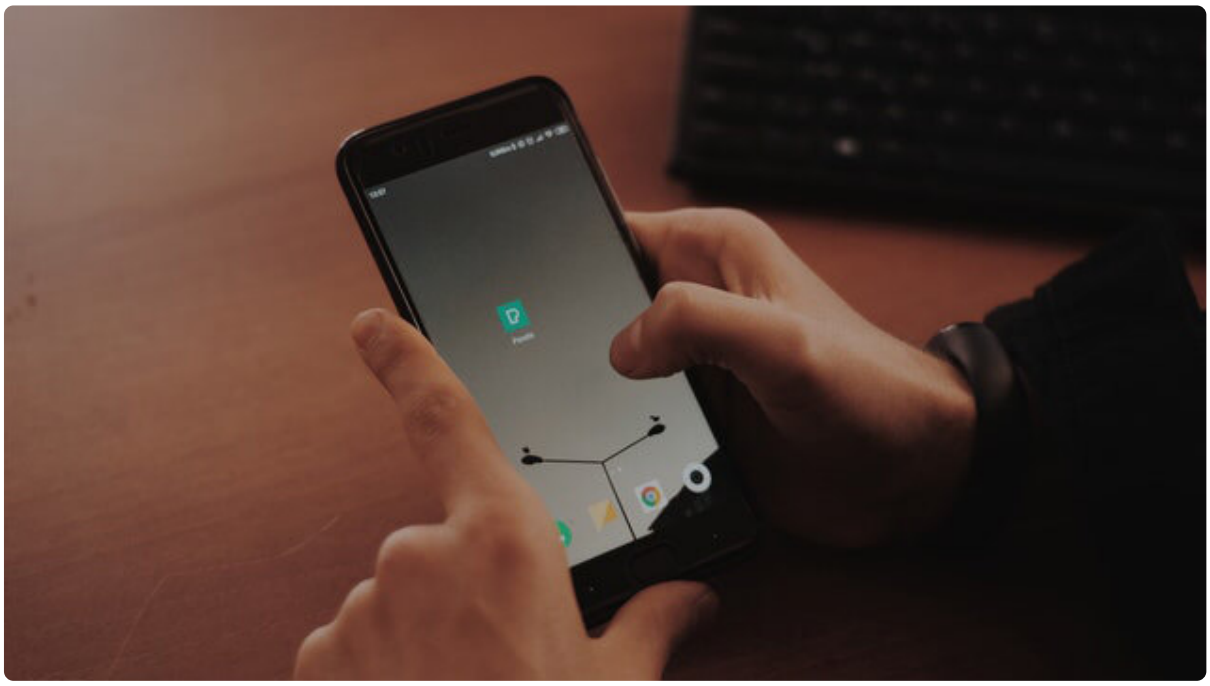
21 使用 Webpack 的 Tree-Shaking

更新时间：2019-06-24 09:32:52



> 你若要喜爱你自己的价值，你就得给世界创造价值。
>
> ——歌德

Tree-Shaking 是一个前端术语，本意为摇树的意思，在前端术语中通常用于描述移除 JavaScript 上下文中没用的代码，这样可以有效地缩减打包体积。关于 Tree-Shaking，Webpack 官方文档有一段很形象的描述：

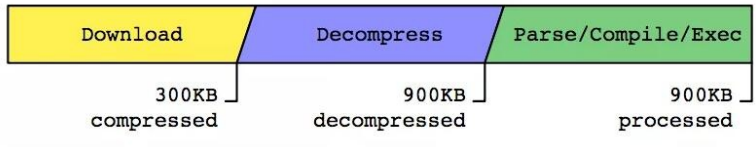
> 你可以将应用程序想象成一棵树。绿色表示实际用到的源码和 library，是树上活的树叶。灰色表示无用的代码，是秋天树上枯萎的树叶。为了除去死去的树叶，你必须摇动这棵树，使它们落下。
>
> 
>

Tree-Shaking 最早是由 Rich Harris 在打包工具 rollup.js提出并且实现的，其实在更早，Google Closure Compiler 也做过类似的事情。在 Webpack 2 中吸取了 Tree-Shaking 功能，并且在 Webpack 中得到实现。

