Glide 源码分析

> 图片加载框架，相对于UniversalImageLoader，Picasso，它还支持video，Gif，SVG格式，支持缩略图请求，旨在打造更好的列表图片滑动体验。Glide有生命周期的概念（主要是对请求进行pause，resume，clear），而且其生命周期与Activity/Fragment的生命周期绑定，支持Volley，OkHttp，并提供了相应的integration libraries，内存方面也更加友好。

开始

添加Glide

Gradle

```
repositories {
  mavenCentral() // jcenter() works as well because it pulls from Maven Central
}
dependencies {
  compile 'com.github.bumptech.glide:glide:3.7.0'
  compile 'com.android.support:support-v4:19.1.0'
}
```

Maven

```
com.google.android
support-v4
r7
```

使用

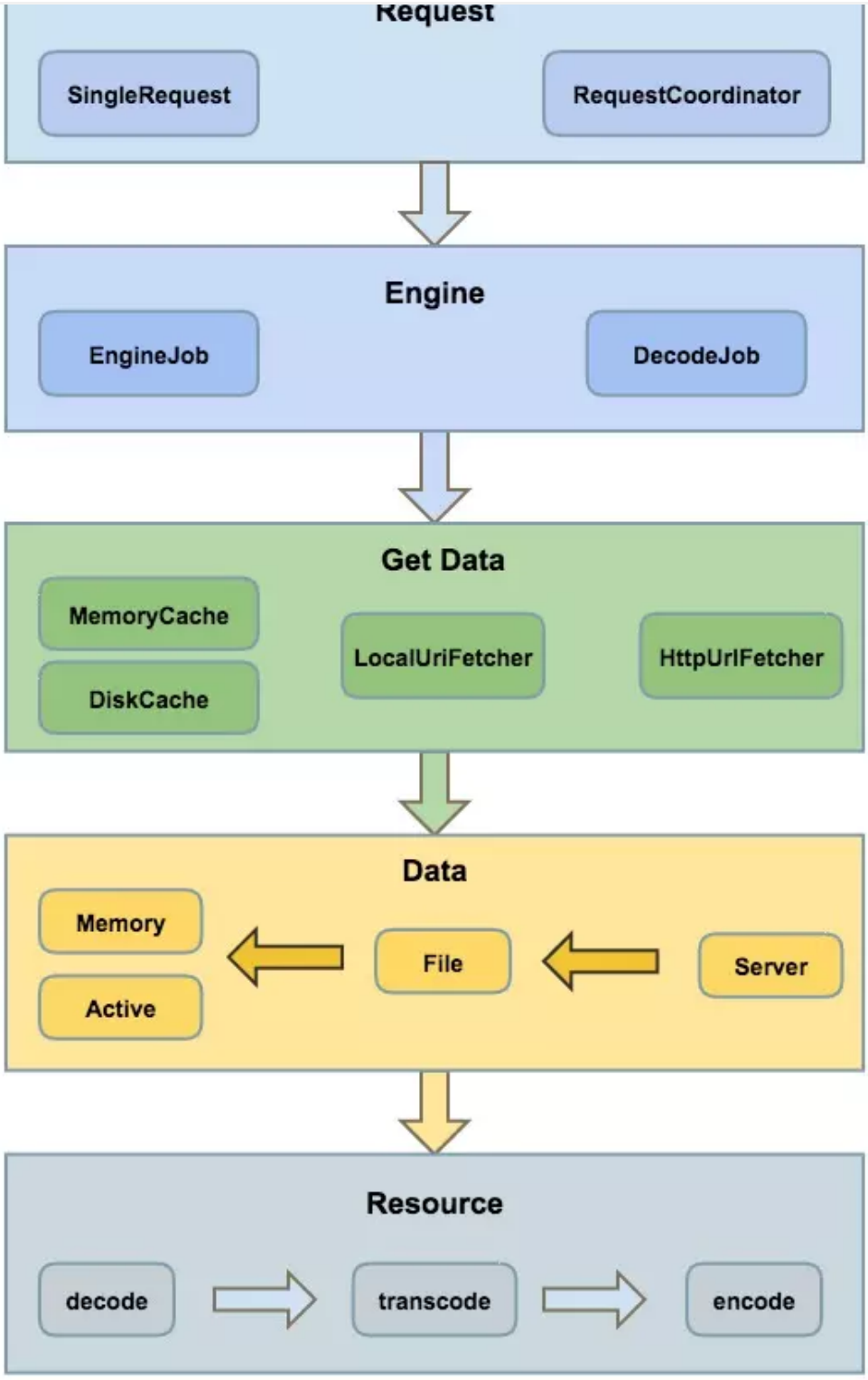
> 就像 Picasso， Glide 库是使用流接口(fluent interface)。对一个完整的功能请求，Glide 建造者要求最少有三个参数。
>
> - with(Context context) - 对于很多 Android API 调用，Context 是必须的。Glide 在这里也一样
> - load(String imageUrl) - 这里你可以指定哪个图片应该被加载，同上它会是一个字符串的形式表示一个网络图片的 URL
> - into(ImageView targetImageView) 你的图片会显示到对应的 ImageView 中。理论解释总是苍白的，所以，看一下实际的例子吧：

```
// For a simple view:
@Override public void onCreate(Bundle savedInstanceState) {
  ...
  ImageView imageView = (ImageView) findViewById(R.id.my_image_view);
  Glide.with(this).load("http://goo.gl/gEgYUd").into(imageView);
}
// For a simple image list:
@Override public View getView(int position, View recycled, ViewGroup container) {
  final ImageView myImageView;
  if (recycled == null) {
    myImageView = (ImageView) inflater.inflate(R.layout.my_image_view, container, false);
  } else {
    myImageView = (ImageView) recycled;
  }
  String url = myUrls.get(position);
  Glide
    .with(myFragment)
    .load(url)
    .centerCrop()
    .placeholder(R.drawable.loading_spinner)
    .crossFade()
```

