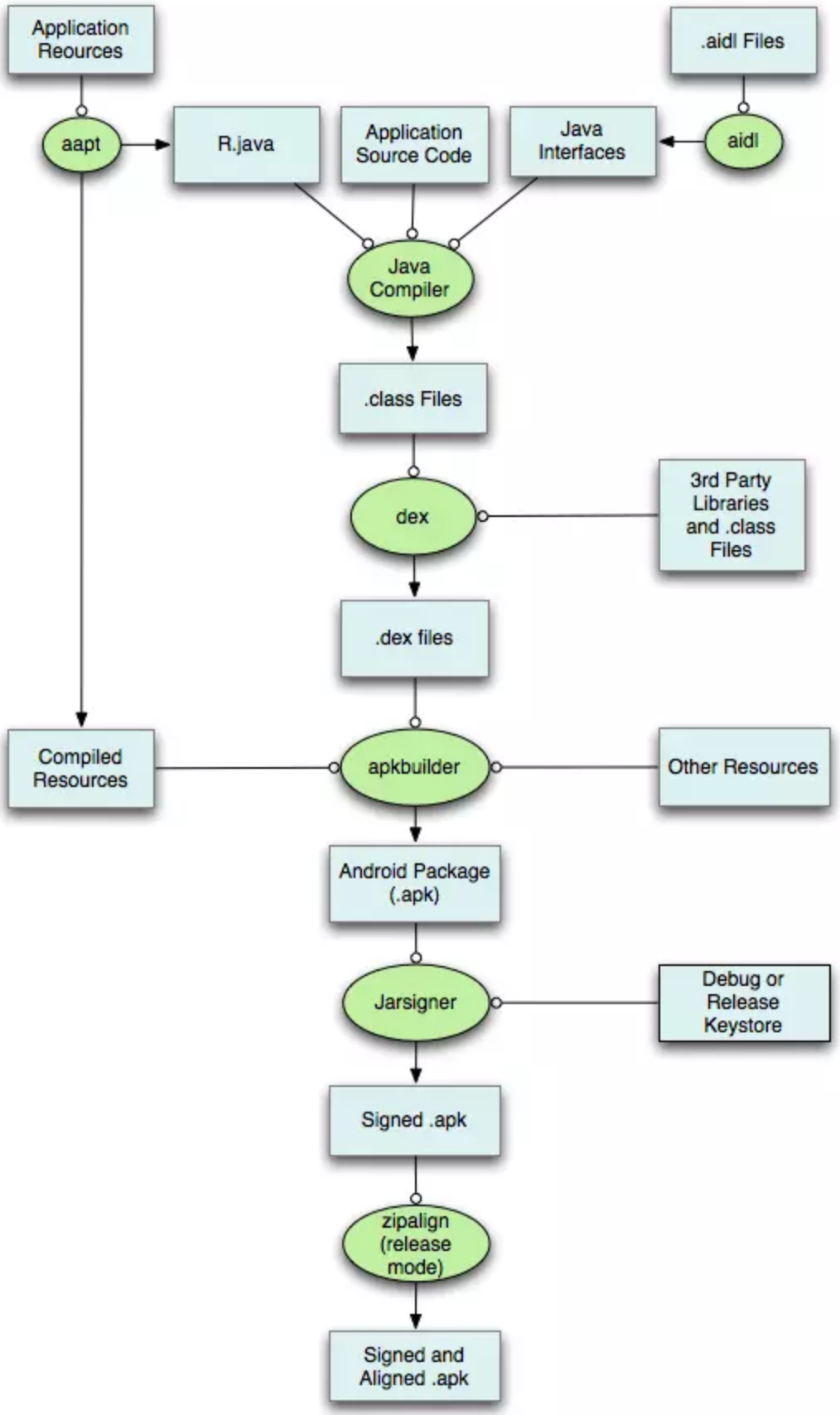
面试杂记

Android项目构建流程

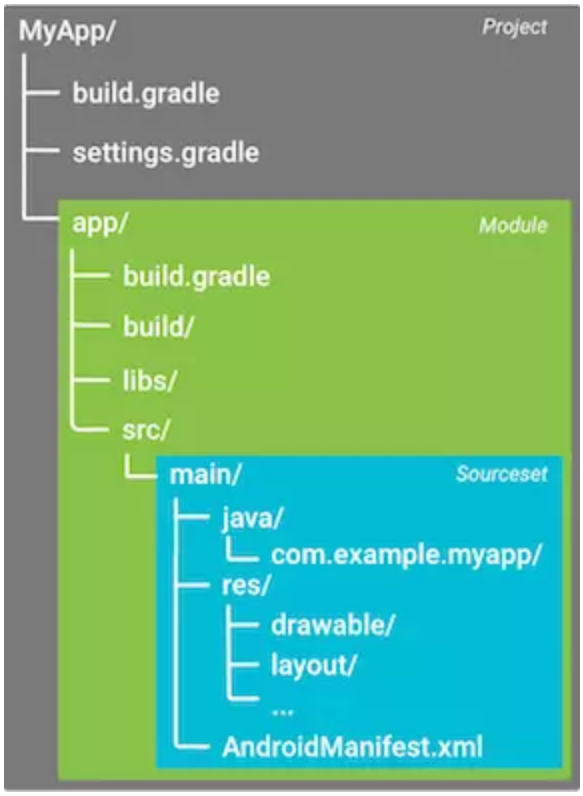
- 首先aapt工具会将资源文件进行转化，生成对应资源ID的R文件和资源文件。
- adil工具会将其中的aidl接口转化成Java的接口
- 至此，Java Compiler开始进行Java文件向class文件的转化，将R文件，Java源代码，由aidl转化来的Java接口，统一转化成.class文件。
- 通过dx工具将class文件转化为dex文件。
- 此时我们得到了经过处理后的资源文件和一个dex文件，当然，还会存在一些其它的资源文件，这个时候，就是将其打包成一个类似apk的文件。但还并不是直接可以安装在Android系统上的APK文件。
- 通过签名工具对其进行签名。
- 通过Zipalign进行优化，提升运行速度。
- 最终，一个可以安装在我们手机上的APK了。



Android 构建系统编译应用资源和源代码，然后将它们打包成可供您测试、部署、签署和分发的 APK。Android Studio 使用 Gradle 这一高级构建工具包来自动化执行和管理构建流程，同时也允许您定义灵活的自定义构建配置。每个构建配置均可自行定义一组代码和资源，同时对所有应用版本共有的部分加以重

复利用。Android Plugin for Gradle 与这个构建工具包协作，共同提供专用于构建和测试 Android 应用的流程和可配置设置。

Gradle 和 Android 插件独立于 Android Studio 运行。这意味着，可以在 Android Studio 内、使用计算机上的命令行工具或在未安装 Android Studio 的计算机（例如持续性集成服务器）上构建 Android 应用。如果您不使用 Android Studio，可以学习如何从命令行构建和运行您的应用。无论您是从命令行、在远程计算机上还是使用 Android Studio 构建项目，构建的输出都相同。

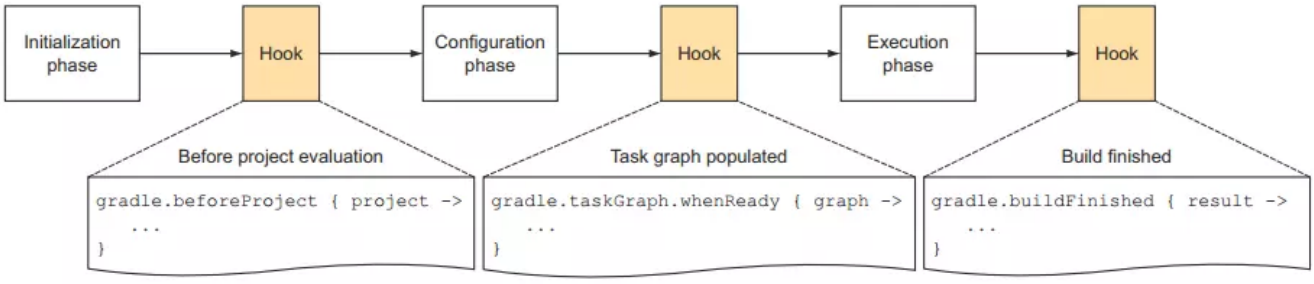


如上图所示，在一个Project中，除了我们项目自身的代码和资源之外，会有多个与项目构建相关的.gradle文件，这些.Gradle文件用来对于我们使用Gradle进行构建项目的整个过程中来使用。

