**Android性能优化**

# 系统优化

## 虚拟机ART 和 Dalvik

Android Runtime (ART) 是 Android 上的应用和部分系统服务使用的托管式运行时。ART 及其前身 Dalvik 最初是专为 Android 项目打造的。作为运行时的 ART 可执行 Dalvik 可执行文件并遵循 Dex 字节码规范。

ART 和 Dalvik 是运行 Dex 字节码的兼容运行时，因此针对 Dalvik 开发的应用也能在 ART 环境中运作。不过，Dalvik 采用的一些技术并不适用于 ART。

### ART 功能

以下是 ART 实现的一些主要功能。

#### 预先 (AOT) 编译

ART 推出了预先 (AOT) 编译，可提高应用的性能。ART 还具有比 Dalvik 更严格的安装时验证。

在安装时，ART 使用设备自带的 **dex2oat** 工具来编译应用。该实用工具接受 [DEX](https://source.android.com/devices/tech/dalvik/dex-format.html) 文件作为输入，并针对目标设备生成已编译的应用可执行文件。该实用工具应能够毫不费力地编译所有有效的 DEX 文件。但是，一些后处理工具会生成无效文件，Dalvik 可以接受这些文件，但 ART 无法编译这些文件。有关详情，请参阅[解决垃圾回收问题](http://developer.android.com/guide/practices/verifying-apps-art.html" \l "GC_Migration)。

#### 优化的垃圾回收

垃圾回收 (GC) 可能会损害应用的性能，从而导致显示不稳定、界面响应速度缓慢以及其他问题。ART 通过以下几种方式优化垃圾回收：

* 采用一个而非两个 GC 暂停
* 在 GC 保持暂停状态期间并行处理
* 采用总 GC 时间更短的回收器清理最近分配的短时对象这种特殊情况
* 优化了垃圾回收人机工程学，这样能够更加及时地进行并行垃圾回收，这使得 [GC\_FOR\_ALLOC](http://developer.android.com/tools/debugging/debugging-memory.html" \l "LogMessages) 事件在典型用例中极为罕见
* 压缩 GC 以减少后台内存使用和碎片

### Android 8.0 中的 ART 功能改进

在 Android 8.0 版本中，Android Runtime (ART) 有了极大改进。下面的列表总结了设备制造商可以在 ART 中获得的增强功能。

#### 并发压缩式垃圾回收器

正如 Google 在 Google I/O 大会上所宣布的那样，ART 在 Android 8.0 中提供了新的并发压缩式垃圾回收器 (GC)。该回收器会在每次执行 GC 时以及应用正在运行时对堆进行压缩，且仅在处理线程根时短暂停顿一次。该回收器具有以下优势：

* GC 始终会对堆进行压缩：堆的大小平均比 Android 7.0 中的小 32%。
* 得益于压缩，系统现可实现线程局部碰撞指针对象分配：分配速度比 Android 7.0 中的快 70%。
* H2 基准的停顿次数比 Android 7.0 GC 的少 85%。
* 停顿次数不再随堆的大小而变化，应用在使用较大的堆时也无需担心造成卡顿。
* GC 实现细节 - 读取屏障：
* 读取屏障是在读取每个对象字段时所做的少量工作。
* 它们在编译器中经过了优化，但可能会减慢某些用例的速度。

#### 循环优化

在 Android 8.0 版本中，ART 采取了多种循环优化措施，具体如下：

* 消除边界检查
* 静态：在编译时证明范围位于边界内
* 动态：运行时测试确保循环始终位于边界内（否则不进行优化）
* 消除归纳变量
* 移除无用归纳
* 用封闭式表达式替换仅在循环后使用的归纳
* 消除循环主体内的无用代码，移除整个死循环
* 强度降低
* 循环转换：逆转、交换、拆分、展开、单模等
* SIMDization（也称为矢量化）

循环优化器位于 ART 编译器中一个独立的优化环节中。大多数循环优化与其他方面的优化和简化类似。由于大多数 CFG 实用工具（请参阅 nodes.h）侧重于编译而不是重写 CFG，因此通过更复杂（与平时相比）的方式进行一些重写 CFG 的优化时，会面临挑战。

#### 类层次结构分析

在 Android 8.0 中，ART 会使用类层次结构分析 (CHA)，这是一种编译器优化，可基于通过分析类层次结构所生成的信息，将虚拟调用去虚拟化为直接调用。虚拟调用代价高昂，因为它们围绕 vtable 查找来实现，且会占用几个依赖负载。另外，虚拟调用也不能内嵌。

以下是对相关增强功能的总结：

* 动态单一实现方法状态更新 - 在类关联时间结束时，如果 vtable 已被填充，ART 会按条目对超类的 vtable 进行比较。
* 编译器优化 - 编译器会利用某种方法的单一实现信息。如果方法 A.foo 设置了单一实现标记，则编译器会将虚拟调用去虚拟化为直接调用，并借此进一步尝试内嵌直接调用。
* 已编译代码无效 - 另外，在类关联时间结束时，如果单一实现信息进行了更新，且方法 A.foo 之前拥有单一实现，但该状态现已变为无效，则依赖方法 A.foo 拥有单一实现这一假设的所有已编译代码都需要变为无效代码。
* 去优化 - 对于堆栈上已编译的有效代码，系统会启动去优化功能，以强制使已编译无效代码进入解释器模式，从而确保正确性。系统会采用结合了同步和异步去优化的全新去优化机制。

#### .oat 文件中的内嵌缓存

ART 现在采用内嵌缓存，并对有足够数据可用的调用站点进行优化。内嵌缓存功能会将额外的运行时信息记录到配置文件中，并利用这类信息将动态优化添加到预先编译中。

#### Dexlayout

Dexlayout 是在 Android 8.0 中引入的一个库，用于分析 dex 文件，并根据配置文件对其进行重新排序。Dexlayout 旨在使用运行时配置信息对 dex 文件的各个部分进行重新排序（在设备的空闲维护编译期间）。通过将经常一起访问的部分 dex 文件集中在一起，程序可以因改进文件位置而拥有更好的内存访问模式，从而节省 RAM 并缩短启动时间。

由于配置文件信息目前仅在运行应用后可用，因此系统会在空闲维护期间将 dexlayout 集成到 dex2oat 的设备编译中。

#### Dex 缓存移除

在 Android 7.0 及更早版本中，DexCache 对象拥有四个大型数组，与 DexFile 中特定元素的数量成正比，即：

* 字符串（每个 DexFile::StringId 一个引用），
* 类型（每个 DexFile::TypeId 一个引用），
* 方法（每个 DexFile::MethodId 一个原生指针），
* 字段（每个 DexFile::FieldId 一个原生指针）。

这些数组用于快速检索我们以前解析的对象。在 Android 8.0 中，除方法数组外，所有数组都已移除。

#### 解释器性能

通过引入 Mterp（一种解释器，具有以汇编语言编写的核心提取/解码/解释机制），Android 7.0 版本中的解释器性能得以显著提升。Mterp 模仿了快速 Dalvik 解释器，并支持 arm、arm64、x86、x86\_64、mips 和 mips64。对于计算代码而言，ART 的 Mterp 大致相当于 Dalvik 的快速解释器。不过，有时候，它的速度可能会显著变慢，甚至急剧变慢：

* 调用性能。
* 字符串操作和 Dalvik 中其他被视为内嵌函数的高频用户方法。
* 堆栈内存使用量较高。

Android 8.0 解决了这些问题。

#### 详细了解内嵌

从 Android 6.0 开始，ART 可以内嵌同一个 dex 文件中的任何调用，但只能内嵌来自其他 dex 文件的叶方法。此项限制具有以下两个原因：

1. 从其他 dex 文件进行内嵌需要使用该 dex 文件的 dex 缓存，这与同一 dex 文件内嵌（只能重复使用调用程序的 dex 缓存）有所不同。已编译代码中需要具有 dex 缓存，以便执行一系列指令，例如静态调用、字符串加载或类加载。
2. 堆栈映射只对当前 dex 文件中的方法索引进行编码。

为了应对这些限制，Android 8.0 做出了以下改进：

1. 从已编译代码中移除 dex 缓存访问（另请参阅“Dex 缓存移除”部分）
2. 扩展堆栈映射编码。

#### 同步方面的改进

ART 团队调整了 MonitorEnter/MonitorExit 代码路径，并减少了我们对 ARMv8 上传统内存屏障的依赖，尽可能将其替换为较新的（获取/释放）指令。

#### 更快速的原生方法

可以使用 [@FastNative](https://android.googlesource.com/platform/libcore/+/master/dalvik/src/main/java/dalvik/annotation/optimization/FastNative.java) 和 [@CriticalNative](https://android.googlesource.com/platform/libcore/+/master/dalvik/src/main/java/dalvik/annotation/optimization/CriticalNative.java) 注解实现对 Java 原生接口 (JNI) 更快速的原生调用。这种内置的 ART 运行时优化加快了 JNI 转换并取代了现在已弃用的 !bang JNI 标记。注解对非原生方法没有任何影响，并且仅适用于 bootclasspath 上的平台 Java 语言代码（无 Play 商店更新）。

@FastNative 注解支持非静态方法。如果某种方法将 jobject 作为参数或返回值进行访问，请使用此注解。

利用 @CriticalNative 注解，可更快速地运行原生方法，但存在以下限制：

* 方法必须是静态的 - 没有参数、返回值或隐式 this 的对象。
* 仅将基元类型传递给原生方法。
* 原生方法在其函数定义中不使用 JNIEnv 和 jclass 参数。
* 该方法必须是使用 RegisterNatives 注册的，而不是依靠动态 JNI 链接。

**@FastNative** 和 **@CriticalNative** 注解在执行原生方法时会停用垃圾回收。请勿使用运行时间较长的方法，包括通常很快但一般不受控制的方法。

停顿垃圾回收可能会导致死锁。如果锁尚未得到本地释放（即尚未返回受管理代码），请勿在原生快速调用期间获取锁。此要求不适用于常规的 JNI 调用，因为 ART 将正执行的原生代码视为已暂停的状态。

@FastNative 可以使原生方法的性能提升高达 3 倍，@CriticalNative 则可以提升高达 5 倍。例如，在 Nexus 6P 设备上测量的 JNI 转换如下：

|  |  |
| --- | --- |
| Java 原生接口 (JNI) 调用 | 执行时间（以纳秒计） |
| 常规 JNI | 115 |
| !bang JNI | 60 |
| @FastNative | 35 |
| @CriticalNative | 25 |

## 低内存配置

Android 4.4 对低内存设备进行了优化

Android 现支持内存为 512MB 的设备。本文档旨在帮助 OEM 优化和配置 Android 4.4，使其能够在低内存设备上运行。在下文所述的优化措施中，有几项非常通用，甚至也可应用于以前的版本。

### Android 4.4 平台优化点

#### 改善了内存管理

* 采用了经验证可节省内存的内核配置：交换到 ZRAM。
* 终止那些即将被取消缓存且过大的缓存进程。
* 不允许大型服务自行返回至 A 服务状态（以免导致启动器终止）。
* 终止那些处于空闲维护状态中的过大进程（甚至终止当前 IME 等通常不可终止的进程）。
* 对后台服务的启动进行序列化。
* 优化了低内存设备的内存使用方式：采用更严格的内存不足 (OOM) 调整级别、缩减图形缓存大小，等等。

#### 减少了系统内存占用

* 删减了 system\_server 和 SystemUI 进程（节省了几兆的内存）。
* 在 Dalvik 中预加载 dex 缓存（节省了几兆的内存）。
* 采用了经验证的 JIT-off 选项（每个进程最多可节省 1.5MB 的内存）。
* 减少了各进程的字体缓存开销。
* 引入了占用内存更少的 ArrayMap/ArraySet，并在框架中广泛地使用其来替代 HashMap/HashSet。

#### Procstats

新增了一个开发者选项，以显示内存状态和应用内存使用情况（按照运行频率和所耗内存量排序）。

#### 内存跟踪

新的 memtrack HAL 可以跟踪图形内存分配情况、dumpsys meminfo 中的更多信息，以及 meminfo 中的阐明性总结（例如，所报告的可用内存包括缓存进程占用的内存，这样 OEM 就不会搞错优化对象）。

### 编译时配置

#### 启用低内存设备标志

我们引入了 ActivityManager.isLowRamDevice() 这个新 API，以便应用确定是否应关闭在低内存设备上表现非常差的某些内存密集型功能。

对于内存为 512MB 的设备，该 API 应返回 true。可以通过在设备 makefile 中使用以下系统属性来启用该 API：

PRODUCT\_PROPERTY\_OVERRIDES += ro.config.low\_ram=true

#### 启动器配置

请务必确保启动器的默认壁纸设置**未**使用动态壁纸。低内存设备不应预装任何动态壁纸。

### 内核配置

#### 优化内核/ActivityManager 以减少直接回收

当进程或内核尝试分配（直接分配或因新页面中存在故障而分配）内存页面并且内核已用尽所有可用内存时，就会发生直接回收。在这种情况下，内核需要释放一个页面，并在此过程中阻断分配操作。而这通常又需要磁盘 I/O 清理一个有文件支持的脏页或等待 lowmemorykiller 终止一个进程。最终可能会导致任意线程（包括界面线程）中出现额外 I/O。

为避免出现直接回收，内核已配有可触发 kswapd 或后台回收的水印。此线程会尝试释放页面，以便下次分配的真实线程能够快速顺利启动。

用于触发后台回收的默认阈值相当低 - 在 2GB 设备上约为 2MB，在 512MB 设备上约为 636KB。而且，内核通过后台回收仅能回收几兆的内存。这意味着，任何快速分配超过几 MB 内容的进程都会快速导致直接回收。

在 android-3.4 内核分支中，我们通过补丁程序 92189d47f66c67e5fd92eafaa287e153197a454f（“添加用于扩展可用内存空间的可调选项”）添加了对新内核可调选项的支持。如果您选择将该补丁程序添加到设备内核中，ActivityManager 会告知内核尝试保留能容纳 3 个全屏 32 bpp 缓冲区的可用内存空间。

这些阈值可通过框架 config.xml 进行配置

<!-- Device configuration setting the /proc/sys/vm/extra\_free\_kbytes tunable

in the kernel (if it exists). A high value will increase the amount of memory

that the kernel tries to keep free, reducing allocation time and causing the

lowmemorykiller to kill earlier. A low value allows more memory to be used by

processes but may cause more allocations to block waiting on disk I/O or

lowmemorykiller. Overrides the default value chosen by ActivityManager based

on screen size. 0 prevents keeping any extra memory over what the kernel keeps

by default. -1 keeps the default. -->

<integer name="config\_extraFreeKbytesAbsolute">-1</integer>

<!-- Device configuration adjusting the /proc/sys/vm/extra\_free\_kbytes

tunable in the kernel (if it exists). 0 uses the default value chosen by

ActivityManager. A positive value will increase the amount of memory that the

kernel tries to keep free, reducing allocation time and causing the

lowmemorykiller to kill earlier. A negative value allows more memory to be

used by processes but may cause more allocations to block waiting on disk I/O

or lowmemorykiller. Directly added to the default value chosen by

ActivityManager based on screen size. -->

<integer name="config\_extraFreeKbytesAdjust">0</integer>

#### 优化 LowMemoryKiller

ActivityManager 可配置 LowMemoryKiller 的阈值，使其符合它对在每个优先级分段中运行进程时所需的文件支持页面（缓存页面）工作集的预期。如果设备对工作集有很高的要求（例如：如果供应商界面需要更多内存，或者如果添加了更多服务），则可增大阈值。

