**1. 代码质量检测工具与代码质量管理机制(QA) -454**

以前：项目代码质量检测管理的手段很单一；通过在wiki文档中定义好具体的代码规则与注释模板，通过checkStyle工具进行代码检测(团队定义了些规则)，还有就是代码合并时进行人工审核；整体实现效果不明显，实施意愿不太高。

现在：重写构建代码检测流程，

1. **Gradle构建优化 -550**

（1）这是之前项目开发中发现的一个问题，Gradle本地构建时间会很长，构建起来会很慢，其实究其原因最核心的问题是我们在项目中集成了太多构建相关的插件，像APM、性能日记采集、神策埋点、注解处理器等等，这些插件都会在我们构建期间生成各种task，然后在构建期间去执行它。还有像注解处理器这种还会在编译期间对代码进行注入； 我的一个解决方案就是生成不同的变体，Gradle有一个变体的概念，就是在不同环境不同变体下对Gradle生成不同的配置；从而删减调一些不必须的插件；像我们开发过程中都会涉及到几个环境，像测试、灰度、正式，每个环境其实对插件的依赖都不同，比如在本地环境构建的时候仅仅想看代码运行效果，那就可把其他不相干的构建插件删减掉，从而提高我们的构建效率；

（2）还有其他的像代码检测、资源文件检测、图片优化等等不需要用的话都可以相应的关闭，也是能够提高本地构建速度的。

（3）还有依赖尽量不要使用动态版本依赖，这样也会每次去远程拉取导致构建变慢。。

gradle的变体可以使用一套代码打出不同的应用程序包，我们可以通过设置一个fastDevelop变体，关闭混淆、删减一些开发时非必须的插件(APM、性能监控、lint静态代码检测、系统图片检测、神策三方构建插件)，从而提高编译速度。但是安全与效率不可兼得，这个变体只在开发过程中使用，在提交给测试人员、发布时还是使用原来的变体。

变体构建主要对build.gradle进行修改，对正常的开发影响不大。但因为增加了一个变体，所以需要开发人员了解各个变体的作用，并在需要时手动切换。

Gradle的apply from 和 apply plugin 区别：

apply from：从本地或URL找到相应Plugin，直接加载进来；

apply plugin：从Maven服务器找到相应的plugin，调用对应的Plugin.apply()； --Plugin插件开发；

自定义Plugin：实际上是在Plugin.apply()上跑各种task(代码扫描、注解处理-代码注入、)；

1. **插件化技术**

插件化的核心是，就是把复杂度很高的业务模块独立成插件，然后根据不同的业务需求对插件模块进行动态替换/动态加载，提供一种实时修复的手段；这有两个概念：一个叫宿主(也就是我们的apk)另一个是插件（也就是我们理解的patch补丁）；像插件化技术最常见的应用就是我们的热更新/热修复，常见的就是我们的tinker热修；

像一般的原理，比如tinker在合并完补丁生成补丁dex后，Tinker是把dex单独存放在本地目录下，在启动时通过反射去获取Application的ClassLoader中的 pathList 字段，然后将 patch 包里面的 dex与基础包的 dexElements(有序数组，存放类信息) 进行合并，最终以此达到类修复手段。 其实就是把补丁包对应的dex插入到原来有序数组dexElements的最前面了。

App启动源码，Application初始化时（ActivityThread.handleBindApplication）会初始化pathClassLoader；

1. **性能分析工具：**

SysTrace：分析方法执行耗时、布局加载耗时；

Layout Inspector：分析布局层级或者布局嵌套情况；

Memory profiler(堆转储:将当前堆内存情况进行快照)：分析当前应用内存状况，检测内存泄漏的情况；

1. **性能优化：**

首页启动速度优化：

布局构建速度优化：

Activity页面启动速度优化：

内存优化(堆转储)：内存泄漏(核心生命周期长的对象持有生命周期短的对象引用，导致短生命对象没被及时回收；静态变量、单例持有；属性动画一直循环执行没有关闭；)

