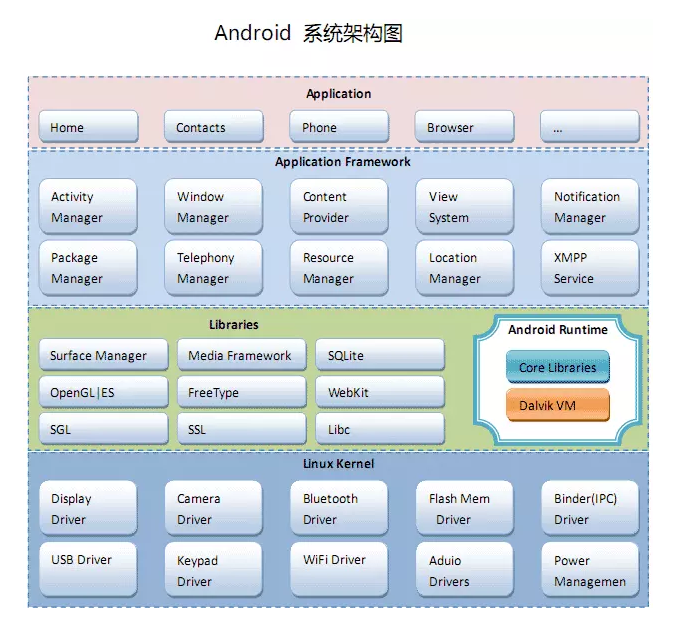
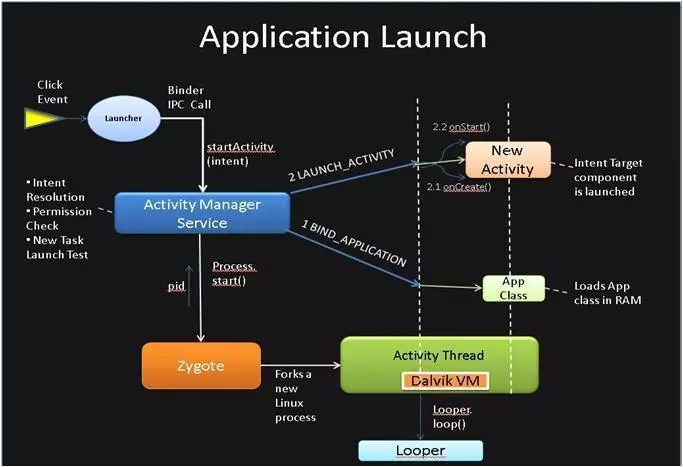
# Android高级进阶



**Android app启动流程**



系统上电，从固定地方加载一段代码到RAM中进行硬件简单的初始化，随后加载kernel，操作系统核心程序。

将kernel加载成功后，从其入口点继续初始化系统，硬件就绪后，管理内存，网络以及进程等，完成后，启动第一个进程

启动第一个init进程，挂载目录，创建一些守护进程，此后所有的进程都是由init进程fork出来，每一个app都是一个单独的dalvik虚拟机和一个单独的进程。在Android中，所有的进程都是由 zygote进程fork而来。

Zygote会通过ZygoteInit.main()创建Android虚拟机，配置JNI，加载动态库等

，然后开启一些system server（系统服务），比如AMS（ActivityManagerService），PMS（PackageManagerService），WMS（WindowMangerService），

AMS使用startProcessLocked方法创建新的进程来开启新的APP,在Process.start方法传入ActivityThread，加载这个类，通过与Zogote通信，开启ActivityThread，在ActivityThread中，启动一个Looper.loop()死循环维持app开启。

SystemServer开启所有Java服务后，调用AMS中的SystemReady方法，开启Launcher（startHomeActivityLocked）

Zygote和AMS通过socket通信，app和AMS通过Binder进行IPC通信，ActivityThread通过ApplicationThread和AMS通信

开启新的Activity最终都调用startActivityForResult方法，最后调用

Instrumentation.execStartActivity方法

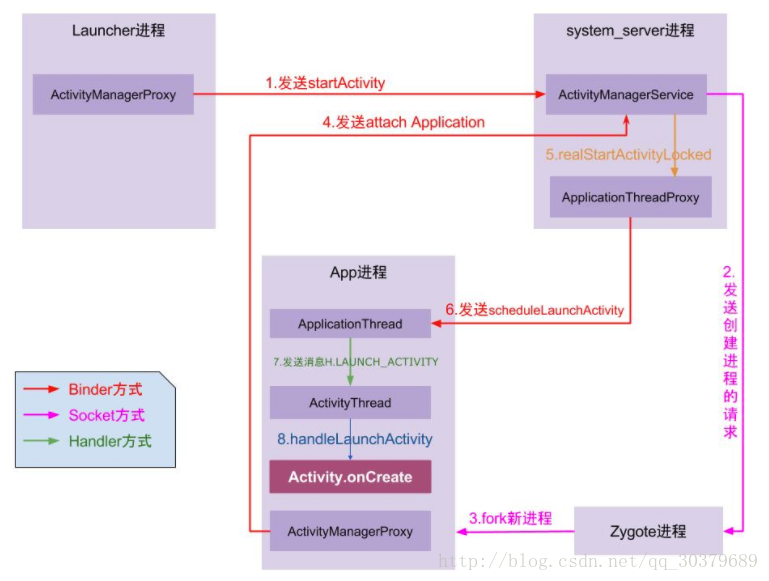
AMS负责系统中所有Activity的生命周期，首先通知launcher进入pause状态，在进入pause状态后，启动新的进程，创建ActivityThread对象，执行其中的main方法。使用startActivity方法启动app的第一个Activity。

AMS中ActivityStack进行管理Activity的状态，ActivityStack中的realStartActivityLocked

方法创建Activity。

ActivityThread通过Handler执行handleLaunchActivity方法，调用两个方法，performLaunchActivity调用Activity的生命周期方法，handleResumeActivity执行Activity中的onResume方法。

