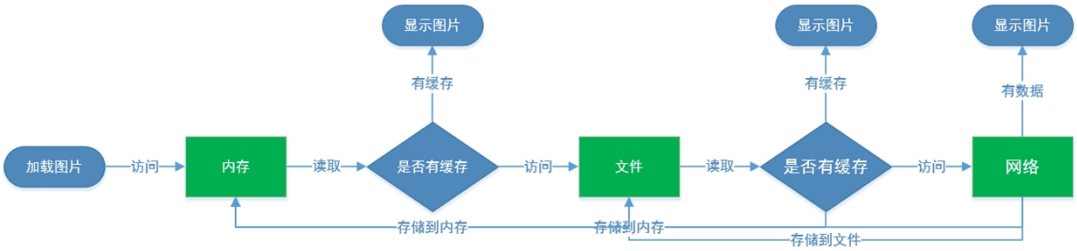
性能优化

1、图片的三级缓存中,图片加载到内存中,如果内存快爆了,会发生什么？怎么处理？

- 参考回答：
  - 首先我们要清楚图片的三级缓存是如何的

图片的三级缓存流程图



如果内存足够时不回收。内存不够时就回收软引用对象

2、内存中如果加载一张 500*500 的 png 高清图片.应该是占用多少的内存?

- 参考回答：
  - **不考虑屏幕比的话**：占用内存=500 * 500 * 4 = 1000000B ≈ 0.95MB
  - **考虑屏幕比的的话**：占用内存= 宽度像素 x （inTargetDensity / inDensity） x 高度像素 x

（inTargetDensity / inDensity）x 一个像素所占的内存字节大小

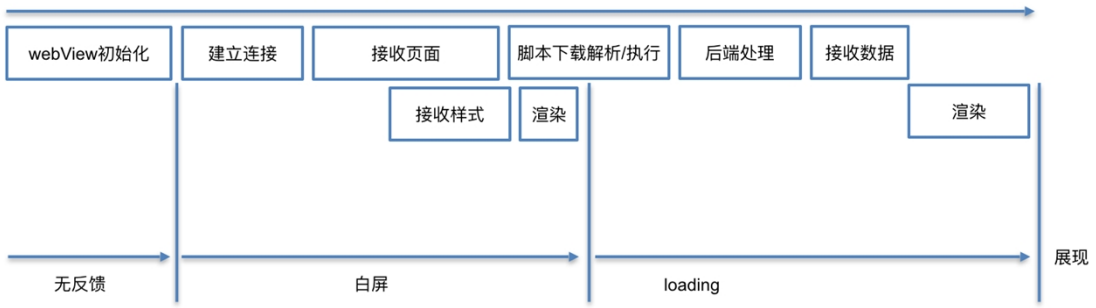
- inDensity 表示目标图片的 dpi（放在哪个资源文件夹下），inTargetDensity 表示目标屏幕的 dpi

密度	dpi范围
ldpi（低）	~120dpi
mdpi（中）	~160dpi
hdpi（高）	~240dpi
xhdpi（超高）	~320dpi
xxhdpi（超超高）	~480dpi
xxxhdpi（超超超高）	~640dpi

3、WebView 的性能优化 ？

- 参考回答：
  - 一个加载网页的过程中，native、网络、后端处理、CPU 都会参与，各自都有必要的工作和依赖关系；让他们相互并行处理而不是相互阻塞才可以让网页加载更快：
    - **WebView 初始化慢，可以在初始化同时先请求数据，让后端和网络不要闲着。**
    - **常用 JS 本地化及延迟加载，使用第三方浏览内核**

- **后端处理慢，可以让服务器分 trunk 输出，在后端计算的同时前端也加载网络静态资源。**
- **脚本执行慢，就让脚本在最后运行，不阻塞页面解析。**
- **同时，合理的预加载、预缓存可以让加载速度的瓶颈更小。**
- **WebView 初始化慢，就随时初始化好一个 WebView 待用。**
- **DNS 和链接慢，想办法复用客户端使用的域名和链接。**

