源码分析：

Context 是一个抽象类 。其实在Apllication、Activity 、Service都会存在一个Context对象，即Context的总数为Activity树加上Service个数加上一

一个Activity的入口时ActivityThread的main函数，在main函数中创建一个新的ActivityThread对象，并且启动消息循环（UI），创建新的Activity、新的Context对象，然后将该Context对象传递给Activity。

在main方法中，我们创建一个ActivityThread对象后，调用了其Attach函数，并且参数为false。在attach函数中，参数为false的情况下，（即非系统应用），会通过Binder机制与ActivityManagerService通信，并且最终调用handleLaunchActivity函数。

Context的实现类事ComtextImple。

从ContextImple类的部分源码看到，在虚拟机第一次加载该类的时候会注册各种serviceFatcher，其中包含了LayoutInflaterServive，将这些服务以健值对形式存储在一个Hashmap中，用户使用时只需要根据key来获取到对应的seviceFetcher，然后通过sericeDetcher对象的getSerice函数来获取具体的服务对象，当第一次获取时，会调用serviceFetcher的createService函数创建对象，然后将该对象缓存到一个列表中，下次再取时直接从缓存中获取，避免重复创建对象，从而达到单例模式。

LayoutInflater 是一个抽象类 在家在COntentImple是会通知LayoutInflater的ServiceFelcher注入到容器中，

这里调用了PolicyMange.makeNewLayoutInflater方法

PolicyManger通过反射构造了Policy实现类，这个类实现了Ipolicy接口，通过这种形式将具体的Policy类对外进行隐藏实现，PolicyManger实际上是一个代理类，具体的功能通过sPolicy对象（Policy）进行实现。

通过Policy代码，我们很清楚真正的LayoutInflater的实现类，就是PhoneLayoutInflater。

代码不多，核心的程序就是覆写了LayoutInflater的onCreateVIew方法，该方法就是在传进来的View名字前面加上android.widget.或者android.wbkiy前缀用以得到该内置的View（如TextView、Button等都是在android.widget包下面）。最后根据完整的路进来构造对应的view对象。

以activity的setContentView对象为例。

Public void setContentView(View view){  
getWindow().setContextView(view)

InitActionBar()

}

Activity的setContentVIew方法实际上调用的事windows的setContextView，而windows是一个抽象类，上下文提到的Windows的具体实现类时PhoneWindow

Public void setContentVeiw(int view)

{  
//当mContextParent为空的时候，先构建DecorView

//并且将DecorVIew包裹到mContentParent中

if(mCOntextParent==null)

{

installDecor();

}else{  
mContextParent.removeAllViews();

}

//解析LayoutResId

mLayoutInflater.inflate(layoutRestId,mContextParent)

mDecor中会加载一个系统定义好的布局，这个布局中又包裹了，mContentParent，而这个mContentParent就是我们设置的布局，并将添加到，parent区域，在phoneWindow的setContentView方法中也验证了这一点，是先构建的mContentparen这个对象，然后通过LayoutInflate的infalte函数指定布局的视图添加到mContentParent中

然后我们分析infalte方法，主要有下面几个步骤

1. 解析xml中的根标签（第一个元素）
2. 如果根标签时merge，那么调用rInfalte进行解析，rInflate会将merge标签下的所有子View直接添加到根标签中
3. 如果标签是普通元素，那么运行到temp=createtiveFrom（root，name，attrs）；调用CreateVIewFromTag对元素进行解析

这里就是通过xml的tag解析layout根视图

那么就是要解析的视图的类名，如RelativeLayout

1. 调用rInflate解析temp跟元素下的所有View，并且将这些子View都添加到，temp下
2. 返回我们temp。

CreateViewFromTag会将该元素的parent以及名字传递过来，当这个tag的名字中没有包含. (在名字中找. 返回-1)LayoutInfalte会认为这是一个内置的view。例如我们在xml中声明一个内置的View时大概是这样的。

<TextVIew/>

这里的TextVIew就是xml元素的名字，因此窒息感infate时就会调用onCreateVIew来解析这个TextView。当我们自定义View时，在xml中必写view的完整路径。

<com.dp.MyView/>

此时就会调用CreateView来解析View，

为什么要这么处理，？  
在上下文中PhoneLayoutInflater中我们知道，PhoneLayoutInflate覆盖了onCreateView方法，就是onCreateVIew方法，该方法就是在View标签名前面设置了 android.widget 前缀，然后再传递给createView进行解析，也就是说内置的View和自定义view最终都调用了createView方法进行解析，只是Google为了让开发者在xml中更方便定义View，只写View名称而不需要完整的路径。在LayoutInflater解析时如果遇到只写类名的view，那么认为是内置的view控件，在onCreateview方法中沪江androdi.widget前缀传递给createView方法，最后，在CreateVIew中构造view的完整路径来进行解析，如果是自定义控件，那么必须写完整的路径，此时调用createview且null进行解析。

CreateVIew比较简单，如果有前缀，那么构造View的完整路径，并且将该类加载到虚拟机中，然后获取该类的构造函数，并且缓存起来，再通过构造函数，来创建该对View的对象，最后将View对象返回，这就是解析单个view的过程。而我们的窗口中是一个视图树，LayoutInflate需要解析这颗完整的树，这个功能就是交给rInflate方法。

rInflate通过深度优先遍历来构造图树，媒解析到一个View就会递归调用rInfalte，知道这条路径下的最后一个元素，然后在回溯过来将每个view元素添加到parent中，通过rinflate的解析之后，整个视图树就构架完毕，当调用了activity的onResume之后，我们通过setContentVIew构建内容就会在我们试图中。‘

## Buidler

Android中最常用的就是AlertDilog.builder，使用Builder来构建复杂饿alertDialog对象，通过title、button、message等，将dialog的构造和表示进行分离，Buidler类（内部类）可以设置AlertDialog中的title、message、button等参数，这些参数都存储在类型为AlertCOntroller.ALertParams的成员变量P中，AlertController.AlertParams包含了了雨AlertDialog是图中对应的成员变量。在调用Buidler的create方法时会创建ALertDialog，并且将Builder成员变量P中的保存的参数对应到alertDailog的mAlert对象后对象中，即P.appl(dialog.mAlert)

Apply 函数中只是将AlertzParams参数设置到ALertCOntroller中，例如，将标题设置到Dialog对应的标题是图中，将Messageshe 设置到内容试图中等，我们获取到AlertDialog对象后，剔骨工show函数就可以显示对话框了，我们看看Dialog的show函数

在show函数中主要做了如下几件事情

1. 通过dispatchOncreate函数来调用ALertDialog的onCreate函数
2. 然后调用AlertDialog的onstart函数
3. 最后将Dialog的DecorView添加到windowsManager中