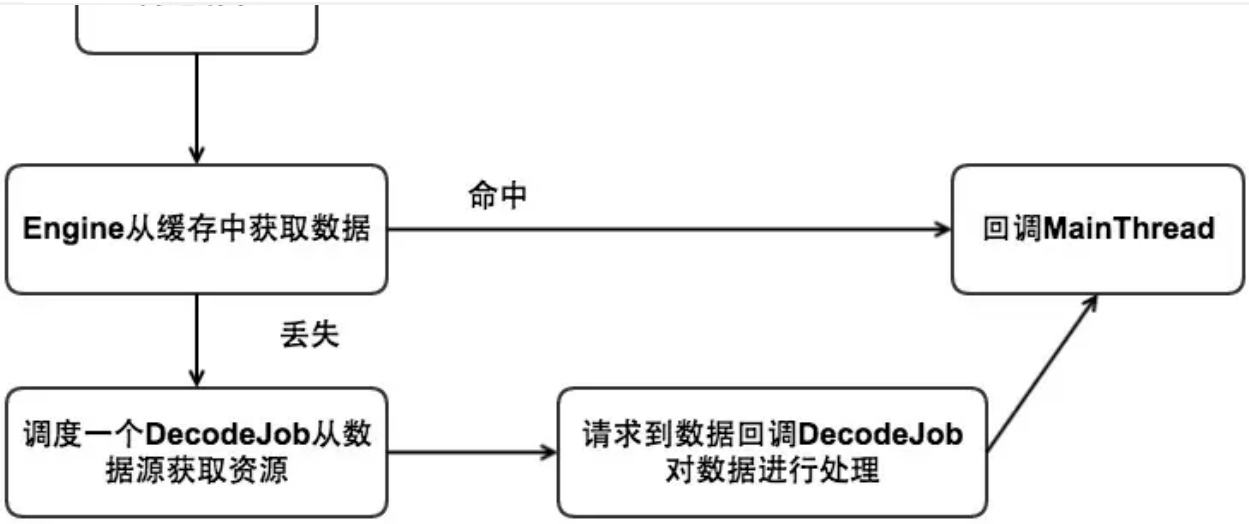
源码分析

图解

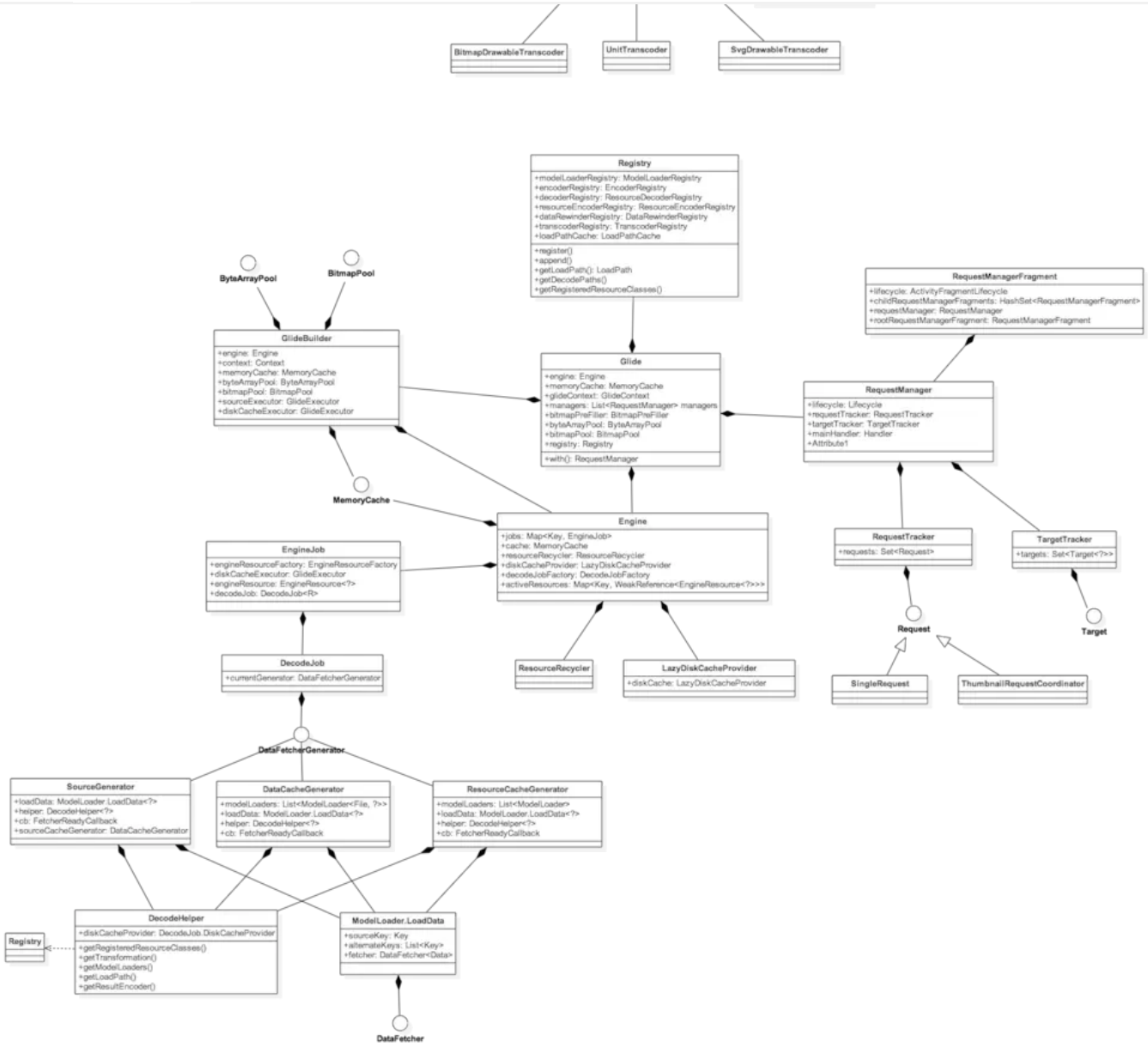
总体设计图


Request
SingleRequest
RequestCoordinator
Engine
EngineJob
DecodeJob
Get Data
MemoryCache
DiskCache
LocalUriFetcher
HttpUrlFetcher
Data
Memory
Active
File
Server
Resource
decode
transcode
encode

流程图


Engine从缓存中获取数据
命中
回调MainThread
丢失
调度一个DecodeJob从数据源获取资源
请求到数据回调DecodeJob对数据进行处理

类关系图



最简请求流程

Glide.with(this).load().into()

> 首先看Glide静态方法with()

```
public static RequestManager with(Context context) {
  RequestManagerRetriever retriever = RequestManagerRetriever.get();
  return retriever.get(context);
```

```
public static RequestManager with(FragmentActivity activity) {
  RequestManagerRetriever retriever = RequestManagerRetriever.get();
  return retriever.get(activity);
}
@TargetApi(Build.VERSION_CODES.HONEYCOMB)
public static RequestManager with(android.app.Fragment fragment) {
  RequestManagerRetriever retriever = RequestManagerRetriever.get();
  return retriever.get(fragment);
}
public static RequestManager with(Fragment fragment) {
  RequestManagerRetriever retriever = RequestManagerRetriever.get();
  return retriever.get(fragment);
}
```

> 可以看到有5个重载方法,都创建了一个RequestManagerRetriever实例然后调用了get()方法

```
public RequestManager get(Context context) {
  if (context == null) {
    throw new IllegalArgumentException("You cannot start a load on a null Context");
  } else if (Util.isOnMainThread() && !(context instanceof Application)) {
    if (context instanceof FragmentActivity) {
      return get((FragmentActivity) context);
    } else if (context instanceof Activity) {
      return get((Activity) context);
    } else if (context instanceof ContextWrapper) {
      return get(((ContextWrapper) context).getBaseContext());
    }
  }
  return getApplicationManager(context);
}
```

> 其实不管到哪个方法最后都会执行getApplicationManager(context)

```
private RequestManager getApplicationManager(Context context) {
  // Either an application context or we're on a background thread.
  if (applicationManager == null) {
    synchronized (this) {
      if (applicationManager == null) {
        // Normally pause/resume is taken care of by the fragment we add to the fragment or
```

```
        applicationManager =
            new RequestManager(
                glide, new ApplicationLifecycle(), new EmptyRequestManagerTreeNode());
      }
    }
  }
  return applicationManager;
}
```

> 使用了单例模式,先创建了Glide实例,其实Glide实例也是通过单例创建,然后创建了RequestManager,显然其和生命周期有关系new ApplicationLifecycle

```
public static Glide get(Context context) {
  if (glide == null) {
    synchronized (Glide.class) {
      if (glide == null) {
        Context applicationContext = context.getApplicationContext();
        List modules = new ManifestParser(applicationContext).parse();
        GlideBuilder builder = new GlideBuilder(applicationContext);
        for (GlideModule module : modules) {
          module.applyOptions(applicationContext, builder);
        }
        glide = builder.createGlide();
        for (GlideModule module : modules) {
          module.registerComponents(applicationContext, glide.registry);
        }
      }
    }
  }
  return glide;
}
```

> 向外暴露单例静态接口，构建Request，配置资源类型，缓存策略，图片处理等，可以直接通过该类完整简单的图片请求和填充。内存持有一些内存变量BitmapPool，MemoryCache，ByteArrayPool，便于低内存情况时自动清理内存。其中先创建了GlideBuilder,然后调用其createGlide创建glide,GlideBuilder为Glide设置一些默认配置，比如：Engine，MemoryCache，DiskCache，RequestOptions，GlideExecutor，MemorySizeCalculato
> 而且可以通过GlideModule进行一些延迟的配置和ModelLoaders的注册。(GlideModule其

```
 * RequestBuilder#load(Object)} with the given model.
 *
 * @return A new request builder for loading a {@link Drawable} using the given model.
 */
public RequestBuilder load(@Nullable Object model) {
  return asDrawable().load(model);
}
```

