

缘起

1. 解答App Team 同事关心的问题。
2. 对Android System App提出一些要求。

概述

(一) Android 权限问题

(二) 电池功耗

(三) 网络

(四) APK安装包瘦身

(五) App 启动时间

(六) UI渲染

Android 权限问题

Android 11 中的权限更新


在 Android 11 中，用户能够针对位置信息、麦克风和摄像头指定更精细的权限。此外，如果以 Android 11 或更高版本为目标平台的应用在一段时间内未使用，系统就会重置这些应用的权限。如果应用使用系统提醒窗口或读取与电话号码相关的信息，可能需要更新它们声明的权限。

- 单次授权 从 Android 11 开始，每当应用请求与位置信息、麦克风或摄像头相关的权限时，面向用户的权限对话框会包含仅限这一次选项。如果用户在对话框中选择此选项，系统会向应用授予临时的单次授权。
- 自动重置未使用的应用的权限

如果应用以 Android 11 或更高版本为目标平台并且数月未使用，系统会通过自动重置用户已授予应用的运行时敏感权限来保护用户数据。此操作与用户在系统设置中查看权限并将应用的访问权限级别更改为拒绝的做法效果一样。如果应用遵循了有关在运行时请求权限的最佳做法，那么您不必对应用进行任何更改。这是因为，当用户与应用中的功能互动时，您应该会验证相关功能是否具有所需权限。

详细了解 [Android 11 中的权限更新](#)

权限分类

- **normal 级别**。权限保护级别的默认值，无须用户确认，只要声明了，就自动默默授权。如：ACCESS_NETWORK_STATE。
- **dangerous 级别**。赋予权限前，会弹出对话框，显式请求权限。如：READ_SMS。因为 Android 需要在安装时赋予权限，所以安装的确认对话框，也会显示列出权限清单。  (image/screen.png 1)
- **signature 级别**。signature 级别的权限是最严格的权限，只会赋予与声明权限使用相同证书的应用程序。以系统内置 signature 级别权限为例，Android 系统应用的签名由平台密钥签发，默认情况下源码树里有 4 个不同的密钥文件：platform、shared、media 和 testkey。
 - 所有核心平台的包（如：设置、电话、蓝牙）均使用 platform 密钥签发；
 - 搜索和通讯录相关的包使用 shared 签发；
 - 图库和媒体相关的包使用 media 密钥签发
 - 其他的应用使用 testkey 签发。定义系统内置权限的 framework-res.apk 文件是使用平台密钥签发的，因此任何试图请求 signature 级别内置权限的应用程序，需要使用与框架资源包相同的密钥进行签名。
- **signatureOrSystem 级别**。可以看做是一种折中的级别，可被赋予与声明权限具有相同签名证书密钥的应用程序（同 signature 级别）或者系统镜像的部分应用，也就是说这允许厂商无须共享签名密钥。
- **Special permissions**

如SYSTEM_ALERT_WINDOW需要应用发送

Settings.ACTION_MANAGE_OVERLAY_PERMISSION intent去提示用户是否开启该权限

权限列表可以参考官网：

<https://developer.android.google.cn/guide/topics/permissions/overview?hl=en>

sharedUserId

如调用隐藏方法，系统关机重启，静默安装升级卸载应用.需要用到
android.uid.system

1. 在应用程序的AndroidManifest.xml中的manifest节点中加入
android:sharedUserId="android.uid.system"这个属性。