很明显这就是一系列典型的生命周期函数，那么按照惯例，AlertDialog的内容是视图构建按理应该在onCreate函数中，

onCreate函数中主要调用了ALertDialog的installCOntent方法，Dialog中的OnCreate只是一个空的实现，可以忽略它，那么ALertDialog的内容视图必然就在installContent函数中

installCOnetent调用了Windows对象，setCOnentView这个setContentView就与Activity中的一模一样·，实际上Activity最终也调用Windows对象的setcontentVIew函数，因此这里就是设置了ALertDIalog的内容布局，这个布局就是mALerDialogLayout字段的值，这个值在AlertController的构造函数中，进行了初始化·，

ALertController的构造函数可以看到，AlerDialog的布局资源就是alert-dialog.xml这个文件，他的布局文件就是系统对话框的布局形式，

当通过Buidler对象的setTIlte、setMessage等方法设置了具体内容时，就是将这些内容对应的视图中，儿ALertDialog也允许你通过setVIew传入视图，这个内容就试图就是替换 标题的中间内容。ALertDialog预留了一个costomPanel区域用来显示用户自定义内容试图。我们来看setupVIew函数

这个setuoVIew顾名思义局势初始化AlertDialog布局中的各个部分，如标题区域，按钮区域、内容区域。在该函数调用之后整个Dialog的视图内容全部设置完毕，而这些各区域的，视图都属于mALertDialogLayout布局中的子元素，Windows对象有关联了mALertDialogLayout的整个布局树，当调用完setUpVIew之后整个视图树的数据填充完毕，，当用户调用show函数时，WIndowsManage会将Windows对象的DecorVIew（也就是mALertDilogLayout对应的视图，当然DecorView还有一个层次）添加到用户的窗口上，并且显示出来，至于整个Dilaog就出现在用户视野中

## WindowsManager

Dialog的内容试图最终是通过windowsManger显示到手机频幕上，那么WIndowsManger：  
不但是Dialog，所有需要显示到屏幕上的内容（包含Activity）都是通过WindowsManger来操作了，看来WIndowManger是一个非常重要的子系统，这就是我们常常说的WMS（windowsManger Service）。我们只关心WindowsManger和WIndeowsMangerSErivve（WMS）Suface、SurfaceFlings等建立关联以及交互的基本过程

与windowsManager联系上的第一步就通过Context的getSystemService方法，上一节单例模式中我们说道，各种系统服务会注册到ContextImpl的一个map容器上，然后通过该服务的字符串来获取，WindowsManger也是在ContextImpl中注册众多服务之一s

registerServic(WINDOw\_SERVIVE,new ServiceFetCher(){  
Display mDefaultDisplay;

public Object getService(COntextImpl ctx){  
Display display=ctx.mDisplay;

if(display == ull)[

displayManger dm =(dispalymanager)ctx.getOuterContext().getSystemService(Context.DISPLAY\_SERIVICE)

mDefaultDisplay=dm.getDisPlay(DIsPlay.DEFAULT\_DISPLAY);  
}

display=mDefualtDispaly;

}

return new WIndowsMagerImpl(display)//构造WindowsMangerImpl对象

我们看到了widowsManger在java层的具体实现，也就是WindowMangerImpl。。那么Dialog对象又是怎么获取到WIndowManger呢？windowsMange是注册到ContextImpl中的，getSystemServie也就是Context定义的接口，因此，需要先从Dialog的构造函数进行分析，然后我们看到最终是通过Windows对象的setWindowsManger函数将Windows与WindowsManger建立联系，该函数是在Windows类中，我们然后再看setWindowsManger函数 最终会调用WIndwosMangerImpl的createLocalWIndowsManger方法：这个方法只是单纯的构建了一个WIndwosMangeImpl对象，鱼ContextImply注册的WindowsMangerImpl不同的是，这里多了一个parentWIndows参数，也就是说，此时构建的WIndowsMangerImpl对象是与具体的Windows关联的，而CotnextImpl注册并没有此参数（只是一个mDisplay）

我们看看WindowsMangerImpl的和核心代码；很显然WindowsMangerImpl也不是具体的干活，他也只是Imp羊皮的狼，添加View、更新删除都是由，WIndowsMangerGlobal这个类，在这一切就绪之后，会调用WindowsManger的addVIew方法请求系统将该View显示到屏幕上，实际上调用的事WIndowsMangerGlobal中的addVIew方法 我们接着跟踪addView方法我们看到：

1. 构建ViewRootImpl
2. 将布局参数设置给View；
3. 存储这些VIewRootImpl、View、LayoutParam到列表中
4. 通过ViewRootImpl的setView将View显示到窗口上。

很多读者可能对ViewRootImpl并不陌生，咋一看他的名字可能会误以为他是一个View，但实际上不是这样的，他继承自Handler类，是作为native层与java层的View系统通信的桥梁，比如我们熟知的performTraversals函数就是收到系统绘制的View消息之后，通过调用视图树的各个节点的meature、layout、draw、方法来绘制整个视图树

。对于widowsManger有些了解读者可能注意到，经过上述分析，我们才到Framework层，而wms可能是运行在Native层的，FrameWOrk层如何与Natice层建立关联？

既然ViewRootImpl事Framework层与View通信的桥梁，我看到ViewRootImp的构造函数，有一WindowMangerGlobal.getWindowsGLobal，getWIndowsSession（）方法，通过函数命名以及WindowsMangerGlobal的作用来看，这是与Native通信的地方

在getWIndowsSession函数中，Framework层首先通过getWindowsMangerservice 函数获取到IWindowsManger对象，该函数中通过ServiceManger.getService函数获取WMS  
，并且将WNS转换为IWIndowMange类型。我们看看ServiveMange.getService函数：  
从程序中可以看到ServifeManger.getService函数返回的是IBinder对象，也就是说AndroidFramework与WMS之间也是通过Binder机制进行通信，到了这一步我们已经与WMS  
建立了初步联系获取WMS之间也是通过IBIinder机制进行通行，到了这一步我们已经与WMS建立了初步的联系。获取WMS之后，又调用了UWIndowsMange.stub类的asInterface函数，将获取到WMS的Binder对象转换成WindowsManger对象，通过opensession函数来雨WMS建立一个通信回话，相当于Framework层与native层建立一个长期的合作，双方都是通过这个session来交换信息，

但是此时Dialog或者Activity的view并不能显示在手机上，WMS  
只是负责管理手机屏幕上View的z-order，也就是说WMS  
管理当前状态下的哪个View应该在最上层显示，仔细想想，你发现其实WMS  
管理器并不是windows，而是view，只不过他管理直属于某个windows下的view。

鱼wms建立session之后就到了调用viewRootImpl的setVIew方法了，该方法会向wms发起显示dialog或者activity中的decorVIew请求，

setView很复杂，但是我们主要关注两步：

1. requestLayout；
2. 向WMS发起显示当前Winwos的请求

我们看看requestalyout函数

就是往handler发送了一DO\_Traversal消息，这个消息会触发整个视图树的绘制操作，也就是最终会执行performTraversal函数，这是一个极为复杂又非常重要的函数，简单来讲它主要做了如下操作：  
1获取Surface对象，用于图形绘制

