|  |
| --- |
| 1、思路：首先了解冷启动的流程；  找到可控的阶段，即优化的方向；  通过工具分析优化优先级、瓶颈；  相对应的策略。    2、应用启动的类型总共分为如下三种：冷启动、热启动、温启动。  （1）冷启动：从点击应用图标到UI界面完全显示且用户可操作的全部过程。  特点：耗时最多，衡量标准。  启动流程：  Click Event -> IPC -> Process.start -> ActivityThread -> bindApplication -> LifeCycle -> ViewRootImpl  注释：先是进程创建 ——> ActivityThread线程创建（Looper、Handler） ——> 创建Application ——> Activity 生命周期——> ViewRootImpl绘制页面  （2）热启动  直接从后台切换到前台。  onResume->  特点：启动速度最快。  （3）温启动  只会重走Activity的生命周期，而不会重走进程的创建、Application的创建与生命周期等。  特点：较快，介于冷启动和热启动之间的一个速度。  启动流程：onCreate->onStart->onResume->ViewRootImpl  3、分析：优化方向是 Application和Activity的生命周期 这个阶段，因为这个阶段的时机对于我们来说是可控的，还有预览窗口。  4、启动耗时检测  最好的检测组合：“Systrace + 函数插桩”  （1）“函数插桩”：通过耗时统计工具，在需要的地方进行打点获取系统时间，就可以获取某段时间的耗时。  特点：精确，可带到线上，但是代码有侵入性，修改成本高。  （2）切面编程，利用AOP(Aspect Oriented Programming)打点  面向切面编程，通过预编译和运行期动态代理实现程序功能统一维护的一种技术，AspectJx 库。  特点：无侵入性；修改方便，建议使用。  （3）启动速度分析工具 — TraceView  使用方式：代码中添加：Debug.startMethodTracing()、检测方法、Debug.stopMethodTracing()。（需要使用adb pull将生成的.trace文件导出到电脑，然后使用Android Studio的Profiler进行加载）。  查看时间轴分布，找到耗时的瓶颈。  （4）启动速度分析工具 — Systrace  使用方式：代码插桩。首先，我们可以定义一个Trace静态工厂类，将Trace.begainSection()，Trace.endSection()封装成i、o方法，然后再在想要分析的方法前后进行插桩即可。  特点：结合Android内核的数据，生成Html报告。  系统版本越高，Android Framework中添加的系统可用Label就越多，能够支持和分析的系统模块也就越多。  必须手动缩小范围，会帮助你加速收敛问题的分析过程，进而快速地定位和解决问题。  5、评估的指标：使用什么指标来衡量启动速度的快慢？  平均启动时间的问题，一些体验很差的用户很可能被平均了。  建议的指标：  1）快开慢开比  如2s快开比，5s慢开比，可以看到有多少比例的用户体验好，多少比例的用户比较糟糕。  2）90%用户的启动时间  如果90%用户的启动时间都小于5s，那么90%区间的启动耗时就是5s。  启动的类型有哪几种？  首次安装启动  覆盖安装启动  冷启动（指标）  热启动（反映程序的活跃或保活能力）  6、主题切换：预览窗口优化  使用Activity的windowBackground主题属性预先设置一个启动图片（layer-list实现），在启动后，在Activity的onCreate()方法中的super.onCreate()前再setTheme(R.style.AppTheme)。  7、第三方库懒加载  按需初始化，特别是针对于一些应用启动时不需要初始化的库，可以等到用时才进行加载。  8、线程调度原理  1）任意时刻，只有一个线程占用CPU，处于运行状态。  2）多线程并发，轮流获取CPU使用权。  3）JVM负责线程调度，按照特定机制分配CPU使用权。  如何干预线程调度？设置线程优先级。  9、Android异步方式  1、Thread  最简单、常见的异步方式。  不易复用，频繁创建及销毁开销大。  复杂场景不易使用。  2、HandlerThread  自带消息循环的线程。  串行执行。  长时间运行，不断从队列中获取任务。  3、IntentService  继承自Service在内部创建HandlerThread。  异步，不占用主线程。  优先级较高，不易被系统Kill。  4、AsyncTask  Android提供的工具类。  无需自己处理线程切换。  需注意版本不一致问题（API 14以上解决）  5、线程池  Java提供的线程池。  易复用，减少频繁创建、销毁的时间。  功能强大，如定时、任务队列、并发数控制等。  6、RxJava  由强大的调度器Scheduler集合提供。  不同类型的Scheduler：  IO  Computation  7、异步方式总结  推荐度：从后往前排列。  正确场景选择正确的方式。  10、Android线程优化实战  线程使用准则：  1、严禁使用new Thread方式。  2、提供基础线程池供各个业务线使用，避免各个业务线各自维护一套线程池，导致线程数过多。  3、根据任务类型选择合适的异步方式：优先级低，长时间执行，HandlerThread；定时执行耗时任务，线程池。  4、创建线程必须命名，以方便定位线程归属，在运行期 Thread.currentThread().setName 修改名字。  5、关键异步任务监控，注意异步不等于不耗时，建议使用AOP的方式来做监控。  6、重视优先级设置（根据任务具体情况），Process.setThreadPriority() 可以设置多次。  11、锁定线程创建背景  项目变大之后收敛线程。项目源码、三方库、aar中都有线程的创建。  线程收敛，就是对所有的线程进行管理。不管是自己业务线的，还是引入第三方库的。  12、异步初始化  子线程分担主线程任务，并行减少时间。  1）耗时任务异化化。但是要注意，子线程不能阻塞主线程。  2）线程池管理线程，控制线程的数量。线程数量太多会相互竞争 CPU 资源，导致分给主线程的时间片减少，从而导致启动速度变慢。  3）避免主线程与子线程之间的锁阻塞等待，特别是防止主线程出现长时间的空转。  4）设置子线程优先级。不重要任务，设置子线程优先级为 THREAD\_PRIORITY\_BACKGROUND，这样子线程最多能获取到10%的时间片，优先保证主线程执行。  注释：麻烦，有些三方库之间可能出现相互依赖的。  13、延迟初始化  14、类预加载优化  在Application中提前异步加载初始化耗时较长的类。  如何找到耗时较长的类？替换系统的ClassLoader，打印类加载的时间，按需选取需要异步加载的类。  注意：  Class.forName()只加载类本身及其静态变量的引用类。  new 类实例 可以额外加载类成员变量的引用类。  15、WebView启动优化  1、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，注意内存泄漏问题。  3、本地离线包，即预置静态页面资源。  16、页面数据预加载  在主页空闲时，将其它页面的数据加载好保存到内存或数据库，等到打开该页面时，判断已经预加载过，就直接从内存或数据库取数据并显示。  17、启动阶段不启动子进程  子进程会共享CPU资源，导致主进程CPU紧张。此外，在多进程情况下一定要可以在onCreate中去区分进程做一些初始化工作。  注意启动顺序：App onCreate之前是ContentProvider初始化。  18、闪屏页与主页的布局优化  1、布局优化。  2、过渡绘制优化。  布局越复杂，测量布局绘制的时间就越长。主要做到以下几点：  布局的层级越少，加载速度越快。  一个控件的属性越少，解析越快，删除控件中的无用属性。  使用<ViewStub/>标签加载一些不常用的布局，做到使用时在加载。  使用<merge/>标签减少布局的嵌套层次。  尽可能少用wrap\_content，wrap\_content会增加布局measure时的计算成本，已知宽高为固定值时，不用wrap\_content。  19、启动优化黑科技  1）启动阶段抑制GC  启动时CG抑制，允许堆一直增长，直到手动或OOM停止GC抑制。（空间换时间）  缺点：需要白名单覆盖所有设备，但维护成本高。  2）保活啦，不被杀死，就容易启动了呗。 |

