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
- o 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)
 - o 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
 - o 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 Reference Queue 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 Reference 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);

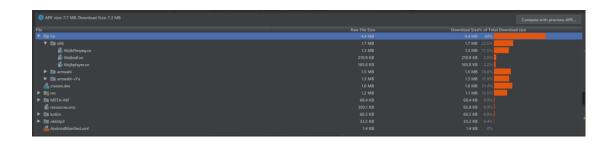
    System.err.println("软引用的对象 ----->" + softReference.get());

    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");
        //清測softReference
    }
}
```

3、Apk 的大小如何压缩?

- 参考回答:
 - o 一个完整 APK 包含以下目录(将 APK 文件拖到 Android Studio):
 - 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,所以这三块是优化 Apk 大小的重点(实际情况不唯一)

。 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"] }
        { manifestPlaceholders = [UMENG CHANNEL VALUE: "UM"] }
   C360 { manifestPlaceholders = [UMENG CHANNEL VALUE: "C360"] }
        { manifestPlaceholders = [UMENG CHANNEL VALUE: "BD"] }
        { manifestPlaceholders = [UMENG CHANNEL VALUE: "HW"] }
   HW
   PP
        { manifestPlaceholders = [UMENG CHANNEL VALUE: "PP"] }
        { manifestPlaceholders = [UMENG CHANNEL VALUE: "XM"] }
   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 出所有渠道的 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。
 - o 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
- o 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

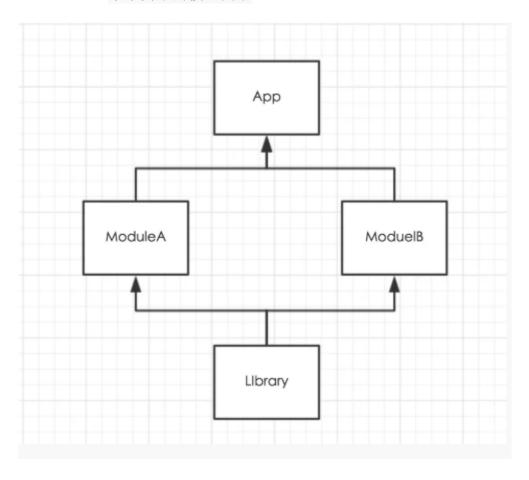
- 原理在于通过反射将插件 apk 的路径加入 AssetManager 中并创建 Resource 对象加载资源, 有两种处理方式:
 - 合并式: addAssetPath 时加入所有插件和主工程的路径; 由于 AssetManager 中加入了所有插件和主工程的路径,因此生成的Resource 可以同时访问插件和主工程的资源。但是由于主工程和各个插件都是独立编译的,生成的资源 id 会存在相同的情况,在访问时会产生资源冲突。
 - 独立式:各个插件只添加自己 apk 路径,各个插件的资源是互相隔离的,不过如果想要实现资源的共享,必须拿到对应的 Resource 对象。

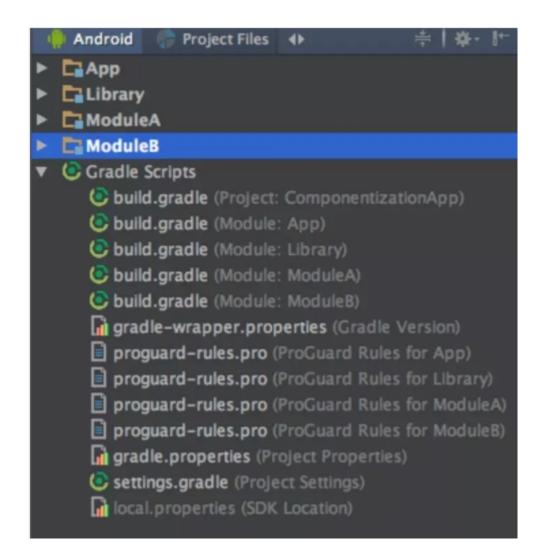
6、组件化原理

- 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)。

。 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。

o 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

		ė.				6
				E EArmeer	東京机件化力室	
对比项	CC	#8≸(DDComponentForAndrold	ModularizationArchitecture	(阿上很多组件化方案的路由引擎)	(基于[東東speter)	ActivitySouter
开解时间	2017-11	2017-9	2017-1	2016-12	2016-9	2016-4
通信机制	相件品株	務会 - 横口下区	担件总裁	落由 - 横口下沉	路由 + 接口下沉	施由 ・ 蒙古方法
octivity#M#0		8	2	100	8	100
最近支持等保险机		×				
ALII ANY PROGRAM	-	1. 通过mr生成自动作入代码		1. 通过apt生成解析参数的代码	•	-
activity变量自动往入	×	2. 在osCreate中範囲 AutowiredService.Factory.getInstance().create().autowire(this);波者继承SeseActivity	×	2. 在osCreate方法申请用 「ABouter, getInstance(), inject(this); 实现自动注 入	×	×
startActivityForResult	支持Activity/Fragment, 但不建议使用Ger)建议使用统一 的组件课程方式	仅支持Activity	仮支持Activity	仮支持Activity	支持Activity/Fragment	仮支持Activity