> 代码可知先调用了asDrawable,过程中又调用了as返回RequestBuilder,然后调用RequestBuilder的load,load()有多个重载方法

```
public RequestBuilder load(@Nullable Object model) {
 return loadGeneric(model);
}
public RequestBuilder load(@Nullable String string) {
 return loadGeneric(string);
}
public RequestBuilder load(@Nullable Uri uri) {
 return loadGeneric(uri);
}
public RequestBuilder load(@Nullable File file) {
 return loadGeneric(file);
}
public RequestBuilder load(@Nullable Integer resourceId) {
 return loadGeneric(resourceId).apply(signatureOf(ApplicationVersionSignature.obtain(context)));
}
@Deprecated
public RequestBuilder load(@Nullable URL url) {
 return loadGeneric(url);
}
public RequestBuilder load(@Nullable byte[] model) {
 return loadGeneric(model).apply(signatureOf(new ObjectKey(UUID.randomUUID().toString()))
     .diskCacheStrategy(DiskCacheStrategy.NONE).skipMemoryCache(true /*skipMemoryCache*/));
}
```

> 可见它们都调用了loadGeneric方法

```
private RequestBuilder loadGeneric(@Nullable Object model) {
  this.model = model;
```

> 至此完成了Glide.with(context).load(),接下来调用RequestBuilder.into()

```
public Target into(ImageView view) {
  //首先判断是否在主线程(跟新UI只能在主线程)
  Util.assertMainThread();
  Preconditions.checkNotNull(view);
  if (!requestOptions.isTransformationSet()
      && requestOptions.isTransformationAllowed()
      && view.getScaleType() != null) {
    if (requestOptions.isLocked()) {
      requestOptions = requestOptions.clone();
    }
    //根据ImageView的ScaleType配置requestOptions,在创建requestbuilder时候有一个默认的requestoptionsthis
    switch (view.getScaleType()) {
      case CENTER_CROP:
        requestOptions.optionalCenterCrop(context);
        break;
      case CENTER_INSIDE:
        requestOptions.optionalCenterInside(context);
        break;
      case FIT_CENTER:
      case FIT_START:
      case FIT_END:
        requestOptions.optionalFitCenter(context);
        break;
      //$CASES-OMITTED$
      default:
        // Do nothing.
    }
  }
  //调用into()
  return into(context.buildImageViewTarget(view, transcodeClass));
}
```

> 经过判断和配置后调用into(context.buildImageViewTarget(view, transcodeClass)),先执行context.buildImageViewTarget,其中参数transcodeClass是创建requestManager调用as方法传进来的(默认Bitmap.class)

```
}
/**
 * A factory responsible for producing the correct type of
 * {@link com.bumptech.glide.request.target.Target} for a given {@link android.view.View} subclass.
 */
public class ImageViewTargetFactory {
  @SuppressWarnings("unchecked")
  public  Target buildTarget(ImageView view, Class clazz) {
    if (Bitmap.class.equals(clazz)) {
      return (Target) new BitmapImageViewTarget(view);
    } else if (Drawable.class.isAssignableFrom(clazz)) {
      return (Target) new DrawableImageViewTarget(view);
    } else {
      throw new IllegalArgumentException(
          "Unhandled class: " + clazz + ", try .as*(Class).transcode(ResourceTranscoder)");
    }
  }
}
```

> 可以看出对clazz进行了判断,如果是bitmap返回Target ,如果是drawable返回Target ,最后调用into(Y target)设置资源的Target，并创建，绑定，跟踪，发起请求

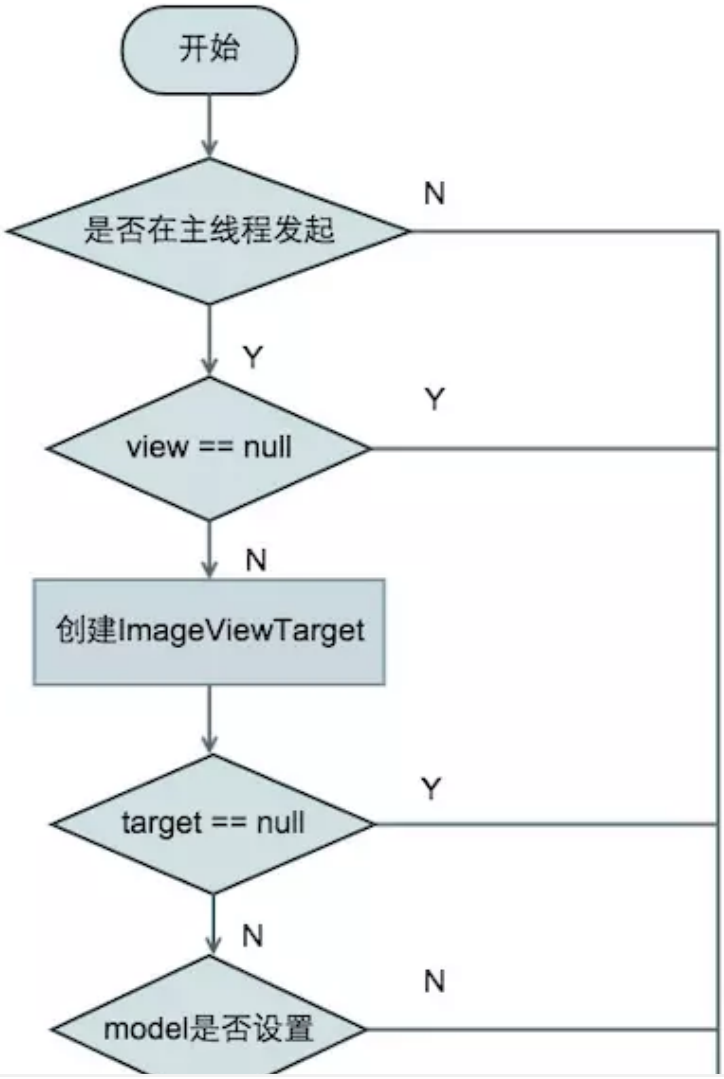
```
public > Y into(@NonNull Y target) {
  Util.assertMainThread();
  Preconditions.checkNotNull(target);
  if (!isModelSet) {
    throw new IllegalArgumentException("You must call #load() before calling #into()");
  }
  Request previous = target.getRequest();
  if (previous != null) {
    requestManager.clear(target);
  }
  requestOptions.lock();
  Request request = buildRequest(target);
  target.setRequest(request);
  requestManager.track(target, request);
  return target;
}
```

