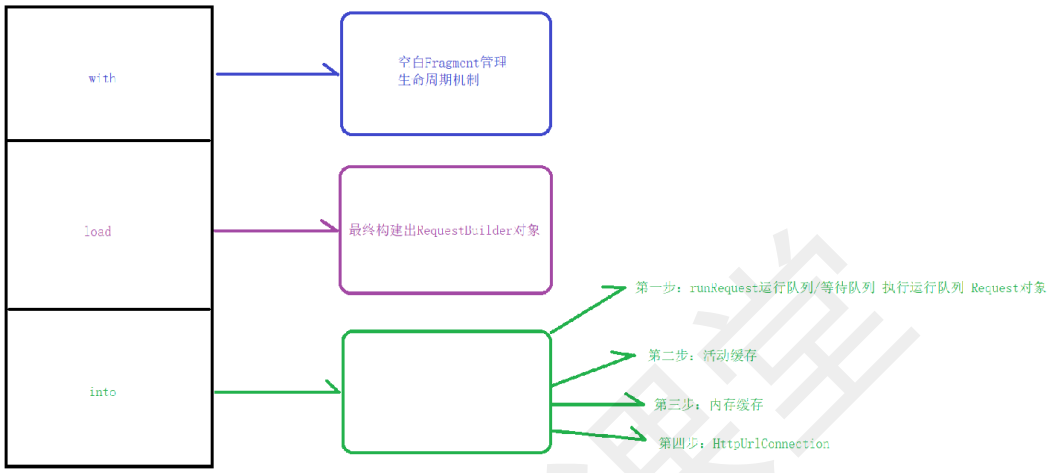
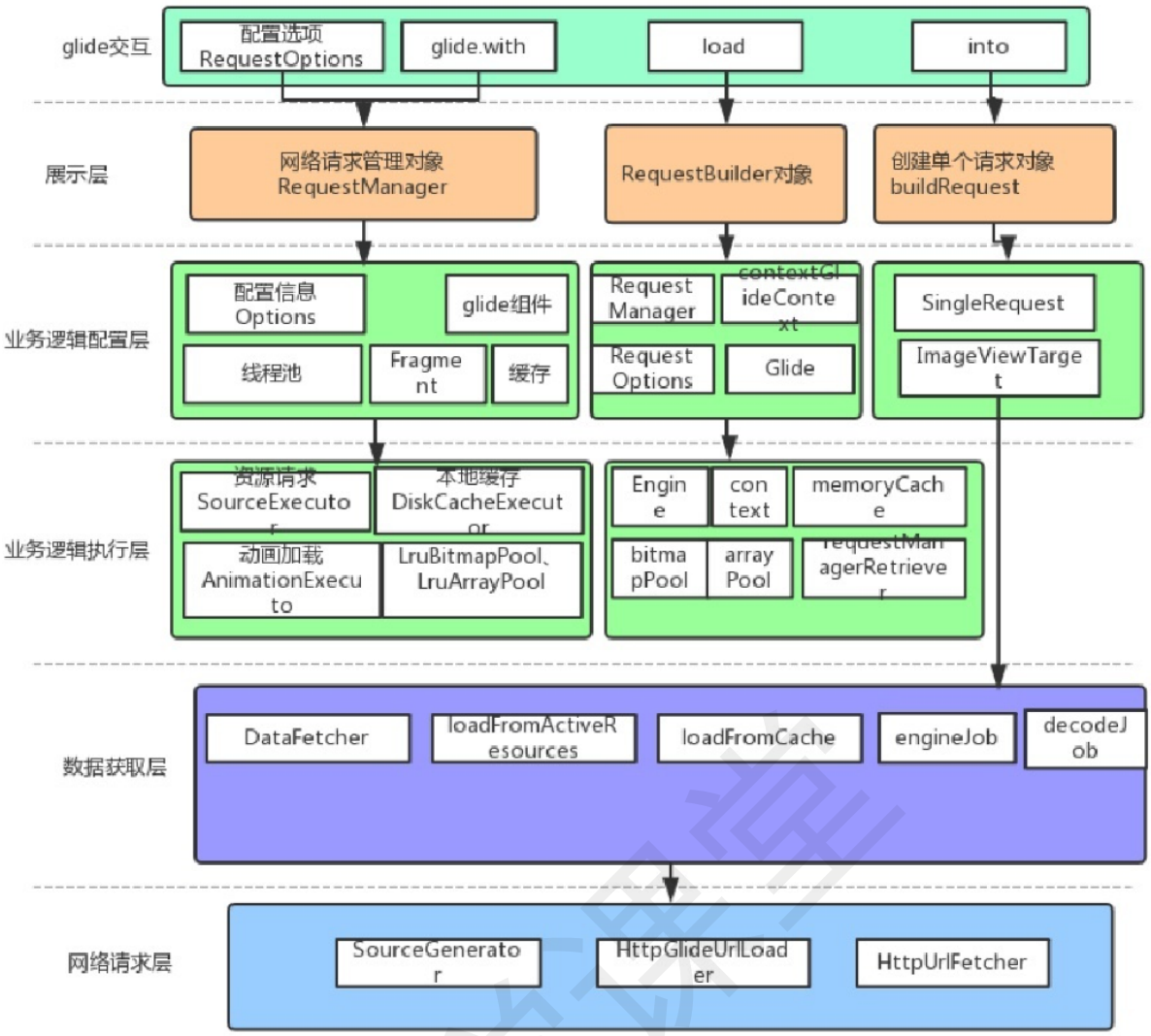
# Glide第二节课(最新Glide4.11源码)

**同学们注意：时序图我已经发给大家了，时序图要看哦，有高清的图片：**

**我们需要关心的 关键点：**

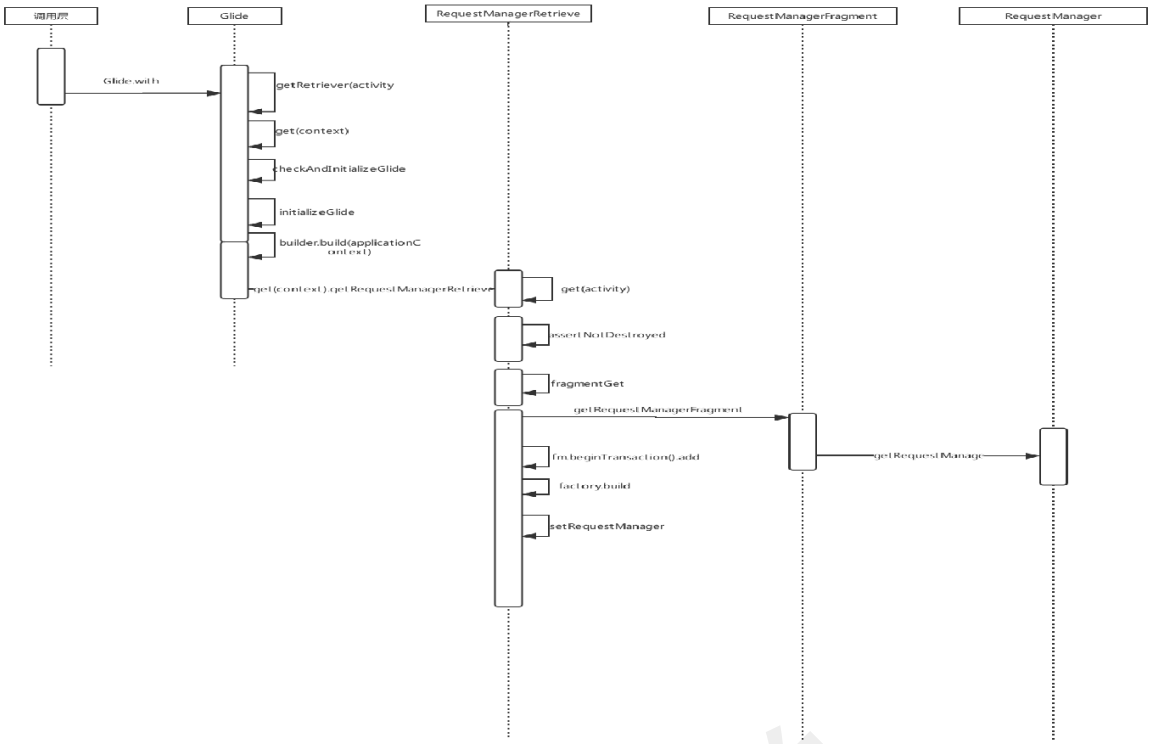



**Glide架构**：（同学们这张图片是网上的图片而已，很模糊，有个大概的认识就行）




【环节一，**with** 源码：】


with时序图：




Glide

- 主要做一些 init 工作，比如缓存，线程池，复用池的构建等等。

RequestManagerRetriever

- 主要是获得一个 `RequestManager` 请求管理类，然后绑定一个 Fragment 。

SupportRequestManagerFragment :

- 用于管理请求的生命周期。

RequestManager

- 主要用于对请求的管理封装。

第一步：调用 with，同学们可以看下 with 这个源码函数，重载有很多：

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

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

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

    @NonNull
    public static RequestManager with(@NonNull Fragment fragment)
{
        return
getRetriever(fragment.getActivity()).get(fragment);
    }

    @Deprecated
    @NonNull
    public static RequestManager with(@NonNull
android.app.Fragment fragment) {
        return
getRetriever(fragment.getActivity()).get(fragment);
    }

    @NonNull
    public static RequestManager with(@NonNull View view) {
        return getRetriever(view.getContext()).get(view);
    }
```

同学们上面其实常用的就 Activity，Fragment, Context 这 3 种形式，下面我们就以 Activity 为主

第二步，getRetriever(activity):

```
    @NonNull
    private static RequestManagerRetriever getRetriever(@Nullable
Context context) {
        Preconditions.checkNotNull(context, "You cannot start a
load on a not yet attached View or xxx.");
        return get(context).getRequestManagerRetriever();
    }
```

同学们继续看 get(context):

```
    @NonNull
    public static Glide get(@NonNull Context context) {
        if (glide == null) {
            Class var1 = Glide.class;
            synchronized(Glide.class) {
                if (glide == null) {
                    checkAndInitializeGlide(context);
                }
            }
        }

        return glide;
    }
```

同学们上面的 Glide get(Context) 是一种双重检测单例模式(DCL)，保证了多线程下安全，仅此而已，非常的简单：

checkAndInitializeGlide(context); 看看做了什么：

```
private static void checkAndInitializeGlide(@NonNull Context
context) {
    if (isInitializing) {
        //【同学们这里不要太在意支线了，听我的只管主线】这里会抛出初始化异常的
信息
    } else {
        //是否初始化标志
        isInitializing = true;
        //开始进行初始化
        initializeGlide(context);
        isInitializing = false;
    }
}
```

同学们接着看 initializeGlide(context);：

```
    private static void initializeGlide(@NonNull Context context)
{
        //实例化一个 GlideBuilder 在进行初始化
        //GlideBuilder 默认的一些配置信息
        initializeGlide(context, new GlideBuilder());
    }
```

辛苦同学们再接着看**initializeGlide**：

```
    private static void initializeGlide(@NonNull Context context,
@NonNull GlideBuilder builder) {
        //1. 拿到应用级别的上下文，这里可以避免内存泄漏，我们实际开发也可以通
过这种形式拿上下文。
        Context applicationContext =
context.getApplicationContext();
        //2. 这里拿到 @GlideModule 标识的注解处理器生成的
GeneratedAppGlideModuleImpl、
//GeneratedAppGlideModuleFactory ...等等。
        GeneratedAppGlideModule annotationGeneratedModule =
getAnnotationGeneratedGlideModules();

      .....