APP启动模型



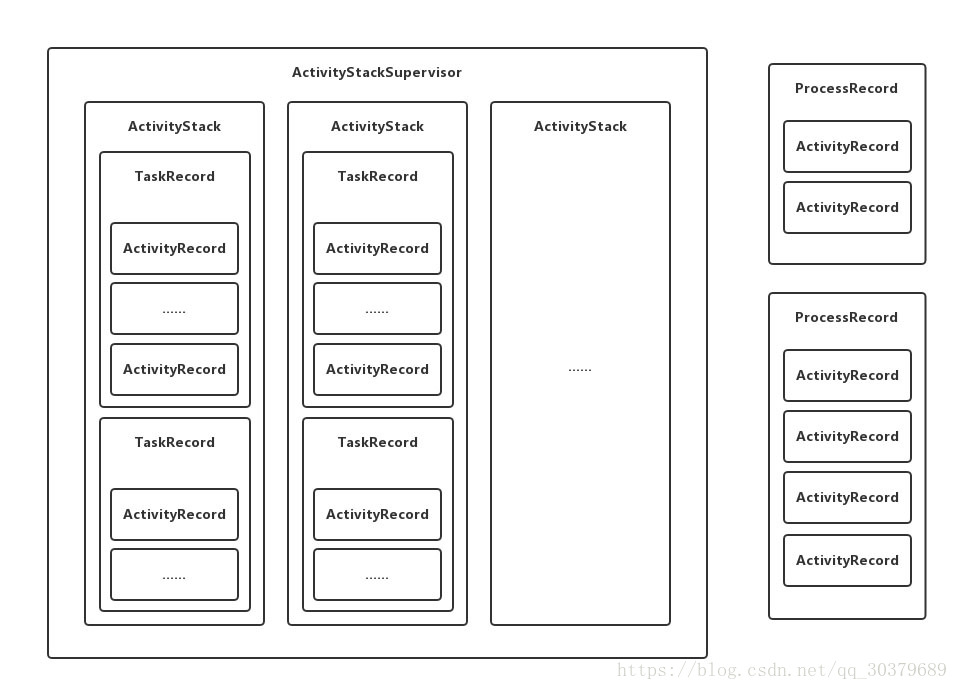
ActivityStackSupervisor，负责管理多个ActivityStack

ActivityStack，栈式管理结构，负责管理栈内TaskRecord，栈顶的TaskRecord表示当前可见的任务

TaskRecord，栈式管理结构，负责管理栈内ActivityRecord，栈顶的ActivityRecord表示当前可见界面

ActivityRecord，负责维护对应的Activity组件的运行状态以及信息

ProcessRecord，记录着属于一个进程的所有ActivityRecord,ActivityRecord可能跟运行在TaskRecord中的ActivityRecord属于同一个



启动过程总结：

Activity#startActivity()=>Activity#startActivityForResult()=>Instrumentation#execStartActvity()=>ActivityManagerNative.getDefault().startActivity()=>ActivityManagerService#startActivity()=>ActivityManagerService#startActivityAsUser()=>ActivityStackSupervisor#startActivityMayWait()=>ActivityStackSupervisor#startActivityLocked()=>ActivityStackSupervisor#startAcitvityUncheckedLocked()=>ActivityStack#startActivityLocked()=>ActivityStackSupervisor#resumeTopActivitiesLocked()=>ActivityStack#resumeTopActivitiesLocked()

=>ActivityStack#resumeTopActivitiesInnerLocked()=>ActivityStackSupervisor#startSpecificActivityLocked()=>ActivityStackSupervisor#realStartActivityLocked()=>ApplicationThreadProxy#scheduleLauncherActivity()=>ApplicationThread#scheduleLaunchActivity()=>

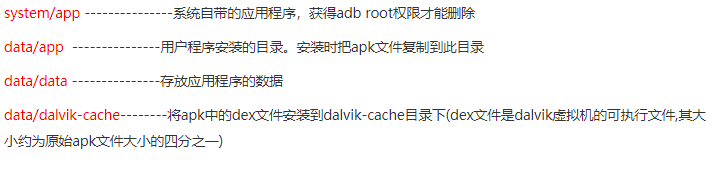
ActivityThread#handleLaunchActivity()=>ActivityThread#performLaunchActivity()=>

Instrumentation#newActivity()=>attach方法将context，application和Activity绑定，创建PhoneWindow

**Android App安装过程源码分析**

安装方式有四种：

1. 系统应用安装—开机完成，没有安装界面
2. 网络下载应用安装，通过market安装，没有界面
3. ADB工具安装，没有安装界面
4. 第三方应用安装，通过SD卡里的APK文件安装，使用PackageInstaller.apk处理安装和卸载的页面。



安装过程是将APK文件复制到data/app目录下，解压并扫描，将dex文件保存到dalvik-

cache目录，并在data/data目录创建对应的应用数据目录

Android启动时候会开启PackageManagerService进程(同AMS一样，是在System Server中启动的)，用来管理应用安装包。安装应用程序的过程，其实是解析应用程序配置文件AndroidManifest.xml的过程，获取其中的信息，通过AMS来调控应用程序

在PMS中，会先创建本身的实例，并将其添加到Service Manager中。

PMS中使用scanDirLI方法扫描五个文件夹中的APK文件

/system/framework，框架jar包

        /system/app，自带应用

        /vendor/app，制造商应用包

        /data/app，用户安装应用

        /data/app-private，受保护的apk文件

如果扫描到Apk文件，就用scanPackageLI方法解析和安装，installPackageLI

使用PackageParser中的parsePackage方法来解析APK文件

解析APK中的AndroidManifest文件后，再使用另一个parsePackage方法解析，解析出每一个标签，例如activity,service,receiver和provider，并将其存入Package Manager Service中（ActivityInfo列表），此时只是在其中注册好了，如果需要在桌面显示，则需要Launcher将其在PMS中取出展示出来。

应用市场应用安装，调用Packagemanager中的installPackage方法

安装包安装，调用PackageInstallerActivity（ROM自定义安装页面在此继承），startInstallConfirm方法，执行installPackageLI

**Android的消息机制Handler**

主线程handler和子线程handler的区别在于，主线程不需要手动创建Looper

1. Looper死循环为什么不会导致应用卡死，会消耗大量资源吗？

主线程在创建的时候，初始化Looper，故可以默认可以使用handler，子线程创建looper应当在用完后，quit方法关闭消息循环。

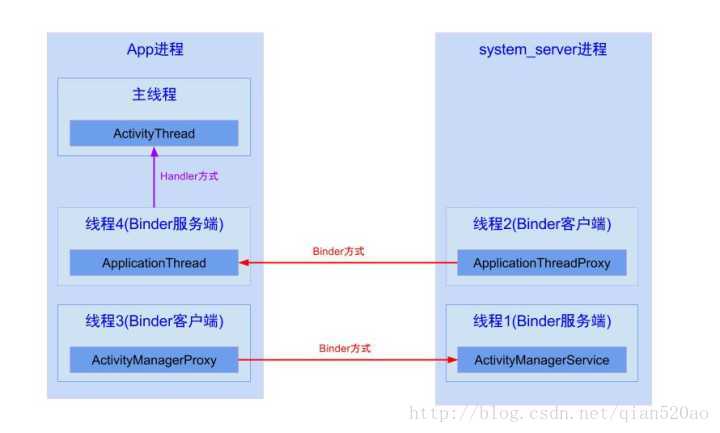
死循环保证app打开不会退出，创建新线程来执行其他操作。

死循环在大多时候是休眠状态，并不会消耗大量资源。

1. 主线程的消息循环机制是什么，如何处理其他事务

Main方法中，创建activityThread和binder线程，接受AMS发来的事件，binder通过handler将message发送给主线程。

ActivityThread通过ApplicationThread和AMS进行进程间通讯，AMS以进程间通信的方式完成Activity Thread的请求后会回调ApplicationThread中的Binder方法，然后ApplicationThread会向H（继承Handler）发送消息，H接收到消息后执行ActivityThread中的生命周期相关方法。