2丈量整个视图树的各个veiw大小，performMeasure函数

3布局整个视图树，，performayout函数

4绘制整个视图树，performDraw函数

在第四步中，framework会获取到图形绘制表面surface对象，然后获取到他可能绘制区域，也就是我们的canvas对象，然后Framework

在—draw函数中会调用到需要绘制的区域，以及判断是否使用GPU进行绘制。通常情况下使用的是GPU绘制，也就是调用的是drawSorware函数来绘制，

上述所属：上述的视图树绘制代码中主要分为：  
1）判断是否使用CPU绘制还是GPU绘制

2）获取绘制表面suerface对象，

1. 通过surface对象获取并且锁住canvas绘图对象
2. 从decorView开始发起整个视图树的绘制流程
3. surface对象解锁Canvas，并且通知SurfaceFlinger更新视图

内容绘制完成之后请求wms显示该窗口的内容，到此、activty、dialog等组的view就显示到用户频幕上了。整个wms系统是极为复杂的，设计的概念，技术非常广泛，各个层次交互错综复杂

## 原型模式

创建对象种类，并通过拷贝这些原型创建新的对象

使用场景：  
1）类初始化需要非常多的资源，这个资源包括书数据、硬件资源等，通过原型拷贝避免这些消耗

2）通过new产生一个对象需要非常繁琐的数据准备或者访问权限，这时可以使用原型模式

3）一个对象需要提供其他对象访问，而且个个调用者能都需要修改其值，可以考虑使用原型拷贝多个对象供调用者使用，保护性拷贝

但是通过cloneable接口的原型模式在调用clone函数构造实力并不一定比new操作速度快，只有当通过new构造对象较消耗高时，通过clone采用一定效率

Uri uri=Uri.parse(“smsto:008”)

Intent shareIntent=new Intent(Intent.ACTION\_SENDTO,uri)

shareIntent.putExtra(“sms\_body”,”sms”)]

Intent intent =(Intent)intentSharedintent.clone();

startActivity(intent)

上述代码首相构造一个发短信的Intent对象，并且设置了啦短信内容，然后通过原始Intent的clone()方法构建了一个副本，在使用者个副本Intent进行跳转，

我们可以看到Clone方法（Intent源码）实际上并没有调用super.clone（）方法来实现对象拷贝，而是调用了new Intent（this），使用clone和new需要根据构造对象的成本来决定，如果对象的构造成本比较高或者构造较为麻烦，使用clone和new需要构造对象的成本决定。

Intent 的查找和匹配：

如何通过Intent来查找对应的组建的呢？

对于第一章我们说过，系统启动之后会就会注册各种系统服务，比如WIndowsMangerserive、activitymanagerservice等，其中一个就是packagemanagerservice（PMS）PMS启动之后，会扫描系统中已经按安装的apk目录，例如系统app的安装目录为/system/app,第三方应用的目录为/data/app,pms会解析apk包下的androidmanifest.xml文件得到app的相关信息，而每个androidamnifest.xml哟包含了activity、service等组件的注册信息，党pws扫描并且解析完这些信息之后就构建好了整个apk信息树

我们解析pws看看他的packagemanagerservice方法

pms不仅需要加载系统的以安装的各类apk，在此之前还需要加载framework资源文件和核心库，加载了资源与核心库之后才开始对扫描的指定目录下的apk文件进行解析，这个函数只给出了系统apk安装目录和第三方应用安装目录，扫描函数为scandirli，我们看看这个函数内部实现；

scandirLI函数通过调用scanpackageLI函数进行解析：  
在scanpackageLI中首先构造了PackageParser，也就是一个apk包解析器，其中调用了packageparser对象的parsePackage函数，

parserPack函数中会根据packageFile的类型来选着不同的解析方式，例如packageFile是一个文件夹，则会解析文件夹下的所有apk：如果是一个文件那么屌用parseMonolithicPackage解析单个apk。我们直接看parseMonolithicPackage函数。

在packageparse的parsepackage函数中我们看到，代码得到一个androidManifest.xml文件解析器，在初学android时我们知道Acgivity、Servie等组件都需要注册到androidManifest.xml中，解析这个文件就是构建一个Intent有关的信息表，然后我们接着看parsePackage(res,parse,flags,errorText)函数：  
parsePackage函数才是正真解析andriodManifest.xml文件的，这个给出两个标签，即Application和use-permission.application中包含了Activity、service、等标签，也就是Intent所需要的标签，我们看看解析Application标签函数parseApplication

看到activity，servife、provider、reeiver等标签我们行的石头总算是陆地了，从parseApplication中我们看到这个过程就是普通的xml解析，根据不同的标签调用不同的解析方法，例如解析activity则会调用parseActivity函数，然后返回一个activity实例，并且将这个实例添加到Package对象的activities列表

此时、我们需要回到上述的packageManageserviec类中的三餐PackageLI最后一，也就是调用了scanPackageLI函数：  
这个函数调用了ScanPageDirtyLI函数，scanPackageDirtyLI函数会将解析到的Activity、servive等添加到对应的mActivity、mService等列表中，这个函数很复杂，大概有一千行，

我们看到，这里将上一步解析道的Activity、service添加到mActtiviyird、mServices中，这些类型定义是PackAgeMangerServiec字段，

到了这一步，整个已安装的apk的信息树建立了，每个apk的应用名、包名

图标、activiy、servive等信息都存储在系统中，当用户使用Intent跳转到Activity或者启动某个Service时，系统则会到这个信息表中进行查找，符合要求的组件会被启动起来，这样就通过Intent将系统的各个组件联系在一起，使得Android系统称为已组件可复用的、灵活的系统

精准匹配：  
上述分析了apk的信息表的构建过程 ，下面我们来分析Intent的查找与匹配过程

startActivity这个函数经过几次转发，最终会调用startActivityForResult  
，startActivtyForResult函数直接调用了Instrumenttation中的execStartActivity函数：  
execStartActivtiy里面其实就是调用了ActivityMangerService的startActivty方法，，这个方法里面其实就是调用了AvtivtiyStackSupervisor的startActvityMaywait方法，该方法最后会调用PMS  
的resolveIntent方法，pms又出现在我们的视野里，在resolveIntent方法终究是调用了自身queryIntentActivtis方法，queryIntentActivities会返回一个activityInfo对象列表，也就是符合Intent的ActivityInfo列表，而每个Av ti vi t y I n fo对象就是一个Actiivty的档案对象，记录了一个Activity相关的信息，queryIntentActivities：  
上述函数运行大致：如果Intent指明了Componet，那么直接通过Componet就可以找到ActivityIfo列表，这个列表的数量只有一个，这个AvtivityInfo就是注定的那个组件：如果没有制定具体的组件，那么Componet为空，此时先查看Intent是否指定了要跳转的目标组件所在的包名，则会通过报名获取到对应的ActivityInfo：否这就需要通过ActivityIntentResolver等类的queryIntentForPackage进行模糊匹配，例如根据根据Action、Category 等，这里需要注意的是：mActivities就是上个章节说到的存储了从AndroidManifest.xml中解析到的Activity，既然已经找到了Activity信息，那么最好就是启动对应的Activity ，对于显式Intent来说，就是直接跳转到具体的Activity中；对于隐式Intent来说，可能会弹出窗口让用户选着具体的应用：  
到最后我们来整理一下思路：在系统启动PackageMangerServive会启动，此时PMS将解析所有以安装的应用的信息，构建以给信息表，当用户通过Intent来跳转到某个·组件时，会根据Intent包含的信PMS中查找对应的组件列表，最后跳转到目标组件中。