如果为文件支持页面预留了太多内存，则可减小阈值，以便系统能够在因缓存变得过小而导致磁盘超负荷之前就终止后台进程。

<!-- Device configuration setting the minfree tunable in the lowmemorykiller

in the kernel. A high value will cause the lowmemorykiller to fire earlier,

keeping more memory in the file cache and preventing I/O thrashing, but

allowing fewer processes to stay in memory. A low value will keep more

processes in memory but may cause thrashing if set too low. Overrides the

default value chosen by ActivityManager based on screen size and total memory

for the largest lowmemorykiller bucket, and scaled proportionally to the

smaller buckets. -1 keeps the default. -->

<integer name="config\_lowMemoryKillerMinFreeKbytesAbsolute">-1</integer>

<!-- Device configuration adjusting the minfree tunable in the

lowmemorykiller in the kernel. A high value will cause the lowmemorykiller to

fire earlier, keeping more memory in the file cache and preventing I/O

thrashing, but allowing fewer processes to stay in memory. A low value will

keep more processes in memory but may cause thrashing if set too low. Directly

added to the default value chosen by ActivityManager based on screen

size and total memory for the largest lowmemorykiller bucket, and scaled

proportionally to the smaller buckets. 0 keeps the default. -->

<integer name="config\_lowMemoryKillerMinFreeKbytesAdjust">0</integer>

#### 交换到 zRAM

zRAM 交换可通过压缩内存页面并将其放入动态分配的内存交换区来增加系统中的可用内存量。

由于这是以牺牲 CPU 时间为代价来增加少量内存，因此您应仔细权衡 zRAM 交换会对您系统的性能造成的负面影响。

Android 会在多个层面上处理 zRAM 交换：

* 首先，必须启用以下内核选项，才能有效地使用zRAM 交换：
* CONFIG\_SWAP
* CONFIG\_CGROUP\_MEM\_RES\_CTLR
* CONFIG\_CGROUP\_MEM\_RES\_CTLR\_SWAP
* CONFIG\_ZRAM
* 然后，您应将一行与下列类似的内容添加到 fstab 中：

/dev/block/zram0 none swap defaults zramsize=<size in bytes>,swapprio=<swap partition priority>

* zramsize 是必要内容，表示您希望 zram 区域占用多少未压缩内存。压缩比通常介于 30-50% 之间。
* swapprio 是可选内容；如果您没有多个交换区，则无需使用此项。

您还应确保在特定于设备的 [sepolicy/file\_contexts](https://source.android.com/security/selinux/implement.html) 中将关联的块设备标记为 swap\_block\_device，以便 SELinux 适当地对其进行处理。

/dev/block/zram0 u:object\_r:swap\_block\_device:s0

* 默认情况下，Linux 内核每次会换入 8 页内存。当使用 ZRAM 时，每次读取 1 页内存产生的增量开销微乎其微，因此在设备承受着极大的内存压力时可能有所助益。要想每次只读取 1 页内存，请将以下内容添加到 init.rc 中：

write /proc/sys/vm/page-cluster 0

* 在 init.rc 中的 mount\_all /fstab.X 行后面，添加以下内容：

swapon\_all /fstab.X

* 如果在内核中启用了此功能，系统便会在启动时自动配置内存 cgroup。
* 如果内存 cgroup 可用，ActivityManager 就会将优先级较低的线程标为比其他线程更适合交换。在需要内存时，Android 内核会开始将内存页面迁移到 zRAM 交换区，并会优先处理那些已被 ActivityManager 标记的内存页面。

#### Carveout、Ion 和连续内存分配 (CMA)

对于低内存设备，需要特别注意 carveout，尤其是不会一直得到充分利用的 carveout，例如用于安全地播放视频的 carveout。有几种解决方案可最大限度地减小 carveout 区域的影响，具体取决于硬件的确切要求。

如果硬件允许不连续的内存分配，则可利用 Ion 系统堆从系统内存中分配内存，这样便无需使用 carveout。它还会尝试增大分配的内存空间以消除外围设备上的 TLB 压力。如果内存区域必须连续或必须限定在某个特定地址范围内，则可以使用连续内存分配器 (CMA)。

以这种方式创建的 carveout 也可供系统用于处理可移动页面。当需要该区域时，可移动页面就会从中移出，以便系统将处于空闲状态的大型 carveout 用于其他目的。您可以直接使用 CMA，也可以借助 Ion（通过使用 Ion CMA 堆）更轻松地使用 CMA。

## CUP频率

## GPU优化

## I/O速度优化

## 存储优化

Android Automotive 内部存储设备使用了支持上千次清空/写入周期的嵌入式多媒体卡 (eMMC)；如果 eMMC 发生故障，系统可能会变得无法使用。由于汽车的使用寿命较长（通常为 10 年以上），因此 eMMC 必须非常稳定可靠。本页面介绍了 eMMC 行为，以及 OEM 如何降低 eMMC 发生故障的风险（从而避免 Android Automotive 系统发生故障）。

### eMMC 行为

eMMC 设备使用耗损均衡技术来解决清空/写入限制的问题，具体方法是在整个系统内均匀地放置数据和分配写入操作（避免单个块因密集写入而发生故障）。eMMC 的预计使用寿命取决于以下因素：

* **写入量**。在手机上，每天写入内部存储设备的数据量可能会超过 10GB。在 Android Automotive 实现中，由于应用数量有限，我们无法提供有关会有多少数据写入的真实数据。不过，据我们观察，当用户流式传输高品质音乐和使用导航时，每分钟写入 eMMC 的数据有 50MB。未来，可能会出现其他类型的写入密集型应用，例如持续录制和存储视频的中控台摄像头应用。另外，有些汽车是每天都会有几个小时处于行使状态的共享车辆。由于这些原因及其他原因，我们预计 Android Automotive 实现会面临比手机更多的 eMMC 写入量。
* **写入模式**。写入和清空操作是在块中进行的。 与不太频繁地在较大的区块中写入数据相比，在较小的区块中频繁地写入相同的数据量会更快地磨损 eMMC。
* **eMMC 的可用空间**。较大的存储空间意味着耗损均衡算法可以将写入分摊到较多的块中。
* 耗损均衡技术。
* **环境因素**。例如：工作温度范围通常介于 -20 和 85 摄氏度之间；如果温度超出此范围，eMMC 的使用寿命会进一步缩短。

对于具有 16GB 可用空间和 3000 次清空/写入周期的 eMMC，我们估算的数据如下：

|  |  |  |
| --- | --- | --- |
| 日写入量 | 16GB | 32GB |
| 预计使用寿命 | 10 年 | 5 年 |

不过，随着可用存储空间的减少，系统可能会早在 eMMc 彻底磨损之前就停止正常运行，并且根据所使用的均衡技术和写入模式，eMMC 的使用寿命甚至可能会更短。此外，这方面的估算并没有将行为异常的应用或恶意应用的影响考虑在内，这些应用可能会在没有特殊权限的情况下将大量垃圾数据写入 eMMC 来攻击 Automotive 系统。

要在潜在的 eMMC 故障实际发生之前就检测出来，应将适当的存储设备运行状况监控功能整合到整体系统运行状况监控功能中。

### Android O实现

借助 Android O 支持的功能，OEM 可以保护和监控 Android Automotive 的内部存储设备，并延长其使用寿命。

#### 限制第三方应用

为了保护 Android Automotive 系统的内部存储设备，Android O 支持 OEM 配置是否允许第三方应用安装在内部存储设备中（应用只能向其安装所在的分区写入数据）。要进行配置，请在资源叠加层中设置以下配置：

<bool name="config\_allow3rdPartyAppOnInternal">false</bool>

#### 减少闪存磨损

如果 OEM 担心内部存储设备上的闪存磨损问题，也可以增加 SD 卡，SD 卡支持足够快的数据写入速度，可用作合并存储设备。SD 卡具有以下行为：

* 合并时，SD 卡会被加密，可以安全存储应用数据。
* SD 卡槽必须位于安全的位置（不要让用户频繁移除 SD 卡）。
* SD 卡不能用于在 Automotive 系统和计算机之间传输数据。
* 弹出 SD 卡不会影响正在运行的系统。不过，除非需要更换，否则不应移除 SD 卡。

要确保第二方应用（由汽车应用开发者构建的应用）可以安装在 SD 卡上（如果汽车提供授权的话），汽车应用开发者必须将 android:[installLocation](https://developer.android.com/guide/topics/manifest/manifest-element.html" \l "install)=["auto" | "preferExternal"] 纳入应用的清单文件中。

如果汽车禁止将第三方应用安装在内部存储设备上（如[限制第三方应用](https://source.android.com/devices/tech/perf/flash-wear" \l "restricting-third-party-apps)中所述），则在没有上述标记（或者如果配置了 [installLocation](https://developer.android.com/guide/topics/manifest/manifest-element.html" \l "install)=internalOnly 设置）的情况下，应用安装将会失败。

#### 使用 storaged 获取磁盘指标

Android O 中引入了 storaged，这是一个新的系统服务，会对磁盘指标和 eMMc 指标进行采样并发布，例如有关总体磁盘使用情况、估算的 eMMC 使用寿命和磁盘 I/O 统计数据（按应用统计）的信息。当内部存储设备开始出现故障或特定应用执行的磁盘 I/O 过多时，OEM 可以使用这些信息来向用户发出警告。有关详情，请参阅[实现 storaged](https://source.android.com/devices/tech/debug/storaged)。

|  |  |
| --- | --- |
| 组件 | 改进 |
| 引导加载程序 | * 通过移除 UART 日志节省了 1.6 秒 * 通过从 GZIP 更改为 LZ4 节省了 0.4 秒 |
| 设备内核 | * 通过移除不使用的内核配置和减少驱动程序大小节省了 0.3 秒 * 通过 dm-verity 预提取优化节省了 0.3 秒 * 通过移除驱动程序中不必要的等待/测试，节省了 0.15 秒 * 通过移除 CONFIG\_CC\_OPTIMIZE\_FOR\_SIZE，节省了 0.12 秒 |
| I/O 调整 | * 正常启动时间节省了 2 秒 * 首次启动时间节省了 25 秒 |
| init.\*.rc | * 通过并行运行 init 命令节省了 1.5 秒 * 通过及早启动 zygote 节省了 0.25 秒 * 通过 cpuset 调整节省了 0.22 秒 |
| 启动动画 | * 在未触发 fsck 的情况下，启动动画的开始时间提前了 2 秒，而触发 fsck 时启动动画则大得多 * 通过立即关闭启动动画在 Pixel XL 上节省了 5 秒 |
| SELinux 政策 | 通过 genfscon 节省了 0.2 秒 |

#### 验证

此功能在 PackageManager 测试中进行测试。

## 电耗管理

## 开机速度优化（原生Android 8.0）

<https://source.android.com/devices/tech/perf/boot-times>

启动时间是系统性能的重要组成部分，因为用户必须等待启动完成后才能使用设备。对于较常进行冷启动的汽车等设备而言，较短的启动时间至关重要（没有人喜欢在等待几十秒后才能输入导航目的地）。

Android 8.0 支持一系列组件的多项改进，因而可以缩短启动时间。下表对这些性能改进（在 Google Pixel 和 Pixel XL 设备上测得）进行了总结。供参考。

### 优化引导加载程序

要优化引导加载程序以缩短启动时间，请遵循以下做法：

* 对于日志记录：
* 停止向 UART 写入日志，因为如果日志记录很多，则可能需要很长时间来处理。（在 Google Pixel 设备上，我们发现这会使引导加载程序的速度减慢 1.5 秒）。
* 仅记录错误情况，并考虑将其他信息存储到具有单独检索机制的内存中。
* 对于内核解压缩，请考虑为当代硬件使用 LZ4 而非 GZIP（例如[补丁程序](https://patchwork.kernel.org/patch/6810841/)）。请注意，不同的内核压缩选项具有不同的加载和解压缩时间，对于特定硬件，某些选项可能比其他选项更适合。
* 检查进入去抖动/特殊模式过程中是否有不必要的等待时间，并最大限度地减少此类时间。
* 将在引导加载程序中花费的启动时间以命令行的形式传递到内核。
* 检查 CPU 时钟并考虑内核加载和初始化 I/O 并行进行（需要多核支持）。

### 优化内核

请按照以下提示优化内核以缩短启动时间。

#### 最大限度地减少设备 defconfig

最大限度地减少内核配置可以减小内核大小，从而更快速地进行加载、解压缩、初始化并缩小受攻击面。要优化设备 defconfig，请执行以下操作：

* ****识别未使用的驱动程序****。查看 /dev 和 /sys 目录，并查找带有常规 SELinux 标签的节点（这种标签表示相应节点未配置为可由用户空间访问）。 如果找到此类节点，请将其移除。
* ****取消设置未使用的配置****。查看由内核版本生成的 .config 文件，以明确取消设置所有已默认启用但并未使用的配置。例如，我们从 Google Pixel 中移除了以下未使用的配置：

CONFIG\_ANDROID\_LOGGER=y  
CONFIG\_IMX134=y  
CONFIG\_IMX132=y  
CONFIG\_OV9724=y  
CONFIG\_OV5648=y  
CONFIG\_GC0339=y  
CONFIG\_OV8825=y  
CONFIG\_OV8865=y  
CONFIG\_s5k4e1=y  
CONFIG\_OV12830=y  
CONFIG\_USB\_EHCI\_HCD=y  
CONFIG\_IOMMU\_IO\_PGTABLE\_FAST\_SELFTEST=y  
CONFIG\_IKCONFIG=y  
CONFIG\_RD\_BZIP2=y  
CONFIG\_RD\_LZMA=y  
CONFIG\_TI\_DRV2667=y  
CONFIG\_CHR\_DEV\_SCH=y  
CONFIG\_MMC=y  
CONFIG\_MMC\_PERF\_PROFILING=y  
CONFIG\_MMC\_CLKGATE=y  
CONFIG\_MMC\_PARANOID\_SD\_INIT=y  
CONFIG\_MMC\_BLOCK\_MINORS=32  
CONFIG\_MMC\_TEST=y  
CONFIG\_MMC\_SDHCI=y  
CONFIG\_MMC\_SDHCI\_PLTFM=y  
CONFIG\_MMC\_SDHCI\_MSM=y  
CONFIG\_MMC\_SDHCI\_MSM\_ICE=y  
CONFIG\_MMC\_CQ\_HCI=y  
CONFIG\_MSDOS\_FS=y  
# CONFIG\_SYSFS\_SYSCALL is not set  
CONFIG\_EEPROM\_AT24=y  
# CONFIG\_INPUT\_MOUSEDEV\_PSAUX is not set  
CONFIG\_INPUT\_HBTP\_INPUT=y  
# CONFIG\_VGA\_ARB is not set  
CONFIG\_USB\_MON=y  
CONFIG\_USB\_STORAGE\_DATAFAB=y  
CONFIG\_USB\_STORAGE\_FREECOM=y  
CONFIG\_USB\_STORAGE\_ISD200=y  
CONFIG\_USB\_STORAGE\_USBAT=y  
CONFIG\_USB\_STORAGE\_SDDR09=y  
CONFIG\_USB\_STORAGE\_SDDR55=y  
CONFIG\_USB\_STORAGE\_JUMPSHOT=y  
CONFIG\_USB\_STORAGE\_ALAUDA=y  
CONFIG\_USB\_STORAGE\_KARMA=y  
CONFIG\_USB\_STORAGE\_CYPRESS\_ATACB=y  
CONFIG\_SW\_SYNC\_USER=y  
CONFIG\_SEEMP\_CORE=y  
CONFIG\_MSM\_SMEM\_LOGGING=y  
CONFIG\_IOMMU\_DEBUG=y  
CONFIG\_IOMMU\_DEBUG\_TRACKING=y  
CONFIG\_IOMMU\_TESTS=y  
CONFIG\_MOBICORE\_DRIVER=y  
# CONFIG\_DEBUG\_PREEMPT is not set

* ****移除导致每次启动时运行不必要测试的配置****。虽然此类配置（即 CONFIG\_IOMMU\_IO\_PGTABLE\_FAST\_SELFTEST）在开发过程中很有用，但应从正式版内核中移除。

#### 最大限度地减小驱动程序大小

如果未使用相应功能，则可以移除设备内核中的某些驱动程序，以便进一步减小内核大小。例如，如果 WLAN 通过 PCIe 连接，则不会用到 SDIO 支持，因此应在编译时将其移除。有关详情，请参阅 Google Pixel 内核：网络：无线：CNSS：添加选项以停用 SDIO 支持。

