资深谷歌安卓工程师对安卓应用开发的建议

with Romain Guy and Chet Haase

擅长Java语言的资深开发者们，多年以来多是工作在网页，服务器，和桌面系统等开发领域。这些领域的经验帮助他们建立起来了自己使用Java语言的模式和自己的Java库的生态系统。但是移动应用的开发却和这些领域的java开发有着天壤之别。优秀的安卓应用开发者需要考虑到移动设备的限制，重新学习怎么样去使用java语言，怎么样去有效地使用实时环境和安卓平台，然后写出更好的安卓应用程序。

Transcription below provided by Realm: a replacement for SQLite that you can use in Java or Kotlin. Check out the docs!

Loading...

So, this talk is
basically structured around

About the Speaker: Romain Guy

Romain是谷歌的安卓工程师。在加入Robotics之前，他在安卓Framwork组参与了安卓1.0到5.0的开发工作。他现在又重新加入了安卓的新UI和图形图像相关的项目。

@romainguy

About the Speaker: Chet Haase

Chet 也是谷歌的工程师。 他现在是安卓UI Toolkit 组的组长，他擅长于动画，图像， UI控件和其他能带来安卓更好的用户体验的UI组件的开发。他还擅长撰写和表演喜剧。

@chethaase

介绍 (0:00)

本次演讲是以安卓平台组写的近10篇文章为基础的。所有的文章都能够在Medium网站上看到，文章的第一部分请看这里. 今天我们会讲到这些文章里面的一些东西，如果你对特定的话题感兴趣或者想深入了解它们，请去阅读原文。

为什么移动开发如此艰难？ (1:47)

有限的内存 (1:47)

我们发现谷歌公司里面的应用开发者有一个大问题，他们对口袋里每天都携带着的安卓手机的本质都有些误解。这些设备内存，CPU的处理能力和电池的待机能力都非常有限。开发者们必须理解你们的应用不是在设备上唯一运行的应用。内存是非常有限的资源，并且被整个系统共享着。我所在的Android平台组非常小心地对待这个问题，这也是为什么有的时候我们建议的一些规则看起来会有一点极端的原因。我们需要有全局的眼光来看待这个问题，因为系统会同时运行20、30、40个的进程。当你只为单一应用开发的时候，牢记这些限制是比较困难的。

今天我们手上的设备往往会有1-3GB的内存，但不是所有的安卓设备都有这么多的内存的。安卓阵营中有14亿的设备，其中许多是两三年前的产品。事实上，他们才是着14亿设备的主力军。大部分的设备不是在美国和西欧，而是在中国和印度这样的国家。在这些国家里，安卓设备必须便宜才能吸引更多的消费者。

安卓在从4.3到4.4的演进过程中，我们使出浑身解数才使安卓系统能够成功地在内存受限的系统上运行起来。很长时间以来，因为人们想在便宜的500MB内存的设备上使用安卓系统，姜饼系统成了他们唯一的选择。当然，现在的情况已经大不相同，姜饼设备已经差不多消失殆尽了。但是，因为现在的安卓版本有着庞大的大内存消耗的应用群，这就需要更强大的框架和平台。

这是一件你必须持续思考的事情，但是这很难，因为它不总在你优先考虑的事情的范畴内。你可能知道Knuth的名言“过早的优化是一些罪恶的根源”。虽然我同意他的观点，可是当你写完你的应用，然后打算从应用中再减少50MB的内存的使用，也是一件非常困难的事情。我们不是说你需要毁掉你的应用系统架构或者牺牲你的测试，但是每一次你能做些改进来改善你的内存使用情况的时候，请马上动手。

为了整个设备有更好的整体用户体验而开发。当你的应用很大的时候，你会导致系统杀掉别的应用，这样你就影响到了别人应用的用户体验。当别人的应用不注意内存使用情况的时候，你的应用体验也不会好。你们需要互相交流然后找到和谐共存的方法，请牢记这条建议。

如果系统需要杀掉除了你的应用之外的所有应用，那么当你的应用退出以后，其他应用需要重新建立他们可运行的环境。这样你的应用会看起来像一个恶意的应用。如果你想要让用户重入你的应用时有一个良好的体验的话，请尽你所能将你的应用保持在后台运行并且消耗最小的内存。

