

# 启动优化方案的“第三板斧”终于齐了！

Dotry 鸿洋 11月6日

本文作者

---

作者：Dotry

链接：

<https://www.jianshu.com/p/418e34d9d253>

本文由作者授权发布。

最近我已经推送过两篇启动优化的文章，都是非常赞的文章，文中给出的方案都不是随便搜索就能找到的：

[都9102年了，Android 冷启动优化除了老三样还有哪些新招？](#)

这篇讲了通过通过 redex 重排列 class 文件来做极致优化。

[面试官：今日头条启动很快，你觉得可能是做了哪些优化？](#)

这篇重点讲解了头条如何multidex 在低端机的优化。

本篇文章也讲解了异步任务依赖关系构建依赖关系图，从而保证高效而不出错的并发。

这三篇文章我称之为启动优化三板斧，如果你都能吸收，启动优化你真的学到了不少东西！

## 1 前言

---

一个应用App的启动速度能够影响用户的首次体验，启动速度较慢(感官上)的应用可能导致用户再次开启App的意图下降，或者卸载放弃该应用程序。本文会通过以下几个方面来介绍应用启动的相关指标和优化，提供应用的启动速度。

整体文章思路如下：



这里图中"无向图"感觉是笔误，不太严谨，改图太麻烦了，大家明白意思即可。

## 2 冷启动&热启动

通常来说，启动方式分为两种：冷启动和热启动。

**冷启动：**当启动应用时，后台没有该应用的进程，这时系统会重新创建一个新的进程分配给该应用，这个启动方式就是冷启动。

**热启动：**当启动应用时，后台已有该应用的进程（例：按back键、home键，应用虽然会退出，但是该应用的进程是依然会保留在后台，可进入任务列表查看），所以在已有进程的情况下，这种启动会从已有的进程中启动应用，这个方式叫热启动。

两者之间的特点如下：

**冷启动:**系统会重新创建一个新的进程分配给该应用，从Application创建到UI绘制等相关流程都会执行一次。

**热启动：**应用还在后台，因此该启动方式不会重建Application，只会重新绘制UI等相关流程。

冷热启动时间的计算命令：

```
adb shell am start -W [packageName]/[packageName.XxxActivity]
```

参数说明：

- 1、ThisTime:一般和TotalTime时间一样。除非在应用启动时开了一个透明的Activity预先处理一些事再显示出主Activity，这样将比TotalTime小。
- 2、TotalTime:应用的启动时间。包含创建进程+Application初始化+Activity初始化到界面显示。
- 3、WaitTime:一般比TotalTime大点，包含系统影响的耗时。

对于我们的应用来说结果如下：

## 冷启动

```
C:\Users\xiuxen.zhan>adb shell am start -W com.hank.android.browser /com.hank.android.browser/BrowserActivity
Starting: Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] cmp=com.hank.android.browser/.BrowserActivity }
Status: ok
Activity: com.hank.android.browser/.BrowserActivity
ThisTime: 2040
TotalTime: 2040
WaitTime: 2093
Complete
```

## 热启动

```
C:\Users\xiuxen.zhan>adb shell am start -W com.hank.android.browser /com.hank.android.browser/BrowserActivity
Starting: Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] cmp=com.hank.android.browser/.BrowserActivity }
Warning: Activity not started, its current task has been brought to the front
Status: ok
Activity: com.hank.android.browser/.BrowserActivity
ThisTime: 357
TotalTime: 357
WaitTime: 391
Complete
```

