技术背景

启动速度尤其是安装后首次启动速度，对用户留存率的影响很大。启动速度越快，用户体验越好，流失的用户也就越少。

启动分类

冷启动、热启动。主要区别在于热启动不需要创建和初始化Application，启动速度较快。

TalkU启动现状

在 Alcatel_4060 （Android5.0）上TalkU首次启动时间（从Launch点击icon到完全展示Welcome界面）大致需要17s+！！！可谓是惨不忍睹。

优化思路

统计APP启动流程中各操作的耗时情况，找出耗时”大户”，分析是否可以优化。

APP启动流程回顾

Application的构造器方法—>attachBaseContext()—>onCreate()—>Activity的构造方法—>onCreate()—>配置主题中背景等属性—>onStart()—>onResume()—>测量布局绘制显示在界面上

TalkU启动流程

DTApplication—>SplashActivity—>WelcomeActivity

耗时分析工具

Traceview。用法网上很多，这里不再介绍。

TalkU耗时情况概览

首次启动耗时统计 冷启动耗时统计 优化策略及case

优化过程

一、loadDtPluginApk优化

需要了解的背景知识:

1.Dalvik和Art。 请参考Dalvik,ART与ODEX相爱相生

2.MultiDex的由来。 APK文件包含 Dalvik Executable (DEX) 文件形式的可执行字节码文件， Dalvik Executable 规范（ dex指令是用16位寄存器来保存dex中的方法数 ）将可在单个 DEX 文件内可引用的方法总数限制在 65,536，其中包括 Android 框架方法、库方法以及您自己代码中的方法。为了突破应用总方法数限制，Google提供了MultiDex支持库（具体如何适配这里不再赘述），适配后APK文件里可以生成多个dex文件。

3.MultiDex拖慢启动速度。确切地说，MultiDex拖慢的是Android5.0以下版本APP的启动速度。由于 Android 5.0（API 级别 21）及更高版本上， ART 在应用安装时执行预编译 ，扫描 classesN.dex 文件，并将它们编译成单个 .oat 文件，供 Android 设备执行，所以安装过程会比较缓慢，执行速度比较快；而Android 5.0以下classesN.dex（N>=2）文件的加载及DexOpt过程需要在首次启动时完成，所以会拖慢首次启动速度。

4.loadDtPluginApk来历。在2015年左右，市场上大部分Android机型还处于5.0以下，MultiDex机制在大部分机型上会拖慢APP的启动速度。有鉴于此，Edward采用了loadDtPluginApk方案：即将广告相关的业务单独打包成一个插件apk，然后在attachBaseContext里动态加载插件apk，把广告业务相关的类加载进来。动态加载插件apk相比于MultiDex机制，省去了dex的验证操作，相比MultiDex机制提升了启动速度。然而在5.0以上，这样做反而会弄巧成拙，大大拖慢启动速度。因为在APP安装时就可以把dex的加载验证优化都搞定，没必要多此一举放在启动时来执行。基于现在市面上Android5.0以下机型已经不多，决定弃用此方案。

二、loadDtPluginApk替代方案

1.Android5.0及以上：弃用插件APK，将广告业务整合到主工程，直接采用MultiDex方案；

2.Android5.0以下：弃用插件APK，将广告业务整合到主工程，直接采用MultiDex方案；由于5.0以下由于加载MultiDex巨慢，再加上机器配置又低，更是慢上加慢，决定在APP首次启动的闪屏页添加进度条，减少用户等待的焦虑感，提升用户体验。

3.由于4.4系统已经可以手动将Dalvik模式切换到Art模式，因此不能单纯通过Build.VERSION是否大于等于21来判定系统是否本身就支持MultiDex机制，而是通过*isVMMultidexCapable*方法来判定，该方法源自MultiDex，更多详情可查看MultiDex源码；

4.对于*isVMMultidexCapable*为true的情况，直接在attachBaseContext里调用MultiDex.*install*(this)即可；

5.对于*isVMMultidexCapable*为false的情况，要先判定之前是否已经执行过dex初始化操作（这里用SharedPreferences来保存是否，键为AppVersion，值为是否已执行过dex初始化操作，如果是，简单执行MultiDex.*install*(this)即可；否则还需显示在SplashActivity显示进度条等待加载完成。

三、关于显示加载进度条（N多坑！！！）

1.首先，要知道，冷启动的时候，当我们点击桌面上的icon，可以马上看到TalkU的闪屏页，事实上，此时对应的Acticvity实例并没有创建好，甚至连Application实例都没创建好，我们看到的只是AndroidManifest通过theme设置的一张背景图而已：