在有些应用当中，实现内存回调接口是非常管用的。你能在API文档中非常容易的找到`onTrim`的内存回调函数，并且能实现多个级别的Trim操作。基于你的Trim操作的级别，你可以释放一些缓存和bitmaps，也可以退出一些Activity，或者其他的任何能帮助你留在后台运行的措施。

CPU 处理器 (8:07)

移动处理器显然比桌面和服务器的处理器能力要慢很多。虽然从外界不容易察觉，但是移动处理器大部分时候都是处在过热降频保护的状态。这意味着尽管CPU的频率很高，但是仍然不能像桌面和服务器的处理器一样快。当诸如用户在屏幕上拖拽的时候，CPU可以达到它的最快处理速度。可是在其他的情况下，CPU处理器是跑在待机频率上的，不然的话，电池不能保证用户的基本待机时间。

如果你买的是2GHz的台式机或者笔记本，CPU大部分时候是能跑在那个参数指标上的。可是你如果买的是一个2GHz的手机，CPU只能有时候跑在2GHz上。如果你买的是八核的手机，硬件上你是有八个处理核，但是他们大部分时候不会一起工作。基本上，手机盒子上的参数表和你实际上能达到的参数是有不一致的地方的。这主要是为了降频保护CPU，提高电池待机时间和减少发热量。

GPU 图像处理器 (10:10)

和CPU一样，GPU也有保护降频的功能。纹理加载（Texture uploads）的开销特别大。所有bitmap的操作或者结果是bitmap的操作都会被加载到GPU。举个例子说，当你在绘制路径时，这些东西会转成bitmap并且作为纹理加载到GPU，这时系统的开销就会特别大。你的性能瓶颈就可能在这。

填充率和分辨率之间也是相关联的. Nexus 6P的分辨率非常高，换句话说，屏幕上有许多的像素点要填充。但是GPU的性能带宽却跟不上。系统为了填充所有的像素点而超负荷的工作。你的要求越高，系统需要的时间就越长。

一个提升UI 性能的技巧就是避免过度刷屏。你需要系统填充屏幕上每次像素的次数越多，当屏幕越来越大和分辨率越来越高的时候，情况就会越来越糟。如果你有一个Playstation 4或者Xbox One的话，他们在运行那些1080p的游戏（每秒30帧）时十分卡顿。在我们的手机上，我们有更高的分辨率，而且我们以每秒60帧的速度完成所有的事情；我们尽可能的完成更多的事情并且消耗更少的电量。这就是为什么我们对图片的处理性能有着诸多的要求以及对应用优化的建议，因为我们没有你想象的那么强的处理能力和电池电量。

内存 = 性能 (12:29)

如果你有大的应用，那么就会发生更多的内存页的置换，更慢的内存分配。实际运行中，系统需要遍历来决定新的对象可以放到内存何处，这需要花费更多的时间。回收也会需要花费更多的时间；每次需要内存的时候都要遍历更多的东西。

整体上内存回收机制也被更多地触发。在我们的演讲中，我们详细阐述了内存回收是如何工作的，而且讨论了ART和Dalvik的改进。详情请看 上面的视频 at 13:06

低端设备 (19:40)

你口袋里面的设备保证比你用户的设备要快很多。你认为的那些老旧设备并没有消失。 A 它们依旧在用户的口袋中，而且用户打算尽可能长地使用它们。B 那些老旧的设备虽然慢但是便宜。这意味着它们能吸引哪些不能够购买最快设备的人们。

流畅的帧频率 (21:01)

找到一个流畅的帧频率是十分困难的。你只有16毫秒的时间去完成所有的事情。这些事情包括触屏事件的处理，计算，布局，绘制帧，然后交换缓存。16毫秒意味着每秒刷新60帧，这是我们在安卓系统上要求的，因为我们是V-Sync。我们不希望屏幕花屏。花屏在有些游戏中你能看到，因为它们在一秒内同时有两个buffer。那个看起来太可怕了，我们不想让你这么做。这就意味着如果你仅仅多花了一点时间，哪怕17毫秒，我们就会跳过一帧。然后跳过一次V-sync，这样系统就不是60fps了，而是30fps。我们叫这种现象为Jank。

