胡凯

- RSS

Search

» RSS

- 首页
- 存档
- Android官方培训课程
- 关于胡凯
- 友情链接

Android性能优化典范 – 第6季

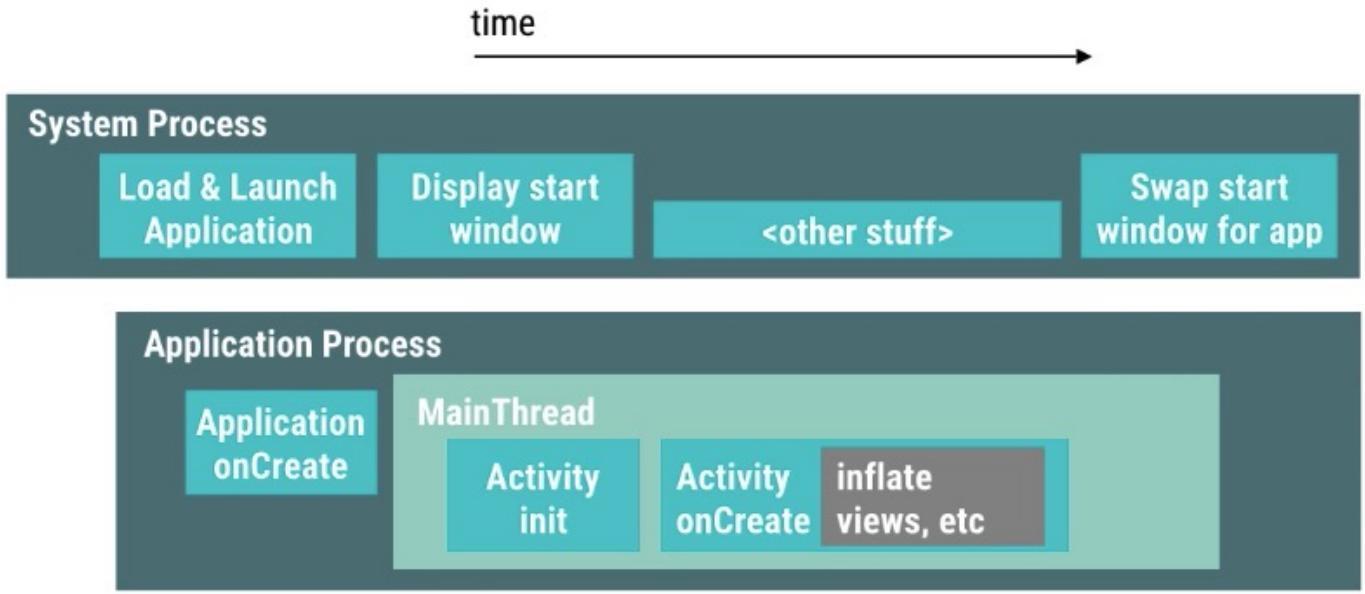
Oct 4th, 2016 | Comments

这里是Android性能优化典范第6季的课程学习笔记，从被@知会到有连载更新，这篇学习笔记就一直被惦记着，现在学习记录分享一下，请多多指教包涵！这次一共才6个小段落，涉及的内容主要有：程序启动时间性能优化的三个方面：优化activity的创建过程，优化application对象的启动过程，正确使用启动显屏达到优化程序启动性能的目的。另外还介绍了减少安装包大小的checklist以及如何使用VectorDrawable来减少安装包的大小。

1）App Launch time 101

提高程序的启动速度意义重大，很显然，启动时间越短，用户才越有耐心等待打开这个APP进行使用，反之启动时间越长，用户则越有可能来不及等到APP打开就已经切换到其他APP了。程序启动过程中的那些复杂错误的操作很可能导致严重的性能问题。Android系统会根据用户的操作行为调整程序的显示策略，用来提高程序的显示性能。例如，一旦用户点击桌面图标，Android系统会立即显示一个启动窗口，这个窗口会一直保持显示直到画面中的元素成功加载并绘制完第一帧。这种行为常见于程序的冷启动，或者程序的热启动场景（程序从后台被唤起或者从其他APP界面切换回来）。那么关键的问题是，用户很可能会因为从启动窗口到显示画面的过程耗时过长而感到厌烦，从而导致用户没有来得及等程序启动完毕就切换到其他APP了。更严重的是，如果启动时间过长，可能导致程序出现ANR。我们应该避免出现这两种糟糕的情况。

从技术角度来说，当用户点击桌面图标开始，系统会立即为这个APP创建独立的专属进程，然后显示启动窗口，直到APP在自己的进程里面完成了程序的创建以及主线程完成了Activity的初始化显示操作，再然后系统进程就会把启动窗口替换成APP的显示窗口。