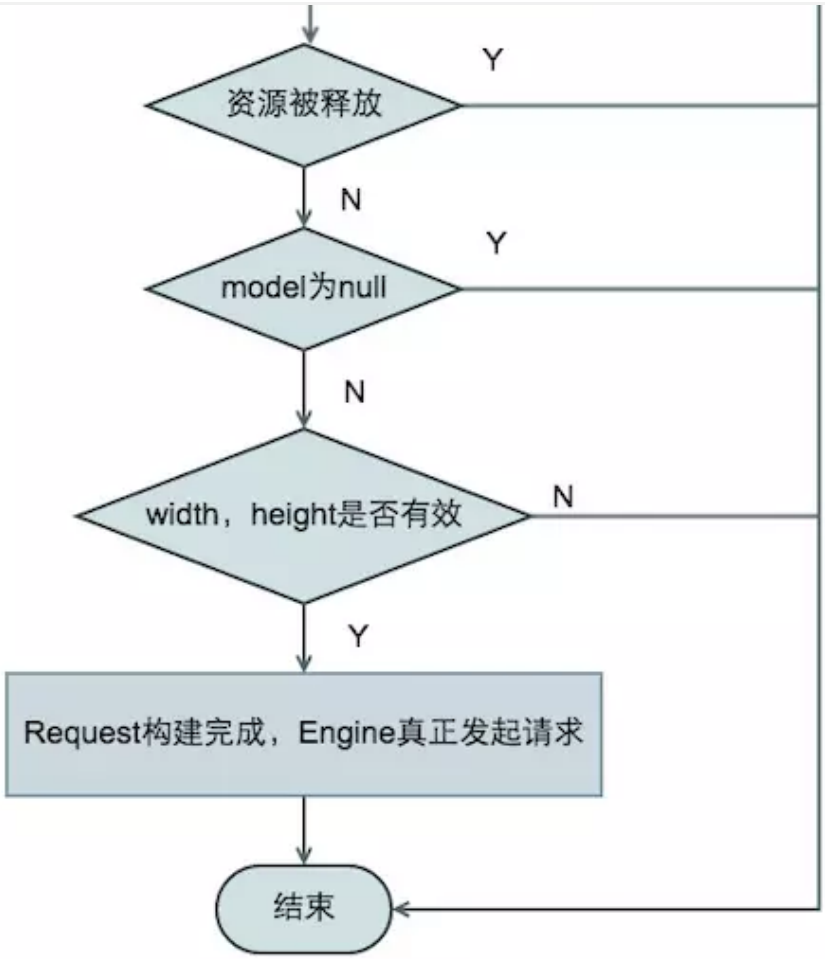
```
private Request buildRequest(Target target) {
  return buildRequestRecursive(target, null, transitionOptions, requestOptions.getPriority(),
      requestOptions.getOverrideWidth(), requestOptions.getOverrideHeight());
}
private Request buildRequestRecursive(Target target,
    @Nullable ThumbnailRequestCoordinator parentCoordinator,
    TransitionOptions transitionOptions,
    Priority priority, int overrideWidth, int overrideHeight) {
  if (thumbnailBuilder != null) {
    // Recursive case: contains a potentially recursive thumbnail request builder.
    if (isThumbnailBuilt) {
      throw new IllegalStateException("You cannot use a request as both the main request and a "
          + "thumbnail, consider using clone() on the request(s) passed to thumbnail()");
    }
    TransitionOptions thumbTransitionOptions =
        thumbnailBuilder.transitionOptions;
    if (DEFAULT_ANIMATION_OPTIONS.equals(thumbTransitionOptions)) {
      thumbTransitionOptions = transitionOptions;
    }
    Priority thumbPriority = thumbnailBuilder.requestOptions.isPrioritySet()
        ? thumbnailBuilder.requestOptions.getPriority() : getThumbnailPriority(priority);
    int thumbOverrideWidth = thumbnailBuilder.requestOptions.getOverrideWidth();
    int thumbOverrideHeight = thumbnailBuilder.requestOptions.getOverrideHeight();
    if (Util.isValidDimensions(overrideWidth, overrideHeight)
        && !thumbnailBuilder.requestOptions.isValidOverride()) {
      thumbOverrideWidth = requestOptions.getOverrideWidth();
      thumbOverrideHeight = requestOptions.getOverrideHeight();
    }
    ThumbnailRequestCoordinator coordinator = new ThumbnailRequestCoordinator(parentCoordinator);
    Request fullRequest = obtainRequest(target, requestOptions, coordinator,
        transitionOptions, priority, overrideWidth, overrideHeight);
    isThumbnailBuilt = true;
    // Recursively generate thumbnail requests.
    Request thumbRequest = thumbnailBuilder.buildRequestRecursive(target, coordinator,
        thumbTransitionOptions, thumbPriority, thumbOverrideWidth, thumbOverrideHeight);
    isThumbnailBuilt = false;
    coordinator.setRequests(fullRequest, thumbRequest);
    return coordinator;
  } else if (thumbSizeMultiplier != null) {
    // Base case: thumbnail multiplier generates a thumbnail request, but cannot recurse.
    ThumbnailRequestCoordinator coordinator = new ThumbnailRequestCoordinator(parentCoordinator);
    Request fullRequest = obtainRequest(target, requestOptions, coordinator, transitionOptions
        priority, overrideWidth, overrideHeight);
    BaseRequestOptions thumbnailOptions = requestOptions.clone()
```

```
    } else {
      // Base case: no thumbnail.
      return obtainRequest(target, requestOptions, parentCoordinator, transitionOptions, priority,
          overrideWidth, overrideHeight);
    }
  }
```

> 首先会执行obtainRequest里的SingleRequest.obtain进行SingleRequest init(),然后执行into的requestManager.track(target, request)进入执行SingleRequest.begin(),然后执行其onSizeReady,调用engine.load()

请求流程图解





Engine

> 任务创建，发起，回调，管理存活和缓存的资源

load()

```
public  LoadStatus load(
    GlideContext glideContext,
    Object model,
    Key signature,
    int width,
    int height,
    Class resourceClass,
    Class transcodeClass,
    Priority priority
```

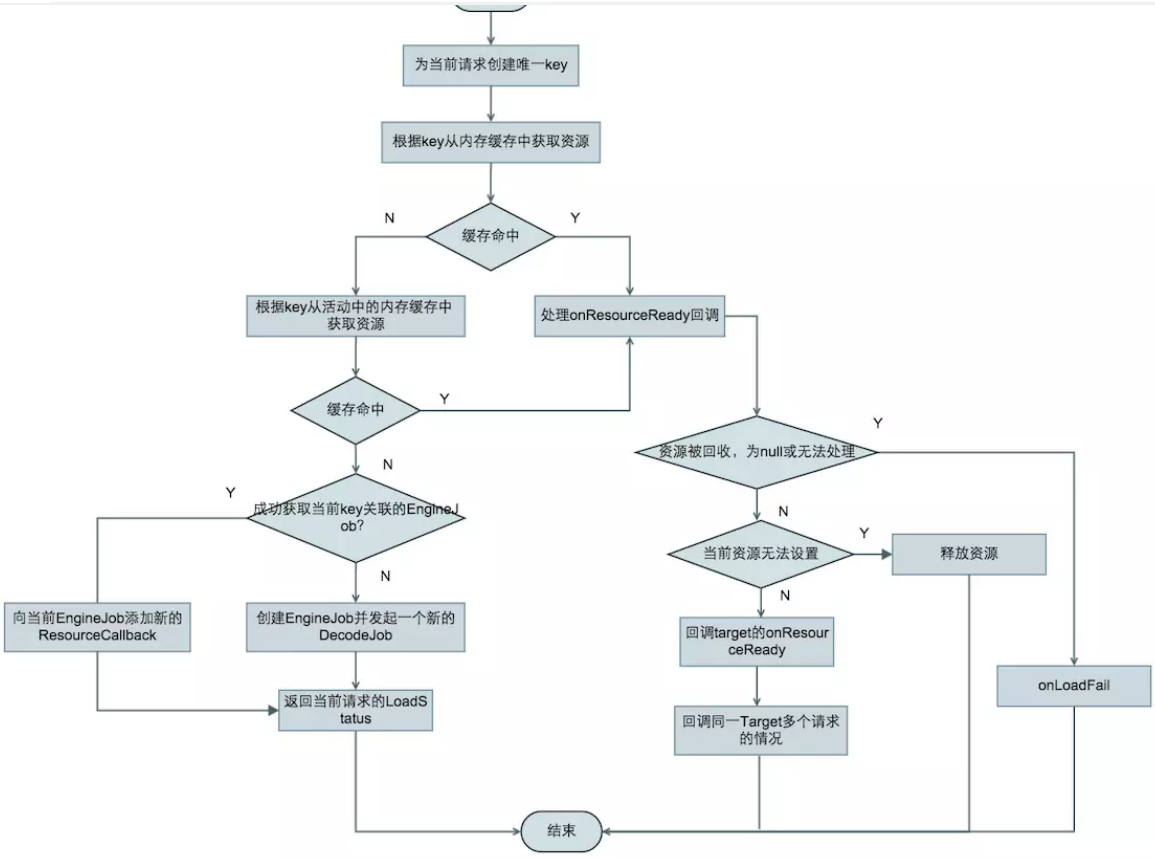
```
    boolean useUnlimitedSourceExecutorPool,
    ResourceCallback cb) {
  Util.assertMainThread();
  long startTime = LogTime.getLogTime();
  //创建key，这是给每次加载资源的唯一标示。
  EngineKey key = keyFactory.buildKey(model, signature, width, height, transformations,
      resourceClass, transcodeClass, options);
  //从内存缓存中获取资源，获取成功后会放入到activeResources中
  EngineResource cached = loadFromCache(key, isMemoryCacheable);
  if (cached != null) {
    //如果有，那么直接利用当前缓存的资源。
    cb.onResourceReady(cached, DataSource.MEMORY_CACHE);
    if (Log.isLoggable(TAG, Log.VERBOSE)) {
      logWithTimeAndKey("Loaded resource from cache", startTime, key);
    }
    return null;
  }
  //这是一个二级内存的缓存引用，很简单用了一个Map>>装载起来的。
  //从存活的资源中加载资源，资源加载完成后，再将这个缓存数据放到一个 value 为软引用的 activeResources map 中
  EngineResource active = loadFromActiveResources(key, isMemoryCacheable);
  if (active != null) {
    cb.onResourceReady(active, DataSource.MEMORY_CACHE);
    if (Log.isLoggable(TAG, Log.VERBOSE)) {
      logWithTimeAndKey("Loaded resource from active resources", startTime, key);
    }
    return null;
  }
  //根据key获取缓存的job。
  EngineJob current = jobs.get(key);
  if (current != null) {
    current.addCallback(cb);
    if (Log.isLoggable(TAG, Log.VERBOSE)) {
      logWithTimeAndKey("Added to existing load", startTime, key);
    }
    return new LoadStatus(cb, current);
  }
  //创建job
  EngineJob engineJob = engineJobFactory.build(key, isMemoryCacheable,
      useUnlimitedSourceExecutorPool);
  DecodeJob decodeJob = decodeJobFactory.build(
      glideContext,
      model,
      key,
      signature,
```

```
        diskCacheStrategy,
        transformations,
        isTransformationRequired,
        options,
        engineJob);
    jobs.put(key, engineJob);
    engineJob.addCallback(cb);
    //放入线程池，执行
    engineJob.start(decodeJob);
    if (Log.isLoggable(TAG, Log.VERBOSE)) {
      logWithTimeAndKey("Started new load", startTime, key);
    }
    return new LoadStatus(cb, engineJob);
}
```

