


Android 应用构建速度提升的十个小技巧



谷歌开发者 
已认证的官方帐号

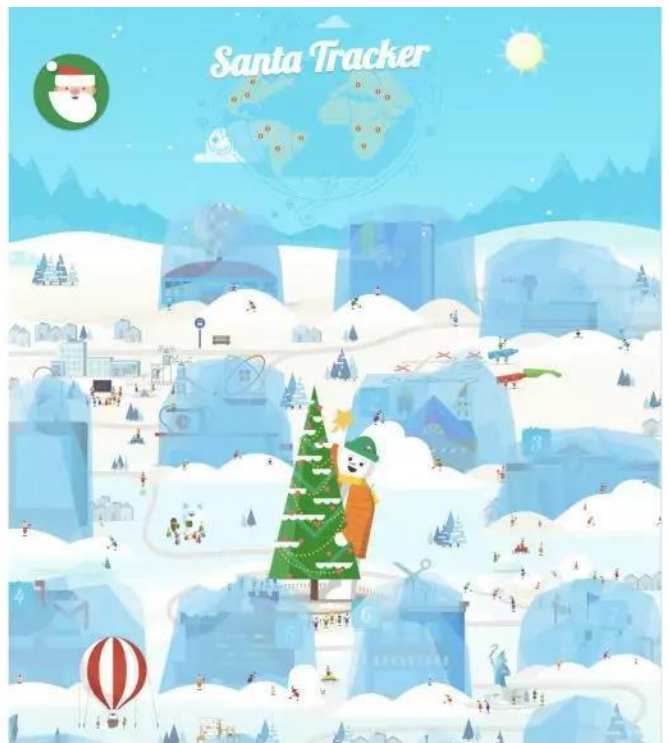
已关注

高爷等 22 人赞同了该文章

应用的构建速度会直接影响开发效率，本文将带您通过改造一个 Android 应用：“Google 追踪圣诞老人 (Google Santa Tracker)” 来为大家提供十个小技巧，帮助提升应用的 Gradle 构建速度，当我们应用了所有的小技巧之后，该演示应用的构建速度快了三倍以上。

Android 应用构建速度提升的十个小技巧

1. 使用最新版的 Android Gradle 插件
2. 避免激活旧版的 Multidex
3. 禁用 Multiple APK 构建
4. 最小化打包资源文件
5. 禁用 PNG 压缩
6. 使用 Instant Run
7. 避免被动的改动
8. 不使用动态版本标识
9. Gradle 内存分配调优
10. 开启 Gradle 构建缓存



首先来了解一下 “Google 追踪圣诞老人” 应用的工程背景：这个应用有约 60M 大小，它包含 9 个模块，有 500 多个 Java 文件，1,700 多个 XML 文件、3,500 多张 PNG 图片资源，用到了 Multidex，没有注解处理器。

其次，在我们开启速度提升调优之前，来了解本次三个性能指标的说明：

- 全量构建，也就是重新开始编译整个工程的 debug 版；
- 代码增量构建，指的是我们修改了工程的 Java / Kotlin 代码；
- 资源增量构建，指的是我们对资源文件的修改，增加减少了图片和字符串资源等。

每个小技巧实施以后，我们会对比如上三个场景的构建时间以作为我们的量化标准。请注意，由于工程规模大小不一、开发环境各异，开发者们在实际的操作中的结果可能会与本文的结果有所不同。