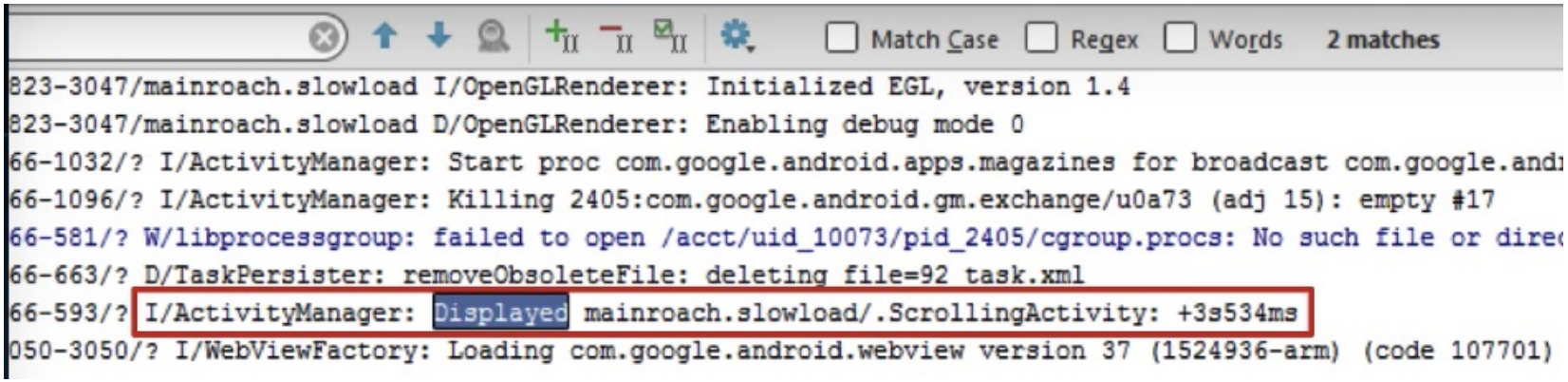


上述流程里面的绝大多数步骤都是由系统控制的，一般来说不会出现什么问题，可是对于启动速度，我们能够控制并且需要特别关注的地方主要有三处：

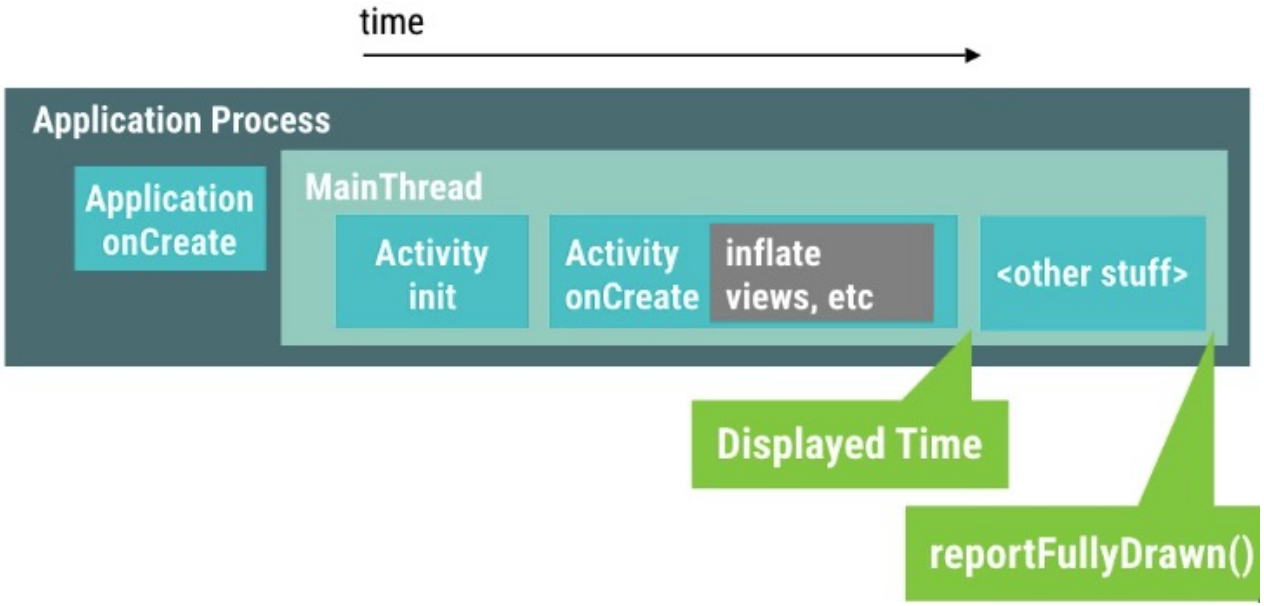
- 1）Activity的onCreate流程，特别是UI的布局与渲染操作，如果布局过于复杂很可能导致严重的启动性能问题。
- 2）Application的onCreate流程，对于大型的APP来说，通常会在这里做大量的通用组件的初始化操作。
- 3）目前有部分APP会提供自定义的启动窗口，这里可以做成品牌宣传界面或者是给用户提供一种程序已经启动的视觉效果。

在正式着手解决问题之前，我们需要掌握一套正确测量评估启动性能的方法。所幸的是，Android系统有提供一些工具来帮助我们定位问题。

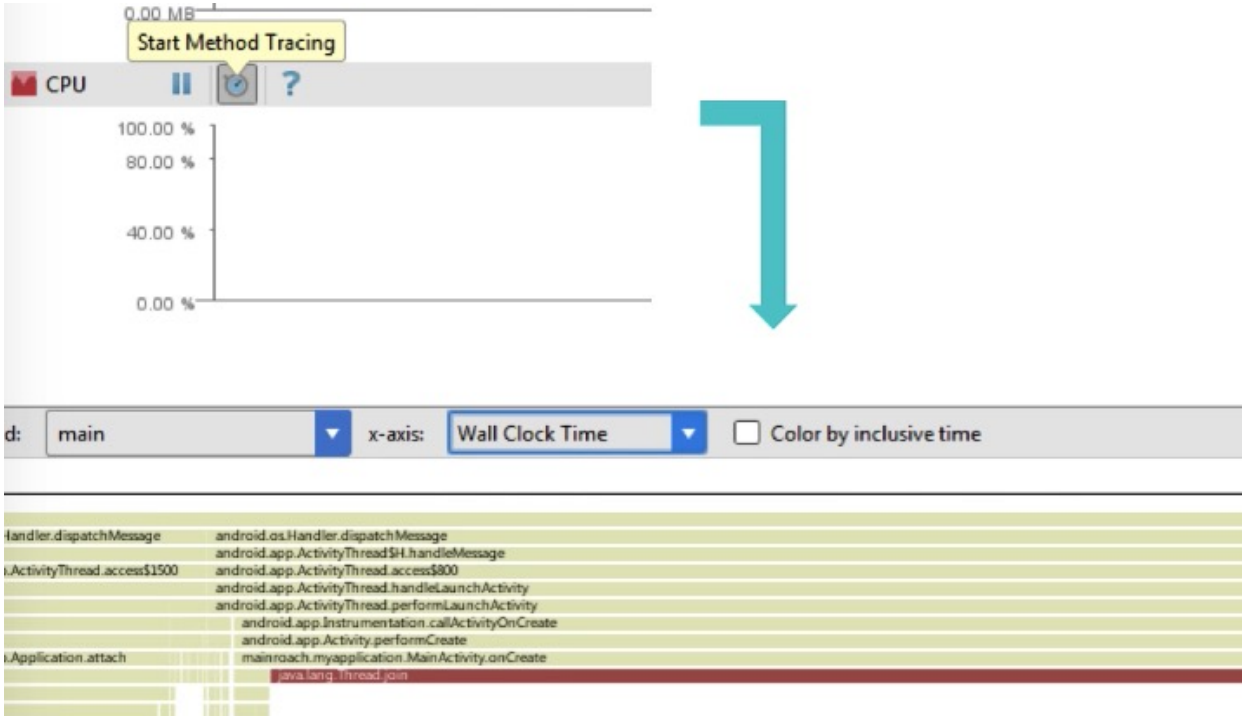
- 1）首先是**display time**：从Android KitKat版本开始，Logcat中会输出从程序启动到某个Activity显示到画面上所花费的时间。这个方法比较适合测量程序的启动时间。