问题：（1）View的绘制从ActivityThread类中Handler的处理RESUME\_ACTIVITY事件开始。是不是就是view的绘制在loop()循环内完成

1. activity 的各个生命周期是在哪完成的，也loop()吗
2. Application对象是被谁创建的

问题：线程池的创建、优化、收敛实践

线程的优先级

不同的android异步方式选择

异步启动器Demo，看懂它的原理和实现。

延迟启动器Demo

问题：application的启动流程

# 1.启动优化的意义

启动速度是用户对我们App的第一体验，打开应用后才能去使用其中提供的强大功能，就算我们应用的内部界面设计的再精美，功能再强大，如果启动速度过慢，用户第一印象就会很差。

其实，应用的安装打开速度也是一个第一体验。

**思路：首先知道冷启动的流程**

**找到可控的阶段，即优化的方向**

**通过工具分析优化优先级、瓶颈**

**相对应的策略**

# 2、应用启动流程

## 2.1应用启动的类型

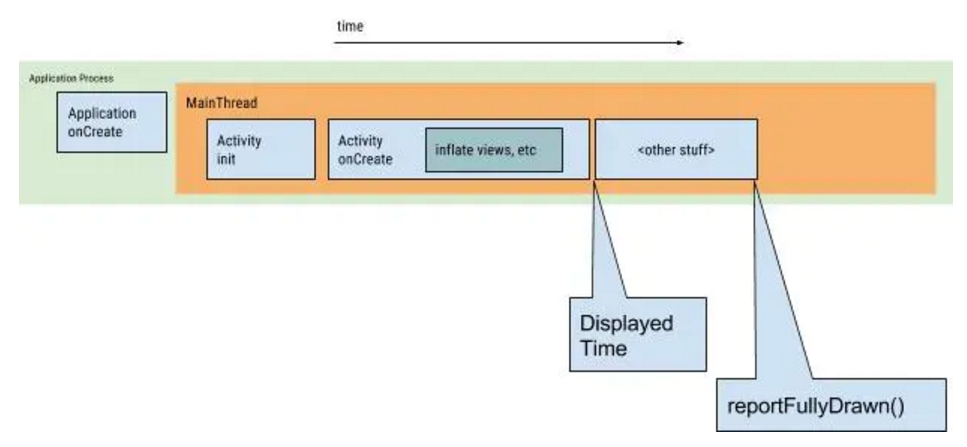
应用启动的类型总共分为如下三种：

冷启动

热启动

温启动

### 2.1.1冷启动



从点击应用图标到UI界面完全显示且用户可操作的全部过程。

特点：耗时最多，衡量标准。

启动流程：

Click Event -> IPC -> Process.start -> ActivityThread -> bindApplication -> LifeCycle -> ViewRootImpl

首先，用户进行了一个点击操作，这个点击事件它会触发一个IPC的操作，之后便会执行到Process的start方法中，这个方法是用于进程创建的，接着，便会执行到ActivityThread的main方法，这个方法可以看做是我们单个App进程的入口，相当于Java进程的main方法，在其中会执行消息循环的创建与主线程Handler的创建，创建完成之后，就会执行到 bindApplication 方法，在这里使用了反射去创建 Application以及调用了 Application相关的生命周期，Application结束之后，便会执行Activity的生命周期，在Activity生命周期结束之后，最后，就会执行到 ViewRootImpl，这时才会进行真正的一个页面的绘制。

注释：先是进程创建 ——> ActivityThread线程创建（Looper、Handler） ——> 创建Application

——> Activity 生命周期——> ViewRootImpl绘制页面

### 2.1.2热启动

直接从后台切换到前台。

onResume->

特点：启动速度最快。

### 2.1.3温启动

只会重走Activity的生命周期，而不会重走进程的创建、Application的创建与生命周期等。

特点：较快，介于冷启动和热启动之间的一个速度。

启动流程：onCreate->onStart->onResume->ViewRootImpl

ViewRootImpl是什么？

它是GUI管理系统与GUI呈现系统之间的桥梁。每一个ViewRootImpl关联一个Window， ViewRootImpl 最终会通过它的setView方法绑定Window所对应的View，并通过其performTraversals方法对View进行布局、测量和绘制。

## 2.2 冷启动分析及其优化方向

### 2.2.1 冷启动任务分析

冷启动指的是应用程序从进程在系统不存在，到系统创建应用运行进程空间的过程。冷启动通常会发生在一下两种情况：

设备启动以来首次启动应用程序

系统杀死应用程序之后再次启动应用程序

（1）冷启动之前

首先，会启动App

然后，加载空白Window

最后，创建进程

需要注意的是，这些都是系统的行为，一般情况下我们是无法直接干预的。

（2）随后任务

首先，创建Application

启动主线程（这个是在创建Application之前后吗，还是之前）

创建MainActivity

加载布局

布置屏幕

首帧绘制

通常到了界面首帧绘制完成后，我们就可以认为启动已经结束了。

### 2.2.2 优化方向

优化方向就是 Application和Activity的生命周期 这个阶段，因为这个阶段的时机对于我们来说是可控的，还有预览窗口。

# 启动耗时检测

检测组合：“Systrace + 函数插桩”

## 3.1查看Logcat（线下，AS）

在Android Studio Logcat中过滤关键字“Displayed”，可以看到对应的冷启动耗时日志。

## 3.2 adb shell（线下，AS, 不精确）

使用adb shell获取应用的启动时间

// 其中的AppstartActivity全路径可以省略前面的packageName

adb shell am start -W [packageName]/[AppstartActivity全路径]

执行后会得到三个时间：ThisTime、TotalTime和WaitTime，详情如下：

ThisTime：表示最后一个Activity启动耗时。

TotalTime：表示所有Activity启动耗时。

WaitTime：表示AMS启动Activity的总耗时。

一般来说，只需查看得到的TotalTime，即应用的启动时间，其包括 创建进程 + Application初始化 + Activity初始化到界面显示 的过程。

特点：

1、线下使用方便，不能带到线上。

2、非严谨、精确时间。

这两个方法，自是一个总时间，对优化的切入点没有帮助。

## 3.3代码打点（函数插桩）

可以写一个统计耗时的工具类来记录整个过程的耗时情况。其中需要注意的有：

在上传数据到服务器时建议根据用户ID的尾号来抽样上报。

在项目中核心基类的关键回调函数和核心方法中加入打点。

其代码如下所示：

|  |
| --- |
| /\*\*  \* 耗时监视器对象，记录整个过程的耗时情况，可以用在很多需要统计的地方，比如Activity的启动耗时和Fragment的启动耗时。  \*/  public class TimeMonitor {  private final String TAG = TimeMonitor.class.getSimpleName();  private int mMonitord = -1;    // 保存一个耗时统计模块的各种耗时，tag对应某一个阶段的时间  private HashMap<String, Long> mTimeTag = new HashMap<>();  private long mStartTime = 0;  public TimeMonitor(int mMonitorId) {  Log.d(TAG, "init TimeMonitor id: " + mMonitorId);  this.mMonitorId = mMonitorId;  }  public int getMonitorId() {  return mMonitorId;  }  public void startMonitor() {  // 每次重新启动都把前面的数据清除，避免统计错误的数据  if (mTimeTag.size() > 0) {  mTimeTag.clear();  }  mStartTime = System.currentTimeMillis();  }  /\*\*  \* 每打一次点，记录某个tag的耗时  \*/  public void recordingTimeTag(String tag) {  // 若保存过相同的tag，先清除  if (mTimeTag.get(tag) != null) {  mTimeTag.remove(tag);  }  long time = System.currentTimeMillis() - mStartTime;  Log.d(TAG, tag + ": " + time);  mTimeTag.put(tag, time);  }  public void end(String tag, boolean writeLog) {  recordingTimeTag(tag);  end(writeLog);  }  public void end(boolean writeLog) {  if (writeLog) {  //写入到本地文件  }  }  public HashMap<String, Long> getTimeTags() {  return mTimeTag;  }  } |

