Android Glide 的

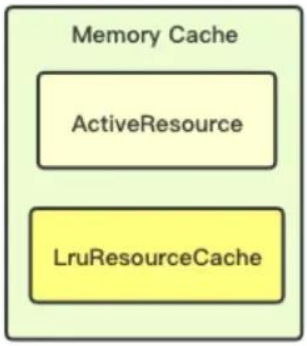
deepwaterooo

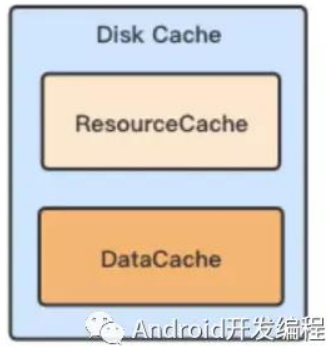
September 15, 2022

Contents

1 一、Glide 中缓存概念简述

- Glide 将它分成了两个模块，一个是内存缓存，一个是硬盘缓存;





1.1 1、内存缓存

- 内存缓存又分为两级，一级是 LruCache 缓存，一级是弱引用缓存
- 内存缓存的作用：防止应用重复将图片数据读取到内存当中。
  - LruCache 缓存：不在使用中的图片使用 LruCache 来进行缓存。
  - 弱引用缓存：把正在使用中的图片使用弱引用来进行缓存，这样的目的保护正在使用的资源不会被 LruCache 算法回收。

1.2 2、硬盘缓存

- 硬盘缓存的作用：防止应用重复从网络或其他地方重复下载和读取数据;

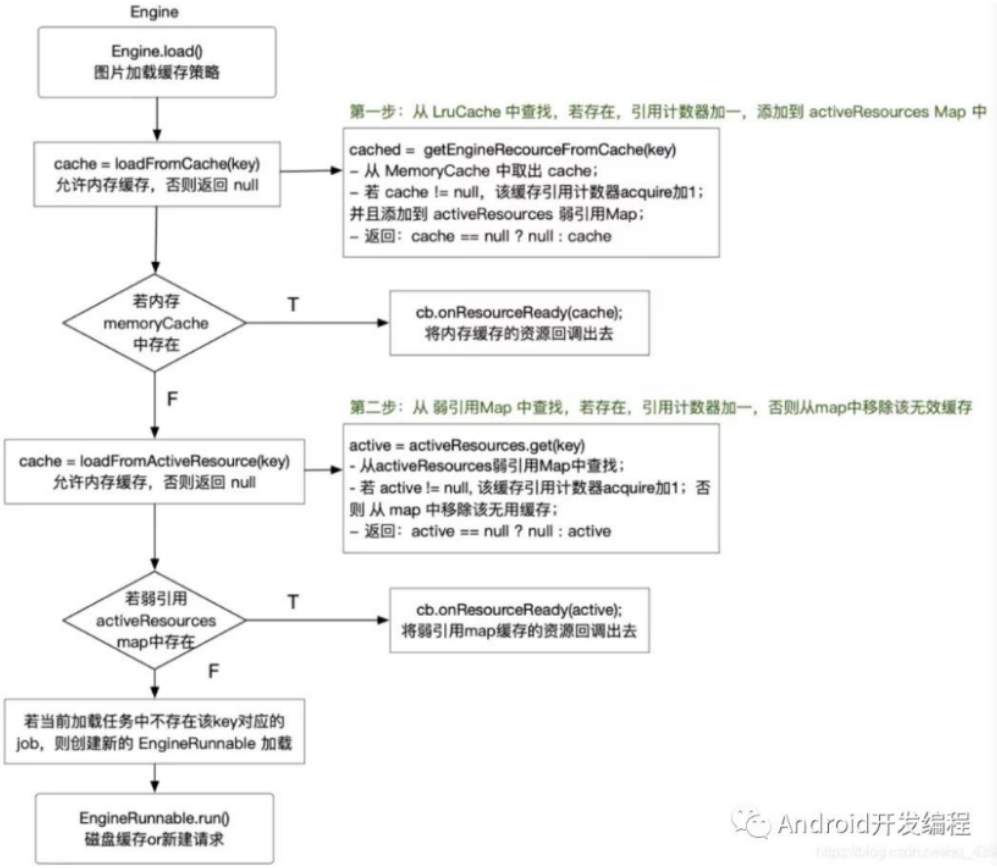
1.3 3、图片请求步骤

- 开始一个新的图片请求之前检查以下多级的缓存：
  - 内存缓存：该图片是否最近被加载过并仍存在于内存中? 即 LruCache 缓存;
  - 活动资源：现在是否有另一个 View 正在展示这张图片? 也就是弱引用缓存;
  - 资源类型：该图片是否之前曾被解码、转换并写入过磁盘缓存?
  - 数据来源：构建这个图片的资源是否之前曾被写入过文件缓存?
- 前两步检查图片是否在内存中，如果是则直接返回图片。后两步则检查图片是否在磁盘上，以便快速但异步地返回图片;
- 如果四个步骤都未能找到图片，则 Glide 会返回到原始资源以取回数据 (原始文件，Uri, Url 等);
- 图片存的顺序是：弱引用、内存、磁盘;
- 图片取的顺序是：内存、弱引用、磁盘。

1.4 4、Glide 中 Bitmap 复用机制

- Bitmap 复用机制：将已经不需要使用的数据空间重新拿来使用，减少内存抖动 (指在短时间内有大量的对象被创建或者被回收的现象);
- BitmapFactory.Options.inMutable 是 Glide 能够复用 Bitmap 的基石，是 BitmapFactory 提供的一个参数，表示该 Bitmap 是可变的，支持复用的。BitmapFactory.Options 中提供了两个属性：inMutable、inBitmap。当进行 Bitmap 复用时，需要设置 inMutable 为 true，inBitmap 设置被复用的已经存在的 Bitmap。Bitmap 复用池使用 LRU 算法实现。

2 二、缓存源码流程



- memory cache 和 disk cache 在 Glide 创建的时候也被创建了，Glide 创建的代码在 GlideBuilder.build(Context) 方法。

```
@NonNull
Glide build(@NonNull Context context) {
    if (memoryCache == null) {
        memoryCache = new LruResourceCache(memorySizeCalculator.getMemoryCacheSize());
    }
    if (diskCacheFactory == null) {
        diskCacheFactory = new InternalCacheDiskCacheFactory(context);
    }
    if (engine == null) {
        engine = new Engine(
                memoryCache,
                diskCacheFactory,
                );
    }
    return new Glide(
        memoryCache,
        );
}
```