        //3. 通过注解生成的代码拿到 RequestManagerFactory
        RequestManagerFactory factory = annotationGeneratedModule
!= null ?
annotationGeneratedModule.getRequestManagerFactory() : null;
        //4. 将拿到的工厂添加到 GlideBuilder
        builder.setRequestManagerFactory(factory);

      ....
        //5. 这里通过 Builder 建造者模式，构建出 Glide 实例对象
        Glide glide = builder.build(applicationContext);
        Iterator var13 = manifestModules.iterator();

        //6. 开始注册组件回调
        while(var13.hasNext()) {
            GlideModule module = (GlideModule)var13.next();
            module.registerComponents(applicationContext, glide,
glide.registry);
        }

        if (annotationGeneratedModule != null) {

annotationGeneratedModule.registerComponents(applicationContext,
glide, glide.registry);
        }

        applicationContext.registerComponentCallbacks(glide);
        //将构建出来的 glide 赋值给 Glide 的静态变量
        Glide.glide = glide;
    }
```

通过上面的注释，相信同学们很容易理解，注意看注释 5 ，这里知道是通过建造者生成的，那么具体内部怎么实现的，接着看：

```
package com.bumptech.glide;

public final class GlideBuilder {

  //管理线程池
  private Engine engine;
  //对象池(享元模式)，这样做避免重复创建对象，对内存开销有一定效果
  private BitmapPool bitmapPool;
  private ArrayPool arrayPool;

  //GlideExecutor 线程池
  private GlideExecutor sourceExecutor;
  private GlideExecutor diskCacheExecutor;
  //本地磁盘缓存
  private DiskCache.Factory diskCacheFactory;
  //内存缓存
  private MemorySizeCalculator memorySizeCalculator;
  private MemoryCache memoryCache;
  private ConnectivityMonitorFactory connectivityMonitorFactory;
  private int logLevel = Log.INFO;
  private RequestOptions defaultRequestOptions = new
RequestOptions();
  @Nullable
  private RequestManagerFactory requestManagerFactory;
  private GlideExecutor animationExecutor;
  private boolean isActiveResourceRetentionAllowed;
  @Nullable
  private List<RequestListener<Object>> defaultRequestListeners;
  private boolean isLoggingRequestOriginsEnabled;

  //都是一些配置信息，用到了 开闭原则。
    ....

  //开始构建，同学们注意：重点留意我写的注释
  @NonNull
  Glide build(@NonNull Context context) {
    //实例化一个网络请求的线程池
    if (sourceExecutor == null) {
      sourceExecutor = GlideExecutor.newSourceExecutor();
    }
        //实例化一个本地磁盘缓存的线程池
    if (diskCacheExecutor == null) {
```

```
      diskCacheExecutor = GlideExecutor.newDiskCacheExecutor();
    }
        //实例化一个加载图片动画的一个线程池
    if (animationExecutor == null) {
      animationExecutor = GlideExecutor.newAnimationExecutor();
    }
        //实例化一个对图片加载到内存的一个计算
    if (memorySizeCalculator == null) {
      memorySizeCalculator = new
MemorySizeCalculator.Builder(context).build();
    }
        //实例化一个默认网络连接监控的工厂
    if (connectivityMonitorFactory == null) {
      connectivityMonitorFactory = new
DefaultConnectivityMonitorFactory();
    }

    //实例化一个 Bitmap 对象池
    if (bitmapPool == null) {
      int size = memorySizeCalculator.getBitmapPoolSize();
      //如果池子里还有可用的，直接加入 最近最少使用的 LruBitmap 容器里
      if (size > 0) {
        bitmapPool = new LruBitmapPool(size);
      } else {
        //如果池子已经满了，那么就装在 BitmapPoolAdapter
        bitmapPool = new BitmapPoolAdapter();
      }
    }

    //实例化一个数组对象池
    if (arrayPool == null) {
      arrayPool = new
LruArrayPool(memorySizeCalculator.getArrayPoolSizeInBytes());
    }

    //资源内存缓存
    if (memoryCache == null) {
      memoryCache = new
LruResourceCache(memorySizeCalculator.getMemoryCacheSize());
    }
    //磁盘缓存的工厂
    if (diskCacheFactory == null) {
      diskCacheFactory = new
InternalCacheDiskCacheFactory(context);
    }
```

```
    //构建执行缓存策略跟线程池的引擎
    if (engine == null) {
      engine =
          new Engine(
              memoryCache,
              diskCacheFactory,
              diskCacheExecutor,
              sourceExecutor,
              GlideExecutor.newUnlimitedSourceExecutor(),
              GlideExecutor.newAnimationExecutor(),
              isActiveResourceRetentionAllowed);
    }

    if (defaultRequestListeners == null) {
      defaultRequestListeners = Collections.emptyList();
    } else {
      defaultRequestListeners =
Collections.unmodifiableList(defaultRequestListeners);
    }
        //实例化一个 RequestManagerRetriever 请求管理类
    RequestManagerRetriever requestManagerRetriever =
        new RequestManagerRetriever(requestManagerFactory);

    //实例化 Glide 的地方
    return new Glide(
        context,
        engine,
        memoryCache,
        bitmapPool,
        arrayPool,
        requestManagerRetriever,
        connectivityMonitorFactory,
        logLevel,
        defaultRequestOptions.lock(),
        defaultTransitionOptions,
        defaultRequestListeners,
        isLoggingRequestOriginsEnabled);
  }
}
```

同学们上面的代码中，builder 主要构建线程池、复用池、缓存策略、执行 Engine，最后构建 Glide 实例，我们看看 Glide 怎么实例化的,主要看对应的构造函数就行了。

```
Glide(
      @NonNull Context context,
      @NonNull Engine engine,
      @NonNull MemoryCache memoryCache,
      @NonNull BitmapPool bitmapPool,
      @NonNull ArrayPool arrayPool,
      @NonNull RequestManagerRetriever requestManagerRetriever,
      @NonNull ConnectivityMonitorFactory
connectivityMonitorFactory,
      int logLevel,
      @NonNull RequestOptions defaultRequestOptions,
      @NonNull Map<Class<?>, TransitionOptions<?, ?>>
defaultTransitionOptions,
      @NonNull List<RequestListener<Object>>
defaultRequestListeners,
      boolean isLoggingRequestOriginsEnabled) {
    //将 Builder 构建的线程池，对象池，缓存池保存到 Glide 中
    this.engine = engine;
    this.bitmapPool = bitmapPool;
    this.arrayPool = arrayPool;
    this.memoryCache = memoryCache;
    this.requestManagerRetriever = requestManagerRetriever;
    this.connectivityMonitorFactory = connectivityMonitorFactory;

    //拿到 Glide 对应需要的编解码
    DecodeFormat decodeFormat =
defaultRequestOptions.getOptions().get(Downsampler.DECODE_FORMAT)
;
    bitmapPreFiller = new BitmapPreFiller(memoryCache,
bitmapPool, decodeFormat);

    final Resources resources = context.getResources();

    registry = new Registry();
    registry.register(new DefaultImageHeaderParser());

  //忽略一些配置信息
  ...
        //用于显示对应图片的工厂
    ImageViewTargetFactory imageViewTargetFactory = new
ImageViewTargetFactory();

  //构建一个 Glide 专属的 上下文
    glideContext =
        new GlideContext(
```

```
            context,
            arrayPool,
            registry,
            imageViewTargetFactory,
            defaultRequestOptions,
            defaultTransitionOptions,
            defaultRequestListeners,
            engine,
            isLoggingRequestOriginsEnabled,
            logLevel);
  }
```

同学们会发现，上面有一个 GlideContentx 这个是什么鬼？ 其实就是 Context一个级别的上下文而已，你看呀

```
public class GlideContext extends ContextWrapper{ }
```

同学们到这里我们已经知道了 缓存策略、Glide、GlideContext 怎么构建出来的了，下面我们看怎么拿到 请求管理类 `RequestManager`

getRetriever(activity).get(activity); 最终是返回 RequestManager：

这里的 get 也有很多重载的函数，同学们只需要看 Activity 参数的重载：

```
public class RequestManagerRetriever implements Handler.Callback
{
  @NonNull
  public RequestManager get(@NonNull Context context) {
   if (context == null) {
      throw new IllegalArgumentException("You cannot start a load
on a null Context");
     //如果在主线程中并且不为 Application 级别的 Context 执行
    } 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) {
        //一直到查找 BaseContext
        return get(((ContextWrapper) context).getBaseContext());
      }
    }
        //如果不在主线程中或为 Application 就直接执行
```

```
    return getApplicationManager(context);
  }

  @NonNull
  public RequestManager get(@NonNull FragmentActivity activity) {
        ....
  }

  @NonNull
  public RequestManager get(@NonNull Fragment fragment) {
    ....
  }

  //通过 Activity 拿到 RequestManager
  @SuppressWarnings("deprecation")
  @NonNull
  public RequestManager get(@NonNull Activity activity) {
    //判断当前是否在子线程中请求任务
    if (Util.isOnBackgroundThread()) {
      //通过 Application 级别的 Context 加载
      return get(activity.getApplicationContext());
    } else {
      //检查 Activity 是否已经销毁
      assertNotDestroyed(activity);
      //拿到当前 Activity 的 FragmentManager
      android.app.FragmentManager fm =
activity.getFragmentManager();
      //主要是生成一个 Fragment 然后绑定一个请求管理 RequestManager
      return fragmentGet(
          activity, fm, /*parentHint=*/ null,
isActivityVisible(activity));
    }
  }
}
```

【再需要辛苦同学们，看下面的代码，虽然已经吐血身亡，但是还是需要坚持，看源码 靠的是谁耐得住寂寞，谁就是赢家】

fragmentGet 函数实现：

```
  private RequestManager fragmentGet(@NonNull Context context,
      @NonNull android.app.FragmentManager fm,
      @Nullable android.app.Fragment parentHint,
      boolean isParentVisible) {
    //1. 在当前的 Acitivty 添加一个 Fragment 用于管理请求的生命周期
```

```
    RequestManagerFragment current =
getRequestManagerFragment(fm, parentHint, isParentVisible);
    //拿到当前请求的管理类
    RequestManager requestManager = current.getRequestManager();
    //如果不存在，则创建一个请求管理者保持在当前管理生命周期的 Fragment 中，
相当于 2 者进行绑定，避免内存泄漏。
    if (requestManager == null) {
      Glide glide = Glide.get(context);
      requestManager =
          factory.build(
              glide, current.getGlideLifecycle(),
current.getRequestManagerTreeNode(), context);
      current.setRequestManager(requestManager);
    }
    //返回当前请求的管理者
    return requestManager;
  }
```

同学们，通过上面的代码可知，这里用于 Fragment 管理请求的生命周期，那么我们具体来看看 Fragment 怎么添加到 Activity 中的呢：

