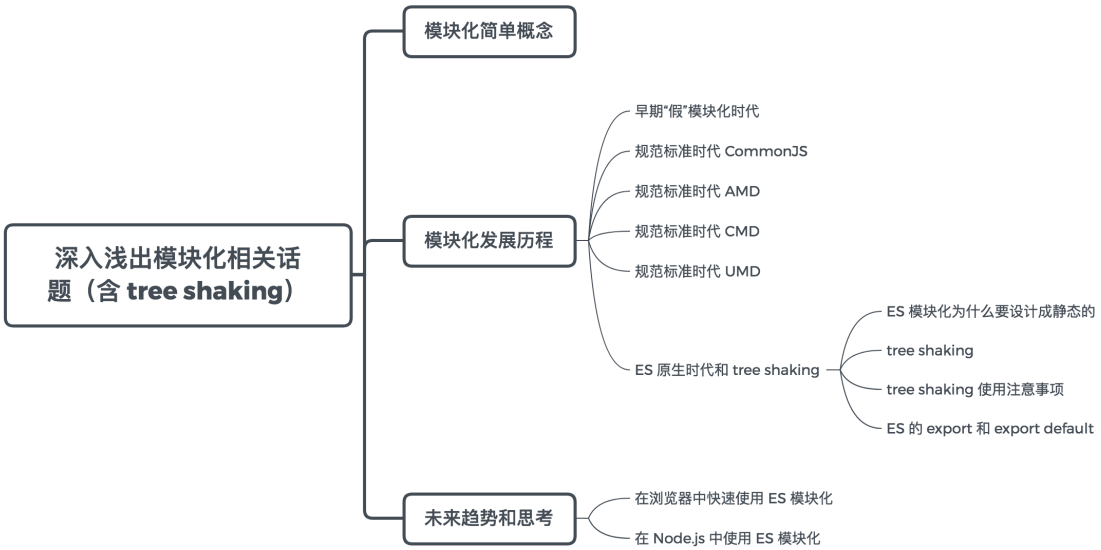
前端开发核心知识进阶：50 讲从夯实基础到突破瓶颈
来自 Lucas ... · 盐选专栏

查看详情 >

深入浅出模块化（含 tree shaking）（下）

本节课程，让我们继续模块化的话题。在此之前，先来回顾一下这个主题的知识点：



模块化发展历程

在上一讲中，介绍了以下方案实现模块化：

早期命名空间模拟模块化

CommonJS

AMD

CMD

UMD

接下来我们来探讨 ES 原生模块化的知识，并就 tree shaking 这个话题展开。

ES 原生时代和 tree shaking

ES 模块化（或称为 ESM）的具体使用方法我们不再具体介绍，请读者先了解相关基础内容。

ES 模块的设计思想是尽量**静态化**，这样能保证在编译时就确定模块之间的依赖关系，每个模块的输入和输出变量也都是确定的。CommonJS 和 AMD 模块，无法保证前置即确定这些内容，只能在运行时确定。这是 ES 模块化和其他规范的显著不同。第二个差别在于，CommonJS 模块输出的是一个值的**拷贝**，ES 模块输出的是值的**引用**。我们来具体看一下：

```
// data.js
export let data = 'data'
export function modifyData() {
   data = 'modified data'
}

// index.js
import { data, modifyData } from './lib'
console.log(data) // data
modifyData()
console.log(data) // modified data
```

我们在 index.js 中调用了 modifyData 方法，之后查询 data 值，得到了最新的变化。

而同样的逻辑，在 CommonJS 规范下的表现为：

```
// data.js
var data = 'data'
function modifyData() {
   data = 'modified data'
```

```
}

module.exports = {
   data: data,
   modifyData: modifyData
}

// index.js
var data = require('./data').data
var modifyData = require('./data').modifyData
console.log(data) // data
modifyData()
console.log(data) // data
```

因为 CommonJS 是输出了值的拷贝，而非引用，因此在调用 modifyData 之后，index.js 的 data 值并没有发生变化，其值为一个全新的拷贝。

ES 模块化为什么要设计成静态的

一个明显的优势是：通过静态分析，我们能够分析出导入的依赖。如果导入的模块没有被使用，我们便可以通过 tree shaking 等手段减少代码体积，进而提升运行性能。这就是基于 ESM 实现 tree shaking 的基础。

这么说可能过于笼统，我们从设计的角度分析这两种规范哲学的利弊。静态性需要规范去强制保证，不像 CommonJS 那样灵活，ES 模块化的静态性带来了限制：

> 只能在文件顶部 import 依赖
>
> export 导出的变量类型严格限制
>
> 变量不允许被重新绑定，import 的模块名只能是字符串常量，即不可以动态确定依赖

