模块化的 Javascript 开发

葛亮@焦点科技

大纲

一．**Javascript** 开发之痛
二．**Javascript** 模块化简介
三．模块化实现原理
四．案例
五．总结

一．Javascript 开发之痛

1. 全局变量冲突


```
var count = 0;
$('#button').click(function () {
    count++;
    $('#text').html('点击了' + count + '次');
});
```

当 以上代码 遇到 以下代码

```
var count = $('.item').size();
$('#dashboard').html('共有' + count + '个');
```

1. 全局变量冲突

- 解决方案
  - 修改变量名 
  - 匿名函数包裹

```
(function () {
    var count = 0;
    $('#button').click(function () {
        count++;
        $('#text').html('点击了' + count + '次');
    });
})();

(function () {
    var count = $('.item').size();
    $('#dashboard').html('共有' + count + '个');
})();
```

```
(function () {
   ...
   ...
   ...
})();
```

全局函数/类该如何处理？

1. 全局变量冲突

```
function getKeys(hash) {
    var keys = [];
    for (var key in hash) {
        keys.push(key);
    }
    return keys;
}
```

```
var User = function () {
    // ...
};
User.prototype.get = function (id) {
    // ...
};
```

用对象模拟命名空间

```
var Focus = {};
Focus.getKeys = function (hash) {
    var keys = [];
    for (var key in hash) {
        keys.push(key);
    }
    return keys;
};
```

```
var Focus = {};
Focus.User = function () {
    // ...
};
Focus.User.prototype.get = function (id) {
    // ...
};
```

- 命名空间依然可能冲突
- 书写/记忆负担

1. 全局变量冲突

```
org.cometd.JSON = {};
org.cometd.JSON.toJSON = org.cometd.JSON.fromJSON = function(object)
{
    throw 'Abstract';
};

org.cometd.Utils = {};

org.cometd.Utils.isString = function(value)
{
    if (value === undefined || value === null)
    {
        return false;
    }
    return typeof value === 'string' ||  value instanceof String;
};

org.cometd.Utils.isArray = function(value)
{
    if (value === undefined || value === null)
```

神啊.....

