目录

1. 如何对 Android 应用进行性能分析以及优化?.....3
2. ddms 和 traceView.....9
3. 性能优化如何分析 systrace？.....10
4. 用 IDE 如何分析内存泄漏？.....11
5. Java 多线程引发的性能问题，怎么解决？.....11
6. 启动页白屏及黑屏解决？.....13
7. 启动太慢怎么解决？.....14
8. 怎么保证应用启动不卡顿？.....15
9. App 启动崩溃异常捕捉.....15
10. 现在下载速度很慢,试从网络协议的角度分析原因,并优化(提示：网络的 5 层都可以涉及)。(暂无).....16
11. Https 请求慢的解决办法（提示：DNS，携带数据，直接访问 IP）.....16
12. 如何保持应用的稳定性.....21
13. RecyclerView 和 ListView 的性能对比.....21
14. ListView 的优化.....29
15. RecycleView 优化.....32
16. View 渲染.....33
17. Bitmap 如何处理大图，如一张 30M 的大图，如何预防 OOM.....39
18. java 中的四种引用的区别以及使用场景.....47
19. 强引用置为 null，会不会被回收？.....48

1. 如何对 Android 应用进行性能分析以及优化?

这个作者做了很多片性能优化的文章，建议看完

https://www.jianshu.com/p/da2a4bfcba68

我们常见的 App 优化会涉几个方面

一般来说, 有以下几个方面:

1. App 启动优化
2. 布局优化
3. 响应优化
4. 内存优化
5. 电池使用优化
6. 网络优化

一、APP 启动优化类：

在 APP 冷启动、热启动、温启动过程中不要进行耗时的操作。

二、布局优化类：

1. 尽量减少布局层级和复杂度：

1) 尽量不要嵌套使用 RelativeLayout.

2) 尽量不要在嵌套的 LinearLayout 中都使用 weight 属性.

3) Layout 的选择, 以尽量减少 View 树的层级为主.

4) 去除不必要的父布局.

5) 善用 TextView 的 Drawable 减少布局层级

6) 如果 H Viewer 查看层级超过 5 层, 你就需要考虑优化下布

局了~

2. **善用 Tag**

1) <include>：使用 include 来重用布局.

2) <merge>：使用<merge>来解决 include 或自定义组合 ViewGroup 导致的冗余层级问题. 例如本例中的 RepoItemView 的布局文件实际可以用一个<merge>标签来减少一级.

3) <ViewStub>：显示时才去加载出来

3. **ListView 优化**

1) contentView 复用

2) 引入 holder 来避免重复的 findViewById.

3) 分页加载

4. **使用 Layout Inspector，用于布局优化**

5. **避免过于复杂的布局：**如果我们的 UI 布局层次太深，或是自定义控件的 onDraw 中有复杂运算，CPU 的相关运算就可能大于 16ms，导致卡顿.

6. **避免过度绘制(Overdraw)：**（Overdraw：用来描述一个像素在屏幕上多少次被重绘在一帧上.通俗的说：理想情况下，每屏每帧上，每个像素点应该只被绘制一次，如果有多次绘制，就是 Overdraw，过度绘制了）

所谓 Overdraw，就是在一个像素点上绘制了多次. 常见的就是:

1) 绘制了多重背景.

2) 绘制了不可见的 UI 元素.

可以通过如下方式去掉 window 的背景.

设置主题:

```
<item name="android:windowBackground">@null</item>
```

或是代码设置，在 onCreate 中:

```
getWindow().setBackgroundDrawable(null);
```

UI 线程的复杂运算会造成 UI 无响应，当然更多的是造成 UI 响应停滞，卡顿.

7. StrictMode 的使用

StrictMode 用来基于线程或 VM 设置一些策略，一旦检测到策略违例，控制台将输出一些警告，包含一个 trace 信息展示你的应用在何处出现问题.

通常用来检测主线程中的磁盘读写或网络访问等耗时操作.

三、响应优化类：

ANR 的处理：针对三种不同的情况，一般的处理情况如下

1) **主线程阻塞的：**开辟单独的子线程来处理耗时阻塞事务.

2) **CPU 满负荷，I/O 阻塞的：**I/O 阻塞一般来说就是文件读写或数据库操作执行在主线程了，也可以通过开辟子线程的方式异步执行.

3) **内存不够用的：**增大 VM 内存，使用 largeHeap 属性，排查内存泄露(这个在内存优化那篇细说吧)等.

四、内存优化类：

避免频繁的 GC：执行 GC 操作的时候，任何线程的任何操作都会需要暂停，等待 GC 操作完成之后，其他操作才能够继续运行，故而如果程序频繁 GC，自然会导致界面卡顿.

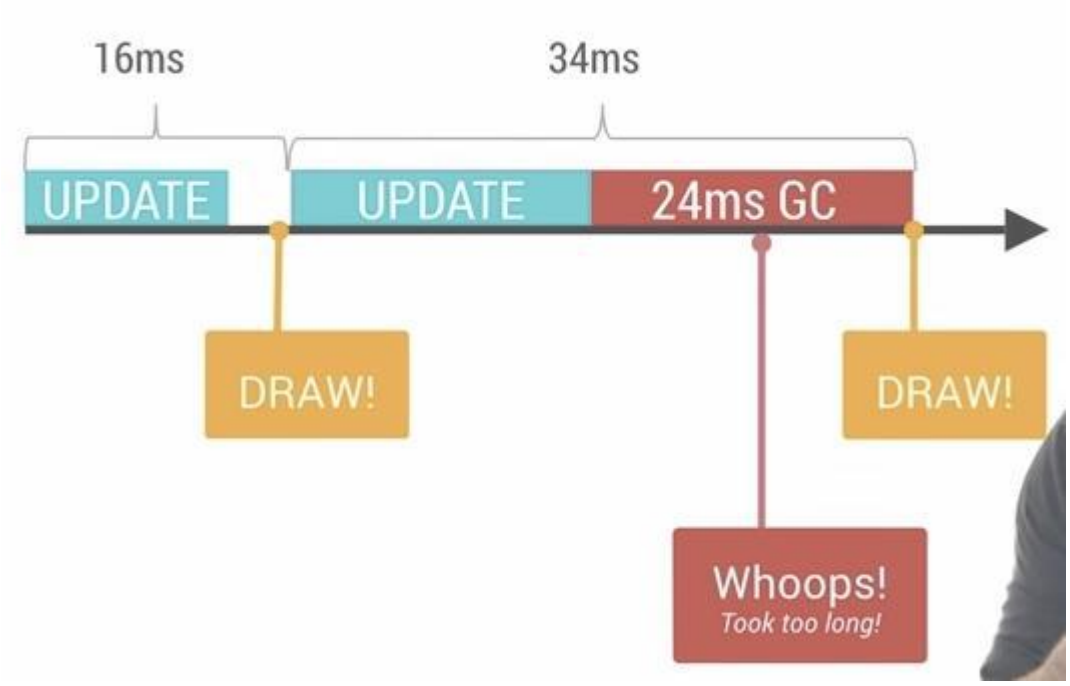
导致频繁 GC 有两个原因:

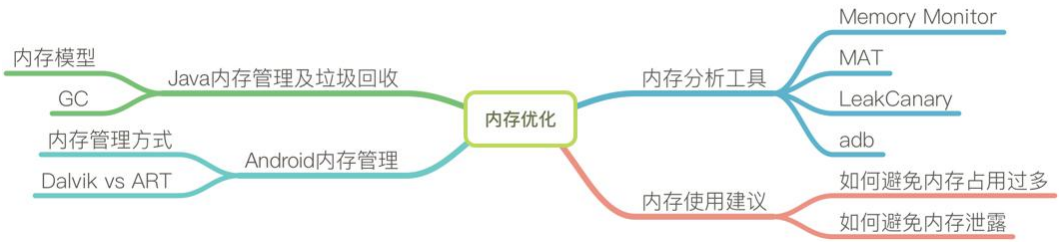
1) 内存抖动(Memory Churn)，即大量的对象被创建又在短时间内马上被释放.

2) 瞬间产生大量的对象会严重占用 Young Generation 的内存区域，当达到阀值，剩余空间不够的时候，也会触发 GC. 即使每次分配的对象需要占用很少的内存，但是他们叠加在一起会增加 Heap 的压力，从而触发更多的 GC.

这些 GC 操作可能会造成上面说到的丢帧，就会让用户感知到卡顿了.

如下:





五、网络优化类：

1. **减少网络数据获取的频次：**这就减少了 radio 的电量消耗，控制电量使用.

2. **减少获取数据包的大小：**可以减少流量消耗，也可以让每次请求更快，在网络情况不好的情况下也有良好表现，提升用户体验.

3. **Gzip 压缩：**使用 Gzip 来压缩 request 和 response，减少传输数据量，从而减少流量消耗.

4. **考虑使用 Protocol Buffer 代替 JSON：**从前我们传输数据使用 XML，后来使用 JSON 代替了 XML，很大程度上也是为了可读性和减少数据量(当然还有映射成 POJO 的方便程度).

Protocol Buffer 是 Google 推出的一种数据交换格式.

如果我们的接口每次传输的数据量很大的话，可以考虑下 protobuf，会比 JSON 数据量小很多.

5. **控制图片的大小：**图片相对于接口请求来说，数据量要大得多. 故而也是我们需要优化的一个点.

我们可以在获取图片时告知服务器需要的图片的宽高，以便服务器给出合适的图片，避免浪费.

6. **网络缓存：**适当的缓存，既可以让我们的应用看起来更快，也能避免一些不必要的流量消耗.

7. **打包网络请求：**当接口设计不能满足我们的业务需求时. 例如可能一个界面需要请求多个接口，或是网络良好，处于 Wifi 状态下时我们想获取更多的数据等.

这时就可以打包一些网络请求，例如请求列表的同时，获取 Header 点击率较高的的 item 项的详情数据.可以通过一些统计数据来帮助我们定位用户接下来的操作是高概率的，提前获取这部分的数据.

8. **监听相关状态并进行相应的优化：**

通过监听设备的状态:

1) 休眠状态

2) 充电状态

3) 网络状态

结合 JobScheduler 来根据实际情况做网络请求. 比方说 Splash 闪屏广告图片, 我们可以在连接到 Wifi 时下载缓存到本地; 新闻类的 App 可以在充电, Wifi 状态下做离线缓存.

9. 弱网下的操作

1) 压缩/减少数据传输量

2) 利用缓存减少网络传输

3) 针对弱网(移动网络), 不自动加载图片

4) 界面先反馈, 请求延迟提交: 例如, 用户点赞操作, 可以直接给出界面的点赞成功的反馈, 使用 JobScheduler 在网络情况较好的时候打包请求.