> 可以看出glide用了两级内存缓存,本人觉得是为了提高命中率吧

load调用处理流程图

> DecodeJob是整个任务的核心部分，在下面DecodeJob中有详细介绍，这里主要整个流程



EngineJob

> 调度DecodeJob，添加，移除资源回调，并notify回调

start(DecodeJob decodeJob)

> 通过线程池调度一个DecodeJob任务

MainThreadCallback

> 实现了Handler.Callback接口，用于Engine任务完成时回调主线程

DecodeJob

> 加载数据，数据加载成功后回调DecodeJob的onDataFetcherReady方法对资源进行处理

入口run()主执行方法

```
private void runWrapped() {
   switch (runReason) {
    case INITIALIZE:
        //初始化 获取下一个阶段状态
      stage = getNextStage(Stage.INITIALIZE);
      currentGenerator = getNextGenerator();
      //运行 load数据
      runGenerators();
      break;
    case SWITCH_TO_SOURCE_SERVICE:
      runGenerators();
      break;
    case DECODE_DATA:
      //处理已经load到的数据
      decodeFromRetrievedData();
      break;
    default:
      throw new IllegalStateException("Unrecognized run reason: " + runReason);
  }
}
```

> 根据不同的runReason执行不同的任务，共两种任务类型：
>
> - runGenerators():load数据
> - decodeFromRetrievedData()：处理已经load到的数据RunReason再次执行任务的原因，三种枚举值：
> - INITIALIZE:第一次调度任务
> - WITCH_TO_SOURCE_SERVICE:本地缓存策略失败，尝试重新获取数据，两种情况；当stage为Stage.SOURCE，或者获取数据失败并且执行和回调发生在了不同的线程
> - DECODE_DATA:获取数据成功，但执行和回调不在同一线程，希望回到自己的线程去处理数据

```
private void runGenerators() {
  currentThread = Thread.currentThread();
```

```
      currentGenerator = getNextGenerator();
      if (stage == Stage.SOURCE) {
        reschedule();
        return;
      }
    }
    // We've run out of stages and generators, give up.
    if ((stage == Stage.FINISHED || isCancelled) && !isStarted) {
      notifyFailed();
    }
    // Otherwise a generator started a new load and we expect to be called back in
    // onDataFetcherReady.
  }
  private Stage getNextStage(Stage current) {
    switch (current) {
      //根据定义的缓存策略来取下一个状态
      //缓存策略来之于RequestBuilder的requestOptions域
      //如果你有自定义的策略，可以调用RequestBuilder.apply方法即可
      //详细的可用缓存策略请参看DiskCacheStrategy.java
      case INITIALIZE:
        return diskCacheStrategy.decodeCachedResource()
            ? Stage.RESOURCE_CACHE : getNextStage(Stage.RESOURCE_CACHE);
      case RESOURCE_CACHE:
        return diskCacheStrategy.decodeCachedData()
            ? Stage.DATA_CACHE : getNextStage(Stage.DATA_CACHE);
      case DATA_CACHE:
        return Stage.SOURCE;
      case SOURCE:
      case FINISHED:
        return Stage.FINISHED;
      default:
        throw new IllegalArgumentException("Unrecognized stage: " + current);
    }
  }
  private DataFetcherGenerator getNextGenerator() {
    switch (stage) {
      case RESOURCE_CACHE:
        return new ResourceCacheGenerator(decodeHelper, this);
      case DATA_CACHE:
        return new DataCacheGenerator(decodeHelper, this);
      case SOURCE:
        return new SourceGenerator(decodeHelper, this);
      case FINISHED:
        return null;
```

getNextStage

> 获取下一步执行的策略，一共5种策略：INITIALIZE，RESOURCE_CACHE，DATA_CACHE，SOURCE，FINISHED
>
> 其中加载数据的策略有三种：RESOURCE_CACHE，DATA_CACHE，SOURCE，分别对应的Generator:
>
> - ResourceCacheGenerator ：尝试从修改过的资源缓存中获取，如果缓存未命中，尝试从DATA_CACHE中获取
> - DataCacheGenerator 尝试从未修改过的本地缓存中获取数据，如果缓存未命中则尝试从SourceGenerator中获取
> - SourceGenerator 从原始的资源中获取，可能是服务器，也可能是本地的一些原始资源策略的配置在DiskCacheStrategy。开发者可通过BaseRequestOptions设置：
> - ALL
> - NONE
> - DATA
> - RESOURCE
> - AUTOMATIC（默认方式，依赖于DataFetcher的数据源和ResourceEncoder的EncodeStrategy）

getNextGenerator

> 根据Stage获取到相应的Generator后会执行currentGenerator.startNext()，如果中途startNext返回true，则直接回调，否则最终会得到SOURCE的stage，重新调度任务

```
public boolean startNext() {
 if (dataToCache != null) {
   Object data = dataToCache;
   dataToCache = null;
   cacheData(data);
 }
 if (sourceCacheGenerator != null && sourceCacheGenerator.startNext()) {
   return true;
 }
 sourceCacheGenerator = null;
```

```
    if (loadData != null
        && (helper.getDiskCacheStrategy().isDataCacheable(loadData.fetcher.getDataSource())
        || helper.hasLoadPath(loadData.fetcher.getDataClass()))) {
      started = true;
      //根据model的fetcher加载数据
      loadData.fetcher.loadData(helper.getPriority(), this);
    }
  }
  return started;
}
```

