Android 进阶延伸点

1、如何进行单元测试，如何保证 App 稳定 ？

- 参考回答：

要测试Android应用程序，通常会创建以下类型自动单元测试

    - **本地测试**：只在本地机器 JVM 上运行，以最小化执行时间，这种单元测试不依赖于 Android 框架，或者即使有依赖，也很方便使用模拟框架来模拟依赖，以达到隔离 Android 依赖的目的，模拟框架如 Google 推荐的Mockito；
    - **检测测试**：真机或模拟器上运行的单元测试，由于需要跑到设备上，比较慢，这些测试可以访问仪器（Android 系统）信息，比如被测应用程序的上下文，一般地，依赖不太方便通过模拟框架模拟时采用这种方式；
  - 注意：单元测试不适合测试复杂的 UI 交互事件
  - App 的稳定主要决定于整体的系统架构设计，同时也不可忽略代码编程的细节规范，正所谓“千里之堤，溃于蚁穴”，一旦考虑不周，看似无关紧要的代码片段可能会带来整体软件系统的崩溃，所以上线之前除了自己**本地化测试**之外还需要进行 **Monkey 压力测试**
  - 少部分面试官可能会延伸，如 Gradle 自动化测试、机型适配测试等

2、Android 中如何查看一个对象的回收情况 ？

- 参考回答：
  - 首先要了解 Java 四种引用类型的场景和使用（强引用、软引用、弱引用、虚引用）
  - 举个场景例子：**SoftReference** 对象是用来保存软引用的，但它同时也是一个 Java 对象，所以当软引用对象被回收之后，虽然这个 **SoftReference** 对象的 get 方法返回null，但 **SoftReference** 对象本身并不是 null，而此时这个 **SoftReference** 对象已经不再具有存在的价值，需要一个适当的清除机制，避免大量 **SoftReference** 对象带来的**内存泄露**
  - 因此，Java 提供**ReferenceQueue** 来处理引用对象的回收情况。当 **SoftReference** 所引用的对象被 GC 后，**JVM** 会先将 **softReference** 对象添加到 **ReferenceQueue** 这个队列

中。当我们调用**ReferenceQueue 的 poll()方法**，如果这个队列中不是空队列，那么将返回并移除前面添加的那个对象。

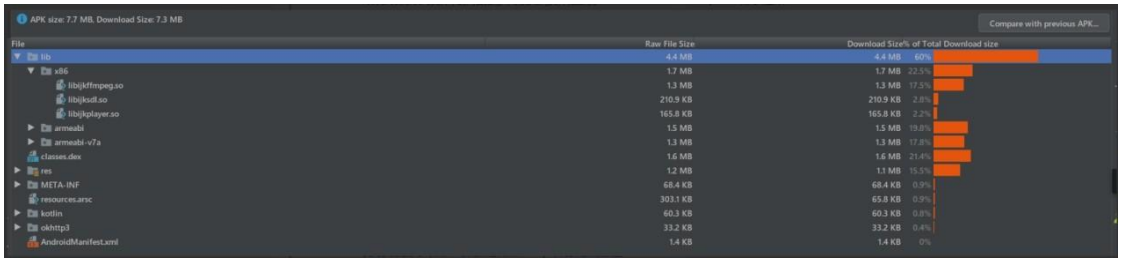
```
public static void main(String[] args) throws InterruptedException {
        //假设当前JVM内存只有8m
        Person person = new Person("张三");
        ReferenceQueue<Person> queue = new ReferenceQueue<>();
        SoftReference<Person> softReference = new SoftReference<Person>(person, queue);

        person = null;//去掉强引用，new Person("张三")的这个对象就只有软引用了

        Person anotherPerson = new Person("李四");//没有足够的空间同时保留两个Person对象,所以触发GC机制
        Thread.sleep(1000);

        System.err.println("软引用的对象 ------->" + softReference.get());

        Reference softPollRef = queue.poll();
        if (softPollRef != null) {
            System.err.println("SoftReference对象中保存的软引用对象已经被GC，准备清理SoftReference对象");
            //清理softReference
        }
    }
```

3、Apk 的大小如何压缩 ？

- 参考回答：
  - 一个完整 包含以下目录（将APK文件拖到Android APK

    - **META-INF/**：包含 **CERT.SF** 和**CERT.RSA** 签名文件以及**MANIFEST.MF** 清单文件。 **assets/**：包含应
    - 用可以使用 **AssetManager** 对象检索的应用资源。
    - **res/**：包含未编译到的资源 resources.arsc。**lib/**：
    - 包含特定于处理器软件层的编译代码。该目录包含
    - 了每种平台的子目录，像**armeabi**，**armeabi-v7a**，**arm64-v8a**，**x86**，**x86_64**，和**mips**。 **resources.arsc**：包含已编译的资源。该文件包含 **res/values/** 文件夹所有配置中的XML内容。打包
    - 工具提取此 XML 内容，将其编译为二进制格式，并将内容归档。此内容包括语言字符串和样式，以及直接包含在**resources.arsc*8 文件中的内容路径，例如布局文件和图像。
      classes.dex：包含以 **Dalvik/ART** 虚拟机可理解的 **DEX** 文件格式编译的类。**AndroidManifest.xml**：
    - 包含核心 Android 清单文件。该文件列出应用程序的名称，版本，访问权限
    -

和引用的库文件。该文件使用Android的二进制XML格式。



- lib、class.dex 和 res 占用了超过 90%的空间，所

一)

- **减少 res，压缩图文文件**
  - 图片文件压缩是针对 jpg和png 格式的图片。我们通常会放置多套不同分辨率的图片以适配不同的屏幕，这里可以进行适当的删减。在实际使用中，只保留一到两套就足够了（保留一套的话建议保留xxhdpi，两套的话就加上 hdpi），然后再对剩余的图片进行压缩(jpg 采用优图压缩，png 尝试采用pngquant 压缩)
- **减少 dex 文件大小**
  - 添加资源混淆

```
buildTypes {
    release {
        shrinkResources true
        minifyEnabled true
        proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
    }
}
```

