## Tree Shaking

Tree Shaking 翻译过来的意思就是“摇树”。伴随着摇树的动作，树上的枯树枝和树叶就会掉落下来。

我们这里要介绍的 Tree-shaking 也是同样的道理，不过通过 Tree-shaking “摇掉”的是代码中那些没有用到的部分，这部分没有用的代码更专业的说法应该叫作未引用代码（dead-code）。

Tree-shaking 最早是 Rollup 中推出的一个特性，Webpack 从 2.0 过后开始支持这个特性。

我们使用 Webpack 生产模式打包的优化过程中，就使用自动开启这个功能，以此来检测我们代码中的未引用代码，然后自动移除它们。

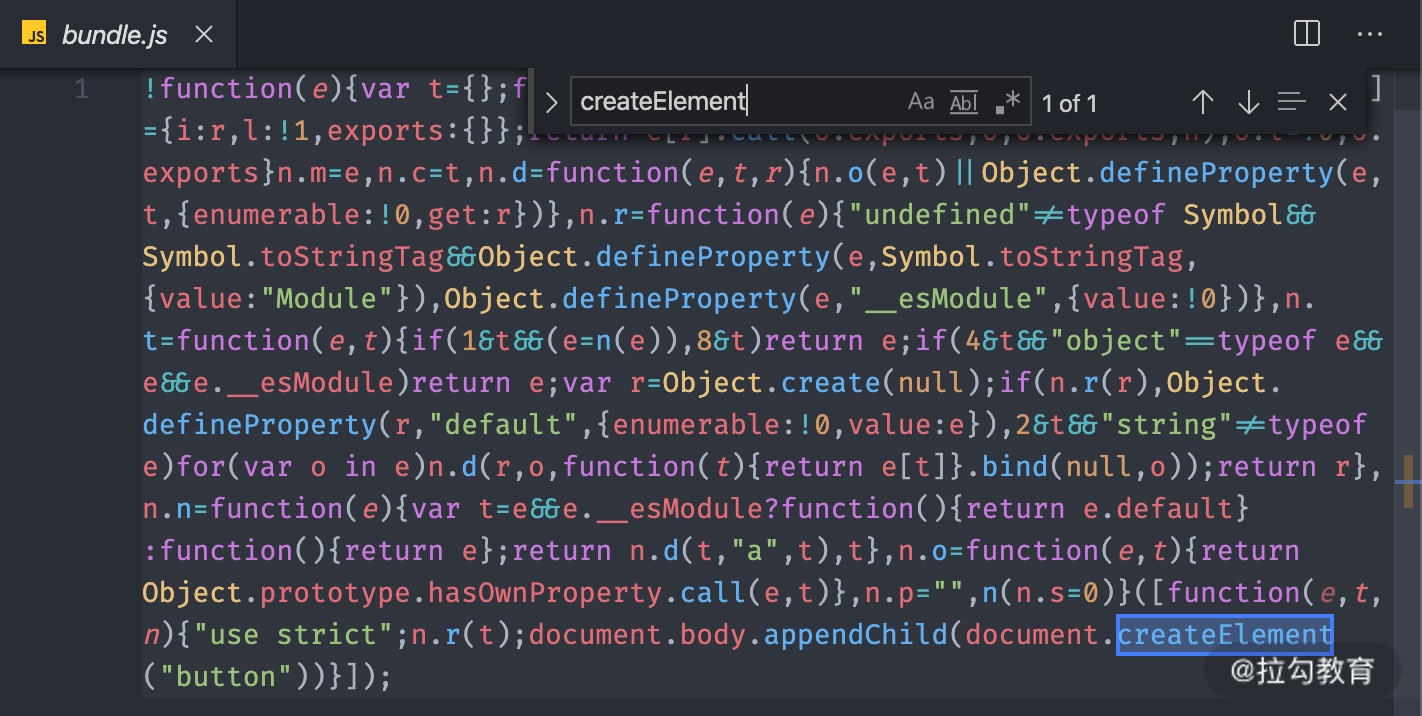
其中 Button 组件函数中，在 return 过后还有一个 console.log() 语句，很明显这句代码永远都不会被执行，所以这个 console.log() 就属于未引用代码。



我们打开命令行终端，这里我们尝试以 production 模式运行打包，具体命令如下：

$ npx webpack --mode=production

Webpack 的 Tree-shaking 特性在生产模式下会自动开启。打包完成以后我们打开输出的 bundle.js，具体结果如下：



通过搜索你会发现，components 模块中冗余的代码根本没有输出。这就是经过 Tree-shaking 处理过后的效果。

试想一下，如果我们在项目中引入 Lodash 这种工具库，大部分情况下我们只会使用其中的某几个工具函数，而其他没有用到的部分就是冗余代码。通过 Tree-shaking 就可以极大地减少最终打包后 bundle 的体积。

需要注意的是，Tree-shaking 并不是指 Webpack 中的某一个配置选项，而是一组功能搭配使用过后实现的效果，这组功能在生产模式下都会自动启用，所以使用生产模式打包就会有 Tree-shaking 的效果。