2.1 1、内存缓存-memoryCache

- 通过代码可以看到 memoryCache 被放入 Engine 和 Glide 实例中。在 Engine 中利用 memoryCache 进行存取操作，Glide 实例中的 memoryCache 是用来在内存紧张的时候，通知 memoryCache 释放内存。Glide 实现了 ComponentCallbacks2 接口，在 Glide 创建完成后，通过 applicationContext.registerComponentCallbacks(glide) 似的 Glide 实例可以监听内存紧张的信号。

```
// Glide
@Override
public void onTrimMemory(int level) {
    trimMemory(level);
}
public void trimMemory(int level) {
    // Engine asserts this anyway when removing resources, fail faster and consistently
    Util.assertMainThread();
    // memory cache needs to be trimmed before bitmap pool to trim re-pooled Bitmaps too. See #687.
    memoryCache.trimMemory(level);
    bitmapPool.trimMemory(level);
    arrayPool.trimMemory(level);
}
```

- memoryCache 是一个使用 LRU(least recently used) 算法实现的内存缓存类 LruResourceCache，继承至 LruCache 类，并实现了 MemoryCache 接口。LruCache 定义了 LRU 算法实现相关的操作，而 MemoryCache 定义的是内存缓存相关的操作。
- LruCache 的实现是利用了 LinkedHashMap 的这种数据结构的一个特性 (accessOrder=true 基于访问顺序) 再加上对 LinkedHashMap 的数据操作上锁实现的缓存策略。
  - 当调用 put() 方法时，就会在集合中添加元素，并调用 trimToSize() 判断缓存是否已满，如果满了就用 LinkedHashMap 的迭代器删除队尾元素，即近期最少访问的元素。
  - 当调用 get() 方法访问缓存对象时，就会调用 LinkedHashMap 的 get() 方法获得对应集合元素，同时会更新该元素到队头。

2.2 2、磁盘缓存

- diskCacheFactory 是创建 DiskCache 的 Factory，DiskCache 接口定义。

```
public interface DiskCache {
    interface Factory {
        /** 250 MB of cache. */
        int DEFAULT_DISK_CACHE_SIZE = 250 * 1024 * 1024;
        String DEFAULT_DISK_CACHE_DIR = "image_manager_disk_cache";
        @Nullable
        DiskCache build();
    }
    interface Writer {
        boolean write(@NonNull File file);
    }
    @Nullable
    File get(Key key);
    void put(Key key, Writer writer);
    @SuppressWarnings("unused")
    void delete(Key key);
    void clear();
}
```

- 接着再来看下 DiskCache.Factory 的默认实现：InternalCacheDiskCacheFactory。

```
public final class InternalCacheDiskCacheFactory extends DiskLruCacheFactory {
    public InternalCacheDiskCacheFactory(Context context) {
        this(context,
             DiskCache.Factory.DEFAULT_DISK_CACHE_DIR,
             DiskCache.Factory.DEFAULT_DISK_CACHE_SIZE);
    }
    public InternalCacheDiskCacheFactory(Context context, long diskCacheSize) {
        this(context,
             DiskCache.Factory.DEFAULT_DISK_CACHE_DIR,
             diskCacheSize);
    }
    public InternalCacheDiskCacheFactory(final Context context,
                                         final String diskCacheName,
                                         long diskCacheSize) {
        super(new CacheDirectoryGetter() {
                @Override
                public File getCacheDirectory() {
                    File cacheDirectory = context.getCacheDir();
                    if (cacheDirectory == null) {
                        return null;
```

```
                    }
                    if (diskCacheName != null) {
                        return new File(cacheDirectory, diskCacheName);
                    }
                    return cacheDirectory;
                }
            }, diskCacheSize);
    }
}
```

- 由以上代码可以看出:默认会创建一个 250M 的缓存目录,其路径为/data/data/{package}/cache/imag
- 继续看其父类 DiskLruCacheFactory 的代码:

```
public class DiskLruCacheFactory implements DiskCache.Factory {

    private final long diskCacheSize;
    private final CacheDirectoryGetter cacheDirectoryGetter;

    public interface CacheDirectoryGetter {
        File getCacheDirectory();
    }

    public DiskLruCacheFactory(CacheDirectoryGetter cacheDirectoryGetter, long diskCacheSize) {
        this.diskCacheSize = diskCacheSize;
        this.cacheDirectoryGetter = cacheDirectoryGetter;
    }
    @Override public DiskCache build() {
        File cacheDir = cacheDirectoryGetter.getCacheDirectory();
        if (cacheDir == null) {
            return null;
        }
        if (!cacheDir.mkdirs() && (!cacheDir.exists() || !cacheDir.isDirectory())) {
            return null;
        }
        return DiskLruCacheWrapper.create(cacheDir, diskCacheSize);
    }
}
```

- DiskLruCacheFactory.build() 方法会返回一个 DiskLruCacheWrapper 类的实例,看下 DiskLruCacheWrapper 的实现。

```
public class DiskLruCacheWrapper implements DiskCache {
    private static final String TAG = "DiskLruCacheWrapper";
    private static final int APP_VERSION = 1;
    private static final int VALUE_COUNT = 1;

    private static DiskLruCacheWrapper wrapper;
    private final SafeKeyGenerator safeKeyGenerator;

    private final File directory;
    private final long maxSize;

    private final DiskCacheWriteLocker writeLocker = new DiskCacheWriteLocker();
    private DiskLruCache diskLruCache;

    @SuppressWarnings("deprecation")
        public static DiskCache create(File directory, long maxSize) {
        return new DiskLruCacheWrapper(directory, maxSize);
    }
    @Deprecated
        @SuppressWarnings({"WeakerAccess", "DeprecatedIsStillUsed"})
        protected DiskLruCacheWrapper(File directory, long maxSize) {
        this.directory = directory;
        this.maxSize = maxSize;
        this.safeKeyGenerator = new SafeKeyGenerator();
    }
// 文件的读写是需要 throw IOException 的; 为什么要上锁呢？同时会有多个客户端多个线程需要拿到这个磁盘文件吗？
    private synchronized DiskLruCache getDiskCache() throws IOException {
        if (diskLruCache == null)
            diskLruCache = DiskLruCache.open(directory, APP_VERSION, VALUE_COUNT, maxSize);
        return diskLruCache;
    }
    @Override
        public File get(Key key) {
        String safeKey = safeKeyGenerator.getSafeKey(key);
```

```
        File result = null;
        try {
            final DiskLruCache.Value value = getDiskCache().get(safeKey);
            if (value != null) {
                result = value.getFile(0);
            }
        } catch (IOException e) {
        }
        return result;
    }
    @Override
        public void put(Key key, Writer writer) {
        String safeKey = safeKeyGenerator.getSafeKey(key);
        writeLocker.acquire(safeKey); // 写，要上锁
        try {
            try {
                DiskLruCache diskCache = getDiskCache();
                Value current = diskCache.get(safeKey);
                DiskLruCache.Editor editor = diskCache.edit(safeKey);
                try {
                    File file = editor.getFile(0);
                    if (writer.write(file)) {
                        editor.commit();
                    }
                } finally {
                    editor.abortUnlessCommitted();
                }
            } catch (IOException e) {
            }
        } finally {
            writeLocker.release(safeKey); // <<<<<<<<<<
        }
    }
}
```