  - shrinkResources 为true 表示移除未引用资源，和代码压缩协同工作。
  - minifyEnabled 为true 表示通过 ProGuard 启用代码压缩，配合 proguardFiles 的配置对代码进行混淆并移除未使用的代码。
  - 代码混淆在压缩 apk 的同时，也提升了安全性。

- **减少 lib 文件大小**

- 由于引用了很多第三方库，lib 文件夹占用的空间通常都很大，特别是有 so 库的情况下。很多 so 库会同时引入armeabi、armeabi-v7a和x86 这几种类型，这里可以只保留 armeabi 或armeabi-v7a 的其中一个就可以了，实际上微信等主流app 都是这么做的。
只需在 build.gradle 直接配置即可，NDK 配置同理
-

```
defaultConfig {
    ndk {
        abiFilters 'armeabi'
    }
}
```

4、如何通过 Gradle 配置多渠道包？

- 参考回答：
  - 首先要了解设置多渠道的原因。在安装包中添加不同的标识，配合自动化埋点，应用在请求网络的时候携带渠道信息，方便后台做运营统计，比如说统计我们的应用在不同应用市场的下载量等信息
  - 这里以友盟统计为例
    - 首先在manifest.xml文件中设置动态渠道变量：

```
<!-- 设置友盟渠道 -->
<meta-data
    android:name="UMENG_CHANNEL"
    android:value="${UMENG_CHANNEL_VALUE}" />
```

    - 接着在 app 目录下的 build.gradle 中配置 productFlavors，也就是配置打包的渠道：

```
//渠道Flavors，配置不同风格的app
productFlavors {

    Other{ manifestPlaceholders = [UMENG_CHANNEL_VALUE: "Other"] }
    UM   { manifestPlaceholders = [UMENG_CHANNEL_VALUE: "UM"] }
    C360 { manifestPlaceholders = [UMENG_CHANNEL_VALUE: "C360"] }
    BD   { manifestPlaceholders = [UMENG_CHANNEL_VALUE: "BD"] }
    HW   { manifestPlaceholders = [UMENG_CHANNEL_VALUE: "HW"] }
    PP   { manifestPlaceholders = [UMENG_CHANNEL_VALUE: "PP"] }
    XM   { manifestPlaceholders = [UMENG_CHANNEL_VALUE: "XM"] }
    YYB  { manifestPlaceholders = [UMENG_CHANNEL_VALUE: "YYB"] }
    VIVO { manifestPlaceholders = [UMENG_CHANNEL_VALUE: "VIVO"] }
    OPPO { manifestPlaceholders = [UMENG_CHANNEL_VALUE: "OPPO"] }
    ST   { manifestPlaceholders = [UMENG_CHANNEL_VALUE: "SMARTISAN"] }
}
```

    - 最后在编辑器下方的Teminal输出命令行
        - **执行./gradlew assembleRelease ，将会打断有渠道的 release 包：**
        - **执行./gradlew assembleVIVO，将会打出 VIVO 渠道的 release 和 debug 版的包：**
        - **执行./gradlew assembleVIVORelease 将生成 VIVO 的 release 包。**

5、插件化原理分析

- 参考回答：
    - **插件化**是指将 APK 分为**宿主**和**插件**的部分。把需要实现的模块或功能当做一个独立的提取出来，在 APP 运行时，我们可以动态的**载入**或者**替换插件**部分，减少**宿主**的规模
        - 宿主：就是当前运行的APP。
        - 插件：相对于插件化技术来说，就是要加载运行的apk类文件。
    - 而**热修复**则是从修复 bug 的角度出发，强调的是在不需要二次安装应用的前提下修复已知的 bug。能

- **类加载机制**
  - Android 中常用的两种类加载器，**DexClassLoader**和**PathClassLoader**，它们都继承于**BaseDexClassLoader**，两者**区别**在于**PathClassLoader** 只能加载**内部存储目录**的 dex/jar/apk 文件。**DexClassLoader** 支持加载**指定目录**(不限于内部)的 dex/jar/apk 文件

- **插件通信**：通过给插件 apk 生成相应的 DexClassLoader 便可以访问其中的类，可分为单 DexClassLoader 和多 DexClassLoader 两种结构。
  - 若使用**多 ClassLoader 机制**，主工程引用插件中类需要先通过插件的 ClassLoader 加载该类再通过**反射**调用其方法。插件化框架一般会通过统一的入口去管理对各个插件中类的访问，并且做一定的限制。

        - 若使用**单 ClassLoader 机制**，主工程则可以**直接通过**类名去访问插件中的类。该方式有个弊端，若两个不同的插件工程引用了一个库的不同版本，则程序可能会出错。

    - **资源加载**

        - 原理在于通过反射将插件 apk 的路径加入 AssetManager 中并创建Resource 对象加载资源，有两种处理方式：

            - 合并式：addAssetPath 时加入所有插件和主工程的路径；由于 AssetManager 中加入了所有插件和主工程的路径，因此生成的Resource 可以同时访问插件和主工程的资源。但是由于主工程和各个插件都是独立编译的，生成的资源 id 会存在相同的情况，在访问时会产生资源冲突。
            - 独立式：各个插件只添加自己 apk 路径，各个插件的资源是互相隔离的，不过如果想要实现资源的共享，必须拿到对应的 Resource 对象。

