

DEC 31ST, 2015 | [COMMENTS](#)

# Android性能优化典范 - 第4季



[Android性能优化典范](#)第4季的课程学习笔记终于在2015年的最后一天完成了，文章共17个段落，包含的内容大致有：优化网络请求的行为，优化安装包的资源文件，优化数据传输的效率，性能优化的几大基础原理等等。因为学习认知水平有限，肯定存在不少理解偏差甚至错误的地方，请多多交流指正！

## 1)Cachematters for networking

想要使得Android系统上的网络访问操作更加的高效就必须做好网络数据的缓存。这是提高网络访问性能最基础的步骤之一。从手机的缓存中直接读取数据肯定比从网络上获取数据要更加的便捷高效，特别是对于那些会被频繁访问到的数据，需要把这些数据缓存到设备上，以便更加快速的进行访问。

Android系统上关于网络请求的Http Response Cache是默认关闭的，这样会导致每次即使请求的数据内容是一样的也会需要重复被调用执行，效率低下。我们可以通过下面的代码示例开启[HttpResponseBodyCache](#)。

```
protected void onCreate(Bundle savedInstanceState) {  
    ...  
  
    try {  
        File httpCacheDir = new File(context.getCacheDir(), "http");  
        long httpCacheSize = 10 * 1024 * 1024; // 10 MiB  
        HttpResponseCache.install(httpCacheDir, httpCacheSize);  
    } catch (IOException e) {  
        Log.i(TAG, "HTTP response cache installation failed:" + e);  
    }  
}  
  
protected void onStop() {  
    ...  
  
    HttpResponseCache cache = HttpResponseCache.getInstalled();  
    if (cache != null) {  
        cache.flush();  
    }  
}
```

开启Http Response Cache之后，Http操作相关的返回数据就会缓存到文件系统上，不仅仅是主程序自己编写的网络请求相关的数据会被缓存，另外引入的library库中的网络相关的请求数据也会被缓存到这个Cache中。

网络请求的场景有可以是普通的http请求，也可以打开某个URL去获取数据，如下图所示：

## Using URL

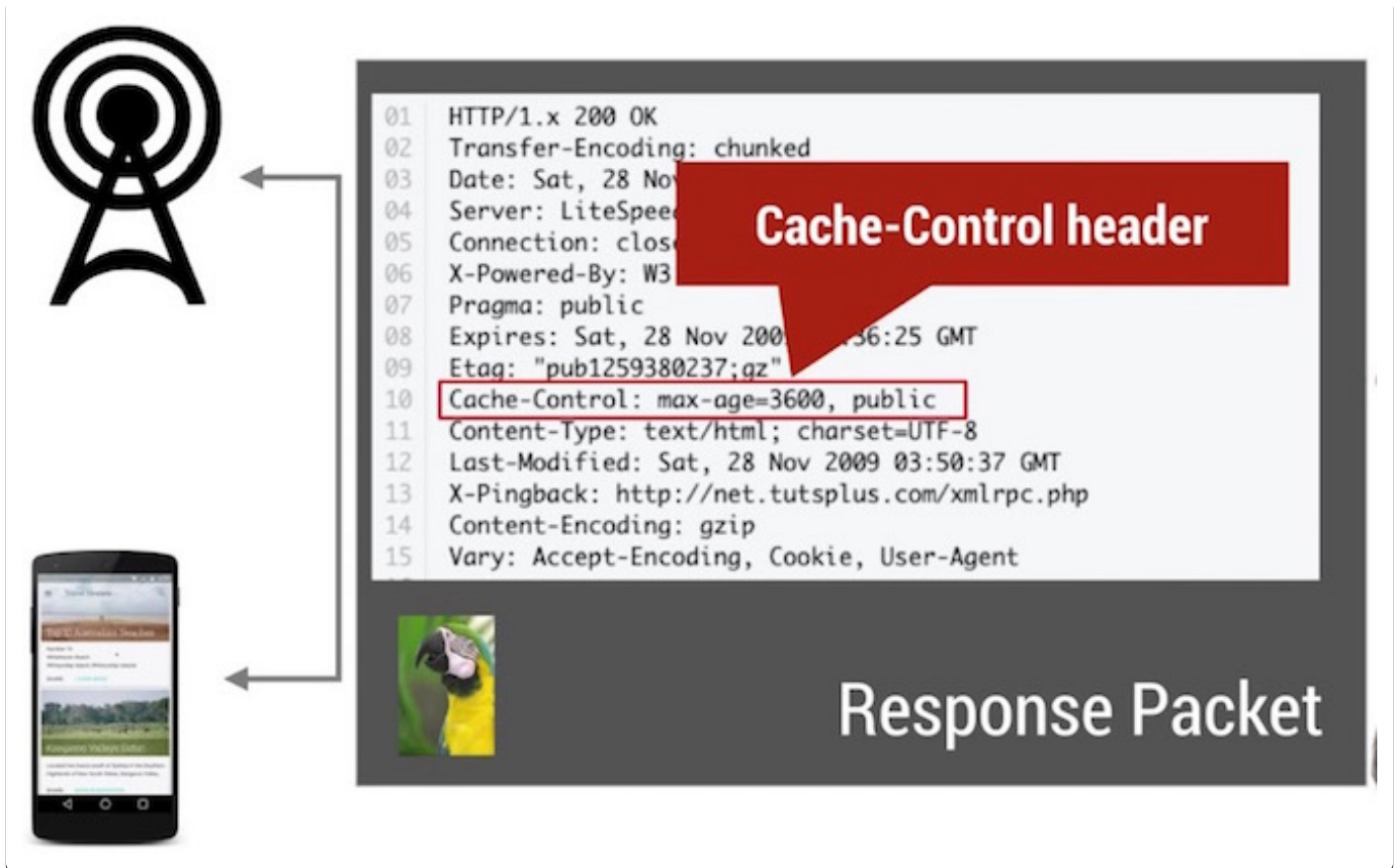
```
BitmapFactory.decodeStream((InputStream)new URL(pathToImage).getContent());
```

## Using HTTP

```
HttpClient client = new DefaultHttpClient();  
HttpGet request = new HttpGet(url);  
HttpResponse response = client.execute(request);  
//...
```

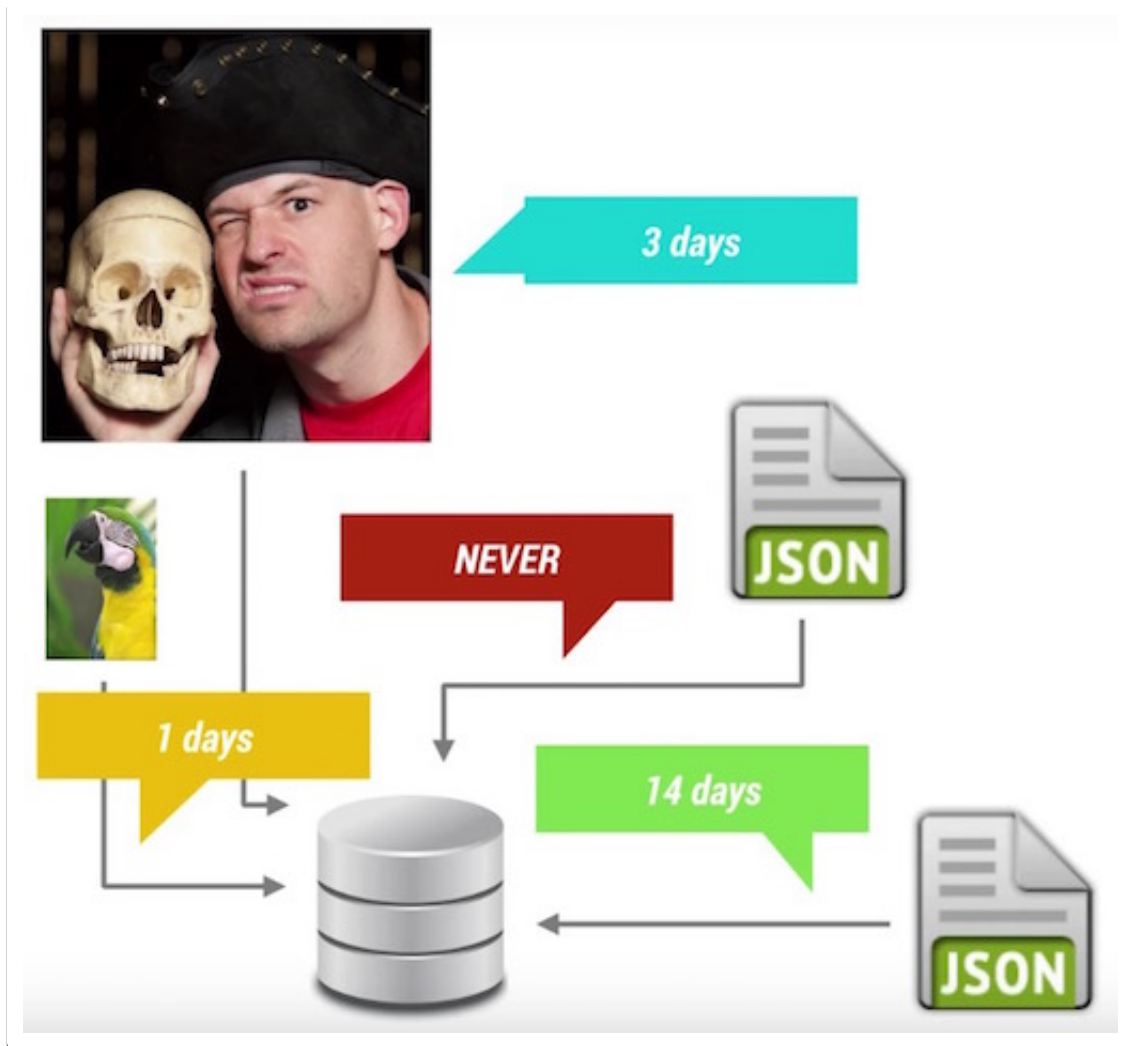
我们有两种方式清除HttpResponseCache的缓存数据：第一种方式是缓存溢出的时候删除最旧最老的文件，第二种方式是通过Http返回Header中的Cache-Control字段来进行控制的。

如下图所示：



通常来说，`HttpServletResponse`会缓存所有的返回信息，包括实际的数据与Header的部分.一般情况下，这个Cache会自动根据协议返回`Cache-Control`的内容与当前缓存的数据量来决定哪些数据应该继续保留，哪些数据应该删除。但是在一些极端的情况下，例如服务器返回的数据没有设置Cache废弃的时间，或者是本地的Cache文件系统与返回的缓存数据有冲突，或者是某些特殊的网络环境导致`HttpServletResponse`工作异常，在这些情况下就需要我们自己来实现Http的缓存Cache。

实现自定义的http缓存，需要解决两个问题：第一个是实现一个`DiskCacheManager`，另外一个则是制定Cache的缓存策略。关于`DiskCacheManager`，我们可以扩展Android系统提供的`DiskLruCache`来实现。而Cache的缓存策略，相对来说复杂一些，我们可能需把部分JSON数据设计成不能缓存的，另外一些JSON数据设计成可以缓存几天的，把缩略图设计成缓存一两天的等等，为不同的数据类型根据他们的使用特点制定不同的缓存策略。



想要比较好的实现这两件事情，如果全部自己从头开始写会比较繁琐复杂，所幸的是，有不少著名的开源框架帮助我们快速的解决了那些问题。我们可以使用[Volly](#)，[okHTTP](#)，[Picasso](#)来实现网络缓存。

