webpack 中的 treeShaking

treeShaking 是什么，为什么需要 treeShaking?

treeShaking 英文直译被称为 “树摇晃“。啥意思，为什么要进行树摇晃。暂时不懂没关系，接下来通过一个例子来说明为什么要 “树摇晃“ TreeShaking

接下来写一个最最普通的 webpack 配置（以下内容都是针对 Webpack4.0+）

index.js

```
import { show } from './utils';
show('Welcome to duyi!');
```

utils.js

```
export const addEvent = function (elem, type, event, flag = false) {
    if (elem.addEventListener) {
        elem.addEventListener(type, event, flag);
    } else if (elem.attachEvent) {
        elem.attachEvent('on' + type, event);
    } else {
        elem['on' + type] = event;
    }
}


export const show = function (value) {
    console.log(value);
}
```

以下是打包之后的结果简化版（webpack --mode development 开发环境）

main.js

```
// ...
/*!**********************!*\
  !*** ./src/utils.js ***!
  \**********************/
/*! exports provided: addEvent, show */
/***/ (function(module, __webpack_exports__, __webpack_require__) {


"use strict";
```

渡一教育 DUYI EDUCATION

```
eval("__webpack_require__.r(__webpack_exports__);\n/* harmony export (binding)
*/ __webpack_require__.d(__webpack_exports__, \"addEvent\", function()
{ return addEvent; });\n/* harmony export (binding) */
__webpack_require__.d(__webpack_exports__, \"show\", function() { return
show; });\nconst addEvent = function (elem, type, event, flag = false) {\r\n
if (elem.addEventListener) {\r\n        elem.addEventListener(type, event,
flag);\r\n    } else if (elem.attachEvent) {\r\n        elem.attachEvent('on'
+ type, event);\r\n    } else {\r\n        elem['on' + type] =
event;\r\n    }\r\n}\r\n\r\nconst show = function (value) {\r\n
console.log(value);\r\n}\n\n//# sourceURL=webpack:///./src/utils.js?");

/***/ })

/******/ });
```

我们能看出来，对于 utils.js 里面的内容，全部打包进去了（show, addEvent）。但实际上我们在 index.js 我们仅仅用了 show 方法。对于一个工具库来说，当他很大，里面有很多的工具方法，但是我们仅仅使用了一个的时候，在打包处理过程中，整个工具库都拿过来了。很明显，这种结果不是我们想要的。我们想要的是： **消除项目一些不必要的代码。**

这就涉及到今天的内容的 treeShaking。

同样是打包，我们在生产环境中进行打包看一下：

以下是打包之后的结果简化版（webpack --mode production 生产环境）

main.js

```
// ...
([function (e, t, r) {
  "use strict";
  r.r(t);
  var n;
  n = "Welcome to duyi!", console.log(n)
}])
```

很明显，对于无用的 addEvent 函数并没有引入，并且对已有函数内容进行简化。在生产环境中，大家看到的这种简化代码，就是 webpack 提供的 treeshaking 功能。

tree-shaking 可以理解为通过工具"摇"我们的 JS 文件，将其中用不到的代码"摇"掉，是一个性能优化的范畴。具体来说，在 webpack 项目中，有一个入口文件，相当于一棵树的主干，入口文件有很多依赖的模块，相当于树枝。实际情况中，虽然依赖了某个模块，

但其实只使用其中的某些功能。通过 tree-shaking，将没有使用的模块摇掉，这样来达到删除无用代码的目的。

tree-shaking 较早由 Rich_Harris 的 rollup 实现，后来，webpack2 也增加了 tree-shaking 的功能。其实在更早，google closure compiler 也做过类似的事情。

treeShaking 的作用与副作用

treeShaking 的本质是消除无用的 js 代码。无用代码消除在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）。

treeShaking 是 DCE 的一种新的实现，Javascript 同传统的编程语言不同的是，javascript 绝大多数情况需要通过网络进行加载，然后执行，加载的文件大小越小，整体执行时间更短，所以去除无用代码以减少文件体积，对 javascript 来说更有意义。

treeShaking 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，而 treeShaking 更关注宇消除没有用到的代码。webpack 的 treeShaking 还和传统意义上的 treeShaking 还不一样。后面文章会介绍一下 DCE 和 Tree-shaking。