6、组件化原理

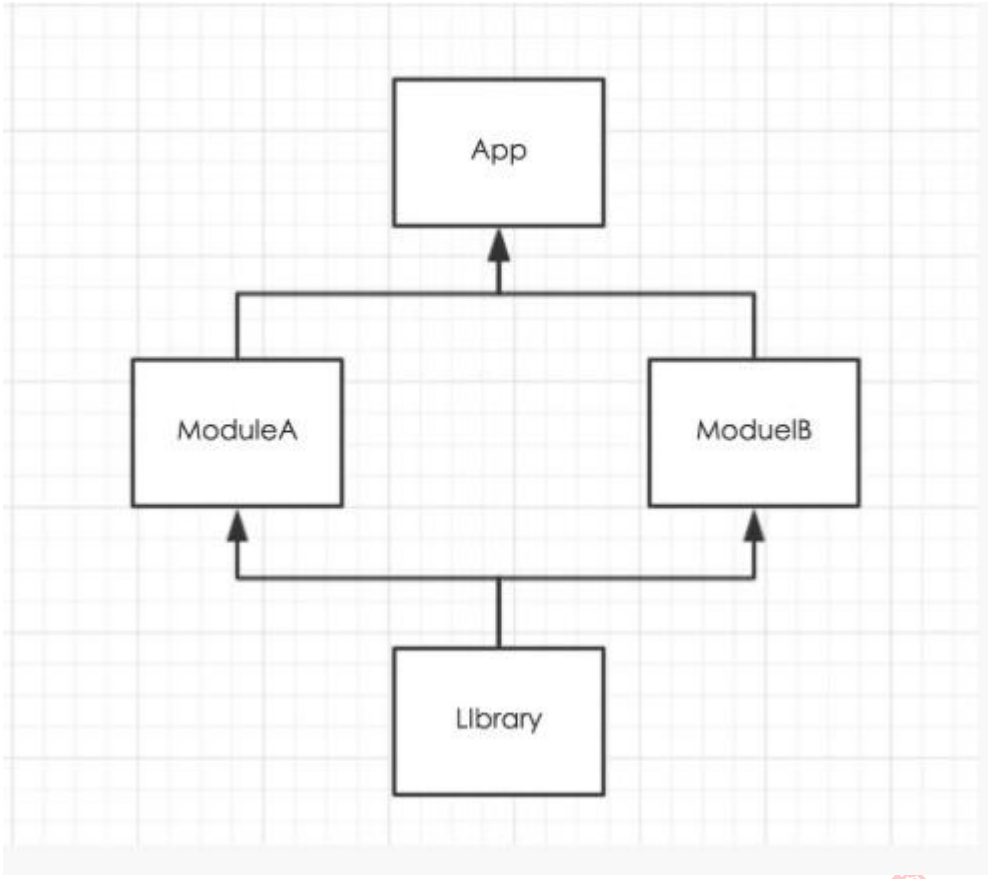
- 参考回答：
    - **引入组件化的原因**：项目随着需求的增加规模变得越来越大，规模的增大导致了各种业务错中复杂的交织在一起，每个业务模块之间，代码没有约束，带来了代码边界的模糊，代码冲突时有发生，更改一个小问题可能引起一些新的问题，牵一发而动全身，增加一个新需求，需要熟悉相关的代码逻辑，增加开发时间

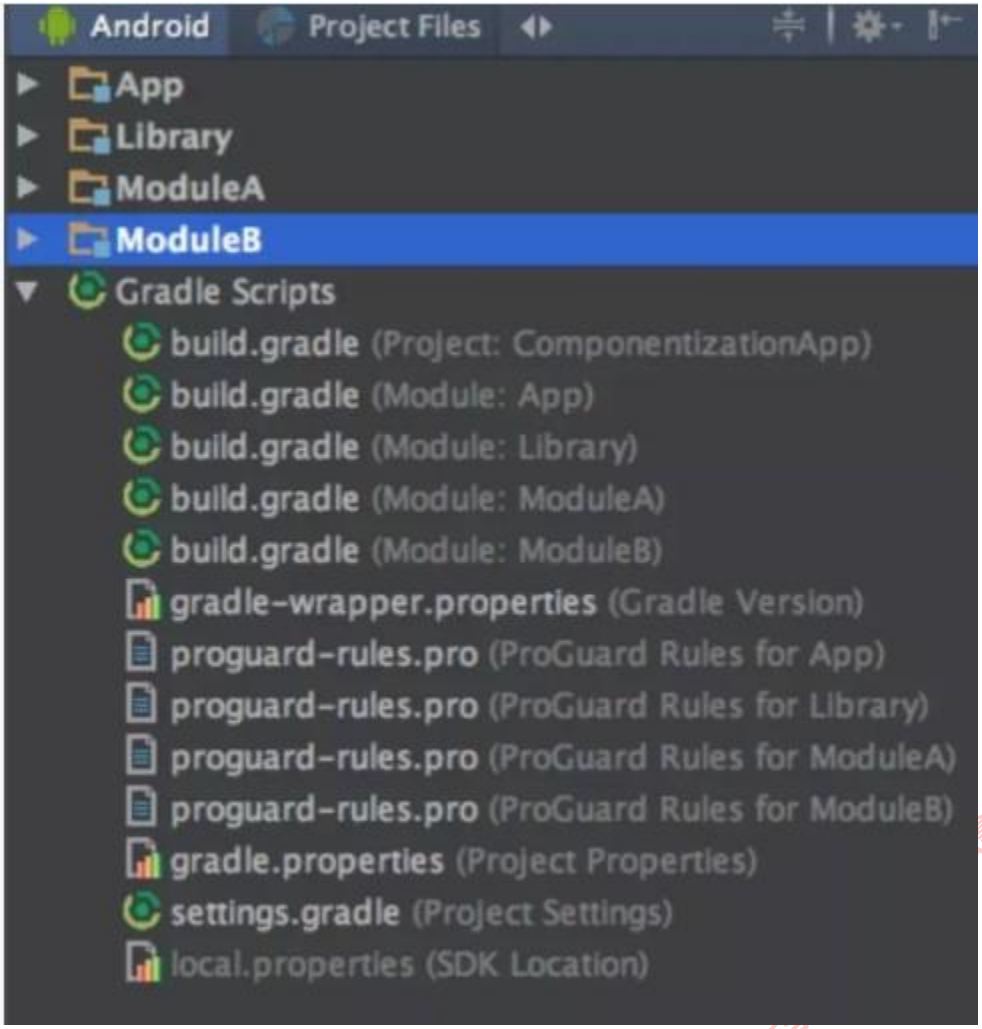
        - **避免重复造轮子，可以节省开发和维护的成本。**
        - **可以通过组件和模块为业务基准合理地安排人力，提高开发效率。**
        - **不同的项目可以共用一个组件或模块，确保整体技术方案的统一性。**
        - **为未来插件化共用同一套底层模型做准备。**