1. Activity Thread的动力是什么

从Linux角度来看，进程和线程除了是否共享资源外，并没有本质的区别。

1. Handler是如何切换线程，发送Message的？

线程间是共享资源的，Handler处理不同线程的问题，只需要注意异步情况。Handler绑定Looper创建的线程，并在其绑定的线程中处理消息。

线程中默认是没有Looper的，如果需要使用Handler，则必须为线程创建Looper，而主线程，即ActivityThread可以直接使用Handler是因为在创建的时候已经初始化了Looper。

系统不允许子线程访问UI，多线程访问UI会导致控件处于不可预期的状态，如果加锁，则会导致UI访问逻辑复杂，降低效率。

1. 子线程有哪些更新UI的方法

如果在onCreate中创建子线程，并立刻更新UI，则不会抛出非UI线程错误。因为，创建viewRootImpl时会检测该线程是否是创建UI的线程，而创建ViewRootImpl是在onResume方法中，所以在onCreate中更新UI不会抛出异常，即可理解为，onCreate方法并没有绘制UI，所以并不检测是否是在UI线程中更新UI。

更新UI方法，主线程定义Handler，子线程通过handler发送消息，主线程Handler的handlerMessage更新UI。

使用Activity的runOnUiThread方法

子线程中创建Handler，传入getMainLooper，获取UI线程的Looper，开启的线程为UI线程

View.post(Runnable r)

1. 子线程中Toast，showDialog的方法

Toast中使用Handler处理消息机制，如果使用在子线程中，子线程中没有Looper，所以需要在子线程中创建Looper。（但会使子线程无法结束，导致内存泄露）

1. 如何处理Handler使用不当导致的内存泄露

有延时消息，在Activity销毁的时候一处Messages；匿名内部类导致的泄露，改成匿名静态内部类，并且对上下文或者Activity使用弱引用。

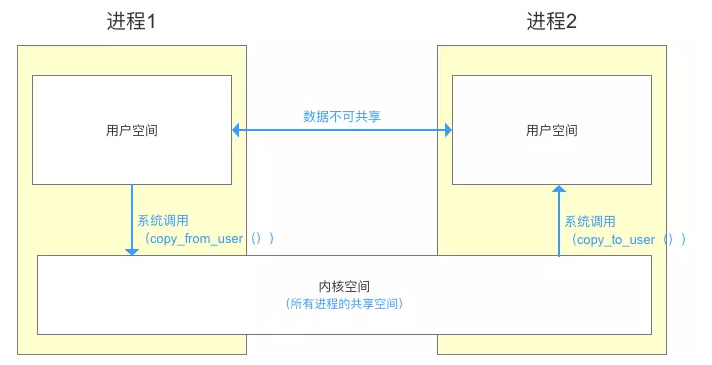
**Looper机制源码心得。**

1. Looper.prepare方法和prepareMainLooper方法，前者可以退出循环，后者不可。
2. Looper的prepare方法创建一个MessageQueue，MessageQueue两个重要方法，一个enqueueMessage，将Message加入队列，一个next，将Message取出队列。
3. Looper中生成一个ThreadLocal对象，prepare方法创建Looper并使用Thread Local的set方法将Looper和当前线程绑定，在获取Looper的时候，使用ThreadLocal的get方法进行获取
4. Looper的loop方法，获取Looper后，获取到其消息队列，有一个无限的for循环，不断取出队列中的Message，通过dispatchMessage方法将消息发送给Handler。



**IPC(进程间通信)方法Binder**





内核空间所有进程共享，但用户彼此不能共享，所以Binder跨进程通信是通过内核空间可共享来实现，利用ioctl等方法。

Binder机制采用C/S架构，主要包括Client，Server，ServiceManager和Binder驱动。



此处的ServiceManager是Native层的ServiceManager，是整个Binder通信的管理者，是AndroidIPC机制Binder的守护进程。

启动ServiceManager后，Client和Server需要先获取ServiceManager接口。

**注册服务**Server进程注册Service到ServiceManager，Server是客户端，SM是服务端

**获取服务**Client使用某个Service，需要向ServiceManager中获取相应的Service，Client是客户端，SM是服务端

**使用服务**Client根据得到的Service信息建立与Service所在的Server进程通信的通路，然后就可以直接与service交互，Client是客户端，Server是服务端

Client，Server和ServiceManager之间的交互实际上均是通过Binder驱动来实现



Binder类代表Server端，BinderProxy类代表Client端

基于Client-Server方式的通信方式，由Server管理，应用程序只需要作为Client与这些Server建立连接，使用这些服务。

各种IPC方式对比：

Socket传输效率低，开销大，主要用于跨网络的进程通信和本机进程间低速通信。

消息队列和管道采用存储-转发方式，数据先从发送方缓存区拷贝到内核的缓存区，然后再拷贝到接收方缓存区

共享内存无需拷贝，直接共享两个进程的内存，但是控制复杂，难以使用

C/S必须实现两点：一是Server必须有确定的访问接入点或地址接收Client访问，且Client可以通过某途径获知Server地址。二是Command-Reply协议来传输数据。

Binder是位于Server中的对象，其提供了一整套方法实现服务的请求，而Client中的Binder作为Server中Binder的代理。

Binder驱动，一种虚拟设备驱动，连接Serivice，Client和ServiceManager的桥梁，使用内存映射的原理。

1. 创建一块接收缓存区
2. 实现地址映射关系，（mmap）将内核缓存区和接收进程用户空间地址同时映射到同一个共享接收缓存区中
3. 发送进程通过调用copy\_from\_user方法将数据发送到虚拟内存区域，拷贝一次。
4. 由于内核缓存区和接收进程的空间地址存在映射，所以接收进程也直接收到了发送进程的数据，即实现了跨进程通信。