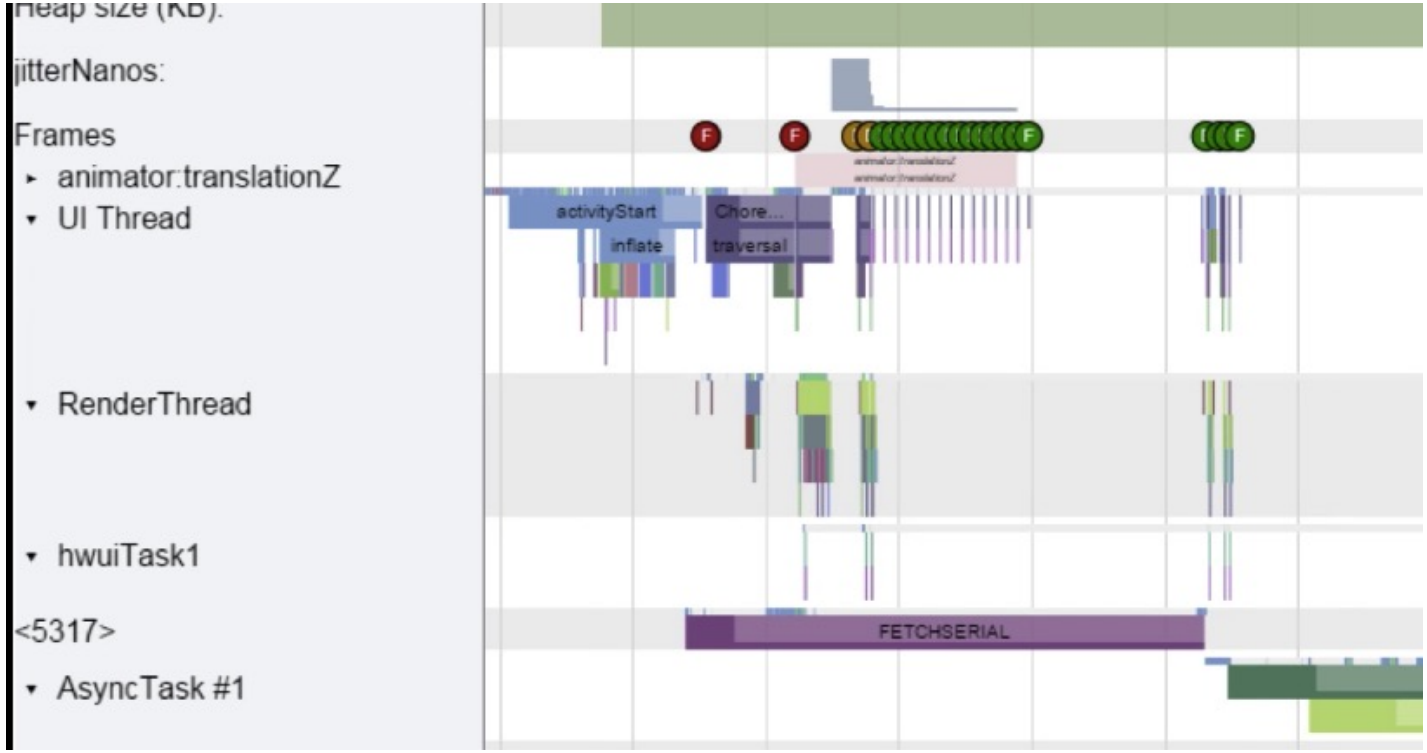
- 2）其次是**reportFullyDrawn**方法：我们通常来说会使用异步懒加载的方式来提升程序画面的显示速度，这通常会导致的一个问题是，程序画面已经显示，可是内容却还在加载中。为了衡量这些异步加载资源所耗费的时间，我们可以在异步加载完毕之后调用`activity.reportFullyDrawn()`方法来告诉系统此时的状态，以便获取整个加载的耗时。



- 3）然后是**Method Tracing**：前面两个方法提供了启动耗时的总时间，可是却无法提供具体的耗时细节。为了获取具体的耗时分布情况，我们可以使用Method Tracing工具来进行详细的测量。

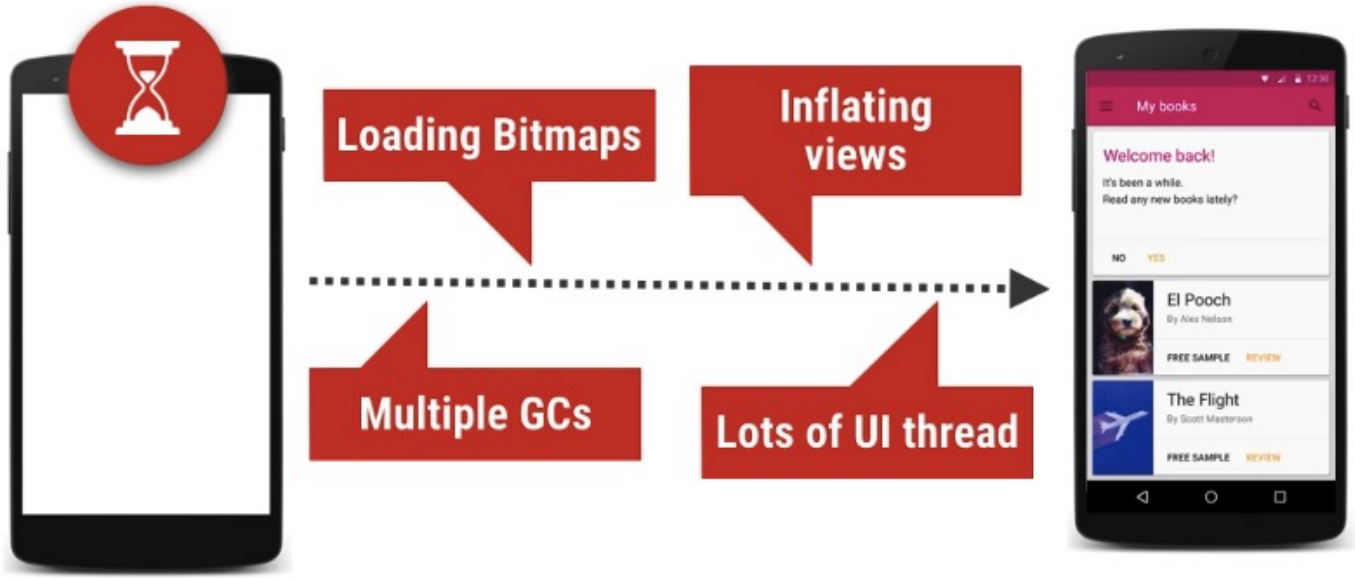


- 4）最后是**Systrace**：我们可以在onCreate方法里面添加trace.beginSection()与trace.endSection()方法来声明需要跟踪的起止位置，系统会帮忙统计中间经历过的函数调用耗时，并输出报表。



2）App Launch Time & Activity Creation

提升Activity的创建速度是优化APP启动速度的首要关注目标。从桌面点击APP图标启动应用开始，程序会显示一个启动窗口等待Activity的创建加载完毕再进行显示。在Activity的创建加载过程中，会执行很多的操作，例如设置页面的主题，初始化页面的布局，加载图片，获取网络数据，读写Preference等等。



上述操作的任何一个环节出现性能问题都可能导致画面不能及时显示，影响了程序的启动速度。上一个段落我们介绍了使用Method Tracing来发现那些耗时占比相对较多的方法。假设我们发现某个方法执行时间过长，接下去就可以使用Systrace来帮忙定位到底是什么原因导致那个方法执行时间过长。

除了使用工具进行具体定位分析性能问题之外，以下两点经验可以帮助我们对Activity启动做性能优化：

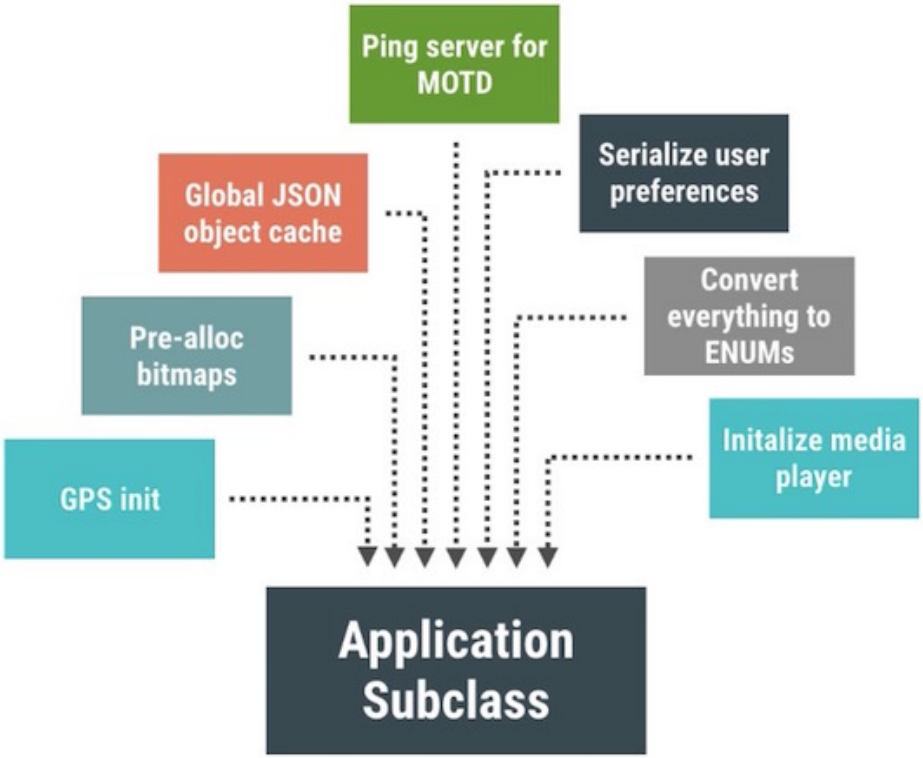
- 1）优化布局耗时：一个布局层级越深，里面包含需要加载的元素越多，就会耗费更多的初始化时间。关于布局性能的优化，这里就不展开描述了！
- 2）异步延迟加载：一开始只初始化最需要的布局，异步加载图片，非立即需要的组件可以做延迟加载。

