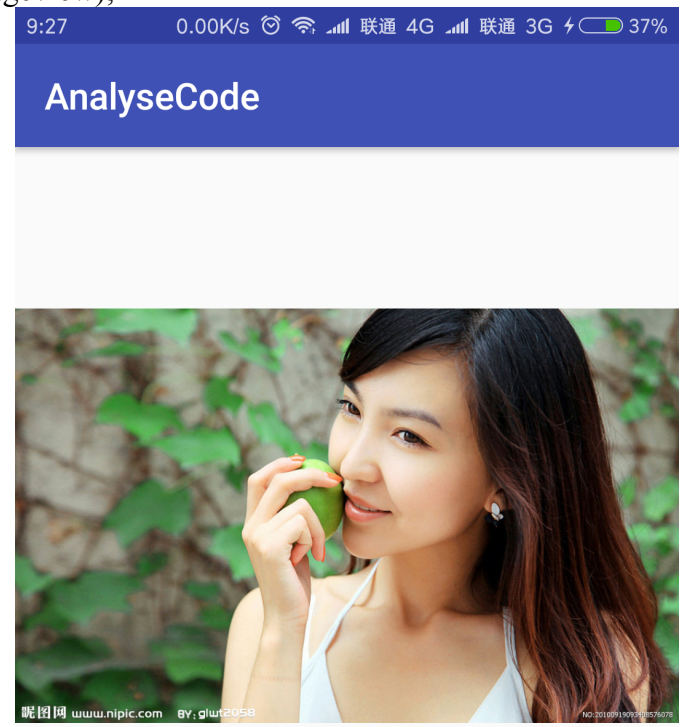
glide 源码分析

李保成

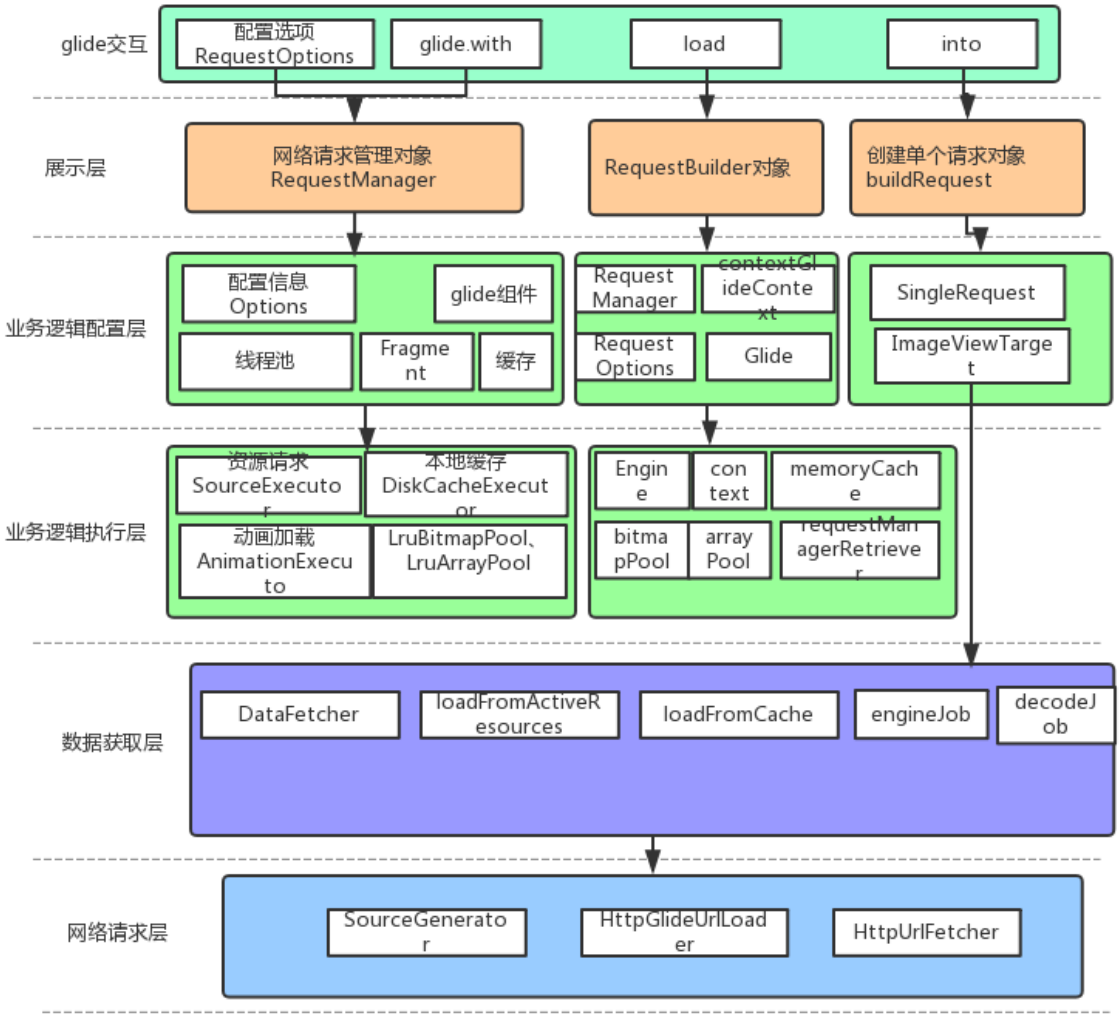
---

glide 是 Bump 出品的一个图片加载库
开源地址：https://github.com/bumptech/glide
glide 使用方法如下：

```
Glide.with(this).load("http://pic9.nipic.com/20100919/5123760_093408576078_2.jpg")
            .into(mImageview);
```



先看一下 glide 的总体框架：



with

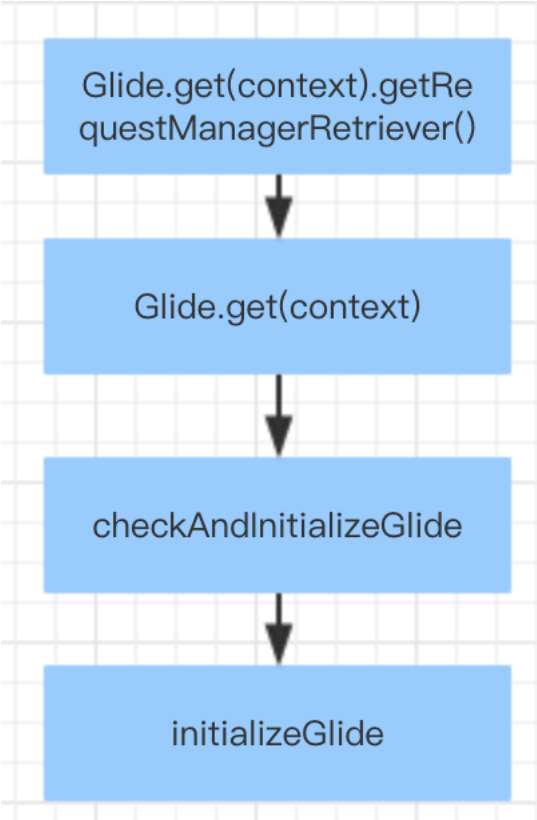
先看一下 with 方法：

```
public static RequestManager with(@NonNull FragmentActivity activity) {
  return getRetriever(activity).get(activity);
}
```

作用：初始化 glide，返回一个 RequestManger 对象

以传入的是 FragmentActivity 为例：

getRetriever （activity）流程如下：



initializeGlide 主要做了以下工作：

1. 获取应用中带注解的 GlideModule（annotationGeneratedModule），这里要解释一下 GlideModule：用户自定义 glide 配置模块，用来修改默认的 glide 配置信息。如果这个为空或者可配置 menifest 里面的标志为 true，则获取 menifest 里面配置的 GlideModule 模块（manifestModules）。

2. 把 manifestModules 以及 annotationGeneratedModule 里面的配置信息放到 builder 里面（applyOptions）替换 glide 默认组件（registerComponents）

3. 各种初始化信息 Glide glide = builder.build(applicationContext);

