

AUG 11TH, 2015 | [COMMENTS](#)

Android性能优化典范 - 第3季

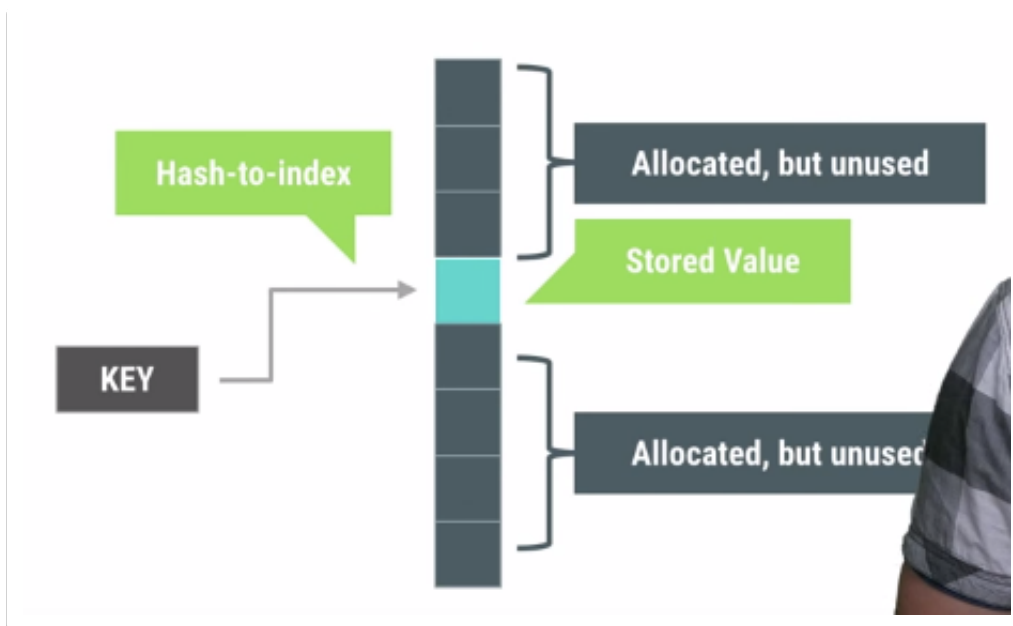


[Android性能优化典范](#)的课程最近更新到第三季了，这次一共12个短视频课程，包括的内容大致有：更高效的ArrayMap容器，使用Android系统提供的特殊容器来避免自动装箱，避免使用枚举类型，注意onLowMemory与onTrimMemory的回调，避免内存泄漏，高效的位置更新操作，重复layout操作的性能影响，以及使用Batching，Prefetching优化网络请求，压缩传输数据等等使用技巧。下面是对这些课程的总结摘要，认知有限，理解偏差的地方请多多交流指正！

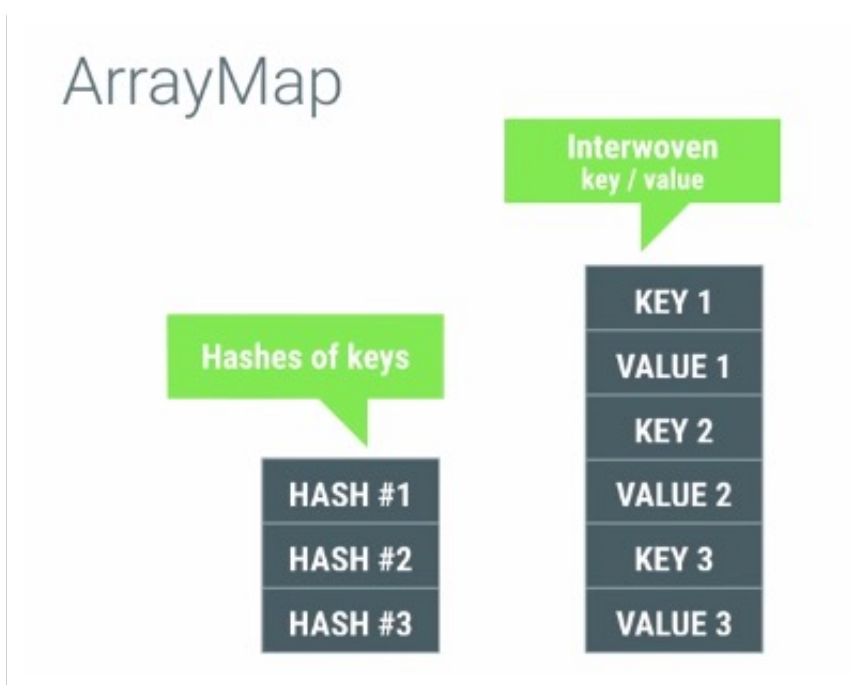
1)Fun with ArrayMaps

程序内存的管理是否合理高效对应用的性能有着很大的影响，有的时候对容器的使用不当也会导致内存管理效率低下。Android为移动操作系统特意编写了一些更加高效的容器，例如SparseArray，今天要介绍的是一个新的容器，叫做[ArrayMap](#)。

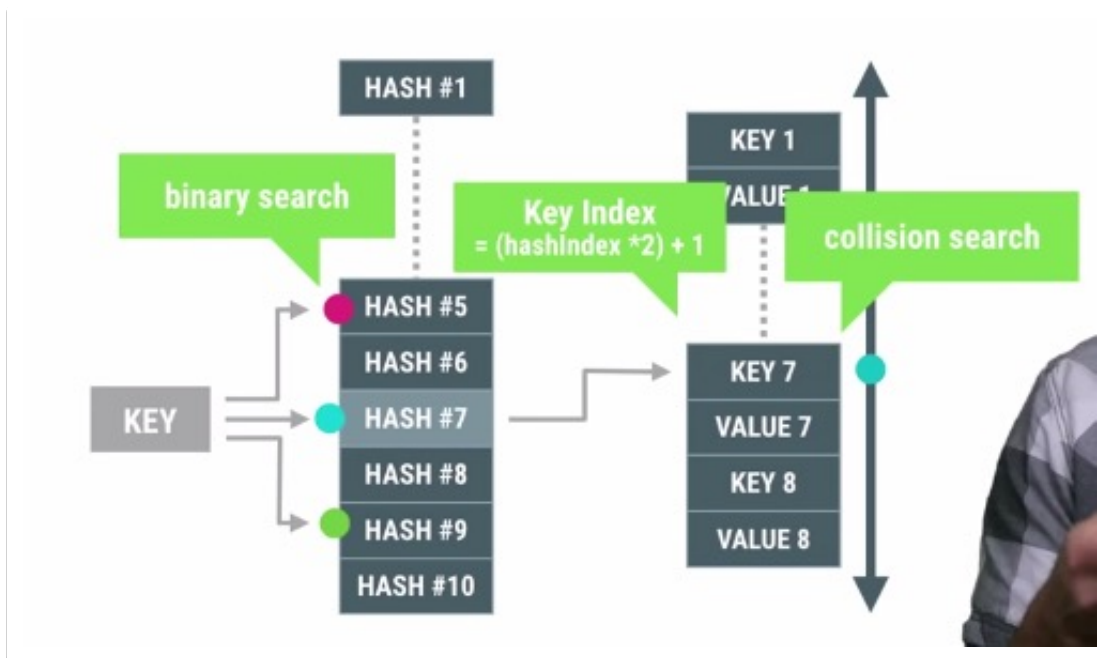
我们经常会使用到HashMap这个容器，它非常好用，但是却很占用内存。下图演示了HashMap的简要工作原理：



为了解决HashMap更占内存的弊端，Android提供了内存效率更高的**ArrayMap**。它内部使用两个数组进行工作，其中一个数组记录key hash过后的顺序列表，另外一个数组按key的顺序记录Key-Value值，如下图所示：

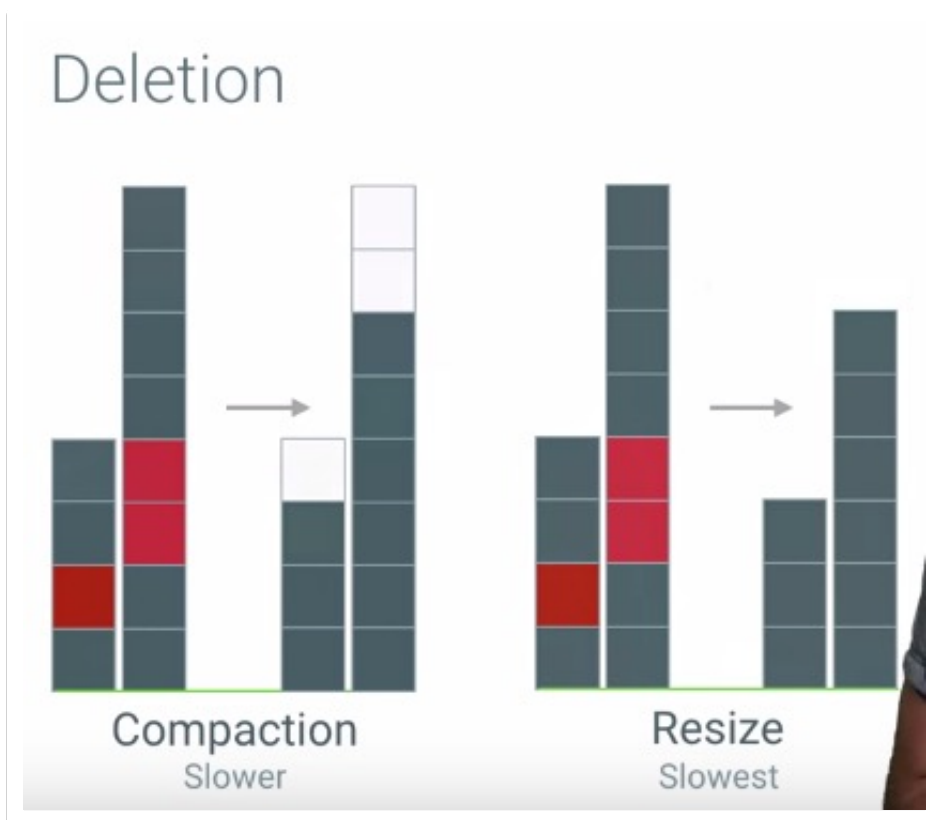


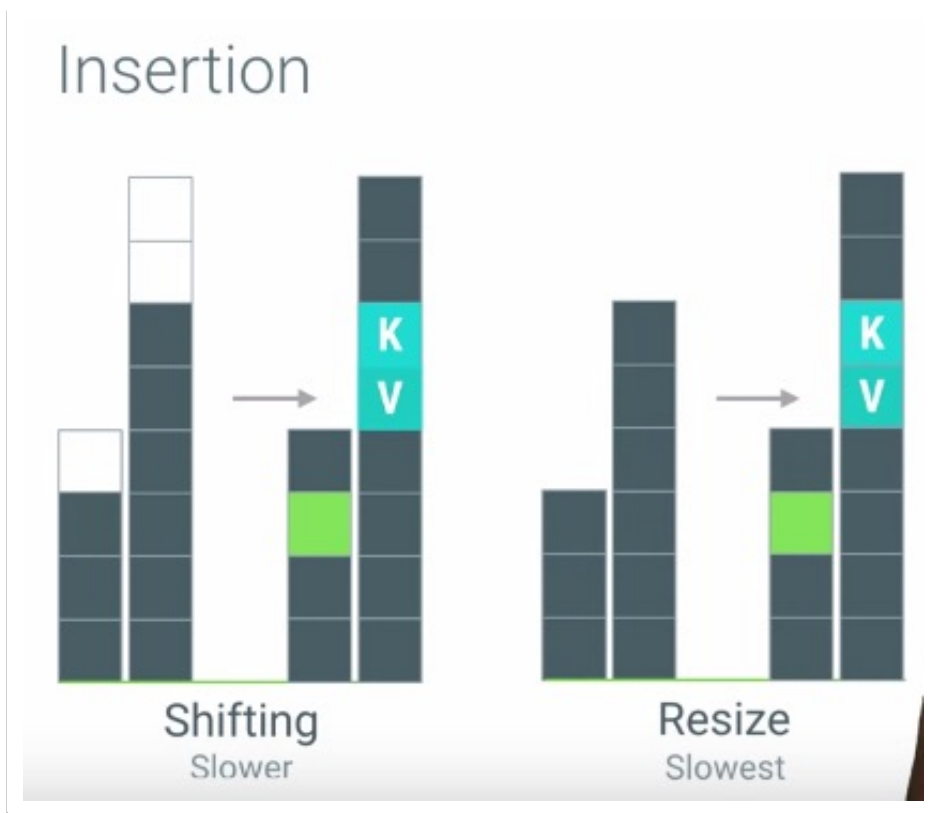
当你想获取某个value的时候，ArrayMap会计算输入key转换过后的hash值，然后对hash数组使用二分查找法寻找到对应的index，然后我们可以通过这个index在另外一个数组中直接访问到需要的键值对。如果在第二个数组键值对中的key和前面输入的查询key不一致，那么就认为是发生了碰撞冲突。为了解决这个问题，我们会以该key为中心点，分别上下展开，逐个去对比查找，直到找到匹配的值。如下图所示：



