Android

Q：Android中进程和线程的关系?
Q：为何需要进行IPC？多进程通信可能会出现什么问题?
Q：什么是序列化？Serializable接口和Parcelable接口的区别？为何推荐使用后者?
Q：Android中为何新增Binder来作为主要的IPC方式?
Q：使用Binder进行数据传输的具体过程?
Q：Binder框架中ServiceManager的作用?
Q：Android中有哪些基于Binder的IPC方式？简单对比下?
Q：是否了解AIDL？原理是什么？如何优化多模块都使用AIDL的情况?

View

Q：MotionEvent是什么？包含几种事件？什么条件下会产生?
Q：scrollTo()和scrollBy()的区别?
Q：Scroller中最重要的两个方法是什么？主要目的是?
Q：谈一谈View的事件分发机制?
Q：如何解决View的滑动冲突?
Q：谈一谈View的工作原理?
Q：MeasureSpec是什么？有什么作用?
Q：自定义View/ViewGroup需要注意什么?
Q：onTouch()、onTouchEvent()和onClick()关系?
Q：SurfaceView和View的区别?
Q：invalidate()和postInvalidate()的区别?
Q：了解哪些Drawable？适用场景?
Q：mipmap系列中xxxhdpi、xxhdpi、xhdpi、hdpi、mdpi和ldpi存在怎样的关系?
Q：dp、dpi、px的区别?
Q：res目录和assets目录的区别?

Animation

Q：Android中有哪几种类型的动画?
Q：帧动画在使用时需要注意什么?
Q：View动画和属性动画的区别?
Q：View动画为何不能真正改变View的位置？而属性动画为何可以?
Q：属性动画插值器和估值器的作用?

Window

Q：Activity、View、Window三者之间的关系?
Q：Window有哪几种类型?
Q：Activity创建和Dialog创建过程的异同?

Handler

Q：谈谈消息机制Hander？作用？有哪些要素？流程是怎样的?
Q：为什么系统不建议在子线程访问UI?
Q：一个Thread可以有几个Looper？几个Handler?
Q：如何将一个Thread线程变成Looper线程？Looper线程有哪些特点?
Q：可以在子线程直接new一个Handler吗？那该怎么做?
Q：这里的ThreadLocal有什么作用?
Q：主线程中Looper的轮询死循环为何没有阻塞主线程?
Q：使用Hanlder的postDealy()后消息队列会发生什么变化?

线程

Q：Android中还了解哪些方便线程切换的类?
Q：AsyncTask相比Handler有什么优点？不足呢?
Q：使用AsyncTask需要注意什么?
Q：AsyncTask中使用的线程池大小?
Q：HandlerThread有什么特点?
Q：快速实现子线程使用Handler
Q：IntentService的特点?
Q：为何不用bindService方式创建IntentService?
Q：线程池的好处、原理、类型?

Activity

Q：说下Activity的生命周期?

> - 技术点：Activity生命周期
> - 思路：分条解释Activity从创建到销毁整个生命周期中涉及到的方法及作用
> - 参考回答：在Activity的生命周期涉及到七大方法，分别是：
>   - onCreate()表示Activity **正在创建**，常做**初始化**工作，如setContentView界面资源、初始化数据
>   - onStart()表示Activity **正在启动**，这时Activity **可见但不在前台**，无法和用户交互
>   - onResume()表示Activity **获得焦点**，此时Activity **可见且在前台**并开始活动
>   - onPause()表示Activity **正在停止**，可做 **数据存储、停止动画**等操作
>   - onStop()表示activity **即将停止**，可做**稍微重量级回收工作**，如取消网络连接、注销广播接收器等
>   - onDestroy()表示Activity **即将销毁**，常做**回收工作、资源释放**
>   - 另外，当Activity由后台切换到前台，由不可见到可见时会调用onRestart()，表示Activity **重新启动**

Q：onStart()和onResume()/onPause()和onStop()的区别?

> - 技术点：Activity生命周期
> - l思路：可从使用细节出发
> - 参考回答：onStart()与onStop()是从Activity是否可见这个角度调用的，onResume()和onPause()是从Activity是否显示在前台这个角度来回调的，在实际使用没其他明显区别。

Q：Activity A启动另一个Activity B会回调哪些方法？如果Activity B是完全透明呢？如果启动的是一个对话框Activity呢?

> - 技术点：Activity生命周期
> - 参考回答：Activity A启动另一个Activity B会回调的方法：Activity A的onPause() -->Activity B的onCreate()-->onStart()-->onResume()-->Activity A的onStop()；如果Activity B是完全透明的，则最后不会调用Activity A的onStop()；如果是对话框Activity，同后种情况。

Q：谈谈onSaveInstanceState()方法？何时会调用?

> - 技术点：Activity重建
> - 思路：可从使用场景、调用次序上解释
> - 参考回答：当非人为终止Activity时，比如系统配置发生改变时导致Activity被杀死并重新创建、资源内存不足导致低优先级的Activity被杀死，会调用 **onSavaInstanceState()** 来保存状态。该方法调用在

onStop之前，但和onPause没有时序关系。
- 引申：可再谈谈它和onRestoreInstanceState()的关系以及后者的作用

Q：onSaveInstanceState()与onPause()的区别?

- 技术点：Activity重建
- 思路：可从适用场景出发
- 参考回答：onSaveInstanceState()适用于对临时性状态的保存，而onPause()适用于对数据的持久化保存。

Q：如何避免配置改变时Activity重建?

- 技术点：Activity重建
- 思路：一种解决思路是设置配置文件中Activity的configChanges属性
- 参考回答：为了避免由于配置改变导致Activity重建，可在AndroidManifest.xml中对应的Activity中设置**android:configChanges="orientation|screenSize"**。此时再次旋转屏幕时，该Activity不会被系统杀死和重建，只会调用onConfigurationChanged。因此，当配置程序需要响应配置改变，指定**configChanges属性**，重写onConfigurationChanged方法即可。

Q：优先级低的Activity在内存不足被回收后怎样做可以恢复到销毁前状态?

- 技术点：Activity重建
- 思路：可从Activity重建会被调用的方法出发
- 参考回答：优先级低的Activity在内存不足被回收后重新打开会引发Activity重建。Activity被重新创建时会调用**onRestoreInstanceState**（该方法在onStart之后），并将onSavaInstanceState保存的**Bundle**对象作为参数传到onRestoreInstanceState与onCreate方法。因此可通过onRestoreInstanceState(Bundle savedInstanceState)和onCreate((Bundle savedInstanceState)来判断Activity是否被重建，并取出数据进行恢复。但需要注意的是，在onCreate取出数据时一定要先判断savedInstanceState是否为空。另外，谷歌更推荐使用onRestoreInstanceState进行数据恢复。

Q：说下Activity的四种启动模式？（有时会出个实际问题来分析返回栈中Activity的情况）

- 技术点：Activity启动模式
- 思路：分条解释四个启动模式的特点
- 参考回答：
  - standard标准模式：每次启动一个Activity就会创建一个新的实例
  - singleTop栈顶复用模式：如果新Activity已经位于任务栈的栈顶，就不会重新创建，并回调 **onNewIntent(intent)** 方法
  - singleTask栈内复用模式：只要该Activity在一个任务栈中存在，都不会重新创建，并回调 **onNewIntent(intent)** 方法。如果不存在，系统会先寻找是否存在需要的栈，如果不存在该栈，就创建一个任务栈，并把该Activity放进去；如果存在，就会创建到已经存在的栈中
  - singleInstance单实例模式：具有此模式的Activity只能单独位于一个任务栈中，且此任务栈中只有**唯一**一个实例

Q：谈谈singleTop和singleTask的区别以及应用场景

- 技术点：Activity启动模式
- 思路：可先解释两个启动模式的含义（见上一个问题），再总结不同点，最后给出应用实例
- 参考回答：singleTop和singleTask的含义分别是……，可见两者大致区别有：

- singleTop：同个Activity实例在栈中可以有多个，即可能重复创建；该模式的Activity会默认进入启动它所属的任务栈，即不会引起任务栈的变更；为防止快速点击时多次startActivity，可以将目标Activity设置为singleTop
- singleTask：同个Activity实例在栈中只有一个，即不存在重复创建；可通过android：taskAffinity设定该Activity需要的任务栈，即可能会引起任务栈的变更；常用于主页和登陆页

Q：onNewIntent()调用时机?

- 技术点：Activity启动模式
- 参考回答：启动模式为singleTop或singleTask的Activity在以下情况会回调onNewIntent()：
  - singleTop：如果新Activity已经位于任务栈的栈顶，就不会重新创建，并回调 **onNewIntent(intent)** 方法
  - singleTask：只要该Activity在一个任务栈中存在，都不会重新创建，并回调 **onNewIntent(intent)** 方法

Q：了解哪些Activity启动模式的标记位?

- 技术点：Activity启动模式
- 参考回答：常见的两个标记为：
  - **FLAG_ACTIVITY_SINGLE_TOP**:对应singleTop启动模式
  - **FLAG_ACTIVITY_NEW_TASK** ： 对应singleTask模式
- 引申：可再谈谈singleTop和singleTask具体效果

Q：如何启动其他应用的Activity?

- 技术点：Activity启动、IntentFilter匹配
- 思路：可从隐式Intent角度出发
- 参考回答：在保证有权限访问的情况下，通过隐式Intent进行目标Activity的IntentFilter匹配，原则是：
  - 一个intent只有**同时**匹配某个Activity的intent-filter中的action、category、data才算**完全匹配**，才能启动该Activity。
  - 一个Activity可以有**多个** intent-filter，一个 intent只要成功匹配**任意一组** intent-filter，就可以启动该Activity。
- 引申：如有必要可展开说明action、category、data的具体匹配规则

Q：Activity的启动过程?

- 技术点：Activity启动、ActivityManagerServie、ApplicationThread
- 思路：可大致介绍Activity启动过程涉及到的类，尤其是ActivityManagerServie、ApplicationThread从中发挥的作用。详见要点提炼|开发艺术之四大组件
- 参考回答：调用startActivity()后经过重重方法会转移到ActivityManagerService的startActivity()，并通过一个IPC回到ActivityThread的内部类**ApplicationThread**中，并调用其scheduleLaunchActivity()将启动Activity的消息发送并交由**Handler H**处理。Handler H对消息的处理会调用handleLaunchActivity()->performLaunchActivity()得以完成Activity对象的创建和启动。
- 引申：由于ActivityManagerService是一个Binder对象，可引申谈谈Binder机制

Fragment

Q：谈一谈Fragment的生命周期?