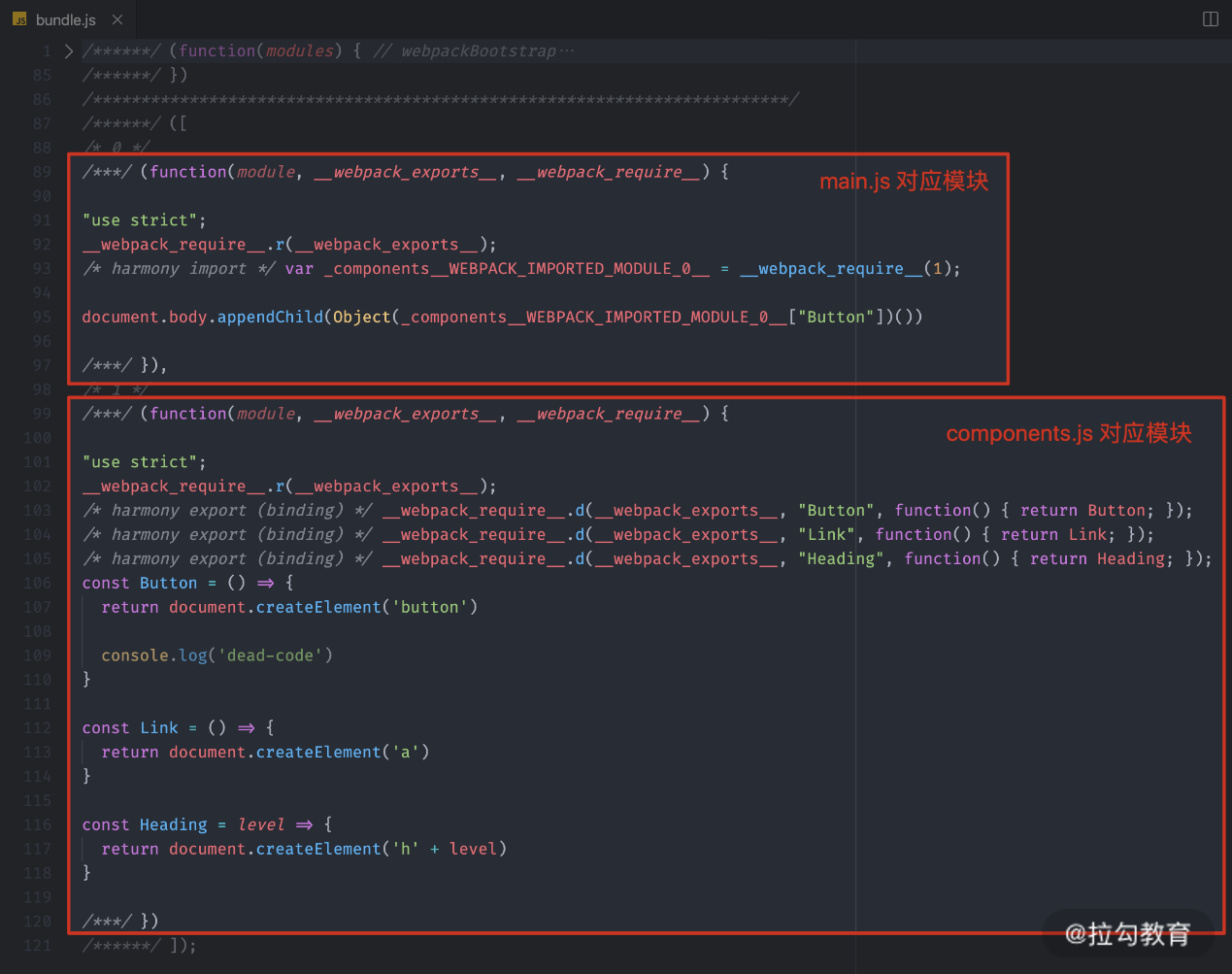
## 开启 Tree Shaking

由于目前官方文档中对于 Tree-shaking 的介绍有点混乱，所以我们这里再来介绍一下在其他模式下，如何一步一步手动开启 Tree-shaking。通过这个过程，还可以顺便了解 Tree-shaking 的工作过程和 Webpack 其他的一些优化功能。

这里还是上述的案例结构，我们再次运行 Webpack 打包，不过这一次我们不再使用 production 模式，而是使用 none，也就是不开启任何内置功能和插件，具体命令如下：

$ npx webpack --mode=none

打包完成过后，我们再次找到输出的 bundle.js 文件，具体结果如下：



这里的打包结果跟我们在第二讲中分析的是一样的，源代码中的一个模块对应这里的一个函数。

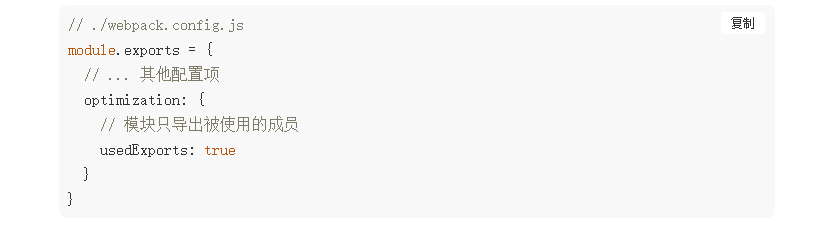
我们这里注意一下 components 对应的这个模块，虽然外部没有使用这里的 Link 函数和 Heading 函数，但是仍然导出了它们，具体如下图所示：