#### 移除针对大小的编译器优化

移除 CONFIG\_CC\_OPTIMIZE\_FOR\_SIZE 的内核配置。此标记是在最初假设较小的代码大小会产生热缓存命中（因此速度更快）时引入的。然而，随着现代移动 SoC 变得更加强大，这一假设不再成立。

此外，移除此标记可以使编译器针对未初始化的变量发出警告，当存在 CONFIG\_CC\_OPTIMIZE\_FOR\_SIZE 标记时，这一功能在 Linux 内核中是停用的（仅这一项更改就已帮助我们在某些 Android 设备驱动程序中发现了很多有意义的错误）。

#### 延迟初始化

很多进程都在设备启动期间启动，但只有关键路径 (bootloader > kernel > init > file system mount > zygote > system server) 中的组件才会直接影响启动时间。在内核启动期间执行 ****initcall**** 来识别对启动 init 进程不重要的外设/组件，然后将这些外设/组件延迟到启动过程的后期来启动。

BoardConfig-common.mk:  
    BOARD\_KERNEL\_CMDLINE += initcall\_debug ignore\_loglevel

### 优化 I/O 效率

提高 I/O 效率对缩短启动时间来说至关重要，对任何不必要内容的读取都应推迟到启动之后再进行（在 Google Pixel 上，启动时大约要读取 1.2GB 的数据）。

#### 调整文件系统

当从头开始读取某个文件或依序读取块时，预读的 Linux 内核便会启动，这就需要调整专门用于启动的 I/O 调度程序参数（与普通应用的工作负载特性不同）。

支持无缝 (A/B) 更新的设备在首次启动时会极大地受益于文件系统调整（例如，Google Pixel 的启动时间缩短了 20 秒）。例如，我们为 Google Pixel 调整了以下参数：

on late-fs  
  # boot time fs tune  
    # boot time fs tune  
    write /sys/block/sda/queue/iostats 0  
    write /sys/block/sda/queue/scheduler cfq  
    write /sys/block/sda/queue/iosched/slice\_idle 0  
    write /sys/block/sda/queue/read\_ahead\_kb 2048  
    write /sys/block/sda/queue/nr\_requests 256  
    write /sys/block/dm-0/queue/read\_ahead\_kb 2048  
    write /sys/block/dm-1/queue/read\_ahead\_kb 2048  
  
on property:sys.boot\_completed=1  
    # end boot time fs tune  
    write /sys/block/sda/queue/read\_ahead\_kb 512  
    ...

#### 其他

* 使用内核配置 DM\_VERITY\_HASH\_PREFETCH\_MIN\_SIZE（默认大小为 128）来启用 dm-verity 哈希预提取大小。
* 为了提升文件系统稳定性及取消每次启动时的强制检查，请在 BoardConfig.mk 中设置 TARGET\_USES\_MKE2FS，以使用新的 ext4 生成工具。

#### 分析 I/O

要了解启动过程中的 I/O 活动，请使用内核 ftrace 数据（systrace 也使用该数据）：

trace\_event=block,ext4 in BOARD\_KERNEL\_CMDLINE

要针对每个文件细分文件访问权限，请对内核进行以下更改（仅限开发版内核；请勿在正式版内核中应用这些更改）：

diff --git a/fs/open.c b/fs/open.c  
index 1651f35..a808093 100644  
--- a/fs/open.c  
+++ b/fs/open.c  
@@ -981,6 +981,25 @@  
 }  
 EXPORT\_SYMBOL(file\_open\_root);  
  
+static void \_trace\_do\_sys\_open(struct file \*filp, int flags, int mode, long fd)  
+{  
+       char \*buf;  
+       char \*fname;  
+  
+       buf = kzalloc(PAGE\_SIZE, GFP\_KERNEL);  
+       if (!buf)  
+               return;  
+       fname = d\_path(&filp-<f\_path, buf, PAGE\_SIZE);  
+  
+       if (IS\_ERR(fname))  
+               goto out;  
+  
+       trace\_printk("%s: open(\"%s\", %d, %d) fd = %ld, inode = %ld\n",  
+                     current-<comm, fname, flags, mode, fd, filp-<f\_inode-<i\_ino);  
+out:  
+       kfree(buf);  
+}  
+  
long do\_sys\_open(int dfd, const char \_\_user \*filename, int flags, umode\_t mode)  
 {  
        struct open\_flags op;  
@@ -1003,6 +1022,7 @@  
                } else {  
                        fsnotify\_open(f);  
                        fd\_install(fd, f);  
+                       \_trace\_do\_sys\_open(f, flags, mode, fd);

使用以下脚本来帮助分析启动性能。

* packages/services/Car/tools/bootanalyze/bootanalyze.py：负责衡量启动时间，并详细分析启动过程中的重要步骤。
* packages/services/Car/tools/io\_analysis/check\_file\_read.py boot\_trace：提供每个文件的访问信息。
* packages/services/Car/tools/io\_analysis/check\_io\_trace\_all.py boot\_trace：提供系统级细分信息。

### 优化 init.\*.rc

Init 是从内核到框架建立之前的衔接过程，设备通常会在不同的 init 阶段花费几秒钟时间。

#### 并行运行任务

虽然当前的 Android init 差不多算是一种单线程进程，但您仍然可以并行执行一些任务。

* 在 Shell 脚本服务中执行缓慢命令，然后通过等待特定属性，在稍后加入。Android 8.0 通过新的 wait\_for\_property 命令支持此用例。
* 识别 init 中的缓慢操作。系统会记录 init 命令 exec/wait\_for\_prop 或任何所需时间较长的操作（在 Android 8.0 中，指所需时间超过 50 毫秒的任何命令）。例如：

init: Command 'wait\_for\_coldboot\_done' action=wait\_for\_coldboot\_done returned 0 took 585.012ms

查看此日志可能会发现可以改进的机会。

* 启动服务并及早启用关键路径中的外围设备。例如，有些 SOC 需要先启动安全相关服务，然后再启动 SurfaceFlinger。在 ServiceManager 返回“wait for service”（等待服务）时查看系统日志 - 这通常表明必须先启动依赖服务。
* 移除 init.\*.rc 中所有未使用的服务和命令。只要是早期阶段的 init 中没有使用的服务和命令，都应推迟到启动完成后再使用。

#### 使用调度程序调整

使用调度程序调整，以便及早启动设备。以下是取自 Google Pixel 的示例：

on init  
    # update cpusets now that processors are up  
    write /dev/cpuset/top-app/cpus 0-3  
    write /dev/cpuset/foreground/cpus 0-3  
    write /dev/cpuset/foreground/boost/cpus 0-3  
    write /dev/cpuset/background/cpus 0-3  
    write /dev/cpuset/system-background/cpus 0-3  
    # set default schedTune value for foreground/top-app (only affects EAS)  
    write /dev/stune/foreground/schedtune.prefer\_idle 1  
    write /dev/stune/top-app/schedtune.boost 10  
    write /dev/stune/top-app/schedtune.prefer\_idle 1

部分服务在启动过程中可能需要进行优先级提升。例如：

init.zygote64.rc:  
service zygote /system/bin/app\_process64 -Xzygote /system/bin --zygote --start-system-server  
    class main  
    priority -20  
    user root  
...

#### 及早启动 zygote

采用文件级加密的设备可以在 zygote-start 触发器的早期阶段启动 zygote（默认情况下，zygote 会在 main 类中启动，比 zygote-start 晚得多）。这样做时，请确保允许 zygote 在所有 CPU 中运行（因为错误的 cpuset 设置可能会强制 zygote 在特定 CPU 中运行）。

#### 停用节电设置

在设备启动期间，可以停用 UFS 和/或 CPU 调节器等组件的节电设置。

****请注意****：为了提高效率，应在充电器模式下启用节电设置。

on init  
    # Disable UFS powersaving  
    write /sys/devices/soc/${ro.boot.bootdevice}/clkscale\_enable 0  
    write /sys/devices/soc/${ro.boot.bootdevice}/clkgate\_enable 0  
    write /sys/devices/soc/${ro.boot.bootdevice}/hibern8\_on\_idle\_enable 0  
    write /sys/module/lpm\_levels/parameters/sleep\_disabled Y  
on property:sys.boot\_completed=1  
    # Enable UFS powersaving  
    write /sys/devices/soc/${ro.boot.bootdevice}/clkscale\_enable 1  
    write /sys/devices/soc/${ro.boot.bootdevice}/clkgate\_enable 1  
    write /sys/devices/soc/${ro.boot.bootdevice}/hibern8\_on\_idle\_enable 1  
    write /sys/module/lpm\_levels/parameters/sleep\_disabled N  
on charger  
    # Enable UFS powersaving  
    write /sys/devices/soc/${ro.boot.bootdevice}/clkscale\_enable 1  
    write /sys/devices/soc/${ro.boot.bootdevice}/clkgate\_enable 1  
    write /sys/devices/soc/${ro.boot.bootdevice}/hibern8\_on\_idle\_enable 1  
    write /sys/class/typec/port0/port\_type sink  
    write /sys/module/lpm\_levels/parameters/sleep\_disabled N

#### 推迟非关键初始化

非关键初始化（如 ZRAM）可以推迟到 boot\_complete。

on property:sys.boot\_completed=1  
   # Enable ZRAM on boot\_complete  
   swapon\_all /vendor/etc/fstab.${ro.hardware}

### 优化启动动画

请按照以下提示来优化启动动画。

#### 配置为及早启动

Android 8.0 支持在装载用户数据分区之前，及早启动动画。然而，即使 Android 8.0 中使用了新的 ext4 工具链，系统也会出于安全原因定期触发 fsck，导致启动 bootanimation 服务时出现延迟。

为了使 bootanimation 及早启动，请将 fstab 装载分为以下两个阶段：

* 在早期阶段，仅装载不需要运行检查的分区（例如 system/ 和 vendor/），然后启动启动动画服务及其依赖项（例如 servicemanager 和 surfaceflinger）。
* 在第二个阶段，装载需要运行检查的分区（例如 data/）。

启动动画将会更快速地启动（且启动时间恒定），不受 fsck 影响。

#### 干净利落地结束

在收到退出信号后，bootanimation 会播放最后一部分，而这一部分的长度会延长启动时间。快速启动的系统不需要很长的动画，如果启动动画很长，在很大程度上就体现不出所做的任何改进。我们建议缩短循环播放和结尾的时间。

### 优化 SELinux

请按照以下提示优化 SELinux 以缩短启动时间。

* ****使用简洁的正则表达式 (regex)****。在为 file\_contexts 中的 sys/devices 匹配 SELinux 政策时，格式糟糕的正则表达式可能会导致大量开销。例如，正则表达式 /sys/devices/.\*abc.\*(/.\*)? 错误地强制扫描包含“abc”的所有 /sys/devices 子目录，导致 /sys/devices/abc 和 /sys/devices/xyz/abc 都成为匹配项。如果将此正则表达式修正为 /sys/devices/[^/]\*abc[^/]\*(/.\*)? ，则只有 /sys/devices/abc 会成为匹配项。
* ****将标签移动到****[genfscon](https://selinuxproject.org/page/FileStatements" \l "genfscon)。这一现有的 SELinux 功能会将文件匹配前缀传递到 SELinux 二进制文件的内核中，而内核会将这些前缀应用于内核生成的文件系统。这也有助于修复错误标记的内核创建的文件，从而防止用户空间进程之间可能出现的争用情况（试图在重新标记之前访问这些文件）。

### 工具和方法

请使用以下工具来帮助您收集用于优化目标的数据。

#### bootchart

bootchart 可为整个系统提供所有进程的 CPU 和 I/O 负载细分。该工具不需要重建系统映像，可以用作进入 systrace 之前的快速健全性检查。

要启用 bootchart，请运行以下命令：

adb shell 'touch /data/bootchart/enabled'adb reboot

在设备启动后，获取启动图表：

$ANDROID\_BUILD\_TOP/system/core/init/grab-bootchart.sh

完成后，请删除 /data/bootchart/enabled 以防止每次都收集日期数据。

#### systrace

systrace 允许在启动期间收集内核和 Android 跟踪记录。 systrace 的可视化可以帮助分析启动过程中的具体问题。（不过，要查看整个启动过程中的平均数量或累计数量，直接查看内核跟踪记录更为方便）。

要在启动过程中启用 systrace，请执行以下操作：

* 在 frameworks/native/atrace/atrace.rc 中，将

write /sys/kernel/debug/tracing/tracing\_on 0

更改为：

#write /sys/kernel/debug/tracing/tracing\_on 0

这将启用跟踪功能（默认处于停用状态）。

* 在 device.mk 文件中，添加下面这行内容：

PRODUCT\_PROPERTY\_OVERRIDES +=    debug.atrace.tags.enableflags=802922

* 在设备 BoardConfig.mk 文件中，添加以下内容：

BOARD\_KERNEL\_CMDLINE := ... trace\_buf\_size=64M trace\_event=sched\_wakeup,sched\_switch,sched\_blocked\_reason,sched\_cpu\_hotplug

要获得详细的 I/O 分析，还需要添加块和 ext4。

* 在设备专用的 init.rc 文件中，进行以下更改：
* on property:sys.boot\_completed=1（这会在启动完成后停止跟踪）
* write /d/tracing/tracing\_on 0
* write /d/tracing/events/ext4/enable 0
* write /d/tracing/events/block/enable 0

在设备启动后，获取跟踪记录：

adb root && adb shell "cat /d/tracing/trace" < boot\_trace\_x000D\_./external/chromium-trace/catapult/tracing/bin/trace2html boot\_trace --output boot\_trace.html

## 代码功能裁剪

### 裁剪未使用的驱动

### 裁剪init.\*.rc

### 裁剪service

## Cache清理机制

### 1.启动cache清理服务

使用init.sun8iw11p1.rc启动cache清理服务。

service drop\_caches /sbin/busybox sh /system/bin/flushcache.sh

user root

group root system

disabled

Oneshot

### 2.Cache清理启动门限

android/frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

在文件中定义：

static final int SW\_MEM\_FLUSH\_THRESTHOD1 = 120000;// =< 512M 25%

static final int SW\_MEM\_FLUSH\_THRESTHOD2 = 300000;// > 512M 38%

这个门限可以根据实际使用进行修改。代码实现如下：

private void flushPageCache() {

MemInfoReader memInfo = new MemInfoReader();

memInfo.readMemInfo();

if ((memInfo.getTotalSizeKb() <= 524288

&& memInfo.getCachedSizeKb() > SW\_MEM\_FLUSH\_THRESTHOD1)

|| (memInfo.getTotalSizeKb() > 524288

&& memInfo.getCachedSizeKb() > SW\_MEM\_FLUSH\_THRESTHOD2)) {

SystemProperties.set("ctl.start", "drop\_caches");

}

Message msg = Message.obtain();

msg.what = SW\_FLUSH\_PAGECACHE\_MSG;

mHandler.sendMessageDelayed(msg, SW\_SEND\_MSG\_DELAYED);

return;