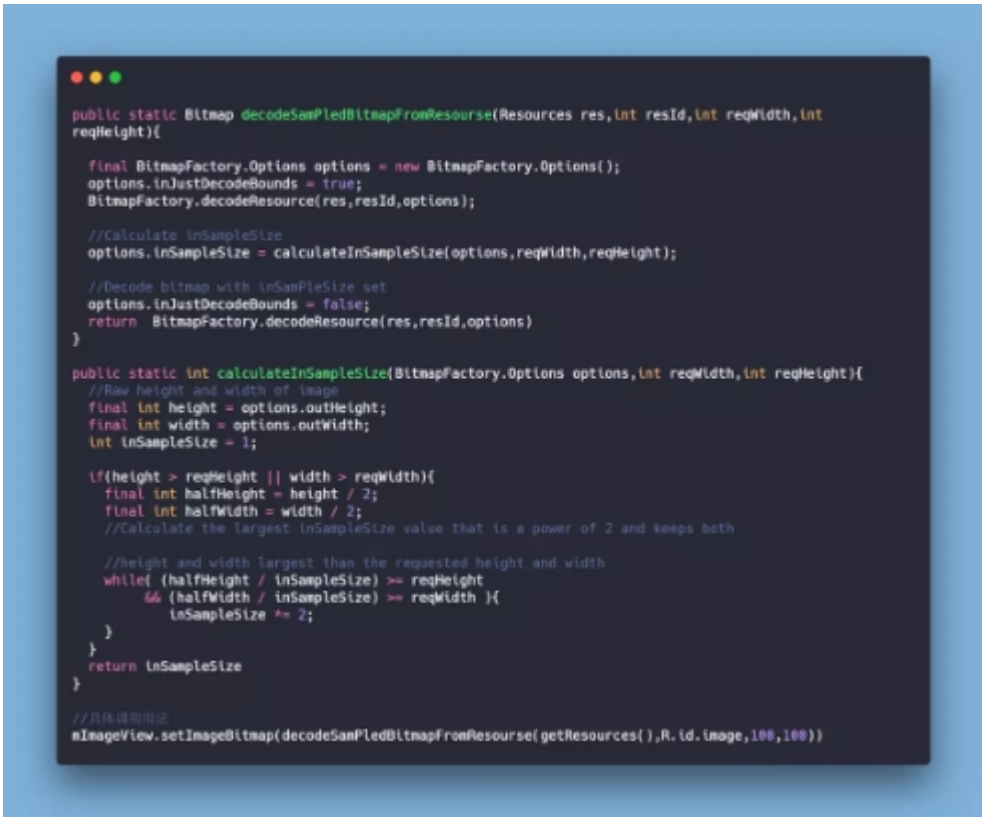


4、Bitmap 如何处理大图，如一张 30M 的大图，如何预防 OOM？

- 参考回答：避免 OOM 的问题就需要对大图片的加载进行管理，主要通过缩放来减小图片的内存占用。
    - BitmapFactory 提供的加载图片的四类方法（**decodeFile、decodeResource、decodeStream、decodeByteArray**）都支持 BitmapFactory.Options 参数，通过 inSampleSize 参数就可以很方便地对一个图片进行采样缩放

- 比如一张 10241024 的高清图片来说。那么它占有的内存为 102410244，即 4MB，如果 inSampleSize 为 2，那么采样后的图片占用内存只有 512512*4,即 1MB（**注意：根据最新的官方文档指出，inSampleSize 的取值应该总是为 2 的指数，即 1、2、4、8 等等，如果外界输入不足为 2 的指数，系统也会默认选择最接近 2 的指数代替，比如 2**）
- 综合考虑。通过采样率即可有效加载图片，流程如下