原型模式：优缺点：

原型模式是在内存中二进制的拷贝：要比直接new以给对象性能很多，特别是是要在在一个循环体内产生大量的对象时，原型模式可以更好体现优点；

缺点；直接在内存中拷贝，构造函数不会执行，在实际开发中应该注意这个问题。优点就是减少了约束，缺点也就是减少了约束

## 工厂模式：

定义一个用于创建对象的接口，让子类决定是实例化那个类，

android 源码用到工厂模式：  
以List和Set为例 list和set都继承Collection接口，而Collection接口继承于Iterable接口很简单，就一个Iterator方法：  
这意味着List和set借口都要继承该方法，平时比较常用的间接实现类ArrayList和Hashset中iterator方法的视线就是构造并返回一个迭代器对象；

HashSet和iterator方法中会返回成员变量backingMap中对应的HashSet对象元素的迭代器对象，最终返回的是keyset中的一个迭代器对象；

ArrayList和HashSet中的iterator方法其实就是相当于一个工厂方法，专为new对象而生，这里iterator方法是构造并返回一个具体的迭代器。当然，java源码中对工厂方法的对应范例也不少

OnCreate(Bundle savedInstanceState){  
super.onCreate(saveInstanceState);

SetContentView(new LinearLayout(this);

}

这里构造一个线性布局LinearLayout对象设置成当前Activity根部局，实质上onCreate方法就相当于一个工厂方法，？因为LinearLayout是一个ViewGroup，而ViewGroup又继承View，简单来说，所有控件都是View的子类，我们在onreate方法中构造了一个View对象，并设置当前也买呢ContentView返回给framework处理，如果现在又有一个Bactivity类，这时我们oncrate设置一个布局，这不就是一个工厂模式？

关于onCreate方法：  
我们知道onCreate相当于一个工厂方法，那么这个方法又是从何而来。

因为真正入口是在AvtivityThread类中，ActivityTHread中含有我们熟悉的main方法；

ActivityThread是一个final类，也就是说他是不能被继承的，一个应用程序对应一个ActivityThread，zygote进程孵化出一个新的进程之后，会执行ActivityThread的main方法，main方法准备looper和消息队列，然后调用ActivityThread的attach方法将其绑定到ActivityManagerServie中，开始不断读取消息队列中的消息并分发消息，这是一个典型的基于事件驱动的模型。我们看看attach 方法

Attach方法里根据boolean类型的参数system分为两部分，我们只看飞system部分，这个部分主要是调用ActivityMangeServive的attach方法将AvtivityThread绑定至ActivityManageService对象，而attachApplicatin方法中，主要还是调用了attachApplicationLocked方法来进行绑定。

AttachApplicationLocked方法逻辑相对来说要很长很多，但是，其中比较重要的部分就是bindApplicatoin和attachApplicationLock这两个方法的调用：

bindApplication参数很多，主要是将ApplicationThread对象绑定到AvtivityMangeServie，这个mStackSupervisor指向的是一个ActivityStacksupervisor类的对象，该类中的attachApplicationLocked方法描述如下：

我们看到这段条件判断式中的realstartActivityLocked方法，在其内部处理了真正启动Activity的逻辑。

RealstartActivityLock方法中首先会启动Activity的参数信息，准备完毕后调用ApplicationThread的scheduleLaunchActivity方法准备启动Activity

scheduleLaunchActivity对启动信息进行准备，这里构造了一个ActivityClienRecord对象，并将相关参数设置，最后通过sendMessage方法发送一个启动消息队列，由于ActivityThread的Handler处理启动，ActivityThread内部有个继承于Handler的子类H，在其内部负责处理消息，比如这里我们发送的是标识LAUNCH\_ACTIVITY，对应到H中则会调用handleLuanchActivity方法：  
这个方法会调用performLaunchAvtivity方法：Instrmentationcall中callActivityonCreate会执行对AvtivityonCreate方法调用

最终会调用到Activity中的onCreate方法

## 抽象工厂模式：

为创建一组相关或者是相互依赖的对象提供一个接口，而不需要指定他们的具体类。

我们知道Activity的onCreate方法可以看作一个工厂方法：

同样的对于另外一个组件servive来说，其onBind方法也可以看作是一个工厂方法：

如果从framework的角度来看Activity和Service都可以看作是一个具体的工厂。另一个更确切的例子就是Android底层对MediaPlayer的创建：

MediaPlayerFactory是Android底层为创建不同的MediaPlayer所定义的一个类，其中就是调用createPlayer函数来创建不同的MediaPlayer，而每一种具体的MediaPlayer创建完毕后最终都会调用MediaPlayerFactory的registerFactory\_l函数将其注册到MediaPlayerFactory中：

这里的MediaPlayerFactory其本质只是用来管理Android内置的四种不同的MediaPlayer，而对于每一种具体的MediaPlayer则由一个具体的Factory类来创建，比如StagefrightPlayerFactory这个抽象工厂类创建StagefrightPlayer：  
NuPlayerFactory这个抽象工厂创建NuPlayerDriver的代码：

SonivoxPlayerFactory这个抽象类工厂创建MidiFile

…

这四种MediaPlayerFactory

分别会生成不同的MediaPlayer基类：StagefrightPlayer…。四者均继承MediaPlayerBase。

## 策略模式

定义了一系列算法，并将每个算法封装起来，并且使用它们还可以相互转换，策略模式让算法独立于使用他的客户而独立变化。

针对同一类型问题的多种处理方式，仅仅是具体行为有差别：

需要安全地封装多种同一类型的操作时。

出现同一抽象类有多个子类，而又需要适应if-else或者switch-case来选着具体子类

android 时间插值器：

········TimeInterpolator也就是时间插值器。他的作用是根据时间流逝的百分比来计算出当前属性值改变的百分比，系统预设的闲心插值器LinearInterpolator用于均速加速，结尾时动画减速；减速插值器DecelerateInterpolator 用于随着时间的推移动画越来越慢

当然插值器并不是一个人，他还有一个TypeEvaluator ，也就是类型估值器。它的作用是根据当前属性改变的百分比来计算改变后的属性值，也就是说TypeEvaluator计算得到的才是属性的值。时间插值器计算得到当前的点的流逝百分比，TypeEvaluator根据这个百分比、属性起始值、目标值计算出当前时刻该属性的值，最后这个值被设置给View，不断地重复这个过程形成了动画。系统预设的有整形属性IntEvaluator。浮点型属性FloatEvaluator和Color属性 ArgbEvaluato