4. 看一下 build 的源码：

   主要做了以下工作：

   1）创建请求图片线程池 sourceExecutor，创建硬盘缓存线程池 diskCacheExecutor。动画线程池 animationExecutor

   2）依据设备的屏幕密度和尺寸设置各种 pool 的 size

3）创建图片线程池 LruBitmapPool，缓存所有被释放的 bitmap, LruBitmapPool 依赖默认的缓存策略和缓存配置。缓存策略在 API 大于 19 时，为 SizeConfigStrategy，小于为 AttributeStrategy。其中 SizeConfigStrategy 是以 bitmap 的 size 和 config 为 key，value 为 bitmap 的 HashMap。

4）创建对象数组缓存池 LruArrayPool，默认 4M

5）创建 LruResourceCache，内存缓存

6）new glide 里面 new Registry()注册管理任务执行对象的类(Registry)，可以简单理解为：Registry 是一个工厂，而其中所有注册的对象都是一个工厂员工，当任务分发时，根据当前任务的性质，分发给相应员工进行处理。

```
Glide build(@NonNull Context context) {
  if (sourceExecutor == null) {
    sourceExecutor = GlideExecutor.newSourceExecutor();
  }

  if (diskCacheExecutor == null) {
    diskCacheExecutor = GlideExecutor.newDiskCacheExecutor();
  }

  if (animationExecutor == null) {
    animationExecutor = GlideExecutor.newAnimationExecutor();
  }

  if (memorySizeCalculator == null) {
    memorySizeCalculator = new MemorySizeCalculator.Builder(context).build();
  }

  if (connectivityMonitorFactory == null) {
    connectivityMonitorFactory = new DefaultConnectivityMonitorFactory();
  }

  if (bitmapPool == null) {
    int size = memorySizeCalculator.getBitmapPoolSize();
    if (size > 0) {
      bitmapPool = new LruBitmapPool(size);
    } else {
      bitmapPool = new BitmapPoolAdapter();
    }
  }

  if (arrayPool == null) {
    arrayPool = new LruArrayPool(memorySizeCalculator.getArrayPoolSizeInBytes());
  }

  if (memoryCache == null) {
    memoryCache = new LruResourceCache(memorySizeCalculator.getMemoryCacheSize());
  }

  if (diskCacheFactory == null) {
    diskCacheFactory = new InternalCacheDiskCacheFactory(context);
  }

  if (engine == null) {
    engine =
        new Engine(
```

```
            memoryCache,
            diskCacheFactory,
            diskCacheExecutor,
            sourceExecutor,
            GlideExecutor.newUnlimitedSourceExecutor(),
            GlideExecutor.newAnimationExecutor(),
            isActiveResourceRetentionAllowed);
  }

  RequestManagerRetriever requestManagerRetriever =
      new RequestManagerRetriever(requestManagerFactory);

  return new Glide(
      context,
      engine,
      memoryCache,
      bitmapPool,
      arrayPool,
      requestManagerRetriever,
      connectivityMonitorFactory,
      logLevel,
      defaultRequestOptions.lock(),
      defaultTransitionOptions);
}
```

glide 参数含义：

context 上下文，engine：任务和资源管理（线程池，内存缓存和硬盘缓存对象），memoryCache：内存缓存，bitmapPool:bitmap 内存缓存，后续会单独介绍。

