Android App启动优化

用户希望应用能够及时响应并快速加载。启动时间过长的应用可能会导致用户在对应用给出很低的评分，甚至完全弃用。

启动状态

应用有三种启动状态，每种状态都会影响应用向用户显示所需的时间：冷启动、温启动与热启动。在冷启动中，应用从头开始启动。在另外两种状态中，系统需要将后台运行的应用带入前台。建议始终在假定冷启动的基础上进行优化。这样做也可以提升温启动和热启动的性能。

> - 冷启动
>   - 冷启动是指应用从头开始启动：系统进程在冷启动后才创建应用进程。发生冷启动的情况包括应用自设备启动后或系统终止应用后首次启动。
> - 热启动：
>   - 在热启动中，系统的所有工作就是将 Activity 带到前台。只要应用的所有 Activity 仍驻留在内存中，应用就不必重复执行对象初始化、布局加载和绘制。
> - 温启动
>   - 温启动包含了在冷启动期间发生的部分操作；同时，它的开销要比热启动高。有许多潜在状态可视为温启动。例如：
>   - 用户在退出应用后又重新启动应用。进程可能未被销毁，继续运行，但应用需要执行 `onCreate()` 从头开始重新创建 Activity。
>   - 系统将应用从内存中释放，然后用户又重新启动它。进程和 Activity 需要重启，但传递到 `onCreate()` 的已保存的实例 state bundle 对于完成此任务有一定助益。

冷启动耗时统计

在性能测试中存在启动时间2-5-8原则：

- 当用户能够在2秒以内得到响应时，会感觉系统的响应很快；
- 当用户在2-5秒之间得到响应时，会感觉系统的响应速度还可以；
- 当用户在5-8秒以内得到响应时，会感觉系统的响应速度很慢，但是还可以接受；
- 而当用户在超过8秒后仍然无法得到响应时，会感觉系统糟透了，或者认为系统已经失去响应。

而Google 也提出一项计划：**Android Vitals** 。该计划旨在改善 Android 设备的稳定性和性能。当选择启用了该计划的用户运行您的应用时，其 Android 设备会记录各种指标，包括应用稳定性、应用启动时间、电池使用情况、呈现时间和权限遭拒等方面的数据。Google Play 管理中心 会汇总这些数据，并将其显示在 Android Vitals 信息中心 内。

当应用启动时间过长时，Android Vitals 可以通过 Play 管理中心提醒您，从而帮助提升应用性能。Android Vitals 在您的应用出现以下情况时将其启动时间视为过长：

- 冷启动用了 5 秒或更长时间。
- 温启动用了 2 秒或更长时间。
- 热启动用了 1.5 秒或更长时间。

实际上不同的应用因为启动时需要初始化的数据不同，启动时间自然也会不同。相同的应用也会因为在不同的设备，因为设备性能影响启动速度不同。所以实际上启动时间并没有绝对统一的标准，我们之所以需要进行启动耗时的统计的，可能在于产品对我们应用启动时间提出具体的要求。

系统日志统计

在 Android 4.4（API 级别 19）及更高版本中，logcat 包含一个输出行，其中包含名为 `Displayed` 的值。此值代表从启动进程到在屏幕上完成对应 Activity 的绘制所用的时间。

```
ActivityManager: Displayed com.android.myexample/.StartupTiming: +3s534ms
```

如果我们使用异步懒加载的方式来提升程序画面的显示速度，这通常会导致的一个问题是，程序画面已经显示，同时 `Displayed` 日志已经打印，可是内容却还在加载中。为了衡量这些异步加载资源所耗费的时间，我们可以在异步加载完毕之后调用 `activity.reportFullyDrawn()` 方法来让系统打印到调用此方法为止的启动耗时。

adb 命令统计

查看启动时间的另一种方式是使用命令：

```
adb [-d|-e|-s <serialNumber>] shell am start -S -W
    com.example.app/.MainActivity
    -c android.intent.category.LAUNCHER
    -a android.intent.action.MAIN
```

