位置。

Android 在源代码分类上，并没有输入系统分类。本章的输入系统研究是一个综合的分析，前面的 GWES 的分析，特别是 View 的 Focus Path 以及 Window Manager Proxy 是本章分析的基础,如果没有理解，请参阅前面的窗口管理的相关章节。

Android 输入系统的组成



输入系统由如下几部分组成：

1）后台窗口管理服务

2）Focus Activity

3）Focus Window

4）Focus View：用来接收键盘消息

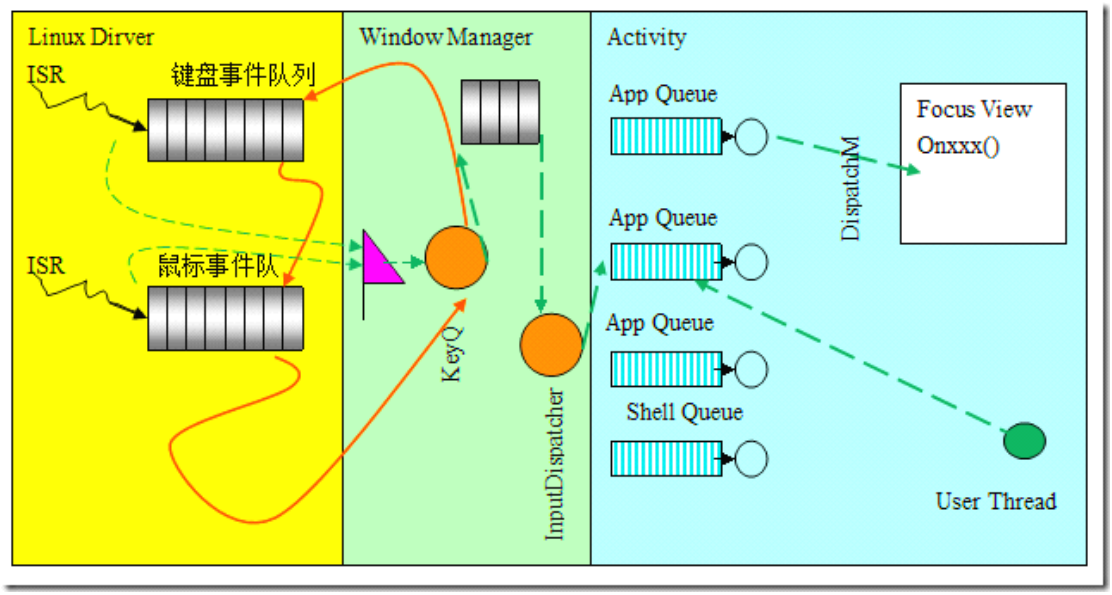
从输入系统这个角度去看 Android 的 Window Manager 服务解决了用户信息输入收集，而 FocusActvitiy，Focus Window、Focus View 这些概念的设计是为了解决用户输入应该放到哪里去这个问题。在整个 Android 系统中，同时只有一个一个 Focus Window，而属于该 Window 的 Focus View 才是真正的 Focus View。

在 Android 系统中，在设计上要求多个 Actvitiy 同时存在运行。在实现中，每次把 Actvitiy 变成 Focused Actvitiy 时（setFocusedActivity@ActivityManagerService.java）激活程序的时候，就把该 Activity 的主窗口设置成前景窗口,即系统中的顶层窗口，AppToken 概念的引进就是为了解决窗口对象的归属问题。在这个过程中，在逻辑上看，我们挑选了一个 Activity 作为了 Focus Activity 来接收系统的消息，实质上这个 Focus Activity 的 Focus 窗口就是前景窗口。

Focus 窗口的改变将改变焦点 View，前景窗口的改变也将引起焦点 View 的变化。焦点和光标的概念用于管理输入设备和输入事件的传送。光标是一个绘制在屏幕之上的小位图，指示当前的输入位置。键盘输入有类似的输入焦点和键盘输入插入符的概念。只有具有输入焦点的窗口才能获取键盘事件。改变窗口的焦点通常由特殊的按键组合或者 TouchEvent 事件完成。具有输入焦点的窗口通常绘制有一个键盘插入符。该插入符的存在、形式、位置，以及该插入符的控制完全是由窗口的事件处理例程完成的。

现在站在更宏观的位置来看 Actvitiy 的输入系统，可以从 Linux Driver 开始到输入框结束的整个链条，我这里给出大输入系统的概念，Android 大输入系统包含：Linux driver, Window Manager, Message System, View Focus Path，Focus View。

Android 输入系统架构图



现在从 Android 的代码分析的角度，来看看输入系统的组成。这个过程从代码中分析处理：

在 Window Manager Service 端

readEvent@com_android_server_KeyInputQueue.cpp

KeyQ@WindowMangerService.java

KeyInputQ@KeyInputeQueue.java

InputDispatcherThread@WindowMangerService.java

在 Client 端

IWindow@ViewRoot.Java

ViewRoot@ViewRoot.java

KeyInputQ 在 WindowMangerService 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 RawEvent，并放入到 KeyQ 消息队列中。

InputDispatcherThread 从 KeyQ 中读取 Events，找到 Window Manager 中的 Focus Window，通过 Focus Window 记录的 mClient 接口，将 Events 专递到 Client 端。Client 端在根据自己的 Focus Path 传递事件，直到事件被处理。

Android 核心分析（15)-------Android 输入系统之输入路径详解

Android 用户事件输入路径

1 输入路径的一般原理

按键，鼠标消息从收集到最终将发送到焦点窗口，要经历怎样的路径，是 Android GWES 设计方案中需要详细考虑的问题。按键，鼠标等用户消息消息的处理可分为不同的情况进行判定：

（1）用户输入根据系统状况是否应该派送。如在 ScreenOff 的情况下，在按键属于特殊按键的情况下等

（2）是否有拦截 Listener

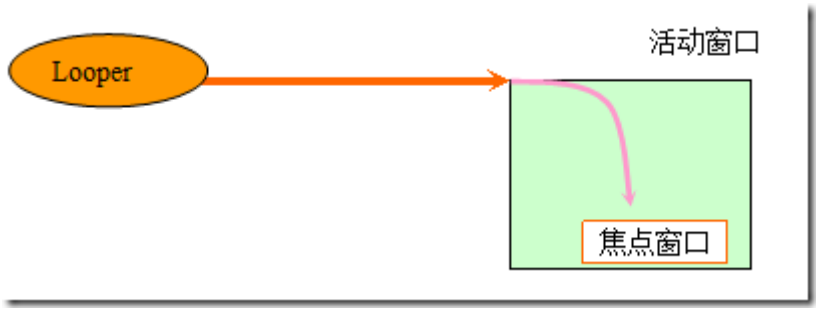
（3）对按键事件来讲，是否存在输入法

（4）是否是焦点终点

（5）是否为焦点切换按相关键

这些情况都是设计输入路径需要考虑的基本条件。

1.1 一般的输入路径设计
该输入路径实际上是指的按键消息（MSG_KEYDOWN，MSG_KEYUP， MSG_LongPress）的输入路径，即从活动主窗口到焦点窗口所经历的路程。

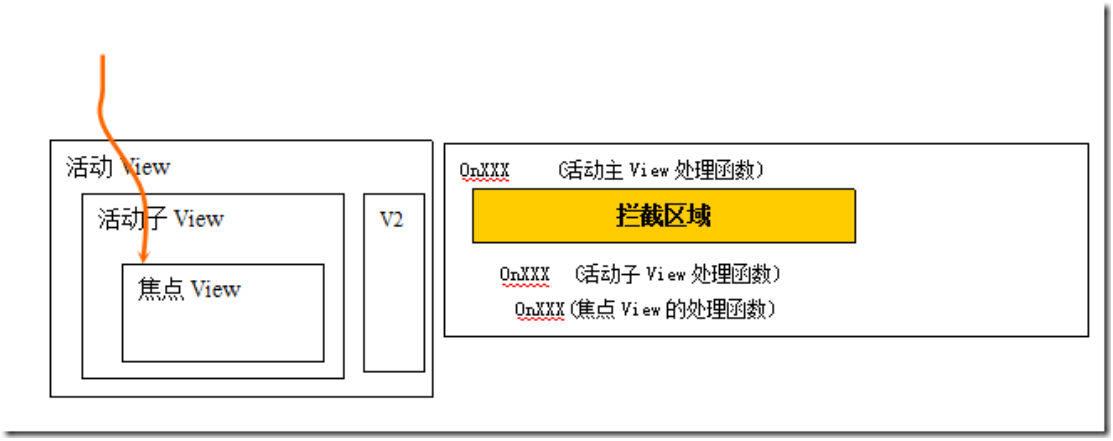


将信息输入路径分为两步：

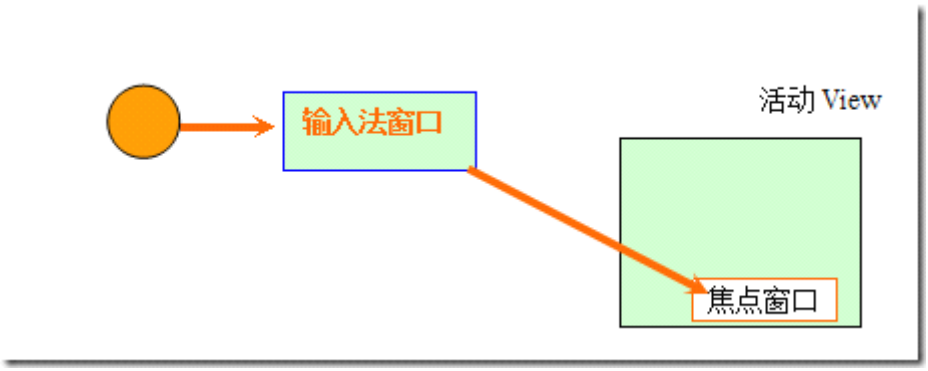
Step 1）窗口管理器将信息发送到活动窗口

Step 2）活动窗口通过缺省处理函数将该消息一层层的传递到焦点。

这样应用程序可以在活动 View 的处理函数中来预先处理用户输入信息，从而增强应用对用户信息的控制力。



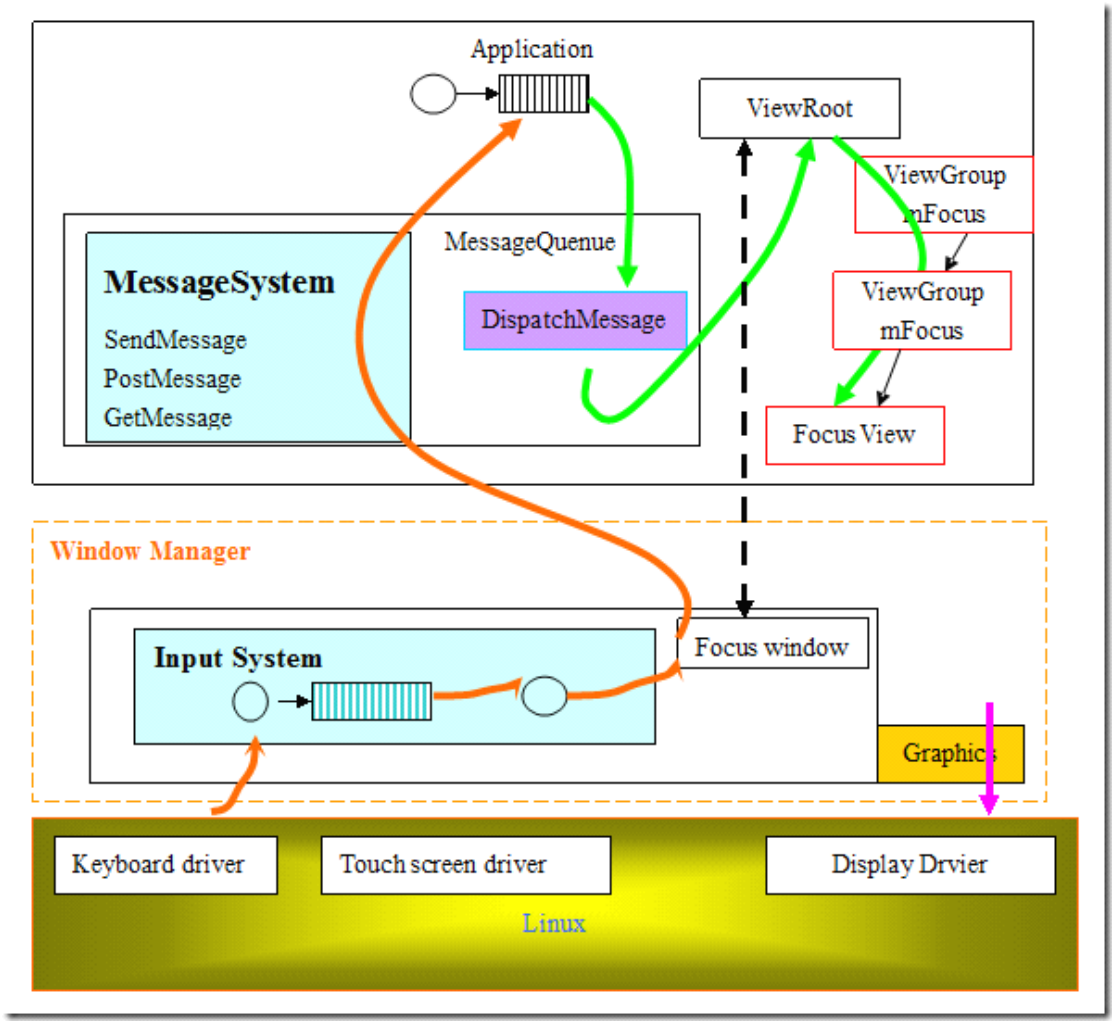
传递路径是通过 View 的缺省处理函数 Onxxx 来完成。通过 ActiveView ->focus->focus->focus 的链条关系，一级一级的将按键消息 MSG_KEYDOWN，MSG_KEYUP， MSG_CHAR 等传递到 focus 窗口。



此时用户按键输入先发送到输入法窗口，经过输入法管理器处理，过滤后将输入法产生的结果放置到焦点 View。

1.3 输入系统整体流程

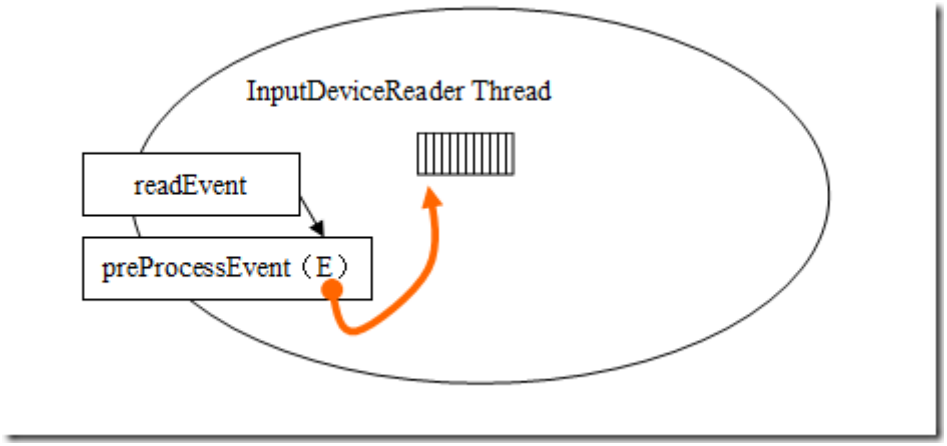
下面示意图是 Android 输入系统的数据流途径，通过 WM 的输入系统线程收集消息，分发到 Focus Activity 消息队列，然后通过消息系统派发。



2 Android 输入路径详细描述

2.1 第一步:用户数据收集及其初步判定

KeyInputQ 在 WindowMangerService 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 RawEvent，放入到 KeyQ 消息队列中。



preProcessEvent()@KeyInptQ@KeyInputQueue.java 这个是在输入系统中的第一个拦截函数原型。KeyQ 重载了 preProcessEvent()@WindowManagerService.java。在该成员函数中进行如下动作：

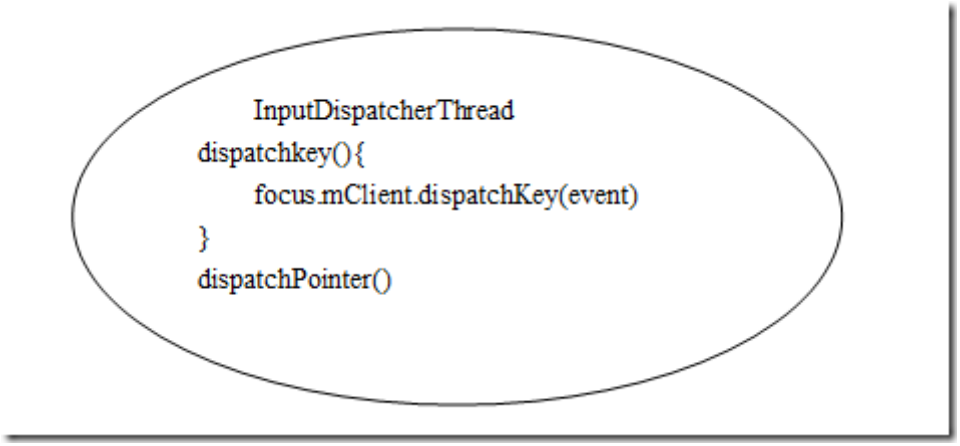
（1） 根据 PowerManager 获取的 Screen on,Screen off 状态来判定用户输入的是否 WakeUPScreen。

（2） 如果按键式应用程序切换按键，则切换应用程序。

（3） 根据 WindowManagerPolicy 觉得该用户输入是否投递。

2.2 第二步 消息分发第一层面

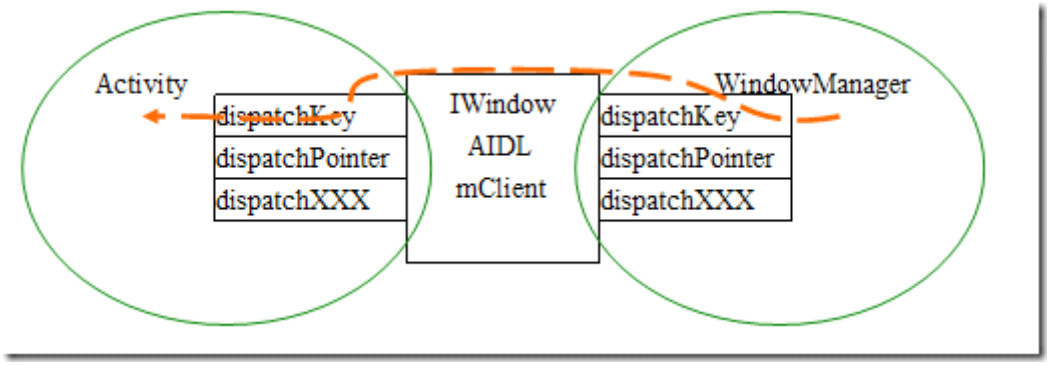
InputDispatcherThread 从 KeyQ 中读取 Events，找到 Window Manager 中的 Focus Window，通过 Focus Window 记录的 mClient 接口，将 Events 专递到 Client 端。



如何将 KeyEvent 对象传到 Client 端：

在前面的章节（窗口管理 ViewRoot，Window Manager Proxy）我们已经知道：在客户端建立 Window Manager Proxy 后，添加窗口到 Window Manager service 时，带了一个跟客户 ViewRoot 相关的 IWindow 接口实例过去，记录在 WindowState 中的 mClient 成员变量中。通过 IWindow 这个 AIDL 接口实例，Service 可以访问客户端的信息，IWindow 是 Service 连

接 View 桥梁。



看看在 Client ViewRootKeyEvent 的分发过程

IWindow：dispatchKey(event)

dispatchKey(event)@W@ViewRoot@ViewRoot.java

ViewRoot.dispatchKey(event)@ViewRoot.java

message>

sendMessageAtTime(msg)@Handler@Handler.java

至此我们通过前面的 Looper，Handler 详解章节的分析结论，我们可以知道 Key Message 已经放入到应用程序的消息队列。

2.3 第三步：应用消息队列分发

消息的分发,在 Looper，Handler 详解章节我们分析了 Looper.loop()在最后后面调用了 handleMesage.

…

ActivityThread.main()

Looper.loop()

ViewRoot$RootHandler().dispatch()

handleMessage

....

注意到在分发的调用 msg.target.dispatch()，而这个 target 在第二层将消息 sendMessageAtTime 到消息队列时填入了 mag.target=this 即为 msg.target=ViewRoot 实例。所

有此时 handleMessage 就是 ViewRoot 重载的 handleMessage 函数。

handlerMessage@ViewRoot@ViewRoot.java

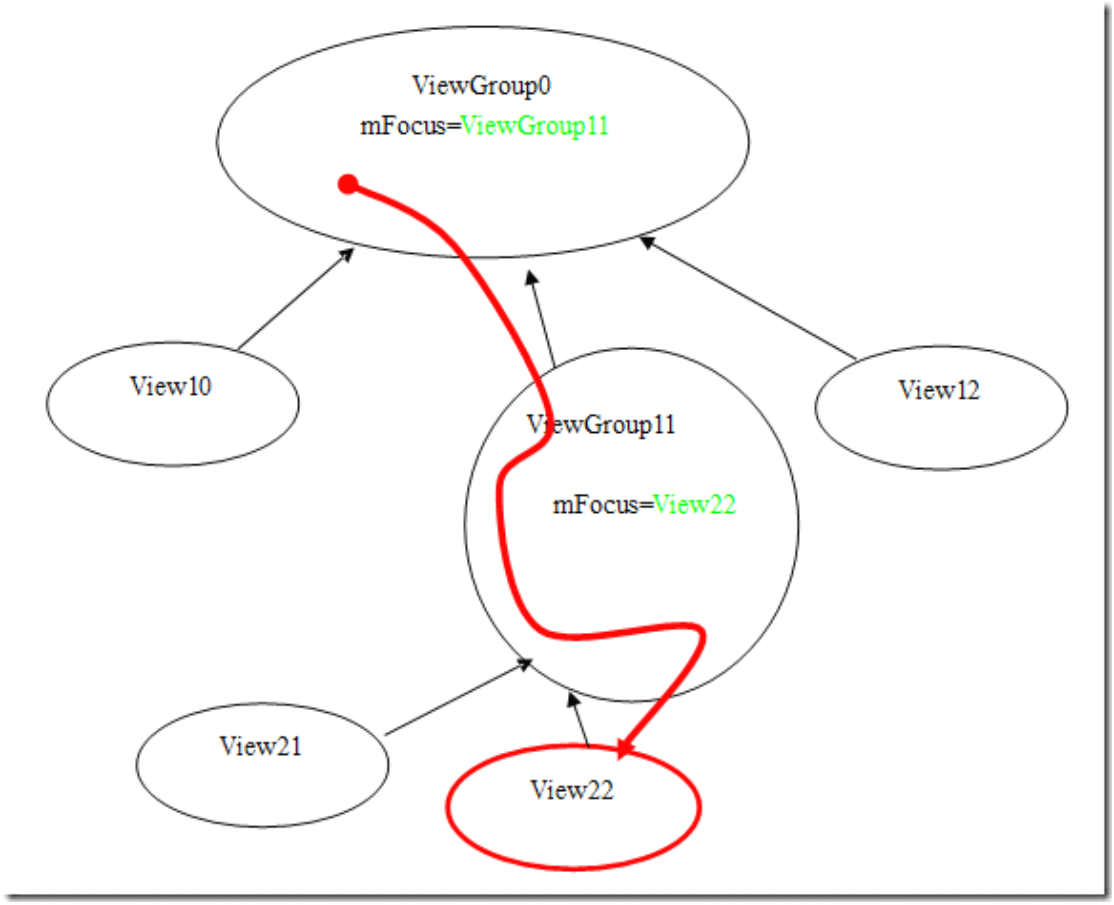
deliverkeyEvent

如果输入法存在，dispatchKey 到输入法服务。

否则 deliverKeyEventToViewHierarchy@ViewRoot.java

在这里需要强调的是，输入法的 KeyEvent 的拦截并没有放入到 Window Manager Service 中，而是放入到了客户端的 RootView 中来处理。

2.4 第四步：向焦点进发，完成焦点路径的遍历。



分发函数调用栈

deliverKeyEventToViewHierarchy@ViewRoot.java

mView.dispatchKeyEvent：mView 是与 ViewRoot 相对应的 Top-Level View.如果 mView 是一个 ViewGroup 则分发消息到他的 mFocus。

mView.dispatchKeyEvent @ViewGroup （ViewRoot@root）

Event.dispatch

mFocus.dispatchKeyEevnet

如果此时的 mFocu 还是一个 ViewGroup，这回将事件专递到下一层的焦点，直到 mFocus 为一个 View。通过这轮调用，就遍历了焦点 Path，至此，用户事件传递完成一个段落。

2.5 第五步 缺省处理
如果事件在上述 Focus View 没有处理掉，并且为方向键之类的焦点转换相关按键，则转移焦点到下一个 View。

Android 核心分析（16）-----Android 电话系统-概述篇

Android 电话系统之概述篇

首先抛开 Android 的一切概念来研究一下电话系统的最基本的描述。我们的手机首先用来打电话的，随后是需要一个电话本，随后是 PIM，随后是网络应用，随后是云计算，随后是想我们的手机无所不能，替代 PC。但是作为一个电话的基本功能如下：

0)拨叫电话,接听电话，挂断电话,发送短信，网络连接,PIM 管理

1)由于电话运营商为我们提供了呼叫等待，电话会议等补充业务，所以我们的手机需要管理多路通话，如何管理?

