Weex源码分析系列（六）之Weex SDK可借鉴细节总结

1、前言

经过前面五篇文章的源码分析及总结，我们对Weex的整体架构及核心源码都有了清晰的认识。**本篇文章主要总结我在Weex SDK源码阅读时觉得可以借鉴的细节。**

备注：本文侧重讲Weex SDK源码级别的可借鉴细节，对大方向上的可借鉴点比如动态化+Native思路、一项技术完整的生态等方面可以参考上一篇文章《深入Weex系列（五）之Weex SDK架构分析》。

2、建造者模式

在使用Weex之前我们都会进行Weex SDK的初始化，对于Weex SDK它的辅助配置类就使用到了建造者模式。

建造者模式主要解决：一个模块各个部分子对象的构建算法可能变化，但是各个部分子对象相互结合在一起的算法确实稳定的。一句话总结就是：模块整体构建过程稳定，但是构建的每一步可能有出入。

我们结合Weex的场景来具体分析下：Weex配置模块的构建过程是稳定的（都需要提供同样的能力），但是构建的每一步则可能有出入（每个配置的能力提供却可以多样）。

举例说明：例如Weex需要提供网络请求的基础能力（这个构建过程稳定），但是网络请求可以有不同的实现方式（具体的构建算法可能变化）。

```
InitConfig config = new InitConfig.Builder().
        setImgAdapter(new WeexImageAdapter()).
        setHttpAdapter(new WeexHttpAdapter).
        setSoLoader(new WeexSoLoaderAdapter).
        build();
```

好处：**调用者无需知道构建模块如何组装，也不会忘记组装某一部分，同时也提供给了开发者定制的能力。**

3、So的加载

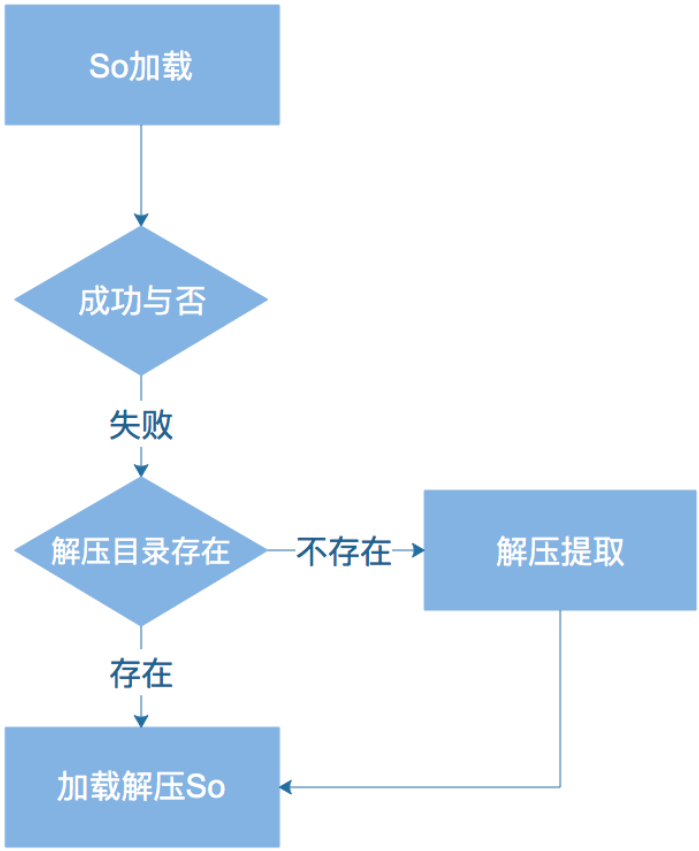
So的成功加载对Weex的运行至关重要，毕竟Weex需要V8引擎执行Js与Native的交互，源码中也可以看出So没有加载成功则Weex的各个模块不会执行。