    - **将 BitmapFactory.Options 的 inJustDecodeBounds 参数设为 true 并加载图片**
    - **从 BitmapFactory.Options 中取出图片的原始宽高信息，它们对应 outWidth 和 outHeight 参数**
    - **根据采样率的规则并结合目标 View 的所需大小计算出采样率 inSampleSize**
    - **将 BitmapFactory.Options 的 inJustDecodeBounds 参数设为 false，重新加载图片**



```
public static Bitmap decodeSamPledBitmapFromResourse(Resources res,int resId,int reqWidth,int
reqHeight){

  final BitmapFactory.Options options = new BitmapFactory.Options();
  options.inJustDecodeBounds = true;
  BitmapFactory.decodeResource(res,resId,options);

  //Calculate inSampleSize
  options.inSampleSize = calculateInSampleSize(options,reqWidth,reqHeight);

  //Decode bitmap with inSamPleSize set
  options.inJustDecodeBounds = false;
  return  BitmapFactory.decodeResource(res,resId,options)
}

public static int calculateInSampleSize(BitmapFactory.Options options,int reqWidth,int reqHeight){
  //Raw height and width of image
  final int height = options.outHeight;
  final int width = options.outWidth;
  int inSampleSize = 1;

  if(height > reqHeight || width > reqWidth){
    final int halfHeight = height / 2;
    final int halfWidth = width / 2;
    //Calculate the largest inSampleSize value that is a power of 2 and keeps both

    //height and width largest than the requested height and width
    while( (halfHeight / inSampleSize) >= reqHeight
        && (halfWidth / inSampleSize) >= reqWidth ){
          inSampleSize *= 2;
    }
  }
  return inSampleSize
}

//具体调用用法
mImageView.setImageBitmap(decodeSamPledBitmapFromResourse(getResources(),R.id.image,100,100))
```

5、内存回收机制与 GC 算法(各种算法的优缺点以及应用场景)；GC 原理时机以及 GC 对象

- 参考回答：
  - 内存判定对象可回收有两种机制：
    - **引用计数算法**：给对象中添加一个引用计数器，每当有一个地方引用它时，计数器值就加 1；当引用失效时，计数器值就减 1；任何时刻计数器为 0 的对象就是不可能再被使用的。然而在主流的 Java 虚拟机里未选用引用计数算法来管理内存，主要原因是它难以解决对象之间

相互循环引用的问题，所以出现了另一种对象存活判定算法。

- **可达性分析法**：通过一系列被称为『GCRoots』的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为**引用链**，当一个对象到 GC Roots 没有任何引用链相连时，则证明此对象是不可用的。其中可作为 GC Roots 的对象：虚拟机栈中引用的对象，主要是指栈帧中的本地变量*、本地方法栈中 **Native** 方法引用的对象、方法区中**类静态**属性引用的对象、方法区中**常量**引用的对象

- GC 回收算法有以下四种：

  - **分代收集算法**：是当前商业虚拟机都采用的一种算法，根据对象存活周期的不同，将 Java 堆划分为新生代和老年代，并根据各个年代的特点采用最适当的收集算法。
  - 新生代：大批对象死去，只有少量存活。使用『复制算法』，只需复制少量存活对象即可。

    - **复制算法**：把可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用尽后，把还存活着的对象『复制』到另外一块上面，再将这一块内存空间一次清理掉。**实现简**

单，运行高效。在对象存活率较高时就要进行较多的复制操作，效率将会变低

- 老年代：对象存活率高。使用『标记—清理算法』或者『标记—整理算法』，只需标记较少的回收对象即可。
  - **标记-清除算法**：首先『标记』出所有需要回收的对象，然后统一『清除』所有被标记的对象。**标记和清除两个过程的效率都不高，清除之后会产生大量不连续的内存碎片，空间碎片太多可能会导致以后在程序运行过程中需要分配较大对象时，无法找到足够的连续内存而不得不提前触发另一次垃圾收集动作。**
  - **标记-整理算法**：首先『标记』出所有需要回收的对象，然后进行『整理』，使得存活的对象都向一端移动，最后直接清理掉端边界以外的内存。**标记整理算法会将所有的存活对象移动到一端，并对不存活对象进行处理，因此其不会产生内存碎片**

6、内存泄露和内存溢出的区别 ？AS 有什么工具可以检测内存泄露

- 参考回答：

- **内存溢出(out of memory)**：是指程序在申请内存时，没有足够的内存空间供其使用，出现 out of memory；比如申请了一个 integer，但给它存了 long 才能存下的数，那就是内存溢出。
- **内存泄露(memory leak)**：是指程序在申请内存后，无法释放已申请的内存空间，一次内存泄露危害可以忽略，但内存泄露堆积后果很严重，无论多少内存,迟早会被占光。**memory leak 会最终会导致 out of memory !**
- 查找内存泄漏可以使用 Android Studio 自带的 **AndroidProfiler** 工具或 **MAT**

7、性能优化,怎么保证应用启动不卡顿？黑白屏怎么处理?

