

# webpack 打包公共库

如果我们打包的目的是生成一个供别人使用的库，那么可以使用 `output.library` 来指定库的名称，库的名称支持占位符和普通字符串：

```
module.exports = {
  output: {
    library: 'myLib' // '[name]'
  }
};
```

## `output.libraryTarget`

使用 `output.library` 确定了库的名称之后，还可以使用 `output.libraryTarget` 指定库打包出来的规范，`output.libraryTarget` 取值范围

为：`var`、`assign`、`this`、`window`、`global`、`commonjs`、`commonjs2`、`commonjs-module`、`amd`、`umd`、`umd2`、`jsonp`，默认是 `var`，下面通过打包后的代码不同，来看下差别。

```
// var config
{
  output: {
    library: 'myLib',
    filename: 'var.js',
    libraryTarget: 'var'
  }
}
// output
var myLib = (function(modules) {})(
  './src/index.js': function(module, exports) {}
);
// =====
// assign config
{
  output: {
    library: 'myLib',
    filename: 'assign.js',
    libraryTarget: 'assign'
  }
}
// output: 少了个 var
myLib = (function(modules) {})(
  './src/index.js': function(module, exports) {}
);
// =====
// this config
```

```

{
  output: {
    library: 'myLib',
    filename: 'this.js',
    libraryTarget: 'this'
  }
}
// output
this["myLib"] = (function(modules) {})(
  './src/index.js': function(module, exports) {}
);
// =====
// window config
{
  output: {
    library: 'myLib',
    filename: 'window.js',
    libraryTarget: 'window'
  }
}
// output
window["myLib"] = (function(modules) {})(
  './src/index.js': function(module, exports) {}
);

// =====
// global config
{
  output: {
    library: 'myLib',
    filename: 'global.js',
    libraryTarget: 'global'
  }
}
// output: 注意 target=node 的时候才是 global, 默认 target=web下global 为 window
window["myLib"] = (function(modules) {})(
  './src/index.js': function(module, exports) {}
);
// =====
// commonjs config
{
  output: {
    library: 'myLib',
    filename: 'commonjs.js',
    libraryTarget: 'commonjs'
  }
}
// output
exports["myLib"] = (function(modules) {})(

```

```

    './src/index.js': function(module, exports) {}
  });
  // =====
  // amd config
  {
    output: {
      library: 'myLib',
      filename: 'amd.js',
      libraryTarget: 'amd'
    }
  }
  // output
  define('myLib', [], function() {
    return (function(modules) {})(
      './src/index.js': function(module, exports) {}
    );
  });

  // =====
  // umd config
  {
    output: {
      library: 'myLib',
      filename: 'umd.js',
      libraryTarget: 'umd'
    }
  }
  // output
  (function webpackUniversalModuleDefinition(root, factory) {
    if (typeof exports === 'object' && typeof module === 'object')
      module.exports = factory();
    else if (typeof define === 'function' && define.amd) define([], factory);
    else if (typeof exports === 'object') exports['myLib'] = factory();
    else root['myLib'] = factory();
  })(window, function() {
    return (function(modules) {})(
      './src/index.js': function(module, exports) {}
    );
  });

  // =====
  // commonjs2 config
  {
    output: {
      library: 'myLib',
      filename: 'commonjs2.js',
      libraryTarget: 'commonjs2'
    }
  }
  // output

```

```

module.exports = (function(modules) {})(
  './src/index.js': function(module, exports) {}
);
// =====
// umd2 config
{
  output: {
    library: 'myLib',
    filename: 'umd2.js',
    libraryTarget: 'umd2'
  }
}
// output
(function webpackUniversalModuleDefinition(root, factory) {
  if (typeof exports === 'object' && typeof module === 'object')
module.exports = factory();
  else if (typeof define === 'function' && define.amd) define([], factory);
  else if (typeof exports === 'object') exports['myLib'] = factory();
  else root['myLib'] = factory();
})(window, function() {
  return (function(modules) {})(
    './src/index.js': function(module, exports) {
    }
  );
});
// =====
// commonjs-module config
{
  output: {
    library: 'myLib',
    filename: 'commonjs-module.js',
    libraryTarget: 'commonjs-module'
  }
}
// =====
// output
module.exports = (function(modules) {})(
  './src/index.js': function(module, exports) {}
);
// =====
// jsonp config
{
  output: {
    library: 'myLib',
    filename: 'jsonp.js',
    libraryTarget: 'jsonp'
  }
}
// output

```

```
myLib((function(modules) {}))({
  './src/index.js': function(module, exports) {}
}));
```

注意： `libraryTarget=global` 的时候，如果 `target=node` 才是 `global`，默认 `target=web` 下 `global` 为 `window`，保险起见可以使用 `this`。

下面介绍下跟 `output` 输出相关的三个配置项： `externals`， `target` 和 `devtool`

## externals

`externals` 配置项用于去除输出的打包文件中依赖的某些第三方 `js` 模块（例如 `jquery`， `vue` 等），减小打包文件的体积。该功能通常在开发自定义 `js` 库（`library`）的时候用到，用于去除自定义 `js` 库依赖的其他第三方 `js` 模块。这些被依赖的模块应该由使用者提供，而不应该包含在 `js` 库文件中。例如开发一个 `jQuery` 插件或者 `Vue` 扩展，不需要把 `jQuery` 和 `Vue` 打包进我们的 `bundle`，引入库的方式应该交给使用者。