2)来电时，我们要播出来电铃声，接通时我们需要切换语音通道，这个又跟多媒体系统打上了交道，例如有耳机插上了，有蓝牙耳机连上了，系统该做如何的管理和切换？

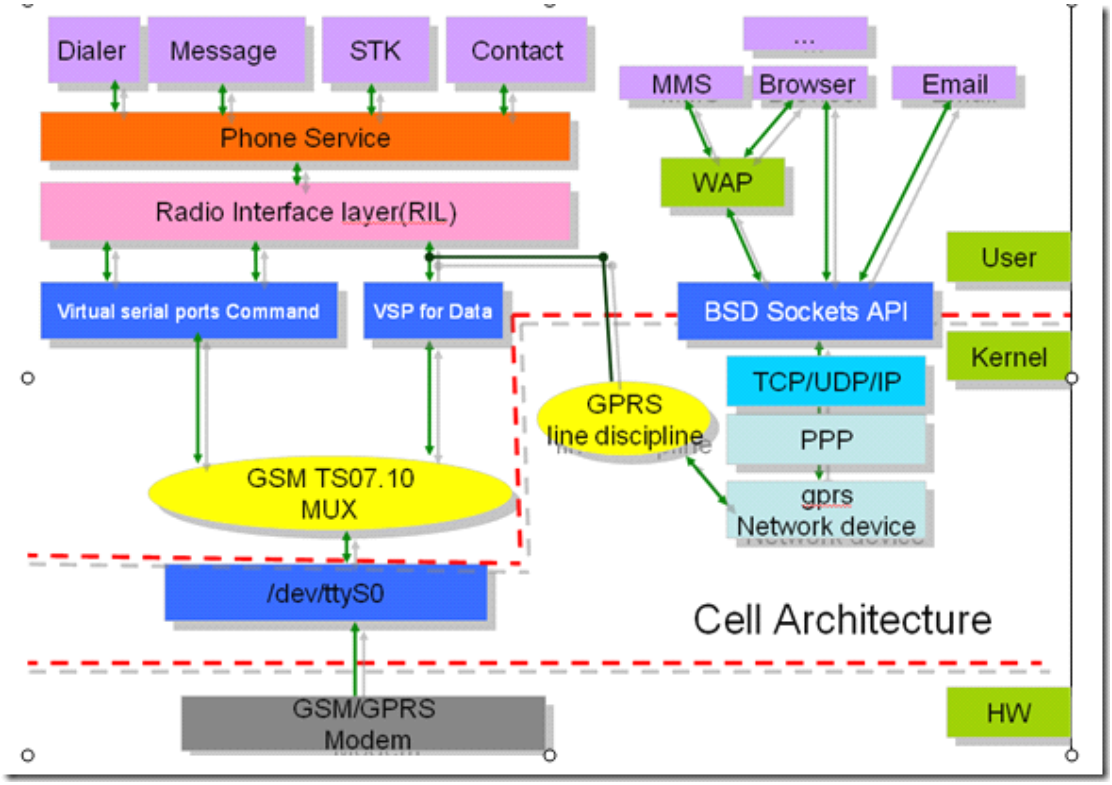
3）上网的网络通路建立（例如 GSM GPRS），如何 PPP 连接并连接到 LinuxSocket 通道上的？系统如何管理数据连接？

4）AP 跟 Modem 通讯时通过 AT 指令的，如何将 AT 指令变成一个个具体的操作函数，如何管理 Modem 发给我们的回应，AT 命令通道，数据通道如何管理？

5）sim 卡的电话本如何管理？

上面的关于手机的基本问题，Android 电话系统设计者必须要解答的问题。该设计如何的管理框架，提出什么概念来表达？所以要分析 Android 的电话部分，还是需要理解电话实现的背景知识，通讯协议，大体框架。

我们回到电话系统基本构成上，先从整体上去把握一下电话模块的大体框架，先从空中俯瞰。我给出的图是一般的智能手机的框架图，该框架基本能够概括所有手机电话模块的构成，当然也包括 Android 的电话系统构成。

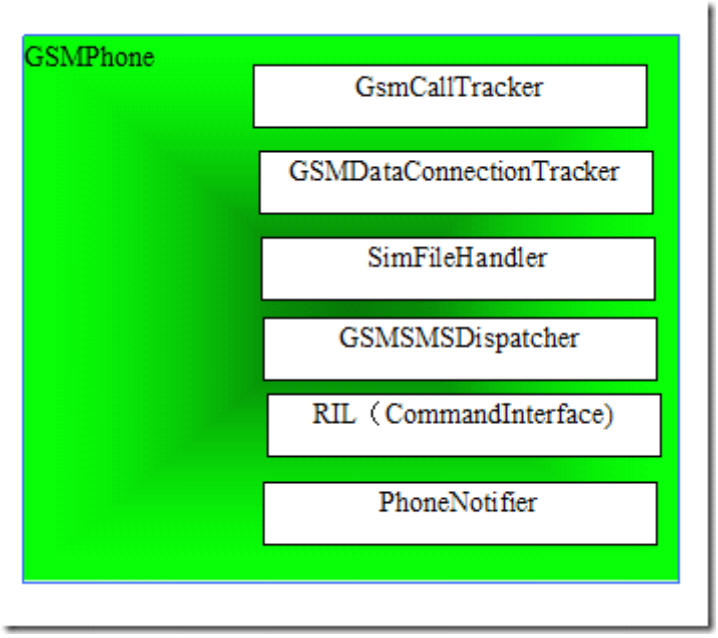


智能机架构一般是应用处理器+Modem。应用处理器与 Modem 的连接使用串口或者 USB。在一个硬件串口通路上实现为了要同时实现数据传输并同时实现控制 Modem,就需要实现多路复用协议（GSM TS07.10），在底层我们在多路复用的基础上虚拟了两个串口，一个用于 CMD 通道，一个用于 DATA 通道。电话的所有控制通路都是在这连个通道上。

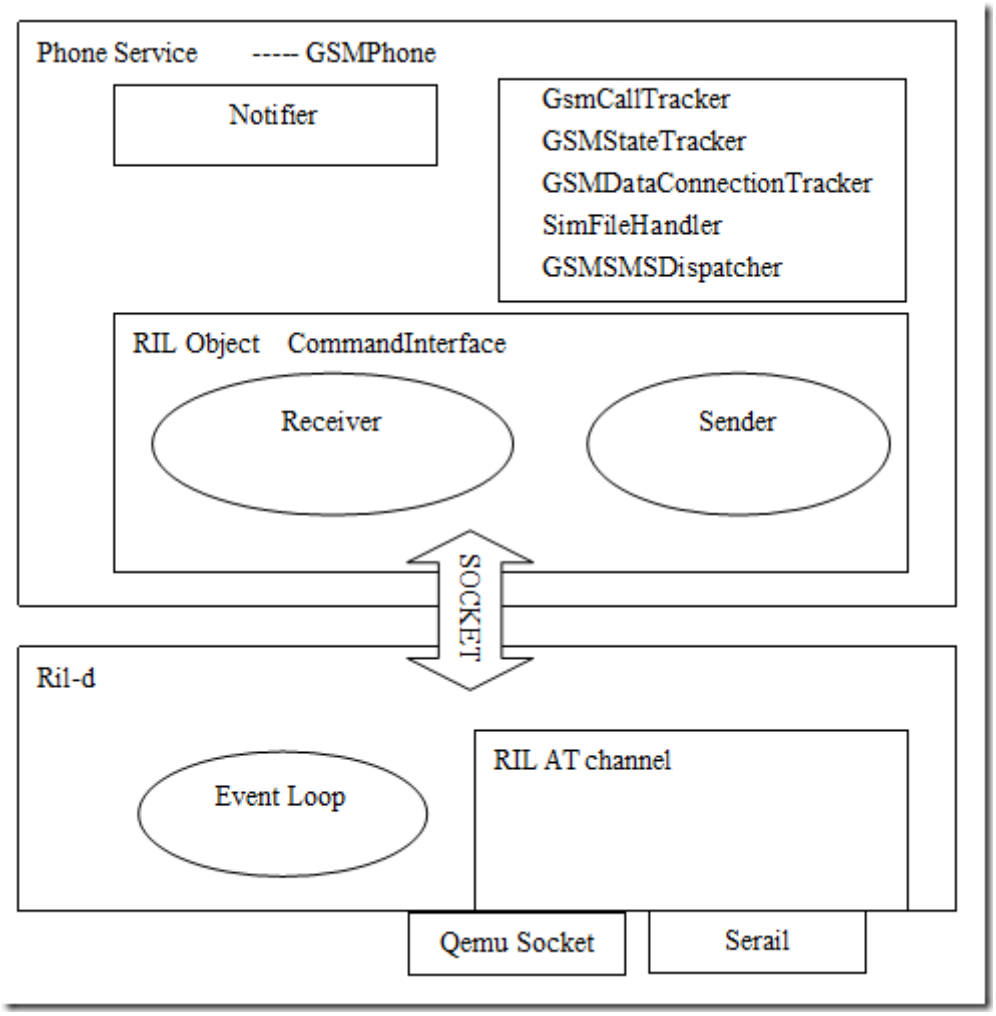
RIL,Radio Interface Layer。本层为一个协议转换层，手机框架需要适应多类型的 Modem 接入到系统中，而对于不同的 Modem 有不同的特性，AT 指令的格式或者回应有所不同，但是这种特性在设计应用时不可能完全考虑和兼容。所以设计者在设计电话系统时，建立了一个虚拟电话系统，为该虚拟电话系统规定了标准的功能，上层的电话管理都是建立在这些标准的功能基础之上。而 RIL 则是将虚拟电话系统的标准功能转换成实际的所使用 的 Modem 的 AT 指令。

Android 设计者将电话系统设计成了三部分。

Andoird 的 Phone Service 其实是 PhoneApp。GSMPhone（CDMAPhone)是 Phone Service 核心的对象，他包含了如下的相关对象。



我们的分析任务就是要把这些对象的相互关系，及其对象间数据传递关系弄清楚。首先我们给出以下的 Android 电话系统的框架，以便对 Android 电话系统有个概要的认识，然后从数据流的角度，以及对象的引用关系来分析系统。下面是 android 电话系统整体框架图。

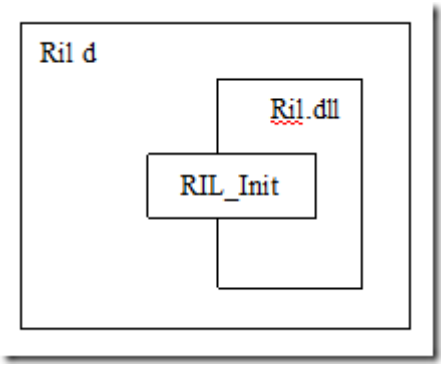


Android 核心分析（17） ------电话系统之 rilD

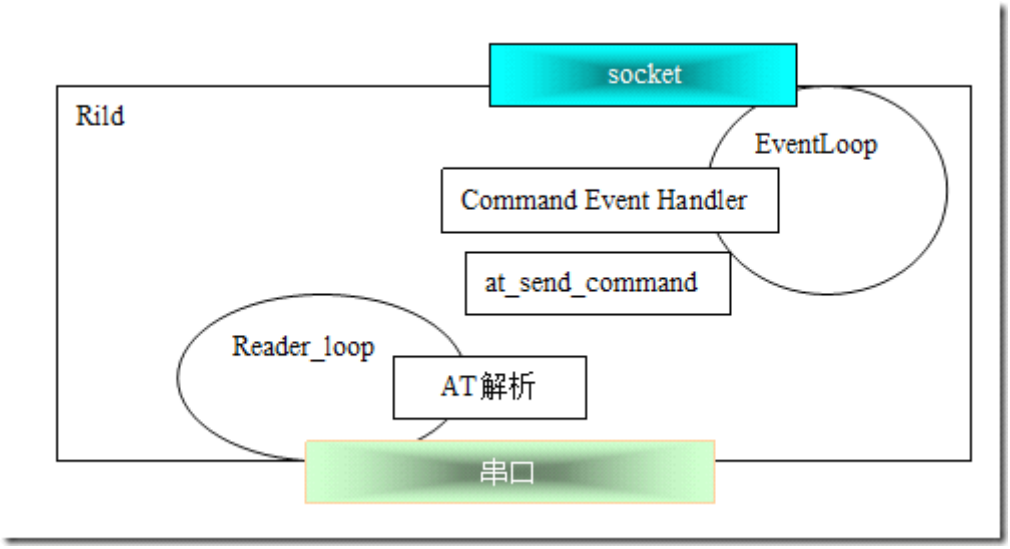
Android 电话系统之-rild
Rild 是 Init 进程启动的一个本地服务，这个本地服务并没有使用 Binder 之类的通讯手段，而是采用了 socket 通讯这种方式。RIL(Radio Interface Layer)

Android 给出了一个 ril 实现框架。由于 Android 开发者使用的 Modem 是不一样的，各种指令格式，初始化序列都可能不一样，GSM 和 CDMA 就差别更大了，所以为了消除这些差别，Android 设计者将 ril 做了一个抽象，使用一个虚拟电话的概念。这个虚拟电话对象就是 GSMPhone（CDMAPhone),Phon 对象所提供的功能协议，以及要求下层的支撑环境都有一个统一的描述，这个底层描述的实现就是靠 RIL 来完成适配。

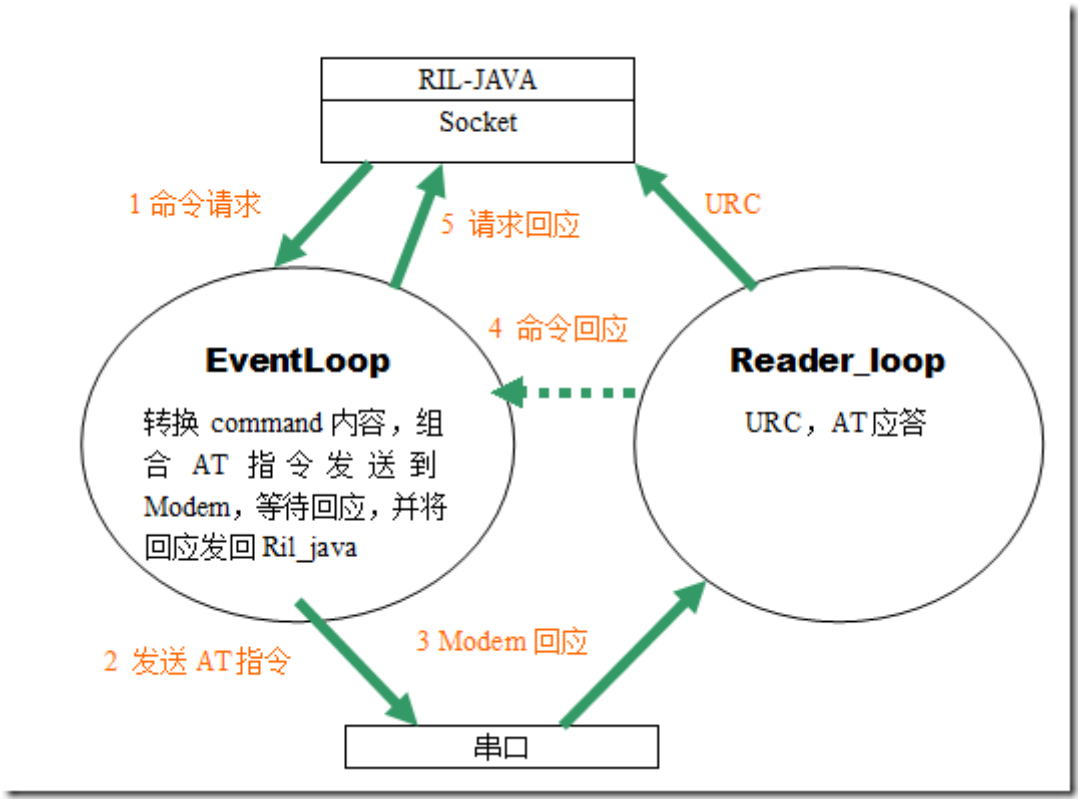
Andoid 将 RIL 层分为两个代码空间：RILD 管理框架，AT 相关的 xxxril.so 动态链接库。将ＲＩＬ独立成一个动态链接库的好处就是Ａｎｄｒｏｉｄ系统适应不同的Ｍｏｄｅｍ，不同的 Mode 可以有一个独立的 Ril 与之对应。从这个层面上看，Rild 更多是一个管理框架。



而 ril 是具体的 AT 指令合成者和应答解析者。从最基本的功能来讲，ril 建立了一个侦听 Socket，等待客户端的连接，然后从该连接上读取 RIL-Java 成传递来的命令并转化成 AT 指令发送到 Modem。并等待 Modem 的回应，然后将结果通过套接口传回到 Ril-Java 层。下图是 Ril-D 的基本框架：



下面的数据流传递描述图表描述了 RIL-JAVA 层发出一个电话指令的 5 步曲。



在 AT 通讯的过程中有两类响应：一种是请求后给出应答，一种是通知类，即为不请自来的，例如短信通知达到，我们称该类通知为 URC。在 Rild 中 URC 和一般的 Response 是分开处理的，概念上 URC 由 handleUnsolicited@Atchannel.c 处理，而 Response 由 handleFinalResponse 来处理。

1 Event Loop

Rild 管理的真正精髓在 ril.cpp,ril_event.cpp 中，在研究的过程中，可以看到设计者在抽象上所下的功夫，设计得很优美。Event Loop 的基本工作就是等待在事件端口（串口，Socket），一旦有数据到达就根据登记的 Event 回调函数进行处理。现在来看 Ril 设计者是如何建立一套管理框架来完成这些工作的？

1.1 Event 对象

Event 对象构成：（fd,index,persist,func,param）

fd 事件相关设备句柄。例如对于串口数据事件，fd 就是相关串口的设备句柄
index
persist 如果是保持的，则不从 watch_list 中删除。
func 回调事件处理函数
param 回调时参数

为了统一管理事件，Android 使用了三个队列：watch_list,timer_list,pending_list,并使用

了一个设备句柄池 readFDS。

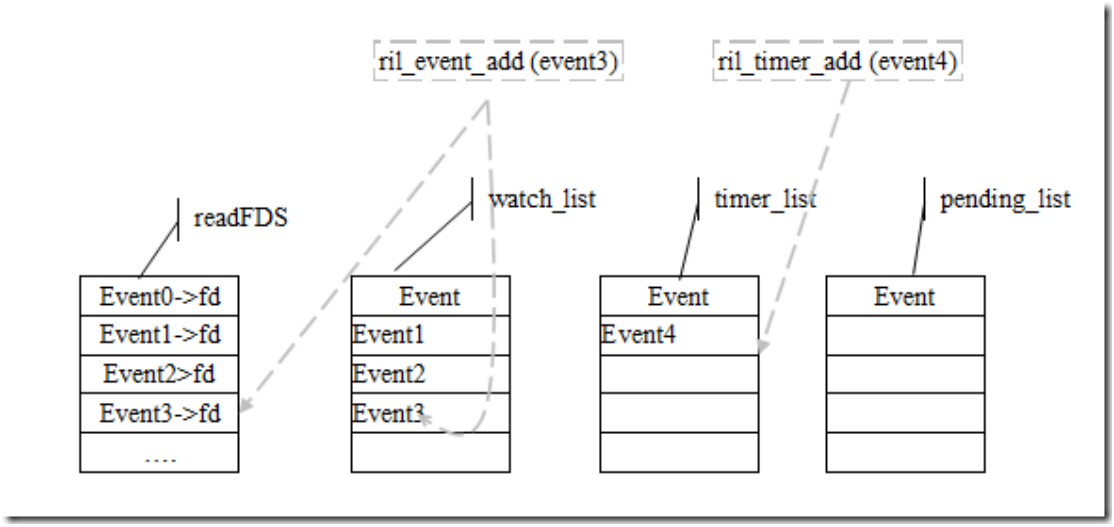
readFDS：是 Linux 的 fd_set，readFDS 保存了 Rild 中所有的设备文件句柄，以便利用 select 函数统一的完成事件的侦听。

watch_list：监测时间队列。需要检测的事件都放入到该队列中。

timer_list：timer 队列

pending_list:待处理事件队列，事件已经触发，需要所回调处理的事件。

事件队列队列的操作：ril_event_add,ril_event_del, ril_timer_add



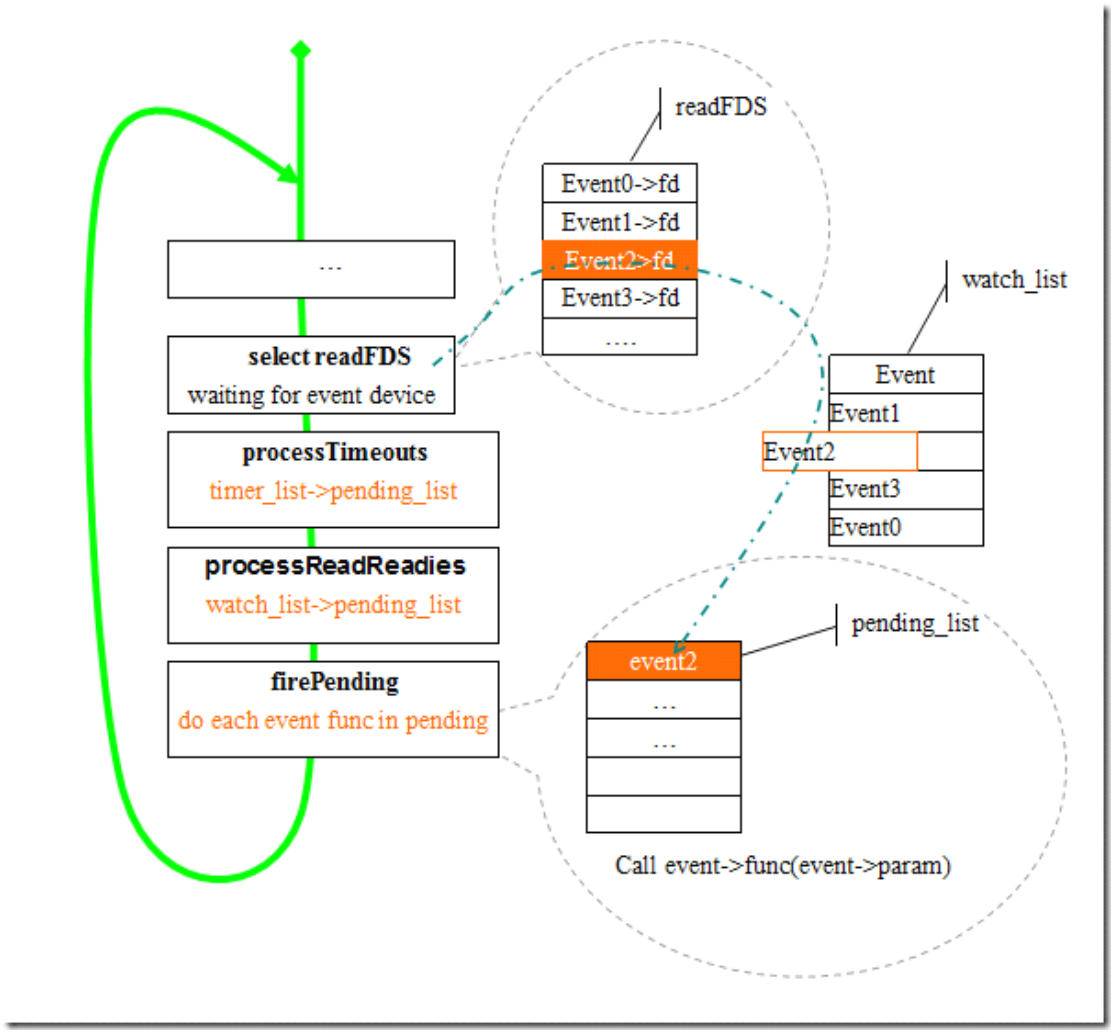
在添加操作中，有两个动作：

(1) 加入到 watch_list

(2) 将句柄加入到 readFDS 事件句柄池。

1.2 ril_event_loop()

我们知道对于 Linux 设备来讲，我们可以使用 select 函数等待在 FDS 上,只要 FDS 中记录的设备有数据到来，select 就会设置相应的标志位并返回。readFDS 记录了所有的事件相关设备句柄。readFDS 中句柄是在在 AddEvent 加入的。所有的事件侦听都是建立在 linux 的 select readFDS 基础上。



ril_event_loop 利用 select 等待在 readFDS（fd_set）上，当 select 设备有数据时，ril_event_loop 会从 select 返回，在 watch_list 中相应的 Event 放置到 pend_list，如果 Event 是持久性的则不从 watch_list 中删除。然后 ril_event_loop 遍历 pengding_list 处理 Event 事件，发起事件回调函数。

1.3 几个重要的 Event

上面分析了 ril-d 的框架，在该框架上跑的事件有什么

（1）s_listen_event- （s_fdListen,listenCallback）

listenCallback 处理函数，

接收客户端连接：s_fdCommand=accepte(..)