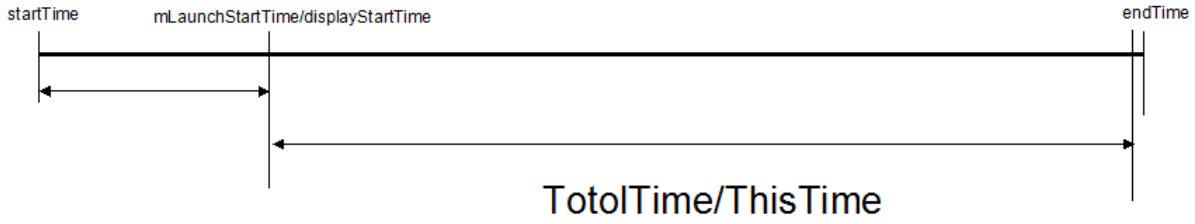
启动完成后，将输出：

```
ThisTime: 415
TotalTime: 415
WaitTime: 437
```

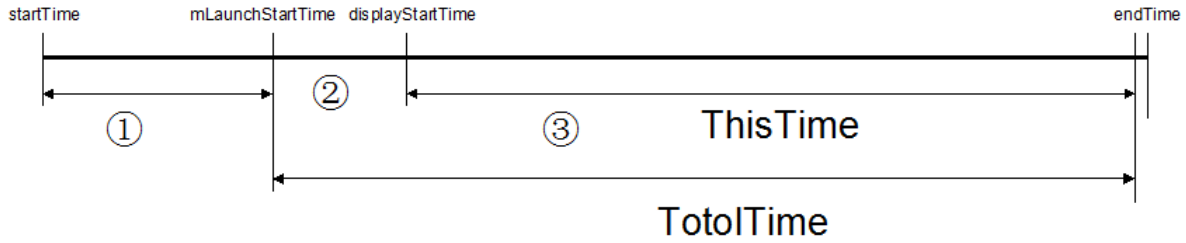
- WaitTime:总的耗时，包括前一个应用Activity pause的时间和新应用启动的时间；
- ThisTime表示一连串启动Activity的最后一个Activity的启动耗时；
- TotalTime表示新应用启动的耗时，包括新进程的启动和Activity的启动，但不包括前一个应用Activity pause的耗时。