- 里面包装了一个 DiskLruCache，该类主要是为 DiskLruCache 提供了一个根据 Key 生成 safeKey 的 SafeKeyGenerator 以及写锁 DiskCacheWriteLocker。
- 回到 GlideBuilder.build(Context) 中，diskCacheFactory 会被传进 Engine 中，在 Engine 的构造方法中会被包装成为一个 LazyDiskCacheProvider，在被需要的时候调用 getDiskCache() 方法，这样就会调用 factory 的 build() 方法返回一个 DiskCache。代码如下：

```
private static class LazyDiskCacheProvider implements DecodeJob.DiskCacheProvider {

    private final DiskCache.Factory factory;
    private volatile DiskCache diskCache;

    LazyDiskCacheProvider(DiskCache.Factory factory) {
        this.factory = factory;
    }

    @Override public DiskCache getDiskCache() {
        if (diskCache == null) {
            synchronized (this) {
                if (diskCache == null) {
                    diskCache = factory.build();
                }
                if (diskCache == null) {
                    diskCache = new DiskCacheAdapter();
                }
            }
        }
        return diskCache;
    }
}
```

- LazyDiskCacheProvider 会在 Engine 后面的初始化流程中作为入参传到 DecodeJobFactory 的构造器。在 DecodeJobFactory 创建 DecodeJob 时也会作为入参会传进去，DecodeJob 中会以全局变量保存此 LazyDiskCacheProvider，在资源加载完毕并展示后，会进行缓存的存储。同时，DecodeJob 也会在 DecodeHelper 初始化时，将此 DiskCacheProvider 设置进去，供 ResourceCacheGenerator、DataCacheGenerator 读取缓存，供 SourceGenerator 写入缓存。

2.3 3、ActiveResources

- ActiveResources 在 Engine 的构造器中被创建，在 ActiveResources 的构造器中会启动一个后台优先级级别 (THREAD_PRIORITY_BACKGROUND) 的线程，在该线程中会调用 cleanReferenceQueue() 方法一直循环清除 ReferenceQueue 中的将要被 GC 的 Resource。

```
final class ActiveResources {
    private final boolean isActiveResourceRetentionAllowed;
    private final Executor monitorClearedResourcesExecutor;

    @VisibleForTesting
    final Map<Key, ResourceWeakReference> activeEngineResources = new HashMap<>();
    private final ReferenceQueue<EngineResource<?>> resourceReferenceQueue = new ReferenceQueue<>();
    private volatile boolean isShutdown;

    ActiveResources(boolean isActiveResourceRetentionAllowed) {
        this(isActiveResourceRetentionAllowed,
             java.util.concurrent.Executors.newSingleThreadExecutor(
                 new ThreadFactory() {
                     @Override
                     public Thread newThread(@NonNull final Runnable r) {
                         return new Thread(new Runnable() {
                                 @Override
                                 public void run() {
                                     Process.setThreadPriority(Process.THREAD_PRIORITY_BACKGROUND); // 后台线程
                                     r.run();
                                 }
                             },
                             "glide-active-resources");
                     }
                 }));
    }
    @VisibleForTesting
    ActiveResources(boolean isActiveResourceRetentionAllowed, Executor monitorClearedResourcesExecutor) {
        this.isActiveResourceRetentionAllowed = isActiveResourceRetentionAllowed;
        this.monitorClearedResourcesExecutor = monitorClearedResourcesExecutor;
        monitorClearedResourcesExecutor.execute(
            new Runnable() {
                @Override
                public void run() {
                    cleanReferenceQueue();
                }
            });
    }

    @SuppressWarnings("WeakerAccess")
    @Synthetic void cleanReferenceQueue() {
        while (!isShutdown) {
            try {
                ResourceWeakReference ref = (ResourceWeakReference) resourceReferenceQueue.remove();
                cleanupActiveReference(ref);
                // This section for testing only.
                DequeuedResourceCallback current = cb;
                if (current != null) {
                    current.onResourceDequeued();
                }
                // End for testing only.
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }
}
```

- 先来看看 ActiveResources 的 activate 方法 (保存)、deactivate 方法 (删除) 的方法。

```
synchronized void activate(Key key, EngineResource<?> resource) {
    ResourceWeakReference toPut = new ResourceWeakReference(
        key, resource, resourceReferenceQueue, isActiveResourceRetentionAllowed);
    ResourceWeakReference removed = activeEngineResources.put(key, toPut);
    if (removed != null) {
        removed.reset();
    }
}
synchronized void deactivate(Key key) {
    ResourceWeakReference removed = activeEngineResources.remove(key);
    if (removed != null) {
```

```
        removed.reset();
    }
}
```

- activate 方法会将参数封装成为一个 ResourceWeakReference，然后放入 map 中，如果对应的 key 之前有值，那么调用之前值的 reset 方法进行清除。deactivate 方法先在 map 中移除，然后调用 resource 的 reset 方法进行清除。ResourceWeakReference 继承 WeakReference，内部只是保存了 Resource 的一些属性。

```
static final class ResourceWeakReference extends WeakReference<EngineResource<?>> {
    @SuppressWarnings("WeakerAccess") @Synthetic final Key key;
    @SuppressWarnings("WeakerAccess") @Synthetic final boolean isCacheable;
    @Nullable @SuppressWarnings("WeakerAccess") @Synthetic Resource<?> resource;

    @Synthetic
        @SuppressWarnings("WeakerAccess")
        ResourceWeakReference(
            @NonNull Key key,
            @NonNull EngineResource<?> referent,
            @NonNull ReferenceQueue<? super EngineResource<?>> queue,
            boolean isActiveResourceRetentionAllowed) {
        super(referent, queue);
        this.key = Preconditions.checkNotNull(key);
        this.resource =
            referent.isCacheable() && isActiveResourceRetentionAllowed
            ? Preconditions.checkNotNull(referent.getResource()) : null;
        isCacheable = referent.isCacheable();
    }
}
```