    - **组件化开发流程**就是把一个功能完整的 App 或模块拆分成**多个子模块（Module）**，每个子模块可以**独立编译运行**，也可以任意组合成另一个新的 App 或模块，每个模块即不相互依赖但又可以相互交互，但是最终发布的时候是将这

些组件合并统一成一个apk，遇到某些特殊情况甚至可以**升级**或者**降级**

- 举个简单的模型例子





App是主application，ModuleA和ModuleB是两个业务模块（**相对独立，互不影响**），Library是基础模块，包含所有模块需要的依赖库，以及一些工具类：如网络访问、时间工具等

- **注意：提供给各业务模块的基础组件，需要根据具体情况拆分成 aar 或者 library，像登录，基础网络层这样较为稳定的组件，一般直接打包成 aar，减少编译耗时。而像自定义 View 组件，由于随着版本迭代会有较多变化，就直接以源码形式抽离成 Library**

7、跨组件通信

- 参考回答：
  - **跨组件通信场景：**
    - 第一种是组件之间的页面跳转（Activity 到Activity，Fragment 到 Fragment，Activity 到 Fragment，Fragment 到Activity）以及跳转时的数

据传递　　(基础数据类型和可序列化的自定义类类)。

- 第二种是组件之间的自定义类和自定义方法的调用(组件向外提供服务)。

- **跨组件通信方案分析**:

  - 第一种**组件之间的页面跳转**实现简单，跳转时想传递不同类型的数据提供有相应的 API即可。
  - 第二种组件之间的自定义类和**自定义方法的调用**要稍微复杂点，需要 ARouter 配合架构中的 公共服务(CommonService) 实现:

    - 提供服务的业务模块:
    - 在公共服务(CommonService) 中声明Service 接口(含有需要被调用的自定义方法), 然后在自己的模块中实现这个 Service 接口, 再通过 ARouter API 暴露实现类。
    - 使用服务的业务模块:

      - 通过 ARouter 的 API 拿到这个Service 接口(多态持有, 实际持有实现类), 即可调用 Service 接口中声明的自定义方法, 这样就可以达到模块之间的交互。

    - 此外，可以使用 AndroidEventBus 其独有的 Tag, 可以在开发时更容易定位发送事件和接受事件的代码, 如果以组件名来作为 Tag 的前缀进行分组，也可以更好的统一管理和查看每个组件的事件, 当然也不建议大家过多使用 EventBus。

- **如何管理过多的路由表？**

  - RouterHub 存在于基础库, 可以被看作是所有组件都需要遵守的通讯协议，里面不仅可以放路由地址常量, 还可以放跨组件传递数据时命名的各种 Key 值, 再配以适当注释，任何组件开发人员不需要事先沟通只要依赖了这个协议，就知道了各自该怎样协同工作, 既提高了效率又降低了出错风险, 约定的东西自然要比口头上说强。
  - Tips: 如果您觉得把每个路由地址都写在基础库的 RouterHub 中, 太麻烦了, 也可以在每个组件内部建立一个私有 RouterHub, 将不需要跨组件的路由地址

放入私有 RouterHub 中管理，只将需要跨组件的路由地址放入基础库的公有 RouterHub 中管理，如果您不需要集中管理所有路由地址的话，这也是比较推荐的一种方式。

- **ARouter 路由原理：**
  - ARouter 维护了一个路由表 Warehouse，其中保存着全部的模块跳转关系，ARouter 路由跳转实际上还是调用了 startActivity 的跳转，使用了原生的Framework 机制，只是通过 apt 注解的形式制造出跳转规则，并人为地拦截跳转和设置跳转条件。

- 常见的组件化方案如下

对比项	CC	得到DDComponentForAndroid	ModularizationArchitecture	阿里Arouter (网上很多组件化方案的路由引擎)	聚美组件化方案 (基于[聚美Router)	ActivityRouter
开源时间	2017-11	2017-9	2017-1	2016-12	2016-9	2016-4
通信机制	组件总线	路由 + 接口下沉	组件总线	路由 + 接口下沉	路由 + 接口下沉	路由 + 静态方法
activity跳转	✓	✓	✓	✓	✓	✓
是否支持降级处理	✓	✗	✓	✓	✓	✓
activity变量自动注入	✗	1. 通过apt生成自动注入代码 2. 在onCreate中调用 `AutowiredService.Factory.getInstance().create().autowire(this)`；或者继承BaseActivity	✗	1. 通过apt生成解析参数的代码 2. 在onCreate方法中调用 `ARouter.getInstance().inject(this);` 实现自动注入	✗	✗
startActivityForResult	支持Activity/Fragment，但不建议使用 建议使用统一的组件调用方式	仅支持Activity	仅支持Activity	仅支持Activity	支持Activity/Fragment	仅支持Activity

8、组件化中路由、埋点的实现

- 参考回答：
  - 因为在组件化中，各个业务模块之间是各自**独立**的，并不会存在相互依赖的关系，所以一个业务模块是访问不了其他业务模块的代码的，如果想从 A 业务模块的 A 页面跳转到 B 业务模块的 B 页面，光靠模块自身是不能实现的，这就需要一种跨组件通信方案—— **路由（Router）**
  - **路由**主要有以下两种场景：
    - 第一种是**组件之间的页面跳转** (Activity 到 Activity，Fragment 到 Fragment，Activity 到 Fragment，Fragment 到 Activity) 以及跳转时的数据传递 (基础数据类型和可序列化的自定义类类型)

