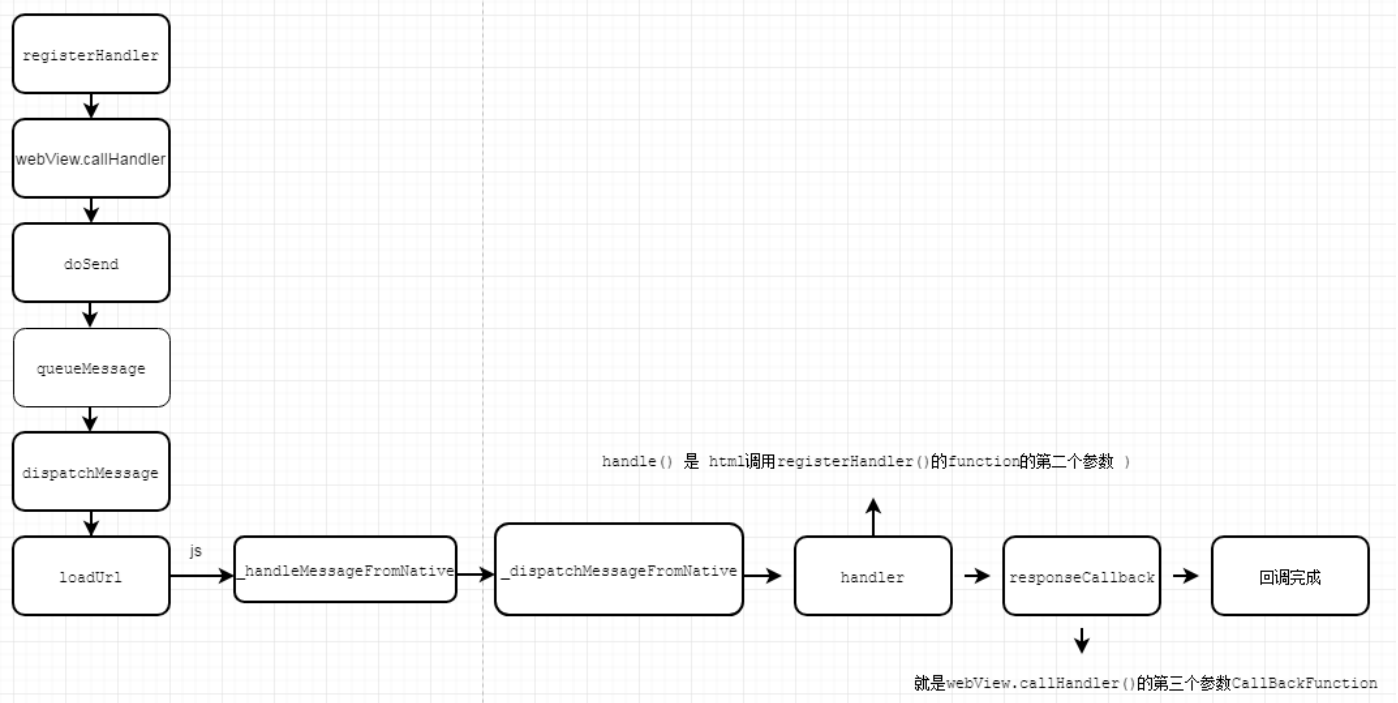
Android 调用 JS 的方法：



实现原理：Android使用 loadUrl方法，调用js的方法，js通过注册的Handler给Android回调。

1.html注册给Android用的方法：

```
//JS注册方法示例，方法名functionInJs【JS代码】
WebViewJavascriptBridge.registerHandler("functionInJs", function(data, responseCallback) {
        document.getElementById("show").innerHTML = ("data from Java: = " + data);
        var responseData = "Javascript Write back something!";
        responseCallback(responseData);
});
```

1的handlerName是调用 WebViewJavascriptBridge.js的registerHandler()注册：

```
//存储注册的Handler（assigned handlerName）
    function registerHandler(handlerName, handler) {
        messageHandlers[handlerName] = handler;
    }
```

注册后保存到 js的变量：

```
var messageHandlers = {};
```

```
//Java调用注册的方法functionInJs【Java代码】
webView.callHandler("functionInJs", new Gson().toJson(user), new CallBackFunction() {
        @Override
        public void onCallBack(String data) {

        }
});
```