2. ddms 和 traceView

ddms: 是 android 开发环境中的 dalvik 虚拟机调试监控服务; ddms 能够提供, 测试设备截屏, 针对特定的进程查看正在运行的线程以及堆信息, Logcat, 广播状态信息, 模拟电话呼叫, 接收 sms,

虚拟地理坐标等。

traceView：是 android 平台配备的性能分析的工具；它可以通过图形化让我们了解要跟踪的程序的性能，并且能具体到方法。

区别：ddms 是一个程序执行查看器，在里面可以看见线程和堆栈等信息，traceView 是程序性能分析器。

3. 性能优化如何分析 systrace？

https://www.jianshu.com/p/6f528e862d31

使用 Systrace 分析 UI 性能

翻译原文地址：https://developer.android.com/studio/profile/systrace.html

其实不完善，但没办法

在开发应用时，通常使用 60fps 的帧率来检测交互是否流畅，如果中途出错了，或者发生了掉帧，解决这个问题的第一步应当是搞清楚当前系统在做什么。

systrace 由 Google Android 和 Google Chrome 共同开发，为 Catapul 开源 t 项目的一部分

Systrace 工具可以在程序运行的时候收集实时的信息，记录时间以及 CPU 的分配情况，记录每个线程和进程在任意时间的运行情况，可以自动分析出一些重要的原因，并且给出建议。

是一个分析 android 性能问题的基础工具，但本质上是其他某些

工具的封装。使用 atrace 开启追踪，然后读取 ftrace 的缓存，并且把它重新转换成 HTML 格式。

。

4. 用 IDE 如何分析内存泄漏？

跑一段你觉得有问题的代码段，gc，再跑，再 gc，看看内存会不会一直上升

https://blog.csdn.net/u010944680/article/details/51721532

Android 虚拟机的垃圾回收采用的是根搜索算法。GC 会从根节点（GC Roots）开始对 heap 进行遍历。到最后，部分没有直接或者间接引用到 GC Roots 的就是需要回收的垃圾，会被 GC 回收掉。而内存泄漏出现的原因就是存在了无效的引用，导致本来需要被 GC 的对象没有被回收掉。

Android Monitor 的使用

5. Java 多线程引发的性能问题，怎么解决？

https://zhuanlan.zhihu.com/p/23389156

主要的影响如下：

(1) **消耗时间：**线程的创建和销毁都需要时间，当有大量的线

程创建和销毁时，那么这些时间的消耗则比较明显，将导致性能上的缺失

(2) **非常耗 CPU 和内存：**大量的线程创建、执行和销毁是非常耗 cpu 和内存的，这样将直接影响系统的吞吐量，导致性能急剧下降，如果内存资源占用的比较多，还很可能造成 OOM

(3) **容易导致 GC 频繁的执行：**大量的线程的创建和销毁很容易导致 GC 频繁的执行，从而发生内存抖动现象，而发生了内存抖动，对于移动端来说，最大的影响就是造成界面卡顿

而针对上述所描述的问题，解决的办法归根到底就是：重用已有的线程，从而减少线程的创建。所以这就涉及到线程池（ExecutorService）的概念了，线程池的基本作用就是进行线程的复用，下面将具体介绍线程池的使用

使用线程池管理线程的优点

1) **节省系统的开销：**线程的创建和销毁由线程池维护，一个线程在完成任务后并不会立即销毁，而是由后续的任务复用这个线程，从而减少线程的创建和销毁，节约系统的开销

2) **节省时间：**线程池旨在线程的复用，这就可以节约我们用以往的方式创建线程和销毁所消耗的时间，减少线程频繁调度的开销，从而节约系统资源，提高系统吞吐量

3) **提高性能：**在执行大量异步任务时提高了性能

4) **方便控制：**Java 内置的一套 ExecutorService 线程池相关的 api，可以更方便的控制线程的最大并发数、线程的定时任务、单

线程的顺序执行等

优先级线程池的优点

从上面我们可以得知，创建一个优先级线程池非常有用，它可以在线程池中线程数量不足或系统资源紧张时，优先处理我们想要先处理的任务，而优先级低的则放到后面再处理，这极大改善了系统默认线程池以 FIFO 方式处理任务的不灵活

6. 启动页白屏及黑屏解决？

https://blog.csdn.net/zivensonice/article/details/51691136

1. 把启动图 bg_splash 设置为窗体背景，避免刚刚启动 App 的时候出现，黑/白屏

```
<item name="android:windowBackground">@drawable/bg_splash</item>
```

2. 设置为背景 bg_splash 显示的时候，后台负责加载资源，同时去下载广告图，广告图下载成功或者超时的时候显示 SplashActivity 的真实样子

3. 随后进入 MainAcitivity

据我观察，淘宝启动的时候和斗鱼逻辑是一样的，有兴趣可以探究下。

```
<style name="ThemeSplash" parent="Theme.AppCompat.Light.NoActionBar">
```

```
        <item name="android:background">@mipmap/bg_splash</item>
        <item name="android:windowNoTitle">true</item>
        <item name="android:windowFullscreen">true</item>
        <item name="windowActionBar">false</item>
        <item name="windowNoTitle">true</item>
    </style>
```

原因分析：在启动 Acitivty 的 onCreate()方法里面，执行 setContentView(R.layout.activity_splash);出现白屏。

onCreate---setContentView这个并不是发生在窗体绘制的第一步，系统会在执行这个步骤之前，先绘制窗体，这时候布局资源还没加载，于是就使用默认背景色。

这种亮色系，造成白色闪屏

```
<style name="ThemeSplash" parent="Theme.AppCompat.Light">
```

这种暗色系主题，造成了黑色闪屏

```
<style name="ThemeSplash" parent="ThemeOverlay.AppCompat.Dark">
```

7. 启动太慢怎么解决？

应用启动速度，取决于你在 application 里面时候做了什么事情，比如你集成了很多 sdk，并且 sdk 的 init 操作都需要在主线程里实现，那自然就慢咯。在非必要的情况下可以把加载延后。或者丢

子线程里。(第一个问题里有答案)

https://www.jianshu.com/p/4f10c9a10ac9

8. 怎么保证应用启动不卡顿?

同上面一个道理，也可以做个闪屏页当缓冲时间。(第一个问题里有答案)

https://www.jianshu.com/p/4f10c9a10ac9

9. App 启动崩溃异常捕捉

https://www.jianshu.com/p/fb28a5322d8a

自定义一个应用异常捕获类 AppUncaughtExceptionHandler，它必须得实现 Thread.UncaughtExceptionHandler 接口，另外还需要重写 uncaughtException 方法，去按我们自己的方式来处理异常。

在 Application 中我们只需要初始化自定义的异常捕获类即可：

完成以上过程后，接着需要重写 uncaughtException 方法：

```
@Overridepublic void uncaughtException(Thread thread, Throwable ex) {
    if (crashing) {
        return;
    }
    crashing = true;
```

```
    // 打印异常信息
    ex.printStackTrace();
    // 我们没有处理异常 并且默认异常处理不为空 则交给系统处理
    if (!handlelException(ex) && mDefaultHandler != null) {
        // 系统处理
        mDefaultHandler.uncaughtException(thread, ex);
    }
    byebye();
}
private void byebye() {
    android.os.Process.killProcess(android.os.Process.myPid());
    System.exit(0);
}
    System.exit(0);
}
```

既然是我们自己处理异常,所以会先执行handlelException(ex)方法:

10. 现在下载速度很慢,试从网络协议的角度分析原因,并优化(提示：网络的 5 层都可以涉及)。(暂无)

11. Https 请求慢的解决办法（提示：

DNS，携带数据，直接访问 IP）

暂时没能力解决

https://www.cnblogs.com/mylanguage/p/5635524.html

前言

HTTPS 在保护用户隐私，防止流量劫持方面发挥着非常关键的作用，但与此同时，HTTPS 也会降低用户访问速度，增加网站服务器的计算资源消耗。

本文主要介绍 https 对用户体验的影响。

HTTP 与 HTTPS 的概念和区别

（1）HTTPS（全称：Hypertext Transfer Protocol over Secure Socket Layer），是以安全为目标的 HTTP 通道，简单讲是 HTTP 的安全版。即 HTTP 下加入 SSL 层，HTTPS 的安全基础是 SSL，因此加密的详细内容就需要 SSL。 它是一个 URI scheme（抽象标识符体系），句法类同 http:体系。用于安全的 HTTP 数据传输。https:URL 表明它使用了 HTTP，但 HTTPS 存在不同于 HTTP 的默认端口及一个加密/身份验证层（在 HTTP 与 TCP 之间）。这个系统的最初研发由网景公司进行，提供了身份验证与加密通讯方法，现在它被广泛用于万维网上安全敏感的通讯，例如交易支付方面。

（2）超文本传输协议（HTTP-Hypertext transfer protocol）是一种详细规定了浏览器和万维网服务器之间互相通信的规则，通过因特网传送万维网文档的数据传送协议。