实现好网络缓存之后，我们可以使用Android Studio里面的 `Network Traffic Tools` 来查看网络数据的请求与返回情况，另外我们还可以使用[AT&TARO](#)工具来抓取网络数据包进行分析查看。

## 2)Optimizing Network Request Frequencies

应用程序的一个基础功能是能够保持确保界面上呈现的信息是即时最新的，例如呈现最新的新闻，天气，信息流等等信息。但是，过于频繁的促使手机客户端应用去同步最新的服务器数据会对性能产生很大的负面影响，不仅仅使得CPU不停的在工作，内存，网络流量，电量等等都会持续的被消耗，所以在进行网络请求操作的时候一定要避免多度同步操作。

退到后台的应用为了能够在切换回前台的时候呈现最新的数据，会偷偷在后台不停的做同步的操作。这种行为会带来很严重的问题，首先因为网络请求的行为异常的耗电，其次不停的进行网络同步会耗费很多带宽流量。

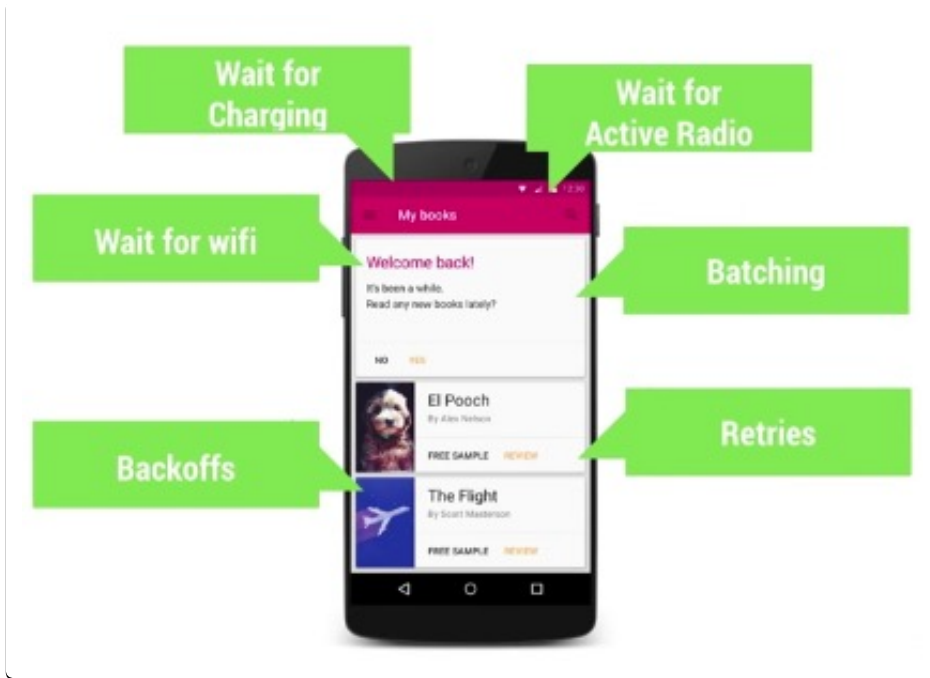


为了能够尽量的减少不必要的同步操作，我们需要遵守下面的一些规则：

- 首先我们要对网络行为进行分类，区分需要立即更新数据的行为和其他可以进行延迟的更新行为，为不同的场景进行差异化处理。
- 其次要避免客户端对服务器的轮询操作，这样会浪费很多的电量与带宽流量。解决这个问题，我们可以使用**Google Cloud Message**来对更新的数据进行推送。
- 然后在某些必须做同步的场景下，需要避免使用固定的间隔频率来进行更新操作，我们应该在返回的数据无更新的时候，使用双倍的间隔时间来进行下一次同步。
- 最后更进一步，我们还可以通过判断当前设备的状态来决定同步的频率，例如判断设备处于休眠，运动等不同的状态设计各自不同时间间隔的同步频率。



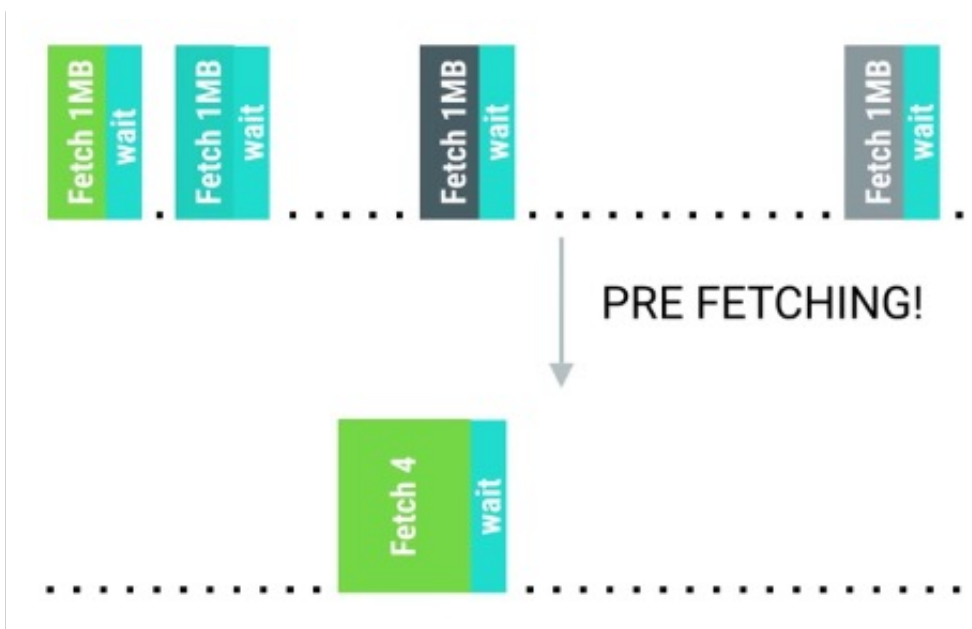
另外，我们还可以通过判断设备是否连接上WiFi，是否正在充电来决定更新的频率。为了能够方便的实现这个功能，Android为我们提供了[GCMNetworkManager](#)来判断设备当下的状态，从而设计更加高效的网络同步操作，如下图所示：



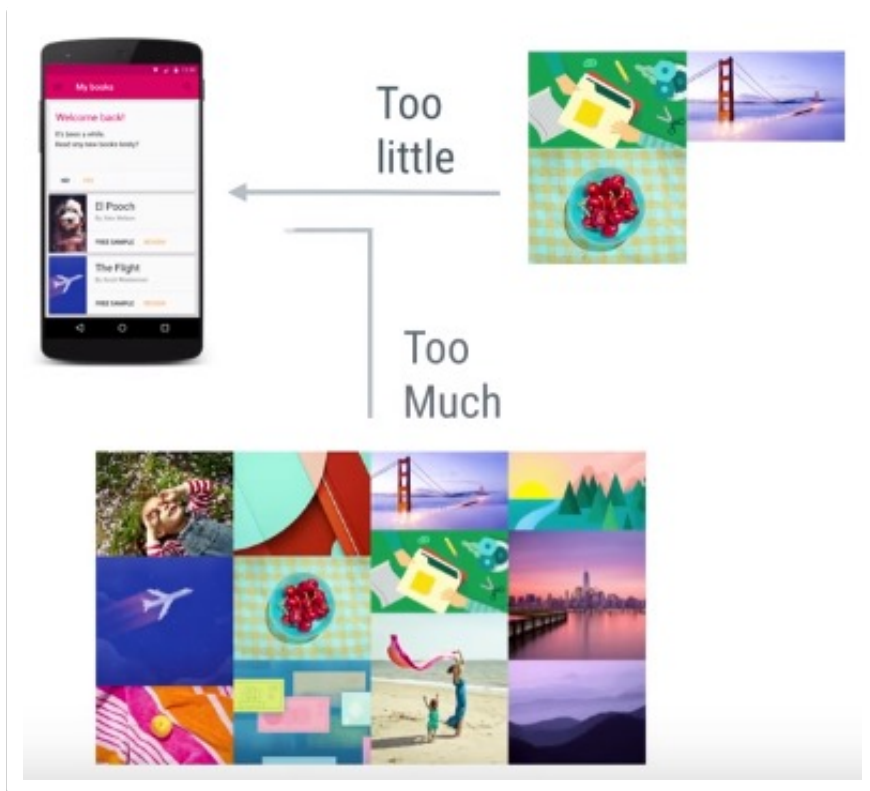
### 3)Effective Prefetching

关于提升网络操作的性能，除了避免频繁的网络同步操作之外，还可以使用捆绑批量访问的方式来减少访问的频率，为了达到这个目的，我们就需要了解**Prefetching**。

举个例子，在某个场景下，一开始发出了网络请求得到了某张图片，隔了10s之后，发出第二次请求想要拿到另外一张图片，再隔了6s发出第三张图片的网络请求。这会导致设备的无线蜂窝一直处于高消耗的状态。**Prefetching**就是预先判定那些可能马上就会使用到的网络资源，捆绑一起集中进行网络请求。这样能够极大的减少电量的消耗，提升设备的续航时间。



使用**Prefetching**的难点在于如何判断事先获取的数据量到底是多少，如果预取的数据量偏少，那么就起不到什么效果，但是如果预取过多，又可能导致访问的时间过长。

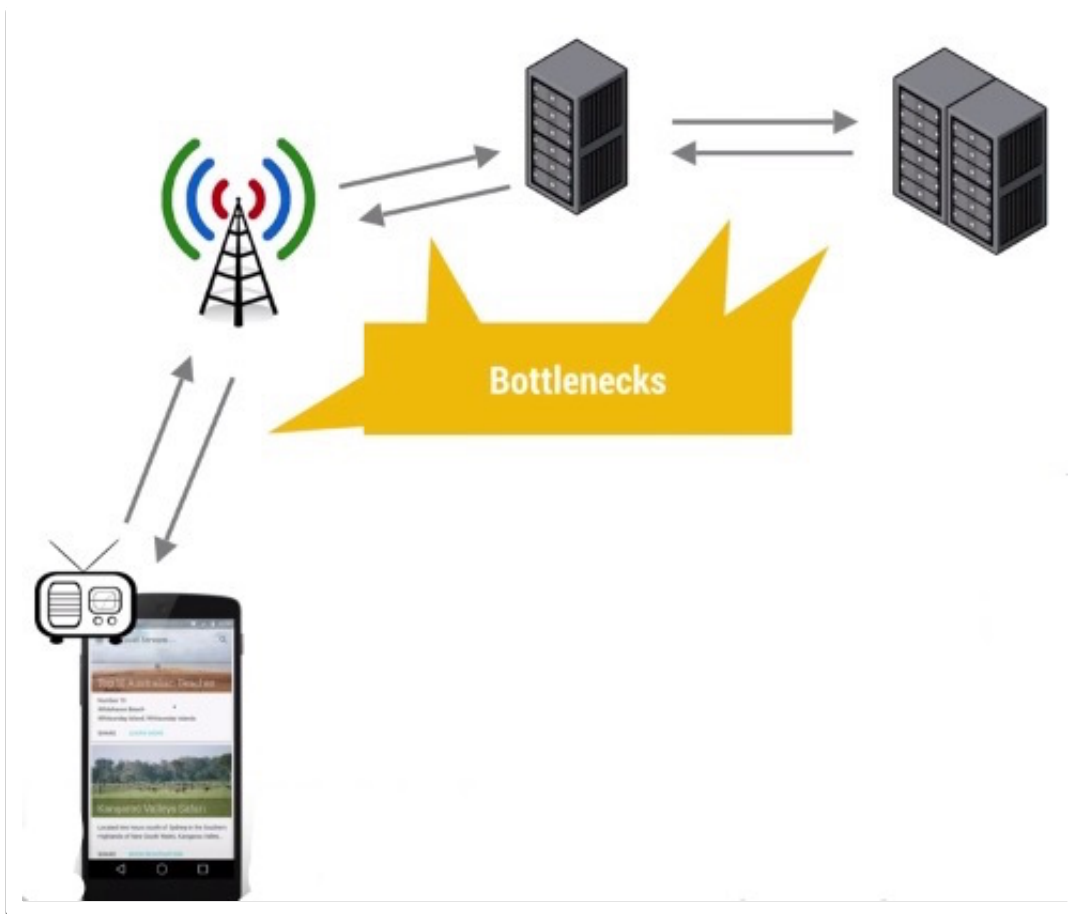


那么问题来了，到底预取多少才比较合适呢？一个比较普适的规则是，在3G网络下可以预取1-5Mb的数据量，或者是按照提前预期后续1-2分钟的数据作为基线标准。在实际的操作当中，我们还需要考虑当前的网络速度来决定预取的数据量，例如在同样的时间下，4G网络可以获取到12张图片的数据，而2G网络则只能拿到3张图片的数据。所以，我们还需要把当前的网络环境情况添加到设计预取数据量的策略当中去。判断当前设备的状态与网络情况，可以使用前面提到过的[GCMNetworkManager](#)。