添加 s_commands_event()

重新建立 s_listen_event，等待下一次连接

（2） s_command_event(s_fdCommand,ProcessCommandsCallback)

从 fdCommand　Socket 连接中读取 StreamRecord

使用 ProcessCommandBufer 处理数据

s_listen_event 在大的功能上处理客户端连接（Ril-JAVA 层发起的 connect）,并建立 s_commands_event 去处理 Socket 连接发来的 Ril 命令。ProcessCommandBufer 实际上包含了 Ril 指令的下行过程。

1.4 下行命令翻译及其组织@ProcessCommandBuffer
RIL_JAVA 传递的命令格式：Parcel ，由命令号，令牌，内容组成。RIL_JAVA 到达 RIL_C 时转为构建本地 RequestInfo，并将被翻译成具体的 AT 指令。由于每条 AT 命令的参数是不同的，所以对不同的 AT 指令，有不同的转换函数，在此 Android 设计在这里做了一个抽象，做了一个分发框架，通过命令号，利用 sCommand 数组，获得该命令的处理函数。

sComand[]={

<...>

}

sComand 存在于 Ril_command.h 中。

&sComand[]=

<

{RIL_REQUEST_GET_IMEI, dispatchVoid, responseString},

{RIL_REQUEST_DIAL, dispatchDial, responseVoid},

{….}

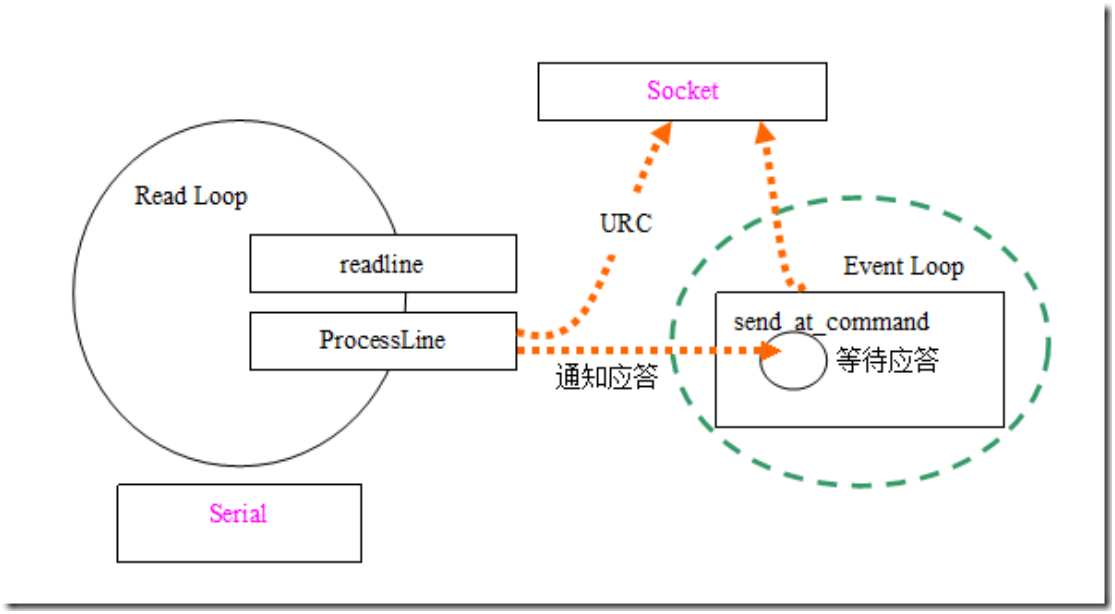
>

dispatchXxx 函数一般都放在在 Reference-ril.c 中，Reference-ril.c 这个就是我们需要根据不同的 Modem 来修改的文件。

1.5 send_at_command 框架
send_at_command 是同步的，命令发送后，send_at_command 将等待在 s_commandcond，直到有 sp_response->finalResponse。

2 read loop@Atchannel.c

Read loop 是解决的问题是：解析从 Modem 发过来的回应。如果遇到 URC 则通过 handleUnsolicited 上报的 RIL_JAVA。如果是命令的应答，则通过 handleFinalResponse 通知 send_at_command 有应答结果。



对于 URC，Rild 同样使用一个抽象数组@Ril.CPP.

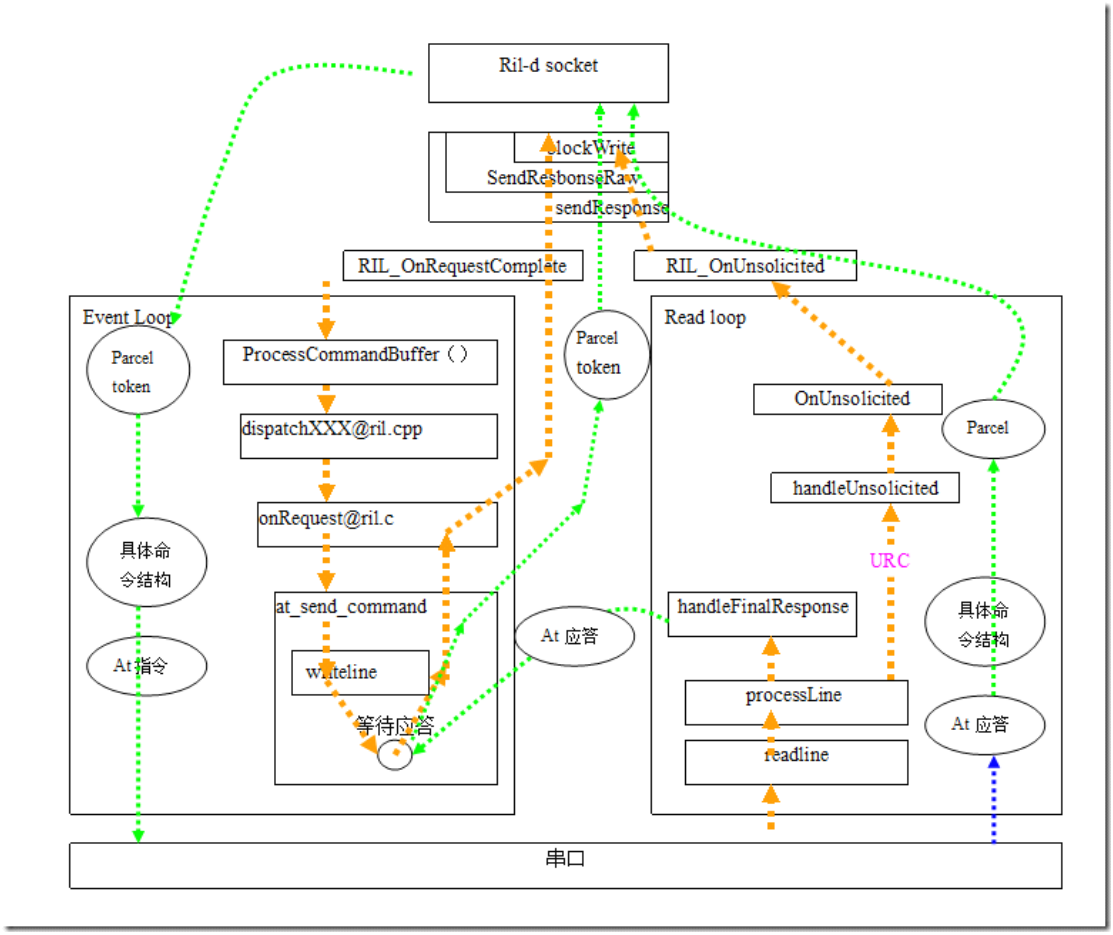
static UnsolResponseInfo s_unsolResponses[] = {

#include "ril_unsol_commands.h"

};

并利用 RIL_onUnsolicitedResponse 将 URC 向上层发送。

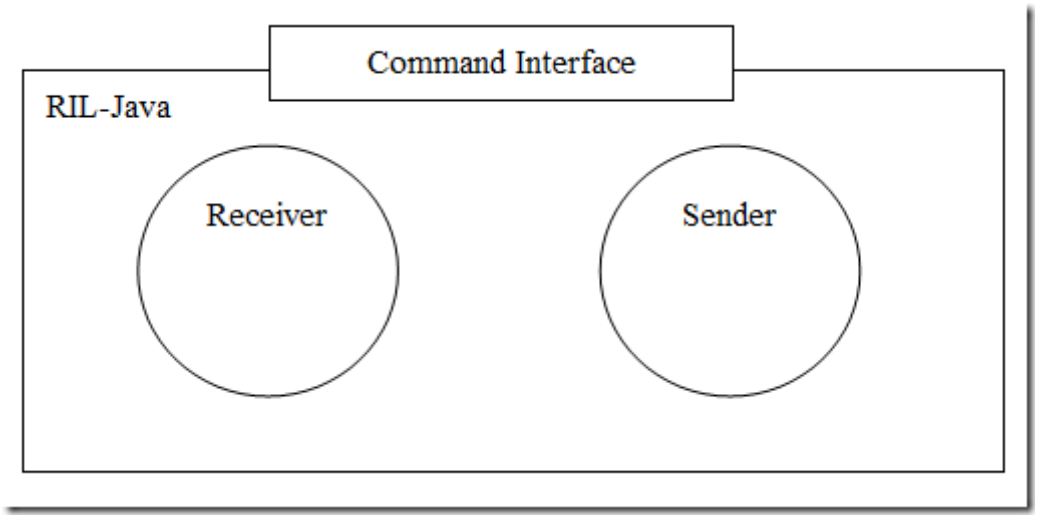
3 Ril-d 的整体数据流及其控制流示意图



Android 核心分析（18）-----Android 电话系统之 RIL-Java

Android RIL-Java

RIL-Java 在本质上就是一个 RIL 代理，起到一个转发的作用，是 Android Java 概念空间中的电话系统的起点。在 RIL-D 的分析中，我们知道 RILD 建立了一个侦听套接口，等待 RIL-Java 的连接。一旦连接成功，RIL-JAVA 就可发起一个请求，并等待应答，并将结构发送到目标处理对象。在 RIL-Java 中，这个请求称为 RILRequest。为了直观起见，我还是不厌其烦的给出 RIL-Java 的框架图。



RIL-Java 的大框架包含了四个方面：

Receiver，Sender，CommandInterface，异步通知机制

（1） Command Interface

在 ril.java 源代码中，我们可以看到 RIL-JAVA 对象提供了如下的 Command Interface：

…

getIccCardStatus

getCurrrentCalls

dial

acceptCall

rejectCall

sendDTMF

sendSMS

setupDataCall

setRadioPower

…

为什么要定义这些接口呢？这函数接口不是凭空捏造出来的，这些都是电话的基本功能的描述，是对 Modem AT 指令的提炼抽象。大多数 Modem 都是根据通讯协议提供接口，我们如果不熟悉通讯协议，请参阅 3GPP 的相关文档，以及自己使用的 Modem 的 SPEC 说明。

V.25ter AT Commands

3GPP 07.07 AT Comamnds-General commands

3GPP 07.07 AT Comamnds-Call Control commans

3GPP 07.07 AT Comamnds-Network Service related commands

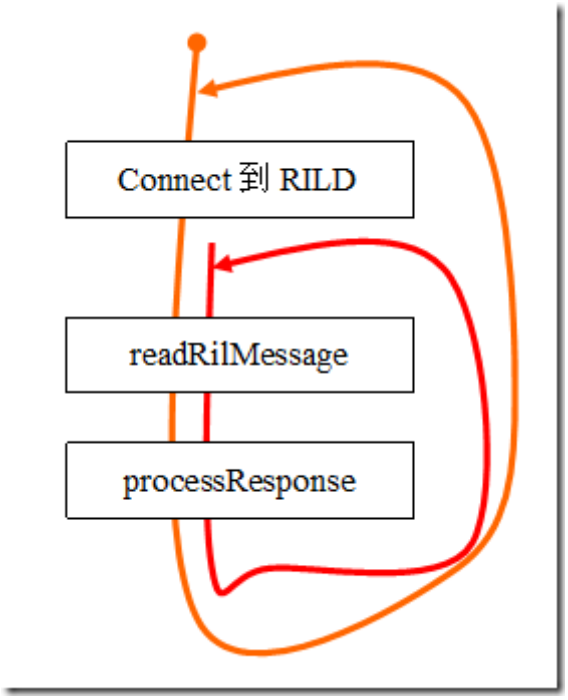
3GPP 07.07 AT Comamnds-MT control and status command

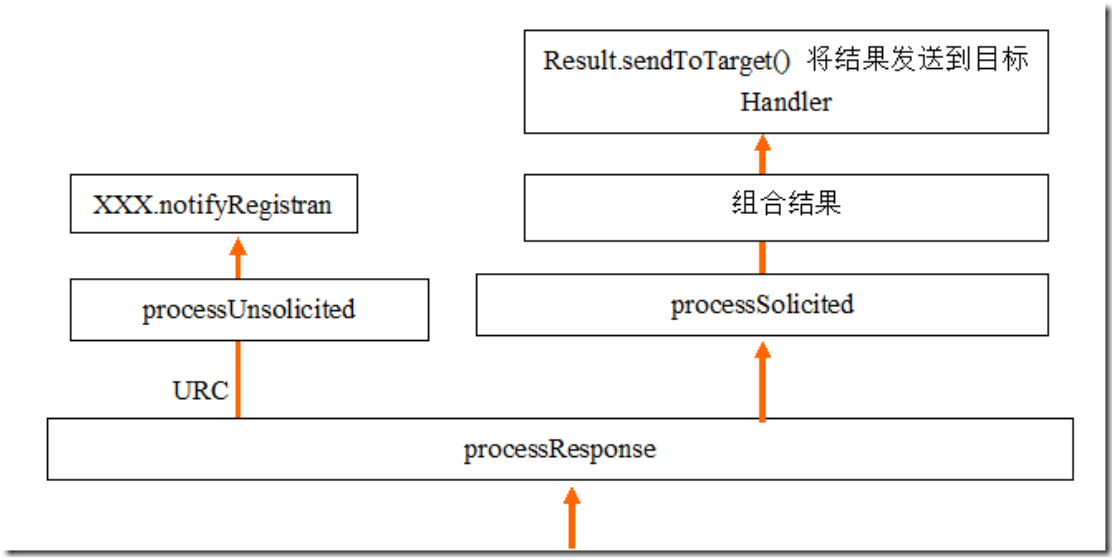
3GPP 07.07 AT Comamnds-GPRS Commands

3GPP 07.07 Mobile Termination Errors

3GPP 07.05 SMS AT Commands

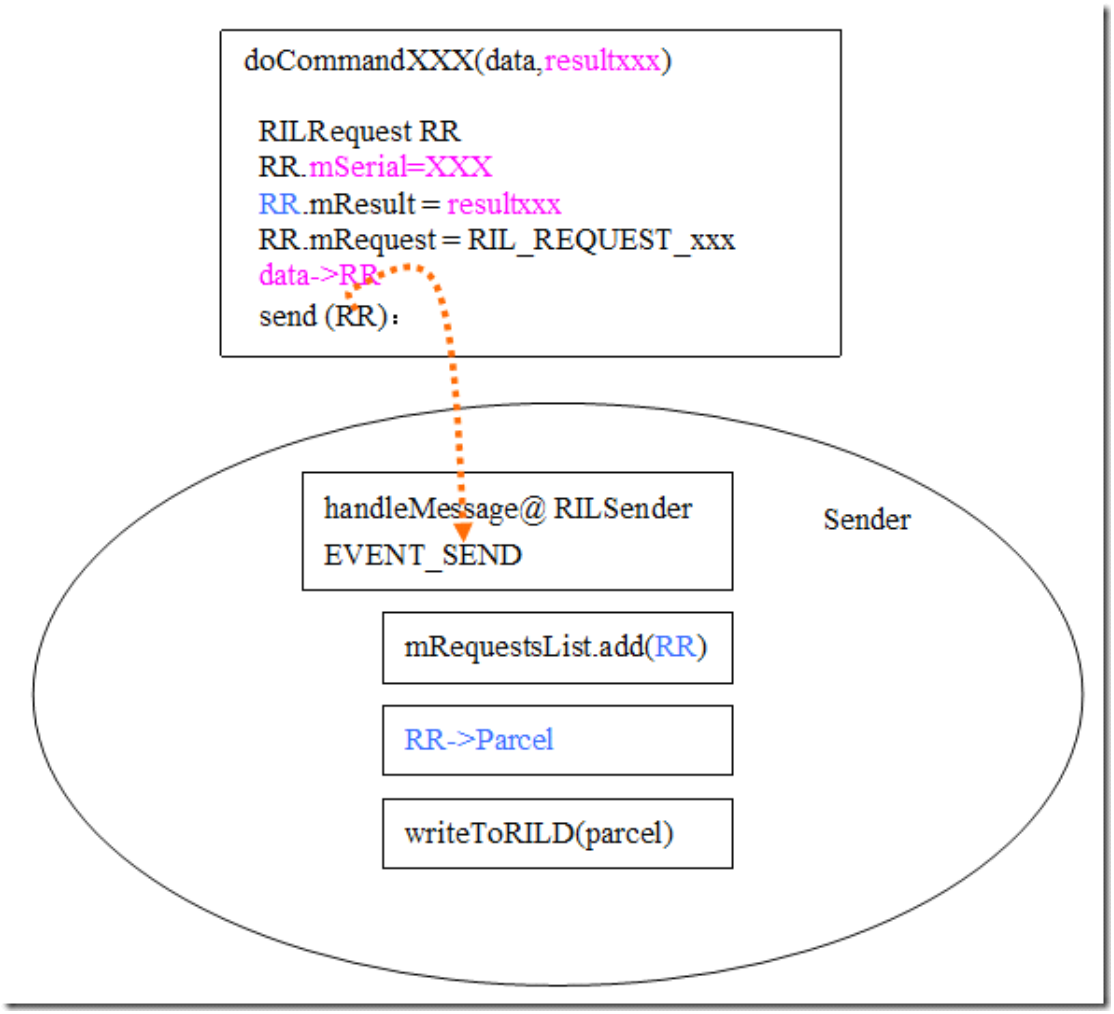
（2）Receiver





Receiver 连接到 RILD 的服务套接口，接收读取 RILD 传递过来的 Response Parcel。Response 分为两种类型，一种是 URC，一种是命令应答。对于 URC 将会直接分发到通知注册表中的 Handler。而命令应答则通过 Receiver 的异步通知机制传递到命令的发送者进行相应处理。

（3）Sender



Sender 应该分为两部分架构，

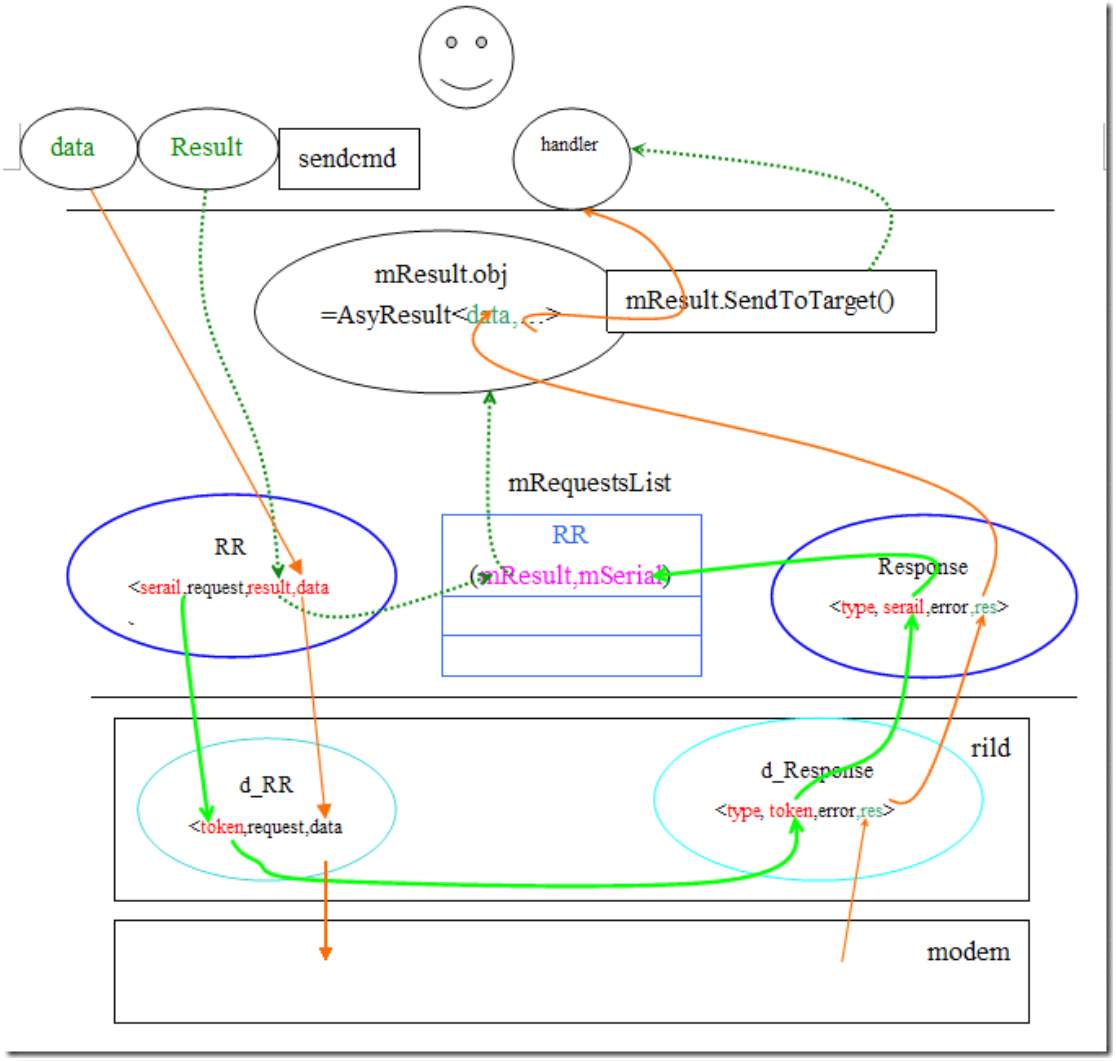
上层函数调用 Command Interface 将请求消息发送到 Sender 的架构。

Sender 接收到 EVENT_SEND 消息后，将请求发送到 RILD 的架构。

（4）异步应答框架

对于异步应答来讲，命令的发起者发送后，并不等待应答就返回，应答的回应是异步的，处理结果通过消息的方式返回。站在设计者的角度思考如何设计合适的框架来完成异步通讯的功能呢？对于异步系统我们首先应该考虑的是如何标识命令和结果，让命令和结果有一个对应关系，还有命令没有响应了，如何管理命令超时？让我们来看看 Android 设计者如何完成这些工作。

Android 设计者利用了 Result Message 和 RILRequest 对象来完成 Request 和 Result 的对应对于关系。在上层做调用的时候生成 Result Message 对象传递到 ril_java，并在 Modem 有应答后，通过 Result Message 对象带回结果。如何保证该应答是该 RILRequest 的呢？Android 设计者还提供了一个 Token（令牌）的概念。在源代码中 RILRequest 的 mSerail 就用作了 Token。Token 用来唯一标识每次发送的请求，并且 Token 将被传递到 RILD，RILD 在组装应答是将 Token 写入，并传回到 ril-java，ril-java 根据该 Token 找到相应的 Request 对象。



（4.1）RIL 命令的发送模式

协议的真正实现是在 rild 中，RIL-JAVA 更多的是一个抽象和代理，我们在研究源代码的过程中就会体会到到 RIL-JAVA 中的命令函数都有一个共同的框架。

SendXxxCmd(传入参数 Data，传出参数 result){

组合 RILRequest(请求号，result，mSerail)

Data->RR

send(RILRequest)： Message

}

1)RILRequest

请求号：

request 将传递到 RILD 用以标识命令，request 代表某个功能。例如拨叫的 request 号为：RIL_REQUEST_DIAL 。 在 libs/telephony/ril_commands.h 有 定 义 。 RILRequest.obtain@RILRequest 根据命令请求号,传入参数 Result Message，mSerail 构造了一个 RILRequest。Result Message 将带回应答信息回到命令的发起者。

mSerail：

Android 使用了一个 RILRequest 对象池来管理 Andoird RILRequest。mSerail 是一个递增的变量，用来唯一标识一个 RILRequest。在发送时正是用了该变量为 Token,在 rild 层看到的 token 就是该 mSerail。

EVENT_END:

EVENT_END@handleMessage@RILSender@RIL.java

2)发送步骤：

第一步：

生成 RILRequest，此时将生成 m_Serial（请求的 Token)并将请求号，数据，及其 Result Message 对象填入到 RILRequest 中

第二步：

使用 send 将 RILRequest 打包到 EVENT_SEND 消息中发送到到 RIL Sender Handler，

第三步：

RilSender 接收到 EVENT_SEND 消息，将 RILRequest 通过套接口发送到 RILD，同时将 RILRequest 保存在 mRequest 中以便应答消息的返回。

