H5和NATIVE交互方案

基本原理

Hybrid 的技术本质是在 WebView 的基础上，与原生客户端建立 JS Bridge 桥接，以达到 JS 调用 Native API 和 Native 执行 JS 方法的目的。

native -> h5

1. loadUrl

'javascript:'+JS代码做跳转地址

比如，当前WebView加载的页面已经定义了一段js方法。

```
<script>
   function callJS(){
      alert("Android调用了JS的callJS方法");
   }
</script>
```

然后native调用这段js，直接调用webview的loadurl方法即可。

```
 mWebView.loadUrl("javascript:callJS()");
```

注意，要调用的js方法可以在网页中要事先定义好，也可以直接调用具体的方法

```
 mWebView.loadUrl("javascript:alert("Android调用了JS的callJS方法");");
```

这两种方式最后的结果都是一样的。

2. evaluatingJavaScript

直接注入执行JS代码，Android 4.4 后才可使用。

比如，当前WebView加载的页面已经定义了一段js方法。这段方法返回了一个字符串。

```
<script>
   function callJS(){
      return "Android调用了JS的callJS方法";
   }
</script>
```

然后native调用这段js。

```
mWebView.evaluateJavascript("javascript:callJS()", new ValueCallback<String>() {
 @Override
 public void onReceiveValue(String value) {
     //这里回调的value是之前js代码里返回的"Android调用了JS的callJS方法"字符串
     }
 });
```

这个方法能够直接在一次执行的时候获取到 JS 返回的结果。如果是使用 loadUrl() 的方式的话，执行完后对客户端来说这句话就结束了，如果想要拿到返回的结果的话另外需要 JS 调用客户端的方法返回。

h5 -> native

1. JS上下文注入

- addJavascriptInterface进行对象映射

直接将一个native对象（or函数）注入到JS里面，可以由web的js代码直接调用，直接操作，可以在loadUrl之前提前准备一个对象，通过这个接口注入给JS上下文，从而让JS能够操作

```
//定义好 Java 接口对象
public class Bridge {
    @JavascriptInterface
    public void calllNative(String msg) {
        ......
    }
}
//注入给JS上下文
mWebview.addJavascriptInterface(new Bridge(), "bridge");
//加载页面
mWebview.loadUrl("www.test.com")
```

在页面加载之后,H5页面触发到

```
window.bridge.calllNative("test");
```

这样就实现了h5到native的通信。这种方式在4.2之前有安全问题，这一点在下文再描述。

2. 跳转请求拦截

- shouldOverrideUrlLoading

根据url来拦截，解析这个url,来执行相应的native操作。

使用通信拦截请求这种方式时使用比较广泛的， `WebViewJavascriptBridge` 和 `cordova` 都是使用的这个方案。

```
mWebview.setWebViewClient(new WebViewClient() {
  @Override
  public boolean shouldOverrideUrlLoading(WebView view, String url) {
    if(/*匹配url的规则*/)) {
      //进行native的操作
      return true;
    }
    return super.shouldOverrideUrlLoading(view, url);
  }
});
```

3. 弹窗拦截

同 URL 拦截类似，弹窗拦截主要是利用 JS 的一些方法执行时会触发 Android 客户端中的一些回调，通过对前端参数进行识别来执行对应的客户端代码。 前端可以发起很多种弹窗包含

- alert() 弹出个提示框，只能点确认
- confirm() 弹出个确认框（确认，取消）
- prompt() 弹出个输入框，让用户输入东西
- console.log

每种弹框都可以由JS发出一串字符串，用于展示在弹框之上，而此字符串恰巧就是可以用来传递数据，我们把所有要传递通讯的信息，都封装进入一个js对象，然后生成字典，最后序列化成json转成字符串。

通过任意一种弹框将字符串传递出来，交给客户端就可以进行拦截，从而实现通信。

在WebView的回调中对应

- onJsPrompt
- onJsConfirm
- onJsAlert
- onConsoleMessage

以 `onJsPrompt` 为例。

native首先设置 `onJsPrompt` 回调。

```
webview.setWebChromeClient(new WebChromeClient(){
    @Override
    public boolean onJsPrompt(WebView view, String message, String defaultValue, J
SPromptResult result) {
      if(/*匹配message的规则*/)) {
      //进行native的操作
      return true;
      }

      return super.onJsPrompt(view, url, message, defaultValue, result);
  }
});
```

