Android开源图片加载库

开发团队：Android

组员：刘文静、吕伟杰、李磊

日期：2016-08-29

备注：技术信息中心-移动部门

**1 概述**

* **开发背景**

加大图片或是加载数量多的图片导致APP处于高内存开销状态，频繁引发系统GC，使得APP运行响应变慢，为解决APP中网络图片加载带来的问题，需要寻找更好的方案来优化网络图片加载。

* **开发目标**

为项目选择最优的网络图片加载库，提升APP中图片的加载速度，降低APP内存开销，尽可能避免APP由于加载大图片或是加载数量多的图片引发的OOM，提高APP的性能。

* **参考资料**

Glide

Github地址：

https://github.com/bumptech/glide

其他：

http://blog.csdn.net/fancylovejava/article/details/44747759

http://www.open-open.com/lib/view/open1438154211785.html

http://www.codeceo.com/article/android-glide-usage.html

Picasso

Github地址：

http://square.github.io/picasso/

其他：

http://www.jcodecraeer.com/a/anzhuokaifa/androidkaifa/2014/0731/1639.html

http://square.github.io/picasso/

Fresco

Github地址：

https://github.com/facebook/fresco

其他：

 http://fresco-cn.org/

http://www.jcodecraeer.com/a/anzhuokaifa/androidkaifa/2015/0402/2683.html

Universal Image Loader

Github地址：

https://github.com/nostra13/Android-Universal-Image-Loader

其他：

http://www.open-open.com/lib/view/open1433940304473.html

http://www.mincoder.com/article/3800.shtml

* **设计原则**

可移植行，可拓展性，低耦合，易使用。

**2 目前比较流行的图片开源库**

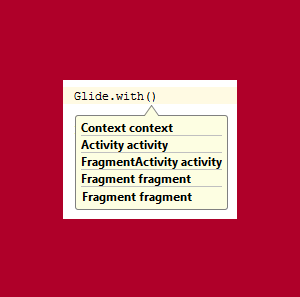
Picasso, Glide, Fresco, Universal Image Loader

* **Glide**
* Google推荐的图片加载库Glide，作者是bumptech。
* 导入依赖库：

dependencies {compile **'com.github.bumptech.glide:glide:3.7.0'** compile **'com.android.support:support-v4:22.0.0'**}

* 代码中使用：

Glide.with(context)  
 .load(**"http://inthecheesefactory.com/uploads/source/glidepicasso/cover.jpg"**)  
 .into(ivImg);

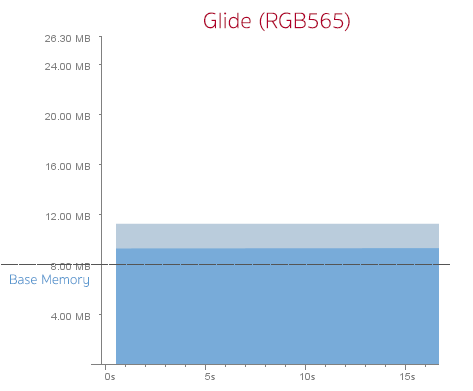


Glide的with方法不光接受Context，还接受Activity 和 Fragment，Context会自动的从他们获取。

同时将Activity/Fragment作为with()参数的好处是：图片加载会和Activity/Fragment的生命周期保持一致，比如Paused状态在暂停加载，在Resumed的时候又自动重新加载。所以我建议传参的时候传递Activity 和 Fragment给Glide，而不是Context。

* 默认Bitmap格式是RGB\_565

Glide默认的Bitmap格式是RGB\_565 ，比ARGB\_8888格式的内存开销要小一半。下面是Glide在RGB565下的内存开销图（应用自身占用了8m，因此以8为基准线比较）：



* Image质量的细节

Glide可以自动计算出任意情况下的ImageView大小，加载的大小和ImageView的大小是一致的，所以在图片质量上会有所下降，但是一般情况下肉眼还是很难察觉出来。

* 磁盘缓存

Glide在磁盘缓存策略上是跟ImageView尺寸相同的， Glide会为每种大小的ImageView缓存一次。尽管一张图片已经缓存了一次，但是假如你要在另外一个地方再次以不同尺寸显示，需要重新下载，调整成新尺寸的大小，然后将这个尺寸的也缓存起来。

具体说来就是：假如在第一个页面有一个200x200的ImageView，在第二个页面有一个100x100的ImageView，这两个ImageView本来是要显示同一张图片，却需要下载两次。

Glide既缓存全尺寸又缓存其他尺寸：

Glide.with(**this**)  
 .load(**"http://nuuneoi.com/uploads/source/playstore/**

**cover.jpg"**)  
 .diskCacheStrategy(DiskCacheStrategy.ALL)  
 .into(ivImgGlide);

下次在任何ImageView中加载图片的时候，全尺寸的图片将从缓存中取出，重新调整大小，然后缓存。 Glide的这种方式优点是加载显示非常快。

* Glide对象池

Glide原理的核心是为bitmap维护一个对象池。对象池的主要目的是通过减少大对象的分配以重用来提高性能（至于对象池的概览，可以查看 Android performance pattern ）。

