# 模块化

ES6之前，JS没有出现模块化系统。因为它在设计之初根本没有想到今天的JS应用场景。

JS主要在前端的浏览器中使用，js文件下载缓存到客户端，在浏览器中执行。
比如简单的表单本地验证，漂浮一个广告。
服务器端使用ASP、JSP等动态网页技术，将动态生成数据嵌入一个HTML模板，里面夹杂着JS后使用`<script>`标签，返回浏览器端执行。`<script>`还可以使用src属性，发起一个GET请求返回一个js文件，嵌入到当前页面执行环境中执行。
这时候的JS只是一些简单函数和语句的组合。

2005年之后，随着Google大量使用了AJAX技术之后，可以异步请求服务器端数据，带来了前端交互的巨大变化。前端功能需求越来越多，代码也越来也多。随着js文件的增多，灾难性的后果产生了：

- 众多js文件通过`<script>`引入到当前页面中，每一个js文件发起一个GET请求，众多的js文件都需要返回到浏览器端。网络开销成本颇高
- 习惯了随便写，js脚本中各种**全局变量污染**，函数名冲突
- JS脚本加载有顺序，JS文件中的代码之间的依赖关系（依赖前后顺序、相互依赖）。

亟待模块化的出现。

2008年V8引擎发布，2009年诞生了Nodejs，支持服务端JS编程。使用JS编程的项目规模越来越大，没有模块化是不可想象的。
之后社区中诞生诸多模块化解决方案。

CommonJS规范（2009年），使用全局require函数导入模块，将所有对象约束在模块对象内部，使用exports导出指定的对象。
最早这种规范是用在Nodejs后端的，后来又向前端开发移植，这样浏览器端开发也可以使用CommonJS了。

AMD（Asynchronous Module Definition）异步模块定义，这是由社区提出的一种浏览器端模块化标准。使用异步方式加载模块，模块的加载不影响它后面语句的执行。所有依赖这个模块的语句，都需要定义在一个回调函数，回调函数中使用模块的变量和函数，等模块加载完成之后，这个回调函数才会执行，就可以安全的使用模块的资源了。其实现就是AMD/RequireJs。AMD虽然是异步，但是会预先加载和执行。目前应用较少。

CMD（Common Module Definition），使用seajs，作者是淘宝前端玉伯，兼容并包解决了RequireJs的问题。CMD推崇as lazy as possible，尽可能的懒加载。

由于社区的模块化呼声很高，ES6开始提供支持模块的语法，但是浏览器目前支持还不够。

# ES6模块化

ES6中模块自动采用严格模式。

import语句，导入另一个模块导出的绑定。
export语句，从模块中导出函数、对象、值的，供其它模块import导入用。

## 导出

建立一个模块目录src，然后在这个目录下新建一个moda.js，内容如下：

```
1 // 缺省导出
2 export default class A{
3     constructor(x){
```

```
        this.x = x;
    }
    show() {
        console.log(this.x);
    }
}

// 导出函数
export function foo() {
    console.log('foo function');
}

// 导出常量
export const B = 'aaa';
```

## 导入

其它模块中导入语句如下

```
import { B, foo } from "./src/moda";
import * as mod_a from "./src/moda";
```

VS Code可以很好的语法支持了，但是nodejs运行环境，包括V8引擎，都不能很好的支持模块化语法。

# 转译工具

转译就是从一种语言代码转换到另一个语言代码，当然也可以从高版本转译到低版本的支持语句。

由于JS存在不同版本，不同浏览器兼容的问题，如何解决对语法的支持问题？
使用transpiler转译工具解决。

## babel

开发中可以使用较新的ES6+语法，通过转译器转换为指定的某些版本代码。

官网 https://babeljs.io/

参考文档

- 官网 https://babeljs.io/docs/en/usage
- https://babel.docschina.org/docs/en/

注意版本7.x较之前版本已经有了较大的变化，6.x文档请参看 https://babeljs.io/docs/en/6.26.3/index.html

## 离线转译安装配置（*）

### 1、初始化npm

在项目目录中使用

```
$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

```

```
See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (js) test
version: (1.0.0)
description: babel
entry point: (test.js)
test command:
git repository:
keywords:
author: wayne
license: (ISC)
About to write to C:\Users\Administrator\Documents\js\package.json:

{
  "name": "test",
  "version": "1.0.0",
  "description": "babel",
  "main": "test.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "wayne",
  "license": "ISC"
}

Is this ok? (yes) yes
```

在项目根目录下会生成package.json文件，内容就是上面花括号的内容。

## 2、设置镜像

https://docs.npmjs.com/cli/v7/configuring-npm/npmrc

.npmrc文件

- 可以放到npm的目录下npmrc文件中
- 可以放到用户家目录中
- 可以放到项目根目录中

参考 http://npm.taobao.org/

本次放到项目根目录中，内容如下`registry=https://registry.npm.taobao.org`
`$ echo "registry=https://registry.npm.taobao.org" > .npmrc`

