什么是 Booster ？

Booster 的目标主要是为了解决随着 APP 复杂度的提升而带来的性能、稳定性、包体积等一系列质量问题。Booster 不仅仅只是一个框架，它还提供了性能检测、多线程优化、资源索引内联、资源去冗余、资源压缩、系统 Bug 修复等一系列功能模块。

Booster 通过 Gradle Plugin 的形式为 Android 工程质量把控提供了一套完整的框架，无论是代码、资源、动态库、依赖关系、包体积、性能等监控，都可以通过 Booster 来完成。

Booster 提供了 Transformer SPI 让开发者更方便的在构建过程中注入字节码、类继承分析等等，并提供了针对线程的重命名能力，结合 APM 平台，让排查问题更加方便与高效。

Booster 提供了静态分析的能力，通过在构建过程中对代码、资源以及构建中间产物进行分析，便可提前发现有性能问题、安全风险、违反应用商店政策的代码等。

Booster 通过 AOP（Aspect-Oriented Programming） 的方式，对 APP 进行无侵入的全局性能优化，而且对于业务方来说，几乎是完全透明的。

Booster 的静态分析解决了什么问题？

booster-task-analyser 通过黑/白名单的方式对 APP 进行扫描，并生成相应的分析报告，使得开发者对 APP 的质量有一个更全面和深入的了解，并为更深层次的优化提供思路：

Analyser 的实现思路

独立的 Task

Booster 的静态分析采用独立的 task 来执行，之所以这样设计，主要有几个方面的考虑：

1、对应用进行静态分析的频率不像构建那么频繁，所以，Task 比 Transformer 更合适；

2、CHA (Class Hierarchy Analysis) 需要提前拿到所有类信息，而 Transformer 是流水线处理，也不太合适；

3、静态分析的过程可能会比较慢，作为 Transformer 可能会严重影响构建效率，而且应用的构建并不依赖静态分析的产出物；

类继承分析

类继承关系分析对于静态分析至关重要，它决定了分析结果的准确性和全面性，在 Transform 中 CHA 是通过 ClassLoader 来实现的，相对来说比较简单，参见：KlassPool & Klass，主要是解决如何判断两个类型是否有继承关系的问题，Analyser 的 CHA 采用的方式是提前加载所有 Class，然后进行分析，主要有以下几个方面的原因：

1、ClassLoader 加载 Class 时，虽然可以不对类进行初始化，但是 ClassLoader 会对 bytecode 进行 verify ，可能会抛出 VerifyError 导致整个分析过程失败；

2、性能开销 —— ClassLoader 加载 Class 的性能跟 ASM 对比相差甚远；

3、除了分析类的继承关系外，还需要分析字段和方法以及注解，通过 Class 反射得到的信息有限；

4、Task 相对于Transform 比较独立，如果在 Transform 的过程中加载所有的 Class ，可能导致内存吃紧，甚至 OOM

静态分析入口

四大组件

像 Application 及四大组件都在 AndroidManifest.xml 里，通过 mergedManifests 就能获取到合并后的 AndroidManifest.xml

自定义 View

查找自定义 View 最直接的方法就是解析 Layout XML ，通过 mergeRes 就能获取到，只不过是 AAPT2 的产物 —— flat 文件，这也就是 booster-aapt2 模块的由来。

通过实测发现：解析 flat 文件的速度不如直接解析 XML 源文件，所以，最终的实现只解析了 flat 文件的 header 部分，然后通过 header 定位到源文件的路径。

线程注释标注的方法和类

Android 本身提供了 Thread Annotations，帮助编译器和静态分析工具提升代码检查的准确性，所以，只要有类或者方法用 Thread Annotations 标注过，则可以认为该类或者方法就是线程入口类或者方法。

考虑到一些主流的应用框架也有线程注解，因此，Analyser 对 Event Bus 做了支持，通过 @Subscribe(threadMode = MAIN) 标的方法会被识别为主线程入口方法。