Dalvik和ART虚拟机都没有使用compacting garbage collector，compacting garbage collector是一种模式，这种模式中GC会遍历堆，同时把活跃对象移到相邻内存区域，让更大的内存块可以用在后续的分配中。因为安卓没有这种模式，就可能会出现被分配的对象分散在各处，对象之间只有很小的内存可用。如果应用试图分配一个大于邻近的闲置内存块空间的对象，就会导致OutOfMemoryError，然后崩溃，即使总的空余内存空间大于对象的大小。

使用对象池还可以帮助提高滚动的性能，因为重用bitmap意味着更少的对象被创建与回收。垃圾回收会导致“停止一切(Stop The World)”事件，这个事件指的是回收器执行期间，所有线程（包括UI线程）都会暂停。这个时候，图像帧无法被渲染同时UI可能会停滞，这在滚动期间尤其明显。

* **Picasso**
* picasso是Square公司开源的一个Android图形缓存库，地址http://square.github.io/picasso/，可以实现图片下载和缓存功能。仅仅只需要一行代码就能完全实现图片的异步加载。
* 导入依赖库：

dependencies {compile **'com.squareup.picasso:picasso:2.5.2'**}

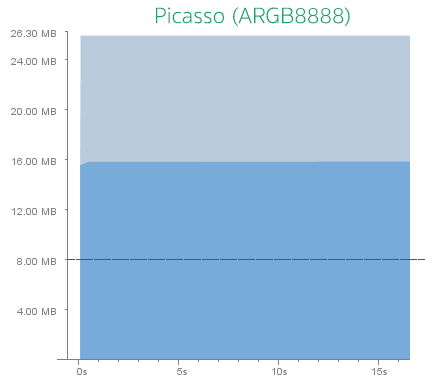
* 代码中使用：

Picasso.with(context)  
 .load(**"http://inthecheesefactory.com/uploads/source/glidepicasso/cover.jpg"**)  
 .into(ivImg);



* 默认Bitmap格式是ARGB8888

下面是Picasso在ARGB8888下的内存开销图（应用自身占用了8m，因此以8为基准线比较）：



* Image质量的细节

Picasso是加载了全尺寸的图片到内存，然后让GPU来实时重绘大小。

Picasso也可以指定加载的图片大小的：

Picasso.with(context)  
 .load(**"http://nuuneoi.com/uploads/source/playstore/cover.jpg"**)  
 .resize(768, 432)  
 .into(ivImgPicasso);

但是问题在于你需要主动计算ImageView的大小，或者说你的ImageView大小是具体的值（而不是wrap\_content），你也可以这样：

Picasso.with(context)  
 .load(**"http://nuuneoi.com/uploads/source/playstore/cover.jpg"**)  
 .fit()  
 .centerCrop()  
 .into(ivImgPicasso);

* 磁盘缓存

Picasso缓存的是全尺寸的，即便将ImageView调整成不同大小，Picasso是不管大小如何，只缓存一个全尺寸的。

在任何ImageView中加载图片的时候，Picasso的方式都是显示之前重新调整大小而导致会有一些延迟。

* **Fresco**
* Fresco是Facebook最新推出的一款用于Android应用中展示图片的强大图片库，可以从网络、本地存储和本地资源中加载图片。

Fresco 是一个强大的图片加载组件。使用它之后，你不需要再去关心图片的加载和显示这些繁琐的事情！ 支持 Android 2.3 及以后的版本。

* Image Pipeline

Fresco 中设计有一个叫做 Image Pipeline 的模块。它负责从网络，从本地文件系统，本地资源加载图片。为了最大限度节省空间和CPU时间，它含有3级缓存设计（2级内存，1级磁盘）。

* Drawees

Fresco 中设计有一个叫做 Drawees 模块，它会在图片加载完成前显示占位图，加载成功后自动替换为目标图片。当图片不再显示在屏幕上时，它会及时地释放内存和空间占用。

* 内存管理

解压后的图片，即Android中的Bitmap，占用大量的内存。大的内存占用势必引发更加频繁的GC。在5.0以下，GC将会显著地引发界面卡顿。

在5.0以下系统，Fresco将图片放到一个特别的内存区域。当然，在图片不显示的时候，占用的内存会自动被释放。这会使得APP更加流畅，减少因图片内存占用而引发的OOM。