2. 文件依赖难以维护

制作简单的页面效果

引入 **jquery.js**，添加 **index.js**

加上图片轮播效果

引入 **jquery.switch** 插件，修改 **index.js**

记住搜索的关键词

引入 **jquery.cookie** 插件，修改 **index.js**

记住图片播放的顺序

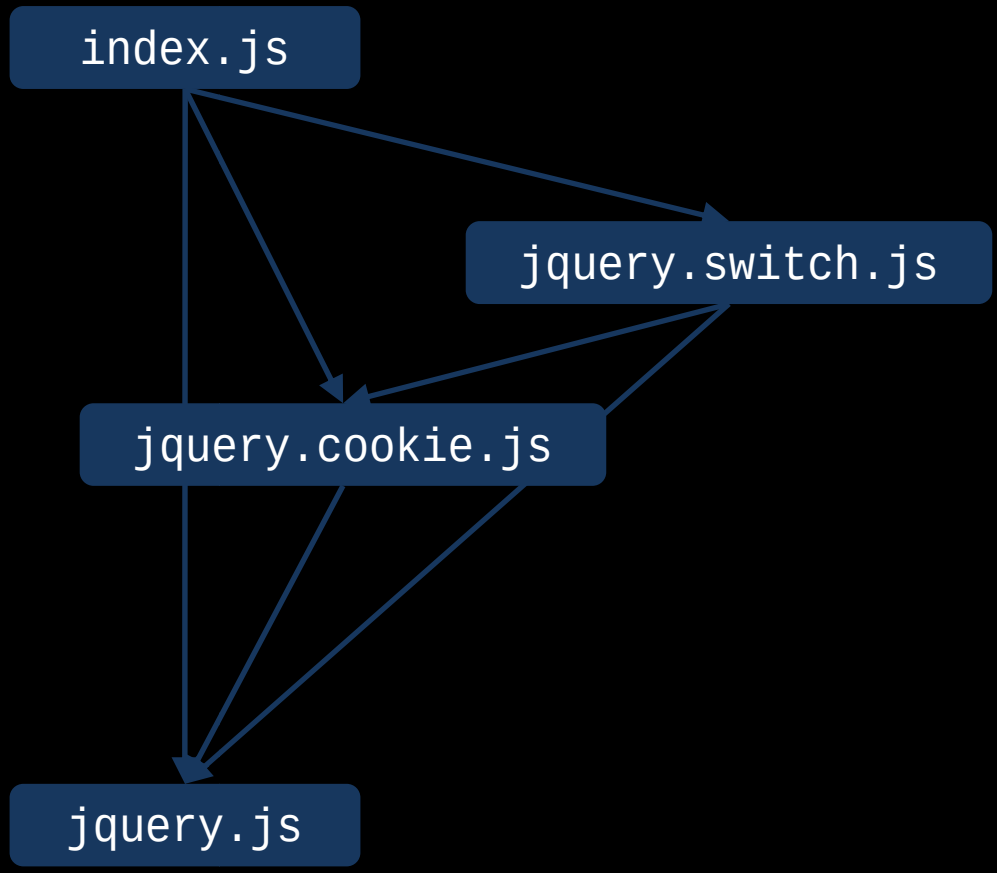
修改 **jquery.switch** 插件

去掉记住关键词功能

可以移除 **jquery.cookie** 插件吗 

```
<script type="text/javascript" src="jquery.js"></script>
<script type="text/javascript" src="jquery.switch.js"></script>
<script type="text/javascript" src="jquery.cookie.js"></script>
```