}

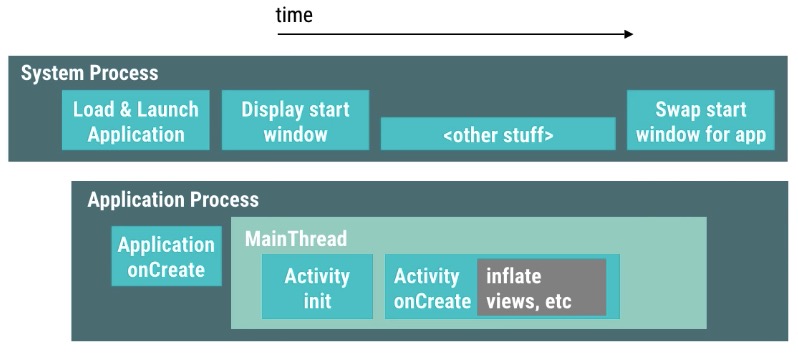
# 应用优化

## 启动优化

### 优化activity的创建过程

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。

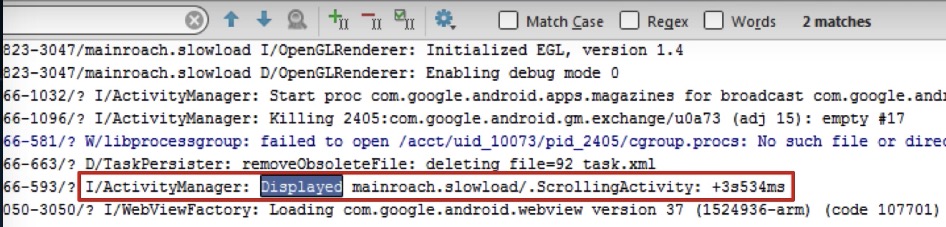


上述流程里面的绝大多数步骤都是由系统控制的，一般来说不会出现什么问题，可是对于启动速度，我们能够控制并且需要特别关注的地方主要有三处：

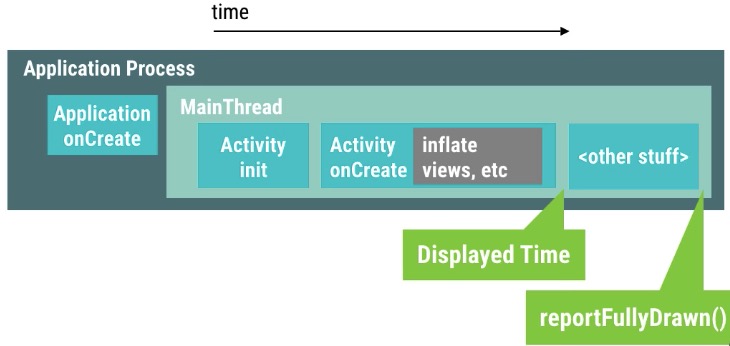
* 1）Activity的onCreate流程，特别是UI的布局与渲染操作，如果布局过于复杂很可能导致严重的启动性能问题。
* 2）Application的onCreate流程，对于大型的APP来说，通常会在这里做大量的通用组件的初始化操作。
* 3）目前有部分APP会提供自定义的启动窗口，这里可以做成品牌宣传界面或者是给用户提供一种程序已经启动的视觉效果。

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。

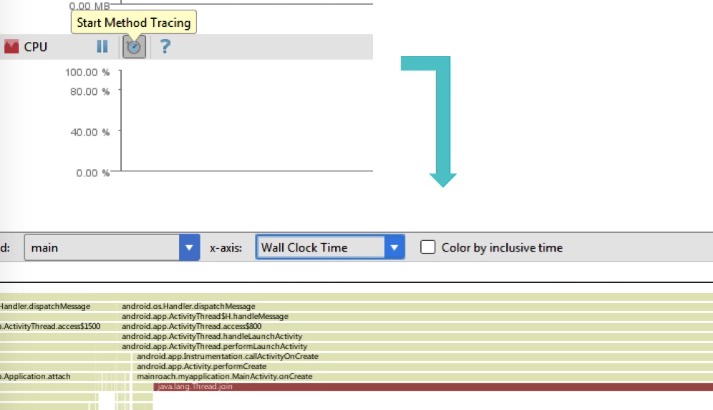
* 1）首先是**display time**：从Android KitKat版本开始，Logcat中会输出从程序启动到某个Activity显示到画面上所花费的时间。这个方法比较适合测量程序的启动时间。



* 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。

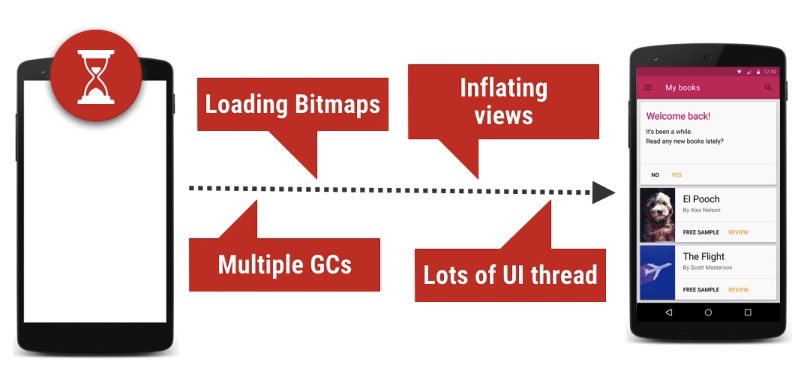


* 4）最后是**Systrace**：我们可以在onCreate方法里面添加trace.beginSection()与trace.endSection()方法来声明需要跟踪的起止位置，系统会帮忙统计中间经历过的函数调用耗时，并输出报表。



### 优化application对象的启动过程

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。

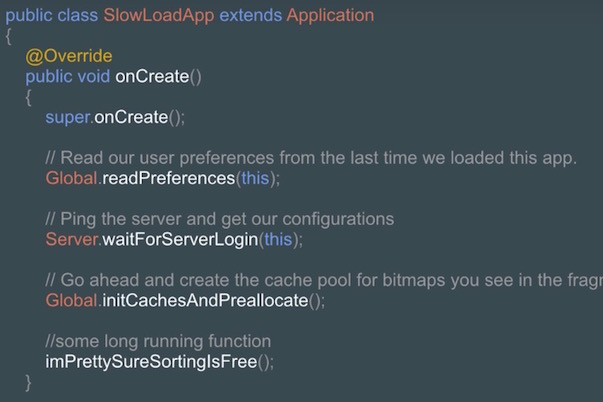


上述操作的任何一个环节出现性能问题都可能导致画面不能及时显示，影响了程序的启动速度。上一个段落我们介绍了使用Method Tracing来发现那些耗时占比相对较多的方法。假设我们发现某个方法执行时间过长，接下去就可以使用Systrace来帮忙定位到底是什么原因导致那个方法执行时间过长。

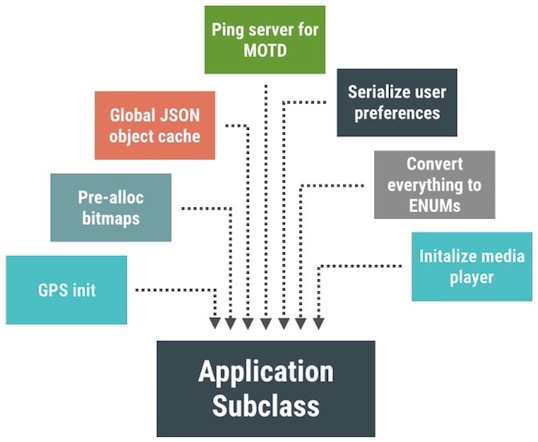
除了使用工具进行具体定位分析性能问题之外，以下两点经验可以帮助我们对Activity启动做性能优化：

* 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。

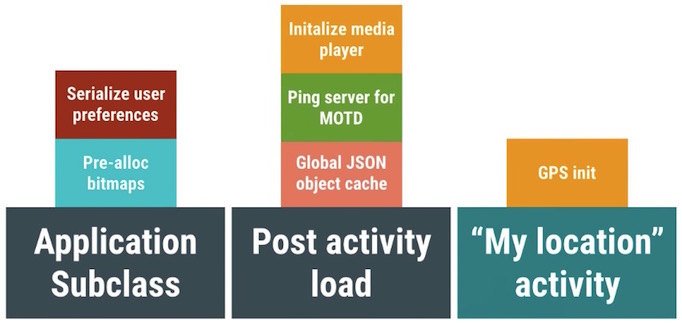
在Application初始化的地方做太多繁重的事情是可能导致严重启动性能问题的元凶之一。Application里面的初始化操作不结束，其他任意的程序操作都无法进行。



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。



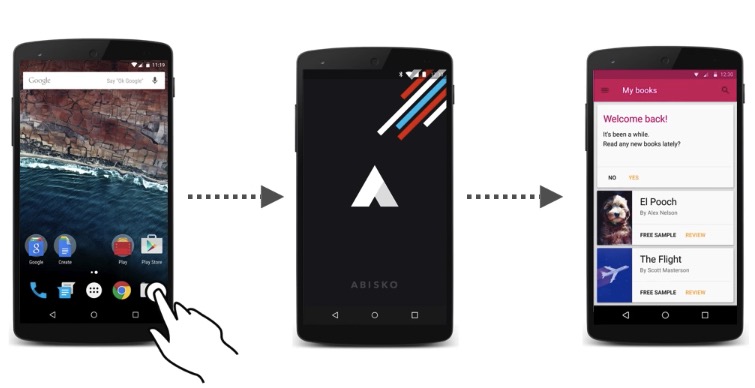
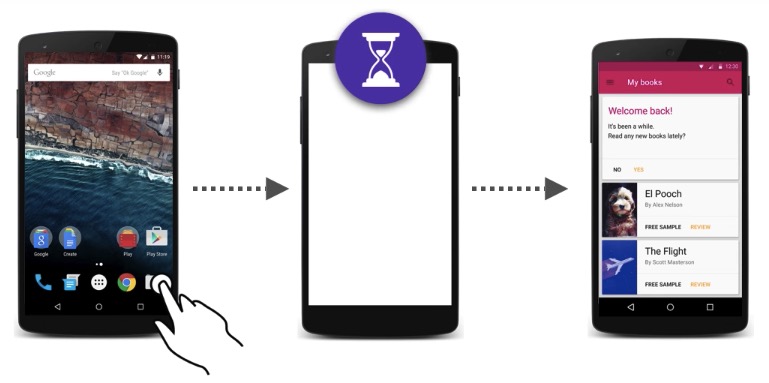
优化这些问题的解决方案是做延迟加载，可以在application里面做延迟加载，也可以把一些初始化的操作延迟到组件真正被调用到的时候再做加载。



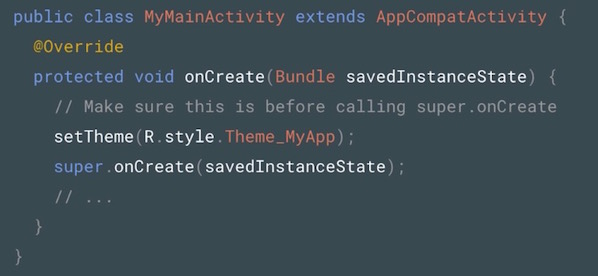
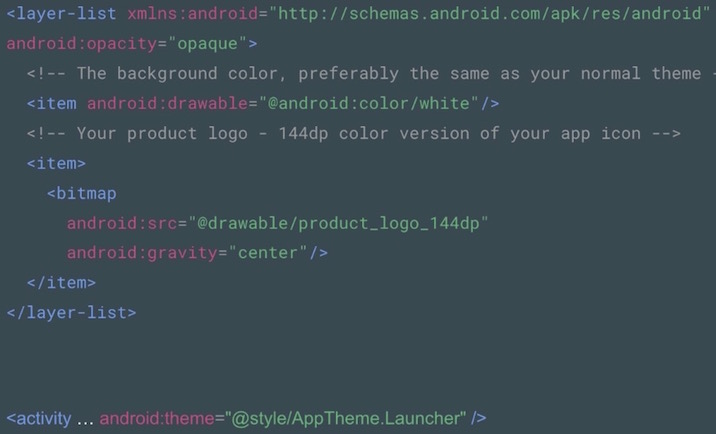
### 正确使用闪屏

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。



## 安装包大小优化

http://hukai.me/android-performance-patterns-season-6/

http://hukai.me/android-performance-patterns-season-4/

## 内存优化

Random Access Memory(RAM)在任何软件开发环境中都是一个很宝贵的资源。这一点在物理内存通常很有限的移动操作系统上，显得尤为突出。尽管Android的Dalvik虚拟机扮演了常规的垃圾回收的角色，但这并不意味着你可以忽视app的内存分配与释放的时机与地点。

为了GC能够从app中及时回收内存，我们需要注意避免内存泄露(通常由于在全局成员变量中持有对象引用而导致)并且在适当的时机(下面会讲到的lifecycle callbacks)来释放引用对象。对于大多数app来说，Dalvik的GC会自动把离开活动线程的对象进行回收。

这篇文章会解释Android是如何管理app的进程与内存分配，以及在开发Android应用的时候如何主动的减少内存的使用。关于Java的资源管理机制，请参考其它书籍或者线上材料。如果你正在寻找如何分析你的内存使用情况的文章，请参考这里[Investigating Your RAM Usage](http://developer.android.com/tools/debugging/debugging-memory.html" \t "http://hukai.me/android-training-course-in-chinese/performance/_blank)。

### 第1部分: Android是如何管理内存的

Android并没有为内存提供交换区(Swap space)，但是它有使用[paging](http://en.wikipedia.org/wiki/Paging" \t "http://hukai.me/android-training-course-in-chinese/performance/_blank)与[memory-mapping(mmapping)](http://en.wikipedia.org/wiki/Memory-mapped_files" \t "http://hukai.me/android-training-course-in-chinese/performance/_blank)的机制来管理内存。这意味着任何你修改的内存(无论是通过分配新的对象还是去访问mmaped pages中的内容)都会贮存在RAM中，而且不能被paged out。因此唯一完整释放内存的方法是释放那些你可能hold住的对象的引用，当这个对象没有被任何其他对象所引用的时候，它就能够被GC回收了。只有一种例外是：如果系统想要在其他地方重用这个对象。

#### 1) 共享内存

Android通过下面几个方式在不同的进程中来实现共享RAM:

每一个app的进程都是从一个被叫做**Zygote**的进程中fork出来的。Zygote进程在系统启动并且载入通用的framework的代码与资源之后开始启动。为了启动一个新的程序进程，系统会fork Zygote进程生成一个新的进程，然后在新的进程中加载并运行app的代码。这使得大多数的RAM pages被用来分配给framework的代码，同时使得RAM资源能够在应用的所有进程中进行共享。

大多数static的数据被mmapped到一个进程中。这不仅仅使得同样的数据能够在进程间进行共享，而且使得它能够在需要的时候被paged out。例如下面几种static的数据:

* + Dalvik 代码 (放在一个预链接好的 .odex 文件中以便直接mapping)
  + App resources (通过把资源表结构设计成便于mmapping的数据结构，另外还可以通过把APK中的文件做aligning的操作来优化)
  + 传统项目元素，比如 .so 文件中的本地代码.
* 在很多情况下，Android通过显式的分配共享内存区域(例如ashmem或者gralloc)来实现一些动态RAM区域能够在不同进程间进行共享。例如，window surfaces在app与screen compositor之间使用共享的内存，cursor buffers在content provider与client之间使用共享的内存。

关于如何查看app所使用的共享内存，请查看[Investigating Your RAM Usage](http://developer.android.com/tools/debugging/debugging-memory.html" \t "http://hukai.me/android-training-course-in-chinese/performance/_blank)

#### 2) 分配与回收内存

这里有下面几点关于Android如何分配与回收内存的事实：

* 每一个进程的Dalvik heap都有一个受限的虚拟内存范围。这就是逻辑上讲的heap size，它可以随着需要进行增长，但是会有一个系统为它所定义的上限。
* 逻辑上讲的heap size和实际物理上使用的内存数量是不等的，Android会计算一个叫做Proportional Set Size(PSS)的值，它记录了那些和其他进程进行共享的内存大小。（假设共享内存大小是10M，一共有20个Process在共享使用，根据权重，可能认为其中有0.3M才能真正算是你的进程所使用的）
* Dalvik heap与逻辑上的heap size不吻合，这意味着Android并不会去做heap中的碎片整理用来关闭空闲区域。Android仅仅会在heap的尾端出现不使用的空间时才会做收缩逻辑heap size大小的动作。但是这并不是意味着被heap所使用的物理内存大小不能被收缩。在垃圾回收之后，Dalvik会遍历heap并找出不使用的pages，然后使用madvise(系统调用)把那些pages返回给kernal。因此，成对的allocations与deallocations大块的数据可以使得物理内存能够被正常的回收。然而，回收碎片化的内存则会使得效率低下很多，因为那些碎片化的分配页面也许会被其他地方所共享到。

#### 3) 限制应用的内存

为了维持多任务的功能环境，Android为每一个app都设置了一个硬性的heap size限制。准确的heap size限制会因为不同设备的不同RAM大小而各有差异。如果你的app已经到了heap的限制大小并且再尝试分配内存的话，会引起OutOfMemoryError的错误。

在一些情况下，你也许想要查询当前设备的heap size限制大小是多少，然后决定cache的大小。可以通过getMemoryClass()来查询。这个方法会返回一个整数，表明你的应用的heap size限制是多少Mb(megabates)。

#### 4) 切换应用

Android并不会在用户切换不同应用时候做交换内存的操作。Android会把那些不包含foreground组件的进程放到LRU cache中。例如，当用户刚开始启动了一个应用，系统会为它创建了一个进程，但是当用户离开这个应用，此进程并不会立即被销毁。系统会把这个进程放到cache中，如果用户后来再回到这个应用，此进程就能够被完整恢复，从而实现应用的快速切换。

如果你的应用中有一个被缓存的进程，这个进程会占用暂时不需要使用到的内存，这个暂时不需要使用的进程，它被保留在内存中，这会对系统的整体性能有影响。因此当系统开始进入低内存状态时，它会由系统根据LRU的规则与其他因素选择综合考虑之后决定杀掉某些进程，为了保持你的进程能够尽可能长久的被缓存，请参考下面的章节学习何时释放你的引用。

对于那些不在foreground的进程，Android是如何决定kill掉哪一类进程的问题，请参考[Processes and Threads](http://developer.android.com/guide/components/processes-and-threads.html" \t "http://hukai.me/android-training-course-in-chinese/performance/_blank).