## 4) Adapting to Latency

网络延迟通常来说很容易被用户察觉到，严重的网络延迟会对用户体验造成很大的影响，用户很容易抱怨应用程序写的不好。

一个典型的网络操作行为，通常包含以下几个步骤：首先手机端发起网络请求，到达网络服务运营商的基站，再转移到服务提供者的服务器上，经过解码之后，接着访问本地的存储数据库，获取到数据之后，进行编码，最后按照原来传递的路径逐层返回。如下图所示：



在上面的网络请求链路当中的任何一个环节都有可能导致严重的延迟，成为性能瓶颈，但是这些环节可能出现的问题，客户端应用是无法进行调节控制的，应用能够做的就只是根据当前的网络环境选择当下最佳的策略来降低出现网络延迟的概率。主要的实施步骤有两步：第1步检测收集当前的网络环境信息，第2步根据当前收集到的信息进行网络请求行为的调整。

关于第1步检测当前的网络环境，我们可以使用系统提供的API来获取到相关的信息，如下图所示：

```
ConnectivityManager cm = (ConnectivityManager) context.getSystemService(Context.CONNECTIVITY_SERVICE);
NetworkInfo activeNetwork = cm.getActiveNetworkInfo();

if (activeNetwork.getType() != ConnectivityManager.TYPE_WIFI)
{
    String typeName = activeNetwork.getSubtypeName();
    int type = activeNetwork.getSubtype();
    switch(type)
    {
        // TelephonyManager.NETWORK_TYPE_* enums
    }
}
```

通过上面的示例，我们可以获取到移动网络的详细子类型，例如4G(LTE),3G等等，详细分类见下图，获取到详细的移动网络类型之后，我们可以根据当前网络的速率来调整网络请求的行为：

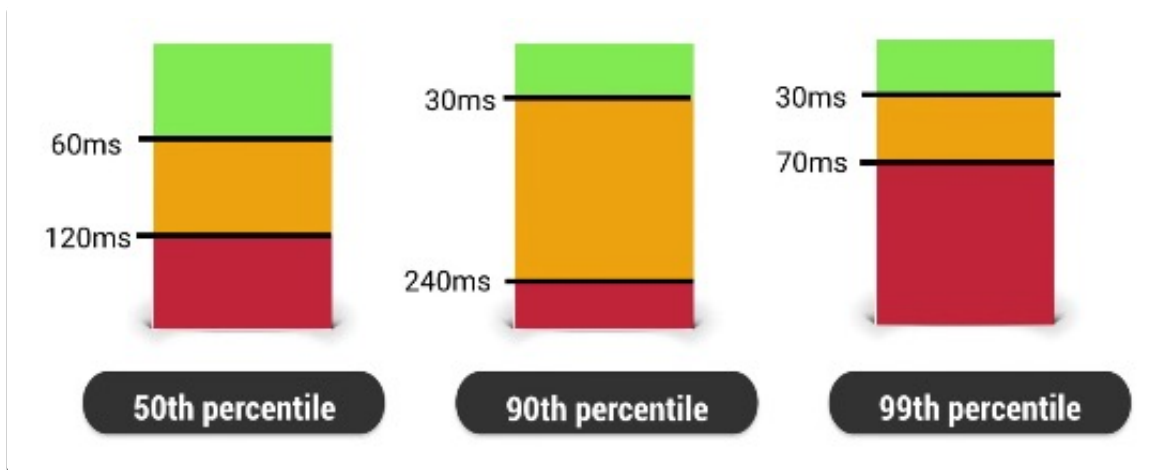


<i>Value</i>	<i>Speed</i>
NETWORK_TYPE_GPRS	<i>SLOW</i>
NETWORK_TYPE_EDGE	<i>SLOW</i>
NETWORK_TYPE_UMTS	<i>MEDIUM</i>
NETWORK_TYPE_CDMA	<i>SLOW</i>
NETWORK_TYPE_EVDO_0	<i>SLOW</i>
NETWORK_TYPE_EVDO_A	<i>MEDIUM</i>
NETWORK_TYPE_1xRTT	<i>SLOW</i>
NETWORK_TYPE_HSDPA	<i>MEDIUM</i>
NETWORK_TYPE_HSUPA	<i>FAST</i>
NETWORK_TYPE_HSPA	<i>FAST</i>
NETWORK_TYPE_IDEN	<i>SLOW</i>
NETWORK_TYPE_EVDO_B	<i>MEDIUM</i>
NETWORK_TYPE_LTE	<i>FAST</i>
NETWORK_TYPE_EHRPD	<i>SLOW</i>

关于第2步根据收集到的信息进行策略的调整，通常来说，我们可以把网络请求延迟划分为三档：例如把网络延迟小于60ms的划分为GOOD，大于220ms的划分为BAD，介于两者之间的划分为OK（这里的60ms，220ms会根据不同的场景提前进行预算推测）。如果网络延迟属于GOOD的范畴，我们就可以做更多比较激进的预取数据的操作，如果网络延迟属于BAD的范畴，我们就应该考虑把当下的网络请求操作Hold住等待网络状况恢复到GOOD的状态再进行处理。



前面提到说60ms，220ms是需要提前自己预测的，可是预测的工作相当复杂。首先针对不同的机器与网络环境，网络延迟的三档阈值都不太一样，出现的概率也不尽相同，我们会需要针对这些不同的用户与设备选择不同的阈值进行差异化处理：



Android官方为了帮助我们设计自己的网络请求策略，为我们提供了模拟器的网络流量控制功能来对实际环境进行模拟测量，或者还可以使用AT&T提供的[AT&T Network Attenuator](#)来帮助预估网络延迟。

## 5)Minimizing Asset Payload

为了能够减小网络传输的数据量，我们需要对传输的数据做压缩的处理，这样能够提高网络操作的性能。首先不同的网络环境，下载速度以及网络延迟是存在差异的，如下图所示：

Name	Generation	Max Down (kbps)	Latency (ms)
GPRS	2G	237	300-1000
EDGE	2G	284	300-1000
UMTS	3G	2,000	100-500
HSPA	3G	3,600	100-500
HSPA+	3G	42,000	100-500
LTE	4G	100,000	<100

如果我们选择在网速更低的网络环境下进行数据传输，这就意味着需要执行更长的时间，而更长的网络操作行为，会导致电量消耗更加严重。另外传输的数据如果不做压缩处理，也同样会增加网络传输的时间，消耗更多的电量。不仅如此，未经过压缩的数据，也会消耗更多的流量，使得用户需要付出更多的流量费。

通常来说，网络传输数据量的大小主要由两部分组成：图片与序列化的数据，那么我们需要做的就是减少这两部分的数据传输大小，分下面两个方面来讨论。

- A)首先需要做的是减少图片的大小，选择合适的图片保存格式是第一步。下图展示了PNG,JPEG,WEBP三种主流格式在占用空间与图片质量之间的对比：

Name	Alpha	Lossy	Quality	Size
PNG	Yes		Great	Bad
JPG		Yes	Good	Good
WEBP	Yes	Yes	Good	Good

对于JPEG与WEBP格式的图片，不同的清晰度对占用空间的大小也会产生很大的影响，适当的减少JPG Quality，可以大大的缩小图片占用的空间大小。

另外，我们需要为不同的使用场景提供当前场景下最合适的图片大小，例如针对全屏显示的情况我们会需要一张清晰度比较高的图片，而如果只是显示为缩略图的形式，就只需要服务器提供一个相对清晰度低很多的图片即可。服务器应该支持到为不同的使用场景分别准备多套清晰度不一样的图片，以便在对应的场景下能够获取到最适合自己的图片。这虽然会增加服务端的工作量，可是这个付出却十分值得！

- B)其次需要做的是减少序列化数据的大小。JSON与XML为了提高可读性，在文件中加入了大量的符号，空格等等字符，而这些字符对于程序来说是没有任何意义的。我们应该使用Protocal Buffers，Nano-Proto-Buffers，FlatBuffer来减小序列化的数据的大小。

Android系统为我们提供了工具来查看网络传输的数据情况，打开Android Studio的Monitor，里面有网络访问的模块。或者是打开AT&T提供的[ARO](#)工具来查看网络请求状态。

## 6)Service Performance Patterns

Service是Android程序里面最常用的基础组件之一，但是使用Service很容易引起电量的过度消耗以及系统资源的未及时释放。学会在何时启用Service以及使用何种方式杀掉Service就显得十分有必要了。