（4.2） 接收模式

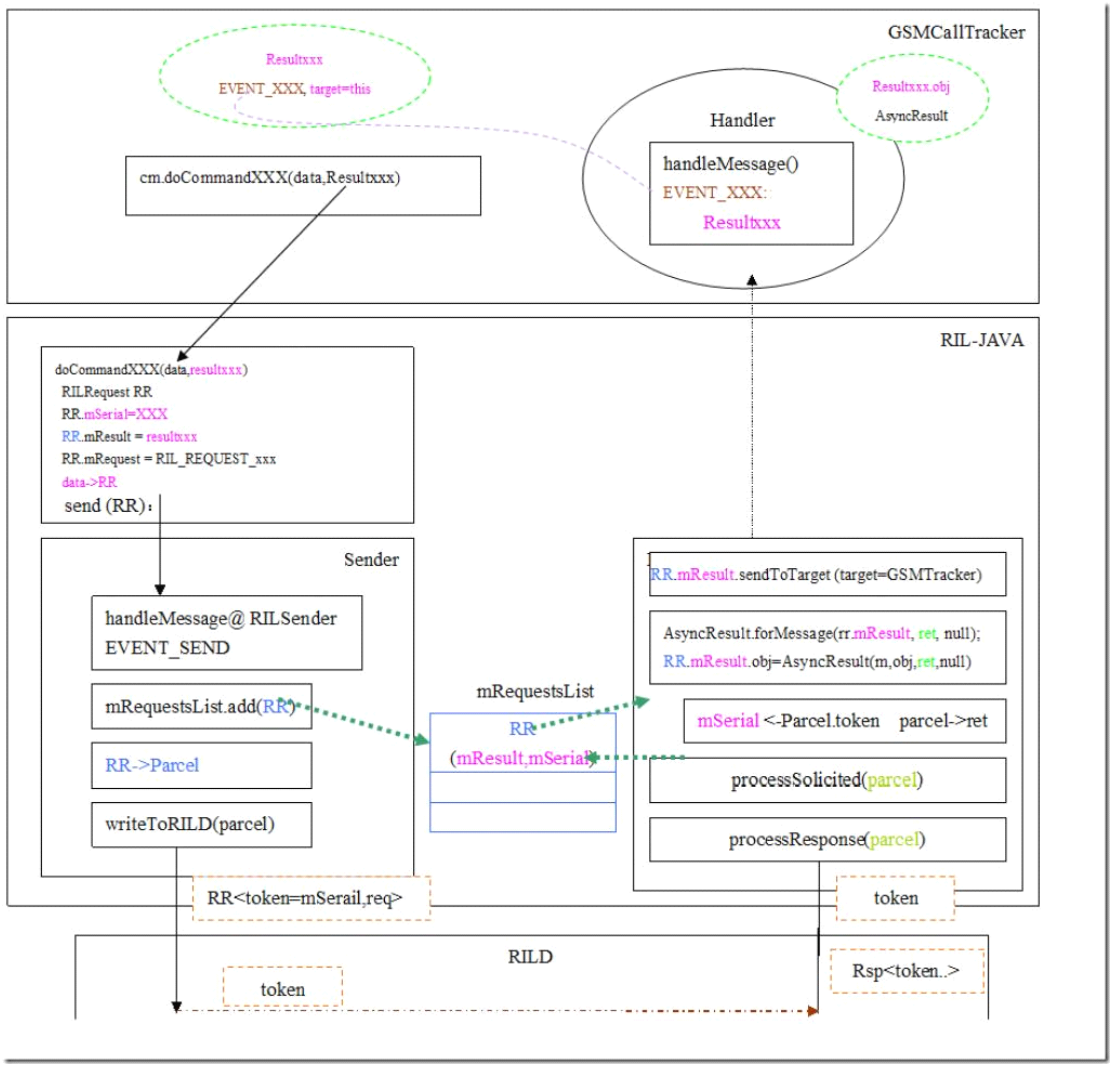
第一步：分析接收到的 Parcel，根据类型不同进行处理。

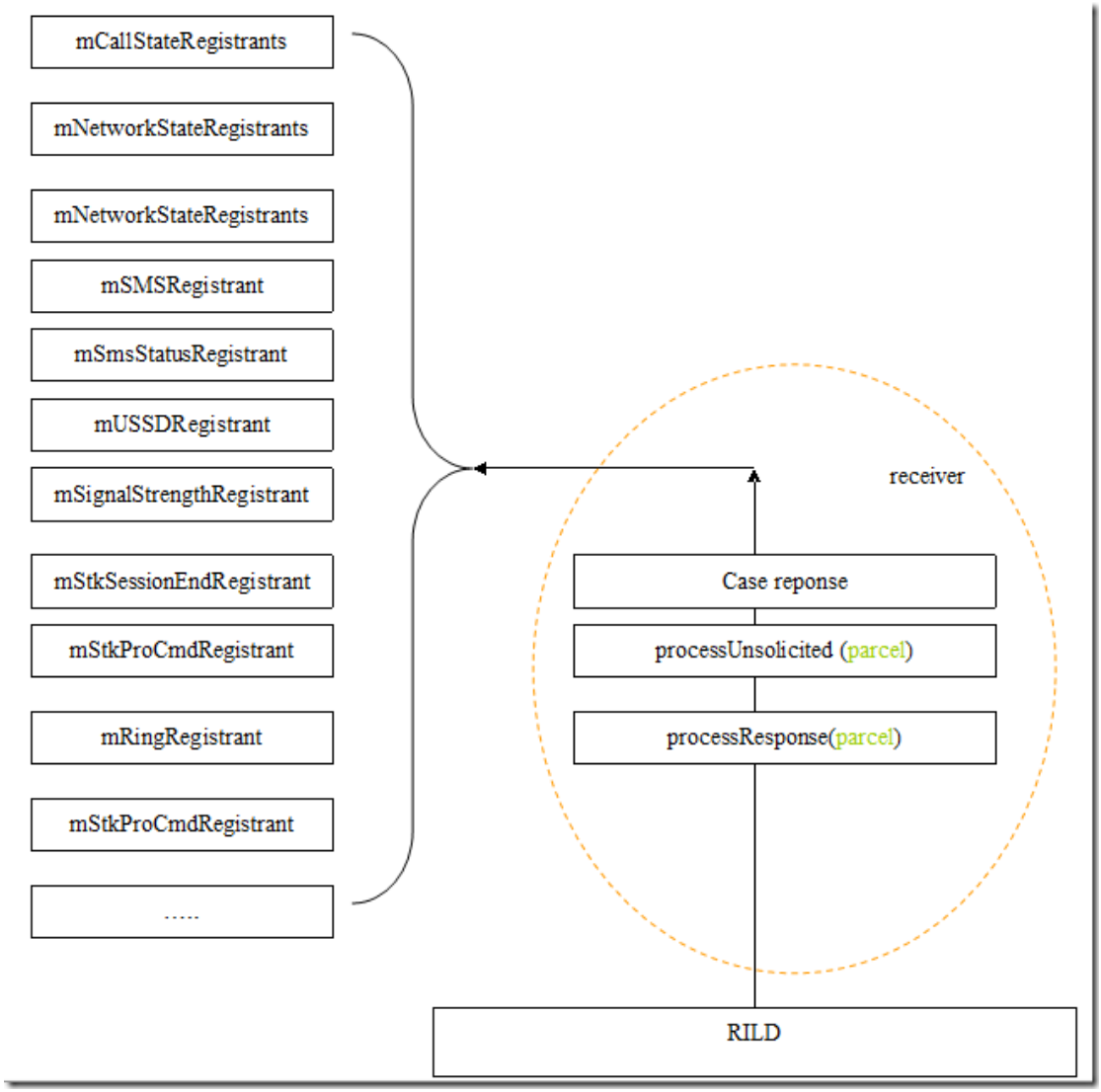
第二步：根据数据中的 Token（mSerail),反查 mRequest,找到对应的请求信息。

第三步：将是数据转换成结果数据。

第四步：将结果放在 RequestMessage 中发回到请求的发起者。

4.3)详细的 GSMCallTracker,RIL-Java 函数对照



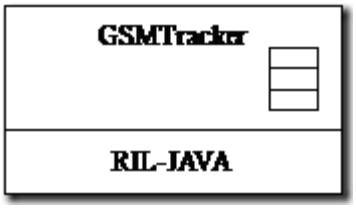


Android 核心分析（19）----电话系统之 GSMCallTacker

Android 电话系统之 GSMCallTracker
通话连接管理

GSMCallTracker 在本质上是一个 Handler。

GSMCallTracker 是 Android 的通话管理层。GSMCallTracker 建立了 ConnectionList 来管理现行的通话连接，并向上层提供电话调用接口。



在 GSMCallTracker 中维护着通话列表：connections。顺序记录了正连接上的通话，这些通话包括：ACTIVE，DIALING，ALERTING，HOLDING，INCOMING，WAITING 等状态的连接。GSMCallTracker 将这些连接分为了三类别进行管理：

RingingCall: INCOMING ,WAITING

ForegourndCall: ACTIVE, DIALING ,ALERTING

BackgroundCall: HOLDING

上层函数通过 getRingCall()，getForegrouandCall()等来获得电话系统中特定通话连接。

为了管理电话状态，GSMCallTracker 在构造时就将自己登记到了电话状态变化通知表中。RIL-Java 一收到电话状态变化的通知，就会使用 EVENT_CALL_STATE_CHANGE 通知到 GSMCallTacker

在一般的实现中，我们的通话 Call Table 是通过 AT+CLCC 查询到的，CPI 可以通知到电话的改变，但是 CPI 在各个 Modem 的实现中差别比较大，所以参考设计都没有用到 CPI 这样的电话连接改变通知，而是使用最为传统的 CLCC 查询 CALL TABLE。在 GSMTracker 中使用 connections 来管理 Android 电话系统中的通话连接。每次电话状态发生变化是 GSMTracker 就会使用 CLCC 查询来更新 connections 内容，如果内容有发生变化，则向上层发起电话状态改变的通知。

1 RIL-JAVA 中发起电话连接列表操作
在 RIL-JAVA 中涉及到 CurrentCallList 查询的有以下几个操作：

（1）hangup

（2）dial

（3）acceptCall

（4）rejectCall

在 GSMcallTracker 在发起这些调用的时候都有一个共同的 ResultMessage 构造函数：obtainCompleteMessage()。obtainCompleteMessage()实际上是调用：

obtainCompleteMessage(EVENT_OPERATION_COMPLETE)

这就意味着在这些电话操作后，GSMCallTracker 会收到 EVENT_OPERATION_COMPLETE 消息，于是我们将目光转移到 handleMessage()@GSMCallTracker 的 EVENT_OPERATION_COMPLETE 事件处理：operationComplete@GSMCallTracker。

operationComplete（）操作会使用 cm.getCurrentCalls(lastRelevantPoll)调用，向 RILD 发起 RIL_REQUEST_GET_CURRENT_CALLS 调用，这个最终就是向 Modem 发起 AT+CLCC，获取到真正的电话列表。

2 在 RILD 中，引起 getCurrentCalls 调用
（1）在 RILD 中，收到 URC 消息：

+CRING

RING

NO CARRIER

+CCWA

将会使用 RIL_onUnsolicitedResponse(RIL_UNSOL_RESPONSE_CALL_STATE_CHANGED)，主动向 ril-java 上报 RIL_UNSOL_RESPONSE_CALL_STATE_CHANGED 消息。

（2） 在处理 requestCurrentCalls 时，使用 CLCC 查询通话连接（CALL TABLE）后，如何发现有 call Table 不为空则开启一个定时器，主动上报 RIL_UNSOL_RESPONSE_CALL_STATE_CHANGED 消息，直到没有电话连接为止。

在 RIL-Java 层收到 RIL_UNSOL_RESPONSE_CALL_STATE_CHANGED 这个 URC，并利用 mCallStateRegistrants.notifyRegistrants(new AsyncResult(null, null, null))来通知电话状态的变化，此时 GSMTracker 会接收到 EVENT_CALL_STATE_CHANGE 消息，并使用

pollCallsWhenSafe()-> cm.getCurrentCalls(lastRelevantPoll);

来发起查询，并更新 JAVA 层的电话列表。

3 handlePollCalls 电话列表刷新
首先我们来看看是什么引起了 handlePollCalls 的调用。

上面的 1,2 分析了，Android 电话系统中所有引起电话连接列表更新的条件及其处理。他们共同的调用了 cm.getCurrentCalls(lastRelevantPoll) 来完成电话列表的获取。

lastRelevantPoll = obtainMessage(EVENT_POLL_CALLS_RESULT)

我们这里就从可以看到获取到的电话列表 Result 使用 handlePollCalls 进行了处理。Result 实际上是一个 DriverCall 列表，handlePollCalls 的工作就是将当前电话列表与 RIL-Java 的电话列表对比，使用 DriverCall 列表更新 CallTracker 的电话列表 connections,并向上传递电话状态改变的通知。

Android 核心分析（20)----Android 应用程序框架之无边界设计意图

Android 应用程序框架

1 无边界设计理念

Android 的应用框架的外特性空间的描述在 SDK 文档（http://androidappdocs.appspot.com/guide/topics/fundamentals.html#acttask）有十分清楚的描述，Android 应用的基本概念，组件生命周期等等有详细的描述。在外特性空间中，Android 提供了 Activity,Service,Broadcast receivers，Content Provider,Intent，task 等概念，我在这里不讨论这些概念定义，因为 SDK 文档已经讲得够详细。

在阅读 SDK 文档和研究 Activity 这个概念时，我感觉到了在 Android 中若隐若现的 Android 自由无边界这个设计意图。Android 的应用只是一个虚的概念，并没有实际的入口，这个不像 Window 平台上的应用程序的概念，Android 更多的是提供组件（Components）的概念。为什么要虚化应用的概念？我想这个虚化就是自由无边界设计意图的直接体现。突出请求和服务，突出组件个体，弱化边界，系统的各个组件可以自由的无边界的交流，服务请求者直接发出请求，不论这个对象在何处和属于谁的，组件是自由独立的个体，一个应用程序可以直接请求使用其他的应用的的组件，这个是 Android 应用框架设计的核心理念，其他的一切都是在为这个核心理念服务。

让程序员忽略应用的概念，甚至彻底的抛弃进程这样的概念，程序员看到的就是一个一个的组件，应用程序员利用这些组件来架构成一个所谓的应用，那么设计者首先要考虑的是什么呢？我想应该是一个抽象的应用模型，在这个模型下产生概念和接口。

我们知道 MicroSoft 提出了 Application，Windows 的概念，有前景应用（Foreground Application）概念，MicroSoft 的应用模型中用户交互则完全交给了 Window，各种界面的呈现都是属于这个应用的是孤立的，应用程序之间的各个构成对象不能相互访问，最多提供一个进程间通讯机制，那个也是应用程序层面的。虽然 Microsoft 后来也提出了组件，分布式组件等概念，但是这些不是根植在 Windows 系统中，而 Android 则是彻底的组件化思想构建，一开始的应用程序概念就是 Activity，Service,Broadcast receivers，Content Provider,Intent，Task。这些概念体现了一个人机交互的模型本质：

界面呈现

发起请求,响应请求

内容交互

消息接收处理

Activity 是 Android 应用的核心概念，简而言之 Activity 为用户交互管理者，有一个可视界面呈现，而 Service 跟 Activity 的区别是他在后台运行，没有界面呈现。而 Intent 的意义是意图，他在 Android 的概念空间中，代表消息，这个消息代表了请求的意图。

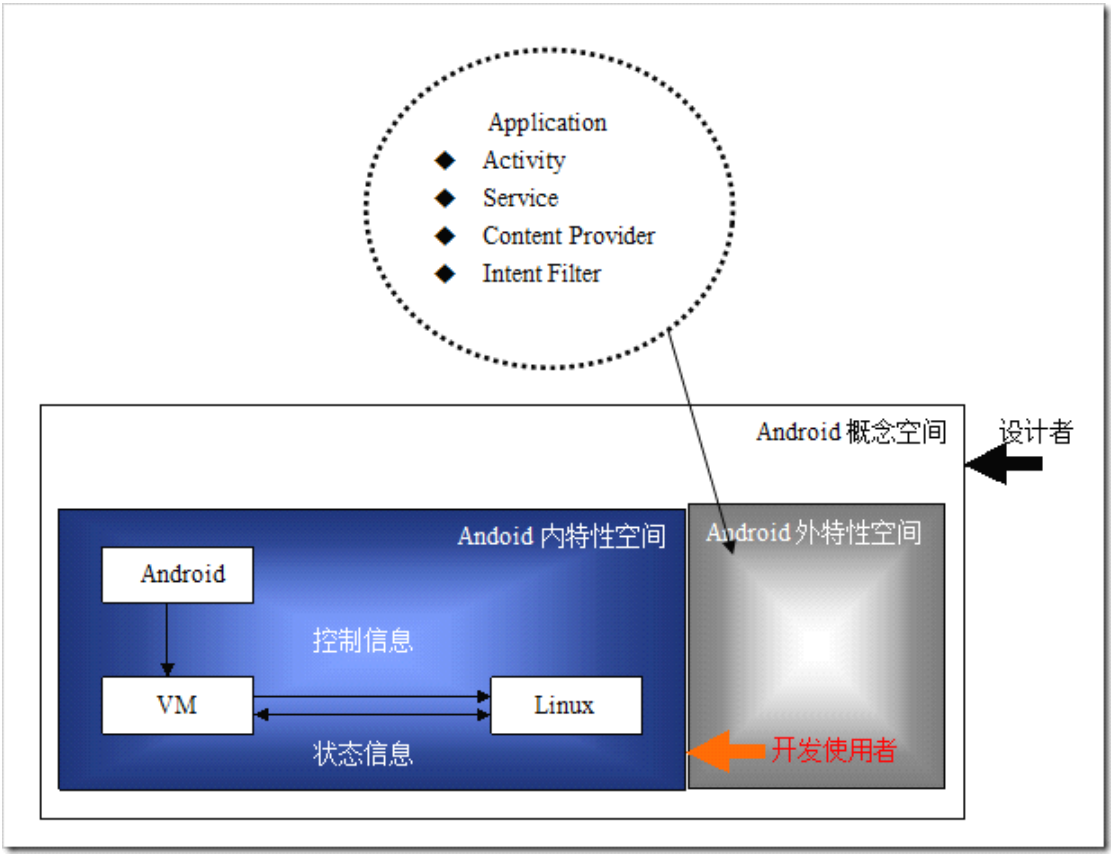
Activity 可以到处存在，提供服务，消除空间差别，Activity 是一个独立的个体，更能表现面向对象的实质。这个个体需要接受另外的个体的消息，可以随时发起对另外一个个体的请求。个体是自由的，Android 中你可以开始一个 Activity，但是没有权利消灭一个 Activity，这是个体权利的体现，个体的消灭是由系统决定的，这个就是 Android 中 Activity 蕴含的人文意义。

Android 核心分析（21）----Android 应用框架之 AndroidApplication

Android Application

Android 提供给开发程序员的概念空间中 Application 只是一个松散的表征概念，没有多少实质上的表征。在 Android 实际空间中看不到实际意义上的应用程序的概念,即使有一个叫 Application 的类，这个也就是个应用程序上下文状态，是一个极度弱化的概念。Application 只是一个空间范畴的概念， Application 就是 Activity， Service 之类的组件上下文描述。Application 并不是 Android 的核心概念， 而 Activity 才是 Android 的核心概念。

从 Android 的 SDK 文档中，我们知道一般情况 Android 应用程序是由以下四种组件构造而成的：Activity，Broadcast Intent Receiver，服务（Service），内容提供器（Content Provider）。我们可以使用下面的图来表示一下 Android 的概念空间。这些组件依附于应用程序中，应用程序并不会一开始就建立起来，而是在这些组件建立起来后，需要运行时，才开始建立应用程序对象。



2.1 应用进程名称

为什么要从应用进程名称开始？作为内核研究，我们还是回到问题的最本质处：不管 Activity，Service 等组件如何设计和运行，它要提供服务，就必须要依附在 Linux 的进程上，建立消息循环，组件才能够真正的运作。Activity 实例是如何 Hosting 在 Linux 进程上的？

这个是我们首先想要弄明白的。

我们在的项目中看到 android:process="string"这个定义。

allowClearUserData=["true" | "false"]
android:allowTaskReparenting=["true" | "false"]
android:backupAgent="string"
…

android:label="string resource"
android:manageSpaceActivity="string"
android:name="string"
android:permission="string"
android:persistent=["true" | "false"]
android:process="string"
android:restoreAnyVersion=["true" | "false"]
android:taskAffinity="string"
android:theme="resource or theme" >
. . .

在 SDK 用已经描述的很清楚到了。

android:process

The name of a process where all components of the application should run. Each component can override this default by setting its own process attribute.

By default, Android creates a process for an application when the first of its components needs to run. All components then run in that process. The name of the default process matches the package name set by the element.

By setting this attribute to a process name that's shared with another application, you can arrange for components of both applications to run in the same process — but only if the two applications also share a user ID and be signed with the same certificate.

为什么要提出这么一个定义？android:process 名称。

默认状态下，Activity Manager Service 在应用程序的第一个组件需要运行时将会为应用程序建立一个进程，而这个进程的名字就是 android:process=”string”所指定，缺省的是应用程序包的名字。该进程一旦建立，后面的该应用的组件都将运行在该进程中，他们绑定的根据就是这个 Android：Process 指定的名称，因为在他们都在同一个应用程序包里，也就具有了同样的进程名字，于是他们都托管在了同一进程中。组件将通过 ClassLoader 从 Package 中获取到应用程序的信息。

在建立 Actvitiy 时，如果在应用进程端没有应用对象，系统在该过程中利用 makeApplication 建立一个 Application 对象，实例化"android.app.Application"，建立一个应用程序上下文完成例如资源，package 等信息管理。

2.2 ActivityThread 运行框架

在分析中，我们可以看到真正对应应用进程的不是 Application 而是 ActivityThread。我们从实际的应用堆栈可以看到：

NaiveStart.main()

ZygoteInit.main

ZygoteInit$MethodAndArgsCall.run

Method.Invoke

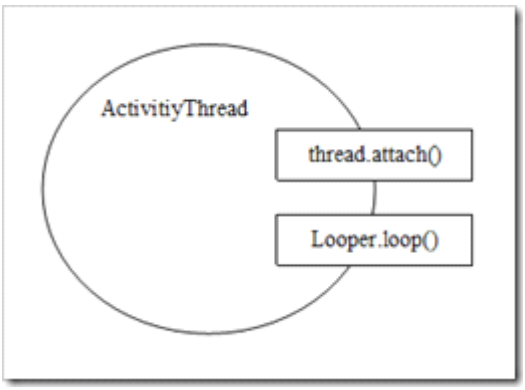
method.invokeNative

ActivityThread.main()

Looper.loop()

....

每个应用程序都以 ActivityThread.main()为入口进入到消息循环处理。对于一个进程来讲，我们需要这个闭合的处理框架。



ActivitiyThread 是应用程序概念空间的重要概念，他建立了应用进程运行的框架，并提供了一个 IActivityThread 接口作为与 Activity Manager Service 的通讯接口.通过该接口 AMS 可以将 Activity 的状态变化传递到客户端的 Activity 对象。

2．3 ActivitiyThread 的建立

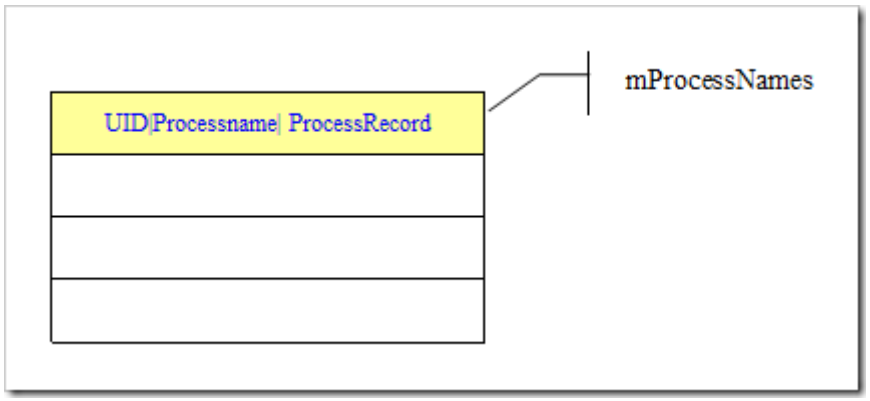
为了叙述的方便我将 Actvitiy Manager Service 简写成 AMS。

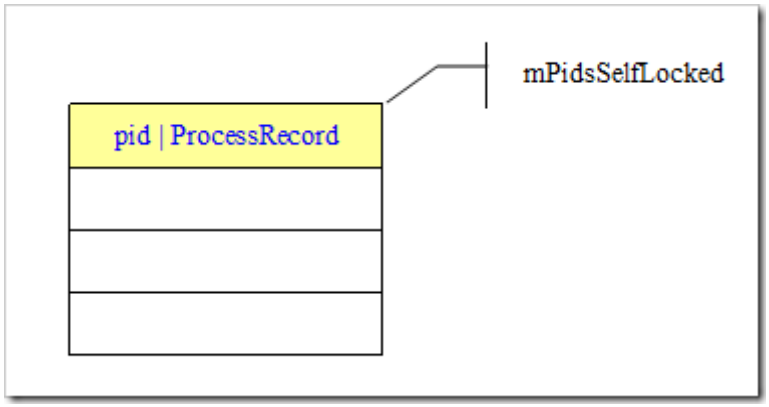
在 AMS 中关于应用程序的概念是 ProcessRecord，请求都是从 Activity，Service…等开始的，在 Activity 需要 Resume 时，此时如果与 Activity 相关的应用进程没有起来，AM 则启动应用进程。

AMS 与应用进程的绑定分为两个部分，第一部分就是 AM 建立应用进程，第二部分就是应用进程 Attach 到 AM，与 AM 建立通讯通道。

1）创建建立进程：startProcessLocked(processName,Appinfo.uid)。该函数在 StartSecificActivityLocked 等调用。

（1）建立 ProcessRecord 对象 app,并将该对象添加到 mProcessNames 中。应用对象在 mProcessNames 中使用应用名字和 uid 来标识自己。如果在同一个 Package 中的 Activity,如果都使用默认设置，那么这些 Activity 都会托管在同一个进程中，这是因为他们在带的 ApplicationInfo 中的 ProcessName 都是一样的。

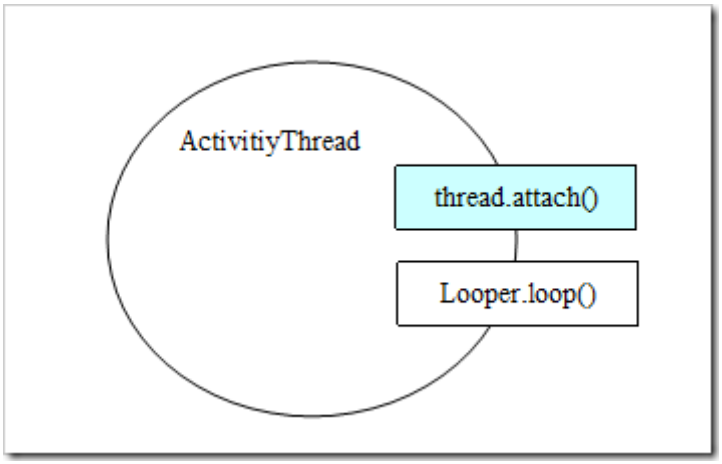




mPidsSelfLocked 数组记录了 PID，这个将会在应用进程跑起来后，将自己 Attach 到 AM 时，根据 pid 找到自己的前世：ProcessRecord.

