Weex源码分析系列（四）之Adapter组件源码解析

1、前言

在上一篇文章《Weex渲染流程分析》中我们分析了Weex的渲染流程，**但是实际上串起来的还是Module以及Component组件，加上Weex特有的渲染流程。**

Module组件和Component组件对Weex Android SDK的意义非常大，属于没有这俩寸步难行的关系，包括今天我们要分析的Adapter也依赖于Module、Component组件，尤其是Js引擎与Module、Component交互的部分，Adapter表面上并不涉及，但是实际上息息相关。如果还有疑惑的话非常建议大家回过头再去看看之前的源码分析文章。

本篇文章我们就开始分析Weex中另一个组件Adapter，**对Weex的设计理解更深一步。**

2、初识Adapter

2.1 Adapter的定位

在《Android 扩展》中我们可以看到Adapter的定位：

> Adapter 扩展 Weex 对一些基础功能实现了统一的接口，可实现这些接口来定制自己的业务。例如：图片下载等。

此处可以看到：**Weex对Adapter的定位是基础功能的定义，可以实现这些接口自己进行实现。**

2.2 Adapter的使用

实际上在我的WeexList项目中已经有了关于Adapter的使用，因为Weex并没有实现默认的图片加载功能。

Adapter的注册：

```
    InitConfig config = new InitConfig.Builder().setImgAdapter(new
WeexImageAdapter()).build();
    WXSDKEngine.initialize(this, config);
```

Adapter的实现：

```
public class WeexImageAdapter implements IWXImgLoaderAdapter {

    @Override
    public void setImage(String url, ImageView view, WXImageQuality
quality, WXImageStrategy strategy) {
        Glide.with(view.getContext())
                .load(url)
                .error(R.mipmap.me_image_man)
                .into(view);
    }
}
```

可以看到我们实现了Weex的IWXImgLoaderAdapter接口，自己实现了图片加载的能力。需要注意的是Adapter的注册和Module、Component的注册方式是不一样的。