多线程优化

线程管理面临的挑战

多线程竞争意味着过多的资源浪费在了线程调度上。

通过对线程进行埋点监控，发现了以下的现象：

1、在某种场景下会无限制的创建新线程，最终导致 OOM

2、在某一时间应用内的线程数达到数百甚至上千

3、即使在空闲的时候，线程池中的线程一直在 WAITING

这些现象最终导致的问题是：

1、Out Of Memory

2、无法分辨出线程所属的模块，导致排查问题困难

booster-transform-thread 提供了一种全新的思路——将所有创建线程的指令在编译期间替换成自定义的方法调用，经过大量的分析和验证后发现，Android 创建线程主要是通过以下几种方式：

1、Thread 及其子类

2、TheadPoolExecutor 及其子类、Executors、ThreadFactory 实现类

3、AsyncTask

4、Timer 及其子类

降低线程数量

降低线程数量并不是说不用线程或少用线程，而是让线程在空闲的时候自动销毁，基于这一理论，通过替换创建线程池的方法，或者直接修改线程池的参数：

1. 将 corePoolSize 设置为 0

new ThreadPoolExecutor(0, maxCorePoolSize, keepAliveTime, timeUnit, workQueue, threadFactory);

将 ScheduledThreadPoolExecutor 的 minPoolSize 设置为 0 在 JDK 9 以下的版本会导致 CPU 负载严重。

2、为 maxPoolSize 设置上限

有些开发者或者第三方库设置的线程池 maxPoolSize 通常是 2 \* NCPU + 1 或者 2 \* NCPU，当有多个模块都这样使用的时候，就容易造成某一时刻出现大量的线程，尤其是 CachedThreadPoolExecutor，通过控制单个线程池的 maxPoolSize 的上限，可以将某一时刻，所有线程池造成的叠加效应降到尽可能低的水平。

new ThreadPoolExecutor(0, Math.min(maxCorePoolSize, NCPU), keepAliveTime, timeUnit, workQueue, threadFactory);

3、允许核心线程在空闲时自动销毁

executor.allowCoreThreadTimeOut(true)

4、尽可能将 HandlerThread 替换成 SingleThreadPoolExecutor

为什么不推荐用 HandlerThread？不可否认，Android 的 HandlerThread 确实简化了多线程开发，而实际的情况是，很少有开发者会主动销毁 HandlerThread，如果大量使用 HandlerThread 而不销毁，HandlerThread 会一直占用内存空间，每个线程栈大小至少是 1040k，如果大量的 HandlerThread 长时间处理 WAITING 状态，则会导致不必要的内存浪费，从而引发 Out Of Memory，采用 SingleThreadPoolExecutor 的好处是可以让核心线程在空闲时自动销毁。

线程重命名

ShadowThread 类的这些静态方法的参数比替换之前多了一个 String 类型的参数，其实，这个 String 参数就是调用 Thread 的构造方法的类的 className，而这个类名，是在 Transform 的过程中扫描出来的，通过这个 className 来为 Thread 重命名，这样，就可以在运行时确定这个线程是从哪儿冒出来的。

线程池优化

理解了线程重命名的实现原理，线程池参数优化也就能理解了，线程池优化主要做了这几件事情：

1、将调用 Executors 类的静态方法替换为 ShadowExecutors 的静态方法；

2、将调用 ThreadPoolExecutor 类的构造方法替换为 ShadowThreadPoolExecutor 的静态方法；

3、在 Application 类的 <clinit>() 中调用 ShadowAsyncTask.optimizeAsyncTaskExecutor() 修改 AsyncTask 的线程池参数；

SharedPreferences 优化

SharedPreferences 是如何造成卡顿和 ANR 的呢？这得从 SharedPreferences.Editor.apply() 说起。在 Activity执行onPause() 的时候，会调用 QueuedWork.waitToFinish()，而 waitToFinish() 就是将之前 Editor.apply() 往 QueuedWork 队列中追加的 Runnable 在主线程同步执行。