Gradle中，每一个待编译的工程都叫一个Project。每一个Project在构建的时候都包含一系列的Task。比如一个Android APK的编译可能包含：Java源码编译Task、资源编译Task、JNI编译Task、lint检查Task、打包生成APK的Task、签名Task等。

Gradle工作流程

Gradle的工作流程如下图所示，在每一个工作流程的前后，我们都可以进行一些hook操作，来满足自己的需求。



Gradle工作包含三个阶段：

- 首先是初始化阶段。对我们前面的multi-project build而言，就是执行settings.gradle

- Initiliazation phase的下一个阶段是Configration阶段。
- Configration阶段的目标是解析每个project中的build.gradle。比如multi-project build例子中，解析每个子目录中的build.gradle。在这两个阶段之间，我们可以加一些定制化的Hook。这当然是通过API来添加的。
- Configuration阶段完了后，整个build的project以及内部的Task关系就确定了。一个Project包含很多Task，每个Task之间有依赖关系。Configuration会建立一个有向图来描述Task之间的依赖关系。所以，我们可以添加一个HOOK，即当Task关系图建立好后，执行一些操作。
- 最后一个阶段就是执行任务了。当然，任务执行完后，我们还可以加Hook。

简言之，Gradle有一个初始化流程，这个时候settings.gradle会执行。 在配置阶段，每个Project都会被解析，其内部的任务也会被添加到一个有向图里，用于解决执行过程中的依赖关系。然后才是执行阶段。你在gradle xxx中指定什么任务，gradle就会将这个xxx任务链上的所有任务全部按依赖顺序执行一遍！

Gradle主要有三种对象

这三种对象和三种不同的脚本文件对应，在gradle执行的时候，会将脚本转换成对应的对象：

- Gradle对象：当我们执行gradle xxx或者什么的时候，gradle会从默认的配置脚本中构造出一个Gradle对象。在整个执行过程中，只有这么一个对象。Gradle对象的数据类型就是Gradle。我们一般很少去定制这个默认的配置脚本。
- Project对象：每一个build.gradle会转换成一个Project对象。
- Settings对象：显然，每一个settings.gradle都会转换成一个Settings对象。

构建的生命周期，首先根据settings.gradle文件构建出一个Seetings对象，然后根据Seetings中的配置，创建Project对象，去找各个project下的build.gradle文件，根据文件内容来对project对象进行配置。

Jetpack

Lifecycle

LiveData

ViewModel

MVC、MVP、MVVM：

MVC：

视图层(View)：对应于xml布局文件和java代码动态view部分
控制层(Controller)：MVC中Android的控制层是由Activity来承担的，Activity本来主要是作为初始化页面，展示数据的操作，但是因为XML视图功能太弱，所以Activity既要负责视图的显示又要加入控制逻辑，承担的功能过多。
模型层(Model)：针对业务模型，建立的数据结构和相关的类，它主要负责网络请求，数据库处理，I/O的操作。
总结：具有一定的分层，model彻底解耦，controller和view并没有解耦
– 层与层之间的交互尽量使用回调或者去使用消息机制去完成，尽量避免直接持有

– controller和view在android中无法做到彻底分离，但在代码逻辑层面一定要分清
– 业务逻辑被放置在model层，能够更好的复用和修改增加业务

MVP：

MVP也是三层，唯一的差别是Model和View之间不进行通讯，都是通过Presenter完成。
Contract 契约类这是Google MVP与其他实现方式的又一个不同，契约类用于定义同一个界面的view的接口和presenter的具体实现。
总结：通过引入接口BaseView，让相应的视图组件如Activity，Fragment去实现BaseView，实现了视图层的独立，通过中间层Preseter实现了Model和View的完全解耦。MVP彻底解决了MVC中View和Controller傻傻分不清楚的问题，但是随着业务逻辑的增加，一个页面可能会非常复杂，UI的改变是非常多，会有非常多的case，这样就会造成View的接口会很庞大。

MVVM：

MVP中我们说过随着业务逻辑的增加，UI的改变多的情况下，会有非常多的跟UI相关的case，这样就会造成View的接口会很庞大。而MVVM就解决了这个问题，通过双向绑定的机制，实现数据和UI内容，只要想改其中一方，另一方都能够及时更新的一种设计理念，这样就省去了很多在View层中写很多case的情况，只需要改变数据就行。
MVVM中 View和ViewModel通过Binding进行关联，他们之前的关联处理通过DataBinding完成。
MVVM是一种思想，DataBinding是谷歌推出的方便实现MVVM的工具。在google推出DataBinding之前，因为xml layout功能较弱，想实现MVVM非常困难。而DataBinding的出现可以让我们很方便的实现MVVM。
总结：看起来MVVM很好的解决了MVC和MVP的不足，但是由于数据和视图的双向绑定，导致出现问题时不太好定位来源，有可能数据问题导致，也有可能业务逻辑中对视图属性的修改导致。如果项目中打算用MVVM的话可以考虑使用官方的架构组件ViewModel、LiveData、DataBinding去实现MVVM

AspectJ

AspectJ主要组成部分：

- aspectjrt.jar包主要是提供运行时的一些注解，静态方法等等东西，通常我们要使用aspectJ的时候都要使用这个包。
- aspectjtools.jar包主要是提供赫赫有名的ajc编译器，可以在编译期将将java文件或者class文件或者aspect文件定义的切面织入到业务代码中。通常这个东西会被封装进各种IDE插件或者自动化插件中。
- aspectjweaverjar包主要是提供了一个java agent用于在类加载期间织入切面(Load time weaving)。并且提供了对切面语法的相关处理等基础方法，供ajc使用或者供第三方开发使用。这个包一般我们不需要显式引用，除非需要使用LTW。

AspectJ的几种标准的使用方法：

- 编译时织入，利用ajc编译器替代javac编译器，直接将源文件(java或者aspect文件)编译成class文件并将切面织入进代码。
- 编译后织入，利用ajc编译器向javac编译期编译后的class文件或jar文件织入切面代码。

- 加载时织入，不使用ajc编译器，利用aspectjweaver.jar工具，使用java agent代理在类加载期将切面织入进代码。

免注册跳转Activity

- hook点1：IActivityManager接口中的startActivity方法，hook方法(通过反射获取到Singleton类中的mInstance(IActivityManager)，用于在IActivityManager接口的动态代理中"瞒天过海"之后，继续执行IActivityManager#startActivity方法)
- hook点2：ActivityThread中的mH(Handler)成员中的mCallBack成员(通过反射获取到，ActivityThread中的mH(Handler)成员中的mCallBack成员，然后通过Handler分发消息顺序原则：msg.callback > mCallback > message，给mCallBack设置我们自定义的callback，然后在自定义cllback的handleMessage方法中，替换回真实的Intent)