为了使代码更好管理，我们需要定义一个打点配置类，如下所示：

|  |
| --- |
| /\*\*  \* 打点配置类，用于统计各阶段的耗时，便于代码的维护和管理。  \*/  public final class TimeMonitorConfig {  // 应用启动耗时  public static final int TIME\_MONITOR\_ID\_APPLICATION\_START = 1;  } |

此外，耗时统计可能会在多个模块和类中需要打点，所以需要一个单例类来管理各个耗时统计的数据：

|  |
| --- |
| /\*\*  \* 采用单例管理各个耗时统计的数据。  \*/  public class TimeMonitorManager {  private static TimeMonitorManager mTimeMonitorManager = null;  private HashMap<Integer, TimeMonitor> mTimeMonitorMap = null;  public synchronized static TimeMonitorManager getInstance() {  if (mTimeMonitorManager == null) {  mTimeMonitorManager = new TimeMonitorManager();  }  return mTimeMonitorManager;  }  public TimeMonitorManager() {  this.mTimeMonitorMap = new HashMap<Integer, TimeMonitor>();  }  /\*\*  \* 初始化打点模块  \*/  public void resetTimeMonitor(int id) {  if (mTimeMonitorMap.get(id) != null) {  mTimeMonitorMap.remove(id);  }  getTimeMonitor(id).startMonitor();  }  /\*\*  \* 获取打点器  \*/  public TimeMonitor getTimeMonitor(int id) {  TimeMonitor monitor = mTimeMonitorMap.get(id);  if (monitor == null) {  monitor = new TimeMonitor(id);  mTimeMonitorMap.put(id, monitor);  }  return monitor;  }  } |

主要在以下几个方面需要打点：

应用程序的生命周期节点。

启动时需要初始化的重要方法，例如数据库初始化，读取本地的一些数据。

其他耗时的一些算法。

例如，启动时在Application和第一个Activity加入打点统计：

Application 打点

|  |
| --- |
| @Override  protected void attachBaseContext(Context base) {  super.attachBaseContext(base);  TimeMonitorManager.getInstance().resetTimeMonitor(TimeMonitorConfig.TIME\_MONITOR\_ID\_APPLICATION\_START);  }  @Override  public void onCreate() {  super.onCreate();  SoLoader.init(this, /\* native exopackage \*/ false);  TimeMonitorManager.getInstance().getTimeMonitor(TimeMonitorConfig.TIME\_MONITOR\_ID\_APPLICATION\_START).recordingTimeTag("Application-onCreate");  } |

第一个Activity打点

|  |
| --- |
| @Override  protected void onCreate(Bundle savedInstanceState) {  TimeMonitorManager.getInstance().getTimeMonitor(TimeMonitorConfig.TIME\_MONITOR\_ID\_APPLICATION\_START).recordingTimeTag("SplashActivity-onCreate");  super.onCreate(savedInstanceState);    initData();    TimeMonitorManager.getInstance().getTimeMonitor(TimeMonitorConfig.TIME\_MONITOR\_ID\_APPLICATION\_START).recordingTimeTag("SplashActivity-onCreate-Over");  }  @Override  protected void onStart() {  super.onStart();  TimeMonitorManager.getInstance().getTimeMonitor(TimeMonitorConfig.TIME\_MONITOR\_ID\_APPLICATION\_START).end("SplashActivity-onStart", false);  } |

特点：精确，可带到线上，但是代码有侵入性，修改成本高。

注意事项

1、在上传数据到服务器时建议根据用户ID的尾号来抽样上报。

2、onWindowFocusChanged只是首帧时间，App启动完成的结束点应该是真实数据展示出来的时候（通常来说都是首帧数据），如列表第一条数据展示，记得使用getViewTreeObserver().addOnPreDrawListener()（在API 16以上可以使用addOnDrawListener），它会把任务延迟到列表显示后再执行，例如，在Awesome-WanAndroid项目的主页就有一个RecyclerView实现的列表，启动结束的时间就是列表的首帧时间，也即列表第一条数据展示的时候。这里，我们直接在RecyclerView的适配器ArticleListAdapter的convert（onBindViewHolder）方法中加上如下代码即可：

if (helper.getLayoutPosition() == 1 && !mHasRecorded) {

mHasRecorded = true;

helper.getView(R.id.item\_search\_pager\_group).getViewTreeObserver().addOnPreDrawListener(new ViewTreeObserver.OnPreDrawListener() {

@Override

public boolean onPreDraw() {

helper.getView(R.id.item\_search\_pager\_group).getViewTreeObserver().removeOnPreDrawListener(this);

LogHelper.i("FeedShow");

return true;

}

});

}

为什么不使用onWindowFocusChanged这个方法作为启动结束点？

因为用户看到真实的界面是需要有网络请求返回真实数据的，但是onWindowFocusChanged只是界面绘制的首帧时机，但是列表中的数据是需要从网络中下载得到的，所以应该以列表的首帧数据作为启动结束点。

## 3.4 AOP(Aspect Oriented Programming) 打点

面向切面编程，通过预编译和运行期动态代理实现程序功能统一维护的一种技术。

1、作用

利用AOP可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的耦合性降低，提高程序的可重用性，同时大大提高了开发效率。

2、AOP核心概念

1、横切关注点

对哪些方法进行拦截，拦截后怎么处理。

2、切面（Aspect）

类是对物体特征的抽象，切面就是对横切关注点的抽象。

3、连接点（JoinPoint）

被拦截到的点（方法、字段、构造器）。

4、切入点（PointCut）

对JoinPoint进行拦截的定义。

5、通知（Advice）

拦截到JoinPoint后要执行的代码，分为前置、后置、环绕三种类型。

3、准备：接入AspectJx进行切面编码

首先，为了在Android使用AOP埋点需要引入AspectJ，在项目根目录的build.gradle下加入：

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

然后，在app目录下的build.gradle下加入：

apply plugin: 'android-aspectjx'

implement 'org.aspectj:aspectjrt:1.8.+'

4、AOP埋点实战

JoinPoint一般定位在如下位置

1、函数调用

2、获取、设置变量

3、类初始化

使用PointCut对我们指定的连接点进行拦截，通过Advice，就可以拦截到JoinPoint后要执行的代码。Advice通常有以下几种类型：

1、Before：PointCut之前执行

2、After：PointCut之后执行

3、Around：PointCut之前、之后分别执行

首先，我们举一个小栗子：

@Before("execution(\* android.app.Activity.on\*\*(..))")

public void onActivityCalled(JoinPoint joinPoint) throws Throwable {

...