可以看到两者时间相差比较大。

根据该命令基本可以看出一个应用的启动速度了，从冷启动热启动的相关关系，当我们需要优化启动速度的时候，优化冷启动速度即可。

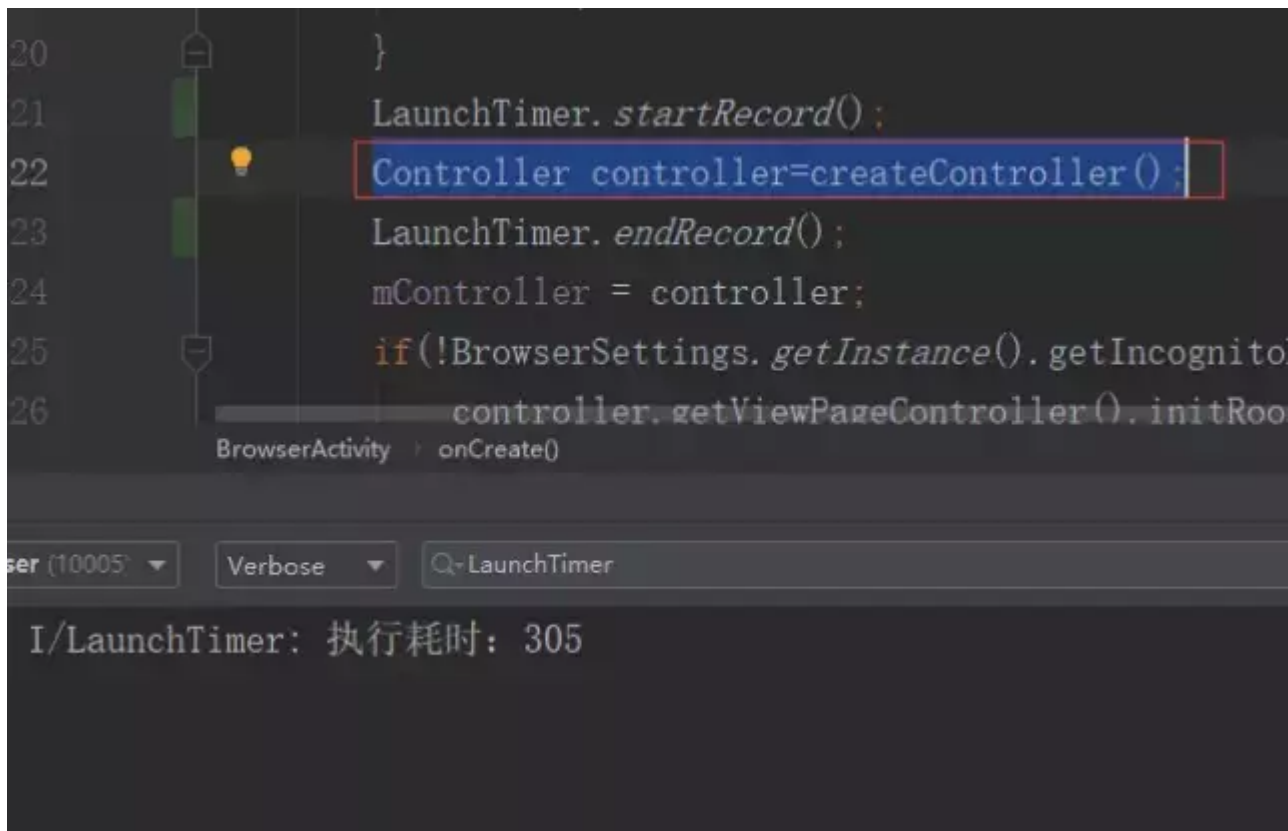
但是该命令我们只是大概知道应用的启动速度，但并不知道我们的应用具体哪个位置耗时，影响启动速度，后续我会介绍如何获取启动具体耗时时间。

## 3 常规获取时间方法

常规获取时间方法无非就是在方法执行前记录下时间，在方法执行完毕后记录时间，两者时间之差就是该方法执行的时间，封装一个基础类如下：

```
public class LaunchTimer {  
  
    private static final String TAG = "LaunchTimer";  
    private static long sTime;  
  
    public static void startRecord() {  
        sTime = System.currentTimeMillis();  
    }  
  
    public static void endRecord() {  
        long cost = System.currentTimeMillis() - sTime;  
        NLog.i(TAG, "执行耗时：%s", cost);  
    }  
}
```

使用方式如下，可以直观的看出createController方法执行的时间



这样已经很直观了，可以具体到该方法的执行时间，如果要继续分析则对该方法内部继续执行该代码即可。但是这里有一个问题如果要知道10个或者更多方法的执行时间，这个方法看起来是可以，但写起来过于繁琐，且不符合程序员的习惯，关于这种场景后面会介绍如何处理。

## 4 TraceView和SysTrace工具使用

TraceView使用：TraceView是Android平台配备一个很好的性能分析工具，它可以通过图形化的方式让我们了解我们要跟踪的程序的性能，并且能具体到方法。

使用方式：

1. 通过Android studio自带的traceview查看（Android profiler）。
2. 通过Android SDK自带的Debug。
3. 通过DDMS中的traceview查看。