2. 文件依赖难以维护



去掉记住关键词功能

2. 文件依赖难以维护

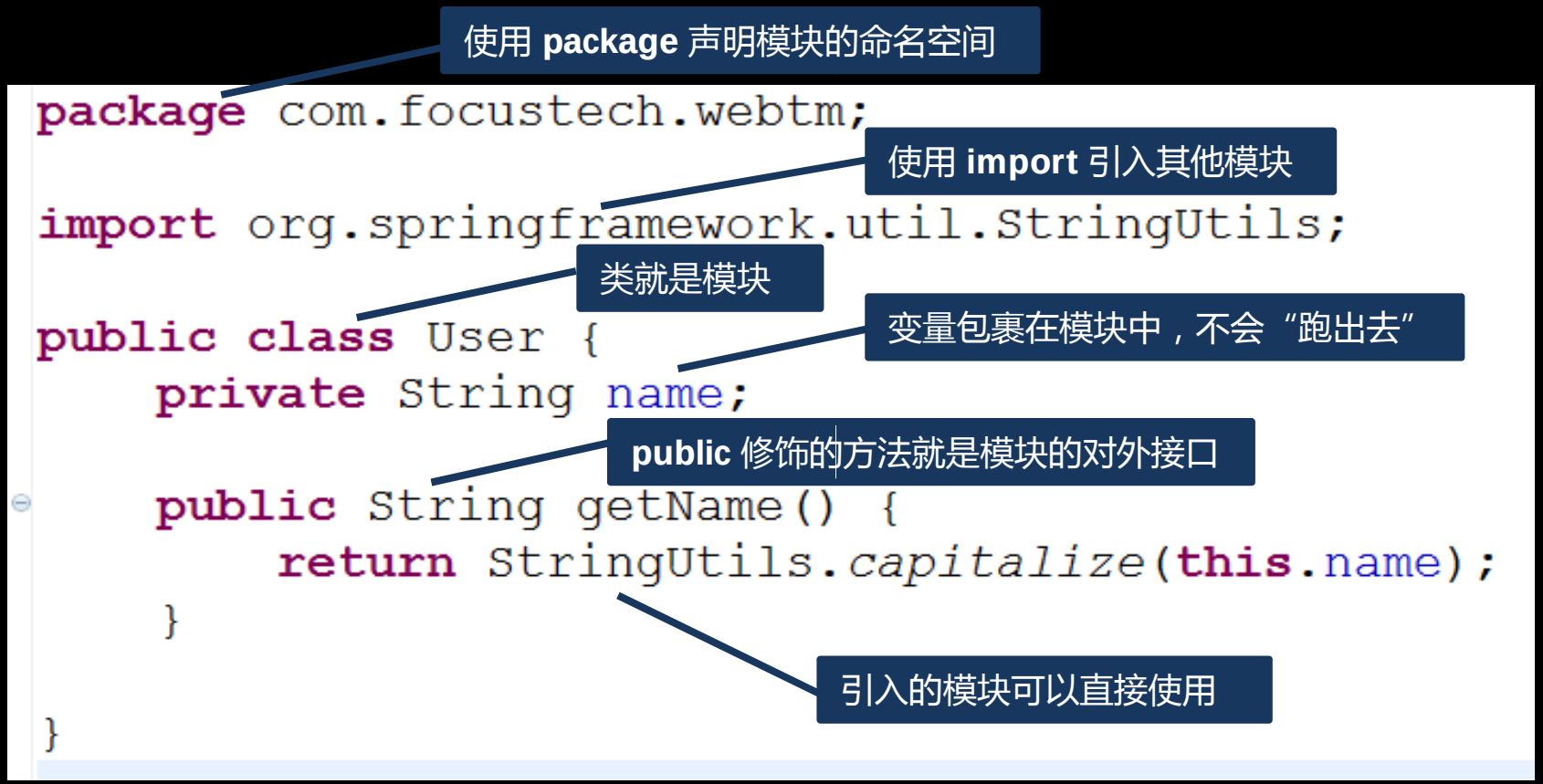
```
ascript" src="/script/jquery.js"></script>
ipt" src="/script/jquery.suggest.js"></script>
ipt" src="/script/jquery.pop.js"></script>
ipt" src="/script/jquery.placeholder.js"></script>
ipt" src="/script/jquery.linkage.js"></script>
ipt" src="/script/jquery.validate.js"></script>
ipt" src="/script/jquery.datepick.js"></script>
ipt" src="/script/jquery.datepick.validation.js"></script>
ipt" src="/script/jquery.multiprovcity.js"></script>
ipt" src="/script/jquery.numeric.js"></script>
ipt" src="/script/jquery.catalog.js"></script>
ipt" src="/script/swfupload.js"></script>
ipt" src="/script/jquery.cookie.js"></script>
ipt" src="/script/common/votips.js"></script>
ipt" src="/script/common/title_tips.js"></script>
ipt" src="/script/common/clone.js"></script>
ipt" src="/script/common/scale_img.js"></script>
ipt" src="/script/vo/demands/upload.js"></script>
ipt" src="/script/vo/demands/select_suppliers.js"></script>
ipt" src="/script/vo/demands/new.js"></script>
ipt" src="/script/vo/demands/score.js"></script>
ipt" src="/script/vo/demands/import.js"></script>
```

- 需要注意 **script** 元素的先后顺序
- 移除某个 **script** 元素时，需要检查依赖

疯了……

问题的根源在哪里？

- Javascript 中没有完备的模块系统
- 看看 Java 中的模块系统



Javascript 需要这样的模块系统！

二．Javascript 模块化简介

Javascript 模块规范

- **CommonJS**
  - 适用于 **NodeJS**，无法用于 **Web**
- **AMD**
  - 既适用于 **Web**，也适用于 **NodeJS**
  - 国外影响力较大
  - 模块加载器 **RequireJS**
- **CMD**
  - 专注于 **Web**，通过扩展的方式用于 **NodeJS**
  - 国内前端开源界的巨星
  - 模块加载器 **SeaJS**