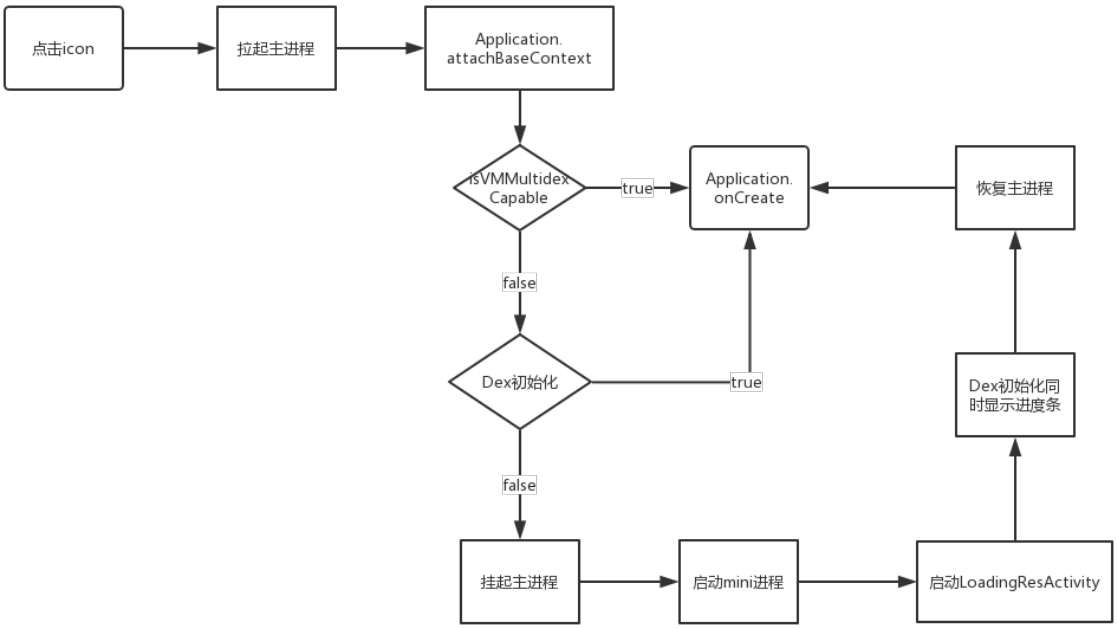
```
<style name="MyTheme.Launch.Img" parent="@android:style/Theme.Light.NoTitleBar">
    <item name="android:background">@drawable/launch_drawable</item>
</style>
```

很多APP都会这样做来避免冷启动时出现白屏or黑屏现象。

2.那么问题来了，如果想要闪屏页显示进度条，就必须要等待闪屏页Activity实例创建完成；闪屏页Activity实例又需要等待Application实例创建完成才能创建；而Application里很多初始化操作又依赖于其他classesN.dex（N>=2），如果classesN.dex（N>=2）没有加载进来，那么初始化时会报 NoClassDefFoundError错误；要加载classesN.dex就需要执行MultiDex.*install*(this)。我们抓取问题的头尾，发现：想要闪屏页显示进度条，必须先执行MultiDex.*install*(this)。但是如果MultiDex.*install*(this)执行完了，我们还需要进度条干什么。脑海里是不是一堆黑人问号？这不是陷入死循环了么？

3.既然进度条的加载依托于Activity，那么有没有办法在MultiDex.*install*(this)没执行的时候也能启动Activity呢？答案是肯定的，那就是多进程。前面我们说过，Application里很多很多初始化操作依赖于除主classes.dex之外的其他classesN.dex（N>=2），但是如果启动一个轻量级进程，里面不进行任何初始化操作，不就不需要加载classesN.dex了么？也就是不需要等待MultiDex.*install*(this)执行完就能启动Activity了。

4. 在attachBaseContext的时候，发现没有执行过Dex初始化操作，那么我们启动一个mini进程，用来显示Loading界面和执行Dex初始化操作。此时主进程退到后台。由于此时classesN.dex还没见加载进来，Application的初始化无法继续进行，只能先挂起主进程，然后每个200ms轮询Dex初始化操作是否已经执行完，就是查询前面提到的SharedPreferences 保存的value是否为true，是的话就继续执行Appliaciton的初始化操作，否则一直等待下去。现在我们梳理一下整个启动流程：



5.是不是觉得已经万事大吉了，那只能说你还图样sometimes纳伊夫。事实上，具体实施过程中容易遇到一个大坑：主进程在后台挂了！！！首先，5.0以下机器大多配置较差，内存吃紧，前台执行的DexOpt操作又很吃资源，退到后台的主进程本身优先级就低，所以很容易被LMK（LowMemoryKiller）机制杀死。然后你会发现，一直卡在Loading界面无法进入Welcome界面了，因为主进程死了无法恢复。