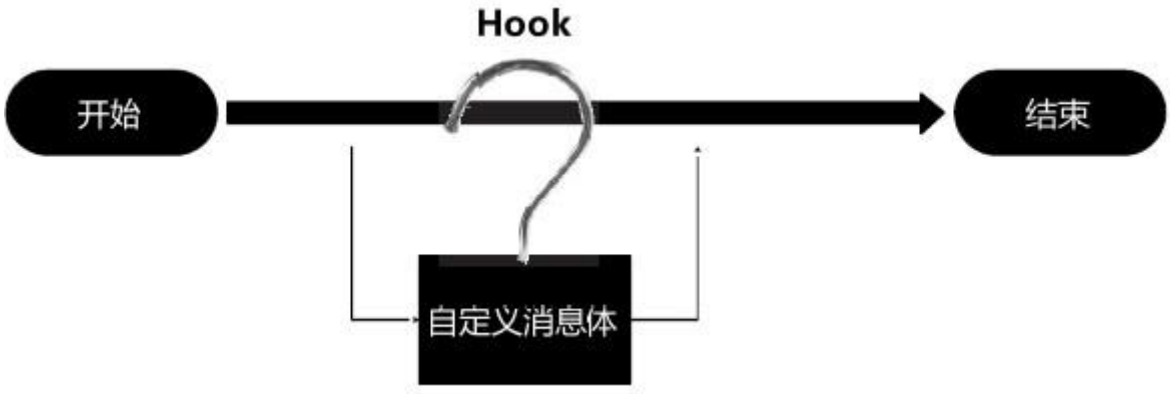
        - 第二种是**组件之间的自定义类**和**自定义方法的调用**(组件向外提供服务)

    - 其**原理**在于将分布在不同组件 module 中的某些类按照一定规则生成映射表（数据结构通常是 Map，Key 为一个字符串，Value 为类或对象），然后在需要用到的时候从映射表中根据字符串从映射表中取出类或对象，本质上是类的查找
    - 埋点则是在应用中特定的流程收集一些信息，用来跟踪应用使用的状况

        - **代码埋点**：在某个事件发生时调用 SDK 里面相应的接口发送埋点数据，百度统计、友盟、TalkingData、Sensors Analytics 等第三方数据统计服务商大都采用这种方案
        - **全埋点**：全埋点指的是将 Web 页面/App 内产生的所有的、满足某个条件的行为，全部上报到后台服务器
        - **可视化埋点**：通过可视化工具（例如 Mixpanel）配置采集节点，在 Android 端自动解析配置并上报埋点数据，从而实现所谓的**自动埋点**
        - **无埋点**：它并不是真正的不需要埋点，而是 Android 端自动采集全部事件并上报埋点数据，在后端数据计算时过滤出有用数据

9、Hook 以及插桩技术

- 参考回答：
    - Hook 是一种用于**改变 API 执行结果**的技术，能够将系统的 API 函数执行**重定向**（应用的**触发事件**和**后台逻辑处理**是根据事件流程一步步地向下执行。而 **Hook** 的意思，就是在事件传送到终点前截获并监控事件的传输，像个钩子钩上事件一样，并且能够在钩上事件时，处理一些自己特定的事件，例如逆向破解 App）



- Android 中的 Hook 机制，大致有两个方式：
  - 要 root 权限，直接 Hook 系统，可以干掉所有的App。
  - 无 root 权限，但是只能 Hook 自身app，对系统其App无能为力。
- **插桩**是以静态的方式修改第三方的代码，也就是从编译阶段，对源代码（中间代码）进行编译，而后重新打包，是**静态的篡改**； 而**Hook** 则不需要再编译阶段修改第三方的源码或中间代码．是在运行时通过反射的方式修改调用，是一种**动态的篡改**

10、Android 的签名机制？

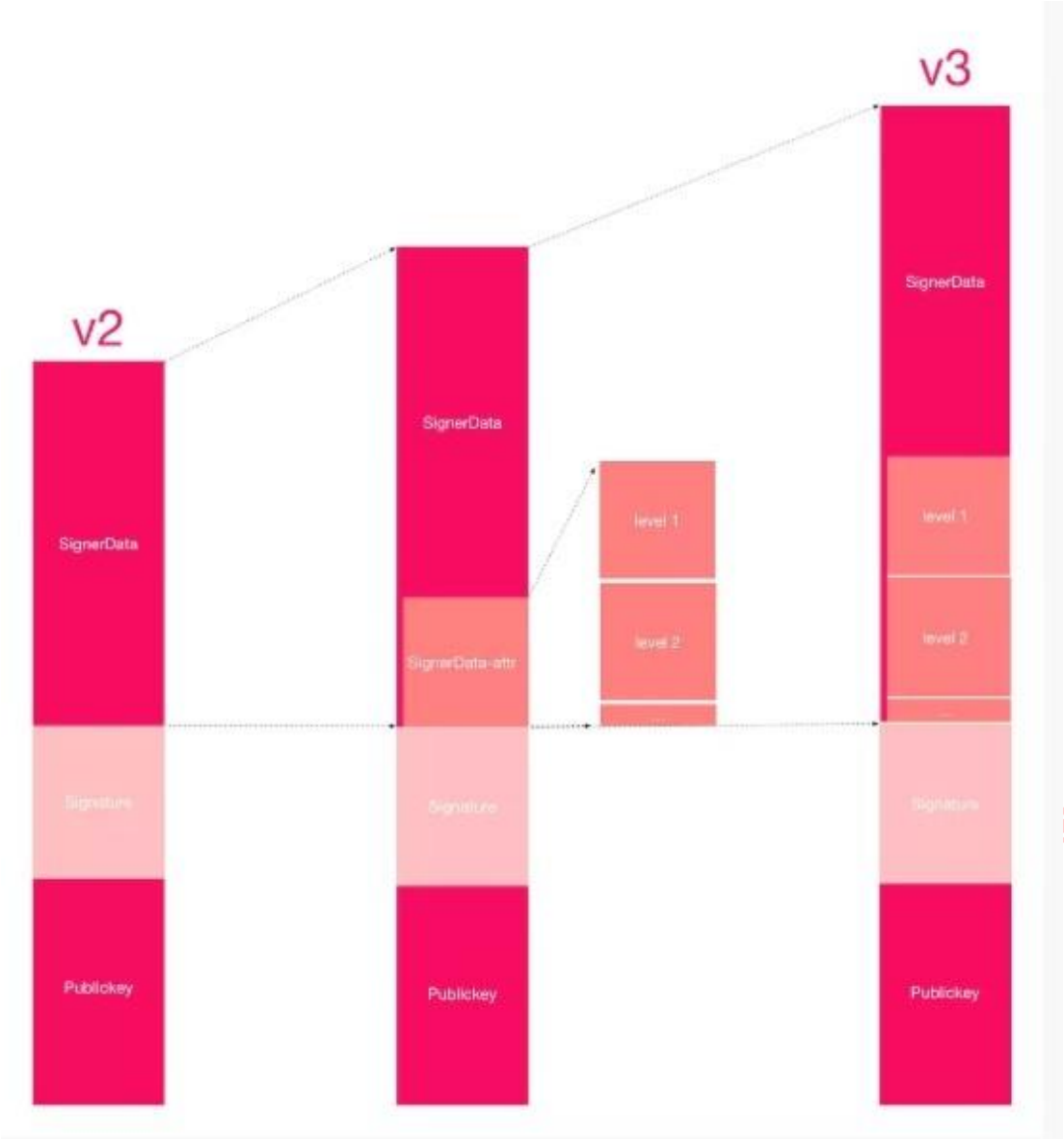
- 参考回答：
  - Android的签名机制包含有**消息摘要**、**数字签名**和**数字证书**
    - **消息摘要**：在消息数据上，执行一个单向的 Hash 函，生成一个固定长度的Hash值
    - **数字签名**：一种以电子形式存储消息签名的方法，一个完整的数字签名方案应该由两部分组成：**签名算法和验证算法**
    - **数字证书**：一个经证书授权（Certificate Authentication）中心数字签名的包含公钥拥有者信息以及公钥的文件