异步调用 commit()

booster v0.1.5 推出的针对 SharedPreferences 的优化，主要是将 Editor.apply() 替换成 Editor.commit() 并在子线程中执行。这个方案的优化点是改动很小，风险相对较小，缺点是有一种场景可能会有 bug —— 在调用 commit() 后立即调用 getXxx()。

自定义 SharedPreferences

为了彻底解决 SharedPreferences 的问题，Booster v0.27.0 推出了BoosterSharedPreferences，通过 SharedPreferencesTransformer 将所有调用 Context.getSharedPreferences(String, int) 的指令替换成 ShadowSharedPreferences.getSharedPreferences(Context, String, int)。通过自定义 SharedPreferences 避开 QueuedWork 在 onPause(), onDestroy() 等生命周期回调时在主线程中的同步操作。

基于 MMKV 的 SharedPreferences

WebView 预加载

WebView 在实例化的时候，需要先初始化 Chromium 引擎，而 Chromium 引擎又是一个重量级的组件，而且很多初始化的工作都需要在主线程中完成，这样就很容易造成主线程卡顿甚至 ANR。

预加载 Chromium 引擎

有人提出了通过 WebView 或者 WebSettings 中的静态方法来间接的初始化 WebView，比如：WebSettings.getDefaultUserAgent(Context)、WebViewFactoryProvider.Statics 中的其它方法。

这些静态方法只能完成一小部分初始化的工作，对于最终的优化结果来说，并不会有明显的作用，主要的初始化工作是由 WebViewChromiumAwInit 来完成的。

ShadowWebView 通过 WebViewTransformer 在 Application onCreate() 回调中注入 ShadowWebView.preloadWebView()，当主线程 IDLE 时，启动 Chromium 引擎。

该方案的优点是无侵入、接入成本低，缺点是由于这种方式反射了非公开 API，可能存在兼容性问题。

PNG 图片压缩

Pngquant

实现思路

根据 Android 的构建流程，我们选择在 mergeRes 和 processRes 任务之间插入 PNG 压缩任务。

WebP 图片压缩

Cwebp

根据 Android 的构建流程，我们选择在 mergeRes 和 processRes 任务之间插入 PNG 压缩任务。

ZIP 文件压缩

AP\_ 文件

在 Android 的构建流程中，真正将 dex, resources, assets, so 等文件合并成 APK 的工作是在 package Task 里完的，而在 APK 生成之前，其实已经有了原型—— AP\_ 文件，也就是 processRes Task 的产物，里面的内容包括：

1、AndroidManifest.xml

2、res/\*

3、resources.arsc

其中 res/\* 是合并后的所有资源，resources.arsc 则是 res/\* 的索引表。

AP\_ 文件就是一个普通的 ZIP 文件，通过 unzip 命令查看该文件内容：

Length Method Size Cmpr Date Time CRC-32 Name

Method 就是存入 ZIP 文件中采用的压缩方法。其中STORED代表已经压缩过的文件、DEFLATED则是未压缩过的文件。

采用 ZIP 打包的文件不一定都是压缩过的，也有未压缩的，这也很容易理解，像图片、音频、视频文件，已经编码压缩过，再用 ZIP 压缩也根本压缩不了多少，有的可能压缩后比原来还要大，所以，这又为我们做包体积瘦身打开了另一扇大门。

既然 AP\_ 文件是 processRes Task 的产物，那我们直接在 processRes Task 任务完成之后对 AP\_ 文件进行重新压缩。

7-zip 与 ZIP 压缩的原理相同，只不过 7-zip 采用了压缩率更高的 LZMA 和 LZMA2 算法，瘦身效果更佳，推荐使用 AndResguard，虽然 7-zip 的压缩效果非常显著，但是会存在一些副作用，可能会导致 Google Play 的优化算法失效。

Google 官方并不推荐对 resources.arsc 和 so 进行压缩，这样会导致它们不能被直接 mmap 到内存，但如果从业务角度来看，如果 APK 的大小成为了阻碍用户增长的一个因素，而通过压缩 resources.arsc 和 so 对用户增长有显著的正向收益。