> 从当前策略对应的Generator获取数据，数据获取成功则回调DecodeJob的onDataFetcherReady对资源进行处理。否则尝试从下一个策略的Generator获取数据。这里的Model必须是实现了GlideModule接口的，fetcher是实现了DataFetcher接口。有兴趣同学可以继续看一下integration中的okhttp和volley工程。Glide主要采用了这两种网络libray来下载图片。

reschedule

> 重新调度当前任务

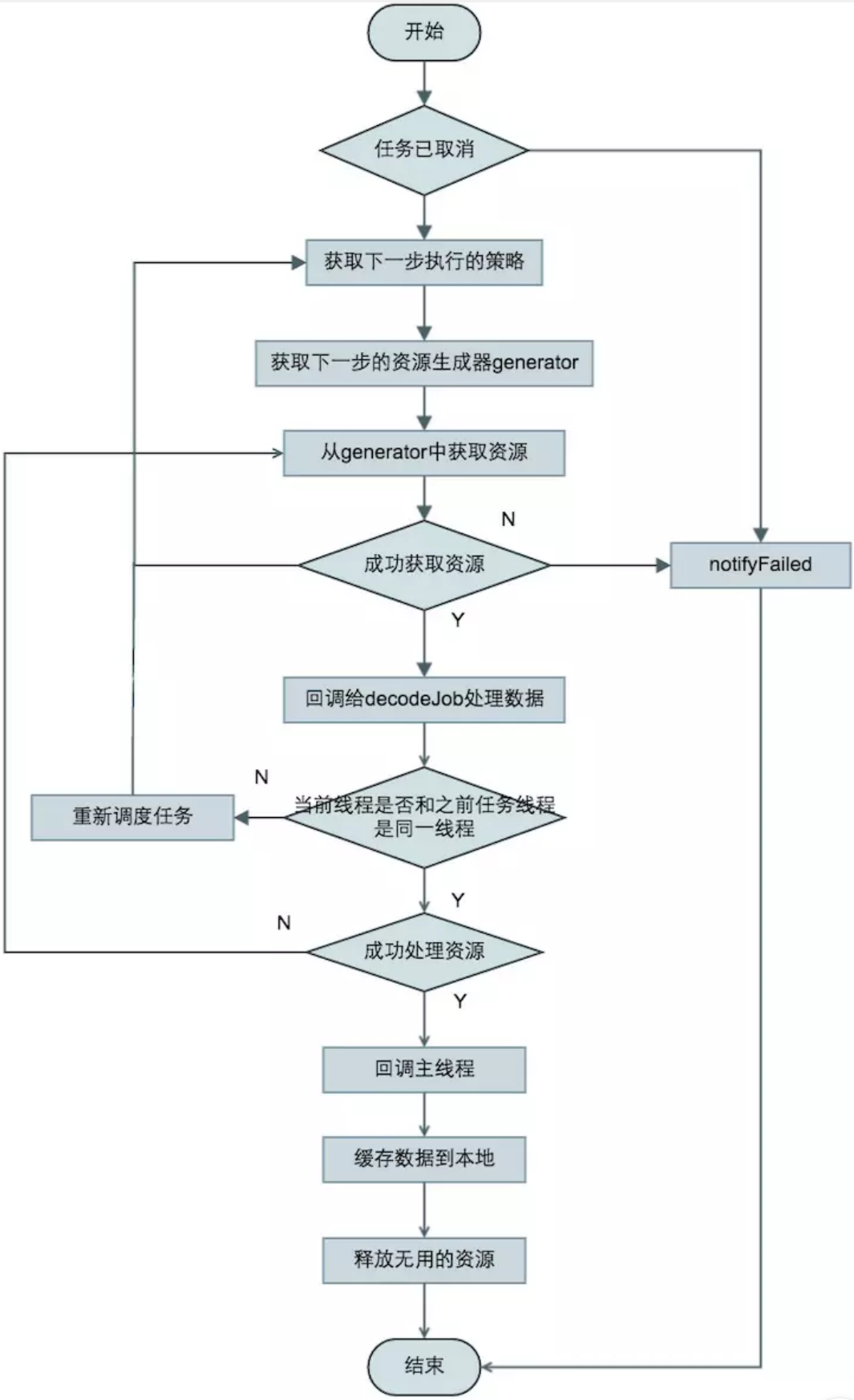
decodeFromRetrievedData

> 获取数据成功后，进行处理，内部调用的是runLoadPath(Data data, DataSource dataSource,LoadPath path)

DecodeCallback.onResourceDecoded

> decode完成后的回调，对decode的资源进行transformpath.load(rewinder, options, width, height,new DecodeCallback(dataSource));

图解



DecodePath

> 根据指定的数据类型对resource进行decode和transcode

RequestTracker

> 追踪，取消，重启失败，正在处理或者已经完成的请求
>
> ### 重要方法
>
> 1. resumeRequests重启所有未完成或者失败的请求，Activity/Fragment的生命周期onStart的时候，会触发RequestManager调用该方法
> 2. pauseRequests停止所有的请求，Activity/Fragment的生命周期onStop的时候，会触发RequestManager调用该方法。
> 3. clearRequests取消所有的请求并清理它们的资源,Activity/Fragment的生命周期onDestory的时候，会触发RequestManager调用该方法。
> 4. restartRequests重启失败的请求，取消并重新启动进行中的请求,网络重新连接的时候，会调用该方法重启请求。
> 5. clearRemoveAndRecycle停止追踪指定的请求，清理，回收相关资源。

TargetTracker

> 持有当前所有存活的Target，并触发Target相应的生命周期方法。方便开发者在整个请求过程的不同状态中进行回调，做相应的处理。

RequestManager

> 核心类之一，用于Glide管理请求。可通过Activity/Fragment/Connectivity（网络连接监听

targetTracker.onStart();回调Target相应周期方法。

2. pauseRequests在onStop方法中调用，其实是通过requestTracker处理，同时也会调用targetTracker.onStop();回调Target相应周期方法
3. onDestroy调用targetTracker.onDestroy();，requestTracker.clearRequests();，lifecycle.removeListener(this);等进行资源清理。
4. resumeRequestsRecursive递归重启所有RequestManager下的所有request。在Glide中源码中没有用到，暴露给开发者的接口。
5. pauseRequestsRecursive递归所有childFragments的RequestManager的pauseRequest方法。同样也只是暴露给开发者的接口。childFragments表示那些依赖当前Activity或者Fragment的所有fragments

如果当前Context是Activity，那么依附它的所有fragments的请求都会中止如果当前Context是Fragment，那么依附它的所有childFragment的请求都会中止如果当前的Context是ApplicationContext，或者当前的Fragment处于detached状态，那么只有当前的RequestManager的请求会被中止注意：在Android 4.2 AP17之前，如果当前的context是Fragment（当fragment的parent如果是activity，fragment.getParentFragment()直接返回null），那么它的childFragment的请求并不会被中止。原因是在4.2之前系统不允许获取parent fragment，因此不能确定其parentFragment。 但v4的support Fragment是可以的，因为v4包的Fragment对应的SupportRequestManagerFragment提供了一个parentFragmentHint，它相当于Fragment的ParentFragment。在RequestManagerRetriever.get(support.V4.Fragment fragment)的时候将参数fragment作为parentFragmentHint。

1. registerFragmentWithRoot获取Activity相应的RequestManagerFragment，并添加到Activity的事务当中去，同时将当前的Fragment添加到childRequestManagerFragments的HashSet集合中去，以便在pauseRequestsRecursive和resumeRequestsRecursive方法中调用RequestManagerTreeNode.getDescendants()的时候返回所有的childFragments。在RequestManagerFragment的onAttach方法以及setParentFragmentHint方法中调用。
2. unregisterFragmentWithRoot对应上面的registerFragmentWithRoot方法，在RequestManagerFragment的onDetach，onDestroy或者重新register前将当前的fragment进行remove

很重要的一个相关类:RequestManagerFragment。当Glide.with(context)获取RequestManager的时候，Glide都会先尝试获取当前上下文相关的RequestManagerFragment。

的纽带。RequestManagerFragment生命周期方法触发的时候，就可以通过ActivityFragmentLifecycle同时触发RequestManager相应的方法，执行相应的操作。