11、v3 签名 key 和v2 还有 v1 有什么区别

- 参考回答：

- 在 **v1 版本**的签名中，签名以文件的形式存在于 apk 包中，这个版本的 apk 包就是一个标准的 zip 包，**V2** 和**V1** 的差别是 **V2** 是对整个 zip 包进行签名，而且在 zip 包中增加了一个 **apk signature block**，里面保存签名信息。

- **v2 版本**签名块（APK Signing Block）本身又主要分成三部分：
  - **SignerData**（签名者数据）：主要包括签名者的证书，整个 APK 完整性校验 hash，以及一些必要信息
  - **Signature**（签名）：开发者对 SignerData 部分数据的签名数据
  - **PublicKey**（公钥）：用于验签的公钥数据
- **v3 版本**签名块也分成同样的三部分，与 v2 不同的是在 SignerData 部分，v3 新增了 attr 块，其中是由更小的 level 块组成。每个 level 块中可以存储一个证书信息。前一个 level 块证书验证下一个 level 证书，以此类推。最后一个 level 块的证书，要符合 SignerData 中本身的证书，即用来签名整个 APK 的公钥所属于的证书



12、Android5.0~10.0 之间大的变化

- 参考回答:
  - **Android5.0 新特性**
    - **MaterialDesign 设计风格**
    - **支持 64 位 ART 虚拟机**（5.0推出的ART虚拟机，在5.0之前都是Dalvik。他们的区别是：Dalvik,每次运行,字节码都需要通过即时编译器转换成机器码(JIT)。 ART,第一次安装应用的时候,字节码就会预编译成机器码(AOT)）
    - 通知详情可以用户自己设计
  - **Android6.0 新特性**

        - **动态权限管理**
        - 支持快速充电的切换
        - 支持文件夹拖拽应用
        - 相机新增专业模式

    - **Android7.0 新特性**

        - **多窗口支持**
        - **V2 签名**
        - 增强的Java8语言模式
        - 夜间模式

    - **Android8.0（O）新特性**

        - **优化通知**：通知渠道(Notification Channel)通知志 休眠 通知超时 通知设置 通知清除
        - **画中画模式**：清单中Activity设置android:supportsPictureInPicture
        - **后台限制**
        - 自动填充框架
        - 系统优化
        - 等等优化很多

    - **Android9.0（P）新特性**

        - **室内 WIFI 定位**
        - **“刘海”屏幕支持**
        - 安全增强
        - 等等优化很多

    - **Android10.0（Q）目前曝光的新特性**

        - **夜间模式**：包括手机上的所有应用都可以为其设置暗黑模式。
        - **桌面模式**：提供类似于PC的体验，但是远远不能代替PC。
        - **屏幕录制**：通过长按“电源”菜单中的"屏幕快照"来开启。

13、说下 Measurepec 这个类

- 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：

子视图布局参数 (LayoutParams) \ 父视图测量模式 (mode)	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

14、请例举 Android 中常用布局类型，并简述其用法以及排版效率

- 参考回答：
  - Android中常用布局分为**传统布局**和**新型布局**
    - 传统布局（编写XML代码、代码生成）：
      - **框架布局（FrameLayout）**：
      - **线性布局（LinearLayout）**：
      - **绝对布局（AbsoluteLayout）**：
      - **相对布局（RelativeLayout）**：
      - **表格布局（TableLayout）**：