简要过一下Service的特性：Service和UI没有关联，Service的创建，执行，销毁Service都是需

要占用系统时间和内存的。另外Service是默认运行在UI线程的，这意味着Service可能会影响到系统的流畅度。

使用Service应该遵循下面的一些规则：

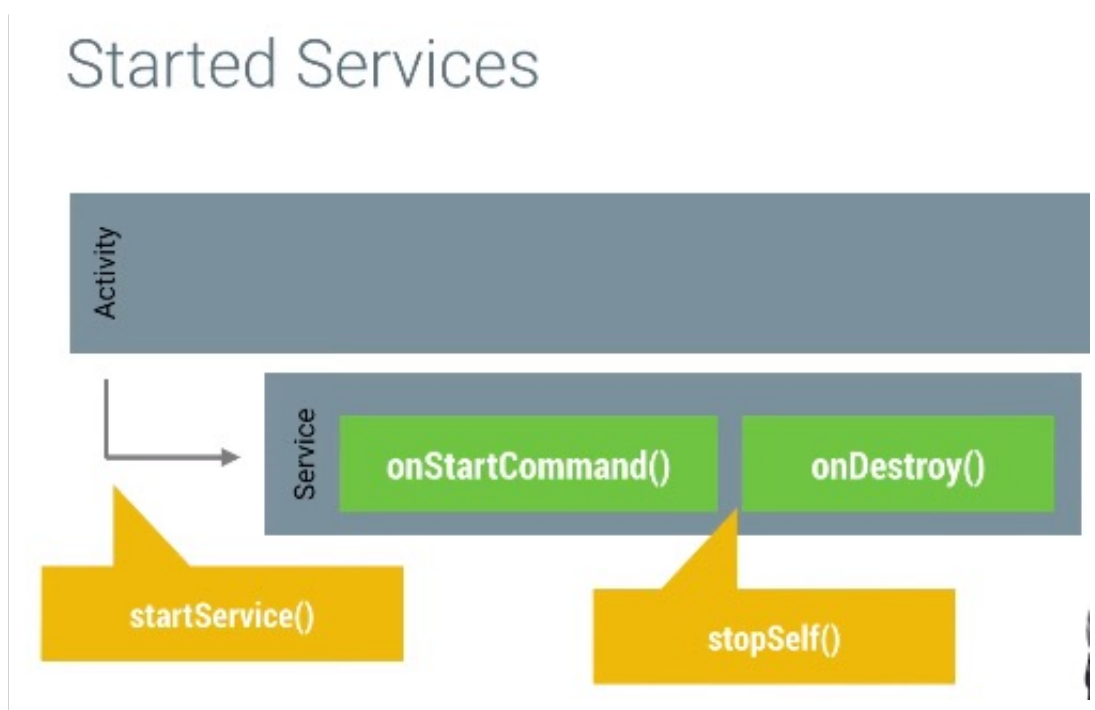
- 避免错误的使用Service，例如我们不应该使用Service来监听某些事件的变化，不应该搞一个Service在后台对服务器不断的进行轮询(应该使用Google Cloud Messaging)
- 如果已经事先知道Service里面的任务应该执行在后台线程(非默认的主线程)的时候，我们应该使用IntentService或者结合HandlerThread，AsyncTask Loader实现的Service。

Android系统为我们提供了以下的一些异步相关的工具类

- GCM
- BroadcastReceiver
- LocalBroadcastReceiver
- WakefulBroadcastReceiver
- HandlerThreads
- AsyncTaskLoaders
- IntentService

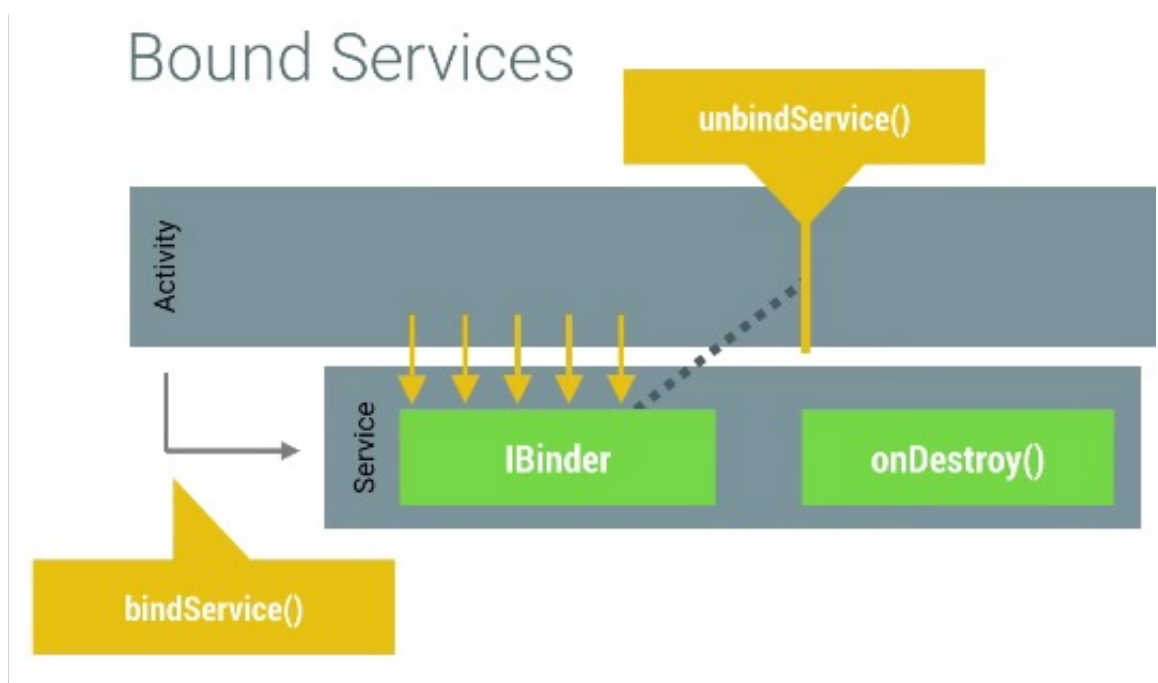
如果使用上面的诸多方案还是无法替代普通的Service，那么需要注意的就是如何正确的关闭Service。

- 普通的Started Service，需要通过stopSelf()来停止Service

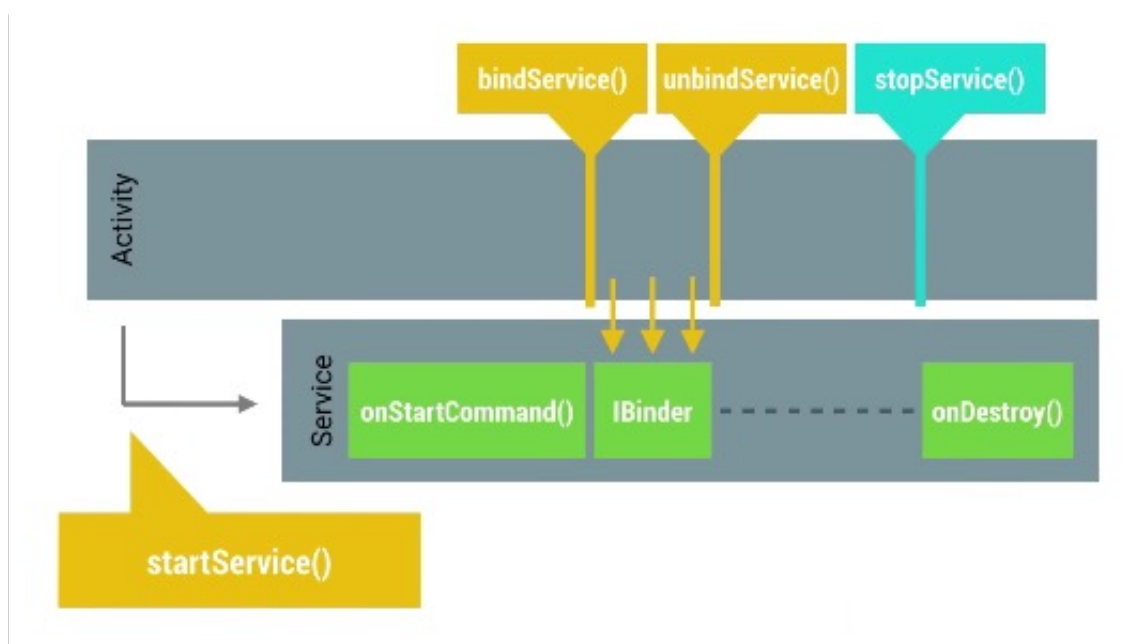




- 另外一种Bound Service，会在其他组件都unBind之后自动关闭自己



把上面两种Service进行合并之后，我们可以得到如下图所示的Service(相关知识，还可以参考<http://hukai.me/android-notes-services/>, <http://hukai.me/android-notes-bound-services/>)



## 7)Removing unused code

使用第三方库(library)可以在不用自己编写大量代码的前提下帮助我们解决一些难题，节约大量的时间，但是这些引入的第三方库很可能会导致主程序代码臃肿冗余。

如果我们处在人力，财力都相对匮乏的情况下，通常会倾向大量使用第三方库来帮助编写应用

程序。这其实是无可厚非的，那些著名的第三方库的可行性早就被很多应用所采用并实践证明过。但是这里面存在的问题是，如果我们因为只需要某个**library**的一小部分功能而把整个**library**都导入自己的项目，这就会引起代码臃肿。一旦发生代码臃肿，用户就会下载到安装包偏大的应用程序，另外因为代码臃肿，还很有可能会超过单个编译文件只能有65536个方法的上限。解决这个问题的办法是使用**MultiDex**的方案，可是这实在是无奈之举，原则上，我们还是应该尽量避免出现这种情况。

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

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

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

但是**Proguard**还是不足够聪明到能够判断哪些类，哪些方法是不能够被混淆的，针对这些情况，我们需要手动的把这些需要保留的类名与方法名添加到**Proguard**的配置文件中，如下图所示：



```
# Needed just to be safe in terms of keeping Google API service model classes
-keep class com.google.api.services.*.model.*
-keep class com.google.api.client.**
-keepattributes Signature,RuntimeVisibleAnnotations,AnnotationDefault
```



```
# Assume dependency libraries Just Work(TM)
-dontwarn com.google.android.youtube.**
-dontwarn com.google.android.analytics.**
-dontwarn com.google.common.**
```



```
-keep class com.android.**
-keep class com.google.android.**
-keep class com.google.android.gms.**
-keep class com.google.android.gms.location.**
-keep class com.google.api.client.**
-keep class com.google.maps.android.**
```

在使用library的时候，需要特别注意这些library在proguard配置上的说明文档，我们需要把这些配置信息添加到自己的主项目中。关于Proguard的详细说明，请看官方文档<http://developer.android.com/tools/help/proguard.html>

## 8)Removing unused resources

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

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

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

为了辅助gradle对资源进行瘦身，或者是某些时候的特殊需要，我们可以通过`tools:keep`或者是`tools:discard`标签来实现对特定资源的保留与废弃，如下图所示：

```
<resources xmlns:tools="http://  
schemas.android.com/tools"  
  
    tools:keep= "@layout/l_used*_c,  
                @layout/l_used_b*"   
  
    tools:discard="@layout/unused2"  
/>
```

Gradle目前无法对`values`，`drawable`等根据运行时来决定使用的资源进行优化，对于这些资源，需要我们自己来确保资源不会有冗余。

## 9)Perf Theory: Caching

当我们讨论性能优化的时候，缓存是最常见最有效的策略之一。无论是为了提高CPU的计算速度还是提高数据的访问速度，在绝大多数的场景下，我们都会使用到缓存。关于缓存是如何提高效率的，这里就不赘述了。

那么在什么地方，在何时应该利用好缓存来提高效率呢？请看下面的例子，很明显的演示了在某些细节上是如何利用缓存的原理来提高代码的执行效率的：




```
matrix4x4 projectionMatrix;  
projectionMatrix.randomizeValues();  
for(int i=0; i < 1024;i++)  
{  
    int determinant = projectionMatrix.determinant()  
    if (determinant < i* magicScalar)  
    {  
        SaveTheWorld();  
        break;  
    }  
}
```



Will always be the same value

```
matrix4x4 projectionMatrix;  
projectionMatrix.randomizeValues();  
    int determinant = projectionMatrix.determinant()  
for(int i=0; i < 1024;i++)  
{  
    if (determinant < i* magicScalar)  
    {  
        SaveTheWorld();  
        break;  
    }  
}
```



Calculate once, use often

类似上面的例子采用缓存原理的地方还有很多，例如缓存到内存里面的图片资源，网络请求返回数据的缓存等等。总之，使用缓存就是为了减少不必要的操作，尽量复用已有的对象来提高效率。