2）android.app.ActivityThread 进程启动

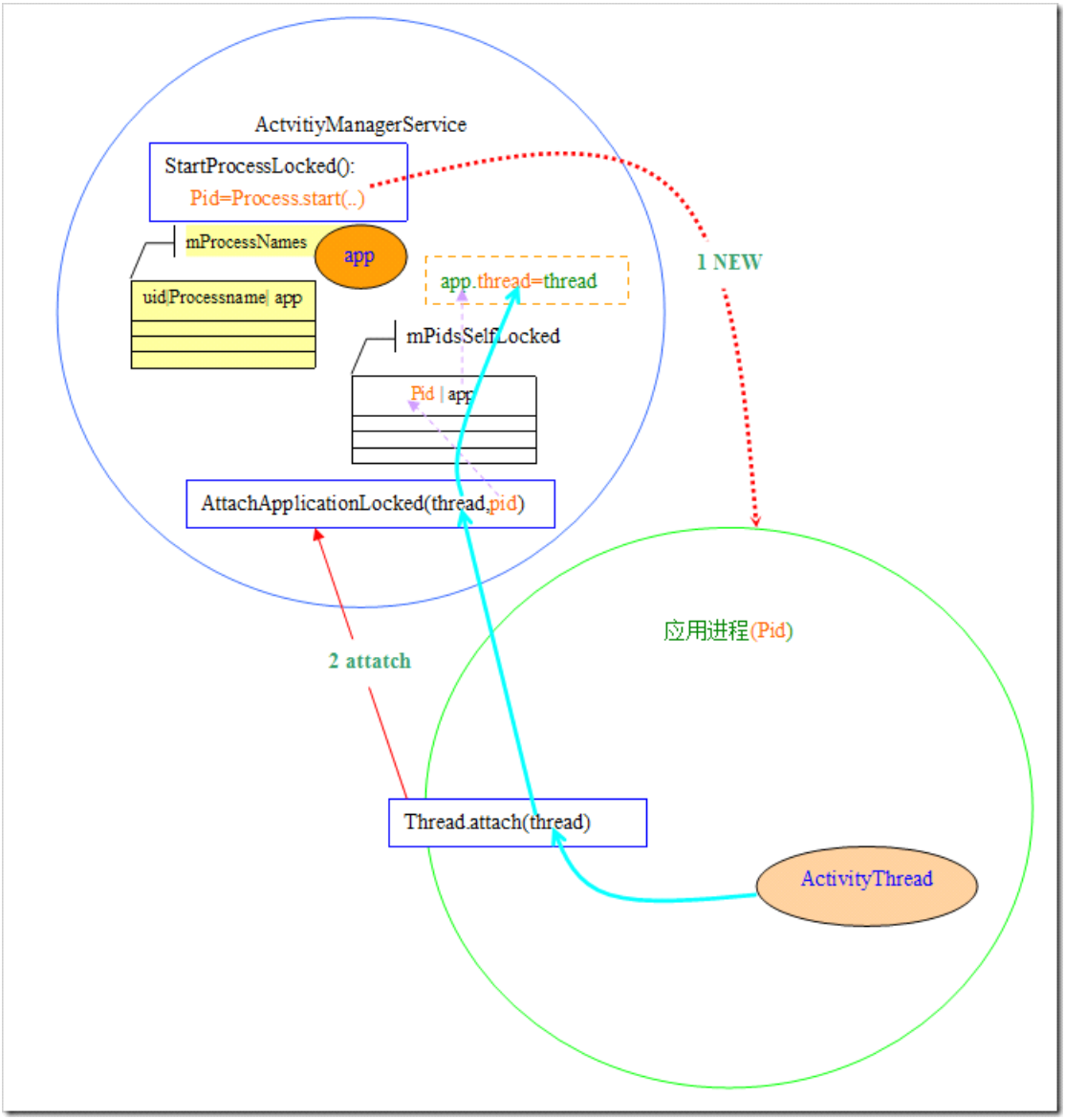
Android.app.ActivityThread 进程建立后，将跳入到 ActivityThread 的 main 函数开始运行，进入消息循环。



应用进程使用 thread.attach()发起 AMS 的 AttachApplicationLocked 调用，并传递 ActvitiyThread 对象和 CallingPid 。AttachApplicationLocked 将根据 CallingPid 在 mPidsSelfLocked 找到对应的 ProcessRecord 实例 app,将 ActvitiyThread 放置 app.thread 中。这样应用进程和 AMS 建立起来双向连接。AM 可以使用 AIDL 接口，通过 app.thread 可以访问应用进程的对象。

应用程序通过 ActivityThread 提供的框架，建立消息循环 Looper 和 Handler。从前面的相关章节我们知道有 Looper 和 Handler,整个系统就可以运作了。

为了更为系统的了解应用程序的建立时序及其涉及到数据操作，我给出了应用进程的建立过程示意图：



Android 核心分析（22）-----Android 应用框架之 Activity

3 Activity 设计框架

3.1 外特性空间的 Activity

我们先来看看，Android 应用开发人员接触的外特性空间中的 Activity，对于 AMS 来讲，这个 Activity 就是客服端的 Activity。应用程序员在建立 Android 应用时，构建 Activity 的子类就是 Andoid 外特性空间展现的接口。我们可以从下面的简单的例子描述看看 Activity，到底如何建立的。

DemoActivity extend Activity

{

onCreate

onResume

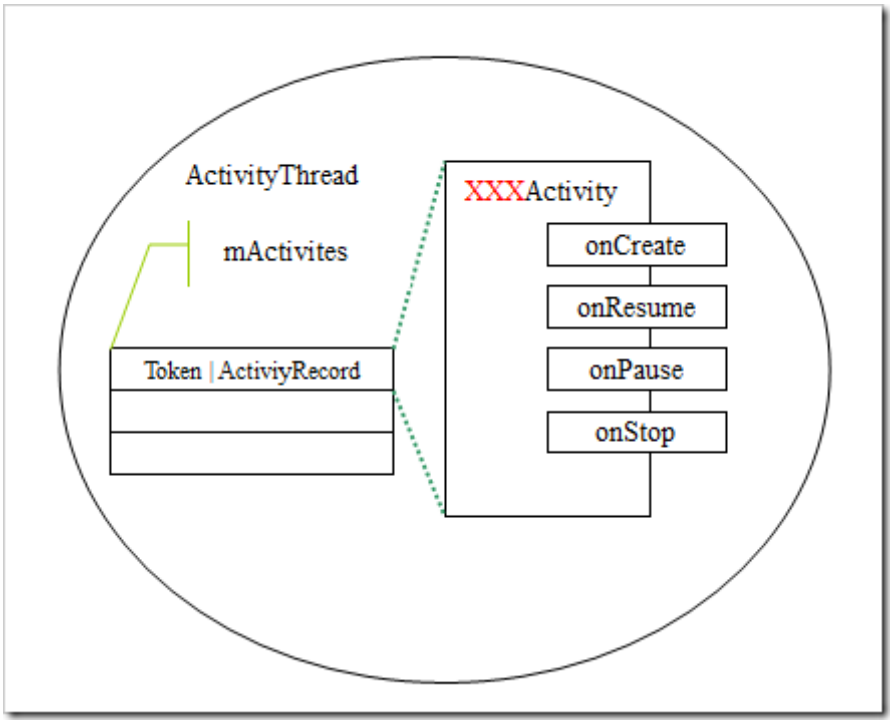
onPause

onStop

}

在 Android 的外特性空间(SDK)中，Android 应用程序员根本不知道进程是什么时候起来的，系统消息是如何传递过来的。这个 DemoActivity 是如何实例化的呢？并且该 Activity 是托管在哪个进程的呢？本节的分析将给出答案。

我们从 ActivityThread 中可以看到在应用进程中的 Activity 都被放置在 mActivities 中。

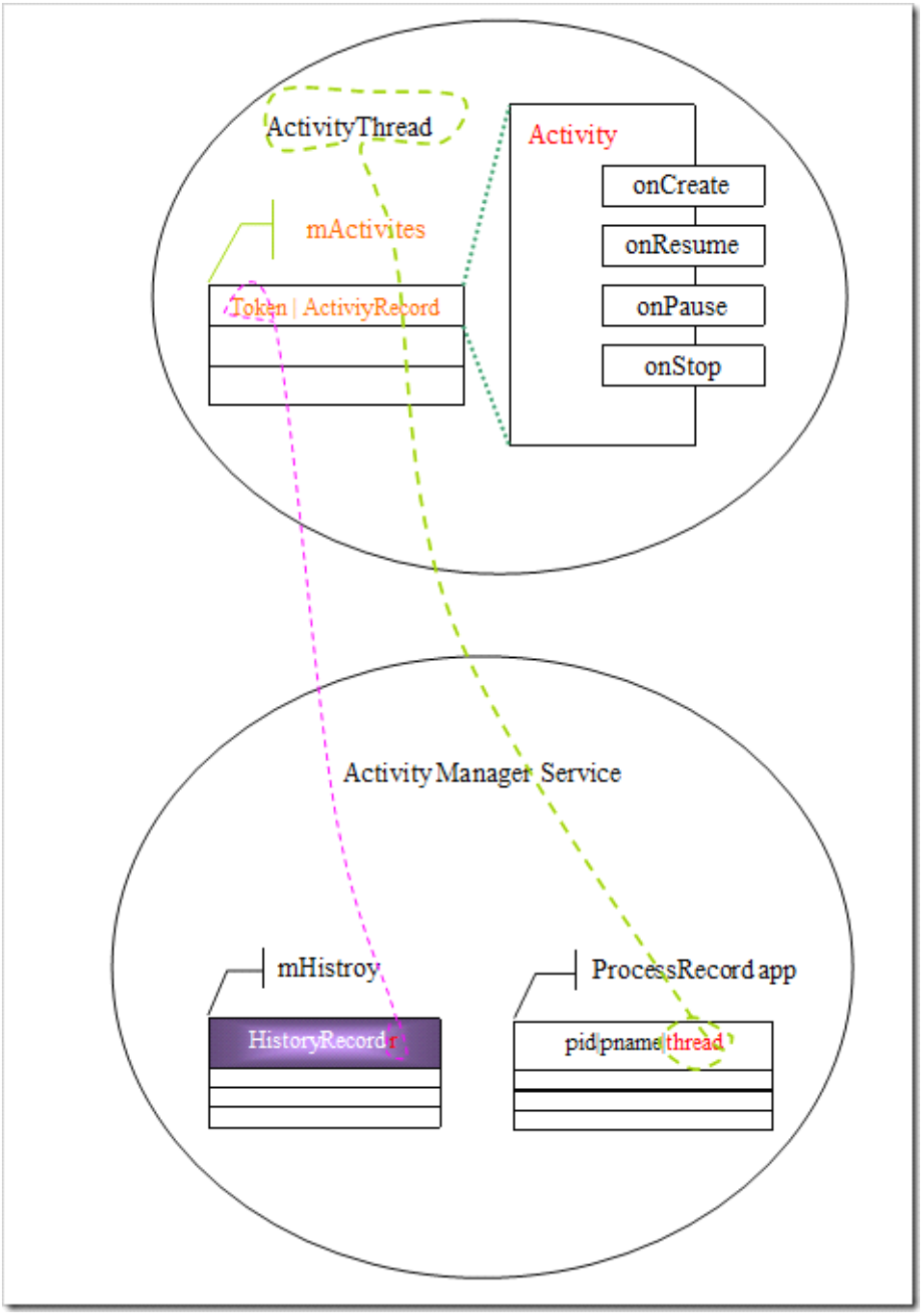


这些 ActivityRecord 记录了应用进程中，程序员建立的 Activity 子类的实例，我们称之为外特性空间的 Activity。这些 Activity 类实例是放在应用程序端进行实际交互的 Activity，而为了管理这些 Activity，AMS 内核中还有一个影子 Activity，被称为 HistoryRecord。

3.2 Activity 与 HistoryRecord 的关系

在整个系统中，Activity 实际上有两个实体。一个在应用进程中跟应用程序员打交道的 Activity，一个是在 AMS 的中具有管理功能的 History Record。应用进程中的 Activity 都登记 ActivityThread 实例中的 mActivity 数组中，而在 AM 端，HistroytRecord 实例放置在 mHistroy 栈中。mHistory 栈是 Android 管理 Activity 的场所，放置在栈顶的就是 User 看到的处于活动状态的 Activity。

Activity 与 HistrotyRecord 的关系图可以表示如下：



Activity 的内核实体是依靠在 ProcessRecord 的成员变量中，通过 ProcessRecord 我们可以访问到所有的属于该 Process 的 Activity。而在 ProcessRecord 记录了与应用进程之间的联系：IActivtityThread 接口。通过该接口，可以访问到所对应的 Activity 的方法。在 Launch Activity 时，AMS 将对应的 HistoryRecord 作为 token 传递到客服端和客服端的 Activity 建立联系。在 AMS 中 Activity 状态变化时，将通过该联系找到客服端的 Activity，从而将消息或者动作传递应用程序面对的接口：xxxActivity。

3.3 Actvity 的 Launch 过程

1)发起请求 startActivity(intent)

2）Activity Service Manager 接收到请求执行 StartActivity 函数。

建立：HistoryRecord 实例 r.

将 r 加入到 mHistory 顶。

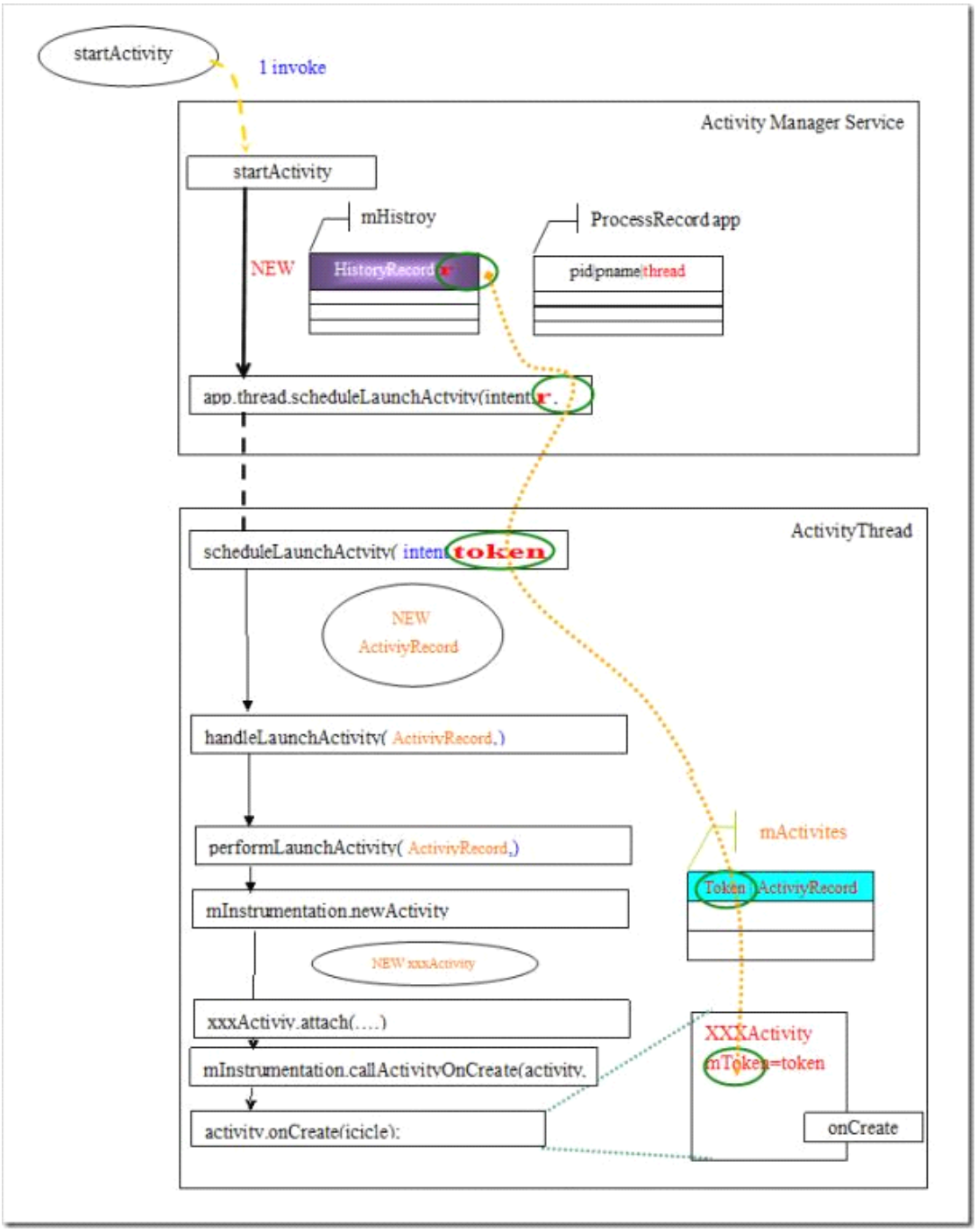
（3）通过 app.thread.scheduleLaunchActvity(app,r)@ActivityThread.java

（4）在 App 应用中建立新的 ActivityRecord。

（5）建立新的 Activity 对象并放入到 ActivityRecord 中。

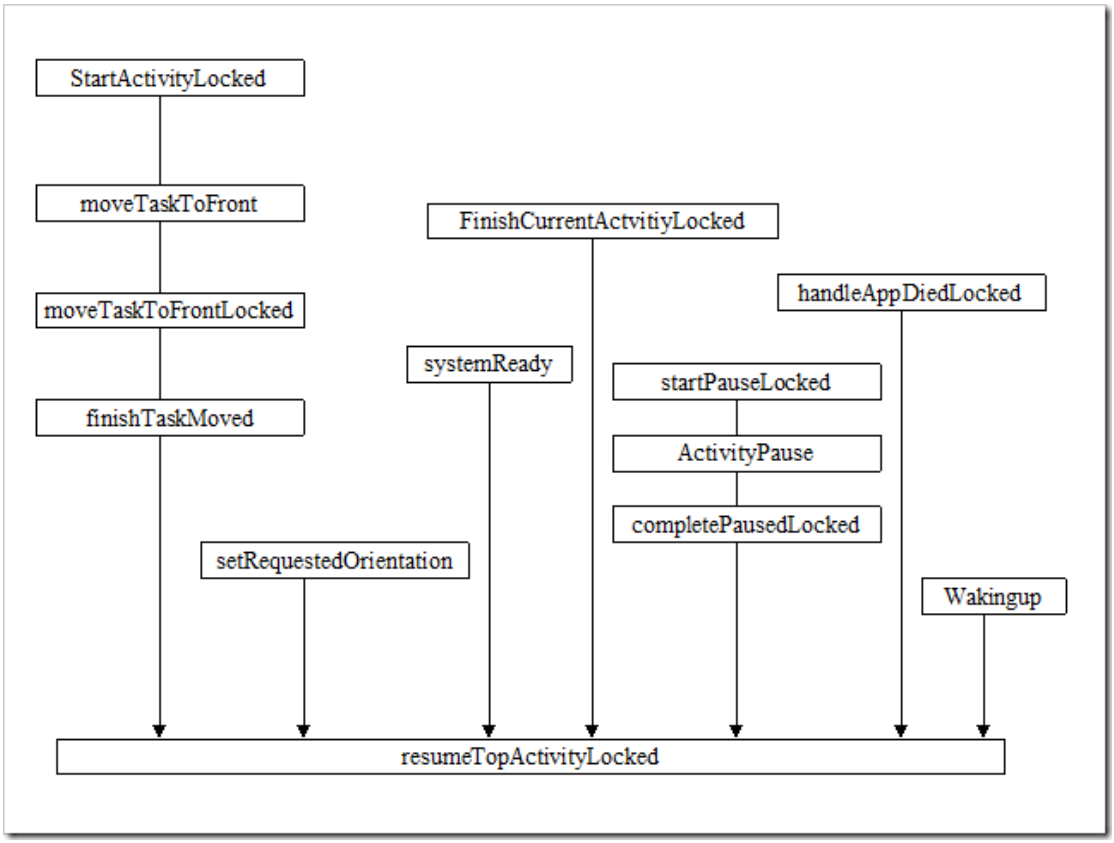
（6）将 ActivityRecord 加入到 mActivites@ActivityThread

(7)发起 Activity.onCreate(..),,该 onCreate 就是在你的应用程序 XXXActivity 中的 onCreate。



3.4 Activity 的 Resume

（1）Activity 什么时候被 Resume



（2）Rusume 的过程

通过该过程的研究我们会进一步的了解到 AMS 与应用进程的交互过程。

在 AMS 端，满足 resume 条件都会调用：Resume 的核心函数：resumeTopActivityLocked@ActivityManagerService

XXX 当前栈顶的 HistroyRecord

1）窗口切换：隐藏前一个 Activity 的窗口，

2）更新 LRUList，（LRUList 是淘汰应用程序的依据之一）

3) XXX.app.thread.scheduleResumeActivity(XXX,

isNextTransitionForward());

4)completeResumeLocked

setFocusedActivityLocked

mFocusActivity=xxx //此时焦点 Actvitiy 切换了。

WM.setFocusedApp(xxx,

mWindowManager.executeAppTransition();

mNoAnimActivities.clear();

在应用程序端：

（5）scheduleResumeActivity

handleResumeActivity(IBinder token, boolean clearHide, boolean isForward) {

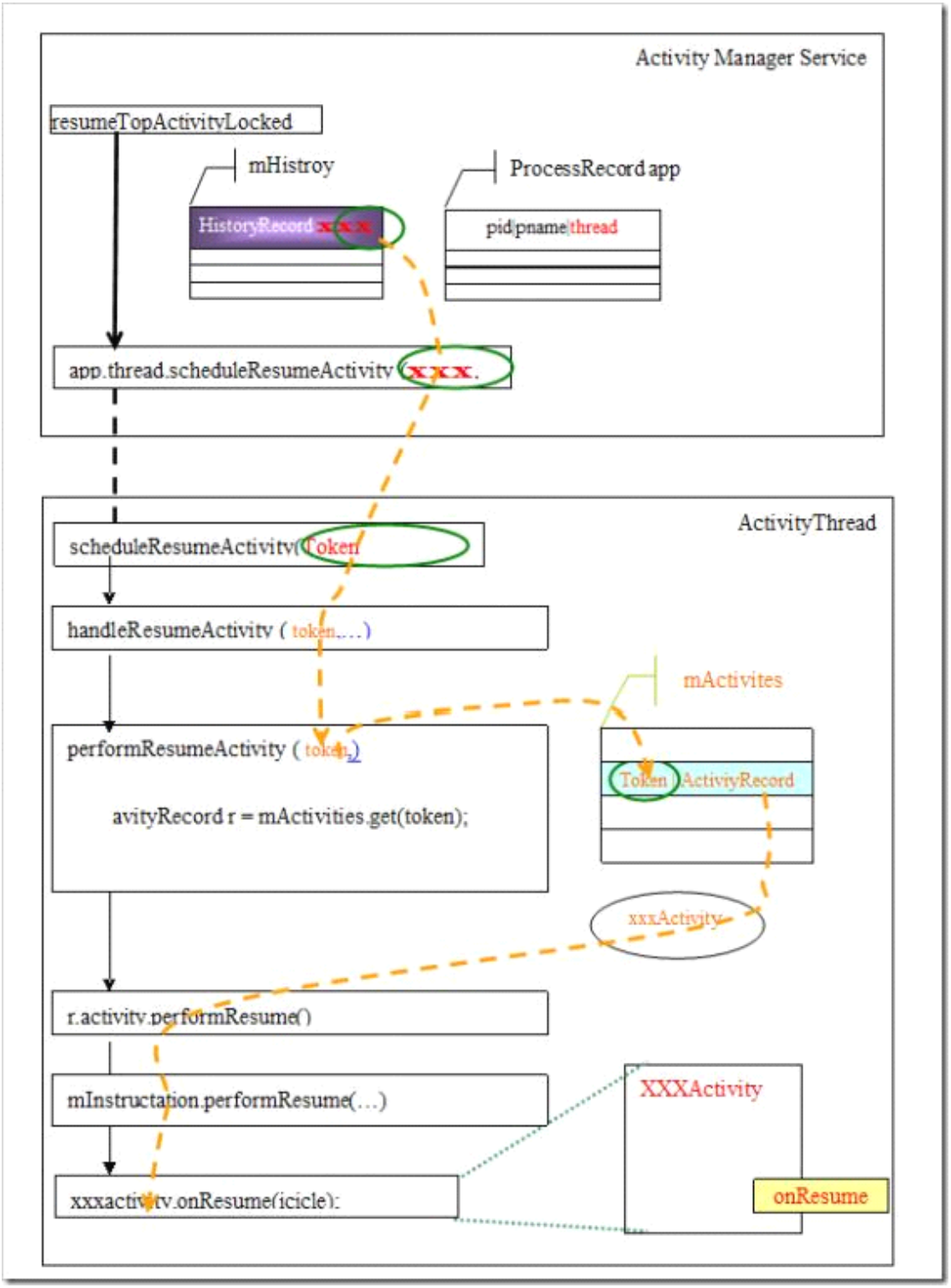
ActivityRecord r = performResumeActivity(token, clearHide);

ActivityRecord r = mActivities.get(token);

r.activity.performResume()

performResume

整个 Resume 的过程如下：

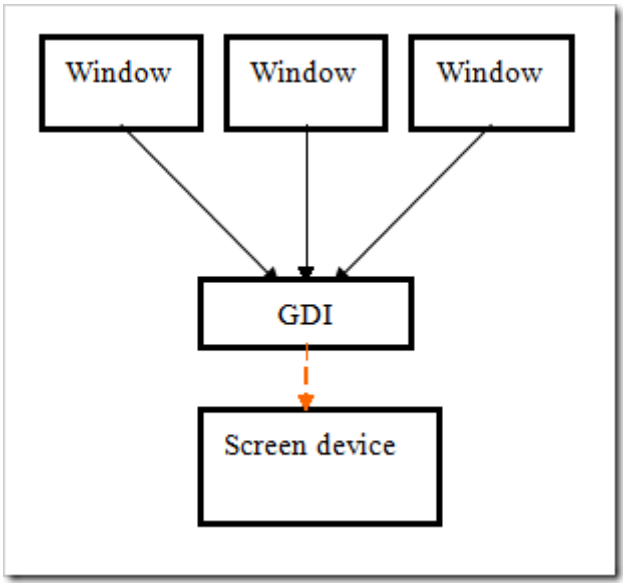


Android 核心分析（23）-----Andoird GDI 之基本原理及其总体框架

Android GDI 基本框架

在 Android 中所涉及的概念和代码最多，最繁杂的就是 GDI 相关的代码了。但是本质从抽象上来讲，这么多的代码和框架就干了一件事情：对显示缓冲区的操作和管理。

GDI 主要管理图形图像的输出,从整体方向上来看,GDI 可以被认为是一个物理屏幕使用的管理器。因为在实际的产品中，我们需要在物理屏幕上输出不同的窗口，而每个窗口认为自己独占屏幕的使用，对所有窗口输出，应用程序不会关心物理屏幕是否被别的窗口占用，而只是关心自己在本窗口的输出，至于输出是否能在屏幕上看见，则需要 GDI 来管理。