arrayPool: connectivityMonitorFactory:回调监听，defaultRequestOptions：默认请求配置，

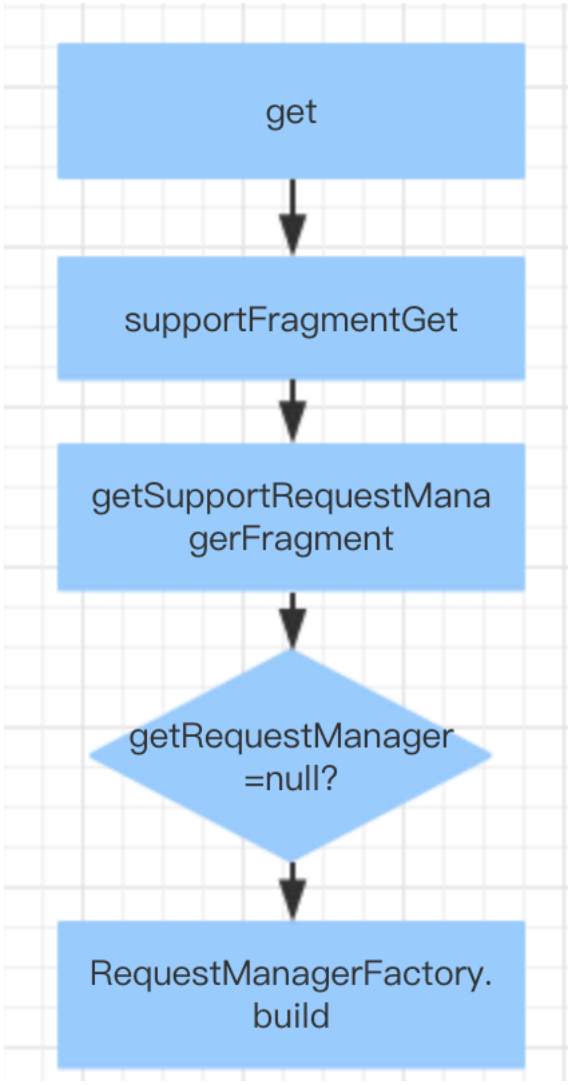
defaultTransitionOptions：默认过度效果

```
getRetriever(activity).get(activity)
```

然后我们看一下 get(activity)的流程：

```
public RequestManager get(@NonNull FragmentActivity activity) {
  if (Util.isOnBackgroundThread()) {
    return get(activity.getApplicationContext());
  } else {
    assertNotDestroyed(activity);
    FragmentManager fm = activity.getSupportFragmentManager();
    return supportFragmentGet(
        activity, fm, /*parentHint=*/ null, isActivityVisible(activity));
  }
}
```

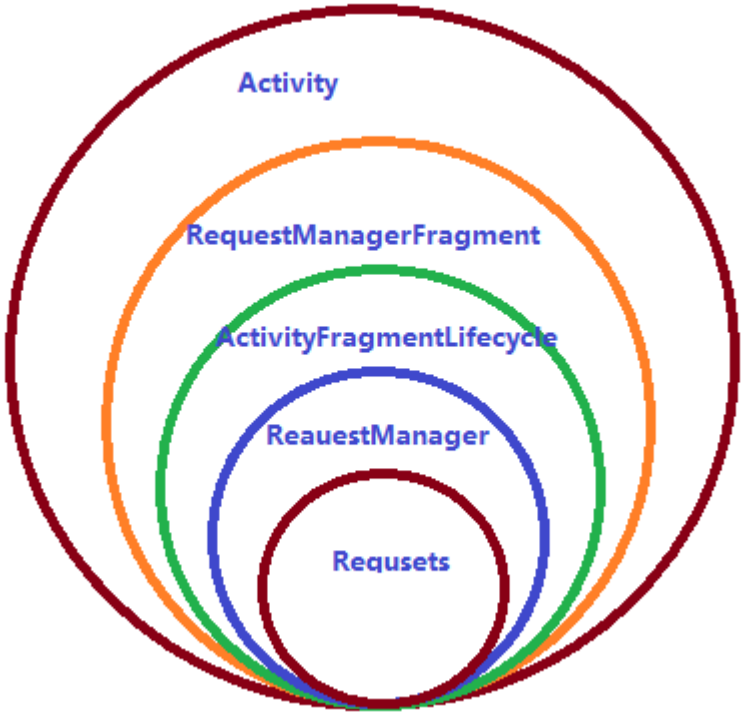
流程图如下：



主要工作是，创建一个 supportFragment（SupportRequestManagerFragment），把 Request 和 Fragment 绑定在一起，主要是生命周期。

引用
https://xiaodanchen.github.io/2016/08/19/%E8%B7%9F%E7%9D%80%E6%BA%90%E7%A0%81%E5%AD%A6%E8%AE%BE%E8%AE%A1%EF%BC%9AGlide%E6%A1%86%E6%9E%B6%E5%8F%8A%E6%BA%90%E7%A0%81%E8%A7%A3%E6%9E%90%EF%BC%88%E4%B8%80%EF%BC%89/