}

在 execution 中的是一个匹配规则，第一个 \* 代表匹配任意的方法返回值，后面的语法代码匹配所有Activity中on开头的方法。

其中execution是处理Join Point的类型，在AspectJx中共有两种类型，如下所示：

1、call：插入在函数体里面

2、execution：插入在函数体外面

如何统计Application中的所有方法耗时？

@Aspect

public class ApplicationAop {

@Around("call (\* com.json.chao.application.BaseApplication.\*\*(..))")

public void getTime(ProceedingJoinPoint joinPoint) {

Signature signature = joinPoint.getSignature();

String name = signature.toShortString();

long time = System.currentTimeMillis();

try {

joinPoint.proceed();

} catch (Throwable throwable) {

throwable.printStackTrace();

}

Log.i(TAG, name + " cost" + (System.currentTimeMillis() - time));

}

}

在上述代码中，我们需要注意 不同的Action类型其对应的方法入参是不同的，具体的差异如下所示：

当Action为Before、After时，方法入参为JoinPoint。

当Action为Around时，方法入参为ProceedingPoint。

Around和Before、After的最大区别:

ProceedingPoint不同于JoinPoint，其提供了proceed方法执行目标方法。

5、总结AOP特性

1、无侵入性

2、修改方便，建议使用

# 启动速度分析工具 — TraceView

## 4.1使用方式

1、代码中添加：Debug.startMethodTracing()、检测方法、Debug.stopMethodTracing()。（需要使用adb pull将生成的.trace文件导出到电脑，然后使用Android Studio的Profiler进行加载）

2、打开 Profiler -> CPU -> 点击 Record -> 点击 Stop -> 查看Profiler下方Top Down/Bottom Up 区域，以找出耗时的热点方法。

## 4.2 Profile CPU

使用 Profile 的 CPU 模块可以帮我们快速找到耗时的热点方法，下面，我们来详细来分析一下这个模块。

1、Trace types

Trace types 有四种，如下所示。

1、Trace Java Methods

会记录每个方法的时间、CPU信息。对运行时性能影响较大。

2、Sample Java Methods

相比于Trace Java Methods会记录每个方法的时间、CPU信息，它会在应用的Java代码执行期间频繁捕获应用的调用堆栈，对运行时性能的影响比较小，能够记录更大的数据区域。

3、Sample C/C++ Functions

需部署到Android 8.0及以上设备，内部使用simpleperf跟踪应用的native代码，也可以命令行使用simpleperf。

4、Trace System Calls

检查应用与系统资源的交互情况。

查看所有核心的CPU瓶颈。

内部采用systrace，也可以使用systrace命令。

2、Event timeline

用于显示应用程序在其生命周期中转换不同状态的活动，如用户交互、屏幕旋转事件等。

3、CPU timeline

用于显示应用程序 实时CPU使用率、其它进程实时CPU使用率、应用程序使用的线程总数。

4、Thread activity timeline

列出应用程序进程中的每个线程，并使用了不同的颜色在其时间轴上指示其活动。

绿色：线程处于活动状态或准备好使用CPU。

黄色：线程正等待IO操作。（重要）

灰色：线程正在睡眠，不消耗CPU时间。

5、检查跟踪数据窗口

Profile提供的检查跟踪数据窗口有四种，如下所示：

1、Call Chart

提供函数跟踪数据的图形表示形式。

水平轴：表示调用的时间段和时间。

垂直轴：显示被调用方。

橙色：系统API。

绿色：应用自有方法。

蓝色：第三方API（包括Java API）。

提示

右键点击 Jump to source 跳转至指定函数。

2、Flame Chart

将具有相同调用方顺序的完全相同的方法收集起来。

水平轴：执行每个方法的相对时间量。

垂直轴：显示被调用方。

使用技巧

看顶层的哪个函数占据的宽度最大（表现为平顶），可能存在性能问题。

3、Top Down

递归调用列表，提供self、children、total时间和比率来表示被调用的函数信息。

Flame Chart是Top Down列表数据的图形化。

4、Bottom Up

展开函数会显示其调用方。

按照消耗CPU时间由多到少的顺序对函数排序。

注意事项

我们在查看上面4个跟踪数据的区域时，应该注意右侧的两个时间，如下所示：

Wall Clock Time：程序执行时间。

Thread Time：CPU执行的时间。

## 4.3 TraceView小结

特点：

1、图形的形式展示执行时间、调用栈等。

2、信息全面，包含所有线程。

3、运行时开销严重，整体都会变慢，得出的结果并不真实。

4、找到最耗费时间的路径：Flame Chart、Top Down。

5、找到最耗费时间的节点：Bottom Up。

作用：主要做热点分析，用来得到以下两种数据：

单次执行最耗时的方法。

执行次数最多的方法。

# 5.启动速度分析工具 — Systrace

## 5.1使用方式：代码插桩

首先，我们可以定义一个Trace静态工厂类，将Trace.begainSection()，Trace.endSection()封装成i、o方法，然后再在想要分析的方法前后进行插桩即可。

然后，在命令行下执行systrace.py脚本，命令如下所示：

python /Users/quchao/Library/Android/sdk/platform-tools/systrace/systrace.py -t 20 sched gfx view wm am app webview -a "com.wanandroid.json.chao" -o ~/Documents/open-project/systrace\_data/wanandroid\_start\_1.html

具体参数含义如下：

-t：指定统计时间为20s。

shced：cpu调度信息。

gfx：图形信息。

view：视图。

wm：窗口管理。

am：活动管理。

app：应用信息。

webview：webview信息。

-a：指定目标应用程序的包名。

-o：生成的systrace.html文件。

如何查看数据？

在UIThread一栏可以看到核心的系统方法时间区域和我们自己使用代码插桩捕获的方法时间区域。

## 5.2 Systrace原理

首先，在系统的一些关键链路（如SystemServcie、虚拟机、Binder驱动）插入一些信息（Label）。

然后，通过Label的开始和结束来确定某个核心过程的执行时间，并把这些Label信息收集起来得到系统关键路径的运行时间信息，最后得到整个系统的运行性能信息;

其中，Android Framework 里面一些重要的模块都插入了label信息，用户App中也可以添加自定义的Lable。

## 5.3 Systrace小结

特点：

结合Android内核的数据，生成Html报告。

系统版本越高，Android Framework中添加的系统可用Label就越多，能够支持和分析的系统模块也就越多。

必须手动缩小范围，会帮助你加速收敛问题的分析过程，进而快速地定位和解决问题。

作用：

主要用于分析绘制性能方面的问题。

分析系统关键方法和应用方法耗时。

# 6.启动监控

## 6.1实验室监控：视频录制

80%绘制

图像识别

覆盖高中低端机型不同的场景。

## 6.2线上监控

目标

需要准确地统计启动耗时。

1、启动结束的统计时机

是否是使用界面显示且用户真正可以操作的时间作为启动结束时间。

2、启动时间扣除逻辑

闪屏、广告和新手引导这些时间都应该从启动时间里扣除。

3、启动排除逻辑

Broadcast、Server拉起，启动过程进入后台都需要排除统计。

4、使用什么指标来衡量启动速度的快慢？

平均启动时间的问题

一些体验很差的用户很可能被平均了。

建议的指标：

1、快开慢开比

如2s快开比，5s慢开比，可以看到有多少比例的用户体验好，多少比例的用户比较糟糕。

2、90%用户的启动时间

如果90%用户的启动时间都小于5s，那么90%区间的启动耗时就是5s。

5、启动的类型有哪几种？

首次安装启动

覆盖安装启动

冷启动（指标）

热启动（反映程序的活跃或保活能力）

# 7.启动优化常规方案

启动过程中的常见问题：

1、点击图标很久都不响应：预览窗口被禁用或设置为透明。

2、首页显示太慢：初始化任务太多。

3、首页显示后无法进行操作：太多延迟初始化任务占用主线程CPU时间片。

优化区域：

Application、Activity创建以及回调等过程。

## 7.1主题切换：预览窗口优化

使用Activity的windowBackground主题属性预先设置一个启动图片（layer-list实现），在启动后，在Activity的onCreate()方法中的super.onCreate()前再setTheme(R.style.AppTheme)。