Fresco 在低端机器上表现一样出色，你再也不用因图片内存占用而思前想后。

* 图片加载

Fresco的Image Pipeline允许你用很多种方式来自定义图片加载过程，比如：

* 为同一个图片指定不同的远程路径，或者使用已经存在本地缓存中的图片
* 先显示一个低清晰度的图片，等高清图下载完之后再显示高清图
* 加载完成回调通知
* 对于本地图，如有EXIF缩略图，在大图加载完成之前，可先显示缩略图
* 缩放或者旋转图片
* 对已下载的图片再次处理
* 支持WebP解码，即使在早先对WebP支持不完善的Android系统上也能正常使用！
* 图片绘制

Fresco 的 Drawees 设计，带来一些有用的特性：

* 自定义居中焦点
* 圆角图，当然圆圈也行
* 下载失败之后，点击重现下载
* 自定义占位图，自定义overlay, 或者进度条
* 指定用户按压时的overlay
* 图片的渐进式呈现

渐进式的JPEG图片格式已经流行数年了，渐进式图片格式先呈现大致的图片轮廓，然后随着图片下载的继续，呈现逐渐清晰的图片，这对于移动设备，尤其是慢网络有极大的利好，可带来更好的用户体验。

Android 本身的图片库不支持此格式，但是Fresco支持。使用时，和往常一样，仅仅需要提供一个图片的URI即可，剩下的事情，Fresco会处理。

* 动图加载

加载Gif图和WebP动图在任何一个Android开发者眼里看来都是一件非常头疼的事情。每一帧都是一张很大的Bitmap，每一个动画都有很多帧。Fresco让你没有这些烦恼，它处理好每一帧并管理好你的内存。

* 导入依赖库：

编辑 build.gradle 文件

dependencies {  
 *// 其他依赖* compile **'com.facebook.fresco:fresco:0.12.0'**}

下面的依赖需要根据需求添加：

dependencies {  
 *// 在 API < 14 上的机器支持 WebP 时，需要添加* compile **'com.facebook.fresco:animated-base-support:0.12.0'**

*// 支持 GIF 动图，需要添加* compile **'com.facebook.fresco:animated-gif:0.12.0'**

*// 支持 WebP （静态图+动图），需要添加* compile **'com.facebook.fresco:animated-webp:0.12.0'** compile **'com.facebook.fresco:webpsupport:0.12.0'**

*// 仅支持 WebP 静态图，需要添加* compile **'com.facebook.fresco:webpsupport:0.12.0'** }

* 开始使用 Fresco
* 初始化Fresco类，在Application 里面调用Fresco.initialize一次即可完成初始化。

*/\*\*\*\*\*\*\*\*\*\*\*  
 \*  
 \** ***@Author*** *rape flower  
 \** ***@Date*** *2016-08-29 16:35  
 \** ***@Describe*** *自定义Application  
 \*  
 \*/* **public class** XApplication **extends** Application{  
  
 @Override  
 **public void** onCreate() {  
 **super**.onCreate();  
 Fresco.*initialize*(**this**);  
 }  
 }

* AndroidManifest.xml 中指定你的 Application 类。为了下载网络图片，请确认你声明了网络请求的权限。

<**manifest xmlns:android="http://schemas.android.com/apk/res/android"  
 package="com.android.lily.image"**>  
  
 <**uses-permission android:name="android.permission.INTERNET"** />  
 <**application  
 android:name=".application.XApplication"  
 android:label="@string/app\_name"**

**...** >  
 ...  
 </**application**>  
 ...  
</**manifest**>

* 在xml布局文件中, 加入命名空间和加入SimpleDraweeView：

<**RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"  
 xmlns:fresco="http://schemas.android.com/apk/res-auto"  
 android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent"  
 android:background="@android:color/white"**>  
  
 <**com.facebook.drawee.view.SimpleDraweeView  
 android:id="@+id/my\_image\_view"  
 android:layout\_width="130dp"  
 android:layout\_height="130dp"  
 fresco:placeholderImage="@mipmap/ic\_launcher"** />  
</**RelativeLayout**>

* 开始加载图片：

*//加载网络图片*Uri uri = Uri.*parse*(**"https://cbu01.alicdn.com/img/ibank/2015/974/624/2383426479\_890623294.summ.jpg"**);  
SimpleDraweeView draweeView = (SimpleDraweeView) findViewById(R.id.***my\_image\_view***);  
draweeView.setImageURI(uri);

剩下的，Fresco会替你完成:

* 显示占位图直到加载完成；
* 下载图片；
* 缓存图片；
* 图片不再显示时，从内存中移除；
* 关键概念
* Drawees

Drawees 负责图片的呈现。它由三个元素组成，有点像MVC模式。

* DraweeView

继承于 View, 负责图片的显示。

一般情况下，使用 SimpleDraweeView 即可。 你可以在 XML 或者在 Java 代码中使用它，通过 setImageUri 给它设置一个 URI 来使用。

