Context的理解：

Context 场景，上下文，应用环境全局信息的接口；

Context的分类：

Application，Activity，Service都继承自Context；Activity直接继承自ContextThemeWrapper（涉及到用户界面的展示，包含了主题相关的接口），Application，Service直接继承自ContextWrapper；

只有Activity的Context能够show dialog，除此之外他们的 Context功能一样，区别就是：Application和Service的Context在InflateLayout时使用的时应用的Theme而不是该Acativity的Theme；Application和Service的Context在启动新的Activity时，需要创建新的Task，因为他们没有任务栈；

Context导致的内存泄漏：

Context被引用后，无法释放，导致的内存泄漏。其中包括：

1.单例模式下，如果该对象持有了Context或者View（非Application Context），那么因为该单例对象的存活时间，大于Activity和Service的生命周期，导致内存泄漏；解决办法是，通过静态方法的参数直接传递Context，而不是通过构造方法传递；

2.非静态内部类（包括匿名内部类），它们会持有外部类的引用，导致外部类无法被释放，解决办法死是，使用静态内部类，如果传入了外部内引用，需要对该引用使用弱引用，并且判断null。

1. 资源未关闭，比如File，Stream，Bitmap，BroadCastReceiver,ContentObserver等，要记得不需要后关闭释放它们。
2. 异步线程导致的内存泄漏问题。

Fragment:

Fragment和Activity相互操作

Fragment对应Activity的生命周期：

|  |  |
| --- | --- |
| onAttach  onCreate  onCreateView  onActivityCreated（Activity的onCreate方法调用完成）  onStart onResume  onPause  onStop    onDestroyView  onDestroy  onDetach | onCreate  onStart  onResume  onPause  onStop  onDestory |

正常启动activity并添加fragment的顺序是：

a:onCreate -> f:onAttach,onCreate,onCreateView,onActivityCreated -> f:onStart -> a:onStart -> a:onResume ->f:onResume

Android消息机制：

ThreadLocal：同一个ThreadLocal对象，可以存储不同线程的某个值，它们互不干扰。

Looper（MessageQueue）

Handler :

线程创建Handler需要Looper，通过Looper.prepare()创建，其实是在该方法中，给ThreadLocal存储一个创建的Looper对象,而该ThreadLocal对象用于存储线程内部的变量。

IdleHandler？

HandlerThread：对Thread进行Looper封装，这样能够直接和Handler配合使用；一个子线程来处理收到的消息，所以就像IntentService一样，是串行处理的。

Service:

Service运行在后台，没有用户界面的；

本地服务，即service依附在主进程上（比如service和启动它的activity在同一个进程）；

远程服务，运行在独立的进程，不受其他进程的影响，方便为其他进程提供服务，**android:process=":service"**

。

两种service的启动方式：

1.通过startService启动，启动后service不会受启动它的组件的影响，即使组件销毁后，service依然能够存在，需要手动销毁service。

1. 通过bindService让组件绑定服务。通过提供一个类似客户端-服务器接口，使得组件和服务之间能够进行交互和通信。一个服务可以被多个组件所绑定，如果所有的组件都没有继续绑定该服务，那么它会被销毁。

|  |  |
| --- | --- |
| startService：  onCreate（只有第一次创建时回调）  onStartCommand  onDestroy  （需要通过stopService终止服务） | bindService：  onCreate（只有第一次创建时回调）  onBind （多次调用绑定服务（此时已经绑定），总共只会调用一次） onRebind  onUnbind  onDestory  （通过unbindService解绑服务） |

客户端（一般指的启动Service的组件），它通过ServiceConnection监听和获取服务端；服务端（指的Service服务），通过创建Binder，Messager，AIDL三种方式实现和客户端的链接。

