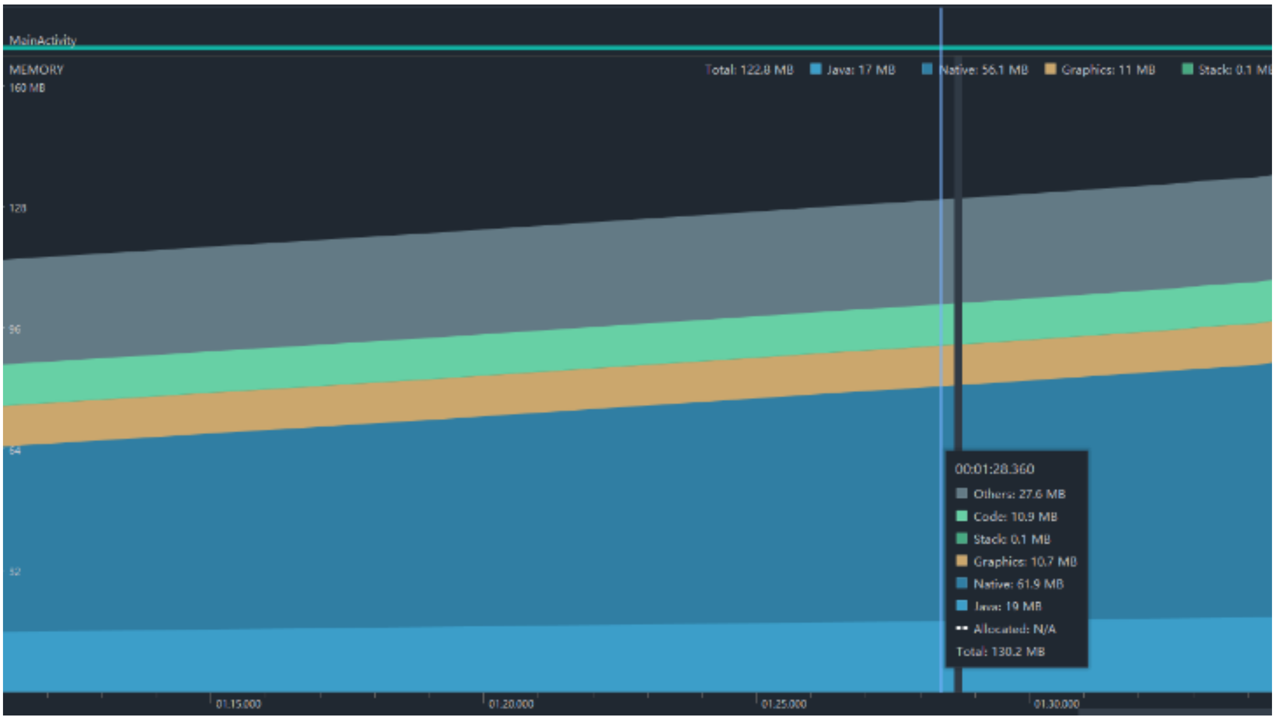
技术背景

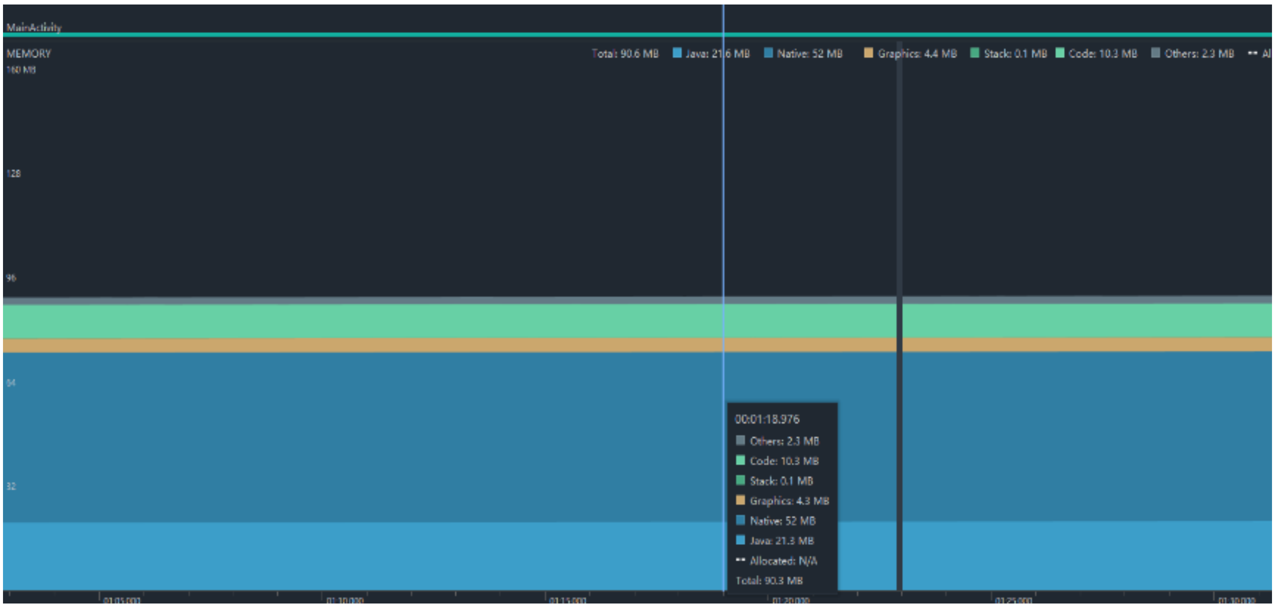
> 1.项目需要加载大量gif图片，众所周知glide自带加载gif功能;
>
> ```
> Glide.with(context).asGif().load(path).into(iv);
> ```
>
> 2.但是真实使用到项目中 glide加载gif会占用大量内存导致应用卡顿，严重的会奔溃。
>
> 3.查看glide源码发现glide加载gif图片，使用java解码，所以导致内存增高。
>
> 4.加载本地gif的三方库（ android-gif-drawable ），用giflib来解码gif，但是他这个库只能加载本地图片，
>
> 5.而我们项目需要加载网络图片，所以就想把glide和giflib做一个结合，使用glide下载图片，bitmap缓存的功能，解码器替换成giflib，经过一天研究终于成功了，写文章记录一下。
>
> ```
> Glide.with(context).as(FrameSequenceDrawable.class).load(path).into(iv);
> ```