Tree-Shaking 实现原理

Tree-Shaking 的本质是消除无用的 JavaScript 代码。无用代码消除（Dead Code Elimination）广泛存在于传统的编程语言编译器中，编译器可以判断出某些代码根本不影响输出，然后消除这些代码，这个称之为 DCE（Dead Code Elimination）。Tree-Shaking 是 DCE 的一种新的实现，Javascript 同传统的编程语言不同的是，JavaScript 绝大多数情况是在浏览器中执行，需要通过网络进行加载，然后解析 JavaScript 文件再执行。

2018 年年中，据 HTTP Archive 统计：移动端 JavaScript 文件的平均传输大小将近 350KB。你要知道，这仅仅是传输的大小。在网络传输的时候，JavaScript 往往是经过压缩的。也就是说，在浏览器解压缩之前，实际的大小会远远大于这个值。而这一点相当重要。如果考虑到浏览器处理数据的资源消耗，其中压缩是不得不考虑的。一个 300KB 的文件解压缩会达到 900KB，并且在分析和编译的时候，体积依然是 900KB。



由于网络的带宽限制，加载的 JavaScript 文件体积越小，整体解析执行时间更短，所以去除无用代码以减少文件体积，对 JavaScript 来说更有意义。

Tree-Shaking 和传统的 DCE 的方法又不太一样，传统的 DCE 消灭不可能执行的代码：

- 程序中没有执行的代码，如不可能进入的分支，`return` 之后的语句等；
- 导致 dead variable 的代码，写入变量之后不再读取的代码。

和 DCE 不同的是，Tree-Shaking 则更关注于消除没有用到的代码。通过之前的章节介绍过，Webpack 是基于 ES6 Modules 静态语法解析的构建工具，Tree-Shaking 之所以能够在 Webpack 实现，也是得益于 ES6 Modules 静态解析的特性。ES6 的模块声明保证了依赖关系是提前确定的，使得静态分析成为可能，这样在 Webpack 中代码不需要执行就可以知道是否被使用，自然就知道哪些是无用的代码了。

ES6 Modules 特点：

- ES6 中 `import` 和 `export` 是显性声明的；
- import 的模块名只能是字符串常量；
- ES6 模块的依赖关系是可以根据 `import` 引用关系推导出来的。
- ES6 模块的依赖关系与运行时状态无关

上面这些 ES6 Modules 的特点是 Tree-Shaking 的基础。所谓静态分析就是不执行代码，从字面量上对代码进行分析，ES6 之前的模块化，比如我们可以动态 require 一个模块，只有执行后才知道引用的什么模块，这个就不能通过静态分析去做优化。这是 ES6 Modules 在设计时的一个重要考量，也是为什么没有直接采用 CommonJS，正是基于这个基础上，才使得 Tree-Shaking 成为可能，这也是为什么 rollup.js 和 Webpack 2 都要用 ES6 Module 语法才能实现 Tree-Shaking。

Webpack Tree-Shaking 代码实战

在 Webpack 中，Tree-Shaking 是需要配合 `mode=production` 来使用的，这是因为 Webpack 的 Tree-Shaking 实际分了两步来实现：

1. Webpack 自己来分析 ES6 Modules 的引入和使用情况，去除不使用的 `import` 引入；
2. 借助工具（如 `uglifyjs-webpack-plugin` 和 `terser-webpack-plugin`）进行删除，这些工具只在 `mode=production` 中会被使用。

我们通过实例来看下这两个步骤，首先我们准备了两个文件：`utils.js`和`index.js`文件，其中`utils.js`中定义了两个方法`isNull`和`isNumber`：

```
// utils.js
export function isNull(obj) {
    console.log('isNull');
    return null === obj;
}

export function isNumber(obj) {
    console.log('isNumber');

    return typeof obj === 'number';
}
```

但是在`index.js`中`import`了`utils`的两个函数方法，但是实际却只用了`isNull`的方法：

```
// index.js
import {isNull, isNumber} from './utils';

isNull(1);
```