HashMap

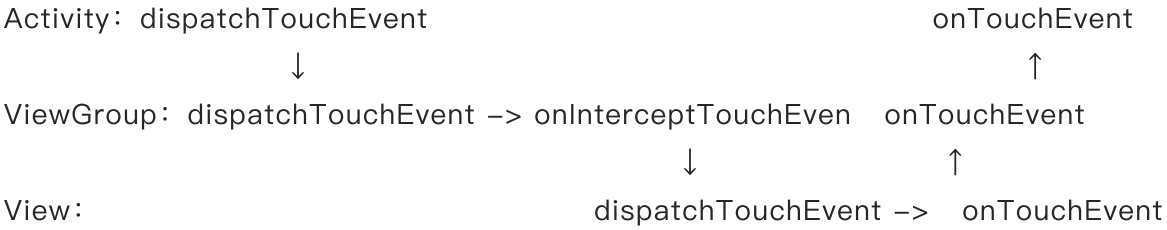
1、hashmap的默认容量是多少，为什么这样设计?
length 的值为2 的整数次幂，h & (length – 1)相当于对 length 取模。这样提高了效率也使得数据分布更加均匀。
为什么会更加均匀？length的值为偶数，length – 1 为奇数，则二进制位的最后以为为1，这样保证了h & (length – 1)的二进制数最后一位可能为1，也可能为0。如果为length为奇数，那么就会浪费一半的空间。
2、hashmap 1.7为什么要先扩容再添加，1.8为什么是先添加再扩容
扩容后数据存储位置的计算方式也不一样：
1. 在JDK1.7的时候是直接用hash值和需要扩容的二进制数进行&（这里就是为什么扩容的时候为啥一定必须是2的多少次幂的原因所在，因为如果只有2的n次幂的情况时最后一位二进制数才一定是1，这样能最大程度减少hash碰撞）（hash值 & length–1）
JDK1.7中的话，是先进行扩容后进行插入的，就是当你发现你插入的桶是不是为空，如果不为空说明存在值就发生了hash冲突，那么就必须得扩容，但是如果不发生Hash冲突的话，说明当前桶是空的（后面并没有挂有链表），那就等到下一次发生Hash冲突的时候在进行扩容，但是当如果以后都没有发生hash冲突产生，那么就不会进行扩容了，减少了一次无用扩容，也减少了内存的使用。
2、而在JDK1.8的时候分两种情况：
1. 该结点下无链表或红黑树：和JDK1.7一致 e.hash & 新扩容的大小 – 1。
2. 该结点下有链表或红黑树：数据存储位置 = 扩容前的原始位置 or 扩容前的原始位置+扩容的大小值。
但是这种方式就相当于只需要判断Hash值的"新增参与运算的位"是0还是1就直接迅速计算出了扩容后的储存方式。
什么是"新增参与运算的位"？ e.hash & length–1
扩容前length = 16，则参与 hash & length–1 的位实际上就是 15的二进制1111四位，因为前面都是0
扩容后length = 32，则参与 hash & length–1 的位实际上就是 32的二进制11111五位，其余前面都是0，因此若新参与运算的第一位，为0则数据仍保存原有位置，否则保存在：扩容前的原始位置+扩容的大小值这样的好处是可以更快速的计算出了扩容后的储存位置。
JDK1.8先添加后扩容是因为，1.7中所有结点的存储位置都需要根据扩容后的大小重新计算位置，而1.8中不需要。
3、1.7&1.8插入数据的规则是什么?

JDK1.7用的是头插法，而JDK1.8及之后使用的都是尾插法，那么他们为什么要这样做呢？因为JDK1.7是用单链表进行的纵向延伸，当采用头插法时会容易出现逆序且环形链表死循环问题。但是在JDK1.8之后是因为加入了红黑树使用尾插法，能够避免出现逆序且链表死循环的问题。
多线程下HashMap的死循环：
HashMap是采用链表解决Hash冲突，因为是链表结构，那么就很容易形成闭合的链路，这样在循环的时候只要有线程对这个HashMap进行get操作就会产生死循环。
在单线程情况下，只有一个线程对HashMap的数据结构进行操作，是不可能产生闭合的回路的。
那就只有在多线程并发的情况下才会出现这种情况，那就是在put操作的时候，如果size>initialCapacity*loadFactor，那么这时候HashMap就会进行rehash操作，随之HashMap的结构就会发生翻天覆地的变化。很有可能就是在两个线程在这个时候同时触发了rehash操作，产生了闭合的回路。
即：put时，若多个线程都需要扩容，则此时线程一拿到时间片，rehash后转移原数据到新的hash表中时，带有链表的节点会从头到尾遍历插入新的链表中，由于1.7是头插法这就导致了原来链表的头结点变成了尾结点，即原来的头结点.next=尾结点；在线程切换至线程二时，此时线程二也需要rehash，此时链表就变成了尾.next=头，就造成了环形链表出现啦。不调用get方法不会出现问题，一旦调用get就有可能死循环。
4. 为什么在JDK1.8中进行对HashMap优化的时候，把链表转化为红黑树的阈值是8,而不是7或者不是20呢（面试蘑菇街问过）？
1). 如果选择6和8（如果链表小于等于6树还原转为链表，大于等于8转为树），中间有个差值7可以有效防止链表和树频繁转换。假设一下，如果设计成链表个数超过8则链表转换成树结构，链表个数小于8则树结构转换成链表，如果一个HashMap不停的插入、删除元素，链表个数在8左右徘徊，就会频繁的发生树转链表、链表转树，效率会很低。
2). 还有一点重要的就是由于treenodes的大小大约是常规节点的两倍，因此我们仅在容器包含足够的节点以保证使用时才使用它们，当它们变得太小（由于移除或调整大小）时，它们会被转换回普通的node节点，容器中节点分布在hash桶中的频率遵循泊松分布，桶的长度超过8的概率非常非常小。所以作者应该是根据概率统计而选择了8作为阀值

事件分发：

Activity => ViewGroup => View 的顺序进行事件分发，然后通过 dispatchTouchEvent(MotionEvent ev) => onInterceptTouchEvent(MotionEvent event) => onTouchEvent(MotionEvent event) 顺序调用。

具体顺序：



事件拦截：

- 外部拦截法：onInterceptTouchEvent条件判断是否拦截
- 内部拦截法：子View的dispatchTouchEvent中请求父布局不拦截
- 注：父布局的onInterceptTouchEvent()方法会比该次事件下子View的dispatchTouchEvent()方法要更早进行（如果该事件会分发给子View的话），所以若子View设置getParent().requestDisallowInterceptTouchEvent(true)代码的那次父布局事件的onInterceptTouchEvent()返回的是true，那么父布局依然会拦截事件，子View的这次禁用父布局拦截事件请求将会失败。若父布局的ACTION_MOVE事件某些条件下返回true而ACTION_DOWN事件返回false，而此时其子布局在其ACTION_DOWN事件下，设置getParent().requestDisallowInterceptTouchEvent(true)禁止父布局拦截的事件，则在发生ACTION_MOVE事件时则不会响应父布局的事件，可禁止成功。原因：在设置禁止父布局拦截为true方法中，实际为上是设置了其成员变量mGroupFlags为FLAG_DISALLOW_INTERCEPT状态，而由于父布局的onInterceptTouchEvent方法比子布局的所有事件分发方法都早，因此设置禁止父布局拦截事件时会在下一次事件发生才生效，直到ACTION_UP事件发生时父布局调用resetTouchState()方法重置触摸状态。

```
底层原理：INotify和Epoll机制
    Android 输入系统处理事件：监控设备节点，当某个设备节点有数据可读时，将数据读出并进行一系列的翻译加工，然后在所有的窗口中找到合适的事件接收者，并派发给它。
    Android 的输入系统InputManagerService（以下简称为IMS）作为系统服务，它像其他系统服务一样在SystemServer进程中创建。
    Linux会为所有可用的输入设备在/dev/input目录在建立event0~n或者其他名称的设备节点，Android输入系统会监控这些设备节点，具体是通过INotify和Epoll机制来进行监控。
    INotify机制：INotify是Linux内核提供的一种文件系统变化通知机制。它可以为应用程序监控文件系统的变化，如文件的新建，删除等。
    Epoll机制：事件是随机发生的，我们也不会本末倒置的采用轮询的方式轮询这个描述符，因为如果这样做的话会浪费大量系统资源。这时候我们Linux的另一个机制就派上用场了，即Epoll机制。Epoll机制简单的说就是使用一次等待来获取多个描述的可读或者可写状态。
Avtivity,Window,PhoneWindow,以及ViewRootImpl之间的联系：
    我们知道Activity的启动流程，Activity对象最先创建，但是Activity的显示是依靠其内部对象Window mWindow,而Window是个抽象类，所以mWindow指向的实际上是Window的实现类PhoneWindow的对象。PhoneWindow作为显示的载体，ViewRootImpl的measure、layout以及draw才是View显示的动力所在。
    当我们的PhoneWindow创建完成之后，我们也在该Window上注册了InputChannel并与IMS通信，IMS把事件写入InputChannel，WindowInputEventReceiver对事件进行处理并最终还是通过InputChannel反馈给IMS。
    InputChannel：本质是一对SocketPair(非网络套接字)。套接字可以用于网络通信，也可以用于本机内的进程通信，是进程间通信的一种方式。
    WindowInputEventReceiver：ViewRootImpl#setView中先初始化InputChannel后，便用它创建WindowInputEventReceiver，WindowInputEventReceiver继承于In
```

putEventReceiver，InputEventReceiver对象可以接收来自InputChannel的输入事件，并触发其onInputEvent方法的回调。