* DraweeHierarchy

DraweeHierarchy 用于组织和维护最终绘制和呈现的 Drawable 对象，相当于MVC中的M。

你可以通过它来在Java代码中自定义图片的展示，具体可参考：

http://fresco-cn.org/docs/using-drawees-code.html

* DraweeController

DraweeController 负责和 image loader 交互（ Fresco 中默认为 image pipeline, 当然你也可以指定别的），可以创建一个这个类的实例，来实现对所要显示的图片做更多的控制。

如果你还需要对Uri加载到的图片做一些额外的处理，那么你会需要这个类的。

* DraweeControllerBuilder

DraweeControllers 由 DraweeControllerBuilder 采用 Builder 模式创建，创建之后，不可修改。具体参见: 使用ControllerBuilder。

* Listeners

使用 ControllerListener 的一个场景就是设置一个 Listener监听图片的下载。

具体可参考：

http://fresco-cn.org/docs/listening-download-events.html

* The Image Pipeline

Fresco 的 Image Pipeline 负责图片的获取和管理。图片可以来自远程服务器，本地文件，或者Content Provider，本地资源。压缩后的文件缓存在本地存储中，Bitmap数据缓存在内存中。

在5.0系统以下，Image Pipeline 使用 pinned purgeables 将Bitmap数据避开Java堆内存，存在ashmem(匿名共享内存驱动)中。这要求图片不使用时，要显式地释放内存。

SimpleDraweeView自动处理了这个释放过程，所以没有特殊情况，尽量使用SimpleDraweeView，在特殊的场合，如果有需要，也可以直接控制Image Pipeline。

* 支持的URI

Fresco 支持许多URI格式。

特别注意：Fresco 不支持 相对路径的URI. 所有的 URI 都必须是绝对路径，并且带上该 URI 的 scheme。

如下：



res 示例:

Uri uri = Uri.*parse*(**"res://包名(实际可以是任何字符串甚至留空)/"** + R.drawable.ic\_launcher);

注意，只有图片资源才能使用在Image

pipeline中，比如(PNG)。其他资源类型，比如字符串，或者XML Drawable在Imagepipeline中没有意义。所以加载的资源不支持这些类型。

像ShapeDrawable这样声明在XML中的drawable可能引起困惑。注意到这毕竟不是图片。如果想把这样的drawable作为图像显示，那么把这个drawable设置为占位图，然后把URI设置为null。

* Drawee 指南

具体参考：

http://fresco-cn.org/docs/using-drawees-xml.html

* Image Pipeline 指南

具体参考：

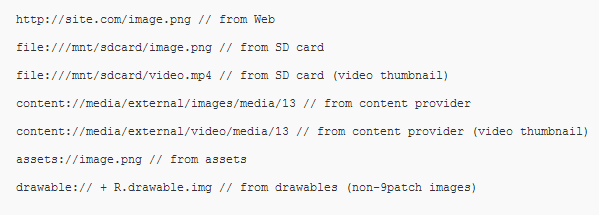
http://fresco-cn.org/docs/intro-image-pipeline.html

* 问题处理

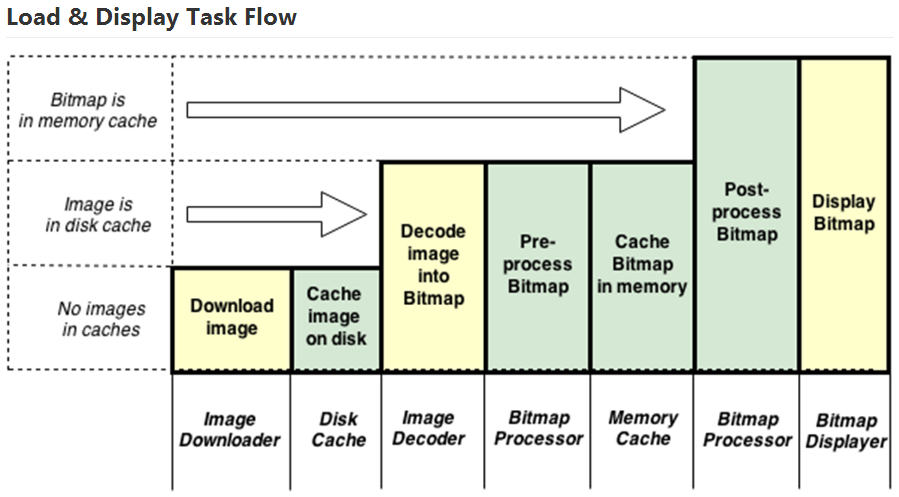
具体参考：

http://fresco-cn.org/docs/troubleshooting.html

* Universal Image Loader
* Android-Universal-Image-Loader是一个开源的UI组件程序，该项目的目的是提供一个可重复使用的仪器为异步图像加载，缓存和显示。
* 功能特性
* 多线程异步加载和显示图片（图片来源于网络、sd卡、assets文件夹，drawable文件夹（不能加载9patch），新增加载视频缩略图）