- 技术点：Fragment生命周期

- 思路：分条解释Fragment从创建到销毁整个生命周期中涉及到的方法及作用
- 参考回答：Fragment从创建到销毁整个生命周期中涉及到的方法依次为：onAttach()->onCreate()->onCreateView()->onActivityCreated()->onStart()->onResume()->onPause()->onStop()->onDestroyView()->onDestroy()->onDetach()，其中和Activity有不少名称相同作用相似的方法，而不同的方法有：
  - **onAttach()**：当Fragment和Activity建立关联时调用
  - **onCreateView()**：当Fragment创建视图时调用
  - **onActivityCreated()**：当与Fragment相关联的Activity完成onCreate()之后调用
  - **onDestroyView()**：在Fragment中的布局被移除时调用
  - **onDetach()**：当Fragment和Activity解除关联时调用

Q：Activity和Fragment的异同?

- 技术点：Fragment作用
- 思路：分别解释“异”“同”
- 参考回答：
  - Activity和Fragment的相似点在于，它们都可包含布局、可有自己的生命周期，Fragment可看似迷你活动。
  - 不同点是，由于Fragment是依附在Activity上的，多了些和宿主Activity相关的生命周期方法，如onAttach()、onActivityCreated()、onDetach()；另外，Fragment的生命周期方法是由宿主Activity而不是操作系统调用的，Activity中生命周期方法都是protected，而Fragment都是public，也能印证了这一点，因为Activity需要调用Fragment那些方法并管理它。
- 引申：可具体谈谈Activity和Fragment的关系

Q：Activity和Fragment的关系?

- 技术点：Fragment作用
- 思路：可从Fragment出现的目的、两者数量关系、调用关系展开
- 参考回答：
  - 正如Fragment的名字“碎片”，它的出现是为了解决Android碎片化，它可作为Activity界面的组成部分，可在Activity运行中实现动态地加入、移除和交换。
  - 一个Activity中可同时出现多个Fragment，一个Fragment也可在多个Activity中使用。
  - 另外，Activity的**FragmentManager**负责调用队列中Fragment的生命周期方法，只要Fragment的状态与Activity的状态保持了同步，宿主Activity的FragmentManager便会继续调用其他生命周期方法以继续保持Fragment与Activity的状态一致。

Q：何时会考虑使用Fragment?

- 技术点：Fragment作用
- 思路：列举更适合使用Fragment的情况
- 参考回答：非常经典的例子，即用两个Fragment封装两个界面模块，这样只使一套代码就能适配两种设备，达到两种界面效果；单一场景切换时使用Fragment更轻量化，如ViewPager和Fragment搭配使用

Service

Q：谈一谈Service的生命周期?

- 技术点：Service生命周期

- 思路：分条解释Service从创建到销毁整个生命周期中涉及到的方法及作用
- 参考回答：在Service的生命周期涉及到六大方法，分别是：
  - **onCreate（）**：服务第一次被创建时调用
  - **onStartComand（）**：服务启动时调用
  - **onBind（）**：服务被绑定时调用
  - **onUnBind（）**：服务被解绑时调用
  - **onDestroy（）**：服务停止时调用
- 引申：谈谈相对应的两种启动方式

Q：Service的两种启动方式？区别在哪？

- 技术点：Service生命周期
- 思路：分别解释两种启动模式如何启动和停止Service，并引起生命周期怎样的变化
- 参考回答：
  - 第一种，其他组件调用Context的 **startService()** 方法可以启动一个Service，并回调服务中的onStartCommand()。如果该服务之前还没创建，那么回调的顺序是onCreate()->onStartCommand()。服务启动了之后会一直保持运行状态，直到 **stopService()** 或 **stopSelf()** 方法被调用，服务停止并回调onDestroy()。另外，无论调用多少次startService()方法，只需调用一次stopService()或stopSelf()方法，服务就会停止了。
  - 第二种，其它组件调用Context的 **bindService()** 可以绑定一个Service，并回调服务中的onBind()方法。类似地，如果该服务之前还没创建，那么回调的顺序是onCreate()->onBind()。之后，调用方可以获取到onBind()方法里返回的IBinder对象的实例，从而实现和服务进行通信。只要调用方和服务之间的连接没有断开，服务就会一直保持运行状态，直到调用了 **unbindService()** 方法服务会停止，回调顺序onUnBind()->onDestroy()。

Q：一个Activty先start一个Service后，再bind时会回调什么方法？此时如何做才能回调Service的destory()方法？

- 技术点：Service生命周期
- 参考回答：startService()启动Service之后，再bindService()绑定，此时只会回调onBind()方法；若想回调Service的destory()方法，需要同时调用 stopService()和 unbindService()方法才能让服务销毁掉。

Q：Service如何和Activity进行通信？

- 技术点：Service信息传递
- 思路：简单介绍Service如何和Activity双向通信的流程
- 参考回答：通过bindService()可以实现Activity调用Service中的方法，具体步骤见学习笔记| AS入门（十）组件篇之Service； 通过广播实现Service向Activity发送消息
- 引申：谈谈底层的Binder机制

Q：用过哪些系统Service？

- 技术点：Service类型（系统Service）
- 参考回答：

常用系统服务

传入的Name	返回的对象	说明
WINDOW_SERVICE	WindowManager	管理打开的窗口程序
LAYOUT_INFLATER_SERVICE	LayoutInflater	取得XML里定义的View
ACTIVITY_SERVICE	ActivityManager	管理应用程序的系统状态
POWER_SERVICE	PowerManger	电源服务
ALARM_SERVICE	AlarmManager	闹钟服务
NOTIFICATION_SERVICE	NotificationManager	状态栏服务
KEYGUARD_SERVICE	KeyguardManager	键盘锁服务

Q：是否能在Service进行耗时操作？如果非要可以怎么做?

- 技术点：Service使用注意
- 参考回答：Service默认并不会运行在子线程中，也不运行在一个独立的进程中，它同样执行在主线程中（UI线程）。换句话说，不要在Service里执行耗时操作，除非手动打开一个子线程，否则有可能出现主线程被阻塞（ANR）的情况。
- 引申：可以引申谈谈开子线程的几种方法

Q：AlarmManager能实现定时的原理?

- 技术点：系统服务（后台定时）
- 思路：AlarmManager
- 参考回答：通过调用AlarmManager的 **set()** 方法就可以设置一个定时任务，并提供三个参数（工作类型，定时任务触发的时间，PendingIntent对象）。其中第三个PendingIntent对象是关键，一般会调用它的 **getBroadcast()** 方法来获取一个能够执行广播的PendingIntent。这样当定时任务被触发的时候，广播接收器的onReceive()方法就可以得到执行。即通过服务和广播的循环触发实现定时服务。
- 引申：可谈谈底层定时机制实现原理

Q：前台服务是什么？和普通服务的不同？如何去开启一个前台服务?

- 技术点：Service类型（前台Service）
- 参考回答：和一般运行在后台的服务不同，前台服务的服务状态可以被用户一眼看到。它和普通服务最大的区别是，前者会一直有一个正在运行的图标在系统的状态栏显示，下拉状态栏后可以看到更加详细的信息，非常类似于通知的效果，且当系统内存不足服务被杀死时，通知会被移除。实现一个前台服务也非常简单，和发送一个通知非常类似，只不过在构建好一个Notification之后，不需要NotificationManager将通知显示出来，而是调用了 **startForeground()** 方法。

Q：是否了解ActivityManagerService，谈谈它发挥什么作用?

- 技术点：Service类型（ActivityManagerService）
- 思路：可谈谈在四大组件创建中ActivityManagerService发挥的作用，详见[要点提炼 | 开发艺术之四大组件]

- 参考回答：ActivityManagerService是Android中最核心的服务， 主要负责系统中四大组件的启动、切换、调度及应用进程的管理和调度等工作，其职责与操作系统中的进程管理和调度模块类似。
- 引申：看源码谈谈AMS启动过程：ActivityManagerService分析——AMS启动流程

Q：如何保证Service不被杀死?

- 技术点：Service应用
- 思路：列举几种解决办法
- 参考回答：可以采取以下几种解决方法：
  - 在Service的onStartCommand()中设置flages值为START_STICKY，使得Service被杀死后尝试再次启动Service
  - 提升Service优先级，比如设置为一个前台服务
  - 在Activity的onDestroy()通过发送广播，并在广播接收器的onReceive()中启动Service

Broadcast Receiver

Q：广播有几种形式？什么特点?

- 技术点：Broadcast类型
- 思路：分条解释每种广播类型的特点
- 参考回答：常见以下四种广播：
  - 普通广播：一种**完全异步**执行的广播，在广播发出之后，所有的广播接收器几乎都会在同一时刻接收到这条广播消息，因此它们接收的先后是随机的。
  - 有序广播：一种**同步执行**的广播，在广播发出之后，同一时刻只会有一个广播接收器能够收到这条广播消息，当这个广播接收器中的逻辑执行完毕后，广播才会继续传递，所以此时的广播接收器是有先后顺序的，且优先级（priority）高的广播接收器会先收到广播消息。有序广播可以被接收器截断使得后面的接收器无法收到它。
  - 本地广播：发出的广播只能够在应用程序的**内部**进行传递，并且广播接收器也只能接收本应用程序发出的广播。
  - 粘性广播：这种广播会一直滞留，当有匹配该广播的接收器被注册后，该接收器就会收到此条广播。

Q：广播的两种注册形式？区别在哪?

- 技术点：Broadcast使用
- 参考回答：广播的注册有两种方法：一种在活动里通过代码**动态注册**，另一种在配置文件里**静态注册**。两种方式的相同点是都完成了对接收器以及它能接收的广播值这两个值的定义；不同点是动态注册的接收器必须要在程序启动之后才能接收到广播，而静态注册的接收器即便程序未启动也能接收到广播，比如想接收到手机开机完成后系统发出的广播就只能用静态注册了。

ContentProvider

Q：ContentProvider了解多少?

- 技术点：ContentProvider
- 思路：ContentProvider功能
- 参考回答：作为四大组件之一，ContentProvider主要负责存储和共享数据。与文件存储、SharedPreferences存储、SQLite数据库存储这几种数据存储方法不同的是，后者保存下的数据只能被该

应用程序使用，而前者可以让不同应用程序之间进行数据共享，它还可以选择只对哪一部分数据进行共享，从而保证程序中的隐私数据不会有泄漏风险。

- 引申：谈谈ContentProvider底层使用Binder机制原理

数据存储

Q：Android中提供哪些数据持久存储的方法?