小技巧 1: 使用最新版本的 Android Gradle 插件

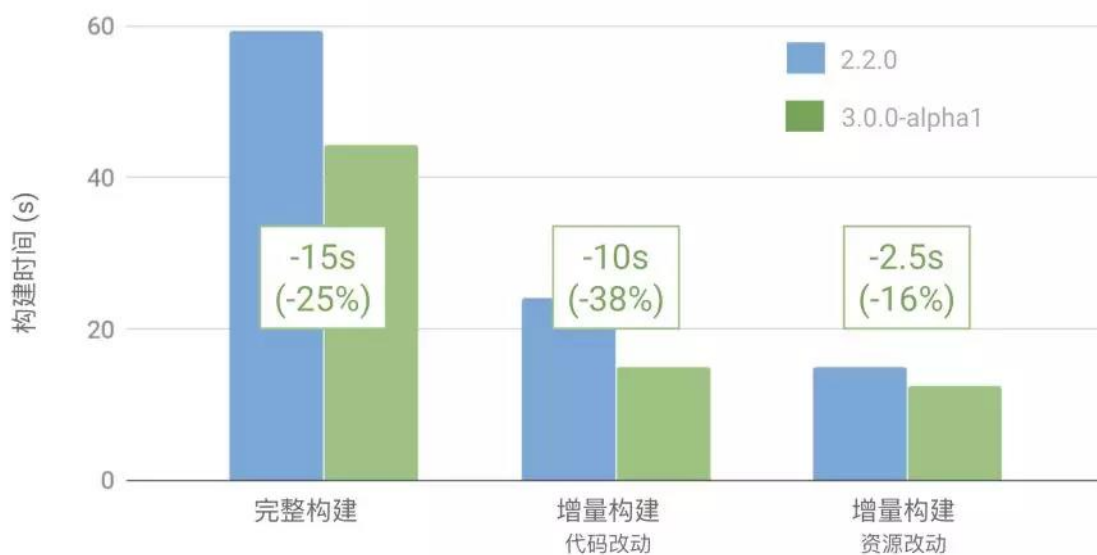


```
repositories {  
    + google()  
    jcenter()  
}  
dependencies {  
    - classpath 'com.android.tools.build:gradle:3.2.0'  
    + classpath 'com.android.tools.build:gradle:3.4.2'  
}  
...  
}
```

每次 Android Gradle 插件的更新都会修复大量的 bug 及提升性能等新特性，因此保持最新的 Android Gradle 插件版本有非常大的必要。

从 3.0 版本开始，我们将通过 google() 的 Maven 仓库分发新的 Android Gradle 插件，所以需要在 repositories 处加入 google() 以获得最新的插件更新 (现在的 Android Studio 新建工程的时候会默认加入 google() 的 Maven 仓库指向)。

Android Gradle 插件版本从 2.2.0 升级到 3.0.0-alpha1



这是将 Android Gradle 插件版本从 2.x 更新到 3.0.0-alpha1 之后得到的结果 (这里的演示是基于 3.0.0-alpha1 版本，随着插件版本的更新，性能的提升会更加明显)，我们可以看出，全量构建一次应用的时间直接减少了 25%，代码改动的增量构建减少了将近 40%，资源改动的增量构建也减少了 16%。

小技巧 2: 避免激活旧版的 Multidex

Multidex + 小于 21 的最低 API 级别

- 旧版 multidex 将会严重拖慢构建速度
- 2.3 版本之后的 Android Studio 会自动规避这个问题

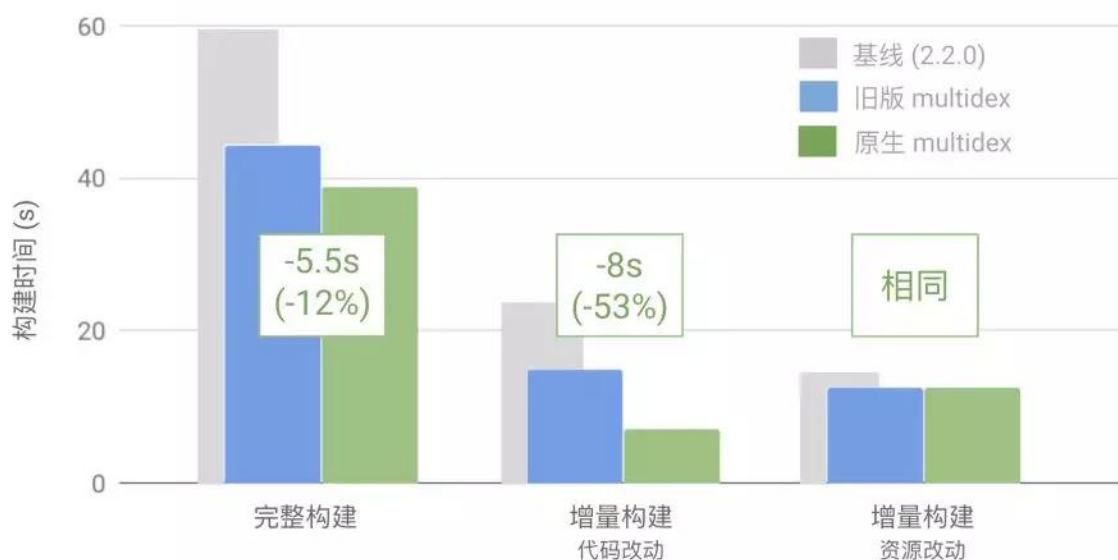
```
development {  
    minSdkVersion 21  
    ...  
}
```