自定义控件优化：避免在频繁调用的方法如onDraw上或动画过程中进行内存分配导致频繁GC卡顿；减少invalidate、requestLayout；保持View整体层级的扁平化，提高构建速度；

1. **属性动画：**

创建属性动画，指定区间数跟动画时间、监听、启动动画、、、计算已完成的动画分数 - 根据插值器改变动画分数、根据这个动画分数计算具体的 Value，监听返回；

1. **Dalvik：**

JIT(just in time)实时编译（运行时）,运行的时候将字节码翻译成机器码,所运行的目标文件与硬件平台无关,app运行效率低

ART:

AOT(Ahead of time)预先编译（安装时）,运行前将字节码翻译成机器码,所运行的目标文件(oat)与硬件平台相关.app运行效率高。但会占用空间。APK安装所需时间增加

**public Class SingleTon{**

**private Static volatile SingleTon mSingleTon;**

**private void SingleTon(){}**

**public Static SingleTon getInstance(){**

**if (mSingleTon == null){**

**synchronized (SingleTon.class){**

**if (mSingleTon == null){**

**mSingleTon = new SingleTon();**

**}**

**}**

**}**

**return mSingleTon;**

**}**

**}**

1. **JetPack：**

**一种概念，可以理解为Google官方推荐的一种开发模式或一种开发的规范，Jetpack集成了各类三方库、AAC组件（LifeCycle、LiveData、ViewModel、Room...）、这些东西都是Google未来大力支持并持续维护的东西。**

1. **Kotlin协程：**

**理解为在主线程与子线程中间加入一个协程管理的管道、消除传统CallBack。**

**GlobalScop。Launch(Dispatcher.Main) {**

**Val value1 = request1(params)**

**Val value2 = asyn {**

**request2(params)**

**}**

**Val value3 = asyn {**

**Request3(params)**

**}**

**updateUI(value2.await(), value3.await())**

**}**

**Suspend fun request1(params :Parameter) {**

**withContext(Dispatchers.io) {**

**repository.getListData()**

**}**

**}**

**10、技术分享分享过什么？**

1. **Kotlin协程的使用和封装。**
2. **Jetpack上AAC组件的介绍、比如LiveData跟ViewModel结合使用对比替换本地项目的Presenter的可行性。ViewModel中Repo数据返回的Model需要进行BaseModel封装逻辑(State、ErrorCode、ErrorMsg、泛型Data)。**
3. **JetPack上AAC组件，Databinding+LiveData+ViewModel绑定UI控件更新，注意一点xml不能做任何逻辑动作，只能绑定具体的view到ViewModel上，然后ViewModel上数据被更新时，通知LiveData然后自动更新所绑定的View。**

**RxJava优势在于它是基于事件流的处理逻辑，比如奇数的图片直角展示、偶数的图片要圆角展示。**

**Kotlin协程优势在于线程切换、异步管理，比如一个页面展示依赖三个接口，Kotlin协程能够通过asyn.await() 并行的处理三个接口而且逻辑清晰。**

****PS：协程是一种并发设计模式，您可以在 Android 平台上使用它来简化异步执行的代码。****

**所以：项目中可以将RxJava的线程切换逻辑更替为Kotlin协程来实现，RxJava专注去处理特定事件流的逻辑。**

****11、尝试去做一个网络框架（HttpUrlContextion、okHttp）：****

1. **底层通过Socket实现。**
2. **考虑线程创建、线程切换的问题、线程池的使用 (连接池的管理)。**
3. **考虑异常处理机制。**
4. **考虑数据缓存机制（缓存对于移动端是非常重要的存在、减少请求次数，减小服务器压力、本地数据读取速度更快、在无网络的情况下提供数据）。**
5. **数据返回的格式处理Converter。**
6. **完善API、暴露上层更易使用的接口方法。**
7. **Https。**

