Javascript模块化

萌芽时代

2006年，ajax的概念被提出，前端拥有了主动向服务端发送请求并操作返回数据的能力，随着Google将此概念的发扬光大，传统的网页慢慢的向“富客户端”发展。前端的业务逻辑越来越多，代码也越来越多，于是一些问题就暴漏了出来：

1. 全局变量的灾难

小明定义了 i=1

小刚在后续的代码里：i=0

小明在接下来的代码里：if(i==1){...} //悲剧

2. 函数命名冲突

项目中通常会把一些通用的函数封装成一个文件，常见的名字有utils.js、common.js...

小明定义了一个函数：function formatData(){ }

小刚想实现类似功能，于是这么写：function formatData2(){ }

小光又有一个类似功能，于是：function formatData3(){ }

.....

避免命名冲突就只能这样靠丑陋的方式人肉进行。

3. 依赖关系不好管理

b.js依赖a.js，标签的书写顺序必须是

```
<script type="text/javascript" src="a.js"></script>
<script type="text/javascript" src="b.js"></script>
```

萌芽时代的解决方案

1. 用自执行函数来包装代码

```
modA = function(){
    var a=123;
    alert(a);
}()
```

这样function内部的变量就对全局隐藏了，达到是封装的目的。但是这样还是有缺陷的，modA这个变量还是暴漏到全局了，随着模块的增多，全局变量还是会越来越多。

2. java风格的命名空间

为了避免全局变量造成的冲突，人们想到或许可以用多级命名空间来进行管理(就是封装在不同的对象中，)于是，代码就变成了这个风格：

```
app.util.modA = xxx;
app.tools.modA = xxx;
app.tools.modA.format = xxx;
```

这样调用函数，写写都会觉得恶心，所以这种方式并没有被很多人采用。

3. jQuery风格的匿名自执行函数

```
(function(window){
    //代码
    window.jQuery = window.$ = jQuery;//通过给window添加属性而暴漏到全局
})(window);
```

jQuery的封装风格曾经被很多框架模仿，通过匿名函数包装代码，所依赖的外部变量传给这个函数，在函数内部可以使用这些依赖，然后在函数的最后把模块自身暴漏给window。

如果需要添加扩展，则可以作为jQuery的插件，把它挂载到$上。

这种风格虽然灵活了些，但并未解决根本问题：所需依赖还是得外部提前提供、还是增加了全局变量。

模块化面临什么问题

从以上的尝试中，可以归纳出js模块化需要解决那些问题：

1. 如何安全的包装一个模块的代码？（不污染模块外的任何代码）
2. 如何唯一标识一个模块？
3. 如何优雅的把模块的API暴漏出去？（不能增加全局变量）
4. 如何方便的使用所依赖的模块？

围绕着这些问题，js模块化开始了一段艰苦而曲折的征途。

三种方案

目前，主流的模块化方案有：**CommonJs** 、 **AMD/CMD** 、 **ES6** 三种方案

CommonJs

2009年，nodejs横空出世，开创了一个新纪元，人们可以用js来编写服务端的代码了。如果说浏览器端的js即便没有模块化也可以忍的话，那服务端是万万不能的。 大牛云集的CommonJs社区发力，制定了Modules/1.0（http://wiki.commonjs.org/wiki/Modules/1.0）规范，首次定义了一个模块应该长啥样。具体来说，Modules/1.0规范包含以下内容：

> 1. 模块的标识应遵循的规则（书写规范）
> 2. 定义全局函数require，通过传入模块标识来引入其他模块，执行的结果即为别的模块暴漏出来的API
> 3. 如果被require函数引入的模块中也包含依赖，那么依次加载这些依赖
> 4. 如果引入模块失败，那么require函数应该报一个异常