12 总结：当我们触摸（点击）屏幕时，Android输入系统IMS通过对事件的加工处理，然后会在所有的窗口中找到合适的事件接收者，并通过InputChannel向Window派发加工后的事件，并触发InputEventReceiver的onInputEvent的调用，由此产生后面一系列的调用，把事件派发给整个控件树的根DecorView。

13 而DecorView又上演了一出偷梁换柱的把戏，先把事件交给Activity处理，在Activity中再传递给Window(if(getWindow().superDispatchTouchEvent(ev)){return;})，最后Window再传递给顶层View DecorView。自此沿着控件树自上向下依次派发事件。若从Window事件传递到最顶层View均不处理事件，则最后事件回到Activity的onTouchEvent方法中。

14 图解Android事件分发机制（深入底层源码）：https://segmentfault.com/a/1190000012227736

如果子view在down事件没有消费事件，在move和up事件还会分发到这个子view吗

- 事件序列是指一组1个down事件，n个move事件和1个up事件组成的。
- 正常情况下，一组事件序列都只能被一个View拦截并消耗，除非通过特殊手段强制转移给其他view处理；
- 当ViewGroup决定拦截一组事件序列后(这里指的是down事件)，后面的点击事件(down、move和up事件)会有它自己处理，而且都不会调用onInterceptTouchEvent()方法，因为既然决定都要它处理了，以后都不需要再次判断是否需要拦截，符合生活规律。
- 如果down事件被消费掉(onTouchEvent返回true)， 以后的move事件和up事件都会发给这个view，如果不消费move和up事件(onTouchEvent返回false)，也会不断接收到move事件或up事件，这些事件会消失，不会被发送给parent的onTouchEvent，最终这些消失的事件会被activity处理。
- 如果down事件没有被消费掉(onTouchEvent返回false)，这个事件就会被parent的onTouchEvent处理；以后的move和up事件都不会被再发给这个view了。实际当中，上级让你完成一个任务，你没有完成，把任务交给上司，那上司在短时间内不会再把任何交给你了，对你不信任了。
- ViewGroup的onInterceptToucheEvent默认是false，不拦截；
- View没有onInterceptToucheEvent，分到事件后直接调用onTouchEvent。
- onTouchEvent默认是true，只要有clickable或者longClickable有一个为true。
- 默认longClickable为false，但是clickable根据不同的控件类型默认值不一样。比如Buttton的clickable默认为true，但是Textview的clickable默认是false。
- setEnable(false)不会影响事件传递。
- onClick能执行的前提是是可点击的(clickable和longClickable至少一个为true)，并且能收到down和up事件。
- setOnTouchListener以后，这个监听器的onTouch方法的返回值将影响后续的onTouchEvent流程。如果onTouch返回true，后续OnTouchEvent不会执行，onClick也不会执行了。如果onTouch返回

false，onTouchEvent接着执行。onTouchEvent返回true并且enable为true，onClick也会在up事件后被调用。

- 事件传递都是由外到内的，由父元素传到子元素的。通过requestDisallowInterceptTouchEvent能让子元素影响父元素的事件分发过程，但是ACTION_DOWN是无法影响的。

点击事件和长按事件的优先级谁高，他们分别是在down、move、up中那个事件触发的。

长按点击的时候默认(默认return false)会触发点击事件，android系统是优先点击的，并且没有返回值；而长按事件是有返回值的，如果返回false，两个事件都会有响应，如果返回true则只响应长按事件。

点击事件触发时机：onClick事件是在手指抬起后触发的
长按事件触发时机：down事件触发后，通过postDelayed延迟发送了一个Runable对象：mPendingCheckForLongPress。延时时间是ViewConfiguration.getLongPressTimeout() – delayOffset的时间差。综上，简单来说，当我们按下屏幕的时候发送了一个延时的Runable，然后等到Runable被执行的时候，在通过一些标志位判断当前是否还满足长按被执行的条件，如果满足，回调Listener中的onLongClick。

Gradle

Gradle执行流程

1. 初始化阶段：执行settings.gradle脚本，解析整个工程中所有Project，构建所有Project对应的project对象。
2. 配置(构建)阶段：解析所有project对象中的task对象，构建好所有task的拓扑图
3. 执行阶段：执行具体的task以及依赖的task

Gradle生命周期

- settings.gradle：
  - gradle.settingsEvaluated：settings.gradle，配置完后调用
  - gradle.projectsLoaded：当settings.gradle中引入的所有project都被创建好后调用
- 配置(构建)阶段：配置每个project前后，以及所有配置前后的回调
- 执行阶段：每个任务执行前后的，以及所有任务执行前后的回调

自定义插件

工程创建步骤：

1. 在工程目录下创建buildSrc文件夹。
2. 在buildSrc目录下，创建src文件夹、build.gradle文件。
3. 在buildSrc/src目录下，再创建main文件夹。
4. 在buildSrc/src/main目录下，再分别创建groovy、resources文件夹。

5. 在buildSrc/src/main/resources再创建一个META-INF文件夹，再在META-INF下创建一个gradle-plugins文件夹。
6. 配置build.gradel文件
7. Async一下工程

元素：

```
class GradleStudyPlugin implements Plugin { @Override  void apply(Project project) {// 插件引入时要执行的方法} }
class ReleaseInfoExtension { // 扩展属性} // GradleStudyPlugin#apply方法中配置：project.extensions.create("releaseInfo", ReleaseInfoExtension)
class ReleaseInfoTask extends DefaultTask { @TaskAction  void doAction() {// @TaskAction注解的任务在doFirst和doLast中间执行 }} // GradleStudyPlugin#apply方法中配置：project.tasks.create("updateReleaseInfo", ReleaseInfoTask)
```

View自定义

View显示流程：

onCreate中先初始化了DecoView(根布局)，然后会调用mLayoutInflater.inflate()方法来填充布局，inflate方法会使用Xml解析器，解析我们传入的xml文件，并保存到mContentParent里，contentParent指的是DecorView的ContentView部分。

ActivityThread#handleResumeActivity(即将执行onResume)中，wm.addView(decor, l)由此WindowManager和DecoView产生了关联，而在WindowManagerImpl#addView又会代理WindowManagerGlobal#addView执行初始化ViewRootImpl，并将view设置到ViewRoot中：root.setView(view, wparams, panelParentView)。

WindowManager是一个接口，里面常用的方法有：添加View，更新View和删除View，WindowManager继承自ViewManager，这三个方法定义在ViewManager中。这些方法传入的参数是View，不是Window，说明WindowManager管理的是Window中的View，我们通过WindowManager操作Window就是在操作Window中的View。WindowManager 会控制Window窗口对象，它们是用于容纳视图对象的容器。窗口对象始终由 Surface 对象提供支持。WindowManager 会监督生命周期、输入和聚焦事件、屏幕方向、转换、动画、位置、变形、Z 轴顺序以及窗口的许多其他方面。WindowManager 会将所有窗口元数据发送到 SurfaceFlinger，以便 SurfaceFlinger 可以使用这些数据在屏幕上合成 Surface。

