# Glide

<http://www.aichengxu.com/java/71733.htm>

Glide基本用法

//设置默认和出错时的图片

Glide.with(this).load(url).placeholder(resId).error(resId).into(mImageView)

//普通的图片加载

Glide.with(this).load(url).into(mImageView);

//可理解为加载动态图的第一帧的Bitmap,比如Gif

Glide.with(this).load(url).asBitmap().into(imageView);

//GIF加载，URL指向的资源必须是gif，如果是普通图片则不显示。

//相反，如果指向正确但没有执行asGif方法，则只是作为普通图片展示

Glide.with(this).asGif().load(url).into(mImageView)；

//缩略图的加载

Glide.with(yourFragment).load(yourUrl).thumbnail(0.1f).into(yourView)

核心：

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内存的分配以重用来提高性能。图片的加载任务会与activity或者Fragment的生命周期绑定，当界面执行onStop的使用自动暂定，而当执行onStart的时候又会自动重新开启，同样的，**动态Gif图的加载**也是如此，以用来节省电量，同时Glide会对网络状态做监听，当网络状态发生改变时，会重启失败的任务，以减少任务因网络连接问题而失败的概率。

默认使用RGB\_565的Bitmap，可以局部或者全局配置为ARGB\_8888

默认使用HttpUrlConnection下载图片，可以配置为OkHttp或者Volley下载，也可以自定义下载方式。

Glide资源层次划分

* Model: 原始资源，比如Url，AndroidResourceId, File等
* Data: 中间资源，比如Stream，ParcelFileDescriptor（ContentProvider共享文件时比较常用，其实就是操作系统的文件描述符的封装，里面有in out err三个取值。也有人说是链接建立好之后的socket句柄。）等
* Resource：直接使用的资源，包括Bitmap，Drawable等

Glide尽量的复用资源来防止不必要的GC\_FOR\_ALLOC引起卡顿

最显著的内存复用就是内存LruResourceCache（第一次从网络或者磁盘上读取到Resource时，并不会保存到LruCache当中，当Resource被release时，也就是View不在需要此Resource时，才会进入LruCache当中）

还有BitmapPool（Glide会尽量用图片池来获取到可以复用的图片，获取不到才会new，而当LruCache触发Evicted时会把从LruCache中淘汰下来的Bitmap回收，也会把transform时用到的中间Bitmap加以复用及回收）

Glide库图片池

4.4以前是Bitmap复用必须长宽相等才可以复用

4.4及以后是Size>=所需就可以复用，只不过需要调用reconfigure来调整尺寸

Glide用AttributeStategy和SizeStrategy来实现两种策略

图片池在收到传来的Bitmap之后，通过长宽或者Size来从KeyPool中获取Key(对象复用到了极致，连Key都用到了Pool)，然后再每个Key对应一个双向链表结构来存储。每个Key下可能有很多个待用Bitmap

取出后要减少图片池中记录的当前Size等，并对Bitmap进行eraseColor(Color.TRANSPAENT)操作确保可用

Glide加载发起流程

1. Glide.with(context)创建RequestManager
   * RequestManager负责管理当前context的所有Request
   * Context可以传Fragment、Activity或者其他Context，当传Fragment、Activity时，当前页面对应的Activity的生命周期可以被RequestManager监控到，从而可以控制Request的pause、resume、clear。这其中采用的监控方法就是在当前activity中添加一个没有view的fragment，这样在activity发生onStart onStop onDestroy的时候，会触发此fragment的onStart onStop onDestroy。
   * RequestManager用来跟踪众多当前页面的Request的是RequestTracker类，用弱引用来保存运行中的Request，用强引用来保存暂停需要恢复的Request。
2. Glide.with(context).load(url)创建需要的Request
   * 通常是DrawableTypeRequest，后面可以添加transform、fitCenter、animate、placeholder、error、override、skipMemoryCache、signature等等
   * 如果需要进行Resource的转化比如转化为Byte数组等需要，可以加asBitmap来更改为BitmapTypeRequest
   * Request是Glide加载图片的执行单位
3. Glide.with(context).load(url).into(imageview)
   * 在Request的into方法中会调用Request的begin方法开始执行
   * 在正式生成EngineJob放入Engine中执行之前，如果并没有事先调用override(width, height)来指定所需要宽高，Glide则会尝试去获取imageview的宽和高，如果当前imageview并没有初始化完毕取不到高宽，Glide会通过view的ViewTreeObserver来等View初始化完毕之后再获取宽高再进行下一步