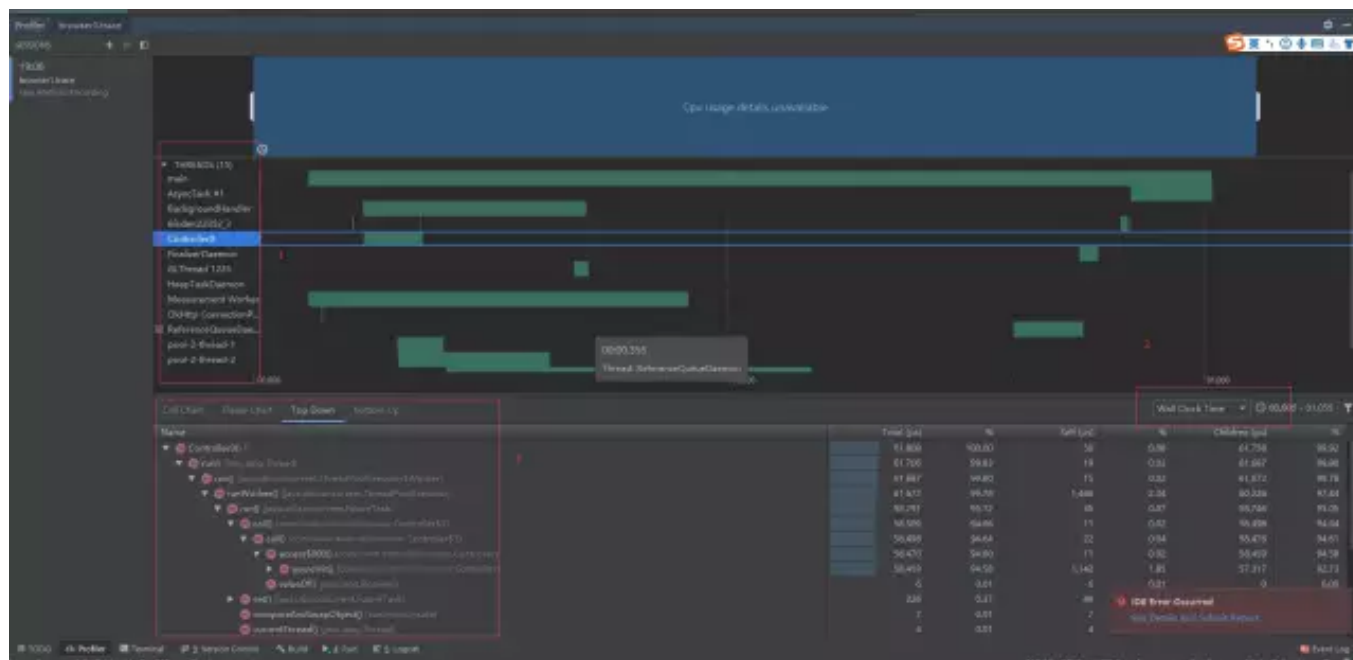
本文主要介绍第二种方式，通过sdk中的方法，对应用进行打点获取相关信息：

```
public void onCreate(final Bundle icle) {
    setTheme(R.style.BrowserTheme);
    Intent intent = getIntent();
    NLog.i(LOGTAG, "onCreate");
    super.onCreate(icle);
    //开始记录 · 且该方法可以设置文件大小和路径
    Debug.startMethodTracing("browser.trace");
    Controller controller=createController();
    mController = controller;
    getWindow().getDecorView().setSystemUiVisibility(
        View.SYSTEM_UI_FLAG_LAYOUT_FULLSCREEN|View.SYSTEM_UI_FLAG_LIGHT_STATUS_BAR);
    controller.handleThirdPartyIntent(intent);
    //结束记录
    Debug.startMethodTracing();
}
```

如上可以在目录下生成如下文件

```
/sdcard/Android/data/com.xxx.xx.browser/files/browser.trace
```

导出改文件，通过Android Studio的profile打开改文件



1 处可以看出有多少线程。

2 处可以看出具体方法的耗时。

3 处有两个选项:

**wall clock time:**

代码在线程上执行的真正时间[有一部分是等待cpu轮询时间]