性能对比

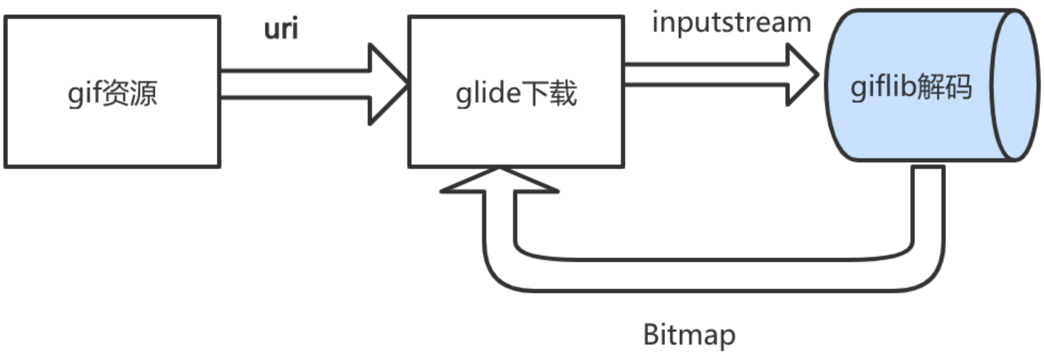
Glide加载8张gif图片



giflib加载gif



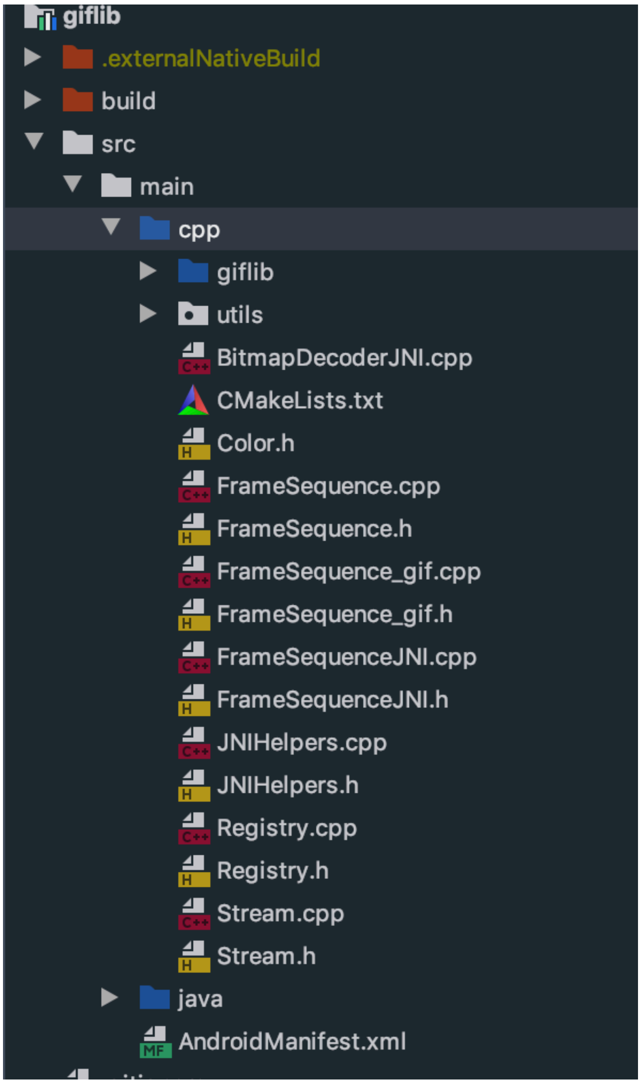
工作流程



开发集成步骤

首先需要下载framesequence(https://android.googlesource.com/platform/frameworks/ex/+/android-9.0.0_r16/framesequence/)及giflib