```
  private RequestManagerFragment getRequestManagerFragment(
      @NonNull final android.app.FragmentManager fm,
      @Nullable android.app.Fragment parentHint,
      boolean isParentVisible) {
    //通过 TAG 拿到已经实例化过的 Fragment ,相当于如果同一个 Activity
Glide.with..多次，那么就没有必要创建多个。
    RequestManagerFragment current = (RequestManagerFragment)
fm.findFragmentByTag(FRAGMENT_TAG);
    //如果在当前 Activity 中没有拿到管理请求生命周期的 Fragment ，那么就从
缓存中看有没有
    if (current == null) {
      current = pendingRequestManagerFragments.get(fm);
      //如果缓存也没有得，就直接实例化一个 Fragment
      if (current == null) {
        current = new RequestManagerFragment();
        current.setParentFragmentHint(parentHint);
        //如果已经有执行的请求就开始
        if (isParentVisible) {
          current.getGlideLifecycle().onStart();
        }
        //添加到 Map 缓存中
        pendingRequestManagerFragments.put(fm, current);
        //通过当前 Activity 的 FragmentManager 开始提交添加一个
Fragment 容器
```

```
            fm.beginTransaction().add(current,
FRAGMENT_TAG).commitAllowingStateLoss();
            //添加到 FragmentManager 成功，发送清理缓存。
            handler.obtainMessage(ID_REMOVE_FRAGMENT_MANAGER,
fm).sendToTarget();
          }
        }
        return current;
      }
```

同学们注意：然后又回到fragmentGet方法，因为已经拿到了RequestManagerFragment了： 就可以**current.XXXXXX** 得到信息了

```
  private RequestManager fragmentGet(@NonNull Context context,
      @NonNull android.app.FragmentManager fm,
      @Nullable android.app.Fragment parentHint,
      boolean isParentVisible) {
    ...
    //如果不存在，则创建一个请求管理者保持在当前管理生命周期的 Fragment 中，
相当于 2 者进行绑定，避免内存泄漏。
    if (requestManager == null) {
      //拿到单例 Glide
      Glide glide = Glide.get(context);
      //构建请求管理，current.getGlideLifecycle(),就是
ActivityFragmentLifecycle 后面我们会讲到这个类
      requestManager =
          factory.build(
              glide, current.getGlideLifecycle(),
current.getRequestManagerTreeNode(), context);
      //将构建出来的请求管理绑定在 Fragment 中。
      current.setRequestManager(requestManager);
    }
    //返回当前请求的管理者
    return requestManager;
  }
```

同学们我们知道，**with之后**，最终返回**RequestManager对象**， 我们需要对RequestManager对象的构建，有一个来龙去脉的学习：

【同学们注意啊：在Glide4.11 这个最新的版本中，大量使用了 工厂模式：】

```
 // 同学们注意：此工厂就是为了构建出 RequestManager对象
 private static final RequestManagerFactory DEFAULT_FACTORY = new
RequestManagerFactory() {
    @NonNull
    @Override
    public RequestManager build(@NonNull Glide glide, @NonNull
Lifecycle lifecycle,
        @NonNull RequestManagerTreeNode requestManagerTreeNode,
@NonNull Context context) {
      //实例化
      return new RequestManager(glide, lifecycle,
requestManagerTreeNode, context);
    }
  };
```

你只要敢 new RequestManager(....); 就会进入 RequestManager的构造方法：

```
  public RequestManager(
      @NonNull Glide glide, @NonNull Lifecycle lifecycle,
      @NonNull RequestManagerTreeNode treeNode, @NonNull Context
context) {
      this(
        glide,
        lifecycle,
        treeNode,
        new RequestTracker(),
        glide.getConnectivityMonitorFactory(),
        context);
  }

  @SuppressWarnings("PMD.ConstructorCallsOverridableMethod")
  RequestManager(
      Glide glide,
      Lifecycle lifecycle,
      RequestManagerTreeNode treeNode,
      RequestTracker requestTracker,
      ConnectivityMonitorFactory factory,
      Context context) {
    this.glide = glide;
    this.lifecycle = lifecycle;
    this.treeNode = treeNode;
    this.requestTracker = requestTracker;
    this.context = context;

    connectivityMonitor =
```

```
        factory.build(
            context.getApplicationContext(),
            new
RequestManagerConnectivityListener(requestTracker));

    //这里只要是添加生命周期监听，Fragment 传递过来的
    if (Util.isOnBackgroundThread()) {
      mainHandler.post(addSelfToLifecycle);
    } else {
      lifecycle.addListener(this);
    }
    //添加网络变化的监听
    lifecycle.addListener(connectivityMonitor);

    defaultRequestListeners =
        new CopyOnWriteArrayList<>
(glide.getGlideContext().getDefaultRequestListeners());

 setRequestOptions(glide.getGlideContext().getDefaultRequestOptio
ns());

    glide.registerRequestManager(this);
  }
```

同学们，到这里请求管理类 RequestManager + Fragment 已经绑定成功了，声明周期监听也设置了，是不是碉堡了：

那他们相互是怎么保证生命周期的传递勒，我们主要看 Fragment 生命周期方法

```
//这里为什么监控 Fragment 的生命周期勒，其实大家应该也知道 Fragment 是依附在
Activity 的 Activity 的生命周期在 Fragment 中都有，所以监听 Fragment 就
行了。
public class RequestManagerFragment extends Fragment {

  //相当于生命周期回调
  private final ActivityFragmentLifecycle lifecycle;

  ....

  @Override
  public void onStart() {
    super.onStart();
    lifecycle.onStart();
  }
```

```
  @Override
  public void onStop() {
    super.onStop();
    lifecycle.onStop();
  }

  @Override
  public void onDestroy() {
    super.onDestroy();
    lifecycle.onDestroy();
  }
  ...
}
```

这里的 lifecycle 是什么，同学们在深入看看去：

```
class ActivityFragmentLifecycle implements Lifecycle {
  private final Set<LifecycleListener> lifecycleListeners =
      Collections.newSetFromMap(new
WeakHashMap<LifecycleListener, Boolean>());
  private boolean isStarted;
  private boolean isDestroyed;


  @Override
  public void addListener(@NonNull LifecycleListener listener) {
    lifecycleListeners.add(listener);

    if (isDestroyed) {
      listener.onDestroy();
    } else if (isStarted) {
      listener.onStart();
    } else {
      listener.onStop();
    }
  }

  @Override
  public void removeListener(@NonNull LifecycleListener listener)
{
    lifecycleListeners.remove(listener);
  }

  void onStart() {
    isStarted = true;
```

```
    for (LifecycleListener lifecycleListener :
Util.getSnapshot(lifecycleListeners)) {
      lifecycleListener.onStart();
    }
  }

  void onStop() {
    isStarted = false;
    for (LifecycleListener lifecycleListener :
Util.getSnapshot(lifecycleListeners)) {
      lifecycleListener.onStop();
    }
  }

  void onDestroy() {
    isDestroyed = true;
    for (LifecycleListener lifecycleListener :
Util.getSnapshot(lifecycleListeners)) {
      lifecycleListener.onDestroy();
    }
  }
}
```

同学们，这里知道了吧，它实现的是 Glide 中的 Lifecycle 生命周期接口，注册是在刚刚我们讲解 RequestManagerFactory 工厂中实例化的 RequestManager 然后在构造函数中添加了生命周期回调监听，具体来看下。

```
public class RequestManager implements LifecycleListener,
    ModelTypes<RequestBuilder<Drawable>> {
  ...
  @Override
  public synchronized void onStart() {
    resumeRequests();
    targetTracker.onStart();
  }

  @Override
  public synchronized void onStop() {
    pauseRequests();
    targetTracker.onStop();
  }

  @Override
  public synchronized void onDestroy() {
    targetTracker.onDestroy();
```

```
    for (Target<?> target : targetTracker.getAll()) {
      clear(target);
    }
    targetTracker.clear();
    requestTracker.clearRequests();
    lifecycle.removeListener(this);
    lifecycle.removeListener(connectivityMonitor);
    mainHandler.removeCallbacks(addSelfToLifecycle);
    glide.unregisterRequestManager(this);
  }
  同学们：省略代码...
}
```

同学们注意：这 3 处回调就是 Fragment 传递过来的，用于实时监听请求的状态。

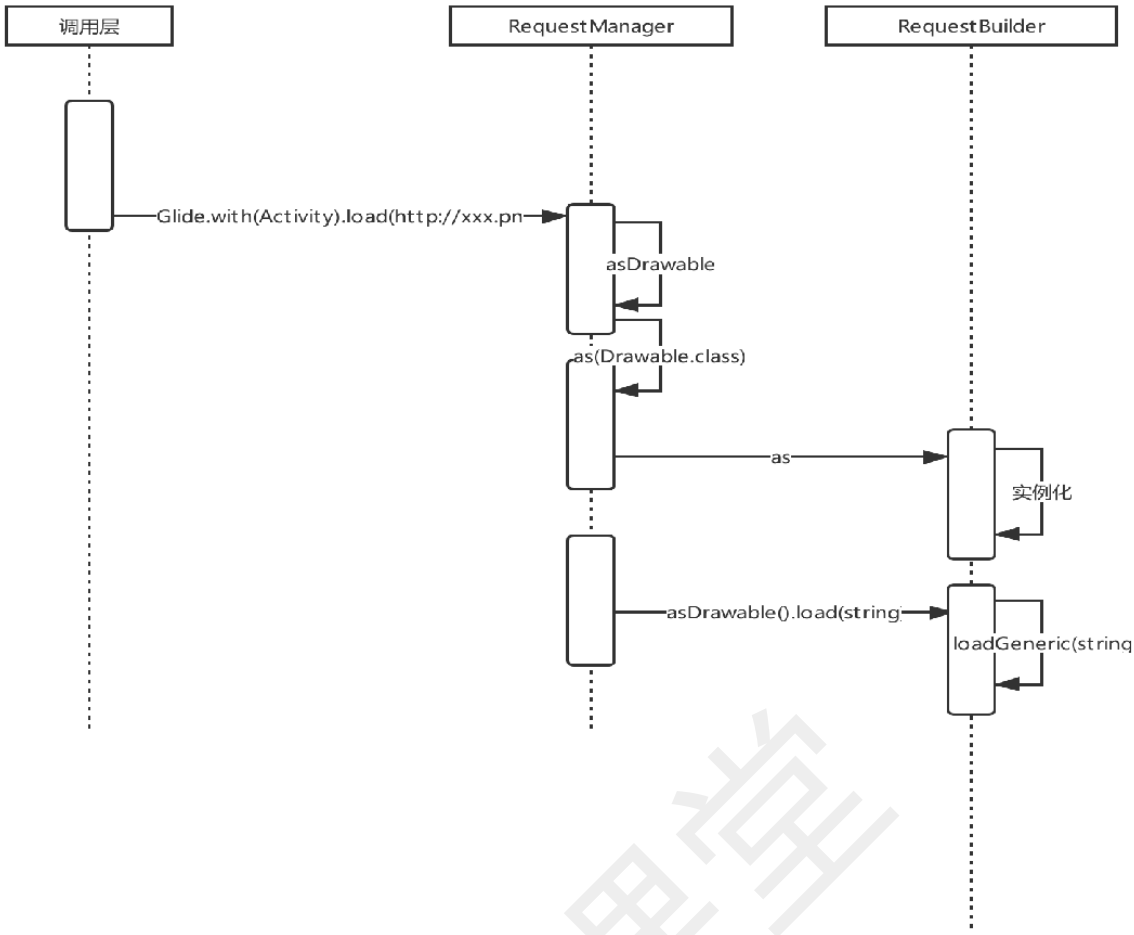
**with总结：**

**根据 with 源码分析，我们知道，Glide.with(Activity) 主要做了 线程池 + 缓存 + 请求管理与生命周期绑定+其它配置初始化的构建，内部的代码其实是很庞大的**

## 同学们要明白：Glide框架的难度，是其他框架不能相提并论的，已经无法同日而语了

【环节二，**load** 源码：】


load时序图：




- RequestBuilder : 这是一个通用请求构建类，可以处理通用资源类型的设置选项和启动负载。

同学们：load 函数加载相对于比较简单。我们看下具体代码实现

