

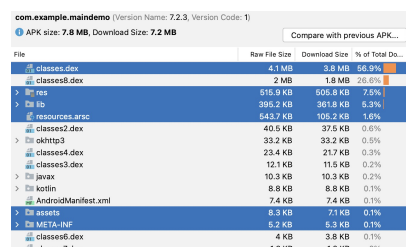
安装包大小优化

▼ 必要性

- 下载转化率
 - 应用市场、渠道限制
- ## ▼ 体积过大对 App 性能的影响
- ▼ 内存RAM占用太大，虚拟内存不够
 - 运行时内存：Resource 资源、Library 以及 Dex 类加载都会占用应用的一部分内存
 - 加载速度
 - 安装速度
 - 安装失败率（ROM）

▼ 原则

- 分析apk包的各种资源占比：



File	Raw File Size	Download Size	% of Total Download Size
classes.dex	4.1 MB	3.9 MB	56.9%
classes2.dex	2 MB	1.8 MB	26.6%
res	315.9 KB	505.8 KB	7.5%
resources.arsc	543.7 KB	105.2 KB	1.6%
classes3.dex	40.5 KB	37.5 KB	0.6%
okhttp3	33.2 KB	33.2 KB	0.5%
classes4.dex	23.4 KB	21.7 KB	0.3%
classes5.dex	12.1 KB	11.5 KB	0.2%
java8	10.3 KB	10.3 KB	0.2%
kotlin	8.8 KB	8.8 KB	0.1%
AndroidManifest.xml	7.4 KB	7.4 KB	0.1%
assets	8.3 KB	7.1 KB	0.1%
META-INF	5.2 KB	5.3 KB	0.1%
classes6.dex	4 KB	3.8 KB	0.1%
classes7.dex	1.3 KB	1.2 KB	0%

- 分析各业务资源占比：
- 分析业务的启动加载流程：缓存业务转成独立模块
- 针对占比大的下手；
- 挑正面收益大，负面收益小；合适的本App的方法下载

▼ 业务方面

▼ 业务梳理

- 用不用
- 用的次数，功能重要性
- 业务边界-依赖

- 删除不用的业务

▼ 子业务组件、动态化

- dynamic feature 插件化

- 子业务之间共用部分独立成单独的模块

▼ 方法

▼ class dex

▼ 1、混淆

▼ D8

- 1)、Dex的编译时间更短。
- 2)、.dex文件更小。
- 3)、D8 编译的 .dex 文件拥有更好的运行时性能。
- 4)、包含 Java 8 语言支持的处理。

▼ R8

- R8 是 Proguard 压缩与优化部分的替代品，并且它仍然使用与 Proguard 一样的 keep 规则。
- R8 在 inline 内联容器类中更有效，并且在删除未使用的类，字段和方法上则更具侵略性
- R8 进行了 ProGuard 尚未提供的一些 Kotlin 的特定的优化

▼ ProGuard

- 和R8都有基本名称混淆：它们 都使用简短，无意义的名称重命名类，字段和方法。他们还可以 删除调试属性

▼ 2、公共lib统一去重

▼ 图片包装模块

- (Picasso、Glide、Fresco)
- 分享，push，支付

▼ 3、删除不用的类

- lint和混淆

▼ 4、dex 的 DebugItem 删除，去除 debug 信息与行号信息

- debugItem 一般占 Dex 的比例有 5% 左右
- 关于如何去除 Dex 中的 Debug 信息是通过 ReDex 的 StripDebugInfoPass 来完成的

▼ 5、dex 分包优化

- ReDex 的 CrossDexDefMinimizer 类分析了类之间的调用关系，并 使用了贪心算法去计算局部的最优解（编译效果和dex优化效果之间的某一个平衡点）。使用 "InterDexPass" 配置项可以把互相引用的类尽量放在同个 Dex，增加类的 pre-verify，以此提升应用的冷启动速度。