文章中的一个图片：



总结以下 with 中的工作主要有以下几点：

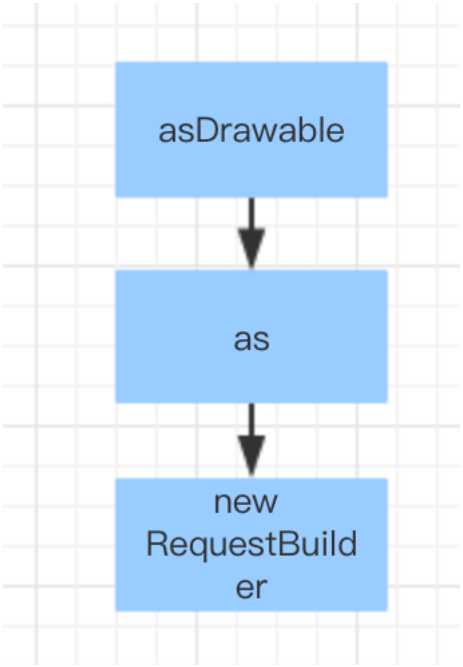
1. 初始化配置信息（包括缓存，请求线程池，大小，图片格式等等）以及 glide 组件，

2. 将 glide 和 Fragment 的生命周期绑定在一块。

Load

```
asDrawable().load(string);
```

先分析一下 asDrawable:



最后是创建了一个 RequestBuilder 对象

```
private RequestBuilder<TranscodeType> loadGeneric(@Nullable Object model) {
   this.model = model;
   isModelSet = true;
   return this;
 }
```

最终返回 RequestBuider 对象

into

```
private <Y extends Target<TranscodeType>> Y into(
    @NonNull Y target,
    @Nullable RequestListener<TranscodeType> targetListener,
    @NonNull RequestOptions options) {
  Util.assertMainThread();
  Preconditions.checkNotNull(target);
  if (!isModelSet) {
    throw new IllegalArgumentException("You must call #load() before calling
#into()");
  }

  options = options.autoClone();
  Request request = buildRequest(target, targetListener, options);

  Request previous = target.getRequest();
  if (request.isEquivalentTo(previous)
      && !isSkipMemoryCacheWithCompletePreviousRequest(options, previous)) {
    request.recycle();
    // If the request is completed, beginning again will ensure the result is re-
delivered,
    // triggering RequestListeners and Targets. If the request is failed, beginning
again will
    // restart the request, giving it another chance to complete. If the request is
already
    // running, we can let it continue running without interruption.
    if (!Preconditions.checkNotNull(previous).isRunning()) {
      // Use the previous request rather than the new one to allow for optimizations
like skipping
      // setting placeholders, tracking and un-tracking Targets, and obtaining View
dimensions
      // that are done in the individual Request.
      previous.begin();
    }
```

```
    return target;
  }

  requestManager.clear(target);
  target.setRequest(request);
  requestManager.track(target, request);

  return target;
}
```

构建 Request 对象，实际是在 buildRequestRecursive 里面创建了一个 buildThumbnailRequestRecursive 的对象以及 errorRequestCoordinator（异常处理对象）

最后调用 requestManager.track(target, request);

track 干了两件事：

```
void track(@NonNull Target<?> target, @NonNull Request request) {
  targetTracker.track(target);
  requestTracker.runRequest(request);
}
```