2.3 Adapter与Module的区别

我们知道Module的定位是非UI性质的功能组件，那和Adapter是不是有点冲突？为什么还多了一个Adapter组件呢？

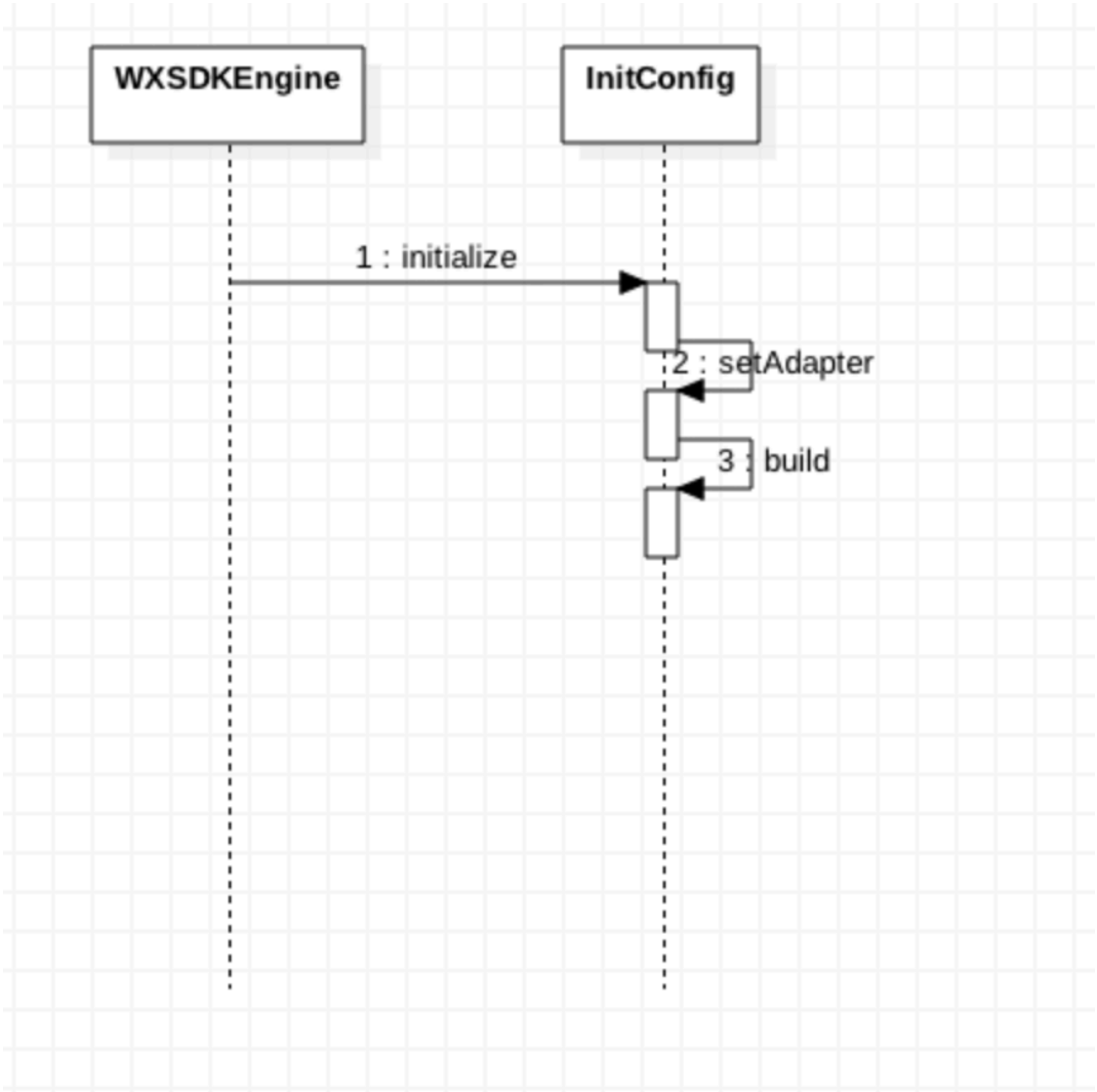
不要Adapter组件可不可以呢？其实是可以的，要做的事情都通过Module来做。但是一些基础功能例如图片加载、网络请求等不同的应用有自己的实现方式、依赖库，如果Weex自己再加一套，就显得冗余而且这些基础能力Weex定义好接口让接入的应用来提供就好了。

这就是Adapter存在的意义，你看Weex团队是多么的贴心啊！

备注： 那么我们此时可以由Adapter和Module的战略意义不一样，猜到两个组件的地位也是不一样的。通过翻阅源码，在InitConfig中Weex使用建造者模式来提供各种各样的Adapter的自定义，但是却不支持自己新增Adapter的类型。**此处也提现了Adapter的定位：基础功能实现了统一的接口。**

3、Adapter源码分析

3.1 Adapter注册



可以看到：**Adapter的注册实际上非常简单，只是通过WXSDKEngine初始化了一些配置信息而已，根本不需要像Module或者Component一样需要通过JsBridge也让Js引擎知道自己的存在。**

再次看出Adapter的定位：基础功能实现了统一的接口，具体的交互交给Module或Component来做，然后Module或Component来调用我们实现的Adapter。

3.2 Adapter调用

关于Adapter的调用就不再画图分析，因为实在不需要怎么分析。实际上对于Adapter的调用本质就是类的调用，**因为刚才在注册的时候设置了各种各样的Adapter，然**

后直接调用接口即可。

4、特定Adapter分析

对于Adapter的调用分为Component调用和Module调用（**Adapter处于调用链的下端**）

4.1 IWXImgLoaderAdapter（Component调用）

我们来介绍下实现图片加载能力的Adapter，这个Adapter没有被Weex默认实现。之前总结过组件的交互是通过JsBridge然后来调用Component被注解修饰的方法。

对于ImageView它在Weex里对应了WXImage这个Component。我们在Vue代码里写的src属性既然是需要通过Adapter实现的，那在WXImage中必定有方法对应src属性，果然我们发现了它。