（3）https 协议需要到 ca 申请证书，一般免费证书很少，需要交费。

http 是超文本传输协议，信息是明文传输，https 则是具有安全性的 ssl 加密传输协议

http 和 https 使用的是完全不同的连接方式，用的端口也不一样,前者是 80,后者是 443。

http 的连接很简单,是无状态的，HTTPS 协议是由 SSL+HTTP 协议构建的可进行加密传输、身份认证的网络协议 ，要比 http 协议安全

3 HTTPS 对访问速度的影响

在介绍速度优化策略之前，先来看下 HTTPS 对速度有什么影响。影响主要来自两方面：

1. 协议交互所增加的网络 RTT(round trip time)。
2. 加解密相关的计算耗时。

网络耗时增加

由于 HTTP 和 HTTPS 都需要 DNS 解析，并且大部分情况下使用了 DNS 缓存，为了突出对比效果，忽略主域名的 DNS 解析时间。

用户使用 HTTP 协议访问 http://www.baidu.com(或者 www.baidu.com) 时会有如下网络上的交互耗时：

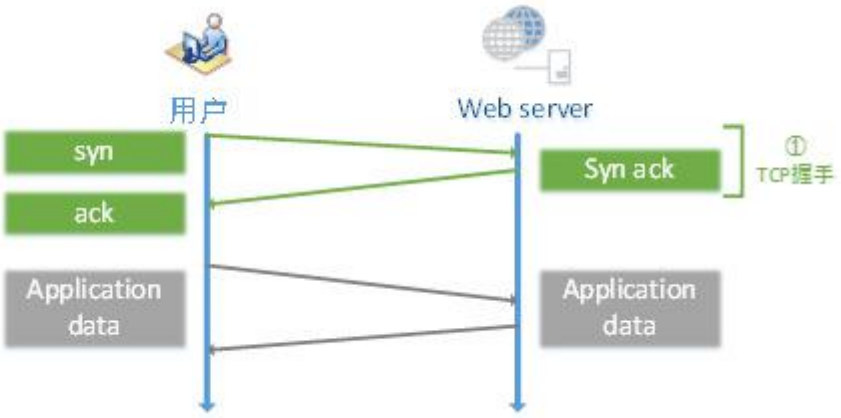


图 1 HTTP 首个请求的网络耗时

可见，用户只需要完成 TCP 三次握手建立 TCP 连接就能够直接发送 HTTP 请求获取应用层数据，此外在整个访问过程中也没有需要消耗计算资源的地方。

接下来看 HTTPS 的访问过程，相比 HTTP 要复杂很多，在部分场景下，使用 HTTPS 访问有可能增加 7 个 RTT。如下图：

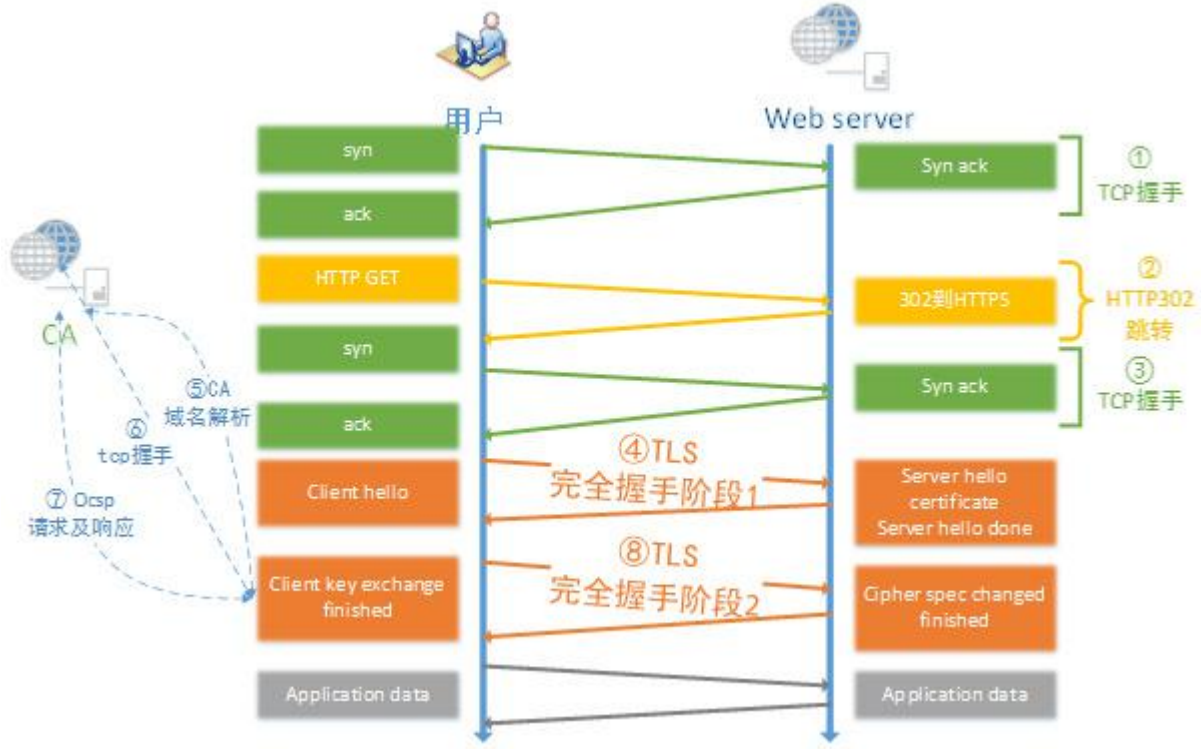


图 2 HTTPS 首次请求对访问速度的影响

HTTPS 首次请求需要的网络耗时解释如下：

1， 三次握手建立 TCP 连接。耗时一个 RTT。

2， 使用 HTTP 发起 GET 请求，服务端返回 302 跳转到 https://www.baidu.com。需要一个 RTT 以及 302 跳转延时。

a) 大部分情况下用户不会手动输入 https://www.baidu.com 来访问 HTTPS，服务端只能返回 302 强制浏览器跳转到 https。

b) 浏览器处理 302 跳转也需要耗时。

3， 三次握手重新建立 TCP 连接。耗时一个 RTT。

a) 302 跳转到 HTTPS 服务器之后，由于端口和服务器不同，需要重新完成三次握手，建立 TCP 连接。

4， TLS 完全握手阶段一。耗时至少一个 RTT。

a) 这个阶段主要是完成加密套件的协商和证书的身份认证。

b) 服务端和浏览器会协商出相同的密钥交换算法、对称加密算法、内容一致性校验算法、证书签名算法、椭圆曲线（非 ECC 算法不需要）等。

c) 浏览器获取到证书后需要校验证书的有效性，比如是否过期，是否撤销。

5， 解析 CA 站点的 DNS。耗时一个 RTT。

a) 浏览器获取到证书后，有可能需要发起 OCSP 或者 CRL 请求，查询证书状态。

b) 浏览器首先获取证书里的 CA 域名。

c) 如果没有命中缓存，浏览器需要解析 CA 域名的 DNS。

6， 三次握手建立 CA 站点的 TCP 连接。耗时一个 RTT。

a) DNS 解析到 IP 后，需要完成三次握手建立 TCP 连接。

7， 发起 OCSP 请求，获取响应。耗时一个 RTT。

8， 完全握手阶段二，耗时一个 RTT 及计算时间。

a) 完全握手阶段二主要是密钥协商。

9， 完全握手结束后，浏览器和服务器之间进行应用层（也就是 HTTP）数据传输。

当然不是每个请求都需要增加 7 个 RTT 才能完成 HTTPS 首次请求交互。大概只有不到 0.01% 的请求才有可能需要经历上述步骤，它们需要满足如下条件：

1， 必须是首次请求。即建立 TCP 连接后发起的第一个请求，该连接上的后续请求都不需要再发生上述行为。

2， 必须要发生完全握手，而正常情况下 80% 的请求能实现简化握手。

3， 浏览器需要开启 OCSP 或者 CRL 功能。Chrome 默认关闭了 ocsp 功能，firefox 和 IE 都默认开启。

4， 浏览器没有命中 OCSP 缓存。Ocsp 一般的更新周期是 7 天，firefox 的查询周期也是 7 天，也就说是 7 天中才会发生一次 ocsp 的查询。

5， 浏览器没有命中 CA 站点的 DNS 缓存。只有没命中 DNS 缓存的情况下才会解析 CA 的 DNS。

计算耗时增加

上节还只是简单描述了 HTTPS 关键路径上必须消耗的纯网络耗时，没有包括非常消耗 CPU 资源的计算耗时，事实上计算耗时也不小（30ms 以上），从浏览器和服务器的角度分别介绍一下：

1， 浏览器计算耗时

a) RSA 证书签名校验，浏览器需要解密签名，计算证书哈希值。如果有多个证书链，浏览器需要校验多个证书。

b) RSA 密钥交换时，需要使用证书公钥加密 premaster。耗时比较小，但如果手机性能比较差，可能也需要 1ms 的时间。

c) ECC 密钥交换时，需要计算椭圆曲线的公私钥。

d) ECC 密钥交换时，需要使用证书公钥解密获取服务端发过来的 ECC 公钥。

e) ECC 密钥交换时，需要根据服务端公钥计算 master key。

f) 应用层数据对称加解密。

g) 应用层数据一致性校验。

2， 服务端计算耗时

a) RSA 密钥交换时需要使用证书私钥解密 premaster。这个过程非常消耗性能。

b) ECC 密钥交换时，需要计算椭圆曲线的公私钥。

c) ECC 密钥交换时，需要使用证书私钥加密 ECC 的公钥。

d) ECC 密钥交换时，需要根据浏览器公钥计算共享的 master key。

e) 应用层数据对称加解密。

f) 应用层数据一致性校验。

由于客户端的 CPU 和操作系统种类比较多，所以计算耗时不能一概而论。手机端的 HTTPS 计算会比较消耗性能，单纯计算增加的延迟至少在 50ms 以上。PC 端也会增加至少 10ms 以上的计算延迟。

服务器的性能一般比较强，但由于 RSA 证书私钥长度远大于客户端，所以服务端的计算延迟也会在 5ms 以上。

12. 如何保持应用的稳定性

内存，布局优化，代码质量，数据结构效率，针对业务合理的设计框架

13. RecyclerView和ListView的性能对比

https://blog.csdn.net/fanenqian/article/details/61191532

https://zhuanlan.zhihu.com/p/23339185

一、首先缓存机制对比

ListView(两级缓存)：

ListView				
	是否需要回调createView	是否需要回调bindView	生命周期	备注
mActiveViews	否	否	onLayout函数周期内	用于屏幕内ItemView快速重用
mScrapViews	否	是	与mAdapter一致，当mAdapter被更换时，mScrapViews即被清空	

RecyclerView(四级缓存)：

RecyclerView				
	是否需要回调createView	是否需要回调bindView	生命周期	备注
mAttachedScrap	否	否	onLayout函数周期内	用于屏幕内ItemView快速重用
mCacheViews	否	否	与mAdapter一致，当mAdapter被更换时，mCacheViews即被缓存至mRecyclerPool	默认上限为2，即缓存屏幕外2个ItemView
mViewCacheExtension				不直接使用，需要用户在定制，默认不实现
mRecyclerPool	否	是	与自身生命周期一致，不再被引用时即被释放	默认上限为5，技术上可以实现所有RecyclerViewPool共用同一个

ListView 和 RecyclerView 缓存机制基本一致：

1). mActiveViews 和 mAttachedScrap 功能相似，意义在于快速重用屏幕上可见的列表项 ItemView，而不需要重新 createView 和 bindView；

2). mScrapView 和 mCachedViews + mReyclerViewPool 功能相似，意义在于缓存离开屏幕的 ItemView，目的是让即将进入屏幕的 ItemView 重用.

3). RecyclerView 的优势在于 a.mCacheViews 的使用，可以做到屏幕外的列表项 ItemView 进入屏幕内时也无须 bindView 快速重用；b.mRecyclerPool 可以供多个 RecyclerView 共同使用，在特定场景下,如 viewpaper+多个列表页下有优势.客观来说,RecyclerView 在特定场景下对 ListView 的缓存机制做了补强和完善。

缓存不同：

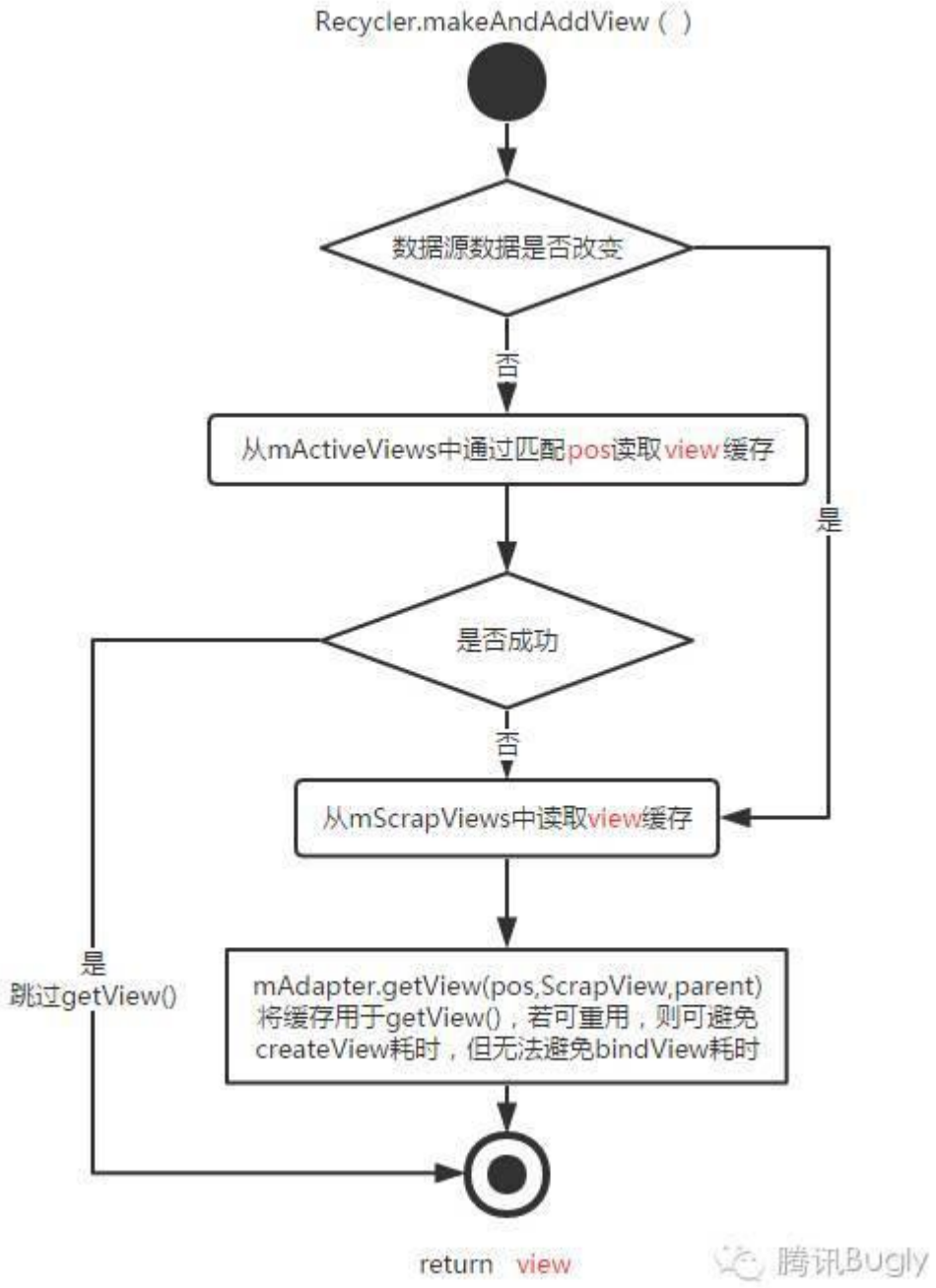
1) RecyclerView 缓存 RecyclerView.ViewHolder，抽象可理解为：