开发者一般只要关心**TotalTime**即可，这个时间才是自己应用真正启动的耗时。

启动单个Activity



启动一连串Activity



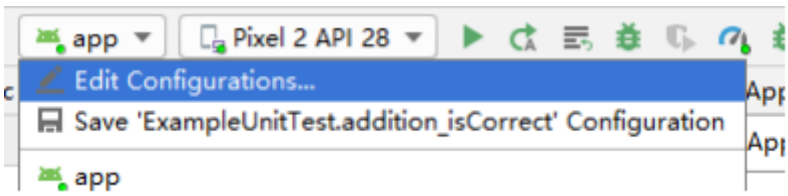
CPU Profile/TraceView

如果发现显示时间比希望的时间长，则可以继续尝试识别启动过程中的瓶颈。 查找瓶颈的一个好方法是使用 Android Studio CPU 性能剖析器。

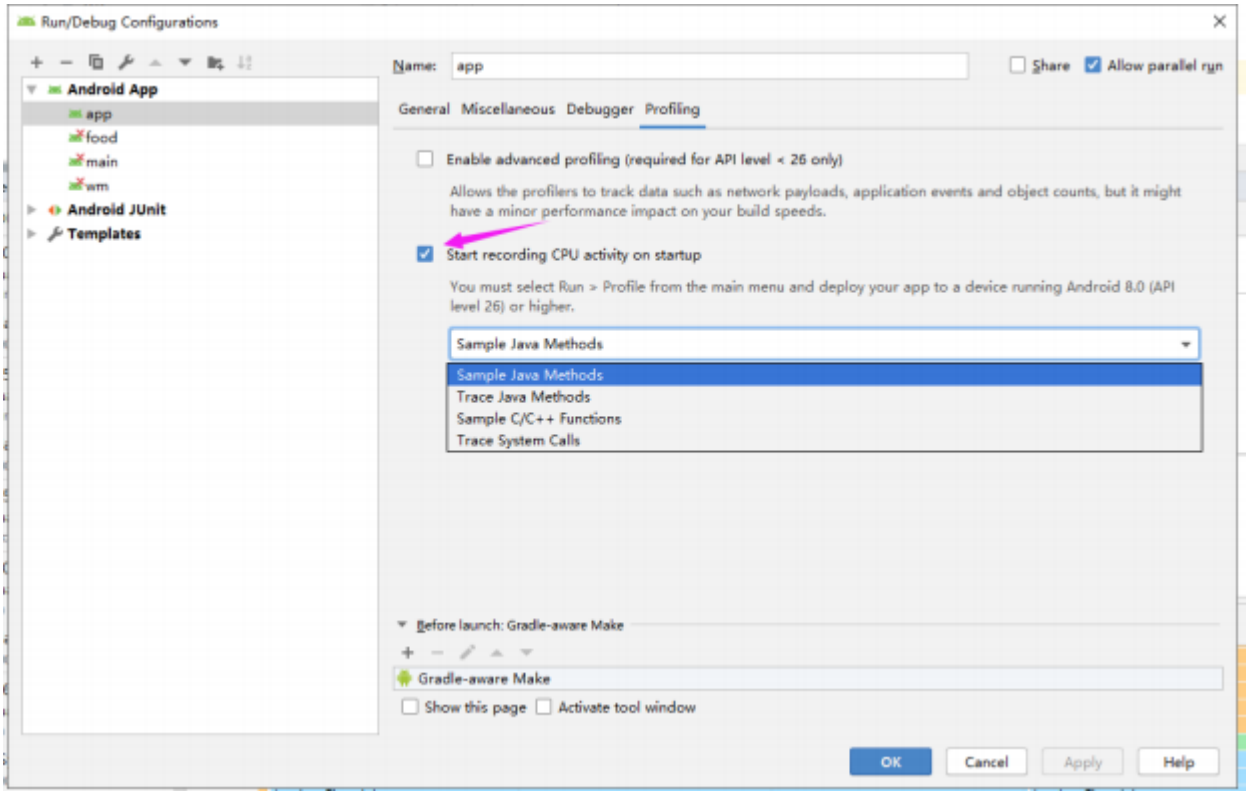
> Traceview是android平台配备一个很好的性能分析的工具。它可以通过图形化的方式让我们了解我们要跟踪的程序的性能，并且能具体到每个方法的执行时间。但是目前**Traceview 已弃用**。如果使用 Android Studio 3.2 或更高版本，则应改为使用 CPU Profiler

要在应用启动过程中自动开始记录 CPU 活动，请执行以下操作：

1. 依次选择 **Run > Edit Configurations。**



2. 在 **Profiling** 标签中，勾选 **Start recording CPU activity on startup** 旁边的复选框。



3. 从菜单中选择 CPU 记录配置。

   Sample Java Methods

   对 Java 方法采样：在应用的 Java 代码执行期间，频繁捕获应用的调用堆栈。分析器会比较捕获的数据集，以推导与应用的 Java 代码执行有关的时间和资源使用信息。如果应用在捕获调用堆栈后进入一个方法并在下次捕获前退出该方法，分析器将不会记录该方法调用。如果您想要跟踪生命周期如此短的方法，应使用检测跟踪。

   Trace Java Methods

   跟踪 Java 方法：在运行时检测应用，以在每个方法调用开始和结束时记录一个时间戳。系统会收集并比较这些时间戳，以生成方法跟踪数据，包括时间信息和 CPU 使用率。

   Sample C/C++ Functions

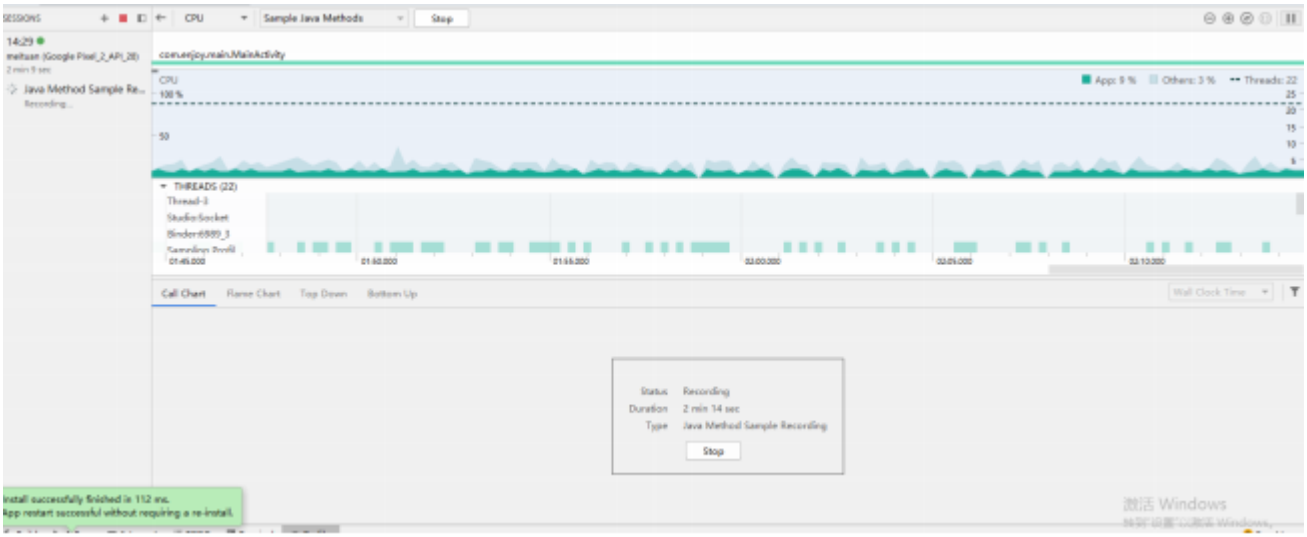
   对 C/C++ 函数采样：捕获应用的原生线程的采样跟踪数据。要使用此配置，您必须将应用部署到搭载 Android 8.0（API 级别 26）或更高版本的设备上。