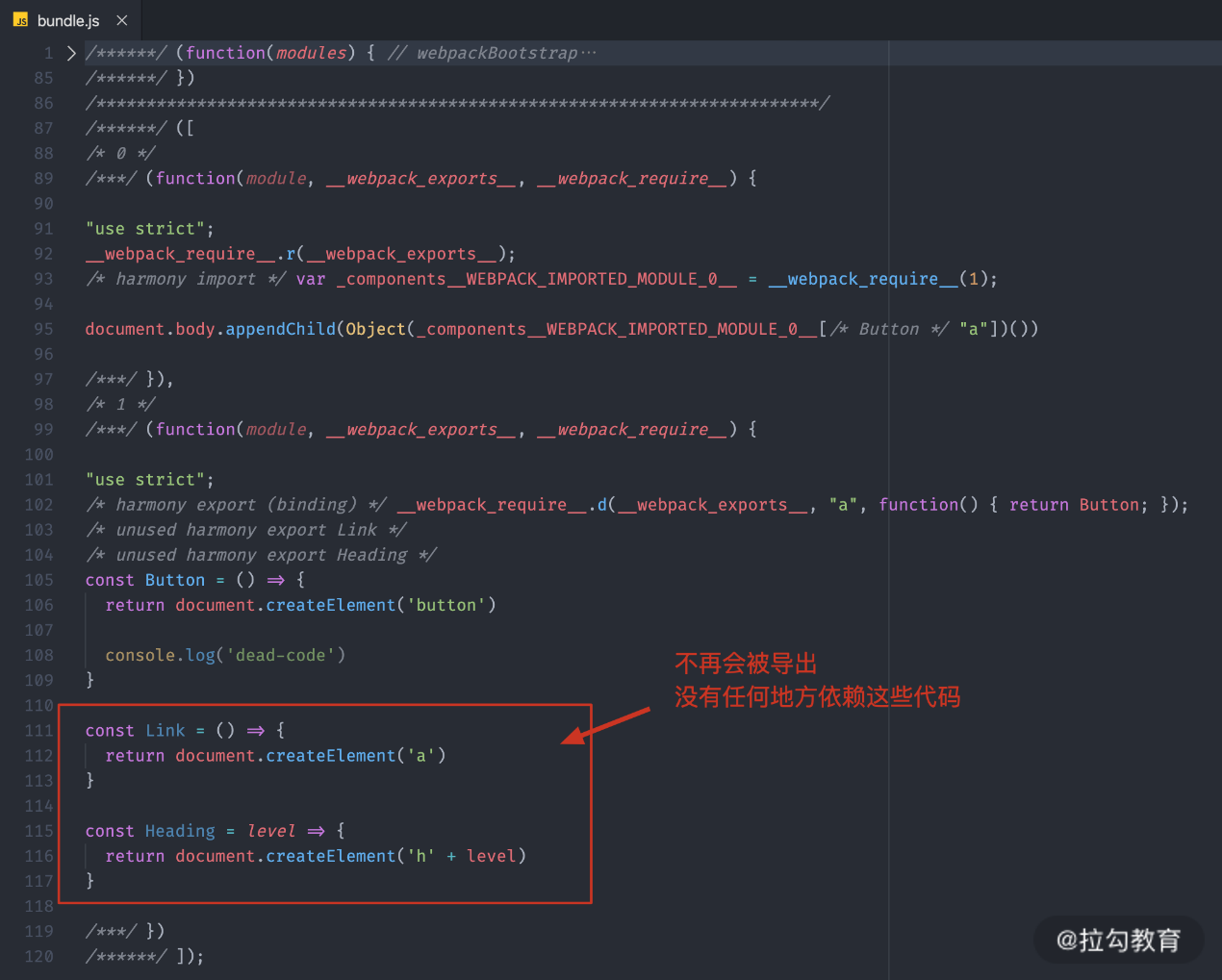


显然这种导出是没有任何意义的。

明确目前打包结果的状态过后，我们打开 Webpack 的配置文件，在配置对象中添加一个 optimization 属性，这个属性用来集中配置 Webpack 内置优化功能，它的值也是一个对象。

在 optimization 对象中我们可以先开启一个 usedExports 选项，表示在输出结果中只导出外部使用了的成员，具体配置代码如下：

配置完成后，重新打包，然后我们再来看一下输出的 bundle.js，具体结果如下图：

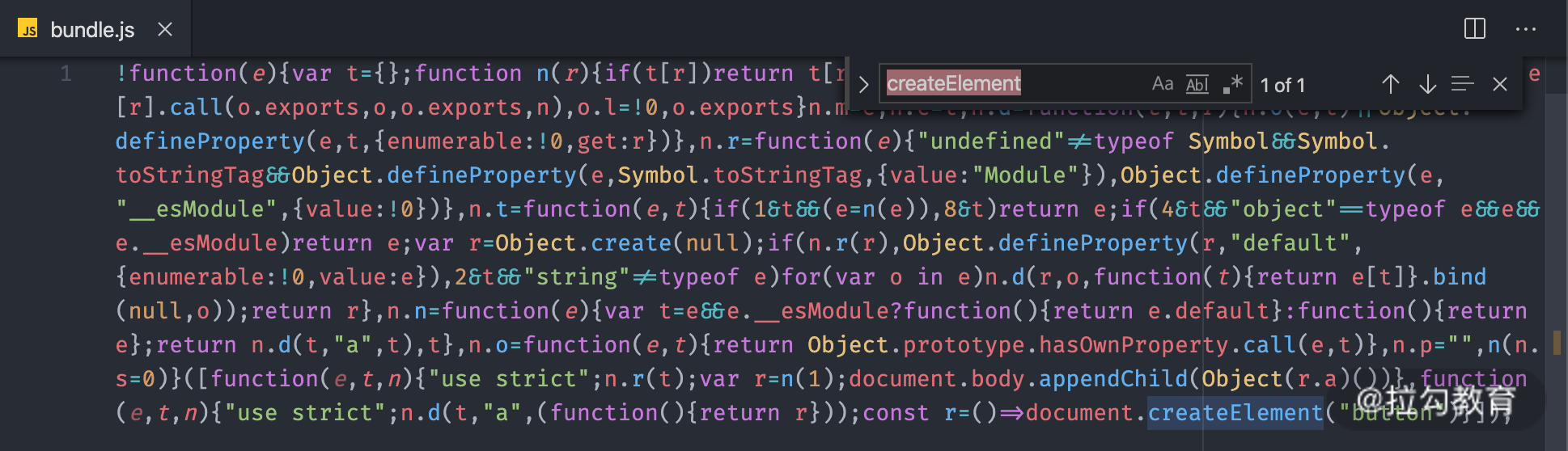


此时你会发现 components 模块所对应的函数，就不再导出 Link 和 Heading 这两个函数了，那它们对应的代码就变成了未引用代码。而且如果你使用的是 VS Code，会发现 VS Code 将这两个函数名的颜色变淡了，这是为了表示它们未被引用。

对于这种未引用代码，如果我们开启压缩代码功能，就可以自动压缩掉这些没有用到的代码。

我们可以回到配置文件中，尝试在 optimization 配置中开启 minimize，具体配置如下：

然后再次回到命令行重新运行打包，具体结果如下图所示：



仔细查看打包结果，你会发现，Link 和 Heading 这些未引用代码都被自动移除了。

这就是 Tree-shaking 的实现，整个过程用到了 Webpack 的两个优化功能：然后回到命令行终端再次运行打包。那此时 bundle.js 中就不再是一个模块对应一个函数了，而是把所有的模块都放到了一个函数中，具体结果如下：

这个特性又被称为 Scope Hoisting，也就是作用域提升，它是 Webpack 3.0 中添加的一个特性。

如果再配合 minimize 选项，打包结果的体积又会减小很多。

结合 babel-loader 的问题

因为早期的 Webpack 发展非常快，那变化也就比较多，所以当我们去找资料时，得到的结果不一定适用于当前我们所使用的版本。而 Tree-shaking 的资料更是如此，很多资料中都表示“为 JS 模块配置 babel-loader，会导致 Tree-shaking 失效”。

针对这个问题，这里我统一说明一下：

首先你需要明确一点：Tree-shaking 实现的前提是 ES Modules，也就是说：最终交给 Webpack 打包的代码，必须是使用 ES Modules 的方式来组织的模块化。