View + ViewHolder(避免每次 createView 时调用 findViewById) + flag(标识状态)；

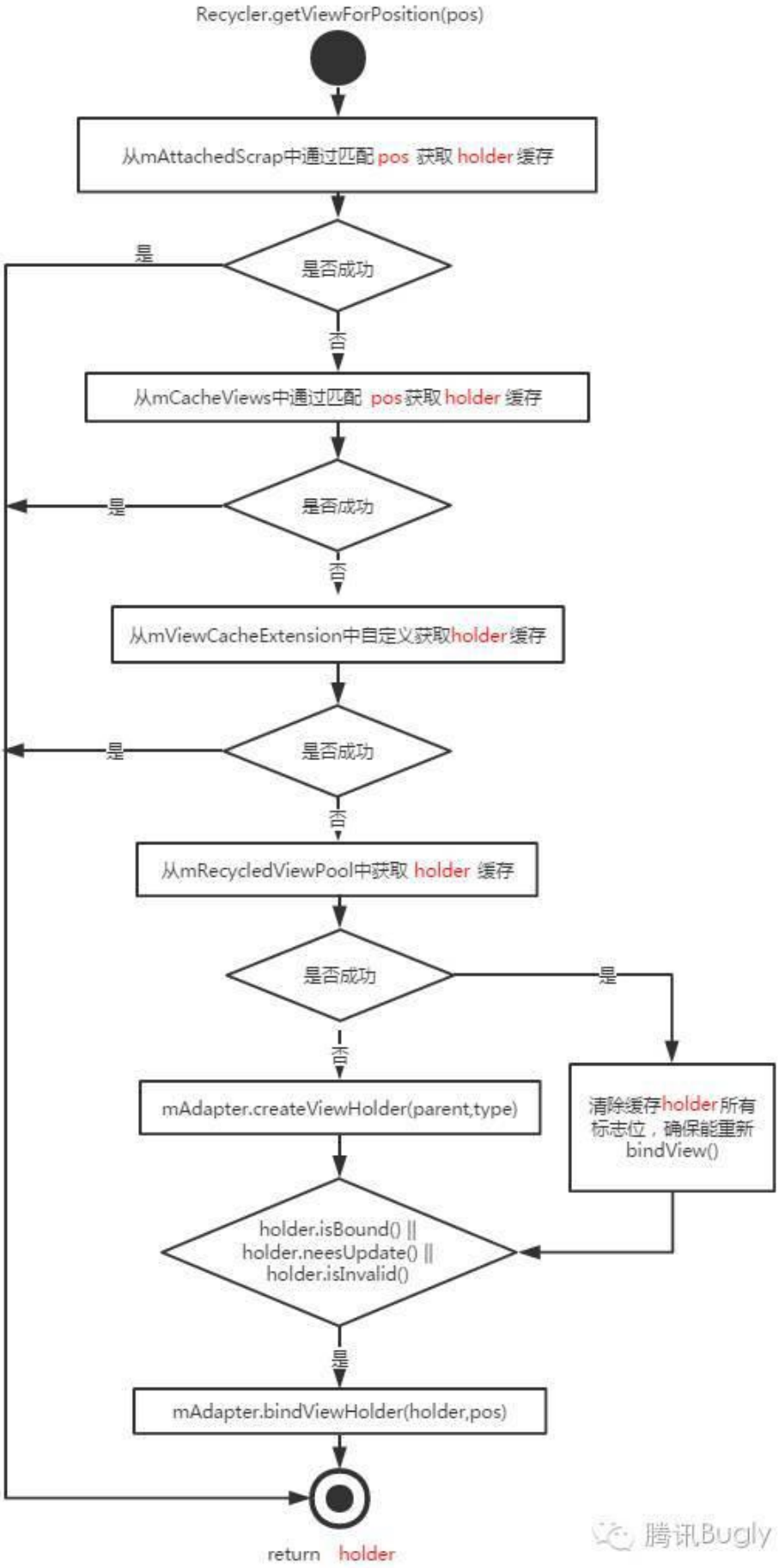
2) ListView 缓存 View。

缓存不同，二者在缓存的使用上也略有差别，具体来说：

ListView 获取缓存的流程：



RecyclerView 获取缓存的流程：


Recycler.getViewForPosition(pos)
从mAttachedScrap中通过匹配 pos 获取 holder 缓存
是否成功
是
否
从mCacheViews中通过匹配 pos 获取 holder 缓存
是否成功
是
否
从mViewCacheExtension中自定义获取holder 缓存
是否成功
是
否
从mRecycledViewPool中获取 holder 缓存
是否成功
是
否
mAdapter.createViewHolder(parent,type)
清除缓存holder 所有标志位，确保能重新bindView()
holder.isBound() ||
holder.neesUpdate() ||
holder.isInvalid()
是
mAdapter.bindViewHolder(holder,pos)
return holder

1)　RecyclerView 中 mCacheViews(屏幕外)获取缓存时，是通过匹配 pos 获取目标位置的缓存，这样做的好处是，当数据源数据不变的情况下，无须重新 bindView；

2)　ListView 中通过 pos 获取的是 view，即 pos→view；RecyclerView 中通过 pos 获取的是 viewholder，即 pos → (view，viewHolder，flag)；

从流程图中可以看出，标志 flag 的作用是判断 view 是否需要重新 bindView，这也是 RecyclerView 实现局部刷新的一个核心.

二、局部刷新

RecyclerView 的缓存机制确实更加完善，但还不算质的变化，RecyclerView 更大的亮点在于提供了局部刷新的接口，通过局部刷新，就能避免调用许多无用的 bindView.

结合 RecyclerView 的缓存机制，看看局部刷新是如何实现的：以 RecyclerView 中 notifyItemRemoved(1)为例，最终会调用 requestLayout()，使整个 RecyclerView 重新绘制，过程为：onMeasure()→onLayout()→onDraw()

其中，onLayout()为重点，分为三步：

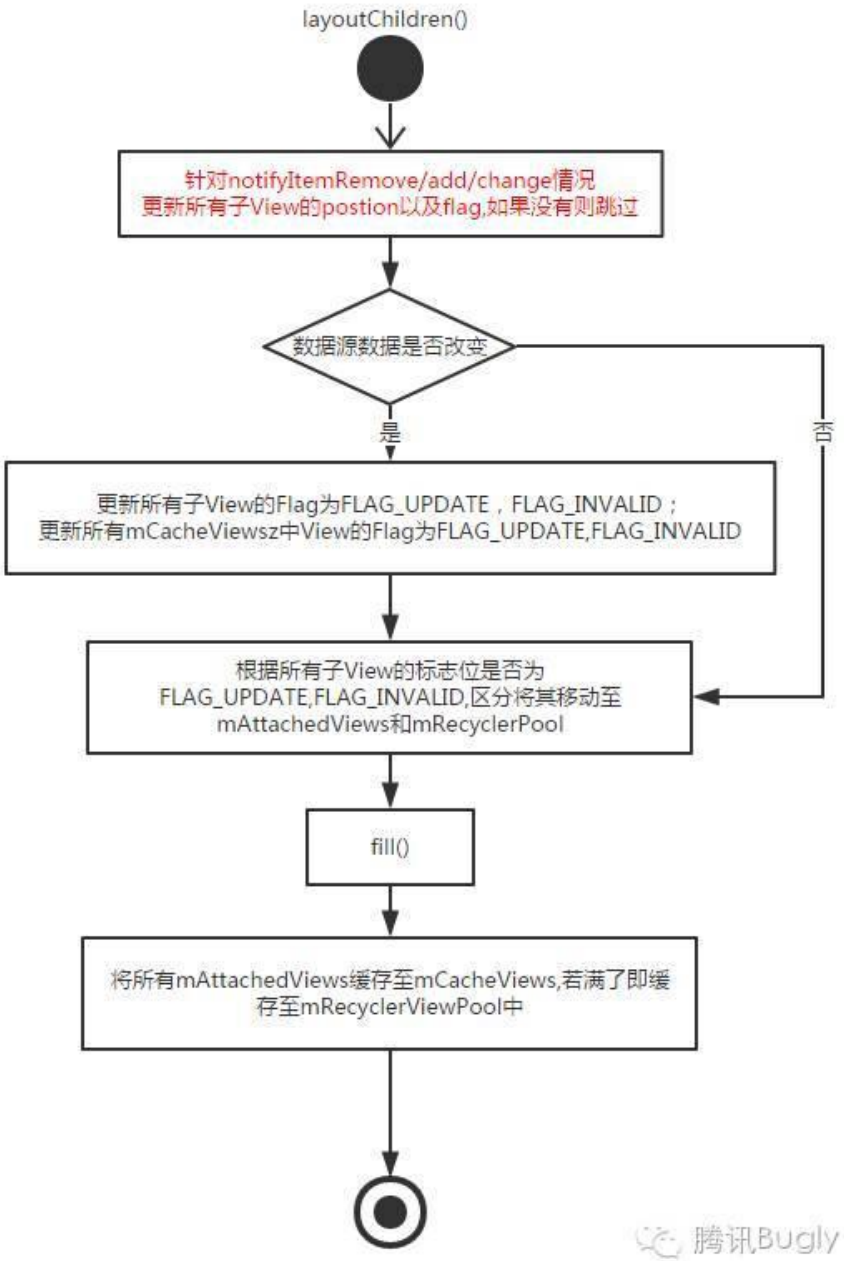
1)　dispathLayoutStep1()：记录 RecyclerView 刷新前列表项 ItemView 的各种信息，如 Top,Left,Bottom,Right，用于动画的相关计算；

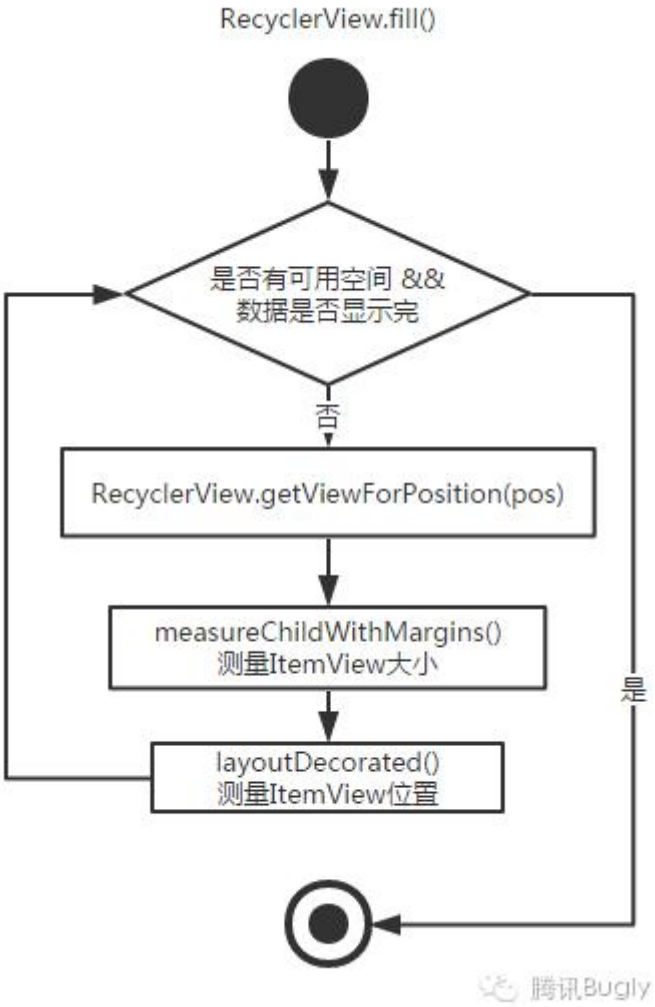
2)　dispathLayoutStep2()：真正测量布局大小，位置，核心

函数为 layoutChildren()；

3)　dispathLayoutStep3()：计算布局前后各个 ItemView 的状态，如 Remove，Add，Move，Update 等，如有必要执行相应的动画.