优点：

使用简单。

避免了启动白屏和点击启动图标不响应的情况。

缺点：

治标不治本，表面上产生一种快的感觉。

对于中低端机，总的闪屏时间会更长，建议只在Android6.0/7.0以上才启用“预览闪屏”方案，让手机性能好的用户可以有更好的体验。

## 7.2 第三方库懒加载

按需初始化，特别是针对于一些应用启动时不需要初始化的库，可以等到用时才进行加载。

## 7.3异步初始化预备知识-线程优化

### 7.3.1 Android线程调度原理剖析

线程调度原理

1、任意时刻，只有一个线程占用CPU，处于运行状态。

2、多线程并发，轮流获取CPU使用权。

3、JVM负责线程调度，按照特定机制分配CPU使用权。

线程调度模型

1、分时调度模型

轮流获取、均分CPU。

2、抢占式调度模型

优先级高的获取。

如何干预线程调度？

设置线程优先级。

Android线程调度

1、nice值

Process中定义。

值越小，优先级越高。

默认是THREAD\_PRIORITY\_DEFAUT，0。

2、cgroup

它是一种更严格的群组调度策略，主要分为如下两种类型：

后台group（默认）。

前台group，保证前台线程可以获取到更多的CPU

注意点

线程过多会导致CPU频繁切换，降低线程运行效率。

正确认识任务重要性以决定使用哪种线程优先级。

优先级具有继承性。

### 7.3.2 Android异步方式

1、Thread

最简单、常见的异步方式。

不易复用，频繁创建及销毁开销大。

复杂场景不易使用。

2、HandlerThread

自带消息循环的线程。

串行执行。

长时间运行，不断从队列中获取任务。

3、IntentService

继承自Service在内部创建HandlerThread。

异步，不占用主线程。

优先级较高，不易被系统Kill。

4、AsyncTask

Android提供的工具类。

无需自己处理线程切换。

需注意版本不一致问题（API 14以上解决）

5、线程池

Java提供的线程池。

易复用，减少频繁创建、销毁的时间。

功能强大，如定时、任务队列、并发数控制等。

6、RxJava

由强大的调度器Scheduler集合提供。

不同类型的Scheduler：

IO

Computation

7、异步方式总结

推荐度：从后往前排列。

正确场景选择正确的方式。

### 7.3.3 Android线程优化实战

线程使用准则：

1、严禁使用new Thread方式。

2、提供基础线程池供各个业务线使用，避免各个业务线各自维护一套线程池，导致线程数过多。

3、根据任务类型选择合适的异步方式：优先级低，长时间执行，HandlerThread；定时执行耗时任务，线程池。

4、创建线程必须命名，以方便定位线程归属，在运行期 Thread.currentThread().setName 修改名字。

5、关键异步任务监控，注意异步不等于不耗时，建议使用AOP的方式来做监控。

6、重视优先级设置（根据任务具体情况），Process.setThreadPriority() 可以设置多次。

4、如何锁定线程创建者

### 7.3.4 锁定线程创建背景

项目变大之后收敛线程。

项目源码、三方库、aar中都有线程的创建。

锁定线程创建方案

特别适合Hook手段，找Hook点：构造函数或者特定方法，如Thread的构造函数。

### 7.3.5 线程收敛优雅实践初步

线程收敛常规方案

根据线程创建堆栈考量合理性，使用同一线程库。

各业务线下掉自己的线程库。

问题：基础库怎么使用线程？

直接依赖线程库，但问题在于线程库更新可能会导致基础库更新。

基础库优雅使用线程

基础库内部暴露API：setExecutor。

初始化的时候注入统一的线程库。

统一线程库时区分任务类型

IO密集型任务：IO密集型任务不消耗CPU，核心池可以很大。常见的IO密集型任务如文件读取、写入，网络请求等等。

CPU密集型任务：核心池大小和CPU核心数相关。常见的CPU密集型任务如比较复杂的计算操作，此时需要使用大量的CPU计算单元。

实现用于执行多类型任务的基础线程池组件

目前基础线程池组件位于启动器sdk之中，使用非常简单，示例代码如下所示：

// 如果当前执行的任务是CPU密集型任务，则从基础线程池组件

// DispatcherExecutor中获取到用于执行 CPU 密集型任务的线程池

DispatcherExecutor.getCPUExecutor().execute(YourRunable());

// 如果当前执行的任务是IO密集型任务，则从基础线程池组件

// DispatcherExecutor中获取到用于执行 IO 密集型任务的线程池

DispatcherExecutor.getIOExecutor().execute(YourRunable());

具体的实现源码也比较简单，并且我对每一处代码都进行了详细的解释，就不一一具体分析了。代码如下所示：

public class DispatcherExecutor {

/\*\*

\* CPU 密集型任务的线程池

\*/

private static ThreadPoolExecutor sCPUThreadPoolExecutor;

/\*\*

\* IO 密集型任务的线程池

\*/

private static ExecutorService sIOThreadPoolExecutor;

/\*\*

\* 当前设备可以使用的 CPU 核数

\*/

private static final int CPU\_COUNT = Runtime.getRuntime().availableProcessors();

/\*\*

\* 线程池核心线程数，其数量在2 ~ 5这个区域内

\*/

private static final int CORE\_POOL\_SIZE = Math.max(2, Math.min(CPU\_COUNT - 1, 5));

/\*\*

\* 线程池线程数的最大值：这里指定为了核心线程数的大小

\*/

private static final int MAXIMUM\_POOL\_SIZE = CORE\_POOL\_SIZE;

/\*\*

\* 线程池中空闲线程等待工作的超时时间，当线程池中

\* 线程数量大于corePoolSize（核心线程数量）或

\* 设置了allowCoreThreadTimeOut（是否允许空闲核心线程超时）时，

\* 线程会根据keepAliveTime的值进行活性检查，一旦超时便销毁线程。

\* 否则，线程会永远等待新的工作。

\*/

private static final int KEEP\_ALIVE\_SECONDS = 5;

/\*\*

\* 创建一个基于链表节点的阻塞队列

\*/

private static final BlockingQueue<Runnable> S\_POOL\_WORK\_QUEUE = new LinkedBlockingQueue<>();

/\*\*

\* 用于创建线程的线程工厂

\*/

private static final DefaultThreadFactory S\_THREAD\_FACTORY = new DefaultThreadFactory();

/\*\*

\* 线程池执行耗时任务时发生异常所需要做的拒绝执行处理

\* 注意：一般不会执行到这里

\*/

private static final RejectedExecutionHandler S\_HANDLER = new RejectedExecutionHandler() {

@Override

public void rejectedExecution(Runnable r, ThreadPoolExecutor executor) {

Executors.newCachedThreadPool().execute(r);

}

};

/\*\*

\* 获取CPU线程池

\*

\* @return CPU线程池

\*/

public static ThreadPoolExecutor getCPUExecutor() {

return sCPUThreadPoolExecutor;

}

/\*\*

\* 获取IO线程池

\*

\* @return IO线程池

\*/

public static ExecutorService getIOExecutor() {

return sIOThreadPoolExecutor;

}

/\*\*

\* 实现一个默认的线程工厂

\*/

private static class DefaultThreadFactory implements ThreadFactory {

private static final AtomicInteger POOL\_NUMBER = new AtomicInteger(1);

private final ThreadGroup group;

private final AtomicInteger threadNumber = new AtomicInteger(1);

private final String namePrefix;

DefaultThreadFactory() {

SecurityManager s = System.getSecurityManager();

group = (s != null) ? s.getThreadGroup() :

Thread.currentThread().getThreadGroup();

namePrefix = "TaskDispatcherPool-" +

POOL\_NUMBER.getAndIncrement() +

"-Thread-";

}

@Override

public Thread newThread(Runnable r) {

// 每一个新创建的线程都会分配到线程组group当中

Thread t = new Thread(group, r,

namePrefix + threadNumber.getAndIncrement(),

0);

if (t.isDaemon()) {

// 非守护线程

t.setDaemon(false);

}

// 设置线程优先级

if (t.getPriority() != Thread.NORM\_PRIORITY) {

t.setPriority(Thread.NORM\_PRIORITY);

}

return t;