   Trace System Calls

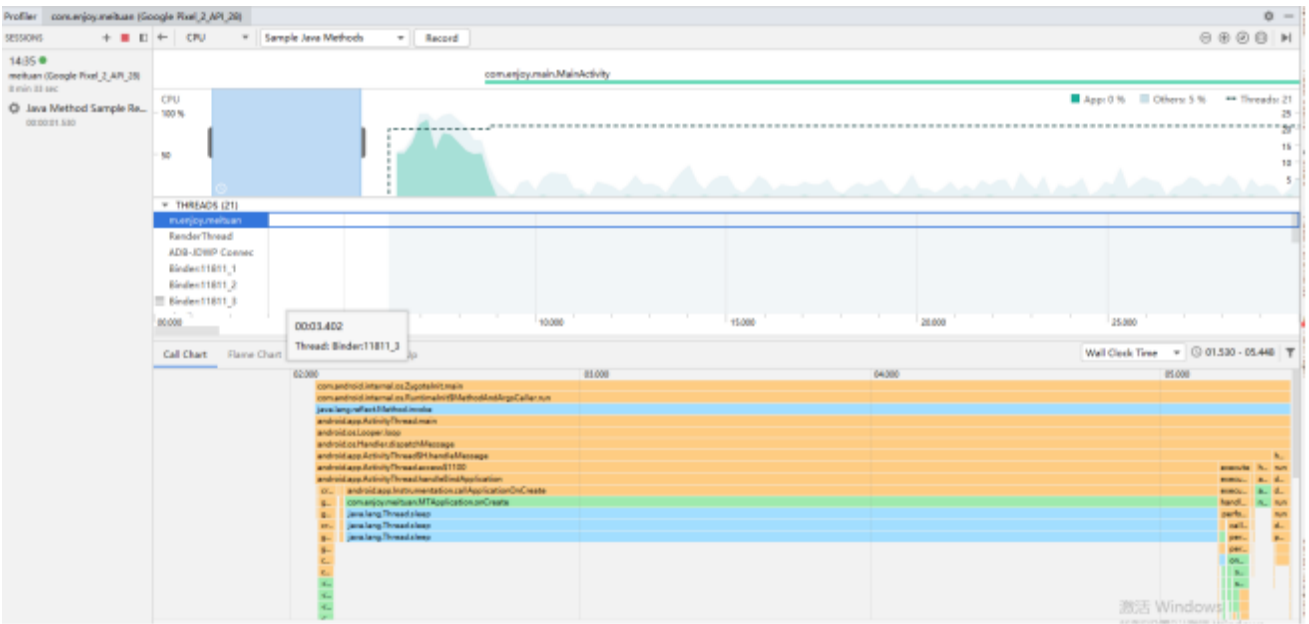
   跟踪系统调用：捕获非常翔实的细节，以便您检查应用与系统资源的交互情况。您可以检查线程状态的确切时间和持续时间、直观地查看所有内核的 CPU 瓶颈在何处，并添加要分析的自定义跟踪事件。要使用此配置，您必须将应用部署到搭载 Android 7.0（API 级别 24）或更高版本的设备上。

   此跟踪配置在 systrace 的基础上构建而成。您可以使用 systrace 命令行实用程序指定除 CPU Profiler 提供的选项之外的其他选项。systrace 提供的其他系统级数据可帮助您检查原生系统进程并排查丢帧或帧延迟问题。

4. 点击 **Apply**。
5. 依次选择 **Run > Profile**，将您的应用部署到搭载 Android 8.0（API 级别 26）或更高版本的设备上。