2. 使用目标系统的platform密钥来重新给apk文件签名。

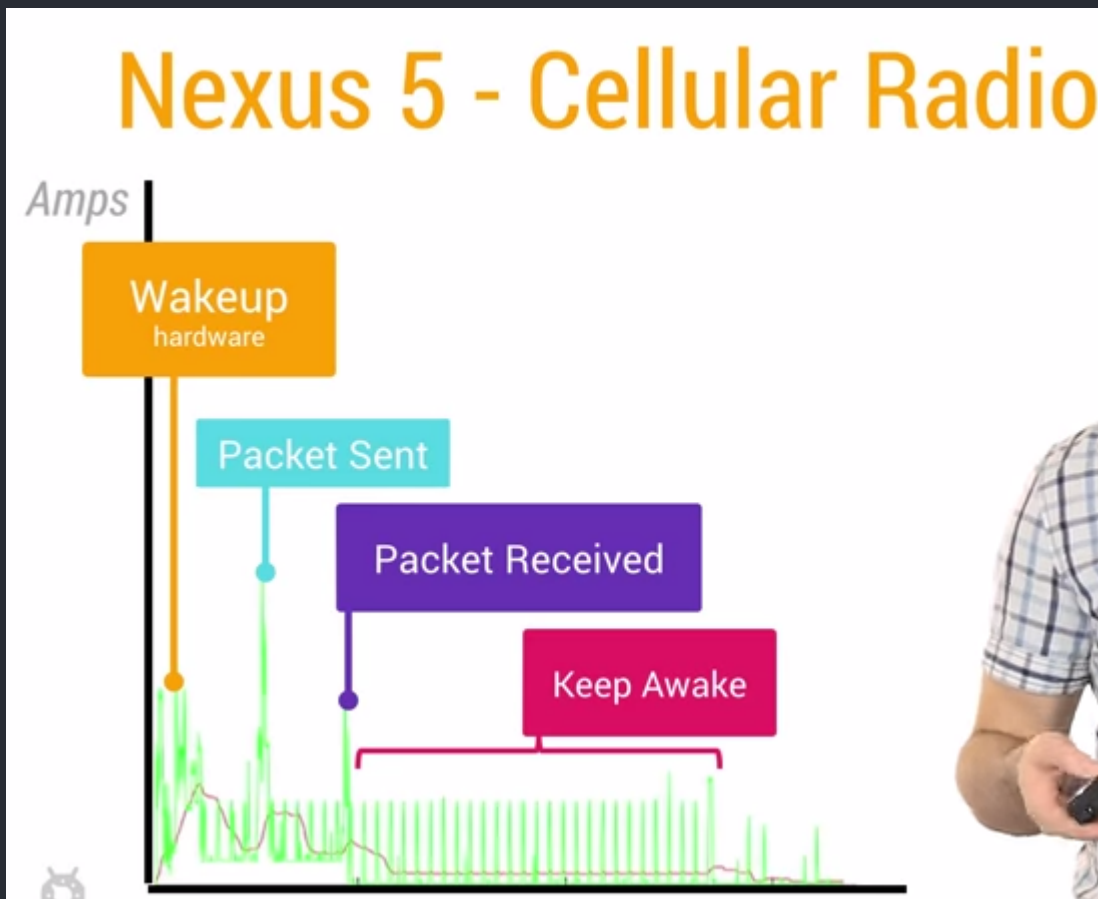
1.系统中所有使用android.uid.system作为共享UID的APK，都会首先在manifest节点中增加android:sharedUserId="android.uid.system"，然后在Android.mk中增加
LOCAL_CERTIFICATE := platform。可以参见Settings等’

2.系统中所有使用android.uid.shared作为共享UID的APK，都会在manifest节点中增加
android:sharedUserId="android.uid.shared"，然后在Android.mk中增加
LOCAL_CERTIFICATE := shared。可以参见Launcher等

3.系统中所有使用android.media作为共享UID的APK，都会在manifest节点中增加
android:sharedUserId="android.media"，然后在Android.mk中增加
LOCAL_CERTIFICATE := media。可以参见Gallery等

电池功耗

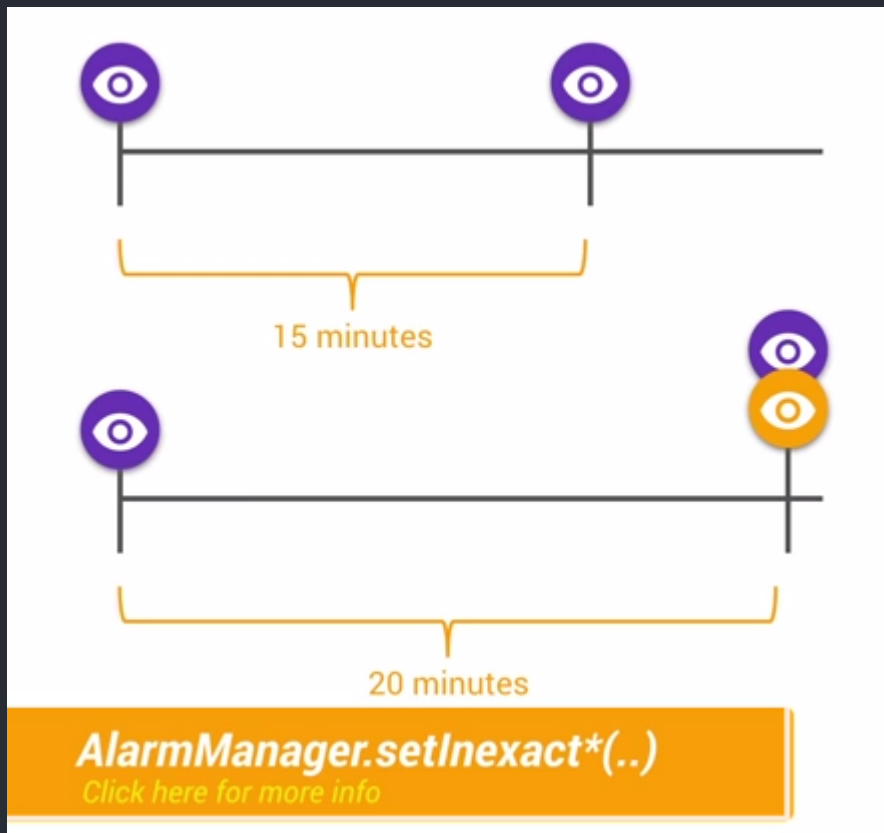
电量其实是目前手持设备最宝贵的资源之一. 对于开发者来说，电量优化是他们最后才会考虑的的事情。但是可以确定的是，千万不能让你的应用成为消耗电量的大户。