在H5页面，把需要传递的信息传入alert()方法即可。

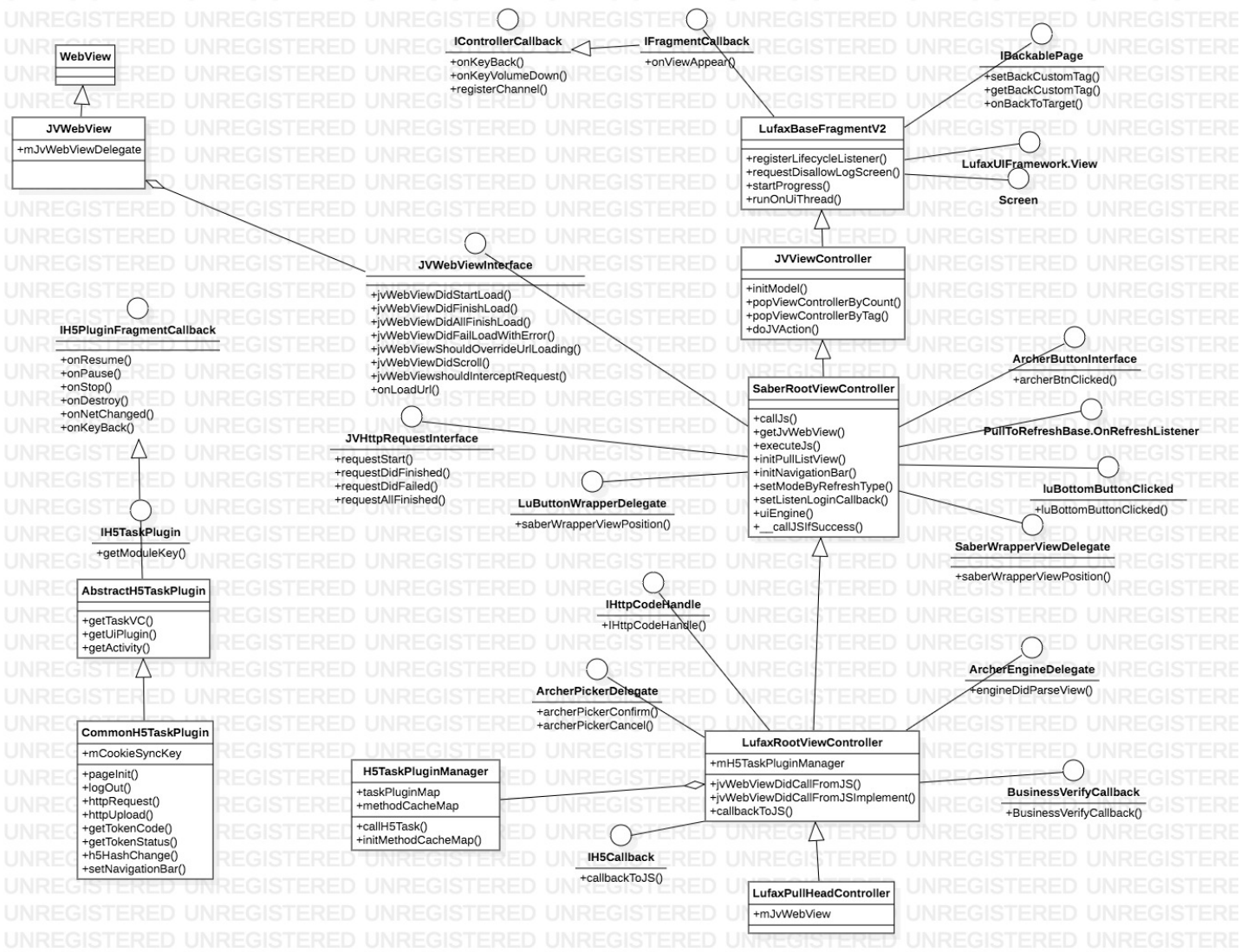
```
   alert("要传递的信息");
```

陆金所task方案

- native -> H5 使用了拼接js，通过 `loadurl` 注入到JVWebview中
- H5 -> native 使用了addJavascriptInterface进行对象映射