- 构造方法中调用了 super(referent, queue)，这样做可以让将要被 GC 的对象放入到 ReferenceQueue 中。而 ActiveResources.cleanReferenceQueue() 方法会一直尝试从 queue 中获取将要被 GC 的 resource，然后调用 cleanupActiveReference 方法将 resource 从 activeEngineResources 中移除。cleanupActiveReference 源码如下：

```
void cleanupActiveReference(@NonNull ResourceWeakReference ref) {
    synchronized (listener) { // listener 所采用的是什么数据结构呢？猜测仍是多线程不安全的结构
        synchronized (this) {
            // 移除 active 资源
// activeEngineResources: 是一个线程不安全的 HashMap; 对它时行操作删除操作就得上锁 (读大概不必上锁)
            activeEngineResources.remove(ref.key);
            if (!ref.isCacheable || ref.resource == null) { // 检查一下用户的配置：如果不需要缓存，就到此结束了
                return;
            }
            // 构造新的 Resource
            EngineResource<?> newResource =
                new EngineResource<>(ref.resource, /*isCacheable=*/ true, /*isRecyclable=*/ false);
            newResource.setResourceListener(ref.key, listener);
            // 回调 Engine 的 onResourceReleased 方法
            // 这会导致此资源从 active 变成 memory cache 状态
            listener.onResourceReleased(ref.key, newResource); // <<<<<<<<<<<<<<<<<<<<
        }
    }
}
```

- Engine 实现了 EngineResource.ResourceListener，此处的 listener 就是 Engine，最终会回调 Engine.onResourceReleased。

```
@Override
public synchronized void onResourceReleased(Key cacheKey, EngineResource<?> resource) {
    activeResources.deactivate(cacheKey);
    if (resource.isCacheable()) {
        cache.put(cacheKey, resource);
    } else {
        resourceRecycler.recycle(resource);
    }
}
```

- 如果资源可以被缓存，则缓存到 memory cache，否则对资源进行回收。

2.4 4、磁盘缓存读取

- 我们分析下缓存的存取代码。我们看下：

```
public synchronized <R> LoadStatus load(...) {
    EngineKey key = keyFactory.buildKey(model, signature, width, height, transformations,
                                        resourceClass, transcodeClass, options);
    EngineResource<?> active = loadFromActiveResources(key, isMemoryCacheable);
    if (active != null) {
        cb.onResourceReady(active, DataSource.MEMORY_CACHE);
        return null;
    }
    EngineResource<?> cached = loadFromCache(key, isMemoryCacheable);
    if (cached != null) {
        cb.onResourceReady(cached, DataSource.MEMORY_CACHE);
        return null;
    }
    EngineJob<?> current = jobs.get(key, onlyRetrieveFromCache);
    if (current != null) {
        current.addCallback(cb, callbackExecutor);
        return new LoadStatus(cb, current);
    }
    EngineJob<R> engineJob = engineJobFactory.build(...);
    DecodeJob<R> decodeJob = decodeJobFactory.build(...);
    jobs.put(key, engineJob);
    engineJob.addCallback(cb, callbackExecutor);
    engineJob.start(decodeJob);
    return new LoadStatus(cb, engineJob);
}
```

- 缓存需要根据 EngineKey 去存取，先看下 EngineKey 的构造方法。

```
EngineKey(
    Object model,
    Key signature,
    int width
    int height,
    Map<Class<?>, Transformation<?>> transformations,
    Class<?> resourceClass,
    Class<?> transcodeClass,
    Options options)
```

- 参数的解释如下：

```
- model：load 方法传的参数；
- signature：BaseRequestOptions 的成员变量，默认会是 EmptySignature.obtain()
  - 在加载本地 resource 资源时会变成 ApplicationVersionSignature.obtain(context);
- width、height：如果没有指定 override(int size)，那么将得到 view 的 size；
- transformations：默认会基于 ImageView 的 scaleType 设置对应的四个 Transformation；
  - 如果指定了 transform，那么就基于该值进行设置；
- resourceClass：解码后的资源，如果没有 asBitmap、asGif，一般会是 Object；
- transcodeClass：最终要转换成的数据类型，根据 as 方法确定，加载本地 res 或者网络 URL，都会调用 asDrawable，所以为 Drawable
- options：如果没有设置过 transform，此处会根据 ImageView 的 scaleType 默认指定一个 option；
```

- 所以，在多次加载同一个 model 的过程中，只要上述任何一个参数有改变，都不会认为是同一个 key；

- 回到 Engine.load 方法，从缓存加载成功后的回调 cb.onResourceReady(cached, DataSource.MEMORY_CACHE)；可以看到：active 状态的资源和 memory cache 状态的资源都是 DataSource.MEMORY_CACHE，并且加载的资源都是 EngineResource 对象，该对象内部采用了引用计数去判断资源是否被释放，如果引用计数为 0，那么会调用 listener.onResourceReleased(key, this) 方法通知外界此资源已经释放了。这里的 listener 是 ResourceListener 类型的接口，只有一个 onResourceReleased(Key key, EngineResource resource) 方法，Engine 实现了该接口，此处的 listener 就是 Engine。在 Engine.onResourceReleased 方法中会判断资源是否可缓存，可缓存则将此资源放入 memory cache 中，否则回收掉该资源，代码如下：

```
public synchronized void onResourceReleased(Key cacheKey, EngineResource<?> resource) {
    // 从 activeResources 中移除
    activeResources.deactivate(cacheKey);
    if (resource.isCacheable()) {
        // 存入 MemoryCache
```

```
        cache.put(cacheKey, resource);
    } else {
        resourceRecycler.recycle(resource);
    }
}
```

- 继续回到 Engine.load 方法，先来看下 active 资源获取的方法。

```
@Nullable
private EngineResource<?> loadFromActiveResources(Key key, boolean isMemoryCacheable) {
    // 设置 skipMemoryCache(true)，则 isMemoryCacheable 为 false，跳过 ActiveResources
    if (!isMemoryCacheable) {
        return null;
    }
    EngineResource<?> active = activeResources.get(key);
    if (active != null) {
        // 命中缓存，引用计数 +1
        active.acquire();
    }
    return active;
}
```