```
public class RequestManager implements LifecycleListener,
    ModelTypes<RequestBuilder<Drawable>> {

  .....

  public RequestBuilder<Drawable> load(@Nullable String string) {
    //这里调用 Drawable 图片加载请求器为其加载
    return asDrawable().load(string);
  }

  public RequestBuilder<Drawable> asDrawable() {
    return as(Drawable.class);
  }

  @NonNull
  @CheckResult
  @Override
  public RequestBuilder<Drawable> load(@Nullable Uri uri) {
```

```
    return asDrawable().load(uri);
  }


  @NonNull
  @CheckResult
  @Override
  public RequestBuilder<Drawable> load(@Nullable File file) {
    return asDrawable().load(file);
  }

  public <ResourceType> RequestBuilder<ResourceType> as(
      @NonNull Class<ResourceType> resourceClass) {
    return new RequestBuilder<>(glide, this, resourceClass,
context);
  }
}
```

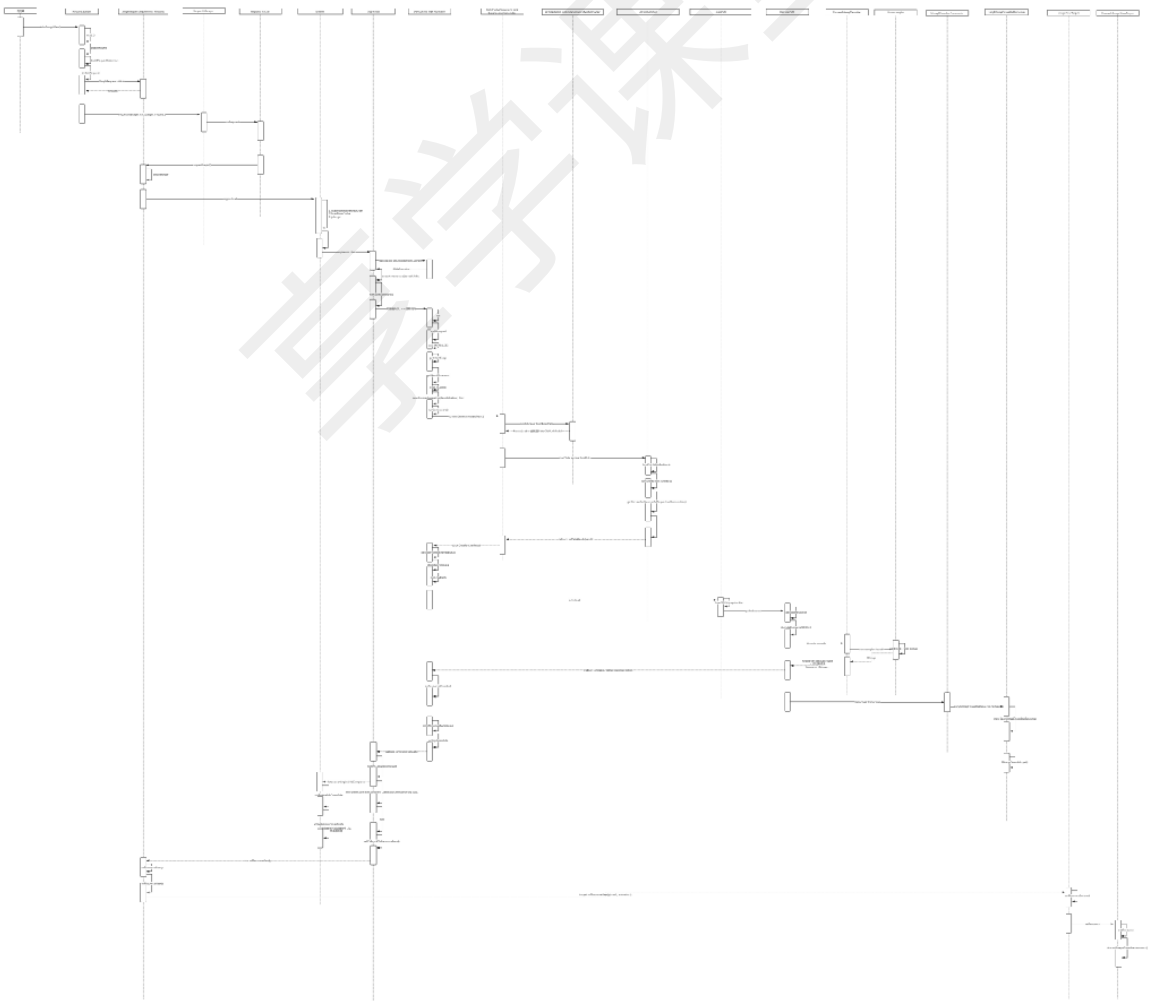
同学们，看看load详情：

```
public class RequestBuilder<TranscodeType> extends
BaseRequestOptions<RequestBuilder<TranscodeType>>
    implements Cloneable,
ModelTypes<RequestBuilder<TranscodeType>> {

  public RequestBuilder<TranscodeType> load(@Nullable String
string) {
    return loadGeneric(string);
  }

  // 描述加载的数据源-这里可以看做是我们刚刚传递进来的 http://xxxx.png
  @Nullable private Object model;
  // 描述这个请求是否已经添加了加载的数据源
  private boolean isModelSet;

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

同学们，到这里 RequestBuilder 就构建好了， RequestBuilder构建出来后，**都是为了后面的into啊**，也意味着，我们目前为止**只是摸到Glide的一点点边而已**，哎，活着zhenlei

【环节三，**into** 源码：】


into时序图：



**【同学们，需要打起十二分精神，因为到目前为止，才开始进入Glide的秘密基地：】**

```
  @NonNull
  public ViewTarget<ImageView, TranscodeType> into(@NonNull
ImageView view) {
    Util.assertMainThread();
    Preconditions.checkNotNull(view);
    // 根据 ImageView 布局中的 scaleType 来重构 requestOptions
    BaseRequestOptions<?> requestOptions = this;
    if (!requestOptions.isTransformationSet()
        && requestOptions.isTransformationAllowed()
        && view.getScaleType() != null) {
      //如果在 xml ImageView 节点中 没有设置 scaleType 那么默认在构造函数
中进行了初始化为   mScaleType = ScaleType.FIT_CENTER;
      switch (view.getScaleType()) {
       .....
        case FIT_CENTER:
        case FIT_START:
        case FIT_END:
          //这里用到了克隆（原型设计模式），选择一个 居中合适 显示的方案，同学
们会发现，到处都是设计模式，它不是为了装B哦
          requestOptions =
requestOptions.clone().optionalFitCenter();
          break;
       ....
      }
    }
    //调用 into 重载函数，创建一个 ViewTarget
    return into(
        //调用 buildImageViewTarget 构建一个 ImageView 类型的
Target(Bitmap/Drawable)
        glideContext.buildImageViewTarget(view, transcodeClass),
        /*targetListener=*/ null,
        requestOptions,
        Executors.mainThreadExecutor());
  }
```

上面代码就两大步：

第一步：先拿到当前 ImageView getScaleType 类型的属性，然后重新 clone 一个进行配置；

第二步：调用 into 重载继续构建；

同学们先来看下 glideContext.buildImageViewTarget 是怎么构建出来 ImageViewTarget 的：

```
  @NonNull
  public <X> ViewTarget<ImageView, X> buildImageViewTarget(
      @NonNull ImageView imageView, @NonNull Class<X>
transcodeClass) {
    //调用 工厂模式 根据 transcodeClass 生成出一个对应的
ImageViewTarget
    return imageViewTargetFactory.buildTarget(imageView,
transcodeClass);
  }
```

```
public class ImageViewTargetFactory {
  @NonNull
  @SuppressWarnings("unchecked")
  public <Z> ViewTarget<ImageView, Z> buildTarget(@NonNull
ImageView view,
      @NonNull Class<Z> clazz) {
    //如果目标的编码类型属于 Bitmap 那么就创建一个 Bitmap 类型的
ImageViewTarget
    if (Bitmap.class.equals(clazz)) {
      return (ViewTarget<ImageView, Z>) new
BitmapImageViewTarget(view);
    ////如果目标的编码类型属于 Drawable 那么就创建一个 Drawable 类型的
ImageViewTarget
    } else if (Drawable.class.isAssignableFrom(clazz)) {
      return (ViewTarget<ImageView, Z>) new
DrawableImageViewTarget(view);
    } else {
      throw new IllegalArgumentException(
          "Unhandled class: " + clazz + ", try .as*
(Class).transcode(ResourceTranscoder)");
    }
  }
}
```

同学们注意：上面 生产 Target 的时候注意一下，只要调用了 `asBitmap` 才会执行生产 BitmapImageViewTarget ,所以这里我们关注 Drawable 类型就行了，我们就先简单看看这个 target 内部怎么实现的，因为最后会讲到这个，先让同学们有个印象：

```
public class DrawableImageViewTarget extends
ImageViewTarget<Drawable> {

  public DrawableImageViewTarget(ImageView view) {
    super(view);
```

```
  }

  @SuppressWarnings({"unused", "deprecation"})
  @Deprecated
  public DrawableImageViewTarget(ImageView view, boolean
waitForLayout) {
    super(view, waitForLayout);
  }

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

同学们从上面代码可以知道 DrawableImageViewTarget 继承的是 ImageViewTarget 重写的 setResource 函数，实现了显示 Drawable 图片的逻辑，好了，这里先有个印象就行，我们只管主线流程，支线细节先跳读，最后会讲到怎么调用的。继续 into 重载

```
  private <Y extends Target<TranscodeType>> Y into(
      @NonNull Y target,
      @Nullable RequestListener<TranscodeType> targetListener,
      BaseRequestOptions<?> options,
      Executor callbackExecutor) {
    Preconditions.checkNotNull(target);
    //这里的 isModelSet 是在 load 的时候赋值为 true 的，所以不会抛异常
    if (!isModelSet) {
      throw new IllegalArgumentException("You must call #load()
before calling #into()");
    }
        //为这个 http://xxx.png 生成一个 Glide request 请求
    Request request = buildRequest(target, targetListener,
options, callbackExecutor);
        //相当于拿到上一个请求
    Request previous = target.getRequest();
    //下面的几行说明是否与上一个请求冲突，一般不用管 直接看下面 else 判断
    if (request.isEquivalentTo(previous)
        && !isSkipMemoryCacheWithCompletePreviousRequest(options,
previous)) {
      request.recycle();
      if (!Preconditions.checkNotNull(previous).isRunning()) {
        previous.begin();
      }
      return target;
```

```
    }
        //清理掉目标请求管理
    requestManager.clear(target);
    //重新为目标设置一个 Glide request 请求
    target.setRequest(request);
    //最后是调用 RequestManager 的 track 来执行目标的 Glide request 请
求
    requestManager.track(target, request);

    return target;
  }
```

以上核心就两个点：

第一点：为 target buildRequest 构建一个 Glide request 请求；

第二点：将构建出来的 Request 交于 RequestManager 来执行；

同学们来简单的来看下怎么构建的 Request：