主要目的： **用于前端优化，减少文件内容，减少网络请求时常，提升用户体验。**

副作用：

这么好用的东西，还有副作用么？这里的副作用是指，webpack 中的 treeShaking 是有问题的。具体问题是什么接着往下看。

utils.js

```
// 在 utils.js 中追加下面内容
export const Person = function () {
    function Person() {

    }
    Person.prototype.getName = function () { return this.name };
    return Person;
  }()
```

在 index.js 内容不变的情况下。看看打包结果有什么不一样？

精简后的打包文件

main.js

```
// ...
([function (e, t, n) {
    "use strict";
    n.r(t);
    ! function () {
        function e() {}
        e.prototype.getName = function () {
            return this.name
        }
    }();
    var r;
    r = "Welcome to duyi!", console.log(r)
}])
```

在之前的 utils.js 中导出了一个立即执行函数。尽管在 index.js 我们并没有引入。但是打包的时候，发现也打进去了。What are you 弄啥嘞？ 说好的 treeShaking 可以把无用的代码去掉，很显然 webpack 中针对 JS 的 treeShaking 是没有做到的，那又是为什么呢？

因为立即执行函数比较特殊，它在被解释时(JS 并非编译型的语言)就会被执行，Webpack 不做程序流分析，它不知道立即执行函数会做什么特别的事情，所以不会删除这部分代码。

副作用在我们项目中，也同样是频繁的出现。知道函数式编程的朋友都会知道这个名词。所谓模块(这里模块可称为一个函数)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。

符合上述就可以称为纯函数，不符合就是不纯的，是具有副作用的，是可能对外界造成影响的。

webpack 自身的 Tree-shaking 不能分析副作用的模块。以 lodash-es 这个模块来举个例子

utils.js

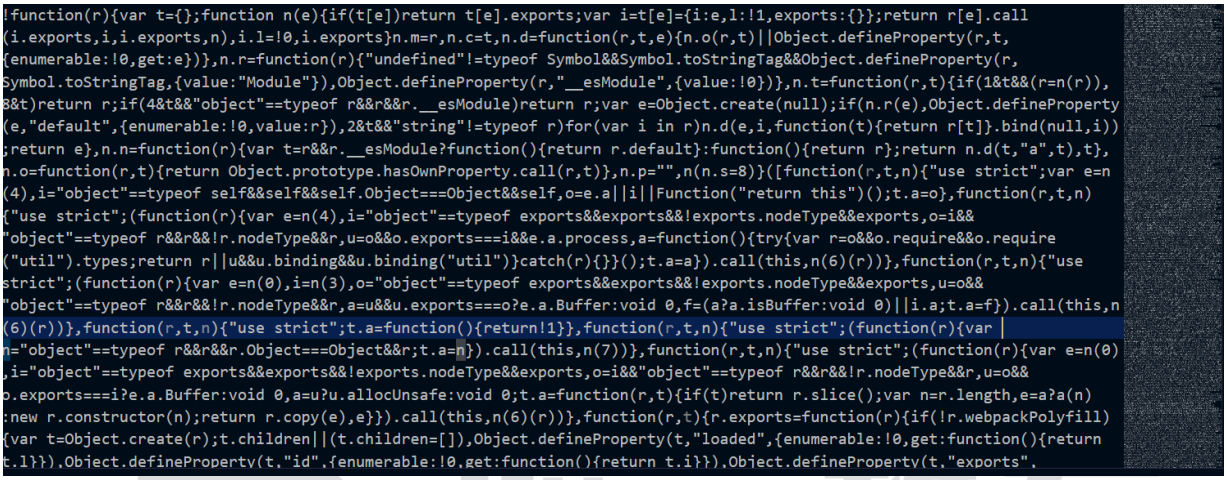
```
// ...
import lodash from 'lodash-es'
//...

const isNaN = function (value) {
    console.log(lodash.isNaN(value))
```

```
}
```