资源索引内联

资源索引的问题

Android 在构建的过程中，会为每个模块（库、应用）生成一份资源索引，诸如：R$id.class，R$layout.class 等等，这对于开发者来说，在代码里引用资源十分的方便。

对于 library 模块来说，R 文件中的索引值并非常量值，以至于 library 的类中引用 R 索引值的方式其实是调用 R 类的 field 来实现的，这也是为什么在 library 工程中不能在 switch-case 语句或者 Annotatio 中使用资源索引的原因。以至于 ButterKnife 创造了独有的 R2 来解决这个问题。

Android 系统中定义了 10 多种资源类型，假设每个模块使用了 5 种资源类型，就会生成 6 个对应的 class 文件（包括 R.class），由于工程结构的复杂度普遍上升，在 APP 工程中直接或间接引用的 library 少则几十，多则上百，假设 APP 中引用了 100 个 library，那对应的 R 文件至少是 500 个以上，无论是类数量、字段数量都是巨大的浪费，毕竟单个 dex 有 65535 的限制，虽然有 multi-dex 技术，但多一个 dex 就会为安装、冷启动增加不必要的性能开销。

删除不必要的 R

对于 Android 工程来说，通常，library 的 R 只是 application 的 R 的一个子集，所以，只要有了全集，子集是可以通通删掉的，而且，application 的 R 中的常量字段，一旦参与编译后，就再也没有利用价值（反射除外）。在 R 的字段，styleable 字段是一个例外，它不是常量，它是 int[]。所以，删除 R 之前，我们要弄清楚要确定哪些是能删的，哪些是不能删的，根据经验来看，不能删的索引有：

1. ConstraintLayout 中引用的字段。Booster 采用了解析 xml 的方式，从 xml 中提取以上属性。
2. 其它通过 TypedArray.getResourceId(int, int) 或 Resources.getIdentifier(String, String, String) 来获取索引值的资源。针对这种情况，需要对字节码进行全盘扫描才能确定哪些地方调用了 TypedArray.getResourceId(int, int) 或 Resources.getIdentifier(String, String, String)，考虑到增加一次 Transform 带来的性能损耗，Booster 提供了通过配置白名单的方式来保留这些资源索引。

删除不必要的 Field

由于 Android 的资源索引只有 32 位整型，格式为：PP TT NNNN，其中：

1、PP 为 Package ID，默认为 0x7f；

2、TT 为 Resource Type ID，从 1 开始依次递增；

3、NNNN 为 Name ID，从 1 开始依次递增；

为了节省空间，在构建 application 时，所有同类型的资源索引会重排，所以，library 工程在构建期间无法确定资源最终的索引值，这就是为什么 library 工程中的资源索引是变量而非常量，既然在 application 工程中可以确定每个资源最终的索引值了，为什么不将 library 中的资源索引也替换为常量呢？这样就可以删掉多余的 field 了，在一定程度上可以减少 dex 的数量，收益是相当的可观。

在编译期间获取索引常量值有很多种方法：

1、反射 R 类文件

2、解析 R 类文件

3、解析 Symbol List (R.txt)

经过 benchmark 测试发现，解析 Symbol List 的方案性能最优，因此，在 Transform 之前拿到所有资源名称与索引值的映射关系，然后在 Transform 的过程中将 getfield 指令替换成 ldc 指令即可。

移除冗余资源

资源限定符

Android 为了便于对特定设备进行适配，定义了一系列资源配置限定符（Configuration Qualifier），开发者只需要根据限定符来对资源进行命名，Android Framework 会根据上下文环境完成适配工作，常用的 Configuration 有：

1、语言和区域

2、幕尺寸

small

normal

large

xlarge

3、屏幕方向

port

land

4、屏幕像素密度

ldpi

mdpi

hdpi

xhdpi

xxhdpi

xxxhdpi

nodpi

tvdpi

anydpi

nnndpi

5、平台版本（API Level）

v3

v4

v7

移除不必要的资源

对于只发布在国内应用市场的 APP 来说，有些不是完全需要的，如：多语言、多屏幕密度等，我们可以在 build.gradle 中通过如下配置来移除不必要的资源：

android {

defaultConfig {

resConfigs "zh"