Window，WindowManager和WindowManagerService之间的关系：

Window：

Window在Android开发中是一个窗口的概念，它是一个抽象类。以Activity对应的Window为例，具体的实现类是PhoneWindow，在PhoneWindow中有一个顶级View—DecorView，继承自FrameLayout，我们可以通过getDecorView()获得它，当我们调用Activity的setContentView时，其实最终会调用Window的setContentView，当我们调用Activity的findViewById时，其实最终调用的是Window的findViewById，这也间接的说明了Window是View的直接管理者。但是Window并不是真实存在的，它更多的表示一种抽象的功能集合，View才是Android中的视图呈现形式，绘制到屏幕上的是View不是Window，但是View不能单独存在，它必需依附在Window这个抽象的概念上面，Android中需要依赖Window提供视图的有Activity，Dialog，Toast，PopupWindow，StatusBarWindow（系统状态栏），输入法窗口等，因此Activity，Dialog等视图都对应着一个Window。

WindowManager(实际为WindowManagerImpl)：

WindowManager是一个接口，里面常用的方法有：添加View，更新View和删除View，WindowManager继承自ViewManager，这三个方法定义在ViewManager中。这些方法传入的参数是View，不是Window，说明WindowManager管理的是Window中的View，我们通过WindowManager操作Window就是在操作Window中的View。WindowManager 会控制Window窗口对象，它们是用于容纳视图对象的容器。窗口对象始终由 Surface 对象提供支持。WindowManager 会监督生命周期、输入和聚焦事件、屏幕方向、转换、动画、位置、变形、Z 轴顺序以及窗口的许多其他方面。WindowManager 会将所有窗口元数据发送到 SurfaceFlinger，以便 SurfaceFlinger 可以使用这些数据在屏幕上合成 Surface。

WindowManager#addView中初始了ViewRootImpl，而ViewRootImpl的setView中，首先通过requestLayout()发起View绘制流程，然后在mWindowSession#addToDisplay中通过Binder与WMS进行跨进程通信，请求显示窗口上的视图，至此View就会显示到屏幕上。这个mWindowSession是一个IWindowSession.AIDL接口类型，用来实现跨进程通信，在WMS内部会为每一个应用的请求保留一个单独的Session，同样实现了IWindowSession接口，应用与WMS之间的通信就通过这个Session。

mWindowSession是通过WindowManagerGlobal的单例类getWindowSession()--> 首先获取WMS的本地代理：IWindowManager windowManager = getWindowManagerService(); --> 最后通过WMS的本地代理的openSession来获取Session：windowManager.openSession

WindowManagerService：

WindowManagerService是一个系统级服务，由SystemService启动，实现了IWindowManager.AIDL接口，它的主要功能分为窗口管理和输入事件的中转站。而上面mWindowSession#addToDisplay中实际上是调用mService.addWindow，即WMS#addWindow添加显示。addWindow主要做的事情是先进行窗口的权限检查，因为系统窗口需要声明权限，然后根据相关的Display信息以及窗口信息对窗口进行校对，再然后获取对应的WindowToken，再根据不同的窗口类型检查窗口的有效性，如果上面一系列步骤都通过了，就会为该窗口创建一个WindowState对象，以维护窗口的状态和根据适当的时机调整窗口状态，最后就会通过WindowState的attach方法与SurfaceFlinger通信。因此SurfaceFlinger能使用这些Window信息来合成surfaces,并渲染输出到显示设备。

```
1 // 注：该方法中的mWindow实际上并不是真正的Window，而是IWindow.Stub，即用于Bind
```

```
ler跨进程通信的回调
// 而此处的mWindow(W即IWindow)内部继承的方法，会通知ViewRootImpl一些事件。这里的事件指的就是按键、触屏等事件。
mWindowSession.addToDisplay(mWindow, mSeq, mWindowAttributes,
getHostVisibility(), mDisplay.getDisplayId(),
mAttachInfo.mContentInsets, mAttachInfo.mStableInsets,
mAttachInfo.mOutsets, mInputChannel);
```

ViewRoot对应ViewViewRootImpl类，它是连接WindowManager和DecorView的纽带，而从root.setView方法中requestLayout()开启了View的绘制流程。View的三大流程（measure,layout,draw）均是通过ViewRoot来完成的。ViewRootIml是View的根类，其控制着View的测量、绘制等操作

自定义View分类：

继承系统控件View，例如：TextView、ImageView等
继承系统ViewGroup，例如LinearLayout、FrameLayout等（一般指自定义组合控件）
继承View，重写onMeasure()、onDraw()等，纯自己测量绘制
继承ViewGroup，重写onMeasure()、onLayout()，纯自己测量布局

自定义View绘制：

- 自定义绘制的方式是重写绘制方法，其中最常用的是 onDraw()
- 绘制的关键是 Canvas 的使用：
  - Canvas 的绘制类方法： drawXXX() （关键参数：Paint）
  - Canvas 的辅助类方法：范围裁切(clipXXX()等)和几何变换(Matrix)
  - Canvas.save() 和 Canvas.restore() 及时保存和恢复绘制范围，分别用在范围裁切和几何变换前后
- 可以使用不同的绘制方法来控制遮盖关系(绘制顺序：背景、主体、子View、滑动边缘渐变和滑动条、前景)

```
drawBackground() 绘制背景，不允许重写
onDraw() 绘制主体
    写在 super.onDraw() 的上面
        如果把绘制代码写在 super.onDraw() 的上面，由于绘制代码会执行在原有内容的绘制之前，所以绘制的内容会被控件的原内容盖住。
dispatchDraw()：绘制子 View 的方法
    写在 super.dispatchDraw() 的下面
        只要重写 dispatchDraw()，并在 super.dispatchDraw() 的下面写上你的绘制代码，这段绘制代码就会发生在子 View 的绘制之后，从而让绘制内容盖住子 View 了。
    写在 super.dispatchDraw() 的上面
        把绘制代码写在 super.dispatchDraw() 的上面，这段绘制就会在 onDraw() 之后、 super.dispatchDraw() 之前发生，也就是绘制内容会出现在主体内容和子 View
```

```
之间。
onDrawForeground() API 23 才引入的，会依次绘制滑动边缘渐变、滑动条和前景。
    写在 super.onDrawForeground() 的下面
        如果你把绘制代码写在了 super.onDrawForeground() 的下面，绘制代码会在滑动边缘渐变、滑动条和前景之后被执行，那么绘制内容将会盖住滑动边缘渐变、滑动条和前景。
    写在 super.onDrawForeground() 的上面
        如果你把绘制代码写在了 super.onDrawForeground() 的上面，绘制内容就会在 dispatchDraw() 和  super.onDrawForeground() 之间执行，那么绘制内容会盖住子 View，但被滑动边缘渐变、滑动条以及前景盖住
draw() 总调度方法
```

自定义布局流程：

- 测量阶段：从上到下递归地调用每个 View 或者 ViewGroup 的 measure()方法，测量他们的尺寸并计算它们的位置；
- 布局阶段：从上到下递归地调用每个 View 或者 ViewGroup 的 layout() 方法，把测得的它们的尺寸和位置赋值给它们。

常用方法：

- getMeasuredWidth() 和 getMeasuredHeight() 来获取到之前的测量结果
- setMeasuredDimension() 来保存新的结果
- resolveSize() 来让子 View 的计算结果符合父 View 的限制
- onLayout() 里调用每个子 View 的 layout()，让它们保存自己的位置和尺寸

动画：

属性动画：

ViewPropertyAnimator：View.animate() 后跟 translationX() 等方法，动画会自动执行。

ObjectAnimator 使用方式：

1. 如果是自定义控件，需要添加 setXXX / getXXX 方法；
2. 用 ObjectAnimator.ofXXX() 创建 ObjectAnimator 对象；
3. 用 start() 方法执行动画。

PropertyValuesHolder 同一个动画中改变多个属性

```
PropertyValuesHolder holder1 = PropertyValuesHolder.ofFloat("scaleX", 1);
PropertyValuesHolder holder2 = PropertyValuesHolder.ofFloat("scaleY", 1);
PropertyValuesHolder holder3 = PropertyValuesHolder.ofFloat("alpha", 1);
```

```
4 ObjectAnimator animator = ObjectAnimator.ofPropertyValuesHolder(view,
  holder1, holder2, holder3)
5 animator.start();
```