这个小技巧大家应该比较熟悉——避免激活旧版的 multidex。当您的应用配置方法数超过 64K 的时候，您需要启用 multidex。当您启用了 multidex，且工程的最低 API 级别在 21 之前时，旧版的 multidex 就会被激活，这将严重拖慢您的构建速度，原因是 21 之前的 API 级别并没有原生的支持 multidex。

如果您是通过 Android Studio 的运行/调试按钮来执行构建，那么无需考虑这个问题，新版本的 Android Studio 会自动检测连接的设备和模拟器，如果系统的 API 级别大于 21 则进行原生的 multidex 支持，同时会忽略工程里对最低 API 级别 (minSdkVersion) 的设置。

习惯通过命令行窗口构建工程的开发者们则需要试着避免这个问题：配置一个新的 productFlavor，设定工程的最低 API 级别为 21 或者以上，在命令行里调用 assembleDevelopmentDebug 即可避免这个问题。

使用旧版和原生 Multidex 的构建时间对比



这一次的性能改进结果效果也非常明显 (灰色的线条是最初的结果)，在全量构建的时候我们又降低了 5.5 秒的时间，而在代码改动的增量构建里时间减少了 50% 以上，资源改动的增量构建与之前

小技巧 3: 禁用 Multiple APK 构建

- 如右图所示，在 `splits{}` 里为不同的 ABI 和屏幕像素密度设定了 Multiple APK 构建
- 在开发阶段不必要生成多个 APK

```
splits {  
    abi {  
        enable true  
        ...  
    }  
    density {  
        enable true  
        ...  
    }  
}
```

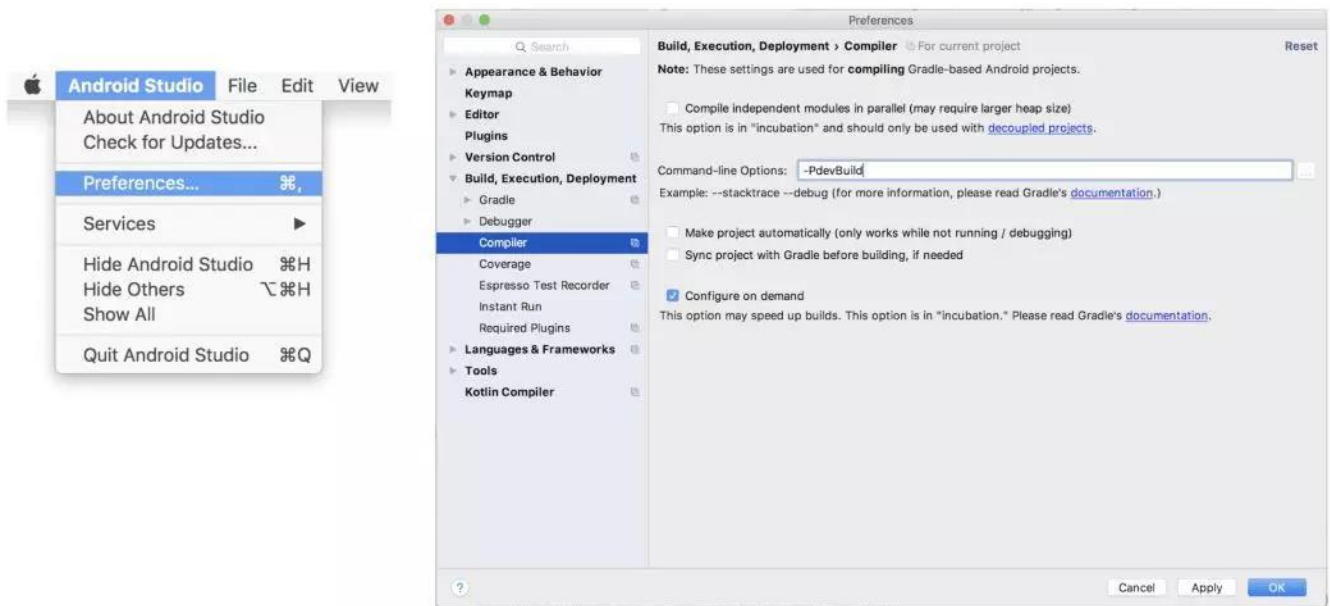
在应用需要发布和上架的时候，我们往往会使用“Multiple APK”构建，它可以根据 ABI 和像素密度创建不同版本的应用，使包体积降低等。但这个在开发阶段似乎显得有些多余，所以我们需要禁用多 APK 构建特性以提高构建速度。