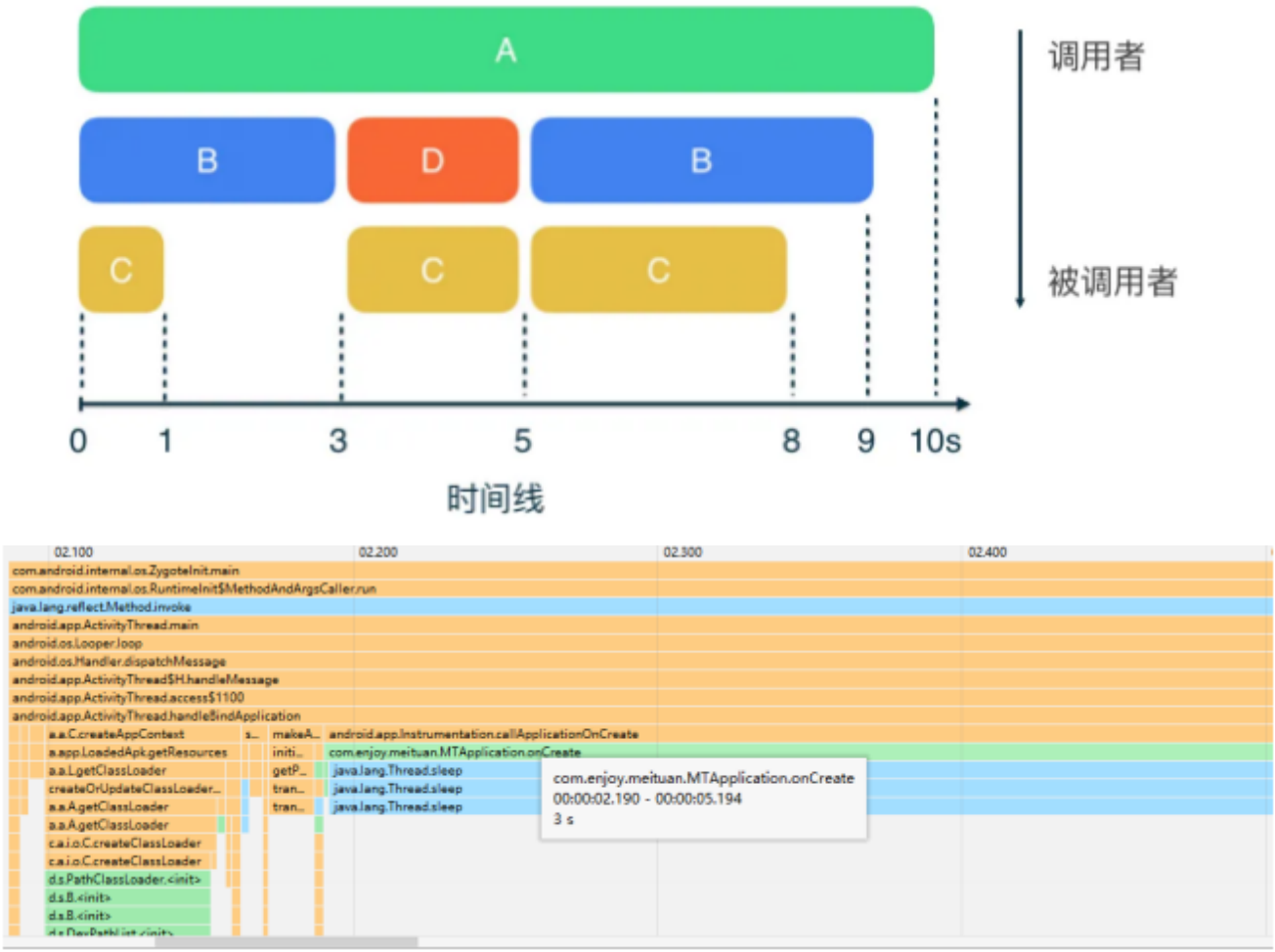


点击Stop，结束跟踪后显示：



Call Chart

以图形来呈现方法跟踪数据或函数跟踪数据，其中调用的时间段和时间在横轴上表示，而其被调用方则在纵轴上显示。对系统 API 的调用显示为橙色，对应用自有方法的调用显示为绿色，对第三方 API（包括 Java 语言 API）的调用显示为蓝色。 **（实际颜色显示有Bug）**