以 **CMD** 规范为例讲解模块化

CMD 规范

定义模块

index.js

一个模块，一个文件

```
define(function() {
    var txt = 'Hello, world!';
    alert(txt);
});
```

使用 **define** 函数定义模块
函数的参数是一个 **function**

模块的代码书写在匿名 **function** 中

使用模块

test.html

引入 **SeaJS** 模块加载器

```
<script src="/sea.js"></script>
<script>
seajs.use('./index');
</script>
```

调用 **seajs.use** 函数来加载并使用模块
函数参数为模块标识

CMD 规范

模块的输出

util.js

```
define(function (require, exports) {
    var getKeys = function () {
        // ...
    };
    exports.getKeys = getKeys;
});
```

给函数增加两个参数：**require**，**exports**

通过 **exports** 参数对象，对外提供接口

使用模块

test.html

```
<script src="/sea.js"></script>
<script>
seajs.use('./util', function (util) {
  util.getKeys(...);
});
</script>
```

use 函数第二个参数为回调函数
回调函数的参数为加载模块的输出

CMD 规范

模块的依赖

index.js

```
define(function (require, exports) {
    var util = require('./util');
    var hash = {'key':'value'};
    util.getKeys(hash);
});
```

require 是一个函数
- 接受的参数为模块的标识（路径）
- 返回值为相应模块的输出

调用 **util** 模块输出的接口方法

使用模块

test.html

```
<script src="/sea.js"></script>
<script>
seajs.use('./index');
</script>
```

只需要加载 **index** 模块即可
SeaJS 会自动加载此模块依赖的其他模块

比较

util.js

```
var Focus = Focus || {};
Focus.util = {};
Focus.util.getKeys = function () {
};
```

```
define(function (require, exports) {
    var getKeys = function () {
    };
    exports.getKeys = getKeys;
});
```

index.js

```
var hash = {'key':'value'};
alert(Focus.util.getKeys(hash));
```

```
define(function (require, exports) {
    var util = require('./util.js');
    var hash = {'key':'value'};
    alert(util.getKeys(hash));
});
```

test.html

```
<script src="./util.js"></script>
<script src="./index.js"></script>
```

```
<script src="/sea.js"></script>
<script>
seajs.use('./index');
</script>
```

- 不用担心类/函数/命名空间的冲突
- 模块依赖由代码决定，程序自动维护
- 需要少量额外的编码

小结

- 遵循规范的代码可以很自然地规避变量/函数/类/命名空间冲突
- 文件之间的依赖由代码决定，并能自动管理

```
define(function (require, exports) {
    var a = require('...');

    // ...

    exports.x = ...;
});
```


模块化是一种更自然的代码组织方式

三．模块化实现原理

模块加载执行的流程

test.html

```
seajs.use('./index');
```

index.js

```
define(function (require, exports) {
    var util = require('./util');
    ...
});
```

1. 解析模块标识 **'./index'**，获取模块路径
2. 下载 **index** 模块，执行 **define**，缓存 **factory** 函数
3. 分析 **index** 模块的依赖，解析 **'./util'**
4. 下载 **util** 模块
5. 执行 **index** 模块的 **factory** 函数

模块加载后不会立刻执行，而是先做语法分析

如何下载模块？

- 异步加载 **script** 脚本的方式
  - XHR eval / XHR Injection
  - document.write script
  - script injection
- SeaJS / RequireJS 采用 script injection

```
var node = document.createElement('script');
node.src = '...';
head.appendChild(node);
```

如何分析模块依赖？

```
seajs.use('./index');
```

*模块被下载后，会自动先执行 **define***

```
define(function () {...});
```

*使用 **factory.toString()** 将函数体以字符串方式输出*

```
var code = factory.toString();
```

*使用正则表达式匹配字符串中的 **require** 语法即可*

```
var REQUIRE_RE = /(?:^|[^.$])\brequire\s*\(\s*(["'])([^"'\s\)]+)\1\s*\)/g
while ((match = REQUIRE_RE.exec(code))) {
...
```

require 是如何实现的？

```
define(function () {...});
```

*分析完依赖后，会先下载 **util** 模块，并缓存其 **factory** 函数*

```
var util = require('./util');
```