页面中，交互的发起者都是H5页面, 在H5 调用native的时候需要传入一个回调方法名，即callbackId，然后客户端直接执行回调方法。这样就完成了一个完整的信息交流的过程。

Android实现



H5->native

- H5网页中直接调用"Bridge.call(options)"

在 `JVWebView` 中的 `setJVWebViewDelegate` 方法将一个native对象注入到JS里面。

```
    /**
     * 设置delegate
     */
    public void setJVWebViewDelegate(JVWebViewInterface delegate) {
        mJvWebViewDelegate = delegate;
        addJavascriptInterface(delegate, "AndroidBridge");
    }
```

当H5页面需要调用native方法时，调用 `Bridge.call(options)` 方法

```
class Bridge {
    /**
     * 调用native
     * @param       {Object} params task参数
     */
    static call(params = {}) {
        console.log(params);
        const sParams = JSON.stringify(params);
        // android与ios实现方式不一致，android采用在webview中注入的方式，native采用iframe
截获的方式
        if (window.AndroidBridge) {
            window.AndroidBridge.jvWebViewDidCallFromJS(sParams);
        } else if (Platform && Platform.ios) {
            if (params.task === 'refresh_webview') {
                window.location.href = 'ios://refresh_webview';
            } else {
                IOSBridge.call(sParams);
            }
        } else {
            console.error('Bridge: Can not support other Platform', params);
        }
    }
}
```

这样就会回调到native的 `jvWebViewDidCallFromJS` 方法。

`LufaxRootViewController` 中的 `jvWebViewDidCallFromJS` 分发h5页面的字符串，根据不同的task名称，最终使用到H5TaskPluginManager里面的方法，调用的方法都实现了H5Task标签。（H5TaskPluginManager会将实现了H5Task的方法都注册到methodCacheMap中）

- H5网页中跳转

JVWebView的shouldOverrideUrlLoading方法中拦截请求。

处理intent://形式的url，在这里进行Schema跳转。这里不做task的处理，不详细解释。

native->H5

`LufaxRootViewController` 重写了 `callbackToJS` 方法，在 `callbackToJS中` 方法里调用了 `callJs` 方法，

`/lu/android/component/src/service/lufax/controller/LufaxRootViewController.java`

```
 callJs("window.Bridge.appCallback('" + Base64.encodeToString(taskJson.toString().
getBytes(), Base64.NO_WRAP) + "')");
```

`callJs` 最终调用到上文所述的 `loadurl` 方法,往H5页面注入这段js,统一使用js框架的全局方法 `window.Bridge.appCallback(options)` 。

```
    public void executeJS(final String js) {
        JVUtility.DLOG_DEBUG(js);

        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                // V342暂时根据url是否为空来确定webview中是否有html，解决没有光标的问题。TO
DO：改造成根据loadstatus状态
                if (getUrl() != null && loadStatus != STATUS_DEAD) {
                    loadUrl("javascript:" + js);
                }
            }
        });
    }
```

JS实现

hybrid端提供了全局回调函数window.Bridge.appCallback，native调用hybrid可全部通过该函数来处理。

对于hybrid调用native功能，如调用后端请求，双方只需要约定taskname，hybrid调用时在前端管理好业务回调，调用Bridge.call({task,sessionId,version,options})交给native，native在完成功能后通过window.Bridge.appCallback({task,sessionId,callbackId,version,data})将数据返回给hybrid。

对于native通知hybrid，如pulldown，双方也只需约定taskname，native调用时直接通过window.Bridge.appCallback({task,data})通知hybrid。

回调管理经过改造后 1.可将native与hybrid功能更好的分离，减少双方耦合；2.可在全局唯一的地方针对数据加密和预处理；3.hybrid可更好的扩展，如果需要针对某个task做拦截或增加更多处理，只需要在Bridge.appCallback中针对某个task做监听处理。

Native调用Hybrid：

```
window.Bridge.appCallback(options)
 {
  task,//native与hybrid约定的task名称，由hybrid带入
  [sessionId,]//sessionId由hybrid生成，代表当前task的一次调用，由hybrid带入；native主动调用hybrid时，无须传该参数
  [...params,]//task请求参数，由hybrid带入
  [callbackId,]//callbackId由hybrid生成，代表当前task中的某一callback，native处理完成后加入带回；native主动调用hybrid时，无须传该参数
  result:{//由native加入到options中，返回给业务
    ...data//native处理完成后回传给hybrid的数据，逐一展开
  }
 }
```