}

}

static {

sCPUThreadPoolExecutor = new ThreadPoolExecutor(

CORE\_POOL\_SIZE, MAXIMUM\_POOL\_SIZE, KEEP\_ALIVE\_SECONDS, TimeUnit.SECONDS,

S\_POOL\_WORK\_QUEUE, S\_THREAD\_FACTORY, S\_HANDLER);

// 设置是否允许空闲核心线程超时时，线程会根据keepAliveTime的值进行活性检查，一旦超时便销毁线程。否则，线程会永远等待新的工作。

sCPUThreadPoolExecutor.allowCoreThreadTimeOut(true);

// IO密集型任务线程池直接采用CachedThreadPool来实现，

// 它最多可以分配Integer.MAX\_VALUE个非核心线程用来执行任务

sIOThreadPoolExecutor = Executors.newCachedThreadPool(S\_THREAD\_FACTORY);

}

}

### 7.3.6 线程优化核心问题

1、线程使用为什么会遇到问题？

项目发展阶段忽视基础设施建设，没有采用统一的线程池，导致线程数量过多。

表现形式：

异步任务执行太耗时，导致主线程卡顿。

问题原因：

1、Java线程调度是抢占式的，线程优先级比较重要，需要区分。

2、没有区分IO和CPU密集型任务，导致主线程抢不到CPU。

2、怎么在项目中对线程进行优化？

核心：线程收敛

通过Hook方式找到对应线程的堆栈信息，和业务方讨论是否应该单独起一个线程，尽可能使用统一线程池。

每个基础库都暴露一个设置线程池的方法，以避免线程库更新导致基础库需要更新的问题。

统一线程池应注意IO、CPU密集型任务区分。

其它细节：重要异步任务统计耗时、注重异步任务优先级和线程名的设置。

## 7.4 异步初始化

### 7.4.1核心思想

子线程分担主线程任务，并行减少时间。

线程优化以下几方面：

第一，耗时任务异步化。子线程处理耗时任务，主线程做的事情越少，越早进入Acitivity绘制阶段，界面越早展现。例如不在主线程做如 IO 、网络等耗时操作。但是要注意，子线程不能阻塞主线程。

1. 线程池管理线程，控制线程的数量。线程数量太多会相互竞争 CPU 资源，导致分给主线程的时间片减少，从而导致启动速度变慢。
2. 避免主线程与子线程之间的锁阻塞等待，特别是防止主线程出现长时间的空转。特别是现在有很多启动框架，会使用 Pipeline 机制，根据业务优先级规定业务初始化时机。比如微信内部使用的 mmkernel 、阿里最近开源的 Alpha 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。如果任务的依赖关系没有配置好，很容易出现下图这种情况，即主线程会一直等待 taskC 结束，空转 2950 毫秒。
3. 设置子线程优先级。不重要任务，设置子线程优先级为 THREAD\_PRIORITY\_BACKG

ROUND，这样子线程最多能获取到10%的时间片，优先保证主线程执行。

### 7.4.2异步优化注意点

1、不符合异步要求。

2、需要在某个阶段完成（采用CountDownLatch确保异步任务完成后才到下一个阶段）。

3、如出现主线程要使用时还没初始化则在此次使用前初始化。

4、区分CPU密集型和IO密集型任务。

### 7.4.3异步初始化方案演进

1、new Thread

2、IntentService

3、线程池（合理配置并选择CPU密集型和IO密集型线程池）

4、异步启动器

### 7.4.4异步优化最优解：异步启动器

看Demo

常规异步优化痛点：

1、代码不优雅：例如使用线程池实现多个并行异步任务时会有多个executorService.submit代码块。

2、场景不好处理：各个初始化任务之间存在依赖关系，例如推送sdk的初始化任务需要依赖于获取设备id的初始化任务。此外，有些任务是需要在某些特定的时候就初始化完成，例如需要在Application的onCreate方法执行完之前就初始化完成。

3、维护成本高。

启动器核心思想

充分利用CPU多核，自动梳理任务顺序。

启动器流程：

启动器的主题流程为上图中的中间区域，即主线程与并发两个区域块。需要注意的是，在上图中的 head task与tail task 并不包含在启动器的主题流程中，它仅仅是用于处理启动前/启动后的一些通用任务，例如我们可以在head task中做一些获取通用信息的操作，在tail task可以做一些log输出、数据上报等操作。

那么，这里我们总结一下启动的核心流程，如下所示：

1、任务Task化，启动逻辑抽象成Task（Task即对应一个个的初始化任务）。

2、根据所有任务依赖关系排序生成一个有向无环图：例如上述说到的推送SDK初始化任务需要依赖于获取设备id的初始化任务，各个任务之间都可能存在依赖关系，所以将它们的依赖关系排序生成一个有向无环图能将并行效率最大化。

3、多线程按照排序后的优先级依次执行：例如必须先初始化获取设备id的初始化任务，才能去进行推送SDK的初始化任务。

异步启动器优化实战与源码剖析

下面，我们就来使用异步启动器来在Application的onCreate方法中进行异步优化，代码如下所示：

// 1、启动器初始化

TaskDispatcher.init(this);

// 2、创建启动器实例，这里每次获取的都是新对象

TaskDispatcher dispatcher = TaskDispatcher.createInstance();

// 3、给启动器配置一系列的（异步/非异步）初始化任务并启动启动器

dispatcher

.addTask(new InitAMapTask())

.addTask(new InitStethoTask())

.addTask(new InitWeexTask())

.addTask(new InitBuglyTask())

.addTask(new InitFrescoTask())

.addTask(new InitJPushTask())

.addTask(new InitUmengTask())

.addTask(new GetDeviceIdTask())

.start();

// 4、需要等待微信SDK初始化完成，程序才能往下执行

dispatcher.await();

这里的 TaskDispatcher 就是我们的启动器调用类。首先，在注释1处，我们需要先调用TaskDispatcher的init方法进行启动器的初始化，其源码如下所示：

public static void init(Context context) {

if (context != null) {

sContext = context;

sHasInit = true;

sIsMainProcess = Utils.isMainProcess(sContext);

}

}

可以看到，仅仅是初始化了几个基础字段。接着，在注释2处，我们创建了启动器实例，其源码如下所示：

/\*\*

\* 注意：这里我们每次获取的都是新对象

\*/

public static TaskDispatcher createInstance() {

if (!sHasInit) {

throw new RuntimeException("must call TaskDispatcher.init first");

}

return new TaskDispatcher();

}

在createInstance方法的中我们每次都会创建一个新的TaskDispatcher实例。然后，在注释3处，我们给启动器配置了一系列的初始化任务并启动启动器，需要注意的是，这里的Task既可以是用于执行异步任务（子线程）的也可以是用于执行非异步任务（主线程）。下面，我们来分析下这两种Task的用法，比如InitStethoTask这个异步任务的初始化，代码如下所示：

/\*\*

\* 异步的Task

\*/

public class InitStethoTask extends Task {

@Override

public void run() {

Stetho.initializeWithDefaults(mContext);

}

}

这里的InitStethoTask直接继承自Task，Task中的runOnMainThread方法返回为false，说明 task 是用于处理异步任务的task，其中的run方法就是Runnable的run方法。下面，我们再看看另一个用于初始化非异步任务的例子，例如用于微信SDK初始化的InitWeexTask，代码如下所示：

/\*\*

\* 主线程执行的task

\*/

public class InitWeexTask extends MainTask {

@Override

public boolean needWait() {

return true;

}

@Override

public void run() {

InitConfig config = new InitConfig.Builder().build();

WXSDKEngine.initialize((Application) mContext, config);

}

}

可以看到，它直接继承了MainTask，MainTask的源码如下所示：

public abstract class MainTask extends Task {

@Override

public boolean runOnMainThread() {

return true;