3）App Launch Time & Bloated Application Objects

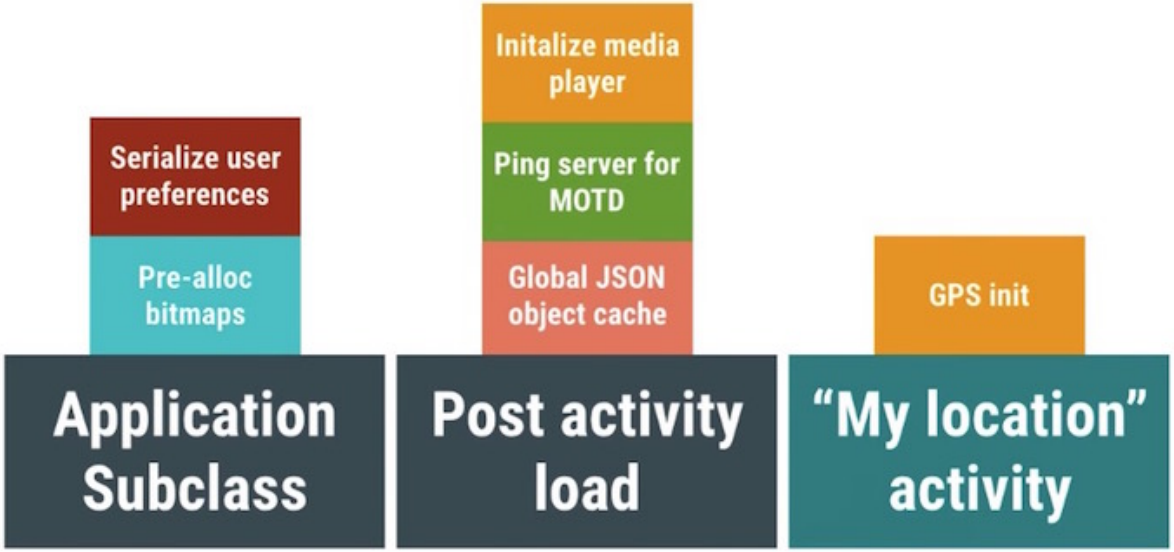
在Application初始化的地方做太多繁重的事情是可能导致严重启动性能问题的元凶之一。Application里面的初始化操作不结束，其他任意的程序操作都无法进行。

```
public class SlowLoadApp extends Application
{
    @Override
    public void onCreate()
    {
        super.onCreate();

        // Read our user preferences from the last time we loaded this app.
        Global.readPreferences(this);

        // Ping the server and get our configurations
        Server.waitForServerLogin(this);

        // Go ahead and create the cache pool for bitmaps you see in the fragr
        Global.initCachesAndPreallocate();

        //some long running function
        imPrettySureSortingIsFree();
    }
```

有时候，我们会一股脑的把绝大多数全局组件的初始化操作都放在Application的onCreate里面，但其实很多组件是需要做区队对待的，有些可以做延迟加载，有些可以放到其他的地方做初始化操作，特别需要留意包含Disk IO操作，网络访问等严重耗时的任务，他们会严重阻塞程序的启动。



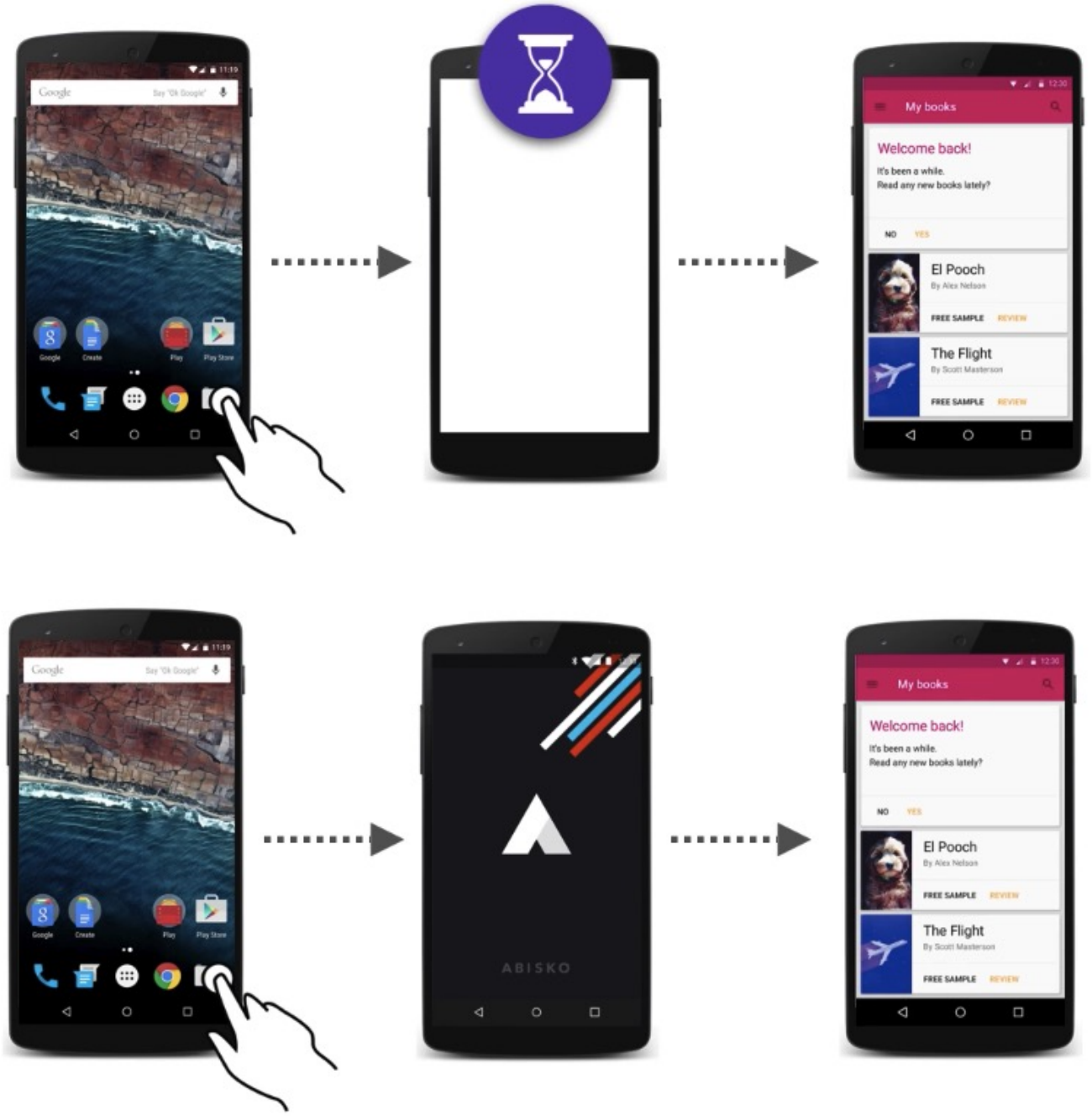
优化这些问题的解决方案是做延迟加载，可以在application里面做延迟加载，也可以把一些初始化的操作延迟到组件真正被调用到的时候再做加载。