- 新型布局（**可视化拖拽控件**、编写XML代码、代码生成）：
  - **约束布局（ConstrainLayout）**：

类型	特有属性	作用	具体使用

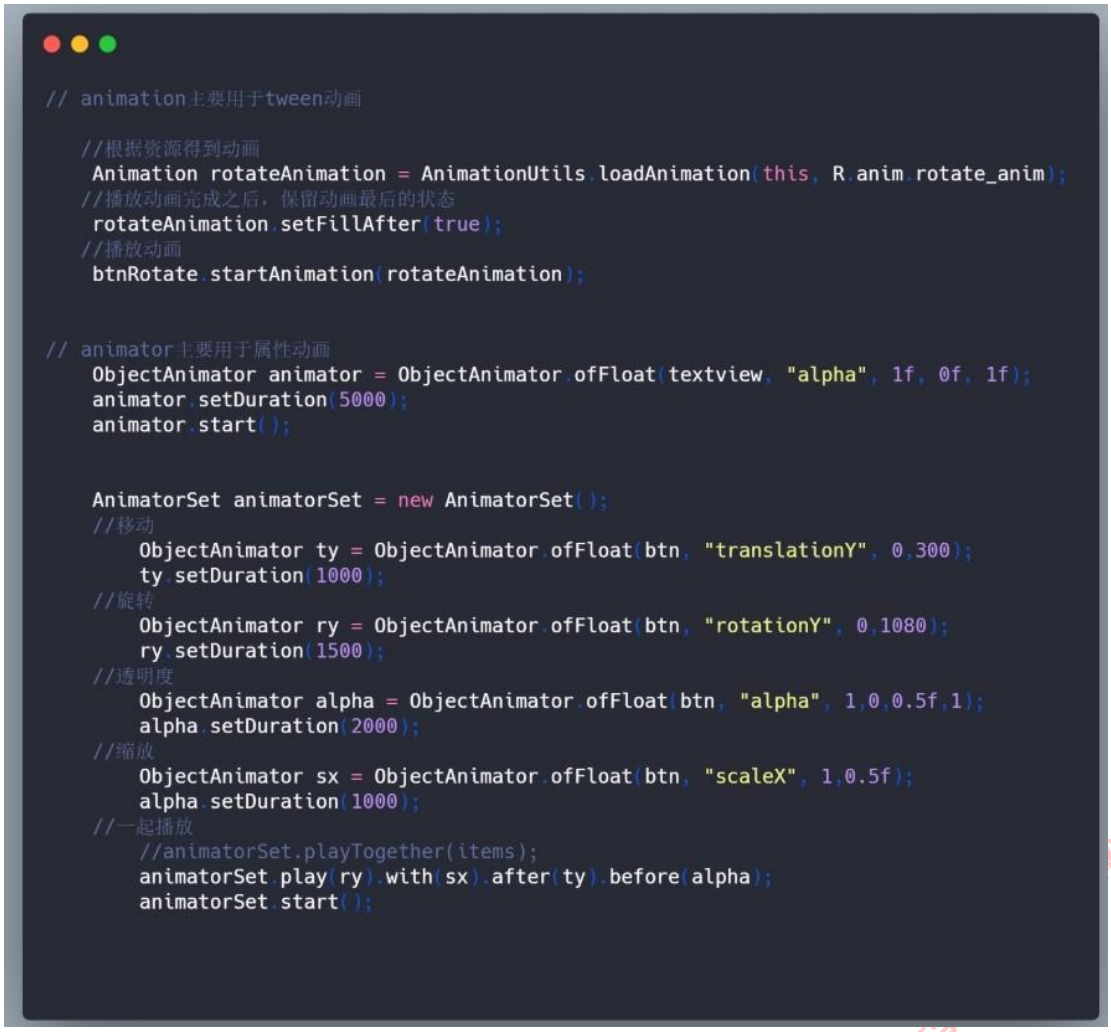
线性布局 (LinearLayout)	orientation	设置布局内控件的排列方式 (水平、垂直)	android:orientation="vertical"; // 垂直排列（默认）
			android:orientation="horizontal"; // 水平排列

	layout_weight	根据设置的权重 将布局的空间按 比例分配 (计算公式：控件宽度 = 控件设置宽度 + 剩余空间所占百分比宽幅)	android:layout_weight="1.0"
相对布局 (RelativeLayout)	layout_alignParentX	当前控件 对齐 父控件 的X方位	android:layout_alignParentTop="true" // 当前控件顶端 对齐 父控件顶端 android:layout_alignParentBottom="true" // 当前控件底端 对齐 父控件底端 android:layout_alignParentLeft="true" // 当前控件左端 对齐 父控件左端 android:layout_alignParentRight="true" // 当前控件右端 对齐 父控件右端 android:layout_centerHorizontal="true" // 当前控件 位于 父控件的水平方向中间位置 android:layout_centerVertical="true" // 当前控件 位于 父控件的垂直方向中间位置 android:layout_centerInParent="true" // 当前控件 位于 父控件的正中间位置
	layout_X	当前控件 位于 某控件的X方位	android:layout_above="@+id/AA" // 当前控件 位于 AA控件的上方 android:layout_below="@+id/AA" // 当前控件 位于 AA控件的下方 android:layout_toLeftOf ="@+id/AA" // 当前控件 位于 AA控件的左方 android:layout_toRightOf ="@+id/AA" // 当前控件 位于 AA控件的右方 android:layout_alignBottom ="@+id/AA" // 当前控件的底部 对齐 AA控件的底部 android:layout_alignLeft ="@+id/AA" // 当前控件的左侧 对齐 AA控件的左侧 android:layout_alignRight ="@+id/AA" // 当前控件的右侧 对齐 AA控件的右侧 android:layout_alignTop ="@+id/AA" // 当前控件的上方 对齐 AA控件的上方
绝对布局 (AbsoluteLayout)	layout_x	指定控件的x坐标	android:layout_x="50dip"
	layout_y	指定控件的x坐标	a ndroid:layout_y="100dip"
表格布局 (TableLayout)	• TableLayout的行（TableRow） = 1个水平排列的线性布局（LinearLayout） • 继承自 线性布局（LinearLayout） • 故具备线性布局（LinearLayout）的所有属性		
框架布局 (FrameLayout)	只具备基础属性		

- 对于嵌套多层View而言，其排版效率：**LinearLayout = FrameLayout >> RelativeLayout**

15、区别 Animation 和 Animator 的用法，概述其原理

- 参考回答：
  - **动画的种类**：前者只有**透明度**，**旋转**，**平移**，**伸缩** 4种属性，而对于后者，只要是该控件的属性，且有setter该属性的方法就都可以对该属性执行一种**动态变化**的效果。
  - **可操作的对象**：前者只能对 **UI 组件**执行动画，但属性动画几乎可以对**任何对象**执行动画（不管它是否显示在屏幕上）。
  - **动画播放顺序**：在Animator中，AnimatorSet正是通过playTogether()、pl aySequentially()、animSet.play().with()、before()、after()这些方法来控制多个动画协同工作，从而做到对动画播放顺序的精确控制