- 目录如下：



```
giflib
▶ .externalNativeBuild
▶ build
▼ src
  ▼ main
    ▼ cpp
      ▶ giflib
      ▶ utils
        BitmapDecoderJNI.cpp
        CMakeLists.txt
        Color.h
        FrameSequence.cpp
        FrameSequence.h
        FrameSequence_gif.cpp
        FrameSequence_gif.h
        FrameSequenceJNI.cpp
        FrameSequenceJNI.h
        JNIHelpers.cpp
        JNIHelpers.h
        Registry.cpp
        Registry.h
        Stream.cpp
        Stream.h
    ▶ java
      AndroidManifest.xml
```

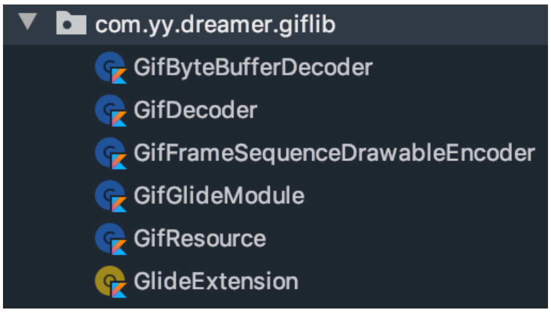
- 编译脚本cmakelist.txt

> 由于项目原工程使用ndk方式，改造为cmake

```
cmake_minimum_required(VERSION 3.4.1)

set(CMAKE_CXX_VISIBILITY_PRESET hidden)
set(CMAKE_C_VISIBILITY_PRESET hidden)
set(CMAKE_VISIBILITY_INLINES_hidden 1)
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -ffunction-sections -fdata-sections -fomit-frame-pointer")
set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -ffunction-sections -fdata-sections -fomit-frame-pointer")
set(CMAKE_EXE_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} -Wl,--gc-sections")

file(GLOB_RECURSE GIF_LIB ${CMAKE_SOURCE_DIR}/giflib/*.*)
file(GLOB_RECURSE FRAME_SEQUENCE ${CMAKE_SOURCE_DIR}/*.cpp*)

add_library(ngif
        SHARED
        ${FRAME_SEQUENCE}
        ${GIF_LIB})
```

```
set(LIBS)
list(APPEND LIBS
        jnigraphics
        android
        GLESv2
        log
        )

target_link_libraries(ngif ${LIBS})
```

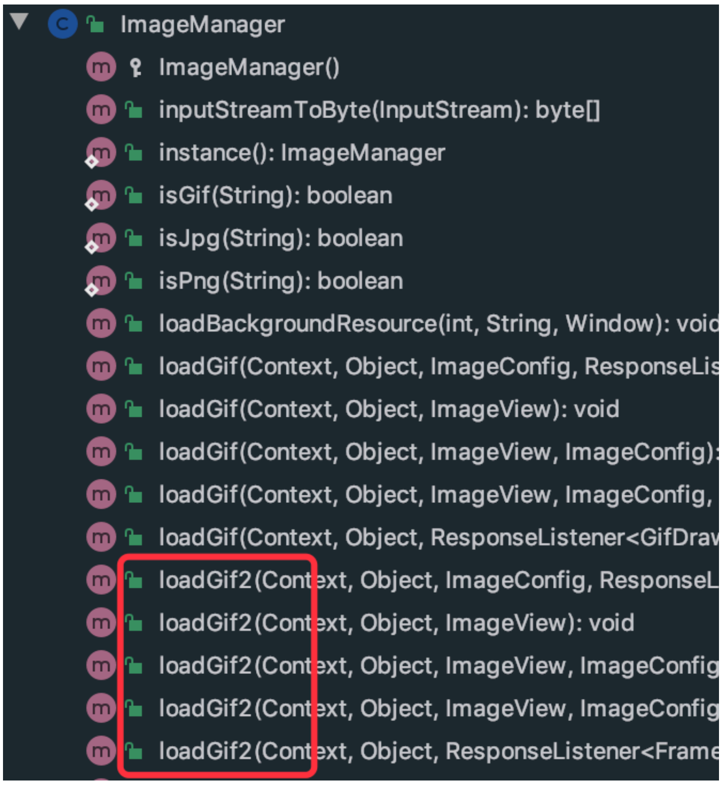
- 集成glide



com.yy.dreamer.giflib
GifByteBufferDecoder
GifDecoder
GifFrameSequenceDrawableEncoder
GifGlideModule
GifResource
GlideExtension

```
fun registerComponents(glide: Glide) {
    glide.registry.append(Registry.BUCKET_GIF,
        InputStream::class.java,
        FrameSequenceDrawable::class.java, GifDecoder(glide.bitmapPool))
    glide.registry.append(Registry.BUCKET_GIF, ByteBuffer::class.java,
FrameSequenceDrawable::class.java,
        GifByteBufferDecoder(glide.bitmapPool))
    glide.registry.append(FrameSequenceDrawable::class.java,
GifFrameSequenceDrawableEncoder())
}
```