}

}

MainTask 直接继承了Task，并仅仅是重写了runOnMainThread方法返回了true，说明它就是用来初始化主线程中的非异步任务的。

此外，我们注意到InitWeexTask中还重写了一个needWait方法并返回了true，其目的是为了在某个时刻之前必须等待InitWeexTask初始化完成程序才能继续往下执行，这里的某个时刻指的就是我们在Application的onCreate方法中的注释4处的代码所执行的地方：dispatcher.await()，其实现源码如下所示：

/\*\*

\* 需要等待的任务数

\*/

private AtomicInteger mNeedWaitCount = new AtomicInteger();

/\*\*

\* 调用了 await 还没结束且需要等待的任务列表

\*/

private List<Task> mNeedWaitTasks = new ArrayList<>();

private CountDownLatch mCountDownLatch;

private static final int WAITTIME = 10000;

@UiThread

public void await() {

try {

// 1、仅仅在测试阶段才输出需等待的任务列表数与任务名称

if (DispatcherLog.isDebug()) {

DispatcherLog.i("still has " + mNeedWaitCount.get());

for (Task task : mNeedWaitTasks) {

DispatcherLog.i("needWait: " + task.getClass().getSimpleName());

}

}

// 2、只要还有需要等待的任务没有执行完成，就调用mCountDownLatch的await方法进行等待，这里我们设定超时时间为10s

if (mNeedWaitCount.get() > 0) {

if (mCountDownLatch == null) {

throw new RuntimeException("You have to call start() before call await()");

}

mCountDownLatch.await(WAITTIME, TimeUnit.MILLISECONDS);

}

} catch (InterruptedException e) {

}

}

首先，在注释1处，我们仅仅只会在测试阶段才会输出需等待的任务列表数与任务名称。然后，在注释2处，只要需要等待的任务数mNeedWaitCount大于0，即只要还有需要等待的任务没有执行完成，就调用mCountDownLatch的await方法进行等待，注意我们这里设定了超时时间为10s。当一个task执行完成后，无论它是异步还是非异步的，最终都会执行到mTaskDispatcher的markTaskDone(mTask)方法，我们看看它的实现源码，如下所示：

/\*\*

\* 已经结束的Task

\*/

private volatile List<Class<? extends Task>> mFinishedTasks = new ArrayList<>(100);

public void markTaskDone(Task task) {

if (ifNeedWait(task)) {

mFinishedTasks.add(task.getClass());

mNeedWaitTasks.remove(task);

mCountDownLatch.countDown();

mNeedWaitCount.getAndDecrement();

}

}

可以看到，这里每执行完成一个task，就会将mCountDownLatch的锁计数减1，与此同时，也会将我们的mNeedWaitCount这个原子整数包装类的数量减1。

此外，我们在前面说到了启动器将各个任务之间的依赖关系抽象成了一个有向无环图，在上面一系列的初始化代码中，InitJPushTask是需要依赖于GetDeviceIdTask的，那么，我们怎么告诉启动器它们两者之间的依赖关系呢？

这里只需要在InitJPushTask中重写dependsOn()方法，并返回包含GetDeviceIdTask的task列表即可，代码如下所示：

/\*\*

\* InitJPushTask 需要在 getDeviceId 之后执行

\*/

public class InitJPushTask extends Task {

@Override

public List<Class<? extends Task>> dependsOn() {

List<Class<? extends Task>> task = new ArrayList<>();

task.add(GetDeviceIdTask.class);

return task;

}

@Override

public void run() {

JPushInterface.init(mContext);

MyApplication app = (MyApplication) mContext;

JPushInterface.setAlias(mContext, 0, app.getDeviceId());

}

}

至此，我们的异步启动器就分析完毕了。下面我们来看看如何高效地进行延迟初始化。

## 7.5延迟初始化

### 7.5.1常规方案：利用闪屏页的停留时间进行部分初始化

new Handler().postDelayed()。

界面UI展示后调用。

### 7.5.2常规初始化痛点

时机不容易控制：handler postDelayed指定的延迟时间不好估计。

导致界面UI卡顿：此时用户可能还在滑动列表。

### 7.5.3延迟优化最优解：延迟启动器

延迟启动器源码及使用demo

核心思想：

利用IdleHandler特性，在CPU空闲时执行，对延迟任务进行分批初始化。

延迟启动器优化实战与源码剖析：

延迟初始化启动器的代码很简单，如下所示：

/\*\*

\* 延迟初始化分发器

\*/

public class DelayInitDispatcher {

private Queue<Task> mDelayTasks = new LinkedList<>();

private MessageQueue.IdleHandler mIdleHandler = new MessageQueue.IdleHandler() {

@Override

public boolean queueIdle() {

// 分批执行的好处在于每一个task占用主线程的时间相对

// 来说很短暂，并且此时CPU是空闲的，这些能更有效地避免UI卡顿

if(mDelayTasks.size()>0){

Task task = mDelayTasks.poll();

new DispatchRunnable(task).run();

}

return !mDelayTasks.isEmpty();

}

};

public DelayInitDispatcher addTask(Task task){

mDelayTasks.add(task);

return this;

}

public void start(){

Looper.myQueue().addIdleHandler(mIdleHandler);

}

}

在DelayInitDispatcher中，我们提供了mDelayTasks队列用于将每一个task添加进来，使用者只需调用addTask方法即可。当CPU空闲时，mIdleHandler便会回调自身的queueIdle方法，这个时候我们可以将task一个一个地拿出来并执行。这种分批执行的好处在于每一个task占用主线程的时间相对来说很短暂，并且此时CPU是空闲的，这样能更有效地避免UI卡顿，真正地提升用户的体验。

至于使用就非常简单了，我们可以直接利用SplashActivity的广告页停留时间去进行延迟初始化，代码如下所示：

@Override

public void onWindowFocusChanged(boolean hasFocus) {

super.onWindowFocusChanged(hasFocus);

GlobalHandler.getInstance().getHandler().post((Runnable) () -> {

if (hasFocus) {

DelayInitDispatcher delayInitDispatcher = new DelayInitDispatcher();

delayInitDispatcher.addTask(new InitOtherTask())

.start();

}

});

}

需要注意的是，能异步的task我们会优先使用异步启动器在Application的onCreate方法中加载（或者是必须在Application的onCreate方法完成前必须执行完的非异task务），对于不能异步的task，我们可以利用延迟启动器进行加载。如果任务可以到用时再加载，可以使用懒加载的方式。

延迟启动器优势：

执行时机明确。

缓解界面UI卡顿。

真正提升用户体验。

## 7.6 Multidex预加载优化

我们都知道，安装或者升级后首次 MultiDex 花费的时间过于漫长，我们需要进行Multidex的预加载优化。

优化步骤：

1、启动时单独开一个进程去异步进行Multidex的第一次加载，即Dex提取和Dexopt操作。

2、此时，主进程Application进入while循环，不断检测Multidex操作是否完成。

3、执行到Multidex时，则已经发现提取并优化好了Dex，直接执行。MultiDex执行完之后主进程Application继续执行ContentProvider初始化和Application的onCreate方法。