*执行 **require** 时，会找到对应的 **factory** 并调用，然后返回 **exports***

```
module.exports = {}

module.factory.call(
  window,
  module.require,
  module.exports,
  module)

return module.exports
```

小结

- 为什么需要 **define**？
  - 让模块代码不会在加载后立刻执行
  - 让模块加载器有机会在执行模块前做预处理
- 为什么不需要人工管理依赖了？
  - 模块代码指明了自身依赖模块的路径
  - 模块加载器在模块执行前，分析代码，并下载依赖的模块

四．案例

案例

- 网页版麦通
  - 网页版的即时聊天系统
  - 交互非常复杂

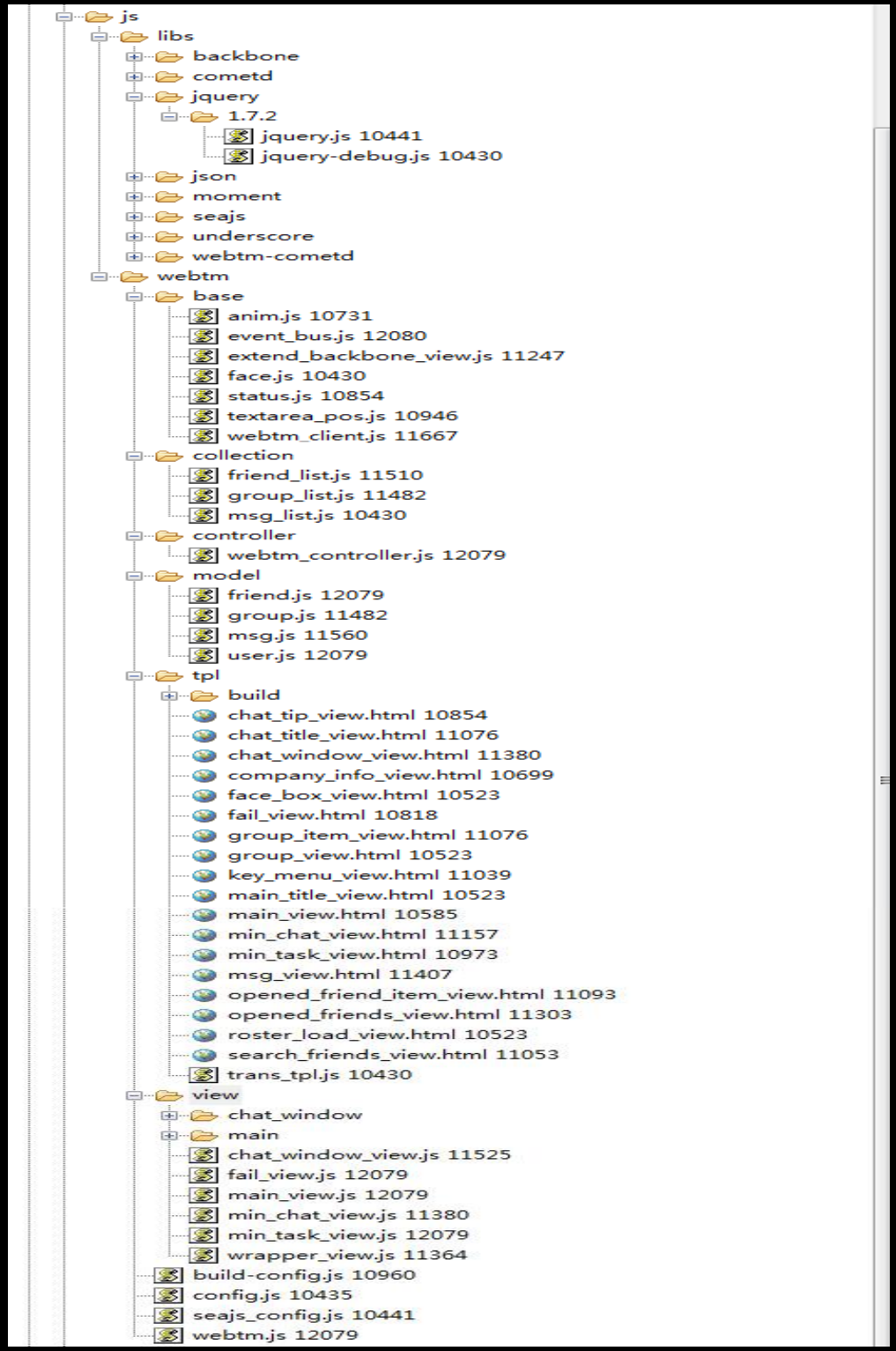
案例

- 项目情况
  - 近 **100** 个 **JS** 文件
  - 超过 **4000** 行代码
  - 采用模块化方式组织代码
  - 遵循 **CMD** 规范，使用 **SeaJS** 加载模块

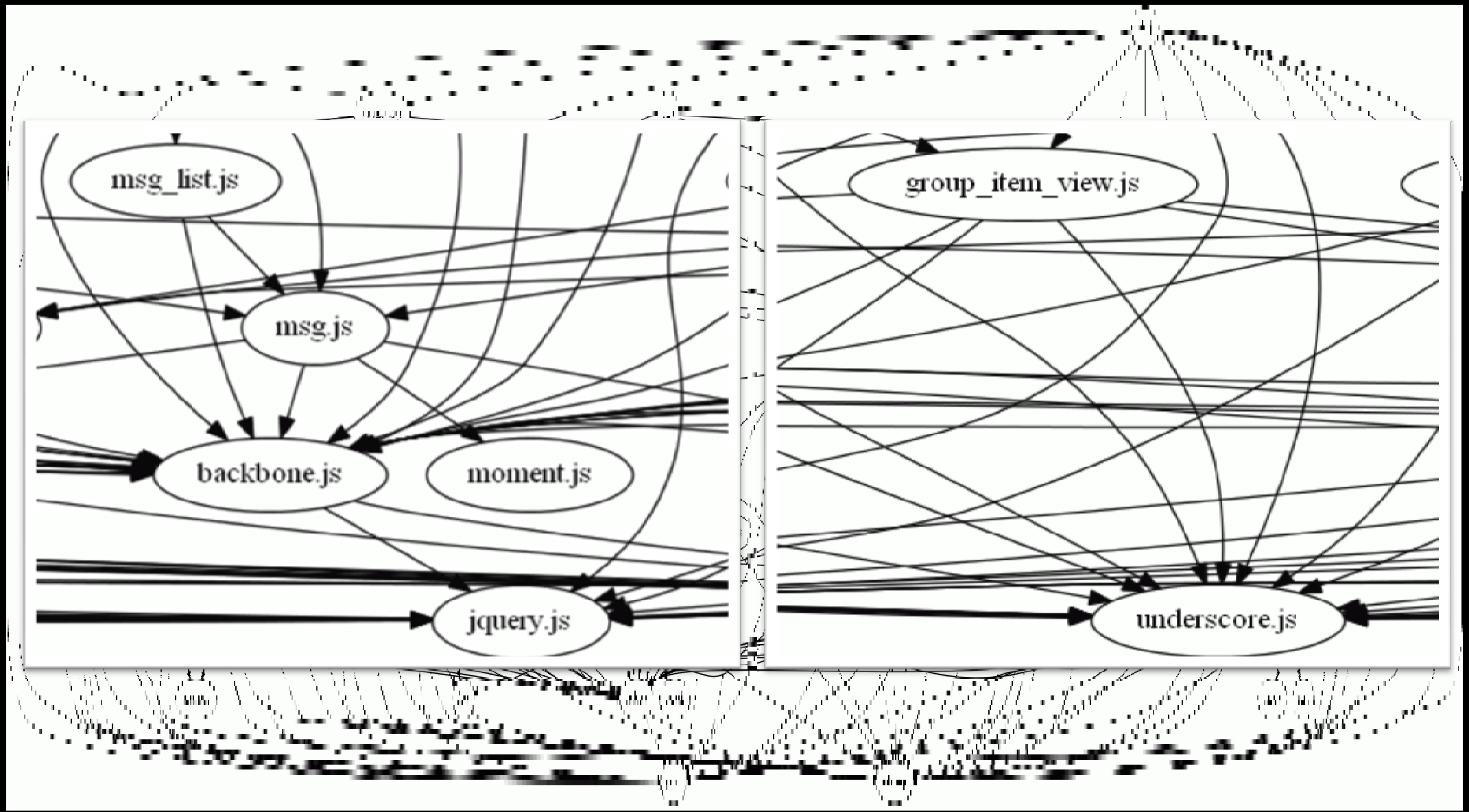
```
define(function (require) {
    var $ = require('$');
    var Backbone = require('backbone');
    var _ = require('underscore');
    var GroupList = require('./collection/group_list');
    var Group = require('./model/group');
    var User = require('./model/user');
    var Friend = require('./model/friend');
    var WrapperView = require('./view/wrapper_view');
    var MainView = require('./view/main_view');
    var MinTaskView = require('./view/min_task_view');
    var MinChatView = require('./view/min_chat_view');
    var ChatWindowView = require('./view/chat_window_view');
    var FailView = require('./view/fail_view');
    var webtmClient = require('./base/webtm_client');
    var config = require('./config');
    var EventBus = require('./base/event_bus');
    var SyncController = require('./controller/sync_controller');
    var NotifyControler = require('./controller/notify_controller');

    require('../../css/style.css');

    var webtm = {
        init:function (options) {
```