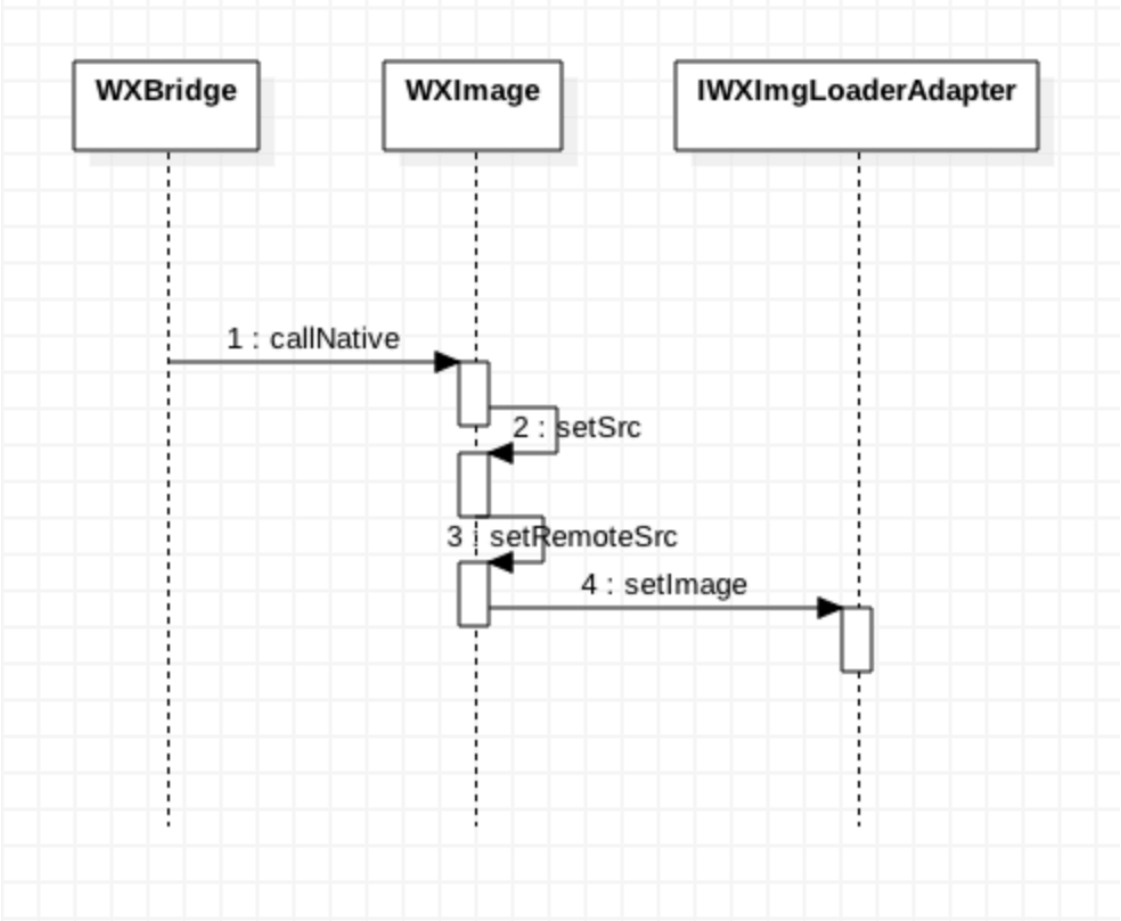
```
    @Component(lazyload = false)
    public class WXImage extends WXComponent<ImageView> {
        @WXComponentProp(name = Constants.Name.SRC)
        public void setSrc(String src) {
            if (src == null) {
                return;
            }
            this.mSrc = src;
            WXSDKInstance instance = getInstance();
            Uri rewrited = instance.rewriteUri(Uri.parse(src),
URIAdapter.IMAGE);

            if (Constants.Scheme.LOCAL.equals(rewrited.getScheme())) {
                setLocalSrc(rewrited);
            } else {
                int blur = 0;
                if (getDomObject() != null) {
                    String blurStr = getDomObject().getStyles().getBlur();
                    blur = parseBlurRadius(blurStr);
                }
                setRemoteSrc(rewrited, blur);
            }
        }

    private void setRemoteSrc(Uri rewrited, int blurRadius) {

        .....
```

```
        IWXImgLoaderAdapter imgLoaderAdapter =
getInstance().getImgLoaderAdapter();
        if (imgLoaderAdapter != null) {
            imgLoaderAdapter.setImage(rewrited.toString(), getHostView(),
                    getDomObject().getAttrs().getImageQuality(),
imageStrategy);
        }
    }
}
```