时间插值器运用于动画中，而动画作用于View上。因此，需要从View上切入到动画，再从动画中找到插值器的身影。当我们要对某个View执行某个动画时，我们首先会创建一个Aniamtion对象，然后调用View的startAnimation（Aniamtio animation）方法

startAnimation中首先设置了动画的起始时间，然后将该动画设置到View中，最后在向ViewGroup请求刷新视图，随后Viewgroup就会调用dispathDraw方法对这个View所在的区域进行重绘。对于某个View的重绘。对于某个View的重绘最总会调用ViewGroup中的drawChild（Canvas canvas，View child，long drawingTime）

这个函数只是进行了一个转发，我们看看draw(canvas canvas,Viewgrop parent,long drawingTime)方法是如何调用Animation

可以看出父类调用View的draw方法中，会先判断是否设置了清楚动画的标记，然后在获取该View动画的信息，如果设置了动画，就会调用Viw中的drawAnimation方法

在drawAnimation中主要是操作动画的初始化、动画操作、界面刷行。在drawAnimation中首先会判断动画是否进行初始化，如果未初始化，先初始化，然后调用动画监听器onstart函数。动画的具体实现是通过Animation中的getTransFormation方法：  
这个函数会获取已经流逝的动画执行的时间百分比，然后再通过插值器来重新计算这个百分比，也就是调用了插值器的getIntrepolation方法来获取当前的时间百分比，并且以此来计算当前动画的属性值，例如，线性插值器的输出百分比就是输入的百分比，不做任何处理，使得动画的速率不会发生变化。在默认情况下，AccelerateInterpolator的getInterpolation方法中会对input进程乘方操作，这个input就是流逝的时间百分比，input的取值范围0.0f-1.0f，当input逐渐增大时，input\*input的变化范围越来越大，使得动画的属性值在同一时间段内的变化范围更大，1从而实现了加速动画的效果，例如，动画执行总时间为1秒，使用的是AccelerateInterpolator插值器，在动画执行了100毫秒时百分比为0.1，那么此时通过插值器的计算，百分比成了0.01；又过了100毫秒，百分比经过插值器计算会变为0.09 在同一时间段百分比差距越来越大形成了加速效果。

在调用了插值器getInterpolation方法之后，会继续调用动画类的applyTransformation方法将属性应用到对应的对象中，applyTransformation在Animatiob基类中是空实现，那么我们选着缩放动画scaleAnimation：

当执行完applyTransformation之后，View的属性就发生了变化，不断重复这个过程，动画随之产生了。

这个过程中，插值器扮演了很重要的角色，他将动画的速率计算封装到一个抽象中也就是interpolator中，该接口中只有一个getInterpolation方法，通过这个方法修改动画流失时间百分比，以此达到动画加速、减速等效果

Interpolator就是这个计算策略的抽象，LinearLayout、cyclerinterpolator等插值器就是具体的实现策略，通过·注入不同的插值器实现不同的动态效果。

深入动画：

在android3.0之前，android提供了集中动画类型：View Animation 、Drawable Animation、Property Animation。View Animation相当简单，不过只能支持简单缩放、平移、旋转、透明度这几个基本动画，且有一定的局限性。 例如希望View有一个颜色的切换动画；希望可以使用3d旋转动画；向当前动画停止时，View的位置就是当前的位置，这些View Animatoin都无法做到。因此Android 3.0提供了属性动画，这使得动画系统变得极其强大·。

Aniamator 通过PropertyValuesholder来更新对象的目标属性，如果用户没有设置目标属性Property对象，那么会通过反射的形式调用目标属性的settter方法来更新属性值。这个属性值则通过KeyFrameSet的计算得到，儿KeyframeSet又是通过时间插值器和类型估值器来计算，在动画的执行过程中不断计算当前时刻目标属性的值，然后更新属性值来达到目的

属性动画的和核心类：

ValueAnimation ：该类是Animator的子类，实现了动画的整个处理逻辑，也就是属性动画最为核心的类。

ObjectAnimator：对象属性动画的操作类，继承自ValueAnimator，通过该类使用动画的形式操作对象的值。

TimeInterpolator：时间插值器。它的作用是根据时间的流逝的百分比来计算出当前属性值改变的百分比，系统的预设有线性插值器LinearInterpolator、加速插值器和减速插值器

TypeEvaluator：TypeEvaluator时类型估值器算法·，它的作用是根据当前的属性值改变的百分比计算改编后的属性值，系统预设的有针对整数属性IntEvaluator、针对浮点型属性FloatEvaluator和针对Color属性

Property ：属性对象

Property ValuesHolder：PropertyValuesHolder是持有目标属性Property、setter和getter方法、以及关键帧的集合的类

KeyframeSet：存储一个动画的关键帧集合

AnimationProxy：在Android3.0以下使用View的属性动画的辅助类

在andorid3.0以上以及以下的系统版本中他的执行代码有什么不一样，这就是我们本节要讲的：首先从他的入口级ObjectAnimation入手

在ObjectAnimation的ofFloat函数中会先构建属性动画对象，然后更具设置的属性值来初始化各个时间段对应的属性值，这个属性值就是values参数，它是一个可变参数，如果是一个参数，那么该参数的目标值：如果是两个参数，那么一个初始值，另一个就是目标值：我们看看setFloatVlaues函数的实现：

我们看到setFloatVlaue出现了一个类PropertyValuesHolder，这个类是该动画库的一个核心类之一，它的作用就是保存属性的名称 和他的settter、getter方法，以及他的目标值。PRopertyValuesHolder：

该类是属性和属性值的辅助类，它保存了属性的名称、setter、getter以及动画属性在duration时间段内各个时刻对应的属性数值。这样，当执行动画时、动画库只需要根据动画的执行时间，到mKeyfrmeSet中查询这个时刻对应的属性值，然后修改执行动画的对象的目标属性值，连续这个过程即可以达到动画的效果，在这个例子我们的属性是scaleX，目标属性值式0.3f。因此对应的属性类FloatPropertyValuesHolder，还有一种是IntPropertyValuesHolde

### 补充

## 状态模式

状态模式的行为是由状态来决定的，不同的状态有不同的行为，状态模式和策略模式结构几乎完全一样，但是他们的目的、本质却完全不一样状态模式的行为是平行的，不可替代的，策略模式的行为是彼此独立，可相互替换的，用一句话来表达，状态状态模式把对象的行为包装在不同的状态对象里，每一个状态对象都有一个共同的抽象状态基类。状态模式的意图是让一个对象在其内部状态改变的时候，其行为也为随之改变。

当一个对象的内在状态改变允许改变其行为，这个对象看起来像是改变了其类

使用场景：

一个对象的行为取决于他的状态，并且他必须的运行根据状态的改变他的行为

代码中包含大量与对象状态有关的条件语句，例如，一个操作类中包含大量的分支语句

状态模式将每一个条件分支放入到一个独立的类中，这使得你可以根据对象自身的情况将对象的状态作为一个对象，这一对象可不能依赖于其他对象而独立变化，这样通过多态来去除过多的、重复的if else等分支语句