禁用 Multiple APK 构建的方法

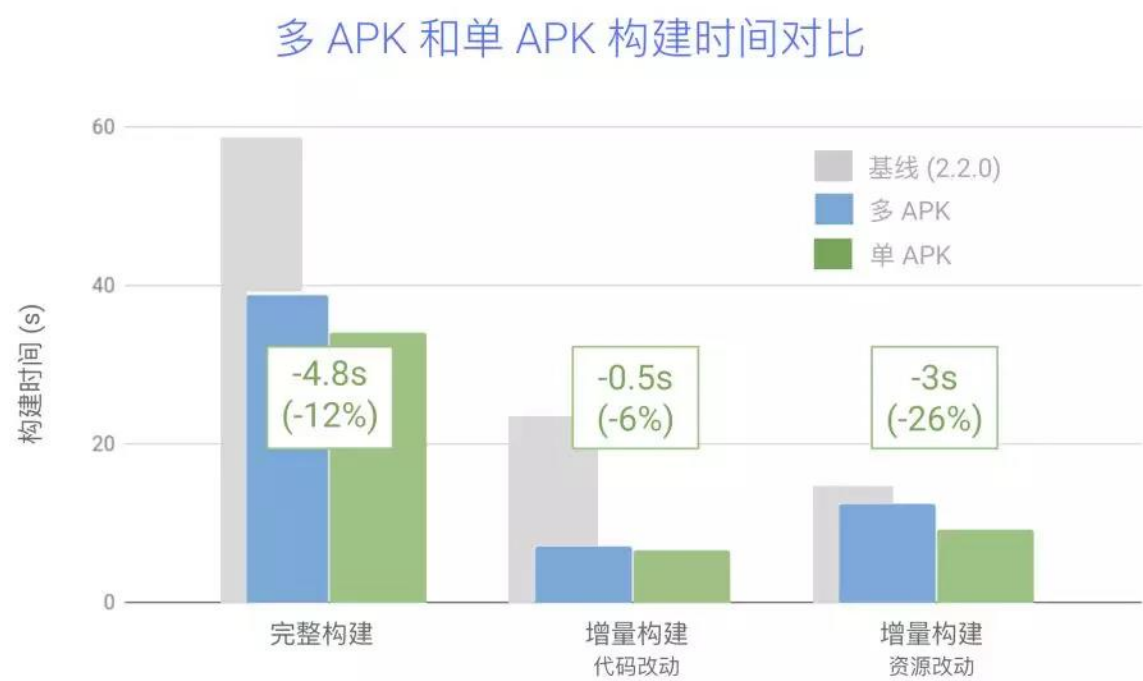
```
android {  
    if (project.hasProperty('devBuild')) {  
        splits.abi.enable = false  
        splits.density.enable = false  
    }  
}
```

```
-----  
$./gradlew santa-tracker:assembleDevelopmentDebug -PdevBuild
```