4）App Launch Time & Theme Launch Screens

启动闪屏不仅仅可以作为品牌宣传页，还能够减轻用户对启动耗时的感知，但是如果使用不恰当，将适得其反。前面介绍过当点击桌面图标启动APP的时候，程序会显示一个启动窗口，一直到页面的渲染加载完毕。如果程序的启动速度足够快，我们看的闪屏窗口停留显示的时间则会很短，但是当程序启动速度偏慢的时候，这个启动闪屏可以一定程度上减轻用户等待的焦虑感，避免用户过于轻易的关闭应用。

目前大多数开发者都会通过设置启动窗口主题的方式来替换系统默认的启动窗口，通过这种方式只是使用『障眼法』弱化了用户对启动时间的感知，但本质上并没有对启动速度做什么优化。也有些APP通过关闭启动窗口属性`android:windowDisablePreview`的方式来直接移除系统默认的启动窗口，但是这样的弊端是用户从点击桌面图标到真的看到实际页面的这段时间当中，画面没有任何变化，这样的用户体验是十分糟糕的！



对于启动闪屏，正确的使用方法是自定义一张图片，把这张图片通过设置主题的方式显示为启动闪屏，代码执行到主页面的onCreate的时候设置为程序正常的主题。

```xml
<layer-list xmlns:android="http://schemas.android.com/apk/res/android"
android:opacity="opaque">
  <!-- The background color, preferably the same as your normal theme -
  <item android:drawable="@android:color/white"/>
  <!-- Your product logo - 144dp color version of your app icon -->
  <item>
    <bitmap
      android:src="@drawable/product_logo_144dp"
      android:gravity="center"/>
  </item>
</layer-list>
```

```xml
<activity ... android:theme="@style/AppTheme.Launcher" />
```

```java
public class MyMainActivity extends AppCompatActivity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    // Make sure this is before calling super.onCreate
    setTheme(R.style.Theme_MyApp);
    super.onCreate(savedInstanceState);
    // ...
  }
}
```

5）Smaller APKs: A Checklist

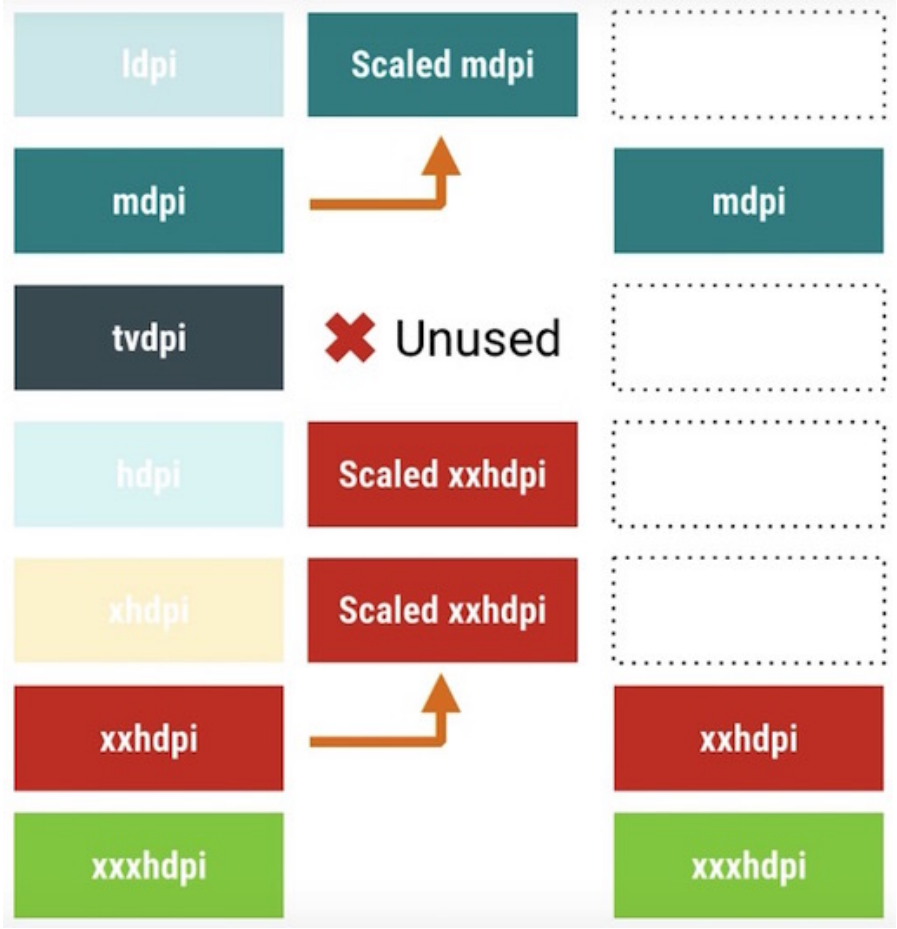
减少应用程序安装包的大小，不仅仅减少了用户的网络数据流量还减少了下载等待的时间。毋庸置疑，尽量减少程序安装包的大小是十分有必要的。通常来说，减少程序安装包的大小有两条规律：要么减少程序资源的大小，要么就是减少程序的代码量。这里总结一个简易版的减少安装包大小的Checklist:

减少程序图片资源的大小

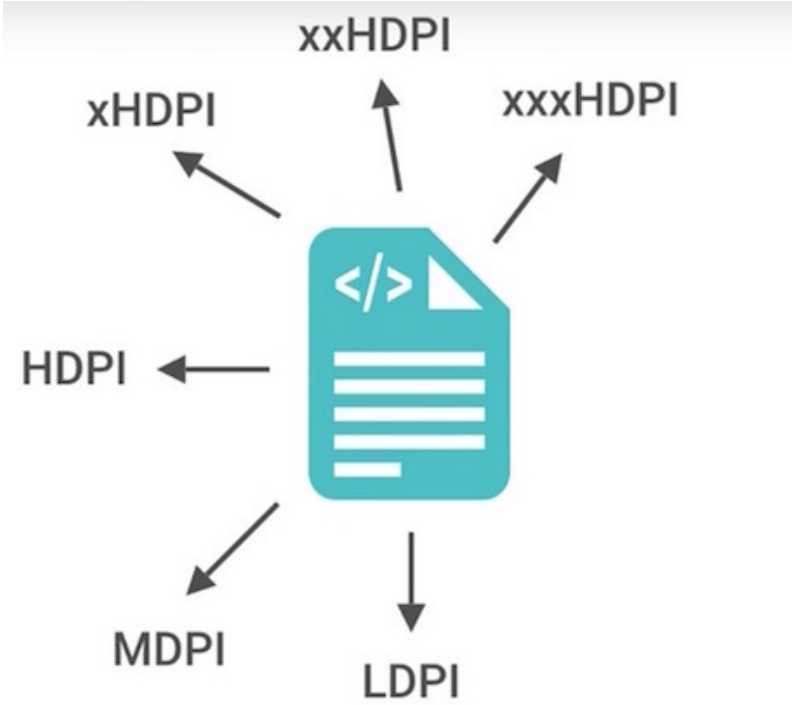
- 1）确保在build.gradle文件中开启了`minifEnabled`与`shrinkResources`的属性，这两个属性可以帮助移除那些在程序中使用不到的代码与资源，帮助减少APP的安装包大小。

```
buildTypes {
    release {
        minifyEnabled true
        shrinkResources true
        proguardFiles getDefaultProguardFile('proguard-android.t
                'proguard-rules.pro'
    }
}
```

- 2）有选择性的提供对应分辨率的图片资源，系统会自动匹配最合适分辨率的图片并执行拉伸或者压缩的处理。



- 3）在符合条件的情况下，使用Vertor Drawable替代传统的PNG/JPEG图片，能够极大的减少图片资源的大小。传统模式下，针对不同dpi的手机都需要提供一套PNG/JPEG的图片，而如果使用Vector Drawable的话，只需要一个XML文件即可。



- 4）尽量复用已经存在的资源图片，使用代码的方式对已有的资源进行复用，如下图所示：


```
<?xml version="1.0" encoding="utf-8"?>
<rotate xmlns:android="http://schemas.android.com/apk/res/
android"
    android:drawable="@drawable/ic_arrow_expand"
    android:fromDegrees="0"
    android:toDegrees="270" />
```

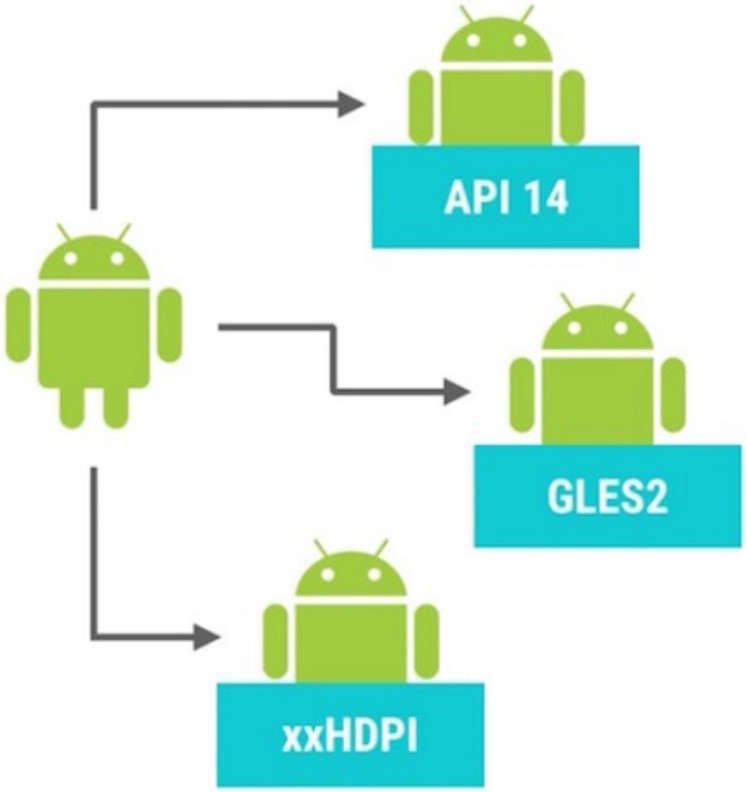
以上几点虽然看起来都微不足道，但是真正执行之后，能够显著减少安装包的资源图片大小。

减少程序的代码量

- 1）开启MinifEnabled，Proguard。打开这些编译属性之后，程序在打包的时候就不会把没有引用到的代码编译进来，以此达到减少安装包大小的目的。
- 2）注意因为编译行为额外产生的方法数，例如类似Enum，Protocal Buffer可能导致方法数与类的个数增加。
- 3）部分引入到工程中的jar类库可能并不是专门针对移动端APP而设计的，他们最开始可能是运用在PC或者Server上的。使用这些类库不仅仅额外增加了包的大小，还增加了编译时间。单纯依靠Proguard可能无法完全移除那些使用不到的方法，最佳的方式是使用一些更加轻量化，专门为Android APP设计的jar类库。

安装包的拆分

设想一下，一个low dpi，API<14的用户手机下载安装的APK里面却包含了大量xxhdpi的资源文件，对于这个用户来说，这个APK是存在很大的资源浪费的。幸好Android平台为我们提供了拆分APK的方法，它能够根据API Level，屏幕大小以及GPU版本的不同进行拆分，使得对应平台的用户下载到最合适自己手机的安装包。

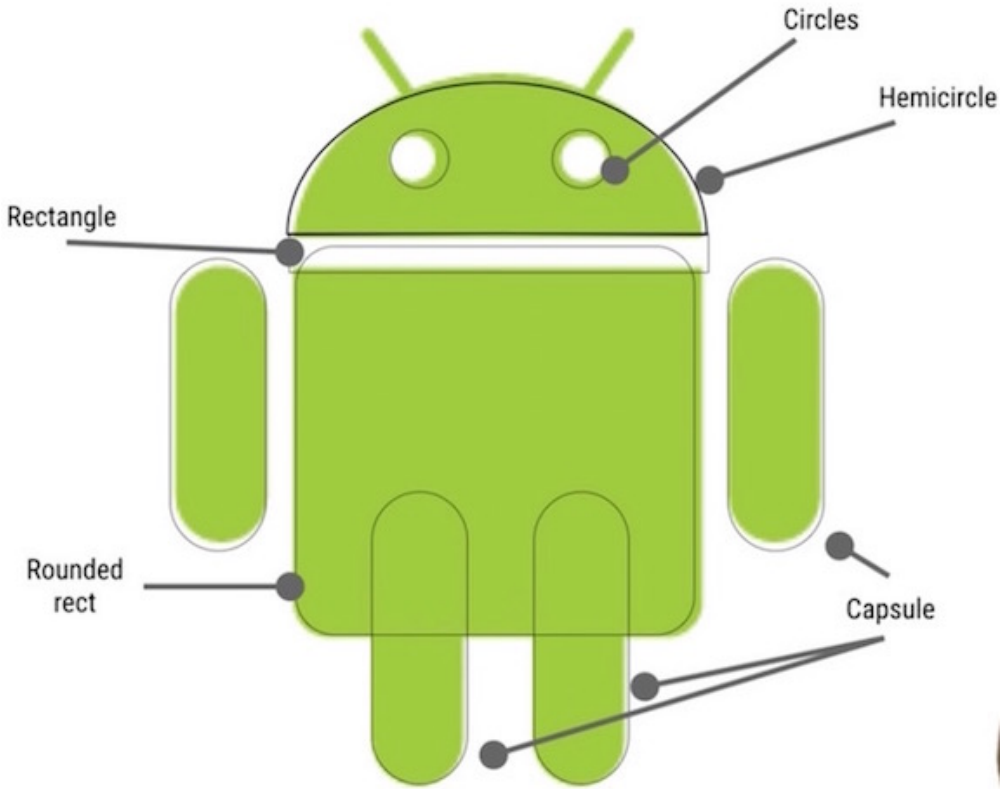


更多关于安装包拆分的信息，请查看Configure APK Splits与Maintaining Multiple APKs(由于国内应用分发市场的现状，这一条几乎没有办法执行)。