- 继续分析 cached 资源获取的方法，如果从 active 资源中没有获取到缓存，则继续从内存缓存中查找。

```
private EngineResource<?> loadFromCache(Key key, boolean isMemoryCacheable) {
    // 设置 skipMemoryCache(true)，则 isMemoryCacheable 为 false，跳过 ActiveResources
    if (!isMemoryCacheable) {
        return null;
    }
    EngineResource<?> cached = getEngineResourceFromCache(key);
    if (cached != null) {
        // 命中缓存，引用计数 +1
        cached.acquire();
        // 将此资源从 memoryCache 中移到 activeResources 中
        activeResources.activate(key, cached);
    }
    return cached;
}
```

- 如果从 memoryCache 中获取到资源则将此资源从 memoryCache 中移到 activeResources 中。第一次加载的时候 activeResources 和 memoryCache 中都没有缓存的，后面继续通过 DecodeJob 和 EngineJob 去加载资源。DecoceJob 实现了 Runnable 接口，然后会被 EngineJob.start 方法提交到对应的线程池中去执行。在 DecoceJob 的 run 方法中，会依次从 ResourceCacheGenerator 和 DataCacheGenerator 中去取缓存数据，当这两者都取不到的情况下，会交给 SourceGenerator 加载网络图片或者本地资源。resource 资源和 data 资源都是磁盘缓存中的资源。

- 先看下 ResourceCacheGenerator.startNext。

```
@Override
public boolean startNext() {
    // list 里面只有一个 GlideUrl 对象
    List<Key> sourceIds = helper.getCacheKeys();
    if (sourceIds.isEmpty()) {
        return false;
    }
    // 获得了三个可以到达的 registeredResourceClasses
    // GifDrawable、Bitmap、BitmapDrawable
    List<Class<?>> resourceClasses = helper.getRegisteredResourceClasses();
    if (resourceClasses.isEmpty()) {
        if (File.class.equals(helper.getTranscodeClass())) {
            return false;
        }
        throw new IllegalStateException(
            "Failed to find any load path from " + helper.getModelClass() + " to "
            + helper.getTranscodeClass());
    }
    // 遍历 sourceIds 中的每一个 key、resourceClasses 中每一个 class，以及其他的一些值组成 key
    // 尝试在磁盘缓存中以 key 找到缓存文件
    while (modelLoaders == null || !hasNextModelLoader()) {
        resourceClassIndex++;
```

```
        if (resourceClassIndex >= resourceClasses.size()) {
            sourceIdIndex++;
            if (sourceIdIndex >= sourceIds.size()) {
                return false;
            }
            resourceClassIndex = 0;
        }
        Key sourceId = sourceIds.get(sourceIdIndex);
        Class<?> resourceClass = resourceClasses.get(resourceClassIndex);
        Transformation<?> transformation = helper.getTransformation(resourceClass);
        // PMD.AvoidInstantiatingObjectsInLoops Each iteration is comparatively expensive anyway,
        // we only run until the first one succeeds, the loop runs for only a limited
        // number of iterations on the order of 10-20 in the worst case.
        // 构造 key
        currentKey =
            new ResourceCacheKey(// NOPMD AvoidInstantiatingObjectsInLoops
                helper.getArrayPool(),
                sourceId,
                helper.getSignature(),
                helper.getWidth(),
                helper.getHeight(),
                transformation,
                resourceClass,
                helper.getOptions());
        // 查找缓存文件
        cacheFile = helper.getDiskCache().get(currentKey);
        // 如果找到了缓存文件，循环条件则会为 false，退出循环
        if (cacheFile != null) {
            sourceKey = sourceId;
            // 1. 找出注入时以 File.class 为 modelClass 的注入代码
            // 2. 调用所有注入的 factory.build 方法得到 ModelLoader
            // 3 . 过滤掉不可能处理 model 的 ModelLoader
            // 此时的 modelLoaders 值为：
            // [ByteBufferFileLoader, FileLoader, FileLoader, UnitModelLoader]
            modelLoaders = helper.getModelLoaders(cacheFile);
            modelLoaderIndex = 0;
        }
    }
    // 如果找到了缓存文件，hasNextModelLoader() 方法则会为 true，可以执行循环
    // 没有找到缓存文件，则不会进入循环，会直接返回 false
    loadData = null;
    boolean started = false;
    while (!started && hasNextModelLoader()) {
        ModelLoader<File, ?> modelLoader = modelLoaders.get(modelLoaderIndex++);
        // 在循环中会依次判断某个 ModelLoader 能不能加载此文件
        loadData = modelLoader.buildLoadData(cacheFile,
                                             helper.getWidth(), helper.getHeight(), helper.getOptions());
        if (loadData != null && helper.hasLoadPath(loadData.fetcher.getDataClass())) {
            started = true;
            // 如果某个 ModelLoader 可以，那么就调用其 fetcher 进行加载数据
// 加载成功或失败会通知自身
            loadData.fetcher.loadData(helper.getPriority(), this);
        }
    }
    return started;
}
```

- 该方法的相关注释代码里都有标明。找缓存时 key 的类型为 ResourceCacheKey，我们先来看下 ResourceCacheKey 的构成

```
currentKey =
    new ResourceCacheKey(// NOPMD AvoidInstantiatingObjectsInLoops
        helper.getArrayPool(),
        sourceId,
        helper.getSignature(),
        helper.getWidth(),
        helper.getHeight(),
        transformation,
        resourceClass,
        helper.getOptions());
ResourceCacheKey(
    ArrayPool arrayPool,
    Key sourceKey,
    Key signature,
    int width,
    int height,
    Transformation<?> appliedTransformation,
```

```
    Class<?> decodedResourceClass,
    Options options)
```