其中，layoutChildren()流程图：





当调用 fill()中 RecyclerView.getViewForPosition(pos)时，RecyclerView 通过对 pos 和 flag 的预处理，使得 bindview 只调用一次.

需要指出，ListView 和 RecyclerView 最大的区别在于数据源改变时的缓存的处理逻辑，ListView 是”一锅端”，将所有的 mActiveViews 都移入了二级缓存 mScrapViews，而 RecyclerView 则是更加灵活地对每个 View 修改标志位，区分是否重新 bindView。

三、结论

1） 在一些场景下，如界面初始化，滑动等，ListView 和 RecyclerView 都能很好地工作，两者并没有很大的差异：

2） 数据源频繁更新的场景，如弹幕：Recyclerview 实现的弹幕 （旧）等 RecyclerView 的优势会非常明显；

进一步来讲，结论是：

列表页展示界面，需要支持动画，或者频繁更新，局部刷新，建议使用 RecyclerView，更加强大完善，易扩展；其它情况(如微信卡包列表页)两者都 OK，但 ListView 在使用上会更加方便，快捷。

14. ListView 的优化

可以说上分页加载哦

https://www.cnblogs.com/yuhanghzsd/p/5595532.html

Adapter:

它在 ListView 和数据源之间起到桥梁的作用，避免 listview 和数据源直接接触，而导致因为数据源的复杂性使 listview 显得臃肿。

Adapter,适配器，把复杂的数据源适配给 listview,很容易联想

到适配器模式。

下面是 listview 的优化：

第一个是优化加载布局，第二个是优化加载控件。还有分页加载

1) **加载布局优化：**convertView 的使用，主要优化加载布局问题。listivew 每次滚动都会调用 getview()方法,所以优化 getview 是重中之重。如果没有缓存就加载布局，如果有缓存就直接用 convertView 对象。所以这样就不用滑动 listview 的时候。调用 getView()方法每次都去加载布局了（如果改布局已经加载）。

2) **加载控件优化：**内部类 ViewHolder 的使用。主要优化 getView 方法中每次回调用 findviewByID()方法来获取一次控件的代码。新增加内部类 ViewHolder,用于对控件的实力存储进行缓存。

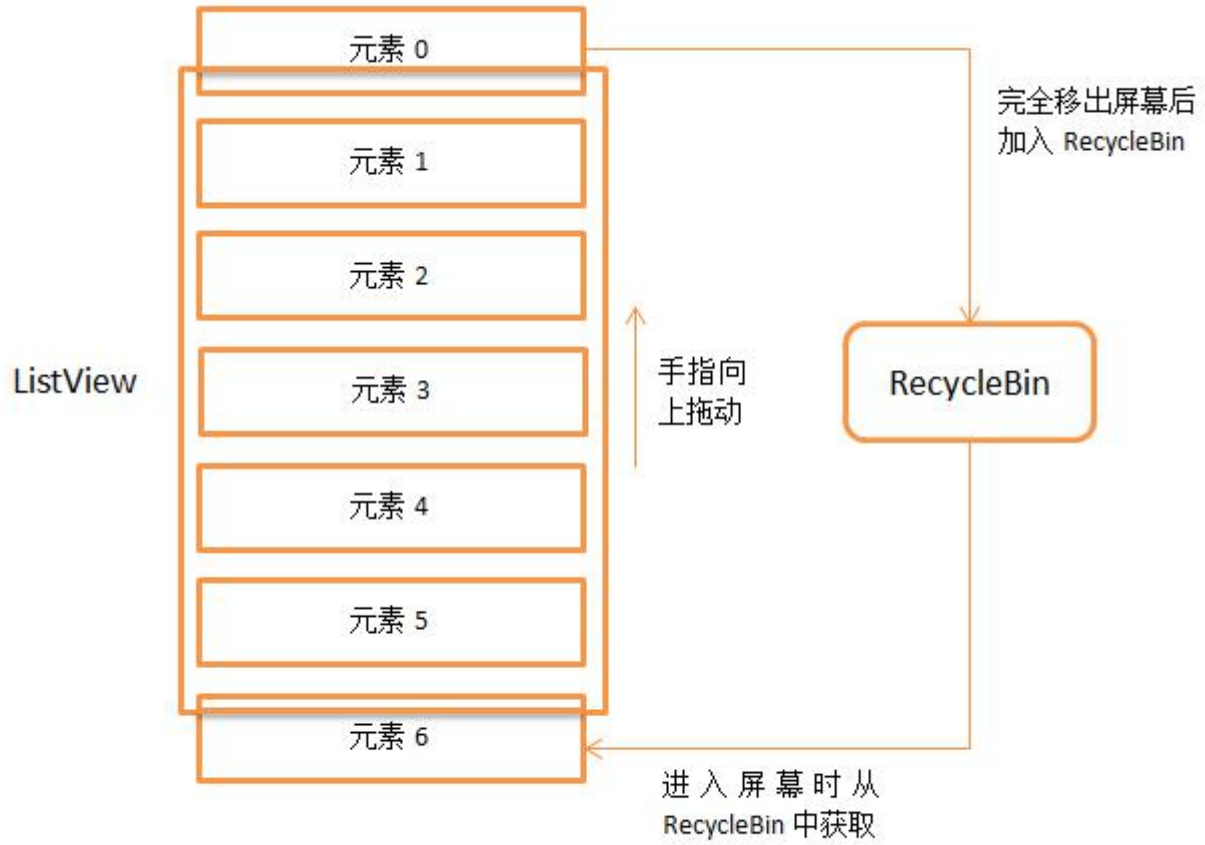
① convertView 为空时，viewHolder 会将空间的实力存放在 ViewHolder 里，然后用 setTag 方法讲 viewHolder 对象存储在 view 里。

② convertView 不为空时，用 getTag 方法获取 viewHolder 对象.

有没有想过 ListView 加载成千上万的数据为什么不出 OOM 错误？

最主要的是因为 RecycleBin 机制。

1) listview 的许多 view 呈现在 Ui 上，这样的 View 对我们来说是可见的，可以称为 OnScreen 的 view(也为 ActionView)。
2) view 被上滚移除屏幕，这样的 view 称为 offScreenView（也称为 ScrapView）。
3) 然后 ScrapView 会被 listview 删除，而 RecycleView 会将这部分保存。
4) 而 listview 底部需要显示的 view 会从 RecycleBin 里面取出一个 ScrapView。
5) 将其作为 convertView 参数传递过去，从而达到 View 复用的目的，这样就不必在 Adapter 的 getView 方法中执行 LayoutInflater.inflate()方法了（不用加载布局了有木有）。



15. RecycleView 优化

https://blog.csdn.net/axi295309066/article/details/52741810

https://www.jianshu.com/p/411ab861034f

(1)使用 RecycledViewPool （RecyclerView 设置一个 ViewHolder 的对象池，这个池称为 RecycledViewPool，这个对象池可以节省你创建 ViewHolder 的开销，更能避免 GC。即便你不给它设置，它也会自己创建一个。）：

RecycledViewPool 使用起来也是非常的简单：先从某个 RecyclerView 对象中获得它创建的 RecycledViewPool 对象，或者是

自己实现一个 RecycledViewPool 对象，然后设置个接下来创建的每一个 RecyclerView 即可。

(2)使用 SortedList （是一个有序列表，数据变动会触发回调 SortedList.Callback 的方法，如 onChanged()）：

构造一个 SortedList 需要实现它的回调 SortedList.Callback，并由其来定义数据的排序和数据的唯一性。它有一个实现类 SortedListAdapterCallback 就是 RecyclerView.Adapter 与 SortedList 交互的秘密武器。调用者却再也不用关心数据的去重与通知更新的问题，比较适合于批量更新(Batched Updates)。

(3)精简代码：

找到重复部分代码，抽取到基类，非重复部分用抽象方法代替，具体让子类实现。利用 SparseArray 来做缓存，把常用方法全部写好，从而避免冗余代码。

(4)扩展功能：

大家都知道 RecyclerView 没有 ItemClick 方法，可以在上面提过的 BaseQuickAdapter 里面添加 ItemClick，可以这样：网上有很多写法都是在 onBindViewHolder 里面写，功能是可以实现但是会导致频繁创建，应该在 onCreateViewHolder()中每次为新建的 View 设置一次就行了。

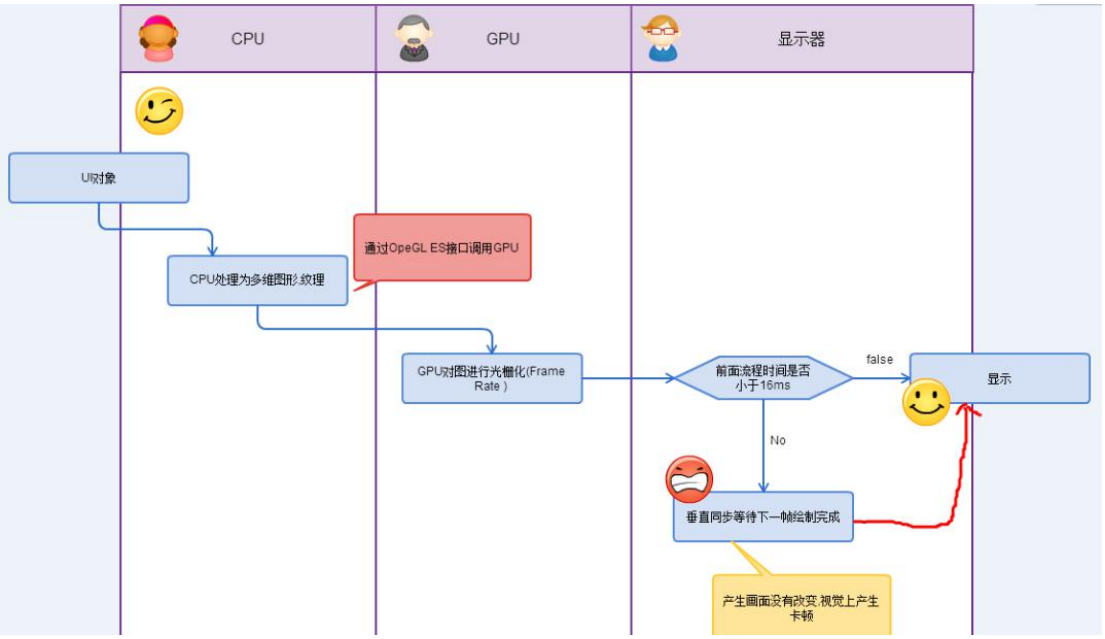
16. View 渲染

https://www.cnblogs.com/ldq2016/p/6668148.html

渲染机制分析

渲染流程线

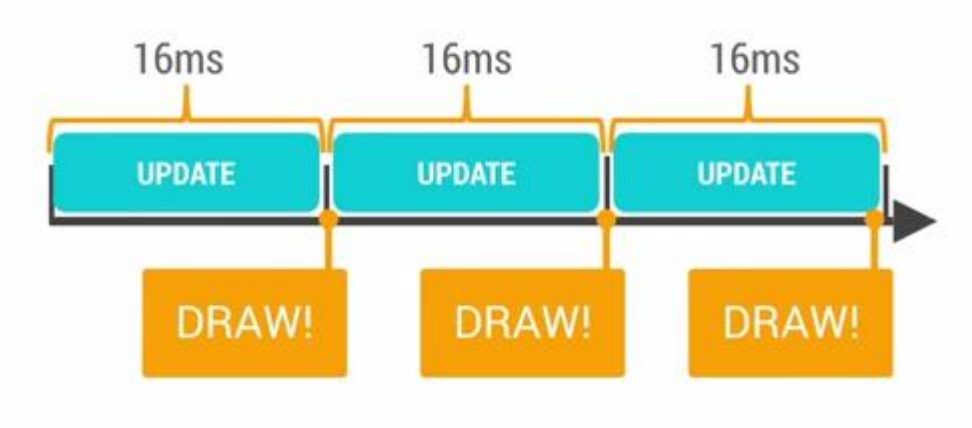
UI 对象—->CPU 处理为多维图形,纹理 —-通过 OpeGL ES 接口调用 GPU—-> GPU 对图进行光栅化(Frame Rate) —->硬件时钟(Refresh Rate)—-垂直同步—->投射到屏幕