## 3、安装

项目根目录下执行

```
$ npm install --save-dev @babel/core @babel/cli @babel/preset-env
```

```
--save-dev, -D说明
当你为你的模块安装一个依赖模块时，正常情况下你得先安装他们（在模块根目录下npm install module-name），然后连同版本号手动将他们添加到模块配置文件package.json中的依赖里（dependencies）。开发用。

--save 默认选项
--save和--save-dev可以省掉你手动修改package.json文件的步骤。
spm install module-name --save 自动把模块和版本号添加到dependencies部分。部署运行时用。
spm install module-name --save-dev 自动把模块和版本号添加到devdependencies部分
```

安装完后，在项目根目录下出现`node_modules目录`，里面有babel相关模块及依赖的模块。

安装时，会一并安装其他依赖组件。

## 4、配置babel和安装预设

Babel 7.8.0+使用配置文件babel.config.json

https://babel.docschina.org/docs/en/babel-preset-env

```
{
  "presets": [
    [
      "@babel/env",
      {
        "targets": {
          "edge": "17",
          "firefox": "60",
          "chrome": "67",
          "safari": "11.1"
        },
        "useBuiltIns": "usage",
        "corejs": "3.6.5"
      }
    ]
  ]
}
```

"useBuiltIns": "usage", 用什么，use了哪些，就打包什么，也称按需打包，否则体积太大。需要配合corejs选项，在7.4.0+之后替代polyfill。所谓polyfill就是给JavaScript提供缺失的功能，比如Promise、Symbol、Generator等。

**注意**：经测试，`"useBuiltIns": "usage"`可能有问题，后面的测试可以改成`"useBuiltIns": "entry"`

安装依赖

```
$ npm install -D @babel/preset-env
```

## 5、准备目录

项目根目录下建立src和dist目录。
`src`是源码目录；
`dist`是目标目录。

习惯上把源码放在src下，配置还是放在根目录。

## 6、修改package.json

替换为`scripts`的部分

```
{
  "name": "js",
  "version": "1.0.0",
  "description": "",
  "main": "test.js",
  "scripts": {
    "build": "babel src -d dist"
  },
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "babel-cli": "^6.26.0",
    "babel-core": "^6.26.0"
  }
}
```

`babel src -d dist`意思是从src目录中转译后的文件输出到lib目录

## 7、准备js文件

在src中的mod.js

```
// 缺省导出
export default class A{
    constructor(x){
        this.x = x;
    }
    show() {
        console.log(this.x);
    }
}

export function foo() {
    console.log('foo function');
}
```

src目录下新建index.js

```
import A, {foo} from './mod';

var a = new A(100);
a.show();

foo();
```

直接在VS Code的环境下执行出错。估计很难有能够正常运行的环境。所以，要转译为ES5的代码。

在项目根目录下执行命令

```
$ npm run build
```

可以看到，2个文件被转译

运行文件

```
$ node dist/index.js
100
foo function
```

使用babel等转译器转译JS非常流行。
开发者可以在高版本中使用新的语法特性，提高开发效率，把兼容性问题交给转译器处理。

npx是包执行器命令，从npm 5.2开始提供。npx可以直接执行已经安装过的包的命令，而不用配置package.json中的run-script。

```
$ npx babel src -d dist

$ node dist/b.js
100
foo function
```

# 导入导出

说明：导出代码都在src/mod.js中，导入代码都写在src/index.js中，不在赘述

## 缺省导入导出

只允许一个缺省导出，缺省导出可以是变量、函数、类，但不能使用let、var、const关键字作为默认导出

```
// 缺省导出 匿名函数
export default function() {
    console.log('default export function')
}

// 缺省导入
import defaultFunc from './mod'
defaultFunc();
```

```
// 缺省导出 命名函数
export default function xyz() {
    console.log('default export function')
}

// 缺省导入
import defaultFunc from './mod'
defaultFunc();
```

缺省导入的时候，可以自己重新命名，可以不需要和缺省导出时的名称一致，但最好一致。

缺省导入，不需要在import后使用花括号。

## 命名导入导出

```
/**
 * 导出举例
 */
// 缺省导出类
export default class {
    constructor(x) {
        this.x = x;
    }
    show(){
        console.log(this.x);
    }
}

// 命名导出 函数
export function foo(){
    console.log('regular foo()');
}

// 函数定义
function bar() {
    console.log('regular bar()');
}

// 变量常量定义
let x = 100;
var y = 200;
const z = 300;

// 导出
export {bar, x, y, z};


/**
 * ~~~~~~~~~~~~~~~
 * 导入举例
 * as 设置别名
 */
import defaultCls, {foo, bar, x, y, z as CONST_C} from './mod';

foo();
bar();
console.log(x); // x只读，不可修改，x++异常
console.log(y); // y只读
console.log(CONST_C);

new defaultCls(1000).show();
```



也可以使用下面的形式，导入所有导出，但是会定义一个新的名词空间。使用名词空间可以避免冲突。

```
import * as newmod from './mod';

newmod.foo();
newmod.bar();
new newmod.default(2000).show();
```

马哥教育
IT人的高薪职业学院