系统在60fps和30fps中来回跳跃是件非常糟糕的事情。这会让你的用户觉得你的应用非常janky和不一致。也有可能你的应用会一直表现糟糕，一直都是30fps。这就是许多游戏当它们认为做不到60fps的时候，它们就一直都是30fps。

实时运行： 语言 ≠ 平台 (22:48)

几周前，有人问我 “当有新版本的JDK发布的时候，你一般做什么？”。 然后我意识到他们并不理解语言和实时运行环境是不一样的。当有新的JDK发布的时候，我基本不关心。它和安卓运行时一点关系都没有。我们使用同一种语言，但那不意味着我们在运行的时候是一样的。

当人们使用Java语言的时候，有三个方面的因素构成了整个java体验。Java编程语言本身，实时运行环境（在有些server环境中叫HotSpot）然后是硬件设备。

对于那些擅长服务器的人来说，服务器的实时运行环境提供诸如移动，压缩收集器的功能，这些都意味着临时的内存分配带来的系统开销非常小。这些事情和移动指针一样快。在服务器环境中，的确如此。然后你会理所当然的认为移动设备有一样快的处理器，一样大的内存，所以处理能力就应该和服务器一样无限制。

在安卓系统中，情况是非常不一样的，特别是你习惯于无限的系统资源和完全不同的实时运行环境的时候。我们有Dalvik和ART。我们没有压缩，这意味着当你分配一个对象的时候，这个对象就会存在堆里面。堆会碎片化，也会使得寻找空余的内存空间变得开销更大和更困难。在ART环境中，我们有空闲时的压缩。ART从来不能在应用是前台运行的时候压缩堆空间，但是当它在后台运行的时候是可以压缩堆空间

的。在进程的生存周期里面，有的时候压缩是会发生的，这会帮助系统寻找空闲内存。

压缩在本地代码上的作用更为关键。如果你的应用使用JNI，并且分配一个指针给一个java对象的时候，那个指针通常会被系统认为一直有效。在ART环境中，如果你的堆被压缩了，那指针就会变成野指针。如果你使用JNI，在开始的时候你就必须对这种情况特别小心。

UI 线程 (26:56)

安卓是一个单线程的UI系统。技术上，你可以有多个UI线程，但这会带来很多麻烦。正如你担心的那样，所有的UI控件都是在同一个线程里。因为是一个UI线程，所以你任何阻塞UI线程的操作都会带来性能上的影响，jankiness（闪屏）和不一致性。在UI线程上分配内存就更糟了。如果你有后台运行的线程分配内存，当虚拟机VM阻塞所有的线程（包括UI线程）十几毫秒的时候，你就会跳过一帧。这样，你就只有30fps，这会非常糟糕。

存储 (28:53)

存储的性能没有一个定论。有时候人们把数据存在SD卡上，这就会有各种各样的性能标准。当存储设备快要满的时候，即使在同一个存储设备上也是有不同的性能体现的。但是如果你的应用是依赖每个测试设备的存储速度的话，那总归是不好的。写flash存储的时候也意味着控制器需要做很多事情，因为它需要收集信息，记录哪些空间已用和哪些空间没用。换句话说，如果你的应用在后台做大量的IO操作的话，你会把这个系统拖慢。你应该有这样的经验，当Play Stoer 在后台安装应用更新的时候，你的设备会突然间感觉慢了起来。因为它在后台做磁盘写操作，所以前台的应用读它们自己资源也会变得很慢。

存储的大小也是各式各样的，所以不要让你的应用对特定的存储大小有依赖。APK的大小也很重要。你应用中资源的大小会影响到你的启动时间。

优化你的应用资源的办法有很多。现在在老的版本的安卓上，也有了矢量图。如果可以的话，请使用安卓的SVG库。虽然对于图标来说不是太合适，但的确能对你的应用起好的作用。你也可以使用WebP格式，这会比PNG省下20%-30%的存储空间。PNG Crush也是一个很好地离线工具。APT做了一部分PNG crush的工作，但是还有很多工具会做的更好，更有效。

网络 (28:53)