渲染时间线

Android 系统每隔 16ms 发出 VSYNC 信号(1000ms/60=16.66ms),触发对 UI 进行渲染， 如果每次渲染都成功，这样就能够达到流畅的画面所需要的 60fps，为了能够实现 60fps，这意味着计算渲染的大多数操作都必须在 16ms 内完成。

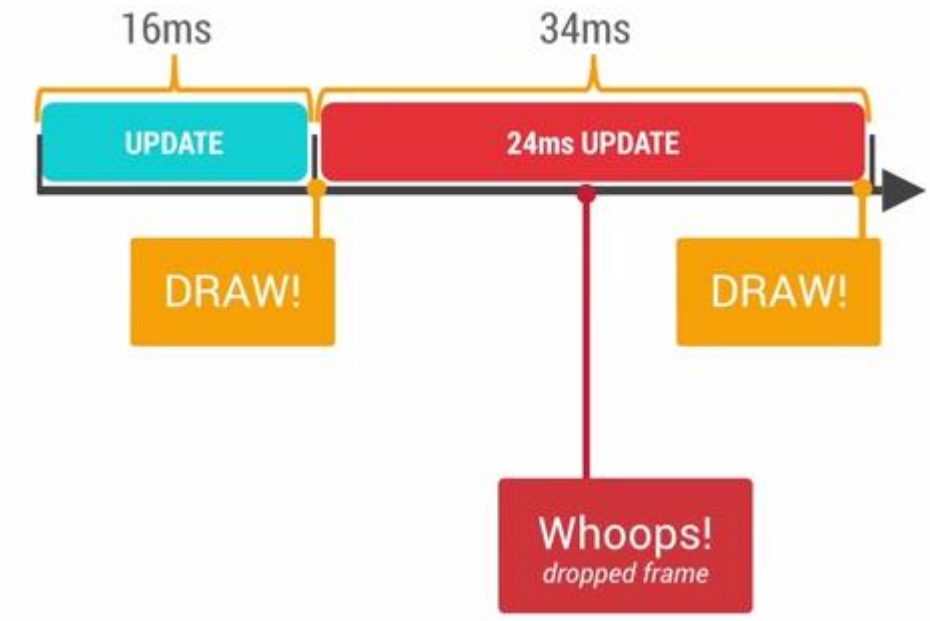
1） 正常情况



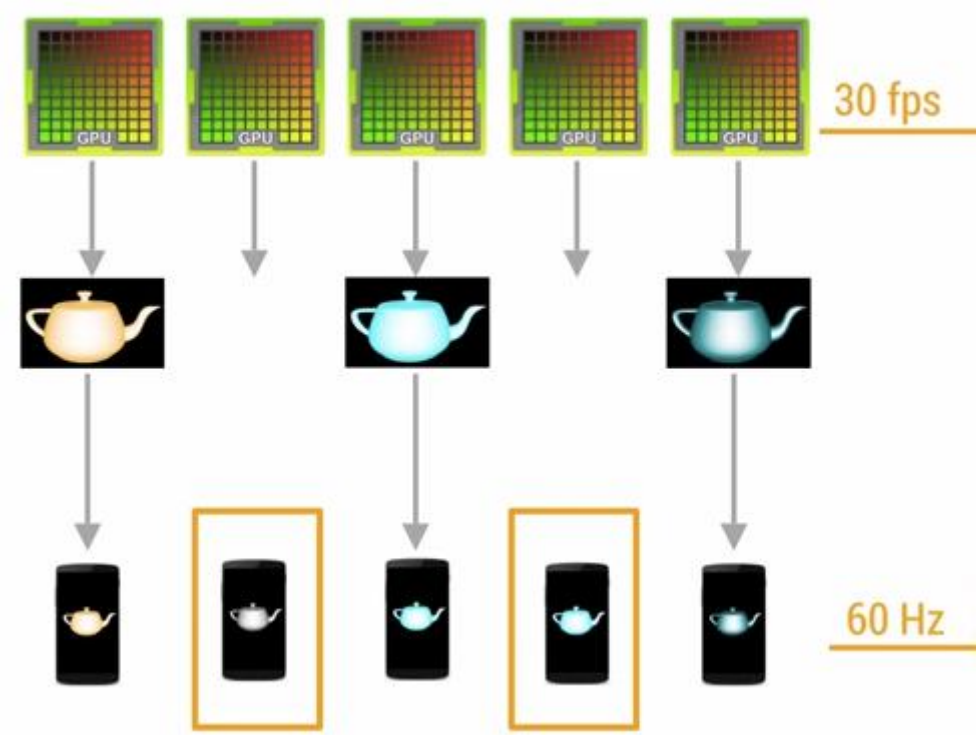
2）渲染超时,计算渲染时间超过 16ms

当这一帧画面渲染时间超过 16ms 的时候,垂直同步机制会让显示器硬件　等待 GPU 完成栅格化渲染操作,

这样会让这一帧画面,多停留了 16ms,甚至更多.这样就这造成了 用户看起来 画面停顿.



当 GPU 渲染速度过慢,就会导致如下情况,某些帧显示的画面内容就会与上一帧的画面相同



渲染时会出现的问题

(1)GPU 过度绘制

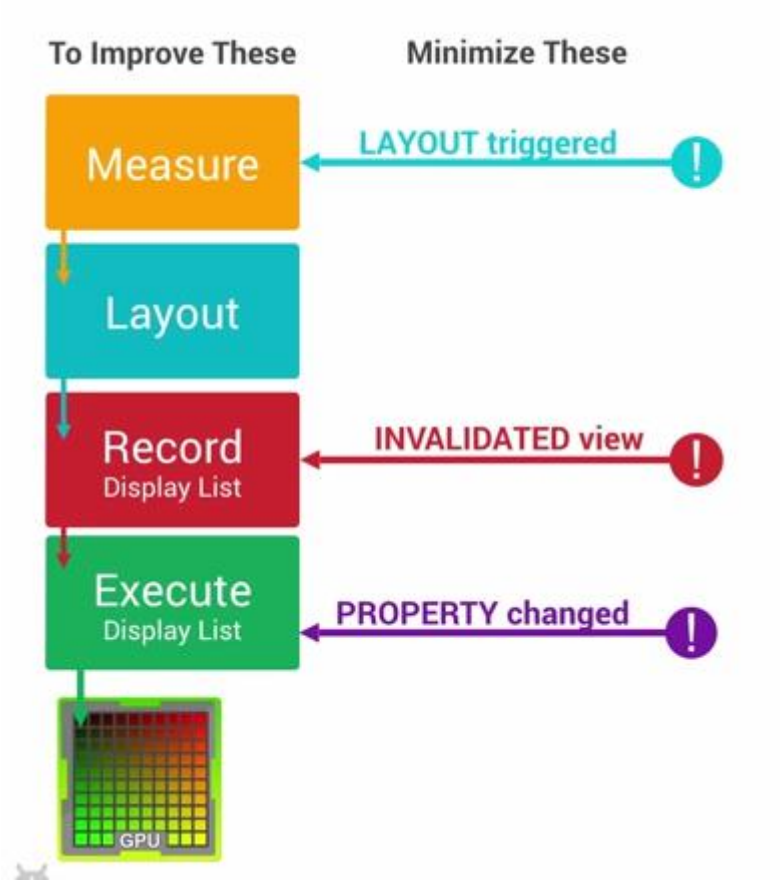
GPU 的绘制过程,就跟刷墙一样,一层层的进行,16ms 刷一次.这样就会造成,图层覆盖的现象,即无用的图层还被绘制在底层,造成不必要的浪费.

(2)计算渲染的耗时

任何时候 View 中的绘制内容发生变化时，都会重新执行创建 DisplayList，渲染 DisplayList，更新到屏幕上等一 系列操作。这个流程的表现性能取决于你的 View 的复杂程度，View 的状态变化以及渲染管道的执行性能。

举个例子,当 View 的大小发生改变,DisplayList 就会重新创建,然后再渲染,而当 View 发生位移,则 DisplayList 不会重新创建,而是执行重新渲染的操作.

当你的 View 过于复杂,操作又过于复杂,就会计算渲染时间超过 16ms,产生卡顿问题.



如何优化：

(1)Android 系统已经对它优化

在 Android 里面那些由主题所提供的资源，例如 Bitmaps，Drawables 都是一起打包到统一的 Texture 纹理当中，然后再传递到 GPU 里面，这意味着每次你需要使用这些资源的时候，都是直接从纹理里面进行获取渲染的。

(2)我们要做的优化

扁平化处理,防止过度绘制 OverDraw

1) 每一个 layout 的最外层父容器 是否需要?

2) 布局层级优化

进行检测时,可能会让多种检测工具冲突,用 Android Device Monitor 的时候,最好关闭相关手机上的开发者检测工具开关.

查看自己的布局，深的层级,是否可以做优化.

渲染比较耗时(颜色就能看出来),想办法能否减少层级以及优化每一个 View 的渲染时间.使用 Hierarchy Viewer 工具检测

3) 图片选择

① **Android 的界面能用 png 最好是用 png：**因为 32 位的 png 颜色过渡平滑且支持透明。jpg 是像素化压缩过的图片，质量已经下降了，再拿来做 9path 的按钮和平铺拉伸的控件必然惨不忍睹，要尽量避免。

② **对于颜色繁杂的，照片墙纸之类的图片（应用的启动画面喜欢搞这种），那用 jpg 是最好不过了：**这种图片压缩前压缩后肉眼分辨几乎不计，如果保存成 png 体积将是 jpg 的几倍甚至几十倍，严重

浪费体积。

4） **清理不必要的背景**

5） **当背景无法避免,尽量用 Color.TRANSPARENT**

因为透明色 Color.TRANSPARENT 是不会被渲染的,他是透明的.

6） **优化自定义 View 的计算**

View 中的方法 OnMeasure,OnLayout,OnDraw.在我们自定义 View 起到了决定作用,我们要学会研究其中的优化方法.

学会裁剪掉 View 的覆盖部分,增加 cpu 的计算量,来优化 GPU 的渲染

最后,Android Device Monitor 是个好东西~ 性能优化的工具基本都在其中.