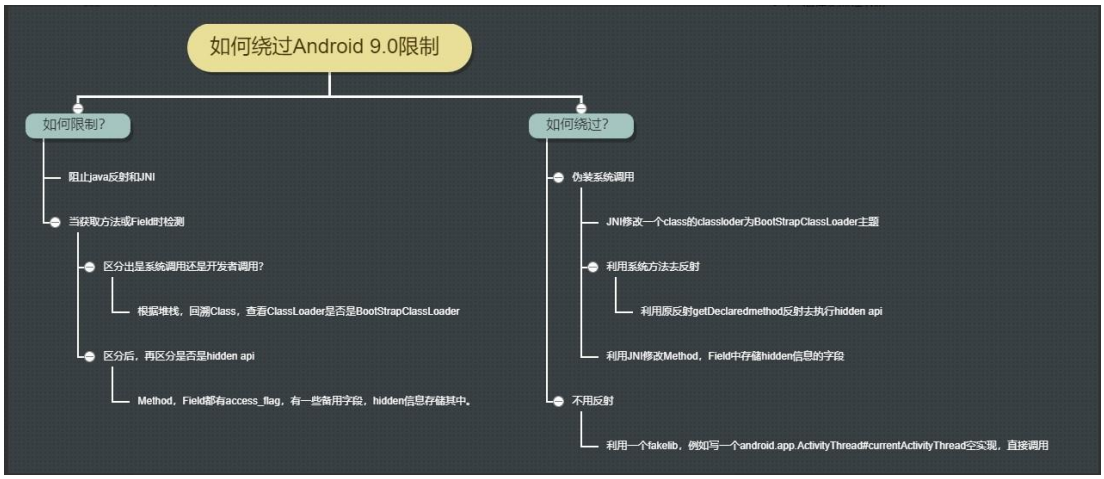
```
// animation主要用于tween动画

    //根据资源得到动画
    Animation rotateAnimation = AnimationUtils.loadAnimation(this, R.anim.rotate_anim);
    //播放动画完成之后，保留动画最后的状态
    rotateAnimation.setFillAfter(true);
    //播放动画
    btnRotate.startAnimation(rotateAnimation);


// animator主要用于属性动画
    ObjectAnimator animator = ObjectAnimator.ofFloat(textview, "alpha", 1f, 0f, 1f);
    animator.setDuration(5000);
    animator.start();


    AnimatorSet animatorSet = new AnimatorSet();
    //移动
        ObjectAnimator ty = ObjectAnimator.ofFloat(btn, "translationY", 0,300);
        ty.setDuration(1000);
    //旋转
        ObjectAnimator ry = ObjectAnimator.ofFloat(btn, "rotationY", 0,1080);
        ry.setDuration(1500);
    //透明度
        ObjectAnimator alpha = ObjectAnimator.ofFloat(btn, "alpha", 1,0,0.5f,1);
        alpha.setDuration(2000);
    //缩放
        ObjectAnimator sx = ObjectAnimator.ofFloat(btn, "scaleX", 1,0.5f);
        alpha.setDuration(1000);
    //一起播放
        //animatorSet.playTogether(items);
        animatorSet.play(ry).with(sx).after(ty).before(alpha);
        animatorSet.start();
```

16、使用过什么图片加载库？Glide 的源码设计哪里很微妙？

- 参考回答：
  - 图片加载库：**Fresco、Glide、Picasso** 等
  - Glide 的设计微妙在于：
    - **Glide 的生命周期绑定**：可以控制图片的加载状态与当前页面的生命周期同步，使整个加载过程随着页面的状态而启动/恢复，停止，销毁
    - **Glide 的缓存设计**：通过（三级缓存，Lru 算法，Bitmap 复用）对 Resource 进行缓存设计
    - **Glide 的完整加载过程**：采用 Engine 引擎类暴露了一系列方法供 Request 操作

17、如何绕过 9.0 限制？

- 参考回答：



18、用过哪些网络加载库？OkHttp、Retrofit 实现原理？

- 参考回答：
    - 网络加载库：OkHttp、Retrofit、xUtils、Volley等

19、对于应用更新这块是如何做的？（灰度，强制更新、分区域更新）

- 参考回答：
    - **内部更新**：
        - 通过接口获取线上版本号，versionCode
        - 比较线上的 versionCode 和本地的 versionCode，弹出更新窗口
        - 下载 APK 文件（文件下载）
        - 安装 APK
    - **灰度更新**：
        - 找单一渠道投放特别版本。
        - 做升级平台的改造，允许针对部分用户推送升级通知甚至版本强制升级。
        - 开放单独的下载入口。
        - 是两个版本的代码都打到 app 包里，然后在 app 端植入测试框架，用来控制显示哪个版本。测试框架负责与服务器端api 通信，由服务器端控制 app 上A/B版本的分布，可以实现指定的一组用户看到 A 版本，其它用户看到 B 版本。服务端会有相应的报表来显示 A/B 版本的数量和效果对比。最后可以由

服务端的后台来控制，全部用户在线切换到A或者B版本~

- **无论哪种方法都需要做好版本管理工作，分配特别版本号以示区别。当然，既然是做灰度，数据监控（常规数据、新特性数据、主要业务数据）还是要做到位，该打的数据桩要打。还有，灰度版最好有收回的能力，一般就是强制升级下一个正式版。**

- **强制更新**:
  - 一般的处理就是进入应用就弹窗通知用户有版本更新，弹窗可以没有取消按钮并不能取消。这样用户就只能选择更新或者关闭应用了，当然也可以添加取消按钮，但是如果用户选择取消则直接退出应用。

- **增量更新**:
  - 二进制差分工具 bsdiff 是相应的补丁合成工具，根据两个不同版本的二进制文件，生成补丁文件.patch 文件。通过bspatch 使旧的apk 文件与不定文件合成新的apk。 注意通过 apk 文件的md5 值进行区分版本。

20、会用 Kotlin、Fultter 吗？ 谈谈你的理解

- 参考回答:
  - Kotlin是一种具有类型推断的跨平台，静态类型的通用编程语言。　Kotlin旨在与Java完全互操作，其标准库的JVM版本依赖于Java类库，但类型推断允许其语法更简洁。
  - Flutter是由Google 创建的开源移动应用程序开发框架。它用于开发 Android和iOS 的应用程序，以及为 Google Fuchsia 创建应用程序的主要方法
  - 关于kotlin 的重要性，相信大家在日常开发可以体会到，应用到实际开发中，需要避免语法糖（例如单列模式、空值判断、高阶函数等）
  - 至于Flutter，目前Google 官方文档还不完善，市面上采用此语言编写的项目较少，如需要具体深入，请参考闲鱼和官方文档