你使用的网络肯定比大多数用户的网络要快。他们可能还在用着2G网络，并且流量很贵。所以你的应用不应该依赖持续的网络连接。也许你应该让你的应用能够智能地下载它需要的内容或者你的应用不需要依赖网络下载的内容就能使用，真正的内容可以晚点再去下载。

开车去Utah的路上，你可以测试在糟糕的网络环境下，你的应用的表现如何。或者你可以用一些模拟糟糕网络的工具。所有这些应用的测试都是手工进行的。

每一台设备都是一个村庄 (33:31)

每一台设备都是一个村庄，这意味着每个在村庄里的人都需要共同努力来维护一个好的用户体验。你可以让这个体验特别差，也可以特别好。如果你的应用想成为好的体验的支持者，你可以试着在你的manifest里面关闭Service和broadcast receivers。在代码里面，当你的应用发现用户打算使用你的应用的时候，你才动态的开启你的services和receivers。邮件客户端就是一个好的例子，当用户安装邮件客户端的时候，所有的一切都应该处于关闭的状态。一旦用户点击了该邮件客户端并且加入了账户的时候，你才开启你的services和receivers。然后下次用户重启设备的时候，Framework就会记录下来，你的服务就真正的运行起来了。这个方法很简单，但也很重要。因为你的services虽然运行起来了，但是没有人用，你会对别人带来不必要的坏的影响。

另外一个现象叫做”公共地带的悲剧“： 每个人都认为他自己的应用是最重要的。如果每个人都是这样的观点，那么每个应用都会非常大而且尽可能被激活，这样设备会承受不了。勿以恶小而为之，勿以善小而不为。

技巧与建议 (35:53)

了解你使用的语言 (35:55)

开发者们可能有了很多年Java的经验，但是却不能最好地利用语言来发挥出移动设备的最大性能。

不要使用Java的序列化

序列化本身是非常有用的，这不是我们现在讨论的话题。如果你用过序列化的话，你会发现不是太好用，因为你需要产生UUID，而且它还有限制。序列化也会慢，因为它用到了Reflection。

使用安卓的数据结构

其他场合Java开发者常用到的一些集合类和方法在Android上也许就不合适了。

Autoboxing 和 Primitive Java 类型的替代品在Java领域里使用的非常广泛。集合的迭代器也是非常常用的。在Android平台上，我们特别创建了一些集合类来替代这些模式。对于Key来说，我们使用基本类型，所以如果hash表里面是一个整形作为key的时候，我们没有autoboxing。你可以以使用SparseArray类，同样的，ArrayMap和SimpleArrayMap也是HashMap的替代者。

为了避免java package中基础类型的额外开销，使用Android的数据结构类型是个不错的选择。HashMap里面的每个条目都会多用4倍的内存空间，像你的int类型一样。你放在hashmap里面的内容越多，你浪费的内存资源也就越多。看看你现在已经使用的数据结构，如果你打算在某些情况下重写你自己的数据类的话，不要犹豫。

小心使用XML和JSON

他们相对来说太大了，对于你的某些应用来说，他们或许太结构化了。如果你使用有线设备上的数据格式会更加简洁，但是至少你能在你做网络传输的时候gzip这些数据。当然，如果你打算序列化你的对象到磁盘上，你可能需要找找其他的替代方案了。

避免使用JNI

有些时候你需要JNI，但是如果不方便就不要使用它了。JNI有些有趣的事情：每一次你越过JAVA和Native的边界的时候，我们都需要检查参数的有效性，这会对GC的行为有影响，而且带来额外的消耗。有的时候这些消耗是非常昂贵的，所以当你使用了很多JNI的调用的时候，你可能在JNI调用本身上花的时间比在你native代码执行的时间都要多。如果你有些老的JNI代码，请仔细检查他们。

有一件你能提高JNI效率的事情就是尽可能的把多次调用集中到一次。避免在JAVA和Native中间来回调用，一次搞定。例如，在Android的Graphic Piplline中，我们在每次调用JNI的时候传入了尽可能多的参数，从而避免了调用JNI的时候，JNI从JAVA对象中抽取参数的开销。我们使用了基础类型，避免了使用奇怪的对象，我们传入的是Left，bottom和right参数，所以我们不需要又返回到JAVA层了。