PropertyValuesHolders.ofKeyframe() 把同一个属性拆分除了合并多个属性和调配多个动画，你还可以在 PropertyValuesHolder 的基础上更进一步，通过设置 Keyframe （关键帧），把同一个动画属性拆分成多个阶段。例如，你可以让一个进度增加到 100% 后再「反弹」回来。

AnimatorSet 多个动画配合执行。可以添加多个ObjectAnimator ，设置依次执行还是同时执行动画。

ValueAnimator 最基本的轮子

- ValueAnimator.ofInt（int values）
- ValueAnimator.ofFloat（float values）
- ValueAnimator.ofObject（int values）

ValueAnimator类 & ObjectAnimator 类的区别：

- ValueAnimator 类是先改变值，然后 手动赋值 给对象的属性从而实现动画；是 间接 对对象属性进行操作；ValueAnimator 类本质上是一种 改变 值 的操作机制
- ObjectAnimator类是先改变值，然后 自动赋值 给对象的属性从而实现动画；是 直接 对对象属性进行操作；可以理解为：ObjectAnimator更加智能、自动化程度更高

帧动画：一系列的静态图片依次播放

补间动画：

开发者指定动画的开始、动画的结束的"关键帧"，而动画变化的"中间帧"由系统计算，并补齐。

补间动画有四种：淡入淡出： alpha、位移：translate、缩放：scale、旋转： rotate

XML 形式补间动画：

```
1 <alpha xmlns:android="http://schemas.android.com/apk/res/android"
2                    android:duration="1000"
3                    android:fromAlpha="1.0"
4                    android:interpolator="@android:anim/accelerate_d
  ecelerate_interpolator"
5                    android:toAlpha="0.0" />
6 interpolator 代表插值器，主要作用是可以控制动画的变化速率，可以通过 @android:an
  im 来选择不同的插值器。
7
8 Java代码加载：
9     final Animation anim = AnimationUtils.loadAnimation(this,R.anim.
  tween_anim);
10    anim.setFillAfter(true); //设置动画结束后保留结束状态
11    image.startAnimation(anim);
```

Java 代码实现补间动画：TranslateAnimation、ScaleAnimation、RotateAnimation、AlphaAnimation、AnimationSet

自定义补间动画：

```
需要继承 Animation。继承 Animation 类关键是要重写一个方法：
applyTransformation(float interpolatedTime,Transformation t)
interploatedTime: 代表了动画的时间进行比。不管动画实际的持续时间如何，当动画播放时，该参数总是从0到 1。
Transformation t:该参数代表了补间动画在不同时刻对图形或组件的变形程度。
在实现自定义动画的关键就是重写 applyTransformation 方法时 根据 interpolatedTime 时间来动态地计算动画对图片或视图的变形程度。
```

IdleHandler一些面试问题

Q：IdleHandler 有什么用?

1. IdleHandler 是 Handler 提供的一种在消息队列空闲时，执行任务的时机；
2. 当 MessageQueue 当前没有立即需要处理的消息时，会执行 IdleHandler；

Q：MessageQueue 提供了 add/remove IdleHandler 的方法，是否需要成对使用?

1. 不是必须；
2. IdleHandler.queueIdle() 的返回值，可以移除加入 MessageQueue 的 IdleHandler；

Q：当 mIdleHanders 一直不为空时，为什么不会进入死循环?

1. 只有在 pendingIdleHandlerCount 为 -1 时，才会尝试执行 mIdleHander；
2. pendingIdlehanderCount 在 next() 中初始时为 -1，执行一遍后被置为 0，所以不会重复执行；

Q：是否可以将一些不重要的启动服务，搬移到 IdleHandler 中去处理?

1. 不建议；
2. IdleHandler 的处理时机不可控，如果 MessageQueue 一直有待处理的消息，那么 IdleHander 的执行时机会很靠后；

Q：IdleHandler 的 queueIdle() 运行在那个线程?

1. 陷进问题，queueIdle() 运行的线程，只和当前 MessageQueue 的 Looper 所在的线程有关；
2. 子线程一样可以构造 Looper，并添加 IdleHandler；

Q：既然 IdleHandler 主要是在 MessageQueue 出现空闲的时候被执行，那么何时出现空闲?

MessageQueue 是一个基于消息触发时间的优先级队列，所以队列出现空闲存在两种场景。

1. MessageQueue 为空，没有消息；
2. MessageQueue 中最近需要处理的消息，是一个延迟消息（when>currentTime），需要滞后执行；

这两个场景，都会尝试执行 IdleHandler。

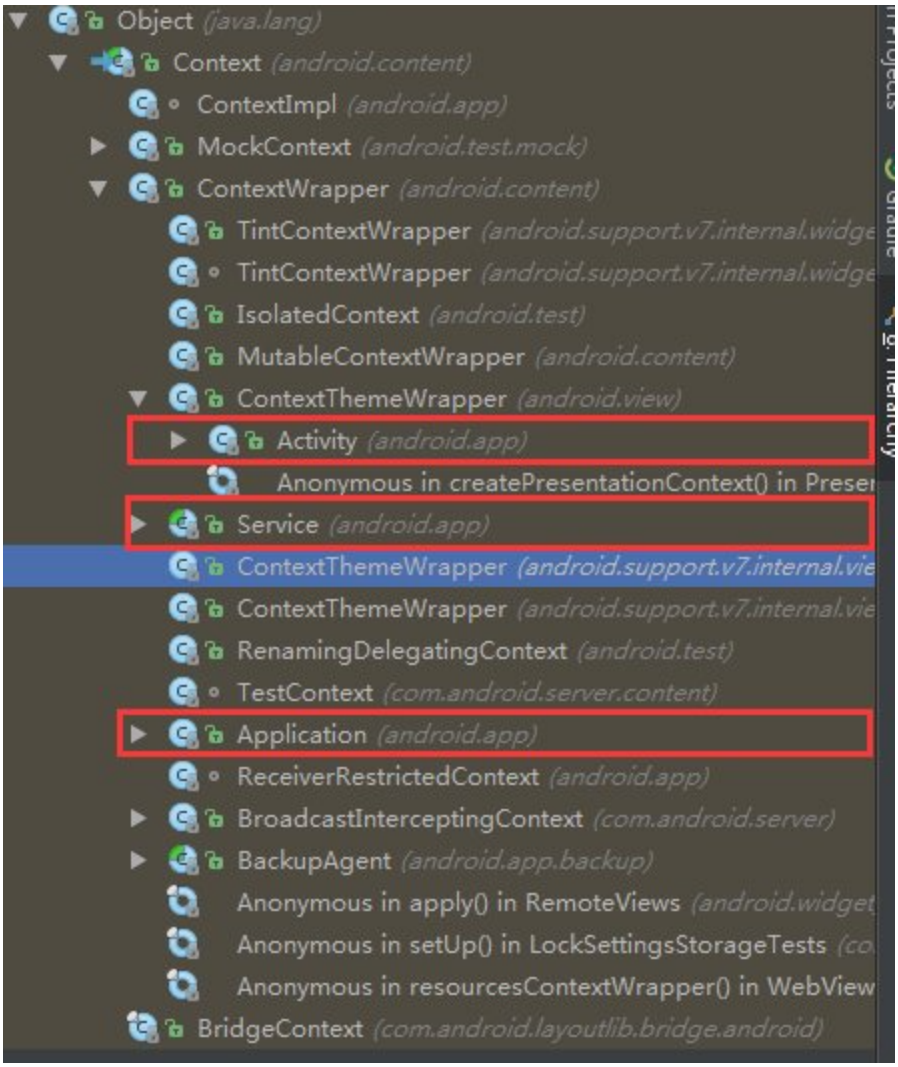
Q： IdleHandler 一次最多会执行几个，执行后是否消失?