如上图，自定义Application的 onCreate 调用了`Thread.sleep`耗时为：3s。

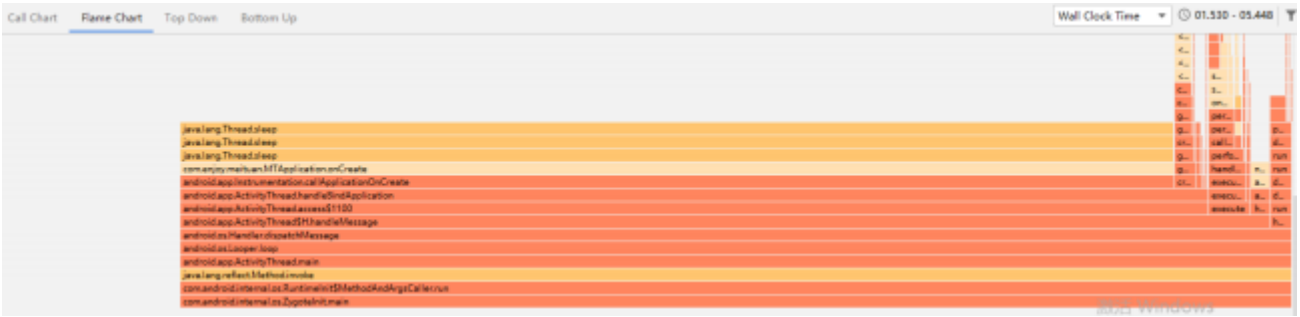
Call Chart 已经比原数据可读性高很多，但它仍然不方便发现那些运行时间很长的代码，这时我们便需要使用 Flame Chart。

Flame Chart

提供一个倒置的调用图表，用来汇总完全相同的调用堆栈。也就是说，将具有相同调用方顺序的完全相同的方法或函数收集起来，并在火焰图中将它们表示为一个较长的横条 。

横轴显示的是百分比数值。由于忽略了时间线信息，Flame Chart 可以展示每次调用消耗时间占用整个记录时长的百分比。 同时纵轴也被对调了，在顶部展示的是被调用者，底部展示的是调用者。此时的图表看起来越往上越窄，就好像火焰一样，因此得名: **火焰图。**