* 支持通过“listener”监视加载的过程，可以暂停加载图片，在经常使用的ListView、GridView中，可以设置滑动时暂
* 停加载，停止滑动时加载图片（便于节约流量，在一些优化中可以使用）
* 缓存图片至内存时，可以更加高效的工作
* 高度可定制化（可以根据自己的需求进行各种配置，如：线程池，图片下载器，内存缓存策略等）
* 支持图片的内存缓存，SD卡（文件）缓存
* 在网络速度较慢时，还可以对图片进行加载并设置下载监听
* 流程及原理图



* 导入依赖库：
* 下载jar包放在libs文件夹中
* Maven dependency

<dependency>

<groupId>com.nostra13.universalimageloader</groupId>

<artifactId>universal-image-loader</artifactId>

<version>1.9.3</version>

</dependency>

* Gradle dependency

dependencies {compile **'com.nostra13.universalimageloader:universal-image-loader:1.9.4'**}

* AndroidManifest.xml添加权限

<uses-permission android:name="android.permission.WRITE\_EXTERNAL\_STORAGE" />

<uses-permission android:name="android.permission.INTERNET" />

* 在应用中配置ImageLoaderConfiguration参数（只能配置一次，如多次配置，则默认第一次的配置参数），配置好ImageLoaderConfiguration，一定不要忘记进行初始化操作（一般在application中进行初始化）：
* 使用默认设置

ImageLoaderConfiguration configuration = ImageLoaderConfiguration.*createDefault*(**this**);

* 自己配置参数

*//缓存文件的目录*File cacheDir = StorageUtils.*getOwnCacheDirectory*(context, **"imageloader/Cache"**);  
ImageLoaderConfiguration config = **new** ImageLoaderConfiguration.Builder(context)  
 .memoryCacheExtraOptions(480, 800) *// max width, max height，即保存的每个缓存文件的最大长宽* .threadPoolSize(3) *//线程池内加载的数量* .threadPriority(Thread.***NORM\_PRIORITY*** - 2)  
 .tasksProcessingOrder(QueueProcessingType.***LIFO***)  
 .denyCacheImageMultipleSizesInMemory()  
 .memoryCache(**new** UsingFreqLimitedMemoryCache(2 \* 1024 \* 1024)) *// You can pass your own memory cache implementation/你可以通过自己的内存缓存实现* .memoryCacheSize(2 \* 1024 \* 1024) *// 内存缓存的最大值* .memoryCacheSizePercentage(13) *// default* .diskCacheSize(50 \* 1024 \* 1024) *// 50 Mb sd卡(本地)缓存的最大值* .diskCacheFileCount(100) *// 可以缓存的文件数量  
 // 由原先的discCache -> diskCache* .diskCache(**new** UnlimitedDiskCache(cacheDir))*//自定义缓存路径 new UnlimitedDiscCache(cacheDir)* .diskCacheFileNameGenerator(**new** Md5FileNameGenerator()) *//将保存的时候的URI名称用MD5 加密* .diskCacheFileNameGenerator(**new** HashCodeFileNameGenerator())  
 .imageDownloader(**new** BaseImageDownloader(context, 5 \* 1000, 30 \* 1000)) *// connectTimeout (5 s), readTimeout (30 s)超时时间* .imageDecoder(**new** BaseImageDecoder(**false**)) *// default* .defaultDisplayImageOptions(DisplayImageOptions.*createSimple*()) *// default* .writeDebugLogs() *// Remove for release app* .build();

* 初始化

*//全局初始化此配置*ImageLoader.*getInstance*().init(config);

如下图：



* 显示图片(此处将代码写在Activity中)
* 首先要得到ImageLoader的实例(使用的单例模式)

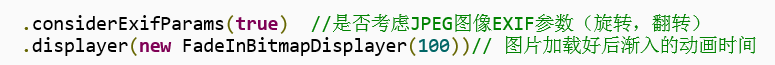
ImageLoader imageLoader = ImageLoader.*getInstance*();

* 相关显示参数配置

*// 使用DisplayImageOptions.Builder()创建DisplayImageOptions*DisplayImageOptions options = **new** DisplayImageOptions.Builder()  
 .showImageOnLoading(R.mipmap.***ic\_launcher***) *// 设置图片下载期间显示的图片* .showImageForEmptyUri(R.mipmap.***ic\_launcher***) *// 设置图片Uri为空或是错误的时候显示的图片* .showImageOnFail(R.mipmap.***ic\_launcher***) *// 设置图片加载或解码过程中发生错误显示的图片* .cacheInMemory(**true**) *// 设置下载的图片是否缓存在内存中* .cacheOnDisk(**true**) *// 设置下载的图片是否缓存在SD卡中* .considerExifParams(**false**) *// default* .imageScaleType(ImageScaleType.***IN\_SAMPLE\_POWER\_OF\_2***) *// default 设置图片以如何的编码方式显示* .bitmapConfig(Bitmap.Config.***ARGB\_8888***) *// default 设置图片的解码类型* .displayer(**new** RoundedBitmapDisplayer(20)) *// 设置成圆角图片* .handler(**new** Handler()) *// default* .build(); *// 构建完成*