Multidex优化Demo地址

注意：

5.0以上默认使用ART，在安装时已将Class.dex转换为oat文件了，无需优化，所以应判断只有在主进程及SDK 5.0以下才进行Multidex的预加载。

## 7.7类预加载优化

在Application中提前异步加载初始化耗时较长的类。

如何找到耗时较长的类？

替换系统的ClassLoader，打印类加载的时间，按需选取需要异步加载的类。

注意：

Class.forName()只加载类本身及其静态变量的引用类。

new 类实例 可以额外加载类成员变量的引用类。

## 7.8 WebView启动优化

1、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，注意内存泄漏问题。

3、本地离线包，即预置静态页面资源。

## 7.9 页面数据预加载

在主页空闲时，将其它页面的数据加载好保存到内存或数据库，等到打开该页面时，判断已经预加载过，就直接从内存或数据库取数据并显示。

## 7.10 启动阶段不启动子进程

子进程会共享CPU资源，导致主进程CPU紧张。此外，在多进程情况下一定要可以在onCreate中去区分进程做一些初始化工作。

注意启动顺序：App onCreate之前是ContentProvider初始化。

## 7.11 闪屏页与主页的绘制优化

1、布局优化。

2、过渡绘制优化。

布局越复杂，测量布局绘制的时间就越长。主要做到以下几点：

布局的层级越少，加载速度越快。

一个控件的属性越少，解析越快，删除控件中的无用属性。

使用<ViewStub/>标签加载一些不常用的布局，做到使用时在加载。

使用<merge/>标签减少布局的嵌套层次。

尽可能少用wrap\_content，wrap\_content会增加布局measure时的计算成本，已知宽高为固定值时，不用wrap\_content。

# 启动优化黑科技

## 8.1启动阶段抑制GC

启动时CG抑制，允许堆一直增长，直到手动或OOM停止GC抑制。（空间换时间）

缺点：需要白名单覆盖所有设备，但维护成本高。

## 8.2 CPU锁频

一个设备的CPU通常都是4核或者8核，但是应用在一般情况下对CPU的利用率并不高，可能只有30%或者50%，如果我们在启动速度暴力拉伸CPU频率，以此提高CPU的利用率，那么，应用的启动速度会提升不少。

在Android系统中，CPU相关的信息存储在/sys/devices/system/cpu目录的文件中，通过对该目录下的特定文件进行写值，实现对CPU频率等状态信息的更改。

缺点：暴力拉伸CPU频率，导致耗电量增加。

## 8.3 IO优化

1、启动过程不建议出现网络IO。

2、为了只解析启动过程中用到的数据，应选择合适的数据结构，如将ArrayMap改造成支持随机读写、延时解析的数据存储结构以替代SharePreference。

这里需要注意的是，需要考虑重度用户的使用场景。

## 8.4数据重排

Dex文件用到的类和APK里面各种资源文件都比较小，读取频繁，且磁盘地址分布范围比较广。我们可以利用Linux文件IO流程中的page cache机制将它们按照读取顺序重新排列在一起，以减少真实的磁盘IO次数。

1、类重排：使用Facebook的 ReDex 的Interdex调整类在Dex中的排列顺序。

2、资源文件重排：

1、最佳方案是修改内核源码，实现统计、度量、自动化，其次也可以使用Hook框架进行统计得出资源加载顺序列表。

2、最后，调整apk文件列表需要修改7zip源码以支持传入文件列表顺序。

## 8.5类加载优化（Dalvik）

1、类预加载原理

对象第一次创建的时候，JVM首先检查对应的Class对象是否已经加载。如果没有加载，JVM会根据类名查找.class文件，将其Class对象载入。同一个类第二次new的时候就不需要加载类对象，而是直接实例化，创建时间就缩短了。

2、类加载优化过程

在Dalvik VM加载类的时候会有一个类校验过程，它需要校验方法的每一个指令。

通过Hook去掉verify步骤 -> 几十ms的优化

最大优化场景在于首次安装和覆盖安装时，在Dalvik平台上，一个2MB的Dex正常需要350ms，将classVerifyMode设为VERIFY\_MODE\_NONE后，只需150ms，节省超过50%时间。

ART比较复杂，Hook需要兼容几个版本。而且在安装时，大部分Dex已经优化好了，去掉ART平台的verify只会对动态加载的Dex带来一些好处。所以暂时不建议在ART平台使用。

## 8.6 保活

1、厂商合作

2、微信Hardcoder

构建了App与系统（ROM）之间可靠的通信框架，让系统知道App的需求。

原理：

1、其实质是让App跨过Framework直接跟厂商ROM通信。

2、分为Client端和Server端，Server端由厂商系统侧自行实现。

3、它们直接采用 LocalSocket 方式，Hardcoder是 Native 实现的，使用了Linux的Socket接口实现了一套自己的LocalSocket。

性能提升有多少？平均10%~30%。

3、OPPO Hyper Boost加速引擎

一种优化资源调度的技术。

原理：让应用程序与系统资源实现实时"双向对话"。当来自应用和游戏程序的不同场景和用户行为被Hyper Boost识别后，手机会智能地匹配到合理的系统资源，让手机SoC的CPU、GPU、ISP、DSP提供的运算资源更加合理地利用，从而让用户使用手机更加流畅。

# 9.启动优化的常见问题

## 9.1启动优化是怎么做的？

1、分析现状、确认问题

2、针对性优化（先概括，引导其深入）

3、长期保持优化效果

在某一个版本之后呢，我们会发现这个启动速度变得特别慢，同时用户给我们的反馈也越来越多，所以，我们开始考虑对应用的启动速度来进行优化。然后，我们就对启动的代码进行了代码层面的梳理，我们发现应用的启动流程已经非常复杂，接着，我们通过一系列的工具来确认是否在主线程中执行了太多的耗时操作。

我们经过了细查代码之后，发现应用主线程中的任务太多，我们就想了一个方案去针对性地解决，也就是进行异步初始化。（引导=>第2题） 然后，我们还发现了另外一个问题，也可以进行针对性的优化，就是在我们的初始化代码当中有些的优先级并不是那么高，它可以不放在Application的onCreate中执行，而完全可以放在之后延迟执行的，因为我们对这些代码进行了延迟初始化，最后，我们还结合了idealHandler做了一个更优的延迟初始化的方案，利用它可以在主线程的空闲时间进行初始化，以减少启动耗时导致的卡顿现象。做完这些之后，我们的启动速度就变得很快了。

最后，我简单说下我们是怎么长期来保持启动优化的效果的。首先，我们做了我们的启动器，并且结合了我们的CI，在线上加上了很多方面的监控。（引导=> 第4题）

## 9.2是怎么异步的，异步遇到问题没有？

1、体现演进过程

2、详细介绍启动器

我们最初是采用的普通的一个异步的方案，即new Thread + 设置线程优先级为后台线程的方式在Application的onCreate方法中进行异步初始化，后来，我们使用了线程池、IntentService的方式，但是，在我们应用的演进过程当中，发现代码会变得不够优雅，并且有些场景非常不好处理，比如说多个初始化任务直接的依赖关系，比如说某一个初始化任务需要在某一个特定的生命周期中初始化完成，这些都是使用线程池、IntentService无法实现的。所以说，我们就开始思考一个新的解决方案，它能够完美地解决我们刚刚所遇到的这些问题。

这个方案就是我们目前所使用的启动器，在启动器的概念中，我们将每一个初始化代码抽象成了一个Task，然后，对它们进行了一个排序，根据它们之间的依赖关系排了一个有向无环图，接着，使用一个异步队列进行执行，并且这个异步队列它和CPU的核心数是强烈相关的，它能够最大程度地保证我们的主线程和别的线程都能够执行我们的任务，也就是大家几乎都可以同时完成。

# 10.总结

## 优化总方针

异步初始化、延迟初始化、懒加载

技术、业务相结合

开发者要未雨绸缪，在编码过程中尽量减少给启动带来性能损耗的工作，主要注意以下几个事项：

尽量避免启动时在主线程做密集繁重的工作，如：避免 I/O 操作、反序列化、网络操作、锁等待等。

对模块以及第三方库按需加载，采取分步加载、异步加载、延期加载等策略。

利用线程池管理线程，避免创建大量线程，造成 CPU 竞争，导致主线程时间片减少。

启动过程中，尽量避免频繁创建的大量对象，减少 GC 给启动性能带来的卡顿影响。

尽量避免在启动过程中调用阻塞性的系统调用。