### 补充

## 责任链模式

责任链模式多个节点首尾连接所构成的模型称为链。对于链式结构，每个节点都可以被拆分开再连接，硬刺，链式结构也具有很好的灵活性，将这样一种结构应用于编程领域，将每一个节点看作是一个对象，每一个对象拥有不同的处理逻辑，将一个请求从链式的首端出发，沿着链的路径依次传递给每一个节点对象，直至对象处理这个请求为止。

使用多个对象都有机会处理请求，从而避免了请求的发送者和接受者之间的耦合关心，将这些对象连成一条链，并沿着这条链传递该请求，知道有对象处理她为止。

使用：

多个对象可以处理同意请求，但具体的由哪个对象处理则在运行时动态决定

在请求处理者不明确的情况下向多个对象中的一个提交一个请求

需要动态指定一组对象处理请求

Android 源码中的责任链式：  
责任链式在android源码中比较类似的实现莫过于对事件的分发处理，每当用户接触屏幕时，android都会将对应的事情包装成一个事件从ViewTree的顶部至上而下地分发传递，关于时间的包装我们会在命令模式中，关于IMS对事件的处理我们也会在访问者模式中讲解，我们主要看看ViewGroup中如何将事件分派到View的，我们知道ViewGroup中执行事件派发的方法是dispathToochEvent

我们再看看dispathTrnsformedTouchEvent方法是如何调度子元素dispatchTouhEvent方法的：

ViewGroup投递的递归调用就类似于一条链式，一旦其寻找到责任者，那么将由责任者持有并消费掉该次事件，具体地体现在onTouhEvent方法中返回值的设置，如果onTouchEvent返回false，那么意味着当前View不会是该次事件的责任人 将不会对其持有；如果为true则相反，此时View会持有改时间并不在向外传递

## 解析器模式

解析器模式是一种用的比较少的行为型模式，其提供了一种解释语言的语法或者表达式的方式，该模式定义了一个表达是接口，通过该接口解释了一个特定的上下文，在这么多的设计模式中，解释器模式在实际运用上相对来说少很多，因此我们很少会自己去构造一个语言的文法，虽然如此，既然他能够在设计模式中：

给定一个语言，定义它的文法表示，并定义一个解析器，他解析器使用该表示来解释语言中的句子

我们PackageParser

PackagePaser为Activity、Serviec、Provider、Permission等构件在其内部内部类的方式创建了对应的类、按照解析器模式的定义，这些类其实都对应AndroidManifest，xml文件中的一个标签，也就是一条文法，其在对该配置文件解析时充分运用了解析器分离实现，、解释执行的特征

在android中国，解析某个apk文件会调用到packageMangerservice中的sendPackageLI方法：该方法有两种实现

两者唯一区别就是签名列表总第一个参数第一种实现为 File类型的对象，而第二种是实现为PackageParse.Package类型的对象，在具体的解析某个文件时会先调用第一种实现解析apk文件，在调用第二种实现解析后的信息保存至PMS

而parsePackage方法的第二种实现逻辑要复杂多，其内部主要对这个AndroidManifest.xml配置文件进行解析

可以看到parseagepack方法主要作用就是对AndroidManifest.xml配置文件中manifest下的每个字节点进行·解析·，这里主要看看parseApplication方法是如何application节点进行解析：

从程序中可以看到，parseApplication除了对applicatin节点内地属性进行解析外，其主要逻辑还是通过遍历解析其子结点，然后继续分别调用不同的解析方法对其进行解析，这里以parseActviity为例，parseActivity的具体逻辑逻辑逻辑如下。

这里主要注意的是，parseActvity方发不仅仅承担着对Activity的解析，其同样承担着对BroadCast的解析，与parseApplicaiton方法类似，parseActivity内部逻辑也是遍历其子标签，并调用相应的方法对其进行解析，如上所述parseIntent和parsezMetaData，其逻辑也很好理解，这里就不再多说。

## PackManagerService

PackageManngerService 起作用就是对我们的apk安装包进行管理服务，其定义在framwworks\base\service\java\com\androdi\server\pm\PackageMangerService.java中，该类在android 19中有一万多行，但是大部分逻辑都是在处理类似我们上面说的文件解析和数据结构操作，作为Android在java服务之一，其与所有的服务一样都是，其与所有其他所有的服务一样都是由SystemServer所启动。

ServerManger是Binder进程间通信机制的守护进程，其目的很简单，就是管理andorid系统里的Bidner对象，这里我们重点看PackageMangerServer的构造过程

整个PackageManagerService的构造过程有点长，但是逻辑并不复杂，调理也很清晰，关于PackAgerMangerService中的一些具体的方法，总体阿里说 PackManagerService的作用就是如他的名字一样，主要是用来管理应用程序包，其与其他系统服务的 通信业表单一，

## 命令模式

命令模式相互对于其他的设计模式更为灵活多变，它是将请求分装成一个对象，从而让用户使用不同的请求把客户端参数化，对请求排序或者记录请求日志，以及支持可撤销的操作。

使用场景：  
需要抽象出待执行的动作，然后以参数的形式提供出来，类似过程设计中的回调机制的一个面向对象的替代品

在不同的时刻指定、排序和执行请求。一个命令对象可以有与处世请求无关的生存期

需要支持取消操作

支持修改日志功能，这样当系统崩溃，这些修改可以被重新做一遍

需要支持事务操作

android事件机制中底层逻辑对事件地转发处理，Android的每一种事件在屏幕上产生后都会经由底层逻辑将其转化为一个NotifyArgs对象，NotifyArgs本身并无任何实现，只是定义了抽象类的方法体：

经过转化后，一次按键事件会被封装成一个NotifyKeyArgs对象：

这里的NotifyArgs实际上就是相当于命令遮抽象，而NotifyKeyArgs等子类相当于一个具体的命令者，那么谁来承当请求者角色呢？由Android事件机制中一个重要角色InterputDispathcher来承当，具体的事件命令者将事件转发给InputDispatcher，并由其来封装具体的实践操作，这里依然以按键为例，InputDispatcher中的执行函数为notifykey

事件对命令模式应用的这部分逻辑还算明朗除了上述的命令者和请求者角色外，接收者角色则由具体的硬件驱动承当，底层的机制讲解不在范畴内，除了事件这块程序外，Android在对应用程序包管理的部分也有对命令模式的应用体现，PackAgeManagerService也是Android系统Service之一，其主要功能在于实现对应用包的解析、管理、卸载等操作，在PackageMangerServive中，其对包的相关消息处理有内部类PackageHandler承当，其将需要处理的请求作为对象通过消息传递给相关的方法，而对于包的安装、移动以及包大小的测量则分别封装为HandlerParas的3个具体的子类InstallParams、MoveParams和MeasureParams、HandlerParams：