为什么这么说呢？

我们都知道 Webpack 在打包所有的模块代码之前，先是将模块根据配置交给不同的 Loader 处理，最后再将 Loader 处理的结果打包到一起。

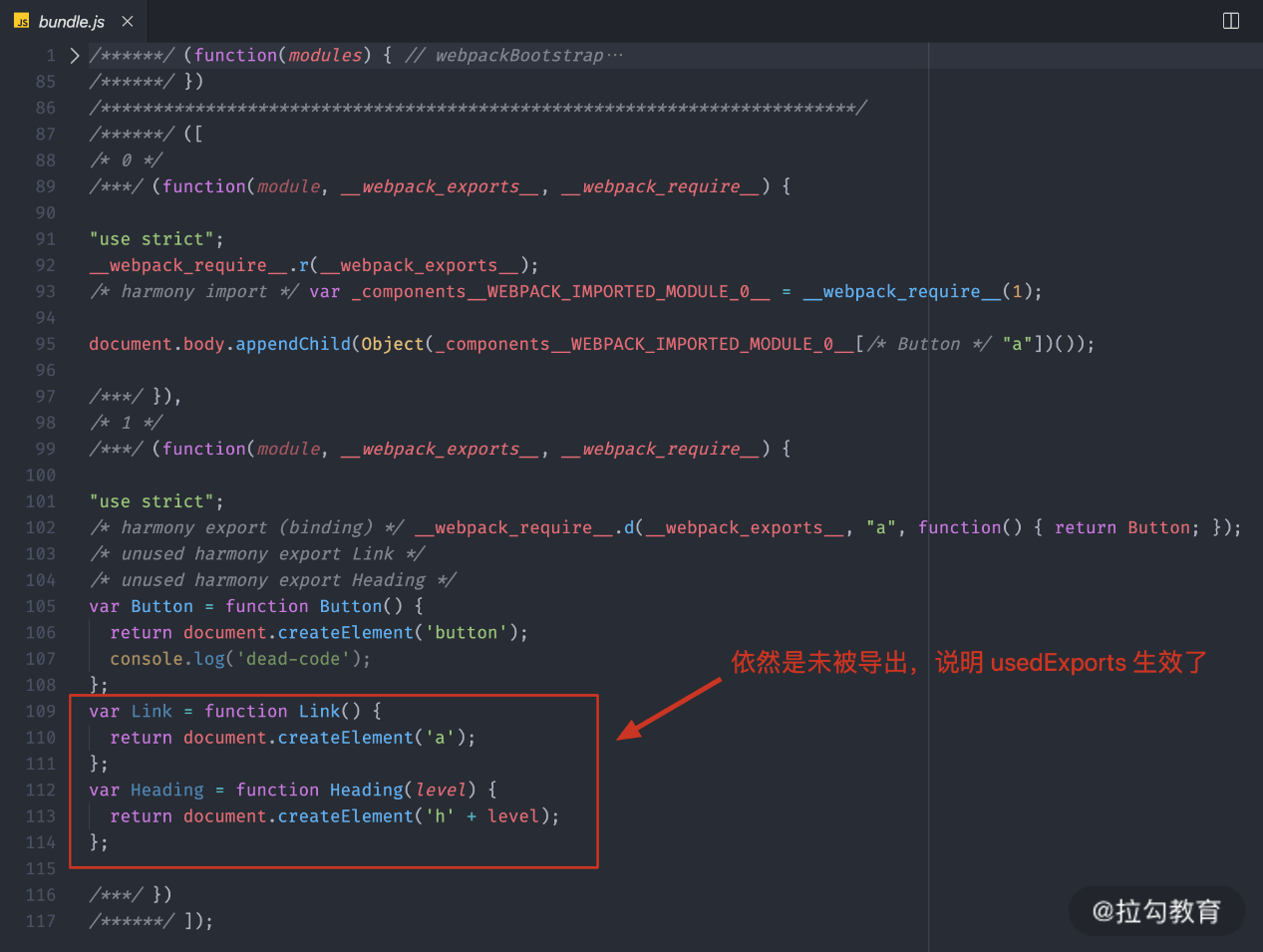
很多时候，我们为了更好的兼容性，会选择使用 babel-loader 去转换我们源代码中的一些 ECMAScript 的新特性。而 Babel 在转换 JS 代码时，很有可能处理掉我们代码中的 ES Modules 部分，把它们转换成 CommonJS 的方式，如下图所示：