Hybrid调用Native：

```
Bridge.call(options)
  {
    task,//native与hybrid约定的task名称
    sessionId,//sessionId由hybrid生成，native处理完成后带回；
    version='2.0',//version为bridge版本，用来控制某个task版本，提供给native做兼容处理；新版该参数为'2.0'，老版无该参数
    ...params//params与之前一致，task所需要的参数，逐一展开
    {
      param1:'',//param和callback名称由hybrid和native约定
      param2:'',
      param3:'',
      callbackId1:'callbackId1',
      callbackId2:'callbackId2'
    }
  }
```

Bridge.call的代码如下

```
class Bridge {
    /**
     * 调用native
     * @param       {Object} params task参数
     */
    static call(params = {}) {
        console.log(params);
        const sParams = JSON.stringify(params);
        // android与ios实现方式不一致，android采用在webview中注入的方式，native采用iframe
截获的方式
        if (window.AndroidBridge) {
            window.AndroidBridge.jvWebViewDidCallFromJS(sParams);
        } else if (Platform && Platform.ios) {
            if (params.task === 'refresh_webview') {
                window.location.href = 'ios://refresh_webview';
            } else {
                IOSBridge.call(sParams);
            }
        } else {
            console.error('Bridge: Can not support other Platform', params);
        }
    }
}
```

具体操作可以在《JSBridge回调原理》中看到setTitle(对应navigation_bar的task)的调用操作。

task列表

H5 Task 列表

重点task介绍

SPA-HYBRID-TASK

安全性

WebViewbe本身就是支持js调用Native代码的，不过WebView的这个功能在Android 4.2（API 17）以下存在高危的漏洞。这个漏洞的原理就是Android系统通过 WebView.addJavascriptInterface(Object o, String interface) 方法注册可供js调用的Java对象，但是系统并没有对注册的Java对象方法调用做限制。导致攻击者可以利用反射调用未注册的其他任何Java对象，攻击者可以根据客户端的能力做任何事情。

出于安全考虑，Android 4.2以后的系统规定允许被js调用的Java方法必须以 `@JavascriptInterface` 进行注解。

目前，陆金所APP最低支持4.0系统的手机，目前没看到针对这部分手机的处理。

android代码里，native->h5进行了加密，h5->native没有看到加解密。

task使用中常见问题

一个完整流程——以返回键实现为例

以单品详情页为例，在 `全部理财` 页面点击 `活期` 中的 `陆金宝T+1` ，此时从native页面跳转到H5页面，创建一个单品详情页，即创建 `LufaxBaseFragmentV2` 的派生类 `LufaxRootViewController` ，在 `onCreate` 回调中首先注册plugin。

```
public void onCreate(Bundle savedInstanceState) {
        //省略...
        mCommonH5TaskPlugin = new CommonH5TaskPlugin(this);
        mH5TaskPluginManager.registerPlugin(mCommonH5TaskPlugin);
        mH5TaskPluginManager.registerPlugin(new UtilH5TaskPlugin(this));
        //省略...
}
```

这里会将 `CommonH5TaskPlugin` 中实现了 `H5Task` 标签的方法都加入到 `H5TaskPluginManager` 类中的 `methodCacheMap` 中。

然后开始用 `JVWebView` 去展示这个页面。页面加载后会从h5回调native的 `track` , `navigation_bar` , `http_request` 这三个task。

其中 `navigation_bar` 这个task实现了控制APP容器顶部的Bar样式的功能。 js代码里的 `setTitle` 方法中最终调用 `navigation_bar` 的task。

具体js代码如下

业务代码

```
componentWillMount() {
  this.setTitle({
    naviBar: {
      color: '526bc2',
      title: '会员成长中心',
      titleColor: 'ffffff',
      hideShadow: '1',
    },
    leftView: {
      color: 'ffffff',
    },
    rightView: {
      title: '等级奖励',
      color: 'ffffff',
      callback: () => {
        this.openPage('https://www.playlu.com/tree/rights', 'out', 'game');
      },
    },
  });
```