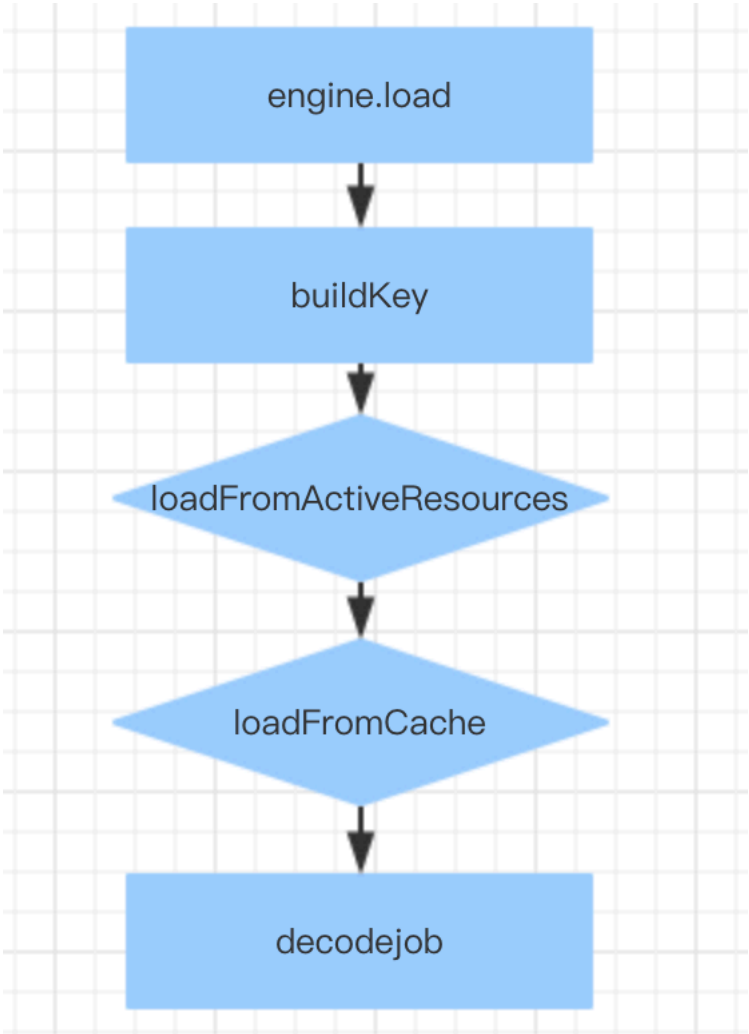
1. 加入 target 目标队列（view）

2. 加入请求 Request 队列，如果缓存中没有开始请求数据

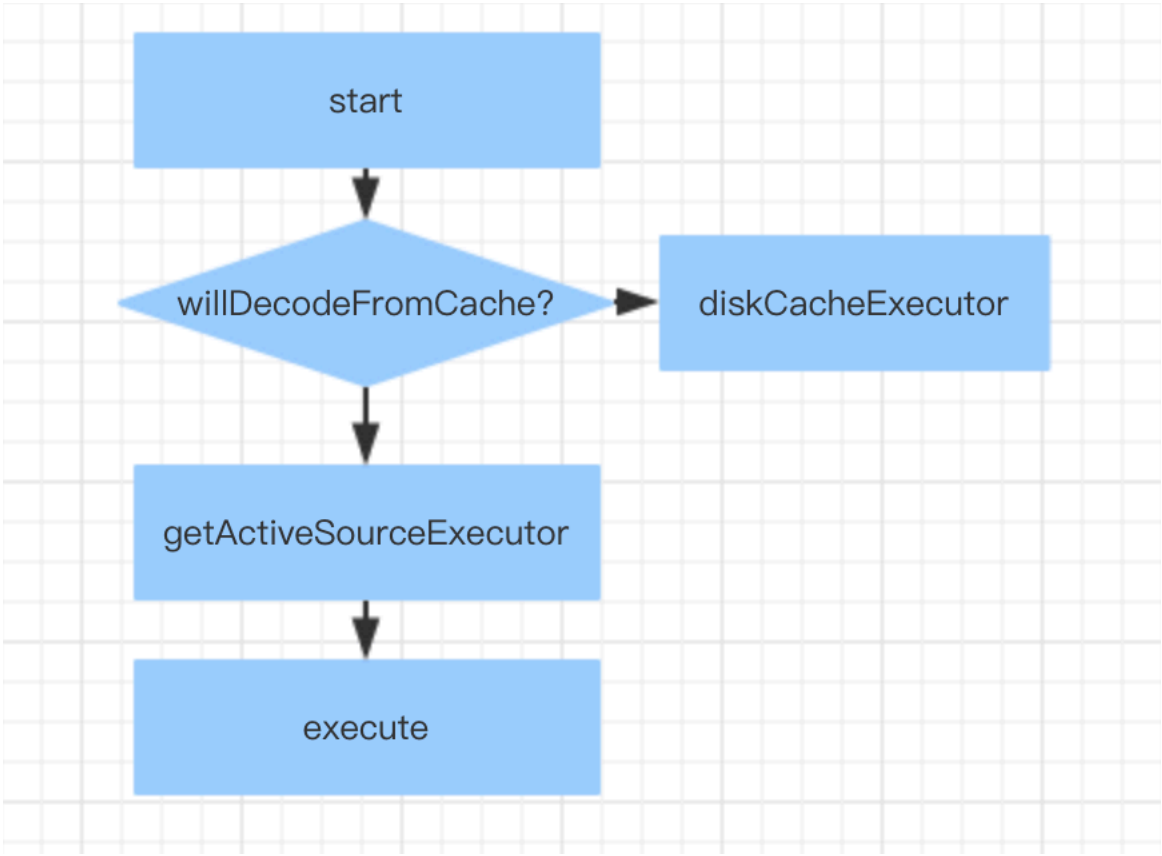
```
3.  @Override
    public void begin() {
      assertNotCallingCallbacks();
      stateVerifier.throwIfRecycled();
      startTime = LogTime.getLogTime();
      if (model == null) {
        if (Util.isValidDimensions(overrideWidth, overrideHeight)) {
          width = overrideWidth;
          height = overrideHeight;
        }
        // Only log at more verbose log levels if the user has set a fallback
    drawable, because
        // fallback Drawables indicate the user expects null models occasionally.
        int logLevel = getFallbackDrawable() == null ? Log.WARN : Log.DEBUG;
        onLoadFailed(new GlideException("Received null model"), logLevel);
        return;
      }

      if (status == Status.RUNNING) {
        throw new IllegalArgumentException("Cannot restart a running request");
      }

      // If we're restarted after we're complete (usually via something like a
    notifyDataSetChanged
      // that starts an identical request into the same Target or View), we can
    simply use the
      // resource and size we retrieved the last time around and skip obtaining a new
    size, starting a
      // new load etc. This does mean that users who want to restart a load because
    they expect that
      // the view size has changed will need to explicitly clear the View or Target
    before starting
      // the new load.
      if (status == Status.COMPLETE) {