**thread time :**

cpu执行的时间。

一般是优化的是cpu执行时间。

关于该图如何查看和阅读可以参照文章Android性能优化—TraceView的使用

<https://www.jianshu.com/p/7e9ca2c73c97>

结合业务代码走查发现Controller0线程为一个线程，因此主线程一些操作可以放进去执行，从而减少main线程的耗时。走查代码可以发现在Controller0线程中执行如下

```
requestPermission();
ExecutorService service = Executors.newSingleThreadExecutor(new NamedThreadFactory("Controller"));
Future<Boolean> future = service.submit(new Callable<Boolean>() {
    @Override
    public Boolean call() throws Exception {
        try {
            asyncInit();

        } catch (Exception e) {
            return false;
        }
        return true;
    }
});
```

requestPermission()方法执行在main线程中，因此我们可以把其放在Controller0线程中执行，从而减少main线程的时间

```
ExecutorService service = Executors.newSingleThreadExecutor(new NamedThreadFactory("Controller"));
Future<Boolean> future = service.submit(new Callable<Boolean>() {
    @Override
    public Boolean call() throws Exception {
        try {
            requestPermission();
            asyncInit();

        } catch (Exception e) {
            return false;
        }
        return true;
    }
});
```

```
    }  
  });
```

经测试发现无问题，且对比此时的trace文件发现修改前后main线程时间相对来说减少很多。

## SysTrace使用：

SysTrace用于收集可帮助您检查原生系统进程的详细系统级数据，例如CPU调度、磁盘活动、应用线程等，并解决掉帧引起的界面卡顿。

使用方式：

在代码的开始位置加上tag

```
TraceCompat.beginSection("AppOnCreate");
```

然后指定位置结束

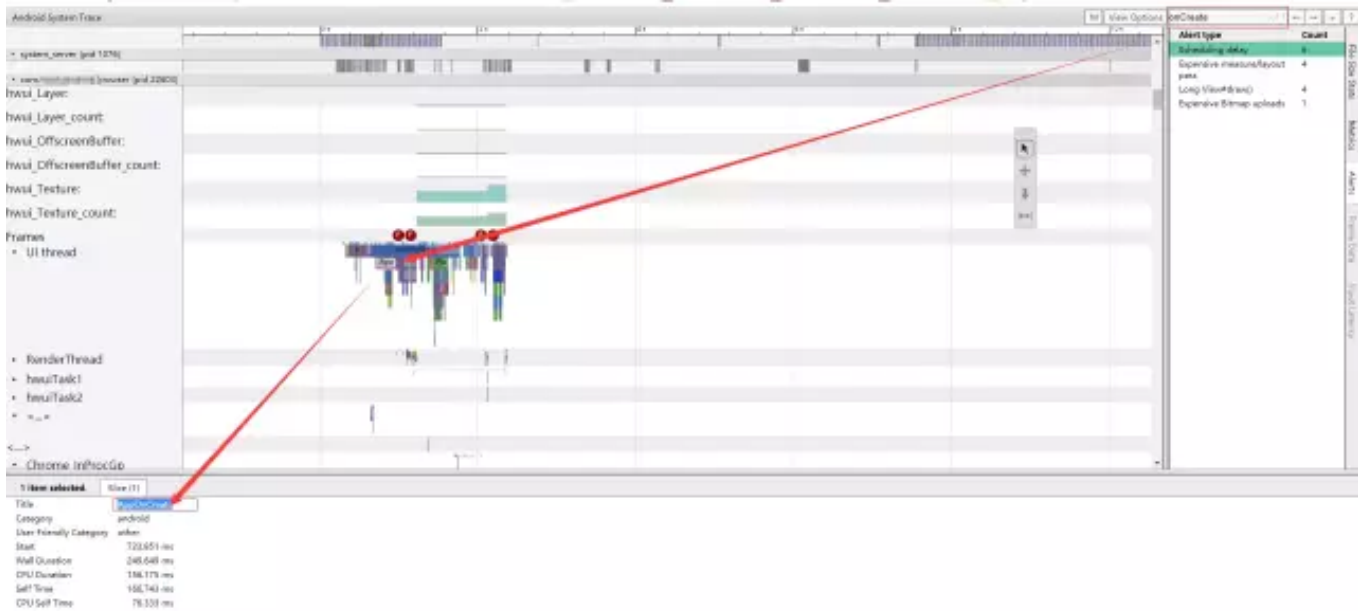
```
TraceCompat.endSection();
```

即可以抓取到整个应用在此过程的相关信息,例如在onCreate方法中添加上述两行代码，执行相关python命令：

```
python systrace.py -b 32768 -t 10 -a com.xxx.xxx.browser -o browser.html sched gfx view wm am app
```