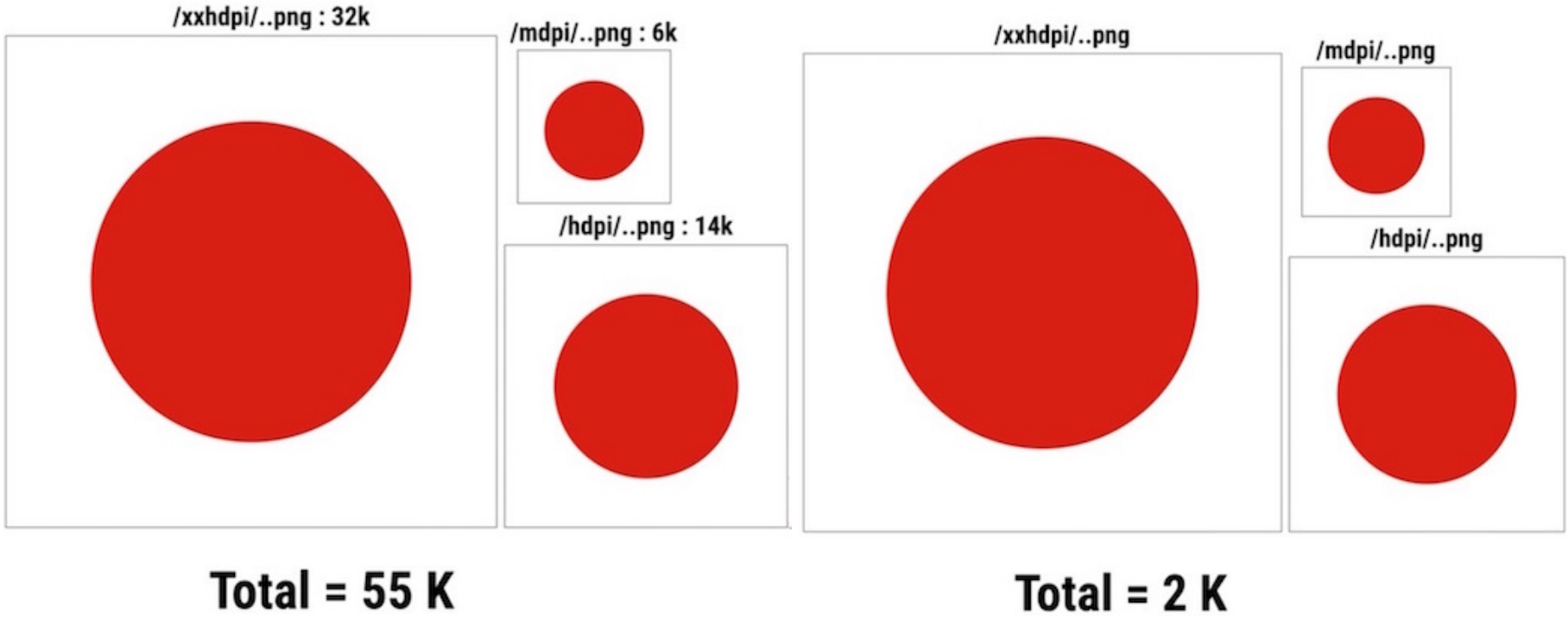
6）VectorDrawable for smaller APKs

针对不同的分辨率提供多张精度的图片会额外增加APK的大小，针对这个问题的解决方案是考虑使用VectorDrawable，它仅仅只需要一个文件，能够动态生成对应分辨率的图片。