## 10)Perf Theory: Approximation(近似法)

很多时候，我们都需要学会在性能更优与体验更好之间做一定的权衡取舍。为了获取更好的表现性能，我们可能会需要牺牲一些用户体验，例如把某些细节做删除或者是降级处理以便有更好的性能。例如，导航类的应用，如果在导航期间是不停的执行定位的操作，这样能够很及时的获取到最新的位置信息以及当下位置相关的其他提示信息，但是这样会导致网络流量以及手机电量的过度消耗。所以我们可以做一定的降级处理，每隔固定的一段时间才去获取一次位置信息，损失一点及时性来换取更长的续航时间。

还有很多地方都会用到近似法则来优化程序的性能，例如使用一张比较接近实际大小的图片来替代原图，换取更快的加载速度。所以对于那些对计算结果要求不需要十分精确的场景，我们

可以使用近似法则来提高程序的性能。

## 11)Perf Theory: Culling(遴选，挑选)

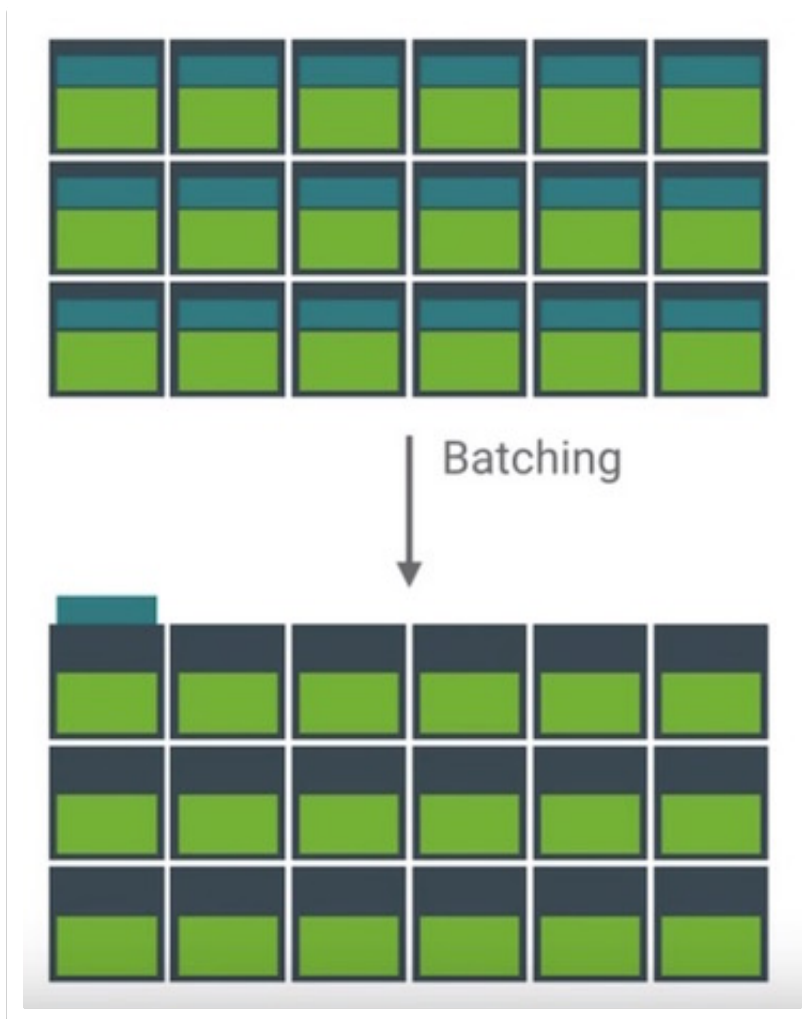
在以前的性能优化课程里面，我们知道可以通过减少**Overdraw**来提高程序的渲染性能（主要手段有移除非必须的**background**，减少重叠的布局，使用**clipRect**来提高自定义**View**的绘制性能），今天在这里要介绍的另外一个提高性能的方法是逐步对数据进行过滤筛选，减小搜索的数据集，以此提高程序的执行性能。例如我们需要搜索到居住在某个地方，年龄是多少，符合某些特定条件的候选人，就可以通过逐层过滤筛选的方式来提高后续搜索的执行效率。

## 12)Perf Theory: Threading

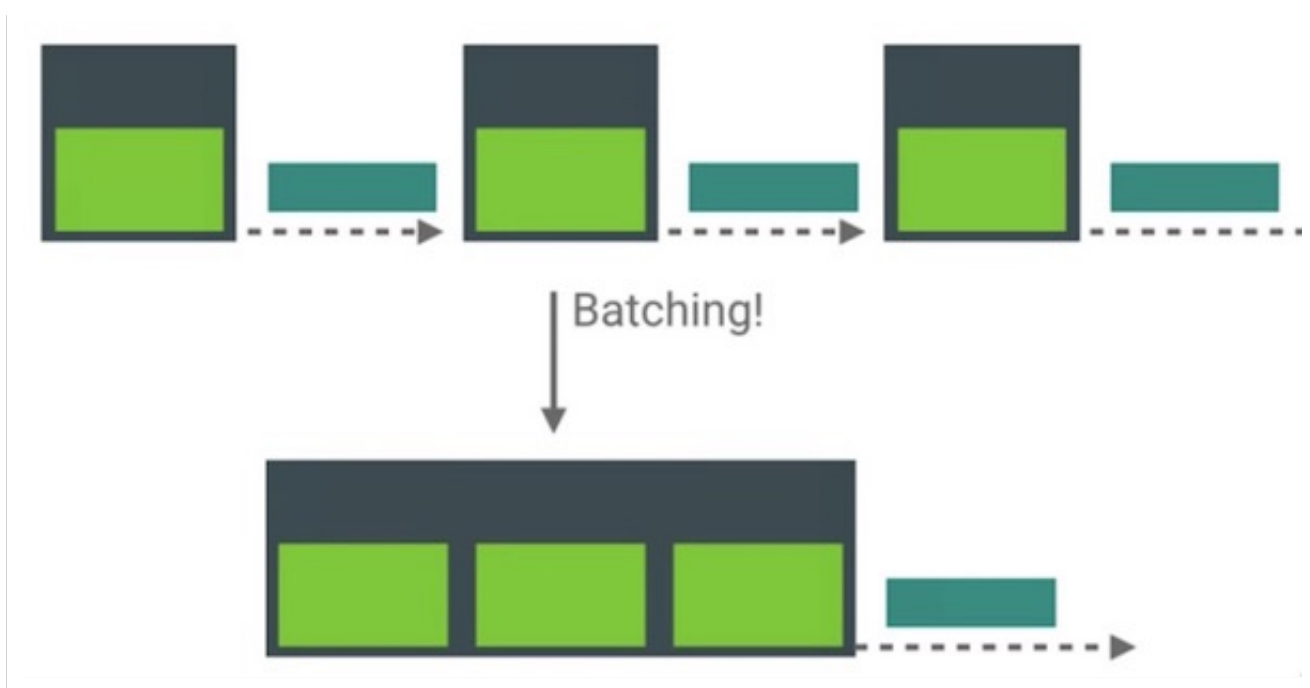
使用多线程并发处理任务，从某种程度上可以快速提高程序的执行性能。对于**Android**程序来说，主线程通常也成为**UI**线程，需要处理**UI**的渲染，响应用户的操作等等。对于那些可能影响到**UI**线程的任务都需要特别留意是否有必要放到其他的线程来进行处理。如果处理不当，很有可能引起程序**ANR**。关于多线程的使用建议，可以参考官方的培训课程<http://developer.android.com/training/best-background.html>

## 13)Perf Theory: Batching

关于**Batching**，在前几季的性能优化课程里面也不止一次提到，下面使用一张图演示下**Batching**的原理：

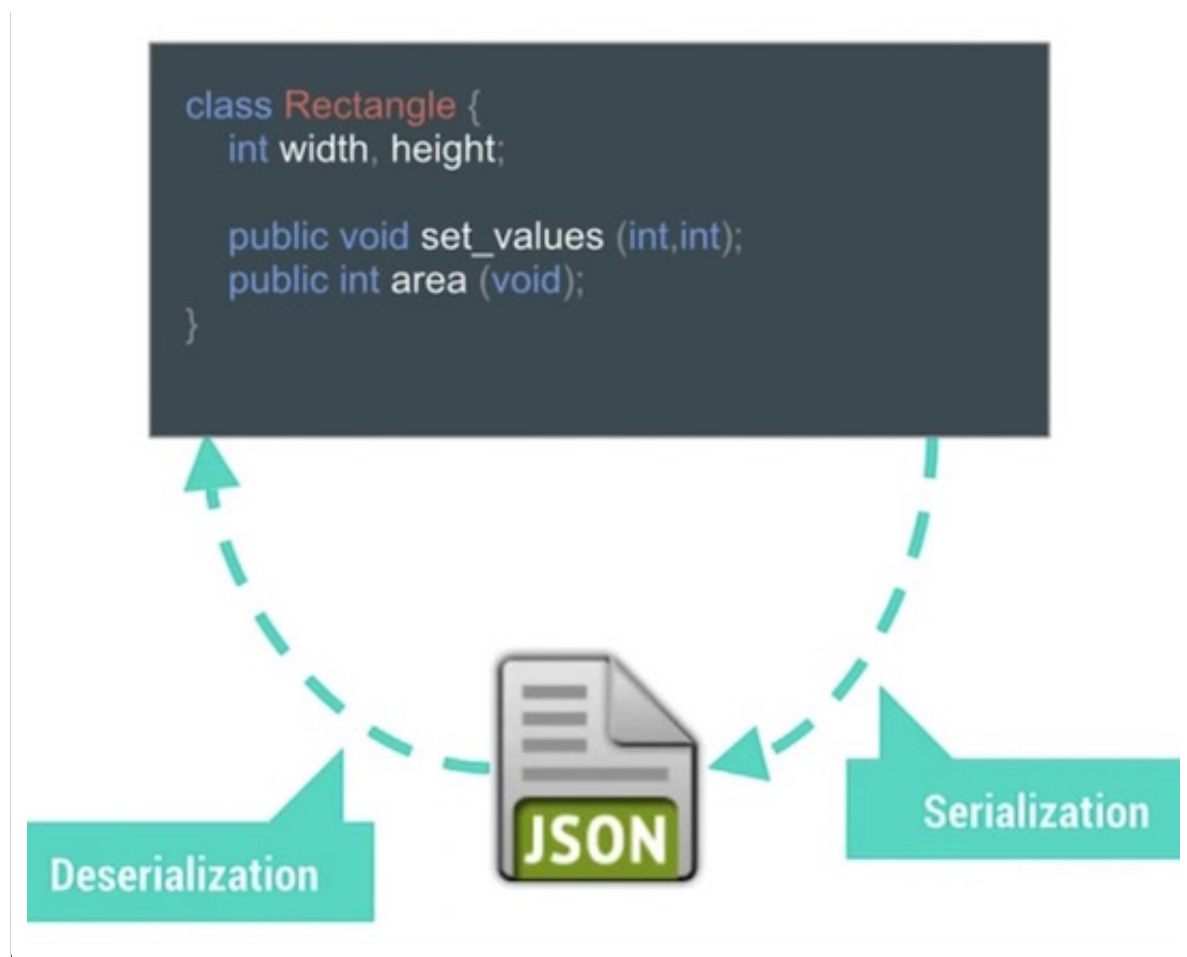


网络请求的批量执行是另外一个比较适合说明**batching**使用场景的例子，因为每次发起网络请求都相对来说比较耗时耗电，如果能够做到批量一起执行，可以大大的减少电量的消耗。