我们看看HandlerParams类

这个结合命令模式来看，HandlerParams也是一个抽象命令者，而对于其3个子类对应于各个功能不同的具体命令角色，

### android事件输入系统介绍：

整个android事件输入系统主要氛围，InputRead’er，其主要作用就是将输入水事件从硬件节点读取后转化为一个Event事件，并将该事件分发给InputDispatcher，InputDispatcher的职责就是接受来自InputReader的Event事件，并将该事件分发给InputDispatcher，InputDispathver的职责就是接收来自InputRead的Event事件，并像其名字所描述那样，将事件分发给合适的窗口并监控ANR的发生，然后则是创建INputReader和InputDispatcher的父类InputManager，尤其创建两者并提供Policy对事件进行预处理，最后则涉及系统Service以及面向用户的几个相关模块，如ActivityManger、WidnowManger和View等

在android4.4中，时间输入的service部分的是现在/framworks／base／service／input下，而在android5.0中该部分实现移到/frameworks/native/services/inputflinger/下虽然我们整个事件输入系统分为如上的四个部分但是，在android中负责管理实践出输入的·是·InputMangerService，其也是系统级的service之一，主要职责就是协调上面我们所述的四个部分，InpitMangerServie和其他系统级的Service一样，在 SystemServer中被初始化·

从上诉与源码中可以看到，在SystemServer中的startotherServices方法中，不但构造并启动了InputManagerService，而且启动InputManagerServie之将它只身注入到windowmanagerServie中进行关联，InpitMangerServie的构造主要是对native层进行初始化，但是是他的实现逻辑非常复杂

：  
这里主要是调用了nativeInit方法进行一系列的初始化操作，nativeInit方法的是现在native层，在其中构造NativeUnputManger对象：

在NativeInputManger逻辑中，首先会创建一个EvetnHub来监听所有的事件输入

之后就会相继初始化InputManager、InputDispatcher、InputReader、I n pu t R e a de rThread和InputDispatcherThread。其中InputReaderThread和InputDispacher是由InputManager管理两个重要的线程，其作用和具体的逻辑均与InputReader和InputDispatcher相关，这一系列的初始化操作完毕后，INputManagerService就会在SystemServer里调用其start方法开始运作，其中最为主要的是，是上面提到的两个线程InpiutReaderThread和InputDispatcherThread运作。inputReaderThread启动后就会进入死循环，每次循环都会调用loopOnce函数，第一次循环是会扫描/dev/input/设备节点目录，并通过ioctl函数获取设备 信息及判断其类型，当设备发生改变后悔则通过NativeInputMangaer的回调函数通知处于java层的InputManagerServive做出相应的处理，如上所述，InputReaderThread只负责从设备端读取事件信息，每一个事件信息在InputReaderThread里最终都会封装成一个NotifyArgs对象，通过NotifyArgs中的Notify方法调用到InputDispatcher相应的Notify接口，并由InputDispatcher来完成事件的分发。这份内容比较繁琐，限于篇幅这里不再描述，

## 观察者模式：

观察者模式是一个使用率非常高的模式，他最常用的地方就是GUI系统、订阅—发布系统。应为这个模式的一个重要作用就是解耦，将被观察者和观察者解耦，使得她们之间的依赖性更小，甚至做到毫无依赖，以GUI系统来说，应用的UI具有易变性，尤其是前期随着业务的改变或者产品的修改，应用界面也会经常变化，但是业务逻辑基本变化不大，此时，GUI系统需要一套机制来应对这种情况，使得UI层与具体的业务逻辑解耦，观察者模式此时就派上用场·

定义：

定义对象件一种一对多的依赖关系，使得每当对象状态改变，则所有依赖他的对象都会得到通知并被自定更新

### ListView源码分析

我们看看notifyDateSethanged，这个方法定义在BaseAdapter中，

public void notifyDataSetChanged(){  
mDataSetObservable.notifyChanged();

}

一看BaseAdapter代码就大体有了这么一个认识：BaseAdapter是一个观察者模式，那么BaseAdapter是如何运作的？这些观察者又是什么呢？

我们看看mDataSetObjservable.norifyChanged()

这个代码很简单就是在mDataSetObservable.notifyChanged()中便利所有的观察者，并调用他们的onChanged方法，从而告知观察者发生变化

那么这些观察者从哪来的：其实这些观察者就是listView通过setAdapter方法设置了Adapter产生的

从程序的角度来看，在设置了Adapter时会创建一个AdapterDataSetObserver，这就是上面所说的观察者，最后，将这个观察者注册到Adapter中，这样我们的被观察者、观察者都有了·，这回事可能你就有点不明包，AdapterDatasetObserver是什么，我们看看AdapterDatasetObserver。AdapterDataSetObserver定义在ListView的父类AbsLisetVidw中，

他继承自AbsListview的父类AdaterView的AdapterDataSetObserver：

到这里我们就知道，当ListView的数据发生变化时，调用Adapter的notifyDataSetChanged函数，这个函数又会dialingDataSetObservable的norifyChanged哈叔，这个函数会调用所有的观察者（AdapterDataSetObserver）的obChanged方法，在onChanged函数中又会调用ListView重新布局的函数使得Listview刷行界面。

最后我们在整理一下：AdapterView中有一个内部类AdapterDataSetObserver，在ListView中设置Adapter时会构建一个AdapterDataObserver，并且注册到Adapter中，这就是一个观察者，而Adapter中包含一个数据集可观察者DataSetObservable，在数据数量发生变化时，开发者手动调用Adapter.notifyDataSetChaned，而notifyDataSetchanged实际上会调用DataSetObservable的NotifyChanged函数，该函数会遍历所有的观察饿onChanged函数，在AdapterDataSetObserver的onChanged函数中会获取Adapter中数据集的新数量，然后调用LisetVIew的RequestLayot方法进行重新进行布局，更行用户界面。

BroadCaseReceive他作为应用内、进程间的一种重要的通信手段，能够将某个消息通过广播的形式传递给他注册的对应广播接收器的对象，接受对象需要通过Context的registerReceiver函数组册到AMS中，当通过SendBroadcast发送广播时，所有注册了对应的InttentFilter的BroadcaseReveiver对象就会接收到这个消息，BroadcaseReveiver的oReveive方法就会被调用

我们在mainActivity的onResume中注册了一个只接受Action的info.update的接收器，应用内的其他发布一个Action为info.udate的广播时，就会触发updatereveicer的onreceive函数，

我们发现registerreceiver函数斌不是在Activity中实现的，因此，我们把目标转向Activity的父类Contextwrapper，registerReceiver函数如下：

这个类的成员变量mBase，是一个ContextImp类的实例·，继续转移到Comimple的registerReceive函数，

注册广播接收器的函数掉调用最终进入到Contextimpl的registerReceiverInternal这个函数，这里的成员变量mPackageInfo是以恶loaderApk实例，它是用来负责处理广播接收。参数broadcastPermission和scheduler都为null，而参数context是上面的函数通过调用函数getOuterContext得到的，这里他指向MainActicvity

由于条件mPackageInfo！=null 和context！==null都成立，而且条件·sfheduler==null也成立，于是就调用mMainThread.getHandler来获取一个Handler，这个Handler是用来分发ActivityMangerService发送过的广播，这里的成员变量mMainThread是一个ActivityThread实例，我们先看看ActivityThread.getHandler函数的实现：