基础类型 vs. 封装后的基础类型

请使用基础类型来代替封装后的基础类型。一个不那么明显的事实是：如果你使用那些集合类并且比较它们是否相等的时候，我们每次都会做autobox。每次都需要分配一个对象去使用，因为它们会被强制转换成boxed equivalent。这里有个例外，你可以使用big boolean，因为它们只有两个。

避免使用反射（Reflection）

比避免使用JNI更进一步，我需要如我建议避免使用JNI一样指出来，animation

framework使用了JNI来避免使用反射。这样当然多了双重内存分配的开销，因为我们需要分配这些对象并且每处都要autobox他们，同时从Java运行环境发起的函数调用才用的内部机制的开销特别大。

小心使用 finalizers

内部我们只在很有限的情况下才用。关于finalizers有个事实不太明显的是：它需要两次完整的GC才能收集完所有的事情。如果你在一个finalizer里面放置某些assert来收集信息的话，我们需要运行两次完整的GC才能回来处理。有些时候这是必要的，但是在其他的一些情况下，把这些处理放在离finalizer之外的近点的地方会更方便。不然的话开销太大了。

网络 (47:19)

不要过度同步

正如前面描述的那样，你的用户的网络可能不那么好，或者他们的网络会很贵。而且，你会给系统带来负担。也许你认为你的应用需要尽可能快地得到那些数据，但是事实上是你会给系统带来很多的负担，仅仅为了保持你的应用处于激活状态。

允许延时下载

这在信号不好的网络和收费很贵的网络下十分重要。你可以把数据打包起来。把所有需要下载的东西收集在一起，然后做一次同步。

谷歌云消息 Google Cloud Messaging

GCM收集了很多东西。他会使用传送层来和后台服务器传送数据，所以你也许可以重用这个系统来避免自己重复工作，也避免了每个应用都创建自己的socket。

GCM 网络管理（Job 调度）

这也是个很好的东西可以利用起来。Job 调度可以让你把你的东西打包起来。你可以说 “我想在空闲的时候才使用这些事务，或者当我被激活的，或者当我在wifi连上的时候”，然后在普通的时间间隙里面收集这些事情。这会更有效，对用户也不会太突兀。

不要轮询

永远不要轮询

只同步你需要的

你知道你的应用里面发生了什么，所以仅仅只要同步当前应用需要的数据，而不是把所有可能需要的数据都同步下来。

网络质量 (50:05)

不要对网络有任何的假设。为低端网络开发，然后保证你的应用在低端网络下测试过。即使是你在使用模拟器，你也需要保证你在我们称为“糟糕的网络”的环境下测试过。在这些情况下，你应用的表现会让你大吃一惊的。

数据 & 协议 (51:13)

如果你拥有服务器，做所有能帮助到设备的事情。比如改变数据格式，改变发送数据的类型。设备来告诉服务器它打算浏览图片的大小，也是个好方法。我们看到过内部的应用接收到了比屏幕大小大4倍的图片，然后在设备端重新处理图片的大小。这个工作明显服务器来做比较合适。

使用缩小和压缩的算法。gzip是个好的选择。如果你能在HTT上传送GZIP的数据，请这样做。这会很有用，特别是在糟糕的网络上。GCM也可以帮忙。它会帮助你保持连接，所以，一个好处就是使用它会有更短的延时，因为不用在每次连接的时候都做一次握手而且在延时方面，移动网络是出了名的糟糕的。

存储 (52:21)

不要写死你的文件路径

路径会改变的，如果用户想把它们存到别处呢?