}

}

另一种方式是针对不同的 Configuration 分别构建 APK，例如，根据屏幕密度来为每种屏幕密度构建一个独立 APK：

android {

splits {

density {

enable true

exclude "ldpi", "xxhdpi", "xxxhdpi"

compatibleScreens 'small', 'normal', 'large', 'xlarge'

}

}

}

国内的现状

国内的大多数 APP 对于不同的屏幕密度适配做得不是很完善，可能有的只适配了 xhdpi，有的适配了 xhdpi 和 xxhdpi，总之不是所有的图片资源适配的密度是一致的，如果用上面的「拆分 APK」的方式来构建的话，就会出现有些资源找不到的情况，为了解决这一问题，Booster 采用了另一种去冗余的方式：

1、先按资源类型分组；

2、同类型资源按名称分组；

3、同名称资源按 density 从高到底排序

4、针对同类型、同名称的不同 denisty 资源，保留最高 density 的资源，移除其它 density 资源

上面的过程看起来很简单，但是，实际情况要比这更复杂，有没有可能存在同类型、同名称、同 density 的资源呢？当然是可能的，毕竟 Configuration 有很多个适配维度，比如：屏幕布局方向。

为系统崩溃兜底

许多 Android 开发者可能经常遇到这样的情况：测试的时候好好的，一上线，各种系统的 crash 就报上来了，而且很多是偶现的，比如：

DeadObjectException

RuntimeException

WindowManager$BadTokenException

Resources.NotFoundException

NullPointerException

SecurityException

IllegalArgumentException

......

很多情况下，这些异常崩溃并不是由 APP 导致的，而且堆栈中也没有半点 APP 的影子，就拿 DeadObjectException 来说，一般都是由于提供远程服务的进程挂掉导致，如果是 APP 代码逻辑的问题，很容易就能在堆栈中发现，那如果是因为系统导致的崩溃，我们难道就无能为力了？

在 Android 系统中，很多的代码逻辑都是在主线程中完成的，例如：四大组件的生命周期，视图相关的操作等等，而 ActivityThread 在其中扮演了一个很重要的角色，几乎所有在主线程中完成的工作都要经过它，如果能把经过 ActivityThread 的所有调用都 try-catch 住，不就能兜住了么？

究竟如何 try-catch 呢？通过分析 ActivityThread 的源码发现，几乎所有的工作都是由 ActivityThread.H 这个内部类来完成的。

由于 ActivityThread.H 是继承自 Handler，而 Handler 支持通过 Handler.Callback 来改变其自行的行为，所以，只要通过反射为 ActivityThread.mH.mCallback 设置一个新的 Handler.Callback ，然后在这个 Handler.Callback 中将系统异常 catch 住就行了。

有人可能会问，如果直接用 try-catch 大法这样粗暴的处理方式的话，那 APP 本身的 bug 是不是就不能及时发现了呢？—— 确实是这样！正是基于这样的考虑，Booster 并不是简单粗暴的一起兜住，虽然这样做可以让崩溃率变得更好看，但是，APP 本身的问题也就被掩盖了，而且可能导致业务流程无法正常进行下去，那到底是如何甄别异常是由 APP 引起的呢？—— 堆栈信息。通过通栈中是否存在非系统的类，便可判断异常是否由 APP 导致的了：

class ActivityThreadCallback implements Handler.Callback {

private static final String[] SYSTEM\_PACKAGE\_PREFIXES = {

"java.",

"android.",

"androidx.",

"dalvik.",

"com.android.",

};

private final Handler mHandler;

private final Handler.Callback mDelegate;

private final Set<String> mIgnorePackages;