調用方式(页面跳转)	用步点性的景成排步相调检索: CCEment remuit = CC. obtainful indr ComponentA*). build().call(); all ring call(d = C. obtainful ider (ComponentA*), build().callAsync(oe * IComponentCallback()());	omdetivityResult短時執限: "URbuter.getInstance().spemlri(getHativity(), url, bundle);"	ROUTEFREQUEST, OCCASIO (MAISTACTIVITY, UNIA)	onhetivityRoumli是把抽集: Alouter.getlimtmcccO.build("/test/activity"). narigation();	onlettvityflesuit延延度; 'Bouter.create(well.openfcontext);'	onActivityResult返回体施; 'Rowters.opem(context, wrl);
调用方式(调用服务)	与其面视和相同	*Router.getInstance().getService(ReadBookService.elass.getSimpleName())	与页面教料相同	'ARouter.getlastance().navigetion(HelloService.class).sayHello();'	'PipeManager.get (LogisPipe.class).logout();'	与实施物和同
机件向外提供服务	与页面跳转一致。在1Component中实现	接口下沉到base中,组件中实现接口并在LipplicationLike中等加代码 注册到kuter中	与页面魏制一致,实现一个对应的Action并在其所属的Provider 中往册即可	并通过注解來暴露服务	機口下記動sse中、銀件中实現機口并在 ApplicationSelegate中兵権口管理委託層 Pipefinnager.register(GarePipe.class, new CorePipelapi());	在静态方法上加拉解來暴露服务。但不支持返回 且参数限定位(context, bundle)
Fragment机件化支持	在IComponent中实现,并支持后续Fragment內部功能資用	调用服务的方式实现。未支持后被Fragment内部的功能调用	不支持	调用服务的方式实现。未支持后续Fregment内部的功 使调用	调用服务的方式实现。未支持后被Fragment内部的功能调用	不支持
和件自构注册方案	TransfersMVI + ASB打解机件类(IComponent接口实现类) 并注册到ComponentReseager中,无罪于动物护机件列表	an生成各modole的解音表 Transformが、juvassist等inpolicationLike的化量代码生成到白定 又application.onCreate分析中,光常子动物产组件到表	未支援目和性症。 1. Action有效位其所属的Provider中注册 2. Provider可混形置的SpolicationLegic中性册 3. ApplicationLegic在主动p的Application中注册	1. apt生成各module的落由表 2. Aroute相對化對古提所有dex提出指定包名下的降 由表。通过反射进行使一位册	apt生成各module的等自表pkg. ReuterBuleCreator美 2. 在ComponentPackages中東文原有ReuterBuleCreator美 3. 在BaseAgplication中反射所有的包名規則所有場由表 RouterBuleCreator 4. 電影を過程が認知のentFackages美中的包名列表	1. apt生成各module的两点表 2. apt在application的module通过Modules注解 MouterInit进行注册 3. 函数手向维护Modules注解中的紹件所表
机件单数运行的方式	初表ibheary/agalication方式権等。提供2時方式。 1. module/build.gradie中切扱 ext.runkskpertrusbfelse 2. 在local.properties中切扱 moduleMane-trusbfelse (作者を提出方がた、不会を変え折く取らなキャ)	切換library/application方式論译。在module/gradle.properties中間 表 isSunklionce-trueStfalse	切挽librury/application方式编译。模型本身没有要供切换方式,并没否自行解决	切換librury/application方式輸導。框架本身役有提供切換方式。开发者自行解決	配件acdale燃料以litrary方式编译。概析提供app完于,可以按定用多个组件依赖近来一起打包。 按定用多个组件依赖近来一起打包。 按处是所有数件调试时包名制同,整误足分享及地图等第三方 UKU时包名的要求	切換library/application方式輸译。框架本身 供切換方式。开发者自行解液
跨spo机件调用支持	3	×	3	×	×	2
跨spp调用开关及权限设置		/	×	/	/	×
机件epplis行时调用其它机 作	组件问时安装在设备上即可,实际开发中一般是当的正在 开发的组件和主spor中的组件互相调用。 通过广播 Sprice + LocalSocket实现。没有UrlSchene 调用好算出的对话选择程	将用要调用的组件一起打包才能调用	批件同时安装在设备上即可。实际开发中一般是当前正在开发的 批件制 Lupy中的批件发程调准。 通过AIL实现	一起打包或者通过uriSchone来统一转发	帝司英调用的机件—起打包才能调用	EriSchene原生支持跨app項局,但件问时安装在 即可 通过中介Activity转发:RouterActivity