禁用多 APK 构建不能仅仅在 `splits` 里设置，因为这里的设置对工程里所有的构建变体都是可见的。正确的禁用多 APK 构建的方法是创建一个属性来做判断，这里我们设置了一个名为“devBuild”的属性，在构建的过程中把这个值传给 gradle，此时 gradle 会将 `splits.abi.enable` 和 `splits.density.enable` 设置为 `false`，它就不会生成多个 APK 了。



在 Android Studio 里，您可以通过偏好设置，构建、执行和部署分类里，选择编译器选项来为命令行加入参数: -PdevBuild，这样每次在构建的时候 Android Studio 会把这个值传递给 gradle 以避免生成多个 APK。



如上图所示，这是我在禁用了多 APK 之后的效果，各项指标都在继续降低。

小技巧 4: 最小化使用资源文件

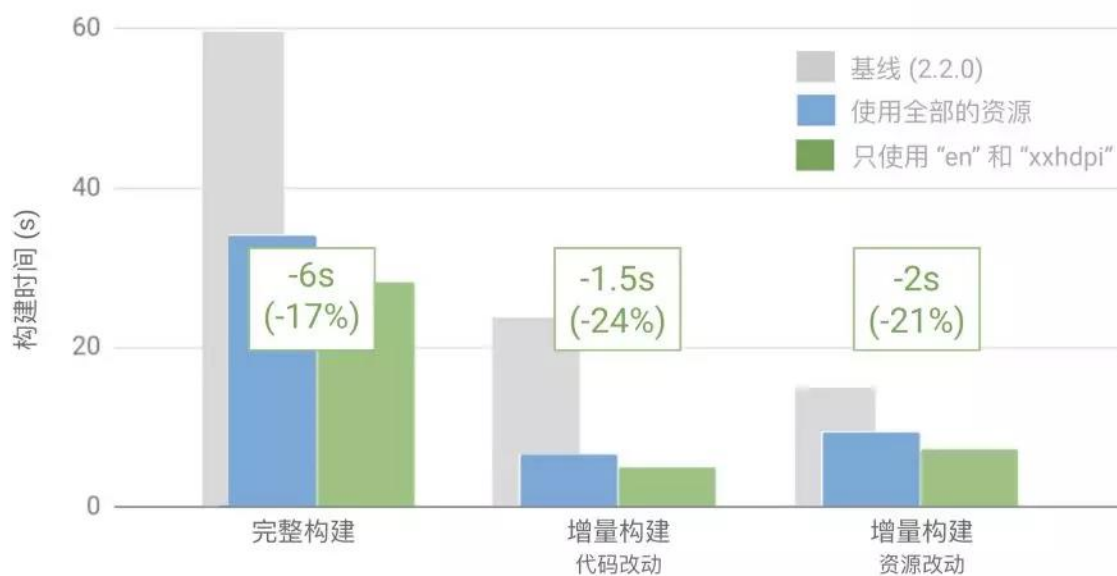



```
development {  
    minSdkVersion 21  
    resConfigs ("en", "xxhdpi")  
    ...  
}  
}
```

当您的应用包含大量本地化资源或者为不同像素密度加入了特别的资源时，您可能需要应用这个小技巧来提高构建速度——最小化开发阶段打包进应用的资源数量。

构建系统默认会将声明过或者使用过的资源全部打包进 APK，但在开发阶段我们可能只用到了其中一套而已，针对这种情况，我们需要使用 `resConfigs()` 来指定构建开发版本时所需要用到的资源，如语言版本和屏幕像素密度。

仅使用 英文 和 xxhdpi 像素密度资源的构建时间对比



这里我们看到了较大程度上的改观，全量构建的时间又降低了 6 秒，增量构建的时间也分别降低了 20% 以上。

小技巧 5: 禁用 PNG 压缩

Android 构建工具会默认压缩工程里的 PNG 图片资源。



与小技巧 4 一样，这个特性本身在打包发布阶段是相当有帮助的——PNG 压缩，但在开发阶段禁用这个功能可以提高构建效率。默认情况下，AAPT 会压缩工程的 PNG 资源以减小 APK 体积，根据图片的数量和大小，这个过程所消耗的时间有长有短。