Android调用callHandler()；经过doSend()后，将callBack 保存到BridgeWebView 本地变量 responseCallbacks 中：

```
Map<String, CallBackFunction> responseCallbacks = new HashMap<String, CallBackFunction>();
```

```
private void doSend(String handlerName, String data, CallBackFunction responseCallback) {
    Message m = new Message();
    // Message存data
    if (!TextUtils.isEmpty(data)) {
        m.setData(data);
```

```
    }
    // Message存callback
    if (responseCallback != null) {
        String callbackStr = String.format(BridgeUtil.CALLBACK_ID_FORMAT, ++uniqueId + (BridgeUtil.UNDERLINE_STR + SystemClo
        // 将callBack保存到本地变量 responseCallbacks 中
        responseCallbacks.put(callbackStr, responseCallback);
        m.setCallbackId(callbackStr);
    }
    // Message存handlerName
    if (!TextUtils.isEmpty(handlerName)) {
        m.setHandlerName(handlerName);
    }
    // 查询消息
    queueMessage(m);
}
```

```
private void queueMessage(Message m) {
    // 一般为空，网页加载完成后会手动清空
    // 主要是用来在JsBridge的js库注入之前，保存Java调用JS的消息，避免消息的丢失或失效。
    // 待页面加载完成后，后续CallHandler的调用，可直接使用loadUrl方法而不需入队。
    // 究其根本，是因为Js代码库必须在onPageFinished（页面加载完成）中才能注入导致的。
    if (startupMessageList != null) {
        startupMessageList.add(m);//之前有消息则在尾部添加
    } else { // 网页加载完成后都是走这直接分发消息
        dispatchMessage(m);
    }
}
```

dispatchMessage()，分发消息的时候调用 webView.load(url);

使用url："javascript:WebViewJavascriptBridge._handleMessageFromNative('%s');"，百分号里面是messageJson。

eg：javascript:WebViewJavascriptBridge._handleMessageFromNative('{\"callbackId\":\"JAVA_CB_2_368\",\"data\":\"data from android webView.callHandler()\",\"handlerName\":\"functionInJs\"}');

```
void dispatchMessage(Message m) {
    String messageJson = m.toJson();
    Log.e("chris", "messageJson==" + messageJson);
    //为json字符串转义特殊字符，格式处理
    messageJson = messageJson.replaceAll("(\\\\)([^utrn])", "\\\\\\\\$1$2");
    messageJson = messageJson.replaceAll("(?<=[^\\\\])(\")", "\\\\\"");
    messageJson = messageJson.replaceAll("(?<=[^\\\\])(\')", "\\\\\'");
    messageJson = messageJson.replaceAll("%7B", URLEncoder.encode("%7B"));
    messageJson = messageJson.replaceAll("%7D", URLEncoder.encode("%7D"));
    messageJson = messageJson.replaceAll("%22", URLEncoder.encode("%22"));
    // javascriptCommand==javascript:WebViewJavascriptBridge._handleMessageFromNative('{\"callbackId\":\"JAVA_CB_2_368\",\"da
    String javascriptCommand = String.format(BridgeUtil.JS_HANDLE_MESSAGE_FROM_JAVA, messageJson);
    // 必须要找主线程才会将数据传递出去
    // webView在主线程中加载该 url schema
    if (Thread.currentThread() == Looper.getMainLooper().getThread()) {
        // 到这里安卓部分已经完成，拦截url 等待js回调，剩下的交给JsBridge的_handleMessageFromNative处理
        this.loadUrl(javascriptCommand);
    }
```

```
}
```

从url 可以看出，webView调用的是 JsBridge 的 _handleMessageFromNative():

```
function _handleMessageFromNative(messageJSON) {
    console.log(messageJSON);
    if (receiveMessageQueue) {
        receiveMessageQueue.push(messageJSON);
    }
    _dispatchMessageFromNative(messageJSON);
}
```