17. Bitmap 如何处理大图，如一张 30M 的大图，如何预防 OOM

重点是在对于对内存的了解以及内存使用率的掌握

https://blog.csdn.net/guolin_blog/article/details/9316683

高效加载大图片

BitmapFactory 这个类提供了多个解析方法(decodeByteArray, decodeFile, decodeResource 等)用于创建 Bitmap 对象，我们应该根据图片的来源选择合适的方法。比如 SD 卡中的图片可以使用 decodeFile 方法，网络上的图片可以使用 decodeStream 方法 ,资源文件中的图片可以使用 decodeResource 方法。

这些方法会尝试为已经构建的 bitmap 分配内存 ,这时就会很容易导致 OOM 出现。为此每一种解析方法都提供了一个可选的 BitmapFactory.Options 参数，将这个参数的 inJustDecodeBounds 属性设置为 true 就可以让解析方法禁止为 bitmap 分配内存 , 返回值也不再是一个 Bitmap 对象 , 而是 null。虽然 Bitmap 是 null 了，但是 BitmapFactory.Options 的 outWidth、outHeight 和 outMimeType 属性都会被赋值。这个技巧让我们可以在加载图片之前就获取到图片的长宽值和 MIME 类型，从而根据情况对图片进行压缩。如下代码所示：

```
BitmapFactory.Options options = new BitmapFactory.Options();

options.inJustDecodeBounds = true;

BitmapFactory.decodeResource(getResources(), R.id.myimage, options);
int imageHeight = options.outHeight;

int imageWidth = options.outWidth;

String imageType = options.outMimeType;
```

为了避免 OOM 异常 , 最好在解析每张图片的时候都先检查一下图片的大小 , 除非你非常信任图片的来源，保证这些图片都不会超出你程序的可用内存。

现在图片的大小已经知道了 ,我们就可以决定是把整张图片加载到内存中还是加载一个压缩版的图片到内存中。以下几个因素是我们需要考虑的：

预估一下加载整张图片所需占用的内存。

为了加载这一张图片你所愿意提供多少内存。

用于展示这张图片的控件的实际大小。

当前设备的屏幕尺寸和分辨率。

下面的方法可以根据传入的宽和高，计算出合适的 inSampleSize 值：

```
public static int calculateInSampleSize(BitmapFactory.Options options,
        int reqWidth, int reqHeight) {
    // 源图片的高度和宽度
    final int height = options.outHeight;
    final int width = options.outWidth;
    int inSampleSize = 1;
    if (height > reqHeight || width > reqWidth) {
        // 计算出实际宽高和目标宽高的比率
        final int heightRatio = Math.round((float) height / (float) reqHeight);
        final int widthRatio = Math.round((float) width / (float) reqWidth);
        // 选择宽和高中最小的比率作为 inSampleSize 的值，这样可以保证最终图片的
宽和高
        // 一定都会大于等于目标的宽和高。
        inSampleSize = heightRatio < widthRatio ? heightRatio : widthRatio;
    }
    return inSampleSize;
}
```

.

使用这个方法，首先你要将 BitmapFactory.Options 的 inJustDecodeBounds 属性设置为 true，解析一次图片。然后将 BitmapFactory.Options 连同期望的宽度和高度一起传递到到 calculateInSampleSize 方法中，就可以得到合适的 inSampleSize 值了。之后再解析一次图

片，使用新获取到的 inSampleSize 值，并把 inJustDecodeBounds 设置为 false，就可以得到压缩后的图片了。

```
public static Bitmap decodeSampledBitmapFromResource(Resources res, int resId,
        int reqWidth, int reqHeight) {
    // 第一次解析将 inJustDecodeBounds 设置为 true，来获取图片大小
    final BitmapFactory.Options options = new BitmapFactory.Options();
    options.inJustDecodeBounds = true;
    BitmapFactory.decodeResource(res, resId, options);
    // 调用上面定义的方法计算 inSampleSize 值
    options.inSampleSize = calculateInSampleSize(options, reqWidth,
reqHeight);
    // 使用获取到的 inSampleSize 值再次解析图片
    options.inJustDecodeBounds = false;
    return BitmapFactory.decodeResource(res, resId, options);
}
```

下面的代码非常简单地将任意一张图片压缩成 100*100 的缩略图，并在 ImageView 上展示。

```
mImageView.setImageBitmap(
    decodeSampledBitmapFromResource(getResources(), R.id.myimage, 100,
100));
```

使用图片缓存技术

为了保证内存的使用始终维持在一个合理的范围，通常会把被移除屏幕的图片进行回收处理。此时垃圾回收器也会认为你不再持有这些图片的引用，从而对这些图片进行 GC 操作。用这种思路来解决问题是非常好的，可是为了能让程序快速运行，在界面上迅速地加载图片，你又必须要考虑到某些图片被回收之后，用户又将它重新滑入屏幕这种情况。这时重新去加载一遍刚刚加载过的图片无疑是性能的瓶颈，你需要想办法去避免这个情况的发生。

使用内存缓存技术来对图片进行缓存，从而让你的应用程序在加载很多图片的时候可以提高响应速度和流畅性。

内存缓存技术对那些大量占用应用程序宝贵内存的图片提供了快速访问的方法。其中最核心的类是 LruCache (此类在 android-support-v4 的包中提供) 。这个类非常适合用来缓存图片，它的主要算法原理是把最近使用的对象用强引用存储在 LinkedHashMap 中，并且把最近最少使用的对象在缓存值达到预设定值之前从内存中移除。

在过去，我们经常会使用一种非常流行的内存缓存技术的实现，即软引用或弱引用 (SoftReference or WeakReference)。但是现在已经不再推荐使用这种方式了，因为从 Android 2.3 (API Level 9)开始，垃圾回收器会更倾向于回收持有软引用或弱引用的对象，这让软引用和弱引用变得不再可靠。另外，Android 3.0 (API Level 11)中，图片的数据会存储在本地的内存当中，因而无法用一种可预见的方式将其释放，这就有潜在的风险造成应用程序的内存溢出并崩溃。

为了能够选择一个合适的缓存大小给 LruCache，有以下多个因素应该放入考虑范围内，例如：

你的设备可以为每个应用程序分配多大的内存？

设备屏幕上一次最多能显示多少张图片？有多少图片需要进行预加载，因为有可能很快也会显示在屏幕上？

你的设备的屏幕大小和分辨率分别是多少？一个超高分辨率的设备（例如 Galaxy Nexus）比起一个较低分辨率的设备（例如 Nexus S），在持有相同数量图片的时候，需要更大的缓存空间。

图片的尺寸和大小，还有每张图片会占据多少内存空间。

图片被访问的频率有多高？会不会有一些图片的访问频率比其它图片要高？如果有的话，你也许应该让一些图片常驻在内存当中，或者使用多个 LruCache 对象来区分不同组的图片。

你能维持好数量和质量之间的平衡吗？有些时候，存储多个低像素的图片，而在后台去开线程加载高像素的图片会更加的有效。

下面是一个使用 LruCache 来缓存图片的例子：

.

```
private LruCache<String, Bitmap> mMemoryCache;



@Override

protected void onCreate(Bundle savedInstanceState) {
```

```
    // 获取到可用内存的最大值，使用内存超出这个值会引起 OutOfMemory 异常。
    // LruCache 通过构造函数传入缓存值，以 KB 为单位。
    int maxMemory = (int) (Runtime.getRuntime().maxMemory() / 1024);
    // 使用最大可用内存值的 1/8 作为缓存的大小。
    int cacheSize = maxMemory / 8;
    mMemoryCache = new LruCache<String, Bitmap>(cacheSize) {
        @Override
        protected int sizeOf(String key, Bitmap bitmap) {
            // 重写此方法来衡量每张图片的大小，默认返回图片数量。
            return bitmap.getByteCount() / 1024;
        }
    };
}

public void addBitmapToMemoryCache(String key, Bitmap bitmap) {
    if (getBitmapFromMemCache(key) == null) {
        mMemoryCache.put(key, bitmap);
    }
}

public Bitmap getBitmapFromMemCache(String key) {
    return mMemoryCache.get(key);
}
```

.

在这个例子当中，使用了系统分配给应用程序的八分之一内存来作为缓存大小。在中高配置的手机当中，这大概会有 4 兆(32/8)的缓存空间。一个全屏幕的 GridView 使用 4 张 800x480 分辨率的图片来填充，则大概会占用 1.5 兆的空间(800*480*4)。因此，这个缓存

大小可以存储 2.5 页的图片。
当向 ImageView 中加载一张图片时,首先会在 LruCache 的缓存中进行检查。如果找到了相应的键值，则会立刻更新 ImageView ，否则开启一个后台线程来加载这张图片。

```
public void loadBitmap(int resId, ImageView imageView) {
    final String imageKey = String.valueOf(resId);
    final Bitmap bitmap = getBitmapFromMemCache(imageKey);
    if (bitmap != null) {
        imageView.setImageBitmap(bitmap);
    } else {
        imageView.setImageResource(R.drawable.image_placeholder);
        BitmapWorkerTask task = new BitmapWorkerTask(imageView);
        task.execute(resId);
    }
}
```

BitmapWorkerTask 还要把新加载的图片的键值对放到缓存中。

```
class BitmapWorkerTask extends AsyncTask<Integer, Void, Bitmap> {
    // 在后台加载图片。
    @Override
    protected Bitmap doInBackground(Integer... params) {
        final Bitmap bitmap = decodeSampledBitmapFromResource(
                getResources(), params[0], 100, 100);
        addBitmapToMemoryCache(String.valueOf(params[0]), bitmap);
        return bitmap;
    }
}
```

掌握了以上两种方法，不管是要在程序中加载超大图片，还是要加载大量图片，都不用担心 OOM 的问题了!

18. java 中的四种引用的区别以及使用场景

https://blog.csdn.net/qq_23547831/article/details/46505287

Java 中存在四种引用，它们分别是：

(1)强引用(StrongReference)

强引用是使用最普遍的引用。如果一个对象具有强引用，那垃圾回收器绝不会回收它。当内存空间不足，Java 虚拟机宁愿抛出 OutOfMemoryError 错误，使程序异常终止，也不会靠随意回收具有强引用的对象来解决内存不足的问题。

(2)软引用(SoftReference)

如果一个对象只具有软引用，则内存空间足够，垃圾回收器就不会回收它；如果内存空间不足了，就会回收这些对象的内存。只要垃圾回收器没有回收它，该对象就可以被程序使用。软引用可用来实现内存敏感的高速缓存。

软引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被垃圾回收器回收，Java 虚拟机就会把这个软引用加入到与之关联的引用队列中。

(3)弱引用(WeakReference)

弱引用与软引用的区别在于：只具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。不过，由于垃圾回收器是一个优先级很低的线程，因此不一定会很快发现那些只具有弱引用的对象。

弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果弱引用所引用的对象被垃圾回收，Java 虚拟机就会把这个弱引用加入到与之关联的引用队列中。

(4)虚引用(PhantomReference)

"虚引用"顾名思义，就是形同虚设，与其他几种引用都不同，虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收器回收。

虚引用主要用来跟踪对象被垃圾回收器回收的活动。虚引用与软引用和弱引用的一个区别在于：虚引用必须和引用队列（ReferenceQueue）联合使用。当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之 关联的引用队列中。

19. 强引用置为 null，会不会被回收？

会，GC 执行时，就被回收掉，前提是没有被引用的对象

https://blog.csdn.net/qq_33048603/article/details/52727991

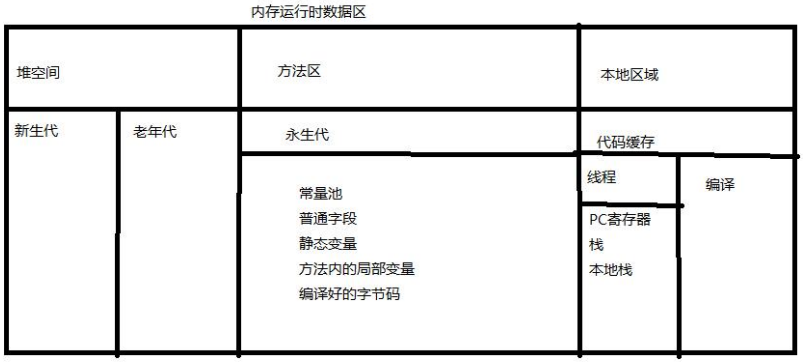
一定要了解垃圾回收原理

首先我们在讲解之前我们需要了解一下 JVM 内存运行时数据区的三个重要的地方

(1) **堆(heap)**：他是最大的一块区域，用于存放对象实例和数组，是全局共享的.（也称为逻辑堆，主要用来存放对象实例与数组，对于所有的线程来说他是共享的，对于 Heap 堆区是动态分配内存的，所以空间大小和生命周期都不是明确的，而 GC 的主要作用就是自动释放逻辑堆里实例对象所占的内存，而在逻辑堆中还分为新生代与老年代，用来区分对象的存活时间，在新生代中还被细致的分为 Eden SurvivorFrom 以及 SurvivorTo 这三部分.）