LuPage

```
/**
 * 设置title, 参数参考LuHeader
 * @description options.leftView.callbackId会被LuPage设置为LuPage.onBack
 * @param       {Object} options 参见LuHeader
 */
setTitle(options) {
  this.hideError();
  const mOpts = options;
  if (mOpts.leftView) { // true/Object
    if (!_.isPlainObject(mOpts.leftView)) { // leftView: true
      mOpts.leftView = {};
    }
    mOpts.leftView.callback = this.onBack;
  }
  this.header.setTitle(mOpts);
}
```

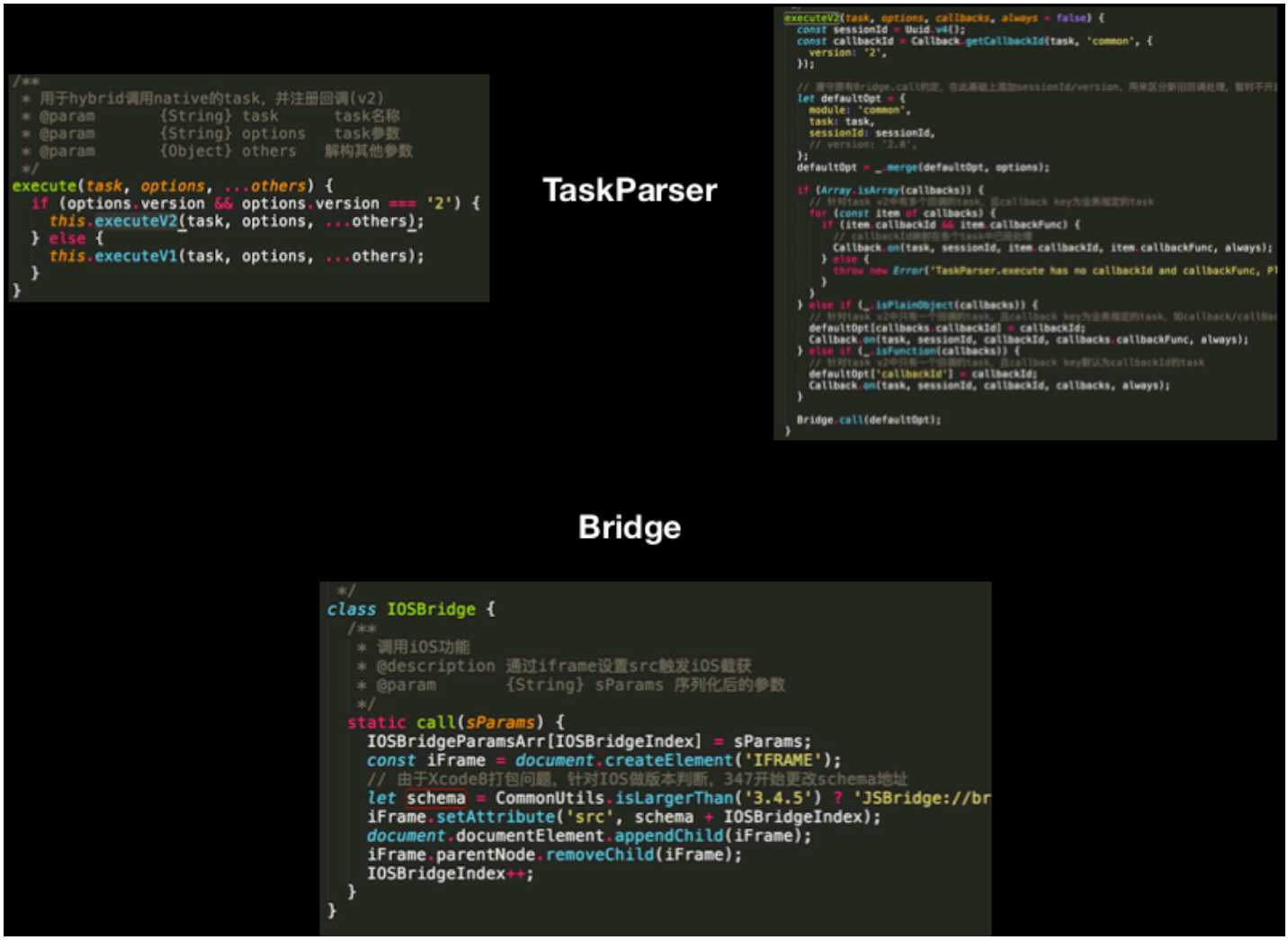
LuHeader

```
 */
setTitle(options) {
  const tOpts = options;
  if (typeof tOpts.naviBar === 'undefined') {
    throw new Error('LuHeader.setTitle: options.naviBar should not be undefined, Ple
  }
  if (Platform.isLU) {
    // leftView
    if (typeof tOpts.leftView !== 'undefined') { // true/false/Object
      if (_.isPlainObject(tOpts.leftView)) { // Object
        tOpts.leftView.isHide = '0';
      } else { // leftView: true/false
        const isShow = tOpts.leftView;
        tOpts.leftView = {
          isHide: isShow ? '0' : '1',
        };
      }
    }
    // rightView
    if (tOpts.rightView) tOpts.rightView.isHide = tOpts.rightView.isHide === '1' ? '
    // rightMenu, 仅用于设置右侧三点按钮
    if (_.isObject(tOpts.rightMenu)) {
      if (tOpts.rightMenu.isHide !== '1' && (!_.isArray(tOpts.rightMenu.items) || tO
        throw new Error('LuHeader.setTitle: options.rightMenu.items should be An Arr
      }
      tOpts.rightMenu.isHide = tOpts.rightMenu.isHide === '1' ? '1' : '0';
    }
    PageParser.setHeader(tOpts);
```