一个进程Binder线程数默认最大是16，超过的请求会被阻塞等待

使用方法：

1. 继承IInterface，写一个管理接口，声明需要提供什么功能
2. 建立Stub抽象类，继承Binder，实现管理接口，重写asInterface方法，返回远程代理的对象proxy。重写onTransact方法，调用管理类中的方法
3. 远程服务中，新建Stub，并实现管理类中的方法
4. 新建远程proxy类，实现管理类中的方法，使用Binder调用远程服务中的方法
5. 在Client中，使用管理类，调用方法，实现功能。

实名Binder：

Server创建Binder实体，并发送给ServiceManager注册并取名管理，Client可以根据名字获得Binder的引用。Server和ServiceManager通信同样需要Binder，前者是Client，后者是Server，ServiceManager预先创建一个Binder为其他线程提供Server，而此时，Server实际是一个Client。

匿名Binder：

Server通过实名Binder建立的与Client连接以后，就可以通过该通道将创建的Binder传输，由于没有在ServiceManager中注册名字，所以是匿名Binder，用来建立私密通道。

Binder协议：

基本格式是命令+数据，使用ioctl(fd,cmd,arg)函数交互，cmd是命令，arg为数据。

基本命令有，读写命令，告知接收方线程池最大线程数，当前进程注册为ServiceManager，通知Binder当前线程退出，获取Binder版本号。

**Android四大引用和源码分析**

1. 强引用，最常见的引用，如果是强引用，垃圾回收器即使抛出OOM异常也绝对不会回收。其可以直接访问目标对象，但可能会导致内存泄露。
2. 软引用，当虚拟机内存不足时，将回收，需要获取对象时，调用get方法，从网络获取图片，显示的同时，通过软引用缓存起来，以备下次使用
3. 弱引用，随时可能被虚拟机回收，同样通过get方法获取对象。

使用案例，Handler中，handler对消息队列进行操作的时候，消息队列会持有handler的引用，如果Activity被结束了，消息队列仍持有handler的对象，此时需要将handler设为静态类型，静态类型持有当前类的弱引用。

1. 虚引用，持有虚引用的对象，和没有引用几乎一样，随时被回收。虚引用必须和引用队列一起使用，目的是跟踪垃圾回收过程。



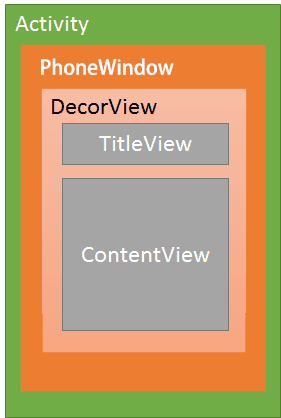
**Android四大组件分析（Activity，ContentProvider，Service，BroadcastReceiver）**

Activity：

ActivityThread通过ApplicationThread和AMS进行进程间通讯，AMS以进程间通信的方式完成Activity Thread的请求后会回调ApplicationThread中的Binder方法，然后ApplicationThread会向H（继承Handler）发送消息，H接收到消息后执行ActivityThread中的生命周期相关方法。

启动Activity，handleLaunchActivity中创建Activity对象，使用performLaunchActivity（创建context，Application，并绑定到Activity Thread），ActivityThread中方法，均通过Instrumentation的方法进行调用，创建Activity等

Activity中视图加载，setContentView方法，调用PhoneWindow的setContentView



Window是一个抽象基类，提供一系列窗口的方法，其实现类是PhoneWindow，其中DecorView是继承自FrameLayout的View，用来加载布局文件，由PhoneWindow负责添加。

Activity中的setContentView方法调用的是PhoneWindow中的setContentView方法。

如果没有放置父容器，需要调用installDecor方法，然后将xml文件通过LayoutInflater转化成View树，并添加至父容器。

在installDecor方法中，会根据用户设定去加载不同的布局，比如Window.FEATURE\_NO\_TITLE，所以在Activity中需要在setContentView前调用requestFeature。

在DecorView中，xml文件包含通知栏，标题栏，内容显示三块。

ContentProvider：

URI



Schema：主机名，Android固定为content://

Authority：用于唯一标识一个ContentProvider

Path：ContentProvider中数据表的表名

Id：数据表中数据的标识

使用：

1. 继承SQLiteOpenHelper,创建数据库
2. 继承ContentProvider实现oncreate，query，insert，update，delete，getType等方法。onCreate运行在主线程，不要耗时操作。
3. 在配置文件，AndroidManifest里声明provider及相关访问权限
4. 通过ContentResolver根据URI进行增删改查

ContentProvider对底层数据库进行封装，底层使用了SQLite，即使替换了，也不影响使用。在contentprovider中用户可以根据需求将自己app的数据暴露出去，而不用担心其他开放数据库的安全问题

ContentResolver通过不同的URI来区分不同的ContentProvider

Service：

本地服务和远程服务，客户端和服务端在同一个进程中，是本地服务，不在同一个进程则是远程服务。

startService启动的入口在ContextImpl（继承自ContextWrapper，继承自Context），调用了AMS中的startService方法，调用ActiveServices（AMS中管理Service的类，类似Instrumentation？）中的startServiceLocked方法，检查是否又ServiceRecord缓存，以及是否是前台服务，最后调用startServiceInnerLocked方法，调用bringUpServiceLocked方法，如果Service被启动，调用onStartCommand方法，未启动，调用realStartServiceLocked启动Service（通过Binder方式，通知ActivityThread创建Service并启动ScheduleCreateService，Handler通知）

bindService启动入口同上，启动方法类似，在通知ActivityThread的时候会搭建与Activity的连接（在Handler处理消息的时候，调用AMS中的publishServiceLocked方法）

BroadcastReceiver：

广播采用Binder机制，因为在Binder中，需要事先设置好server端和Client端，而广播可以不用知道接收者的存在，降低了耦合度，提高了扩展性。

分为静态注册和动态注册，静态在配置文件中声明，不受其他组件影响，即使关闭，也能接收广播，Android8以后大多数静态广播失效。动态广播在代码中注册，注意在退出时销毁广播

1. 新建类，继承BroadcastReceiver，重写onReceive方法
2. 静态文件中注册监听器
3. Activity中发送广播

注册是在ContextImpl中完成，调用registerReceiverInternal方法，通过LoadedApk获取IintentReceiver对象，生成ReceiverDispatcher实例，生成IintentReceiver实例并返回。然后通过AMS调用registerReceiver方法