仅仅固定相对路径

你知道你的数据相对你的APK的路径，这是正确的，安卓有APIs得到那些路径。你不需要做hard code的事情，你只需要坚守和你APK实际安装位置的相对路径。

使用存储缓存来处理临时文件

有APIs可以调用，请使用它们。

简单情况下不要使用SQLite

SQlite的消耗在某种程度上也是很大的，所以如果你只需要些简单的方式，比如 key-value存储会更加合适些。

避免使用太多的数据库

数据库整体上开销是很大的。也许你可以试试只用一个数据库，然后服务于多个不同的用户。

让用户选择内容存储位置

当用户有可移除的存储设备的时候这些尤为重要。或者他们可以使用adoptable storage，这是在棉花糖版本上的新的方法。如果用户打算采用新的存储设备，然后把数据都迁移到它上面，你的应用应该允许他们这样做，这样你的应用就不会乱掉了。

问和答 (54:00)

问：你提到了一个平滑的策略是降低你的帧频率，有可能一个应用说:“我需要30fps而不是60fps吗？”

Romain：某种程度上。有一些你可以使用的API来达到和屏幕同步的目的。你可以每隔一帧同步一次。在普通的应用中，这会是很困难的，除非你知道你会因为外面各种各样的设备而有许多的工作量。但是，如果你使用的是OpenGL或者Canvas，你又有自己的线程做渲染，并且你对渲染线程有完整的控制权的时候，情况就不一样了。在这种情况下，你可以控制你的帧，你可以等待，然后你记录V-Sync，诸如此类的动作。如果你想深入的了解这些高级的技术细节，你可以看看那些游戏开发者的文章，他们在做类似的事情。

Chet：一般情况下，你可以尝试60fps，然后你可以使用一个叫GPU Profile的工具，这个工具在Android Studio上就有。工具里有一个彩色的显示条，你需要保证你持续的呆在绿线以下。你可以在developers.android.com 查到工具的详细信息。

Romain: 还有，我们经常用 Systrace。Systrace是一个底层的，轻量级的，系统层面的监测工具。如果你的应用在任何地方有性能问题，它不需要嵌入在你的代码中。Systrace不但能显示你在那你花费了时间，而且还能显示你的线程是否被调度，CPU的运行频率，你是不是被从一个CPU转移到另一个CPU上（这也会影响到你的性能），是否后台线程导致了锁定，或者优先级倒置。文档在developer.android.com上也能找到。这也是个很有用的工具。

问：有一个类叫‘android.os.memfile’。 它和linux 共享内存联系在一起。当我使用它的时候，我能从linux 内核中得到300-400MB的内存来共享。所以，我使用它来分享拍摄的视频并且存到了临时区域。同时我把它设为 non-purgable. 因为使用了400MB的内存，Android总会参与进来刷新内存。你对使用这个技术来存储奇怪的内存有什么建议吗，或者不要使用共享内存?

Romain：这是个非常有趣的问题。我没有什么想法。我也不知道系统有这样的行为，也许你应该问问内核组来理解这里的行为。如果是如设计一样，那么后果是什么。对不起，我没有一个更好的答案。

问：我想指出当Android在切换Activity状态的时候，会使用共享内存。他们发现如果

你一直监视着你的Android设备中共享内存的使用状态，你可以准确地猜出Activity是处于什么状态，如：是从onResume到onStop，所以这看起来像是个安全漏洞。显然，安卓再每次做Activity的状态转换的时候都使用了共享内存，有时候是12bytes。所以实时监测共享内存的状态，你就可以知道Activity是在什么状态。

Romain: 如果你一直监视着内存状态，你可能还可以知道其他的发生在设备上的事情。比如看着屏幕。但是，很高兴你能分享这点。

问：你之前讨论到应该避免使用JNI，也需要避免使用Reflection。然后你提到你在动画实现中忍受了许多开销来避免使用Reflection。我想知道如果使用了Reflection，会在动画对象中发生些什么呢?

Chet: 我们没有用Reflection，我们用的是JNI。JNI里面有Reflection的代码。它是这样工作的：他会调用到JNI说，“我需要一个方法”。所以，当你想使用一个动画的属性叫做“foo”。你传入了一个串，然后说“好吧，我想找到一个设置的方法叫做setFoo”。进入JNI层后，看起来JNI层会说：“有这样的方法的签名吗？”。如果JNI找不到，就会返回false然后就会使用Reflection。我认为Reflection的代码应该永远都不被调用到。所以JNI需要返回false。我以前也使用过Reflection，只要是我写的代码，我会使用已经存在的接口来保证没有其他奇怪的事情发生。然后，在使用Reflection的时候我还会有些backup的机制。但是我不认为Reflection需要这样用。这种情况下可以使用JNI。在有些代码中，也许不太明显，但是如果你找到代码正确的地方，你会发现一个情况是“JNI够用吗？如果他够用，就使用JNI好了。”