PageParser

```
 */
setHeader(options) {
  const defaultOpt = {
    version: '2',
    naviBar: {
      isHide: '0',
      color: 'ffffff',
      title: '',
      titleColor: '13334d',
    },
    leftView: {
      isHide: '1',
      color: '526bc2',
    },
    rightView: {
      isHide: '1',
      color: '526bc2',
      title: '',
    },
  };
  const callbacks = [];
  // 处理通过到Native上一览
  if (options.leftView) {
    let leftViewCallbackId = Uuid.v4(); // V353之后建议使用 this.getCallbackId
    if (_.isFunction(options.leftView.callback)) {
      callbacks.push({
        callbackId: leftViewCallbackId,
        callbackFunc: options.leftView.callback,
      });
    } else {
      // for back to native
      leftViewCallbackId = '';
    }
    // 删除callback
    delete options.leftView.callback;
    // callback转换成callbackId传给native
    options.leftView.callbackId = leftViewCallbackId;
  }
```

```
  if (options.rightView) {
    if (_.isFunction(options.rightView.callback)) {
      const rightViewCallbackId = Uuid.v4(); // V353之后建议
      callbacks.push({
        callbackId: rightViewCallbackId,
        callbackFunc: options.rightView.callback,
      });
      delete options.rightView.callback;
      options.rightView.callbackId = rightViewCallbackId;
    }
  }

  if (options.rightMenu) {
    options.rightMenu.items && options.rightMenu.items.for
      if (item.itemType !== 'cs' && _.isFunction(item.call
        const itemCallbackId = this.getCallbackId('navigat
        callbacks.push({
          callbackId: itemCallbackId,
          callbackFunc: item.callback,
        });
        delete item.callback;
        item.callbackId = itemCallbackId;
      }
    });
  }

  options = _.merge(defaultOpt, options);
  this.execute('navigation_bar', options, callbacks, true)
}
```



其中 `Callback.on` 方法会把刚才的callbackId和callbackFunc存进 `taskCallbackMap` 中。

```
static on(task, sessionId, callbackId, callback, always = false) {
    if (!taskCallbackMap[task]) taskCallbackMap[task] = {};
    if (sessionId && callbackId) {
      // 针对主动调用
      if (!taskCallbackMap[task][sessionId]) taskCallbackMap[task][sessionId] = {}
;
      taskCallbackMap[task][sessionId][callbackId] = {
        callback: callback,
        always: always,
      };
    } else {
      // 针对被动触发
      taskCallbackMap[task] = {
        callback: callback,
        always: always,
      };
    }
  }
```