6.既然卡在Loading界面是因为主进程在后台被杀了，那么我们在Dex初始化操作结束时判断一下主进程是否存活，如果存活则恢复主进程，继续往下执行；否则，在Loading界面重新启动闪屏页。这样总可以了吧？呵呵，你还是太天真。还存在这样的情形：主进程在后台因为系统内存不足被Kill了，但是要知道我们的APP是注册了很多广播接收器的，监听了很多系统广播比如网络状态变化。主进程在被杀后，可能因为这样的广播重新被拉起。如果此时Dex初始化操作执行完了恢复主进程，同样是不能调出闪屏页的（因为主进程的Application创建完后接下来就是去执行广播接收器里的操作了而不是启动闪屏页Activity）。

7.考虑到上面这些情况，最终决定在Loading界面dex初始化操作结束后，不管主进程是否存活都主动跳转到闪屏页（更细致的做法是判断前后两次的主进程号是否一致，如果是则恢复主进程；否则跳转到闪屏页，此方案是否有缺陷有待验证）。这样可以彻底杜绝卡在Loading页进不去Welcome界面的尴尬。

8.但是依然有坑！！！可真是磨人。前面有提到5.0以下大部分设备配置都很差，再加上TalkU本身优化工作没做到位，Applicaiton和闪屏页里的初始化工作实在太多（并没有做到按需加载，很多初始化工作可以延后进行），

从Loading界面恢复主进程再到展示闪屏页（实际上应该是Welcom界面，但是我们的很多初始化工作又是在闪屏页执行，执行完后自动跳转到Welcome界面），这个过程十分漫长（超过5s很正常），很容易出现ANR。再者就算这个过程很迅速，但是"点击icon->闪屏页背景->Loading页->闪屏页->Welcom页"这个跳转如此频繁的启动过程，你没有发现一些异常么？是的，过渡问题。从Loading页到Welcome页，中间多了一个闪屏页，本应该从99%的加载进度页直接跳到Welcome页，现在中间会闪现一个不协调的闪屏页！！！本想把这一步骤干掉，但是一看到闪屏页的初始化工作之多，涉及业务面之广，顿时有种深深的无力感，还是保留吧。

9.于是想到把中间的闪屏页Activity设为透明，这样跳转的时候不再会有明显的闪烁现象。但是如果在透明的闪屏页按下Home键再返回，会有很长的等待时间或者黑屏，这一点，现在依然不能很好处理。但是毕竟会这样操作的用户不多，就算有，他按下Home键的时机也不一定刚好是APP处在透明闪屏页的时候。这点瑕疵，还是可以接受的。

10.关于进度条进度的分配，是参考了手Q的做法。dex初始化执行完之前，都会停留在89%而不是99%。后面10个百分点用来等待Applicaiton初始化和透明闪屏页初始化。这个时间可能够也可能不够，都会走到99%然后直接跳转或者是等待。不够的话最多等待25s，Loading界面会自己finish，并退出mini进程。一是释放不用的系统资源，二是防止某些异常情况下会出现Loading界面（99%）乱入的场景。

11.这个思路主要是参考Android拆分与加载Dex的多种方案对比 这篇文章，稍作修改。有兴趣可以详细阅读，虽然是15年的，但依然十分有帮助。

四、EventBus版本升级

从耗时操作统计情况来看，EventBus实例的创建也十分耗时。之前项目用的EventBus版本是2.4.0。现在最新版已经是3.0了。网上搜了一下，发现3.0在性能上确实有提升。详情可参考【腾讯Bugly干货】老司机教你“飙”EventBus3

需要注意的有几点：

- 混淆。 被Subscribe注解标注的方法不能被混淆，否则运行时会报错误。
- **DTEventBusIndex**。这个类是编译过程中动态生成的，首次下载工程后是找不到该类的，编译一下就可以了。
- 全局添加**Subscribe**注解标注。执行gradle脚本task replaceEventBusCall（dingtone_lib目录下的build.gradle文件最底部注释掉部分）即可。部分代码格式不规范的地方需要手动修改。
- 测试用例。EventBus升级涉及文件很多，测试用例也很多，参考EventBus测试用例。

总结

一、正确科学的启动优化方式应该是给Application和SplashActivity减负。紧急性不高的的单例创建和初始化工作按需执行，不要一股脑全部丢在APP进程创建之初，这样启动慢是必然的。

二、第三方SDK的使用（单例创建初始化）可能会很耗时，需要及时关注是否有新版本发布以及新版本是否有性能上的提升。

三、每个开发者都应该有性能优化的意识。日常代码维护或者是新功能开发的时候应该多考虑是否会影响性能而不是单纯实现业务功能就了事。