组件依赖指挥	无背极赖、完全隔离	通过插件实现只在打 _印 à包时才添加依赖。福码期间不能直接调用其它组件的代码	无罪佐赖、完全隔离	未隔离	未開客	无需依赖、完全隔离
ANF支持	打電器 · 划件内部Action进行AIP	×	组件內部Action进行ADP	打破器ACP	×	×
##B	0	×	×	2		×
SHORESHIP TO		×	×	×	×	×
创作课程的取消		×	×	×	×	×
动态注册/注解批件	<u> </u>	<u> </u>	×	×	×	×
10,42	可以跨worliki和,初期改选的印可单数编译组件记行 股份统一的组件询明及实现方式(不管是否跨worlin),直接的统一、服务调用,可加/排步访明) 即行自由证明,无理能学、无理能学、无理能学、无理能学、无理能学、表现能学、表现能学、表现能学、表现的工作。 取代了ActionProcessor包需用的支持	1. 鐵码期间机件依赖道过插件进行隔离。避免直接调用其它机件的代码 2. 领售了需等场动ter的方案 3. 机件自动记录,无常维护	3. 组件累积定同步实现还是异步实现。调用组件时统一拿到		配件modale可以处件以library方式编译。由统一的mp分子未 安备词说。不需要切象library/mplication编译方式、避免 了第三方508高更指定包包的问题。自定及程度重复导致安装 冲突的问题。误操作导致pk upload到meres包字的问题等	1. 业内最早的银行化支持库 2. 通过注解费布方法的方式暴露推务
柜件定义代码侵入性	新增1Component接口的实现类来定义组件。 侵入性低	注解定义路由及参数自动往入。侵入性高	新增接口实现类。使入性低	注新定义路由及参数自动往入。 侵入性高	拉懈定义路由,侵入性高	注解定义路由。侵入性高
机件调用代码侵入性	a	ä	*		a	ā
現所配置		所有下記接口、框架中相关接口的实现类等	'-dontwarn com. spinytech. **	就是中的所有委员规型相关培口的实现委	所有WouterRuleCreator要	概集中的所有着
	44	-60	A STATE OF S	一般 一般	# # HOUNCERSTON TO	60 ACT 123 OF 13 ACT
ALC: U CO ALC: A LEAD			*			
老项目改造成本评价						
方案使用的学习成本评估	16					-
	15, 15, 595941523		A	65 592278657 / 336755078	一般 108895031	

8、组件化中路由、埋点的实现

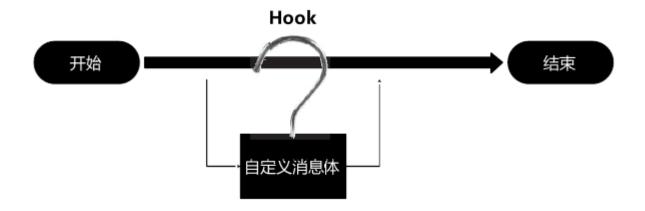
• 参考回答:

- 。 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)
- o 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)



- o 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,对系统其它 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:
 - o 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 函数, 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

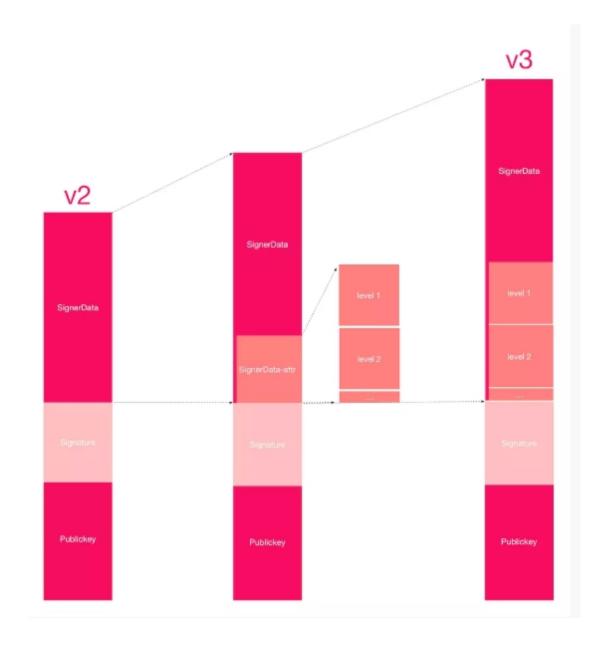
11、v3 签名 key 和 v2 还有 v1 有什么区别

• 参考回答:

o 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。



- o 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
- o 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:
 - o 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 先编译成机器码(AOT))
 - 通知详情可以用户自己设计
 - o 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
- o 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
- o Android8.0(0)新特性
 - 优化通知: 通知渠道 (Notification Channel) 通知标志 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
- o Android9.0(P)新特性
 - 室内 WIFI 定位
 - "刘海"屏幕支持
 - 安全增强
 - 事等优化很多
- o 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:

父视图测量模式 子视图布局参数 (mode) (LayoutParams)	EXACTLY	AT_MOST	UNSPECIFIED
具体数值(dp / px)	EXACTLY + childSize	EXACTLY + childSize	EXACTLY + childSize
match_parent	EXACTLY + parentSize (父容器的剰余空间)	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:
 - o 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	orientation	设置布局内控件的排列方式 (水平、量宜)	android:orientation="vertical"; // 垂直排列(默认) android:orientation="horizontal"; // 水平排列