- 参考回答：
  - **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所以会有卡顿的感觉。在非必要的情况下可以把加载延后或则开启子线程处理
  - 另外，影响**界面卡顿**的两大因素，分别是**界面绘制和数据处理。**
    - 布局优化(使用 include，merge 标签，复杂布局推荐使用 ConstraintLayout 等)

        - onCreate() 中不执行耗时操作 把页面显示的 View 细分一下，放在 AsyncTask 里逐步显示，用 Handler 更好。这样用户的看到的就是有层次有步骤的一个个的 View 的展示，不会是先看到一个黑屏，然后一下显示所有 View。最好做成动画，效果更自然。
        - 利用多线程的目的就是尽可能的减少 onCreate() 和 onReume() 的时间，使得用户能尽快看到页面，操作页面。
        - 减少主线程阻塞时间。
        - 提高 Adapter 和 AdapterView 的效率。

    - **黑白屏产生原因**：当我们在启动一个应用时，系统会去检查是否已经存在这样一个进程，如果不存在，系统的服务会先检查 startActivity 中的 intent 的信息，然后在去创建进程，最后启动 Acitivy，即冷启动。而启动出现白黑屏的问题，就是在这段时间内产生的。系统在绘制页面加载布局之前，首先会初始化窗口（Window），而在进行这一步操作时，系统会根据我们设置的 Theme 来指定它的 Theme 主题颜色，我们在 Style 中的设置就决定了显示的是白屏还是黑屏。

        - windowIsTranslucent 和 windowNoTitle，将这两个属性都设置成 true (会有明显的卡顿体验，不推荐)

        - 如果启动页只是是一张图片，那么为启动页专一设置一个新的主题，设置主题的 android:windowBackground 属性为启动页背景图即可
        - 使用 layer-list 制作一张图片 launcher_layer.xml，将其设置为启动页专一主题的背景，并将其设置为启动页布局的背景。

8、强引用置为 null，会不会被回收？

- 参考回答：
    - **不会立即释放对象占用的内存。** 如果对象的引用被置为 null，只是断开了当前线程栈帧中对该对象的引用关系，而 垃圾收集器是运行在后台的线程，只有当用户线程运行到安全点(safe point)或者安全区域才会扫描对象引用关系，扫描到对象没有被引用则会标记对象，这时候仍然不会立即释放该对象内存，因为有些对象是可恢复的（在 finalize 方法中恢复引用 ）。只有确定了对象无法恢复引用的时候才会清除对象内存。