注意：

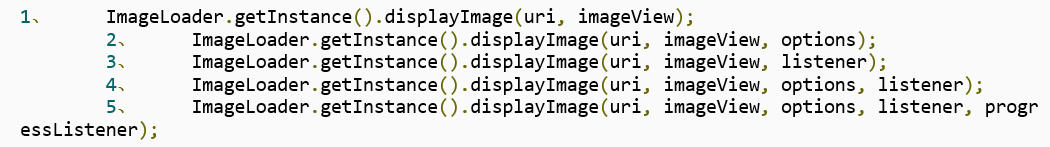




* 调用 displayImage(…)显示图片

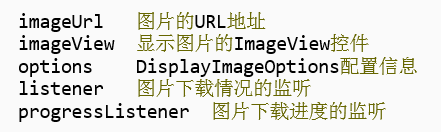
imageLoader.displayImage(  
**"https://cbu01.alicdn.com/img/ibank/2015/974/624/2383426479\_890623294.summ.jpg"**,  
 iv,  
 options);

* 显示图片的5中方法：



*//1.定义要显示的图片的URL和要显示图片的ImageView* ImageLoader.*getInstance*().displayImage(uri, imageView);  
 *//2.加载自定义配置的一个图片* ImageLoader.*getInstance*().displayImage(uri, imageView, options);  
 *//3.加载带监听的一个图片* ImageLoader.*getInstance*().displayImage(uri, imageView, listener);  
 *//4.加载自定义配置且带监听的一个图片* ImageLoader.*getInstance*().displayImage(uri, imageView, options, listener);  
 *//5.加载自定义配置且带监听和进度条的一个图片* ImageLoader.*getInstance*().displayImage(uri, imageView, options, listener, progressListener);

* 参数解析：



* 自定义ImageLoadingListener：

/\*开始\*/

ImageLoader.*getInstance*().displayImage(uri, imageView, options,  
 **new** ImageLoadingListener() {  
 @Override  
 **public void** onLoadingStarted(String arg0, View arg1) {  
 *//开始加载* }  
 @Override  
 **public void** onLoadingFailed(String arg0, View arg1,  
 FailReason arg2) {  
 *//加载失败* }  
 @Override  
 **public void** onLoadingComplete(String arg0, View arg1,  
 Bitmap arg2) {  
 *//加载成功* }  
 @Override  
 **public void** onLoadingCancelled(String arg0, View arg1) {  
 *//加载取消* }  
 }, **new** ImageLoadingProgressListener() {  
 @Override  
 **public void** onProgressUpdate(String imageUri, View view,  
 **int** current, **int** total) {  
 *//加载进度* }  
 });

/\*结束\*/

* 提示和技巧
* 只有在你需要让Image的尺寸比当前设备的尺寸大的时候，你才需要配置maxImageWidthForMemoryCach(...)和 maxImageHeightForMemoryCache(...)这两个参数，比如放大图片的时候。其他情况下，不需要做这些配置，因为默 认的配置会根据屏幕尺寸以最节约内存的方式处理Bitmap。
* 在设置中配置线程池的大小是非常明智的。一个大的线程池会允许多条线程同时工作，但是也会显著的影响到UI 线程的速度。但是可以通过设置一个较低的优先级来解决：当ImageLoader在使用的时候，可以降低它的优先级，这 样UI线程会更加流畅。在使用List的时候，UI 线程经常会不太流畅，所以在你的程序中最好设置threadPoolSize(...)和 threadPriority(...)这两个参数来优化你的应用。
* memoryCache(...)和memoryCacheSize(...)这两个参数会互相覆盖，所以在ImageLoaderConfiguration中使用一个就好了
* diskCacheSize(...)、diskCache(...)和diskCacheFileCount(...)这三个参数会互相覆盖，只使用一个