```
  private Request buildRequest(
      Target<TranscodeType> target,
      @Nullable RequestListener<TranscodeType> targetListener,
      BaseRequestOptions<?> requestOptions,
      Executor callbackExecutor) {
    return buildRequestRecursive(
        target,
        targetListener,
        /*parentCoordinator=*/ null,
        transitionOptions,
        requestOptions.getPriority(),
        requestOptions.getOverrideWidth(),
        requestOptions.getOverrideHeight(),
        requestOptions,
        callbackExecutor);
  }
```

```
  private Request obtainRequest(
      Target<TranscodeType> target,
      RequestListener<TranscodeType> targetListener,
      BaseRequestOptions<?> requestOptions,
      RequestCoordinator requestCoordinator,
      TransitionOptions<?, ? super TranscodeType>
transitionOptions,
```

```
      Priority priority,
      int overrideWidth,
      int overrideHeight,
      Executor callbackExecutor) {
    return SingleRequest.obtain(
        context,
        glideContext,
        model,
        transcodeClass,
        requestOptions,
        overrideWidth,
        overrideHeight,
        priority,
        target,
        targetListener,
        requestListeners,
        requestCoordinator,
        glideContext.getEngine(),
        transitionOptions.getTransitionFactory(),
        callbackExecutor);
  }
```

同学们最后我们发现是 `SingleRequest.obtain` 来为我们构建的 Request 请求对象，开始只是初始化一些配置属性，下面我们就来找 `begin` 开始的地方，先来看下 `track` 函数执行:

```
 //这里对当前 class 加了一个同步锁避免线程引起的安全性
 synchronized void track(@NonNull Target<?> target, @NonNull
Request request) {
    //添加一个目标任务
    targetTracker.track(target);
    //执行 Glide request
    requestTracker.runRequest(request);
  }
```


```
 //这里对当前 class 加了一个同步锁避免线程引起的安全性
 synchronized void track(@NonNull Target<?> target, @NonNull
Request request) {
    //添加一个目标任务
    targetTracker.track(target);
    //执行 Glide request
    requestTracker.runRequest(request);
  }
```

```
  public void runRequest(@NonNull Request request) {
    //添加一个请求
    requests.add(request);
    //是否暂停
    if (!isPaused) {
      //没有暂停，开始调用 Request begin 执行
      request.begin();
    } else {
      //如果调用了 暂停，清理请求
      request.clear();
      pendingRequests.add(request);
    }
  }
```

上面的逻辑是先为 `requests` 添加一个请求，看看是否是停止状态，如果不是就调用 `request.begin();`执行。

这里的 `Request` 是一个接口，通过之前我们讲到 `buildRequest` 函数可知 `Request` 的实现类是 `SingleRequest` 我们就直接看它的 `begin` 函数.

```
  @Override
  public synchronized void begin() {
    assertNotCallingCallbacks();
    stateVerifier.throwIfRecycled();
    startTime = LogTime.getLogTime();
    if (model == null) {
      //检查外部调用的尺寸是否有效
      if (Util.isValidDimensions(overrideWidth, overrideHeight))
{
        width = overrideWidth;
        height = overrideHeight;
      }
      //失败的回调
      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 (status == Status.COMPLETE) {
```

```
      //表示资源准备好了
      onResourceReady(resource, DataSource.MEMORY_CACHE);
      return;
    }


    status = Status.WAITING_FOR_SIZE;
    //这里表示大小已经准备好了
    if (Util.isValidDimensions(overrideWidth, overrideHeight)) {
      //开始
      onSizeReady(overrideWidth, overrideHeight);
    } else {
      target.getSize(this);
    }

    //这里是刚刚开始执行的回调，相当于显示开始的进度
    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));
    }
  }
```

同学们我们直接看 onSizeReady：

```
  public synchronized void onSizeReady(int width, int height) {
    stateVerifier.throwIfRecycled();
    ....//都是一些初始化状态，配置属性，我们不用管。

    loadStatus =
            //加载
        engine.load(
            glideContext,
            model,
            requestOptions.getSignature(),
            this.width,
            this.height,
            requestOptions.getResourceClass(),
            transcodeClass,
            priority,
            requestOptions.getDiskCacheStrategy(),
```

```
            requestOptions.getTransformations(),
            requestOptions.isTransformationRequired(),
            requestOptions.isScaleOnlyOrNoTransform(),
            requestOptions.getOptions(),
            requestOptions.isMemoryCacheable(),
            requestOptions.getUseUnlimitedSourceGeneratorsPool(),
            requestOptions.getUseAnimationPool(),
            requestOptions.getOnlyRetrieveFromCache(),
            this,
            callbackExecutor);
  }
```

load:

```
  public synchronized <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) {

    //拿到缓存或者请求的 key
    EngineKey key = keyFactory.buildKey(model, signature, width,
height, transformations,
        resourceClass, transcodeClass, options);
        //根据 key 拿到活动缓存中的资源
    EngineResource<?> active = loadFromActiveResources(key,
isMemoryCacheable);
    //如果 ActiveResources 活动缓存中有就回调出去
    if (active != null) {
      cb.onResourceReady(active, DataSource.MEMORY_CACHE);
```

```
      return null;
    }

    //尝试从 LruResourceCache 中找寻这个资源
    EngineResource<?> cached = loadFromCache(key,
isMemoryCacheable);
    if (cached != null) {
      //如果内存缓存 Lru 中资源存在回调出去
      cb.onResourceReady(cached, DataSource.MEMORY_CACHE);
      return null;
    }

    //------------- 走到这里说明活动缓存 跟内存 缓存都没有找到 ----------
-

    //根据 Key 看看缓存中是否正在执行
    EngineJob<?> current = jobs.get(key, onlyRetrieveFromCache);
    if (current != null) {
      //如果正在执行，把数据回调出去
      current.addCallback(cb, callbackExecutor);
      if (VERBOSE_IS_LOGGABLE) {
        logWithTimeAndKey("Added to existing load", startTime,
key);
      }
      return new LoadStatus(cb, current);
    }

    // -------------   走到这里说明是一个新的任务  ---------------
    // -------------   构建新的请求任务  ---------------
    EngineJob<R> engineJob =
        engineJobFactory.build(
            key,
            isMemoryCacheable,
            useUnlimitedSourceExecutorPool,
            useAnimationPool,
            onlyRetrieveFromCache);

    DecodeJob<R> decodeJob =
        decodeJobFactory.build(
            glideContext,
            model,
            key,
            signature,
            width,
            height,
```

```
            resourceClass,
            transcodeClass,
            priority,
            diskCacheStrategy,
            transformations,
            isTransformationRequired,
            isScaleOnlyOrNoTransform,
            onlyRetrieveFromCache,
            options,
            engineJob);
        //把当前需要执行的 key 添加进缓存
    jobs.put(key, engineJob);
        //执行任务的回调
    engineJob.addCallback(cb, callbackExecutor);
    //开始执行。
    engineJob.start(decodeJob);

    return new LoadStatus(cb, engineJob);
  }
```

通过 `engine.load` 这个函数里面的逻辑，同学们我们可以总结3点：

1. 先构建请求或者缓存 KEY；
2. 根据 KEY 从内存缓存中查找对应的资源数据(ActiveResources（活动缓存，内部是一个 Map 用弱引用持有）,LruResourceCache)，如果有就回调 对应监听的 onResourceReady 表示数据准备好了。
3. 从执行缓存中查找对应 key 的任务
    1. 如果找到了，就说明已经正在执行了，不用重复执行。
    2. 没有找到，通过 EngineJob.start 开启一个新的请求任务执行。

同学们下面我们就来看下 `engineJob.start` 具体执行逻辑:

```
  public synchronized void start(DecodeJob<R> decodeJob) {
    this.decodeJob = decodeJob;
    //拿到 Glide 执行的线程池
    GlideExecutor executor = decodeJob.willDecodeFromCache()
        ? diskCacheExecutor
        : getActiveSourceExecutor();
    //开始执行
    executor.execute(decodeJob);
  }
```

通过 `DecodeJob` 源码得知，它是实现的 `Runnable` 接口，这里 GlideExecutor 线程池开始执行，就会启动 DecodeJob 的 run 函数，我们跟踪 run 的实现:

```
class DecodeJob<R> implements
DataFetcherGenerator.FetcherReadyCallback,
    Runnable,
    Comparable<DecodeJob<?>>,
    Poolable {

      // 线程执行调用 run
      @Override
      public void run() {

        GlideTrace.beginSectionFormat("DecodeJob#run(model=%s)",
model);

        DataFetcher<?> localFetcher = currentFetcher;
        try {
          //是否取消了当前请求
          if (isCancelled) {
            notifyFailed();
            return;
          }
          //执行
          runWrapped();
        } catch (CallbackException e) {

         .....//一些错误回调
      }
    }
```

分析runWrapped:

```
  private void runWrapped() {
    switch (runReason) {
      case INITIALIZE:
        //获取资源状态
        stage = getNextStage(Stage.INITIALIZE);
        //根据当前资源状态，获取资源执行器
        currentGenerator = getNextGenerator();
        //执行
        runGenerators();
        break;
      ...
    }
```

```
  }

  private Stage getNextStage(Stage current) {
    switch (current) {
      case INITIALIZE:
        //如果外部调用配置了资源缓存策略，那么返回 Stage.RESOURCE_CACHE
        //否则继续调用 Stage.RESOURCE_CACHE 执行。
        return diskCacheStrategy.decodeCachedResource()
            ? Stage.RESOURCE_CACHE :
getNextStage(Stage.RESOURCE_CACHE);
      case RESOURCE_CACHE:
        //如果外部配置了源数据缓存，那么返回 Stage.DATA_CACHE
        //否则继续调用 getNextStage(Stage.DATA_CACHE)
        return diskCacheStrategy.decodeCachedData()
            ? Stage.DATA_CACHE : getNextStage(Stage.DATA_CACHE);
      case DATA_CACHE:
        //如果只能从缓存中获取数据，则直接返回 FINISHED，否则，返回SOURCE。
        //意思就是一个新的资源
        return onlyRetrieveFromCache ? Stage.FINISHED :
Stage.SOURCE;
      case SOURCE:
      case FINISHED:
        return Stage.FINISHED;
      default:
        throw new IllegalArgumentException("Unrecognized stage: "
+ current);
    }
  }
```

通过上面代码可以知道，我们在找资源的执行器，这里由于我们没有在外部配置缓存策略所以，直接从源数据加载，看下面代码：

```
  private DataFetcherGenerator getNextGenerator() {
    switch (stage) {
      //从资源缓存执行器
      case RESOURCE_CACHE:
        return new ResourceCacheGenerator(decodeHelper, this);
      //源数据磁盘缓存执行器
      case DATA_CACHE:
        return new DataCacheGenerator(decodeHelper, this);
      //什么都没有配置，源数据的执行器
      case SOURCE:
        return new SourceGenerator(decodeHelper, this);
      case FINISHED:
        return null;
```

```
      default:
        throw new IllegalStateException("Unrecognized stage: " +
stage);
    }
  }
```

同学们知道，由于我们什么都没有配置，返回的是 `SourceGenerator` 源数据执行器。继续下面代码执行：

```
  private void runGenerators() {
    currentThread = Thread.currentThread();
    startFetchTime = LogTime.getLogTime();
    boolean isStarted = false;
    //判断是否取消，是否开始
    //调用 DataFetcherGenerator.startNext() 判断是否是属于开始执行的任
务
    while (!isCancelled && currentGenerator != null
        && !(isStarted = currentGenerator.startNext())) {