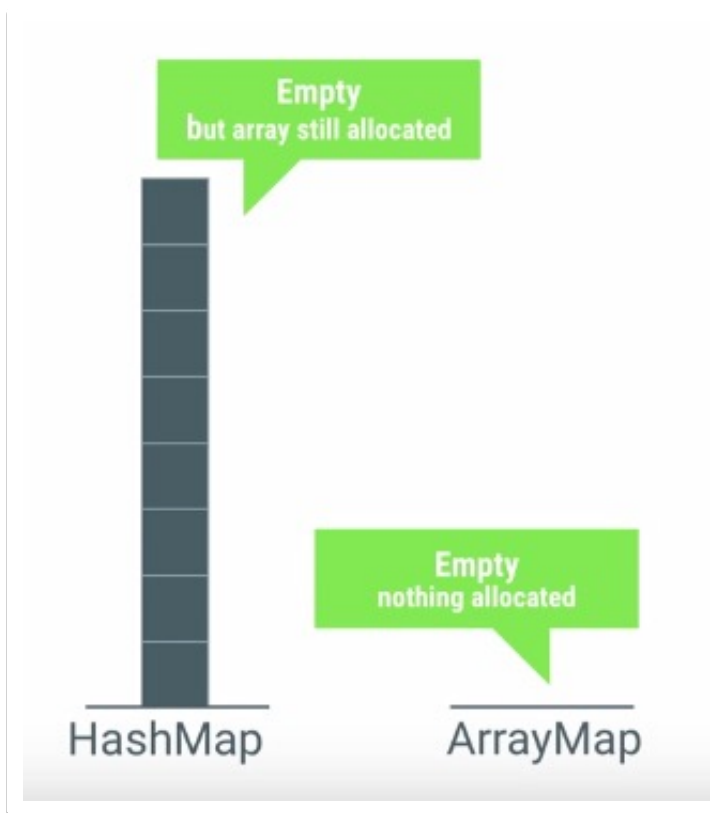
随着数组中的对象越来越多，查找访问单个对象的花费也会跟着增长，这是在内存占用与访问时间之间做权衡交换。

既然ArrayMap中的内存占用是连续不间断的，那么它是如何处理插入与删除操作的呢？请看下图所示，演示了Array的特性：





很明显，**ArrayMap**的插入与删除的效率是不够高的，但是如果数组的列表只是在一百这个数量级上，则完全不用担心这些插入与删除的效率问题。**HashMap**与**ArrayMap**之间的内存占用效率对比图如下：



与**HashMap**相比，**ArrayMap**在循环遍历的时候也更加简单高效，如下图所示：

```
// ArrayMap
for(int i = 0; i < map.size(); i++){
    Object keyObj = map.keyAt(i)
    Object valObj = map.valueAt(i)
    ...
}

// HashMap
for(Iterator it = map.iterator(); it.hasNext();){
    Object obj = it.next();
    ...
}
```

前面演示了很多ArrayMap的优点，但并不是所有情况下都适合使用ArrayMap，我们应该在满足下面2个条件的时候才考虑使用ArrayMap：

- 对象个数的数量级最好是千以内
- 数据组织形式包含Map结构

我们需要学会在特定情形下选择相对更加高效的实现方式。

2) Beware Autoboxing

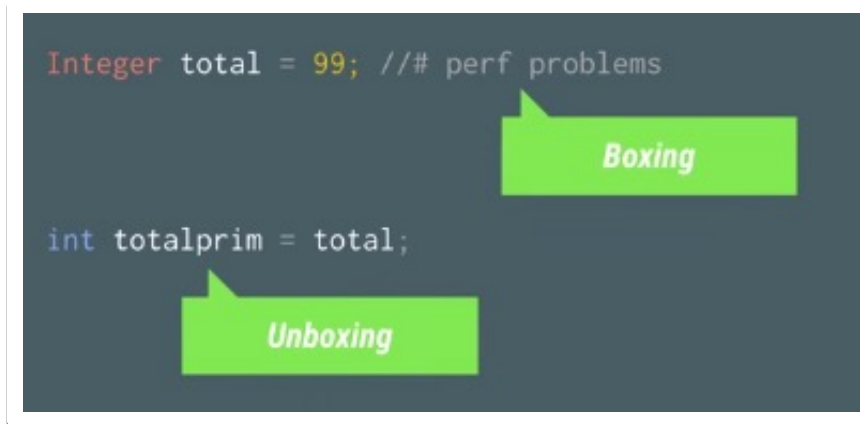
有时候性能问题也可能是因为那些不起眼的小细节引起的，例如在代码中不经意的“自动装箱”。我们知道基础数据类型的大小：boolean(8 bits), int(32 bits), float(32 bits), long(64 bits)，为了能够让这些基础数据类型在大多数Java容器中运作，会需要做一个autoboxing的操作，转换成Boolean，Integer，Float等对象，如下演示了循环操作的时候是否发生autoboxing行为的差异：

```
// Primitive version
int total = 0;
for (int i = 0; i < 100; i++)
    total += i;
```

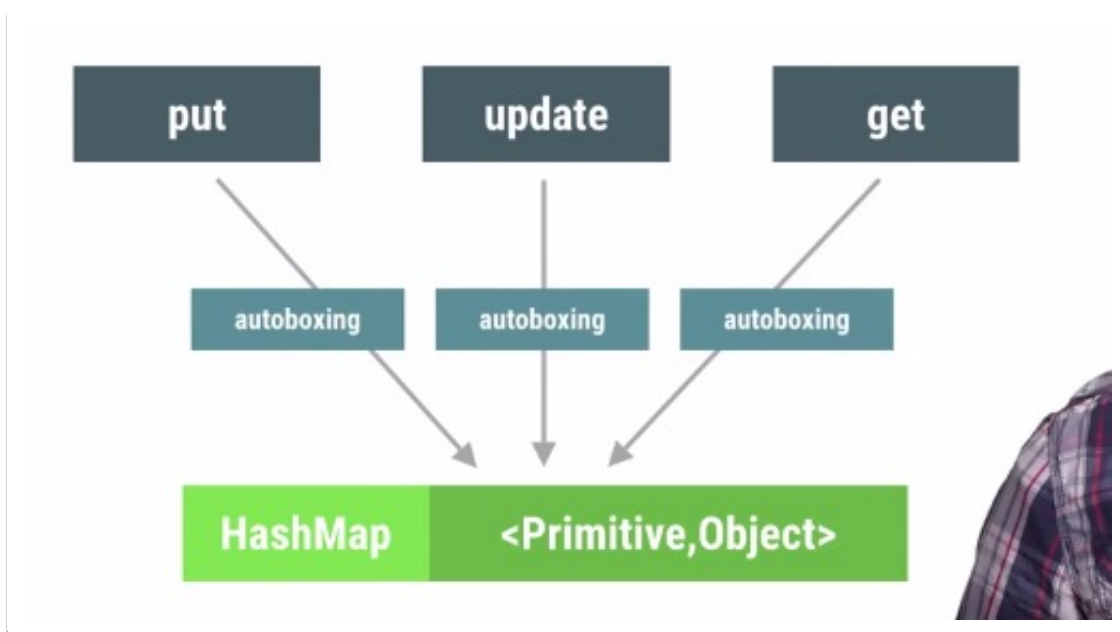
Allocates 0 new objects!

```
// Generic version
Integer total = 0;
for (int i = 0; i < 100; i++)
    //total += i;
    // create new Integer(),
    // push in new value
    // add to total
```

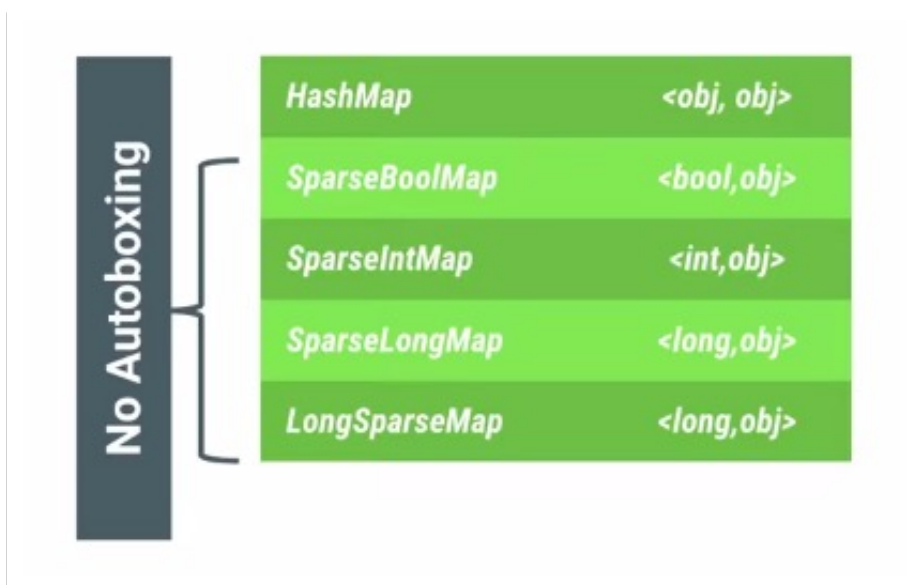
*Allocates 83 new objects!
(yea, I know it's confusing)*



Autoboxing的行为还经常发生在类似HashMap这样的容器里面，对HashMap的增删改查操作都会发生了大量的autoboxing的行为。



为了避免这些autoboxing带来的效率问题，Android特地提供了一些如下的Map容器用来替代HashMap，不仅避免了autoboxing，还减少了内存占用：



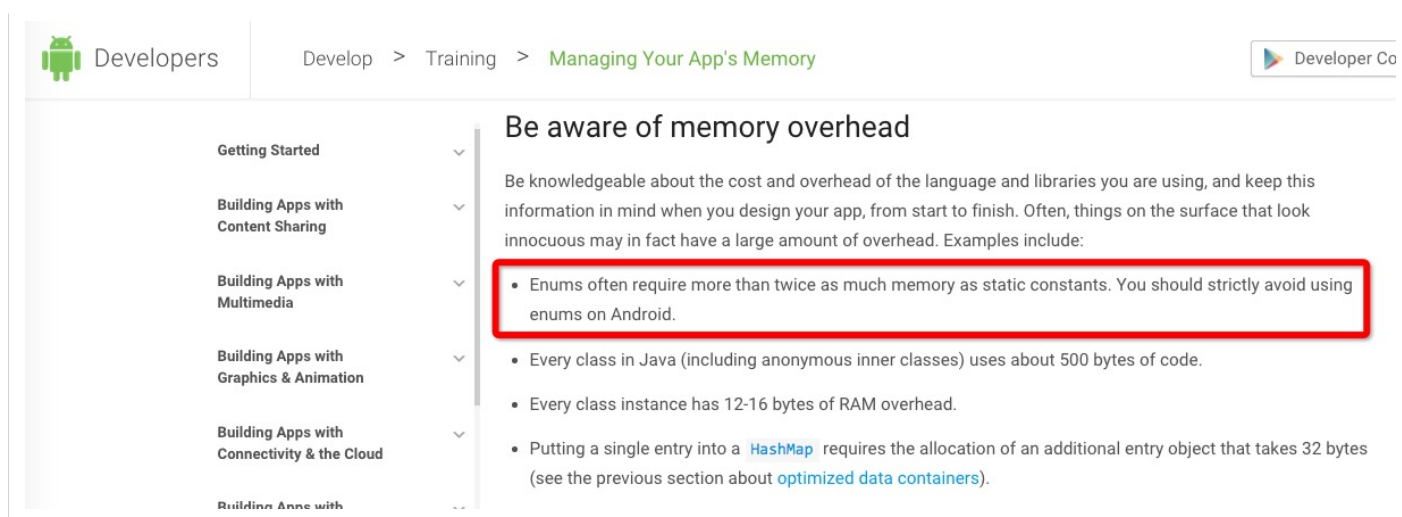
3)SparseArray Family Ties

为了避免HashMap的autoboxing行为，Android系统提供了SparseBoolMap，SparseIntMap，SparseLongMap，LongSparseMap等容器。关于这些容器的基本原理请参考前面的ArrayMap的介绍，另外这些容器的使用场景也和ArrayMap一致，需要满足数量级在千以内，数据组织形式需要包含Map结构。