在这种睡眠状态下，大多数应用还是会尝试进行工作，他们将不断的唤醒手机。一个最简单的唤醒手机的方法是使用PowerManager.WakeLock的API来保持CPU工作并防止屏幕变暗关闭。

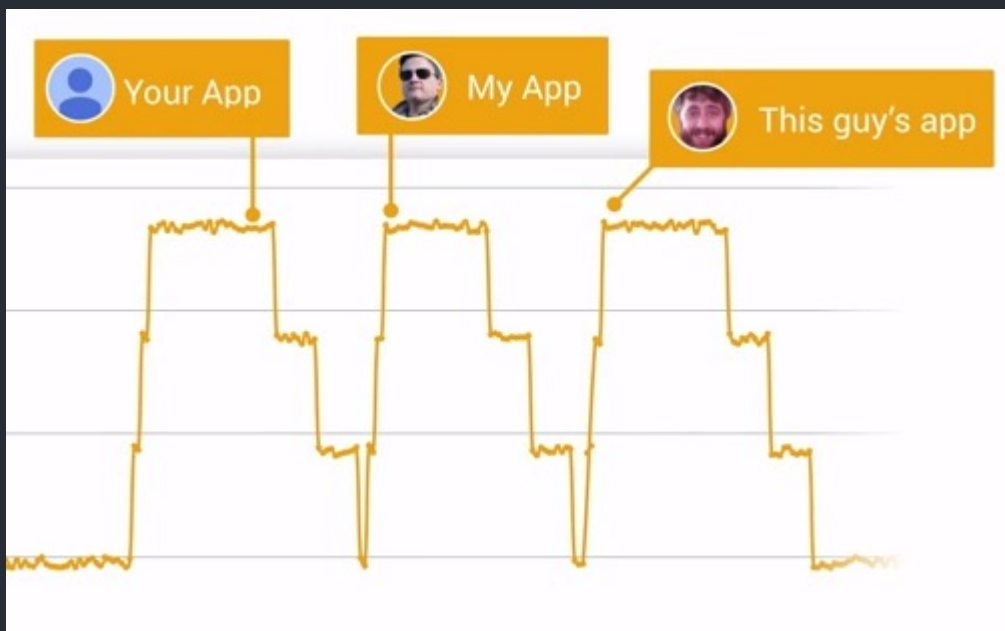
唤醒要求

- 我们应该尽量减少唤醒屏幕的次数与持续的时间，使用WakeLock来处理唤醒的问题，能够正确执行唤醒操作并根据设定及时关闭操作进入睡眠状态。
- 某些非必须马上执行的操作，例如上传歌曲，图片处理等，可以等到设备处于充电状态或者电量充足的时候才进行。
- 使用非精准定时器,例如，如果有另外一个程序需要你设定的时间晚5分钟唤醒，最好能够等到那个时候，两个任务捆绑一起同时进行。