- arrayPool：默认值是 LruArrayPool，不参与 key 的 equals 方法;
- sourceKey：如果请求的是 URL，此处就是 GlideUrl(GlideUrl implements Key);
- signature：BaseRequestOptions 的成员变量，默认会是 EmptySignature.obtain()，
  - 在加载本地 resource 资源时会变成 ApplicationVersionSignature.obtain(context);
- width、height：如果没有指定 override(int size)，那么将得到 view 的 size;
- appliedTransformation：默认会根据 ImageView 的 scaleType 设置对应的 BitmapTransformation;
  - 如果指定了 transform，那么就会是指定的值;
- decodedResourceClass：可以被编码成的资源类型，如 BitmapDrawable 等;
- options：如果没有设置过 transform，此处会根据 ImageView 的 scaleType 默认指定一个 option;
- 在 ResourceCacheKey 中，arrayPool 并没有参与 equals 方法;
- 生成 ResourceCacheKey 之后会根据 key 去磁盘缓存中查找 cacheFile = helper.getDiskCache().get(
- helper.getDiskCache() 返回 DiskCache 接口，它的实现类是 DiskLruCacheWrapper，看下 DiskLruCacheWrapper.get 方法。

```
@Override
public File get(Key key) {
    String safeKey = safeKeyGenerator.getSafeKey(key);
    File result = null;
    try {
        final DiskLruCache.Value value = getDiskCache().get(safeKey);
        if (value != null) {
            result = value.getFile(0);
        }
    } catch (IOException e) {
    }
    return result;
}
```

- 这里调用 SafeKeyGenerator 生成了一个 String 类型的 SafeKey，实际上就是对原始 key 中每个字段都使用 SHA-256 加密，然后将得到的字节数组转换为 16 进制的字符串。生成 SafeKey 后，接着根据 SafeKey 去 DiskCache 里面找对应的缓存文件，然后返回文件。
- 回到 ResourceCacheGenerator.startNext 方法中,如果找到了缓存会调用 loadData.fetcher.loadData this); 这里的 fetcher 是 ByteBufferFetcher，ByteBufferFetcher 的 loadData 方法中最终会执行 callback.onDataReady(result) 这里 callback 是 ResourceCacheGenerator。

```
public void onDataReady(Object data) {
    cb.onDataFetcherReady(sourceKey, data, loadData.fetcher,
                          DataSource.RESOURCE_DISK_CACHE,
                          currentKey);
}
```

- ResourceCacheGenerator 的 onDataReady 方法又会回调 DecodeJob 的 onDataFetcherReady 方法进行后续的解码操作。
- 如果 ResourceCacheGenerator 没有找到缓存，就会交给 DataCacheGenerator 继续查找缓存。该类大体流程和 ResourceCacheGenerator 一样，有点不同的是，DataCacheGenerator 的构造器有两个构造器，其中的 DataCacheGenerator(List, DecodeHelper, FetcherReadyCallback) 构造器是给 SourceGenerator 准备的。因为如果没有磁盘缓存，那么从源头加载后，肯定需要进行磁盘缓存操作的。所以，SourceGenerator 会将加载后的资源保存到磁盘中，然后转交给 DataCacheGenerator 从磁盘中取出交给 ImageView 展示。

- 看下 DataCacheGenerator.startNext：

```
public boolean startNext() {
    while (modelLoaders == null || !hasNextModelLoader()) {
        sourceIdIndex++;
        if (sourceIdIndex >= cacheKeys.size()) {
            return false;
        }
        Key sourceId = cacheKeys.get(sourceIdIndex);

        Key originalKey = new DataCacheKey(sourceId, helper.getSignature());
        cacheFile = helper.getDiskCache().get(originalKey);

        while (!started && hasNextModelLoader()) {
            ModelLoader<File, ?> modelLoader = modelLoaders.get(modelLoaderIndex++);
            loadData =
                modelLoader.buildLoadData(cacheFile, helper.getWidth(), helper.getHeight(),
                                          helper.getOptions());
            if (loadData != null && helper.hasLoadPath(loadData.fetcher.getDataClass())) {
                started = true;
                loadData.fetcher.loadData(helper.getPriority(), this);
            }
        }
        return started;
    }
}
```

- 这里的 originalKey 是 DataCacheKey 类型的，DataCacheKey 构造方法如下：
- DataCacheKey(Key sourceKey, Key signature)
- 这里的 sourceKey 和 signature 与 ResourceCacheKey 中的两个变量一致，从这里就可以看出：DataCache 缓存的是原始的数据，ResourceCache 缓存的是是被解码、转换后的数据。
- 如果 DataCacheGenerator 没有取到缓存，那么会交给 SourceGenerator 从源头加载。看下 SourceGenerator 的 startNext 方法。

```
@Override
public boolean startNext() {
    // 首次运行 dataToCache 为 null
    if (dataToCache != null) {
        Object data = dataToCache;
        dataToCache = null;
        cacheData(data);
    }
    // 首次运行 sourceCacheGenerator 为 null
    if (sourceCacheGenerator != null && sourceCacheGenerator.startNext()) {
        return true;
    }
    sourceCacheGenerator = null;
    loadData = null;
    boolean started = false;
    while (!started && hasNextModelLoader()) {
        loadData = helper.getLoadData().get(loadDataListIndex++);
        if (loadData != null
            && (helper.getDiskCacheStrategy().isDataCacheable(loadData.fetcher.getDataSource())
                || helper.hasLoadPath(loadData.fetcher.getDataClass()))) {
            started = true;
            loadData.fetcher.loadData(helper.getPriority(), this);
        }
    }
    return started;
}
```

- 加载成功后，依然会回调 SourceGenerator 的 onDataReady 方法。

```
@Override
public void onDataReady(Object data) {
    DiskCacheStrategy diskCacheStrategy = helper.getDiskCacheStrategy();
    if (data != null && diskCacheStrategy.isDataCacheable(loadData.fetcher.getDataSource())) {
        dataToCache = data;
        // cb 为 DecodeJob
        cb.reschedule();
    } else {
```

```
        // cb 为 DecodeJob
        cb.onDataFetcherReady(loadData.sourceKey, data, loadData.fetcher,
                              loadData.fetcher.getDataSource(), originalKey);
    }
}
```

- 先判断获取到的数据是否需要进行磁盘缓存，如果需要磁盘缓存，则经过 DecodeJob、EngineJob 的调度，重新调用 SourceGenerator.startNext 方法，此时 dataToCache 已经被赋值，则会调用 cacheData(data); 进行磁盘缓存的写入，并转交给 DataCacheGenerator 完成后续的处理; 否则就通知 DecodeJob 已经加载成功。

- 先看下 SourceGenerator 的 startNext 方法中调用的 SourceGenerator.cacheData(data)。

```
private void cacheData(Object dataToCache) {
    long startTime = LogTime.getLogTime();
    try {
        Encoder<Object> encoder = helper.getSourceEncoder(dataToCache);
        DataCacheWriter<Object> writer =
            new DataCacheWriter<>(encoder, dataToCache, helper.getOptions());
        originalKey = new DataCacheKey(loadData.sourceKey, helper.getSignature());
        helper.getDiskCache().put(originalKey, writer);

    } finally {
        loadData.fetcher.cleanup();
    }
    sourceCacheGenerator =
        new DataCacheGenerator(Collections.singletonList(loadData.sourceKey), helper, this);
}
```