## 14)Serialization performance

数据的序列化是程序代码里面必不可少的组成部分，当我们讨论到数据序列化的性能的时候，需要了解有哪些候选的方案，他们各自的优缺点是什么。首先什么是序列化？用下面的图来解释一下：



数据序列化的行为可能发生在数据传递过程中的任何阶段，例如网络传输，不同进程间数据传递，不同类之间的参数传递，把数据存储到磁盘上等等。通常情况下，我们会把那些需要序列化的类实现**Serializable**接口(如下图所示)，但是这种传统的做法效率不高，实施的过程会消耗更多的内存。



```
class Rectangle implements
Serializable {
    int width, height;

    public void set_values (int,int);
    public int area (void);
}
```

***BAD performance!***

但是我们如果使用GSON库来处理这个序列化的问题，不仅仅执行速度更快，内存的使用效率也更高。Android的XML布局文件会在编译的阶段被转换成更加复杂的格式，具备更加高效的执行性能与更高的内存使用效率。

***GREAT performance!***

```
import com.google.gson.annotations.SerializedName;
class Rectangle {
    @SerializedName("w")
    public int width;

    @SerializedName("h")
    public int height;
}
```

***Serialization***

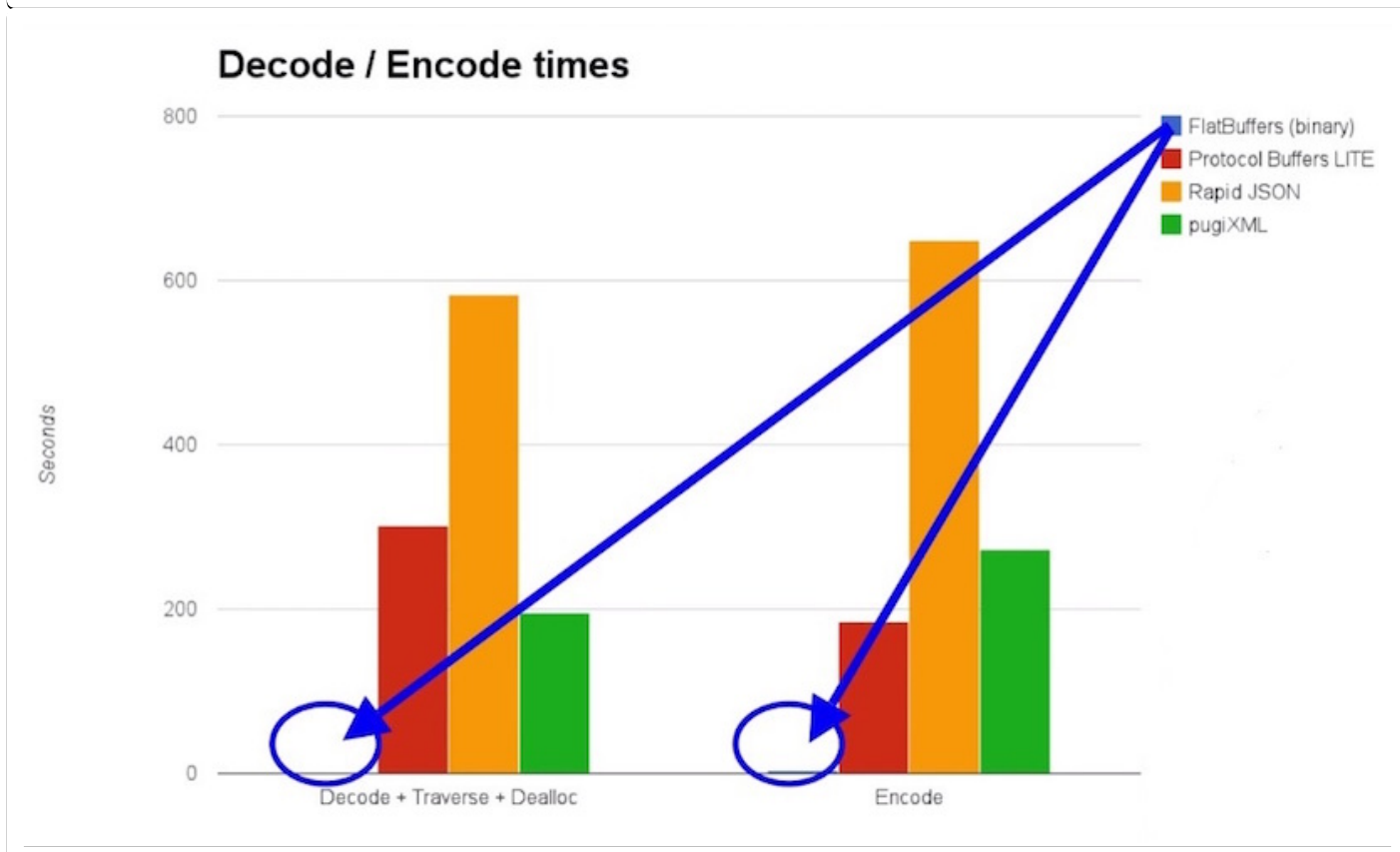
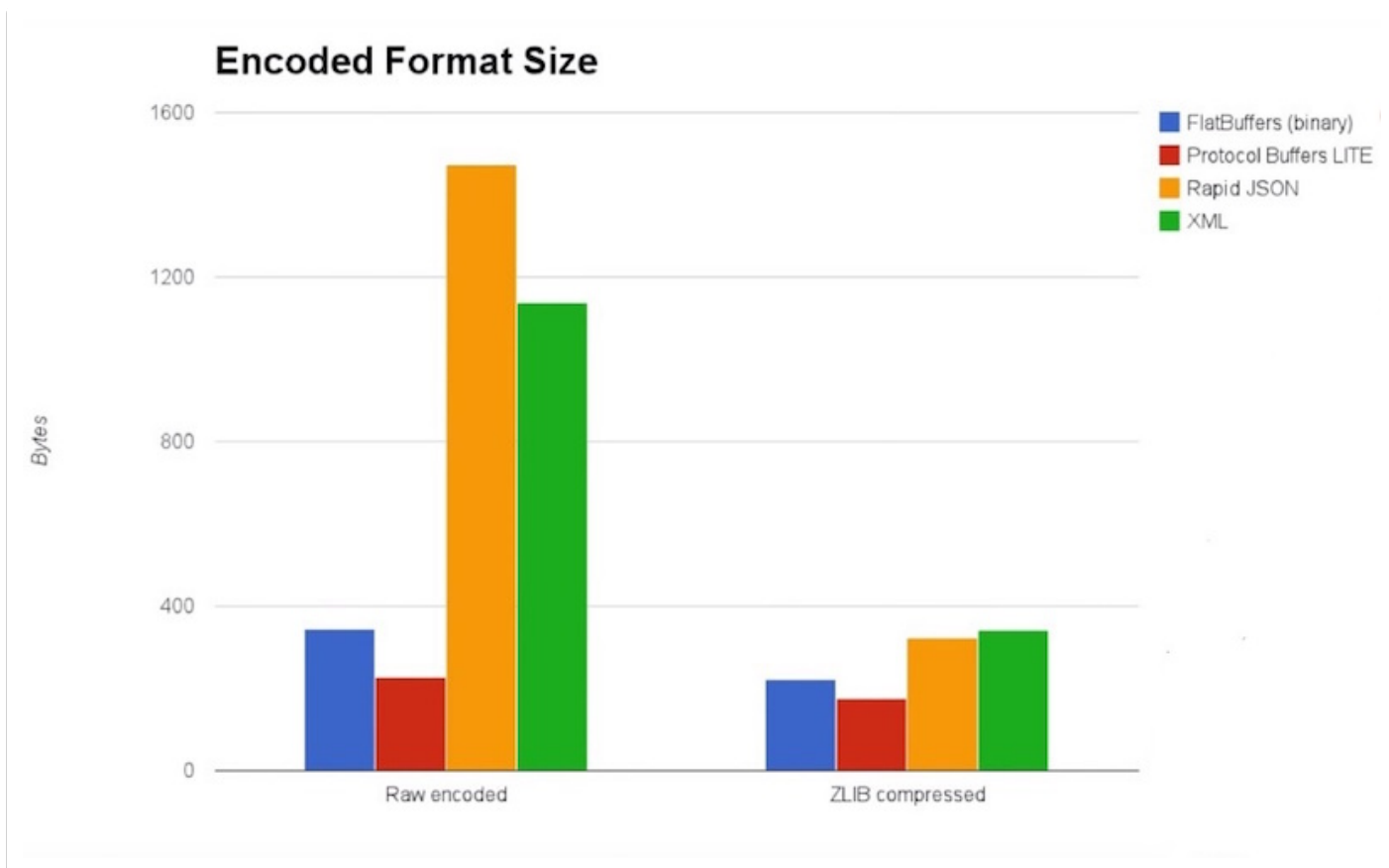


下面介绍三个数据序列化的候选方案：

- **Protocol Buffers**：强大，灵活，但是对内存的消耗会比较大，并不是移动终端上的最佳选择。

- **Nano-Proto-Buffers**: 基于Protocol, 为移动终端做了特殊的优化, 代码执行效率更高, 内存使用效率更佳。
- **FlatBuffers**: 这个开源库最开始是由Google研发的, 专注于提供更优秀的性能。

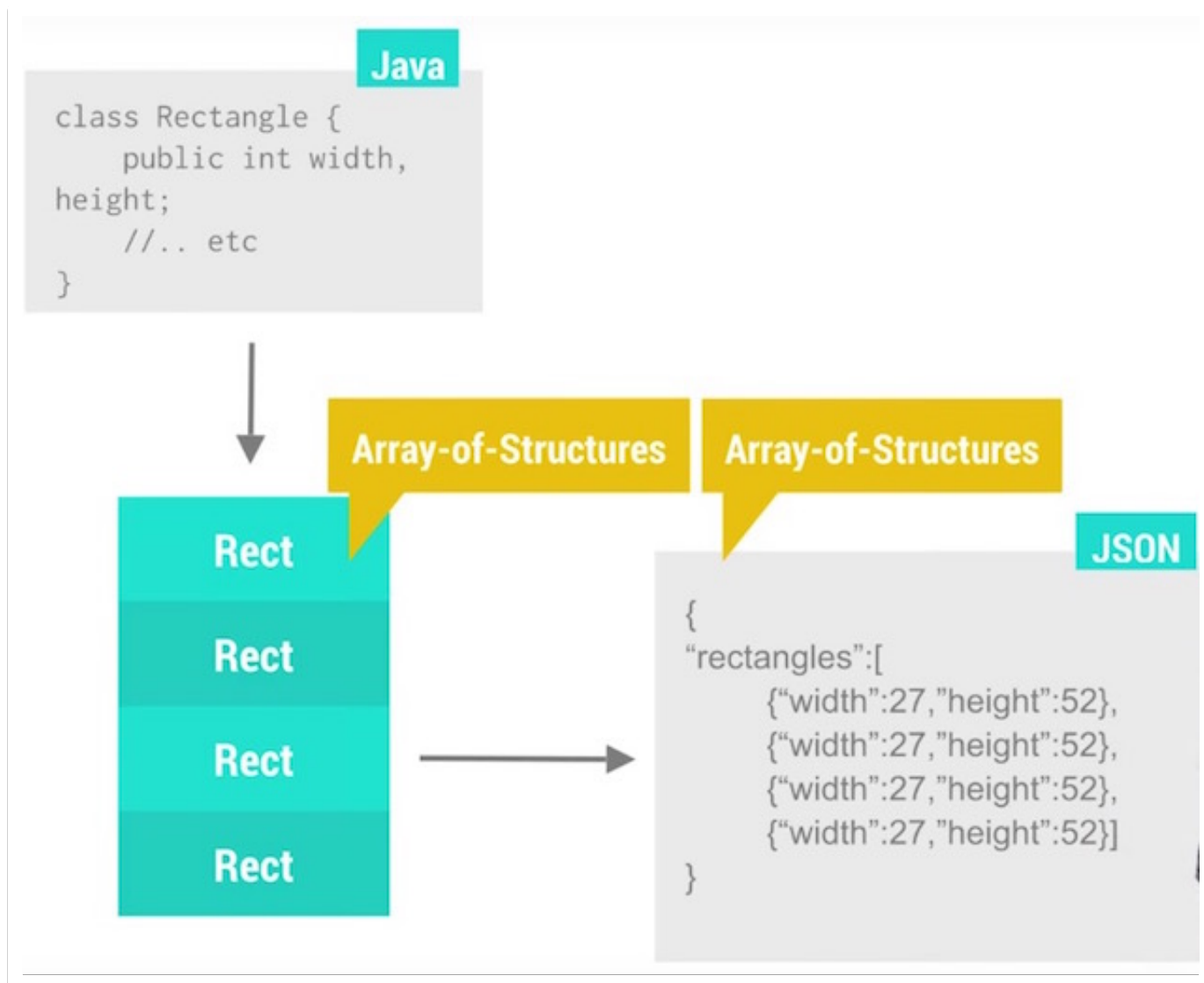
上面这些方案在性能方面的数据对比如下图所示:



为了避免序列化带来的性能问题，我们其实可以考虑使用SharedPreference或者SQLite来存储那些数据，避免需要先把那些复杂的数据进行序列化的操作。

## 15)Smaller Serialized Data

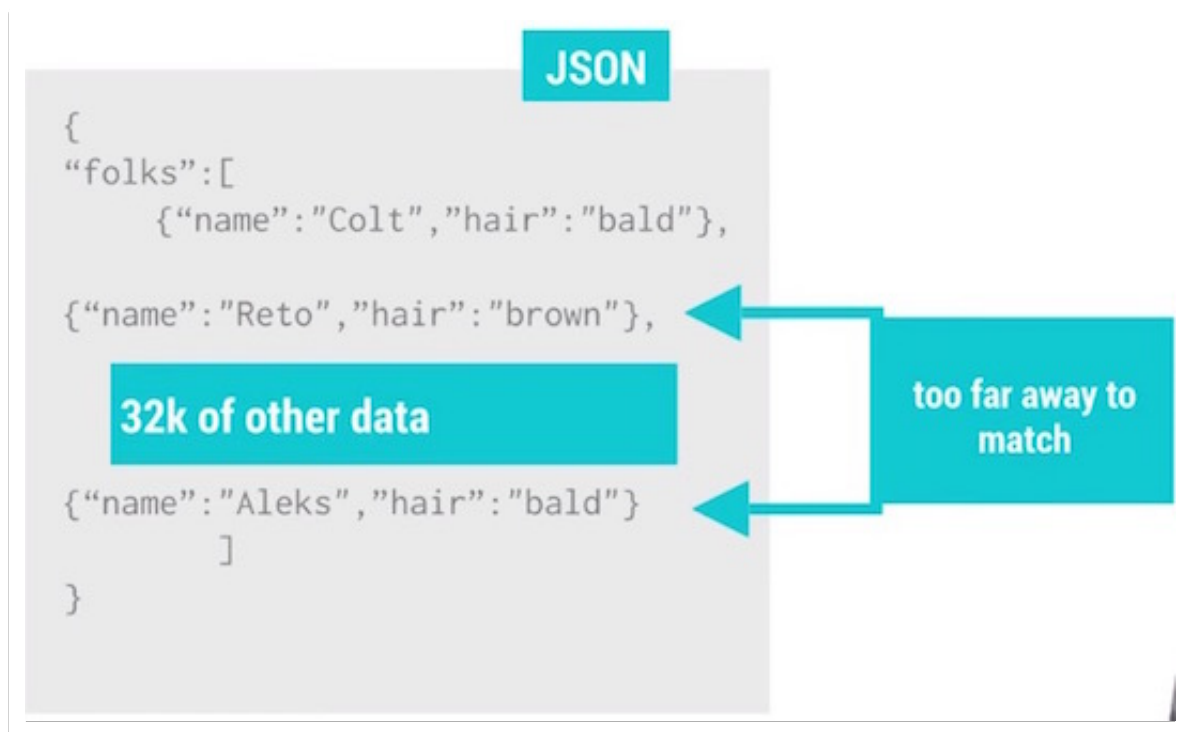
数据呈现的顺序以及结构会对序列化之后的空间产生不小的影响。通常来说，一般的数据序列化的过程如下图所示：



上面的过程，存在两个弊端，第一个是重复的属性名称：

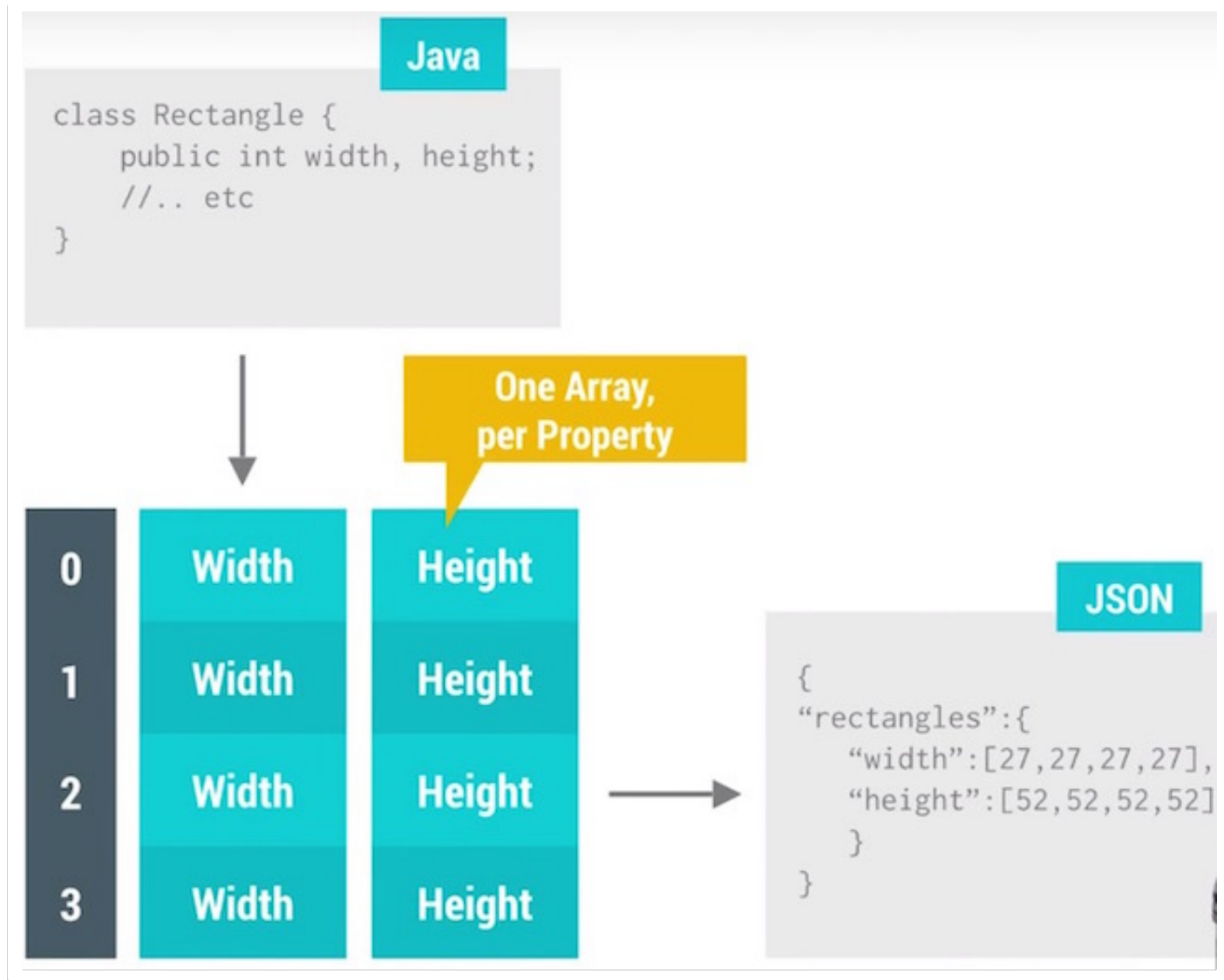


另外一个GZIP没有办法对上面的数据进行更加有效的压缩，假如相似数据间隔了32k的数据量，这样GZIP就无法进行更加有效的压缩：



但是我们稍微改变下数据的记录方式，就可以得到占用空间更小的数据，如下图所示：





通过优化，至少有三方面的性能提升，如下图所示：

1) 减少了重复的属性名：

### Array-of-Structs

```
{
  "rectangles":[
    {"width":27,"height":52},
    {"width":27,"height":52},
    {"width":27,"height":52},
    {"width":27,"height":52}]
}
```

**Property Name  
Duplicated Often**

### Struct-of-Arrays

```
{
  "rectangles":{
    "width":[27,27,27,27],
    "height":[52,52,52,52]
  }
}
```

**Property Name  
Listed once**

2) 使得GZIP的压缩效率更高:

### Array-of-Structs

```
{
  "folks":[
    {"name":"Colt","hair":"bald"},
    {"name":"Reto","hair":"brown"},
    {"name":"Aleks","hair":"bald"}
  ]
}
```

**32k of other data**

**Similar values  
far away;  
not compressed**

### Struct-of-Arrays

```
{
  "folks":{
    "names":["Colt","Reto","Aleks"],
    "hair":["bald","brown","bald"]
  }
}
```

**32k of other data**

**Similar values, super  
close!**

3) 同样的数据类型可以批量优化:

```
{
  "folks":{
    "names":["Colt","Reto","Aleks"],
    "ID":[101,102,105],
    "hair":["bald","brown","bal"]
  }
}
```