Glide加载资源

* GlideBuilder在初始化Glide时，会生成一个执行机Engine
* Engine中包含LruCache缓存及一个当前正在使用的active资源Cache（弱引用）
* activeCache辅助LruCache，当Resource从LruCache中取出使用时，会从LruCache中remove并进入acticeCache当中
* Cache优先级LruCache>activeCache
* Engine在初始化时要传入两个ExecutorService，即会有两个线程池，一个用来从DiskCache获取resource，另一个用来从Source中获取（通常是下载）
* 线程的封装单位是EngineJob，有两个顺序状态，先是CacheState，在此状态先进入DiskCacheService中执行获取，如果没找到则进入SourceState，进到SourceService中执行下载

Glide的Target

* 负责图片加载的回调

优缺点

* 代码扩展性极强，4.0版本更加如此，但来的问题就是过于复杂，不太便于阅读
* 但仍会遇到有些需求Glide无法满足
* 设置加载图片的最大宽高
* PlaceHolder的图片形状不与加载的Bitmap相同会产生的抖动问题
* 无法指定删除某一个图片缓存的问题（可以用加signature的方式试其失效并重新下载，但不可以删除）

# Picasso

<https://www.2cto.com/kf/201511/451087.html>

特点：

绝对是最轻量的图片加载库,120kb. 自带监控功能，可以检测cache hit/内存大小等等数据 **图片预加载** 线程并发数依网络状态变化而变化、优先级调度 图片变换 图片压缩、自适应 易扩展

使用：

Picasso.with(context)

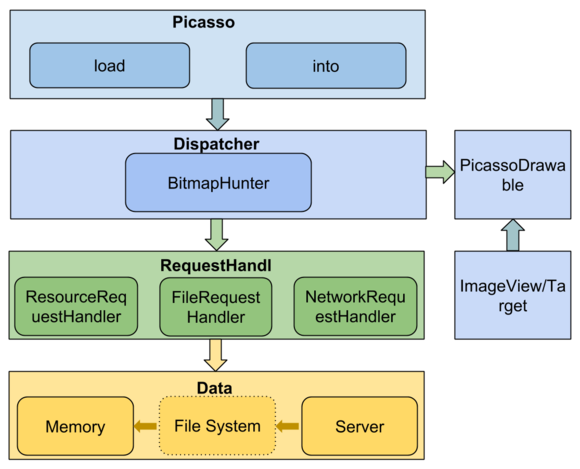
    .load(url)

    .placeholder(R.drawable.user\_placeholder)

    .error(R.drawable.user\_placeholder\_error)

    .into(imageView);//此种策略并不会压缩图片

原理：



Picasso类是一个负责图片下载、变换、缓存的管理器,当它收到一个图片下载请求的时候，它会创建Request并提交给Dispatcher,  
Dispatcher会寻找对应的处理器RequestHandler,并将请求与该处理器一起提交给线程池执行,图片获取成功后，最终会交给  
PicassoDrawable显示到Target上。

Picasso.with用**单例模式**创建Picasso实例（里面用build方法最终完成创建）

内存缓存基于LruCache,磁盘缓存基于http缓存,HttpResponseCache 创建默认的下载器 创建默认的线程池(3个worker线程) 创建默认的Transformer,这个Transformer什么事情也不干，只负责转发请求 创建默认的监控器(Stats),用于统计缓存命中率、下载时长等等 创建默认的处理器集合,即RequestHandlers.它们分别会处理不同的加载请求

处理器集合的初始化在Picasso的构造器中，处理器包括从网络、file、assert、contactsphoto等地方加载图片

load()方法用于从不同地方加载图片，比如网络、resource、File等，该方法内部逻辑很简单，只是创建了一个RequestCreator

RequestCreator从名字就可以知道这是一个封装请求的类,请求在Picasso中被抽象成Request。RequestCreator类提供了诸如placeholder、tag、error、memoryPolicy、networkPolicy等方法.由于可配置项太多，所以Request也使用了**Builder**模式

into方法会去将Request创建，并丢到线程池或者分发器中执行。into方法有多种重载，因为Picasso不仅仅可以将图片加载到ImageView上，还可以加载到Target或者RemoteView上。into方法会先从缓存里面查找图片，如果找不到的话，则会创建Action即一个加载任务，交给Dispatcher执行。