- cacheData 方法先构建了一个 DataCacheKey 将 data 写入了磁盘，然后 new 了一个 DataCacheGenerator 赋值给 sourceCacheGenerator。回到 startNext 继续向下执行，此时 sourceCacheGenerator 不为空，就调用其 startNext() 方法从磁盘中加载刚写入磁盘的数据，并返回 true 让 DecodeJob 停止尝试获取数据。此时，从磁盘缓存中读取数据的逻辑已经完成，接下来是写磁盘缓存。

- 假如 SourceGenerator 的 onDataReady 方法中的磁盘缓存策略不可用，则会回调 DecodeJob.onDataFetcherReady 方法。

```
// DecodeJob
@Override
public void onDataFetcherReady(Key sourceKey, Object data, DataFetcher<?> fetcher,
                               DataSource dataSource, Key attemptedKey) {
    this.currentSourceKey = sourceKey;
    this.currentData = data;
    this.currentFetcher = fetcher;
    this.currentDataSource = dataSource;
    this.currentAttemptingKey = attemptedKey;
    if (Thread.currentThread() != currentThread) {
        runReason = RunReason.DECODE_DATA;
        callback.reschedule(this);
    } else {
        GlideTrace.beginSection("DecodeJob.decodeFromRetrievedData");
        try {
            decodeFromRetrievedData();
        } finally {
            GlideTrace.endSection();
        }
    }
}
private void decodeFromRetrievedData() {
    Resource<R> resource = null;
    try {
        resource = decodeFromData(currentFetcher, currentData, currentDataSource);
    } catch (GlideException e) {
        e.setLoggingDetails(currentAttemptingKey, currentDataSource);
        throwables.add(e);
    }
    if (resource != null) {
        notifyEncodeAndRelease(resource, currentDataSource);
    } else {
        runGenerators();
    }
}
```

- decodeFromRetrievedData(); 后续的方法调用链在之前的文章中分析过，主要做的事情就是：将原始的 data 数据转变为可以供 ImageView 显示的 resource 数据并将其显示在 ImageView 上。
- 将原始的 data 数据转变为 resource 数据后，会调用 DecodeJob.onResourceDecoded(dataSource, decoded)。

```
@Synthetic
@NonNull
<Z> Resource<Z> onResourceDecoded(DataSource dataSource,
                                  @NonNull Resource<Z> decoded) {
    @SuppressWarnings("unchecked")
        Class<Z> resourceSubClass = (Class<Z>) decoded.get().getClass();
    Transformation<Z> appliedTransformation = null;
    Resource<Z> transformed = decoded;
    // 不是 resource cache 时要 transform
    if (dataSource != DataSource.RESOURCE_DISK_CACHE) {
        appliedTransformation = decodeHelper.getTransformation(resourceSubClass);
        transformed = appliedTransformation.transform(glideContext, decoded, width, height);
    }
    // TODO: Make this the responsibility of the Transformation.
    if (!decoded.equals(transformed)) {
        decoded.recycle();
    }
    final EncodeStrategy encodeStrategy;
    final ResourceEncoder<Z> encoder;
    if (decodeHelper.isResourceEncoderAvailable(transformed)) {
        encoder = decodeHelper.getResultEncoder(transformed);
        encodeStrategy = encoder.getEncodeStrategy(options);
    } else {
        encoder = null;
        encodeStrategy = EncodeStrategy.NONE;
    }
    Resource<Z> result = transformed;
    boolean isFromAlternateCacheKey = !decodeHelper.isSourceKey(currentSourceKey);
    if (diskCacheStrategy.isResourceCacheable(isFromAlternateCacheKey, dataSource,
                                              encodeStrategy)) {
        if (encoder == null) {
            throw new Registry.NoResultEncoderAvailableException(transformed.get().getClass());
        }
        final Key key;
        switch (encodeStrategy) {
        case SOURCE:
            key = new DataCacheKey(currentSourceKey, signature);
            break;
        case TRANSFORMED:
            key = new ResourceCacheKey(
                decodeHelper.getArrayPool(),
                currentSourceKey,
                signature,
                width,
                height,
                appliedTransformation,
                resourceSubClass,
                options);
            break;
        default:
            throw new IllegalArgumentException("Unknown strategy: " + encodeStrategy);
        }
        LockedResource<Z> lockedResult = LockedResource.obtain(transformed);
        deferredEncodeManager.init(key, encoder, lockedResult);
        result = lockedResult;
    }
    return result;
}
```

- 然后是此过程中的磁盘缓存过程，影响的因素有 encodeStrategy、DiskCacheStrategy.isResourceCac encodeStrategy 根据 resource 数据的类型来判断，如果是 Bitmap 或 BitmapDrawable，那么就是 TRANSFORMED; 如果是 GifDrawable，那么就是 SOURCE。磁盘缓存策略默认是 DiskCacheStrategy.AUTOMATIC。源码如下：

```
public static final DiskCacheStrategy AUTOMATIC = new DiskCacheStrategy() {
    public boolean isDataCacheable(DataSource dataSource) {
        return dataSource == DataSource.REMOTE;
```

```
    }
    public boolean isResourceCacheable(boolean isFromAlternateCacheKey, DataSource dataSource, EncodeStrategy encodeStrateg
        return (isFromAlternateCacheKey && dataSource == DataSource.DATA_DISK_CACHE || dataSource == DataSource.LOCAL) && e
    }
    public boolean decodeCachedResource() {
        return true;
    }
    public boolean decodeCachedData() {
        return true;
    }
};
```

- 只有 dataSource 为 DataSource.LOCAL 且 encodeStrategy 为 EncodeStrategy.TRANSFORMED 时，才允许缓存。也就是只有本地的 resource 数据为 Bitmap 或 BitmapDrawable 的资源才可以缓存。
- 在 DecodeJob.onResourceDecoded 中会调用 deferredEncodeManager.init(key, encoder, lockedResult); 去初始化 deferredEncodeManager。
- 在 DecodeJob 的 decodeFromRetrievedData(); 中拿到 resource 数据后会调用 notifyEncodeAndRelease(resource, currentDataSource) 利用 deferredEncodeManager 对象进行磁盘缓存的写入;

```
private void notifyEncodeAndRelease(Resource<R> resource, DataSource dataSource) {
    // 通知回调，资源已经就绪
    notifyComplete(result, dataSource);
    stage = Stage.ENCODE;
    try {
        if (deferredEncodeManager.hasResourceToEncode()) {
            deferredEncodeManager.encode(diskCacheProvider, options);
        }
    } finally {
        if (lockedResource != null) {
            lockedResource.unlock();
        }
    }
    onEncodeComplete();
}
```