### 第2部分: 你的应用该如何管理内存

你应该在开发过程的每一个阶段都考虑到RAM的有限性，甚至包括在开始编写代码之前的设计阶段就应该考虑到RAM的限制性。我们可以使用多种设计与实现方式，他们有着不同的效率，即使这些方式只是相同技术的不断组合与演变。

为了使得你的应用性能效率更高，你应该在设计与实现代码时，遵循下面的技术要点。

#### 1) 珍惜Services资源

如果你的应用需要在后台使用service，除非它被触发并执行一个任务，否则其他时候service都应该是停止状态。另外需要注意当这个service完成任务之后因为停止service失败而引起的内存泄漏。

当你启动一个service，系统会倾向为了保留这个service而一直保留service所在的进程。这使得进程的运行代价很高，因为系统没有办法把service所占用的RAM空间腾出来让给其他组件，另外service还不能被paged out。这减少了系统能够存放到LRU缓存当中的进程数量，它会影响app之间的切换效率。它甚至会导致系统内存使用不稳定，从而无法继续保持住所有目前正在运行的service。

限制你的service的最好办法是使用[IntentService](http://developer.android.com/reference/android/app/IntentService.html" \t "http://hukai.me/android-training-course-in-chinese/performance/_blank)， 它会在处理完交代给它的intent任务之后尽快结束自己。更多信息，请阅读[Running in a Background Service](http://developer.android.com/training/run-background-service/index.html" \t "http://hukai.me/android-training-course-in-chinese/performance/_blank).

当一个Service已经不再需要的时候还继续保留它，这对Android应用的内存管理来说是**最糟糕的错误之一**。因此千万不要贪婪的使得一个Service持续保留。不仅仅是因为它会使得你的应用因为RAM空间的不足而性能糟糕，还会使得用户发现那些有着常驻后台行为的应用并且可能卸载它。

#### 2) 当UI隐藏时释放内存

当用户切换到其它应用并且你的应用 UI不再可见时，你应该释放你的应用UI上所占用的所有内存资源。在这个时候释放UI资源可以显著的增加系统缓存进程的能力，它会对用户体验有着很直接的影响。

为了能够接收到用户离开你的UI时的通知，你需要实现Activtiy类里面的onTrimMemory()回调方法。你应该使用这个方法来监听到TRIM\_MEMORY\_UI\_HIDDEN级别的回调，此时意味着你的UI已经隐藏，你应该释放那些仅仅被你的UI使用的资源。

请注意：你的应用仅仅会在所有UI组件的被隐藏的时候接收到onTrimMemory()的回调并带有参数TRIM\_MEMORY\_UI\_HIDDEN。这与onStop()的回调是不同的，onStop会在activity的实例隐藏时会执行，例如当用户从你的app的某个activity跳转到另外一个activity时前面activity的onStop()会被执行。因此你应该实现onStop回调，并且在此回调里面释放activity的资源，例如释放网络连接，注销监听广播接收者。除非接收到[onTrimMemory(TRIM\_MEMORY\_UI\_HIDDEN)](http://developer.android.com/reference/android/content/ComponentCallbacks2.html" \l "onTrimMemory(int" \t "http://hukai.me/android-training-course-in-chinese/performance/_blank))的回调，否者你不应该释放你的UI资源。这确保了用户从其他activity切回来时，你的UI资源仍然可用，并且可以迅速恢复activity。

#### 3) 当内存紧张时释放部分内存

在你的app生命周期的任何阶段，onTrimMemory的回调方法同样可以告诉你整个设备的内存资源已经开始紧张。你应该根据onTrimMemory回调中的内存级别来进一步决定释放哪些资源。

* [TRIM\_MEMORY\_RUNNING\_MODERATE](http://developer.android.com/reference/android/content/ComponentCallbacks2.html" \l "TRIM_MEMORY_RUNNING_MODERATE" \t "http://hukai.me/android-training-course-in-chinese/performance/_blank)：你的app正在运行并且不会被列为可杀死的。但是设备此时正运行于低内存状态下，系统开始触发杀死LRU Cache中的Process的机制。
* [TRIM\_MEMORY\_RUNNING\_LOW](http://developer.android.com/reference/android/content/ComponentCallbacks2.html" \l "TRIM_MEMORY_RUNNING_LOW" \t "http://hukai.me/android-training-course-in-chinese/performance/_blank)：你的app正在运行且没有被列为可杀死的。但是设备正运行于更低内存的状态下，你应该释放不用的资源用来提升系统性能（但是这也会直接影响到你的app的性能）。
* [TRIM\_MEMORY\_RUNNING\_CRITICAL](http://developer.android.com/reference/android/content/ComponentCallbacks2.html" \l "TRIM_MEMORY_RUNNING_CRITICAL" \t "http://hukai.me/android-training-course-in-chinese/performance/_blank)：你的app仍在运行，但是系统已经把LRU Cache中的大多数进程都已经杀死，因此你应该立即释放所有非必须的资源。如果系统不能回收到足够的RAM数量，系统将会清除所有的LRU缓存中的进程，并且开始杀死那些之前被认为不应该杀死的进程，例如那个包含了一个运行态Service的进程。

同样，当你的app进程正在被cached时，你可能会接受到从onTrimMemory()中返回的下面的值之一:

* [TRIM\_MEMORY\_BACKGROUND](http://developer.android.com/reference/android/content/ComponentCallbacks2.html" \l "TRIM_MEMORY_BACKGROUND" \t "http://hukai.me/android-training-course-in-chinese/performance/_blank): 系统正运行于低内存状态并且你的进程正处于LRU缓存名单中**最不容易杀掉的位置**。尽管你的app进程并不是处于被杀掉的高危险状态，系统可能已经开始杀掉LRU缓存中的其他进程了。你应该释放那些容易恢复的资源，以便于你的进程可以保留下来，这样当用户回退到你的app的时候才能够迅速恢复。
* [TRIM\_MEMORY\_MODERATE](http://developer.android.com/reference/android/content/ComponentCallbacks2.html" \l "TRIM_MEMORY_MODERATE" \t "http://hukai.me/android-training-course-in-chinese/performance/_blank): 系统正运行于低内存状态并且你的进程已经已经接近LRU名单的**中部位置**。如果系统开始变得更加内存紧张，你的进程是有可能被杀死的。
* [TRIM\_MEMORY\_COMPLETE](http://developer.android.com/reference/android/content/ComponentCallbacks2.html" \l "TRIM_MEMORY_COMPLETE" \t "http://hukai.me/android-training-course-in-chinese/performance/_blank): 系统正运行与低内存的状态并且你的进程正处于LRU名单中**最容易被杀掉的位置**。你应该释放任何不影响你的app恢复状态的资源。

因为onTrimMemory()的回调是在**API 14**才被加进来的，对于老的版本，你可以使用[onLowMemory](http://developer.android.com/reference/android/content/ComponentCallbacks.html" \l "onLowMemory(" \t "http://hukai.me/android-training-course-in-chinese/performance/_blank))回调来进行兼容。onLowMemory相当与TRIM\_MEMORY\_COMPLETE。

**Note:** 当系统开始清除LRU缓存中的进程时，尽管它首先按照LRU的顺序来操作，但是它同样会考虑进程的内存使用量。因此消耗越少的进程则越容易被留下来。

#### 4) 检查你应该使用多少的内存

正如前面提到的，每一个Android设备都会有不同的RAM总大小与可用空间，因此不同设备为app提供了不同大小的heap限制。你可以通过调用[getMemoryClass()](http://developer.android.com/reference/android/app/ActivityManager.html" \l "getMemoryClass(" \t "http://hukai.me/android-training-course-in-chinese/performance/_blank))来获取你的app的可用heap大小。如果你的app尝试申请更多的内存，会出现OutOfMemory的错误。

在一些特殊的情景下，你可以通过在manifest的application标签下添加largeHeap=true的属性来声明一个更大的heap空间。如果你这样做，你可以通过[getLargeMemoryClass()](http://developer.android.com/reference/android/app/ActivityManager.html" \l "getLargeMemoryClass(" \t "http://hukai.me/android-training-course-in-chinese/performance/_blank))来获取到一个更大的heap size。

然而，能够获取更大heap的设计本意是为了一小部分会消耗大量RAM的应用(例如一个大图片的编辑应用)。**不要轻易的因为你需要使用大量的内存而去请求一个大的heap size。**只有当你清楚的知道哪里会使用大量的内存并且为什么这些内存必须被保留时才去使用large heap. 因此请尽量少使用large heap。使用额外的内存会影响系统整体的用户体验，并且会使得GC的每次运行时间更长。在任务切换时，系统的性能会变得大打折扣。

另外, large heap并不一定能够获取到更大的heap。在某些有严格限制的机器上，large heap的大小和通常的heap size是一样的。因此即使你申请了large heap，你还是应该通过执行getMemoryClass()来检查实际获取到的heap大小。

#### 5) 避免bitmaps的浪费

当你加载一个bitmap时，仅仅需要保留适配当前屏幕设备分辨率的数据即可，如果原图高于你的设备分辨率，需要做缩小的动作。请记住，增加bitmap的尺寸会对内存呈现出2次方的增加，因为X与Y都在增加。

**Note:**在Android 2.3.x (API level 10)及其以下, bitmap对象的pixel data是存放在native内存中的，它不便于调试。然而，从Android 3.0(API level 11)开始，bitmap pixel data是分配在你的app的Dalvik heap中, 这提升了GC的工作效率并且更加容易Debug。因此如果你的app使用bitmap并在旧的机器上引发了一些内存问题，切换到3.0以上的机器上进行Debug。

#### 6) 使用优化的数据容器

利用Android Framework里面优化过的容器类，例如[SparseArray](http://developer.android.com/reference/android/util/SparseArray.html" \t "http://hukai.me/android-training-course-in-chinese/performance/_blank), [SparseBooleanArray](http://developer.android.com/reference/android/util/SparseBooleanArray.html" \t "http://hukai.me/android-training-course-in-chinese/performance/_blank), 与 [LongSparseArray](http://developer.android.com/reference/android/support/v4/util/LongSparseArray.html" \t "http://hukai.me/android-training-course-in-chinese/performance/_blank)。 通常的HashMap的实现方式更加消耗内存，因为它需要一个额外的实例对象来记录Mapping操作。另外，SparseArray更加高效在于他们避免了对key与value的autobox自动装箱，并且避免了装箱后的解箱。

#### 7) 请注意内存开销

对你所使用的语言与库的成本与开销有所了解，从开始到结束，在设计你的app时谨记这些信息。通常，表面上看起来无关痛痒(innocuous)的事情也许实际上会导致大量的开销。例如：

* Enums的内存消耗通常是static constants的2倍。你应该尽量避免在Android上使用enums。
* 在Java中的每一个类(包括匿名内部类)都会使用大概500 bytes。
* 每一个类的实例花销是12-16 bytes。
* 往HashMap添加一个entry需要额一个额外占用的32 bytes的entry对象。

#### 8) 请注意代码“抽象”

通常，开发者使用抽象作为"好的编程实践"，因为抽象能够提升代码的灵活性与可维护性。然而，抽象会导致一个显著的开销：通常他们需要同等量的代码用于可执行。那些代码会被map到内存中。因此如果你的抽象没有显著的提升效率，应该尽量避免他们。

#### 9) 为序列化的数据使用nano protobufs

[Protocol buffers](https://developers.google.com/protocol-buffers/docs/overview" \t "http://hukai.me/android-training-course-in-chinese/performance/_blank)是由Google为序列化结构数据而设计的，一种语言无关，平台无关，具有良好扩展性的协议。类似XML，却比XML更加轻量，快速，简单。如果你需要为你的数据实现协议化，你应该在客户端的代码中总是使用nano protobufs。通常的协议化操作会生成大量繁琐的代码，这容易给你的app带来许多问题：增加RAM的使用量，显著增加APK的大小，更慢的执行速度，更容易达到DEX的字符限制。

关于更多细节，请参考[protobuf readme](https://android.googlesource.com/platform/external/protobuf/+/master/java/README.txt" \t "http://hukai.me/android-training-course-in-chinese/performance/_blank)的"Nano version"章节。

#### 10) 避免使用依赖注入框架

使用类似[Guice](https://code.google.com/p/google-guice/" \t "http://hukai.me/android-training-course-in-chinese/performance/_blank)或者[RoboGuice](https://github.com/roboguice/roboguice" \t "http://hukai.me/android-training-course-in-chinese/performance/_blank)等framework injection包是很有效的，因为他们能够简化你的代码。

Notes：RoboGuice 2 通过依赖注入改变代码风格，让Android开发时的体验更好。你在调用 getIntent().getExtras() 时经常忘记检查 null 吗？RoboGuice 2 可以帮你做。你认为将 findViewById() 的返回值强制转换成 TextView 是本不必要的工作吗？ RoboGuice 2 可以帮你。RoboGuice 把这些需要猜测性的工作移到Android开发以外去了。RoboGuice 2 会负责注入你的 View, Resource, System Service或者其他对象等等类似的细节。

然而，那些框架会通过扫描你的代码执行许多初始化的操作，这会导致你的代码需要大量的RAM来mapping代码，而且mapped pages会长时间的被保留在RAM中。

#### 11) 谨慎使用第三方libraries

很多开源的library代码都不是为移动网络环境而编写的，如果运用在移动设备上，，这样的效率并不高。当你决定使用一个第三方library的时候，你应该针对移动网络做繁琐的迁移与维护的工作。

即使是针对Android而设计的library，也可能是很危险的，因为每一个library所做的事情都是不一样的。例如，其中一个lib使用的是nano protobufs, 而另外一个使用的是micro protobufs。那么这样，在你的app里面就有2种protobuf的实现方式。这样的冲突同样可能发生在输出日志，加载图片，缓存等等模块里面。

同样不要陷入为了1个或者2个功能而导入整个library的陷阱。如果没有一个合适的库与你的需求相吻合，你应该考虑自己去实现，而不是导入一个大而全的解决方案。

#### 12) 优化整体性能

官方有列出许多优化整个app性能的文章：[Best Practices for Performance](http://developer.android.com/training/best-performance.html" \t "http://hukai.me/android-training-course-in-chinese/performance/_blank)。这篇文章就是其中之一。有些文章是讲解如何优化app的CPU使用效率，有些是如何优化app的内存使用效率。

你还应该阅读[optimizing your UI](http://developer.android.com/tools/debugging/debugging-ui.html" \t "http://hukai.me/android-training-course-in-chinese/performance/_blank)来为layout进行优化。同样还应该关注lint工具所提出的建议，进行优化。

#### 13) 使用ProGuard来剔除不需要的代码

[ProGuard](http://developer.android.com/tools/help/proguard.html" \t "http://hukai.me/android-training-course-in-chinese/performance/_blank)能够通过移除不需要的代码，重命名类，域与方法等方对代码进行压缩，优化与混淆。使用ProGuard可以使得你的代码更加紧凑，这样能够使用更少mapped代码所需要的RAM。

#### 14) 对最终的APK使用zipalign

在编写完所有代码，并通过编译系统生成APK之后，你需要使用[zipalign](http://developer.android.com/tools/help/zipalign.html" \t "http://hukai.me/android-training-course-in-chinese/performance/_blank)对APK进行重新校准。如果你不做这个步骤，会导致你的APK需要更多的RAM，因为一些类似图片资源的东西不能被mapped。

**Notes:**Google Play不接受没有经过zipalign的APK。

#### 15) 分析你的RAM使用情况

一旦你获取到一个相对稳定的版本后，需要分析你的app整个生命周期内使用的内存情况，并进行优化，更多细节请参考[Investigating Your RAM Usage](http://developer.android.com/tools/debugging/debugging-memory.html" \t "http://hukai.me/android-training-course-in-chinese/performance/_blank).