4)The price of ENUMs

在StackOverflow等问答社区常常出现关于在Android系统里面使用枚举类型的性能讨论，关于这一点，Android官方的Training课程里面有下面这样一句话：

*Enums often require more than twice as much memory as static constants.
You should strictly avoid using enums on Android.*



Developers

Develop > Training > Managing Your App's Memory

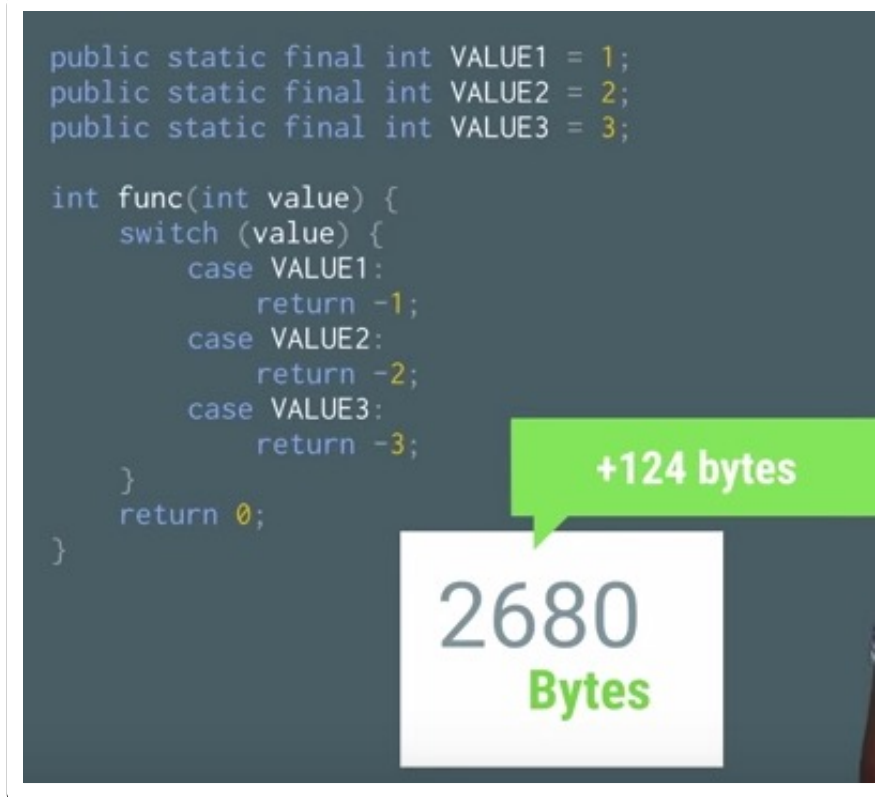
Developer Co

Be aware of memory overhead

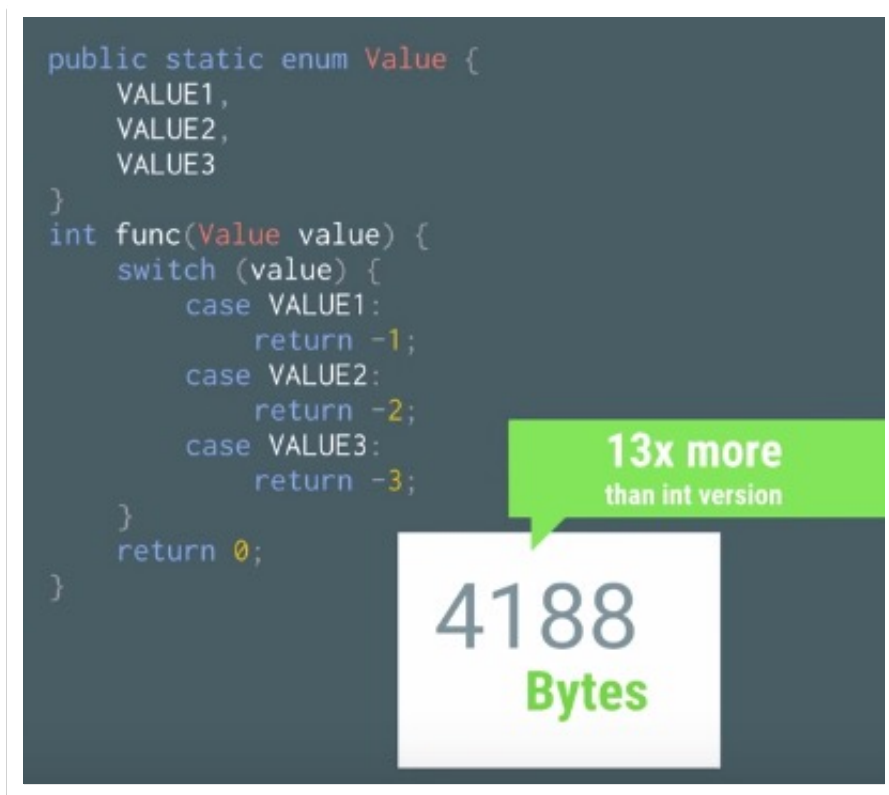
Be knowledgeable about the cost and overhead of the language and libraries you are using, and keep this information in mind when you design your app, from start to finish. Often, things on the surface that look innocuous may in fact have a large amount of overhead. Examples include:

- Enums often require more than twice as much memory as static constants. You should strictly avoid using enums on Android.
- Every class in Java (including anonymous inner classes) uses about 500 bytes of code.
- Every class instance has 12-16 bytes of RAM overhead.
- Putting a single entry into a [HashMap](#) requires the allocation of an additional entry object that takes 32 bytes (see the previous section about [optimized data containers](#)).

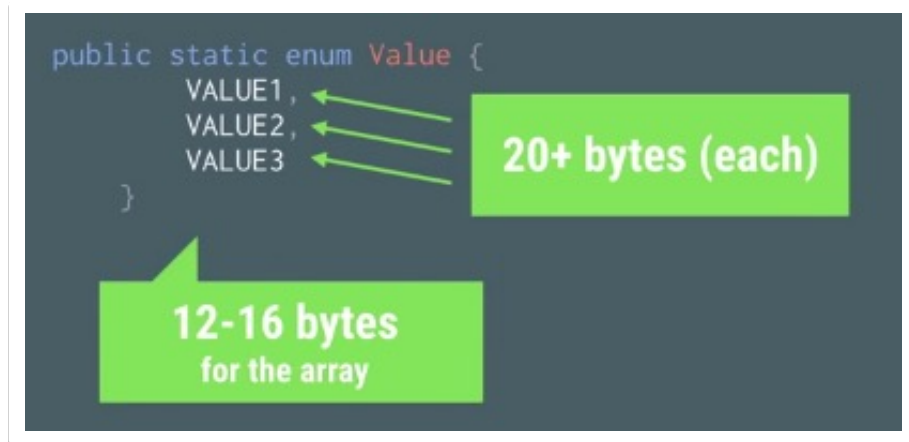
关于enum的效率，请看下面的讨论。假设我们有这样一份代码，编译之后的dex大小是2556 bytes，在此基础上，添加一些如下代码，这些代码使用普通static常量相关作为判断值：



增加上面那段代码之后，编译成dex的大小是2680 bytes，相比起之前的2556 bytes只增加124 bytes。假如换做使用enum，情况如下：



使用enum之后的dex大小是4188 bytes，相比起2556增加了1632 bytes，增长量是使用static int的13倍。不仅如此，使用enum，运行时还会产生额外的内存占用，如下图所示：

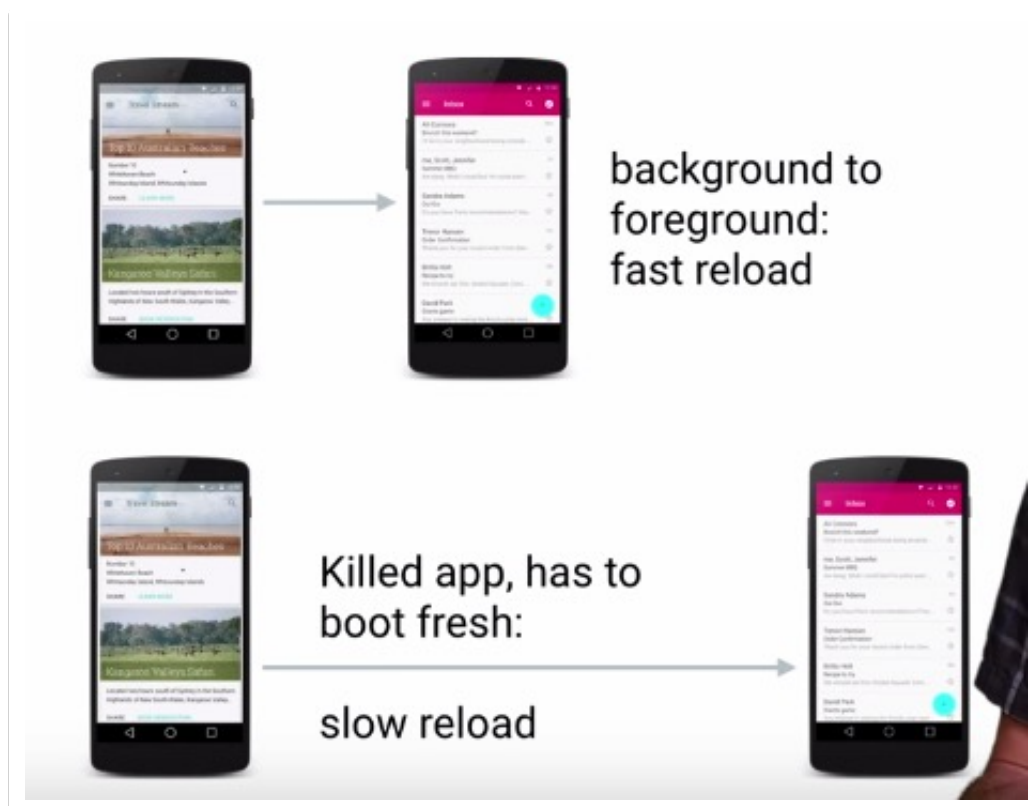


Android官方强烈建议不要在Android程序里面使用到enum。

5)Trimming and Sharing Memory

Android系统的一大特色是多任务，用户可以随意在不同的app之间进行快速切换。为了确保你的应用在这种复杂的多任务环境中正常运行，我们需要了解下面的知识。

为了让background的应用能够迅速的切换到foreground，每一个background的应用都会占用一定的内存。Android系统会根据当前的系统内存使用情况，决定回收部分background的应用内存。如果background的应用从暂停状态直接被恢复到foreground，能够获得较快的恢复体验，如果background应用是从Kill的状态进行恢复，就会显得稍微有点慢。



Android系统提供了一些回调来通知应用的内存使用情况，通常来说，当所有的background应

用都被kill掉的时候，foreground应用会收到**onLowMemory()**的回调。在这种情况下，需要尽快释放当前应用的非必须内存资源，从而确保系统能够稳定继续运行。Android系统还提供了**onTrimMemory()**的回调，当系统内存达到某些条件的时候，所有正在运行的应用都会收到这个回调，同时在这个回调里面会传递以下的参数，代表不同的内存使用情况，下图介绍了各种不同的回调参数：