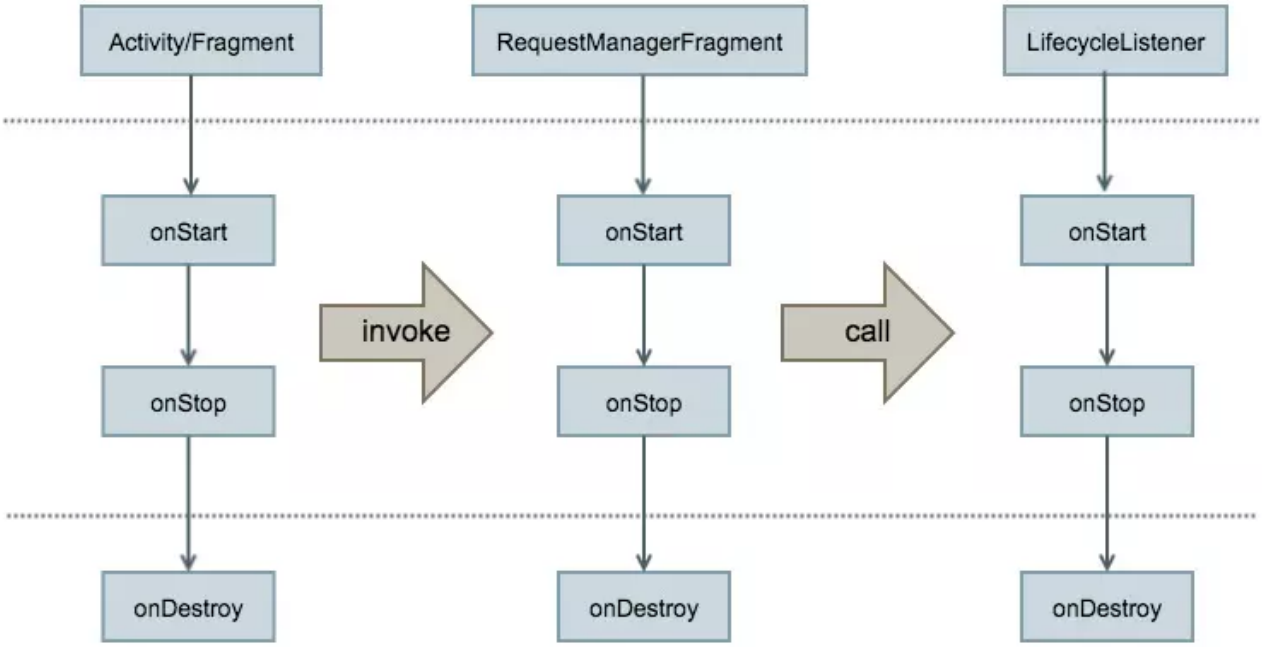
Request Manager通过ActivityFragmentLifecycle的addListener方法注册一些LifecycleListener。当RequestManagerFragment生命周期方法执行的时候，触发ActivityFragmentLifecycle的相应方法，这些方法会遍历所有注册的LifecycleListener并执行相应生命周期方法。

RequestManager注册的LifecycleListener类型

- RequestManager自身RequestManager自己实现了LifecycleListener。主要的请求管理也是在这里处理的。
- RequestManagerConnectivityListener，该listener也实现了LifecycleListener，用于网络连接时进行相应的请求恢复。 这里的请求是指那些还未完成的请求，已经完成的请求并不会重新发起。另外Target接口也是直接继承自LifecycleListener，因此RequestManager在触发相应的生命周期方法的时候也会调用所有Target相应的生命周期方法，这样开发者可以监听资源处理的整个过程，在不同阶段进行相应的处理。

生命周期的管理主要由RequestTracker和TargetTracker处理。

生命周期事件的传递



RequestManagerFragment

> 发listener相应的生命周期方法。复写了onLowMemory和onTrimMemory，低内存情况出现的时候，会调用RequestManager的相应方法进行内存清理。
>
> 释放的内存有：
>
> - bitmapPool：
> - memoryCache：
> - byteArrayPool：

RequestManagerRetriever

> 提供一些静态方法，用语创建或者从Activity/Fragment获取RequestManager。get(Activity activity)get(android.app.Fragment fragment)get(Activity activity)get(FragmentActivity activity)getSupportRequestManagerFragment

RequestManagerTreeNode

> 上文提到获取所有childRequestManagerFragments的RequestManager就是通过该类获得，就一个方法：getDescendants，作用就是基于给定的Context，获取所有层级相关的RequestManager。上下文层级由Activity或者Fragment获得，ApplicationContext的上下文不会提供RequestManager的层级关系，而且Application生命周期过长，所以Glide中对请求的控制只针对于Activity和Fragment。

LifecycleListener

> 用于监听Activity或者Fragment的生命周期方法的接口，基本上请求相关的所有类都实现了该接口
>
> - void onStart();
> - void onStop();
> - void onDestroy();

DataFetcher

每一次通过ModelLoader加载资源的时候都会创建的实例。loadData ：异步方法，如果目标资源没有在缓存中找到时才会被调用,cancel方法也是。cleanup：清理或者回收DataFetcher使用的资源，在loadData提供的数据被decode完成后调用。

主要方法

1. DataCallback用于数据加载结果的回调,三种Generator实现了该接口

```
//数据load完成并且可用时回调
void onDataReady(@Nullable T data);
//数据load失败时回调
void onLoadFailed(Exception e);
```

2. getDataClass()返回fetcher尝试获取的数据类型

3. getDataSource()获取数据的来源
4. DataSource

```
public enum DataSource {
  //数据从本地硬盘获取，也有可能通过一个已经从远程获取到数据的Content Provider
  LOCAL,
  //数据从远程获取
  REMOTE,
  //数据来自未修改过的硬盘缓存
  DATA_DISK_CACHE,
  //数据来自已经修改过的硬盘缓存
  RESOURCE_DISK_CACHE,
  //数据来自内存
  MEMORY_CACHE,
}
```

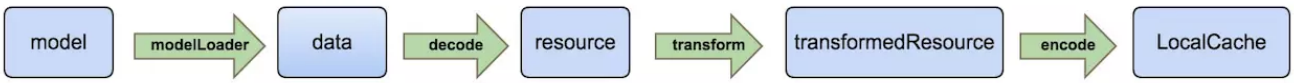
DataFetcherGenerator

> startNext当Generator从DataFetcher完成loadData时回调，含有的方法：
> onDataFetcherReady：load完成onDataFetcherFailed：load失败

Registry

> 管理组件（数据类型 + 数据处理）的注册
>
> ### 主要成员变量
>
> - ModelLoaderRegistry ：注册所有数据加载的loader
> - ResourceDecoderRegistry：注册所有资源转换的decoder
> - TranscoderRegistry：注册所有对decoder之后进行特殊处理的transcoder
> - ResourceEncoderRegistry：注册所有持久化resource（处理过的资源）数据的encoder
> - EncoderRegistry ： 注册所有的持久化原始数据的encoder

标准的数据处理流程：



Glide在初始化的时候，通过Registry注册以下所有组件， 每种组件由功能及处理的资源类型组成：

loader	model + data + ModelLoaderFactory
decoder	dataClass + resourceClass + decoder
transcoder	resourceClass + transcodeClass
encoder	dataClass + encoder
resourceEncoder	resourceClass + encoder
rewind	缓冲区处理

Decoder	**数据源**	**解码后的资源**
BitmapDrawableDecoder	Bitmap	Drawable
StreamBitmapDecoder	InputStream	Bitmap
ByteBufferBitmapDecoder	ByteBuffer	Bitmap
GifFrameResourceDecoder	GifDecoder	Bitmap
StreamGifDecoder	InputStream	GifDrawable
ByteBufferGifDecoder	ByteBuffer	Gif
SvgDecoder	InputStream	SVG
VideoBitmapDecoder	ParcelFileDescriptor	Bitmap
FileDecoder	File	file

BitmapBytesTranscoder	Bitmap	Bytes
BitmapDrawableTranscoder	Bitmap	Drawable
GifDrawableBytesTranscoder	GifDrawable	Bytes
SvgDrawableTranscoder	Svg	Drawable

> LoadPath是对decodePath的封装，持有一个decodePath的List。在通过modelloader.fetchData获取到data后，会对data进行decode，具体的decode操作就是通过loadPath来完成。resourceClass就是asBitmap，asDrawable方法的参数