(2) **栈(stack)**：全称为虚拟机栈，主要存储基本数据类型，以及对象的引用，私有线程（在每一个对象被创建的时候，在堆栈区都有一个对他的引用，在这里我们可以这样理解。）

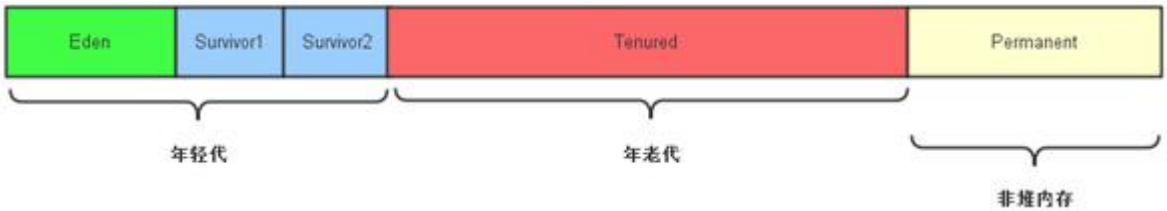
(3) **方法区(Method Area)**：在 class 被加载后的一些信息 如常量，静态常量这些被放在这里，在 Hotspot 里面我们将它称之为永生代（方法区主要存储（类加载器）ClassLoader 加载的类信息，在这里我们可以理解为已经编译好的代码储存区，所以存储包括类的元数据，常量池，字段，静态变量与方法内的局部变量以及编译好的字节码，等等）



Object obj = new Object();

上面的代码左边的 Object obj 等于在堆栈区申请了一个内存，这里也就是对类的引用了，而 new Object()则是生成了一个实例，=则是 将对象的内容则可通过 obj 进行访问，在 Java 里都是通过引用来操纵对象的。

我们知道对象的实例是存在于逻辑堆中，而 GC 在逻辑堆是怎样运行的呢，下面我们看下逻辑堆的具体结构



逻辑堆分为 年轻代与年老代，而年轻代则被分为 eden、survivor1、 survivor2 ，对于一个新被实例化的对象都是存在于年轻代中的 eden 区

按照 GC 的运行机制，会回收掉已经死掉的对象，而对象一般都是在年轻代就会死去，所以年轻代比老年代需要更频繁的 GC 清理

年轻代与年老代使用不同算法来进行 GC 操作

年轻代 ：

在年轻代中 jvm 使用的是 Mark-copy 算法，就像算法名字说的那样有两个步骤，第一是标记(Mark) 第二是 copy（复制），Mark 主要用于标记出还活着的实例，然后清除掉没有被标记的实例，释放内存，然后 Copy 部分则是将还活着的实例根据年龄拷贝到不同的年龄代

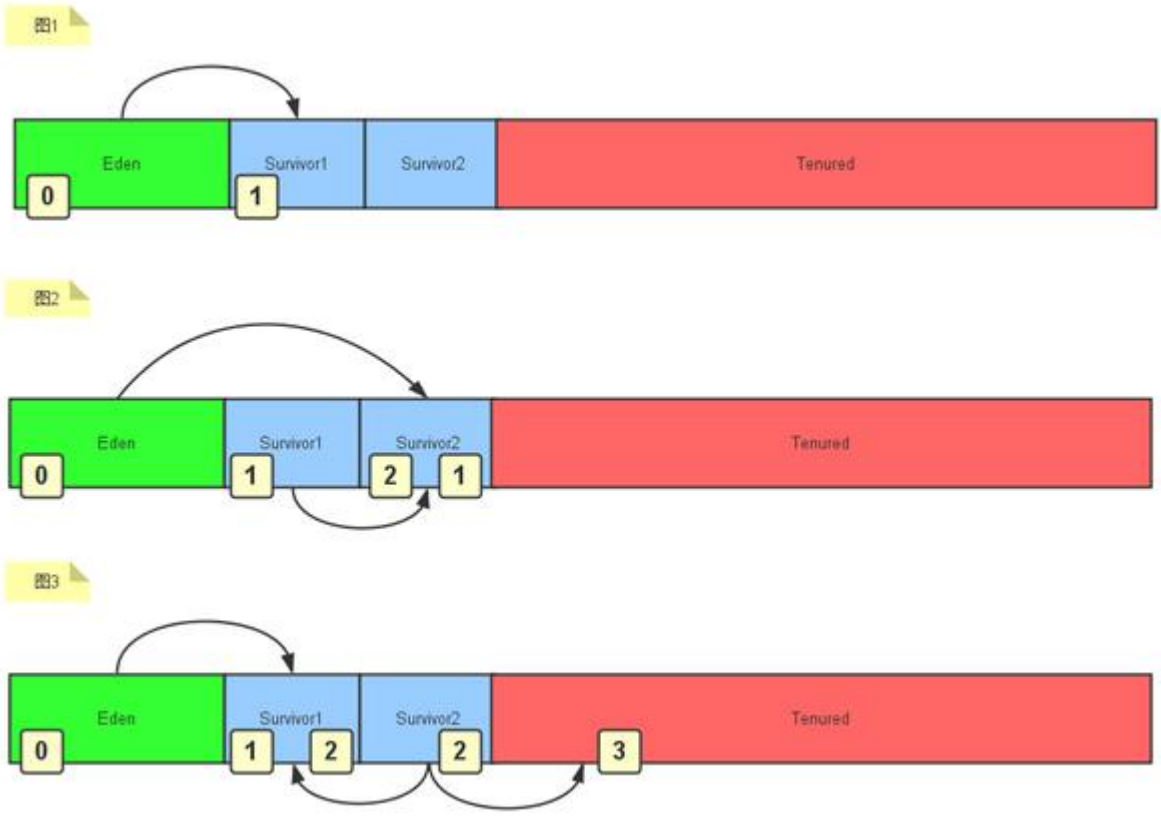
jvm 区在年轻代分年龄代方法：

对于标记 与区分年龄代的技术 我们一般都是用到的都是引用计数器，在每一个对象中都含有引用计数器，都有引用指向对象的时候 引用计数器就会加 1（应该是使用 gcroot 查找，如果对象可达，则对象年龄+1.到下一个晋升的空间去），不在被引用 计数器 减 1，对与垃圾回收的策略则是标记所有活着的实例，将没有被标记的实例全部回收 释放内存，

对于静态，我们都知道静态方法与静态变量是不会产生实例的，直接通过类的引用，使用 ClassLoader 进行加载的类数据如前面所说是不存在逻辑堆里面的，直接存在于永生代里面也就是 方法区里面，这个类一旦被清除掉里面所有的静态变量都会被清除

当我们在 Object obj 的时候 向逻辑堆中的 Eden 区域 申请内存，当 Eden 区域的内存不足的时候，这个时候会触发 GC 这个时候称 gc 为小型垃圾回收，每个实例都有一个独有的年龄，每个引用被经历过一次 GC 后就会年龄加一，同时就会将没有被清理掉的对象全都

copy 到上图的 survivor1 区域，如图 1 所示：



当第二次 GC 执行的时候就会使用 Mark 算法找到存活的对象，然后将他们的年龄加 1，并且将他们拷贝到 survivor2 区域，然后执行 GC，这样就可以实现 survivor1 与 survivor2 两个一样大的区域进行交替使用，当对象的年龄足够大的时候，对象就会被移动到老年代，这里移动到老年代的标准由 JVM 的参数所决定

年老代 ：

当 GC 被触发的时候 eden 的对象会转到 survivor1 然后再次就会转到 survivor2 ，当 survivor1 的对象太大了 survivor2 的区域无法容纳得部分就会转到 Tenured 的区域，当 Tenured 的区域也容不下的时候就会自动移动到年老代，在移动年老代的时候会先触发年老

代上面的 GC 然后在将 Tenured 容纳不下的对象放入年老代，对于年老代的 GC 算法与年轻代的 Mrak-copy 算法有很大不同

年老代的 GC 算法在 jdk 1.7 中分为五种，

1) Serial GC

2) Parallel GC

3) Parallel Old GC(Parallel Compacting GC)

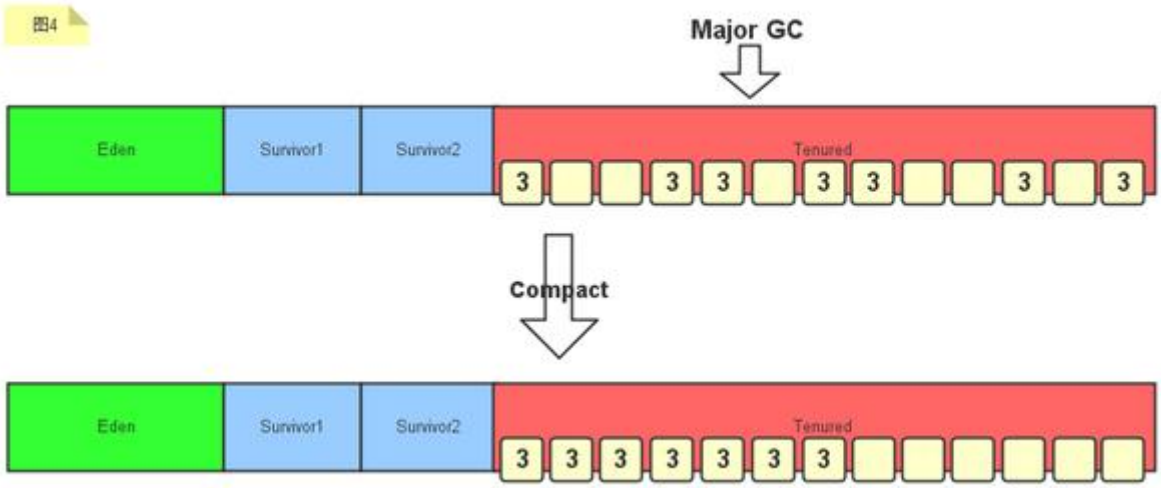
4) Concurrent Mark & Sweep GC (or “CMS”)

5) Garbage First (G1) GC

在这里我只讲解两种 Parallel Scavenge 与 Concurrent Mark sweeps 对与这两种接下来会进行简单的讲解

Parallel Scavenge ：

在这里我们简称他为 PS 算法，PS 算法执行的是 Mark-compact 算法的过程 ，并且是用多线程进行执行这样提高了执行效率，这里的 Mrak 还是与之前的年轻代的 Mark 原理是一样的，但是 Compat 算法则是将年老代的对象进行碎片化的整理，并且年老代是没有像年轻代的那样有 survivor1 与 survivor2 来将残留的对象全部 copy 过去，考虑到年老代的对象比较多，所以就需要进行碎片化整理如下图：



Concurrent Mark sweeps：

我们简称他为 CMS 算法。GC 调优通常就是为了改善 stop-the-world 的时间。在 CMS GC 开始时的初始标记(initial mark)比较简单，只有靠近类加载器的存活对象会被标记，因此停顿时间(stop-the-world)比较短暂。

在并发标记(concurrent mark)阶段，由刚被确认和标记过的存活对象所关联的对象将被会跟踪和检测存活状态。此步骤的不同之处在于有多个线程并行处理此过程。

在重标记(remark)阶段，由并发标记所关联的新增或中止的对象瘵被会检测。在最后的并发清理(concurrent sweep)阶段，垃圾回收过程被真正执行。

在垃圾回收执行过程中，其他线程依然在执行。得益于 CMS GC 的执行方式，在 GC 期间系统中断时间非常短暂。CMS GC 也被称为低延迟 GC，

Stop the world：

对于 Stop the world，不管选择哪种 GC 算法，stop-the-world

都是不可避免的。Stop-the-world 意味着从应用中停下来并进入到 GC 执行过程中去。一旦 Stop-the-world 发生，除了 GC 所需的线程外，其他线程都将停止工作，中断了的线程直到 GC 任务结束才继续它们的任务。