        onResourceReady(resource, DataSource.MEMORY_CACHE);
        return;
      }

      // Restarts for requests that are neither complete nor running can be treated
    as new requests
      // and can run again from the beginning.
```

```
    status = Status.WAITING_FOR_SIZE;
    if (Util.isValidDimensions(overrideWidth, overrideHeight)) {
      onSizeReady(overrideWidth, overrideHeight);
    } else {
      target.getSize(this);
    }

    if ((status == Status.RUNNING || status == Status.WAITING_FOR_SIZE)
        && canNotifyStatusChanged()) {
      target.onLoadStarted(getPlaceholderDrawable());
    }
    if (IS_VERBOSE_LOGGABLE) {
      logV("finished run method in " + LogTime.getElapsedMillis(startTime));
    }
  }
```

begin 在 onSizeReady 执行 engine.load, 先从弱引用中查找 loadFromActiveResources()，如果有的话直接返回，没有再从内存中查找 loadFromCache, 有的话会取出并放到 ActiveResources 里面，如果内存中没有，则创建 engineJob（decodejob 的回调类，管理下载过程以及状态）线程 decodeJob，先下载，后解析，并把 Job 放到 Hashmap 里面。流程图如下：



开启线程是从 engineJob.start 开始的，流程图如下：



看一下 DecodeJob 线程的 run 方法，实际起作用的是 runWrapped 方法：

```
private void runWrapped() {
  switch (runReason) {
    case INITIALIZE:
      stage = getNextStage(Stage.INITIALIZE);
      currentGenerator = getNextGenerator();
      runGenerators();
      break;
    case SWITCH_TO_SOURCE_SERVICE:
      runGenerators();
      break;
    case DECODE_DATA:
      decodeFromRetrievedData();
      break;
    default:
      throw new IllegalStateException("Unrecognized run reason: " + runReason);
  }
}
```

完整执行的情况下，会依次调用 ResourceCacheGenerator、DataCacheGenerator 和 SourceGenerator 中的 startNext()