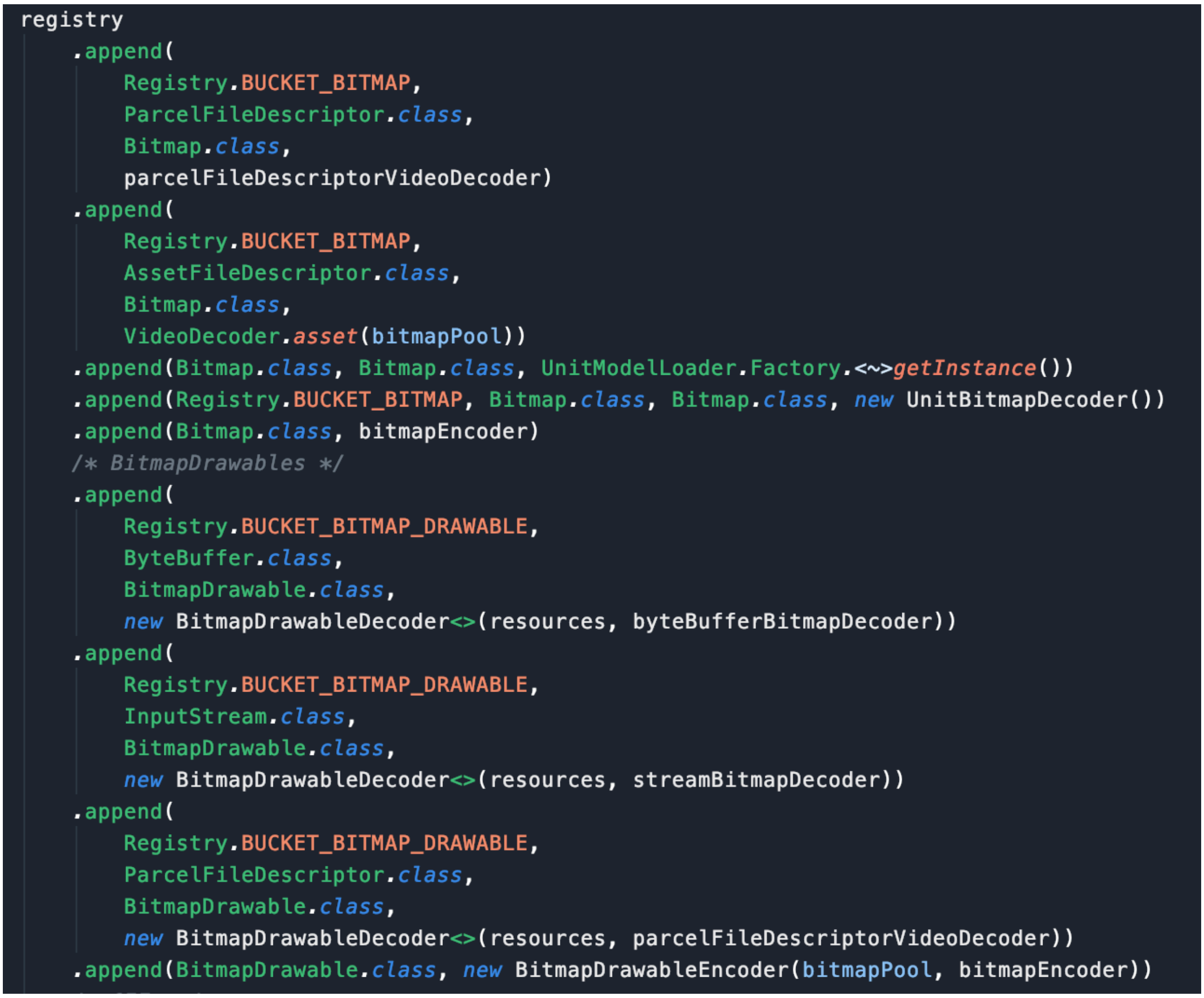
- 如何调用



```
ImageManager
    ImageManager()
    inputStreamToByte(InputStream): byte[]
    instance(): ImageManager
    isGif(String): boolean
    isJpg(String): boolean
    isPng(String): boolean
    loadBackgroundResource(int, String, Window): void
    loadGif(Context, Object, ImageConfig, ResponseLis
    loadGif(Context, Object, ImageView): void
    loadGif(Context, Object, ImageView, ImageConfig):
    loadGif(Context, Object, ImageView, ImageConfig,
    loadGif(Context, Object, ResponseListener<GifDraw
    loadGif2(Context, Object, ImageConfig, ResponseL
    loadGif2(Context, Object, ImageView): void
    loadGif2(Context, Object, ImageView, ImageConfig
    loadGif2(Context, Object, ImageView, ImageConfig
    loadGif2(Context, Object, ResponseListener<Frame
```

glide核心源码解析

> 使用Glide来加载网络图片非常简单，通过`Glide.with(this).load(url).into(imageView)`这样的一句代码就可以搞定，虽然很简单，但还是需要知其所以然。下面就来梳理一下Glide是如何加载网络图片。