> 说白了就是将Call Chart上下调用栈倒过来。

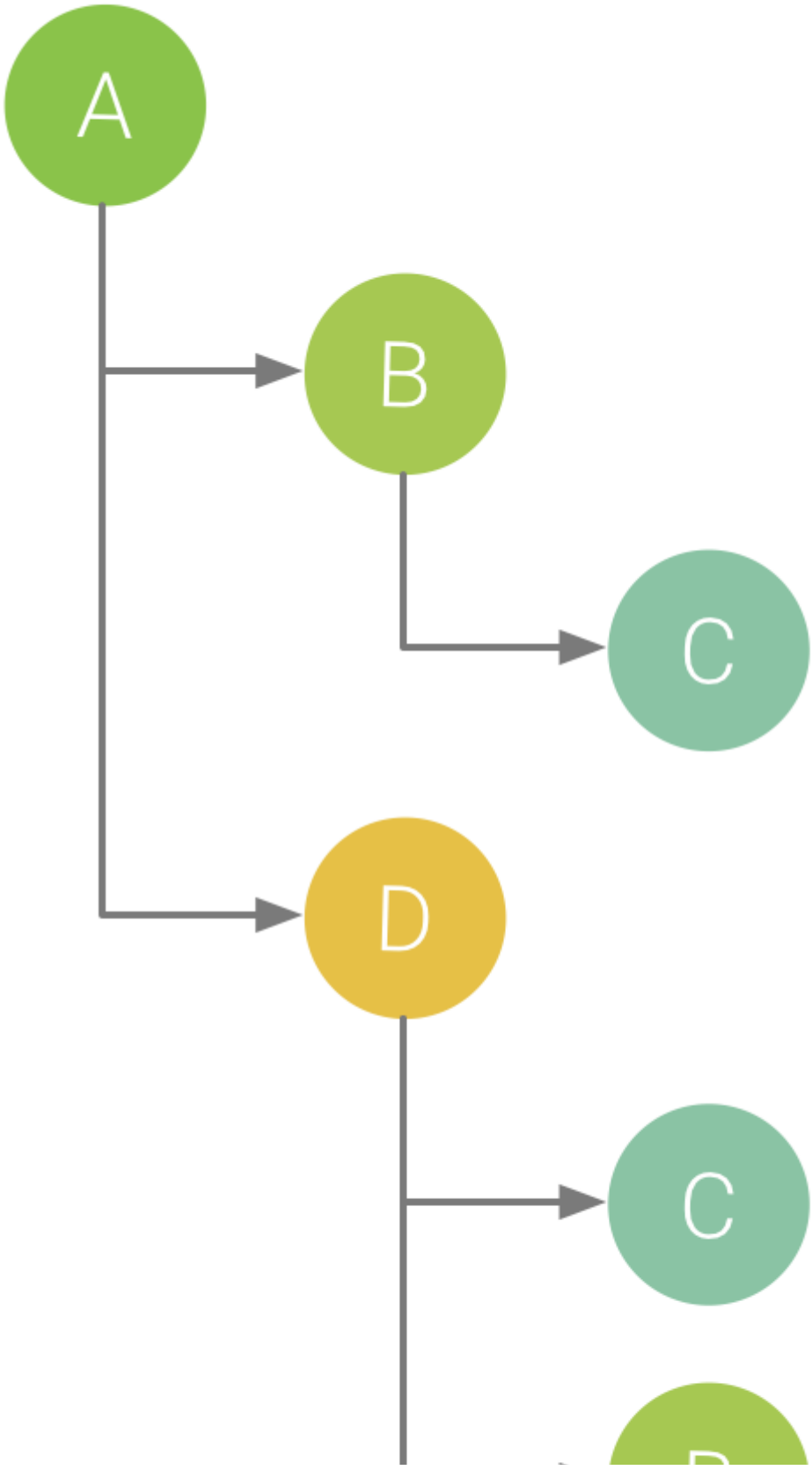


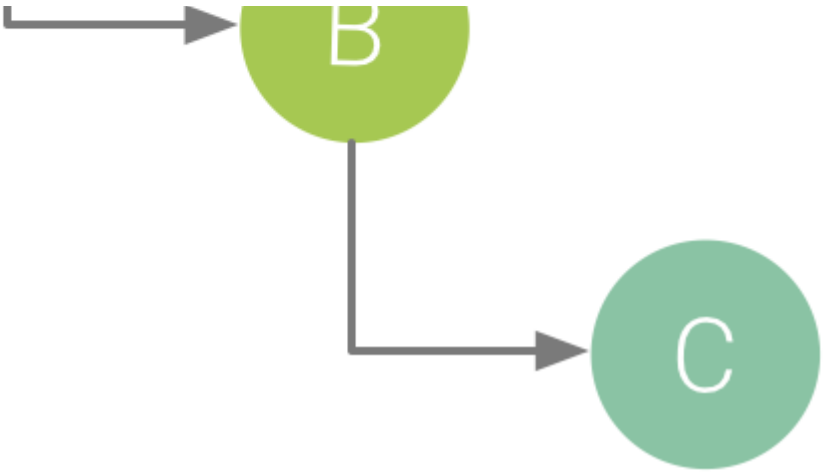
耗时最长的为：**Thread.sleep**

Top Down Tree

如果我们需要更精确的时间信息，就需要使用 Top Down Tree。 Top Down Tree显示一个调用列表，在该列表中展开方法或函数节点会显示它调用了的方法节点。

Top Down





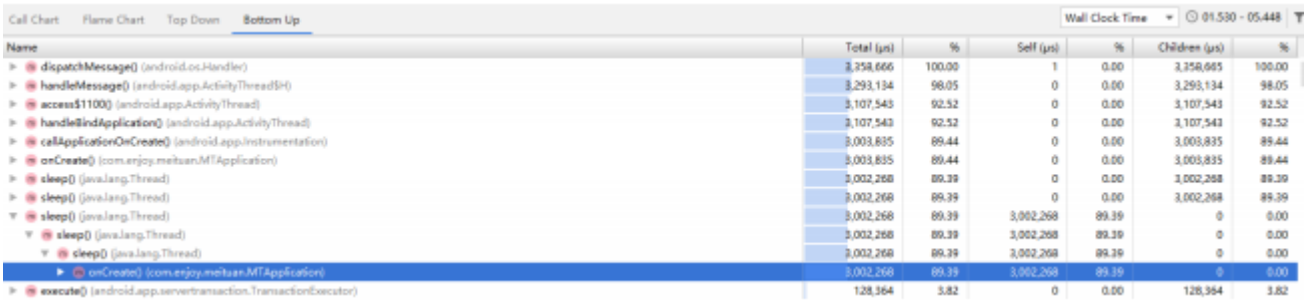
对于每个节点，三个时间信息:

- Self Time —— 运行自己的代码所消耗的时间;
- Children Time —— 调用其他方法的时间;
- Total Time —— 前面两者时间之和。

> 此视图能够非常方便看到耗时最长的方法调用栈。

Bottom Up Tree

方便地找到某个方法的调用栈。在该列表中展开方法或函数节点会显示哪个方法调用了自己。



通过工具可以定位到耗时代码，然后查看是否可以进行优化。对于APP启动来说，启动耗时包括Android系统启动APP进程加上APP启动界面的耗时时长，我们可做的优化是APP启动界面的耗时，也就是说从Application的构建到主界面的`onWindowFocusChanged`的这一段时间。

因此在这段时间内，我们的代码需要尽量避免耗时操作，检查的方向包括：主线程IO；第三方库初始化或程序需要使用的数据等初始化改为异步加载/懒加载；减少布局复杂度与嵌套层级；Multidex(5.0以上无需考虑)等。

Debug API

除了直接使用 **Profile** 启动之外，我们还可以借助Debug API生成trace文件。

```
public class MyApplication extends Application {

    public MyApplication() {
        Debug.startMethodTracing("enjoy");
    }
    //.....
}
public class MainActivity extends AppCompatActivity {

    @Override
    public void onWindowFocusChanged(boolean hasFocus) {
        super.onWindowFocusChanged(hasFocus);
        Debug.stopMethodTracing();
    }
    //.......
}
```

运行App，则会在sdcard中生成一个enjoy.trace文件（需要sdcard读写权限）。将手机中的trace文件保存至电脑，随后拖入Android Studio即可。

StrictMode严苛模式

StrictMode是一个开发人员工具，它可以检测出我们可能无意中做的事情，并将它们提请我们注意，以便我们能够修复它们。

StrictMode最常用于捕获应用程序主线程上的意外磁盘或网络访问。帮助我们让磁盘和网络操作远离主线程，可以使应用程序更加平滑、响应更快。

```
public class MyApplication extends Application {
    @Override
    public void onCreate() {
        if (BuildConfig.DEBUG) {
            //线程检测策略
            StrictMode.setThreadPolicy(new StrictMode.ThreadPolicy.Builder()
                    .detectDiskReads()   //读、写操作
                    .detectDiskWrites()
                    .detectNetwork()   // or .detectAll() for all detectable problems
                    .penaltyLog()
                    .build());
            StrictMode.setVmPolicy(new StrictMode.VmPolicy.Builder()
                    .detectLeakedSqlLiteObjects()   //Sqlite对象泄露
```

```
                    .detectLeakedClosableObjects()  //未关闭的Closable对象泄露
                    .penaltyLog()  //违规打印日志
                    .penaltyDeath() //违规崩溃
                    .build());
        }
}
```

启动黑白屏

当系统加载并启动 App 时，需要耗费相应的时间，这样会造成用户会感觉到当点击 App 图标时会有 “延迟” 现象，为了解决这一问题，Google 的做法是在 App 创建的过程中，先展示一个空白页面，让用户体会到点击图标之后立马就有响应。

如果你的application或activity启动的过程太慢，导致系统的BackgroundWindow没有及时被替换，就会出现启动时白屏或黑屏的情况（取决于Theme主题是Dark还是Light）。

消除启动时的黑/白屏问题，大部分App都采用自己在Theme中设置背景图的方式来解决。

```
<style name="AppTheme.Launcher">
        <item name="android:windowBackground">@drawable/bg</item>
</style>

<activity
    android:name=".activity.SplashActivity"
    android:screenOrientation="portrait"
    android:theme="@style/AppTheme.Launcher">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

然后在Activity的onCreate方法，把Activity设置回原来的主题。

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    //替换为原来的主题，在onCreate之前调用
    setTheme(R.style.AppTheme);
    super.onCreate(savedInstanceState);
}
```

这么做，只是提高启动的用户体验。并不能做到真正的加快启动速度。

总结

启动速度优化也会涉及到布局优化与卡顿优化，包括内存抖动等问题。优化是一条持续的道路，很多时候我们会发现通过各种检测手段花费了大量的精力去对代码进行修改得到的优化效果可能并不理想。因为优化就是一点一滴积累下来的，我们平时在编码的过程中就需要多注意自己的代码性能。

可能实际过程中优化并不会很顺利，不同的设备上可能表现不一样。我们只能结合对业务、对自己代码的了解去不断去实践。