链滴

浅谈 App 的启动优化

作者：xuexiangjys

原文链接：https://ld246.com/article/1669044813401

来源网站：链滴

1. 应用启动的方式

在Android中，应用启动一般可分为三种：**冷启动**、**温启动**、**热启动**。

那么什么是**冷启动**、**温启动**和**热启动**呢？下面我们来简单看一下它们的定义：

- 冷启动：当启动应用时，后台没有该应用的进程。这时系统会又一次创建一个新的进程分配给该应，这个启动方式就是冷启动。
- 温启动：当启动应用时，后台已有该应用的进程，但是Activity可能因为内存不足被回收。这样系会从已有的进程中来启动这个Activity，这个启动方式叫温启动。
- 热启动：当启动应用时，后台已有该应用的进程，且Activity仍然存在内存中没有被回收。这样系直接把这个Activity拉到前台即可，这个启动方式叫热启动。

由于冷启动相对于其他启动方式多了进程的创建（Zygote进程fork创建进程）以及应用的资源加载和始化（Application的创建及初始化），所以相对来说会比较耗时，所以我们一般说的App启动优化般指的都是App的冷启动优化。

2. 优化方案

App启动优化的本质就是：启动速度和体验的优化。

这就好比早些年你去饭店吃饭，你想要点餐但等了半天都没有服务人员过来，可能就等得不耐烦直接开了。同样的，对于APP来说，如果用户点击App后长时间都打不开，用户就很可能失去耐心而卸载用。

所以**启动速度是用户对我们App的第一体验。**如果启动速度过慢，用户第一印象就会很差，这样即使功能做出花来，用户也不会愿意去使用。

其实要想优化App的启动体验，关键就是要让用户更快地获取到应用的内容（流畅，不卡顿、不等待，那么我们应该怎么做呢？

2.1 案例分析

这里我们还是先以之前说的去饭店吃饭为例来展开讨论。现在的饭店竞争是尤为得激烈，为了能够提顾客的体验、留住顾客，真的是使出了浑身解数，除了口味之外，服务品质也是被摆在了越来越重要位置。

就比方说：

- 为了能够更快地给用户提供点餐服务，现在的饭店每个座子上基本都贴上了点餐的二维码。
- 一些热门需要排队的餐馆，在前台张贴了二维码，提供排队取号和提前点餐服务。
- 一些常年爆满需要排队的餐馆，还提供了零食、茶水、棋牌以及美甲等服务。
- 有些餐厅为了提高上菜的速度和体验，设置了倒计时免单的服务。

以上的改进措施，都是这些年餐饮行业在竞争下，不断提高服务质量的产物。可以说谁的服务质量落了，谁就有可能被淘汰。

其实我们仔细分析一下上面所列举的，不难看出有很多是可以让我们借鉴的。

（1）案例1

分析：饭店提供了点餐的二维码，本质上是将一件被动等待转变为主动请求的一种过程，客观上减少等待的时间，从而提高了速度。

类比：这对应我们的应用程序，就像原先一些耗时不必要的三方库需要被动等待其初始化完毕程序才继续进行，转变为先不初始化这部分耗时的三方库，等真正用到时再进行初始化；又类似我们应用程的游客模式，无需被迫进行一堆复杂的用户注册过程，就可以直接进入程序使用，待涉及一些用户信功能的时候再提示用户注册。

（2）案例2

分析：热门餐馆同时提供排队取号和提前点餐服务，本质上是将原先无法同时进行的操作（需要先排队等到座位，才能扫描座位号点餐）变成了可同时进行的操作，将串行任务转化为并行任务，从而节了时间。

类比：这对应我们的应用程序，就是将一些原本在主线程串行执行的耗时资源/数据加载，改为在子程中并发执行。这在几个耗时任务耗时差距不大的时候优化尤为明显。

（3）案例3

分析：在顾客排队等待的时候，提供零食、茶水、棋牌以及美甲等服务，本质上就是让顾客提前享受餐馆的服务，从而缓解顾客等待的焦虑。

类比：这对应我们的应用程序，就是开屏启动页。在Android12上，Google强制增加了这个开屏页就是为了让用户提前看到你的应用页面，让用户产生应用启动很流畅的假象，从而提高用户的启动体。