Request关注的是请求本身，比如请求的源、id、开始时间、图片变换配置、优先级等等，而Action则代表的是一个加载任务，所以不仅需要Request对象的引用，还需要Picasso实例，是否重试加载等等

Action持有的是Target(比如ImageView..)的弱引用，这样可以保证加载时间很长的情况下  
也不会影响到Target的回收了.

Action会传入enqueueAndSubmit方法，它会先从action任务上拿到对应target，也就是imageView，然后从weakHashMap中通过这个imageView索引到对应的action，如果发现这个action跟传进来的action不一样的话，那就取消掉之前的加载任务。最后将当前加载任务提交.（最终调用的是Dispatcher的dispatchSubmit(action)）

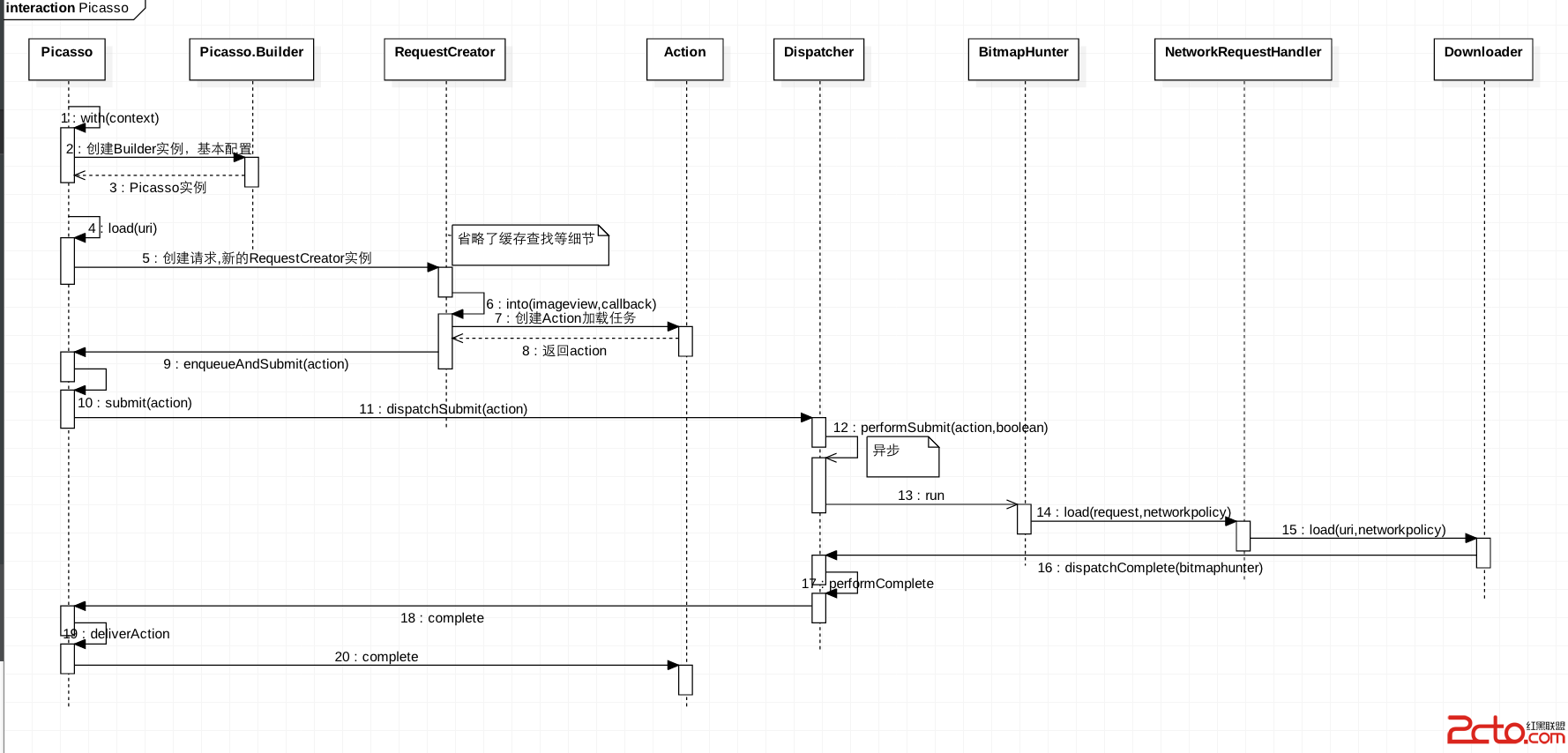
（插曲）每一个Dispatcher都需要关联线程池(service)、下载器(downloader)、主线程的Handler(HANDLER)、缓存(cache)、监控器(stats).这里先看线程池,Picasso默认的线程池叫PicassoExecutorService,它继承自ThreadPoolExecutor,默认线程数量为3.但是PicassoExecutorService的特性是可以根据网络情况调整线程数量，wifi下是4个线程，而2g网只有一个线程。具体是通过在Dispatcher中注册了监听网络变化的广播接收者。另外，PicassoExecutorService中还有一个很重要的方法叫submit,它会去执行一个runnable.