而在线上Bug收集中我们会遇到UnsatisfiedLinkError错误，虽然不是频发性Bug，但是对于Weex而言一旦出现那么Weex就不可能再运行。于是Weex SDK对So加载这块做了优化，我们看下So加载的代码逻辑：

```
public static boolean initSo(String libName, int version,
IWXUserTrackAdapter utAdapter) {
    String cpuType = _cpuType();
    if (cpuType.equalsIgnoreCase(MIPS) ) {
      return false; // mips架构不支持，直接返回
    }

    boolean InitSuc = false;
    if (checkSoIsValid(libName, BuildConfig.ARMEABI_Size)
||checkSoIsValid(libName, BuildConfig.X86_Size)) {  // 校验So大小是否正常
      /**
       * Load library with {@link System#loadLibrary(String)}
       */
      try {
        // If a library loader adapter exists, use this adapter to load
library
        // instead of System.loadLibrary.
        if (mSoLoader != null) {
          mSoLoader.doLoadLibrary(libName);// 自定义SoLoader加载的话自己去加
载
        } else {
          System.loadLibrary(libName);// 默认加载的方式
        }
        commit(utAdapter, null, null);

        InitSuc = true;
      } catch (Exception | Error e2) {// So加载失败
        if (cpuType.contains(ARMEABI) || cpuType.contains(X86)) {
          commit(utAdapter, WXErrorCode.WX_ERR_LOAD_SO.getErrorCode(),
WXErrorCode.WX_ERR_LOAD_SO.getErrorMsg() + ":" + e2.getMessage());
        }
        InitSuc = false;
      }

      try {
        if (!InitSuc) {
```

```
          // 没有加载成功的话则从文件中加载
          //File extracted from apk already exists.
          if (isExist(libName, version)) {
            boolean res = _loadUnzipSo(libName, version, utAdapter);// 从
解压包中加载So
            if (res) {
              return res;
            } else {
              //Delete the corrupt so library, and extract it again.
              removeSoIfExit(libName, version);// 解压包也加载失败，删除；
            }
          }

          //Fail for loading file from libs, extract so library from so
and load it.
          if (cpuType.equalsIgnoreCase(MIPS)) {
            return false;
          } else {
            try {
              InitSuc = unZipSelectedFiles(libName, version, utAdapter);//
从apk中解压出来So，然后加载；
            } catch (IOException e2) {
              e2.printStackTrace();
            }
          }
        }
      } catch (Exception | Error e) {
        InitSuc = false;
        e.printStackTrace();
      }
    }
    return InitSuc;
}
```

可以看到Weex中有多项保障去保证So的成功加载，总结下流程图：



4、Weex的线程模型

各位老司机都知道多线程的好处也知道Android只有主线程才能更新UI，对于Weex来说它有自己完整的一套工作机制，如果所有任务都在主线程那势必会积压太多任务，导致任务得不到及时执行同时也有卡顿的风险。

Weex SDK也考虑到了这些，**分析Weex的机制可以知道任务主要花费在三方面：JSBridge相关、Dom相关、UI相关。于是对这三方面进行了细分，JSBridge相关的操作挪到JSBridge线程执行，Dom相关操作在Dom线程执行，避免了主线程积压太多任务。此处我们可以想到使用异步线程。同时对于单项的任务例如Dom操作，需要是串行的。如果使用线程池，实际上也发挥不出线程池的威力。**

分析到了这里。我们的需求其实就很明确了：避免异步线程的创建及销毁过程消耗资源，同时支持串行执行。我们可以设想一种线程能力：有任务的时候则执行，没有任务的时候则等待，是不是完美的符合我们的需求。

幸运的是Android其实已经为我们提供了这样的一个类：HandlerThread。大家可以参考我之前的一篇文章《Android性能优化（十一）之正确的异步姿势》。

```
// 贴出Weex中使用的HandlerThread实例
// JSBridge工作的Thread
mJSThread = new WXThread("WeexJSBridgeThread", this);
mJSHandler = mJSThread.getHandler();

// Dom工作的Thread
mDomThread = new WXThread("WeeXDomThread", new WXDomHandler(this));
mDomHandler = mDomThread.getHandler();
```

总结下Weex的线程模型：

- **JSBridge在WeexJSBridgeThread负责JS与Native的通信；**
- **切换具体的Dom指令到WeeXDomThread负责关于Dom的各项如：解析、Rebuild Dom Tree、Layout等操作；**
- **切换到UI线程，负责原生View的创建、布局、事件添加、数据绑定等；**

优势：

- 避免主线程的卡顿风险；
- 避免了线程的创建与销毁等资源消耗；
- 同时支持串行操作；

5、交互函数参数类型的处理

对于Weex的RunTime，再怎么强大也少不了与Native的交互（方法调用，使用Native的能力），前面的系列文章也详细分析了Module的交互原理。**但是有一个细节问题前面没有说到，就是JS与Native交互的方法签名，参数类型只能是String吗？**