      ....
  }
```

同学们注意：上面代码先看 `currentGenerator.startNext()` 这句代码，`DataFetcherGenerator` 是一个抽象类，那么这里执行的实现类是哪一个，可以参考下面说明：

Stage.RESOURCE_CACHE【状态标记】 ---- 从磁盘中获取缓存的资源数据【作用】 --- ResourceCacheGenerator【执行器】

Stage.DATA_CACHE【状态标记】 ---- 从磁盘中获取缓存的源数据【作用】 --- DataCacheGenerator【执行器】

Stage.SOURCE【状态标记】 --- 一次新的请求任务 --- SourceGenerator【执行器】

因为这里我们没有配置缓存，那么直接看 `SourceGenerator`

```
  @Override
  public boolean startNext() {
        ...
    loadData = null;
    boolean started = false;
    while (!started && hasNextModelLoader()) {
      //获取一个 ModelLoad 加载器
      loadData = helper.getLoadData().get(loadDataListIndex++);
      if (loadData != null
```

```
          &&
(helper.getDiskCacheStrategy().isDataCacheable(loadData.fetcher.getDataSource())
          ||
helper.hasLoadPath(loadData.fetcher.getDataClass()))) {
        started = true;
        //使用加载器中的 fetcher 根据优先级加载数据
        loadData.fetcher.loadData(helper.getPriority(), this);
      }
    }
    return started;
  }
```

这里同学们看 `helper.getLoadData()` 获取的是一个什么样的加载器，我们可以先猜一下，因为没有配置任何缓存，所以可以猜得到是 http 请求了，那么是不是猜测的那样的，同学们我们一起来验证下。

```
  List<LoadData<?>> getLoadData() {
    if (!isLoadDataSet) {
      isLoadDataSet = true;
      loadData.clear();
      //从 Glide 注册的 Model 来获取加载器（注册是在 Glide 初始化的时候通过 registry
       // .append（）添加的）
      List<ModelLoader<Object, ?>> modelLoaders =
glideContext.getRegistry().getModelLoaders(model);

      for (int i = 0, size = modelLoaders.size(); i < size; i++) {
        ModelLoader<Object, ?> modelLoader = modelLoaders.get(i);
        LoadData<?> current =
          //开始构建加载器
            modelLoader.buildLoadData(model, width, height,
options);
        //如果架子啊器不为空，那么添加进临时缓存
        if (current != null) {
          loadData.add(current);
        }
      }
    }
    return loadData;
  }
```

首先拿到一个加载器的容器，加载器是在 Glide 初始化的时候 通过 `Registry.append()` 添加的，这里因为同学们我们以网络链接举例的。所以，ModelLoad 的实现类是 `HttpGlideUrlLoader` 加载器，我们看下它的具体实现：

**【同学们注意：之前有开发者看了一周Glide源码，也找不到网络请求的地方，我们现在就已经找到了，很伟大了，可以给自己鼓掌】**

```
  @Override
  public LoadData<InputStream> buildLoadData(@NonNull GlideUrl
model, int width, int height,
      @NonNull Options options) {
    GlideUrl url = model;
    if (modelCache != null) {
      url = modelCache.get(model, 0, 0);
      if (url == null) {
        modelCache.put(model, 0, 0, model);
        url = model;
      }
    }
    int timeout = options.get(TIMEOUT);
    // 【同学们注意：之前有开发者看了一周Glide源码，也找不到网络请求的地方，我
们现在就已经找到了，很伟大了，可以给自己鼓掌】
    return new LoadData<>(url, new HttpUrlFetcher(url, timeout));
  }
```

这里看到是返回的一个 `HttpUrlFetcher` 给加载器。加载器我们拿到了，现在开始加载，返回到刚刚的源码，请看下面：

```
class DataCacheGenerator implements DataFetcherGenerator,
    DataFetcher.DataCallback<Object> {

      //挑重要代码
       @Override
  public boolean startNext() {
....
    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;
```

```
        //通过拿到的加载器，开始加载数据
        loadData.fetcher.loadData(helper.getPriority(), this);
      }
    }
    return started;
  }
}
```

因为刚刚同学们知道了这里拿到的加载器是`HttpUrlFetcher`所以我们直接看它的 loadData 实现:

```
  @Override
  public void loadData(@NonNull Priority priority,
      @NonNull DataCallback<? super InputStream> callback) {
    long startTime = LogTime.getLogTime();
    try {
      //http 请求，返回一个 InputStream 输入流
      InputStream result =
loadDataWithRedirects(glideUrl.toURL(), 0, null,
glideUrl.getHeaders());
      //将 InputStream 以回调形式回调出去
      callback.onDataReady(result);
    } catch (IOException e) {
      callback.onLoadFailed(e);
    } finally {
     ...
    }
  }
```

同学们继续看`loadDataWithRedirects`这个函数是怎么生成的一个`InputStream`:

```
  private InputStream loadDataWithRedirects(URL url, int
redirects, URL lastUrl,
      Map<String, String> headers) throws IOException {
    if (redirects >= MAXIMUM_REDIRECTS) {
      throw new HttpException("Too many (> " + MAXIMUM_REDIRECTS
+ ") redirects!");
    } else {

      try {
        if (lastUrl != null &&
url.toURI().equals(lastUrl.toURI())) {
          throw new HttpException("In re-direct loop");
        }
```

```
      } catch (URISyntaxException e) {
        // Do nothing, this is best effort.
      }
    }

    urlConnection = connectionFactory.build(url);
    for (Map.Entry<String, String> headerEntry :
headers.entrySet()) {
      urlConnection.addRequestProperty(headerEntry.getKey(),
headerEntry.getValue());
    }
    urlConnection.setConnectTimeout(timeout);
    urlConnection.setReadTimeout(timeout);
    urlConnection.setUseCaches(false);
    urlConnection.setDoInput(true);

    urlConnection.setInstanceFollowRedirects(false);
    urlConnection.connect();

    stream = urlConnection.getInputStream();
    if (isCancelled) {
      return null;
    }
    final int statusCode = urlConnection.getResponseCode();
    if (isHttpOk(statusCode)) {
      return getStreamForSuccessfulRequest(urlConnection);
    }
    ...//抛的异常我们暂时先不管
  }
```

已经到了同学们熟悉的 Http 请求了，这里是 HttpURLConnection 作为 Glide 底层成网络请求的。请求成功之后直接返回的是一个输入流，最后会通过 `onDataReady` 回调到 DecodeJob onDataFetcherReady 函数中。同学们我们跟下回调，回调到 `SourceGenerator`：

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

这里会有 else 因为我们没有配置缓存,继续回调:

```
class DecodeJob<R> implements
DataFetcherGenerator.FetcherReadyCallback,
    Runnable,
    Comparable<DecodeJob<?>>,
    Poolable {
      ...
      @Override
      public void onDataFetcherReady(Key sourceKey, Object data,
DataFetcher<?> fetcher,
          DataSource dataSource, Key attemptedKey) {
        this.currentSourceKey = sourceKey; //当前返回数据的 key
        this.currentData = data; //返回的数据
        this.currentFetcher = fetcher; //返回的数据执行器，这里可以理解
为 HttpUrlFetcher
        this.currentDataSource = dataSource; //数据来源 url
        this.currentAttemptingKey = attemptedKey;
        if (Thread.currentThread() != currentThread) {
          runReason = RunReason.DECODE_DATA;
          callback.reschedule(this);
        } else {

 GlideTrace.beginSection("DecodeJob.decodeFromRetrievedData");
          try {
            //解析返回回来的数据
            decodeFromRetrievedData();
          } finally {
            GlideTrace.endSection();
```

```
            }
          }
        }
        ...
      }

    //解析返回的数据
    private void decodeFromRetrievedData() {
      Resource<R> resource = null;
      try {
        // 调用 decodeFrom 解析 数据：HttpUrlFetcher , InputStream ,
currentDataSource
        resource = decodeFromData(currentFetcher, currentData,
currentDataSource);
      } catch (GlideException e) {
        e.setLoggingDetails(currentAttemptingKey,
currentDataSource);
        throwables.add(e);
      }
      //解析完成后，通知下去
      if (resource != null) {
        notifyEncodeAndRelease(resource, currentDataSource);
      } else {
        runGenerators();
      }
    }
```

同学们继续跟 decodeFromData 看看怎么解析成 Resource 的：

```
    private <Data> Resource<R> decodeFromData(DataFetcher<?>
fetcher, Data data,
        DataSource dataSource) throws GlideException {
      ...
      Resource<R> result = decodeFromFetcher(data, dataSource);
      ....
      return result;
    } finally {
      fetcher.cleanup();
    }
  }

  @SuppressWarnings("unchecked")
  private <Data> Resource<R> decodeFromFetcher(Data data,
DataSource dataSource)
      throws GlideException {
```

```
    //获取当前数据类的解析器 LoadPath
    LoadPath<Data, ?, R> path =
decodeHelper.getLoadPath((Class<Data>) data.getClass());
    //通过 LoadPath 解析器来解析数据
    return runLoadPath(data, dataSource, path);
  }

  private <Data, ResourceType> Resource<R> runLoadPath(Data data,
DataSource dataSource,
      LoadPath<Data, ResourceType, R> path) throws GlideException
{
    Options options = getOptionsWithHardwareConfig(dataSource);

    //因为这里返回的是一个 InputStream 所以 这里拿到的是
InputStreamRewinder
    DataRewinder<Data> rewinder =
glideContext.getRegistry().getRewinder(data);
    try {
      //将解析资源的任务转移到 Load.path 方法中
      return path.load(
          rewinder, options, width, height, new
DecodeCallback<ResourceType>(dataSource));
    } finally {
      rewinder.cleanup();
    }
  }
```

同学们注意上面代码，为了解析数据首先构建一个 LoadPath, 然后创建一个 InputStreamRewinder 类型的 DataRewinder, 最终将数据解析的操作放到了 LoadPath.load 方法中， 接下来看下 LoadPath.load 方法的具体逻辑操作：

```
  public Resource<Transcode> load(DataRewinder<Data> rewinder,
@NonNull Options options, int width,
      int height, DecodePath.DecodeCallback<ResourceType>
decodeCallback) throws GlideException {

    try {
      return loadWithExceptionList(rewinder, options, width,
height, decodeCallback, throwables);
    } finally {
      listPool.release(throwables);
    }
  }
```

```
  private Resource<Transcode>
loadWithExceptionList(DataRewinder<Data> rewinder,
      @NonNull Options options,
      int width, int height,
DecodePath.DecodeCallback<ResourceType> decodeCallback,
      List<Throwable> exceptions) throws GlideException {
    Resource<Transcode> result = null;

    //遍历内部存储的 DecodePath 集合，通过他们来解析数据
    for (int i = 0, size = decodePaths.size(); i < size; i++) {
      DecodePath<Data, ResourceType, Transcode> path =
decodePaths.get(i);
      try {
        //这里才是真正解析数据的地方
        result = path.decode(rewinder, width, height, options,
decodeCallback);
      } catch (GlideException e) {
       ...
      }
     ...
    return result;
  }
```

同学们看看path.decode：

```
  public Resource<Transcode> decode(DataRewinder<DataType>
rewinder, int width, int height,
      @NonNull Options options, DecodeCallback<ResourceType>
callback) throws GlideException {
    //调用 decodeResourec 将数据解析成中间资源
    Resource<ResourceType> decoded = decodeResource(rewinder,
width, height, options);
    //解析完数据回调出去
    Resource<ResourceType> transformed =
callback.onResourceDecoded(decoded);
    //转换资源为目标资源
    return transcoder.transcode(transformed, options);
  }
```

同学们看看 decodeResource 怎么解析成中间资源的：