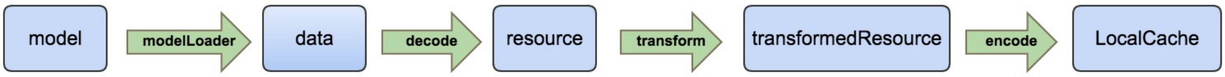
```
registry
    .append(
        Registry.BUCKET_BITMAP,
        ParcelFileDescriptor.class,
        Bitmap.class,
        parcelFileDescriptorVideoDecoder)
    .append(
        Registry.BUCKET_BITMAP,
        AssetFileDescriptor.class,
        Bitmap.class,
        VideoDecoder.asset(bitmapPool))
    .append(Bitmap.class, Bitmap.class, UnitModelLoader.Factory.<~>getInstance())
    .append(Registry.BUCKET_BITMAP, Bitmap.class, Bitmap.class, new UnitBitmapDecoder())
    .append(Bitmap.class, bitmapEncoder)
    /* BitmapDrawables */
    .append(
        Registry.BUCKET_BITMAP_DRAWABLE,
        ByteBuffer.class,
        BitmapDrawable.class,
        new BitmapDrawableDecoder<>(resources, byteBufferBitmapDecoder))
    .append(
        Registry.BUCKET_BITMAP_DRAWABLE,
        InputStream.class,
        BitmapDrawable.class,
        new BitmapDrawableDecoder<>(resources, streamBitmapDecoder))
    .append(
        Registry.BUCKET_BITMAP_DRAWABLE,
        ParcelFileDescriptor.class,
        BitmapDrawable.class,
        new BitmapDrawableDecoder<>(resources, parcelFileDescriptorVideoDecoder))
    .append(BitmapDrawable.class, new BitmapDrawableEncoder(bitmapPool, bitmapEncoder))
```

构造方法最重要的就是`Register`这个类

> 管理组件（数据类型＋数据处理）的注册

它主要是用于管理组件注册以扩展或替换Glide的默认加载，解码和编码逻辑，比如我们可以使用giflib替换Glide自带的GIF解码器，来提高性能，可以使用OKHttp来替换Glide默认的下载实现，也可以自己定义比Glide默认性能更好的编解码器等。构造方法里默认注册了`HttpGlideUrlLoader`这个类，默认的下载实现就在这个类。

标准的数据处理流程：



那么图片的下载是从哪里开始的尼？就是通过`into`方法来实现的，来看一下`into`方法的实现。

```
public ViewTarget<ImageView, TranscodeType> into(@NonNull ImageView view) {
  ...

  return into(
      glideContext.buildImageViewTarget(view, transcodeClass),
      /*targetListener=*/ null,
      requestOptions,
      Executors.mainThreadExecutor());
}
```

> 由于`transcodeClass`是一个`Drawable`类型，所以`glideContext.buildImageViewTarget(view, transcodeClass)`创建了一个`DrawableImageViewTarget`对象，来看看`DrawableImageViewTarget`的实现。

再来看`into`方法的实现。

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

  Request request = buildRequest(target, targetListener, options,
callbackExecutor);
 ...
  requestManager.clear(target);
  target.setRequest(request);
  requestManager.track(target, request);

  return target;
}
```

> 默认创建的`Request`对象是`SingleRequest`，由于本文分析的是第一次加载图片，所以我们来看`RequestManager`的`track`方法。

由于这里`Request`的具体实现是`SingleRequest`，所以我们来看它的`begin`方法。

```
public void begin() {
  synchronized (requestLock) {
    ...
    if (status == Status.COMPLETE) {
      onResourceReady(
     resource, DataSource.MEMORY_CACHE, /* isLoadedFromAlternateCacheKey= */ false);
   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());
  }
 }
}
```

如果图片的宽高已经确定就直接调用`onSizeReady`，否先确定宽高再调用`onSizeReady`方法，该方法中最关键的是调用`Engine`的`load`方法，来看一下实现。

```
public <R> LoadStatus load(
    GlideContext glideContext,
    Object model,
    Key signature,
    int width,
    int height,
    Class<?> resourceClass,
    Class<R> transcodeClass,
    Priority priority,
    DiskCacheStrategy diskCacheStrategy,
    Map<Class<?>, Transformation<?>> transformations,
    boolean isTransformationRequired,
    boolean isScaleOnlyOrNoTransform,
    Options options,
    boolean isMemoryCacheable,
    boolean useUnlimitedSourceExecutorPool,
    boolean useAnimationPool,
    boolean onlyRetrieveFromCache,
    ResourceCallback cb,
    Executor callbackExecutor) {
  long startTime = VERBOSE_IS_LOGGABLE ? LogTime.getLogTime() : 0;
  EngineKey key =
      keyFactory.buildKey(
          model,
          signature,
          width,
          height,
          transformations,
          resourceClass,
          transcodeClass,
          options);