从最上层到最底层的数据流的分析可以看到实际上 GDI 在上层为 GUI 提供一个抽象的概念,就好像操作系统中的文件系统所提供文件，目录等抽象概念一样，GDI 输出抽象成了文本,画笔,位图操作等设备无关的操作,让应用程序员只需要面对逻辑的设备上下文进行输出操作,而不要涉及到具体输出设备,以及输出边界的管理。GDI 负责将文本、线条、位图等概念对象映射到具体的物理设备，所以 GDI 的在大体方向上可以分为以下几大要素：

画布

字体

文本输出

绘画对象

位图输出

Android 的 GDI 系统

Android 的 GDI 系统所涉及到概念太多，加之使用了 OpenGL 使得 Android 的层次和代码很繁杂。但是我们对于 Android 的 GDI 系统需要了解的方面不是他的静态的代码关系，而是动态的对象关系，在逻辑运行的架构上理解 GDI。我们首先还是需要从代码结构开始我

们的理解。

Frameworks/Libs/Surfaceflinger

Frameworks/base/core/jni/android_view_Surface.cpp

Frameworks/base/core/java/android/view/surface.java

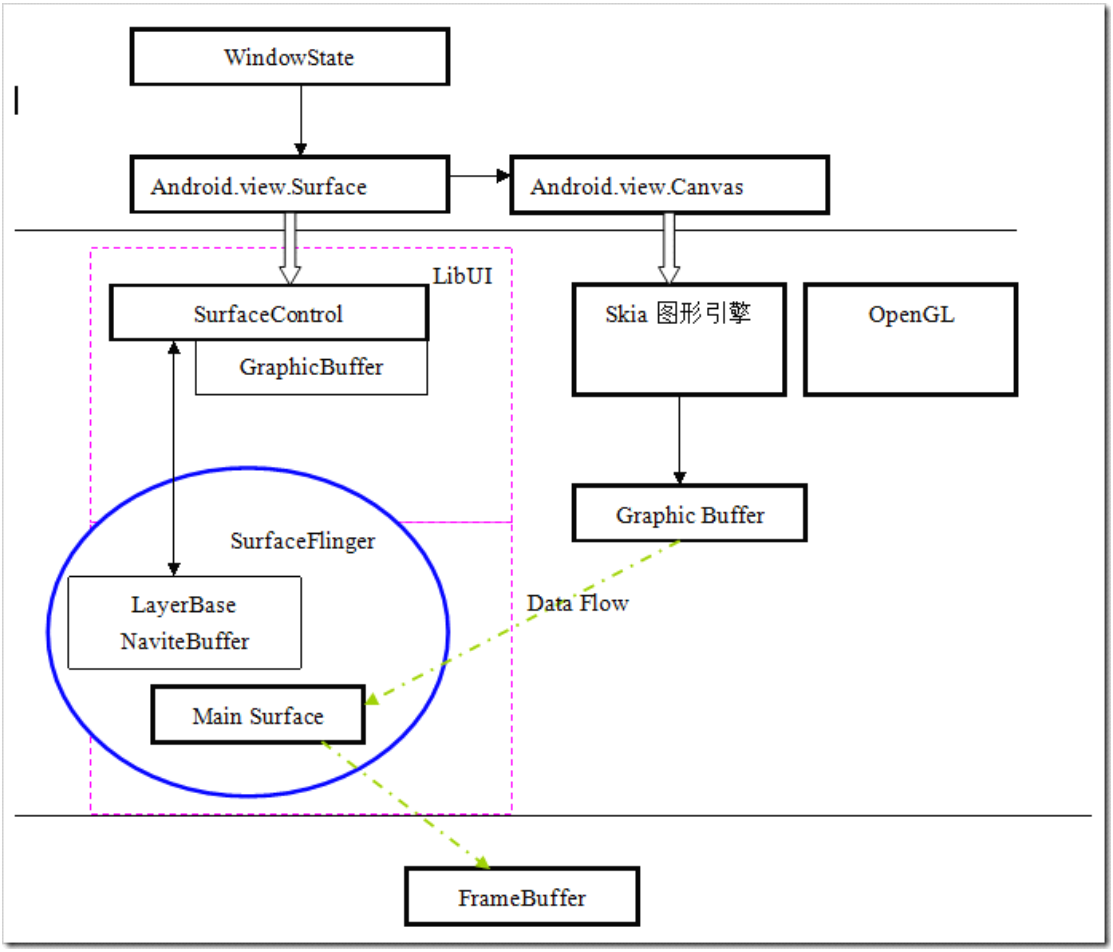
Frameworks/base/Graphics：绘图接口

Frameworks/Libs/Ui

External/Skia

其中 External/Skia 是一个 C++的 2D 图形引擎库，Android 的 2D 绘制系统都是建立在该基础之上.Skia 完成了：文本输出，位图，点，线，图像解码等功能。

我在这里给出 Android GDI 的基本框架示意图。



对于上面的 GDI 架构图我们只是一个大概的了解，我们有太多的问题需要解决，有太

多的疑问需要得到答案，我就一直在想，为什么设计者有提出如此众多的概念，这个概念的背景是什么？他要管理什么，他要抽象什么？从前面知道，Android 的整个设计理念就是无边界化，他是如何穿透 Linux 进程这个鸿沟来达到无边界的？Surface，Canvas， Layer，LayerBase, NativeBuffer，SurfaceFlinger，SurfaceFlingerClient 这些到底是一个什么东西？如何管理，传递的是什么？创建的是什么？这些都是抽象的概念，绘画的终极的缓冲区到底是如何管理的？缓冲区到底在哪里？

我们还是看看做终极的，最本质的设计概念，在从这些概念出发，来探讨这些概念的形成过程，是否有必要去生成写概念。SurfaceFlinger 本质上干什么的？SurfaceFlinger 的确就是这个意义：应用程序通过 SurfaceFlinger 将自己的“Surface”投掷到屏幕缓冲区。至于如何投掷的，我们将会在后面详细描述。

Android 核心分析（24)-----Android GDI 之显示缓冲管理

Android 核心分析（24)-----Android GDI 之显示缓冲管理 收藏
Android GDI 之屏幕设备管理-动态链接库

万丈高楼从地起，从最根源的硬件帧缓冲区开始。我们知道显示 FrameBuffer 在系统中就是一段内存,GDI 的工作就是把需要输出的内容放入到该段内存的某个位置。我们从基本的点（像素点）和基本的缓冲区操作开始。

1 基本知识
1.1 点的格式

对于不同的 LCD 来讲，FrameBuffer 的二进制格式不一样，并且可以分为两部分：

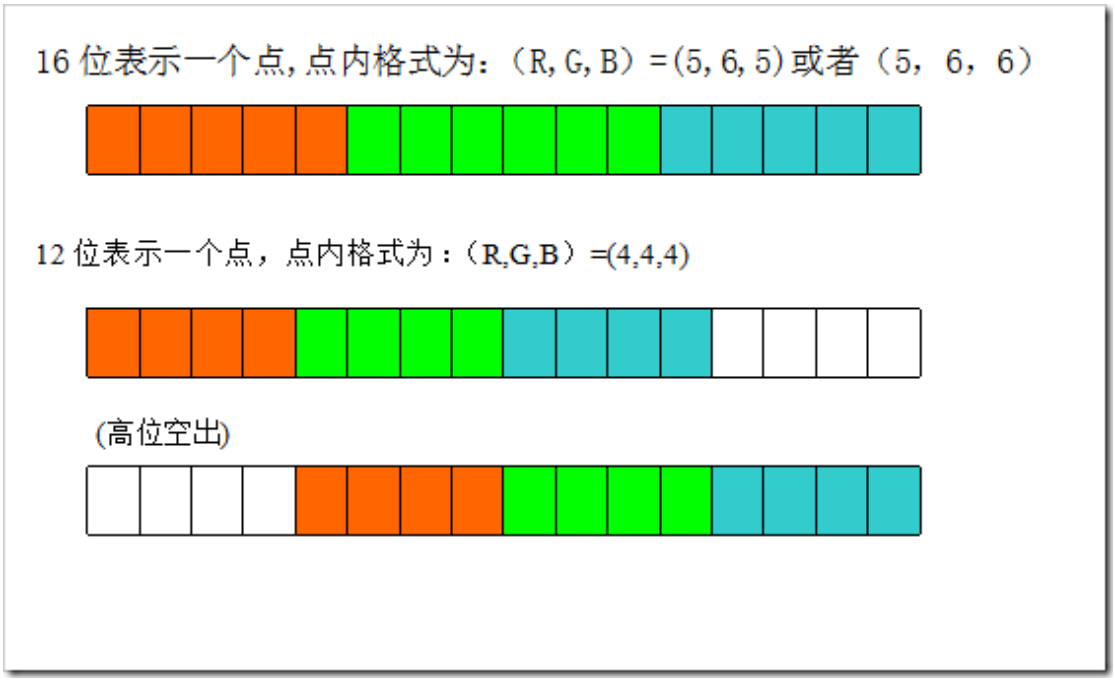
1)点的格式：通常将 Depth,即表示多少位表示一个点。
1 位表示一个点

2 位表示一个点

16 位表示一个点

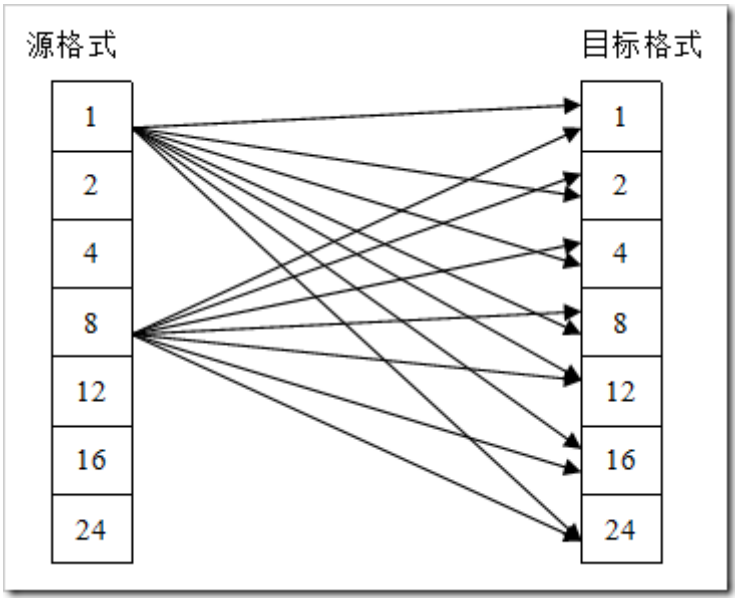
32 位表示一个点（Alpha 通道）

2) 点内格式：RGB 分量分布表示。
例如对于我们常见的 16 位表示一个点

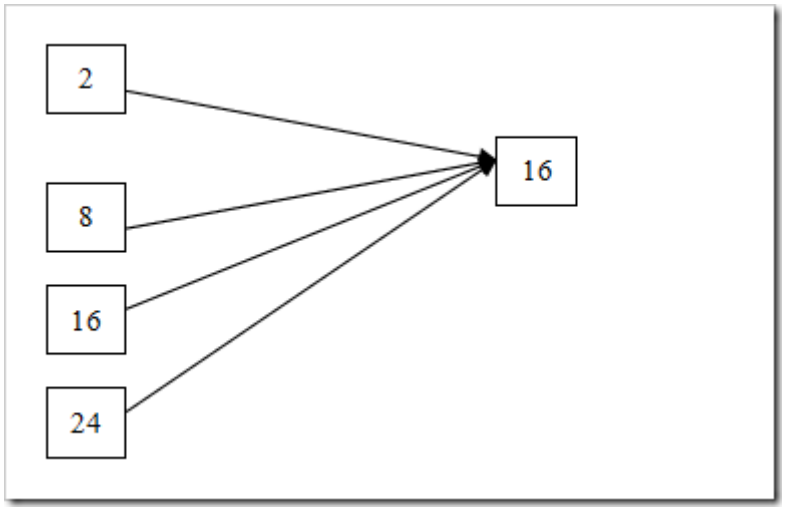


1.2．格式之间的转换

所以屏幕输出实际上是一个值映射的关系。我们可以有如下的点格式转换，



源格式可能来自单色位图和彩色位图，对于具体的目标机来讲，我们的目标格式可能就是一种，例如 16 位（5/6/5）格式。其实就只存在一种格式的转换，即从目标格式都是 16 位格式。



但是，在设计 GDI 时，基本要求有一个可移植性好，所以我们还是必须考虑对于不同点格式 LCD 之间的转换操作。所以在 GDI 的驱动程序中涉及到如下几类主要操作:

区域操作(Blit)：我们在显示缓冲区上做的最多的操作就是区块搬运。由此，很多的应用处理器使用了硬件图形加速器来完成区域搬运:blit.从我们的主要操作的对象来看,可以分为两个方向:

1)内存区域到屏幕区域

2)屏幕区域到屏幕区域

3)屏幕区域到内存区域

4)内存区域到内存区域

在这里我们需要特别提出的是，由于在 Linux 不同进程之间的内存不能自由的访问，使得我们的每个 Android 应用对于内存区域和屏幕缓冲区的使用变得很复杂。在 Android 的设计中，在屏幕缓冲区和显示内存缓冲区的管理分类很多的层次，最上层的对象是可以在进程间自由传递，但是对于缓冲区内容则使用共享内存的机制。

基于以上的基础知识，我们可以知道：

（1）代码中 Config 及其 Format 的意义所在了。也就理解了兼容性的意义：采用同硬件相同的点的描述对象

（2）所有屏幕上图形的移动都是显示缓冲区搬运的结果。

1.2 图形加速器

应用处理器都可能带有图形加速器，对于不同的应用处理器对其图形加速器可能有不同的处理方式，对于 2D 加速来讲，都可归结为 Blit。多为数据的搬运，放大缩小，旋转等。

2 Android 的缓冲区抽象定义

不同的硬件有不同的硬件图形加速设备和缓冲内存实现方法。Android Gralloc 动态库抽象的任务就是消除不同的设备之间的差别，在上层看来都是同样的方法和对象。在 Moudle 层隐藏缓冲区操作细节。Android 使用了动态链接库 gralloc.xxx.so，来完成底层细节的封装。

2.1 本地定义@hardware\libhandware\modules\gralloc

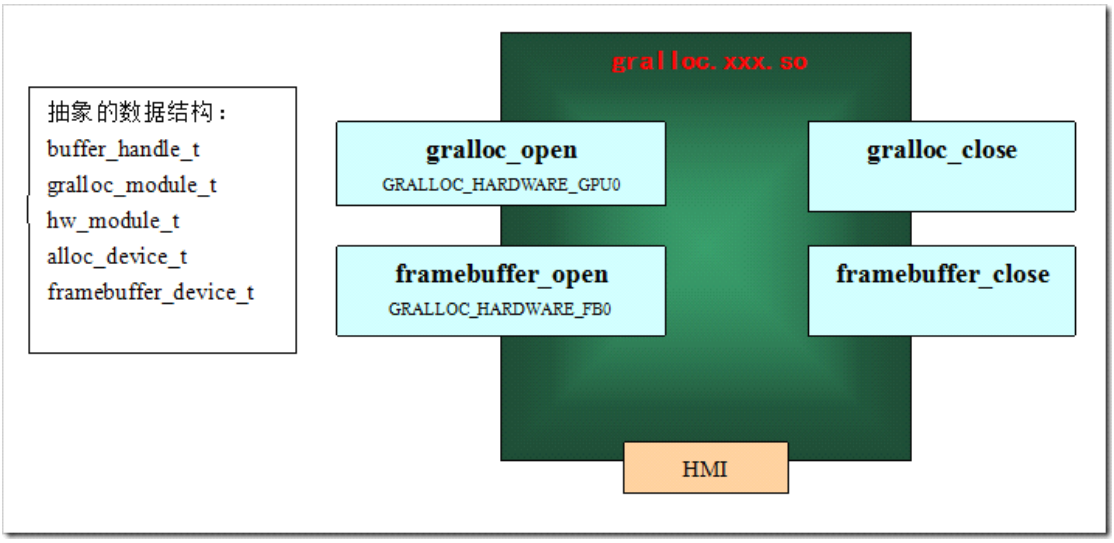
每个动态链接库都是用相同名称的调用接口：

1)硬件图形加速器的抽象：BlitEngine，CopyBit 的加速操作。

2)硬件 FrameBuffer 内存管理

3)共享缓存管理

从数据关系上我们来考察..动态链接库的抽象行为：在层次：Hardware.c@hardware\libhardware 中对动态链接库中的内容作了全新的包装。/system/lib/hw/gralloc.xxx.so 动态库文件。从文件 Gralloc.h(handware\libhardware\include\hardware) 是抽象的结果： hw_get_module 从 gralloc.xxx.so 提取了 HAL_MODULE_INFO_SYM（SYM 变量）



从展露在外部的数据结构，我们在@Gralloc.cpp 看到到了这样的布局：

static struct hw_module_methods_t gralloc_module_methods = {

open: gralloc_device_open

};

```
struct private_module_t HAL_MODULE_INFO_SYM = {
    base: {
        common: {
            tag: HARDWARE_MODULE_TAG,
             …
            id: GRALLOC_HARDWARE_MODULE_ID,
            name: "Graphics Memory Allocator Module",
            author: "The Android Open Source Project",
            methods: &gralloc_module_methods
        },
registerBuffer: gralloc_register_buffer,
unregisterBuffer: gralloc_unregister_buffer,
lock:    gralloc_lock,
unlock:    gralloc_unlock,
    },
    framebuffer: 0,
    flags: 0,
    numBuffers: 0,
    bufferMask: 0,
…
};
```

我们建立了什么对象来支撑缓冲区的操作？

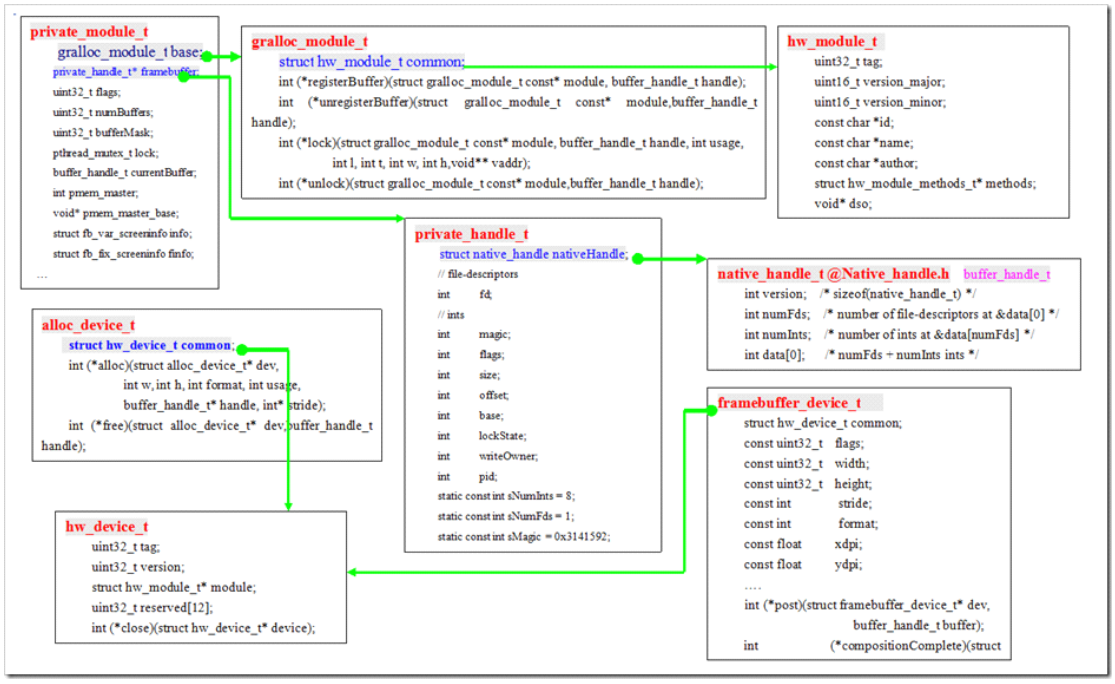
buffer_handle_t：外部接口。

methods.open，registerBuffer，unregisterBuffer，lock，unlock

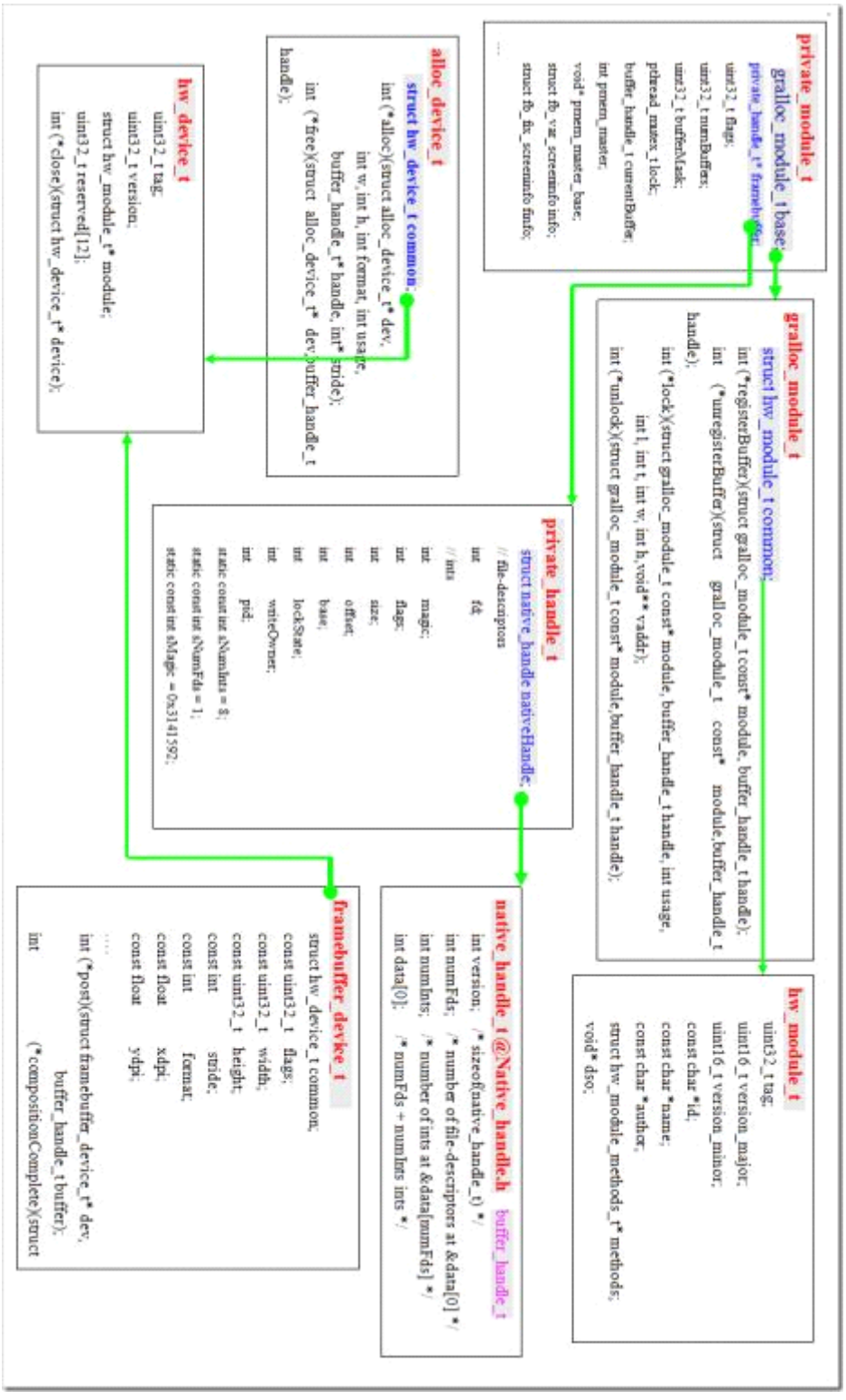
下面是外部接口和内部对象的结构关系，该类型的结构充分利用 C Struct 的数据排列特性：基本结构体放置在最前面，本地私有放置在后面，满足了抽象的需要。

typedef const native_handle* buffer_handle_t;

private_module_t　HAL_MODULE_INFO_SYM 向往暴露的动态链接库接口，通过该接口，我们直接可以使用该对象。



看不清楚上面图，可以偏一下头横着看：

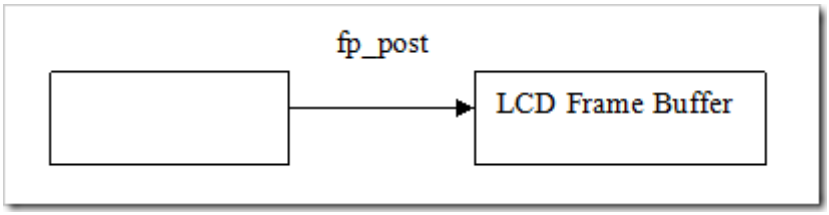


几个接口函数的解释：

(1)fb_post

对于帧缓冲区实际地址并不需要向上层报告，所有的操作都是通过 fb_post 了完成。

fp_post 的任务就是将一个 Buffer 的内容传递到硬件缓冲区。其实现方式有两种：



（方式 1）无需拷贝动作，是把 Framebuffer 的后 buffer 切为前 buffer，然后通过 IOCTRL 机制告诉 FB 驱动切换 DMA 源地地址。这个实现方式的前提是 Linux 内核必须分配至少两个缓冲区大小的物理内存和实现切换的 ioctrol，这个实现快速切换。