```
  @NonNull
  private Resource<ResourceType>
decodeResource(DataRewinder<DataType> rewinder, int width,
```

```
      int height, @NonNull Options options) throws GlideException
{
   ...
    try {
      return decodeResourceWithList(rewinder, width, height,
options, exceptions);
    } finally {
    ...
    }
  }

  @NonNull
  private Resource<ResourceType>
decodeResourceWithList(DataRewinder<DataType> rewinder, int
width,
      int height, @NonNull Options options, List<Throwable>
exceptions) throws GlideException {
    Resource<ResourceType> result = null;
    //noinspection ForLoopReplaceableByForEach to improve perf
    for (int i = 0, size = decoders.size(); i < size; i++) {
      ResourceDecoder<DataType, ResourceType> decoder =
decoders.get(i);
      try {
        DataType data = rewinder.rewindAndGet();
        if (decoder.handles(data, options)) {
          data = rewinder.rewindAndGet();
          // 调用 ResourceDecoder.decode 解析数据
          result = decoder.decode(data, width, height, options);
        }

      } catch (IOException | RuntimeException | OutOfMemoryError
e) {
                ...
      }
    return result;
  }
```

同学们可以看到数据解析的任务最终是通过 DecodePath 来执行的, 它内部有三大步操作

第一大步：deResource 将源数据解析成资源（源数据: InputStream， 中间产物: Bitmap）

第二大步：调用 DecodeCallback.onResourceDecoded 处理资源

第三大步：调用 ResourceTranscoder.transcode 将资源转为目标资源（目标资源类型: Drawable)

同学们可以发现，通过上面的 decoder.decode 源码可知，它是一个接口，由于我们这里的源数据是 InputStream,所以，它的实现类是 **StreamBitmapDecoder类** ,同学们我们就来看下 它内部的解码过程：

```
  @Override
  public Resource<Bitmap> decode(@NonNull InputStream source, int
width, int height,
      @NonNull Options options)
      throws IOException {

    // Use to fix the mark limit to avoid allocating buffers that
fit entire images.
    final RecyclableBufferedInputStream bufferedStream;
    final boolean ownsBufferedStream;

    ....

    try {
      // 根据请求配置来对数据进行采样压缩，获取一个 Resource<Bitmap>
      return downsampler.decode(invalidatingStream, width,
height, options, callbacks);
    } finally {
      ....
    }
  }
```

同学们注意：具体怎么采样压缩，同学们先不用关注具体实现(**先不关心支线，只管主线**)，现在拿到了一个 Bitmap 数据，我们需要通过回调出去，请看下面代码：

```
  public Resource<Transcode> decode(DataRewinder<DataType>
rewinder, int width, int height,
      @NonNull Options options, DecodeCallback<ResourceType>
callback) throws GlideException {
    //第一步： 调用 decodeResourec 将数据解析成中间资源 Bitmap
    Resource<ResourceType> decoded = decodeResource(rewinder,
width, height, options);
    //第二步： 解析完数据回调出去
    Resource<ResourceType> transformed =
callback.onResourceDecoded(decoded);
    //第三步： 转换资源为目标资源 Bitmap to Drawable
    return transcoder.transcode(transformed, options);
  }
```

同学们只看第二注释里面回调，最后会回调到 DecodeJob：

```
class DecodeJob<R> implements
DataFetcherGenerator.FetcherReadyCallback,
    Runnable,
    Comparable<DecodeJob<?>>,
    Poolable {
      ...
    @Override
    public Resource<Z> onResourceDecoded(@NonNull Resource<Z>
decoded) {
      return DecodeJob.this.onResourceDecoded(dataSource,
decoded);
    }
      ...
}
```

同学们辛苦了，在坚持看下去：

```
  @Synthetic
  @NonNull
  <Z> Resource<Z> onResourceDecoded(DataSource dataSource,
      @NonNull Resource<Z> decoded) {
    @SuppressWarnings("unchecked")
    //获取资源类型
    Class<Z> resourceSubClass = (Class<Z>)
decoded.get().getClass();
    Transformation<Z> appliedTransformation = null;
    Resource<Z> transformed = decoded;
    //如果不是从磁盘资源中获取需要进行 transform 操作
```

```
    if (dataSource != DataSource.RESOURCE_DISK_CACHE) {
      appliedTransformation =
decodeHelper.getTransformation(resourceSubClass);
      transformed = appliedTransformation.transform(glideContext,
decoded, width, height);
    }
    ...
    //构建数据编码的策略
    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;
    }

    //根据编码策略，构建缓存 Key
    Resource<Z> result = transformed;
    boolean isFromAlternateCacheKey =
!decodeHelper.isSourceKey(currentSourceKey);
    if
(diskCacheStrategy.isResourceCacheable(isFromAlternateCacheKey,
dataSource,
        encodeStrategy)) {
      if (encoder == null) {
        throw new
Registry.NoResultEncoderAvailableException(transformed.get().getC
lass());
      }
      final Key key;
      switch (encodeStrategy) {
        case SOURCE:
          //源数据 key
          key = new DataCacheKey(currentSourceKey, signature);
          break;
          ... 省略 成吨的代码
      }
      //初始化编码管理者，用于提交内存缓存
      LockedResource<Z> lockedResult =
LockedResource.obtain(transformed);
      deferredEncodeManager.init(key, encoder, lockedResult);
      result = lockedResult;
    }
```

```
    //返回转换后的 Bitmap
    return result;
  }
```

同学们可以看到 onResourceDecoded 中, 主要是对中间资源做了如下的操作:

第一步：对资源进行了转换操作。比如 Fit_Center,CenterCrop, 这些都是在请求的时候配置的

第二步：构建磁盘缓存的 key

同学们注意：最终就是将 **Bitmap** 转换成 **Drawable** 了操作了，请看下面代码

```
public class DecodePath<DataType, ResourceType, Transcode> {
  省略成吨的代码 ...
  Resource<Transcode> decode(DataRewinder<DataType> rewinder, int
width, int height,
      @NonNull Options options, DecodeCallback<ResourceType>
callback) throws GlideException {
    //第一步： 调用 decodeResourec 将数据解析成中间资源 Bitmap
    Resource<ResourceType> decoded = decodeResource(rewinder,
width, height, options);
    //第二步： 解析完数据回调出去
    Resource<ResourceType> transformed =
callback.onResourceDecoded(decoded);
    //第三步： 转换资源为目标资源 Bitmap to Drawable
    return transcoder.transcode(transformed, options);
  }
  省略成吨的代码 ...
}
```

同学们只看第三步，通过源码可知，ResourceTranscoder 是一个接口，又因为解析完的数据是 Bitmap 所以它的实现类是 BitmapDrawableTranscoder，最后看下它的 transcode 具体实现:

```
public class BitmapDrawableTranscoder implements
ResourceTranscoder<Bitmap, BitmapDrawable> {
  @Nullable
  @Override
  public Resource<BitmapDrawable> transcode(@NonNull
Resource<Bitmap> toTranscode,
      @NonNull Options options) {
    return LazyBitmapDrawableResource.obtain(resources,
toTranscode);
  }
}
```

具体同学们辛苦看下 `LazyBitmapDrawableResource.obtain`，【希望同学们理解：这也是没有办法，这个框架太庞大了】

```
public final class LazyBitmapDrawableResource implements
Resource<BitmapDrawable>,
    Initializable {

  private final Resources resources;
  private final Resource<Bitmap> bitmapResource;

  @Deprecated
  public static LazyBitmapDrawableResource obtain(Context
context, Bitmap bitmap) {
    return
        (LazyBitmapDrawableResource)
            obtain(
                context.getResources(),
                BitmapResource.obtain(bitmap,
Glide.get(context).getBitmapPool()));
  }

  @Deprecated
  public static LazyBitmapDrawableResource obtain(Resources
resources, BitmapPool bitmapPool,
      Bitmap bitmap) {
    return
        (LazyBitmapDrawableResource) obtain(resources,
BitmapResource.obtain(bitmap, bitmapPool));
  }

  @Nullable
  public static Resource<BitmapDrawable> obtain(
```

```
      @NonNull Resources resources, @Nullable Resource<Bitmap>
bitmapResource) {
    if (bitmapResource == null) {
      return null;
    }
    return new LazyBitmapDrawableResource(resources,
bitmapResource);
  }

  private LazyBitmapDrawableResource(@NonNull Resources
resources,
      @NonNull Resource<Bitmap> bitmapResource) {
    this.resources = Preconditions.checkNotNull(resources);
    this.bitmapResource =
Preconditions.checkNotNull(bitmapResource);
  }

  @NonNull
  @Override
  public Class<BitmapDrawable> getResourceClass() {
    return BitmapDrawable.class;
  }

  // Get 方法反回了一个 BitmapDrawable 对象
  @NonNull
  @Override
  public BitmapDrawable get() {
    return new BitmapDrawable(resources, bitmapResource.get());
  }

  @Override
  public int getSize() {
    return bitmapResource.getSize();
  }

  @Override
  public void recycle() {
    bitmapResource.recycle();
  }

  @Override
  public void initialize() {
    if (bitmapResource instanceof Initializable) {
      ((Initializable) bitmapResource).initialize();
    }
```

```
  }
}
```

同学们转化终于完成了，将我们解析到的 bitmap 存放到 LazyBitmapDrawableResource 内部, 然后外界通过 get 方法就可以获取到一个 BitmapDrawable 的对象了，解析完就到了展示数据了，同学们请看下面代码:

```
class DecodeJob<R> implements
DataFetcherGenerator.FetcherReadyCallback,
    Runnable,
    Comparable<DecodeJob<?>>,
    Poolable {

  //解析返回的数据
  private void decodeFromRetrievedData() {
    Resource<R> resource = null;
    try {
      //第一步： 调用 decodeFrom 解析 数据：HttpUrlFetcher ,
InputStream , currentDataSource
      resource = decodeFromData(currentFetcher, currentData,
currentDataSource);
    } catch (GlideException e) {
      e.setLoggingDetails(currentAttemptingKey,
currentDataSource);
      throwables.add(e);
    }
    //第二步： 解析完成后，通知下去
    if (resource != null) {
      notifyEncodeAndRelease(resource, currentDataSource);
    } else {
      runGenerators();
    }
 }
```

第一步就解析完了数据， 现在第二步执行 **notifyEncodeAndRelease**函数:

```
private void notifyEncodeAndRelease(Resource<R> resource,
DataSource dataSource) {
    ...
    //通知调用层数据已经装备好了
    notifyComplete(result, dataSource);

    stage = Stage.ENCODE;
    try {
      //这里就是将资源磁盘缓存
```

```
      if (deferredEncodeManager.hasResourceToEncode()) {
        deferredEncodeManager.encode(diskCacheProvider, options);
      }
    } finally {
     ...
    }
    //完成
    onEncodeComplete();
  }

  private void notifyComplete(Resource<R> resource, DataSource
dataSource) {
    setNotifiedOrThrow();
    // 在 DecodeJob 的构建中，我们知道这个 Callback 是 EngineJob
    callback.onResourceReady(resource, dataSource);
  }
}
```

同学们可以看到上面的 DecodeJob.**decodeFromRetrievedData** 中主要做了三个处理：

第一个处理：解析返回回来的资源。

第二个处理：拿到解析的资源，如果配置了本地缓存，就缓存到磁盘。

第三个处理：通知上层资源准备就绪，可以使用了。

同学们我们直接看 EngineJob 的 onResourceReady 回调函数：