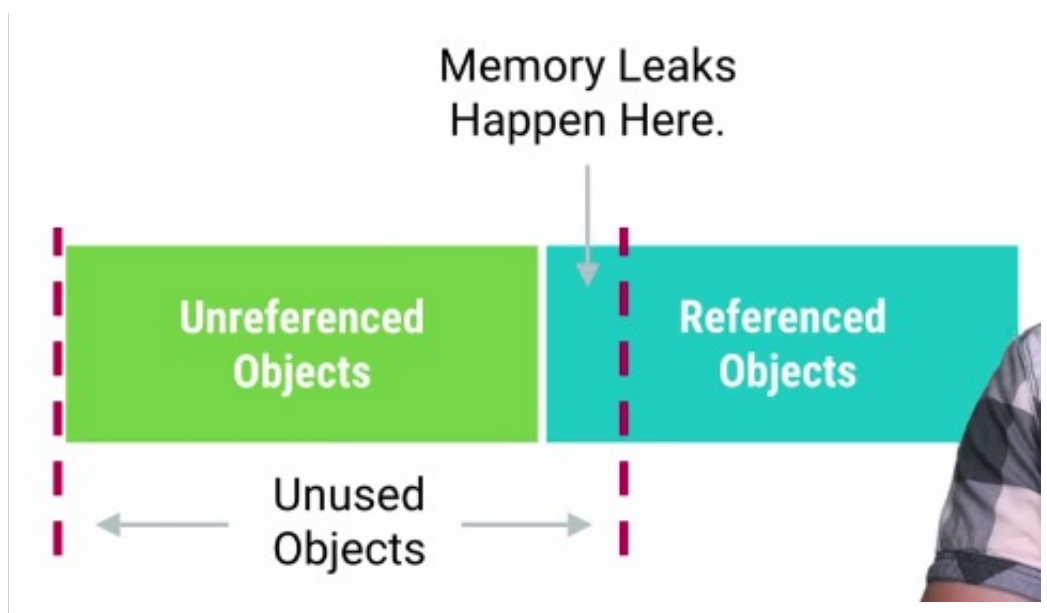
TRIM_MEMORY_RUNNING_MODERATE	<i>this is your first warning</i>
TRIM_MEMORY_RUNNING_MODERATE	<i>This is like the yellow light. It is your second warning to begin to trim resources to improve performance.</i>
TRIM_MEMORY_RUNNING_CRITICAL	<i>This is the red light. If you keep on executing without clearing up memory resource, the system is going to begin killing background processes to get more memory for you. Unfortunately that will lower the performance of your application.</i>
TRIM_MEMORY_UI_HIDDEN	<i>Your application was just moved off the screen, so this is a good time to release large UI resources. Now your application is on the list of cached applications. If there are memory problems, your process may be killed</i>
TRIM_MEMORY_BACKGROUND	<i>Being a background app - release as much as you can so that your app can resume faster than a pure restart</i>
TRIM_MEMORY_BACKGROUND	<i>You are a background app, but near the end of the list</i>
TRIM_MEMORY_MODERATE	<i>You are a background app, but in the middle</i>
TRIM_MEMORY_COMPLETE	<i>You are a background app, but about to be killed</i>

关于每个参数的更多介绍，请参考这里 http://hukai.me/android-training-managing_your_app_memory/，另外onTrimMemory()的回调可以发生在Application，Activity，Fragment，Service，Content Provider。

从Android 4.4开始，ActivityManager提供了**isLowRamDevice()**的API，通常指的是Heap Size低于512M或者屏幕大小<=800*480的设备。

6)DO NOT LEAK VIEWS

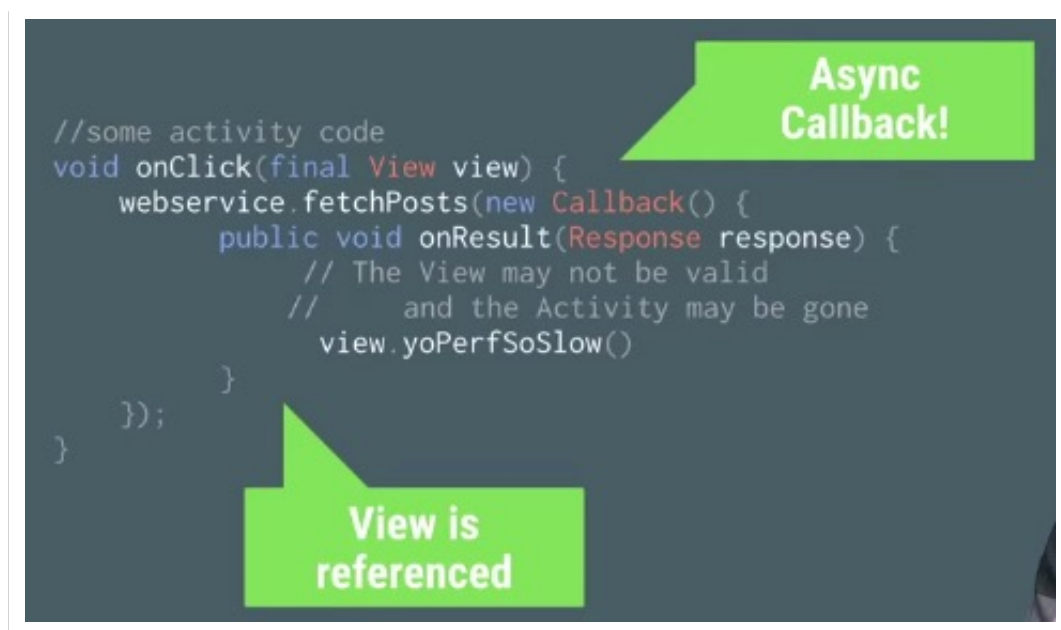
内存泄漏的概念，下面一张图演示下：



通常来说，View会保持Activity的引用，Activity同时还和其他内部对象也有可能保持引用关系。当屏幕发生旋转的时候，activity很容易发生泄漏，这样的话，里面的view也会发生泄漏。Activity以及view的泄漏是非常严重的，为了避免出现泄漏，请特别留意以下的规则：