  EngineResource<?> memoryResource;
  synchronized (this) {
    memoryResource = loadFromMemory(key, isMemoryCacheable, startTime);

    if (memoryResource == null) {
      return waitForExistingOrStartNewJob(
          glideContext,
          model,
          signature,
          width,
          height,
          resourceClass,
          transcodeClass,
          priority,
          diskCacheStrategy,
          transformations,
          isTransformationRequired,
          isScaleOnlyOrNoTransform,
          options,
          isMemoryCacheable,
          useUnlimitedSourceExecutorPool,
          useAnimationPool,
          onlyRetrieveFromCache,
          cb,
          callbackExecutor,
          key,
          startTime);
    }
  }

  // Avoid calling back while holding the engine lock, doing so makes it easier
for callers to
  // deadlock.
  cb.onResourceReady(
      memoryResource, DataSource.MEMORY_CACHE, /*
isLoadedFromAlternateCacheKey= */ false);
  return null;
}
```

Glide会首先从缓存中获取数据，如果没有的话再从网络获取。`EngineJob`与`DecodeJob`两个类非常重要，`EngineJob`主要进行线程之间的切换，`DecodeJob`主要是从本地或者网络获取数据的实现，来看`EngineJob`的`start`的实现。

```
public synchronized void start(DecodeJob<R> decodeJob) {
  this.decodeJob = decodeJob;
  GlideExecutor executor =
      decodeJob.willDecodeFromCache() ? diskCacheExecutor :
getActiveSourceExecutor();
  executor.execute(decodeJob);
}
```

由于`DecodeJob`实现了`Runnable`接口，所以就来看`run`的方法

```
public void run() {
  DataFetcher<?> localFetcher = currentFetcher;
  try {
    if (isCancelled) {
      notifyFailed();
      return;
    }
    runWrapped();
  } catch (CallbackException e) {
...
}
 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);
    }
  }
```

很明显这里的重点是`runGenerators`，来看看`runGenerators`的实现。

```
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();
  }
}
```

由于这里不涉及到缓存，所以调用`SourceGenerator`的`startNext`的方法，当网络返回数据时则`dataToCache`不为null，就会存储数据到本地。否则就从网络获取数据。

```
@Override
public boolean startNext() {
  if (dataToCache != null) { //当下载成功后，dataToCache 则不为null，需要写入缓存，后
面会用到
    Object data = dataToCache;
    dataToCache = null;
    cacheData(data);
  }
  //从缓存拿数据
  if (sourceCacheGenerator != null && sourceCacheGenerator.startNext()) {
    return true;
  }
  sourceCacheGenerator = null;
  //从网络获取数据
  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;//加载数据，loadData的实现是MultiModelLoader,
loadData.fetcher的实现是MultiFetcher
```

```
      startNextLoad(loadData);
    }
  }
  return started;
}
```

这里的`loadData`的实现是`MultiModelLoader`，`fetcher`的实现是`MultiFetcher`，然后调用`loadData`方法来加载数据。

由于我们没有任何定制fetcher，所以调用的是`HttpUrlFetcher`的`load`方法

```
 @Override
  public void loadData(
      @NonNull Priority priority, @NonNull DataCallback<? super InputStream>
callback) {
    long startTime = LogTime.getLogTime();
    try {
      InputStream result = loadDataWithRedirects(glideUrl.toURL(), 0, null,
glideUrl.getHeaders());
      callback.onDataReady(result);
    } catch (IOException e) {
      if (Log.isLoggable(TAG, Log.DEBUG)) {
        Log.d(TAG, "Failed to load data for url", e);
      }
      callback.onLoadFailed(e);
    } finally {
      if (Log.isLoggable(TAG, Log.VERBOSE)) {
        Log.v(TAG, "Finished http url fetcher fetch in " +
LogTime.getElapsedMillis(startTime));
      }
    }
  }
```

将数据通过`callback.onDataReady(result);`返回，这个callback其实就是`MultiFetcher`。

```
@Override
public void onDataReady(@Nullable Data data) {
  if (data != null) {
    callback.onDataReady(data);
  } else {
    startNextOrFail();
  }
}
```

这个`callback`其实就是`SourceGenerator`。

这个cb其实就是`DecodeJob`。

```
private void startNextLoad(final LoadData<?> toStart) {
  loadData.fetcher.loadData(
      helper.getPriority(),
      new DataCallback<Object>() {
        @Override
        public void onDataReady(@Nullable Object data) {
          if (isCurrentRequest(toStart)) {
            onDataReadyInternal(toStart, data);
          }
        }

        @Override
        public void onLoadFailed(@NonNull Exception e) {
          if (isCurrentRequest(toStart)) {
            onLoadFailedInternal(toStart, e);
          }
        }
      });
}
```

接下来进入onDataReadyInternal里面