- 技术点：数据持久化
- 思路：分条解释每种数据持久存储的特点
- 参考回答：Android平台实现数据存储的常见几种方式：
  - File 文件存储：写入和读取文件的方法和 Java中实现I/O的程序一样。
  - SharedPreferences存储：一种轻型的数据存储方式，常用来存储一些简单的配置信息，本质是基于XML文件存储key-value键值对数据。
  - SQLite数据库存储：一款轻量级的关系型数据库，它的运算速度非常快，占用资源很少，在存储大量复杂的关系型数据的时可以使用。
  - ContentProvider：四大组件之一，用于数据的存储和共享，不仅可以让不同应用程序之间进行数据共享，还可以选择只对哪一部分数据进行共享，可保证程序中的隐私数据不会有泄漏风险。

Q：Java中的I/O流读写怎么做?

- 技术点：数据持久化（文件存储）
- 思路：大致介绍核心类和核心方法
- 参考回答：和 Java中实现I/O的程序是一样的，Context类中提供了openFileInput()和openFileOutput()方法来打开数据文件里的文件IO流，有关读写涉及的主要方法详见文章学习笔记| AS入门（七） 数据存储篇

Q：SharePreferences适用情形？使用中需要注意什么?

- 技术点：数据持久化（SharePreferences存储）
- 参考回答：SharePreferences是一种轻型的数据存储方式，适用于存储一些简单的配置信息，如int、string、boolean、float和long。由于系统对SharedPreferences的读/写有一定的缓存策略，即在内存中有一份该文件的缓存，因此在多进程模式下，其读/写会变得不可靠，甚至丢失数据。
- 引申：谈谈Android中多进程通信（Binder）

Q：了解SQLite中的事务处理吗？是如何做的?

- 技术点：数据持久化（SQLite）
- 参考回答：SQLite在做CRDU操作时都默认开启了事务，然后把SQL语句翻译成对应的SQLiteStatement并调用其相应的CRUD方法，此时整个操作还是在rollback journal这个临时文件上进行，只有操作顺利完成才会更新.db数据库，否则会被回滚。
- 引申：谈谈如何模仿SQLite中事务的思想更高效进行批量操作

Q：使用SQLite做批量操作有什么好的方法吗?

- 技术点：数据持久化（SQLite）
- 思路：模仿SQLite的事务处理
- 参考回答：使用SQLiteDatabase的beginTransaction()方法开启一个事务，将批量操作SQL语句转化成SQLiteStatement并进行批量操作，结束后endTransaction()

Q：如果现在要删除SQLite中表的一个字段如何做?

- 技术点：数据持久化（SQLite）
- 参考回答：SQLite数据库只允许增加表字段而不允许修改和删除表字段，只能采取复制表思想，即创建一个新表保留原表想要的字段、再将原表删除

Q：使用SQLite时会有哪些优化操作?

- 技术点：数据持久化（SQLite）
- 思路：列举可优化点
- 参考回答：
  - 使用事务做批量操作：具体操作见上
  - 及时关闭Cursor，避免内存泄漏
  - 耗时操作异步化：数据库的操作属于本地IO，通常比较耗时，建议将这些耗时操作放入异步线程中处理
  - ContentValues的容量调整：ContentValues内部采用HashMap来存储Key-Value数据，ContentValues初始容量为8，扩容时翻倍。因此建议对ContentValues填入的内容进行估量，设置合理的初始化容量，减少不必要的内部扩容操作
  - 使用索引加快检索速度：对于查询操作量级较大、业务对要求查询要求较高的推荐使用索引

IPC

Q：Android中进程和线程的关系?

- 技术点：进程、线程
- 参考回答：
  - 形象理解：如果把安卓系统比喻成一片土壤，可以把App看做扎根在这片土壤上的工厂，每个APP一般对应一个进程，那么线程就像是工厂的生产线。其中，主线程好比是主生产线，只有一条，子线程就像是副生产线，可以有很多条。
  - 关系：一个APP一般对应一个进程和有限个线程
    - 一般对应一个进程，当然，可以在AndroidMenifest中给四大组件指定属性`android:process`开启多进程模式
    - 有限个线程：线程是一种**受限**的系统资源，不可无限制的产生且线程的创建和销毁都有一定的开销。

Q：为何需要进行IPC？多进程通信可能会出现什么问题?

- 技术点：多进程通信
- 思路：讨论多进程通信会出现的问题得出IPC的必要性
- 参考回答：
- （1）

  多进程造成的影响

  可总结为以下四方面：
  - 静态变量和单例模式失效：由独立的虚拟机造成
  - 线程同步机制失效：由独立的虚拟机造成
  - SharedPreference的不可靠下降：不支持两个进程同时进行读写操作，即不支持并发读写，有一定几率导致数据丢失

    - Application多次创建： Android系统会为新的进程分配独立虚拟机，相当于系统又把这个应用重新启动了一次。
- （2）**需要进程间通信的必要性**：所有运行在不同进程的四大组件，只要它们之间需要通过内存在共享数据，都会共享失败。这是由于Android为每个应用分配了**独立**的虚拟机，不同的虚拟机在内存分配上有不同的地址空间，这会导致在不同的虚拟机中访问同一个类的对象会产生多份副本。
- 引申： 谈谈IPC的使用场景

Q：什么是序列化？Serializable接口和Parcelable接口的区别？为何推荐使用后者?

- 技术点：序列化
- 参考回答：序列化表示将一个对象转换成

  可存储或可传输

  的状态。序列化后的对象可以在网络上进行传输，也可以存储到本地。
    - 应用场景：需要通过Intent和Binder等传输**类对象**就必须完成对象的序列化过程。
    - 两种方式：实现Serializable/Parcelable接口。不同点如图：

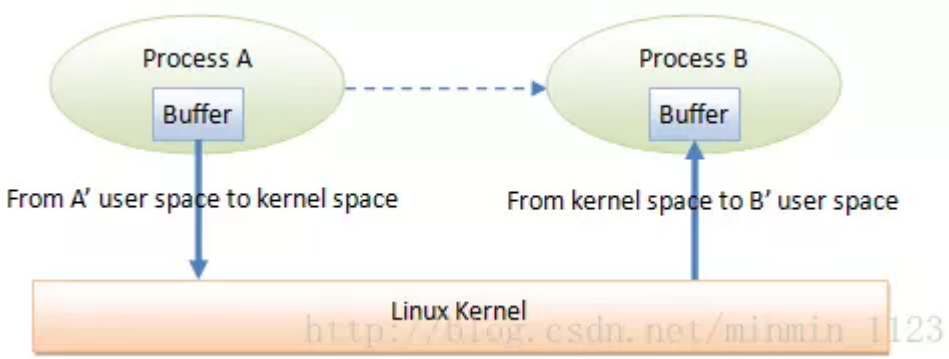
	Serializable 接口	**Parcelable** 接口
平台	**Java**的序列化接口	**Android**的序列化接口
序列化原理	将一个**对象**转换成可存储或可传输的状态。	将一个对象进行**分解**，且分解后的每一部分都是传递可支持的数据类型。
优缺点	简单但效率较低，开销大。 由于序列化（ObjectOutputStream类）和反序列化（ObjectInputStream类）过程都需要大量的I/O操作。	高效但使用较麻烦
使用场景	适合将对象序列化到**存储**设备或者将对象序列化后通过**网络**设备传输	主要用在**内存**的序列化

Q：Android中为何新增Binder来作为主要的IPC方式?

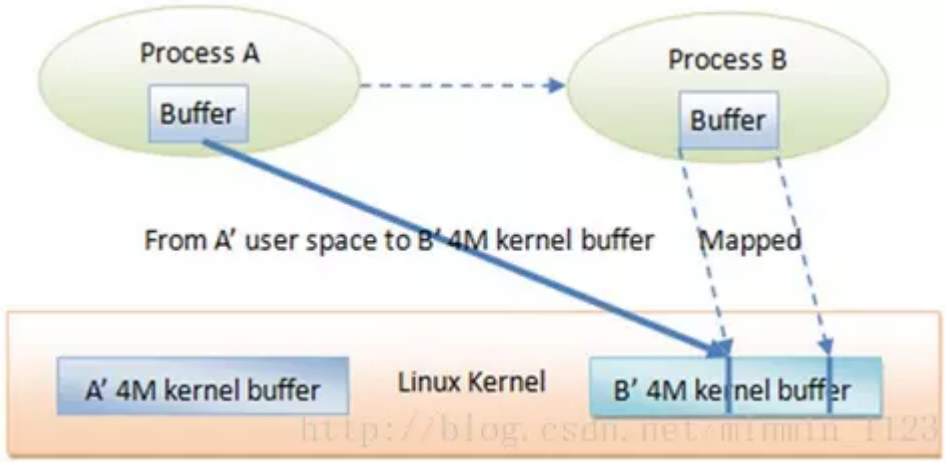
- 技术点：Binder机制
- 思路：回答Binder优点
- 参考回答：Binder机制有什么几条

  优点

  ：
    - 传输效率高、可操作性强：传输效率主要影响因素是内存拷贝的次数，拷贝次数越少，传输速率越高。从Android进程架构角度分析：对于消息队列、Socket和管道来说，数据先从发送方的缓存区拷贝到内核开辟的缓存区中，再从内核缓存区拷贝到接收方的缓存区，一共两次拷贝，如图：



而对于Binder来说，数据从发送方的缓存区拷贝到内核的缓存区，而接收方的缓存区与内核的缓存区是映射到同一块物理地址的，节省了一次数据拷贝的过程，如图：

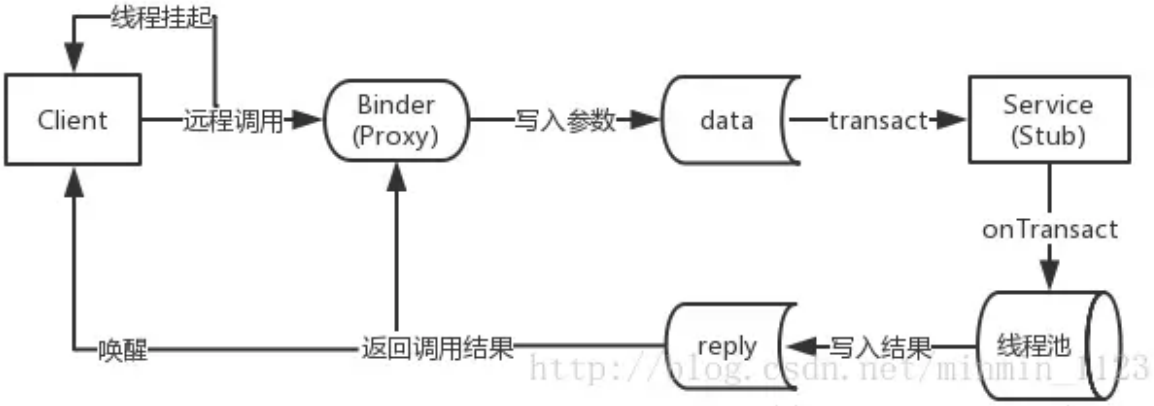


由于共享内存操作复杂，综合来看，Binder的传输效率是最好的。

- 实现C/S架构方便：Linux的众IPC方式除了Socket以外都不是基于C/S架构，而Socket主要用于网络间的通信且传输效率较低。Binder基于C/S架构 ，Server端与Client端相对独立，稳定性较好。
- 安全性高：传统Linux IPC的接收方无法获得对方进程可靠的UID/PID，从而无法鉴别对方身份；而Binder机制为每个进程分配了UID/PID且在Binder通信时会根据UID/PID进行有效性检测。

Q：使用Binder进行数据传输的具体过程?

