# JavaScript模块化

**模块化雏形**

1. 早期，JavaScript在项目地位较低，实现的功能有限
2. IE6之前，没有专门的JavaScript引擎，JavaScript的解析在浏览器渲染引擎中，是渲染引擎的一部分
3. 通常网页与用户交互的功能简单，代码量较少，JavaScript代码基本都写在 `<script></script>` 标签中
4. 随着互联网的发展，浏览器的能力越来越强，JavaScript能做的事情越来越多，代码越来越多，如果还将JavaScript代码放在标签中显然不合理
5. 此时的做法是将JavaScript代码单独提取到一个文件，在页面引入
6. 如果有多个页面需要使用到该脚本文件，这会导致每个页面都会引入一个无比之大的脚本
7. 将脚本按页面分离，按照页面引入对应的脚本
8. 最初的模块化的概念：**以页面为基准将脚本分成不同的代码块，将HTML结构与JavaScript逻辑分开**

```
// index.html
<script type="text/javascript">
   // some JavaScript code ...
</script>

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

// index1.html
<script src="./index1.js"></script>
// index2.html
<script src="./index2.js"></script>
```

**公共代码抽离**

- 遇到公共的逻辑代码段，将其抽离到一个单独的JavaScript文件中，然后在要使用的页面导入
- 但一个页面并不是用到了公共文件中的所有代码段，这样会导致页面引入大量无用代码
- 不能单独以页面为基准来划分程序块

```
// index.js
function test1() {
  console.log('test1');
}

// index1.js
function test1() {
  console.log('test1');
}

// common.js
function test1() {
  console.log('test1');
}
function test2() {
  console.log('test1');
}

// index.html
<script src="./common.js"></script>
<script src="./index.js"></script>
```

**按照程序来划分模块**

- 程序块之间若有依赖关系，需按照一定的顺序导入
- 引入的脚本共用全局作用域，会污染全局环境

```
// module_a.js
var a = [1, 2, 3, 4, 5].reverse();
// module_b.js
var b = a.concat([6, 7, 8, 9, 10]);
// module_c.js
var c = b.join('-');
// index.js
console.log(a);
console.log(b);
```

```
console.log(c);

<script src="./module_a.js"></script>
<script src="./module_b.js"></script>
<script src="./module_c.js"></script>
<script src="./index.js"></script>
```

**模块化解决的问题**

- 加载顺序
- 污染全局
  - 可控的声明
  - 类似于数据的声明

**闭包与立即执行函数**

- 函数有自己的作用域，模块的独立作用域
- 由于闭包的存在使得立即执行函数作用域仍然存在，只是函数被销毁
- 通过注入(传参)依赖其它模块，并且解决了污染全局的问题
- 模块独立并且可相互依赖，但依然没有解决加载顺序的问题

```
// module_a.js
var moduleA = (function () {
  var a = [1, 2, 3, 4, 5].reverse();
  return {
    a: a,
  };
})();

// module_b.js
var moduleB = (function (moduleA) {
  var b = moduleA.a.concat([6, 7, 8, 9, 10]);
  return {
    b: b,
  };
})(moduleA);

// module_c.js
var moduleC = (function (moduleB) {
  var c = moduleB.b.join('-');
  return {
    c: c,
  };
})(moduleB);
```

```
// index.js
(function (moduleA, moduleB, moduleC) {
  console.log(moduleA.a);
  console.log(moduleB.b);
  console.log(moduleC.c);
})(moduleA, moduleB, moduleC);

// index.html
<script src="./module_a.js"></script>
<script src="./module_b.js"></script>
<script src="./module_c.js"></script>
<script src="./index.js"></script>
```

### CommonJS

- CommonJS是一种模块化规范，来源于Node.js
- Node.js的诞生，依靠JavaScript实现模块导入导出，可互相依赖
- 模块化不再基于HTML页面，而是基于JavaScript语言本身实现的模块化
- 同步加载、缓存模块、用于服务端
- 一个模块就是一个文件，有独立的作用域，文件中的变量和函数都是私有的
- `exports` 导出， `require` 导入，引入后会被解析成立即执行函数

```
// module_a.js
var a = (function () {
  return [1, 2, 3, 4, 5].reverse();
})();

module.exports = {
  a,
};

// module_b.js
var moduleA = require('./module_a.js');
var b = (function () {
  return moduleA.a.concat([6, 7, 8, 9, 10]);
})();

module.exports = {
  b,
};

// module_c.js
var moduleB = require('./module_b.js');
var c = (function () {
```

```
  return moduleB.b.join('-');
})();

module.exports = {
  c,
};

// index.js
var moduleA = require('./module/module_a.js');
var moduleB = require('./module/module_b.js');
var moduleC = require('./module/module_c.js');
console.log(moduleA.a);
console.log(moduleB.b);
console.log(moduleC.c);
```

- 导出导入方式

```
module.exports = { xxx1, xxx2 }
module.exports = xxx

exports.xxx = xxx
// exports = xxx

const module = require('./xxx.js')
const { a, num } = require('./xxx.js')
```