> 5. 模块通过变量exports来向往暴漏API，exports只能是一个对象，暴漏的API须作为此对象的属性。

遵循commonjs规范的代码看起来是这样的：

```
//aaa.js-----------------------------------
exports.add = function() {
    var sum = 0;
    for(var i=0; i<arguments.length;i++){
        sum+=arguments[i];
    }
    return sum;
};

//bbb.js-----------------------------------------
var add = require('aaa').add;
exports.increment = function(val) {
    return add(val, 1);
};

//ccc.js-----------------------------------------
var inc = require('bbb').increment;
var a = 1;
inc(a); // 2
```

Modules/1.0规范源于服务端（CommonJs原名叫ServerJs，从名字可以看出是专攻服务端的，为了统一前后端而改名CommonJs）。无法直接用于浏览器端，原因表现为：

1. 外层没有function包裹，变量全暴漏在全局。如上面例子中increment.js中的add。
2. 资源的加载方式与服务端完全不同，下表列出了js脚本在不同环境下运行的差异：

js脚本	**服务器端**	**浏览器**
require一个模块	本地从硬盘或者内存中读取	需要花费一个http请求，从服务器获取，模块请求到以后才会运行
对脚本运行的影响	无	require后面的一行代码，需要资源请求完成才能执行。没办法让代码同步执行 (由于浏览器端是以插入<script>标签的形式来加载资源的，所以像commonjs那样的写法会直接报错)

AMD/CMD

AMD---requirejs

AMD全称是Asynchronous Module Definition，即异步模块定义。AMD的思想正如其名，异步加载所需的模块，然后在回调函数中执行主逻辑。这正是我们在浏览器端开发所习惯了的方式。

AMD规范包含以下内容：

> 1. 用全局函数define来定义模块，用法为：define（ id?， dependencies?，factory）

2. id为模块标识，遵从CommonJS Module Identifiers规范
3. dependencies为依赖的模块数组，在factory中需传入形参与之一一对应
4. 如果dependencies的值中有”require”、”exports”或”module”，则与commonjs中的实现保持一致
5. 如果dependencies省略不写，则默认为[“require”, “exports”, “module”]，factory中也会默认传入require,exports,module
6. 如果factory为函数，模块对外暴漏API的方法有三种：return任意类型的数据、exports.xxx=xxx、module.exports=xxx
7. 如果factory为对象，则该对象即为模块的返回值

依据AMD规范开发的出了**requirejs**，遵循requirejs规范的代码看起来是这样的：

```
//a.js
define(function(){
     console.log('a.js执行');
     return {
          hello: function(){
               console.log('hello, a.js');
          }
     }
});

//b.js
define(function(){
     console.log('b.js执行');
     return {
          hello: function(){
               console.log('hello, b.js');
          }
     }
});

//main.js
require(['a', 'b'], function(a, b){
     console.log('main.js执行');
     a.hello();
     $('#btn').click(function(){
          b.hello();
     });
})

/*
上面的main.js被执行的时候，会有如下的输出：
a.js执行
b.js执行
main.js执行
hello, a.js
在点击按钮后，会输出：
hello, b.js
*/
```

requirejs在定义模块的时候，要把所有依赖模块都罗列一遍，而且还要在factory中作为形参传进去，要写两遍很大一串模块名称，像这样：

```
define(['a', 'b', 'c', 'd', 'e', 'f', 'g'], function(a, b, c, d, e, f, g){  ..... })
```

还要思考一个问题：

b.js 里的hello方法，是在#btn点击事件的时候才会运行，如果，这个点击事件从未发生，那么，b.js还有没有预加载的意义呢？

代码可以如下这样改写一下：

```
define(function(){
     console.log('main2.js执行');

     require(['a'], function(a){
          a.hello();
     });

     $('#b').click(function(){
          require(['b'], function(b){
               b.hello();
          });
     });
});

/*
define的参数中未写明依赖，那么main2.js在执行的时候，就不会预先加载a.js和b.js，只是执行到require语句的时候才会去加载，上述代码的输出如下：
main2.js执行
a.js执行
hello, a.js
*/
```