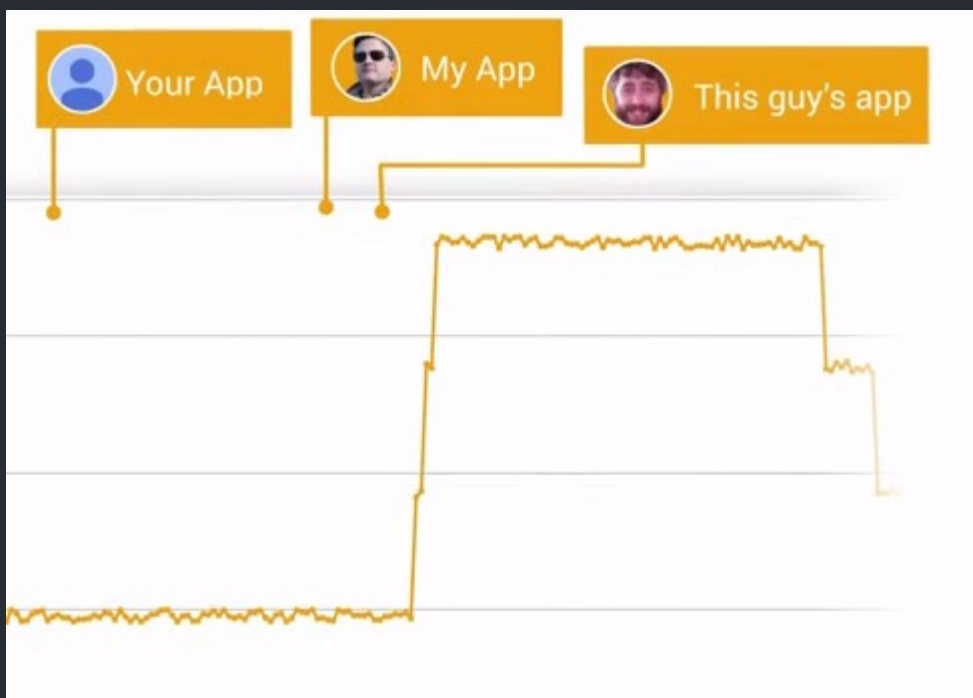


电量消耗对比

下图是每个应用程序各自执行后台任务导致的电量消耗示意图：



因为像上面那样做会导致浪费很多电量，我们需要做的是把部分应用的任务延迟处理，等到一定时机，这些任务一并进行处理。结果如下面的示意图：



延迟任务

执行延迟任务，通常有下面三种方式：

- 1)AlarmManager 使用AlarmManager设置定时任务，可以选择精确的间隔时间，也可以选择非精确时间作为参数。除非程序有很强烈的需要使用精确的定时唤醒，否者一定要避免使用他，我们应该尽量使用非精确的方式。
- 2)SyncAdapter 我们可以使用SyncAdapter为应用添加设置账户，这样在手机设置的账户列表里面可以找到我们的应用。这种方式功能更多，但是实现起来比较复杂。我们可以从这里看到官方的培训课程：
<http://developer.android.com/training/sync-adapters/index.html>
- 3)JobScheduler 这是最简单高效的方法，我们可以设置任务延迟的间隔，执行条件，还可以增加重试机制。