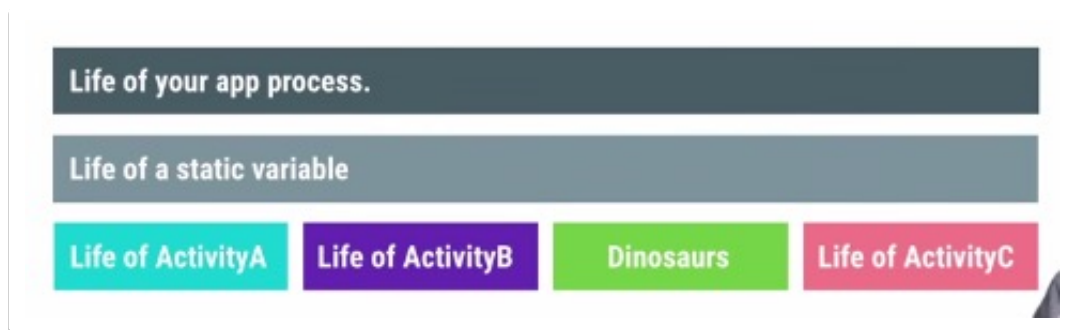
6.1)避免使用异步回调

异步回调被执行的时间不确定，很有可能发生在activity已经被销毁之后，这不仅仅很容易引起crash，还很容易发生内存泄露。



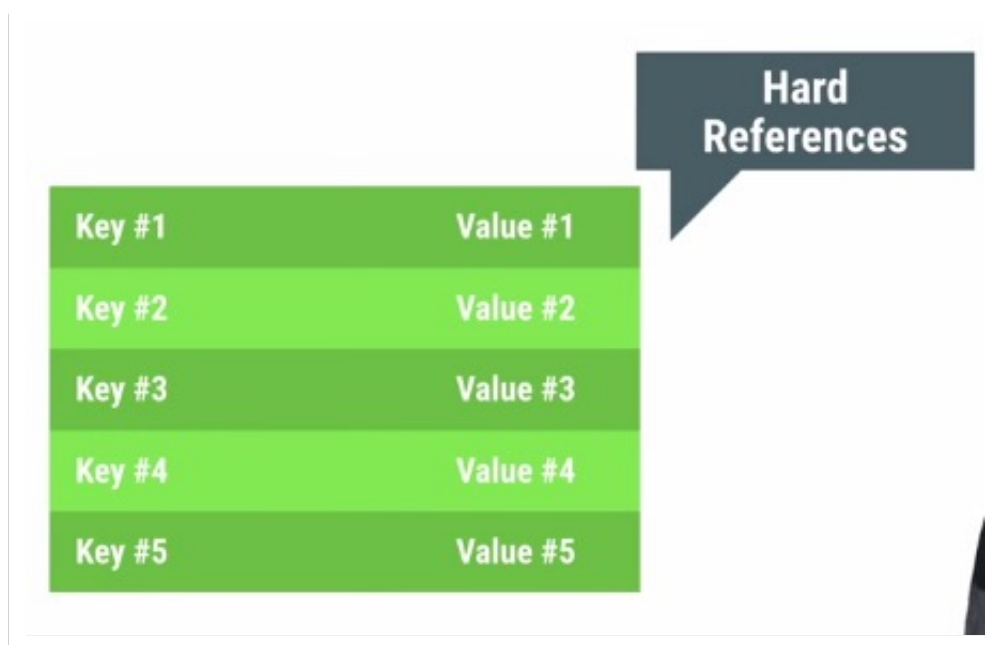
6.2)避免使用Static对象

因为static的生命周期过长，使用不当很可能导致leak，在Android中应该尽量避免使用static对象。



6.3)避免把View添加到没有清除机制的容器里面

假如把view添加到[WeakHashMap](#)，如果没有执行清除操作，很可能会导致泄漏。

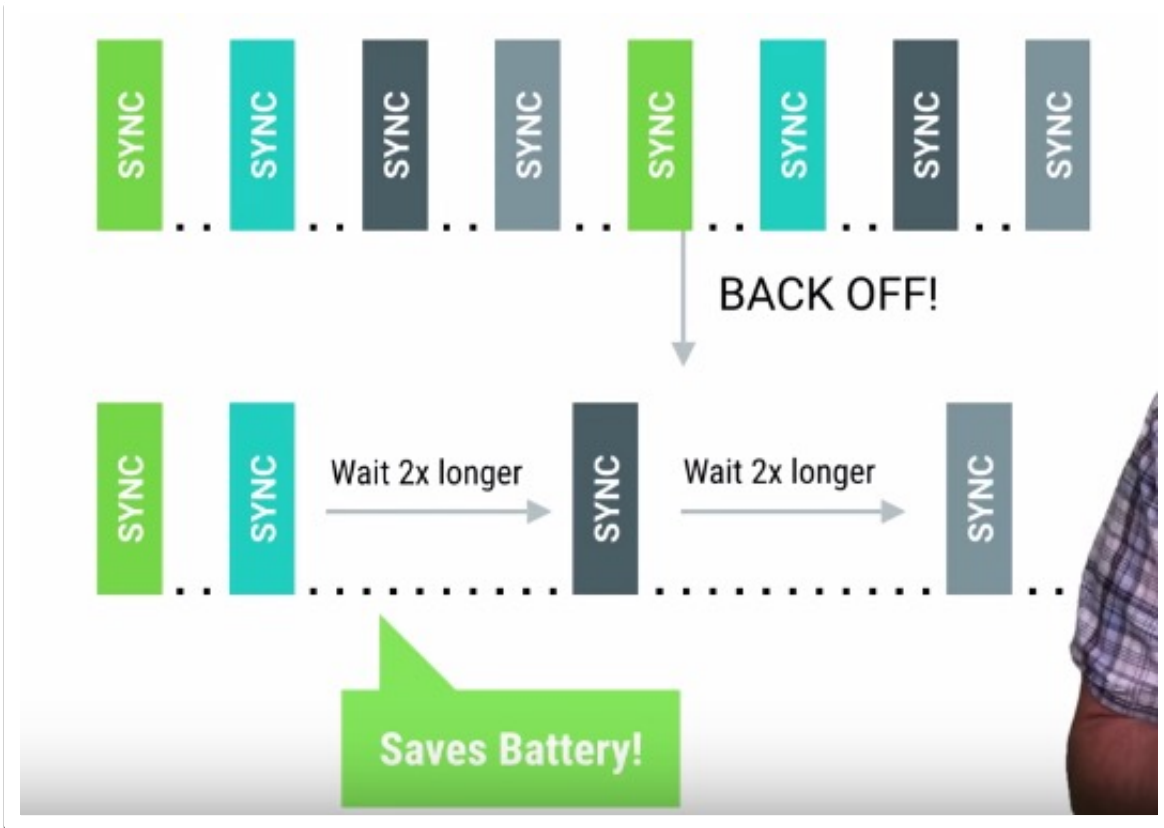


7)Location & Battery Drain

开启定位功能是一个相对来说比较耗电的操作，通常来说，我们会使用类似下面这样的代码来发出定位请求：

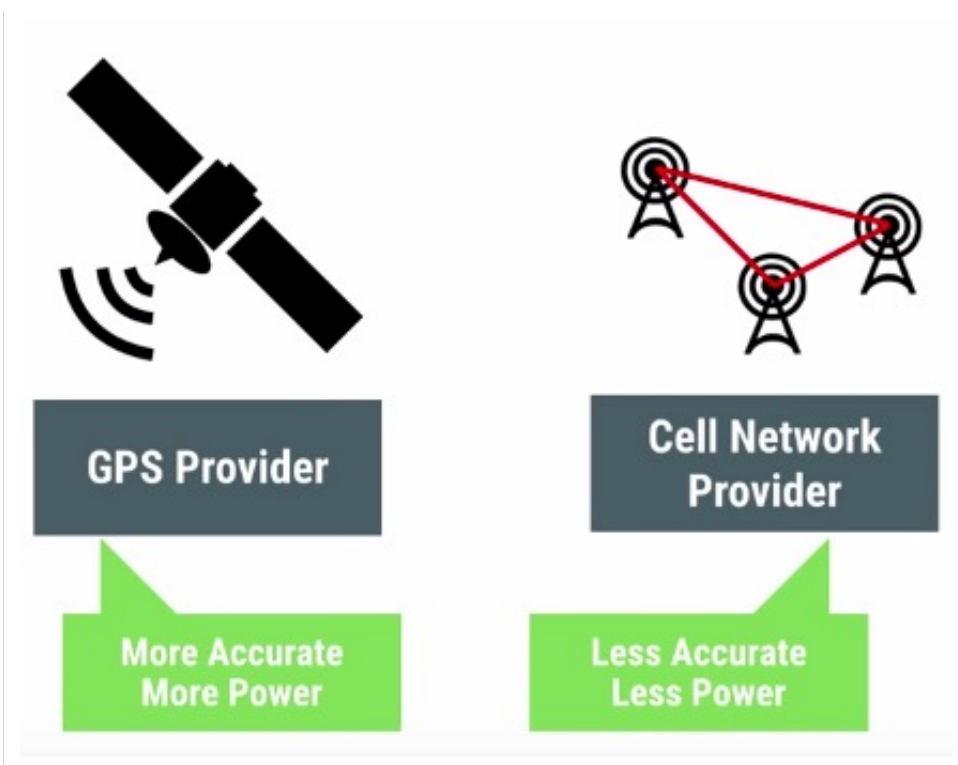
```
LocationRequest mLocationRequest = new LocationRequest();  
mLocationRequest.setInterval(10000);  
mLocationRequest.setFastestInterval(5000);  
mLocationRequest.setPriority(LocationRequest.PRIORITY_HIGH_ACCURACY);
```

上面演示中有一个方法是**setInterval()**指的意思是每隔多长的时间获取一次位置更新，时间相隔越短，自然花费的电量就越多，但是时间相隔太长，又无法及时获取到更新的位置信息。其中存在的一个优化点是，我们可以通过判断返回的位置信息是否相同，从而决定设置下次的更新间隔是否增加一倍，通过这种方式可以减少电量的消耗，如下图所示：

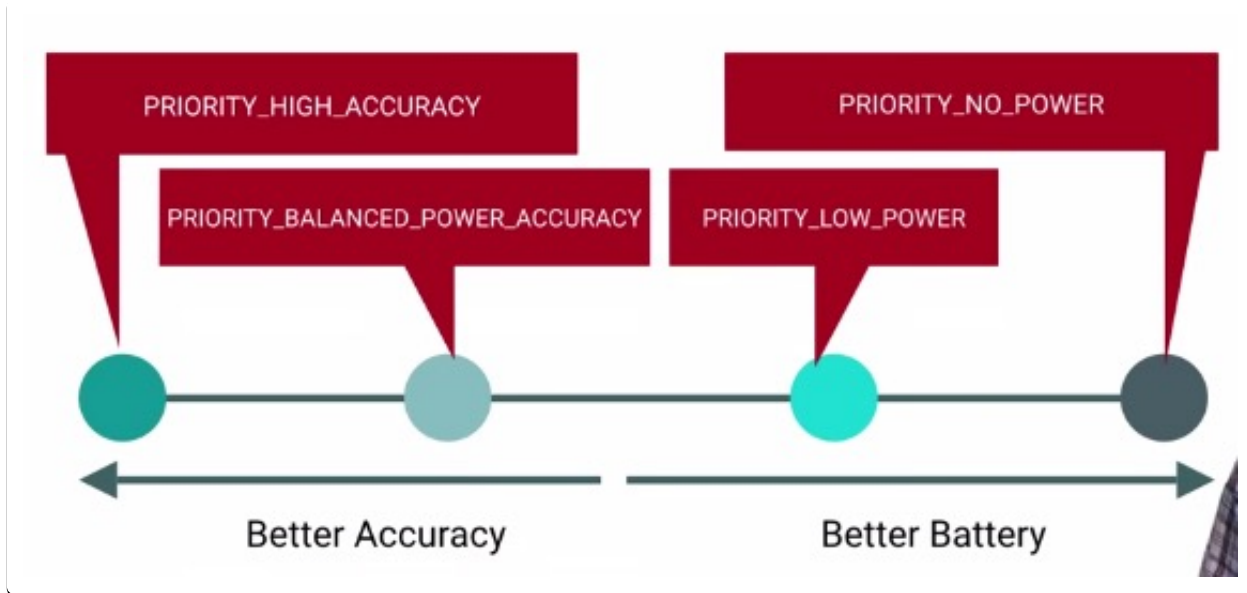


在位置请求的演示代码中还有一个方法是**`setFastestInterval()`**，因为整个系统中很可能存在其他的应用也在请求位置更新，那些应用很有可能设置的更新间隔时间很短，这种情况下，我们就可以通过**`setFastestInterval`**的方法来过滤那些过于频繁的更新。

通过**GPS**定位服务相比起使用网络进行定位更加的耗电，但是也相对更加精准一些，他们的图示关系如下：

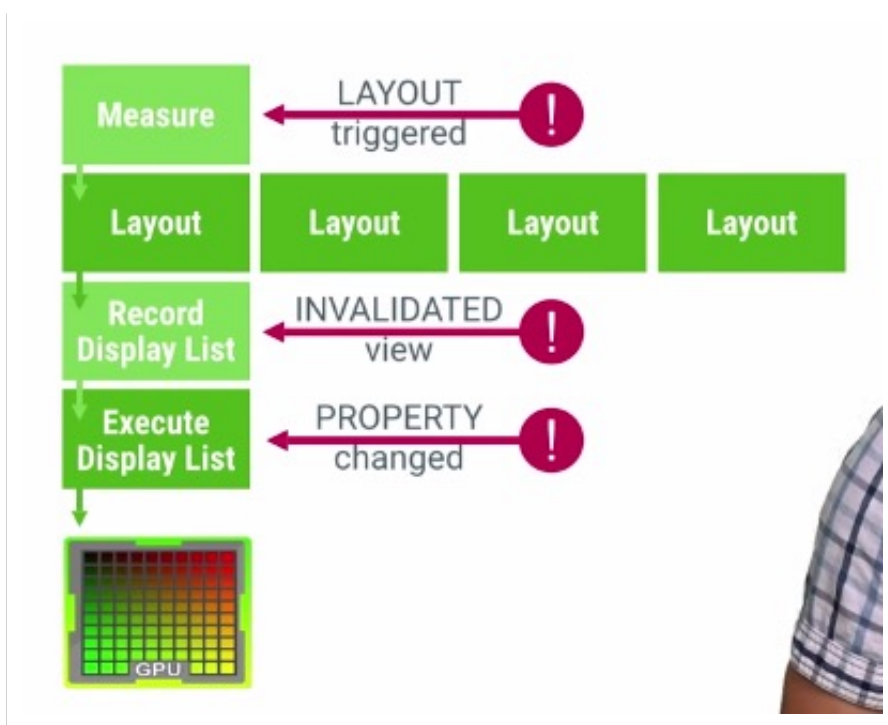


为了提供不同精度的定位需求，同时屏蔽实现位置请求的细节，Android提供了下面4种不同精度与耗电量的参数给应用进行设置调用，应用只需要决定在适当的场景下使用对应的参数就好了，通过`LocationRequest.setPriority()`方法传递下面的参数就好了。

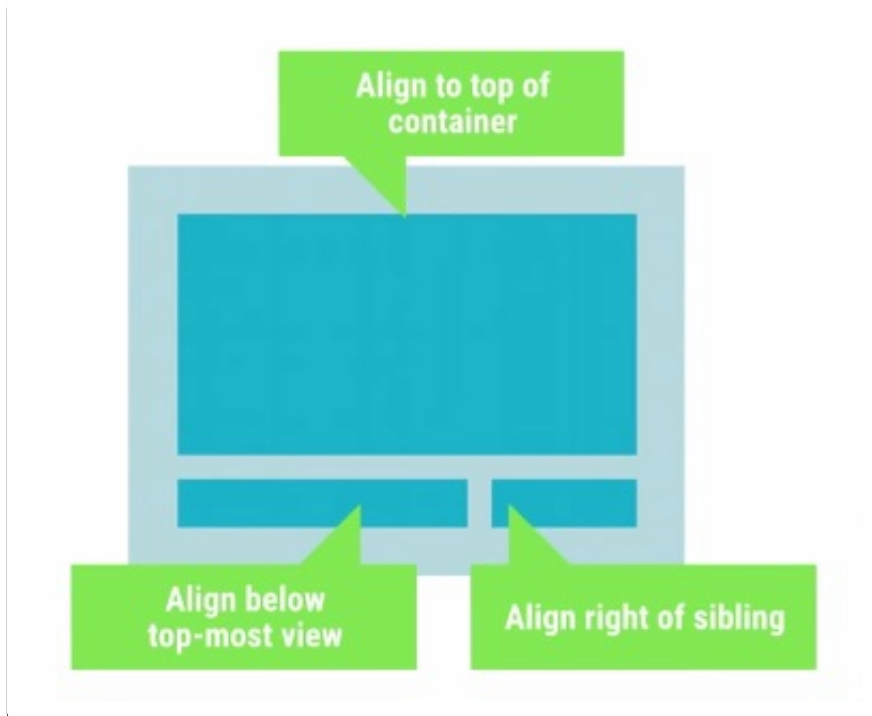


8)Double Layout Taxation

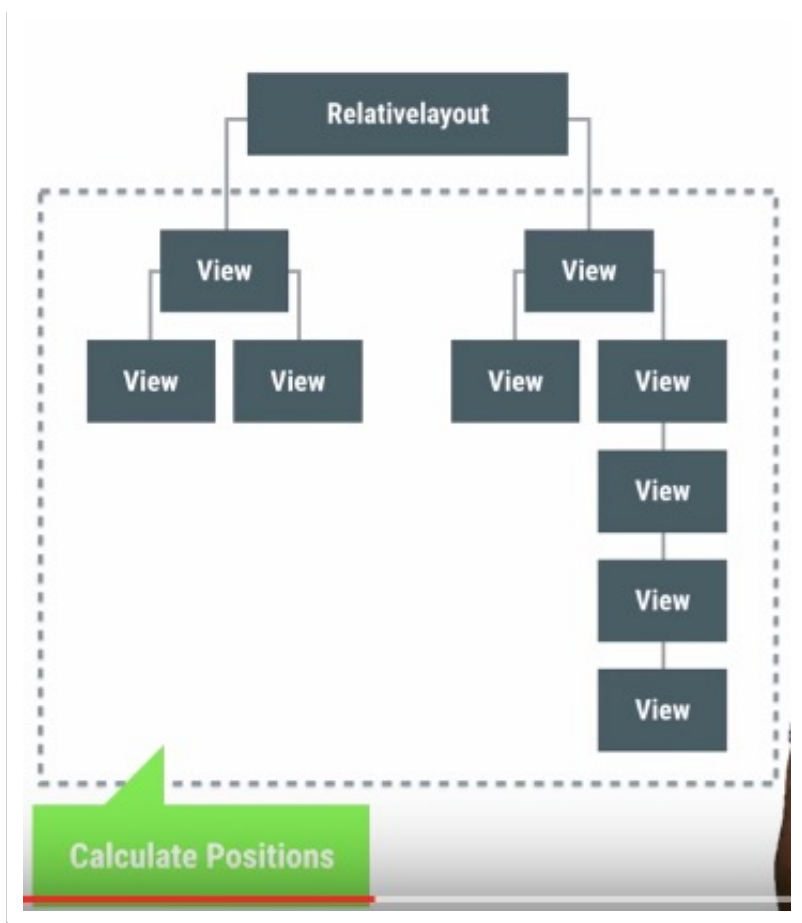
布局中的任何一个View一旦发生一些属性变化，都可能引起很大的连锁反应。例如某个button的大小突然增加一倍，有可能会引起兄弟视图的位置变化，也有可能引起父视图的大小发生改变。当大量的`layout()`操作被频繁调用执行的时候，就很可能引起丢帧的现象。