9、ListView 跟 RecyclerView 的区别

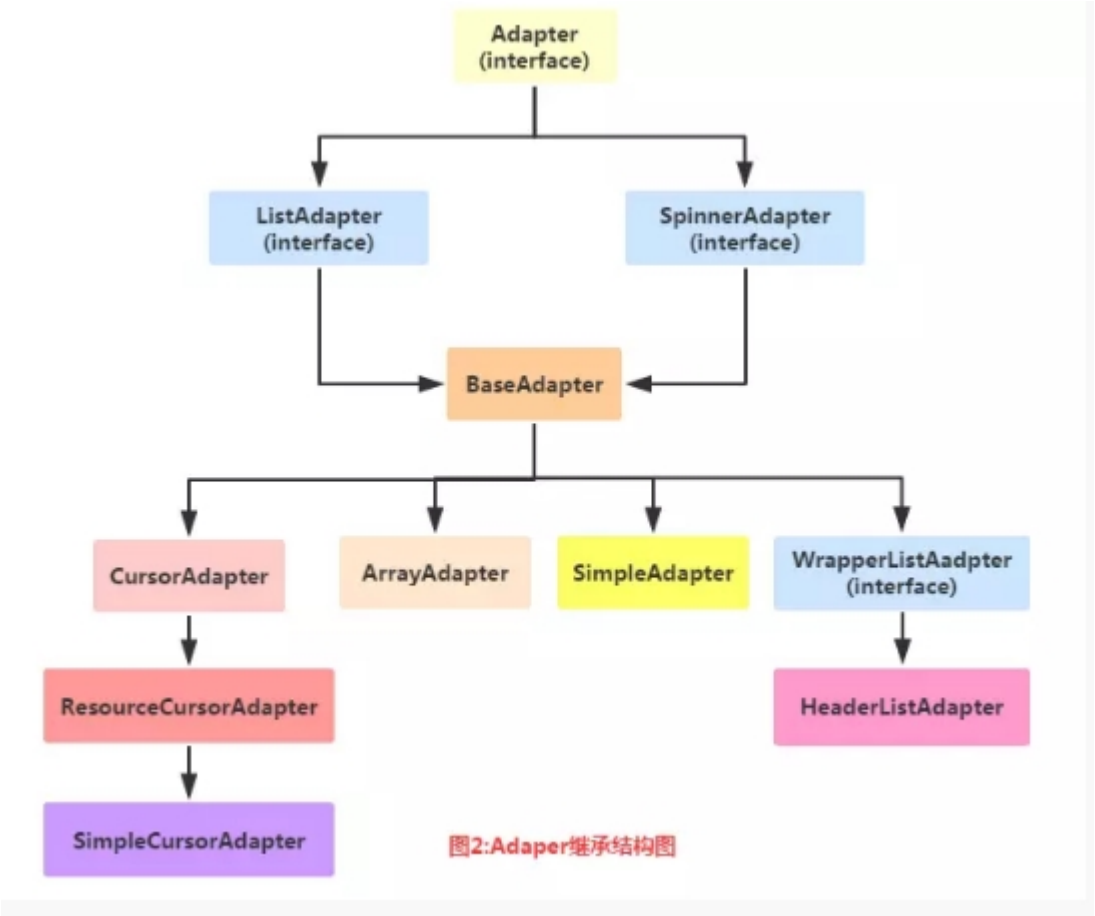
- 参考回答：

- 动画区别：
    - 在 **RecyclerView** 中，内置有许多动画 API，例如：notifyItemChanged(), notifyDataInserted(), notifyItemMoved()等等；如果需要自定义动画效果，可以通过实现（RecyclerView.ItemAnimator 类）完成自定义动画效果，然后调用 RecyclerView.setItemAnimator()；
    - 但是 **ListView** 并没有实现动画效果，但我们可以在 Adapter 自己实现 item 的动画效果；
- 刷新区别：
    - ListView 中通常刷新数据是用全局刷新 notifyDataSetChanged()，这样一来就会非常消耗资源；**本身无法实现局部刷新**，但是如果要在 ListView 实现**局部刷新**，依然是可以实现的，当一个 item 数据刷新时，我们可以在 Adapter 中，实现一个 onItemChanged()方法，在方法里面获取到这个 item 的 position（可以通过 getFirstVisiblePosition()），然后调用 getView()方法来刷新这个 item 的数据；
    - RecyclerView 中可以实现局部刷新，例如：notifyItemChanged()；

- 缓存区别：
    - RecyclerView 比 ListView 多两级缓存，支持多个离 ItemView 缓存，支持开发者自定义缓存处理逻辑，支持所有 RecyclerView 共用同一个 RecyclerViewPool(缓存池)。
    - ListView 和 RecyclerView 缓存机制基本一致，但缓存使用不同

10、ListView 的 adapter 是什么 adapter

- 参考回答：



- **BaseAdapter**：抽象类，实际开发中我们会继承这个类并且重写相关方法，用得最多的一个适配器！
- **ArrayAdapter**：支持泛型操作，最简单的一个适配器，只能展现一行文字～
- **SimpleAdapter**：同样具有良好扩展性的一个适配器，可以自定义多种效果！
- **SimpleCursorAdapter**：用于显示简单文本类型的 listView，一般在数据库那里会用到，不过有点过时，不推荐使用！

11、LinearLayout、FrameLayout、RelativeLayout 性能对比，为什么？

- 参考回答：
    - RelativeLayout 会让子 View 调用 2 次 onMeasure，LinearLayout 在有 weight 时，也会调用子 View 2 次 onMeasure
    - RelativeLayout 的子 View 如果高度和 RelativeLayout 不同，则会引发效率问题，当子 View 很复杂时，这个问题会更加严重。如果可以，尽量使用 padding 代替 margin。
    - 在不影响层级深度的情况下,使用 LinearLayout 和 FrameLayout 而不是 RelativeLayout。