注：不要使用discCacheSize(...)、discCache(...)和discCacheFileCount(...)这三个参数已经弃用

* 如果你的程序中使用displayImage（）方法时传入的参数经常是一样的，那么一个合理的解决方法是，把这些选项 配置在ImageLoader的设置中作为默认的选项（通过调用defaultDisplayImageOptions(...)方法）。之后调用 displayImage(...)方法的时候就不必再指定这些选项了，如果这些选项没有明确的指定给 defaultDisplayImageOptions(...)方法，那调用的时候将会调用UIL的默认设置。
* 注意事项
* 如果你经常出现oom，你可以尝试:
* 禁用在内存中缓存cacheInMemory(false)，如果oom仍然发生那么似乎你的应用程序有内存泄漏，使用MemoryAnalyzer来检测它。否则尝试以下步骤(尝试所有或几个)
* 减少配置的线程池的大小(.threadPoolSize(...))，建议1~5
* 在显示选项中使用 .bitmapConfig(Bitmap.Config.RGB\_565) . RGB\_565模式消耗的内存比ARGB\_8888模式少两倍.
* 配置中使用.diskCacheExtraOptions(480, 320, null)
* 配置中使用 .memoryCache(newWeakMemoryCache()) 或者完全禁用在内存中缓存(don't call .cacheInMemory()).
* 在显示选项中使用.imageScaleType(ImageScaleType.EXACTLY) 或 .imageScaleType(ImageScaleType.IN\_SAMPLE\_INT)
* 7)避免使用 RoundedBitmapDisplayer. 调用的时候它使用ARGB-8888模式创建了一个新的Bitmap对象来显示，对于内存缓存模式 (ImageLoaderConfiguration.memoryCache(...)) 你可以使用已经实现好的方法.
* ImageLoader是根据ImageView的height，width确定图片的宽高
* 一定要对ImageLoaderConfiguration进行初始化,否则会报错
* 开启缓存后默认会缓存到外置SD卡如下地址(/sdcard/Android/data/[package\_name]/cache).如果外置SD卡不存在，会缓存到手机. 缓存到Sd卡需要在AndroidManifest.xml文件中进行如下配置

<uses-permission android:name="android.permission.WRITE\_EXTERNAL\_STORAGE" />

* 内存缓存模式可以使用以下已实现的方法 (ImageLoaderConfiguration.memoryCache(...))
* 缓存只使用强引用

LruMemoryCache (缓存大小超过指定值时，删除最近最少使用的bitmap) --默认情况下使用

* 缓存使用弱引用和强引用

UsingFreqLimitedMemoryCache (缓存大小超过指定值时,删除最少使的bitmap)

LRULimitedMemoryCache (缓存大小超过指定值时,删除最近最少使用的<span style="font-family: 'Helvetica Neue', Helvetica, 'Segoe UI', Arial, freesans, sans-serif;">bitmap) --默认值</span>

FIFOLimitedMemoryCache (缓存大小超过指定值时,按先进先出规则删除的<span style="font-family: 'Helvetica Neue', Helvetica, 'Segoe UI', Arial, freesans, sans-serif;">bitmap)</span>

LargestLimitedMemoryCache (缓存大小超过指定值时,删除最大的bitmap)

LimitedAgeMemoryCache (缓存对象超过定义的时间后删除)

* 缓存使用弱引用

WeakMemoryCache（没有限制缓存）

* 本地缓存模式可以使用以下已实现的方法 (ImageLoaderConfiguration.diskCache(...)) ：

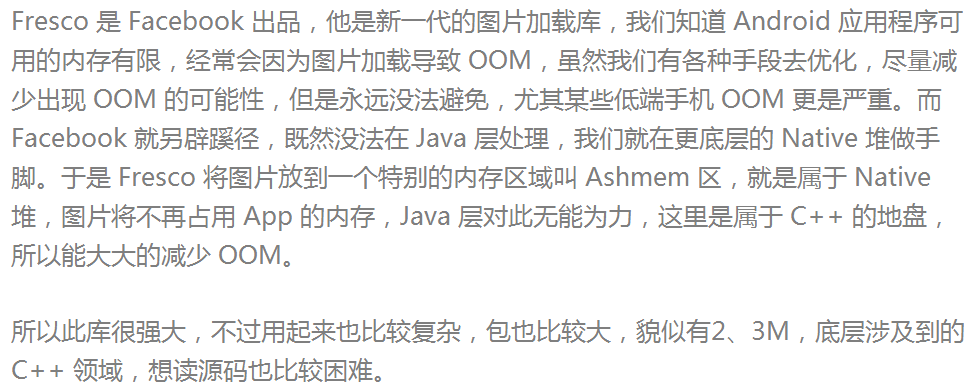
UnlimitedDiskCache 不限制缓存大小（默认）TotalSizeLimitedDiskCache (设置总缓存大小，超过时删除最久之前的缓存)

FileCountLimitedDiskCache (设置总缓存文件数量，当到达警戒值时，删除最久之前的缓存。如果文件的大小都一样的时候，可以使用该模式)

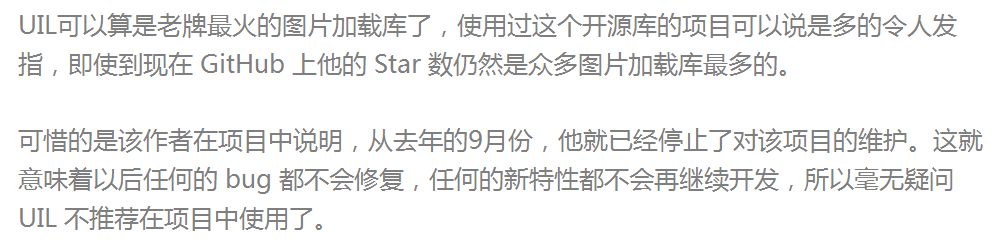
LimitedAgeDiskCache (不限制缓存大小，但是设置缓存时间，到期后删除)