操作相关应用，即可以抓取整个过程的相关信息：





即可以看到添加的tag“AppOnCreate”，对应的时间信息：

**Wall Duration** 代表的方法从开始到结束的耗时

**CPU Duration** 代表CPU的执行时间

通过这两个参数可以看出此流程中执行的时间等相关信息。

我们都知道CPU是轮询模式，因此优化的方向可以说是两个方向，提高CPU的核数和优化CPU执行的时间。

关于Sysrtrace的具体使用及命令可以参照:

## 性能优化工具 (二) - Systrace

<https://www.jianshu.com/p/fa6cfad8ccc2>

## 了解 Systrace

<https://source.android.com/devices/tech/debug/systrace>

## 5 通过AOP获取时间

AOP:面向切面编程(Aspect-Oriented Programming)。

如果说，OOP如果是把问题划分到单个模块的话，那么AOP就是把涉及到众多模块的某一类问题进行统一管理。打个比方Android 里面PMS，AMS都拥有各自的职责，但是他们都需要通过log系统管理log，这就是一种AOP思想。

AspectJ实际上是对AOP编程思想的一个实践，当然，除了AspectJ以外，还有很多其它的AOP实现，例如ASMDex，但目前最好、最方便的，依然是AspectJ。

AspectJ的使用如下：

根目录gradle下引用：

```
classpath 'com.hujiang.aspectjx:gradle-android-plugin-aspectjx:2.0.0'
```

app目录gradle文件下引用：

```
implementation 'org.aspectj:aspectjrt:1.8.+'
```

此两处引用完成之后，就是代码编写：

```
package com.xx.xxx.browser.aspect;

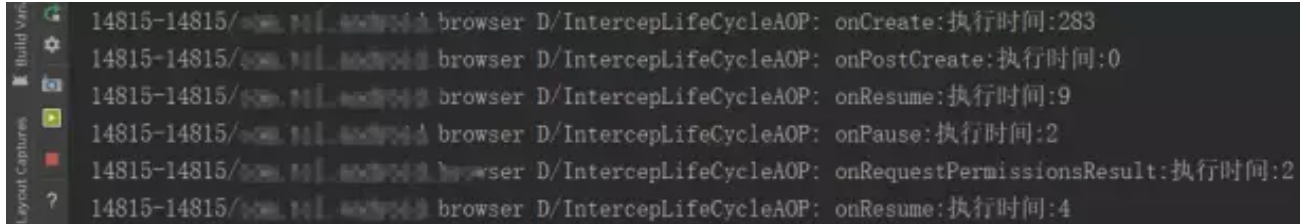
import android.util.Log;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.Signature;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class IntercepLifeCycleAOP {
    //获取该Activity下的所有on开头的方法耗时
    @Around("execution(* com.xxx.xxx.BrowserActivity.on**(..))")
    public Object getTime(ProceedingJoinPoint joinPoint) {
        Object proceed = null;
        long start = System.currentTimeMillis();
        try {
            proceed = joinPoint.proceed();
        } catch (Throwable throwable) {
            throwable.printStackTrace();
        }
        long end = System.currentTimeMillis();
        Log.d("IntercepLifeCycleAOP", joinPoint.getSignature().getName() + ":执行时间:" + (end - start));
        return proceed;
    }
}
```

```
}  
  
}
```

引入之后结果如下：



```
14815-14815/com.hujiang.aspectjx.browser D/InterceptLifeCycleAOP: onCreate:执行时间:283  
14815-14815/com.hujiang.aspectjx.browser D/InterceptLifeCycleAOP: onPostCreate:执行时间:0  
14815-14815/com.hujiang.aspectjx.browser D/InterceptLifeCycleAOP: onResume:执行时间:9  
14815-14815/com.hujiang.aspectjx.browser D/InterceptLifeCycleAOP: onPause:执行时间:2  
14815-14815/com.hujiang.aspectjx.browser D/InterceptLifeCycleAOP: onRequestPermissionsResult:执行时间:2  
14815-14815/com.hujiang.aspectjx.browser D/InterceptLifeCycleAOP: onResume:执行时间:4
```