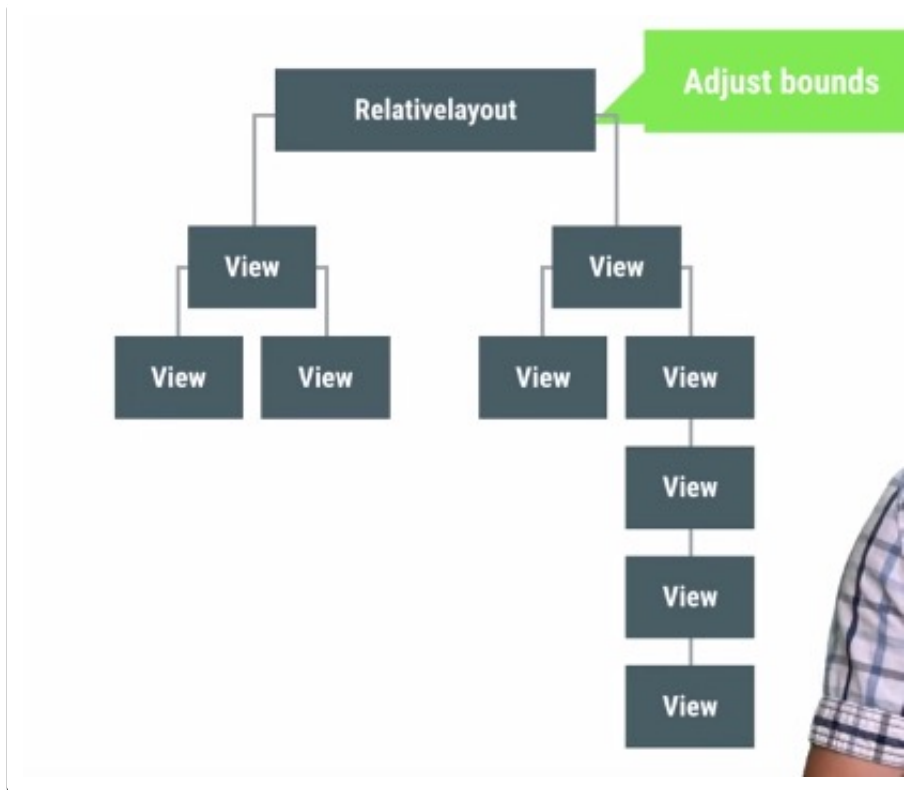


例如，在`RelativeLayout`中，我们通常会定义一些类似`alignTop`，`alignBelow`等等属性，如图所示：



为了获得视图的准确位置，需要经过下面几个阶段。首先子视图会触发计算自身位置的操作，然后`RelativeLayout`使用前面计算出来的位置信息做边界的调整的操作，如下面两张图所示：

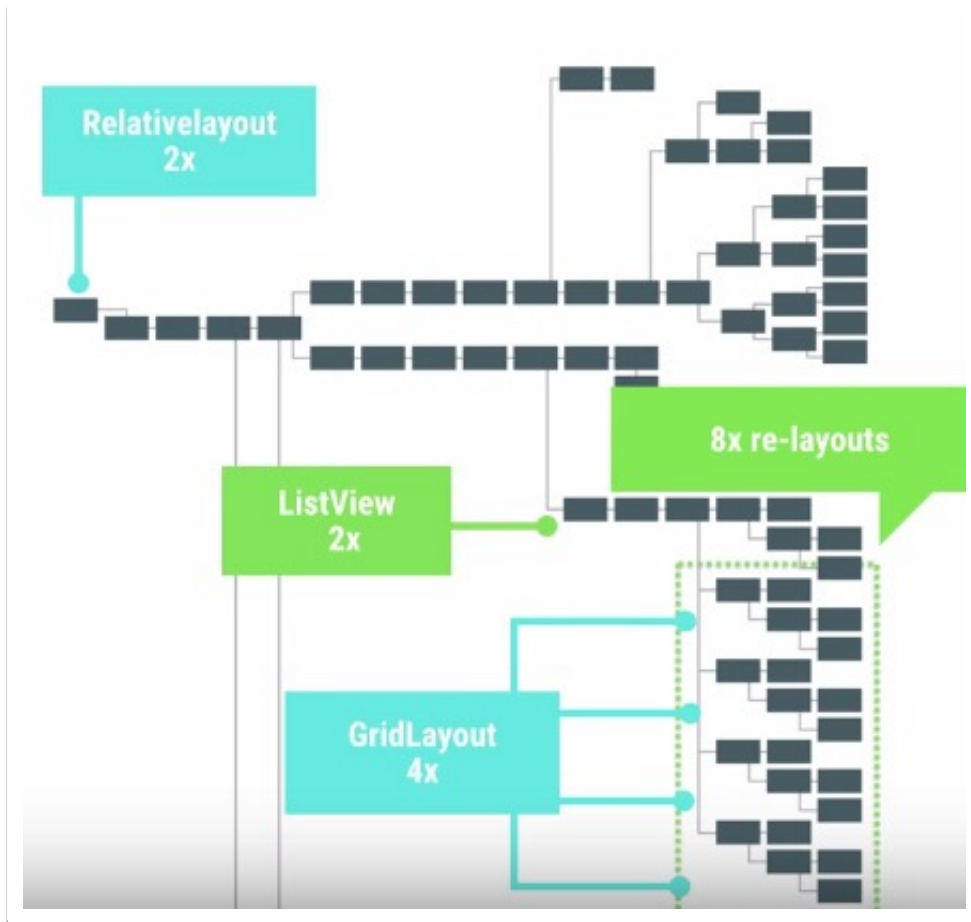




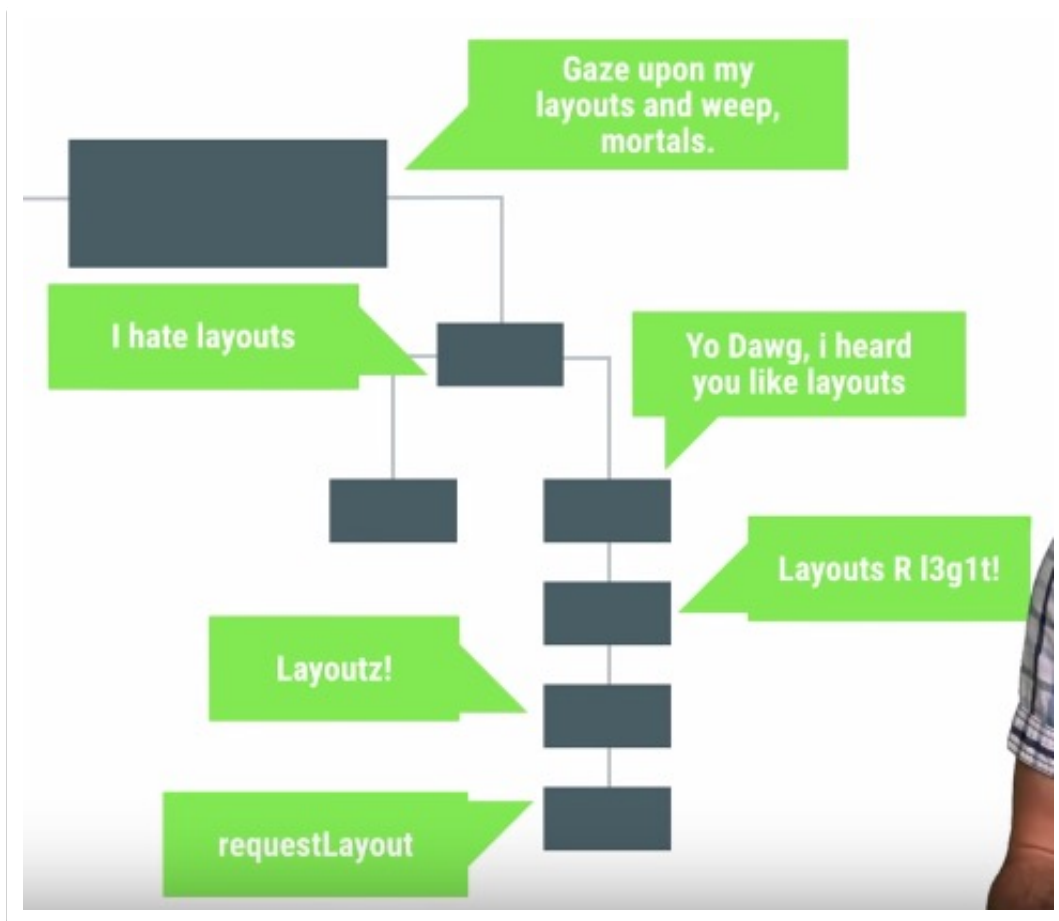
经历过上面2个步骤，**RelativeLayout**会立即触发第二次**layout()**的操作来确定所有子视图的最终位置与大小信息。

除了**RelativeLayout**会发生两次**layout**操作之外，**LinearLayout**也有可能触发两次**layout**操作，通常情况下**LinearLayout**只会发生一次**layout**操作，可是一旦调用了**measureWithTargetChild()**方法就会导致触发两次**layout**的操作。另外，通常来说，**GridLayout**会自动预处理子视图的关系来避免两次**layout**，可是如果**GridLayout**里面的某些子视图使用了**weight**等复杂的属性，还是会导致重复的**layout**操作。

如果只是少量的重复**layout**本身并不会引起严重的性能问题，但是如果它们发生在布局的根节点，或者是**ListView**里面的某个**Listitem**，这样就会引起比较严重的性能问题。如下图所示：



我们可以使用**Systrace**来跟踪特定的某段操作，如果发现了疑似丢帧的现象，可能就是因为重复**layout**引起的。通常我们无法避免重复**layout**，在这种情况下，我们应该尽量保持**View Hierarchy**的层级比较浅，这样即使发生重复**layout**，也不会因为布局的层级比较深而增大了重复**layout**的倍数。另外还有一点需要特别注意，在任何时候都请避免调用**requestLayout()**的方法，因为一旦调用了**requestLayout**，会导致该**layout**的所有父节点都发生重新**layout**的操作。