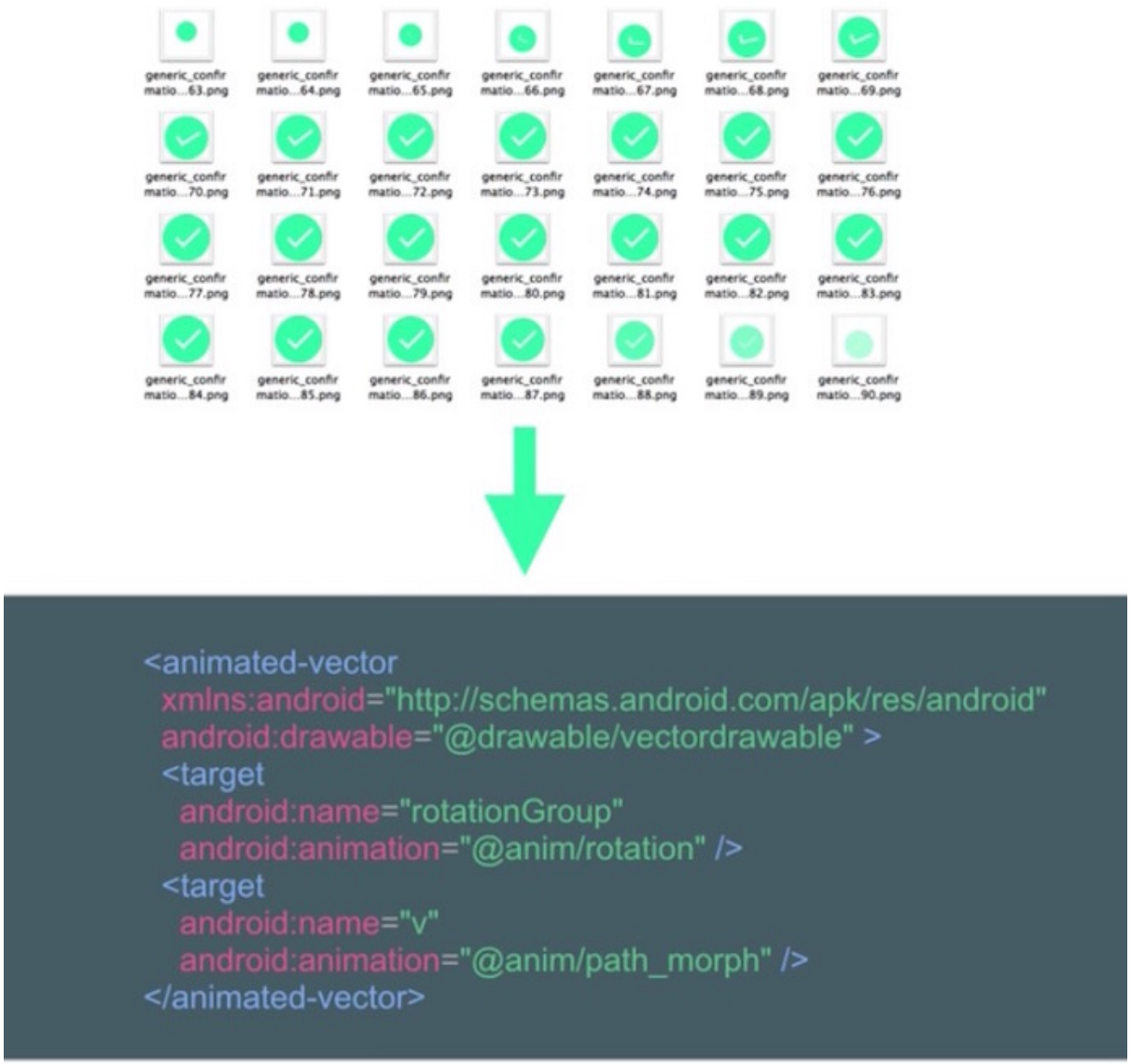
VectorDrawable通过XML文件描述图片的形状，大小，样式。



通过这种方式，我们可以显著减少图片资源对安装包大小的影响。

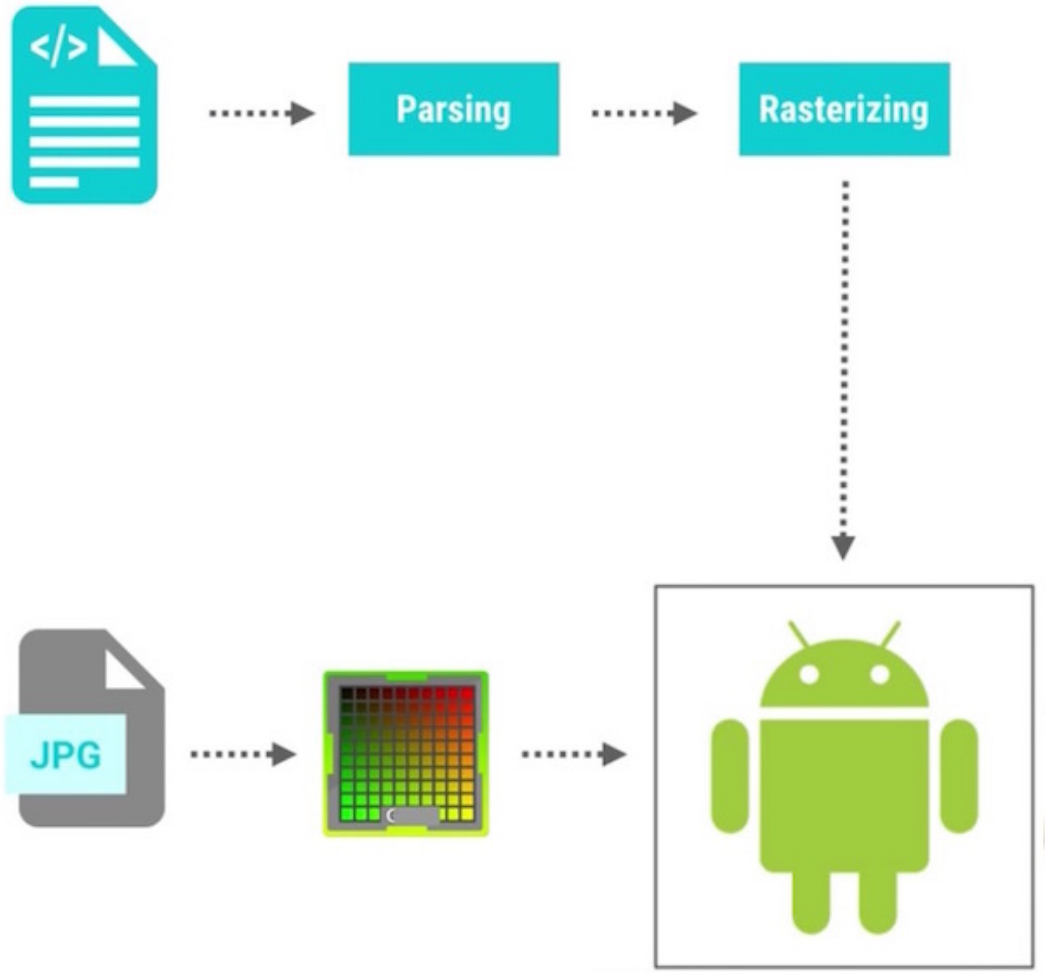


使用VectorDrawable还可以避免因为使用帧动画导致的图片资源过多的情况，如下图所示



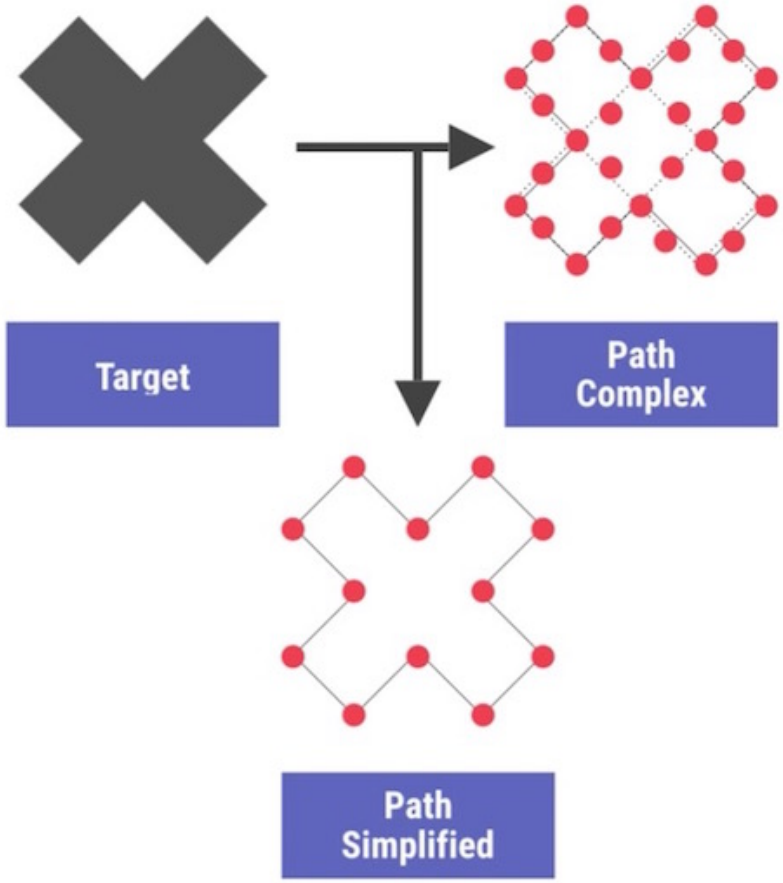
前面介绍了VectorDrawable(VD)的优势，但是在使用VectorDrawable的时候，还是有以下的问题需要特别注意的?

- 首先VD的加载有异于JPEG/PNG文件，图片文件可以依靠硬件进行纹理的渲染，而VD文件需要先进行加载解析，然后才能够进行纹理的渲染。



- 其次VD文件适用于简单有规则的图片渲染，不适用于那些纹理过于复杂的图片，这样不仅仅会过度增加描述文件的复杂度还可能无法获取到想要的渲染效果。

- 最后VD文件中关于Path的描述需要尽量简化，复杂冗余的Path信息不仅对得到想要的图片没有益处，还增加了加载渲染的难度。



首发于CSDN：Android性能优化典范（六）

Posted by HuKai Oct 4th, 2016 Android, Android:Performance

« Google I/O 2016随笔 Android相机开发 – 1)基础概览篇 »

Comments