禁用 PNG 压缩的方法

```
android {  
    if (project.hasProperty('devBuild')) {  
        splits.abi.enable = false  
        splits.density.enable = false  
        aaptOptions.cruncherEnabled = false  
    }  
}
```

```
-----  
$./gradlew santa-tracker:assembleDevelopmentDebug -PdevBuild
```

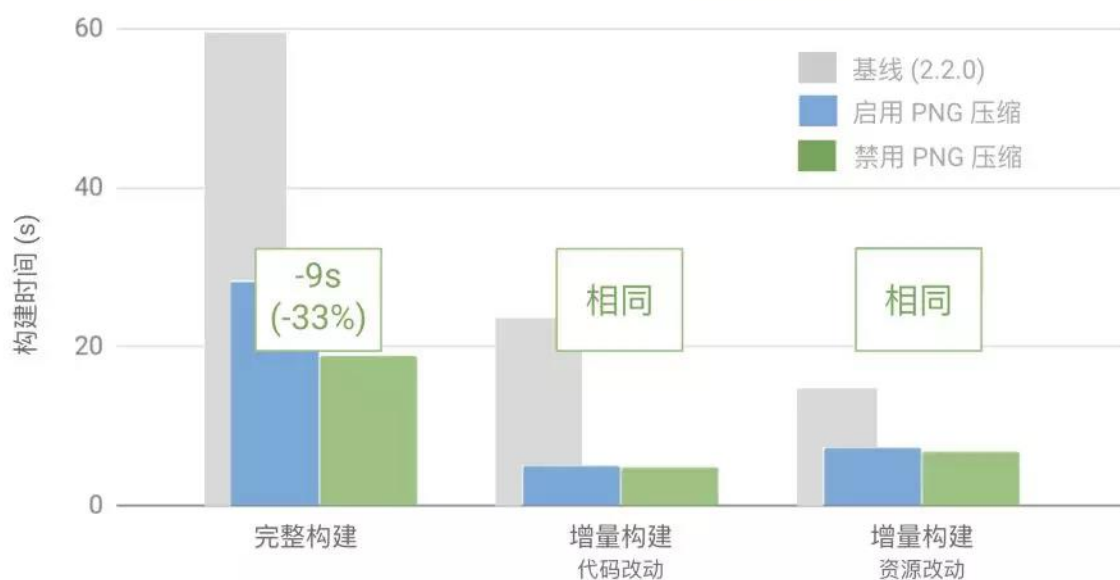
如果要避免使用 PNG 压缩，我们可以在小技巧 3 里提到的，在 devBuild 属性里加入 `aaptOptions.cruncherEnabled = false` 来实现，在构建的过程中把这个值传给 gradle，它就可以避免执行 PNG 压缩命令了。



另外一个避免压缩 PNG 的方法是使用把 PNG 转换成 WebP 格式的图片，对比 PNG 格式，WebP 可以减少最多 25% 的大小，同时 2.3 以上版本的 Android Studio 直接支持 PNG 到 WebP 格式的转换。