（4）案例4

分析：设置倒计时免单服务，本质上就是给等待加上了进度条上限以及超时赔偿。俗话说最让人害怕是等待，比等待更让人害怕的是看不见尽头的等待。而为等待设置上限，可以极大地缓解顾客等待的虑，毕竟等待超时了也是会有补偿的。

类比：这对应我们的应用程序，就是一些应用（比如游戏）初次启动会非常耗时，所以它们通常会在动页增加一个初始化/加载进度条页面，来告诉用户啥时候能加载完，而不是无止境未知的等待。

通过分析，我们可以看到（1）、（2）两种操作是从技术的角度来实现的优化，而（3）、（4）两种作则更多的是从业务的角度去实现的优化。

2.2 优化策略

从上面的案例分析，我们可以得出，应用启动优化，我们可以分别从技术和业务的角度来进行决策。

2.2.1 技术优化

（1）针对启动流程任务进行梳理。

- 无耗时任务 -> 主线程
- 耗时且需要同步任务 -> 异步线程 + 同步锁
- 耗时且无需同步任务 -> 异步线程

（2）非必要不执行。

- 非必要数据 -> 懒加载

- 非必要任务 -> 延迟/空闲执行
- 非必要界面/布局 -> 延迟加载
- 非必要功能 -> 删除/插件化

（3）数据结构优化，减小初始化时间。

- 数据结构尽可能复用，避免内存抖动
- 数据结构申请的空间要恰到好处，不能过大（占用空间，创建慢）也不能过小（频繁扩充，效率低）
- 对于必要且量大的数据，可采取分段加载。
- 重要的资源本地缓存

2.2.2 业务优化

（1）业务流程整合。

- 多个相关的串行业务整合为统一的一个业务。
- 不相关的串行业务整合为并行的业务。

（2）业务流程拆分调整。

- 对业务进行拆分，拆分出主要（必要）业务和次要（非必要）业务。
- 分别对主要业务和次要业务进行优先级评估，业务执行按优先级从高到底依次执行。

2.3 优化方向

在优化之前，让我们先来分析一下冷启动的过程：

Zygote创建应用进程 -> AMS请求ApplicationThread -> Application创建 -> attachBaseContext onCreate -> ActivityThread启动Activity -> Activity生命周期(创建、布局加载、屏幕布置、首帧制)

以上过程，只有Application和Activity的生命周期这两个阶段对我们来说是可控的，所以这就是我们优化方向。

3. 优化措施

3.1 启动流程优化

1.依据之前我们列举的技术优化策略，首先需要对启动的所有任务流程进行梳理，然后对其执行方式行优化。

- 只有必要且非耗时的任务在 主线程执行。
- 耗时且需要同步的任务，使用 异步线程 + 同步锁的方式执行。
- 耗时且无需同步的任务在 异步线程执行。

2.第三方SDK初始化优化。

- 对于那些在启动时非必要的第三方SDK，可以延迟初始化。
- 对于初始化耗时的第三方SDK，可以开启一个后台服务/异步线程进行初始化。

3.使用任务执行框架。

这里我们还可以使用一些第三方的任务启动框架，对启动流程进行优化。下面我就拿我开源的XTask单介绍一下：

这里我们模拟了三种类型的任务：

- 1.优先级最重要的任务，执行时间50ms；
- 2.单独的任务，没有执行上的先后顺序，但是需要同步，每个执行200ms；
- 3.耗时、最不重要的任务，等所有任务执行完毕后执行，每个执行需要1000ms。

优化前

```
/**
 * 优化前的写法, 这里仅是演示模拟，实际的可能更复杂
 */
private void doJobBeforeImprove(long startTime) {
    new TopPriorityJob(logger).doJob();
    for (int i = 0; i < 4; i++) {
        new SingleJob((i + 1), logger).doJob();
    }
    new LongTimeJob(logger).doJob();
    log("任务执行完毕，总共耗时:" + (System.currentTimeMillis() - startTime) + "ms");
}
```