```
void onDataReadyInternal(LoadData<?> loadData, Object data) {
  DiskCacheStrategy diskCacheStrategy = helper.getDiskCacheStrategy();
  if (data != null && diskCacheStrategy.isDataCacheable(loadData.fetcher.getDataSource())) {
    dataToCache = data;
    // We might be being called back on someone else's thread. Before doing anything, we should
    // reschedule to get back onto Glide's thread.
    cb.reschedule();
  } else {
    cb.onDataFetcherReady(
        loadData.sourceKey,
        data,
        loadData.fetcher,
        loadData.fetcher.getDataSource(),
        originalKey);
  }
}
```

这个cb其实就是`DecodeJob`。

```
@Override
public void reschedule(DecodeJob<?> job) {
  // Even if the job is cancelled here, it still needs to be scheduled so that
it can clean itself
  // up.
  getActiveSourceExecutor().execute(job);
}
```

> 这里是切换到缓存数据线程，那么就会执行`DecodeJob`的`run`方法，前面介绍过，在该方法内执行的是`runWrapped`方法，由于前面将`runReason`的值修改为`SWITCH_TO_SOURCE_SERVICE`，所以就会直接执行`runGenerators`然后再次调用`SourceGenerator`的`startNext`方法，前面在介绍该方法时，说过如果有数据就写入缓存，这时候就会将数据写入缓存并调用`DataCacheGenerator`的`startNext`方法。

```
public boolean startNext() {
  while (modelLoaders == null || !hasNextModelLoader()) {
    ...

  loadData = null;
  boolean started = false;
  while (!started && hasNextModelLoader()) {
    ModelLoader<File, ?> modelLoader = modelLoaders.get(modelLoaderIndex++);
    loadData = //加载从缓存中获取数据
      //loadData的实现类是ByteBufferFileLoader
      //loadData.fetcher的实现类是ByteBufferFetcher
        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;
}
```

然后又继续刚刚的流程，回到cb(DecodeJob)..onDataFetcherReady

```
private void decodeFromRetrievedData() {
  if (Log.isLoggable(TAG, Log.VERBOSE)) {
    logWithTimeAndKey(
        "Retrieved data",
        startFetchTime,
        "data: "
            + currentData
            + ", cache key: "
            + currentSourceKey
            + ", fetcher: "
            + currentFetcher);
  }
  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,
isLoadingFromAlternateCacheKey);
  } else {
    runGenerators();
  }
}
```

进入notifyEncodeAndRelease中的notifyComplete然后回调

```
private void notifyComplete(
    Resource<R> resource, DataSource dataSource, boolean
isLoadedFromAlternateCacheKey) {
  setNotifiedOrThrow();
  callback.onResourceReady(resource, dataSource,
isLoadedFromAlternateCacheKey);
}
```

这里的callback是EngineJob,然后调用notifyCallbacksOfResult

```
void notifyCallbacksOfResult() {
  ResourceCallbacksAndExecutors copy;
  Key localKey;
  EngineResource<?> localResource;
  synchronized (this) {
    ...

  engineJobListener.onEngineJobComplete(this, localKey, localResource);

  for (final ResourceCallbackAndExecutor entry : copy) {
    entry.executor.execute(new CallResourceReady(entry.cb));
    //通过execute执行一个 CallResourceReady的runnable对象
  }
  decrementPendingCallbacks();
}
```

```
public void run() {
  // Make sure we always acquire the request lock, then the EngineJob lock to
avoid deadlock
  // (b/136032534).
  synchronized (cb.getLock()) {
    synchronized (EngineJob.this) {
      if (cbs.contains(cb)) {
        // Acquire for this particular callback.
        engineResource.acquire();
        callCallbackOnResourceReady(cb);
        removeCallback(cb);
      }
      decrementPendingCallbacks();
    }
  }
}
```

进入callCallbackOnResourceReady，这里cb是singleRequest

```
public void onResourceReady(
    Resource<?> resource, DataSource dataSource, boolean
isLoadedFromAlternateCacheKey) {
  stateVerifier.throwIfRecycled();
  Resource<?> toRelease = null;
  try {
    synchronized (requestLock) {
      ...

      onResourceReady(
          (Resource<R>) resource, (R) received, dataSource,
isLoadedFromAlternateCacheKey);
    }
  } finally {
    if (toRelease != null) {
      engine.release(toRelease);
    }
  }
```

回调onResourceReady

```
private void onResourceReady(
    Resource<R> resource, R result, DataSource dataSource, boolean
isAlternateCacheKey) {
  // We must call isFirstReadyResource before setting status.
  boolean isFirstResource = isFirstReadyResource();
  status = Status.COMPLETE;
  this.resource = resource;
...
```