当然了，Babel 具体会不会处理 ES Modules 代码，取决于我们有没有为它配置使用转换 ES Modules 的插件。

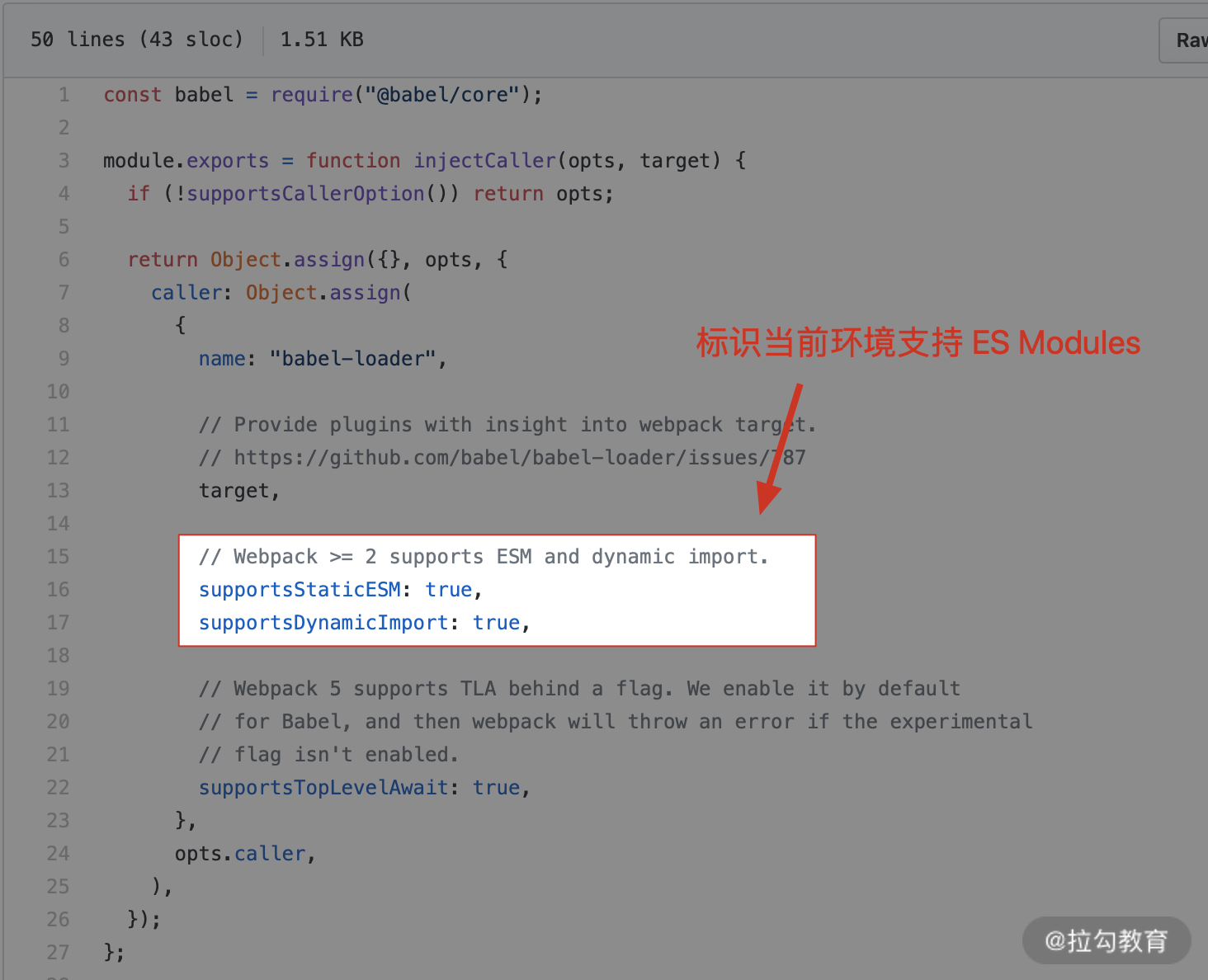
很多时候，我们为 Babel 配置的都是一个 preset（预设插件集合），而不是某些具体的插件。例如，目前市面上使用最多的 @babel/preset-env，这个预设里面就有转换 ES Modules 的插件。所以当我们使用这个预设时，代码中的 ES Modules 部分就会被转换成 CommonJS 方式。那 Webpack 再去打包时，拿到的就是以 CommonJS 方式组织的代码了，所以 Tree-shaking 不能生效。

那我们这里具体来尝试一下。为了可以更容易分辨结果，我们只开启 usedExports，完整配置如下：

配置完成过后，我们打开命令行终端，运行 Webpack 打包命令，然后再找到 bundle.js，具体结果如下：

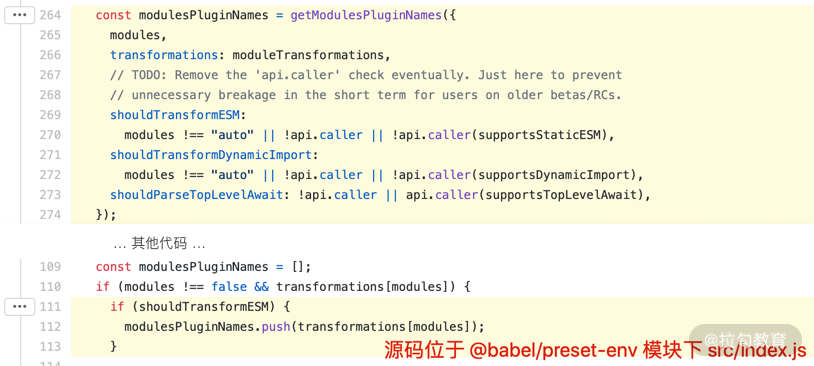
仔细查看你会发现，结果并不是像刚刚说的那样，这里 usedExports 功能仍然正常工作了，此时，如果我们压缩代码，这些未引用的代码依然会被移除。这也就说明 Tree-shaking 并没有失效。

那到底是怎么回事呢？为什么很多资料都说 babel-loader 会导致 Tree-shaking 失效，但当我们实际尝试后又发现并没有失效？

其实，这是因为在最新版本（8.x）的 babel-loader 中，已经自动帮我们关闭了对 ES Modules 转换的插件，你可以参考对应版本 babel-loader 的源码，核心代码如下：