其他文件都不变的情况下，大家可以感受一下结果

就不给大家看代码，看一下压缩后的代码截图



```
!function(r){var t={};function n(e){if(t[e])return t[e].exports;var i=t[e]={i:e,l:!1,exports:{}};return r[e].call
(i.exports,i,i.exports,n),i.l=!0,i.exports}n.m=r,n.c=t,n.d=function(r,t,e){n.o(r,t)||Object.defineProperty(r,t,
{enumerable:!0,get:e})},n.r=function(r){"undefined"!=typeof Symbol&&Symbol.toStringTag&&Object.defineProperty(r,
Symbol.toStringTag,{value:"Module"}),Object.defineProperty(r,"__esModule",{value:!0})},n.t=function(r,t){if(1&t&&(r=n(r)),
8&t)return r;if(4&t&&"object"==typeof r&&r&&r.__esModule)return r;var e=Object.create(null);if(n.r(e),Object.defineProperty
(e,"default",{enumerable:!0,value:r}),2&t&&"string"!=typeof r)for(var i in r)n.d(e,i,function(t){return r[t]}.bind(null,i))
;return e},n.n=function(r){var t=r&&r.__esModule?function(){return r.default}:function(){return r};return n.d(t,"a",t),t},
n.o=function(r,t){return Object.prototype.hasOwnProperty.call(r,t)},n.p="",n(n.s=8)}([function(r,t,n){"use strict";var e=n
(4),i="object"==typeof self&&self&&self.Object===Object&&self,o=e.a||i||Function("return this")();t.a=o},function(r,t,n)
{"use strict";(function(r){var e=n(4),i="object"==typeof exports&&exports&&!exports.nodeType&&exports,o=i&&
"object"==typeof r&&r&&!r.nodeType&&r,u=o&&o.exports===i&&e.a.process,a=function(){try{var r=o&&o.require&&o.require
("util").types;return r||u&&u.binding&&u.binding("util")}catch(r){}}();t.a=a}).call(this,n(6)(r))},function(r,t,n){"use
strict";(function(r){var e=n(0),i=n(3),o="object"==typeof exports&&exports&&!exports.nodeType&&exports,u=o&&
"object"==typeof r&&r&&!r.nodeType&&r,a=u&&u.exports===o?e.a.Buffer:void 0,f=(a?a.isBuffer:void 0)||i.a;t.a=f}).call(this,n
(6)(r))},function(r,t,n){"use strict";t.a=function(){return!1}},function(r,t,n){"use strict";(function(r){var e=n(0)
n="object"==typeof r&&r&&r.Object===Object&&r;t.a=n}).call(this,n(7))},function(r,t,n){"use strict";(function(r){var e=n(0)
,i="object"==typeof exports&&exports&&!exports.nodeType&&exports,o=i&&"object"==typeof r&&r&&!r.nodeType&&r,u=o&&
o.exports===i?e.a.Buffer:void 0,a=u?u.allocUnsafe:void 0;t.a=function(r,t){if(t)return r.slice();var n=r.length,e=a?a(n)
:new r.constructor(n);return r.copy(e),e}}).call(this,n(6)(r))},function(r,t){r.exports=function(r){if(!r.webpackPolyfill)
{var t=Object.create(r);t.children||(t.children=[]),Object.defineProperty(t,"loaded",{enumerable:!0,get:function(){return
t.l}}),Object.defineProperty(t,"id",{enumerable:!0,get:function(){return t.i}}),Object.defineProperty(t,"exports",
```

```
Built at: 2019-03-26 14:27:32
  Asset    Size  Chunks             Chunk Names
main.js  86 KiB       0  [emitted]  main
Entrypoint main = main.js
[6] (webpack)/buildin/harmony-module.js 573 bytes {0} [built]
[7] (webpack)/buildin/global.js 472 bytes {0} [built]
[8] ./src/index.js + 611 modules 572 KiB {0} [built]
    | ./src/index.js 60 bytes [built]
    | ./src/utils.js 662 bytes [built]
    |     + 610 hidden modules
    + 28 hidden modules
```

main.js 86KB，可见这个结果是符合我们的预期的，因为 isNaN 函数的副作用，webpack 自身的 Tree-shaking 并没有检测到这里有没必要的模块。那该怎么解决呢？

解决办法还是用的，webpack 的插件系统是很强大的。

webpack-deep-scope-plugin

webpack-deep-scope-plugin 这个插件主要用于填充 webpack 自身 Tree-shaking 的不足，通过作用域分析来消除无用的代码。

小结：

Webpack Tree shaking 从 ES6 顶层模块开始分析，可以清除未使用的模块

Webpack Tree shaking 会对多层调用的模块进行重构，提取其中的代码，简化函数的调用结构

Webpack Tree shaking 不会清除立即调用函数表达式

如果要更好的使用 Webpack Tree shaking,请满足：

使用 ES2015(ES6)的模块

避免使用立即执行函数

总结

tree-shaking 对 web 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。理想是美好的，但目前还处在发展阶段，还比较困难，有各个方面的，甚至有目前看来无法完全解决的问题，但还是应该相信新技术能带来更好的前端世界。优化是一种态度，不因小而不为，不因艰而不攻。

渡一教育 DUYI EDUCATION