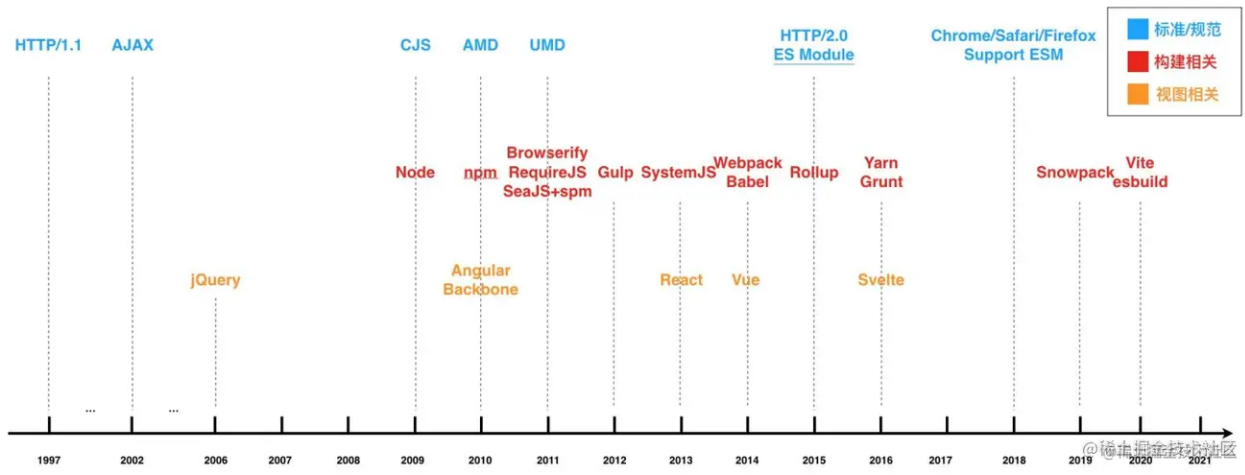
模块标准：为什么 ESM 是前端模块化的未来？

发布于 2022-05-09

2002 年 AJAX 诞生至今，前端从刀耕火种的年代，经历了一系列的发展，各种标准和工具百花齐放。下图中我们可以看到，自 2009 年 Node.js 诞生，前端先后出现了 `CommonJS` 、 `AMD` 、 `CMD` 、 `UMD` 和 `ES Module` 等模块规范，底层规范的发展催生出了一系列工具链的创新，比如 AMD 规范提出时社区诞生的模块加载工具 `requireJS` ，基于 CommonJS 规范的模块打包工具 `browserify` ，还有能让用户提前用上 `ES Module` 语法的 JS 编译器 `Babel` 、兼容各种模块规范的重量级打包工具 `Webpack` 以及基于浏览器原生 ES Module 支持而实现的 **no-bundle** 构建工具 `Vite` 等等。



总体而言，业界经历了一系列**由规范、标准引领工程化改革**的过程。构建工具作为前端工程化的核心要素，与底层的前端模块化规范和标准息息相关。接下来的时间，我就带你梳理一下前端模块化是如何演进的。这样你能更清楚地了解到各种模块化标准诞生的背景和意义，也能更好地理解 ES Module 为什么能够成为现今最主流的前端模块化标准。

无模块化标准阶段

早在模块化标准还没有诞生的时候，前端界已经产生了一些模块化的开发手段，如 文件划分 、 命名空间 和 `IIFE` 私有作用域 。下面，我来简单介绍一下它们的实现以及背后存在的问题。

1. 文件划分

文件划分方式是最原始的模块化实现，简单来说就是将应用的状态和逻辑分散到不同的文件中，然后通过 HTML 中的 script 来一一引入。下面是一个通过 文件划分 实现模块化的具体例子:

```
// module-a.js
let data = "data";


// module-b.js
function method() {
  console.log("execute method");
}


// index.html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
  </head>
  <body>
    <script src="./module-a.js"></script>
    <script src="./module-b.js"></script>
    <script>
      console.log(data);
      method();
    </script>
  </body>
</html>
```

从中可以看到 `module-a` 和 `module-b` 为两个不同的模块，通过两个 script 标签分别引入到 HTML 中，这么做看似是分散了不同模块的状态和运行逻辑，但实际上也隐藏着一些风险因素:

> 模块变量相当于在全局声明和定义，会有变量名冲突的问题。比如 `module-b` 可能也存在 `data` 变量，这就会与 `module-a` 中的变量冲突。
>
> 由于变量都在全局定义，我们很难知道某个变量到底属于哪些模块，因此也给调试带来了困难。
>
> 无法清晰地管理模块之间的依赖关系和加载顺序。假如 `module-a` 依赖 `module-b` ，