执行结果：

由于所有的任务都是执行在主线程，串行执行，所以花了大约1865ms。

优化后

```
/**
 * 优化后的写法, 这里仅是演示模拟，实际的可能更复杂
 */
private void doJobAfterImprove(final long startTime) {
    ConcurrentGroupTaskStep groupTaskStep = XTask.getConcurrentGroupTask();
    for (int i = 0; i < 4; i++) {
        groupTaskStep.addTask(buildSingleTask(i));
    }
    XTask.getTaskChain()
            .addTask(new MainInitTask(logger))
            .addTask(groupTaskStep)
            .addTask(new AsyncInitTask(logger))
            .setTaskChainCallback(new TaskChainCallbackAdapter() {
                @Override
                public void onTaskChainCompleted(@NonNull ITaskChainEngine engine, @NonNull
ITaskResult result) {
                    log("任务完全执行完毕，总共耗时:" + (System.currentTimeMillis() - startTime) + "m
");
                }
            }).start();
    log("主线程任务执行完毕，总共耗时:" + (System.currentTimeMillis() - startTime) + "ms");
}
```

原文链接：浅谈 App 的启动优化

```
@NonNull
private XTaskStep buildSingleTask(int finall) {
    return XTask.getTask(new TaskCommand() {
        @Override
        public void run() throws Exception {
            new SingleJob((finall + 1), logger).doJob();
        }
    });
}
```

执行结果：

由于只有优先级最重要的任务在主线程执行，其他任务都是异步执行，所以主线程任务执行消耗57m，所有任务执行消耗1267ms。

3.2 IO优化

3.网络请求优化。

● 启动过程中避免不必要的网络请求。对于那些在启动时非必要执行的网络请求，可以延时请求或者用缓存。

● 对于需要进行多次串行网络请求的接口进行优化整合，控制好请求接口的粒度。比如后台有获取用信息的接口、获取用户推荐信息的接口、获取用户账户信息的接口。这三个接口都是必要的接口，且

在先后关系。如果依次进行三次请求，那么时间基本上都花在网络传输上，尤其是在网络不稳定的情下耗时尤为明显。但如果将这三个接口整合为获取用户的启动（初始化）信息，这样数据在网络中传的时间就会大大节省，同时也能提高接口的稳定性。

4.磁盘IO优化

- 启动过程中避免不必要的磁盘IO操作。这里的磁盘IO包括：文件读写、数据库（sqlite）读写和SharePreference等。
- 对于启动过程中所必须的数据加载，选择合适的数据结构。可以选择支持随机读写、延时解析的数存储结构以替代SharePreference。
- 启动过程中避免大量的序列化和反序列化。

3.3 线程优化

我们在开发应用的过程中，都或多或少会使用到线程。当我们创建一个线程时，需要向系统申请资源分配内存空间，这是一笔不小的开销，所以我们平时开发的过程中都不会直接操作线程，而是选择使线程池来执行任务。

但问题就在于如果线程池设置不对的话，很容易被人滥用，引发内存溢出的问题。而且通常一个应用有多个线程池，不同功能、不同模块乃至是不同三方库都会有自己的线程池，这样大家各用各的，就难做到资源的协调统一，劲不往一处使。

3.3.1 线程优化分析

1.线程池耗时分析。

要想进行线程优化，首先我们就需要了解线程池在使用过程中，哪些地方比较耗时。

- 首先，从常规上来讲，线程池在使用过程中，线程创建、线程切换和CPU调度比较耗时。
- 其次，需要从业务的角度去分析当前应用的线程使用情况，到底是哪些任务导致线程池大量的创建执行耗时。这里我们可以使用Hook的方式，去Hook线程的创建(ThreadFactory的 newThread方法)执行进行统计。
- 最后，我们需要结合业务的重要性（优先级）以及其相应的线程执行开销，进行综合考量，调整线池的执行策略。