- ▼ 6、使用 XZ Utils 进行 Dex 压缩
 - XZ Utils 是具有高压缩率的免费通用数据压缩软件
 - XZ Utils 的输出比 gzip 小 30%，比 bzip2 小 15%。
- 7、R field内联（ByteX 也增加了 shrink_r_class ThinRPlugin）
- ▼ 8、移除无用代码
 - 业务代码只增不减
 - 代码太多不敢删除
 - 用aop将类的入口加打点：有统计，则在用
- ▼ 9、ReDex
 - 1)、Interdex：类重排和文件重排、Dex 分包优化。其中对于类重排和文件重排，Google 在 Android 8.0 的时候引入了 Dexlayout，它是一个用于分析 dex 文件，并根据配置文件对其进行重新排序的库。与 ReDex 类似，Dexlayout 通过将经常一起访问的部分 dex 文件集中在一起，程序可以因改进文件位置从而拥有更好的内存访问模式，以节省 RAM 并缩短启动时间。不同于ReDex的是它使用了运行时配置信息对 Dex 文件的各个部分进行重新排序。因此，只有在应用运行之后，并在系统空闲维护的时候才会将 dexlayout 集成到 dex2oat 的设备进行编译。
 - 2)、Oatmeal：直接生成 Odex 文件。
 - 3)、StripDebugInfo：去除 Dex 中的 Debug 信息。
 - 4)、源码中 access-marking 模块：删除 Java access 方法。
 - 5)、源码中 type-erasure 模块：类型擦除。
- ▼ 10、利用 ByteX Gradle 插件平台中的代码优化插件
 - 编译期间 内联常量字段：const_inline
 - 编译期间 移除多余赋值代码：field_assign_opt
 - 编译期间 移除 Log 代码：method_call_opt
 - 编译期间 内联 Get / Set 方法：getter-setter-inline-plugin
 - 避免产生 Java access 方法，使用 ByteX 的 access_inline 插件
- ▼ res resources.arsc
 - ▼ 1、冗余资源优化
 - 使用 Lint 的 Remove Unused Resource
 - 优化 shrinkResources 流程真正去除无用资源
 - android-chunk-utils

▼ 2、重复资源优化

- 内容相同，但名不同的问题
- 通过资源包中的每个ZipEntry的CRC-32 checksum来筛选出重复的资源
- android-chunk-utils

▼ 3、图片压缩

- tinypng
- AAPT 会使用内置的压缩算法来优化 res/drawable/ 目录下的 PNG 图片，但这可能会导致本来已经优化过的图片体积变大

▼ 4、使用针对性的图片格式

▼ 尽量用webp

▼ 子主题 1

- 相同的图片转换为 webp 格式之后会有大幅度的压缩

▼ 少用png

- 对于 png 来说，它是一个无损格式

▼ 图片放置优化的大概思路

- 1)、聊天表情出一套图 => xhdpi。
- 2)、纯色小 icon 使用 VD => raw。
- 3)、背景大图出一套 => xxhdpi。
- 4)、logo 等权重比较大的图片出两套 => xhdpi, xxhdpi。
- 5)、若某些图在真机中有异常，则用多套图。
- 6)、若遇到奇葩机型，则针对性补图。
- 我们统一只把图片放到 xxhdpi 这个目录下
- ▶ VectorDrawable (矢量图形) 8
- ▶ iconFont (字体图标) 10
- ▼ VD (纯色icon) ->
 - ▼ WebP (非纯色icon) ->
 - ▼ Png (更好效果) ->
 - jpg (若无alpha通道)

▼ 5、资源混淆

- 同代码混淆类似，资源混淆将 资源路径混淆成单个资源的路径

- ▼ AndroidResGuard
 - 例如将 res/drawable/wechat 变为 r/d/a
- ▼ 6、R Field 的内联优化
 - 进一步对代码进行瘦身
 - 解决了 R Field 过多导致 MultiDex 65536 的问题
- ▼ 蘑菇街的 ThinRPlugin
 - android 中的 R 文件，除了 styleable 类型外，所有字段都是 int 型变量/常量，且在运行期间都不会改变。所以可以在编译时，记录 R 中所有字段名称及对应值，然后利用 ASM 工具遍历所有 Class，将除 R\$styleable.class 以外的所有 R.class 删除掉，并且在引用的地方替换成对应的常量
- ▼ ByteX 也增加了 shrink_r_class
 - 可以在编译阶段对 R 文件常量进行内联
 - 针对 App 中无用 Resource 和无用 assets 的资源进行检查
- ▼ 7、资源合并方案
 - 一个大资源文件就相当于换肤方案中的一套皮肤
- ▼ 8、资源文件最少化配置
 - 按需配置资源（语言）
 - app bundle so, res , language
 - resConfigs "zh", "zh-rCN"
resConfigs "nodpi", "hdpi", "xhdpi", "xxhdpi", "xxxhdpi"
- ▼ 9、资源在线化
 - 将一些图片资源放在服务器，然后 结合图片预加载 的技术手段，这些 既可以满足产品的需要，同时可以减小包大小。
- ▼ 10、统一应用风格
 - 如设定统一的 字体、尺寸、颜色和按钮按压效果、分割线 shape、selector 背景 等等
- ▼ lib so
 - ▼ 1、So 移除方案
 - 去了不要的平台：X86, mips, armeabi, 只要armeabi-v7a,arm64-v8a
 - 按需加入-armeabi兼容性好-但性能差
 - 大多数设备（大于android 7）支持arm64-v8a