JS的 _dispatchMessageFromNative(messageJSON)会从 var messageHandlers = {} 取出指定的 handler，走指定的 handle();

```
handler(message.data, responseCallback);
```

handle()是 html的registerHandler()的第二个参数，也就是 name为 functionInJs的这个handler：

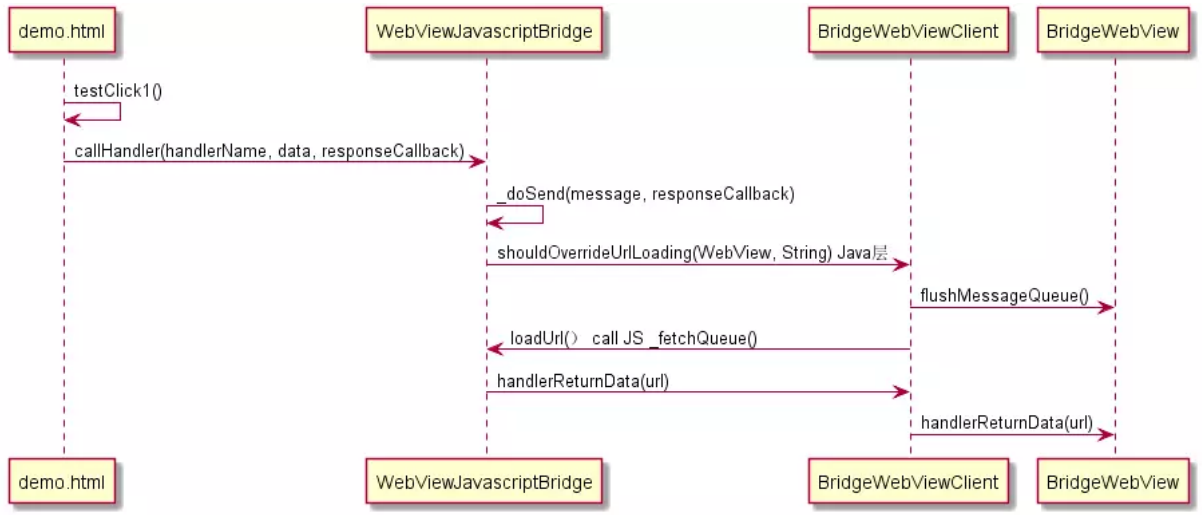
```
//注册方法供java调用
          bridge.registerHandler("functionInJs", function(data, responseCallback) {
              document.getElementById("show").innerHTML = ("data from Java: = " + data);
              var responseData = "Javascript Says Right back aka!";
              responseCallback(responseData);
          });
```

Android收到对应的回调：

```
//Java调用注册的方法functionInJs【Java代码】
webView.callHandler("functionInJs", new Gson().toJson(user), new CallBackFunction() {
       @Override
       public void onCallBack(String data) {
            // 这里处理 data
       }
});
```

其中responseCallback 对应 onCallback。

JS 调用 Android 的方法：



实现原理：

利用js的iFrame（不显示）的src（url）动态变化，触发java层WebViewClient的shouldOverrideUrlLoading方法，然后让本地去调用js。js代码执行完成后，最终调用_doSend方法处理回调。

1.没有传 handlerName 的方式，使用 defaultHandler 处理，不用注册 Handler。

demo.html页面中点击"发消息给Native"按钮，触发WebViewJavascriptBridge.js中send方法的调用:（第二个参数是responseCallback）

```
function testClick() {

    //send message to native, 通过 send(), 只是少了一个参数 handlerName
    var data = {id: 1, content: "这是一个图片 <img src=\"a.png\"/> test\r\nhahaha"};
    window.WebViewJavascriptBridge.send(
        data
        , function(responseData) {
            document.getElementById("show").innerHTML = "repsonseData from android, data = " + responseData
        }
    );

}
```

然后调用 _doSend()，更换iFrame的src，触发BridgeWebViewClient的shouldOverrideUrlLoading方法。

```
//sendMessage add message, 触发native处理 sendMessage
function _doSend(message, responseCallback) {
    if (responseCallback) {
        var callbackId = 'cb_' + (uniqueId++) + '_' + new Date().getTime();
        responseCallbacks[callbackId] = responseCallback;
        message.callbackId = callbackId;
    }

    // 添加消息到 sendMessageQueue
    sendMessageQueue.push(message);//保存 js 发送的消息，这里并没有去真的发消息
    messagingIframe.src = CUSTOM_PROTOCOL_SCHEME + '://' + QUEUE_HAS_MESSAGE;
}
```