#### 16) 使用多进程

如果合适的话，有一个更高级的技术可以帮助你的app管理内存使用：通过把你的app组件切分成多个组件，运行在不同的进程中。这个技术必须谨慎使用，大多数app都不应该运行在多个进程中。因为如果使用不当，它会显著增加内存的使用，而不是减少。当你的app需要在后台运行与前台一样的大量的任务的时候，可以考虑使用这个技术。

一个典型的例子是创建一个可以长时间后台播放的Music Player。如果整个app运行在一个进程中，当后台播放的时候，前台的那些UI资源也没有办法得到释放。类似这样的app可以切分成2个进程：一个用来操作UI，另外一个用来后台的Service.

你可以通过在manifest文件中声明'android:process'属性来实现某个组件运行在另外一个进程的操作。

<service android:name=".PlaybackService" android:process=":background" />

更多关于使用这个技术的细节，请参考原文，链接如下。[http://developer.android.com/training/articles/memory.html](http://developer.android.com/training/articles/memory.html" \t "http://hukai.me/android-training-course-in-chinese/performance/_blank)

<http://hukai.me/android-performance-oom/>

<http://hukai.me/android-performance-patterns/>

## 提升Layout的性能

### 优化layout的层级

一个常见的误区是，用最基础的 Layout 结构可以提高 Layout 的 性能。然而，因为程序的每个组件和 Layout 都需要经过初始化、布局和绘制的过程，如果布局嵌套导致层级过深，上面的初始化，布局和绘制操作就更加耗时。例如，使用嵌套的 LinearLayout 可能会使得 View 的层级结构过深，此外，嵌套使用了 layout\_weight 参数的 LinearLayout 的计算量会尤其大，因为每个子元素都需要被测量两次。这对需要多次重复 inflate 的 Layout 尤其需要注意，比如嵌套在 ListView 或 GridView 时。

在本课中，你将学习使用 [Hierarchy Viewer](http://developer.android.com/tools/help/hierarchy-viewer.html" \t "http://hukai.me/android-training-course-in-chinese/performance/improving-layouts/_blank)和[Layoutopt](http://developer.android.com/tools/help/layoutopt.html" \t "http://hukai.me/android-training-course-in-chinese/performance/improving-layouts/_blank)来检查和优化 Layout。

#### 检查 Layout

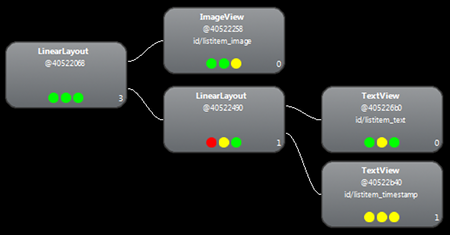
Android SDK 工具箱中有一个叫做 [Hierarchy Viewer](http://developer.android.com/tools/help/hierarchy-viewer.html" \t "http://hukai.me/android-training-course-in-chinese/performance/improving-layouts/_blank) 的工具，能够在程序运行时分析 Layout。你可以用这个工具找到 Layout 的性能瓶颈。

Hierarchy Viewer 会让你选择设备或者模拟器上正在运行的进程，然后显示其 Layout 的树型结构。每个块上的交通灯分别代表了它在测量、布局和绘画时的性能，帮你找出瓶颈部分。

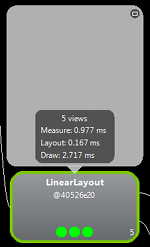
比如，下图是 ListView 中一个列表项的 Layout 。列表项里，左边放一个小位图，右边是两个层叠的文字。像这种需要被多次 inflate 的 Layout ，优化它们会有事半功倍的效果。



hierarchyviewer 这个工具在 <sdk>/tools/ 中。当打开时，它显示一张可使用设备的列表，和它正在运行的组件。点击 **Load View Hierarchy** 来查看所选组件的层级。比如，下图就是前一个图中所示 Layout 的层级关系。



在上图中，你可以看到一个三层结构，其中右下角的 TextView 在布局的时候有问题。点击这个TextView可以看到每个步骤所花费的时间。

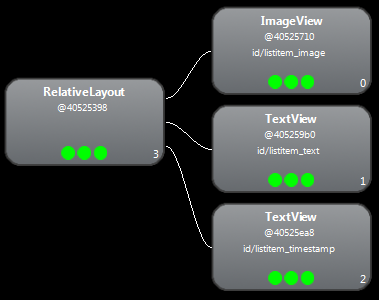


可以看到，渲染一个完整的列表项的时间就是：

* 测量: 0.977ms
* 布局: 0.167ms
* 绘制: 2.717ms

#### 修正 Layout

上面的 Layout 由于有这个嵌套的 LinearLayout 导致性能太慢，可能的解决办法是将 Layout 层级扁平化 - 变浅变宽，而不是又窄又深。RelativeaLayout 作为根节点时就可以达到目的。所以，当换成基于 RelativeLayout 的设计时，你的 Layout 变成了两层。新的 Layout 变成这样：



现在渲染列表项的时间：

* 测量: 0.598ms
* 布局: 0.110ms
* 绘制: 2.146ms

可能看起来是很小的进步，但是由于它对列表中每个项都有效，这个时间要翻倍。

这个时间的主要差异是由于在 LinearLayout 中使用 layout\_weight 所致，因为会减慢“测量”的速度。这只是一个正确使用各种 Layout 的例子，当你使用 layout\_weight 时有必要慎重。

#### 使用 Lint

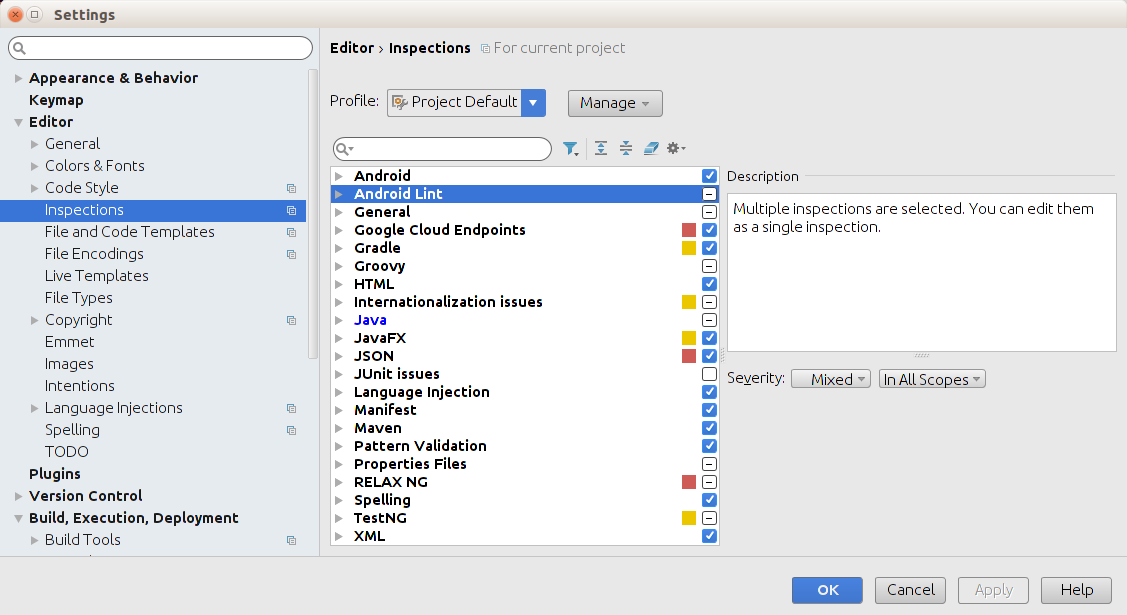
大部分叫做 lint 的编程工具，都是类似于代码规范的检测工具。比如JSLint，CSSLinkt， JSONLint 等等。译者注。

运行 [Lint](http://tools.android.com/tips/lint" \t "http://hukai.me/android-training-course-in-chinese/performance/improving-layouts/_blank) 工具来检查 Layout 可能的优化方法，是个很好的实践。Lint 已经取代了 Layoutopt 工具，它拥有更强大的功能。Lint 中包含的一些检测[规则](http://tools.android.com/tips/lint-checks" \t "http://hukai.me/android-training-course-in-chinese/performance/improving-layouts/_blank)有：

* 使用compound drawable — 用一个compound drawable 替代一个包含 ImageView 和 TextView 的 LinearLayout 会更有效率。
* 合并根 frame — 如果 FrameLayout 是 Layout 的根节点，并且没有使用 padding 或者背景等，那么用 merge 标签替代他们会稍微高效些。
* 没用的子节点 — 一个没有子节点或者背景的 Layout 应该被去掉，来获得更扁平的层级
* 没用的父节点 — 一个节点如果没有兄弟节点，并且它不是 ScrollView 或根节点，没有背景，这样的节点应该直接被子节点取代，来获得更扁平的层级
* 太深的 Layout — Layout 的嵌套层数太深对性能有很大影响。尝试使用更扁平的 Layout ，比如 RelativeLayout 或 GridLayout 来提高性能。一般最多不超过10层。

另一个使用 Lint 的好处就是，它内置于 Android Studio 中。Lint 在你导编译程序时自动运行。Android Studio 中，你可以为单独的 build variant 或者所有 variant 运行 lint。

你也可以在 Android Studio 中管理检测选项，在 **File > Settings > Project Settings** 中。检测配置页面会显示支持的检测项目。



Lint 有自动修复、提示建议和直接跳转到问题处的功能。

### 使用include标签重用layouts

android:layout\_height="wrap\_content" android:background="@color/titlebar\_bg"> <ImageView android:layout\_width="wrap\_content" android:layout\_height="wrap\_content" android:src="@drawable/gafricalogo" /></FrameLayout>

根节点 View 就是你想添加入的 Layout 类型。

#### 使用<include>标签

使用 <include> 标签，可以在 Layout 中添加可重用的组件。比如，这里有一个来自 G-Kenya codelab 的 Layout 需要包含上面的那个标题栏：

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android" android:orientation="vertical" android:layout\_width=”match\_parent” android:layout\_height=”match\_parent” android:background="@color/app\_bg" android:gravity="center\_horizontal"> <include layout="@layout/titlebar"/> <TextView android:layout\_width=”match\_parent” android:layout\_height="wrap\_content" android:text="@string/hello" android:padding="10dp" /> ...</LinearLayout>

你也可以覆写被添加的 Layout 的所有 Layout 参数（任何 android:layout\_\* 属性），通过在 <include/> 中声明他们来完成。比如：

<include android:id="@+id/news\_title" android:layout\_width="match\_parent" android:layout\_height="match\_parent" layout="@layout/title"/>

然而，如果你要在 <include> 中覆写某些属性，你必须先覆写 android:layout\_height 和 android:layout\_width。

#### 使用<merge>标签

<merge /> 标签在你嵌套 Layout 时取消了 UI 层级中冗余的 ViewGroup 。比如，如果你有一个 Layout 是一个竖直方向的 LinearLayout，其中包含两个连续的 View 可以在别的 Layout 中重用，那么你会做一个 LinearLayout 来包含这两个 View ，以便重用。不过，当使用一个 LinearLayout 作为另一个 LinearLayout 的根节点时，这种嵌套 LinearLayout 的方式除了减慢你的 UI 性能外没有任何意义。

为了避免这种情况，你可以用 <merge> 元素来替代可重用 Layout 的根节点。例如：

<merge xmlns:android="http://schemas.android.com/apk/res/android"> <Button android:layout\_width="fill\_parent" android:layout\_height="wrap\_content" android:text="@string/add"/>

<Button android:layout\_width="fill\_parent" android:layout\_height="wrap\_content" android:text="@string/delete"/></merge>

现在，当你要将这个 Layout 包含到另一个 Layout 中时（并且使用了 <include/> 标签），系统会忽略 <merge> 标签，直接把两个 Button 放到 Layout 中 <include> 的所在位置。

### 按需加载视图

有时你的 Layout 会用到不怎么重用的复杂视图。不管它是列表项 细节，进度显示器，或是撤销时的提示信息，你可以仅在需要的时候载入它们，提高 UI 渲染速度。

#### 定义 ViewStub

[ViewStub](http://developer.android.com/reference/android/view/ViewStub.html" \t "http://hukai.me/android-training-course-in-chinese/performance/improving-layouts/_blank) 是一个轻量的视图，不需要大小信息，也不会在被加入的 Layout 中绘制任何东西。每个 ViewStub 只需要设置 android:layout 属性来指定需要被 inflate 的 Layout 类型。

以下 ViewStub 是一个半透明的进度条覆盖层。功能上讲，它应该只在新的数据项被导入到应用程序时可见。

<ViewStub android:id="@+id/stub\_import" android:inflatedId="@+id/panel\_import" android:layout="@layout/progress\_overlay" android:layout\_width="fill\_parent" android:layout\_height="wrap\_content" android:layout\_gravity="bottom" />

#### 载入 ViewStub Layout

当你要载入用 ViewStub 声明的 Layout 时，要么用 setVisibility(View.VISIBLE) 设置它的可见性，要么调用其 inflate() 方法。

((ViewStub) findViewById(R.id.stub\_import)).setVisibility(View.VISIBLE);// orView importPanel = ((ViewStub) findViewById(R.id.stub\_import)).inflate();

**Notes**：inflate() 方法会在渲染完成后返回被 inflate 的视图，所以如果你需要和这个 Layout 交互的话， 你不需要再调用 findViewById() 去查找这个元素，。

一旦 ViewStub 可见或是被 inflate 了，ViewStub 就不再继续存在View的层级机构中了。取而代之的是被 inflate 的 Layout，其 id 是 ViewStub 上的 android:inflatedId 属性。（ViewStub 的 android:id 属性仅在 ViewStub 可见以前可用）

**Notes**：ViewStub 的一个缺陷是，它目前不支持使用 <merge/> 标签的 Layout 。

### 使得ListView滑动顺畅

保持程序流畅的关键，是让主线程（UI 线程）不要进行大量运算。你要确保在其他线程执行磁盘读写、网络读写或是 SQL 操作等。为了测试你的应用的状态，你可以启用 [StrictMode](http://developer.android.com/reference/android/os/StrictMode.html" \t "http://hukai.me/android-training-course-in-chinese/performance/improving-layouts/_blank)。

#### 使用后台线程

你应该把主线程中的耗时间的操作，提取到一个后台线程（也叫做“worker thread工作线程”）中，使得主线程只关注 UI 绘画。很多时候，使用 [AsyncTask](http://developer.android.com/reference/android/os/AsyncTask.html" \t "http://hukai.me/android-training-course-in-chinese/performance/improving-layouts/_blank) 是一个简单的在主线程以外进行操作的方法。系统会自动把execute()的请求放入队列中并线性调用执行。这个行为是全局的，这意味着你不需要考虑自己定义线程池的事情。