静态注册，则是在系统启动的时候，使用PMS去解析，当AMS调用查询时候，PMS将信息返回给AMS

广播发送和接受，仍然是在ContextImpl中，直接调用AMS的broadcastIntent方法＝＞broadcastIntentLocked方法，在广播解析器中查询相关，然后创建broadcastQueue，，将信息逐个发送，获取时，会从中找到对应的IintentReceiver实例，找到对应的BroadcastReceiver实例，调用其OnReceive方法进行广播。

**Android Context详解**

Context是一个很重要的基类，提供了与APP相关非常重要的接口（获取APP基本信息，四大组件接口，基本信息创建接口，文件操作接口等）。ContextWrapper继承Cotext，提供一个mBase，构造器或者attach传入mBase，其实现的所有Context方法，都是调用mBase的context实例中的方法。

ContextThemeWrapper，实现了诸多有关Theme的方法，getTheme，getResource等

ContextImpl是Context中方法的具体实现类，提供诸多接口：

创建ActivityContext，创建AppContext，创建SystemContext，创建SystemUiContext，创建PackageContext。

DecorContext持有PhoneWindow和WindowManager，属于主题方面的方法

Application继承ContextWrapper，Activity继承ContextThemeWrapper。

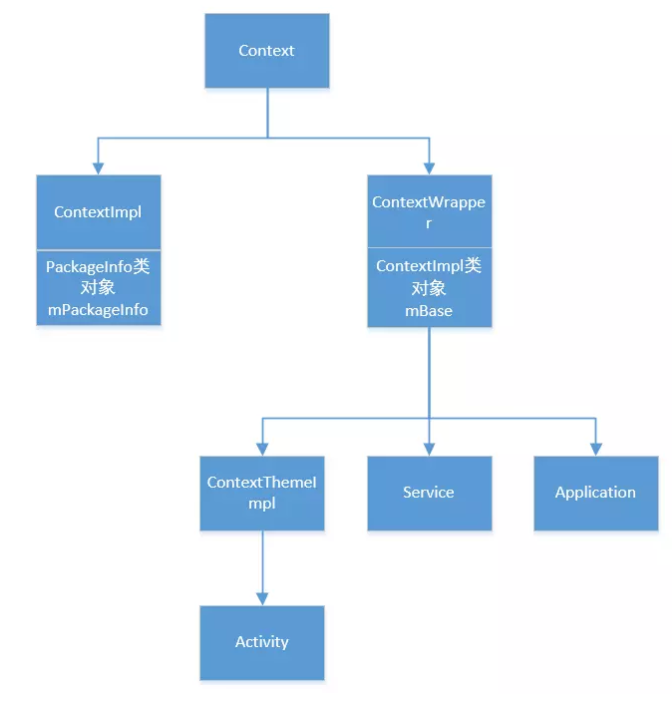


ActivityContext的创建在ActivityThread创建Activity时候，通过ContextImpl的create ActivityContext来实现

ApplicationContext创建在ActivtyThread中，通过Instrumentation的newApplication创建Application，通过LoadedApk中的makeApplication方法创建其Context。

DecorContext，算是View的Context，是Application的实例实例

一个APP总共的Context数为Service个数，Activity个数，Application个数



**Intent**

Intent连接各个组件的桥梁，通信的载体，负责数据传递，例如启动Activity，service，注册，注销，发送广播。

显示Intent，明确指定要启动的组件；隐式Intent，通过IntentFilter指定要启动的组件，Android5以后，不能隐式启动service。

PendingIntent用于延时启动

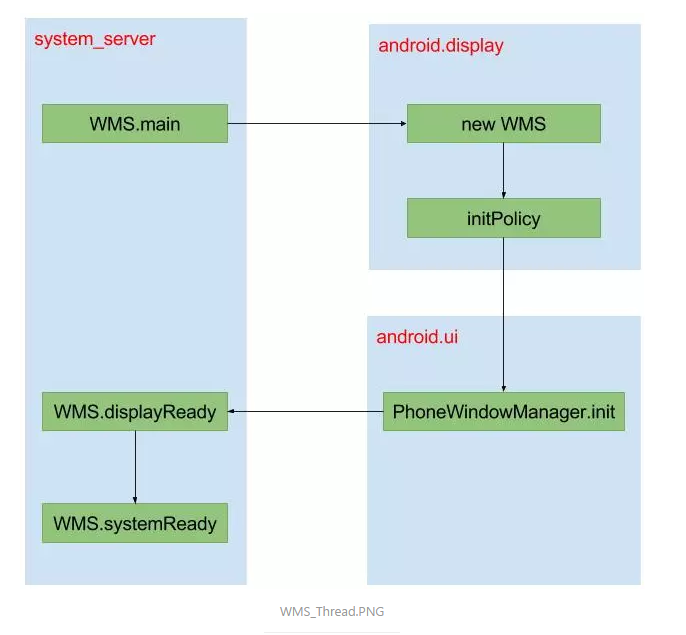
Intent的setFlags方法参数详解：

原理：Intent创建后，将所携带的信息等传入StartActivity中，执行开启新的Activity方法，在AMS中，会调用PMS中的resolveIntent方法查找PMS中注册的app信息，并启动那个Activity。

如果Intent指定了Component对象，直接通过其找到ActivityInfo列表。如果没有，即隐式调用，则根据包名，如果包名为空，则进行模糊匹配，根据Action和category等。

**WMS（WindowManagerService）源码**

在SystemServer中的startOtherServices中创建WMS（调用WMS中main方法，并将其放入ServiceManager中）。创建WMS是DisplayThread（系统前台线程）通过WMS的构造方法创建WMS对象



在启动Activity的时候，startActivityLocked主要完成以下工作：

1. 将Activity所在的Task移动到栈顶
2. 创建WindowToken并加入WindowToken列表中
3. 设置Activity切换动画的相关参数
4. 开启startingWindow

在Activity执行onResume方法时，会在调用handleResumeActivity时，在添加decorView之前，设为不可见，然后调用WindowManager的addView方法（实际实现是在WindowManagerImpl中），添加decorView到Window。WindowManagerImpl中addView调用WindowManagerGlobal中的addView，创建viewRootImpl对象，将DécorView，viewRootImpl，LayoutParams添加对应的list中，最后调用viewRootImpl的setView方法，执行DecorView的绘制过程。

