小程序框架项目说明文档

刘家萱

一、模块加载部分

先建立 a.js、b.js、c.js、d.js 等需加载的模块：

a.js

```
module-loading > JS a.js > ...
1   module.exports = "hello world";
```

b.js

```
module-loading > JS b.js > add
1   function add(a, b) {
2     return a + b;
3   }
4
5   module.exports = add;
```

c.js

```
module-loading > JS c.js > ...
1   let c = 1;
2
3   c = c + 1;
4
5   module.exports = c;
6
7   c = 6;
```

d.js

```
module-loading > JS d.js > ...
1   let d = {
2     num: 1
3   };
4
5   d.num++;
6
7   module.exports = d;
8
9   d.num = 6;
```

然后在 index.js 中用 require 函数使用写好的模块：

```
JS index.js
module-loading > JS index.js > ...
1   const a = require('./a.js');
2   const add = require('./b.js');
3   const c = require('./c.js');
4   const d = require('./d.js');
5   const e = require('./e.json');
6
7   console.log(a);
8   console.log(add(1, 2));
9   console.log(c);
10  console.log(d);
11
12  d.num = 7;
13  console.log(d);
14
15  console.log(e);
```

在终端中输入 node index.js 可查看到对应输出，模块加载成功：

```
问题  输出  调试控制台  终端  JUPYTER

Windows PowerShell
版权所有 (C) Microsoft Corporation。保留所有权利。

PS D:\node> cd module-loading
PS D:\node\module-loading> node index.js
hello world
3
2
{ num: 6 }
{ num: 7 }
{ testField: 'testName' }
PS D:\node\module-loading>
```

二、双线程通信

主线程采用 new 命令，调用 Worker()构造函数，新建一个 Worker 线程：

```
1   var worker = new Worker('work.js');
```

主线程调用 worker.postMessage()方法，向 Worker 发消息：

```
3   worker.postMessage('Hello World');
4   worker.postMessage({method: 'echo', args: ['Work']});
```

主线程通过 worker.onmessage 指定监听函数，接收子线程发回来的消息：

```
worker.onmessage = function (event) {
  console.log('Received message ' + event.data);
  doSomething();
}

function doSomething() {
  // 执行任务
  worker.postMessage('Work done!');
}
```

Worker 完成任务以后，主线程就可以把它关掉：

```
worker.terminate();
```

Worker 线程内部需要有一个监听函数，监听 message 事件：

work.js

```
worker > JS work.js > ...
self.addEventListener('message', function (e) {
  self.postMessage('You said: ' + e.data);
}, false);

```

三、渲染

首先写一段 vdom：

```
const vdom = {
    type: 'ul',
    props: {
        className: 'list'
    },
    children: [
        {
            type: 'li',
            props: {
                className: 'item',
                style: {
                    background: 'blue',
                    color: '#fff'
                },
                onClick: function() {
                    alert(1);
                }
            },
            children: [
                'aaaa'
```

```
        {
            type: 'li',
            props: {
                className: 'item'
            },
            children: [
                'bbbbddd'
            ]
        },
        {
            type: 'li',
            props: {
                className: 'item'
            },
            children: [
                'cccc'
            ]
        }
    ]
};
```

render 函数如下所示：

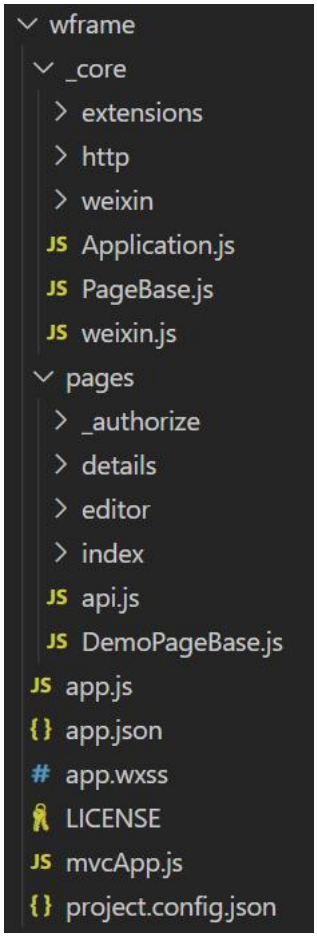
```
const render = (vdom, parent = null) => {
    const mount = parent ? (el => parent.appendChild(el)) : (el => el);
    if (isTextVdom(vdom)) {
        return mount(document.createTextNode(vdom));
    } else if (isElementVdom(vdom)) {
        const dom = mount(document.createElement(vdom.type));
        for (const child of vdom.children) {
            render(child, dom);
        }
        for (const prop in vdom.props) {
            setAttribute(dom, prop, vdom.props[prop]);
        }
        return dom;
    } else {
        throw new Error(`Invalid VDOM: ${vdom}.`);
    }
};
```