**3 总结**

* **Facebook的Fresco图片框架，需要连同自定义控件一起使用，功能不是一般的强大，但是对于已经在维护的项目修改成本那可不是一般的高。**



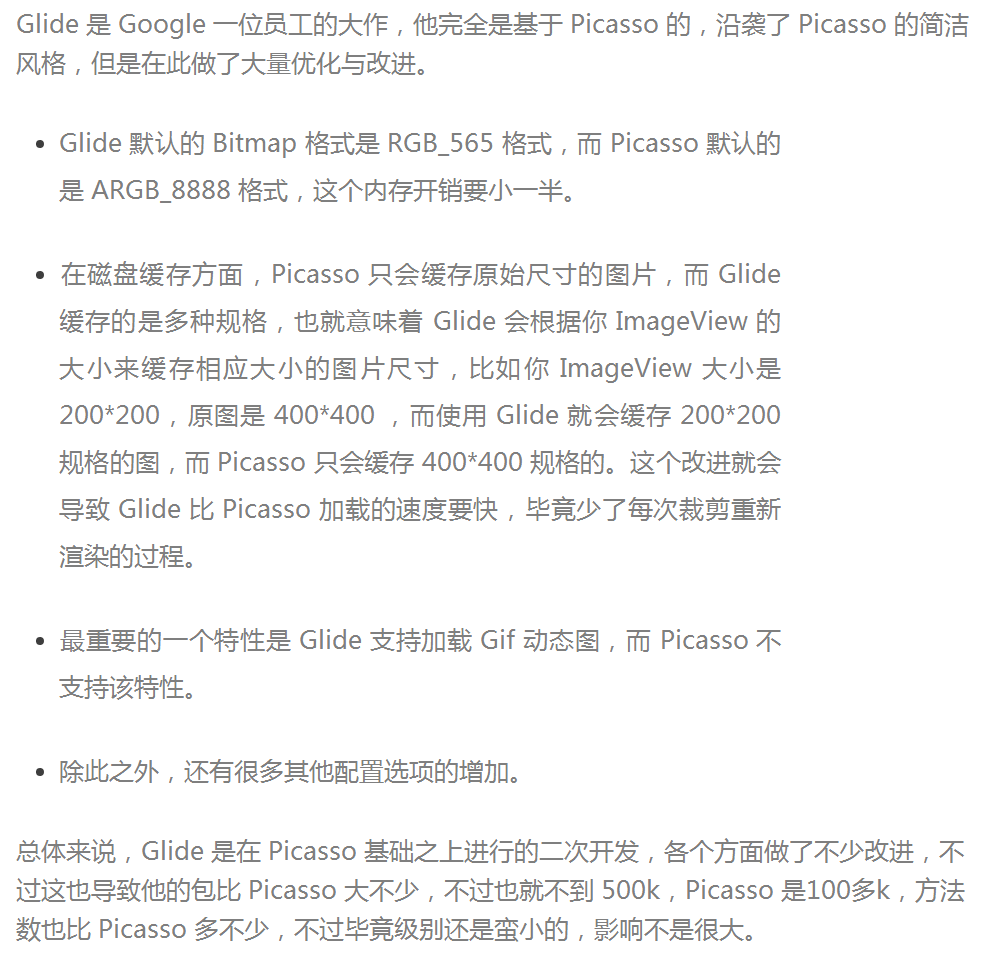
* **Universal Image Loader：一个强大的图片加载库，包含各种各样的配置，最老牌，使用也最广泛。**



* **Picasso: Square出品，必属精品。和OkHttp搭配起来更配呦！**



* **Volley ImageLoader：Google官方出品，可惜不能加载本地图片~**
* **Glide：Google推荐的图片加载库，专注于流畅的滚动。**



综合来看，毫无疑问 Glide 与 Picasso 之间优先推荐选择 Glide，尤其是如果你的项目想要支持 Gif 动态图，那更该选择 Glide 。

但是如果你的项目使用了 Square 公司的全家桶，如 Retrofit 或者 OkHttp ，那么搭配 Picasso 一起使用也不是不可，兼容性可能会更好些，占用体积也会少些。

对于一般的 App 使用 Fresco 未免有些大材小用了，大部分情况 Glide 都能满足你的需求了，但是如果你的 App 中大量使用图片，比如是类似 Instagram 一类的图片社交 App ，那么推荐使用 Fresco ，虽然稍复杂，但是还是推荐使用 Fresco ，对提升你 App 的性能与体验有不少帮助，值得花时间去研究并应用到自己的 App 上来。