下面我们使用`webpack --mode=development`打包看下结果：

```
{
'./src/index.js': function(module, __webpack_exports__, __webpack_require__) {
    'use strict';
    // 注意！注意！注意！
    // 注意这里打包后_utils__WEBPACK_IMPORTED_MODULE_0__
    __webpack_require__.r(__webpack_exports__);
    /* harmony import */ var _utils__WEBPACK_IMPORTED_MODULE_0__ = __webpack_require__(
        /*! ./utils */ './src/utils.js'
    );

    Object(_utils__WEBPACK_IMPORTED_MODULE_0__['isNull'])(1);

},
'./src/utils.js': function(module, __webpack_exports__, __webpack_require__) {
    'use strict';
    __webpack_require__.r(__webpack_exports__);
    /* harmony export (binding) */ __webpack_require__.d(__webpack_exports__, 'isNull', function() {
        return isNull;
    });
    /* harmony export (binding) */ __webpack_require__.d(__webpack_exports__, 'isNumber', function() {
        return isNumber;
    });

    function isNull(obj) {
        console.log('isNull');
        return null === obj;
    }

    function isNumber(obj) {
        console.log('isNumber');

        return typeof obj === 'number';
    }
}
```

我们发现`index.js`的打包结果中，只保留了`isNull`的使用，而虽然我们同时`import`了`isNumber`和`isNull`，但最终`isNumber`并没有出现在`index.js`的打包结果内：

```
function(module, __webpack_exports__, __webpack_require__) {
    'use strict';
    // 注意！注意！注意！
    // 注意这里打包后_utils__WEBPACK_IMPORTED_MODULE_0__
    __webpack_require__.r(__webpack_exports__);
    /* harmony import */ var _utils__WEBPACK_IMPORTED_MODULE_0__ = __webpack_require__(
        /*! ./utils */ './src/utils.js'
    );

    Object(_utils__WEBPACK_IMPORTED_MODULE_0__['isNull'])(1);

}
```

但是`utils.js`打包后的内容没有变化，保留了`isNumber`的方法。

这说明，Webpack 的 Tree-Shaking 第一步只不过是去掉了无用的引用，但是并没有删除无用的代码，删除无用的代码是`mode=production`时候使用压缩工具实现的。那么我们在使用`webpack  --mode=production`来看下结果，格式化后，我们看到`isNumber`部分的关键字没有了（因为有`console.log('isNumber')`，可以搜索`isNumber`字符串关键字）：

```
!(function(e) {
    var t = {};
    //... 忽略内容
})({
    9: function(e, t, r) {
        'use strict';
        r.r(t), console.log('isNull');
    }
});
```

到此，我们已经理解 Tree-Shaking 的原理和使用方法了。

Tree-Shaking 并不是银弹

通过上面的实验，可能大家认为 Tree-Shaking 已经很了不起了，可以帮助我们缩减代码，但是很多情况下 Tree-Shaking 并不是银弹！首先基于 Tree-Shaking 的原理，所以我们的代码必须遵循 ES6 的模块规范，即使用`import`和`export`语法，如果是 CommonJS 规范（使用`require`）则无法使用 Tree-Shaking 功能。除了这点之外，在使用 Tree-Shaking 还有什么注意点或者 Tree-Shaking 处理不到的地方呢？

Tree-Shaking 处理 Class

下面我们再来看下 Tree-Shaking 对类的处理，首先创建一个`class.js`，内容如下：

```
// class.js
class Utils {
    foo() {
        console.log('foo');
    }
    bar() {
        console.log('bar');
    }
}
export default Utils;
```

然后我们在`class-entry.js`中引入这个`Class`，并且只是用`foo`的方法：

```
// class-entry.js
import Utils from './class';
const u = new Utils();
console.log(u.foo());
```

我们希望 Tree-Shaking 能够帮我们把不使用的`bar`方法干掉，但是实际 Tree-Shaking 做不了这样的事情：

```
function(e, t, o) {
    'use strict';
    o.r(t);
    const n = new class {
        foo() {
            console.log('foo');
        }
        bar() {
            console.log('bar');
        }
    }();
    console.log(n.foo());
}
```