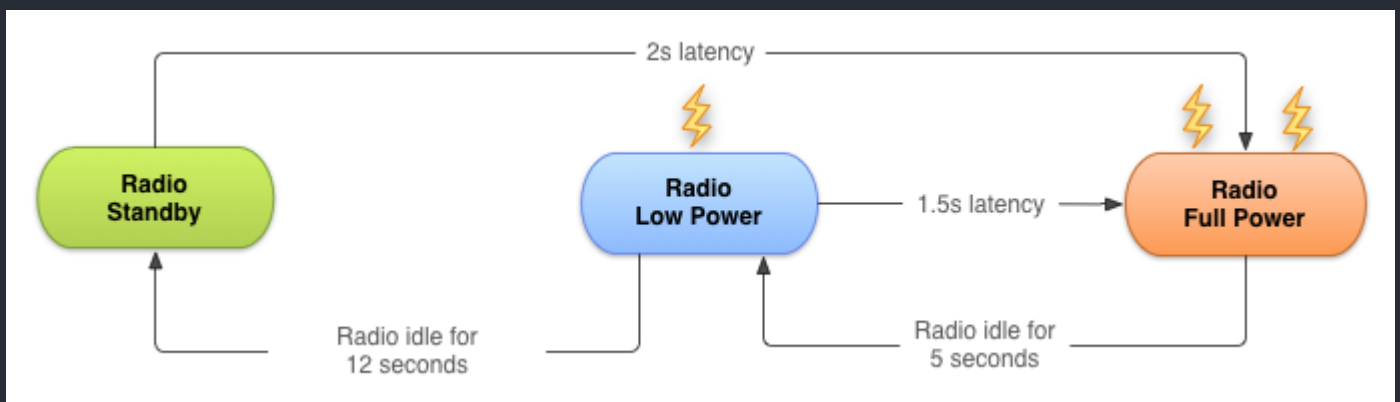
网络

无线电波状态机

使用无线电波（wireless radio）进行传输数据很可能是我们 app 最耗电的来源之一。为了最小化网络连接对电量的消耗，懂得连接模式（connectivity model）会如何影响底层的无线电硬件设备是至关重要的。

典型的 无线网络有三种能量状态：

- Full power：当无线连接被激活的时候，允许设备以最大的传输速率进行操作。
- Low power：一种中间状态，对电量的消耗差不多是 Full power 状态下的50%。
- Standby：最小的能量状态，没有被激活或者需求的网络连接。



捆绑（bundle）与预取（prefetching）

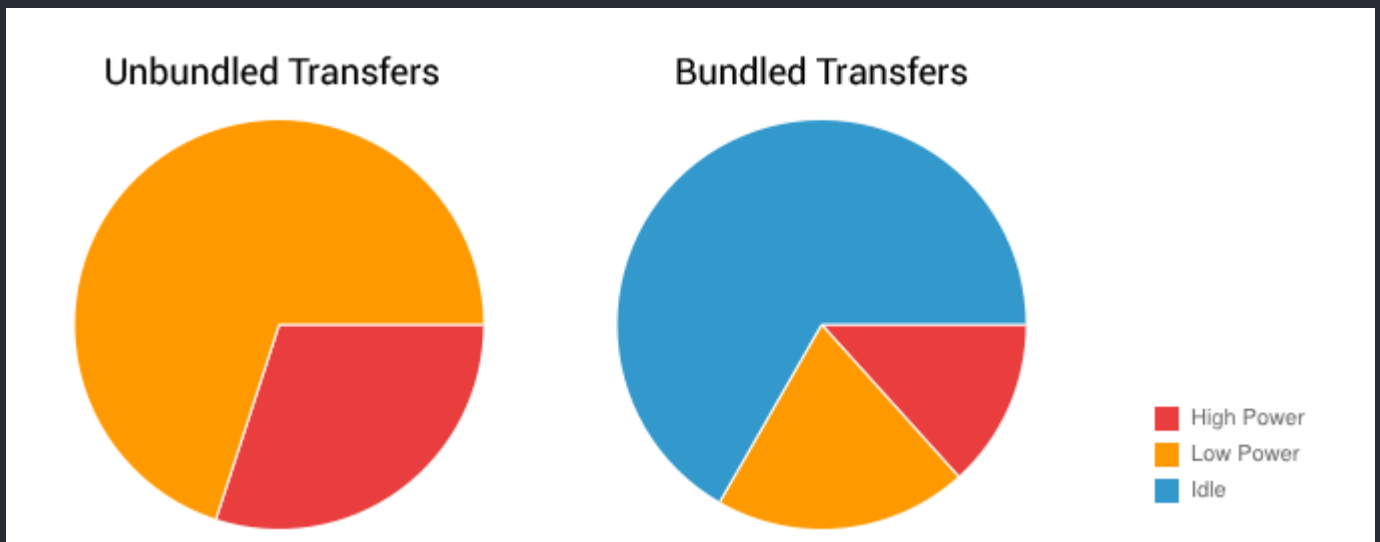
每次创建一个新的网络连接，无线电波就切换到 full power 状态。无线电波会在传输数据时保持在 full power 的状态（加上一个附加的5秒拖尾时间），再之后会经过12秒的 low power 能量状态。因此对于典型的 设备，每一次数据传输的会话都会导致无线电波消耗大概20秒时间来提取电能。

实际上，这意味着一个每18秒传输1秒数据的 app，会一直保持激活状态（18 = 1秒的传输数据 + 5秒过渡时间回到 low power + 12秒过渡时间回到standby）。因此，每分钟会消耗18秒 high power 的电量，42秒 low power 的电量。

通过比较，同一个 app，每分钟传输持续3秒的捆绑数据（bundle data），会使得无线电波持续在 high power 状态仅仅8秒，在 low power 状态仅仅12秒钟。