dispatchSubmit会将其封装丢进线程池，如果缓存中有就直接用，否则接下来调用RequestHandler的load方法（NetworkRequestHandler网络handler）

load方法传入了downloader，创建Downloader时先反射下，看有没有依赖okhttp，如果依赖的话，那就使用OkHttpClient喽，否则就使用默认的HttpUrlConnection了。  
注:其实从4.4开始，okhttp已经作为HttpUrlConnection的实现引擎了。

之后会将结果回调给dispatcher，dispatcher执行dispatchcomplete方法，首先会根据事先设置的缓存策略决定是否将结果加到内存缓存。然后调用batch方法，从名字就可以知道，这个方法会把结果暂存，  
然后批量处理(等待200ms)，这样做也是为了防止短时间大量任务阻塞消息队列。到时间后，就会执行performBatchComplete,此方法会将这个批次的所有结果一次性发给主线程的Handler，也就是Picasso中定义的Handler

主线程收到消息后会进行处理，对batch中每个BitmapHunter调用complete方法,而complete方法会调用deliverAction方法,最终其实调用的是具体action的complete方法，如果是ImageView的话，那就是ImageViewAction的complete方法



简单总结下，当我们执行Picasso.with(context).load(url).into(imageview)时，首先会构造Picasso实例，然后会  
根据url创建请求，然后请求会被交给Dispatcher,Dispatcher将在子线程对请求任务进行调度，将请求任务交给线程池  
执行，执行完毕后，将结果传给主线程的handler，最后在主线程中将图片设置到ImageView上.

关于缓存策略

Picasso的缓存是内存缓存+磁盘缓存，内存缓存基于LruCache类，可配置替换。磁盘缓存依赖于http缓存，不可配置。  
先看内存缓存.内存缓存比较简单，是通过LinkedHashMap实现.  
读缓存时机:生成了请求Request对象，准备创建Action加载任务之前，会先去缓存里面查找下

写缓存时机:图片从网络或者其他地方加载成功后，即在BitmapHunter的run方法执行结束的时候.  
Dispatcher#performComplete

关于磁盘缓存。  
如果你是使用UrlConnectionDownloader的话,那很不幸，缓存只在Api>14上生效，因为缓存依赖于HttpResponseCache.  
如果你依赖了okhttp，那么缓存策略始终是有效的。另外需要说明的是，既然是http缓存，那么缓存的可用性依赖于http响应是  
否允许缓存，也就是说得看响应中是否携带Cache-Control、Expires等字段.

关于压缩

图片压缩的原理通常都是利用BitmapFactory#Options类，先将injustDecodeBounds设置为true,对Bitmap进行一次解码，拿到outWidth/outHeight，即实际宽高,然后根据期望压缩到的宽和高算出inSampleSize,最后将injustDecodeBounds设置为false，再对Bitmap进行一次解码即可。另一种压缩的方法是设置图片的显示效果,比如ARGB\_8888等等.Picasso综合了利用这两种方案.