总结：

- **Js引擎通过JsBridge发送消息给客户端，最终调用到相关Component的具体方法；**
- **对于WXImage来说，被调用setSrc方法之后，会调用设置的IWXImgLoaderAdapter的setImage方法；**

4.2 IWXHttpAdapter（Module调用）

对于网络请求比较常用，Weex就做了默认的实现，但是之前在《Weex系列（四）之Module组件源码解析》分析过，实现比较简陋，最好重新实现。

我们先看下平时写Vue代码的时候使用网络请求是怎么做的：

```
    var stream = weex.requireModule('stream')
    stream.fetch
```

既然require的是名叫"stream"的Module，那么我们就在WXSDKEngine中找一下，果然在register()方法中找到了：

```
    registerModule("stream", WXStreamModule.class);
```

接下来，我们不用猜测，坚信WXStreamModule类中必定存在一个fetch方法；

```
  @JSMethod(uiThread = false)
  public void fetch(String optionsStr, final JSCallback callback,
JSCallback progressCallback){

    JSONObject optionsObj = null;
    try {
      optionsObj = JSON.parseObject(optionsStr);
    }catch (JSONException e){
      WXLogUtils.e("", e);
    }

    boolean invaildOption = optionsObj==null ||
optionsObj.getString("url")==null;
    if(invaildOption){
      if(callback != null) {
        Map<String, Object> resp = new HashMap<>();
        resp.put("ok", false);
        resp.put(STATUS_TEXT, Status.ERR_INVALID_REQUEST);
        callback.invoke(resp);
      }
      return;
    }

    //此处可以看到Http请求的那些参数最终是被怎么取的，这样即便是文档没写的一些设
置我们也可以根据取参数的方法反推出来怎么设置。
    String method = optionsObj.getString("method");
    String url = optionsObj.getString("url");
```

```
    JSONObject headers = optionsObj.getJSONObject("headers");
    String body = optionsObj.getString("body");
    String type = optionsObj.getString("type");
    int timeout = optionsObj.getIntValue("timeout");

    if (method != null) method = method.toUpperCase();
    Options.Builder builder = new Options.Builder()
            .setMethod(!"GET".equals(method)
                    &&!"POST".equals(method)
                    &&!"PUT".equals(method)
                    &&!"DELETE".equals(method)
                    &&!"HEAD".equals(method)
                    &&!"PATCH".equals(method)?"GET":method)
            .setUrl(url)
            .setBody(body)
            .setType(type)
            .setTimeout(timeout);

    extractHeaders(headers,builder);
    final Options options = builder.createOptions();

    // 真正请求网络去了，里面会调用IWXHttpAdapter
    sendRequest(options, new ResponseCallback() {
      @Override
      public void onResponse(WXResponse response, Map<String, String>
headers) {
        if(callback != null) {
          Map<String, Object> resp = new HashMap<>();
          if(response == null|| "-1".equals(response.statusCode)){
            resp.put(STATUS,-1);
            resp.put(STATUS_TEXT,Status.ERR_CONNECT_FAILED);
          }else {
            int code = Integer.parseInt(response.statusCode);
            resp.put(STATUS, code);
            resp.put("ok", (code >= 200 && code <= 299));
            if (response.originalData == null) {
              resp.put("data", null);
            } else {
              String respData = readAsString(response.originalData,
                      headers != null ? getHeader(headers, "Content-Type")
: ""
              );
              try {
                resp.put("data", parseData(respData, options.getType()));
              } catch (JSONException exception) {
                WXLogUtils.e("", exception);
                resp.put("ok", false);
                resp.put("data","{'err':'Data parse failed!'}");
```

```
              }
            }
            resp.put(STATUS_TEXT,
Status.getStatusText(response.statusCode));
          }
          resp.put("headers", headers);
          callback.invoke(resp);
        }
      }
    }, progressCallback);
  }


  private void sendRequest(Options options,ResponseCallback
callback,JSCallback progressCallback){
    WXRequest wxRequest = new WXRequest();
    wxRequest.method = options.getMethod();
    wxRequest.url = mWXSDKInstance.rewriteUri(Uri.parse(options.getUrl()),
URIAdapter.REQUEST).toString();
    wxRequest.body = options.getBody();
    wxRequest.timeoutMs = options.getTimeout();

    if(options.getHeaders()!=null)
    if (wxRequest.paramMap == null) {
      wxRequest.paramMap = options.getHeaders();
    }else{
      wxRequest.paramMap.putAll(options.getHeaders());
    }

    IWXHttpAdapter adapter = (mAdapter==null && mWXSDKInstance != null) ?
mWXSDKInstance.getWXHttpAdapter() : mAdapter;
    if (adapter != null) {
    // 调用了IWXHttpAdapter的sendRequest方法
      adapter.sendRequest(wxRequest, new
StreamHttpListener(callback,progressCallback));
    }else{
      WXLogUtils.e("WXStreamModule","No HttpAdapter found,request
failed.");
    }
  }
```