MessageQueue#next()源码中可以看到，为了不妨碍主线程运行，即便在空闲时执行IdleHandler，也每次只会执行4个。若IdleHandler#queueIdle()方法返回false，说明此IdleHandler是一次性的，则执行后会将该IdleHandler移出**mIdleHandlers**数组，若为true则会保留，以备下一次主线程空闲时继续执行。

Application中的Context和Activity中的Context的区别

Context是什么

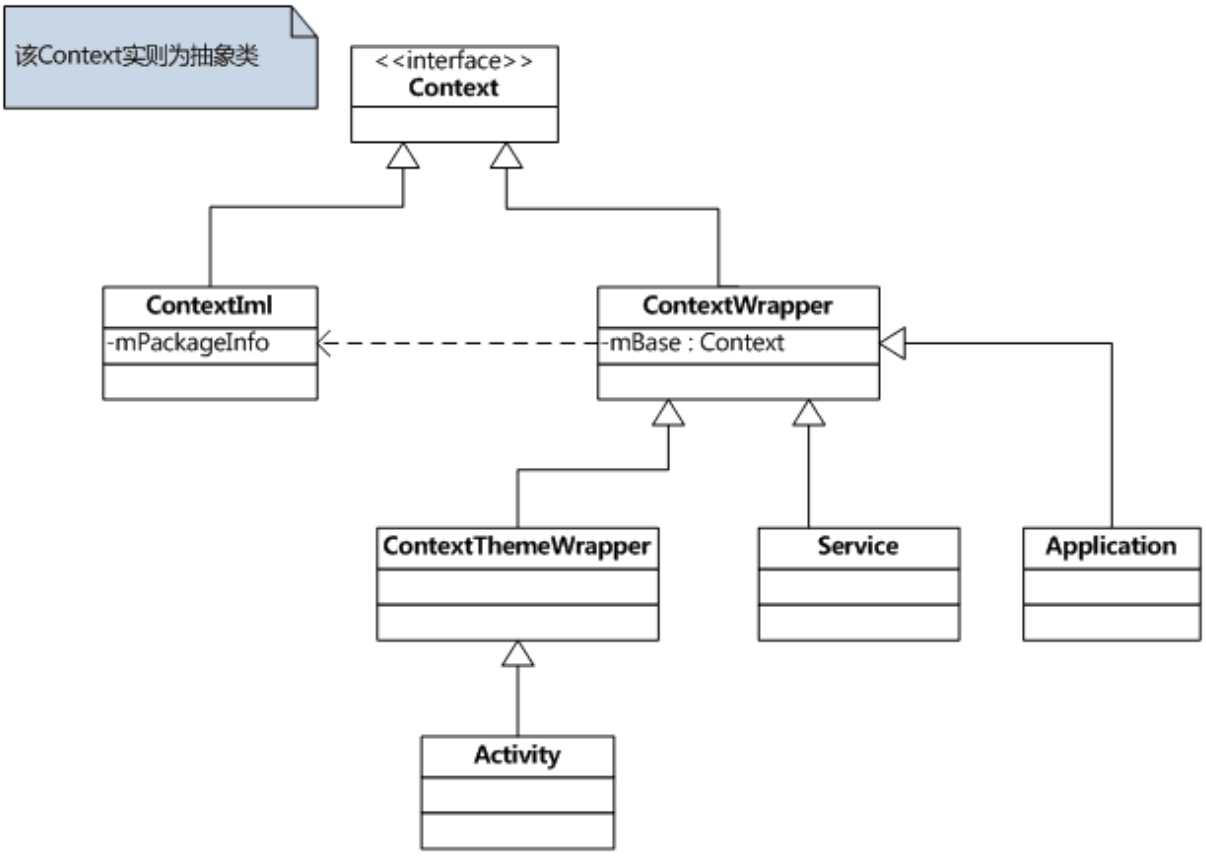
Context是维持Android程序中各组件能够正常工作的一个核心功能类，我们选中Context类，按下快捷键F4，右边就会出现一个Context类的继承结构图啦，如下图所示：



看下我用红线标出来的地方，从这里可以看到Activity、Service、Application都是Context的子类；

源码中我们可以看见Context是个抽象类，里面定义了各种抽象方法，包括获取系统资源，获取系统服务，发送广播，启动Activity,Service等，结合上图也就是说Activity、Service、Application等都是Context类的一个实现。再仔细看一下上图：Activity、Service、Application都是继承自ContextWrapper（上下文功能的封装类），而在ContextWrapper的源码中我们可以看到，ContextWrapper内部会包含一个base context（这里就不贴出来了，

大家去看源码即可），由这个base context去实现了绝大多数的方法。



通过继承关系可以看到，Context直接子类为ContextIml（具体实现类）和ContextWrapper（上下文功能包装类），而ContextWrapper又有三个子类，分别是ContextThemeWrapper、Service和Application。基于Activity和Service、Application不在一个继承层级里，而是又继承了ContextThemeWrapper。细心的朋友看到ContextWrapper和ContextThemeWrapper这两个类的名字，相信你心里已经有了答案，对的，区别在Theme。ContextThemeWrapper是一个带主题的封装类，内部包含了主题(Theme)相关的接口，当Activity在启动的时候系统都会加载一个主题，也就是我们在配置文件AndroidManifest.xml里面写的android:theme="@style/AppTheme"的属性啦！可是Service和Applicaton并不需要加载主题，因此他们继承自ContextWrapper。

Application中的Context和Activity中的Context的区别

首先Activity.this和getApplicationContext()返回的不是同一个对象，一个是当前Activity的实例，一个是项目的Application的实例，这两者的生命周期是不同的，它们各自的使用场景不同，this.getApplicationContext()取的是这个应用程序的Context，它的生命周期伴随应用程序的存在而存在；而Activity.this取的是当前Activity的Context，它的生命周期则只能存活于当前Activity，这两者的生命周期是不同的。getApplicationContext() 生命周期是整个

应用，当应用程序摧毁的时候，它才会摧毁；Activity.this的context是属于当前Activity的，当前Activity摧毁的时候，它就摧毁。

Activity Context 和Application Context两者的使用范围存在着差异，具体如下图所示：

	Application	Activity	Service	ContentProvider	BroadcastReceiver
Show a Dialog	NO	YES	NO	NO	NO
Start an Activity	NO[1]	YES	NO[1]	NO[1]	NO[1]
Layout Inflation	NO[2]	YES	NO[2]	NO[2]	NO[2]
Start a Service	YES	YES	YES	YES	YES
Bind to a Service	YES	YES	YES	YES	NO
Send a Broadcast	YES	YES	YES	YES	YES
Register BroadcastReceiver	YES	YES	YES	YES	NO[3]
Load Resource Values	YES	YES	YES	YES	YES

我们就只看Activity和Application，可以看到前三个操作不在Application中出现，也就是Show a Dialog、Start an Activity和Layout Inflation。开发的过程中，我们主要记住一点，凡是跟UI相关的，都用Activity做为Context来处理。

Context数量

在创建Activity、Service、Application时都会自动创建Context，它们各自维护着自己的上下文。在Android系统中Context类的继承结构里面我们讲到Context一共有Application、Activity和Service三种类型，因此如果要统计一个app中Context数量，我们可以这样来表示：Context数量 = Activity数量 + Service数量 + 1。

这里要解释一下，上面的1表示Application数量。一个应用程序中可以有多个Activity和多个Service，但只有一个Application。可能有人会说一个应用程序里面可以有多个Application啊，我的理解是：一个应用程序里面可以有多个Application，可是在配置文件AndroidManifest.xml中只能注册一个，只有注册的这个Application才是真正的Application，才会调用到全部的生命周期，所以Application的数量是1。

LayoutInflater.from参数Context传Activity、Application区别

```
1 LayoutInflater部分源码
2     /**
```

```
     * Obtains the LayoutInflater from the given context.
     */
    public static LayoutInflater from(Context context) {
        LayoutInflater LayoutInflater =
                (LayoutInflater) context.getSystemService(Context.LA
YOUT_INFLATER_SERVICE);
        if (LayoutInflater == null) {
            throw new AssertionError("LayoutInflater not found.");
        }
        return LayoutInflater;
    }
```