/\*\*

\* @param ignorePackages packages to ignore

\*/

public ActivityThreadCallback(final String[] ignorePackages) {

final Set<String> packages = new HashSet<>(Arrays.asList(SYSTEM\_PACKAGE\_PREFIXES));

for (final String pkg : ignorePackages) {

if (null == pkg) {

continue;

}

packages.add(pkg.endsWith(".") ? pkg : (pkg + "."));

}

packages.add(getClass().getPackage().getName() + ".");

this.mIgnorePackages = Collections.unmodifiableSet(packages);

this.mHandler = getHandler(getActivityThread());

this.mDelegate = getFieldValue(this.mHandler, "mCallback");

}

private boolean isCausedByUser(final Throwable t) {

if (null == t) {

return false;

}

for (Throwable cause = t; null != cause; cause = cause.getCause()) {

for (final StackTraceElement element : cause.getStackTrace()) {

if (isUserStackTrace(element)) {

return true;

}

}

}

return false;

}

private boolean isUserStackTrace(final StackTraceElement element) {

final String name = element.getClassName();

for (final String prefix : this.mIgnorePackages) {

if (!name.startsWith(prefix)) {

return true;

}

}

return false;

}

}

Finalizer 导致的 TimeoutException

通过查看 AOSP 源码，我们很容易就能定位到异常发生在 Daemons$FinalizerWatchdogDaemon.finalizerTimedOut 。我们可以看到 finalizerTimedOut 会抛出 UncaughtException。FinalizerWatchdogDaemon 会异步等待 Finalization 结束。FinalizerWatchdogDaemon 会等待 FinalizerDaemon.doFinalize() 的结果，如果在 MAX\_FINALIZE\_NANOS 时间之内没有完成，就会抛出 TimeoutException。

从源码中我们可以看到，Daemons 启了 4 个 Daemon 线程：

1、HeapTaskDaemon 用来启动用于处理 GC 相关的任务，如：Heap Trimming, Heap Transition 以及 Concurrent GC。

2、ReferenceQueueDaemon 负责将 FinalizerReference （重写了 finalize() 方法的类在实例化的时候，会被 FinalizerReference 引用，当该实例具有且仅有 FinalizerReference 引用它时，则认为该对象适合被 GC 回收）入队到 ReferenceQueue 中。

3、FinalizerDaemon 负责在 GC 被触发时，执行被 FinalizerReference 引用的对象的 finalize 方法。

4、FinalizerWatchdogDaemon 顾名思义，它就是 Finalizer 的「看门狗」，一旦在规定的时间之内，没有给它「喂骨头」，则认为 Finalizer 过程被阻塞了，它就会抛异常了，而 FinalizerDaemon 就是给它「喂骨头」的线程。

Booster 的解决方案是解决抛出异常的「看门狗」—— 在应用启动后，停掉 FinalizerWatchdogDaemon 线程，这样做对于 APP 来说，并没有什么实质性的影响。

资源为 null 的问题

这种问题基本上都是发生在覆盖安装之后，App 由 Broadcast 唤起，由于系统 bug 导致加载了旧的 APK，一般表现出来的异常有：

Context.getResources() 返回 null

Class.getResourceAsStream() 返回 null

Context.getAssets() 抛出 NullPointerException

Context.getSystemService(Context.AUDIO\_SERVICE) 抛出 NullPointerException

像这样的系统 bug，可能跟系统版本有关，也没有什么通用的解决方案，所以 Booster 的解决思路是，在 Application 启动的时候，对 Resources 和 Assets 进行检查，如果 Resources 或者 AssetManager 为 null，直接杀进程。

Android 7.1 Toast 崩溃

在 Android O (API Level 26) 源码中，这个问题已经被修复了，修复方法就是简单粗暴的 try-catch。

所以，Booster 也简单粗暴的将异常 catch 住，只不过，catch 的不是 mWM.addView(mView, mParams)，而是 Toast$TN 这个内部类的 mHandler，给它设置一个 Handler.Callback。在 booster-transform-toast 中，将 Toast 类及其子类的 show() 方法调用替换成 ShadowToast.show(Toast)。