上面第二种传输捆绑数据（bundle data）的例子，可以看到减少了大量的电量消耗。图示如下：

电池功耗



- 预取数据

预取数据是一种减少独立数据传输会话数量的有效方法。预取技术指的是在一定时间内，单次连接操作，以最大的下载能力来下载所有用户可能需要的数据。

通过前面的传输数据的技术，减少了大量下载数据所需的无线电波激活时间。这样不仅节省了电量，也改善了延迟，降低了带宽，减少了下载时间。

另外WiFi连接下，网络传输的电量消耗要比移动网络少很多，应该尽量减少移动网络下的数据传输，多在WiFi环境下传输数据。

APK安装包瘦身

代码

使用第三方库(library)可以在不用自己编写大量代码的前提下帮助我们解决一些难题，节约大量的时间，但是这些引入的第三方库很可能会导致主程序代码臃肿冗余。如果我们因为只需要某个library的一小部分功能而把整个library都导入自己的项目，这就会引起代码臃肿。

- 使用proguard混淆代码

Android为我们提供了Proguard的工具来帮助应用程序对代码进行瘦身，优化，混淆的处理。它会帮助移除那些没有使用到的代码，还可以对类名，方法名进行混淆处理以避免程序被反编译。举个例子，Google I/O 2015这个应用使用了大量的library，没有经过Proguard处理之前编译出来的包是8.4Mb大小，经过处理之后的包仅仅是4.1Mb大小。

使用Proguard相当的简单，只需要在build.gradle文件中配置minifyEnable为true即可，如下图所示：

```
android {  
    ...  
    buildTypes {  
        release {  
            minifyEnabled true  
            proguardFiles getDefaultProguardFile('proguard-android.txt'),  
                'proguard-rules.pro'  
        }  
    }  
}
```

- 保持良好的编程习惯，不要重复或者不用的代码，谨慎添加libs，移除使用不到的libs。
- native code的部分，大多数情况下只需要支持armabi与x86的架构即可。如果非必须，可以考虑拿掉x86的部分。

资源

减少APK安装包的大小也是Android程序优化中很重要的一个方面，我们不应该给用户下载到一个臃肿的安装包。假设这样一个场景，我们引入了Google Play Service的library，是想要使用里面的Maps的功能，但是里面的登入等等其他功能是不需要的，可是这些功能相关的代码与图片资源，布局资源如果也被引入我们的项目，这样就会导致我们的程序安装包臃肿。

所幸的是，我们可以使用Gradle来帮助我们分析代码，分析引用的资源，对于那些没有被引用到的资源，会在编译阶段被排除在APK安装包之外，要实现这个功能，对我们来说仅仅只需要在build.gradle文件中配置shrinkResource为true就好了，如下图所示：

```
android {  
    ...  
  
    buildTypes {  
        release {  
            minifyEnabled true  
            shrinkResources true  
            proguardFiles  
            getDefaultProguardFile('proguard-android.txt'),  
                'proguard-rules.pro'  
        }  
    }  
}
```

- 使用Lint工具查找没有使用到的资源。去除不使用的图片，String，XML等等。

[美团外卖Android Lint代码检查实践](#)

App 启动时间

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();