然后我们看下DefaultWXHttpAdapter的默认网络请求实现；

```
public class DefaultWXHttpAdapter implements IWXHttpAdapter {

  private static final IEventReporterDelegate DEFAULT_DELEGATE = new
```

```
NOPEventReportDelegate();
  private ExecutorService mExecutorService;

  private void execute(Runnable runnable){
    if(mExecutorService==null){
      mExecutorService = Executors.newFixedThreadPool(3);
    }
    mExecutorService.execute(runnable);
  }

  @Override
  public void sendRequest(final WXRequest request, final OnHttpListener
listener) {
    if (listener != null) {
      listener.onHttpStart();
    }
    execute(new Runnable() {
      @Override
      public void run() {
        WXResponse response = new WXResponse();
        IEventReporterDelegate reporter = getEventReporterDelegate();
        try {
          HttpURLConnection connection = openConnection(request,
listener);
          reporter.preConnect(connection, request.body);
          Map<String,List<String>> headers = connection.getHeaderFields();
          int responseCode = connection.getResponseCode();
          if(listener != null){
            listener.onHeadersReceived(responseCode,headers);
          }
          reporter.postConnect();

          response.statusCode = String.valueOf(responseCode);
          if (responseCode >= 200 && responseCode<=299) {
            InputStream rawStream = connection.getInputStream();
            rawStream = reporter.interpretResponseStream(rawStream);
            response.originalData = readInputStreamAsBytes(rawStream,
listener);
          } else {
            response.errorMsg =
readInputStream(connection.getErrorStream(), listener);
          }
          if (listener != null) {
            listener.onHttpFinish(response);
          }
        } catch (IOException|IllegalArgumentException e) {
          e.printStackTrace();
          response.statusCode = "-1";
```

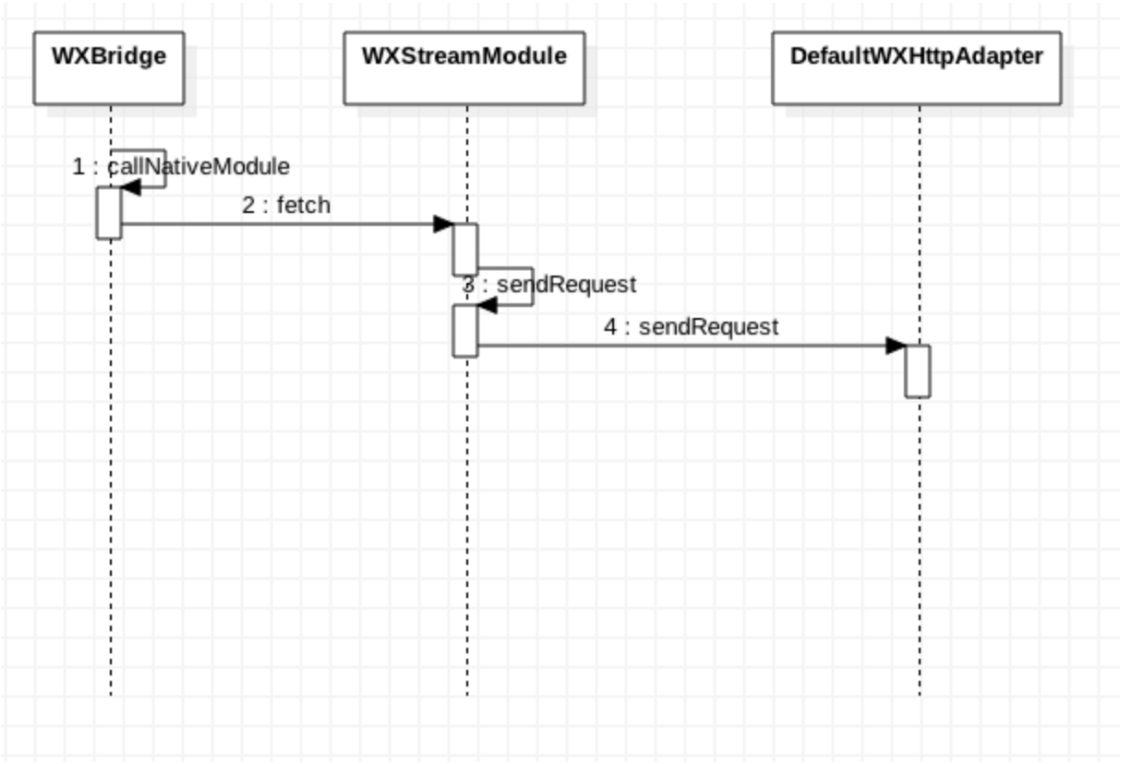
```
            response.errorCode="-1";
            response.errorMsg=e.getMessage();
            if(listener!=null){
              listener.onHttpFinish(response);
            }
            if (e instanceof IOException) {
              reporter.httpExchangeFailed((IOException) e);
            }
          }
        }
      });
    }
  }
```