**Numerics**  
Delta encode

**Repeatable strings**  
gzip

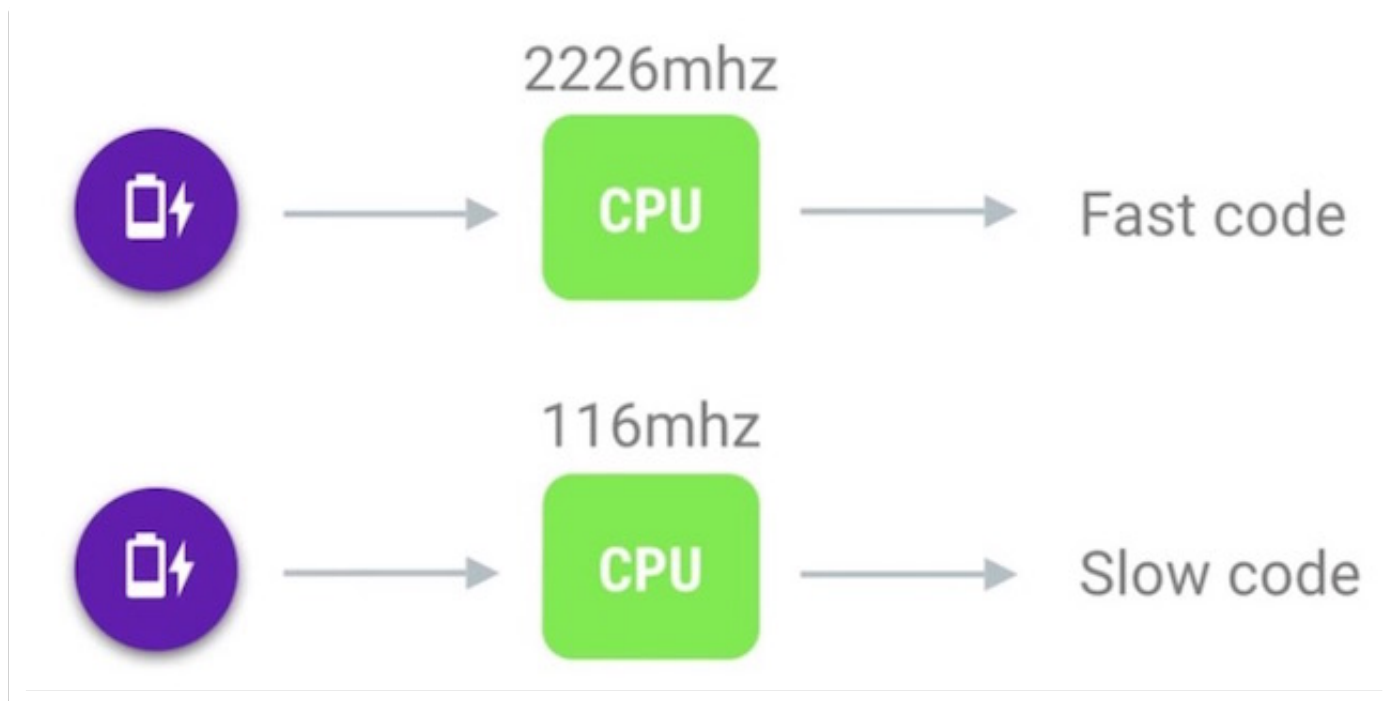
## 16)Caching UI data

如今绝大多数的应用界面上呈现的数据都依赖于网络请求返回的结果，如何做到在网络数据返回之前避免呈现一个空白的等待页面呢（当然这里说的是非首次冷启动的情况）？这就会涉及到如何缓存UI界面上的数据。

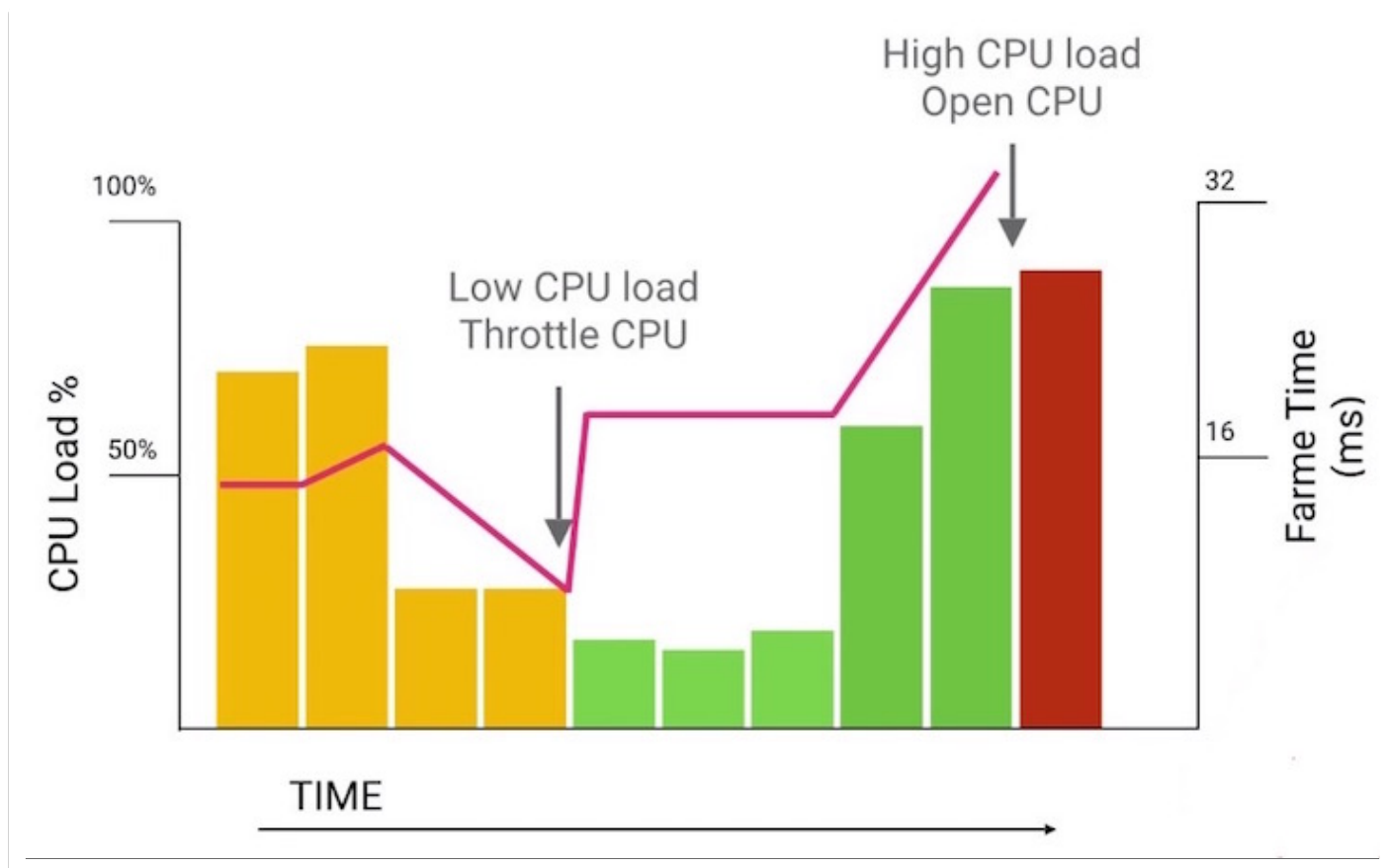
缓存UI界面上的数据，可以采用方案有存储到文件系统，Preference，SQLite等等，做了缓存之后，这样就可以在请求数据返回结果之前，呈现给用户旧的数据，而不是使用正在加载的方式让用户什么数据都看不到，当然在请求网络最新数据的过程中，需要有正在刷新的提示。至于到底选择哪个方案来对数据进行缓存，就需要根据具体情况来做选择了。

## 17)CPU Frequency Scaling

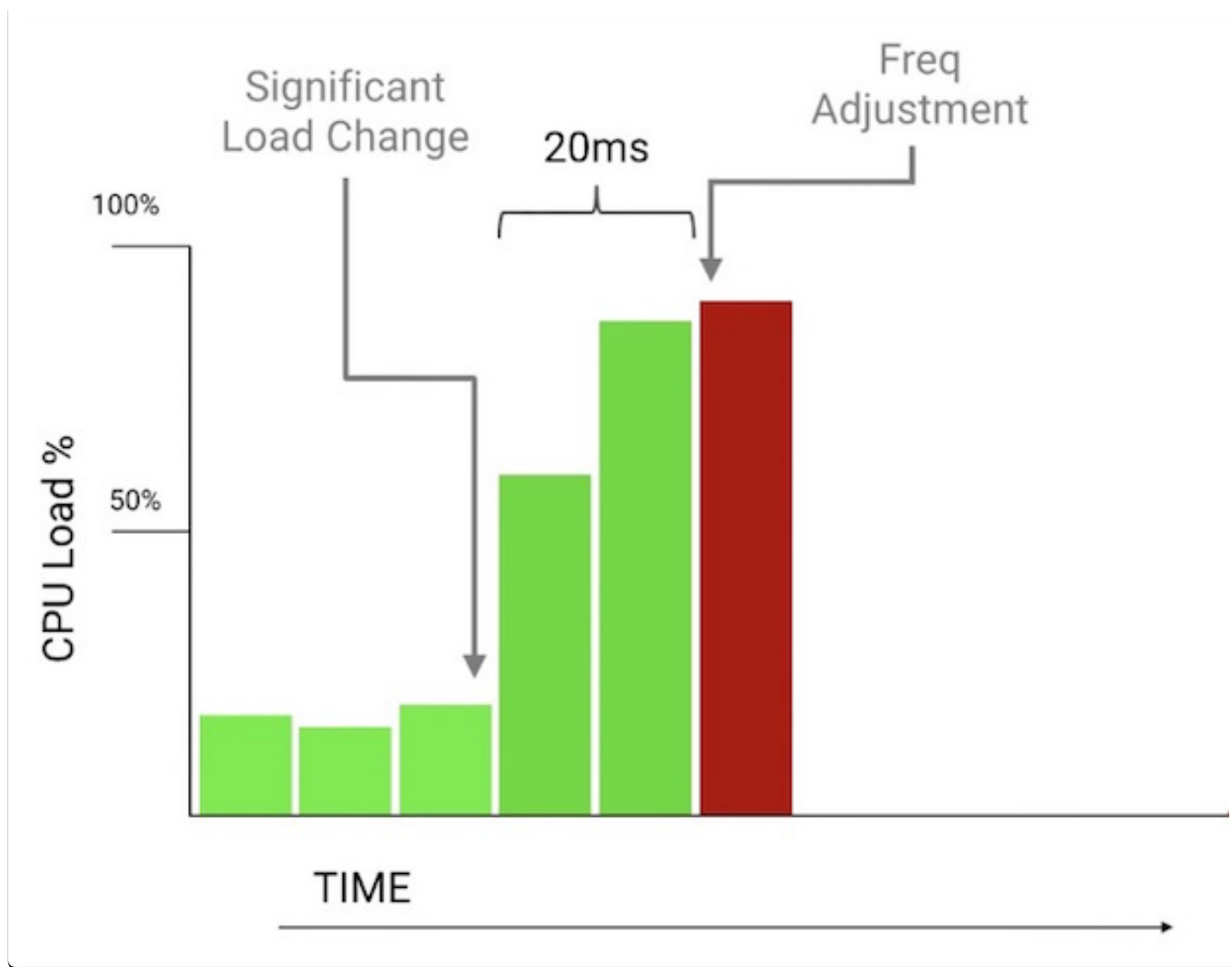
调节CPU的频率会执行的性能产生较大的影响，为了最大化的延长设备的续航时间，系统会动态调整CPU的频率，频率越高执行代码的速度自然就越快。



Android系统会在电量消耗与表现性能之间不断的做权衡，当有需要的时候会迅速调整CPU的频率到一个比较高负荷的状态，当程序不需要高性能的时候就会降低频率来确保更长的续航时间。



Android系统检测到需要调整CPU的频率到CPU频率真的达到对应频率会需要花费大概20ms的时间，在此期间很有可能会因为CPU频率不够而导致代码执行偏慢。



我们可以使用Systrace工具来导出CPU的执行情况，以便帮助定位性能问题。

首发于CSDN: [Android性能优化典范（四）](#)



[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