通过查阅 babel-loader 模块的源码，我们发现它已经在 injectCaller 函数中标识了当前环境支持 ES Modules。

然后再找到我们所使用的 @babal/preset-env 模块源码，部分核心代码如下：



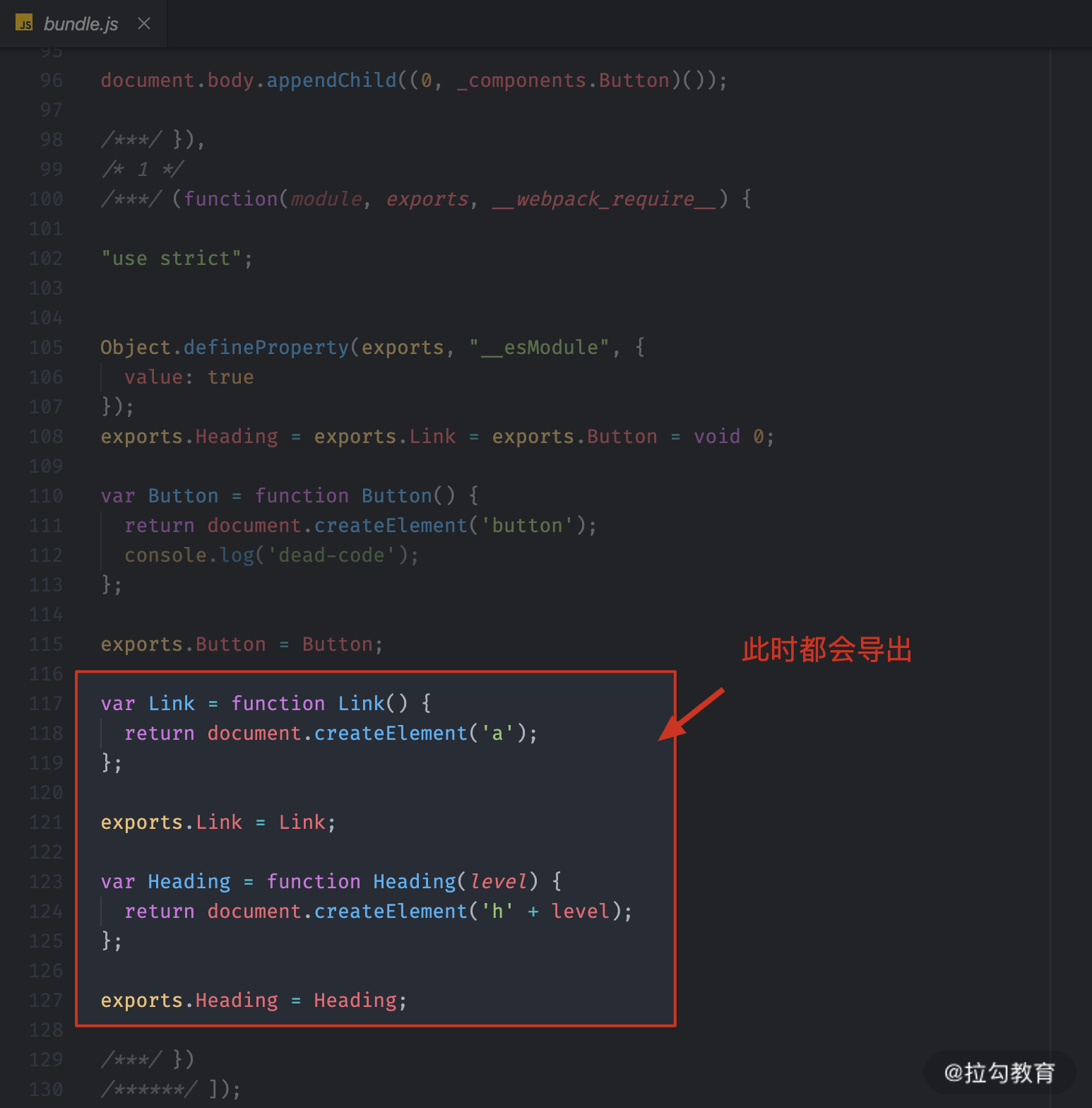
在这个模块中，根据环境标识自动禁用了对 ES Modules 的转换插件，所以经过 babel-loader 处理后的代码默认仍然是 ES Modules，那 Webpack 最终打包得到的还是 ES Modules 代码，Tree-shaking 自然也就可以正常工作了。

我们也可以在 babel-loader 的配置中强制开启 ES Modules 转换插件来试一下，具体配置如下：

给 Babel preset 添加配置的方式比较特别，这里很多人都会配错，一定要注意。它需要把预设数组中的成员定义成一个数组，然后这个数组中的第一个成员就是所使用的 preset 的名称，第二个成员就是给这个 preset 定义的配置对象。

我们在这个对象中将 modules 属性设置为 "commonjs"，默认这个属性是 auto，也就是根据环境判断是否开启 ES Modules 插件，我们设置为 commonjs 就表示我们强制使用 Babel 的 ES Modules 插件把代码中的 ES Modules 转换为 CommonJS。

完成以后，我们再次打开命令行终端，运行 Webpack 打包。然后找到 bundle.js，结果如下：



此时，你就会发现 usedExports 没法生效了。即便我们开启压缩代码，Tree-shaking 也会失效。

总结一下，这里通过实验发现，最新版本的 babel-loader 并不会导致 Tree-shaking 失效。如果你不确定现在使用的 babel-loader 会不会导致这个问题，最简单的办法就是在配置中将 @babel/preset-env 的 modules 属性设置为 false，确保不会转换 ES Modules，也就确保了 Tree-shaking 的前提。