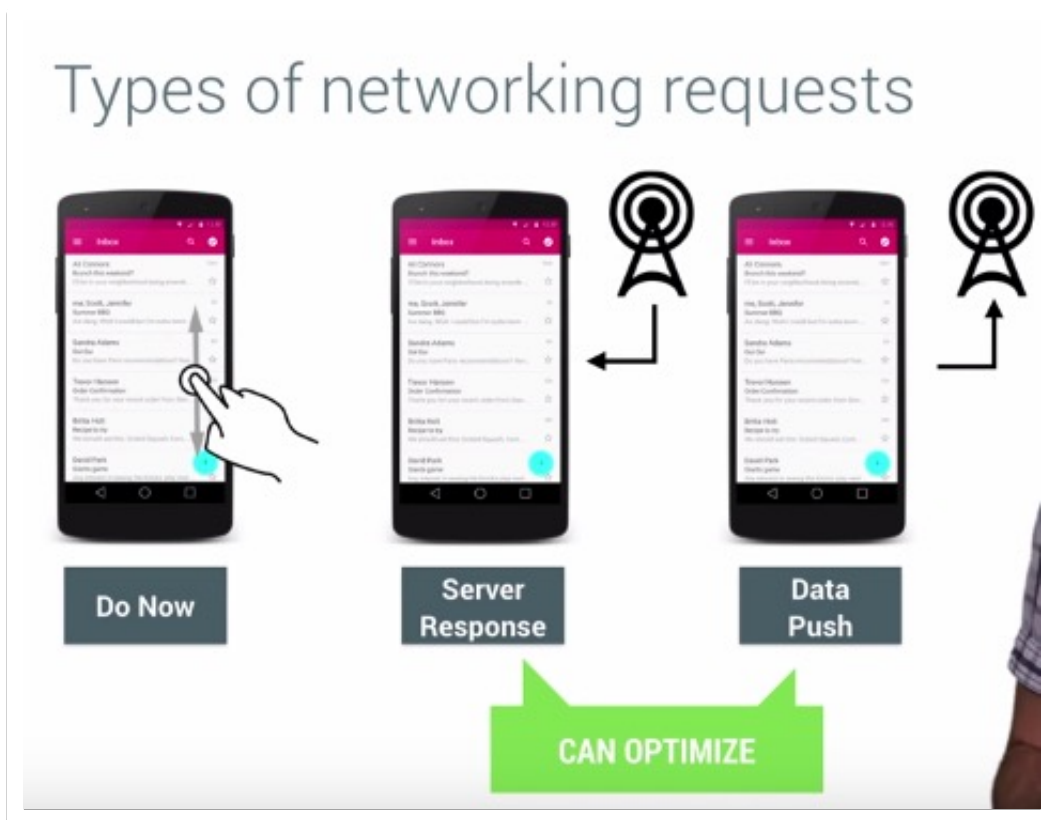


9)Network Performance 101

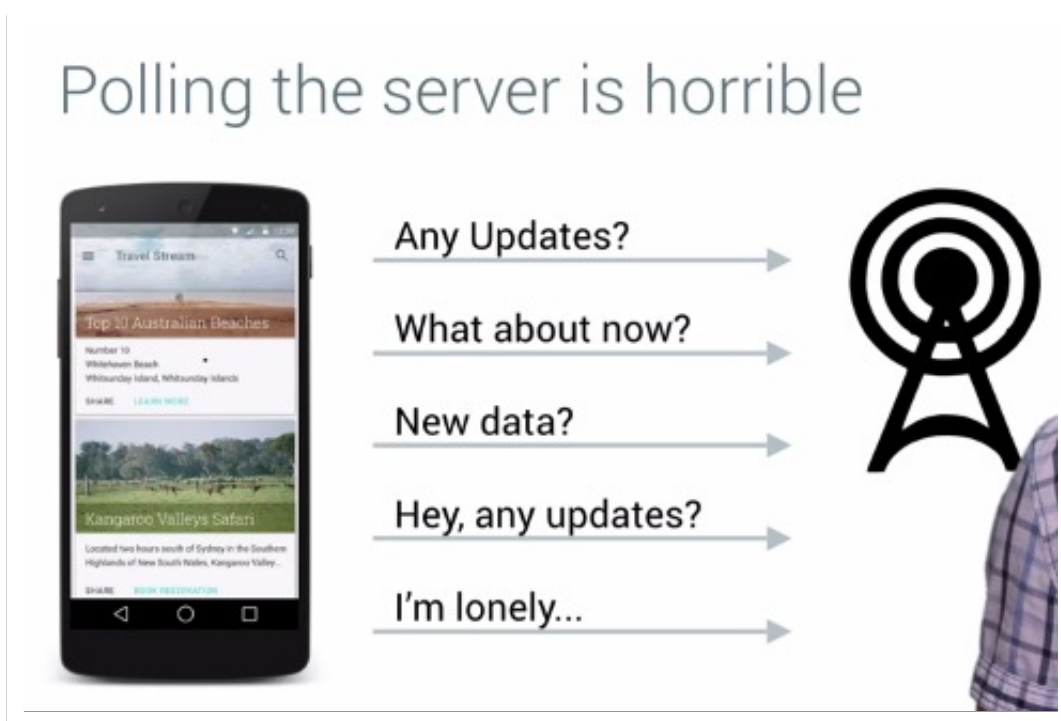
在性能优化第一季与第二季的课程里面都介绍过，网络请求的操作是非常耗电的，其中在移动蜂窝网络情况下执行网络数据的请求则尤其比较耗电。关于如何减少移动网络下的网络请求的耗电量，有两个重要的原则需要遵守：第一个是减少移动网络被激活的时间与次数，第二个是压缩传输数据。

9.1)减少移动网络被激活的时间与次数

通常来说，发生网络行为可以划分为如下图所示的三种类型，一个是用户主动触发的请求，另外被动接收服务器的返回数据，最后一个数据上报，行为上报，位置更新等等自定义的后台操作。



我们绝对坚决肯定不应该使用**Polling**(轮询)的方式去执行网络请求，这样不仅仅会造成严重的电量消耗，还会浪费许多网络流量，例如：

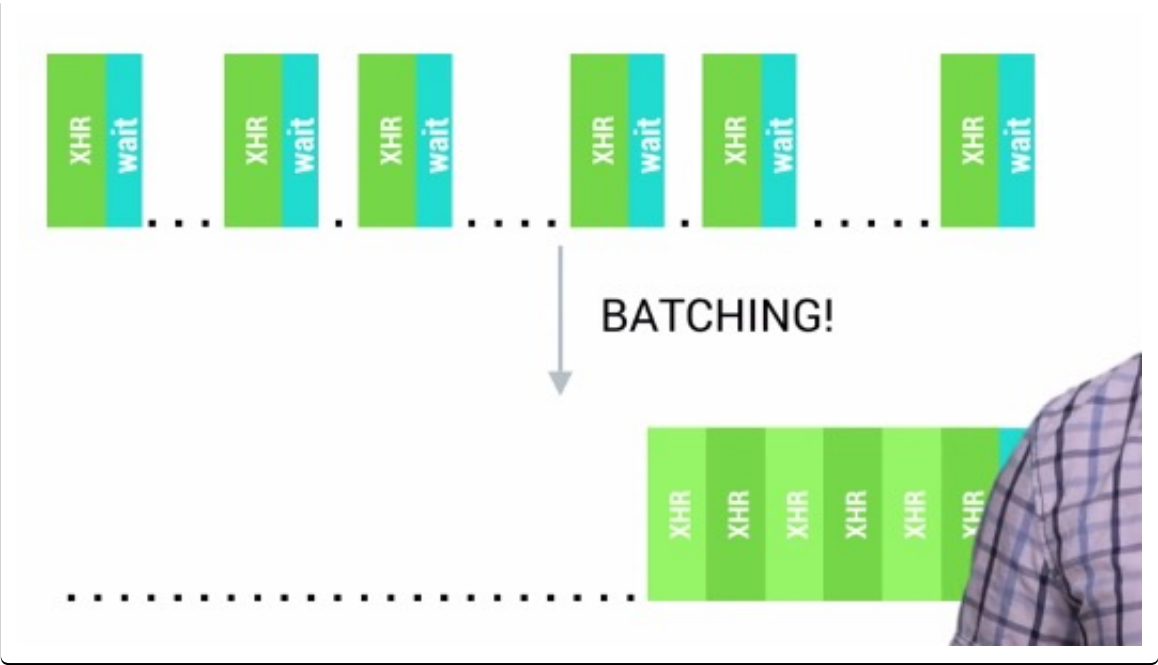


Android官方推荐使用[Google Cloud Messaging](https://cloud.google.com/messaging/)(在大陆，然并卵)，这个框架会帮助把更新的数据推送给手机客户端，效率极高！我们应该遵循下面的规则来处理数据同步的问题：

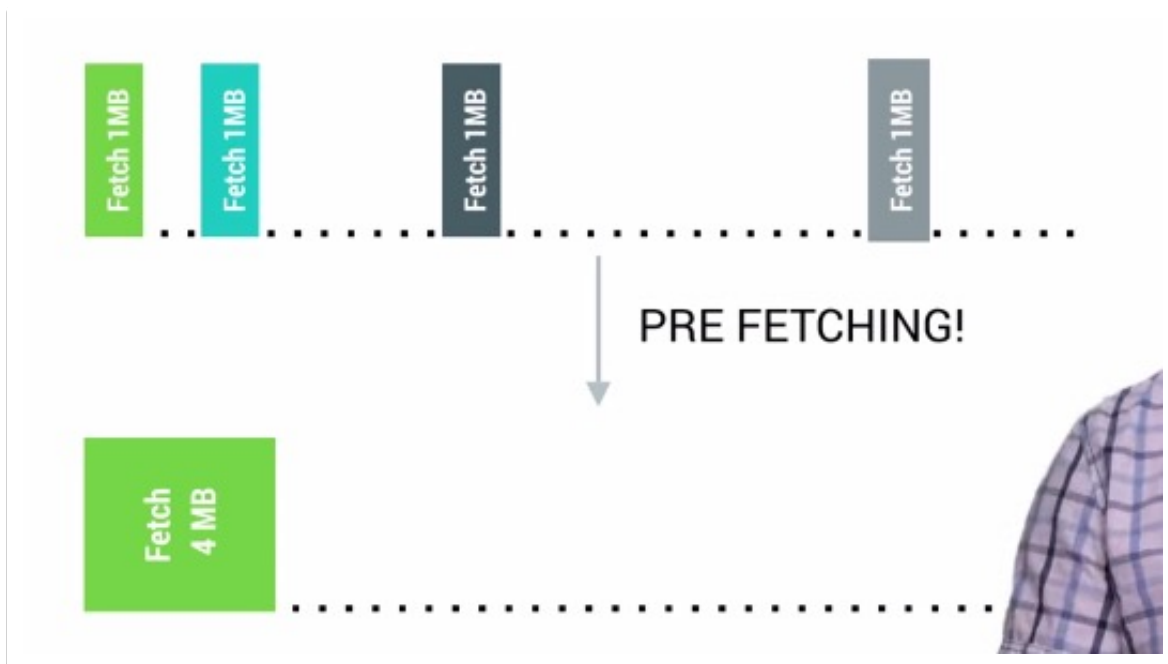
首先，我们应该使用回退机制来避免固定频繁的同步请求，例如，在发现返回数据相同的情况下，推迟下次的请求时间，如下图所示：



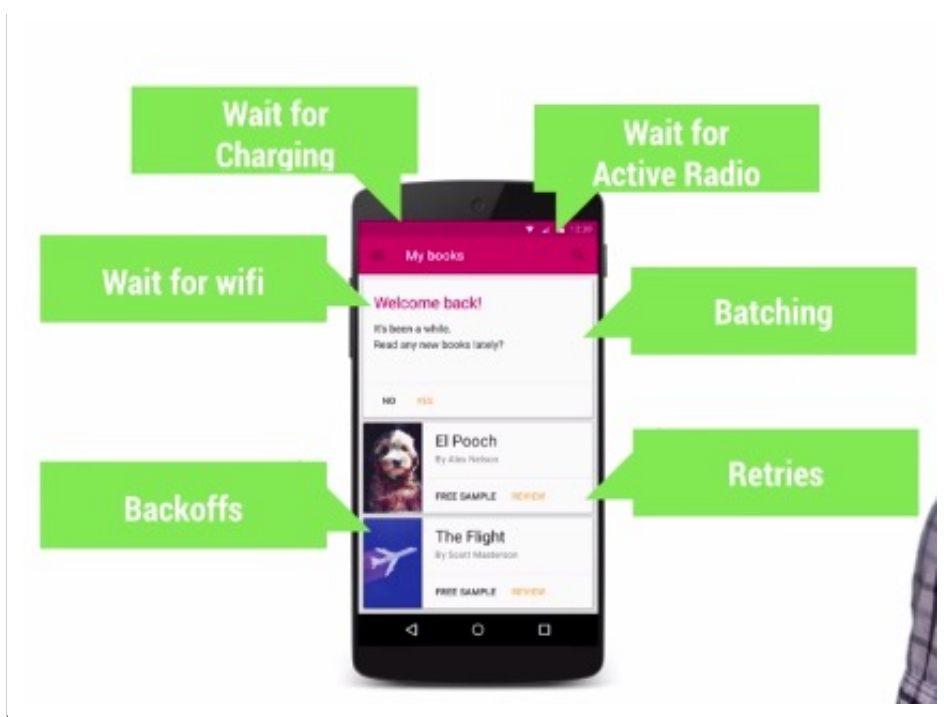
其次，我们还可以使用**Batching**(批处理)的方式来集中发出请求，避免频繁的间隔请求，如下图所示：



最后，我们还可以使用**Prefetching**(预取)的技术提前把一些数据拿到，避免后面频繁再次发起网络请求，如下图所示：



Google Play Service中提供了一个叫做[GCMNetworkManager](#)的类来帮助我们实现上面的那些功能，我们只需要调用对应的API，设置一些简单的参数，其余的工作就都交给Google来帮我们实现了。