使用 jsx 即将 vdom 改写成如下更简洁的写法：

```
const jsx = <ul className="list">
    <li className="item" style={{ background: 'blue', color: 'pink' }} onClick={() => alert(2)}>aaa</li>
    <li className="item">{data.item1}<i>aaa</i></li>
    <li className="item">{data.item2}</li>
</ul>
```

四、小程序框架

项目结构如下图所示：其中_core 是小程序框架；pages 存放所有的页面；app.js 为程序主入口；mvcApp.js 为 require 函数库引用主入口。



1、app.js 和 Application 类

app.js 中定义了程序入口：

```
var mvcApp = require('mvcApp.js');
var Application = require('_core/Application.js');

function MvcApplication() {
    Application.call(this);
    this.initUrl = 'https://www.somdomain.com/api/client-config/get?key=wx_applet_wframe';
    this.host = 'http://localhost:18007';
    this.confgis = {
        host: 'http://localhost:18007',
        cdn: 'https://images.local-dev.cdn.somedomain.com'
    };
    this.mock = true;
    this.accessToken = null;
    this.useDefaultConfigsOnInitFailed = false;
};

MvcApplication.prototype = new Application();
```

```
MvcApplication.prototype.onInitialized = function (configs) {
    if (configs != null && configs !== '') {
        this.configs = JSON.parse(configs);
        this.host = this.configs.host;
    }
};

App(new MvcApplication());
```

可以看到 app.js 定义了一个 MvcApplication 类，继承自框架中的 Application 类，重写父类的 onInitialized 方法。

框架中的 Application 类如下所示：

```
var WebClient = require('http/WebClient.js');
var AuthorizeManager = require('weixin/AuthorizeManager.js');
var weixin = require('weixin.js');


function Application() {
    this.initUrl = '';
    this.host = '';
    this.session = null;
    this.initialized = false;
    this.mock = false;
    this.useDefaultConfigsOnInitFailed = false;
    this.authorizeManager = new AuthorizeManager();
    this._userInfo = null;
    this._readyHandlers = [];
};

Application.prototype = {
    onLaunch: function () {
        var me = this;
        if(this.initUrl === ''){
            throw 'please create YourOwnApplication class in app.js that in
        }
        var client = new WebClient();
        client.post(this.initUrl, null, function(result){
            if (result.success || me.useDefaultConfigsOnInitFailed){
                me.initialized = true;
                me.onInitialized(result.success ? result.value : null);
                me.triggerReady();
            }
            else{
                weixin.alert('小程序初始化失败', result.message);
            }
        }, '初始化中...');
    },
    onShow: function () {

    },
```

```
  onHide: function () {

  },
  onError: function () {

  },
  onPageNotFound: function () {

  },
  ready: function (callback) {
      var me = this;
      if (this.initialized === true) {
          callback && callback();
          return;
      }
      this._readyHandlers.push(callback);
  },
  triggerReady: function () {
      for (var i = 0; i < this._readyHandlers.length; i++) {
          var callback = this._readyHandlers[i];
          callback && callback();
      }
      this._readyHandlers = [];
  },
```

```
    onInitialized: function(configs){

    },
    getUserInfo: function(callback){
        var me = this;
        if(this._userInfo != null){
            callback && callback(this._userInfo.userInfo);
            return;
        }
        this.authorizeManager.getUserInfo(function(result){
            me._userInfo = result;
            callback && callback(me._userInfo.userInfo);
        });
    },
    getCurrentPage: function(){
        var pages = getCurrentPages();
        return pages.length > 0 ? pages[0] : null;
    }
};

module.exports = Application;
```

Applicaiton 类（及其子类）在 wframe 框架中的主要工作：

1. 应用程序初始化的时候从服务器获取一个配置，比如服务器域名（实现域名实时切换）、CDN 域名，以及其他程序配置信息；
2. 全局存储用户的授权信息和登陆之后的会话信息；
3. 全局 mock 开关；
4. 其他快捷方法，比如获取当前页面等。