**View和ViewGroup**

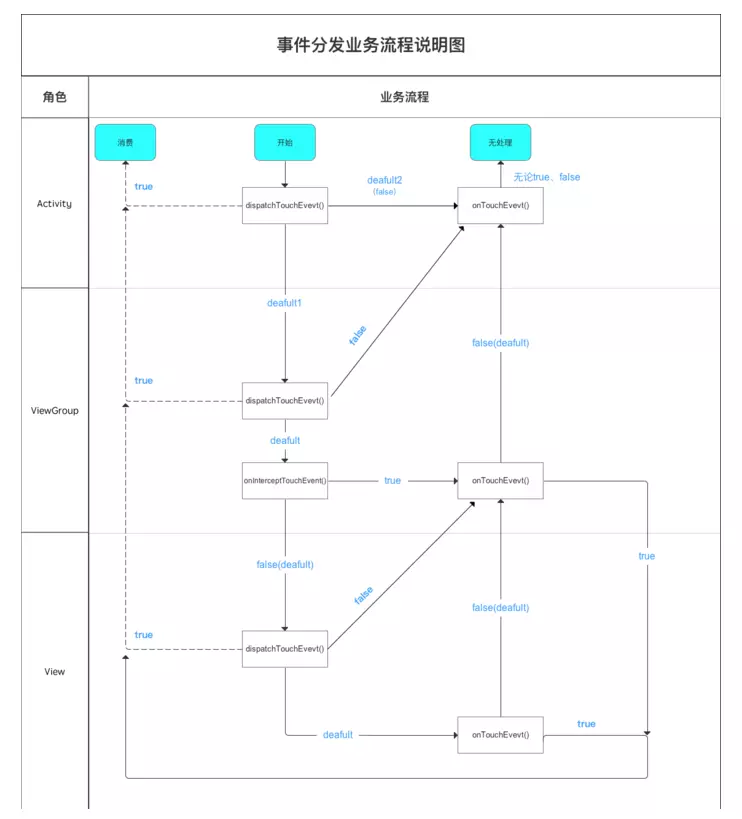
view事件分发机制：

用户点击屏幕，首先会调用Activity中的dispatchTouchEvent方法，首先向下分发，调用window的superDispatchTouchEvent，调用顶层DecorView的dispatchTouchEvent，再传递到viewGroup中的dispatchTouchEvent。

再viewGroup中，会首先调用onInterceptTouchEvent方法，检测是否拦截，如果不拦截，则循环其中的每一个view，判断其是否包含点击区域，然后调用子View的dispatchTouchEvent方法，如果消费，则返回true，最终一层层返回到Activity的dispatchTouchEvent，返回True，即表示被消费。如果被拦截，或者是子类view没有消费，返回false，那就调用viewGroup自己的onTouchEvent进行消费。

子类view的dispatchTouchEvent方法会调用View.onTouch方法，如果事件被消费，则返回true，如果没有被消费，调用onTouchEvent去消费。

Activity中，如果dispatchTouchEvent中向下传递过程中没有被消费，则调用自己的onTouchEvent方法进行消费。



在PhoneWindow中，setContentView方法中解析XML文件，并将其通过viewGroup的addview添加到Decorview中的contentParent中。实际调用viewRootImpl中的setview，

findViewById通过控件ID获取实例：

view中的findviewbyid调用findviewTraversal方法，如果id是本view的Id，则直接返回这个view，该方法被view group重写，因为一般是viewGroup使用这个方法。

findviewTraversal方法，循环查找子view，并根据findviewbyid找到其view实例。

View绘制三大方法onMeasure,onLayout,onDraw

MeasureSpec，View静态内部类，提供获取测量模式和测量尺寸两个方法，测量模式分为EXACTLY精准模式，表示match\_parent和具体值，AT\_MOST最大模式表示wrap\_content。

onMeasure方法中，通过setMeasureDemension来设置view的宽高，其参数getDefaultSize返回的是测量的宽高，根据match\_parent和wrap\_content设置宽高。

在viewGroup中，需要测量子View，调用measureChildren方法，遍历子view，并分别测量。

在Activity中获取view的宽高：

1. onWindowFocusChanged方法
2. view.post方法，将runable放入消息队列尾部
3. viewTreeObserver，回调onGlobalLayout回调接口。

View的layout：viewGroup用来确定子view的位置，调用子类view的onlayout方法，确定子元素的位置。Layout方法执行两个方法，setFrame确定自身位置，onlayout确定子元素位置。onLayout方法没有具体实现，需要根据子元素的具体情况去实现

View的draw：绘制内容到屏幕，绘制背景绘制自身元素onDraw，绘制子元素dispatchDraw，绘制装饰onDrawForeground

每一个Window都对应着一个View和ViewRootImpl，Window通过ViewRootImpl来和view建立联系，Window管理所有的view，触摸事件，均是由Window传递给DecorView。在创建Activity的过程中，在Activity的attach方法中，创建Window，使用WindowManagerImpl中的addview将view添加到Window，才能在Window的管理下，接收用户各种输入。

在WMI中，将addView方法交给WindowManagerGlobal处理，并创建ViewRootImpl，调用起的setview方法，调用requestLayout调用performTraversals绘制view，再将Window通过addToDisplay添加到WMS。

**Android Drawable**

Drawable大多数时候是把各种不同类型的资源转化成drawable以便于view进行渲染，但不接受任何事件和用户交互

在设置背景的时候，view的draw方法调用drawable的draw方法，将内容绘制到view上。

Bitmap位图，后缀格式bmp，Drawable是Android平台下通用图像对象，包括bmp。

bitmap获取resource，通过bitmapFactory的decodeResource方法根据ID获取资源（调用resources中获取资源的方法）。

Drawable获取resource，通过resources中的getDrawable方法。

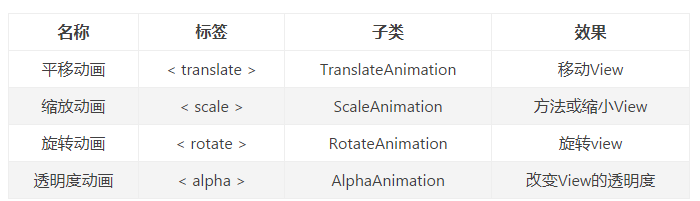
Drawable绘制流程：

使用drawable的方式主要是在xml文件中，设置背景，资源，代码中setimageResource，setImageDrawable，setBackgroudDrawable等