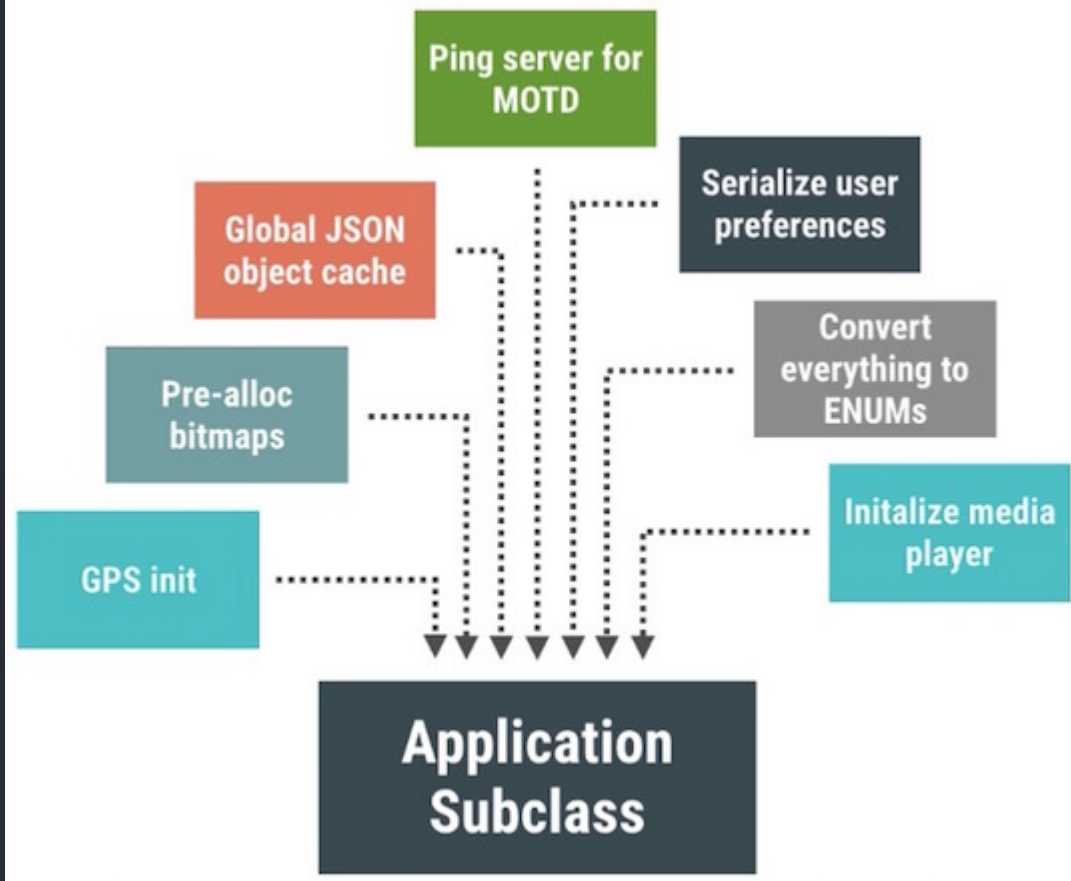
        // Read our user preferences from the last time we loaded this app.
        Global.readPreferences(this);

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

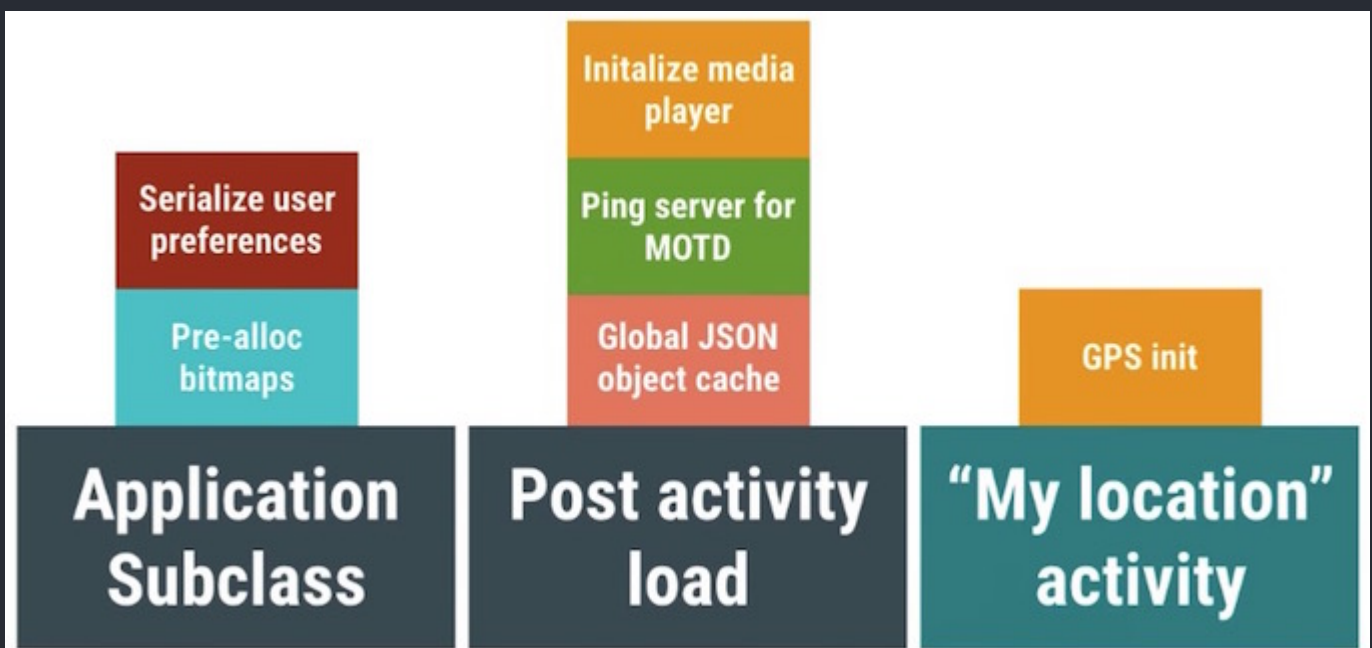
        // Go ahead and create the cache pool for bitmaps you see in the frag
        Global.initCachesAndPreallocate();