```
  @Override
  public void onResourceReady(Resource<R> resource, DataSource
dataSource) {
    synchronized (this) {
      this.resource = resource;
      this.dataSource = dataSource;
    }
    notifyCallbacksOfResult();
  }

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

```
      if (isCancelled) {
        resource.recycle();
        release();
        return;
      } else if (cbs.isEmpty()) {
      ...
      }
      engineResource = engineResourceFactory.build(resource,
isCacheable);
      hasResource = true;
      copy = cbs.copy();
      incrementPendingCallbacks(copy.size() + 1);

      localKey = key;
      localResource = engineResource;
    }

    //回调上层 Engine 任务完成了
    listener.onEngineJobComplete(this, localKey, localResource);

    //遍历资源回调给 ImageViewTarget
    for (final ResourceCallbackAndExecutor entry : copy) {
      entry.executor.execute(new CallResourceReady(entry.cb));
    }
    decrementPendingCallbacks();
  }
```

通过上面 EngineJob 的 onResourceReady 回调函数 主要做了 两个处理：

第一个处理：通知上层任务完成。

第二个处理：回调 ImageViewTarget 用于展示数据。

辛苦同学们看下 listener.onEngineJobComplete 具体实现：

```
  @SuppressWarnings("unchecked")
  @Override
  public synchronized void onEngineJobComplete(
      EngineJob<?> engineJob, Key key, EngineResource<?>
resource) {
    if (resource != null) {
      resource.setResourceListener(key, this);
            //收到下游返回回来的资源，添加到活动缓存中
      if (resource.isCacheable()) {
        activeResources.activate(key, resource);
      }
    }
    jobs.removeIfCurrent(key, engineJob);
  }
```

最终通知 ImageViewTarget，同学们看下具体操作：

```
    //遍历资源回调给 ImageViewTarget
    for (final ResourceCallbackAndExecutor entry : copy) {
      entry.executor.execute(new CallResourceReady(entry.cb));
    }
```

```
  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)) {
         ...
          //返回准备好的资源
          callCallbackOnResourceReady(cb);
          removeCallback(cb);
        }
        decrementPendingCallbacks();
      }
    }
  }
```

同学们可以看到 CallResourceReady 实现 Runnable， 当 entry.executor.execute 线程池执行的时候就会调用 run， 最后我们继续跟 callCallbackOnResourceReady 函数：

```
  @Synthetic
  synchronized void callCallbackOnResourceReady(ResourceCallback
cb) {
    try {
      //回调给 SingleRequest
      cb.onResourceReady(engineResource, dataSource);
    } catch (Throwable t) {
      throw new CallbackException(t);
    }
  }
```

辛苦同学们看，SingleRequest onResourceReady 回调实现：

```
 public synchronized void onResourceReady(Resource<?> resource,
DataSource dataSource) {
    stateVerifier.throwIfRecycled();
    loadStatus = null;
    ... 省略成吨的代码
    Object received = resource.get();
    if (received == null ||
!transcodeClass.isAssignableFrom(received.getClass())) {
      releaseResource(resource);
      ... 省略成吨的代码
      onLoadFailed(exception);
      return;
    }

    if (!canSetResource()) {
      releaseResource(resource);
      status = Status.COMPLETE;
      return;
    }

   //当资源准备好的时候
    onResourceReady((Resource<R>) resource, (R) received,
dataSource);
  }


private synchronized void onResourceReady(Resource<R> resource, R
result, DataSource dataSource) {
```

```
      ... 省略成吨的代码
      anyListenerHandledUpdatingTarget |=
          targetListener != null
              && targetListener.onResourceReady(result, model,
target, dataSource, isFirstResource);

      if (!anyListenerHandledUpdatingTarget) {
        Transition<? super R> animation =
            animationFactory.build(dataSource, isFirstResource);
        //回调给目标 ImageViewTarget 资源准备好了
        target.onResourceReady(result, animation);
      }
    } finally {
      isCallingCallbacks = false;
    }
    //加载成功
    notifyLoadSuccess();
  }
```

同学们这一步主要把准备好的资源回调给显示层，看下面代码【重要在显示阶段了，同学们看到曙光了】

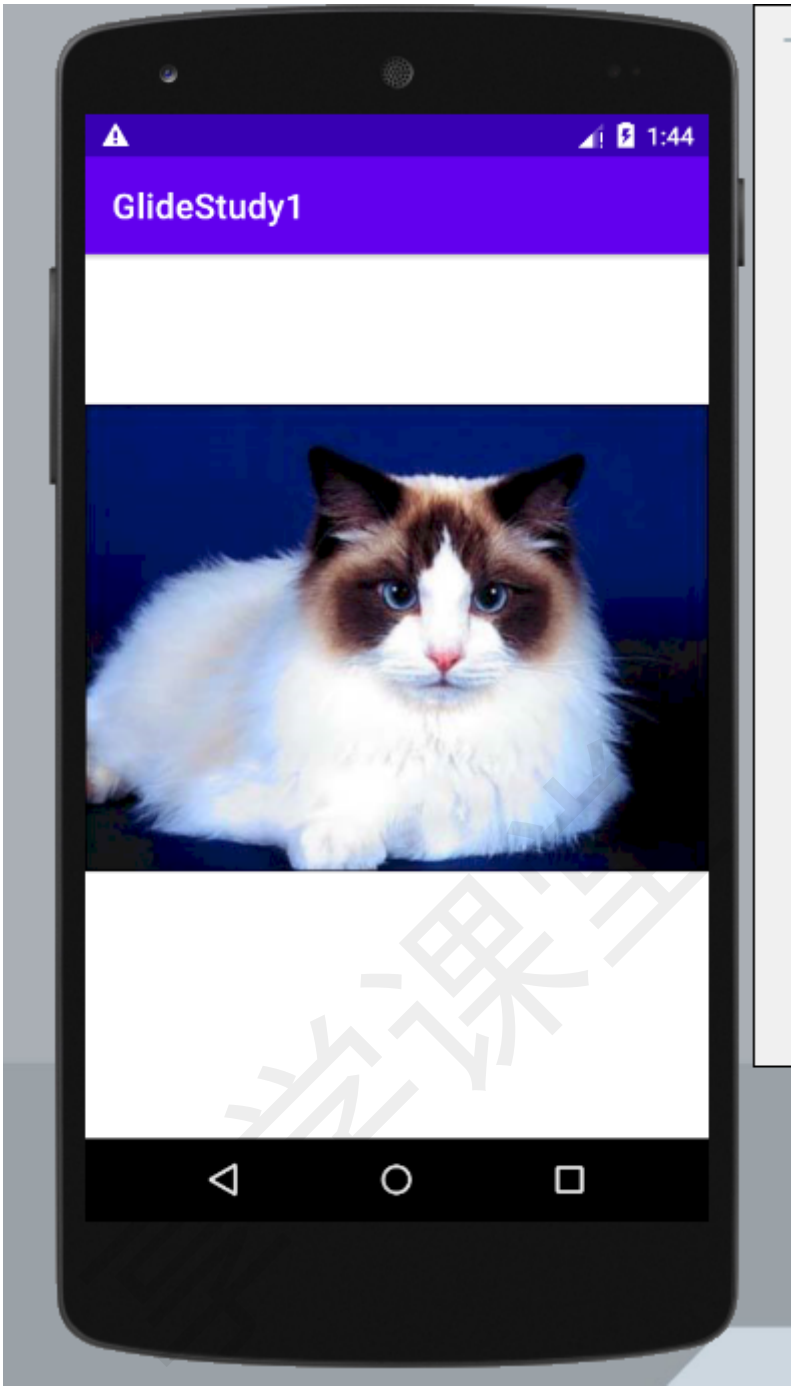
```
public abstract class ImageViewTarget<Z> extends
ViewTarget<ImageView, Z>
    implements Transition.ViewAdapter {
  ...
  @Override
  public void onResourceReady(@NonNull Z resource, @Nullable
Transition<? super Z> transition) {
    if (transition == null || !transition.transition(resource,
this))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

  protected abstract void setResource(@Nullable Z resource);
  ...
}

  private void setResourceInternal(@Nullable Z resource) {
    //调用 setResource 函数，将资源显示出来
    setResource(resource);
    ...
```

```
  }
```

同学们还记得么？在最开始构建的时候，我们知道只有调用 asBitmap 的时候实现类是 `BitmapImageViewTarget`,在这里的测试，并没有调用这个函数，所以它的实现类是 `DrawableImageViewTarget`,具体看下它内部实现：

```
public class DrawableImageViewTarget extends
ImageViewTarget<Drawable> {

  public DrawableImageViewTarget(ImageView view) {
    super(view);
  }

  // Public API.
  @SuppressWarnings({"unused", "deprecation"})
  @Deprecated
  public DrawableImageViewTarget(ImageView view, boolean
waitForLayout) {
    super(view, waitForLayout);
  }

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

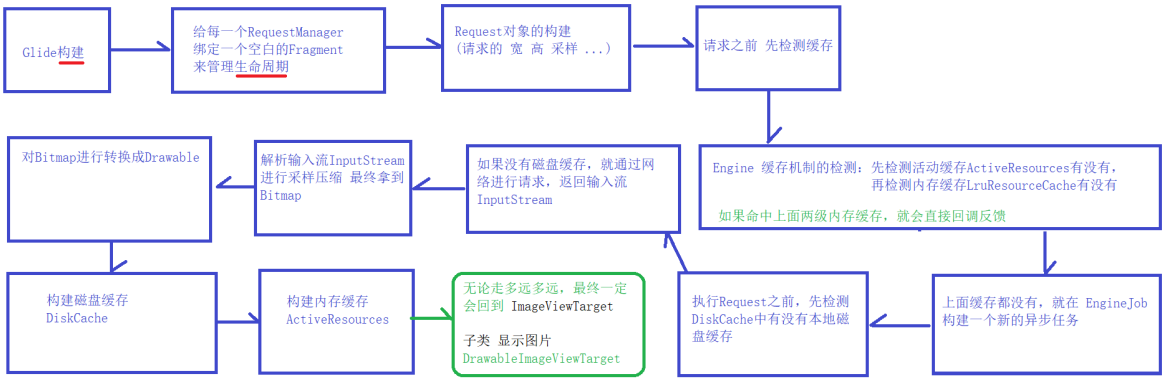
同学们注意，同学们注意，同学们注意，这里看到抽象类中调用了 setResource ,子类实现并调用了 **view.setImageDrawable(resource);** 图片现在算是真正的显示出来了。 我们就看到了图片的显示:




**淹死在源码中的原因是什么?**

答：给大家看源码的忠告，千万不要把每一行代码都看懂 的方式去看，这样的方式百分之九十以上的开发者，都淹死在代码中

最后，我给同学们来一个，最简单最简单的Glide流程简化图总结：




【环节四，**生命周期** 的意义：】


**生命周期的意义：**

**【就是 Glide框架内部 会搞一个空白的Fragment 关联到 用户的 Activity或者Fragment，当用户的Activity或者Fragment 发生Stop Start 的时候，空白的Fragment就监听到了，从而根据用户Activity或者Fragment的变化，从而做出自己框架的处理】**