服务第一次被绑定时，会调用onBInd创建并传递Binder，后面无论多次绑定，也不会调用onBind方法，因为已经存在了Binder对象，可以直接使用；

客户端（Activity）在绑定服务时，要注意其生命周期的影响，比如onCreate绑定，然后在onDestory解绑，这样该服务会在组件的整个生命周期中存在和运行；onStart绑定，onStop解绑，那么该服务只会在组件对用户可见时运行。

对于一个服务同时存在startService和bindService两种方式，startService的优先级高于bindService（或者说影响力更强大）：

比如：服务先通过bindService绑定，然后通过startService启动，此时服务会变成startService启动的状态，即使bindService的客户端解绑，也对他没有影响。同理，服务先通过startService启动在通过bindService绑定，仍然时startService状态。

Service是运行在进程的主线程当中的，和Activity一样，所以耗时操作需要另外开辟子线程处理。

前台服务，优先级更高，存活时间和几率更高，且会在通知栏下拉框中显示通知，比如音乐播放器一般使用了该方式。只需在onStartCommand时，调用startForeground（true），并设置对应通知栏即可，然后在onDestroy中使用startForegroud（false）关闭它。

QQQ。如果服务存在启动和绑定两种方式，那么分别调用stopService和unBIndService对他们有啥影响？

Service的隐式启动：Android5.0以上，不同应用之间服务需要通过隐式启动，否则两种都可以；

提高Service的存活率：

1. 启动前台服务。
2. 单独进程启用服务。
3. onStartCommand的返回值，START\_STICKY
4. 提高service的优先权

IntentService：继承自Service的，处理异步操作。

1. 包含了一个子线程，用来处理耗时操作。
2. 处理完耗时操作后，会自动关闭service。
3. 多次启用服务，该服务的子线程会串行的处理这些耗时操作。

本质其实是：继承Service后，添加了HandlerThread和Handler来处理耗时操作。

Android动画：

FrameAnimation：一系列的图片顺序播放产生的动画效果，图片过多是可能造成OOM。

TweenAnimation：对场景的view进行图像变化（平移，缩放，透明度，旋转）。

AttributeAnimation：动态改变对象属性产生的动画属性。

TweenAnimation并未改变view的属性，只是更改了绘制位置。

对于属性动画：ValueAnimator（子类ObjectAnimator），TypeEvaluator（决定了运动轨迹），TimeInterpolator（决定了行进速率），ObjectAnimator通过不断的控制属性值的变化，并不断的自动赋值给属性，每次赋值后都会调用invalidate/postInvalidate方法刷新视图。

ObjectAnimator的使用：

两个前提条件：1.该属性必须有setXxxx的方法来修改该属性，如果没有传递动画属性的初始值，那么该属性必须有getXxxx的方法，获取初始值。2.该属性值的改变，必须要产生某种特定的影响。

如果无法满足上面的条件，可以通过1.手动添加get/set方法。2.用一个新的类包装它，从而添加get/set方法。3.使用ValueAnimator监听动画的改变，自己实现属性的变化过程。

线程池ThreadPoolExecutor：

线程的创建和销毁需要付出较高的代价，通过池化技术高效的利用线程。

工作线程的数量：

0～～～核心线程数量～～～最大线程数量。

阻塞队列，用于处理工作线程已达到核心线程数量，阻塞队列会存放这些任务，如果阻塞队列被装满，则会创建新的工作线程处理这些任务，直到工作线程数量达到最大线程数量，如果工作线程数量达到最大线程数量，那么会触发饱和策略（抛异常，丢弃任务，替换任务。。。）

，如果是无界的阻塞队列，则没有限制，并且因为不会被装满，所以工作线程最大数量不会超过核心线程数量。

核心线程的身份不是由创建者决定的，而是和工作线程数量有关，如果工作线程数量大于核心线程数，则部分工作线程可能被抛弃，留下来的就是核心线程，相反如果工作线程数量小于核心线程数量，则它们都是核心线程。

线程的阻塞和非阻塞：

线程池的分类：