90 多个 JS 文件
目录结构也很复杂



恐怖的依赖关系

感谢模块化开发！感谢 **SeaJS！**

实战技巧

- **JS 如何组装 HTML**

*使用字符串保存 **HTML***

```
var html = '<div style="...">'
    + '<span style="...">'
    + '</span>'
    + '</div>';
var div = $(html);
```

可维护性极差，但代码简单

*用程序生成 **DOM** 元素*

```
var div = $('<div></div>');
var span = $('<span></span>');
div.css({
    'border' : '1px solid black',
    'position' : 'absolute',
    ...
});
span.css('color', 'red');
div.append(span);
```

可维护性稍强，但代码过于复杂

JS 如何组装 HTML

使用 **SeaJS**

div.html

```
<div class="wrap">
  <span class="red">
  </span>
</div>
```

div.css

```
.wrap {
    ...
}
.red {
    color : red;
}
```

div.js

```
define(function (require, exports) {
    require('./div.css');
    var html = require('./div.html');
    var div = $(html);
});
```

SeaJS 会将 **css** 文件插入 **dom**

SeaJS 会使用 **ajax** 获取 **html**
然后将文件内容作为字符串返回

可维护性高，代码也很简单

实战技巧

- **JS** 文件拆分的矛盾
  - 优点
    - 可维护性高
    - 多人协作更方便
  - 缺点
    - **http** 请求数变多，拖慢页面加载速度
  - 解决办法