        //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别对待的，有些可以做延迟加载，有些可以放到其他地方做初始化操作，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。

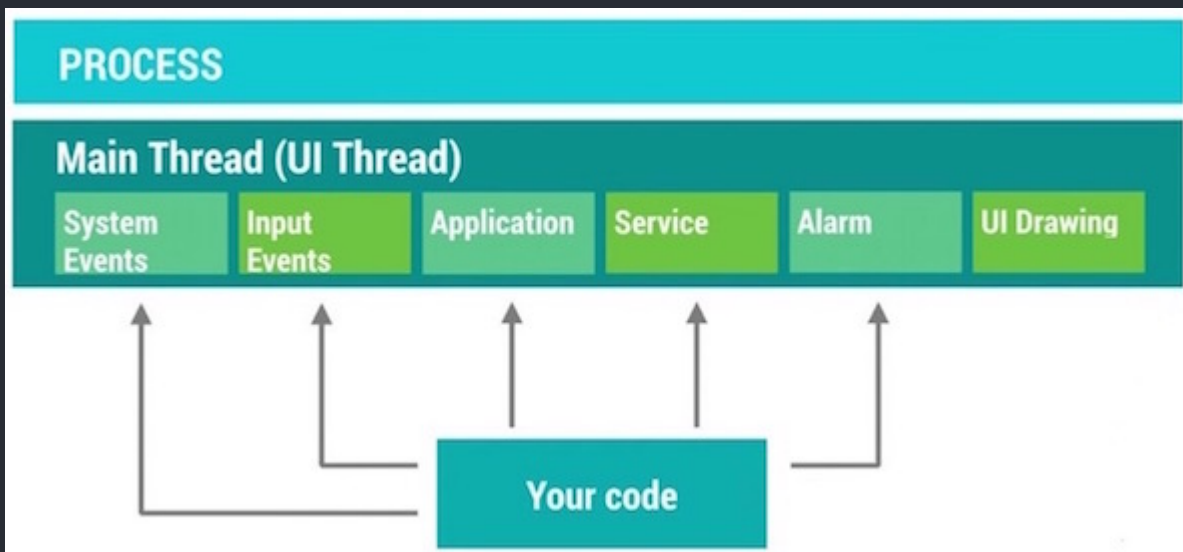


UI 渲染

Threading Performance

在程序开发的实践当中，为了让程序表现得更加流畅，我们肯定会需要使用到多线程来提升程序的并发执行性能。但是编写多线程并发的代码一直以来都是一个相对棘手的问题，所以想要获得更佳的程序性能，我们非常有必要掌握多线程并发编程的基础技能。

众所周知，Android程序的大多数代码操作都必须执行在主线程，例如系统事件(例如设备屏幕发生旋转)，输入事件(例如用户点击滑动等)，程序回调服务，UI绘制以及闹钟事件等等。那么我们在上述事件或者方法中插入的代码也将执行在主线程。



一旦我们在主线程里面添加了操作复杂的代码，这些代码就很可能阻碍主线程去响应点击/滑动事件，阻碍主线程的UI绘制等等。我们知道，为了让屏幕的刷新帧率达到60fps，我们需要确保16ms内完成单次刷新的操作。一旦我们在主线程里面执行的任务过于繁重就可能导致接收到刷新信号的时候因为资源被占用而无法完成这次刷新操作，这样就会产生掉帧的现象，刷新帧率自然也就跟着下降了

PROCESS

Main Thread (UI Thread)



16ms

16ms

16ms

Dropped frame

彩蛋

[Android 源代码文档 https://source.android.google.cn/](https://source.android.google.cn/)

[Android 开发这网址 https://developer.android.google.cn/](https://developer.android.google.cn/)

[Android性能优化](#)

[10 条提升 Android 性能的建议](#)