那么上述 HTML 的 script 执行顺序需要手动调整，不然可能会产生运行时错误。

2. 命名空间

命名空间 是模块化的另一种实现手段，它可以解决上述文件划分方式中 全局变量定义 所带来的一系列问题。下面是一个简单的例子:

```
// module-a.js
window.moduleA = {
  data: "moduleA",
  method: function () {
    console.log("execute A's method");
  },
};

// module-b.js
window.moduleB = {
  data: "moduleB",
  method: function () {
    console.log("execute B's method");
  },
};

<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
  </head>
  <body>
    <script src="./module-a.js"></script>
    <script src="./module-b.js"></script>
    <script>
      // 此时 window 上已经绑定了 moduleA 和 moduleB
      console.log(moduleA.data);
      moduleB.method();
    </script>
  </body>
</html>
```

这样一来，每个变量都有自己专属的命名空间，我们可以清楚地知道某个变量到底属于哪个 模块 ，同时也避免全局变量命名的问题。

3. IIFE(立即执行函数)

不过，相比于 命名空间 的模块化手段， IIFE 实现的模块化安全性要更高，对于模块作用域的区分更加彻底。你可以参考如下 IIFE 实现模块化 的例子:

```
// module-a.js
(function () {
  let data = "moduleA";

  function method() {
    console.log(data + "execute");
  }

  window.moduleA = {
    method: method,
  };
})();


// module-b.js
(function () {
  let data = "moduleB";

  function method() {
    console.log(data + "execute");
  }

  window.moduleB = {
    method: method,
  };
})();


// index.html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
  </head>
  <body>
    <script src="./module-a.js"></script>
    <script src="./module-b.js"></script>
    <script>
      // 此时 window 上已经绑定了 moduleA 和 moduleB
      console.log(moduleA.data);
      moduleB.method();
    </script>
  </body>
</html>
```

我们知道，每个 IIFE 即 立即执行函数 都会创建一个私有的作用域，在私有作用域中的变量外界是无法访问的，只有模块内部的方法才能访问。拿上述的 module-a 来说:

```
// module-a.js
(function () {
  let data = "moduleA";

  function method() {
    console.log(data + "execute");
  }

  window.moduleA = {
    method: method,
  };
})();
```

对于其中的 `data` 变量，我们只能在模块内部的 `method` 函数中通过闭包访问，而在其它模块中无法直接访问。这就是模块 私有成员 功能，避免模块私有成员被其他模块非法篡改，相比于 命名空间 的实现方式更加安全。

但实际上，无论是 命令空间 还是 `IIFE` ，都是为了解决全局变量所带来的命名冲突及作用域不明确的问题，也就是在 文件划分方式 中所总结的 问题 1 和 问题 2 ，而并没有真正解决另外一个问题——**模块加载**。如果模块间存在依赖关系，那么 script 标签的加载顺序就需要受到严格的控制，一旦顺序不对，则很有可能产生运行时 Bug。

而随着前端工程的日益庞大，各个模块之间相互依赖已经是非常常见的事情，模块加载的需求已经成为了业界刚需，而以上的几种非标准模块化手段不能满足这个需求，因此我们需要指定一个行业标准去统一前端代码的模块化。

不过前端的模块化规范统一也经历了漫长的发展阶段，即便是到现在也没有实现完全的统一。接下来，我们就来熟悉一下业界主流的三大模块规范: `CommonJS` 、 `AMD` 和 `ES Module` 。

CommonJS 规范

CommonJS 是业界最早正式提出的 JavaScript 模块规范，主要用于服务端，随着 Node.js 越来越普及，这个规范也被业界广泛应用。对于模块规范而言，一般会包含 2 方面内容:

- 统一的模块化代码规范

- 实现自动加载模块的加载器(也称 `loader`)

对于 CommonJS 模块规范本身，相信有 Node.js 使用经验的同学都不陌生了，为了方便你理解，我举一个使用 CommonJS 的简单例子:

```
// module-a.js
var data = "hello world";
function getData() {
  return data;
}
module.exports = {
  getData,
};

// index.js
const { getData } = require("./module-a.js");
console.log(getData());
```

代码中使用 `require` 来导入一个模块，用 `module.exports` 来导出一个模块。实际上 Node.js 内部会有相应的 loader 转译模块代码，最后模块代码会被处理成下面这样:

```
(function (exports, require, module, __filename, __dirname) {
  // 执行模块代码
  // 返回 exports 对象
});
```

对 CommonJS 而言，一方面它定义了一套完整的模块化代码规范，另一方面 Node.js 为之实现了自动加载模块的 `loader` ，看上去是一个很不错的模块规范，但也存在一些问题:

> 模块加载器由 Node.js 提供，依赖了 Node.js 本身的功能实现，比如文件系统，如果 CommonJS 模块直接放到浏览器中是无法执行的。当然, 业界也产生了 browserify 这种打包工具来支持打包 CommonJS 模块，从而顺利在浏览器中执行，相当于社区实现了一个第三方的 loader。
> CommonJS 本身约定以同步的方式进行模块加载，这种加载机制放在服务端是没问题的，一来模块都在本地，不需要进行网络 IO，二来只有服务启动时才会加载模块，而服务通常启动后会一直运行，所以对服务的性能并没有太大的影响。但如果这种加载机制放到浏览器端，会带来明显的性能问题。它会产生大量同步的模块请求，浏览器要等待响应返回后才能继续解析模块。也就是说，**模块请求会造成浏览器 JS 解析过程的阻塞**，导致页面加载速度缓慢。

总之，CommonJS 是一个不太适合在浏览器中运行的模块规范。因此，业界也设计出了全新的规范来作为浏览器端的模块标准，最知名的要数 `AMD` 了。

AMD 规范

AMD 全称为 Asynchronous Module Definition，即异步模块定义规范。模块根据这个规范，在浏览器环境中会被异步加载，而不会像 CommonJS 规范进行同步加载，也就不会产生同步请求导致的浏览器解析过程阻塞的问题了。我们先来看看这个模块规范是如何来使用的:

```
// main.js
define(["./print"], function (printModule) {
  printModule.print("main");
});

// print.js
define(function () {
  return {
    print: function (msg) {
      console.log("print " + msg);
    },
  };
});
```

在 AMD 规范当中，我们可以通过 define 去定义或加载一个模块，比如上面的 main 模块和 print 模块，如果模块需要导出一些成员需要通过在定义模块的函数中 return 出去(参考 print 模块)，如果当前模块依赖了一些其它的模块则可以通过 define 的第一个参数来声明依赖(参考 main 模块)，这样模块的代码执行之前浏览器会先**加载依赖模块**。

当然，你也可以使用 require 关键字来加载一个模块，如:

```
// module-a.js
require(["./print.js"], function (printModule) {
  printModule.print("module-a");
});
```

不过 require 与 define 的区别在于前者只能加载模块，而 不能定义一个模块 。

由于没有得到浏览器的原生支持，AMD 规范需要由第三方的 loader 来实现，最经典的就是 requireJS 库了，它完整实现了 AMD 规范，至今仍然有不少项目在使用。

不过 AMD 规范使用起来稍显复杂，代码阅读和书写都比较困难。因此，这个规范并不能成为前端模块化的终极解决方案，仅仅是社区中提出的一个妥协性的方案，关于新的模块化规范的探索，业界从仍未停止脚步。

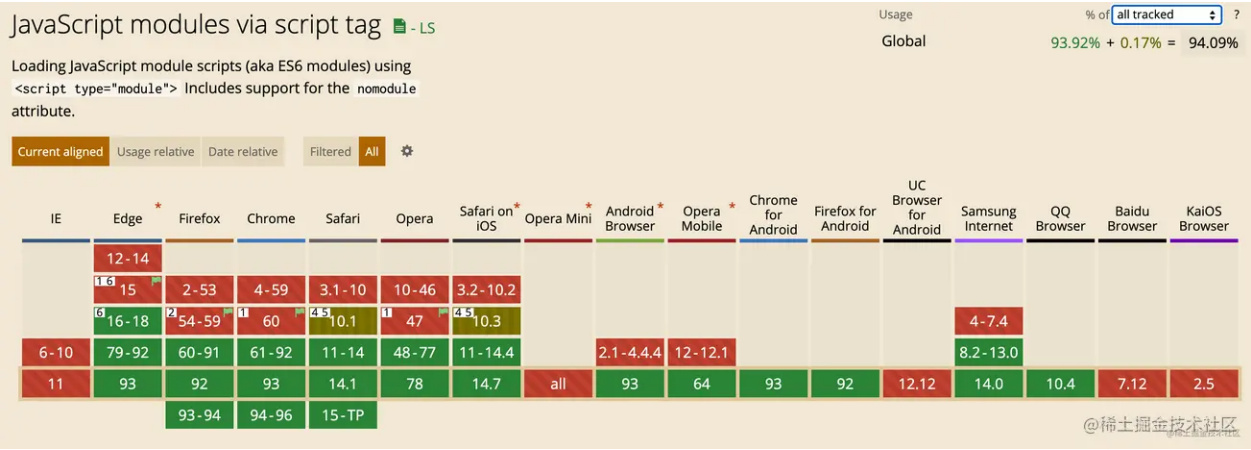
同期出现的规范当中也有 CMD 规范，这个规范是由淘宝出品的 `SeaJS` 实现的，解决的问题和 AMD 一样。不过随着社区的不断发展，SeaJS 已经被 `requireJS` 兼容了。