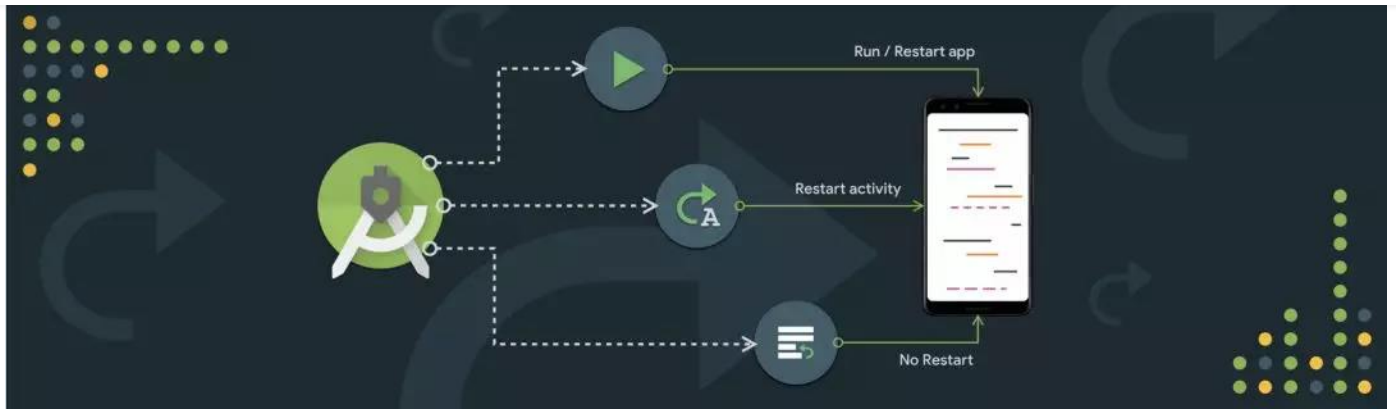
需要注意的是，API 级别 15 及更高可以支持不透明的 WebP 格式图片，如果是透明格式的 WebP，需要 API 级别 18 以及更高。

启用和禁用 PNG 压缩的构建时间对比



这可以看到全量构建又减少了 9 秒的时间，这也是因为 Google 追踪圣诞老人应用里有 3,500 多张 PNG 图片，这要花费大量的时间进行压缩计算，所以这方面的效率提升显得很明显，而其他增量构建只是维持了之前的情况。

特别提出一下关于 APK 体积的问题——对比了启用和禁用 PNG 压缩之后的 APK 体积之后，我们发现前后的体积并没有太大改变，这说明该工程里使用的 PNG 图片在导入之前已经经过了充分优化，PNG 压缩在这里实属多此一举。



从 Android Studio 3.5 版开始 (3.5 版目前在 Beta 构建渠道发布), 开发者们可以使用 Apply Changes 功能来提高构建性能, 它可以让代码和资源的改动直接生效而无需重启应用, 有时候甚至无需重启当前的 Activity。与 Instant Run 的实现方式不一样, Apply Changes 充分利用了 Android 8.0 以上版本操作系统的特性进行运行时检测, 从而动态的对类进行重新定义。因此, 如果您希望使用 Apply Changes, 则需要让您的工程运行在 Android 8.0 (API级别26) 以上的真机或者模拟器上。

小技巧 7: 避免被动的改动

```
def buildDateTime = new Date().format('yyMMddHHmm').toInteger()

android {
    defaultConfig {
        versionCode buildDateTime
        ...
    }
}
```

: (

我们通过一个很小的例子来说明这个小技巧: 我们把工程的版本号设定为基于当前时间的数字 (实际上大家应该不会这么操作), 这样的结果是每次构建的时候版本号都是新的, 工程的清单文件会因此发生改变, 最后带来的结果就是拖慢了本次的构建速度。



如图所示，我们发现增量构建的时间甚至增加了一倍，因此尽量不要在构建脚本里加入太多无意义的内容。

在 devBuild 构建里定义固定的版本号

```
def buildDateTime = project.hasProperty('devBuild')? 100: new
Date().format('yyMMddHHmm').toInteger()

android {
    ...
    defaultConfig {
        versionCode buildDateTime
    }
    ...
}
```

:)

解决这个问题并不难，我们可以通过在构建脚本里判断是否有 devBuild 标记，如果有的话，我们就把版本号设置为一个固定值就可以了。

```
apply plugin: 'io.fabric'


...

android {
    buildTypes {
        debug {
            ext.alwaysUpdateBuildId = false
            ...
        }
    }
}
```

这个例子里，我们故意在构建脚本中加入一些捣乱的代码以展现其带来的损失。同时也举一个在使用 Crashlytics 时的实际例子，这个插件默认会为每次构建中都加入唯一 ID 作为构建标识，这会带来不必要的时间损失，您可以通过在构建脚本里加入 `ext.alwaysUpdateBuildId = false` 来避免这个，当然也可以选择完全关闭 Crashlytics。