2.线程池执行分析。

然后再让我们看看线程池的执行逻辑：

我们知道，一个线程池通常由一个核心线程池和一个阻塞队列组成。那么当我们调用线程池去执行一任务的时候，线程池是如何执行的呢？

核心线程池 -> 阻塞队列 -> 最大线程数(新建线程） -> RejectedExecutionHandler（拒绝策略）

- 当线程池中的核心线程池线程饱和后，任务会被塞进阻塞队列中等待。
- 如果阻塞队列也满了，线程池会创建新的线程。
- 但是如果目前线程池中的线程总数已经达到了最大线程数，这个时候会调用线程池的拒绝策略（默是直接中断，抛出异常）。

其中核心线程池的最大线程数并不是设置的越大越好，为什么这么说？因为CPU的处理能力是有限的

四核的CPU一次也只能同时执行四个任务，如果核心线程池数设置过大，那么各任务之间就会互相竞CPU资源，加大CPU的调度消耗，这样反而会降低线程池的执行效率。

一般来说，核心线程池的最大线程数满足下面的公式：

最佳核心线程数目 = （（线程等待时间 + 线程CPU时间）/ 线程CPU时间 ）* CPU数目

（1）线程等待时间所占比例越高，需要越多线程。

（2）线程CPU时间所占比例越高，需要越少线程。

这里我们考虑两种最常见的情况：

● **CPU密集型任务**:这种任务绝大多数时间都在进行CPU计算，线程等待时间几乎为0，因此这时最佳心线程数为 n + 1（或者为n），这里n为CPU核心数。

● **IO密集型任务**:这种任务，CPU通常需要等待I/O（读/写）操作，这样CPU的处理时间和等待时间几差不多，因此这时最佳核心线程数为 2n + 1（或者为2n），这里n为CPU核心数。

3.3.2 线程优化目标

通过上面对线程池的分析，我们可以知道：

● 线程池设置过大，会侵占内存，互相竞争CPU资源，增加CPU的调度耗时。

● 线程池设置过小，任务会阻塞串行，降低线程池执行效率。

因此，我们**线程优化的目标**是：

（1）拥有可以统筹全局的统一的线程池。

（2）能根据机器的性能来控制数量，合理分配线程池大小。

（3）能够根据业务的优先级进行调度，优先级高的先执行。

3.3.3 线程优化具体措施

1.建立主线程池+副线程池的组合线程池，由线程池管理者统一协调管理。主线程池负责优先级较高任务，副线程池负责优先级不高以及被主线程池拒绝降级下来的任务。

这里执行的任务都需要设置优先级，任务优先级的调度通过 PriorityBlockingQueue队列实现，以下主副线程池的设置，仅供参考：

● 主线程池：核心线程数和最大线程数：2n（n为CPU核心数），60s keepTime，PriorityBlockingQueue（128）。

● 副线程池：核心线程数和最大线程数：n（n为CPU核心数），60s keepTime，PriorityBlockingQue（64）。

2.使用Hook的方式，收集应用内所以使用 newThread方法的地方，改为由线程池管理者统一协调管。

3.将所有提供了设置线程池接口的第三方库，通过其开放的接口，设置为线程池管理者管理。没有提设置接口的，考虑替换库或者插桩的方式，替换线程池的使用。

3.4 闪屏优化

闪屏优化属于启动用户体验的优化。毕竟谁也不想使用页面一闪一闪的应用。

1.设置自定义闪屏页。

设置自定义的闪屏页可以提高我们启动的"视觉速度"。通常会设置一个背景，然后把logo居中显示，以使用xml文件来布局（注意，该图片不可展示动画，并且展示时间也不可控）。这种方式可以给用一种启动非常快的感觉，不仅解决了启动白屏的问题，并且展示了品牌logo也有助于提升品牌认知。

(1) 使用xml自定义一张带有logo的图片。

```
<?xml version="1.0" encoding="utf-8"?>
<layer-list xmlns:android="http://schemas.android.com/apk/res/android"
    android:opacity="opaque">
    <item android:drawable="?attr/xui_config_color_splash_bg"/>

    <item android:bottom="?attr/xui_config_app_logo_bottom">
        <bitmap
            android:gravity="center"
            android:src="?attr/xui_config_splash_app_logo"/>
    </item>

    <item android:bottom="?attr/xui_config_company_logo_bottom">
        <bitmap
            android:gravity="bottom"
            android:src="?attr/xui_config_splash_company_logo"/>
    </item>
</layer>
```