（方式 2）利用 Copy 的方式。不修改内核，则在适配层利用从拷贝的方式进行，但是这个是费时了。

(2)gralloc 的主要功能是要完成：

1）打开屏幕设备 "/dev/fb0",，并映射硬件显示缓冲区。

2）提供分配共享显示缓存的接口

3）提供 BiltEngine 接口（完成硬件加速器的包装）

（3）gralloc_alloc 输出 buffer_handle_t 句柄。

这个句柄是共享的基本依据，其基本原理在后面的章节有详细描述。

3 总结

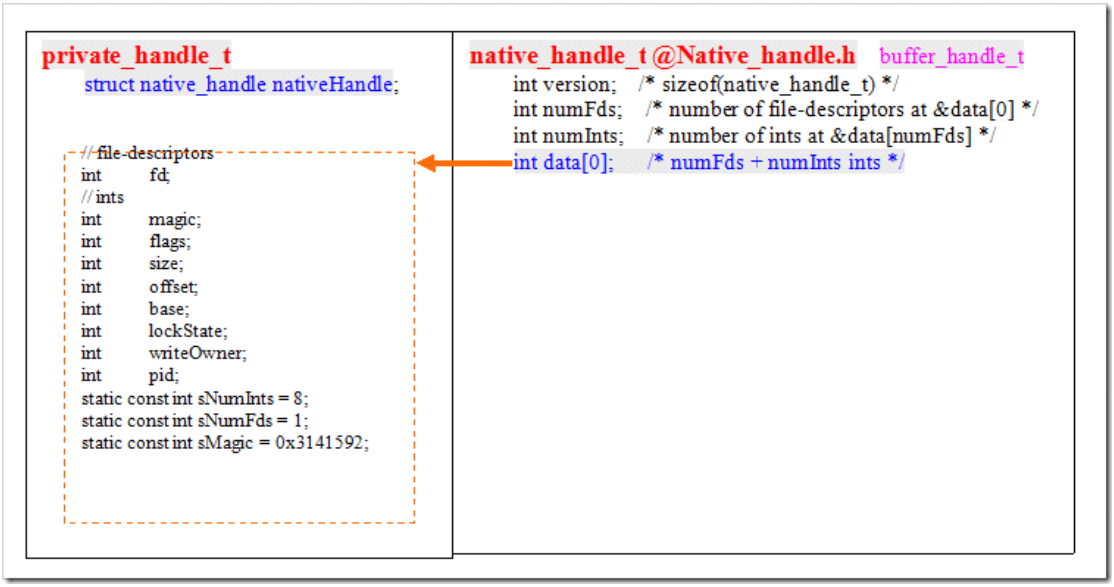
总结一下，/system/lib/hw/gralloc.xxx.so 是跟硬件体系相关的一个动态链接库，也可以叫做 Android 的硬件抽象层。他实现了 Android 的硬件抽象接口标准，提供显示内存的分配机制和 CopyBit 等的加速实现。而如何具体实现这些功能，则跟硬件平台的配备有关系，所以我们看到了对于与不同的硬件架构，有不同的配置关系。

Android 核心分析（25）-----Android GDI 之共享缓冲区机制

Androird GDI 之共享缓冲区机制

1　native_handle_t 对 private_handle_t 的包裹

private_handle_t 是 gralloc.so 使用的本地缓冲区私有的数据结构，而 Native_handle_t 是上层抽象的可以在进程间传递的数据结构。在客户端是如何还原所传递的数据结构呢？首先看看 native_handle_t 对 private_handle_t 的抽象包装。



numFds= sNumFds=1;

numInts= sNumInts=8;

这个是 Parcel 中描述句柄的抽象模式。实际上是指的 Native_handle 所指向句柄对象的具体内容：

numFds=1 表示有一个文件句柄：fd/

numInts= 8 表示后面跟了 8 个 INT 型的数据：magic,flags,size,offset,base,lockState,writeOwner,pid;

由于在上层系统不要关心 buffer_handle_t 中 data 的具体内容。在进程间传递 buffer_handle_t(native_handle_t)句柄是其实是将这个句柄内容传递到 Client 端。在客户端通过 Binder 读取 readNativeHandle @Parcel.cpp 新生成一个 native_handle。

```
native_handle* Parcel::readNativeHandle() const

{

…

native_handle* h = native_handle_create(numFds, numInts);

    for (int i=0 ; err==NO_ERROR && i

        h->data[i] = dup(readFileDescriptor());

        if (h->data[i] < 0) err = BAD_VALUE;

    }

    err = read(h->data + numFds, sizeof(int)*numInts);

    ….

return h;

}
```

这里需要提到的是为在构造客户端的 native_handle 时，对于对方传递过来的文件句柄的处理。由于不是在同一个进程中，所以需要 dup(…)一下为客户端使用。这样就将 Native_handle 句柄中的，客户端感兴趣的从服务端复制过来。这样就将 Private_native_t 的数据：magic,flags,size,offset,base,lockState,writeOwner,pid;复制到了客户端。

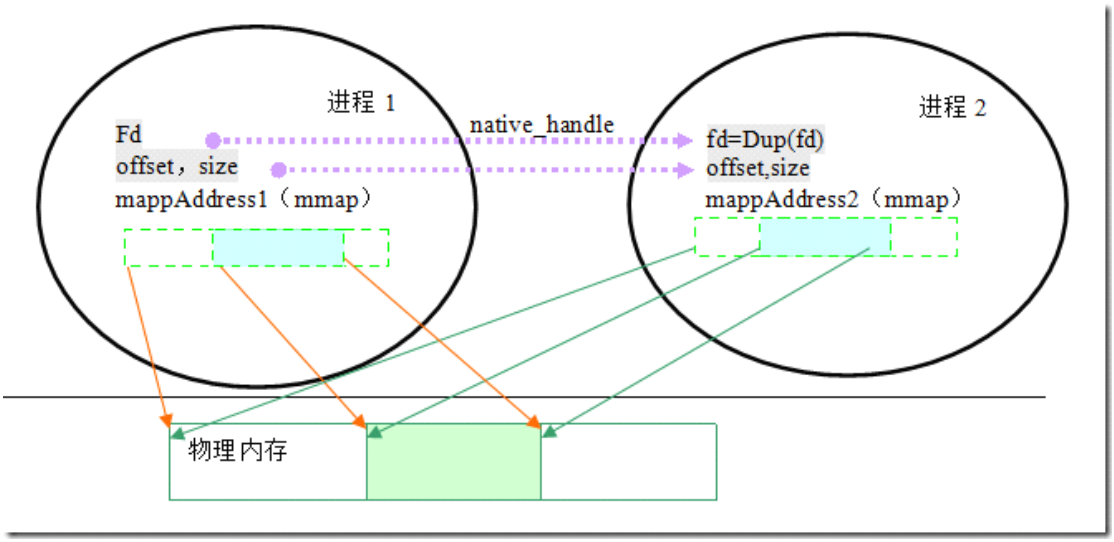
客户端利用这个新的 Native_buffer 被 Mapper 传回到 gralloc.xxx.so 中，获取到 native_handle 关联的共享缓冲区映射地址，从而获取到了该缓冲区的控制权，达到了客服端和 Server 间的内存共享。从 SurfaceFlinger 来看就是作图区域的共享。

2 Graphic Mapper 是干什么的？

服务端（SurfaceFlinger）分配了一段内存作为 Surface 的作图缓冲，客户端怎样在这个作图缓冲区上工作呢？这个就是 Mapper(GraphicBufferMapper)y 要干的事情。两个进程间如何共享内存，如何获取到共享内存？Mapper 就是干这个得。需要利用到两个信息：共享缓冲区设备句柄，分配时的偏移量。Mapper 利用这样的原理：

客户端只有 lock,unlock,实质上就是 mmap 和 ummap 的操作。对于同样一个共享缓冲区，偏移量才是总要的，起始地址不重要。实际上他们操作了同一物理地址的内存块。我们在上面讨论了 native_handle_t 对 private_handle_t 的包裹过程，从中知道服务端给客户端传递了什么东西。

进程 1 在共享内存设备上预分配了 8M 的内存。以后所有的分配都是在这个 8M 的空间进行。对以该文件设备来讲，8M 物理内存提交后，就实实在在的占用了 8M 内存。每个每个进程都可以同个该内存设备共享该 8M 的内存，他们使用的工具就会 mmap。由于在 mmap 都是用 0 开始获取映射地址，所以所有的客户端进程都是有了同一个物理其实地址，所以此时偏移量和 size 就可以标识一段内存。而这个偏移量和 size 是个数值，从服务进程传递到客户端直接就可以使用。



3 GraphicBuffer（缓冲区代理对象）

typedef struct android_native_buffer_t

{

struct android_native_base_t common;

int width;

int height;

int stride;

int format;

int usage;

…

buffer_handle_t handle;

…

} android_native_buffer_t;

关系图表：

GraphicBuffer :EGLNativeBase :android_native_buffer_t

GraphicBuffer(parcel &)建立本地的 GraphicBuffer 的数据 native_buffer_t。在通过接收对方的传递的 native_buffer_t 构建 GraphicBuffer。我们来看看在客户端 Surface：：lock 获取操作缓冲区的函数调用：

Surface::lock(SurfaceInfo* other, Region* dirtyIn, bool blocking)

{int Surface::dequeueBuffer(android_native_buffer_t** buffer)（Surface）

{status_t Surface::getBufferLocked(int index, int usage)

{

sp buffer = s->requestBuffer(index, usage);

{

virtual sp requestBuffer(int bufferIdx, int usage)

{ remote()->transact(REQUEST_BUFFER, data, &reply);

sp buffer = new GraphicBuffer(reply);

Surface：：Lock 建立一个在 Client 端建立了一个新的 GraphicBuffer 对象，该对象通过（1）描述的原理将 SurfaceFlinger 的 buffer_handle_t 相关数据构成新的客户端 buffer_handle_t 数据。在客户端的 Surface 对象就可以使用 GraphicMapper 对客户端 buffer_handle_t 进行 mmap 从而获取到共享缓冲区的开始地址了。

4 总结

Android 在该节使用了共享内存的方式来管理与显示相关的缓冲区，他设计成了两层，上层是缓冲区管理的代理机构 GraphicBuffer,及其相关的 native_buffer_t,下层是具体的缓冲区的分配管理及其缓冲区本身。上层的对象是可以在经常间通过 Binder 传递的，而在进程间并不是传递缓冲区本身，而是使用 mmap 来获取指向共同物理内存的映射地址。

Android 核心分析（26）-----Android GDI 之 SurfaceFlinger

Android GDI 之 SurfaceFlinger
SurfaceFinger 按英文翻译过来就是 Surface 投递者。SufaceFlinger 的构成并不是太复杂，复杂的是他的客户端建构。SufaceFlinger 主要功能是：

1） 将 Layers （Surfaces） 内容的刷新到屏幕上

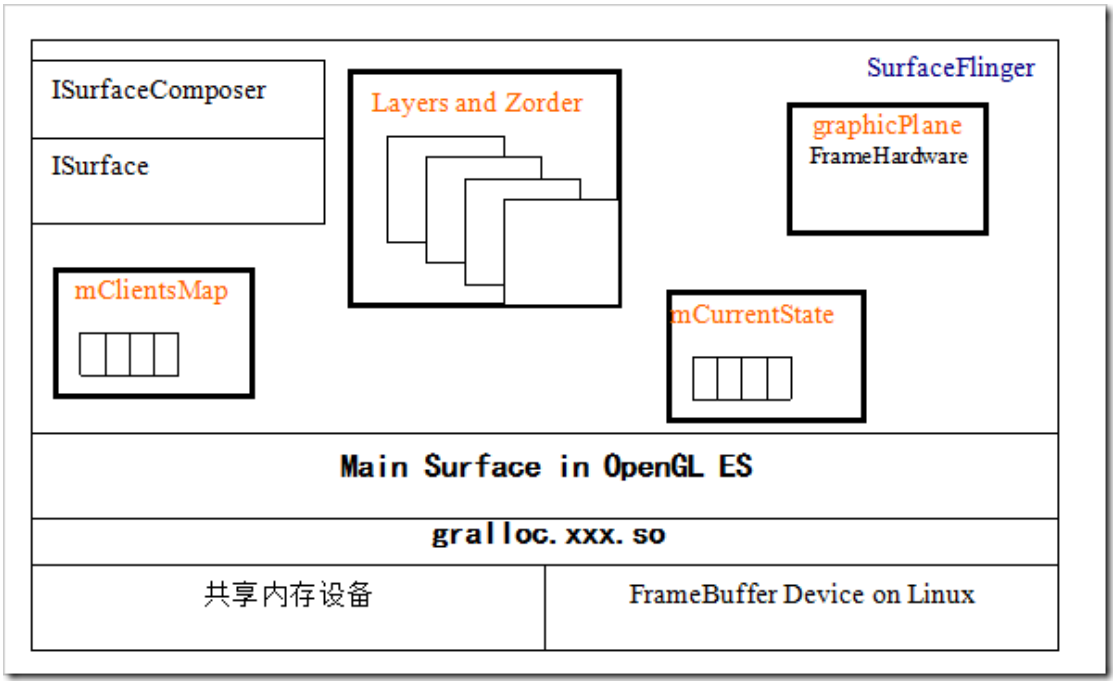
2） 维持 Layer 的 Zorder 序列，并对 Layer 最终输出做出裁剪计算。

3） 响应 Client 要求，创建 Layer 与客户端的 Surface 建立连接

4） 接收 Client 要求，修改 Layer 属性（输出大小，Alpha 等设定）

但是作为投递者的实际意义，我们首先需要知道的是如何投递，投掷物，投递路线，投递目的地。

1　SurfaceFlinger 的基本组成框架



SurfaceFlinger 管理对象为：

mClientsMap：管理客户端与服务端的连接。

ISurface，IsurfaceComposer：AIDL 调用接口实例

mLayerMap：服务端的 Surface 的管理对象。

mCurrentState.layersSortedByZ ： 以 Surface 的 Z-order 序列排列的 Layer 数组。

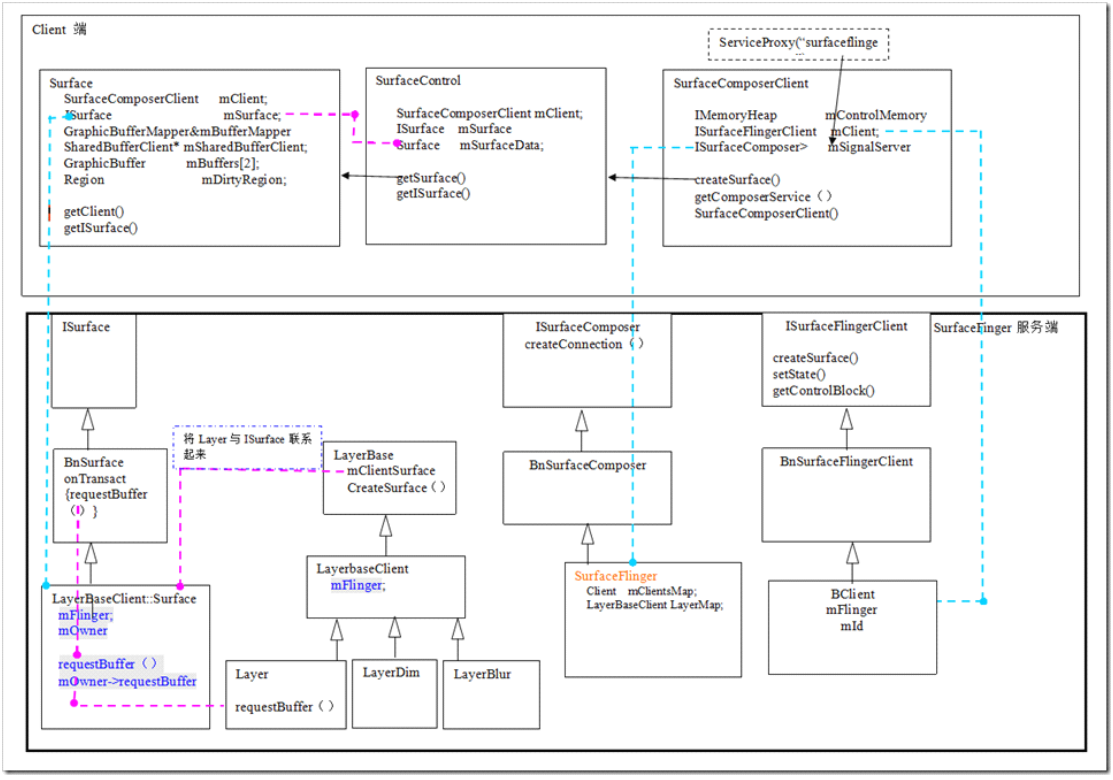
graphicPlane 缓冲区输出管理

OpenGL ES：图形计算，图像合成等图形库。

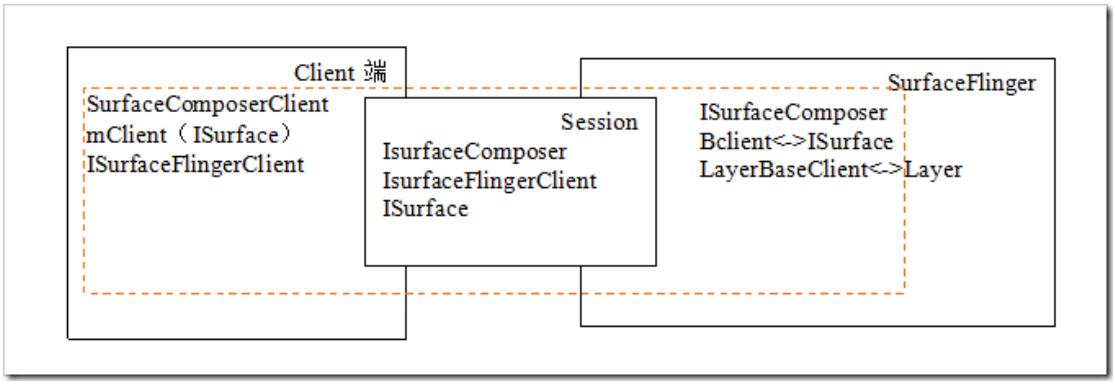
gralloc.xxx.so 这是个跟平台相关的图形缓冲区管理器。

pmem Device：提供共享内存，在这里只是在 gralloc.xxx.so 可见，在上层被 gralloc.xxx.so 抽象了。

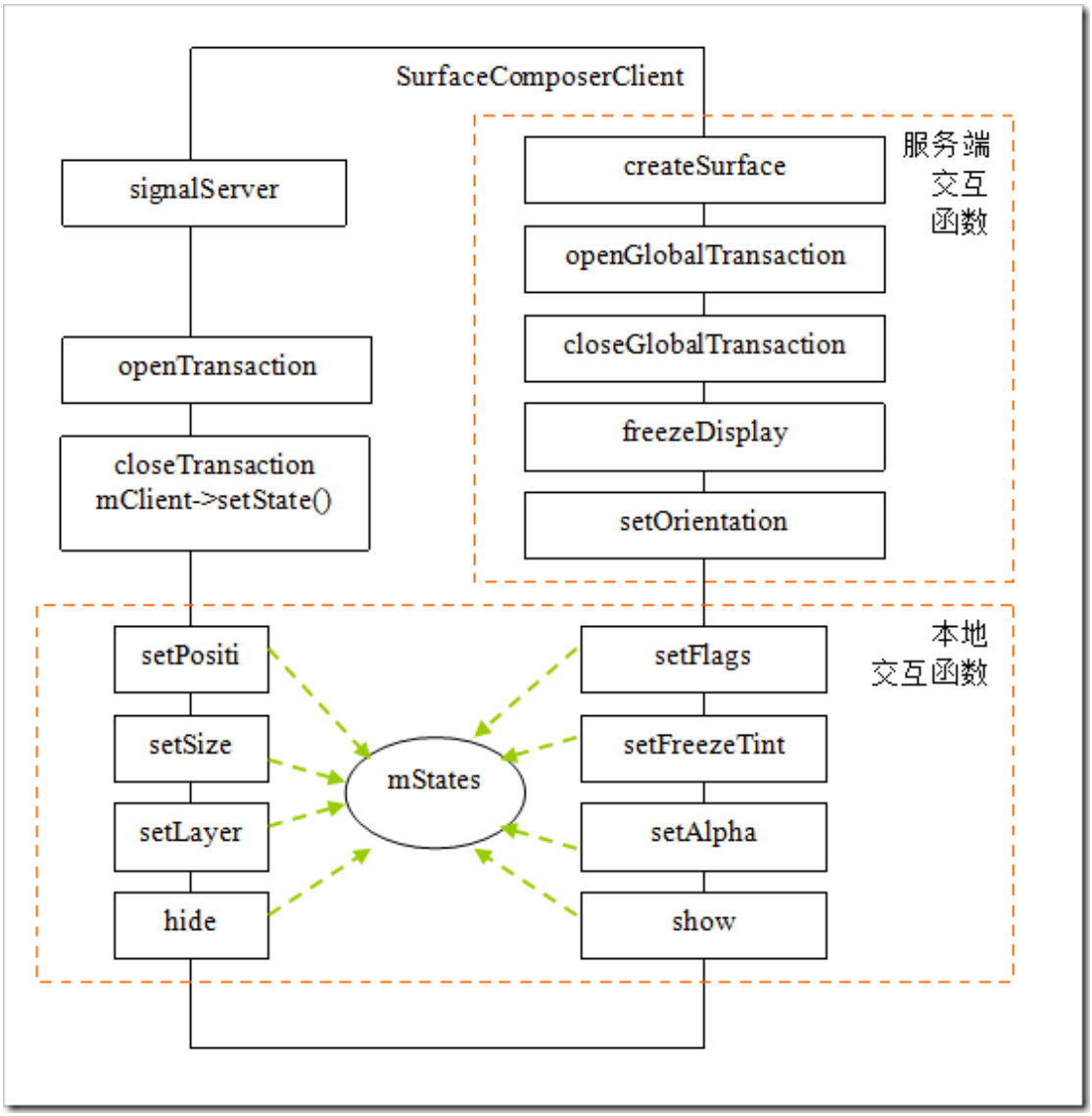
2 SurfaceFinger Client 和服务端对象关系图



Client 端与 SurfaceFlinger 连接图：



Client 对象：一般的在客户端都是通过 SurfaceComposerClient 来跟 SurfaceFlinger 打交道。

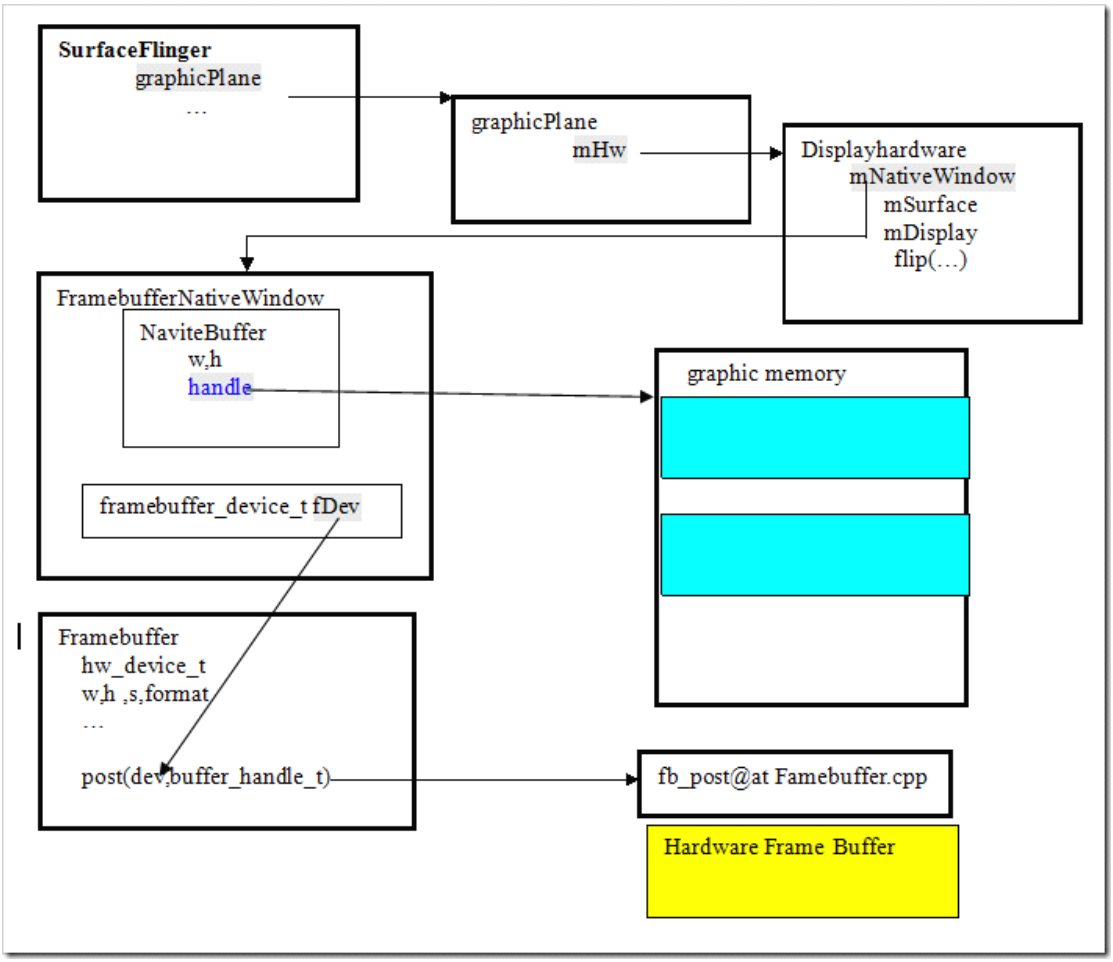


3 主要对象说明

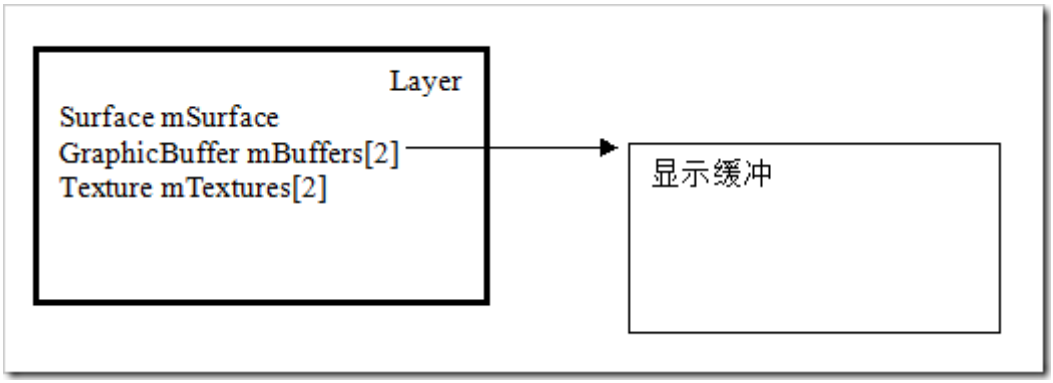
3.1 DisplayHardware &FrameBuffer

首先 SurfaceFlinger 需要操作到屏幕，需要建立一个屏幕硬件缓冲区管理框架。Android 在设计支持时，考虑多个屏幕的情况，引入了 graphicPlane 的概念。在 SurfaceFlinger 上有一个 graphicPlane 数组，每一个 graphicPlane 对象都对应一个 DisplayHardware.在当前的 Android（2.1）版本的设计中，系统支持一个 graphicPlane，所以也就支持一个 DisplayHardware。