最后从H5传递到native的完整的json如下。

```
{
    "module":"common",
    "task":"navigation_bar",
    "sessionId":"7b10a4e7-2d3e-49a9-bd01-ecc13b2084ba",
    "version":"2",
    "naviBar": {
        "title":"项目详情"
    },
    "leftView": {
        "isHide":"0",
        "callbackId":"75df13f9-274a-46a4-891e-242fc2452fb5"
    },
    "rightView": {
        "title":"分享",
        "isHide":"0",
        "callbackId":"bd4b869a-612f-49c4-9685-dea1964f84df"
    },
    "rightMenu": {
        "items": [
             {
                "itemType":"cs",
                "scene":"陆金宝T+1"
            }
        ],
        "isHide":"0"
    }
}
```

其中 `leftView` 部分控制左边按钮，目前只有一个返回箭头样式。其中的 `callbackId` 代表左边按钮的回调。

```
    private void navigationBarLeftView(JSONObject taskJson) {
        JSONObject leftViewJsObj = taskJson.optJSONObject("leftView");
        //省略
        final String callbackId = leftViewJsObj.optString("callbackId");
        if (StringUtil.isNotEmpty(callbackId)) {
           H5CallbackModel h5CallbackModel = new H5CallbackModel(callbackId, "", t
askJson);
           getTaskVC().setBackCallbackV2(h5CallbackModel);
        } else {
           getTaskVC().setBackCallbackV2(null);
         }
         //省略
          }
        }
    }
```

其中，会把task中的 `callbackId` 设置到 `LufaxRootViewController` 中。

```
  public void setBackCallbackV2(H5CallbackModel h5CallbackModel) {
        mBackCallbackV2Model = h5CallbackModel;
   }
```

页面展示出来之后，点击左上角的返回键。会调用到 `onKeyBack` 方法

```
    @Override
    public boolean onKeyBack(boolean fromBar) {
        //省略
        // 1. 优先处理H5中的返回模型
        if (mBackCallbackV2Model != null && StringUtil.isNotEmpty(mBackCallbackV2M
odel.callbackId)) {
            callbackToJS(mBackCallbackV2Model);
            return true;
        }
       //省略
    }
```

此时，会判断 `mBackCallbackV2Model` 这个变量已经其中的 `callbackId` 成员是否为空。而这个变量就是之前在调用 `navigation_bar` 这个task时进行设置的。

此时会重新将之前的callbackId拼接成js，重新注入到 `JVWebView` 中。将options解码之后，根据callbackId等信息从 `taskCallbackMap` 中取出callback。

```
window.Bridge.appCallback = function appCallback(options) {
  const rOptions = JSON.parse(Base64.decode(options));
  console.log(rOptions);
  const task = rOptions.task;
  const sessionId = rOptions.sessionId;
  const callbackId = rOptions.callbackId;
  const data = rOptions.result; // native返回传结果
  const params = rOptions.params; // native主动调动传参数
  let callback = null;
  let always = null;

  if (sessionId && callbackId) { // hybrid调用native，并注册回调
    callback = taskCallbackMap[task][sessionId][callbackId]['callback'];
    always = taskCallbackMap[task][sessionId][callbackId]['always'];
    //省略
    }
```