- 问题
  - CommonJS为什么是同步加载?
  - MO `module.exports` 与 `exports` 的异同

AMD

- Asynchronous Module Definition 异步模块定义
- 浏览器本身不支持，但 `requirejs` 实现了AMD规范，使其可在浏览器运行
- `define` 定义模块， `require` 引入模块
- 异步加载，前置依赖，所有模板加载完毕才会执行回调函数

```
// module_a.js
define('moduleA', [], function () {
  var a = [1, 2, 3, 4, 5];
  return {
    a: a.reverse(),
  };
```

```
});

// module_b.js
define('moduleB', ['moduleA'], function (moduleA) {
  return {
    b: moduleA.a.concat([6, 7, 8, 9, 10]),
  };
});

// module_c.js
define('moduleC', ['moduleB'], function (moduleB) {
  return {
    c: moduleB.b.join('-'),
  };
});

// index.js
require.config({
  paths: {
    moduleA: 'module_a',
    moduleB: 'module_b',
    moduleC: 'module_c',
  },
});
require(['moduleA', 'moduleB', 'moduleC'], function (
  moduleA,
  moduleB,
  moduleC
) {
  console.log(moduleA.a);
  console.log(moduleB.b);
  console.log(moduleC.c);
});

// index.html
<script src="./js/require.js"></script>
<script src="./index.js"></script>

<script
    type="text/javascript"
    charset="utf-8"
    async=""
  data-requirecontext="_"
  data-requiremodule="moduleA"
  src="./module_a.js">
</script>
<script
    type="text/javascript"
    charset="utf-8"
    async=""
  data-requirecontext="_"
```

```
  data-requiremodule="moduleB"
  src="./module_b.js">
</script>
<script
    type="text/javascript"
    charset="utf-8"
    async=""
  data-requirecontext="_"
  data-requiremodule="moduleC"
  src="./module_c.js">
</script>
```

### CMD

- Common Module Definition 通用模块定义
- `define` 定义， `require` 加载， `exports` 导出， `module` 操作
- 依赖加载完毕执行回调函数
- 依赖就近，按需加载

```
// module_a.js
define(function (require, exports, module) {
  var a = [1, 2, 3, 4, 5];
  return {
    a: a.reverse(),
  };
});

// module_b.js
define(function (require, exports, module) {
  var moduleA = require('module_a'),
    b = [6, 7, 8, 9, 10];
  return {
    b: moduleA.a.concat(b),
  };
});

// module_c.js
define(function (require, exports, module) {
  var moduleB = require('module_b');
  return {
    c: moduleB.b.join('-'),
  };
});

// index.js
seajs.use(
  ['module_a.js', 'module_b.js', 'module_c.js'],
```

```
  function (moduleA, moduleB, moduleC) {
    console.log(moduleA.a);
    console.log(moduleB.b);
    console.log(moduleC.c);
  }
);

// index.html
<script src="./js/sea.js"></script>
<script src="./index.js"></script>
```

### UMD

- Universal Module Definition 统一模块定义
- AMD和CommonJS的结合

```
(function (root, factory) {
  if (typeof define === 'function' && define.amd) {
    define(['jQuery'], factory);
  } else if (typeof exports === 'object') {
    module.exports = factory();
  } else {
    root.testModule = factory(root.jQuery);
  }
})(this, function () {
    function myModule() {}
    return myModule
});
```

以上所有的模块化方案都是基于开发者社区及一些知名大公司发起的，并非官方标准，且得不到ES官方的认可。因此，呼之欲出，ES6开始，ES官方给出了权威答案：ECMAScript Module

### ESM

- ES官方提供的模块化标准

```
// module_a.js
var a = [1, 2, 3, 4, 5].reverse();
export default a;

// module_b.js
import a from './module_a.js';
var b = a.concat([6, 7, 8, 9, 10]);
export default b;
```

```
// module_c.js
import b from './module_b.js';
var c = b.join('-');
export default c;

// index.js
import a from './module_a.js';
import b from './module_b.js';
import c from './module_c.js';
console.log(a);
console.log(b);
console.log(c);

// index.html
<script src="./index.js" type="module"></script>
```

- export 导出， import 导入

```
// tool.js
const arr = [1,2,3,4,5];
class Person {}

// 1. 默认导出 数据类型
export default arr;
// 2. 命名导出1
export const arr = [1,2,3,4,5]
export class Person {}
// 3. 命名导出2 模块集合的容器
export {
    arr,
    Person
}
// 4. 二者结合
export {
    arr
}
export default Person;

// 导入
import arr from './tool.js'

// 模块集合的解构
import {arr, Person } from './tool.js'
import {arr, Person } from './tool.js'
import * as toolModule from './tool.js'

import { arr }, Person from './tool.js'
```

- CommJS与ESM区别
  - CommonJS值输出值的拷贝，ESM输出值的引用
  - CommonJS运行时加载，ESM编译时输出
- CommonJS目前已兼容ESM
  - 文件后缀 `.mjs`
  - package.json指定 `type` 属性为 `module`

**TypeScript模块化**

- TypeScript支持AMD、CMD、CommonJS、ESM等模块化规范及标准

```
// person.ts
export interface Person {
    name: string,
    age: number
}
import { Person } from './person'

// person.ts
interface Person {
    name: string,
    age: number
}
export { Person }
export { Person as Man}
import { Man } from './person'
import * as Human from './person'

// person.ts
interface Person {
    name: string,
    age: number
}
type PeronAreaType {
    city: 'Bejing'
}
export dafault {
    Person,
    PeronAreaType
}
```

- `import types`

```
// api.ts
```

```
type APIRequestType {
    method: 'GET',
    token: '********'
}
export { APIRequestType }

import type { APIRequestType } from './api'
```

- export/require

```
// info.ts
const arrInfo = [1,2,3,4,5]
export = arrInfo;

import info = require('./info')
```

## 友情链接

- 闭包
- 立即执行函数
- CommonJS
- RequireJS
- SeaJS
- UMD
- ESM
- TypeScript