Application 类核心执行流程：

1. 应用程序初始化时首先从服务器获取客户端配置信息；
2. 获取完成之后会触发 onInitialized 方法（在子类中覆写）和 ready 方法。

2、PageBase 类

框架中的 PageBase 类如下所示：

```
console.log("PageBae.js entered");

const app = getApp();

function PageBase(title) {
    this.vm = null;
    this.title = title;
    this.requireLogin = true;
};

PageBase.prototype = {
    onLoad: function (options) {
        var me = this;
        if (this.title != null) {
            this.setTitle(this.title);
        }
        this.onPreload(options);
        app.ready(function () {
            if (me.requireLogin && app.session == null) {
                app.getUserInfo(function (info) {
                    me.login(info, function (session) {
                        app.session = session;
                        me.ready(options);
                    });
                });
            }
            else {
                me.ready(options);
            }
        });
    },
    ready: function (options) {

    },
    onPreload: function(options){

    },
```

```
  render: function () {
      var data = {};
      for (var p in this.vm) {
          var value = this.vm[p];
          if (!this.vm.hasOwnProperty(p)) {
              continue;
          }
          if (value == null || typeof (value) === 'function') {
              continue;
          }
          if (value.__route__ != null) {
              continue;
          }
          data[p] = this.vm[p];
      }
      this.setData(data);
  },
```

```
    go: function (url, addToHistory) {
        if (addToHistory === false) {
            wx.redirectTo({ url: url });
        }
        else {
            wx.navigateTo({ url: url });
        }
    },
    goBack: function () {
        wx.navigateBack({});
    },
    setTitle: function (title) {
        this.title = title;
        wx.setNavigationBarTitle({ title: this.title });
    },
    login: function (userInfo, callback) {
        throw 'please implement PageBase.login method.';
    },
    getFullUrl: function () {
        var url = this.route.indexOf('/') === 0 ? this.route : '/' + this.route;
        var parts = [];
        for (var p in this.options) {
            if (this.options.hasOwnProperty(p)) {
                parts.push(p + "=" + this.options[p]);
            }
        }
        if (parts.length > 0) {
            url += "?" + parts.join('&');
        }
        return url;
    },
    isCurrentPage: function(){
        return this === getApp().getCurrentPage();
    }
};
```

```
PageBase.extend = function (prototypeObject) {
    var fn = new PageBase();
    for (var p in prototypeObject) {
        fn[p] = prototypeObject[p];
    }
    return fn;
};

module.exports = PageBase;
```

PageBase 类的三个实例属性：

1. vm：即 ViewModel 实例，可以理解为官方文档中的 Page 实例的 data 属性；
2. title：页面标题
3. requireLogin：是否需要登录，如果设置为 true，则页面 onLoad 执行后自动进入登录流程，登录完成后才会触发页面的 ready 方法；

PageBase 类的实例方法：

1. onLoad：wframe 框架自动会处理 requireLogin 属性，处理完成后才触发 ready 方法；
2. ready：每个业务级页面的主入口，每个业务级页面都应该实现 ready 方法，而不一定实现 onLoad 方法；
3. onPreload：在执行 onLoad 之前执行的方法，不支持异步；
4. render：将 ViewModel（即 data）呈现到页面上，在业务页面中直接使用 this.render()即可将更新的数据呈现出来；
5. go：页面跳转；
6. goBack：等于 wx.navigateBack；
7. setTitle：直接设置页面标题；
8. login：由业务级框架中的 DemoPageBase 实现；
9. getFullUrl：获取页面完整地址，包括路径和参数，便于直接跳转；
10. isCurrentPage：判断该页面实例是否在应用程序页面栈中处于当前页面，主要用于 setInterval 函数中判断用户是否已离开了页面；

3、DemoPageBase 类

这个类可以封装跟业务相关的很多逻辑，方便子类直接通过 this 调用相关方法。

```
var PageBase = require('../_core/PageBase.js');
var api = require('api.js');
const app = getApp();

function DemoPageBase(title) {
    PageBase.call(this, title);
};

DemoPageBase.prototype = new PageBase();

DemoPageBase.prototype.login = function (userInfo, callback) {
    var me = this;
    wx.login({
        success: function (res) {
            api.login(userInfo, res.code, function(value){
                app.accessToken = value.token;
                callback && callback(value);
            });
        }
    });
};


module.exports = DemoPageBase;
```