另外，我们刚刚探索的过程也值得你仔细再去琢磨一下，通过这样的探索能够帮助你了解很多背后的原因，做到“知其然，知其所以然”。

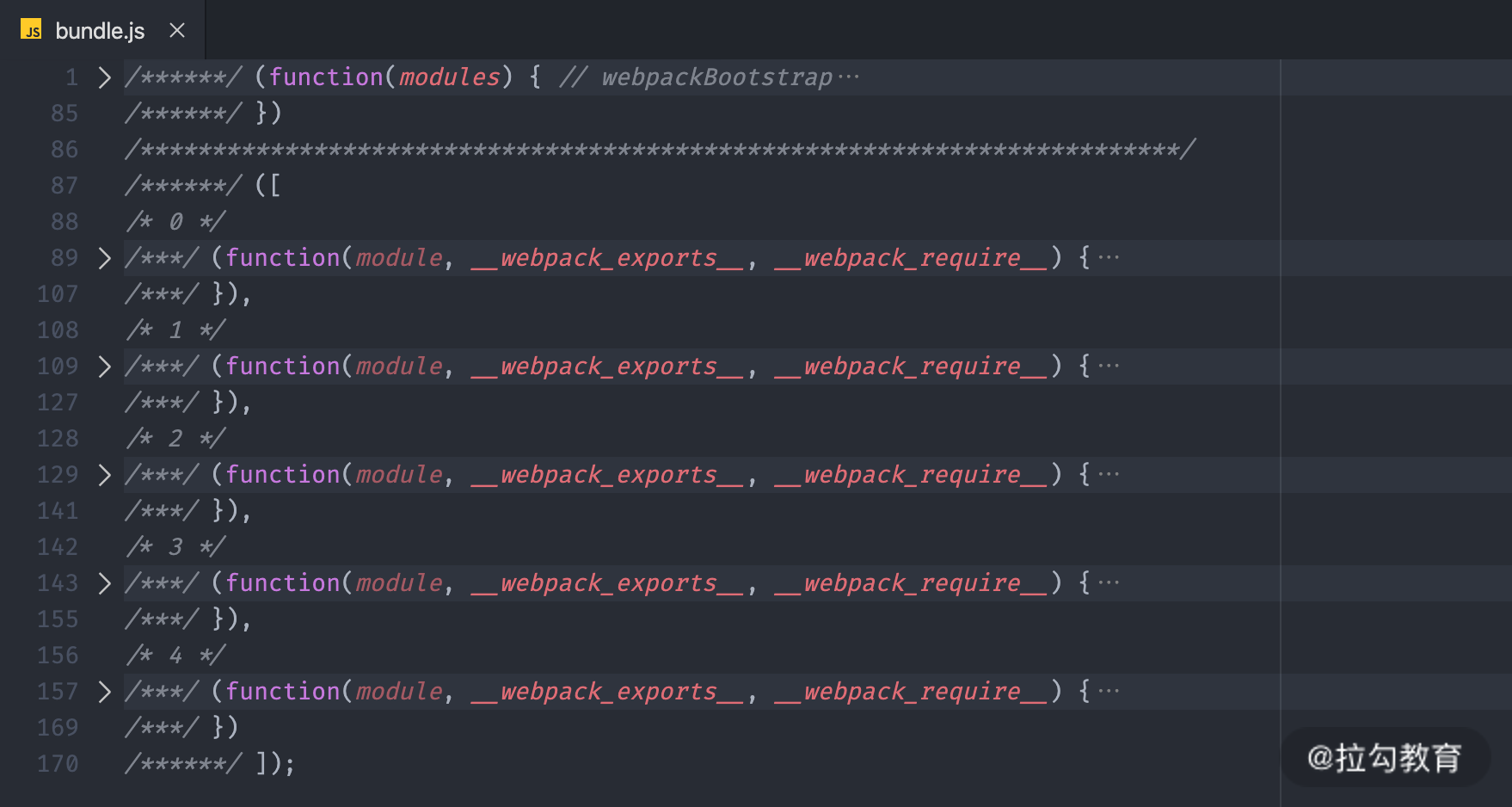
## sideEffects

Webpack 4 中新增了一个 sideEffects 特性，它允许我们通过配置标识我们的代码是否有副作用，从而提供更大的压缩空间。

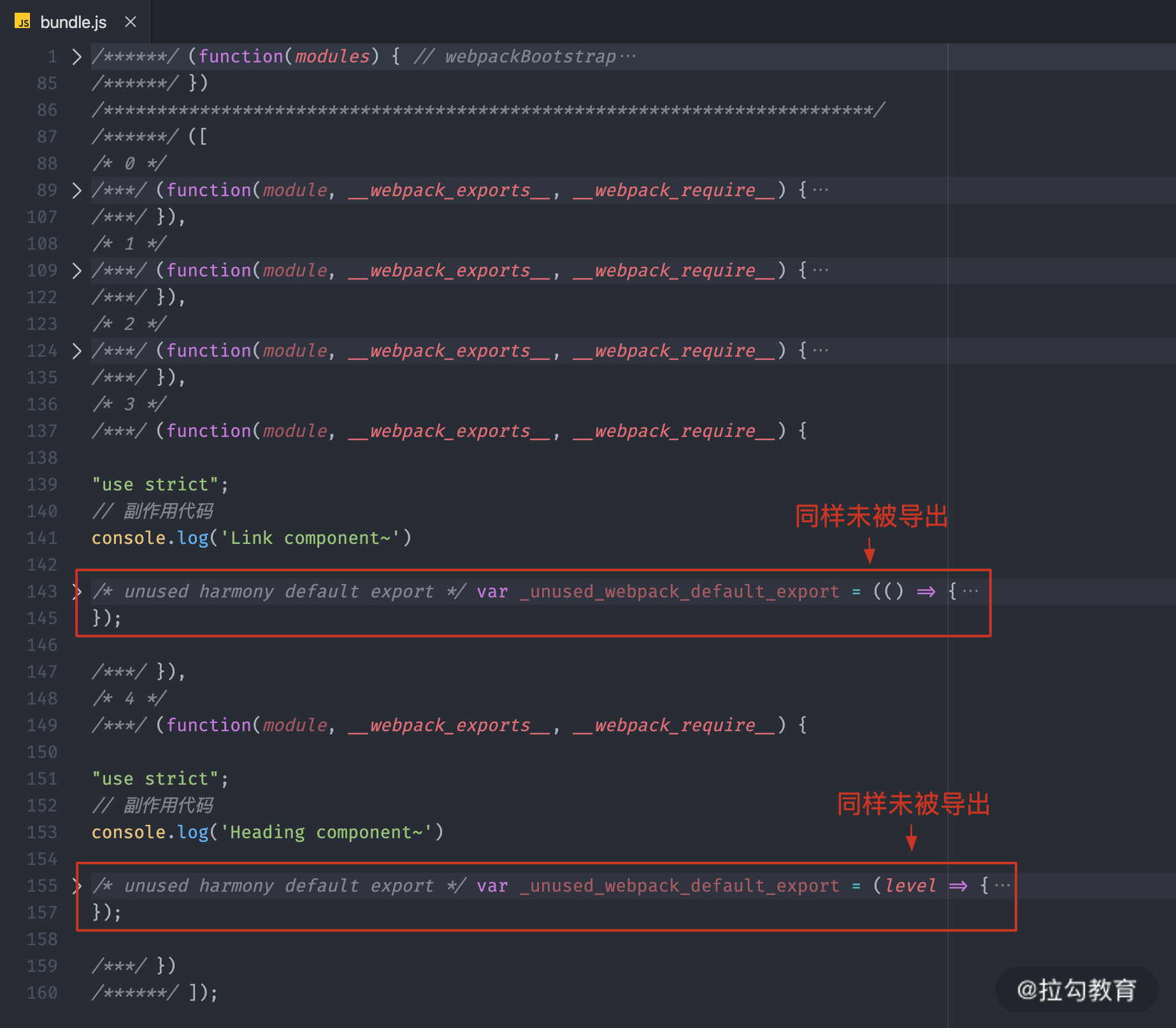
TIPS：模块的副作用指的就是模块执行的时候除了导出成员，是否还做了其他的事情。

这个特性一般只有我们去开发一个 npm 模块时才会用到。因为官网把对 sideEffects 特性的介绍跟 Tree-shaking 混到了一起，所以很多人误认为它们之间是因果关系，其实它们没有什么太大的关系。

我们先把 sideEffects 特性本身的作用弄明白，你就更容易理解为什么说它跟 Tree-shaking 没什么关系了。

我们打开命令行终端，尝试运行打包，打包完成过后找到打包结果，具体结果如下：

根据打包结果发现，所有的组件模块都被打包进了 bundle.js。

此时如果我们开启 Tree-shaking 特性（只设置 useExports），这里没有用到的导出成员其实最终也可以被移除，打包效果如下：

但是由于这些成员所属的模块中有副作用代码，所以就导致最终 Tree-shaking 过后，这些模块并不会被完全移除。

可能你会认为这些代码应该保留下来，而实际情况是，这些模块内的副作用代码一般都是为这个模块服务的，例如这里我添加的 console.log，就是希望表示一下当前这个模块被加载了。但是最终整个模块都没用到，也就没必要留下这些副作用代码了。