**Android动画详解**

视图动画view animation，帧动画drawable animation，属性动画property animation

视图动画作用对象是view，支持四种效果，平移，缩放，旋转，透明度，对一个view的动画。



该动画并未改变view的真实位置

Interpolator插值器：

定义动画的变化，使基础的动画可以加速减速重复等，除了诸多自带的插值器，还可以自定义插值器

Xml实现：

只能改变固有插值器的属性，不能修改属性。

Java方式：

继承Interpolator接口并实现getInterpolation方法。该方法输入动画的进度，根据计算，输出当前属性值的百分比。

Drawable Animation帧动画：顺序播放一组预先设置好的图片，不同于view动画，通过AnimationDrawable使用帧动画。

在xml文件中animation-list放入图片，设置view背景资源，并使用animationdrawable开启动画

Property Animation属性动画：修改view属性来实现的动画



持续时间duration，动画持续时间

时间插值TimeInterpolator计算属性值作为动画当前已用事件的函数

重复计数和行为Repeat Country and behavoir重复次数和方式

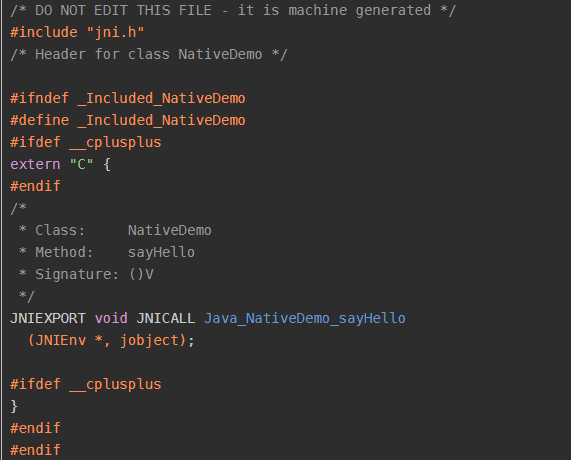
动画设置animationset，将动画分组为一起，或按顺序播放

帧刷新延迟frameRefreshDelay，指定刷新动画帧的频率

TypeEvaluator估值器：定义属性值计算方式的接口，根据属性起始结束插值一起计算出当前时间的属性值。

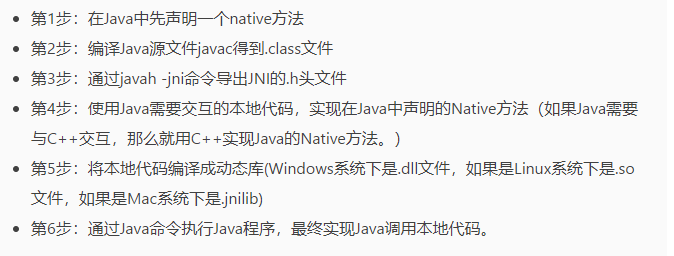
**JNI和NDK**

Java Native Interface，提供API，使Java和其他语言通信。在Java中，给方法加入native关键字，代表jni.h会据此找到C/C++编写的真正函数。



JNIEXPORT，作为JNI接口的标志，JNICALL，代表JNI要调用的东西，Java\_NativeDemo\_sayHello表示方法名，包名+类名+方法名。JNIEnv代表jni接口的对象，jobject代表native方法的调用者

在java类中，将编译好的so文件通过System.load加载进去



Java VM是JVM在JNI层的代表，封装了一些函数指针对JVM操作的接口

JNIEnv是当前java线程的执行环境，一个JVM对应一个Java VM结构，而一个JVM创建多个线程，每个线程对应一个JNIEnv结构，保存在线程本地存储TLS中，不同线程JNIEnv不同，不能互相共享，在本地代码中通过JNIEnv函数表来操作Java数据和方法，只要在本地代码中拿到JNIEnv结构，就可以调用java代码

JavaVM是Java虚拟机的JNI曾的代表，JNI全局仅有一个

JNIEnv是javaVM在线程中的代码，每个线程一个，JNI有非常多个JNIEnv

JNIEnv代表了java的执行环境，能够调用Java中的代码，还可以操作java代码

**AndroidSDK**

**Android数据库原理**

SQLiteOpenHelper，数据库操作帮助类，用来创建和升级数据库，提供两个抽象方法oncreate和onupgrade，还有两个实例方法，getReadableDatabase和getWritableDatabase前者打开或创建一个可读数据，后者打开或创建一个可读写数据库。

自定义Helper类，继承SQLiteOpenHelper，重写oncreate和onupgrade方法

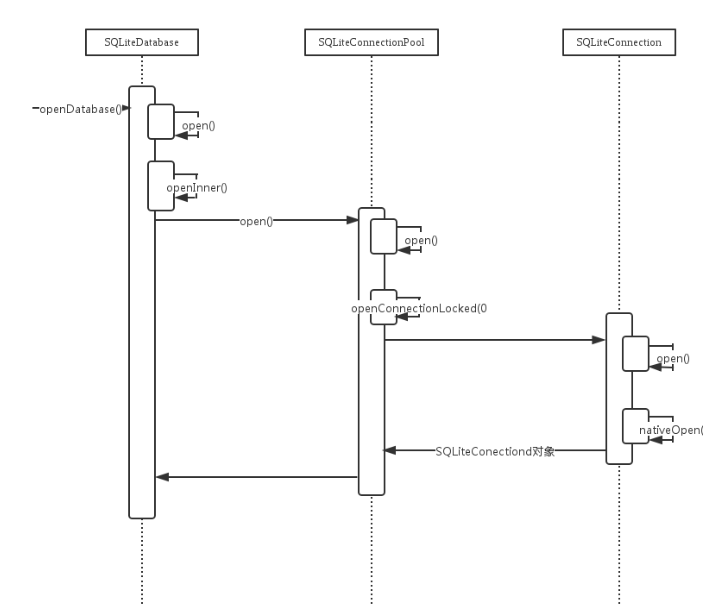
调用的时候，调用SQLiteOpenHelper中的getWritableDatabase方法。

在getWritableDatabase中，会使用SQLiteDatabase中的方法打开或创建数据库。

SQLiteDatabase中使用SQLiteConnectionPool数据库连接池中的SQLiteConnection来连接执行数据库操作

SQLiteSession，通过其获取SQLiteConnection连接，每一个线程都有自己的SQLiteSession对象，属于线程私有数据。

最终调用native的SQLiteConnection进行数据库的操作



**类加载，注解，反射**

类加载：.java文件经过Java编译器编译成.class文件，保存着java代码经转换后的虚拟机指令，当需要使用某个类时，虚拟机会加载它的.class文件，创建对应的class对象，将class文件加载到虚拟机内存，叫做类加载。

注解的原理：

注解通过反射获取，首先通过class对象的isAnnotationPresent方法判断它是否应用了某个注解，再通过getAnnotation获取Annotation对象，然后就可以调用它的属性方法了。