- 技术点：Binder机制
- 思路：通过AIDL实现方式解释Binder数据传输的具体过程
- 参考回答：服务端中的Service给与其绑定的客户端提供Binder对象，客户端通过AIDL接口中的asInterface()将这个Binder对象转换为代理Proxy，并通过它发起RPC请求。客户端发起请求时会挂起当前线程，并将参数写入data然后调用transact()，RPC请求会通过系统底层封装后由服务端的onTransact()处理，并将结果写入reply，最后返回调用结果并唤醒客户端线程。

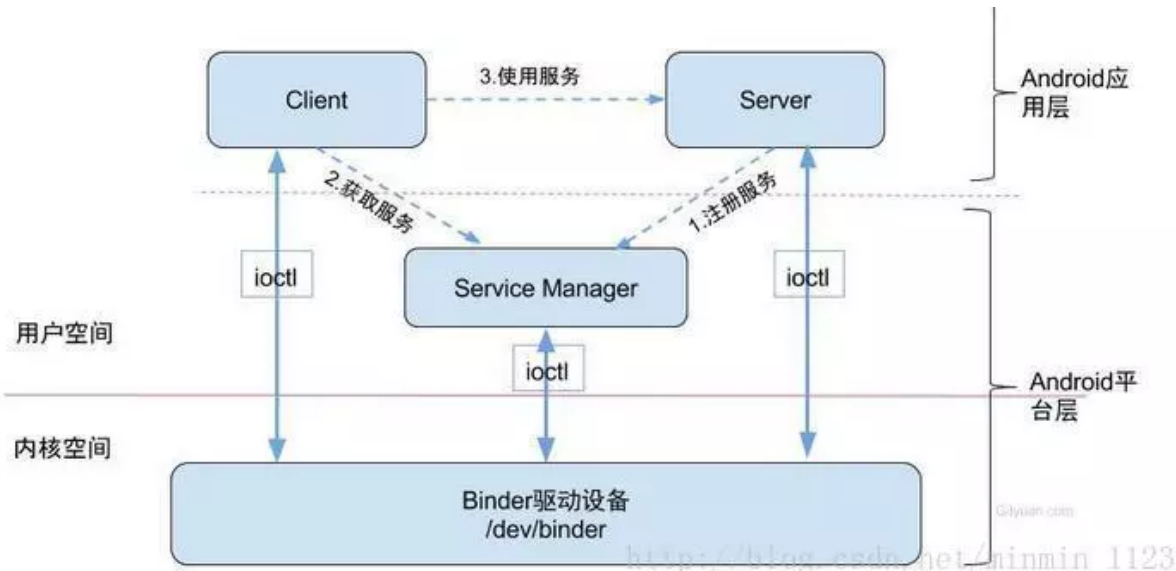


Q：Binder框架中ServiceManager的作用?

- 技术点：Binder机制
- 思路：从Binder框架出发讨论每个元素的作用
- 参考回答：在

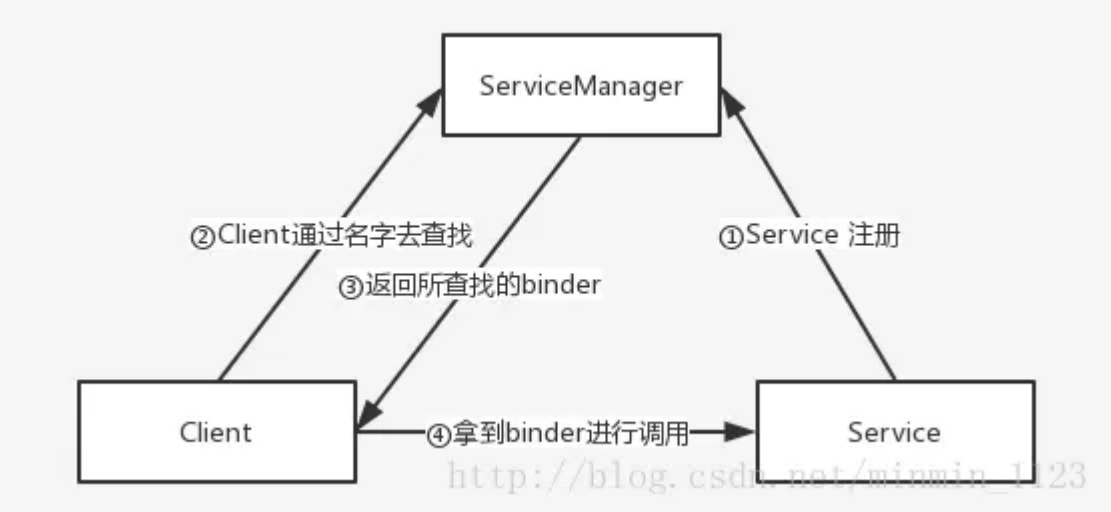
  Binder框架

  定义了四个角色：Server，Client，ServiceManager和Binder驱动。其中Server、Client、ServiceManager运行于用户空间，Binder驱动运行于内核空间。关系如图：



  - **Server**&**Client**：服务器&客户端。在Binder驱动和Service Manager提供的基础设施上，进行Client-Server之间的通信。
  - ServiceManager

    服务的管理者，将Binder名字转换为Client中对该Binder的引用，使得Client可以通过Binder名字获得Server中Binder实体的引用。流程如图：



- Binder驱动

  :

  - 与硬件设备没有关系，其工作方式与设备驱动程序是一样的，工作于内核态。
  - 提供**open()**、**mmap()**、**poll()**、**ioctl()** 等标准文件操作。
  - 以字符驱动设备中的misc设备注册在设备目录/dev下，用户通过/dev/binder访问该它。
  - 负责进程之间binder通信的建立，传递，计数管理以及数据的传递交互等底层支持。
  - 驱动和应用程序之间定义了一套接口协议，主要功能由**ioctl()** 接口实现，由于ioctl()灵活、方便且能够一次调用实现先写后读以满足同步交互，因此不必分别调用write()和read()接口。
  - 其代码位于linux目录的drivers/misc/binder.c中。

Q：Android中有哪些基于Binder的IPC方式？简单对比下?

- 技术点：IPC方式
- 思路：分析每种IPC方式的优缺点和使用场景的差异
-

  参考回答：

表 2-2　IPC 方式的优缺点和适用场景

名　称	优　点	缺　点	适 用 场 景
Bundle	简单易用	只能传输 Bundle 支持的数据类型	四大组件间的进程间通信
文件共享	简单易用	不适合高并发场景，并且无法做到进程间的即时通信	无并发访问情形，交换简单的数据实时性不高的场景
AIDL	功能强大，支持一对多并发通信，支持实时通信	使用稍复杂，需要处理好线程同步	一对多通信且有 RPC 需求
Messenger	功能一般，支持一对多串行通信，支持实时通信	不能很好处理高并发情形，不支持 RPC，数据通过 Message 进行传输，因此只能传输 Bundle 支持的数据类型	低并发的一对多即时通信，无 RPC 需求，或者无须要返回结果的 RPC 需求
ContentProvider	在数据源访问方面功能强大，支持一对多并发数据共享，可通过 Call 方法扩展其他操作	可以理解为受约束的 AIDL，主要提供数据源的 CRUD 操作	一对多的进程间的数据共享
Socket	功能强大，可以通过网络传输字节流，支持一对多并发实时通信	实现细节稍微有点烦琐，不支持直接的 RPC	网络数据交换

Q：是否了解AIDL？原理是什么？如何优化多模块都使用AIDL的情况？

- 技术点：AIDL
- 思路：
- 参考回答：
  - AIDL(Android Interface Definition Language，**Android接口定义语言**)：如果在一个进程中要调用另一个进程中对象的方法，可使用AIDL生成可序列化的参数，AIDL会生成一个服务端对象的**代理类**，通过它客户端实现间接调用服务端对象的方法。
  - AIDL的

    本质

    是系统提供了一套可快速实现Binder的工具。关键类和方法：
    - **AIDL接口**：继承**IInterface。**
    - **Stub类**：Binder的实现类，服务端通过这个类来提供服务。
    - **Proxy类**：服务器的本地代理，客户端通过这个类调用服务器的方法。
    - asInterface()

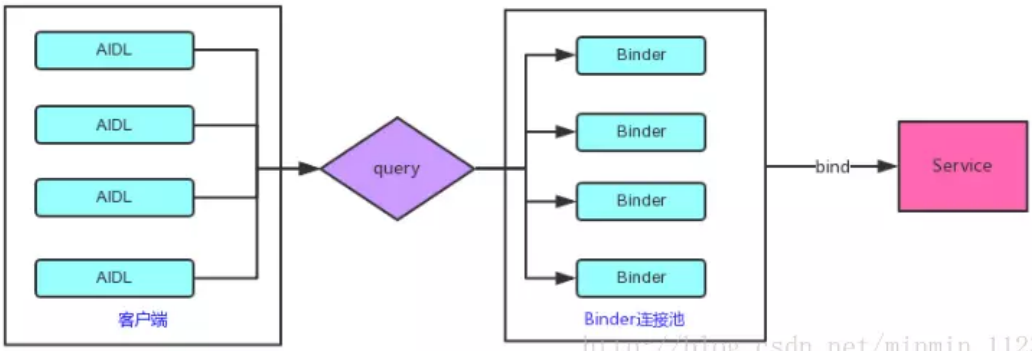
      ：客户端调用，将服务端的返回的Binder对象，转换成客户端所需要的AIDL接口类型对象。返回对象：
      - 若客户端和服务端位于同一进程，则直接返回Stub对象本身；
      - 否则，返回的是系统封装后的Stub.proxy对象。
    - **asBinder()**：根据当前调用情况返回代理Proxy的Binder对象。
    - **onTransact()**：运行服务端的Binder线程池中，当客户端发起跨进程请求时，远程请求会通过系统底层封装后交由此方法来处理。
    - **transact()**：运行在客户端，当客户端发起远程请求的同时将当前线程挂起。之后调用服务端的onTransact()直到远程请求返回，当前线程才继续执行。