SurfaceFlinger，Hardware 硬件缓冲区的数据结构关系图。



3.2 Layer



method:setBuffer　在 SurfaceFlinger 端建立显示缓冲区。这里的缓冲区是指的 HW 性质的，PMEM 设备文件映射的内存。

1) layer 的绘制

void Layer::onDraw(const Region& clip) const

```
{
    int index = mFrontBufferIndex;
    GLuint textureName = mTextures[index].name;
…
  drawWithOpenGL(clip, mTextures[index]);
}
```

3.2 mCurrentState.layersSortedByZ

以 Surface 的 Z-order 序列排列的 LayerBase 数组，该数组是层显示遮挡的依据。在每个层计算自己的可见区域时，从 Z-order 顶层开始计算，是考虑到遮挡区域的裁减，自己之前层的可见区域就是自己的不可见区域。而绘制 Layer 时，则从 Z-order 底层开始绘制，这个考虑到透明层的叠加。

4 SurfaceFlinger 的运行框架

我们从前面的章节<Android Service>的基本原理可以知道，SurfaceFlinger 的运行框架存在于：threadLoop,他是 SurfaceFlinger 的主循环体。SurfaceFlinger 在进入主体循环之前会首先运行：SurfaceFlinger::readyToRun()。

4.1 SurfaceFlinger::readyToRun()

（1）建立 GraphicPanle

（2）建立 FrameBufferHardware(确定输出目标)

初始化：OpenGL ES

建立兼容的 mainSurface.利用 eglCreateWindowSurface。

建立 OpenGL ES 进程上下文。

建立主 Surface（OpenGL ES）。 DisplayHardware 的 Init()@DisplayHardware.cpp 函数对 OpenGL 做了初始化，并创建立主 Surface。为什么叫主 Surface，因为所有的 Layer 在绘制时，都需要先绘制在这个主 Surface 上，最后系统才将主 Surface 的内容”投掷”到真正的屏幕上。

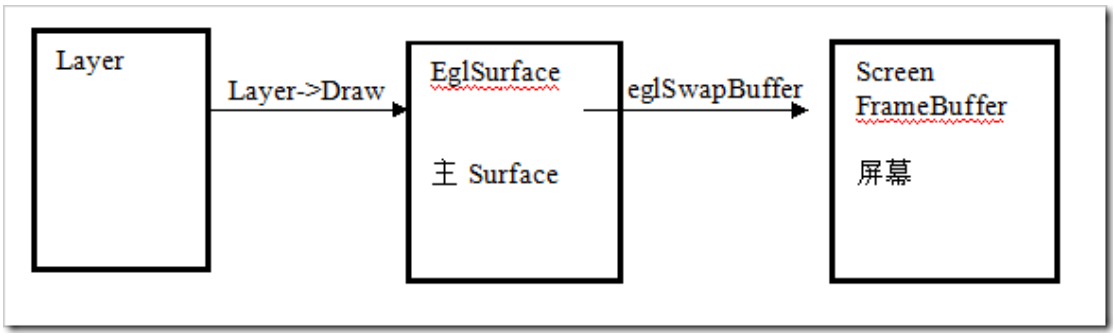
（3） 主 Surface 的绑定

1）在 DisplayHandware 初始完毕后，hw.makeCurrent()将主 Surface，OpenGL ES 进程上下文

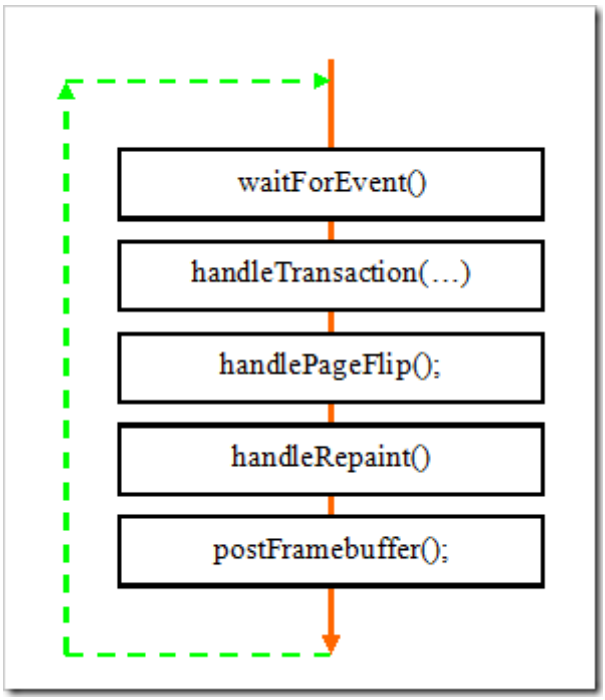
绑定到 SurfaceFlinger 的上下文中，

2）之后所有的 SurfaceFlinger 进程中使用 EGL 的所有的操作目的地都是 mSurface@DisplayHardware。

这样，在 OpenGL 绘制图形时，主 Surface 被记录在进程的上下文中，所以看不到显示的主 Surfce 相关参数的传递。下面是 Layer-Draw，Hardware.flip 的动作示意图：



4.2 ThreadLoop



(1)handleTransaction(…):主要计算每个 Layer 有无属性修改，如果有修改着内用需要重画。

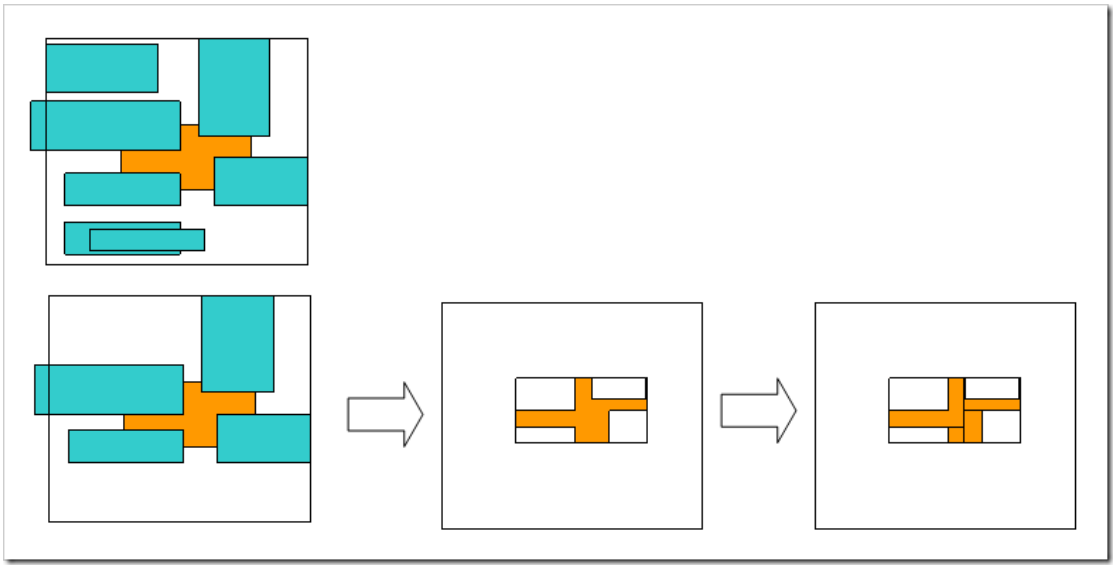
(2)handlePageFlip()

computeVisibleRegions：根据 Z-Order 序列计算每个 Layer 的可见区域和被覆盖区域。裁剪输出范围计算-

在生成裁剪区域的时候，根据 Z_order 依次，每个 Layer 在计算自己在屏幕的可显示区域时，需要经历如下步骤：

1）以自己的 W,H 给出自己初始的可见区域

2）减去自己上面窗口所覆盖的区域



在绘制时，Layer 将根据自己的可将区域做相应的区域数据 Copy。

（3）handleRepaint()

composeSurfaces（需要刷新区域）：

根据每个 Layer 的可见区域与需要刷新区域的交集区域从 Z-Order 序列从底部开始绘制到主 Surface 上。

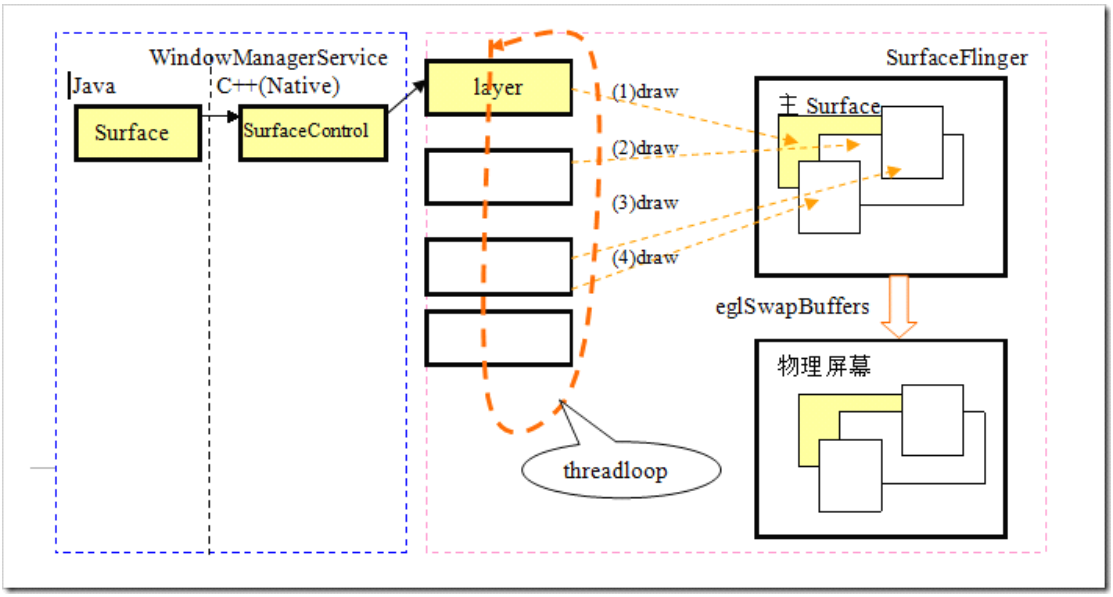
（4）postFramebuffer()

（DisplayHardware）hw.flip(mInvalidRegion);

eglSwapBuffers(display,mSurface) :将 mSruface 投递到屏幕。

5 总结

现在 SurfaceFlinger 干的事情利用下面的示意图表示出来：



Android 核心分析（27）-----Android GDI 之 SurfaceFlinger 之动态结构示意图

SurfaceFlinger 对象建立过程示意

1 SurfaceSession 的建立

客户端请求建立 Surface 时，首先在要与 SurfaceFlinger 建立一个 Session,然后再 Session 上建立一个 Connection 通过概念返回 Bclient 对象。WindowManagerService 在添加第一个窗口前会检查 SurfaceSession 是否建立，如何没有建立，将会新建立一个实例来代表与 SurfaceFlinger 的一个连接。

new SurfaceSession()@windowAddedLocked() @WindowManagerService.java。

SurfaceSession 的建立过程大部分是在 C++ Native 空间中完成的，表现在 SurfaceSession 的初始化函数：init()本地函数上。从下面的初始化函数可以看到：

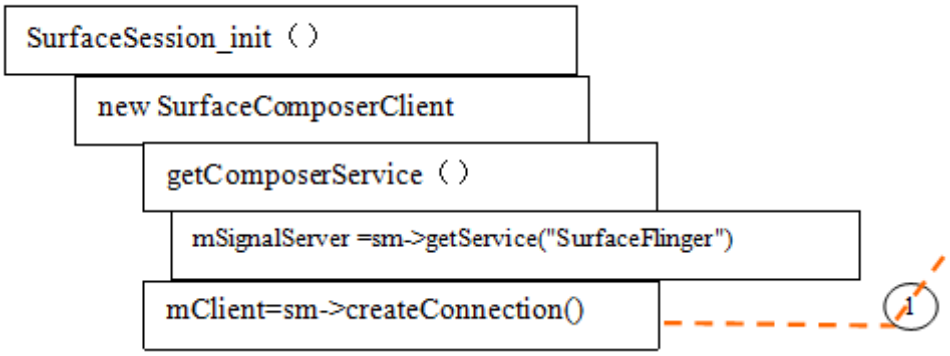
Init()<->SurfaceSession_init@android_view_Surface.cpp

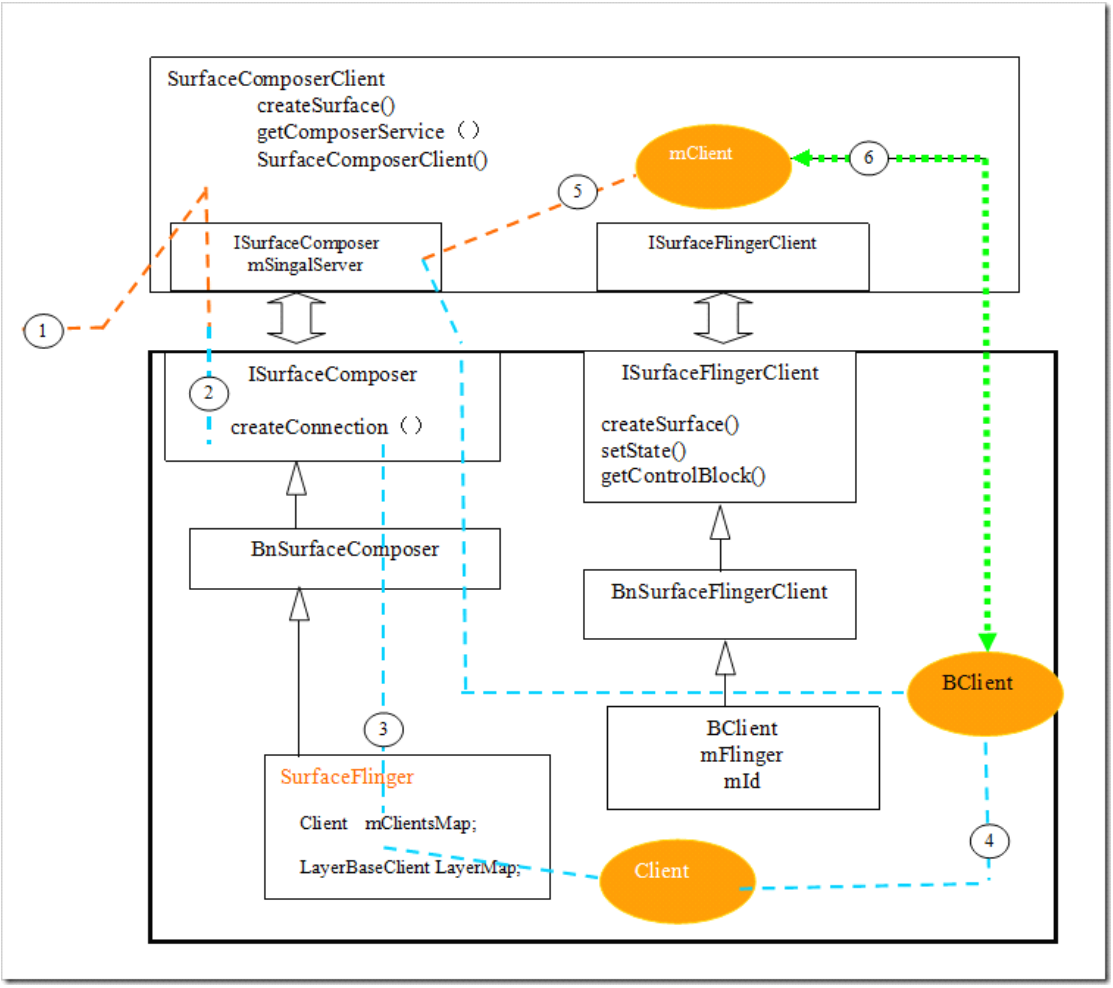
new SurfaceComposerClient

SurfaceSession 在 C++Native 空间建立一个 SurfaceComposerClient 实例。而该实例的建立实现了如下的与 SurfaceFlinger 通讯基础：

（1）建立了代理 SurfaceFlinger 服务的代理服务端

（2)建立了 IsurfaceFlingerClient 连接，在 SurfaceFlinger 端建立了对应的 Client，并将 BClient 返回给 WindowManagerService。





2 Surface 的建立

在 WindowManagerService 中 WindowState 类中，我们知道每个主窗口子啊需要是都需

要建立一个 Surface 与之对应。win.createSurfaceLocked()@relayoutWindow

Surface.java

Init()< -- >Surface_init(….,session,pid,dpy,w,h,format)@android_view_Surface.cpp

SurfaceControl surface(client->createSurface

在 mClient 的连接上：建立 ISurface 接口：

M_Client->greateSurface（...）@

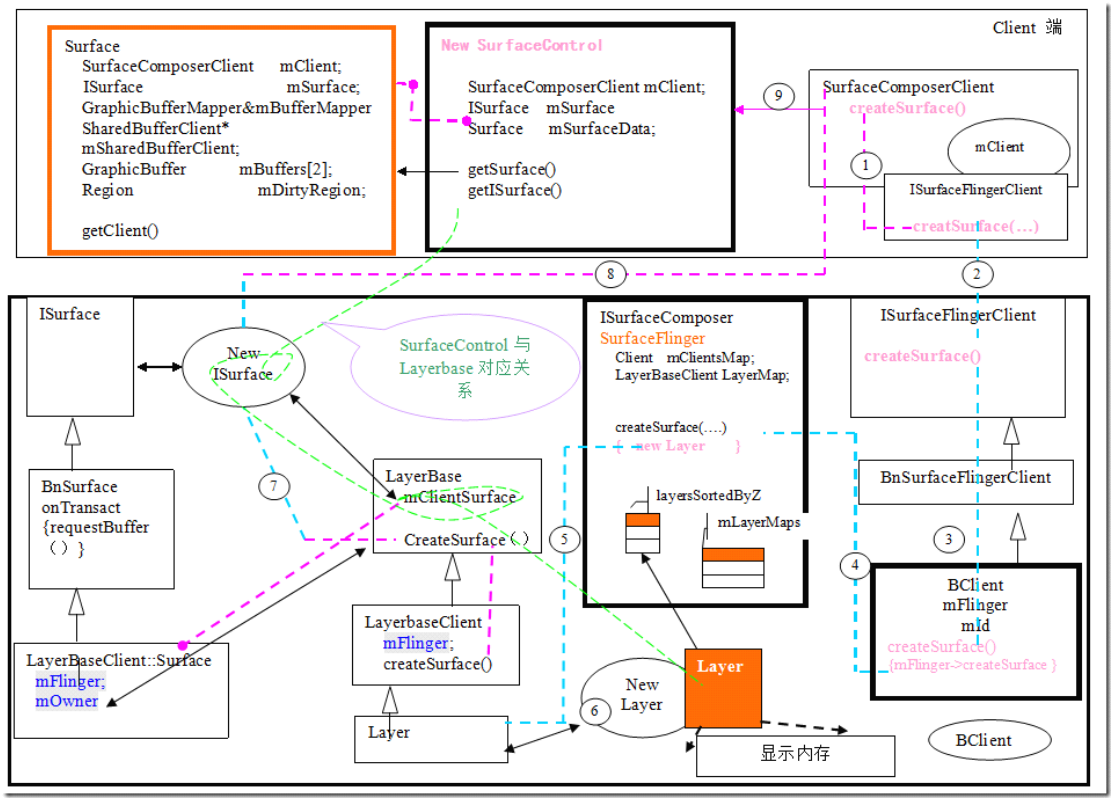
Bclient ::createSurface(mId...)@SurfaceFlinger.cpp

mFlinger->createSurface(clientid....)

createNormalSurfaceLocked

*createNormalSurfaceLocked:建立一个 Layer 分配显示内存

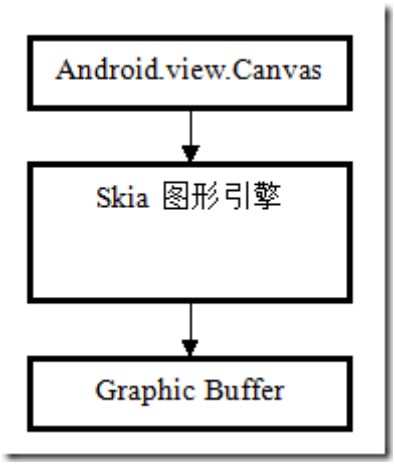
*createPushBuffersSurfaceLocked:建立一个 LayBuffer 但是不分配显示内存。



Android 核心分析（28）-----Android GDI 之 Surface&Canvas

Surface&Canvas

Canvas 为在画布的意思。Android 上层的作图几乎都通过 Canvas 实例来完成，其实 Canvas 更多是一种接口的包装。drawPaints ，drawPoints，drawRect，drawBitmap ...



1 Canvas 与 Surface 之间本质关系

对于本节，我们不去研究 Skia 图形引擎本身，我们需要了解的我们的所做的图形到底放置到了那个地方，并且这个 Canvas 如何与 Surface 连接在一起的。

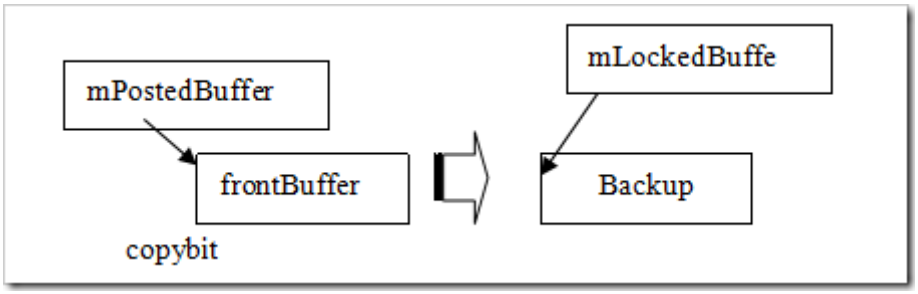
Canvas（Java）在 C++Native 层有一个 Native Canvas 的 C++对象所对应。

lockCanvas()@java

Surface_lockCanvas@android_view_Surface.cpp

SurfaceControl->new Surface(control) @Surface.cpp

Surface: lock 操作：



GraphicBuffer :lock

getBufferMapper().lock<-> GraphicBufferMapper ::lock

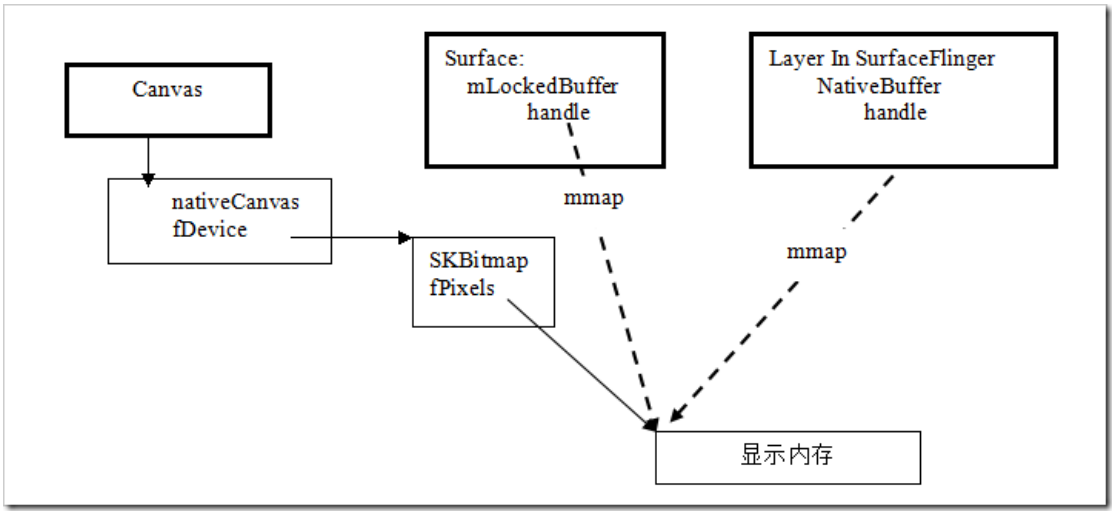
mAllocMod->lock<->gralloc_module_t::lock

通过 SurfaceLock 可取得 Surface（mLockedBuffe）所对应的图形缓冲区地址。

（1） 建立与 SkCanvas 连接的位图设备，而该位图使用上面取得的图形缓冲区地址做自己的位图内存。

（2） 设置 SkCanvas 的作图目标设备为该位图。

通过该过程就建立起了 SurfaceControl 与 Canvas 之间的联系。



2 View:OnDraw 的本源

不是使用 OpenGL 绘制时，Android 在 View 属性发生变化，新建 View 时，或者 Z-order 发生变化时，需要对系统屏幕上的 View 重新绘制，此时我们的 View 会执行 OnDraw(canvas)，这个根源在哪里呢？

ViewRoot.Java

performTraversals(..)

…

draw()

canvas = surface.lockCanvas(dirty);

…

mView.draw(canvas);

draw(cavas)@view.java

background.draw(canvas);

onDraw(cavas)

dispatchDraw(cavas)

onDrawScrolbars(cavas)

surface.unlockCanvasAndPost(canvas);