所以说，Tree-shaking 只能移除没有用到的代码成员，而想要完整移除没有用到的模块，那就需要开启 sideEffects 特性了。

## sideEffects 作用





完成以后我们再次运行打包，然后同样找到打包输出的 bundle.js 文件，结果如下：



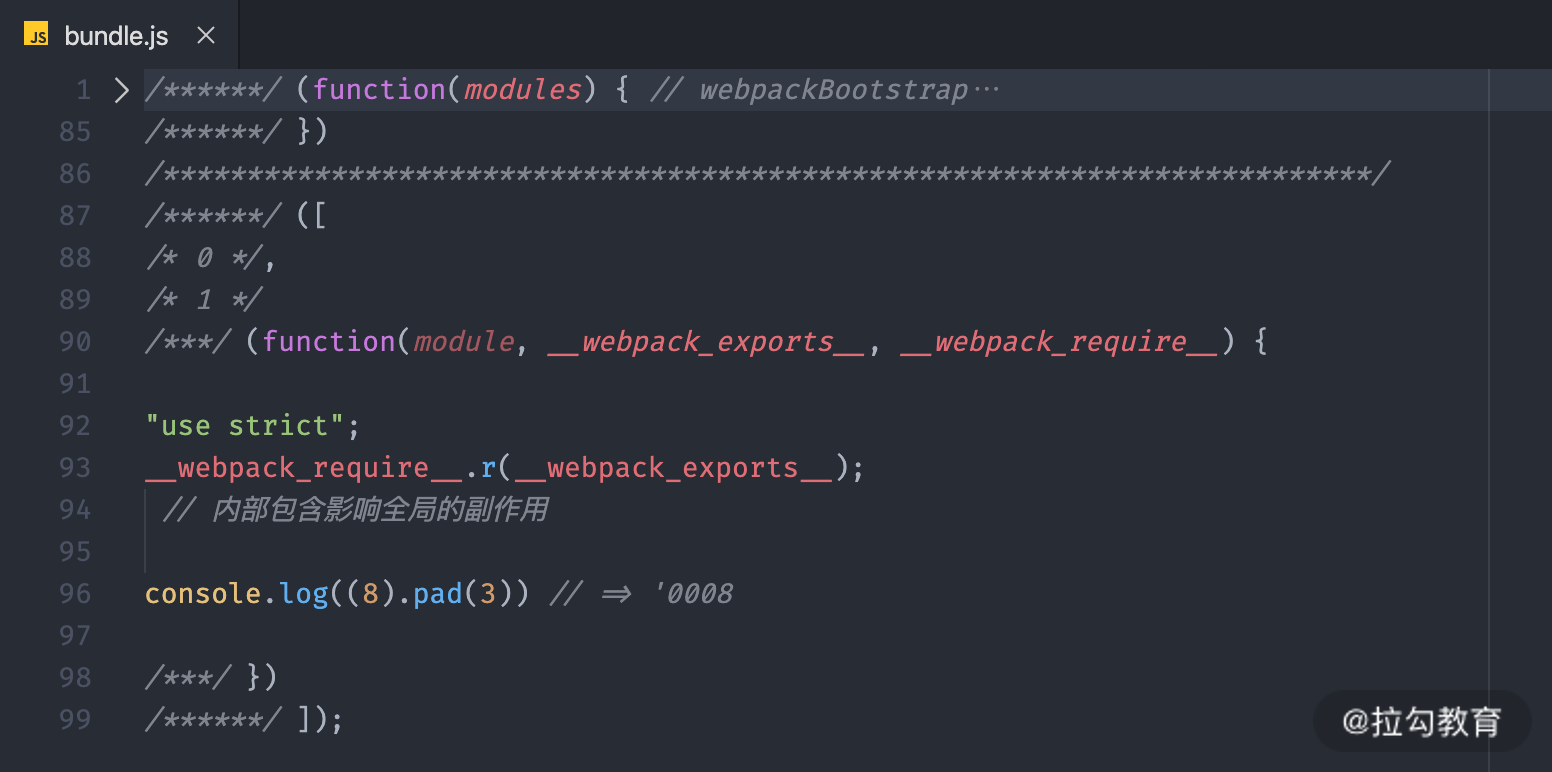
此时那些没有用到的模块就彻底不会被打包进来了。那这就是 sideEffects 的作用。

这里设置了两个地方：

webpack.config.js 中的 sideEffects 用来开启这个功能；

package.json 中的 sideEffects 用来标识我们的代码没有副作用。

目前很多第三方的库或者框架都已经使用了 sideEffects 标识，所以我们再也不用担心为了一个小功能引入一个很大体积的库了。例如，某个 UI 组件库中只有一两个组件会用到，那只要它支持 sideEffects，你就可以放心大胆的直接用了。



我们看到，对 Number 的扩展模块并不会打包进来。缺少了对 Number 的扩展操作，我们的代码再去运行的时候，就会出现错误。这种扩展的操作属于对全局产生的副作用。

这种基于原型的扩展方式，在很多 Polyfill 库中都会大量出现，比较常见的有 es6-promise，这种模块都属于典型的副作用模块。

除此之外，我们在 JS 中直接载入的 CSS 模块，也都属于副作用模块，同样会面临这种问题。

## 写在最后

最后我们来总结一下，今天介绍到了两个 Webpack 中的高级特性，分别是 Tree-shaking 和 sideEffects。

Tree-shaking 的本身没有太多需要你理解和思考的地方，你只需要了解它的效果，以及相关的配置即可。

而 sideEffects 可能需要你花点时间去理解一下，重点就是想明白哪些副作用代码是可以随着模块的移除而移除，哪些又是不可以移除的。总结下来其实也很简单：对全局有影响的副作用代码不能移除，而只是对模块有影响的副作用代码就可以移除。

总之不管是 Tree-shaking 还是 sideEffects，我个人认为，它们都是为了弥补 JavaScript 早期在模块系统设计上的不足。随着 Webpack 这类技术的发展，JavaScript 的模块化确实越来越好用，也越来越合理。

除此之外，我还想强调一点，当你对这些特性有了一定的了解之后，就应该意识到：尽可能不要写影响全局的副作用代码。