然后H5页面又以 `pop_view` 的task的形式回调到native。代码如下。

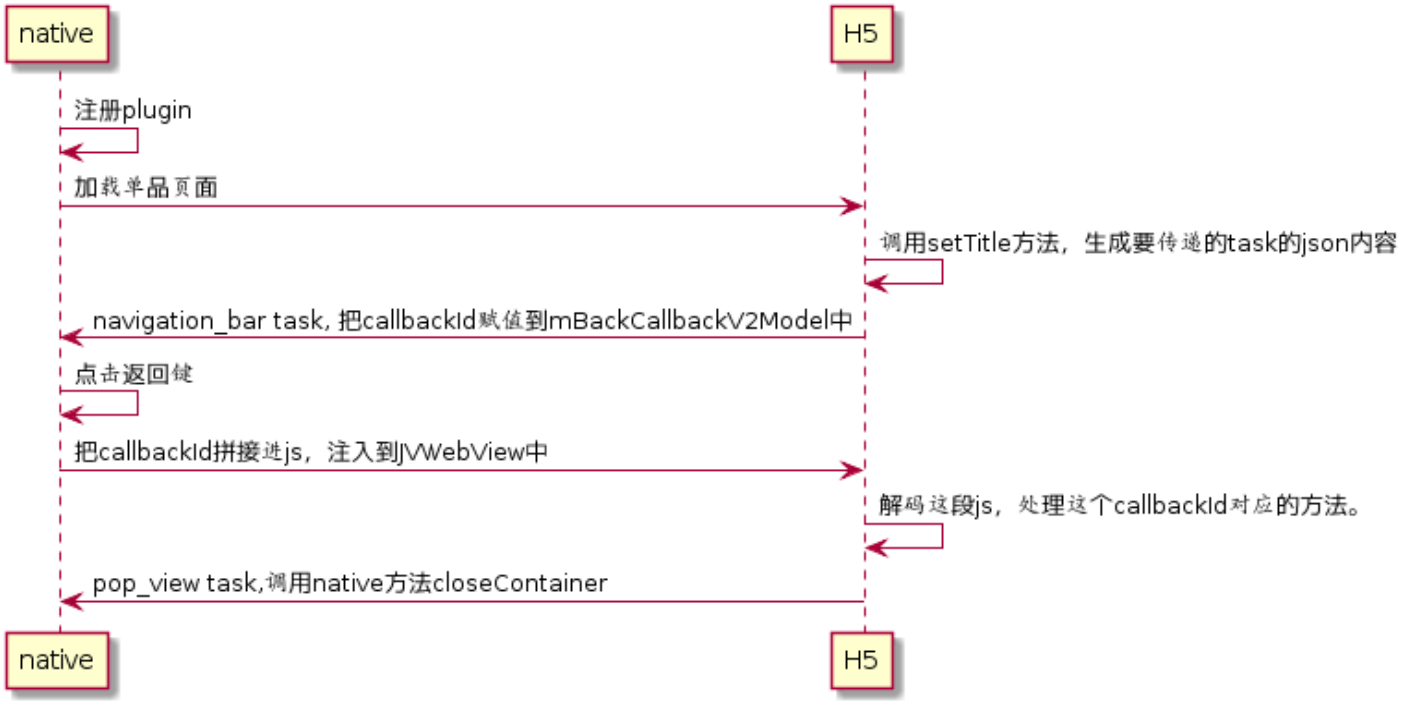
```
    @H5Task(value = "pop_view", version = "2")
    public void popView(final JSONObject taskJson) {
        //省略
        if (StringUtil.isEmpty(backTag) && !"1".equals(needRefresh)) {
            closeContainer(getTaskVC());
            return;
        }
        //省略
    }

    //关闭容器统一用一个逻辑
    public static void closeContainer(LufaxRootViewController taskVC) {
        // clear backcallback
        taskVC.setBackCallback("");
        taskVC.setBackCallbackV2(null);

        if (taskVC.getActivity() != null) {
            LufaxUtils.invokeKeyBack(taskVC.getActivity(), true);
        }
    }
```

在 `popView` 方法中会调用到 `closeContainer` ,最终实现了在native端返回键的功能。

综上所述，native开始展示一个h5页面，会首先在h5页面的容器中注册h5 task方法，然后h5以task方式调用navigation*bar,把callbackId赋值到LufaxRootViewController中的mBackCallbackV2Model中去。用户点击返回键之后，将之前赋值的callbackId拼接进js，然后注入到H5页面，H5页面再以task方式调用pop*view,调用native方法closeContainer。最终实现了返回键的操作。

native为H5提供各种操作的能力，具体的操作还是从H5页面发动，最终达到页面动态控制的效果

全部流程如下图所示。



坑

目前的框架下，appCallBack和call方法，入口都是统一的，但是参数是可变的，根据不同的业务会有不同的字段，这个内容增加了整个方案的复杂度。

- H5 Task common模块功能列表记录的接口和字段与安卓现存代码里面的不一致。
- 只有common模块的h5 task文档，缺少UtilH5TaskPlugin等其他类的文档，只有

几种容器task简单介绍

- Android端和Ios端都是使用同一份H5页面，对应的字段不统一。

参考

hybrid架构改造一期-Bridge回调管理改造

Android schema跳转规范

谈谈Android App混合开发

浅谈 2018 移动端跨平台开发方案

聊聊移动端跨平台开发的各种技术

[从零收拾一个hybrid框架（一）-- 从选择JS通信方案开始]

[WebView 远程代码执行漏洞浅析]