- 当有多个业务模块都需要AIDL来进行IPC，此时需要为每个模块创建特定的aidl文件，那么相应的Service就会很多。必然会出现系统资源耗费严重、应用过度重量级的问题。解决办法是建立Binder连接池，即将每个业务模块的Binder请求

  统一

  转发到一个远程Service中去执行，从而避免重复创建Service。

  - 工作原理：每个业务模块创建自己的AIDL接口并实现此接口，然后向服务端提供自己的唯一标识和其对应的Binder对象。服务端只需要一个Service，服务器提供一个queryBinder接口，它会根据业务模块的特征来返回相应的Binder对像，不同的业务模块拿到所需的Binder对象后就可进行远程方法的调用了。流程如图：



View

Q：MotionEvent是什么？包含几种事件？什么条件下会产生？

- 技术点：View触控
- 参考回答：

  MotionEvent

  是手指触摸屏幕锁产生的一系列事件。包含的事件有：

  - **ACTION_DOWN**：手指刚接触屏幕
  - **ACTION_MOVE**：手指在屏幕上滑动
  - **ACTION_UP**：手指在屏幕上松开的一瞬间
  - **ACTION_CANCEL**：手指保持按下操作，并从当前控件转移到外层控件时会触发

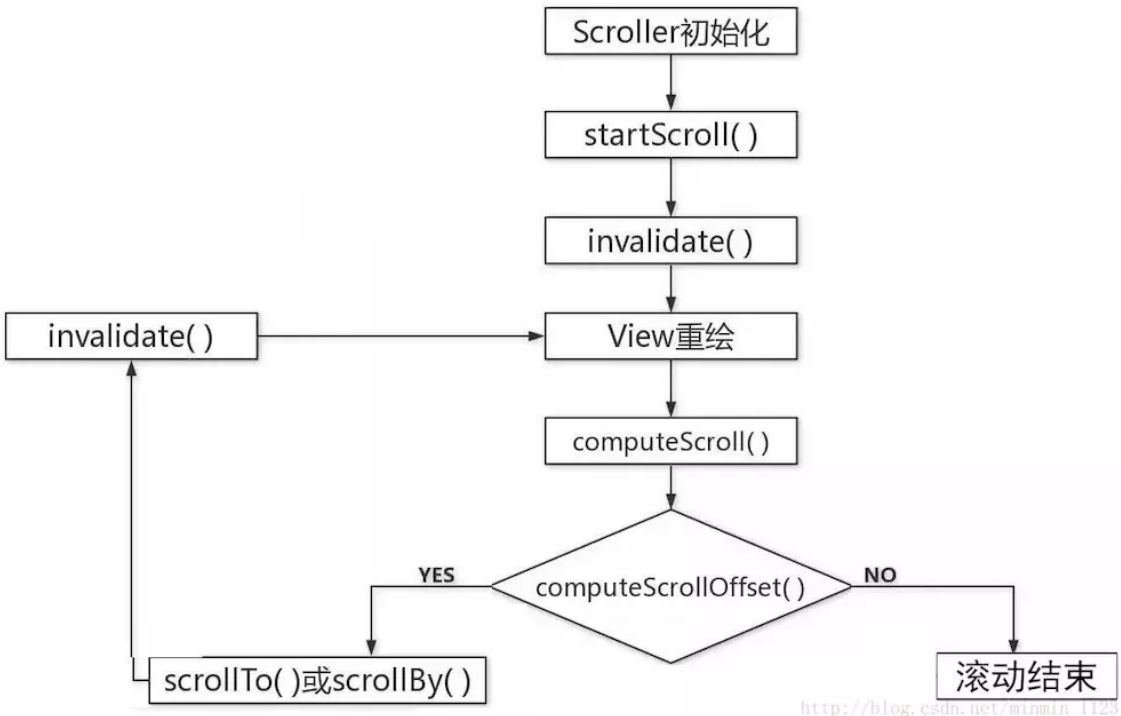
Q：scrollTo()和scrollBy()的区别？

- 技术点：View滑动
- 参考回答：scrollBy内部调用了scrollTo，它是基于当前位置的**相对滑动**；而scrollTo是**绝对滑动**，因此如果利用相同输入参数多次调用scrollTo()方法，由于View初始位置是不变只会出现一次View滚动的效果而不是多次。
- 引申：两者都只能对**view内容**进行滑动，而不能使view本身滑动，且非平滑，可使用Scroller有过渡滑动的效果

Q：Scroller中最重要的两个方法是什么？主要目的是？

- 技术点：View滑动

- 思路：从Scroller实现滑动的具体过程出发，
- 参考回答：Scroller实现滑动的具体过程：
  - 在MotionEvent.ACTION_UP事件触发时调用startScroll()方法，该方法并没有进行实际的滑动操作，而是记录滑动相关量
  - 马上调用invalidate/postInvalidate()方法，请求View重绘，导致View.draw方法被执行
  - 紧接着会调用View.computeScroll()方法，此方法是空实现，需要自己处理逻辑。具体逻辑是：先判断computeScrollOffset()，若为true（表示滚动未结束），则执行scrollTo()方法，它会再次调用postInvalidate()，如此反复执行，直到返回值为false。流程图如下：



其中，最重要的两个方法是startScroll()和computeScroll()

Q：谈一谈View的事件分发机制?

- 技术点：View事件分发
- 思路：从分发本质、传递顺序、核心方法展开
- 参考回答：
  - 事件分发本质：就是对**MotionEvent事件**分发的过程。即当一个MotionEvent产生了以后，系统需要将这个点击事件传递到一个具体的View上。
  - 点击事件的传递顺序：**Activity（Window） -> ViewGroup -> View**
  - 三个主要方法：
    - **dispatchTouchEvent**：进行事件的分发（传递）。返回值是 boolean 类型，受当前onTouchEvent和下级view的dispatchTouchEvent影响
    - **onInterceptTouchEvent**：对事件进行拦截。该方法只在ViewGroup中有，View（不包含ViewGroup）是没有的。一旦拦截，则执行ViewGroup的onTouchEvent，在ViewGroup中处理事件，而不接着分发给View。且只调用一次，所以后面的事件都会交给ViewGroup处理。

        - **onTouchEvent**：进行事件处理。

Q：如何解决View的滑动冲突?

- 技术点：View滑动冲突
- 思路：从处理规则和具体实现方法展开讨论
- 参考回答：
- （1）处理规则：
    - 对于由于外部滑动和内部滑动**方向不一致**导致的滑动冲突，可以根据滑动的方向判断谁来拦截事件。
    - 对于由于外部滑动方向和内部滑动**方向一致**导致的滑动冲突，可以根据业务需求，规定何时让外部View拦截事件何时由内部View拦截事件。
    - 对于上面两种情况的嵌套，相对复杂，可同样根据需求在业务上找到突破点。
- （2）实现方法：
    - **外部拦截法**：指点击事件都先经过**父容器的拦截处理**，如果父容器需要此事件就拦截，否则就不拦截。具体方法：需要重写父容器的**onInterceptTouchEvent**方法，在内部做出相应的拦截。
    - **内部拦截法**：指父容器不拦截任何事件，而将所有的事件都传递给子容器，如果子容器需要此事件就直接消耗，否则就交由父容器进行处理。具体方法：需要配合**requestDisallowInterceptTouchEvent**方法。

Q：谈一谈View的工作原理?

- 技术点：View工作流程
- 思路：围绕三大流程展开
- 参考回答：View工作流程简单来说就是，先measure测量，用于确定View的

  测量宽高

  ，再 layout布局，用于确定View的

  最终宽高

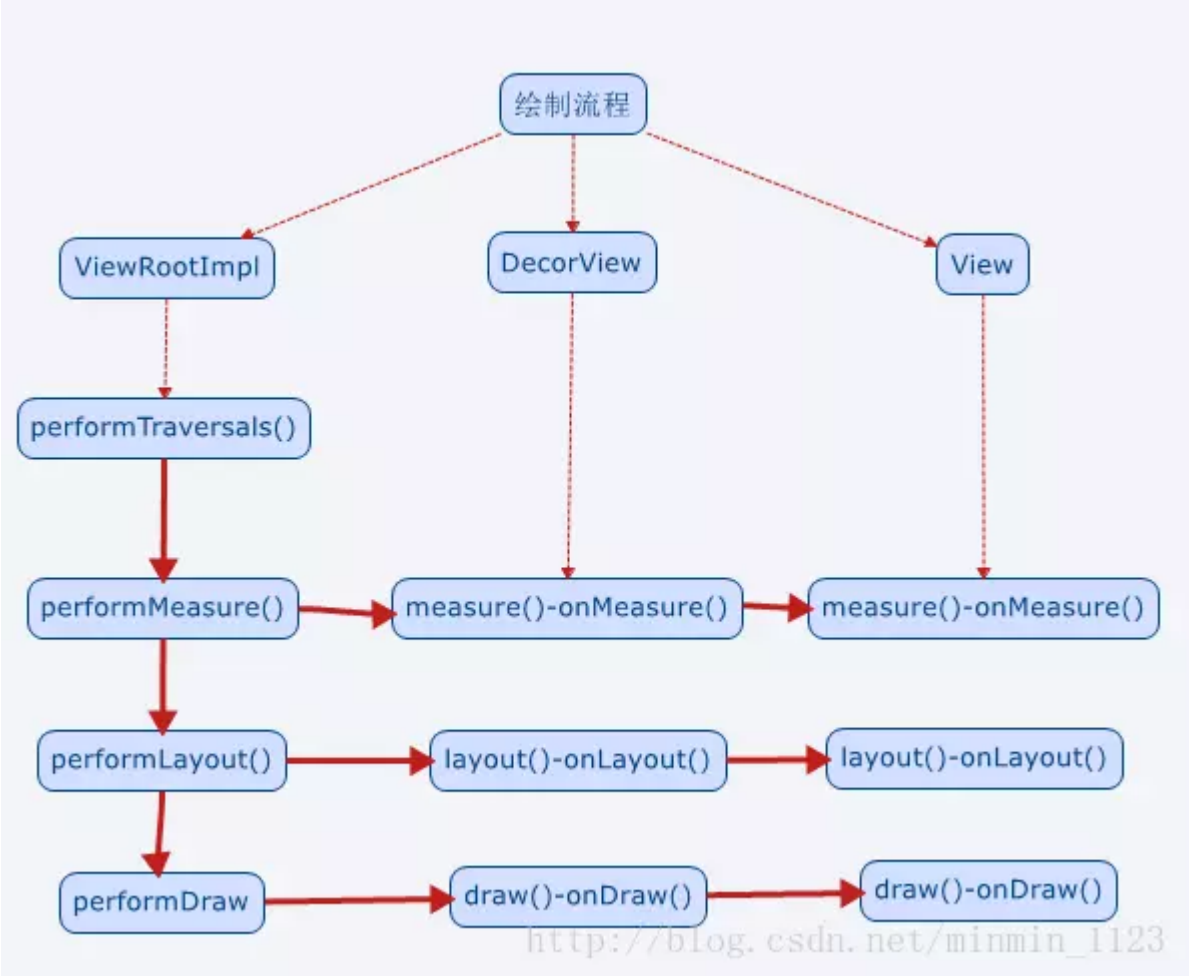
  和

  四个顶点的位置

  ，最后 draw绘制，用于将View

  绘制到屏幕

  上。具体过程图见：



- ViewRoot对应于ViewRootImpl类，它是连接WindowManager和DecorView的纽带。
- View的绘制流程是从**ViewRoot**和**performTraversals**开始。
- performTraversals()依次调用performMeasure()、performLayout()和performDraw()三个方法，分别完成**顶级** View的绘制。
- 其中，performMeasure()会调用measure()，measure()中又调用onMeasure()，实现对其所有子元素的measure过程，这样就完成了一次measure过程；接着子元素会重复父容器的measure过程，如此反复至完成整个View树的遍历。layout和draw同理。

Q：MeasureSpec是什么？有什么作用?