这表明 webpack Tree-Shaking 只处理顶层内容，例如类和对象内部都不会再被分别处理，这主要也是由于 JavaScript 的动态语言特性所致，例如下面的代码：

```
import Utils from './class';
const u = new Utils();
console.log(u[Math.random() > 0.5 ? 'foo' : 'bar']());
```

JavaScript 的编译器并不能识别一个方法名字究竟是以直接调用的形式出现（`u.foo()`）还是以字符串的形式（`u['foo']()`）或者其他更加离奇的方式。因此误删方法只会导致运行出错，反而得不偿失。

副作用（**Side Effect**）代码

知道函数式编程的朋友都会知道**副作用**（Side Effect）这个词，副作用会在我们项目中频繁的出现。我们称模块（函数）具有副作用，就是说这个模块是不纯的，这里可以引入纯函数的概念：

> 对于相同的输入就有相同的输出，不依赖外部环境，也不改变外部环境。

符合上面描述的函数就可以称为**纯函数**，不符合就是不纯的，不纯就具有副作用的，是可能对外界造成影响的。我们通过代码示例来理解下：

```
// 函数内调用外部方法
import {isNumber} from 'lodash-es';
export function foo(obj) {
    return isNumber(obj);
}
// 直接使用全局对象
function goto(url) {
    location.href = url;
}
// 直接修改原型
Array.prototype.hello = () => 'hello';
```

上面几种方式的代码都是有副作用的代码，这样的代码在 Webpack 中因为并不知道代码内部究竟做了什么事情，所以不会被 Tree-Shaking 删除。那么怎么解决副作用呢？有两种方式：

1. 代码中消除副作用；
2. 配置`sideEffects`告诉 webpack 模块是安全的，不会带有副作用，可以放心优化。

代码中消除副作用

例如我们按照纯函数的定义，可以将需要用到的方法通过参数的方式传入：

```
// 函数内调用外部方法
export function foo(isNumber, obj) {
    return isNumber(obj);
}
// 直接使用全局对象
function goto(location, url) {
    location.href = url;
}
```

配置 `sideEffects`

Webpack 的项目中，可以在 `package.json` 中使用 `sideEffects` 来告诉 webpack 哪些文件中的代码具有副作用，从而对没有副作用的文件代码可以放心的使用 Tree-Shaking 进行优化。

```
// package.json
{
    "name": "tree-shaking-side-effect",
    "sideEffects": ["./src/utils.js"]
}
```

如果自己的项目是个类库或者工具库，需要发布给其他项目使用，并且项目是使用 ES6 Modules 编写的，没有副作用，那么可以在该项目 `package.json` 设置 `sideEffects:false` 来告诉使用该项目的 webpack 可以放心的对该项目进行 Tree-Shaking，而不必考虑副作用。

总结

Tree-Shaking 对前端项目来说可谓意义重大，是一个极致优化的理想世界，是前端进化的又一个终极理想。但是理想是美好的，现实是骨感的，真正发挥 Tree-Shaking 的强大作用，还需要我们在日常的代码中保持良好的开发习惯：

1. 要使用 Tree-Shaking 必然要保证引用的模块都是 ES6 规范的，很多工具库或者类库都提供了 ES6 语法的库，例如 lodash 的 ES6 版本是 `lodash-es`；
2. 按需引入模块，避免「一把梭」，例如我们要使用 lodash 的 `isNumber`，可以使用 `import isNumber from 'lodash-es/isNumber';`，而不是 `import {isNumber} from 'lodash-es'`；
3. 减少代码中的副作用代码。

> Tips：另外一些组件库，例如 AntDesign 和 ElementUI 这些组件库，本身自己开发了 Babel 的插件，通过插件的方式来按需引入模块，避免一股脑的引入全部组件。

> 本小节 Webpack 相关面试题：
>
> 1. 什么是 Tree-Shaking？
> 2. 怎么在 Webpack 中做 Tree-Shaking？
> 3. Webpack 中 Tree-Shaking 应该注意什么？

20 Webpack 优化之速度优化

22 为你准备了一份 Webpack 工程化最佳实践总结

欢迎在这里发表留言，作者筛选后可公开显示

目前暂无任何讨论