(2) 把这张图片通过设置主题的 android:windowBackground属性方式显示为启动闪屏。

```
<style name="XUITheme.Launch.Demo">
    <item name="android:windowBackground">@drawable/xui_config_bg_splash</item>
    <item name="xui_config_splash_app_logo">@drawable/ic_splash_app_logo_xui</item>
    <item name="xui_config_splash_company_logo">@drawable/ic_splash_company_logo_xue
iang</item>
</style>
```

(3) 在manifest中将主页的主题设置为刚才带启动图片的 Launch主题。

```
<activity
    android:name=".activity.MainActivity"
    android:theme="@style/XUITheme.Launch.Demo">
</activity>
```

(4) 代码执行到主页面的onCreate的时候设置为程序正常的主题，这样就切回到正常主题背景了。

```
public class BaseActivity extends XPageActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        setTheme(R.style.AppTheme);
        super.onCreate(savedInstanceState);
    }
}
```

2.使用Android12提供的启动画面Splash Screen。

https://developer.android.google.cn/guide/topics/ui/splash-screen

3.如果是因为某些不可抗拒的原因（例如一些大型游戏的首次加载），导致第一次启动非常慢的话，以在主页面进入之前，增加一个进度条加载页面。这样的好处就是要告诉用户，到底需要多久才能加完毕，给用户一个明确的信息，缓解用户的等待焦虑。

4.减少首页的跳转层次。除非某些不可抗拒的原因（比如广告作为一项重要收入来源），尽量不要设启动页（SplashActivity），因为打开和关闭任何一个Activity都是需要消耗时间的，每多一个Activit的跳转就意味着我们主页面的打开时间会被延长。