可以看到Weex默认的网络请求是基于HttpURLConnection，一个核心池和最大池都是3的FixThreadPool。对于网络请求来说缺点显而易见：

- **没有Https的实现；**
- **线程池使用可以更优；**

但是对Weex来说实际上只是提供默认的简单实现，也没错，需要自己去重新定义。

接下来看图总结下：



总结：

- **Js引擎发消息来执行网络请求；**
- **调用到了WXStreamModule，调用其fetch方法；**
- **WXStreamModule里会调用IWXHttpAdapter的sendRequest方法，实现真正的网络请求；**

5、问题

Weex除了使用网络图片之外可以使用别的类型图片吗，例如直接使用drawable文件夹里的图片？

在刚开始接触到Weex的时候我内心的答案也是NO，毕竟drawable里的图片在常规的安卓开发中都是需要使用R文件来调用的。源码面前，了无秘密！我们就来看下WXImage的实现吧，在setSrc方法中有一个判断：

```
    if (Constants.Scheme.LOCAL.equals(rewrited.getScheme())) {
      setLocalSrc(rewrited);// 以local 开头的话则走到了这里
    } else {
      int blur = 0;
      if(getDomObject() != null) {
        String blurStr = getDomObject().getStyles().getBlur();
        blur = parseBlurRadius(blurStr);
      }
      setRemoteSrc(rewrited, blur);
    }
```

最终会走到这里：

```
  public static Drawable getDrawableFromLoaclSrc(Context context, Uri
rewrited) {
    Resources resources = context.getResources();
    List<String> segments = rewrited.getPathSegments();
    if (segments.size() != 1) {
      WXLogUtils.e("Local src format is invalid.");
      return null;
    }
    int id = resources.getIdentifier(segments.get(0), "drawable",
context.getPackageName());
    return id == 0 ? null : ResourcesCompat.getDrawable(resources, id,
null);
```

```
    }
```

老司机们已经明白了吧：通过资源名生成uri，然后还是拿到了资源对应的id，获取的图片。

备注：

- **如果不是仔细跟踪Weex源码的话，我们很容易给Weex贴上一个不能加载本地图片的标签。**
- **实际上，Weex也支持别的类型的图片调用方式，老司机们可以自己探索实现下。**

6、Adapter总结

- **Module和Component类是理解Adapter的前提；与Js引擎交互的部分被Module和Component做了，但是对理解很重要；**
- **Adapter的定位是扩展Weex对一些基础功能实现了统一的接口，可实现这些接口来定制自己的业务；**
- **Module自身的源码其实很简单，复杂的是上面与Module、Component相关的调用链；**
- **通过细读源码，可以发现很多问题的答案；带着问题去读，更加事半功倍；**

欢迎持续关注Weex源码分析项目：Weex-Analysis-Project

欢迎关注微信公众号：定期分享Java、Android干货！