这样的限制在语言层面带来的便利之一是：我们可以通过作用域分析，分析出代码里变量所属的作用域以及它们之间的引用关系，进而可以推导出变量和导入依

赖变量的引用关系，在没有明显引用时，就可以进行去冗余。

tree shaking

上面说到的「在没有明显引用时，就可以进行去冗余」，就是我们经常提到的 **tree shaking**，它的目的就是减少应用中写出，但没有被实际运用的 JavaScript 代码。这样一来，无用代码的清除，意味着更小的代码体积，bundle size 的缩减，对用户体验起到了积极作用。

在计算机科学当中，一个典型去除无用代码、冗余代码的手段是 DCE，dead code elimination。**那么 tree shaking 和 DCE（Dead Code Elemination）有什么区别?**

Rollup 的主要贡献者 Rich Harris 做过这样的比喻：假设我们用鸡蛋做蛋糕。显然，我们不需要蛋壳而只需要蛋清和蛋黄，那么如何去除蛋壳呢？DCE 是这样做的：直接把整个鸡蛋放到碗里搅拌，蛋糕做完后再慢慢地从里面挑出蛋壳

相反，与 DCE 不同，tree shaking 是开始阶段就把蛋壳剥离，留下蛋清和蛋黄。事实上，也可以将 tree shaking 理解为广义 DCE 的一种，它在前置打包时即排除掉不会用到的代码。

当然说到底，tree shaking 只是一种辅助手段，良好的模块拆分和设计才是减少代码体积的关键。

Tree shaking 也有局限性，它还有很多不能清除无用代码的场景，比如 Rollup 的 tree shaking 实现只处理函数和顶层的 `import/export` 导入的变量，不能把没用到的类的方法消除；对于 tree shaking 来说，具有副作用的脚本无法被优化。

更多情况读者可以参考：

- tree-shaking 不完全指南
- webpack-common-shake

你的 Tree-Shaking 并没什么卵用

Webpack Tree shaking 深入探究

tree shaking 使用注意事项

webpack 和 Rollup 构建工具目前都有成熟的方案，但是笔者并不建议马上引入到项目中。事实上，是否要在成熟的项目上立即实施 tree shaking 需要妥善考虑。这里我也提供几篇收藏的文章，介绍了 tree shaking 的使用方法，这些基本操作内容，我们不再展开，可以按照官方文档实施，我也在文档之外推荐这些内容供大家学习。

Webpack 之 treeShaking

体积减少 80%！释放 webpack tree-shaking 的真正潜力

Tree-Shaking 性能优化实践 - 原理篇

ES 的 export 和 export default

ES 模块化导出有 export 和 export default 两种。这里我们建议减少使用 export default 导出，原因是一方面 export default 导出整体对象结果，不利于 tree shaking 进行分析；另一方面，export default 导出的结果可以随意命名变量，不利于团队统一管理。

Nicholas C. Zakas 有一篇文章： Why I've stopped exporting defaults from my JavaScript modules ，表达了类似的观点。

未来趋势和思考

个人认为，ES 模块化是未来不可避免的发展趋势，它的优点毫无争议，比如开箱即用的 tree shaking 和未来浏览器兼容性支持。Node.js 的 CommonJS 模块化方案甚至也会慢慢过渡到 ES 模块化上。如果你正在使用 webpack 构建应用项目，那么 ES 模块化是首选；如果你的项目是一个前端库，也建议使用 ES 模块化。这么看来，或许只有在编写 Node.js 程序时，才需要考虑 CommonJS。

在浏览器中快速使用 ES 模块化

目前各大浏览器较新版本都已经开始逐步支持 ES 模块了。如果我们想在浏览器中使用原生 ES 模块方案，只需要在 script 标签上添加一个 type="module" 属性。通过该属性，浏览器知道这个文件是以模块化的方式运行的。而对于不支持的浏览器，需要通过 nomodule 属性来指定某脚本为 fallback 方案：

使用 type="module" 的另一个作用是进行 ES Next 兼容性的嗅探。因为支持 ES 模块化的浏览器，都支持 ES Promise 等特性，基于此，应用场景较多。

在 Node.js 中使用 ES 模块化

Node.js 从 9.0 版本开始支持 ES 模块，执行脚本需要启动时加上 --experimental-modules，不过这一用法要求相应的文件后缀名必须为 *.mjs：

```
node --experimental-modules module1.mjs
import module1 from './module1.mjs'
console.log(module1)
```

另外，也可以安装 babel-cli 和 babel-preset-env，配置 .babelrc 文件后，执行：

```
./node_modules/.bin/babel-node
```

或：

```
npx babel-node
```

在工具方面，webpack 本身维护了一套模块系统，这套模块系统兼容了几乎所有前端历史进程下的模块规范，包括 AMD/CommonJS/ES 模块化等，具体分析咱们见后续课程《webpack 工程师 > 前端工程师》（即下一讲的内容）。

总结

通过本课程的学习，我们了解了 JavaScript 模块化的历史，重点分析了不同过渡方案的不同实现以及 ES 模块化标准的细节。希望读者对模块系统有一个更清

晰的认识，同时希望大家可以仔细阅读源码，对代码设计有自己的理解和体会。

点击查看下一节

webpack 工程师 > 前端工程师（上）