所以，这里就有个重要的问题，使用者应该怎么提供这些被依赖的模块给我们的 `js` 库（`library`）使用呢？这就要看我们的 `js` 库的导出方式是什么，以及使用者采用什么样的方式使用我们的库。例如：

js library 导出方式	output.libraryTarget	使用者引入方式	使用者提供给被依赖模块的方式
默认的导出方式	<code>output.libraryTarget='var'</code>	只能以 <code>&lt;script&gt;</code> 标签的形式引入我们的库	只能以全局变量的形式提供这些被依赖的模块
<code>commonjs</code>	<code>output.libraryTarget='commonjs'</code>	只能按照 <code>commonjs</code> 的规范引入我们的库	被依赖模块需要按照 <code>commonjs</code> 规范引入
<code>amd</code>	<code>output.libraryTarget='amd'</code>	只能按照 <code>amd</code> 规范引入	被依赖模块需要按照 <code>amd</code> 规范引入
<code>umd</code>	<code>output.libraryTarget='umd'</code>	可以用 <code>&lt;script&gt;</code> 、 <code>commonjs</code> 、 <code>amd</code> 引入	被依赖模块需要按照对应方式引入

如果不是在开发一个 `js` 库，即没有设置 `output.library`， `output.libraryTarget` 等配置信息，那么我们生成的打包文件只能以 `<script>` 标签的方式在页面中引入，因此那些被去除的依赖模块也只能以全局变量的方式引入。

## target

在项目开发中，我们不仅仅是开发 `web` 应用，还可能开发的是 `Node.js` 服务应用、或者 `electron` 这类跨平台桌面应用，这时候因为对应的宿主环境不同，所以在构建的时候需要特殊处理。`webpack` 中可以通过设置 `target` 来指定构建的目标（`target`）。

```
module.exports = {
  target: 'web' // 默认是 web，可以省略
};
```

`target` 的值有两种类型：string 和 function。

`string` 类型支持下面的七种：

- `web`：默认，编译为类浏览器环境里可用；
- `node`：编译为类 Node.js 环境可用（使用 Node.js require 加载 chunk）；
- `async-node`：编译为类 Node.js 环境可用（使用 fs 和 vm 异步加载分块）；
- `electron-main`：编译为 Electron 主进程；
- `electron-renderer`：编译为 Electron 渲染进程；
- `node-webkit`：编译为 Webkit 可用，并且使用 jsonp 去加载分块。支持 Node.js 内置模块和 nw.gui 导入（实验性质）；
- `webworker`：编译成一个 WebWorker。

后面章节介绍 webpack 特殊项目类型配置的时候还会介绍 `target` 相关的用法。

除了 `string` 类型，`target` 还支持 `function` 类型，这个函数接收一个 `compiler` 作为参数，如下面代码可以用来增加插件：

```
const webpack = require('webpack');

const options = {
  target: compiler => {
    compiler.apply(new webpack.JsonpTemplatePlugin(options.output), new
webpack.LoaderTargetPlugin('web'));
  }
};
```

## demo

---

- 创建一个library库项目
- webpack打包
  - 打包出两个文件，压缩和未压缩
- 发布到npm
- 项目使用

## 创建公共库：library-ouyi

```
// 目录结构
dist
node_modules
src
  --index.js
index.js
package.json
webpack.config.js
```

## 实现库的逻辑

```
// ./src/index.js

export default function add(a, b) {
  return a + b;
}
```

## 实现打包配置

```
// ./webpack.config.js
const TerserPlugin = require("terser-webpack-plugin"); // @4.2.0
module.exports = {
  entry: {
    "add-number": "./src/index.js",
    "add-number.min": "./src/index.js",
  },
  output: {
    filename: "[name].js",
    library: "addNumber",
    libraryTarget: "umd",
    libraryExport: "default",
  },
  mode: "none",
  optimization: {
    minimize: true,
    minimizer: [
      new TerserPlugin({
        test: /\.min\.js$/,
      }),
    ],
  },
}
```

```
};
```

这里推荐 [terser-webpack-plugin](#)，它是 Webpack 官方维护的插件，使用[terser](#)来压缩 JavaScript 代码。UglifyJS 在压缩 ES5 方面做的很优秀，但是随着 ES6 语法的普及，UglifyJS 在 ES6 代码压缩上做的不够好，所以有了 `uglify-es` 项目，但是之后 `uglify-es` 项目不在维护了，terser 是从 `uglify-es` 项目拉的一个分支，来继续维护。

## Npm publish

- npm 注册账号
- npm login 登录
- 注意npm 源

```
npm config set registry https://registry.npm.taobao.org
```

```
npm publish
```

## 项目验证

```
// const kkb = require("library-kkb"); //common
// import kkb from "library-kkb"; //es module

console.log(kkb(10, 10));

// scripts
  <script src="./add-number.min.js">
  <script>
    console.log(addNumber.default(10,10))
  </script>
//
```