在下面的例子中，一个 AsyncTask 被用于在后台线程载入图片，并在载入完成后把图片显示到 UI 上。当图片正在载入时，它还会显示一个进度提示。

// Using an AsyncTask to load the slow images in a background threadnew AsyncTask<ViewHolder, Void, Bitmap>() { private ViewHolder v; @Override protected Bitmap doInBackground(ViewHolder... params) { v = params[0]; return mFakeImageLoader.getImage(); } @Override protected void onPostExecute(Bitmap result) { super.onPostExecute(result); if (v.position == position) { // If this item hasn't been recycled already, hide the // progress and set and show the image v.progress.setVisibility(View.GONE); v.icon.setVisibility(View.VISIBLE); v.icon.setImageBitmap(result); } }}.execute(holder);

从 Android 3.0 (API level 11) 开始, AsyncTask 有个新特性，那就是它可以在多个 CPU 核上运行。你可以调用 executeOnExecutor()而不是execute()，前者可以根据CPU的核心数来触发多个任务同时进行。

#### 在 ViewHolder 中填入视图对象

你的代码可能在 ListView 滑动时经常使用 findViewById()，这样会降低性能。即使是 Adapter 返回一个用于回收的 inflate 后的视图，你仍然需要查看这个元素并更新它。避免频繁调用 findViewById() 的方法之一，就是使用 ViewHolder（视图占位符）的设计模式。

一个 ViewHolder 对象存储了他的标签下的每个视图。这样你不用频繁查找这个元素。第一，你需要创建一个类来存储你会用到的视图。比如：

static class ViewHolder { TextView text; TextView timestamp; ImageView icon; ProgressBar progress; int position;}

然后，在 Layout 的类中生成一个 ViewHolder 对象：

ViewHolder holder = new ViewHolder();holder.icon = (ImageView) convertView.findViewById(R.id.listitem\_image);holder.text = (TextView) convertView.findViewById(R.id.listitem\_text);holder.timestamp = (TextView) convertView.findViewById(R.id.listitem\_timestamp);holder.progress = (ProgressBar) convertView.findViewById(R.id.progress\_spinner);convertView.setTag(holder);

这样你就可以轻松获取每个视图，而不是使用 findViewById() 来不断查找子视图，节省了宝贵的运算时间。

## 流畅性优化

http://hukai.me/android-performance-patterns-season-2/

http://hukai.me/android-performance-patterns/

## 电量优化

### 监测电池的电量与充电状态

当你想通过改变后台更新操作的频率来减少对电池寿命的影响时，那么首先需要检查当前电量与充电状态。

执行应用更新对电池寿命的影响是与电量和充电状态密切相关的。当使用交流电对设备充电时，更新操作的影响可以忽略不计，所以在大多数情况下，如果使用壁式充电器对设备进行充电，我们可以将刷新频率设置到最大。相反的，如果设备没有在充电状态，那么我们就需要尽量减少设备的更新操作来延长电池的续航能力。

同样的，如果我们监测到电量即将耗尽时，那么应该尽可能降低甚至停止更新操作。

#### 判断当前充电状态

首先来看一下应该如何确定当前的充电状态。[BatteryManager](http://developer.android.com/reference/android/os/BatteryManager.html" \t "http://hukai.me/android-training-course-in-chinese/performance/monitor-device-state/_blank)会广播一个带有电池与充电详情的[Sticky Intent](http://developer.android.com/guide/topics/fundamentals/services.html" \t "http://hukai.me/android-training-course-in-chinese/performance/monitor-device-state/_blank)

因为广播的是一个sticky Intent，所以不需要注册[BroadcastReceiver](http://developer.android.com/reference/android/content/BroadcastReceiver.html" \t "http://hukai.me/android-training-course-in-chinese/performance/monitor-device-state/_blank)。仅仅只需要调用一个以null作为Receiver参数的registerReceiver()方法就可以了。如下面的代码片段中展示的那样，它返回了保存当前电池信息的Intent。你也可以在这里传入一个实际的[BroadcastReceiver](http://developer.android.com/reference/android/content/BroadcastReceiver.html" \t "http://hukai.me/android-training-course-in-chinese/performance/monitor-device-state/_blank)对象，但这并不是必须的。

IntentFilter ifilter = new IntentFilter(Intent.ACTION\_BATTERY\_CHANGED);Intent batteryStatus = context.registerReceiver(null, ifilter);

我们可以提取出当前的充电状态，以及设备处于充电时，是通过USB还是交流充电器充电的。

// Are we charging / charged?int status = batteryStatus.getIntExtra(BatteryManager.EXTRA\_STATUS, -1);boolean isCharging = status == BatteryManager.BATTERY\_STATUS\_CHARGING || status == BatteryManager.BATTERY\_STATUS\_FULL;// How are we charging?int chargePlug = batteryStatus.getIntExtra(BatteryManager.EXTRA\_PLUGGED, -1);boolean usbCharge = chargePlug == BatteryManager.BATTERY\_PLUGGED\_USB;boolean acCharge = chargePlug == BatteryManager.BATTERY\_PLUGGED\_AC;

通常，我们可以在设备使用交流充电时最大化后台更新频率，在使用USB充电时降低更新频率，在非充电状态时，将更新频率进一步降低。

#### 监测充电状态的改变

充电状态随时可能改变，所以我们应该检查充电状态的改变来调整更新频率。

[BatteryManager](http://developer.android.com/reference/android/os/BatteryManager.html" \t "http://hukai.me/android-training-course-in-chinese/performance/monitor-device-state/_blank)会在设备连接或者断开充电器的时候广播一个Action。即使应用没有运行，我们也应该接收这些事件的广播，主要原因是因为这些事件会影响到应用启动（从而进行更新）的频率，因此我们应该在Manifest文件里面注册一个[BroadcastReceiver](http://developer.android.com/reference/android/content/BroadcastReceiver.html" \t "http://hukai.me/android-training-course-in-chinese/performance/monitor-device-state/_blank)来监听含有[ACTION\_POWER\_CONNECTED](http://developer.android.com/reference/android/content/Intent.html" \l "ACTION_POWER_CONNECTED" \t "http://hukai.me/android-training-course-in-chinese/performance/monitor-device-state/_blank) 与 [ACTION\_POWER\_DISCONNECTED](http://developer.android.com/reference/android/content/Intent.html" \l "ACTION_POWER_DISCONNECTED" \t "http://hukai.me/android-training-course-in-chinese/performance/monitor-device-state/_blank)的Intent。

<receiver android:name=".PowerConnectionReceiver"> <intent-filter> <action android:name="android.intent.action.ACTION\_POWER\_CONNECTED"/> <action android:name="android.intent.action.ACTION\_POWER\_DISCONNECTED"/> </intent-filter></receiver>

我们可以在该[BroadcastReceiver](http://developer.android.com/reference/android/content/BroadcastReceiver.html" \t "http://hukai.me/android-training-course-in-chinese/performance/monitor-device-state/_blank)的实现中，提取出当前的充电状态，如下所示：

public class PowerConnectionReceiver extends BroadcastReceiver { @Override public void onReceive(Context context, Intent intent) { int status = intent.getIntExtra(BatteryManager.EXTRA\_STATUS, -1); boolean isCharging = status == BatteryManager.BATTERY\_STATUS\_CHARGING || status == BatteryManager.BATTERY\_STATUS\_FULL; int chargePlug = intent.getIntExtra(BatteryManager.EXTRA\_PLUGGED, -1); boolean usbCharge = chargePlug == BatteryManager.BATTERY\_PLUGGED\_USB; boolean acCharge = chargePlug == BatteryManager.BATTERY\_PLUGGED\_AC; }}

#### 判断当前电池电量

在一些情况下，获取到当前电池电量也很有帮助。我们可以在获知电量少于某个级别的时候减少后台的更新频率。 我们可以通过电池状态Intent获取到电池电量与容量等信息，如下所示：

int level = batteryStatus.getIntExtra(BatteryManager.EXTRA\_LEVEL, -1);int scale = batteryStatus.getIntExtra(BatteryManager.EXTRA\_SCALE, -1);float batteryPct = level / (float)scale;

#### 检测电量的有效改变

我们不能不停地监测电池状态，实际上这也是不必要的。通常来说，不间断地监测电量信息对电池的影响会远大于应用本身对电池的影响。所以我们应该仅监测电量的一些显著性变化，特别是当设备进入或者离开低电量状态时。

在下面的Manifest文件片段中，BroadcastReceiver仅仅监听ACTION\_BATTERY\_LOW与ACTION\_BATTERY\_OKAY，这样它就只会在设备电量进入低电量或者离开低电量的时候被触发。

<receiver android:name=".BatteryLevelReceiver"><intent-filter> <action android:name="android.intent.action.ACTION\_BATTERY\_LOW"/> <action android:name="android.intent.action.ACTION\_BATTERY\_OKAY"/> </intent-filter></receiver>

通常我们都需要在进入低电量的情况下，关闭所有后台更新来维持设备的续航，因为这个时候做任何更新等操作都极有可能是无用的，因为也许在你还没来得及处理更新的数据时，设备就因电量耗尽而自动关机了。

在很多时候，用户往往会将设备放入某种底座中充电（译注：比如车载的底座式充电器），在下一节课程当中，我们将会学习如何确定当前的底座状态，以及如何监听设备底座的变化。

### 判断并监测设备的底座状态与类型

Android设备可以放置在许多不同的底座中，包括车载底座，家庭底座还有数字信号底座以及模拟信号底座等。由于许多底座会向设备充电，因此底座状态通常与充电状态密切相关。

你的应用类型决定了底座类型会对更新频率产生怎样的影响。对于一个体育类应用，可以让设备在笔记本底座状态下增加更新的频率，或者当设备在车载底座状态下停止更新。相反的，如果你的后台服务用来更新交通数据，你也可以选择在车载底座模式下最大化更新的频率。

底座状态也是以Sticky Intent方式来广播的，这样可以通过查询Intent里面的数据来判断目前设备是否放置在底座中，以及底座的类型。

#### 判断当前底座状态

底座状态的具体信息会以Extra数据的形式，包含在具有[ACTION\_DOCK\_EVENT](http://developer.android.com/reference/android/content/Intent.html" \l "ACTION_DOCK_EVENT" \t "http://hukai.me/android-training-course-in-chinese/performance/monitor-device-state/_blank)这一Action的某个Sticky广播中 ，因此，你不需要为其注册一个[BroadcastReceiver](http://developer.android.com/reference/android/content/BroadcastReceiver.html" \t "http://hukai.me/android-training-course-in-chinese/performance/monitor-device-state/_blank)。如下所示，仅需要将null作为参数传递给[registerReceiver()](http://developer.android.com/reference/android/content/Context.html" \l "registerReceiver(android.content.BroadcastReceiver, android.content.IntentFilter))方法就可以了：

IntentFilter ifilter = new IntentFilter(Intent.ACTION\_DOCK\_EVENT);Intent dockStatus = context.registerReceiver(null, ifilter);

你可以从EXTRA\_DOCK\_STATE这一Extra数据中，提取出当前的底座状态：

int dockState = battery.getIntExtra(EXTRA\_DOCK\_STATE, -1);boolean isDocked = dockState != Intent.EXTRA\_DOCK\_STATE\_UNDOCKED;

#### 判断当前底座类型

如果设备被放置在了底座中，那么它可以有下面四种底座类型：

* Car
* Desk
* Low-End (Analog) Desk
* High-End (Digital) Desk

注意最后两种底座类型仅在API Level 11及以后版本的Android系统中才被支持。如果你只在乎底座的类型而不管它是数字的还是模拟的，那么可以仅监测三种类型：

boolean isCar = dockState == EXTRA\_DOCK\_STATE\_CAR;boolean isDesk = dockState == EXTRA\_DOCK\_STATE\_DESK || dockState == EXTRA\_DOCK\_STATE\_LE\_DESK || dockState == EXTRA\_DOCK\_STATE\_HE\_DESK;

#### 监测底座状态或者类型的改变

当设备被放置在或者拔出底座时，系统会发出一个具有[ACTION\_DOCK\_EVENT](http://developer.android.com/reference/android/content/Intent.html" \l "ACTION_DOCK_EVENT" \t "http://hukai.me/android-training-course-in-chinese/performance/monitor-device-state/_blank)这一Action的广播。为了监听底座状态的变化，我们只需要在应用的Manifest文件中注册一个BroadcastReceiver，如下所示：

<action android:name="android.intent.action.ACTION\_DOCK\_EVENT"/>

之于该BroadcastReceiver的具体实现，可以参考前面提到的那些方法，以此来提取出当前的底座类型和状态。

### 判断并监测网络连接状态

重复闹钟和后台服务最常见的功能之一，是用来从网络上获取应用更新，存储数据或者执行大文件的下载。但是如果没有获得网络连接，或者连接的速度太慢以至于下载无法完成，那么就没有必要唤醒设备并执行那些更新等操作了。

我们可以使用[ConnectivityManager](http://developer.android.com/reference/android/net/ConnectivityManager.html" \t "http://hukai.me/android-training-course-in-chinese/performance/monitor-device-state/_blank)来检查设备是否连接到网络，以及网络的类型（译注：通过网络的连接状况改变，相应的改变app的行为，减少无谓的操作，从而延长设备的续航能力）。

#### 判断当前是否有网络连接

如果没有网络连接，那么就没有必要做那些需要联网的事情。下面的代码片段展示了如何通过[ConnectivityManager](http://developer.android.com/reference/android/net/ConnectivityManager.html" \t "http://hukai.me/android-training-course-in-chinese/performance/monitor-device-state/_blank)检查当前活动的网络类型，并确定它是否可以连接到互联网：

ConnectivityManager cm = (ConnectivityManager)context.getSystemService(Context.CONNECTIVITY\_SERVICE);NetworkInfo activeNetwork = cm.getActiveNetworkInfo();

boolean isConnected = activeNetwork != null && activeNetwork.isConnectedOrConnecting();

#### 判断连接网络的类型

我们还可以获取到当前的网络连接类型。

设备通常可以有移动网络，WiMax，Wi-Fi与以太网连接等类型。通过查询当前活动的网络类型，可以根据网络的带宽对更新频率进行调整：

boolean isWiFi = activeNetwork.getType() == ConnectivityManager.TYPE\_WIFI;

移动网络的使用费会比Wi-Fi更高，所以多数情况下，如果设备正在使用移动网络，我们应该减少应用的更新频率；同样地，还应该临时地挂起一些文件下载任务直到有Wi-Fi连接时再继续下载。

如果已经关闭了更新操作，那么需要监听网络连接的变化，这样就可以在建立了互联网访问之后，重新恢复它们。

#### 监听网络连接的变化

当网络连接发生改变时，[ConnectivityManager](http://developer.android.com/reference/android/net/ConnectivityManager.html" \t "http://hukai.me/android-training-course-in-chinese/performance/monitor-device-state/_blank)会广播[CONNECTIVITY\_ACTION](http://developer.android.com/reference/android/net/ConnectivityManager.html" \l "CONNECTIVITY_ACTION" \t "http://hukai.me/android-training-course-in-chinese/performance/monitor-device-state/_blank)（android.net.conn.CONNECTIVITY\_CHANGE）的Action消息。 我们可以在Manifest文件里面注册一个BroadcastReceiver，来监听这些变化，并适当地恢复（或挂起）你的后台更新:

<action android:name="android.net.conn.CONNECTIVITY\_CHANGE"/>

设备的网络变化可能会比较频繁，因此每当你在移动网络与Wi-Fi之间切换的时候，这一广播就会被触发。因此，我们可以仅在之前的更新或者下载任务被挂起的时候去监听这一广播（用来恢复那些任务）。通常我们可以在开始更新前检查一下网络连接，如果当前没有连接到互联网，那么就将更新任务挂起，直到连接恢复。