回到WXBridge这个通信的桥梁，调用Native的方法都会走到callNative方法，然后走到WxBridgeManager.callNative方法，会发现函数体内有一行：

```
JSONArray array = JSON.parseArray(tasks);
```

由此可以断定JS传递给Native的参数首先不仅仅是普通String字符串，而是

Json格式。实际上不管是断点查看或者翻阅WXStreamModule的代码，都可以发现Json的踪影。

```
  @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);
        }
        ......
    }
```

不过以上发现还不足以解决我们的疑惑：参数类型只能是String吗？那必须不是！

首先回顾下在Module的注册过程中会有一步是获取Module中被打上注解的方法然后存在mMethodMap中；而在真正调用方法的地方是NativeInvokeHelper的invoke方法：

```
  public Object invoke(final Object target,final Invoker invoker,JSONArray
args) throws Exception {
    final Object[] params =
prepareArguments(invoker.getParameterTypes(),args);// 解析参数
    if (invoker.isRunOnUIThread()) {// 要求在主线程执行则抛到主线程执行；
      WXSDKManager.getInstance().postOnUiThread(new Runnable() {
        @Override
        public void run() {
          try {
            invoker.invoke(target, params);// 反射调用方法执行
          } catch (Exception e) {
            throw new RuntimeException(e);
          }
        }
      }, 0);
    } else {
      return invoker.invoke(target, params);
    }
    return null;
  }
```

我们再来详细跟踪下解析参数这步：

```
  private Object[] prepareArguments(Type[] paramClazzs, JSONArray args)
throws Exception {
    Object[] params = new Object[paramClazzs.length];
    Object value;
    Type paramClazz;
    for (int i = 0; i < paramClazzs.length; i++) {
      paramClazz = paramClazzs[i];
      if(i>=args.size()){
        if(!paramClazz.getClass().isPrimitive()) {
          params[i] = null;
          continue;
        }else {
          throw new Exception("[prepareArguments] method argument list not
match.");
        }
      }
      value = args.get(i);
      // JSONObject与JSCallback类型单独处理
      if (paramClazz == JSONObject.class) {
        params[i] = value;
      } else if(JSCallback.class == paramClazz){
        if(value instanceof String){
          params[i] = new SimpleJSCallback(mInstanceId,(String)value);
        }else{
          throw new Exception("Parameter type not match.");
        }
      } else {
        // 其它类型的参数
        params[i] = WXReflectionUtils.parseArgument(paramClazz,value);
      }
    }
    return params;
  }
```

看下其它参数类型的解析：

```
  public static Object parseArgument(Type paramClazz, Object value) {
    if (paramClazz == String.class) {
      return value instanceof String ? value : JSON.toJSONString(value);
    } else if (paramClazz == int.class) {
      return value.getClass().isAssignableFrom(int.class) ? value :
WXUtils.getInt(value);
    } else if (paramClazz == long.class) {
```

```
      return value.getClass().isAssignableFrom(long.class) ? value :
WXUtils.getLong(value);
    } else if (paramClazz == double.class) {
      return value.getClass().isAssignableFrom(double.class) ? value :
WXUtils.getDouble(value);
    } else if (paramClazz == float.class) {
      return value.getClass().isAssignableFrom(float.class) ? value :
WXUtils.getFloat(value);
    } else {
      return JSON.parseObject(value instanceof String ? (String) value :
JSON.toJSONString(value), paramClazz);
    }
  }
```

跟踪到此处就显而易见：JS与Native的交互参数不仅仅支持String。

我们再来总结下Weex是如何实现不同方法签名的交互的：

- **Module注册阶段保存下来Method；**
- **JS发送指令调用Module方法传递的原始参数是Json格式；**
- **真正反射调用方法的时候从Method中拿到参数的具体类型，然后从Json中读到相应的值，再进行转换。**

6、后记

本文主要记录了我在Weex源码阅读过程中觉得不错可以借鉴的细节，限于文章篇幅不能面面俱到。实际上不仅Weex的整体思路，Weex SDK的代码也非常优秀，非常建议大家仔细阅读，学习优秀的源码对自己的编码能力会有一定程度的提升！

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