ModelLoaderRegistry

> 持有多个ModelLoader，model和数据类型按照优先级进行处理loader注册示例：

```
registry
.append(Integer.class, InputStream.class, new ResourceLoader.StreamFactory())
.append(GifDecoder.class, GifDecoder.class, new UnitModelLoader.Factory())
```

主要函数

1. register，append，prepend注册各种功能的组件
2. getRegisteredResourceClasses(Class modelClass, Class resourceClass, Class transcodeClass)获取Glide初始化时注册的所有resourceClass
3. getModelLoaders(Model model)
4. hasLoadPath(Class<?> dataClass)判断注册的组件是否可以处理给定的dataClass
    - 直接调用getLoadPath(dataClass, resourceClass, transcodeClass)
    - 该方法先从loadPathCache缓存中尝试获取LoadPath,如果没有，则先根据dataClass, resourceClass, transcodeClass获取所有的decodePaths，如果decodePaths不为空，则创建一个LoadPath<>(dataClass, resourceClass, transcodeClass, decodePaths,exceptionListPool) 并缓存起来。
5. getDecodePaths根据dataClass, resourceClass, transcodeClass从注册的组件中找到

到符合条件的decoder和transcoder然后创建DecodePath。

ModelLoader

ModelLoader是一个工厂接口。将任意复杂的model转换为准确具体的可以被DataFetcher获取的数据类型。每一个model内部实现了一个ModelLoaderFactory，内部实现就是将model转换为Data

重要成员

LoadDataKey sourceKey，用于表明load数据的来源。List alternateKeys：指向相应的变更数据DataFetcher fetcher：用于获取不在缓存中的数据

重要方法

1. buildLoadData返回一个LoadData
2. handles(Model model)判断给定的model是否可以被当前modelLoader处理

ModelLoaderFactory

根据给定的类型，创建不同的ModelLoader，因为它会被静态持有，所以不应该维持非应用生命周期的context或者对象。

DataFetcherGenerator

通过注册的DataLoader生成一系列的DataFetcherDataCacheGenerator：根据未修改的缓存数据生成DataFetcherResourceCacheGenerator：根据已处理的缓存数据生成DataFetcherSourceGenerator：根据原始的数据和给定的model通过ModelLoader生成的DataFetcher

如何监测当前context的生命周期

> 为当前的上下文Activity或者Fragment绑定一个TAG为”com.bumptech.glide.manager”的RequestManagerFragment，然后把该fragment作为rootRequestManagerFragment，并加入到当前上下文的FragmentTransaction事务中，从而与当前上下文Activity或者Fragment的生命周期保持一致。
>
> 关键就是RequestManagerFragment，用于绑定当前上下文以及同步生命周期。比如当前的context为activity，那么activity对应的RequestManagerFragment就与宿主activity的生命周期绑定了。同样Fragment对应的RequestManagerFragment的生命周期也与宿主Fragment保持一致。

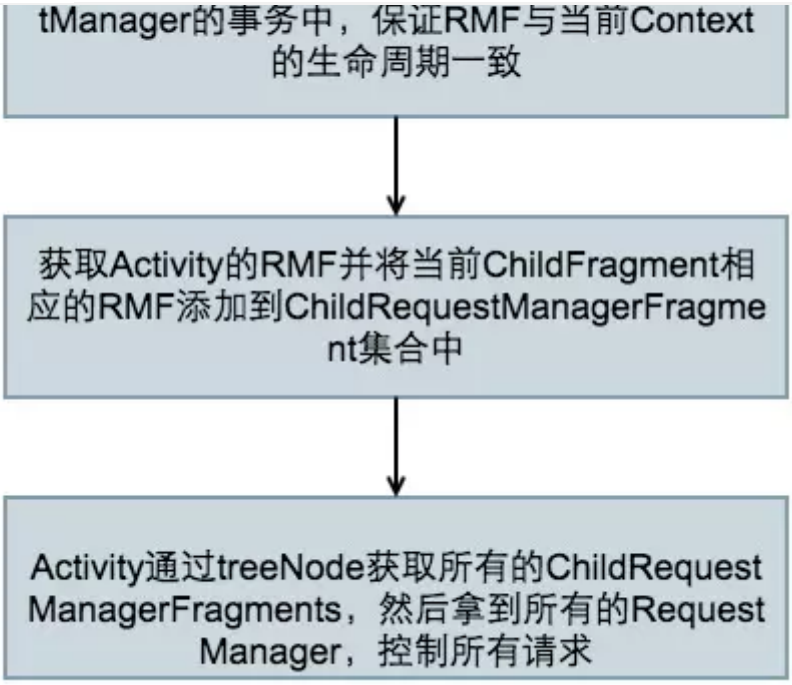
请求管理的实现

> pauseRequests，resumeRequests在RequestManagerFragment对应Request Manager的生命周期方法中触发，

如何控制当前上下文的所有ChildFragment的请求？

情景：

> 假设当前上下文是Activity（Fragment类似）创建了多个Fragment，每个Fragment通过Glide.with(fragment.this)方式加载图片



- 首先Glide会为Activity以及每一个Fragment创建一个RequestManagerFragment（原因看下面）并提交到当前上下文的事务中。以上保证了每个Fragment以及对应的RequestManagerFragment生命周期是与Activity的生命周期绑定的。
- 在RequestManagerFragment的onAttach方法中通过Glide.with(activity.this)先获得Activity（宿主）的RequestManagerFragment(rootRequestManagerFragment)，并将每个Fragment相应的RequestManagerFragment添加到childRequestManagerFragments集合中。
- Activity通过自己的RequestManager的childRequestManagerFragments获取所有childFragment的RequestManager，然后对请求进行pause，resume。同理，如果当前context是Fragment，Fragment对应的RequestManagerFragment可以获取它自己所有的Child Fragment的RequestManagerFragment。

如何管理没有ChildFragment的请求？

> 很简单，只会存在当前context自己的RequestManagerFragment，那么伴随当前上下文的生命周期触发，会调用RequestManagerFragment的RequestManager相应的lefecycle方法实现请求的控制，资源回收。

为何每一个上下文会创建自己的RequestManagerFragment ？

- 如果传入到Glide.with(...)的context是activityfm = activity.getSupportFragmentManager();
- 如果传入到Glide.with(...)的context是Fragmentfm = fragment.getChildFragmentManager();因为上下文不同导致得到的fm不同，从而RequestManagerRetriever.getSupportRequestManagerFragment(fm)方法返回的RequestManagerFrament不同。而且如果一个activity下面有多个Fragment，并以Glide.with(fragment.this)的方式加载图片。那么每个Fragment都会为自己创建一个fm相关的RequestManagerFragment。

关键在于每一个上下文拥有一个自己的RequestManagerFragment。而传入的context不同，会返回不同的RequestManagerFragment，顶层上下文会保存所有的childRequestManagerFragments。

杂谈

> Glide优点在于其生命周期的管理，资源类型的支持多。但相对于简洁的UniversalImageLoader和Picasso，无论从设计上还是细节实现上，都复杂的多，从代码的实现上可以看出，正式因为Glide的生命周期管理，内存友好，资源类型支持多这些优点相关。一些设计概念很少碰到，比如decodePath，loadpath。整个数据处理流程的拆分三个部分，每个部分所支持的数据以及处理方式全部通过组件注册的方式来支持，很多方法或者构造函数会接收10多个参数，看着着实眼花缭乱。这里的分析把大体的功能模块分析了，比如请求的统一管理，生命周期的同步，具体的实现细节还需要一部分的工作量。对于开源项目的初学者来说，Glide并不是一个好的项目，门槛太高。也因为如此，所以Glide的使用并没有其它几种图片库的使用那么广泛，相关文档很欠缺，本篇分析希望成为一个很好的参考，也希望大家提出自己的建议和意见，继续优化，让更多开发者能更快了解，使用这个强大的库。

本文标题:Glide源码分析

文章作者:Hpw123

发布时间:2016-12-30, 13:39:26

最后更新:2016-12-30, 22:19:56

原始链接:hpw123.coding.me/2016/12/30/