```
  isCallingCallbacks = true;
  try {
    boolean anyListenerHandledUpdatingTarget = false;
    if (requestListeners != null) {
      for (RequestListener<R> listener : requestListeners) {
        anyListenerHandledUpdatingTarget |=
            listener.onResourceReady(result, model, target, dataSource,
isFirstResource);
      }
    }
    anyListenerHandledUpdatingTarget |=
        targetListener != null
            && targetListener.onResourceReady(result, model, target,
dataSource, isFirstResource);

    if (!anyListenerHandledUpdatingTarget) {
      Transition<? super R> animation = animationFactory.build(dataSource,
isFirstResource);
      //在前面说过，target的实现类是DrawableImageViewTarget。但在该类中并没有
onResourceReady这个方法，于是去父类查找
      target.onResourceReady(result, animation);
    }
  } finally {
    isCallingCallbacks = false;
  }

  notifyLoadSuccess();
}
```

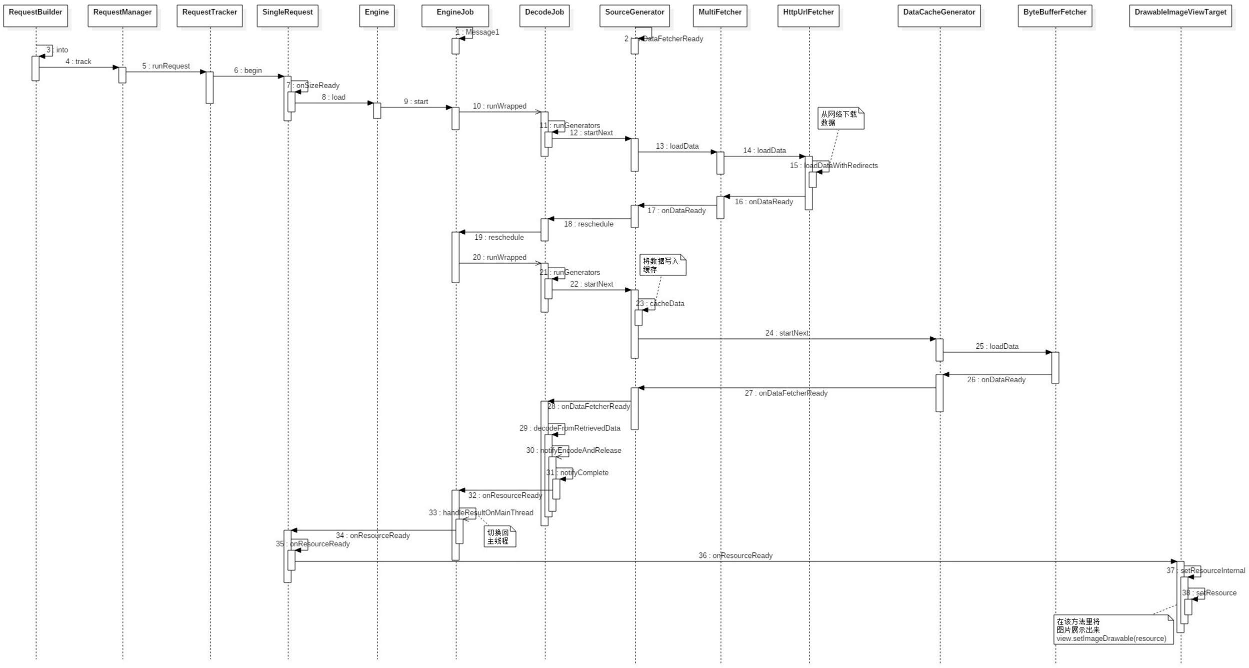
ImageViewTarget中的onResourceReady

```
@Override
public void onResourceReady(@NonNull Z resource, @Nullable Transition<? super
Z> transition) {
  if (transition == null || !transition.transition(resource, this)) {
    setResourceInternal(resource);
  } else {
    maybeUpdateAnimatable(resource);
  }
}
```

最终又回到DrawableImageViewTarget的

```
@Override
protected void setResource(@Nullable Drawable resource) {
  view.setImageDrawable(resource);
}
```

> 到此Glide加载网络图片的流程就完结了，太复杂了，特别是`into`方法，由于很复杂，所以画了张时序图，如下:


RequestBuilder
RequestManager
RequestTracker
SingleRequest
Engine
EngineJob
DecodeJob
SourceGenerator
MultiFetcher
HttpUrlFetcher
DataCacheGenerator
ByteBufferFetcher
DrawableImageViewTarget
1 : Message1
2 : DataFetcherReady
3 : into
4 : track
5 : runRequest
6 : begin
7 : onSizeReady
8 : load
9 : start
10 : runWrapped
11 : runGenerators
12 : startNext
13 : loadData
14 : loadData
从网络下载数据
15 : loadDataWithRedirects
16 : onDataReady
17 : onDataReady
18 : reschedule
19 : reschedule
20 : runWrapped
21 : runGenerators
将数据写入缓存
22 : startNext
23 : cacheData
24 : startNext
25 : loadData
26 : onDataReady
27 : onDataFetcherReady
28 : onDataFetcherReady
29 : decodeFromRetrievedData
30 : notifyEncodeAndRelease
31 : notifyComplete
32 : onResourceReady
33 : handleResultOnMainThread
切换回主线程
34 : onResourceReady
35 : onResourceReady
36 : onResourceReady
37 : setResourceInternal
38 : setResource
在该方法里将图片展示出来
view.setImageDrawable(resource)