(LinearLayout)	layout_weight	根据设置的权重 将布局的空间按 比例分配 (计算公式: 控件复度 = 控件设置宽度 + 剩余空间所占百分比宽幅)	android:layout_weight="1.0"
	layout_alignParentX	当前控件 对齐 父控件 的X方位	androidiayout_alignParentTop="true" // 当前控件顶端 对齐 父控件顶端 androidiayout_alignParentBottom="true" // 当前控件底端 对齐 父控件底端 androidiayout_alignParentLeft="true" // 当前控件左端 对齐 父控件右端 androidiayout_alignParentRight="true" // 当前控件石端 对齐 父控件右端 androidiayout_centerHorizontal="true" // 当前控件 位于 父控件的本于向中间位置 androidiayout_centerHorizontal="true" // 当前控件 位于 父控件的垂直方向中间位置 androidiayout_centerInParent="true" // 当前控件 位于 父控件的正中间位置
相对布局 (RelativeLayout)	layout_X	当前控件 位于 某控件的X方位	android/ayout_above="@-id/AA" // 当前控件 位于 AA控件的上方 android/ayout_below="@-id/AA" // 当前控件 位于 AA控件的下方 android/ayout_toLetfOf="@-id/AA" // 当前控件 位于 AA控件的方方 android/ayout_toRightOf="@-id/AA" // 当前控件 位于 AA控件的方方 android/ayout_alignBottom="@-id/AA" // 当前控件的底部 对齐 AA控件的后部 android/ayout_alignLeft="@-id/AA" // 当前控件的无侧 对齐 AA控件的后即 android/ayout_alignRight="@-id/AA" // 当前控件的方侧 对齐 AA控件的后侧 android/ayout_alignTop="@-id/AA" // 当前控件的方则 对齐 AA控件的后侧 android/ayout_alignTop="@-id/AA" // 当前控件的上方 对齐 AA控件的上方
绝对布局	layout_x	指定控件的x坐标	android:layout_x="50dip"
(AbsoluteLayout)	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)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)。
 - o **动画播放顺序**: 在 Animator 中,AnimatorSet 正是通过 playTogether()、playSequentially()、 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

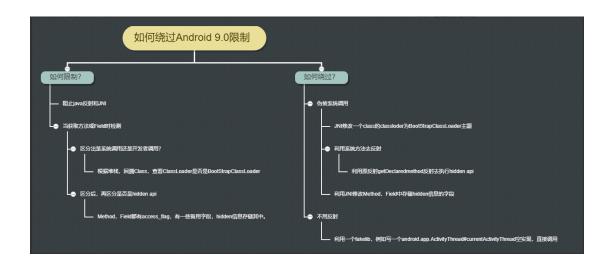
```
• • •
    Animation rotateAnimation = AnimationUtils.loadAnimation(this, R.anim.rotate_anim)
    rotateAnimation.setFillAfter(true)
    btnRotate.startAnimation(rotateAnimation)
    ObjectAnimator animator = ObjectAnimator ofFloat(textview, "alpha", 1f, 0f, 1f);
    animator.setDuration(5000)
    animator.start(
    AnimatorSet animatorSet = new AnimatorSet();
        ObjectAnimator ty = ObjectAnimator.ofFloat(btn, "translationY", 0,300);
        ty.setDuration(1000)
        ObjectAnimator ry = ObjectAnimator.ofFloat(btn, "rotationY", 0,1080);
       ry.setDuration(1500)
        ObjectAnimator alpha = ObjectAnimator.ofFloat(btn, "alpha", 1,0,0.5f,1);
       alpha.setDuration(2000)
        ObjectAnimator sx = ObjectAnimator.ofFloat(btn, "scaleX", 1,0.5f);
       alpha.setDuration(1000)
       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:
 - o 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:
 - o 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,分配特别的版本号以示区别。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)
- o 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