▼ 2、extractNativeLibs

- 此属性指示软件包安装程序是否将原生库从 APK 提取到文件系统
- 如果设置为“false”，则原生库以未压缩的形式存储在 APK 中，虽然您的 APK 可能较大，但应用应该加载得更快，因为库是在应用运行时直接从 APK 加载，但apk会变大
- ▼ 如果为true：so加被压缩，
 - 缺点是：因为so是压缩存储的，因此用户安装时，系统会将so解压出来，重新存储一份。因此安装时间会变长，占用的用户磁盘存储空间反而会增大
 - 好处是：用户在应用市场下载和升级时，因为消耗的流量较小，用户有更强的下载和升级意愿
- minSdkVersion < 23 或 Android Gradle plugin < 3.6.0情况下，打包时 android:extractNativeLibs=true；
- minSdkVersion >= 23 并且 Android Gradle plugin >= 3.6.0情况下，打包时 android:extractNativeLibs=false；

▪ 3、使用 XZ Utils 对 Native Library 进行压缩

▼ 4、对 Native Library 进行合并（减少体积，影响按需加载）

- 可以删除部分动态符号表项，减小 so 总体积。具体来讲，就是可以删除 liba.so 和 libb.so 的动态符号表中的所有导出符号，以及 libx.so 的动态符号表中从 liba.so 和 libb.so 中导入的符号。
- 可以删除部分 PLT 表项和 GOT 表项，减小 so 总体积。具体来讲，就是可以删除 libx.so 中与 liba.so、libb.so 相关的 PLT 表项和 GOT 表项。
- 可以减轻优化的工作量。如果没有合并 so，对 liba.so 和 libb.so 做体积优化时需要确定 libx.so 依赖了它们的哪些符号，才能对它们进行优化，做了 so 合并后就不需要了。链接器会自动分析引用关系，保留使用到的所有符号的对应内容。
- 由于链接器对原 liba.so 和 libb.so 的导出符号拥有了更全的上下文信息，LTO 优化也能取得更好的效果

▼ 5、删除 Native Library 中无用的导出 symbol

▼ 精简动态符号表

- RegisterNatives 方式可以提前检测到方法签名不匹配的问题，并且可以减少导出符号的数量所以在最优情况下只需导出 JNI_OnLoad（在其中使用 RegisterNatives 对 Java native 方法进行动态注册）和 JNI_OnUnload（可以做一些清理工作）这两个符号即可。
- 使用 visibility 和 attribute 控制符号可见性

- 使用 static 关键字控制符号可见性
- 使用 exclude libs 移除静态库中的符号
- 使用 version script 控制符号可见性
- ▼ 移除无用代码
 - 在实际的项目中，有一些代码在 Release 版中永远不会被使用到（例如历史遗留代码、用于测试的代码等）
 - 开启 LTO，是 Link Time Optimization 的缩写，即链接期优化
 - 开启 GC sections
- ▼ 优化指令长度
 - 实现某个功能的指令并不是固定的，编译器有可能能用更少的指令完成相同的功能，从而实现优化。由于指令是 so 的主要组成部分，因此优化这一部分的潜在收益也比较大
 - 使用 Oz/Os 优化级别
- ▼ 其它
 - 禁用 C++ 的异常机制
 - 禁用 C++ 的 RTTI 机制
- 6、So 动态下载
- Android 构建工具默认为 so 体积做的优化 strip 优化（移除调试信息和符号表）
- ▼ 提取多 so 共同依赖库
 - 这里典型的例子是 libc++ 库：如果存在多个 so 都静态依赖 libc++ 库的情况，可以优化为这些 so 都动态依赖于 libc++_shared.so。
 - 上面“合并 so”是减小 so 总个数，而这里是增加 so 总个数。当多个 so 以静态方式依赖了某个相同的库时，可以考虑将此库提取成一个单独的 so，原来的几个 so 改为动态依赖该 so。例如 liba.so 和 libb.so 都静态依赖了 libx.a，可以优化为 liba.so 和 libb.so 均动态依赖 libx.so。提取多 so 共同依赖库，可以对不同 so 内的相同代码进行合并，从而减小总的 so 体积。
- ▼ assets
 - 压缩-去重
 - 在线化，动态化
 - 子主题 3
- ▼ flutter
 - 混淆

▼ META-INF

- META-INF目录下存放的是签名信息
- 用来保证apk包的完整性和系统的安全性，帮助用户避免安装来历不明的盗版apk
- 除了CERT.RSA没有压缩机会外，其余的两个文件都可以通过混淆资源名称的方式进行压缩

▼ 整体

- 7-zip
- zipAlignEnabled
- minifyEnabled
- shrinkResources

▼ 长效化

- apk分析工具
- CICD集成apk对比任务
- 变动分析、报警