首次下载图片创建的是 SourceGenerator：

runGenerators 流程如下：

```
private void runGenerators() {
  currentThread = Thread.currentThread();
  startFetchTime = LogTime.getLogTime();
  boolean isStarted = false;
  while (!isCancelled && currentGenerator != null
      && !(isStarted = currentGenerator.startNext())) {
    stage = getNextStage(stage);
    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.
}
```

调用了 SourceGenerator 的 startNext 方法：

1、dataToCache 数据不为空的话缓存到硬盘（非第一次）

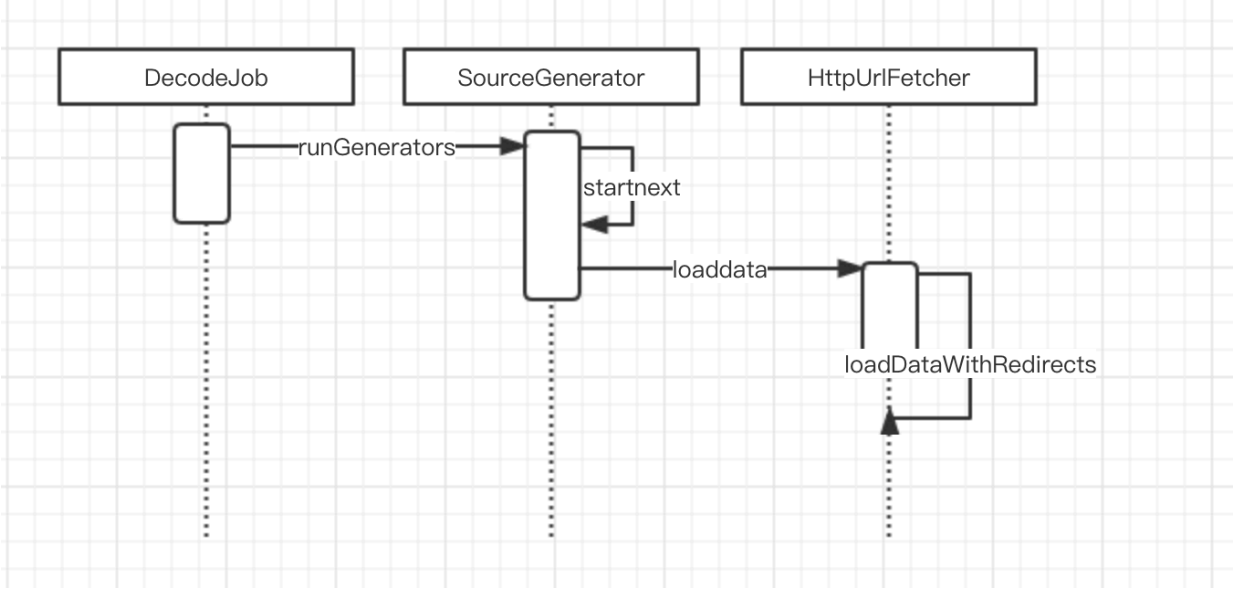
2、 在 modelLoaders 里面找到 ModelLoder 对象（每个 Generator 对应一个 ModelLoader）

3、通过（HttpGlideUrlLoader）buildLoadData 获取到实际的 loadData 对象（key 为 URL，value 创建的 HttpUrlFetcher 对象）

4、通过 loadData 对象的 fetcher 对象（HttpUrlFetcher）的 loadData 方法来获取图片数据。

5、HttpUrlFetcher 通过 HttpURLConnection 网络请求数据

时序图如下：



参考：

https://blog.csdn.net/guolin_blog/article/details/78357251

https://xiaodanchen.github.io/2016/08/22/%E8%B7%9F%E7%9D%80%E6%BA%90%E7%A0%81%E5%AD%A6%E8%AE%BE%E8%AE%A1%EF%BC%9AGlide%E6%A1%86%E6%9E%B6%E5%8F%8A%E6%BA%90%E7%A0%81%E8%A7%A3%E6%9E%90%EF%BC%88%E4%B8%89%EF%BC%89/

https://blog.csdn.net/zsago/article/details/57964228