### 按需操控BroadcastReceiver

监测设备状态变化最简单的方法，是为你所要监听的每一个状态创建一个[BroadcastReceiver](http://developer.android.com/reference/android/content/BroadcastReceiver.html" \t "http://hukai.me/android-training-course-in-chinese/performance/monitor-device-state/_blank)，并在Manifest文件中注册它们。之后就可以在每一个BroadcastReceiver中，根据当前设备的状态调整一些计划任务。

上述方法的副作用是：一旦你的接收器收到了广播，应用就会唤醒设备。唤醒的频率可能会远高于需要的频率

更好的方法是在程序运行时开启或者关闭BroadcastReceiver。这样的话，你就可以让这些接收器仅在需要的时候被激活。

#### 切换是否开启接收器以提高效率

我们可以使用[PackageManager](http://developer.android.com/reference/android/content/pm/PackageManager.html" \t "http://hukai.me/android-training-course-in-chinese/performance/monitor-device-state/_blank)来切换任何一个在Mainfest里面定义好的组件的开启状态。通过下面的方法可以开启或者关闭任何一个BroadcastReceiver：

ComponentName receiver = new ComponentName(context, myReceiver.class);PackageManager pm = context.getPackageManager();pm.setComponentEnabledSetting(receiver, PackageManager.COMPONENT\_ENABLED\_STATE\_ENABLED, PackageManager.DONT\_KILL\_APP)

使用这种技术，如果我们确定网络连接已经断开，那么可以在这个时候关闭除了监听网络状态变化的接收器之外的其它所有接收器。

相反的，一旦重新建立网络连接，我们可以停止监听网络连接的改变，而仅仅在执行需要联网的操作之前判断当前网络是否可以用。

同样地，你可以使用上面的技术来暂缓一个需要更高带宽的下载任务。这仅需要启用一个监听网络连接变化的BroadcastReceiver，并在连接到Wi-Fi时，初始化下载任务。

## 多线程并发的性能

http://hukai.me/android-performance-patterns-season-5/

## 代码优化

通常来说，选择合适的算法与数据结构会是你首要考虑的因素，在这篇文章中不会涉及这方面的知识点。你应该使用这篇文章中的小技巧作为平时写代码的习惯，这样能够提升代码的效率。

通常来说，高效的代码需要满足下面两个原则：

* 不要做冗余的工作
* 尽量避免执行过多的内存分配操作

在优化App时其中一个难点就是让App能在各种型号的设备上运行。不同版本的虚拟机在不同的处理器上会有不同的运行速度。你甚至不能简单的认为“设备X的速度是设备Y的F倍”，然后还用这种倍数关系去推测其他设备。另外，在模拟器上的运行速度和在实际设备上的速度没有半点关系。同样，设备有没有JIT也对运行速度有重大影响：在有JIT情况下的最优化代码不一定在没有JIT的情况下也是最优的。

为了确保App在各设备上都能良好运行，就要确保你的代码在不同档次的设备上都尽可能的优化。

### 避免创建不必要的对象

创建对象从来不是免费的。**Generational GC**可以使临时对象的分配变得廉价一些，但是执行分配内存总是比不执行分配操作更昂贵。

随着你在App中分配更多的对象，你可能需要强制gc，而gc操作会给用户体验带来一点点卡顿。虽然从Android 2.3开始，引入了并发gc，它可以帮助你显著提升gc的效率，减轻卡顿，但毕竟不必要的内存分配操作还是应该尽量避免。

因此请尽量避免创建不必要的对象，有下面一些例子来说明这个问题：

* 如果你需要返回一个String对象，并且你知道它最终会需要连接到一个StringBuffer，请修改你的函数实现方式，避免直接进行连接操作，应该采用创建一个临时对象来做字符串的拼接这个操作。
* 当从已经存在的数据集中抽取出String的时候，尝试返回原数据的substring对象，而不是创建一个重复的对象。使用substring的方式，你将会得到一个新的String对象，但是这个string对象是和原string共享内部char[]空间的。

一个稍微激进点的做法是把所有多维的数据分解成一维的数组:

* 一组int数据要比一组Integer对象要好很多。可以得知，两组一维数组要比一个二维数组更加的有效率。同样的，这个道理可以推广至其他原始数据类型。
* 如果你需要实现一个数组用来存放(Foo,Bar)的对象，记住使用Foo[]与Bar[]要比(Foo,Bar)好很多。(例外的是，为了某些好的API的设计，可以适当做一些妥协。但是在自己的代码内部，你应该多多使用分解后的容易）。

通常来说，需要避免创建更多的临时对象。更少的对象意味者更少的gc动作，gc会对用户体验有比较直接的影响。

### 选择Static而不是Virtual

如果你不需要访问一个对象的值，请保证这个方法是static类型的，这样方法调用将快15%-20%。这是一个好的习惯，因为你可以从方法声明中得知调用无法改变这个对象的状态。

### 常量声明为Static Final

考虑下面这种声明的方式

static int intVal = 42;static String strVal = "Hello, world!";

编译器会使用一个初始化类的函数，然后当类第一次被使用的时候执行。这个函数将42存入intVal，还从class文件的常量表中提取了strVal的引用。当之后使用intVal或strVal的时候，他们会直接被查询到。

我们可以用final关键字来优化：

static final int intVal = 42;static final String strVal = "Hello, world!";

这时再也不需要上面的方法了，因为final声明的常量进入了静态dex文件的域初始化部分。调用intVal的代码会直接使用42，调用strVal的代码也会使用一个相对廉价的“字符串常量”指令，而不是查表。

**Notes：**这个优化方法只对原始类型和String类型有效，而不是任意引用类型。不过，在必要时使用static final是个很好的习惯。

### 避免内部的Getters/Setters

像C++等native language，通常使用getters(i = getCount())而不是直接访问变量(i = mCount)。这是编写C++的一种优秀习惯，而且通常也被其他面向对象的语言所采用，例如C#与Java，因为编译器通常会做inline访问，而且你需要限制或者调试变量，你可以在任何时候在getter/setter里面添加代码。

然而，在Android上，这不是一个好的写法。虚函数的调用比起直接访问变量要耗费更多。在面向对象编程中，将getter和setting暴露给公用接口是合理的，但在类内部应该仅仅使用域直接访问。

在没有JIT(Just In Time Compiler)时，直接访问变量的速度是调用getter的3倍。有JIT时，直接访问变量的速度是通过getter访问的7倍。

请注意，如果你使用[ProGuard](http://developer.android.com/tools/help/proguard.html" \t "http://hukai.me/android-training-course-in-chinese/performance/_blank)，你可以获得同样的效果，因为ProGuard可以为你inline accessors.

### 使用增强的For循环

增强的For循环（也被称为 for-each 循环）可以被用在实现了 Iterable 接口的 collections 以及数组上。使用collection的时候，Iterator会被分配，用于for-each调用hasNext()和next()方法。使用ArrayList时，手写的计数式for循环会快3倍（不管有没有JIT），但是对于其他collection，增强的for-each循环写法会和迭代器写法的效率一样。

请比较下面三种循环的方法：

static class Foo { int mSplat;}Foo[] mArray = ...public void zero() { int sum = 0; for (int i = 0; i < mArray.length; ++i) { sum += mArray[i].mSplat; }}public void one() { int sum = 0; Foo[] localArray = mArray; int len = localArray.length; for (int i = 0; i < len; ++i) { sum += localArray[i].mSplat; }}public void two() { int sum = 0; for (Foo a : mArray) { sum += a.mSplat; }}

* zero()是最慢的，因为JIT没有办法对它进行优化。
* one()稍微快些。
* two() 在没有做JIT时是最快的，可是如果经过JIT之后，与方法one()是差不多一样快的。它使用了增强的循环方法for-each。

所以请尽量使用for-each的方法，但是对于ArrayList，请使用方法one()。

**Tips：**你还可以参考 Josh Bloch 的 《Effective Java》这本书的第46条

### 使用包级访问而不是内部类的私有访问

参考下面一段代码

public class Foo { private class Inner { void stuff() { Foo.this.doStuff(Foo.this.mValue); } } private int mValue; public void run() { Inner in = new Inner(); mValue = 27; in.stuff(); } private void doStuff(int value) { System.out.println("Value is " + value); }}

这里重要的是，我们定义了一个私有的内部类（Foo$Inner），它直接访问了外部类中的私有方法以及私有成员对象。这是合法的，这段代码也会如同预期一样打印出"Value is 27"。

问题是，VM因为Foo和Foo$Inner是不同的类，会认为在Foo$Inner中直接访问Foo类的私有成员是不合法的。即使Java语言允许内部类访问外部类的私有成员。为了去除这种差异，编译器会产生一些仿造函数：

/\*package\*/ static int Foo.access$100(Foo foo) { return foo.mValue;}/\*package\*/ static void Foo.access$200(Foo foo, int value) { foo.doStuff(value);}

每当内部类需要访问外部类中的mValue成员或需要调用doStuff()函数时，它都会调用这些静态方法。这意味着，上面的代码可以归结为，通过accessor函数来访问成员变量。早些时候我们说过，通过accessor会比直接访问域要慢。所以，这是一个特定语言用法造成性能降低的例子。

如果你正在性能热区（hotspot:高频率、重复执行的代码段）使用像这样的代码，你可以把内部类需要访问的域和方法声明为包级访问，而不是私有访问权限。不幸的是，这意味着在相同包中的其他类也可以直接访问这些域，所以在公开的API中你不能这样做。

### 避免使用float类型

Android系统中float类型的数据存取速度是int类型的一半，尽量优先采用int类型。

就速度而言，现代硬件上，float 和 double 的速度是一样的。空间而言，double 是两倍float的大小。在空间不是问题的情况下，你应该使用 double 。

同样，对于整型，有些处理器实现了硬件几倍的乘法，但是没有除法。这时，整型的除法和取余是在软件内部实现的，这在你使用哈希表或大量计算操作时要考虑到。

### 使用库函数

除了那些常见的让你多使用自带库函数的理由以外，记得系统函数有时可以替代第三方库，并且还有汇编级别的优化，他们通常比带有JIT的Java编译出来的代码更高效。典型的例子是：Android API 中的 String.indexOf()，Dalvik出于内联性能考虑将其替换。同样 System.arraycopy()函数也被替换，这样的性能在Nexus One测试，比手写的for循环并使用JIT还快9倍。

**Tips：**参见 Josh Bloch 的 《Effective Java》这本书的第47条

### 谨慎使用native函数

结合Android NDK使用native代码开发，并不总是比Java直接开发的效率更好的。Java转native代码是有代价的，而且JIT不能在这种情况下做优化。如果你在native代码中分配资源（比如native堆上的内存，文件描述符等等），这会对收集这些资源造成巨大的困难。你同时也需要为各种架构重新编译代码（而不是依赖JIT）。你甚至对已同样架构的设备都需要编译多个版本：为G1的ARM架构编译的版本不能完全使用Nexus One上ARM架构的优势，反之亦然。

Native 代码是在你已经有本地代码，想把它移植到Android平台时有优势，而不是为了优化已有的Android Java代码使用。

如果你要使用JNI,请学习[JNI Tips](http://developer.android.com/guide/practices/jni.html" \t "http://hukai.me/android-training-course-in-chinese/performance/_blank)

**Tips：**参见 Josh Bloch 的 《Effective Java》这本书的第54条

### 关于性能的误区

在没有JIT的设备上，使用一种确切的数据类型确实要比抽象的数据类型速度要更有效率（例如，调用HashMap map要比调用Map map效率更高）。有误传效率要高一倍，实际上只是6%左右。而且，在JIT之后，他们直接并没有大多差异。

在没有JIT的设备上，读取缓存域比直接读取实际数据大概快20%。有JIT时，域读取和本地读取基本无差。所以优化并不值得除非你觉得能让你的代码更易读（这对 final, static, static final 域同样适用）。

### 关于测量

在优化之前，你应该确定你遇到了性能问题。你应该确保你能够准确测量出现在的性能，否则你也不会知道优化是否真的有效。

本章节中所有的技巧都需要Benchmark（基准测试）的支持。Benchmark可以在 [code.google.com "dalvik" project](http://code.google.com/p/dalvik/source/browse/" \l "svn/trunk/benchmarks" \t "http://hukai.me/android-training-course-in-chinese/performance/_blank)中找到

Benchmark是基于Java版本的 [Caliper](http://code.google.com/p/caliper/" \t "http://hukai.me/android-training-course-in-chinese/performance/_blank) microbenchmarking框架开发的。Microbenchmarking很难做准确，所以Caliper帮你完成这部分工作，甚至还帮你测了你没想到需要测量的部分（因为，VM帮你管理了代码优化，你很难知道这部分优化有多大效果）。我们强烈推荐使用Caliper来做你的基准微测工作。

我们也可以用[Traceview](http://developer.android.com/tools/debugging/debugging-tracing.html" \t "http://hukai.me/android-training-course-in-chinese/performance/_blank) 来测量，但是测量的数据是没有经过JIT优化的，所以实际的效果应该是要比测量的数据稍微好些。

关于如何测量与调试，还可以参考下面两篇文章：

* [Profiling with Traceview and dmtracedump](http://developer.android.com/tools/debugging/debugging-tracing.html" \t "http://hukai.me/android-training-course-in-chinese/performance/_blank)
* [Analysing Display and Performance with Systrace](http://developer.android.com/tools/debugging/systrace.html" \t "http://hukai.me/android-training-course-in-chinese/performance/_blank)

http://hukai.me/android-performance-patterns-season-3/

http://hukai.me/android-performance-patterns-season-2/

## 网络优化

http://hukai.me/android-performance-patterns-season-2/

## 性能优化的几大基础原理

http://hukai.me/android-performance-patterns-season-4/

http://hukai.me/android-performance-patterns-season-2/

## 稳定性优化

http://www.apkfuns.com/android-performance-optimization-summary/

# 性能测试

## 分析和调试预构建 APK

<https://developer.android.google.cn/studio/debug/apk-debugger.html>

## 使用 CPU Profiler 检查 CPU Activity 和函数跟踪

<https://developer.android.google.cn/studio/profile/cpu-profiler.html>

## 使用 Memory Profiler 查看 Java 堆和内存分配

<https://developer.android.google.cn/studio/profile/memory-profiler.html>

## 检查 GPU 渲染速度和绘制过度

<https://developer.android.google.cn/studio/profile/inspect-gpu-rendering.html>

## 利用 Network Profiler 检查网络流量

<https://developer.android.google.cn/studio/profile/network-profiler.html>

## LeakCanary

## **[redex](https://github.com/facebook/redex)**、stetho

## Systrace

## Traceview

# 参考：

## Android官方文档

[https://source.android.com/devices/tech/](https://source.android.com/devices/tech/dalvik/)

<https://developer.android.com/topic/performance/index.html>

## 其他

<http://hukai.me/android-training-course-in-chinese/best-performance.html>

<http://hukai.me/blog/categories/android-performance/>

<http://hukai.me/android-performance-patterns-season-6/>

<http://hukai.me/android-performance-patterns-season-2/>

<http://hukai.me/android-performance-patterns/>

<http://gityuan.com/2016/09/17/android-lowmemorykiller/>

<http://blog.csdn.net/carson_ho/article/details/79549417>

<https://zhuanlan.zhihu.com/p/24414378>

<https://zhuanlan.zhihu.com/p/24534940>

Android 性能优化—Android memory 参数tuning

<http://www.10tiao.com/html/431/201601/401579694/1.html>

<https://bbs.pediy.com/thread-183668.htm>

**为你的应用加速 - 安卓优化指南.md**

[https://github.com/hehonghui/android-tech-frontier/blob/master/issue-27/%E4%B8%BA%E4%BD%A0%E7%9A%84%E5%BA%94%E7%94%A8%E5%8A%A0%E9%80%9F%20-%20%E5%AE%89%E5%8D%93%E4%BC%98%E5%8C%96%E6%8C%87%E5%8D%97.md](https://github.com/hehonghui/android-tech-frontier/blob/master/issue-27/ä¸ºä½ çåºç¨å é - å®åä¼åæå.md)