这个函数获取Handler之后，就可以分发消息给应用程序处理。

再回到上一步的ConteztImp.registerReceiverInternal函数中，他通过mPackaageInfo.getReceiverDispatcher函数获取一个internaReceicer接口对象rd，这是一个Binder对象，接下来会把它传递给ActivityManagerService，ActivityManagerService在收到相应的广播时，就是通过·这个Binder对象来通知MainActivity来接受的

我们看看mPackageInfo。GetRceiverDispatcher函数的实现：  
在LoadedApk类的getReceiverDispatcher函数中，首先看一下参数，r是不是已经有相应的ReceiverDispatcher存在了，如果有，就直接返回，否者就创建一个ReceiverDispatcher，并且以r为key保存在一个HashMAp中，而这个HashMap又被存储在mReceivers这个map，mRecieVers以Context为key，map为value。即这里的MainActivity为key保存在LaoderApk的成员变量mReceivers中，这样只要给定一个Activity和BroadecaseReceiver，就可以查看LaoderAPk里面是否已经存在相应的广播接收发布器ReceiverDispatcher

新创建广播接收者发布器ReveiverDispatcher时，会在构造函数里创建一个Binder对象，实现了IIntentReceiver接口，可以通过ReceiverDispatcher.getIntentReceiver函数来获得，获得后就会把它传给ActivityManagerZService，以便接受广播，在ReceiverDispatcher类的构函数中，还会把传进来Handler类型的参数activityThread保存下来，以便后面再发布广播的时候使用。

现在，再回到ContextImpl.registerCeiverInternal函数，在获得IIntReceiver类型的Binder对象后，就开始把他注册到ActivityManagerService中：

这个函数通过Binder驱动程序就进入到ActivityManagerServive中的registerReceiver函数中了

## 备忘录模式

备忘录模式是一种行为模式，该模式用于保存对象当前的状态，并且在之后可以再次恢复到此状态，。备忘录模式实现的方式需要保证被保存的对象状态不能被对象外部访问，目的是为了保护好被保存的这些对象状态的完整性以及内部实现不想外部暴露

在android开发中，状态模式应用时Activity中的保存状态，也就是里面的onSaveInstanceState和onRestoreInstanceState。当Activity不是正常方式退出，切Activity在随后的时间内被系统杀死之前会调用这两个方法让开发者有机会存储Activity相关信息。并且在下次放回Activty时恢复这些数据。通过这两个函数，开发人员能够在某些特殊的场景下存储与界面相关的信息，提升用户的体验，例如，用户在写了一大段表白短信后，此时一个电话打进来，用户的断行输入界面被推到后台，如果在打电话的过程中短信应用被系统杀死了，那么在用户在此进入到短信应用时..我们直接看onSaceInstanceState和onREstoreInstanceState都是Activity中的函数，

protected void onSavaInstanveState(Bundler oustate){  
outstate.putBunde(WIND\_  
)

这个函数主要分为3个部分：

1 存储窗口的视图树状态

2 存储Fragment的状态

3 调用Activity的ActivityLifecyclerCallbacks和onSaveInstaneState函数进行状态存储

我们看第一步，在这一步Windows对象中的视图树中的各个View状态存储到Bundler中，这样一来，当用户重新进入到Activty时，用户UI的结构，状态才会被重新恢复，以此来保证用户界面的一致性，Windows类的具体实现类时PhoneWindow 我们看看PhoneWindows中的saveHirearchyState函数的实现：

在saveHierachyzState函数中，主要是存储了与当前UI、ActionBar相关的View状态，这里用mZContentparent来分析，这个mCodntextParent就是我们通过Activity的setContentView函数设置的内容视图。它是整个内容试图的根节点，存储他层级结构中的View状态也就是存储了用户界面的状态，mContentParent是一个ViewGroup对象，但是，saveHierachyState并不是在VewiGroup中，而是在ViewGroup的父类View中：

在View类中SvaeHierarchyState函数调用了dispatchSaveInstanceState函数来存储自身以及子视图的状态，ViewGroup的dispatchSaveInstancezstate函数：  
 在ViewGroup的dispachSaveInstanceState函数中调用super.dispatchInstanceState函数储存自身的状态，然后便利所有的子视图，并调用子视图的dispatchSvaInstave函数存储她们的状态，如果这个子视图时ViewGroup类型，那么则再次执行这个过程，如果这个子视图是View类型，那调用的是Veiw的dispatchSaveInstnce函数，在View类dispatchSaveInstanceState函数的就是在View没有设置id时，这个View的状态不会被存储到Bundler中这个id就是我们平时在xml中通过android;id属性来设置标识View的唯一性的

这里的i在一个Activity的布局中必须是唯一的，应为这些View的状态是通过SpareArray来存储的，这相当于一个map，他的key时View的id，value为View的状态，当在一个视图树中含有两个相同的id时，那么必然会出现状态覆盖

这些被存储的状态通过onSaveInstanceState函数得到，但在View类中我们看到返回的是一个空的状态，这就意味着，当我们需要存储View的状态时，需要复写onSaveInStanceState方法，将要存储的数据放到Parcelable对象中，并且将他返回：我们看看TextView的实现onSaveInstancezstate

调用了View的onSaveInstanceState函数之后就得到了View要存储的数据，此时执行到 fdsafds

if(state!=null)

{

container.put(Mid,state);

}

经过一层一层地遍历之后，整个视图树的状态就被存储起来了

存储完Windows的视图树的状态信息之后，便会执行存储Fragemnt中的状态信息，回退栈等，这个存储Fragment状态信息也是调用它的onSafeInstaneState方法，存储Fragment的View视图树状态，最后就是调用用户设置的ActivityLifecycyleCallback的zonSaveInstanceState方法，让用户能够在做一些而外的处理，至此，整个存储过程就完成了

我们上文说到，调用onSaveInstanceState是在Activty被销毁之前，更确切的说法是在调用Activity的onstop之前，Activity的onStop方法在ActivityThread的performStopActivity函数中：  
 在performstop函数中通过token从mActivities中获取了一个ActivityClineRecord对象，这个对象就存储了Activity的信息，我们的状态信息就包含在其中，获取到这个ActivityClientRecord这个对象就存储了Activity信息，我们的状态信息就包含在其中，获取到这个ActivityClientRecord之后，调用了performStopActivityInner函数这个函数分为四部分：

判断是否需要存储Activity状态：

如果需要存储Activity状态，调用onSaveInstane函数：  
将状态信息存储到ActiityClienetRecord对象state字段中

也就是说在执行onstop函数之前，系统会根据情况来选着是否存储Activity状态，并且将这些状态存储到mActivities中，这个mActivities维护了一个Activity信息表，当Activity重新启动时，会从mActivities查询对应的ActivityClinetRecord，如果这个记录对象中含有状态信息，那么则调用Activity的onRestoreInstanceState函数：  
  
onsaveInstancezstate调用时机：

在上下文中，onSaveInstanceState并不是在每次Actiity推出之前都会调用，只是在某些情况才会调用：  
1 当用户按下home键时 这