3.5 主页面优化

主页面的启动和显示也是app启动非常重要的一部分。

3.5.1 布局优化

布局优化的核心就是：提高页面渲染的速度，防止页面过度渲染导致耗时。

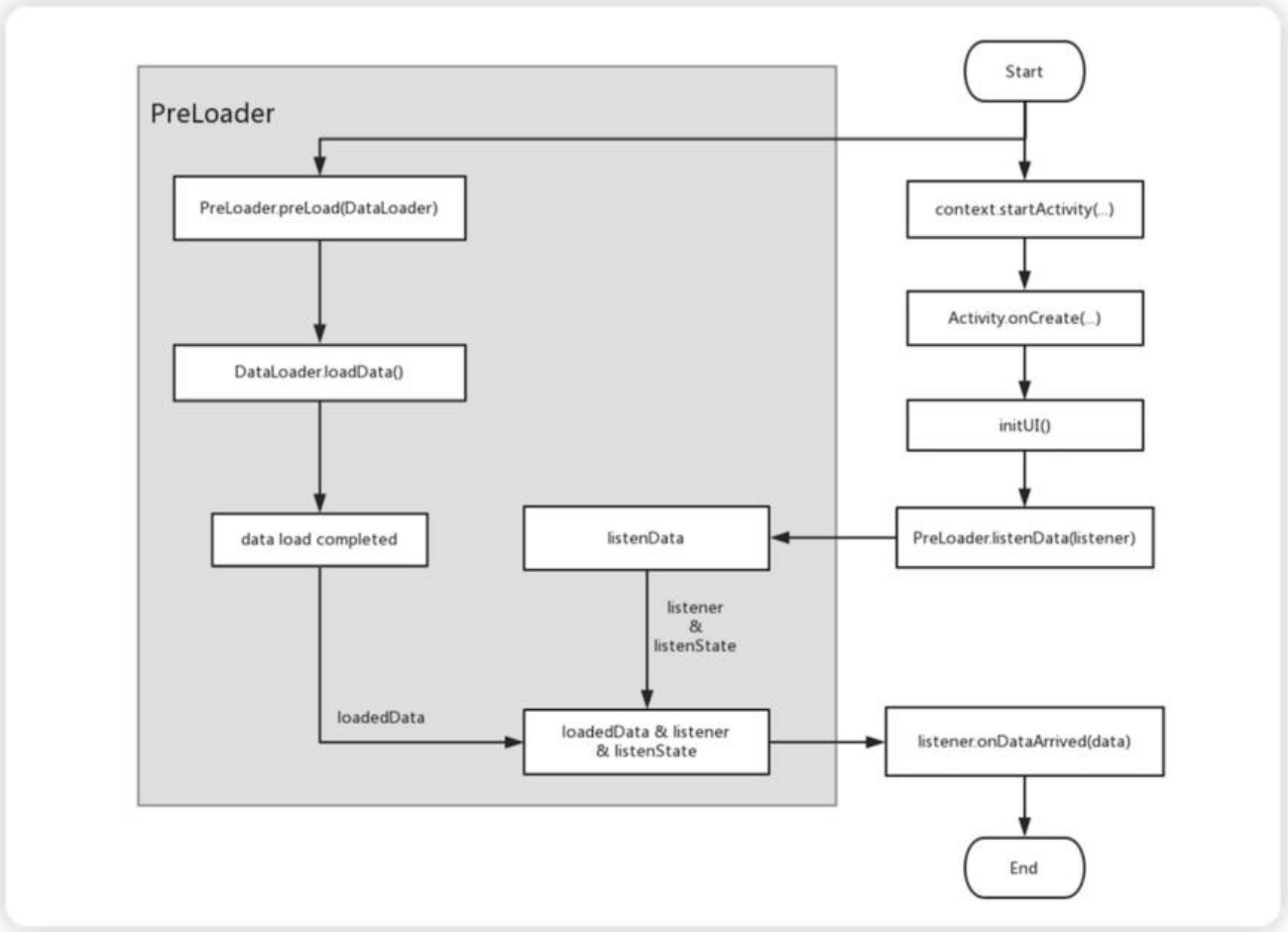
- 降低视图层级。减少冗余或者嵌套布局，防止页面过度渲染。合理使用 merge标签和约束布局 ConstraintLayout。
- 不必要的布局延迟加载。用 ViewStub替代在启动过程中不需要显示的 UI控件。
- 首页懒加载。首页不需要立即显示的页面，可以使用懒加载。
- 使用自定义 View替代复杂的 View叠加。

3.5.2 页面数据预加载

一般来说，我们喜欢在每个页面内部才开始加载和显示数据，因为这样写可能更容易让人看懂，利于后的维护。但是如果等页面UI布局初始化完毕后，我们才去加载数据的话，势必会增加页面启动显示时间。

因为每个页面（Activity）的启动本身就是比较耗时的过程，我们可以将需要显示的数据进行预加载即页面启动和数据加载同时进行，串行->并行），这样等页面UI布局初始化完毕后，我们就可以拿着加载的数据直接渲染显示了，这样可以减少数据加载的等待，从而达到加快页面显示的目的。

这里我们可以参考开源预加载库：PreLoader



3.6 系统调度优化

● 1. **启动阶段不启动子进程**，只在主进程执行Application的 onCreate方法。因为子进程会共享CP的资源，导致主进程CPU紧张。

● 2. **启动过程中减少系统调用**，避免与 AMS、WMS竞争锁。因为 AMS和 WMS在应用启动的过程承担了很多工作，且这些方法很多都是带锁的，这时应用应当避免与它们进行通信，避免出现大量的等待，阻塞关键操作。

● 3. **启动过程中除了 Activity之外的组件启动要谨慎。**因为四大组件的启动都是通过主线程Handle进行驱动的，如果在应用启动的同时他们也启动，Handler中的Message势必会增加，从而影响应用启动速度。

● 4. **启动过程中减少主线程Handler的使用。**原因同上面一样，Activity的启动都是由主线程Handle进行驱动的，应用启动期间减少主线程Handler的使用，可以减小对主页面启动的影响。对于那些量且频繁的任务调度，可以使用 HandlerThread中的 Looper创建属于子线程的handler来代替。

● 5. **使用IdleHandler。**利用 IdleHandler特性，在消息队列空闲时，对延迟任务进行分批初始化。

3.7 GC 优化

启动过程中应当减少 GC的次数。因为GC会暂停程序的执行，从而会带来延迟的代价。那么我们应当何避免频繁的GC呢?