- 技术点：View工作流程（measure）
- 思路：从MeasureSpec作用、组成、模式和决定因素展开
- 参考回答：
  - 作用：通过宽测量值**widthMeasureSpec**和高测量值**heightMeasureSpec**决定View的大小
  - 组成：一个32位int值，高2位代表**SpecMode**(测量模式)，低30位代表**SpecSize**(某种测量模式下的规格大小)。
  - 三种模式：
    - **UNSPECIFIED**：父容器不对View有任何限制，要多大有多大。常用于系统内部。
    - **EXACTLY**(精确模式)：父视图为子视图指定一个确切的尺寸SpecSize。对应LyaoutParams中的**match_parent**或**具体数值**。

        - **AT_MOST**(最大模式)：父容器为子视图指定一个最大尺寸SpecSize，View的大小不能大于这个值。对应LayoutParams中的**wrap_content**。
    - 决定因素：值由子

      View的布局参数LayoutParams

      和

      父容器的MeasureSpec值

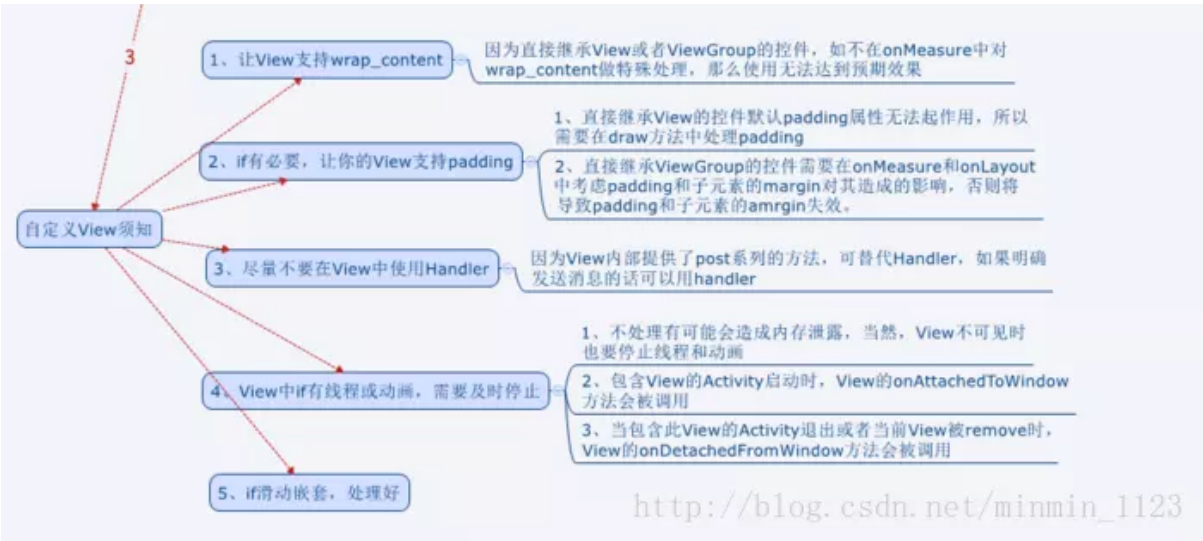
      共同决定。具体规则见下图：

父视图测量模式 (mode) / 子视图布局参数 (LayoutParams)	EXACTLY	AT_MOST	UNSPECIFIED
具体数值（dp / px）	EXACTLY + childSize	EXACTLY + childSize	EXACTLY + childSize
match_parent	EXACTLY + parentSize (父容器的剩余空间)	AT_MOST + parentSize (大小不超过父容器的剩余空间)	UNSPECIFIED + 0
wrap_content	AT_MOST + parentSize (大小不超过父容器的剩余空间)	AT_MOST + parentSize (大小不超过父容器的剩余空间)	UNSPECIFIED + 0

- 引申：直接继承View的自定义View需要重写onMeasure()并设置wrap_content时的自身大小，否则效果相当于macth_parent

Q：自定义View/ViewGroup需要注意什么?

- 技术点：自定义View
- 参考回答：



Q：onTouch()、onTouchEvent()和onClick()关系?

- 技术点：View事件分发

> - 参考回答：优先度onTouch()>onTouchEvent()>onClick()。因此onTouchListener的onTouch()方法会先触发；如果onTouch()返回false才会接着触发onTouchEvent()，同样的，内置诸如onClick()事件的实现等等都基于onTouchEvent()；如果onTouch()返回true，这些事件将不会被触发。
> - 引申：OnTouchListener、OnClickListener的冲突

Q：SurfaceView和View的区别?

> - 技术点：View、SurfaceView
> - 参考回答：SurfaceView是从View基类中派生出来的显示类，他和View的区别有：
>   - View需要在UI线程对画面进行刷新，而SurfaceView可在子线程进行页面的刷新
>   - View适用于主动更新的情况，而SurfaceView适用于被动更新，如频繁刷新，这是因为如果使用View频繁刷新会阻塞主线程，导致界面卡顿
>   - SurfaceView在底层已实现双缓冲机制，而View没有，因此SurfaceView更适用于需要频繁刷新、刷新时数据处理量很大的页面

Q：invalidate()和postInvalidate()的区别?

> - 技术点：View刷新
> - 参考回答：invalidate()与postInvalidate()都用于刷新View，主要区别是invalidate()在主线程中调用，若在子线程中使用需要配合handler；而postInvalidate()可在子线程中直接调用。

- Drawable等资源

Q：了解哪些Drawable？适用场景?

> - 技术点：res资源
> - 参考回答：BitmapDrawable表示一张图片、NinePatchDrawable可自动地根据所需的宽/高对图片进行相应的缩放并保证不失真、ShapeDrawable表示纯色、有渐变效果的基础几何图形、StateListDrawable表示一个Drawable的集合且每个Drawable对应着View的一种状态、LayerDrawable可通过将不同的Drawable放置在不同的层上面从而达到一种叠加后的效果

Q：mipmap系列中xxxhdpi、xxhdpi、xhdpi、hdpi、mdpi和ldpi存在怎样的关系?

> - 技术点：res资源
> - 参考回答：表示不同密度的图片资源，像素从高到低依次排序为xxxhdpi>xxhdpi>xhdpi>hdpi>mdpi>ldpi，根据手机的dpi不同加载不同密度的图片

Q：dp、dpi、px的区别?

> - 技术点：Android适配
> - 参考回答：
>   - px：像素，如分辨率1920x1080表示高为1920个像素、宽为1080个像素
>   - dpi：每英寸的像素点，如分辨率为1920x1080的手机尺寸为4.95英寸，则该手机DPI为（1920x1920+ 1080x1080）½/4.95≈445dpi
>   - dp：密度无关像素，是个相对值

Q：res目录和assets目录的区别?

> - 技术点：res、assets
> - 参考回答：
>   - res/raw中的文件会被映射到R.java文件中，访问时可直接使用资源ID，不可以有目录结构

- assets文件夹下的文件不会被映射到R.java中，访问时需要AssetManager类，可以创建子文件夹

Animation

Q：Android中有哪几种类型的动画?

- 技术点：动画类型
- 参考回答： 常见三类动画
  - View动画（View Animation）/补间动画（Tween animation）：对View进行平移、缩放、旋转和透明度变化的动画，不能真正的改变view的位置。应用如布局动画、Activity切换动画
  - 逐帧动画（Drawable Animation）：是View动画的一种，它会按照顺序播放一组预先定义好的图片
  - 属性动画（Property Animation）：对该类对象进行动画操作，真正改变了对象的属性

Q：帧动画在使用时需要注意什么?

- 技术点：帧动画
- 参考回答：使用祯动画要注意不能使用尺寸过大的图片，否则容易造成OOM

Q：View动画和属性动画的区别?

- 技术点：View动画、属性动画
-

  参考回答：

	View 动画	属性动画
动画实现	通过不断**图形变换**(TranslateAnimation,ScaleAnimation,RotateAnimation,AlphaAnimation)	通过动态改变**对象属性**(ValueAnimator,ObjectAnimator,AnimatorSet)
作用对象	View 对象	任何对象
存放位置	res/anim/	res/animator/
状态变化	未真正改变 view 位置	真正改变 view 位置

Q：View动画为何不能真正改变View的位置？而属性动画为何可以?

- 技术点：View动画
- 参考回答：View动画改变的只是View的显示，而没有改变View的响应区域；而属性动画会通过反射技术来获取和执行属性的get、set方法，从而改变了对象位置的属性值。

Q：属性动画插值器和估值器的作用?

- 技术点：属性动画
- 参考回答：
  - 插值器(Interpolator)

    ：根据

    时间流逝的百分比

    计算出当前

    属性值改变的百分比

    。确定了动画效果变化的模式，如匀速变化、加速变化等等。View动画和属性动画均可使用。常用的系统内置插值器：
    - 线性插值器(LinearInterpolator)：匀速动画
    - 加速减速插值器(AccelerateDecelerateInterpolator)：动画两头慢中间快
    - 减速插值器(DecelerateInterpolator)：动画越来越慢
  - 类型估值器(TypeEvaluator)

    ：根据当前

    属性改变的百分比

    计算出

    改变后的属性值

    。针对于属性动画，View动画不需要类型估值器。常用的系统内置的估值器：
    - 整形估值器(IntEvaluator)
    - 浮点型估值器(FloatEvaluator)
    - Color属性估值器(ArgbEvaluator)

Window

Q：Activity、View、Window三者之间的关系?