注解：通过@interface关键字定义，实际使继承了Anotation接口的接口

元注解：基本注解，可以注解到注解上的注解。

@Retention：保留期，应用到某个注解的时候，说明了该注解的存活时间

@Documented：将注解的元素包含到Javadoc中。

@Target，指定了注解运用的地方，被注解的注解，就被限定了运用的场景

@Inherited，如果一个超类被其注解过的注解注解，那么其子类自动拥有其父类的注解

@Repeatable，一个注解多次使用，需要使用该注解进行注解

注解的属性：注解只有成员变量，没有方法，在使用的时候，在注解的括号里赋值。如果成员变量有默认值，则可以不用赋值。

Java预置的注解

@Deprecated，标记过时的方法，类，成员变量等

@Override，提示子类复写父类被注解的方法

@SuppressWarnings，被@Deprecated注解过的方法，会被编译器警告，通过此注解忽略该警告

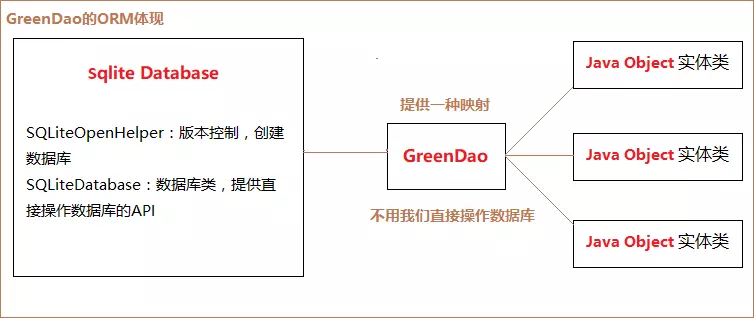
@SafeVarargs，提醒开发者不要用参数做一些不安全的操作

@Functionallnterface，函数式接口注解，将函数时接口转换成lambda表达式

反射：反射机制允许开发者在运行时发现和使用类型的信息。用户创建任何类，JAVA虚拟机均会创建该类对应class类的class对象，该对象保存了这个类的类型信息，并存在同名类的.class文件中，Class类的对象作用就是在运行时提供或获得某个对象的类型信息。

**常用框架源码分析（GreenDAO,ButterKnife,EventBus,RxJava,Glide）**

GreenDao：对象关系映射框架。对象关系映射，是一种将对象层次结构映射成关系型结构的方法。即将一张数据表按照统一的规则映射成一个Java实体类。



ButterKnife：依赖注入框架，使用注解的方式，实现view和onClickListener的注入，省去了findviewbyid和Setonclicklistener的过程。

利用编译时注解技术，在编译时扫描所有注解，对注解进行处理，在项目中生成一份代码，并在项目中直接调用。

1. 编译时，利用ButterKnifeProcessor类的process方法实现：获取所有标记了@BindView，@OnClick等注解的元素，定义一个key为类级别元素，value为注解信息的Map集合，根据获取的元素可以拿到当前类级别元素和注解标记的信息，将其put进去，map里保存诸多生成类和类中信息
2. 通过javapoet生成辅助类，即含有findViewById和setOnClickListener等的类
3. 运行时，ButterKnife.bind(this)会反射创建辅助类，辅助类中会构造findviewbyid等方法，在ondestory中解除绑定。

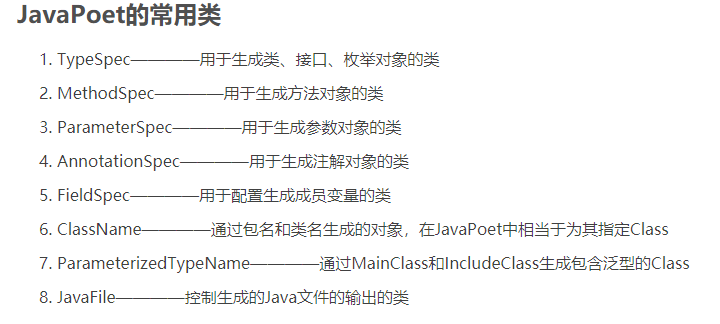
Bind方法，通过传入的activity对象，获取其view，然后createBinding方法创建辅助类，其中findBindingConstructorForClass，维护一个map集合，key为class，value为构造器。

在这个构造器中，构建辅助类，类名在Activity名后加viewbinding，创建构造器，最后放入map集合，返回构造器。

在编译期间，注解处理器ButterKnifeProcessor执行其中的process方法，解析注解并生成辅助类文件

扫描注解过的元素，放入集合，循环查找，并调用BindingSet，使用JavaPoet生成辅助类文件

JavaPoet：自动生成Java文件的第三方依赖。



常用方法：

addModifiers，设置修饰符

addAnnotation，设置注解对象

addJavadoc，设置注释

addParameter，设置方法参数

addCode,addStatement，设置方法体

returns，设置返回值

addException，设置异常

Rxjava

使用方法，创建被观察者Observable，调用其create方法，将ObservableOnSubscribe作为参数传入。

创建观察者Observer，将二者通过订阅关系绑定。

Create方法：

创建ObservableCreate对象，重写subscribeActual方法，在其中创建CreateEmitter对象，传入observer，observer订阅createEmitter对象，ObserverOnSubscribe订阅createEmitter对象（实现ObserverOnSubscribe中的subscribe方法，将ObserverOnSubscribe事件源与订阅者实现订阅关系）

CreateEmitter对象，实现onNext等方法，调用observer的onNext方法等，并在ObserverOnSubscribe中的subscribe中调用

创建观察者Observer，实现其中四个方法。

订阅关系，依然是调用subscribeActual方法。

Map方法，将事件源的参数进行转化，map方法中同样调用subscribeActual，新建一个事件源，并与订阅者订阅

EventBus

使用的时候，通过EventBus.getDefault方法获取EventBus的单例，然后调用register方法，注册。Register方法中，获取传入的类的class信息，通过findSubscriberMethods方法，获取class中被注解的方法，并存入列表中，然后循环列表，调用subscribe方法，将该类和注解的方法进行注册。

findSubscriberMethods方法，首先通过缓存寻找已注册的信息，如果没有，通过反射的方式，获取到类中被@subscribe注解过的方法，并返回（第一种是通过最顶层的父类遍历获取订阅方法信息，如果没有，就是用反射获取注解过的方法）。

Subscribe方法，将对应的方法放入一个队列中，完成注册

发送消息，使用post方法，将当前事件放入事件队列，通过循环，使用postSingleEvent方法发送消息，最后通过postToSubscription方法，对注解中不同的线程进行不同的消息分发，继承Handler实现多个方法。

Glide

Picasso

Retrofit

okhttp

**线程和线程池**

**设计模式**

**Object类**

**IO**

**多线程详解**

**Android打包过程**

**Android热更新和插件化**