● 1.避免进行大量的字符串操作，特别是序列化和反序列化。不要使用+(加号)进行字符串拼接。

● 2.避免临时对象的频繁创建，频繁创建的对象需要考虑复用。

原文链接：浅谈 App 的启动优化

- 3.避免大量bitmap的绘制。
- 4.避免在自定义View的 onMeasure、onLayout和 onDraw中创建对象。

3.8 Webview启动优化

如果你的应用使用到了Webview，可以按需对Webview进行优化。

- 1.由于WebView首次创建比较耗时，需要预先创建WebView，提前将其内核初始化。
- 2.使用WebView缓存池，用到WebView的时候都从缓存池中拿。
- 3.应用内置本地离线包，如一些字体和js脚本，即预置静态页面资源。

3.9 应用瘦身

对应用瘦身，可以最直接地加快资源加载的速度，从而提高应用启动的效率。

- 1.删除没有引用的资源。我们可以使用Inspect Code或者开启资源压缩，自动删除无用的资源。
- 2.能用xml写Drawable的，就不要使用UI切图。
- 3.重用资源。同一图像的着色不同，我们可以用android:tint和tintMode属性调整使用。
- 4.压缩png和jpeg文件。这里推荐一个非常好用的图片压缩插件： img-optimizer
- 5.使用webp文件格式。Android Studio可以直接将现有的bmp，jpg，png或静态gif图像转换为webp格式。
- 6.使用矢量图形，尤其是那些与分辨率无关，且可伸缩的小图标尽可能使用矢量图形。
- 7.开启代码混淆。使用proGuard代码混淆器工具，包括压缩、优化、混淆等功能。
- 8.对于一些独立也非必须的功能模块，采用插件化，按需加载。

3.10 资源重排

利用Linux的IO读取策略，PageCache和ReadAhead机制，按照读取顺序重新排列，减少磁盘IO次。具体可参见《支付宝 App 构建优化解析：通过安装包重排布优化 Android 端启动性能》 这篇文章这种技术门槛较高，一般应用都不会用到。

3.11 类重排

类重排的实现通过ReDex的Interdex调整类在Dex中的排列顺序。调整Dex中类的顺序，把启动时需加载的类按顺序放到主dex里。具体实现可以参考《Redex初探与Interdex：Andorid冷启动优化》篇文章。

4. 如何进行优化

上面讲了那么多应用启动优化的策略和措施，可能有些人就会问了：那么具体到我们每个不同的项目，我们应该如何进行优化呢?

以下是我个人的优化步骤，仅供参考：

- 1. **明确优化的内容和目标。**首先，做任何优化一定是需要带着问题（目的）去优化的。任何不带的进行的优化都是耍流氓。

● 2. **分析现状、确认问题。**当我们目前需要优化的内容后，接下来就是需要进行大量的埋点统计、较与分析，确认到底是因为什么原因导致的应用启动过慢，找到需要优化的部位。

● 3. **进行针对性的优化。**找到导致应用启动过慢的问题之后，就是按照本篇讲述的优化策略和措施进行针对性的优化。

● 4. **对优化结果进行总结并进行持续跟进。**对优化前后的数据进行统计和比较，总结优化的经验并组内进行分享，并在后续的版本中进行持续跟进。有条件的可以结合CI，增加线上的启动性能监控。

最后

讲了这么多，还是希望大家在平时开发的过程中，多重视一些应用启动优化的相关技巧，这样等别人你优化应用启动的时候，也就不会那么手足无措了。

我是xuexiangjys，一枚热爱学习，爱好编程，勤于思考，致力于Android架构研究以及开源项目经分享的技术up主。获取更多资讯，欢迎微信搜索公众号：**【我的Android开源之旅】**