小技巧 8: 不使用动态版本标识

```
android {
    dependencies {
        compile 'com.android.support:appcompat-v7:+'
        ...
    }
}
```




Gradle 提供了一个非常方便的依赖库版本号管理功能，方便开发者们通过使用一个加号 “+” 标识希望使用这个依赖库的最新版本。但是使用动态版本有几个风险，从性能角度来说，Gradle 会每隔 24 小时去检查一次依赖库的更新，如果您的依赖库很多，而且都使用了动态获取最新版本的这个设定，那会对构建时候的性能产生一定的影响。

即使您不是特别在意这些性能损耗，但是它仍然是有风险的——依赖库的版本更新会让您的构建充满不确定性，可能两周之后您就在构建一个完全不一样的工程了，因为依赖库代码的更新对开发者们是不可见的。

gradle.properties:

```
org.gradle.jvmargs=-Xmx1536m
```

 调优这个数值

build.gradle

```
dexOptions {  
    javaMaxHeapSize = "4g"  
}
```

 可以移除

默认的构建环境里，我们会给 Gradle 分配 1.5G 的内存，但这个并非适用于所有的项目，您需要通过对这个数字对调优来得到适合您工程的最佳 Gradle 内存分配。

与此同时，从 Android Gradle 插件 2.1 版本之后，dex 已经默认在进程里了，所以如果您之前设定过 javaMaxHeapSize 值，可以选择删掉它了。

小技巧 10: 开启 Gradle 构建缓存

- Gradle 3.5 以上版本可以使用
- 与 2.3 版本中提供的构建缓存实现方法不同

```
# 在 gradle.properties 文件里加入  
org.gradle.caching=true
```

Gradle 新推出的缓存机制效果非常出色，我们建议大家尝试开启，最新的 Gradle 支持了 Kotlin 项目使用构建缓存，构建速度可以降低很多。Gradle 的构建缓存默认是不开启的，您可以通过在命令行里加入 `--build-cache` 参数或者在工程根目录的 `gradle.properties` 里加入 `org.gradle.caching=true` 为所有人启用构建缓存。您可以在[这个文档里](#)了解更多关于 Gradle 构建缓存的内容。

总结

全量构建

增量构建
代码改动

增量构建
资源改动

59秒 ➡ 19秒

24秒 ➡ 2秒

15秒 ➡ 4.5秒

快了 3 倍以上

快 12 倍以上

快了 3 倍以上

在实践了所有的速度提升小技巧之后，得到的整体的改善结果，全量构建的速度比之前快了三倍以上，而代码改动的增量构建则快了 12 倍以上，我们在 GitHub 上创建了一个[代码仓库](#)，大家可以下载并实践一下我们今天所提到的构建速度提升的技巧。更多关于如何提高应用构建速度的内容，请关注我们的[官方文档](#)。

[点击这里](#)提交产品反馈建议

android

发布于 2019-08-15

[Android](#) [Android 开发](#)

▲ 赞同 22 ▼

● 3 条评论

🚀 分享

♥ 喜欢

★ 收藏

📄 申请转载

...

推荐阅读

第一行代码Android(第二版) 观后感和分享（想学Android的...

深入学习Android：虚拟机&运行时



。最近忙得没时间，自己...
android的《第一行代码（第二
反）》。自己做Android也有好几年
了，这本书给我的第一个感觉就...

alazy

何继续深入的研究技术层的知识？
请教各位前辈指条明路 Android
Framework 如何学习，如...

weish... 发表于维术不多的...



ZYLAB 发表于Andro...
全面解析 Android 热修复原理

3 条评论

⇌ 切换为时间排序

写下你的评论...



乐未极

10-20

老师，那么，Android studio为什么显示设计编辑器在项目同步成功之前不可用？

👍 赞



乐未极

10-20

希望得到你的解答，我将非常感激。

👍 赞



乐未极

10-20

有人说同步，可是想同步Gradle也是失败啊！[捂脸]

👍 赞