Romain: 当你使用JNI的时候，你其实可以有效地访问到所有的成员和方法，这就是Reflection做的事情。我们发现使用JNI会比Reflection快上很多。

Chet: 而且部分原因是内存的消耗。因为至少算上运行的开销，JNI是不会额外再分配内存的。通过任何机制实现方法查找都不简单。但是，我们只会在你第一次调用的时候做方法查找。所以，当你创建动画对象的时候，第一次你运行的时候，就开始寻找setter或者getter，然后缓存它们。这样同样的情况不会再次发生。

Romain: 另一个事情是当你通过Reflection调用一个set alpha的函数的时候，你会在方法对象上调用该方法，这需要一个对象。所以当你需要传入一个Float时，你需要分配内存来封装（boxing）你的Float。但是如果你用的是JNI，你就不需要boxing。

Chet: 当然还有第三种机制，如果你打算习惯它的话，就是属性（properties）。这就是我们为什么很早前就加入属性（properties）。为一些特殊的视图我们创建了属性，alpha属性，translation X属性， rotation X属性等等。他们直接使用setters。所以当你使用属性的时候，他直接调用了静态的属性对象。你在视图的实例中传递它们，它会调用视图的seter，这样是开销最小的方法。

问：你刚才说你们不太关心Java的版本发布，因为Java语言和实时运行是分开的。但是我还是很好奇你们如何看待 retrolambda，它让你使用lambdas，这是Java 8 的功能，但是它把Lambdas编译成二进制代码。而且作为dexing的一部分，它把他们重编译成匿名类。你有什么建议吗?

Romain: 我知道retrolambda，而且我喜欢它。我觉得它太棒了。使用它之前，我会反编译retrolambda的结果看看发生了什么。因为使用lambda，我敢肯定你会有些讨厌的惊喜。我非常肯定它们在某些地方会变成匿名类，所以在你会遇到大量的内存分配，分配情况取决于它们是如何介入的。例如，如果我在一个循环中传入一个lambda，内存分配会发生什么呢？所以，我会去看看反汇编代码，然后在我决定前看看发生了什么。你知道，匿名类可以被缓存，或者它会在每次使用的时候分配，我不知道具体会发生那种情况。我需要看看。作为一个软件工程师来说，在决定如何使用前，尽量的去理解背后到底发生了什么是非常重要的。因为这样做，就不会在之后的开发中遇到讨厌的惊喜。当然，在某些情况下，如果你是在用户点击一个按钮时使用了lambda，并不重要，对吗？当你点击一个按钮时，性能不是个特别大的问题。

问：我想知道的是什么时候你们会支持Java 8， 然后你不需要把他们编译成匿名类。

Romain: 不知道，实话实说，这几个月我们确实讨论了不少关于桌面系统的JAVA 8的事情。我使用过streams，parallel streams和lambdas。这些都太棒了。代码看起来特别棒，写起来都特别有趣。但是当你意识到在某些情况下，他们确实会比老的循环更慢些的时候，感觉就不一样了。再说一遍，小心一些，尽量去理解你现在正在做的取舍。关于Java 8语言什么时候会被Android支持，我不知道，因为我们没有讨论过这个事情。而且即使我知道，我也不能告诉你。

Chet: 我有必要提一下，我最喜欢的一个工具就是DDMS里面的Allocation Tracker，它现在可能是在Monitor的什么地方。反编译二进制代码是一个好的方法，这样可以看看到底编译器干了什么，但是运行的时候看看发生了什么也是个好方法。当我们刚开始开发动画框架的时候，我们使用Allocation Tracker去找到我们在每一帧的时候分配了些什么，然后消除它们。所以，如果你用retralambda，你需要确定在你应用的关键点，你不要分配那些从外部看不需要的内存。

问：我们日常的工作都是feature驱动的，对吗？内部，我们如何去维护一个性能的标杆？你们有专职的QA来跟踪你们的代码，然后你可以看栈的trace，然后时刻提醒你，或者作为一个feature的开发者在你们提交给QA前你们有什么标准吗?