shouldOverrideUrlLoading方法根据url的前缀，进入了BridgeWebView的flushMessageQueue方法。

```
//BridgeWebViewClient.java
@Override
public boolean shouldOverrideUrlLoading(WebView view, String url) {
    try {
        url = URLDecoder.decode(url, "UTF-8");
    } catch (UnsupportedEncodingException e) {
        e.printStackTrace();
    }

    if (url.startsWith(BridgeUtil.YY_RETURN_DATA)) { // 如果是返回数据
         //url以yy://return/开头
        webView.handlerReturnData(url);
        return true;
    } else if (url.startsWith(BridgeUtil.YY_OVERRIDE_SCHEMA)) { //
         //url以yy://开头
        webView.flushMessageQueue();
        return true;
    } else {
```

```
        return super.shouldOverrideUrlLoading(view, url);
    }
}
```

Android的flushMessageQueue()方法，通过loadUrl调用到WebViewJavascriptBridge.js中的_fetchQueue()方法

```
void flushMessageQueue() {
    MyCallBackFunction myCallBackFunction = new MyCallBackFunction();
    // jsBridge抓取消息 _fetchQueue 即返回messageQueueString的数据 ，抓取之后回调数据到 myCallBackFunction
    if (Thread.currentThread() == Looper.getMainLooper().getThread()) {
        // JS_FETCH_QUEUE_FROM_JAVA = "javascript:WebViewJavascriptBridge._fetchQueue();";
        loadUrl(BridgeUtil.JS_FETCH_QUEUE_FROM_JAVA, myCallBackFunction);
    }
}
```

_fetchQueue方法将sendMessageQueue数组中的所有消息，序列化为json字符串，通过更改iFrame的src，触发shouldOverrideUrlLoading方法

```
// 提供给native调用,该函数作用:获取sendMessageQueue返回给native,由于android不能直接获取返回的内容,所以使用url shouldOverridel
function _fetchQueue() {
    var messageQueueString = JSON.stringify(sendMessageQueue);// 读取 js发送的消息
    sendMessageQueue = [];//置空
    //android can't read directly the return data, so we can reload iframe src to communicate with java
    if (messageQueueString !== '[]') {
        /*触发安卓的 shouldOverrideUrlLoading，真正把消息传给 Android*/
        bizMessagingIframe.src = CUSTOM_PROTOCOL_SCHEME + '://return/_fetchQueue/' + encodeURIComponent(messageQueueString);
    }
}
```

shouldOverrideUrlLoading方法根据url的前缀，进入了BridgeWebView的handlerReturnData方法。

```
void handlerReturnData(String url) {
    String functionName = BridgeUtil.getFunctionFromReturnUrl(url);
    CallBackFunction callBackFunction = responseCallbacks.get(functionName);
    String data = BridgeUtil.getDataFromReturnUrl(url);
    if (callBackFunction != null) {
        callBackFunction.onCallBack(data);
        responseCallbacks.remove(functionName); // 移除已经处理过的回调
        return;
    }
}
```

返回结果回调到 onCallBack()

2.传 handlerName 的方式：注册可供js调用的handler。最终handler在java端存放在webview的messageHandlers变量中

```
//MainActivity.java
webView.registerHandler("submitFromWeb", new BridgeHandler() {

    @Override
    public void handler(String data, CallBackFunction function) {
        Log.i(TAG, "handler = submitFromWeb, data from web = " + data);
                function.onCallBack("submitFromWeb exe, response data 中文 from Java");
    }

});

//BridgeWebView.java
public void registerHandler(String handlerName, BridgeHandler handler) {
```

```
    if (handler != null) {
        messageHandlers.put(handlerName, handler);
    }
}
```

html中调用Native端提供的方法

```
//demo.html
function testClick1() {
    var str1 = document.getElementById("text1").value;
    var str2 = document.getElementById("text2").value;

    //call native method
    window.WebViewJavascriptBridge.callHandler(
        'submitFromWeb'
        , {'param': '中文测试'}
        , function(responseData) { // 这个参数是responseCallback
                document.getElementById("show").innerHTML = "js send get responseData from java, data = " + responseData
        }
    );
}
```

callHandler最终调用_doSend方法

```
//WebViewJavascriptBridge.js
function callHandler(handlerName, data, responseCallback) {
    _doSend({
        handlerName: handlerName,
        data: data
    }, responseCallback);
}
```

后面是和1一样的流程。