这就是名副其实的“懒加载”了。这样的懒加载无疑会大大减轻初始化时的损耗（下载和执行都被省去了），但是弊端也是显而易见的，在后续执行a.hello和b.hello时，必须得实时下载代码然后在回调中才能执行，这样的用户体验是不好的，用户的操作会有明显的延迟卡顿。

CMD---seaJS

CMD（Common Module Definition）规范，由国内淘宝前端大牛玉伯在开发SeaJS的时候提出来的，属于CommonJS的一种规范。

玉伯（王保平），淘宝前端类库 KISSY、前端模块化开发框架SeaJS、前端基础类库Arale的创始人。

```
/a.js
define(function(require, exports, module){
     console.log('a.js执行');
     return {
          hello: function(){
               console.log('hello, a.js');
          }
     }
});

//b.js
define(function(require, exports, module){
     console.log('b.js执行');
     return {
          hello: function(){
               console.log('hello, b.js');
          }
     }
});

//main.js
define(function(require, exports, module){
     console.log('main.js执行');

     var a = require('a');
     a.hello();

     $('#b').click(function(){
          var b = require('b');
          b.hello();
     });
 });
```

```
/*
上面的main.js执行会输出如下：
main.js执行
a.js执行
hello, a.js

a.js和b.js都会预先下载，但是b.js中的代码却没有执行，因为还没有点击按钮。当点击按钮的时候，会输出如下：
b.js执行
hello, b.js
可以看到b.js中的代码此时才执行。这样就真正实现了“就近书写，延迟执行“，不可谓不优雅。
*/
```

定义模块时无需罗列依赖数组，在factory函数中需传入形参require,exports,module，然后它会调用factory函数的toString方法，对函数的内容进行正则匹配，通过匹配到的require语句来分析依赖，这样就真正实现了commonjs风格的代码。

AMD 与 CMD的区别

CMD定义一个模块的时候不需要立即制定依赖模块，在需要的时候require就可以了（相当于按需加载）

AMD则相反，定义模块的时候需要制定依赖模块，并以形参的方式引入factory中。

```
//AMD
define(['dep1','dep2'],function(dep1,dep2){
     //内部只能使用制定的模块
      return function(){};
});

//CMD
define(function(require,exports,module){
   //此处如果需要某XX模块，可以引入
   var xx=require('XX');
});
```

方案	优势	劣势	特点
AMD	速度快	会浪费资源	预先加载所有的依赖，直到使用的时候才执行
CMD	只有真正需要才加载依赖	性能较差	直到使用的时候才定义依赖

ES6

ES6移除了关于模块如何加载/执行的内容，只保留了输出、引入模块的语法。

export输出

```
//a.js 方式一 分别输出每一个，可以是变量 函数 对象 类
export var a = 1;
export var obj = {name: 'abc', age: 20};

export function run(){console.log("aaaaa")};
```

```
//b.js 方式二 export default 对象方式输出全部
var a = 1;
var obj = {name: 'abc', age: 20};
function run(){console.log("bbbbbb")};

export default {a, run};
//default 表示不给导出的对象起名字，名字交给引入方（引入时也无需查看名称），并且可以选择性导出
```

import引入

输入文件一定要使用 "/" "./" "../" 路径

```
//在花括号中指明需使用的API，并且可以用as指定别名
import {a} from "./a.js";
import {run as go} from './a.js';
go();

//可以输入用export default定义的对象，
import mod from "./b.js";
mod.run();
```

目前**不是所有浏览器都能支持**ES6的模块，如果你要在开发中使用ES6语法来操作模块，目前的解决办法有三种：

1. 可以使用一些第三方模块来对ES6进行编译，转化为可以使用的ES5代码
2. 使用符合AMD规范的模块，例如：ES6 module transpiler。
3. 另外有一个项目也提供了加载ES6模块的方法，[es6-module-loader]

不过这都是一些临时的方案，随着ES后续版本的更新发布，模块的加载有了标准，浏览器给与了实现，这些工具也就没有用武之地了。