9.2)压缩传输数据

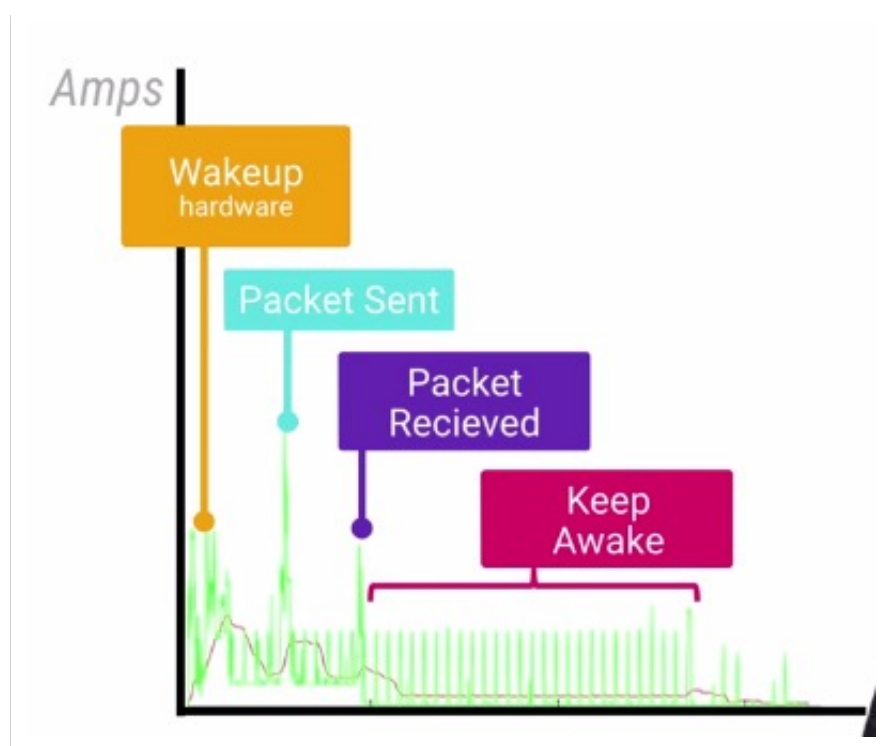
关于压缩传输数据，我们可以学习以下的一些课程(真的够喝好几壶了):

- [CompressorHead](#): 这系列的课程会介绍压缩的基本概念以及一些常见的压缩算法知识。
- [Image Compression](#): 介绍关于图片的压缩知识。
- [Texture Wranglin](#): 介绍了游戏开发相关的知识。

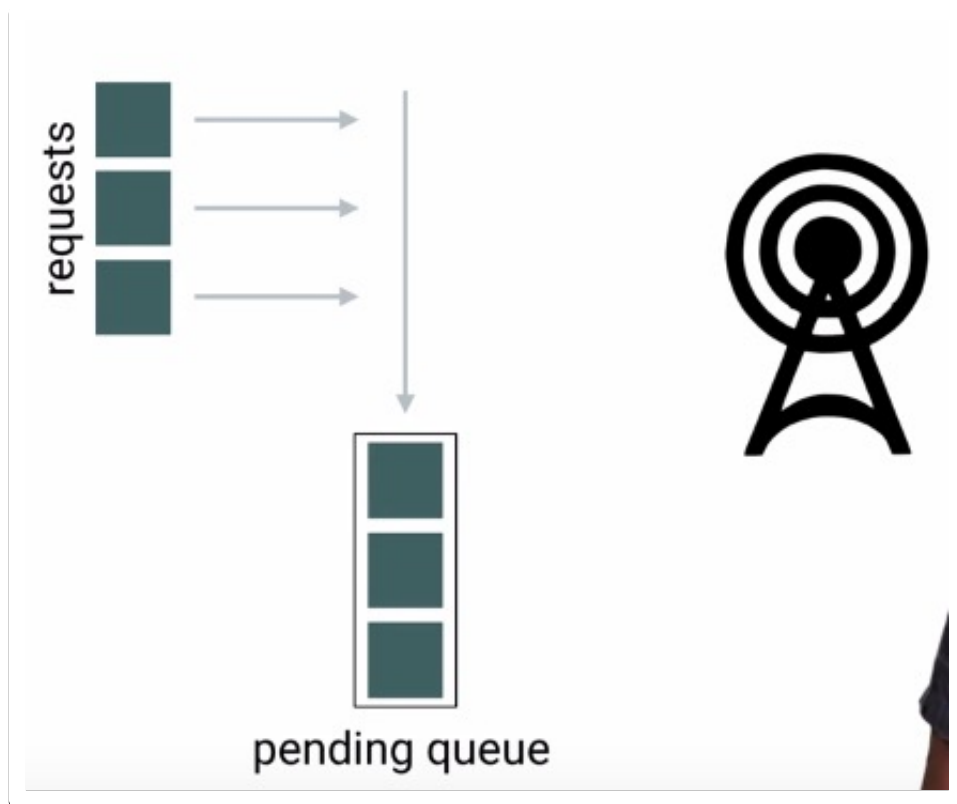
- [Grabby](#) : 介绍了游戏开发相关的知识。
- [Gzip is not enough](#)
- [Text Compression](#)
- [FlatBuffers](#)

10)Effective Network Batching

在性能优化课程的第一季与第二季里面，我们都有提到过下面这样一个网络请求与电量消耗的示意图：



发起网络请求与接收返回数据都是比较耗电的，在网络硬件模块被激活之后，会继续保持几十秒的电量消耗，直到没有新的网络操作行为之后，才会进入休眠状态。前面一个段落介绍了使用Batching的技术来捆绑网络请求，从而达到减少网络请求的频率。那么如何实现Batching技术呢？通常来说，我们可以会把那些发出的网络请求，先暂存到一个PendingQueue里面，等到条件合适的时候再触发Queue里面的网络请求。



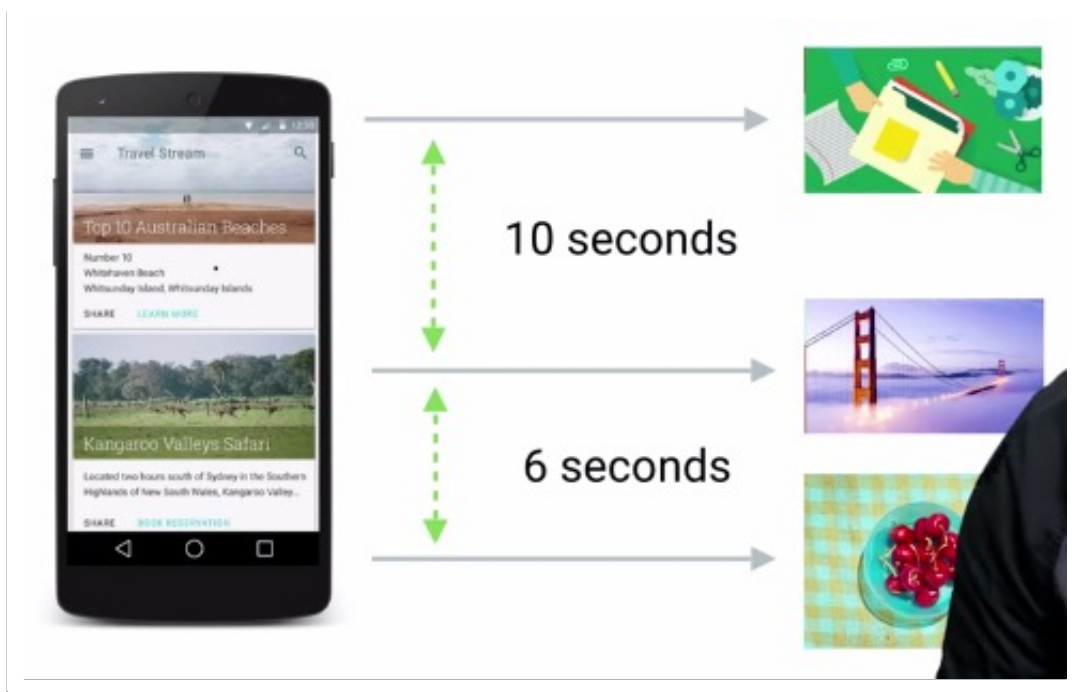
可是什么时候才算是条件合适了呢？最简单粗暴的，例如我们可以在Queue大小到10的时候触发任务，也可以是当手机开始充电，或者是手机连接到WiFi等情况下才触发队列中的任务。手动编写代码去实现这些功能会比较复杂繁琐，Google为了解决这个问题，为我们提供了GCMNetworkManager来帮助实现那些功能，仅仅只需要调用API，设置触发条件，然后就OK了。

11)Optimizing Network Request Frequencies

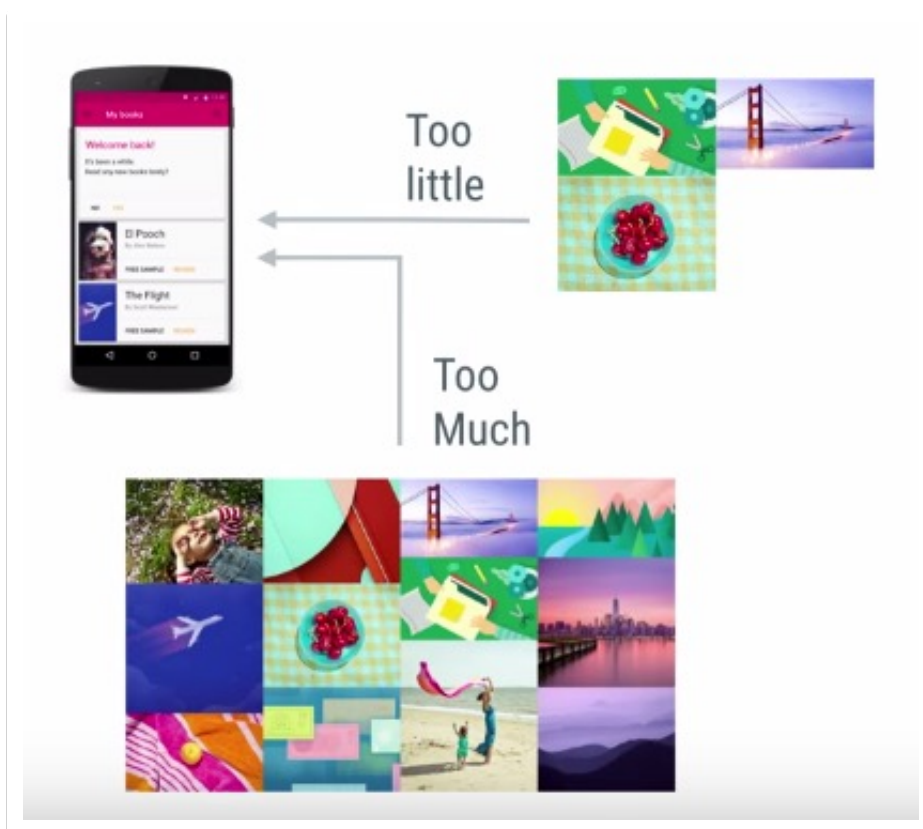
前面的段落已经提到了应该减少网络请求的频率，这是为了减少电量的消耗。我们可以使用Batching，Prefetching的技术来避免频繁的网络请求。Google提供了GCMNetworkManager来帮助开发者实现那些功能，通过提供的API，我们可以选择在接入WiFi，开始充电，等待移动网络被激活等条件下再次激活网络请求。

12)Effective Prefetching

假设我们有这样的一个场景，最开始网络请求了一张图片，隔了10秒需要请求另外一张图片，再隔6秒会请求第三张图片，如下图所示：



类似上面的情况会频繁触发网络请求，但是如果我们能够预先请求后续可能会使用到网络资源，避免频繁的触发网络请求，这样就能够显著的减少电量的消耗。可是预先获取多少数据量是很值得考量的，因为如果预取数据量偏少，就起不到减少频繁请求的作用，可是如果预取数据过多，就会造成资源的浪费。



我们可以参考在WiFi，4G，3G等不同的网络下设计不同大小的预取数据量，也可以是按照图片数量或者操作时间来作为阈值。这需要我们根据特定的场景，不同的网络情况设计合适的方案。

首发于CSDN: [Android性能优化典范 \(三\)](#)



知识共享许可协议：本站作品由[HuKai](#)创作，采用[知识共享 署名-非商业性使用-相同方式共享 4.0 国际 许可协议](#)进行许可。

如果你觉得这篇文章对你有帮助，请点击下面的分享链接，你还可以选择扫描二维码进行打



赏，承诺所有打赏都会用于公益!

Posted by HuKai Aug 11th, 2015 [Android](#), [Android:Performance](#)