可以看到具体方法的耗时。

采用注解方式，其中Around 需要有一定的AspectJ相关的语法，具体参考：

AspectJ 在 Android 中的使用

<https://blog.csdn.net/yxhuang2008/article/details/94193201>

gradle\_plugin\_android\_aspectjx

[https://github.com/HujiangTechnology/gradle\\_plugin\\_android\\_aspectjx](https://github.com/HujiangTechnology/gradle_plugin_android_aspectjx)

hugo

<https://github.com/JakeWharton/hugo>

## 6 用户体验优化

1. 主题优化：通过给应用设置一个透明主题，在应用启动完成之后，再给其赋予本该有的主题，通过对启动页的主题设置后，就会将白屏/黑屏抹去，用户点击App的图标就展示启动图，让用户先产生启动很快的“错觉”。

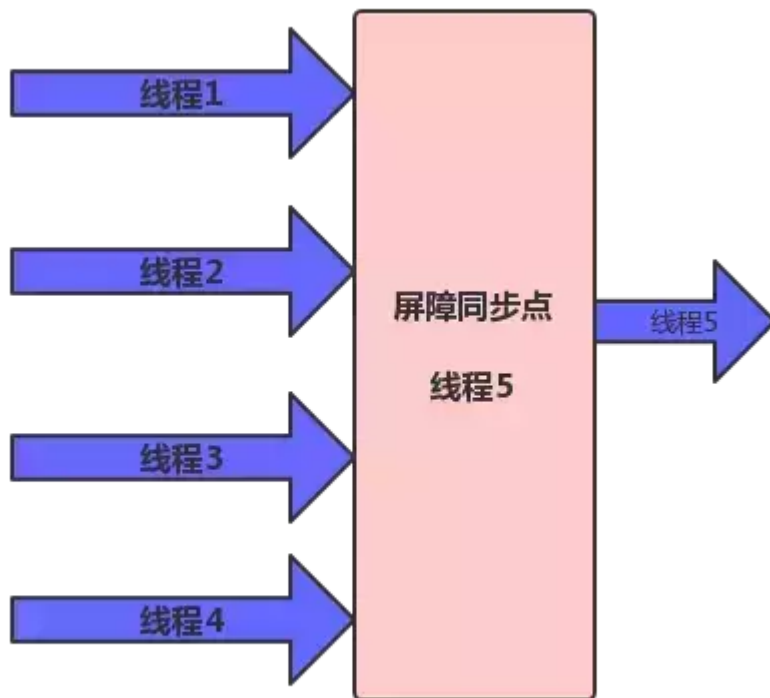
2. 动画兼容：根据不同年代的机型可以选择执行或不执行相关动画，或者延迟其他相关操作,可根据device-year-class来判断具体年份

<https://github.com/facebook/device-year-class>

3. UI布局优化。

## 7 异步加载

1. 采用线程加载一些资源，比如sdk初始化，配置信息拉取等相关资源。线程，线程池，IntentServices均可以，配合延迟效果更好。
2. 当我们采用线程之间的可能会存在各线程之间相互等待依赖等相关问题，资源A线程必须在资源B加载完成，才能加载，但两者又会在不同的线程之间，此时简单的办法可以采用CountDownLatch来实现。其整体思路如下图



具体使用方法可以参照文章Android 并发之CountDownLatch、CyclicBarrier的简单应用

<https://www.jianshu.com/p/31a100fc945d>

3. 使用 Pipeline 机制，根据业务优先级规定业务初始化时机，制定启动框架，它们为各个任务建立依赖关系，最终构成一个有向无环图。对于可以并发的任务，会通过线程池最大程度提升启动速度。无论是微信的mmkernel 还是阿里的Alpha 都具备这种能力。

<https://github.com/alibaba/alpha>

#### 4. 其他方案：

除了上述几种，我们也可以利用IdealHandler，dex分包等相关方式做到启动优化。

## 8 总结

上面主要介绍了如何获取启动的相关事件和相关优化知识点。关于时间就是尽量使用工具，关于优化整体思路就是能预加载能延迟加载的资源尽量去预加载去延迟加载，能异步的业务尽量异步。

当然优化这个话题也是要根据具体的业务逻辑来定，总之：

**对于启动优化要警惕 KPI 化，我们要解决的不是一个数字，而是用户真正的体验问题。**

上述只是提供一些思路和方式，还有很多技巧，欢迎给位大佬评论指出。

推荐阅读：

[千万级别的app中验证过，5分钟让你的 SDK 拥有热修复能力](#)

[我开发了一个神器！](#)

[走心推荐几个必备的插件](#)



哒哒哒哒哒



扫一扫 关注我的公众号

如果你想要跟大家分享你的文章，欢迎投稿~

r(^ 0 ^)明天见！

[阅读原文](#)