- 技术点：Activity、View、Window联系
- 思路：围绕Window是Activity和View的桥梁展开
- 参考回答：在Activity启动过程其中的attach()方法中初始化了PhoneWindow，而PhoneWindow是Window的唯一实现类，然后Activity通过setContentView将View设置到了PhoneWindow上，而View通过WindowManager的addView()、removeView()、updateViewLayout()对View进行管理。

Q：Window有哪几种类型?

- 技术点：Window类型
- 参考回答：Window有三种类型：
  - **应用Window**：对应一个Activity。
  - **子Window**：不能单独存在，需附属特定的父Window。如Dialog。
  - **系统Window**： 需申明权限才能创建。如Toast。

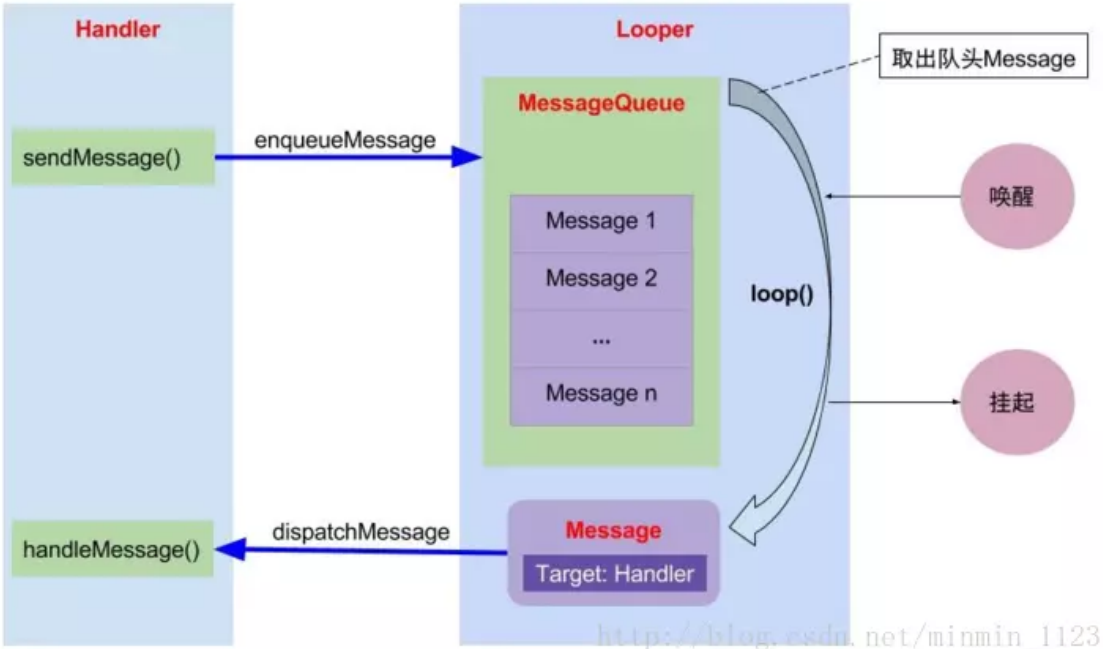
Q：Activity创建和Dialog创建过程的异同?

> - 技术点：Window创建
> - 参考回答：
>
>   Dialog
>
>   的Window创建过程：
>
>   - 创建WindowDialog。和Activity类似，同样是通过**PolicyManager.makeNewWindow()** 来实现。
>   - 初始化DecorView并将Dialog的视图添加到DecorView中去。和Activity类似，同样是通过 **Window.setContentView()** 来实现。
>   - 将DecorView添加到Window中显示。和Activity一样，都是在自身要出现在前台时才会将添加Window。
>   - **Dialog.show()** 方法：完成DecorView的显示。
>   - **WindowManager.remoteViewImmediate()** 方法：当Dialog被dismiss时移除DecorView。

Handler

Q：谈谈消息机制Hander？作用？有哪些要素？流程是怎样的?

> - 技术点：消息机制
> - 参考回答：
>   - 作用：**跨线程通信。**当子线程中进行耗时操作后需要更新UI时，通过Handler将有关UI的操作切换到主线程中执行。
>   - 四要素：
>   - **Message（消息）**：需要被传递的消息，其中包含了消息ID，消息处理对象以及处理的数据等，由MessageQueue统一列队，最终由Handler处理。
>   - **MessageQueue（消息队列）**：用来存放Handler发送过来的消息，内部通过**单链表**的数据结构来维护消息列表，等待Looper的抽取。
>   - **Handler（处理者）**：负责Message的发送及处理。通过 Handler.sendMessage() 向消息池发送各种消息事件；通过 Handler.handleMessage() 处理相应的消息事件。
>   - **Looper（消息泵）**：通过Looper.loop()不断地从MessageQueue中抽取Message，按分发机制将消息分发给目标处理者。
>   - 具体流程如图



- `Handler.sendMessage()`发送消息时，会通过`MessageQueue.enqueueMessage()`向MessageQueue中添加一条消息；
- 通过`Looper.loop()`开启循环后，不断轮询调用`MessageQueue.next()`；
- 调用目标`Handler.dispatchMessage()`去传递消息，目标Handler收到消息后调用`Handler.handlerMessage()`处理消息。

Q：为什么系统不建议在子线程访问UI?

- 技术点：UI线程、子线程
- 参考回答：系统不建议在子线程访问UI的原因是，UI控件

  非线程安全

  ，在多线程中并发访问可能会导致UI控件处于不可预期的状态。而不对UI控件的访问加上锁机制的原因有：

  - 上锁会让UI控件变得复杂和低效
  - 上锁后会阻塞某些进程的执行

Q：一个Thread可以有几个Looper？几个Handler?

- 技术点：Looper、Handler
- 参考回答：一个Thread只能有一个Looper，可以有多个Handler
- 引申：更多数量关系：Looper有一个MessageQueue，可以处理来自多个Handler的Message；MessageQueue有一组待处理的Message，这些Message可来自不同的Handler；Message中记录了负责发送和处理消息的Handler；Handler中有Looper和MessageQueue；

Q：如何将一个Thread线程变成Looper线程？Looper线程有哪些特点?

- 技术点：Looper
- 参考回答：通过Looper.prepare()可将一个Thread线程转换成Looper线程。Looper线程和普通Thread不同，它通过MessageQueue来存放消息和事件、Looper.loop()进行消息轮询。

Q：可以在子线程直接new一个Handler吗？那该怎么做?

- 技术点：Handler
- 参考回答：不同于主线程直接new一个Handler，由于子线程的Looper需要手动去创建，在创建Handler时需要多一些方法：

```
new Thread(new Runnable() {
            @Override
            public void run() {
                Looper.prepare();//为子线程创建Looper
                new Handler() {
                @Override
                public void handleMessage(Message msg) {
                    super.handleMessage(msg);
                    //子线程消息处理
                }
            };
                Looper.loop(); //开启消息轮询
            }
        }).start();
```

Q：Message可以如何创建？哪种效果更好，为什么？

- 技术点：Message
- 参考回答：创建Message对象的几种方式：
  - Message msg = new Message();
  - Message msg = Message.obtain();
  - Message msg = handler1.obtainMessage(); 后两种方法都是从整个Messge池中返回一个新的Message实例，能有效避免重复Message创建对象，因此更鼓励这种方式创建Message

Q：这里的ThreadLocal有什么作用？

- 技术点：ThreadLocal
- 参考回答：ThreadLocal类可实现线程本地存储的功能，把共享数据的可见范围限制在同一个线程之内，无须同步就能保证线程之间不出现数据争用的问题，这里可理解为ThreadLocal帮助Handler找到本线程的Looper。
  - 底层数据结构：每个线程的Thread对象中都有一个ThreadLocalMap对象，它存储了一组以ThreadLocal.threadLocalHashCode为key、以本地线程变量为value的键值对，而ThreadLocal对象就是当前线程的ThreadLocalMap的访问入口，也就包含了一个独一无二的threadLocalHashCode值，通过这个值就可以在线程键值值对中找回对应的本地线程变量。

Q：主线程中Looper的轮询死循环为何没有阻塞主线程？

- 技术点：Looper
- 参考回答：Android是依靠事件驱动的，通过Loop.loop()不断进行消息循环，可以说Activity的生命周期都是运行在 Looper.loop()的控制之下，一旦退出消息循环，应用也就退出了。而所谓的导致ANR多是因为某个事件在主线程中处理时间太耗时，因此只能说是对某个消息的处理阻塞了Looper.loop()，反之则不然。

Q：使用Hanlder的postDealy()后消息队列会发生什么变化？

- 技术点：Handler

- 参考回答：post delay的Message并不是先等待一定时间再放入到MessageQueue中，而是直接进入并阻塞当前线程，然后将其delay的时间和队头的进行比较，按照触发时间进行排序，如果触发时间更近则放入队头，保证队头的时间最小、队尾的时间最大。此时，如果队头的Message正是被delay的，则将当前线程堵塞一段时间，直到等待足够时间再唤醒执行该Message，否则唤醒后直接执行。

线程

Q：Android中还了解哪些方便线程切换的类?

- 技术点：线程通信
- 参考回答：对Handler进一步的封装的几个类：
  - **AsyncTask**：底层封装了线程池和**Handler**，便于执行后台任务以及在子线程中进行UI操作。
  - **HandlerThread**：一种具有**消息循环**的线程，其内部可使用Handler。
  - **IntentService**：是一种**异步、会自动停止**的服务，内部采用HandlerThread。
- 引申：更多是对消息机制的理解

Q：AsyncTask相比Handler有什么优点？不足呢?

- 技术点：AsyncTask、Handler
- 参考回答：
  - Handler机制存在的**问题**：多任务同时执行时不易精确控制线程。
  - 引入AsyncTask的**好处**：创建异步任务更简单，直接继承它可方便实现后台异步任务的执行和进度的回调更新UI，而无需编写任务线程和Handler实例就能完成相同的任务。

Q：使用AsyncTask需要注意什么?

- 技术点：AsyncTask
- 参考回答：
  - 不要直接调用onPreExecute()、doInBackground()、onProgressUpdate()、onPostExecute()和onCancelled()方法
  - 一个异步对象只能调用一次execute()方法
- 引申：谈谈AsyncTask初始化、五个核心方法如何配合进而体现Handler的作用

Q：AsyncTask中使用的线程池大小?

- 技术点：AsyncTask
- 参考回答：在AsyncTask内部实现有两个线程池：
  - **SerialExecutor**：用于任务的排队，默认是**串行**的线程池，在3.0以前核心线程数为5、线程池大小为128，而3.0以后变为同一时间只能处理一个任务
  - **THREAD_POOL_EXECUTOR**：用于真正执行任务。
- 引申：谈谈对线程池的理解

Q：HandlerThread有什么特点?