> 当然，你可能也听说过 `UMD` (Universal Module Definition)规范，其实它并不算一个新的规范，只是兼容 AMD 和 CommonJS 的一个模块化方案，可以同时运行在浏览器和 Node.js 环境。顺便提一句，后面将要介绍的 ES Module 也具备这种跨平台的能力。

ES6 Module

`ES6 Module` 也被称作 `ES Module` (或 `ESM`)， 是由 ECMAScript 官方提出的模块化规范，作为一个官方提出的规范， `ES Module` 已经得到了现代浏览器的内置支持。在现代浏览器中，如果在 HTML 中加入含有 `type="module"` 属性的 script 标签，那么浏览器会按照 ES Module 规范来进行依赖加载和模块解析，这也是 Vite 在开发阶段实现 no-bundle 的原因，由于模块加载的任务交给了浏览器，即使不打包也可以顺利运行模块代码，具体的模块加载流程我们会在下一节进行详细的解释。

大家可能会担心 ES Module 的兼容性问题，其实 ES Module 的浏览器兼容性如今已经相当好了，覆盖了 90% 以上的浏览器份额，在 CanIUse 上的详情数据如下图所示:



不仅如此，一直以 CommonJS 作为模块标准的 Node.js 也紧跟 ES Module 的发展步伐，从 `12.20` 版本开始正式支持原生 ES Module。也就是说，如今 ES Module 能够同时在浏览器与 Node.js 环境中执行，拥有天然的跨平台能力。

下面是一个使用 ES Module 的简单例子:

```
// main.js
```

```
// main.js
import { methodA } from "./module-a.js";
methodA();

//module-a.js
const methodA = () => {
  console.log("a");
};

export { methodA };


<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <link rel="icon" type="image/svg+xml" href="/src/favicon.svg" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Vite App</title>
  </head>
  <body>
    <div id="root"></div>
    <script type="module" src="./main.js"></script>
  </body>
</html>
```

如果在 Node.js 环境中，你可以在 `package.json` 中声明 `type: "module"` 属性:

```
// package.json
{
  "type": "module"
}
```

然后 Node.js 便会默认以 ES Module 规范去解析模块:

```
node main.js
// 打印 a
```

顺便说一句，在 Node.js 中，即使是在 CommonJS 模块里面，也可以通过 `import` 方法顺利加载 ES 模块，如下所示:

```
async function func() {
  // 加载一个 ES 模块
  // 文件名后缀需要是 mjs
  const { a } = await import("./module-a.mjs");
  console.log(a);
}

func();

module.exports = {
```

```
  func,
};
```

ES Module 作为 ECMAScript 官方提出的规范，经过五年多的发展，不仅得到了众多浏览器的原生支持，也在 Node.js 中得到了原生支持，是一个能够跨平台的模块规范。同时，它也是社区各种生态库的发展趋势，尤其是被如今大火的构建工具 Vite 所深度应用。可以说，ES Module 前景一片光明，成为前端大一统的模块标准指日可待。

当然，这一讲我们只简单介绍了 ESM。至于高级特性，我们将在「高级应用篇」专门介绍。你可以先利用我这里给到的官方资料提前预习： MDN 官方解释、ECMAScript 内部提案细节。

小结

这一节，我们要重点掌握**前端模块化的诞生意义**、**主流的模块化规范**和 **ESM 规范的优势**。

由于前端构建工具的改革与底层模块化规范的发展息息相关，从一开始我就带你从头梳理了前端模块化的演进史，从无模块化标准的时代开始谈起，跟你介绍了 文件划分 的模块化方案，并分析了这个方案潜在的几个问题。随后又介绍了 命名空间 和 IIFE 两种方案，但这两种方式并没有解决模块自动加载的问题。由此展开对前端模块化规范的介绍，我主要给你分析了三个主流的模块化标准: CommonJS 、 AMD 以及 ES Module ，针对每个规范从 模块化代码标准 、 模块自动加载方案 这两个维度给你进行了详细的拆解，最后得出 ES Module 即将成为主流前端模块化方案的结论。

本小节的内容就到这里了，希望能对你有所启发，也欢迎你把自己的学习心得打到评论区，我们下一节再见~

上一篇：开篇 ： 让 Vite 助力你的前端工程化之路　　下一篇：快速上手: 如何用 Vite 从零搭建前端项目?