从上面代码观察，都是调用的Context.getSystemService() 下面看下getSystemService源码

1. 在Context.java中是抽象方法
2. Context的子类是ContextWrapper
3. ContextWrapper的子类ContextThemeWrapper
4. Activity继承ContextThemeWrapper，Application继承ContextWrapper

```

/**
 * ContextWrapper
 */
public class ContextWrapper extends Context {
    Context mBase;

    public ContextWrapper(Context base) {
        mBase = base;
    }

    /**
     * Set the base context for this ContextWrapper.  All calls will
then be
     * delegated to the base context.  Throws
     * IllegalStateException if a base context has already been set.
     *
     * @param base The new base context for this wrapper.
     */
    protected void attachBaseContext(Context base) {
        if (mBase != null) {
```

```
            throw new IllegalStateException("Base context already set");
        }
        mBase = base;
    }

    @Override
    public Object getSystemService(String name) {
        return mBase.getSystemService(name);
    }
}


public class ContextThemeWrapper extends ContextWrapper {
    public Object getSystemService(String name) {
        if ("layout_inflater".equals(name)) {
            if (this.mInflater == null) {
                this.mInflater = LayoutInflater.from(this.getBaseContext()).cloneInContext(this);
            }
            return this.mInflater;
        } else {
            return this.getBaseContext().getSystemService(name);
        }
    }
```

- **ContextWrapper**：mBase具体的实现类是ContextImpl，getSystemService返回的Inflater对象是PhoneLayoutInflater。
- **ContextThemeWrapper**：this.getBaseContext()返回的也是ContextImpl对象，所以LayoutInflater.from(this.getBaseContext())返回的也是PhoneLayoutInflater。 接着调用了cloneInContext(this)，设置Activity对象Clone给了Inflate对象中的Context。 而Activity.setContentView()会调用PhoneWindow.setContentView()，因为Activity继承ContextThemeWrapper，所以会解析Theme和Resource等数据。

总结： 当LayoutInflater.from(Context context) context 是 Application实例，则不会包含Theme等；而 context 是 Activity实例，会包含Theme等。

ContextThemeWrapper应用

问题：如何对单个元素设置Theme而不影响到整个Activity?

分析：该问题的实际意义是对于有些属性我们只能从Theme中获取而不能通过内联的方式嵌入在单个元素标签内，因此要想修改这个属性值，只能在Activity使用的Theme中指定。于是，该修改的作用域就会是整个Activity——这不是我们希望的！
基础概念：一般来说Activity的getTheme返回的对象就是在注册Activity时指定的Theme了（当然也有可能在运行时通过Context#setTheme改变）。由于在inflate布局文件时，默认View会从父亲节点继承Context，于是整个布局文件中的所有元素都是使用和根元素一样的Context。自然我们在Theme中定义的某些属性会影响到所有元素。

ContextThemeWrapper

ContextThemeWrapper是一个类，在Android源码中有对其简短的说明，说这是一个允许你更改或者替换Theme的Context包装。

```
// 新指定的Theme会被置于原始Theme的顶上（Top），在新的Theme中未指定的值会使用base中的值。
// 意思是这个类除了应用新的Theme外，还保留了Base Theme中的值，只不过会优先使用新的Theme中的值。
public ContextThemeWrapper(Context base, @StyleRes int themeResId) {
    super(base);
    mThemeResource = themeResId;
}
```

正如其名，它就是一个可以指定Theme的Context包装。用于在View构造时为其提供Theme属性集。

如何使用ContextThemeWrapper

代码中使用

这种方式适合手动添加View组件的情况，通过上面提供的ContextThemeWrapper构造方法生成ContextThemeWrapper，然后作为Context提供给View即可。

```
// 手动创建ContextThemeWrapper
Context context = new ContextThemeWrapper(base, resId);
// 将ContextThemeWrapper作为context提供给TextView
TextView tv = new TextView(context);
// 将ContextThemeWrapper作为context提供给LayoutInflater
// LayoutInflater.from(context).inflate(R.layout.activity_test, null);
```

布局中使用

相比起需要在代码中手动构造ContextThemeWrapper，在布局中填充的View也是有办法的。且更方便！默认情况下，View在被inflate时会使用与父节点一样的Context，既然是默认情况，那也有特殊的情况，

就是对其指定android:theme（注意！一定是带“android”命名空间的）属性，示例如下：

```
<!-- “FrameStyle”包裹在Activity的Theme之上，且对FrameLayout本身及所有子元素起作用 -->
<FrameLayout
        android:theme="@style/FrameStyle"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="20dp">

    <!-- 此处指定的“RedTextStyle” 包裹在“FrameStyle”之上 -->
    <TextView
        android:theme="@style/RedTextStyle"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="red text"/>

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textColor="hello"/>
</FrameLayout>
```

ArrayList

1. 在索引中ArrayList的增加或者删除某个对象的运行过程？效率很低吗？解释一下为什么？

   效率是很低的，因为ArrayList无论是增加或者删除某个对象，我们都要通过对数组中的元素进行移位来实现。

   - 增加元素时，我们要把要增加位置及以后的所有元素都往后移一位，先腾出一个空间，然后再进行添加。
   - 删除某个元素时，我们也要把删除位置以后的元素全部元素往前挪一位，通过覆盖的方式来删除。

   而这种移位就需要不断的arraycopy，是很耗时间的，所以效率自然也很低。
2. ArrayList如何顺序删除节点 ？ ArrayList要顺序删除节点，可以有2中方式来实现。
   - 一种是通过普通for循环的方式，但是要注意从后往前删出。否则的话会出现删不干净的问题。原因：删除节点时，把删除位置以后的元素全部元素往前挪一位，通过覆盖的方式来删除。

```
//通过一般for循环，必须从后往前删除！
for (int i=(arraylist.size()-1);i>=0;i--){
    arraylist.remove(i);
}
```

- 另一种是，通过迭代器，并调用迭代器的remove方法来删除。

```
//迭代器的方式顺序移除节点
Iterator<String> iterator = list.iterator();
while (iterator.hasNext()) {
    iterator.next();//没有这一行， 就会抛出java.lang.IllegalStateExceptio
n异常
    iterator.remove();//这里只能调用迭代器对应的remove()方法
}
```

- 注意：不调用iterator.next();这一行， 会抛出IllegalStateException异常。
  原因是：通过Iterator来删除，首先需要使用next方法迭代出集合中的元素，然后才能调用remove方法，否则集合可能抛出IllegalStateException异常。

Java中isAssignableFrom()方法与instanceof关键字区别

isAssignableFrom()方法与instanceof关键字的区别总结为以下两个点：

- isAssignableFrom()方法是从类继承的角度去判断，instanceof关键字是从实例继承的角度去判断。
- isAssignableFrom()方法是判断是否为某个类的父类，instanceof关键字是判断是否某个类的子类。

isAssignableFrom()方法的调用者和参数都是Class对象，调用者为父类，参数为本身或者其子类。
instanceof关键字两个参数，前一个为类的实例，后一个为其本身或者父类的类型。

String、StringBuffer和StringBuilder的区别

String str = new String("abc");创建了几个对象?

2个：new String(); 创建一个 "abc"：若字符串常量池中没有此字符串，则继续创建一个。
1个：new String(); 创建一个 而"abc"在字符串常量池中已存在，则直接引用。
结论：两个或者一个

String连接符"+"的底层实现原理

Java8之前：str1 + str2 === new StringBuffer(str1).append(str2);
Java8：str1 + str2 === new StringBuilder(str1).append(str2);
Java9：开始使用了invokeDynamic指令，可以动态指定要调用的方法，而不是一开始就编译好的。

String：常量，不可变，不适合用来字符串拼接，每次都是新创建的对象，消耗较大。
StringBuffer：适合用来作字符串拼接，线程安全(方法以**synchronized**来保证线程安全)
StringBuilder：JDK1.5引入，适合用来作字符串拼接，与StringBuffer区别是它不是线程安全的

Java9的改进

Java9改进了字符串（包括String、StringBuffer、StringBuilder）的实现。在Java9以前字符串采用char[]数组来保存字符，因此字符串的每个字符占2字节；而Java9的字符串采用byte[]数组再加一个

encoding-flag字段来保存字符，因此字符串的每个字符只占1字节。所以Java9的字符串更加节省空间，字符串的功能方法也没有受到影响。

设计模式

单例
工厂
策略
建造者
责任链
门面
适配器模式
观察者
代理
装饰