- 技术点：HandlerThread
- 参考回答：HandlerThread是一个线程类，它继承自Thread。与普通Thread不同，HandlerThread具有**消息循环**的效果，这是因为它内部`HandlerThread.run()`方法中有Looper，能通过`Looper.prepare()`来创建消息队列，并通过`Looper.loop()`来开启消息循环。

Q：快速实现子线程使用Handler

- 技术点：HandlerThread
- 思路：不同于之前手动在子线程创建Looper再构建Handler的想法，这里从HandlerThread角度去快速实现在子线程使用Handler
- 参考回答：HandlerThread实现方法
  - 实例化一个HandlerThread对象，参数是该线程的名称；
  - 通过 HandlerThread.start()开启线程；
  - 实例化一个Handler并传入HandlerThread中的looper对象，使得与HandlerThread绑定；
  - 利用Handler即可执行异步任务；
  - 当不需要HandlerThread时，通过HandlerThread.quit()/quitSafely()方法来终止线程的执行。

Q：IntentService的特点?

- 技术点：IntentService
- 思路：和普通线程和普通Service比较突出其特点
- 参考回答： 不同于线程，IntentService是服务，优先级比线程高，更不容易被系统杀死，因此较适合执行一些**高优先级**的后台任务；不同于普通Service，IntentService可**自动创建**子线程来执行任务，且任务执行完毕后**自动退出。**

Q：为何不用bindService方式创建IntentService?

- 技术点：IntentService
- 思路：从底层实现出发
- 参考回答：IntentService的工作原理是，在IntentService的onCreate()里会创建一个HandlerThread，并利用其内部的Looper实例化一个ServiceHandler对象；而这个ServiceHandler用于处理消息的handleMessage()方法会去调用IntentService的onHandleIntent()，这也是为什么可在该方法中处理后台任务的逻辑；当有Intent任务请求时会把Intent封装到Message，然后ServiceHandler会把消息发送出，而发送消息是在onStartCommand()完成的，只能通过startService()才可走该生命周期方法，因此不能通过bindService创建IntentService。

Q：线程池的好处、原理、类型?

- 技术点：线程池
- 参考回答：
- （1）线程池的好处：
  - **重用**线程池中的线程，避免线程的创建和销毁带来的性能消耗；
  - 有效控制线程池的**最大并发数**，避免大量的线程之间因互相抢占系统资源而导致阻塞现象；
  - 进行**线程管理**，提供定时/循环间隔执行等功能
- （2）线程池的分类：
  - **FixThreadPool**：线程数量固定的线程池，所有线程都是**核心线程**，当线程空闲时**不会**被回收；能**快速**响应外界请求。
  - **CachedThreadPool**：线程数量不定的线程池（最大线程数为**Integer.MAX_VALUE**），只有**非核心线程**，空闲线程有超时机制，超时回收；适合于执行大量的**耗时较少**的任务
  - **ScheduledThreadPool**：核心线程数量**固定**，非核心线程数量**不定**；可进行**定时**任务和**固定**周期的任务。
  - **SingleThreadExecutor**：只有**一个核心线程**，可确保所有的任务都在同一个线程中**按顺序**执行；好处是无需处理**线程同步**问题。
- （3）线程池的原理：实际上通过ThreadPoolExecutor并通过一系列参数来配置各种各样的线程池，具体的参数有：

    - **corePoolSize**核心线程数：一般会在线程中一直存活
    - **maximumPoolSize**最大线程数：当活动线程数达到这个数值后，后续的任务将会被阻塞
    - **keepAliveTime**非核心线程超时时间：超过这个时长，闲置的非核心线程就会被回收
    - **unit**：用于指定keepAliveTime参数的时间单位
    - **workQueue**任务队列：通过线程池的`execute()`方法提交的Runnable对象会存储在这个参数中。
    - **threadFactory**：线程工厂，可创建新线程
    - **handler**：在线程池无法执行新任务时进行调度
- 引申：使用Executors各个方法创建线程池的弊端

Q：ThreadPoolExecutor的工作策略?

- 技术点：线程池
- 参考回答：ThreadPoolExecutor的

  默认工作策略

  ：
    - 若程池中的线程数量**未达到**核心线程数，则会直接启动一个核心线程执行任务。
    - 若线程池中的线程数量

      已达到

      或者超过核心线程数量，则任务会被插入到任务列表等待执行。
        - 若任务

          无法插入

          到任务列表中，往往由于任务列表已满，此时如果
            - 线程数量**未达到**线程池最大线程数，则会启动一个非核心线程执行任务；
            - 线程数量**已达到**线程池规定的最大值，则拒绝执行此任务，ThreadPoolExecutor会调用RejectedExecutionHandler的rejectedExecution方法来通知调用者。
- 引申：ThreadPoolExecutor的拒绝策略

Q：什么是ANR？什么情况会出现ANR？如何避免？在不看代码的情况下如何快速定位出现ANR问题所在?

- 技术点：ANR
- 思路：
- 参考回答：
    - ANR（Application Not Responding，应用无响应）：当操作在一段时间内系统无法处理时，会在系统层面会弹出ANR对话框
    - 产生ANR可能是因为5s内无响应用户输入事件、10s内未结束BroadcastReceiver、20s内未结束Service
    - 想要避免ANR就不要在主线程做耗时操作，而是通过开子线程，方法比如继承Thread或实现Runnable接口、使用AsyncTask、IntentService、HandlerThread等
- 引申：快读定位ANR方法：使用命令导出ANR日志，并分析关键信息，详见如何分析ANR

Bitmap

Q：加载图片的时候需要注意什么?

- 技术点：Bitmap高效加载
- 参考回答：

- 直接加载大容量的高清Bitmap很容易出现显示不完整、内存溢出OOM的问题，所以最好按一定的**采样率**将图片缩小后再加载进来
- 为减少流量消耗，可对图片采用内存缓存策略，又为了避免图片占用过多内存导致内存溢出，最好以软引用方式持有图片
- 如果还需要网上下载图片，注意要开子线程去做下载的耗时操作

Q：LRU算法的原理?

- 技术点：LRU算法
- 参考回答：为减少流量消耗，可采用缓存策略。常用的缓存算法是LRU(Least Recently Used)：
  - 核心思想：当缓存满时, 会优先淘汰那些近期最少使用的缓存对象。主要是两种方式：
    - LruCache(内存缓存)：LruCache类是一个线程安全的**泛型类**：内部采用一个`LinkedHashMap`以**强引用**的方式存储外界的缓存对象，并提供`get`和`put`方法来完成缓存的获取和添加操作，当缓存满时会移除较早使用的缓存对象，再添加新的缓存对象。
    - DiskLruCache(磁盘缓存)： 通过将缓存对象**写入文件**系统从而实现缓存效果
- 引申：感兴趣可了解具体实现算法

性能优化

Q：项目中如何做性能优化的?

- 技术点：性能优化实例
- 思路：举例说明项目注意了哪些方面的性能优化，如布局优化、绘制优化、内存泄漏优化、 响应速度优化、列表优化、Bitmap优化、 线程优化.....

Q：了解哪些性能优化的工具?

- 技术点：性能优化工具
- 思路：做项目时是否使用过的系统自带的性能优化工具？公司是否有自己的性能优化工具？实现原理怎样的?

Q：布局上如何优化?

- 技术点：布局优化
- 参考回答：布局优化的核心就是尽量减少布局文件的

  层级

  ，常见的方式有：
  - 多嵌套情况下可使用RelativeLayout减少嵌套。
  - 布局层级相同的情况下使用LinearLayout，它比RelativeLayout更高效。
  - 使用`<include>`标签重用布局、`<merge>`标签减少层级、`<ViewStub>`标签懒加载。

Q：内存泄漏是什么？为什么会发生？常见哪些内存泄漏的例子？都是怎么解决的?

- 技术点：内存泄漏
- 参考回答：内存泄漏(Memory Leak)是指程序在申请内存后，

  无法释放

  已申请的内存空间。简单地说，发生内存泄漏是由于长周期对象持有对短周期对象的引用，使得短周期对象不能被及时回收。常见的几个例子和解决办法：

    - 单例模式导致的内存泄漏：单例传入参数this来自Activity，使得持有对Activity的引用。
        - 解决办法：传参context.getApplicationContext()
    - Handler导致的内存泄漏：Message持有对Handler的引用，而非静态内部类的Handler又隐式持有对外部类Activity的引用，使得引用关系会保持至消息得到处理，从而阻止了Activity的回收。
        - 解决办法：使用静态内部类+WeakReference弱引用；当外部类结束生命周期时清空消息队列。
    - 线程导致的内存泄漏：AsyncTask/Runnable以

      匿名内部类

      的方式存在，会隐式持有对所在Activity的引用。
        - 解决办法：将AsyncTask和Runnable设为静态内部类或独立出来；在线程内部采用弱引用保存Context引用
    - 资源未关闭导致的内存泄漏：未及时注销资源导致内存泄漏，如BraodcastReceiver、File、Cursor、Stream、Bitmap等。
        - 解决办法：在Activity销毁的时候要及时关闭或者注销。
            - BraodcastReceiver：调用`unregisterReceiver()`注销；
            - Cursor，Stream、File：调用`close()`关闭；
            - 动画：在`Activity.onDestroy()`中调用`Animator.cancel()`停止动画
- 引申：谈谈项目中是如何注意内存泄漏的问题

Q：内存泄漏和内存溢出的区别

- 技术点：内存泄漏、内存溢出
- 参考回答：
    - 内存泄漏(Memory Leak)是指程序在申请内存后，**无法释放**已申请的内存空间。是造成应用程序OOM的主要原因之一。
    - 内存溢出(out of memory)是指程序在申请内存时，没有足够的内存空间供其使用。

Q：什么情况会导致内存溢出?

- 技术点：内存溢出
- 参考回答：内存泄漏是导致内存溢出的主要原因；直接加载大图片也易造成内存溢出
- 引申：谈谈如何避免内存溢出（如何避免内存泄漏、避免直接加载大图片）

- 开源框架（略）
- 谷歌新动态

```
Q：是否了解和使用过谷歌推出的新技术？`
 `Q：有了解刚发布的Androidx.0的特性吗？`
 `Q：Kotlin对Java做了哪些优化?
```

- 可能意图：了解候选者对谷歌&安卓的关注度、共同探讨对新技术的看法、学习主动性、平时学习习惯
- 思路：谷歌的安卓官方网站（中文版）：https://developer.android.google.cn，了解最新动态