Romain: 这是个混合的问题。对于那些非常关心他们应用的性能的工程师来说，这是个他们开发过程中非常小心的问题。也有QA会关心这个话题，但是这几年来，

Android组写了很多自动化的测试用例来测试性能。例如，拿Butter项目来说，图像组为每一个CL做了一个jank的dashboard。他会运行一些像在Gmail中滚动滚动条的用例，然后计算我们错过的帧。每一次数字的变动，我们都会收到一封邮件，然后有人就会受到批评。当然应用的开发者会先收到批评，然后他们会发给我们框架组说："你们太烂，你们的东西太慢"。然后我们会回复它，这样会来来回回讨论好长时间。

Chet: 我们也会写出许多工具并且改进它们。Systrace就是一个我们内部正在使用并且持续改进的一个很好的工具，之后我们提供给所有人使用。并且我们持续迭代地开发它。这样我们就像最近做的事情一样，在一些不明显的情况下，给你一些关于你的问题的提醒和信息。

Romain: 事实上，大部分的工具你可以通过工程的UI看到很多信息。所以诸如Hierarchy Viewer， devel-draw，debug tool 和 GPU profiler都在设备上可以运行。所有的这些工具都是我们内部需要的，所以我们开发出来并且提供给所有人使用。当然，还有很多的事情可以做，但是关于性能最重要的事情就是你们能写出自动的测试用例来捕捉它们。这是非常困难的一件事情。我们暴露了一些内部的计数器，我想我们有文档描述它们。在adb shell的dumpsys帮助下，你可以获取其中的一些信息，帧的时间戳，janks的次数，特别是在棉花糖和棒棒糖中我们加入了更多的方法。你可以做的事情就像UI automater一样，你可以创建一个用例，驱动你的应用，然后输出这些计数器看看能否告诉你些什么有用的信息。更有效的方式是有个dashboard。它抓取了这些命令的输出，然后给你画一个特别好看的图表。另一件事情是，当你做些事情的时候想想你的性能。说起来简单，做起来难，评判起来更难，理解你在量化什么还要困难一些。但是，这是唯一的途径。

问：你提到不要使用序列化。你能再详细的描述一下吗？在Activity和Fragment间使用Serializable和Parcelable传输数据包含在里面吗？

Romain: 我们是在讨论Serializable的接口。Parcelable是Android提供的Serialization的一个变种，是非常有效的，但是仅仅用来进程间的数据传输。现在我们有新的方式了，搜索"persistent parcel"，你就可以找到它了。但是对了，我们讨论的是Serializable接口。

问：你之前提到flat buffers。有一种说法是如果网络请求需要很长时间的话，节省100多毫秒不太重要。

Romain: 我之前给的例子比较有意思是因为他用了JSON作为磁盘上文件存储格式。这里的想法是你已经缓存了数据了，所以你已经承受了网络的开销，所以可以优化的地方在你多次读取这些数据的时候。在这种情况下，使用更快的方式是有意义的。但是你是对的，如果解析JSON需要10毫秒，解析你的flat buffer只需要5毫秒，

但是网络传输需要200多毫秒，这样的优化还有作用吗？显然，我希望你的应用在后台线程处理所有的网络操作。所以这里讨论的更多的是UI的延时，而不是别的。如果你有什么东西是经常访问的，你当然需要在flat buffer 这样的地方里面寻找它。

还有一个Cap'n Proto，也是protobuf v2的开发者开发的。他写了一些类似flat buffer的东西，但是更加深入。因为它可以被作为有线传输格式，可以被用作RPC机制，而且我相信它有Java 封装层。这就回到我们之前的坚持，当你考虑性能的时候，你需要量化标准。确定你修改的是正确的问题。如果你修改的地方不是个问题可能还无所谓。作为我们来说，在你的应用里面到底什么会是个问题更难。因为这完全依赖于你的应用。我们只能分享我们在应用中一次又一次看到的问题，再一次强调，这不意味着你的应用也有同样的问题。但是真的，如果你在磁盘上做serialization，看看其他的格式吧。总而言之，找到你最好的方式。