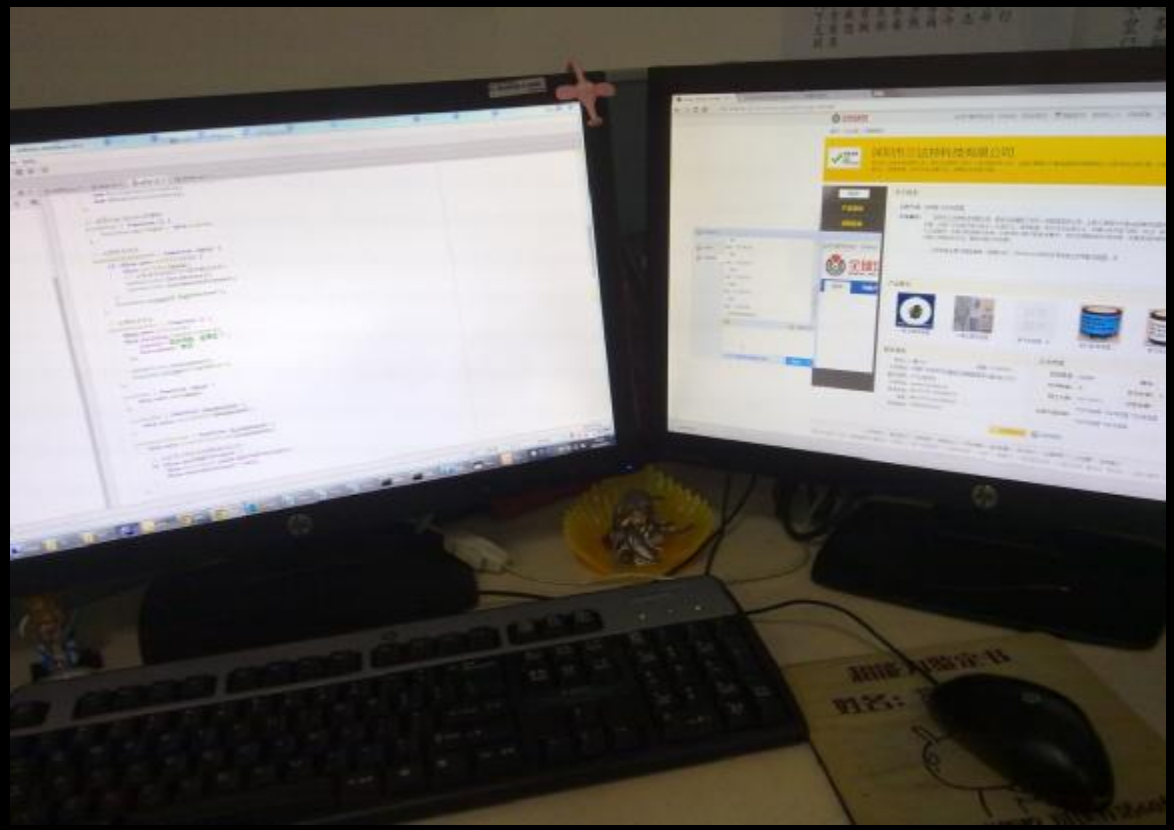
    使用 **SeaJS** 配套的 **spm** 工具合并 **JS**

    spm 通过语法分析，将所有的依赖文件合并到一起并压缩

```
spm build init.js --combine
Building init.js
... process init.js
... process stdin.js
... process calculator.js
... process stdout.js
... process math.js
Combined to __build/init.js
```

实战技巧

- 修改文件，自动刷新页面
  - 安装 **SeaJS** 配套的 **reload-server**
  - 进入代码目录，执行 **reload-server**
  - 给访问的 **url** 带上 **?seajs-reload** 参数



五．总结

- 优点
  - 解决了长期困扰 **Javascript** 开发的两大难题
  - 提高了程序的可维护性
  - 有利于多人协作
  - 纯前端的解决方案，简单，适用广泛

- 缺点
  - 由于 **Javascript** 语法的先天不足，模块化需要额外代码
  - 旧代码迁移的代价

使用 SeaJS 的网站

```
define({
    name   : '葛亮',
    email  : 'geliang@made-in-china.com',
    answer : function (question) {
        return getAnswer(question);
    }
});
```

```
define(function (require) {
    var geliang = require('./geliang');
    geliang.answer(...)
});
```