- deferredEncodeManager.encode 行磁盘缓存的写入。

```
// DecodeJob
private static class DeferredEncodeManager<Z> {
    private Key key;
    private ResourceEncoder<Z> encoder;
    private LockedResource<Z> toEncode;
    @Synthetic
    DeferredEncodeManager() { }
    // We just need the encoder and resource type to match, which this will enforce.
    @SuppressWarnings("unchecked")
    <X> void init(Key key, ResourceEncoder<X> encoder, LockedResource<X> toEncode) {
        this.key = key;
        this.encoder = (ResourceEncoder<Z>) encoder;
        this.toEncode = (LockedResource<Z>) toEncode;
    }
    void encode(DiskCacheProvider diskCacheProvider, Options options) {
        GlideTrace.beginSection("DecodeJob.encode");
        try {
            // 存入磁盘缓存
            diskCacheProvider.getDiskCache().put(key,
                                                 new DataCacheWriter<>(encoder, toEncode, options));
        } finally {
            toEncode.unlock();
            GlideTrace.endSection();
        }
    }
    boolean hasResourceToEncode() {
        return toEncode != null;
    }
    void clear() {
        key = null;
        encoder = null;
        toEncode = null;
    }
}
```

- diskCacheProvider.getDiskCache() 获取到 DiskLruCacheWrapper，并调用 DiskLruCacheWrapper 的 put 写入。DiskLruCacheWrapper 在写入的时候会使用到写锁 DiskCacheWriteLocker，锁对象由对象池 WriteLockPool 创建，写锁 WriteLock 实现是一个不公平锁 ReentrantLock。
- 在缓存写入前，会判断 key 对应的 value 存不存在，若存在则不写入。缓存的真正写入会由 DataCacheWriter 交给 ByteBufferEncoder 和 StreamEncoder 两个具体类来写入，前者负责将 ByteBuffer 写入到文件，后者负责将 InputStream 写入到文件。
- 目前为止，磁盘缓存的读写流程都已分析完成。

2.5 5、内存缓存：ActiveResource 与 MemoryCache 读取

- 回到 DecodeJob.notifyEncodeAndRelease 方法中，经过 notifyComplete、EngineJob.onResourceRea notifyCallbacksOfResult 方法中。
- 在该方法中一方面会将原始的 resource 包装成一个 EngineResource，然后通过回调传给 Engine.onEngineJobComplete。

```
@Override
public synchronized void onEngineJobComplete(
    EngineJob<?> engineJob, Key key, EngineResource<?> resource) {
    // 设置资源的回调为自己，这样在资源释放时会通知自己的回调方法
    if (resource != null) {
        resource.setResourceListener(key, this);
        // 将资源放入 activeResources 中，资源变为 active 状态
        if (resource.isCacheable()) {
            activeResources.activate(key, resource);
        }
    }
    // 将 engineJob 从 Jobs 中移除
    jobs.removeIfCurrent(key, engineJob);
}
```

- 在这里会将资源放入 activeResources 中，资源变为 active 状态。后面会使用 Executors.mainThreadExecutor() 调用 SingleRequest.onResourceReady 回调进行资源的显示。在触发回调前后各有一个地方会对 engineResource 进行 acquire() 和 release() 操作，这两个操作分别发生在 notifyCallbacksOfResult() 方法的 incrementPendingCallbacks、decrementPendingCallbacks() 调用中。

```
@Synthetic
void notifyCallbacksOfResult() {
    ResourceCallbacksAndExecutors copy;
    Key localKey;
    EngineResource<?> localResource;
    synchronized (this) {
        engineResource = engineResourceFactory.build(resource, isCacheable);
        hasResource = true;
        copy = cbs.copy();
        incrementPendingCallbacks(copy.size() + 1);
        localKey = key;
        localResource = engineResource;
    }
    listener.onEngineJobComplete(this, localKey, localResource);
    for (final ResourceCallbackAndExecutor entry : copy) {
        entry.executor.execute(new CallResourceReady(entry.cb));
    }
    decrementPendingCallbacks();
}
synchronized void incrementPendingCallbacks(int count) {
    if (pendingCallbacks.getAndAdd(count) == 0 && engineResource != null) {
        engineResource.acquire();
    }
}
synchronized void decrementPendingCallbacks() {
    int decremented = pendingCallbacks.decrementAndGet();
    if (decremented == 0) {
        if (engineResource != null) {
            engineResource.release();
```

```
        }
        release();
    }
}
private class CallResourceReady implements Runnable {
    private final ResourceCallback cb;
    CallResourceReady(ResourceCallback cb) {
        this.cb = cb;
    }
    @Override
        public void run() {
        synchronized (EngineJob.this) {
            if (cbs.contains(cb)) {
                // Acquire for this particular callback.
                engineResource.acquire();
                callCallbackOnResourceReady(cb);
                removeCallback(cb);
            }
            decrementPendingCallbacks();
        }
    }
}
```

- CallResourceReady 的 run 方法中也会调用 engineResource.acquire()，上面的代码调用结束后，engineResource 的引用计数为 1。engineResource 的引用计数会在 RequestManager.onDestory 方法中最终调用 SingleRequest.clear() 方法，SingleRequest.clear() 内部调用 releaseResource()、Engine.release 进行释放，这样引用计数就变为 0。引用计数就变为 0 后会通知 Engine 将此资源从 active 状态变成 memory cache 状态。如果我们再次加载资源时可以从 memory cache 中加载，那么资源又会从 memory cache 状态变成 active 状态。也就是说，在资源第一次显示后，我们关闭页面，资源会由 active 变成 memory cache; 然后我们再次进入页面，加载时会命中 memory cache，从而又变成 active 状态。

2.6 总结

- 读取内存缓存时，先从 LruCache 算法机制的内存缓存读取，再从弱引用机制的内存缓存读取;
- 写入内存缓存时，先写入弱引用机制的内存缓存，等到图片不再被使用时，再写入到 LruCache 算法机制的内存缓存;
- 读取磁盘缓存时，先读取转换后图片的缓存，再读取原始图片的缓存。