教程目的

系统全面了解 webpack 打包工具的使用，掌握从零构建大型项目的技术，是作为一名高级前端工程师不可或缺的能力。通常我们接手的项目都有了较为完整、成熟的架构，一方面降低了前端入门门槛，另一方面也划出一道初中级向高级迈进的门槛。试想（未来）作为团队的一名核心成员，不熟悉升级项目需要了解的技术点将会是多么被动。本教程从**应用角度**出发，抛开官方文档的晦涩难懂的细节，引领大家一步步实现一套可用的前端大型项目架构。

安装 webpack

创建文件夹如 webpack，并在终端内执行

```
yarn init // 一路回车，当然仓库的信息也可根据需求填写
yarn add webpack webpack-cli -D
```

教程当前安装的版本

```
"webpack": "^5.70.0",
"webpack-cli": "^4.9.2"
```

webpack 从零到一

1. 创建基础文件目录如下：

```
webpack/
  node_modules/
  src/
    index.js
  index.html
  package.json
```

2. `src/index.js` 文件内容：

```
function test(content) {
  document.querySelector('#app').innerHTML = content;
}

test('something');
```

3. `index.html` 文件内容（`./output/main.js` 是打包后的 `js` 代码路径）：

```
<!DOCTYPE html>
<html>
  <head>
```

```
    <meta charset="utf-8">
    <title>webpack</title>
  </head>
  <body>
    <div id="app"></div>
  </body>
</html>
<script type="text/javascript" src="./output/main.js"></script>
```

4. 修改 `package.json` 文件，补充打包命令：对 `./src/index.js` 文件进行编译，输出目录为 `output` ，指定开发环境参数和 `devtool` 类型可以不对产物进行压缩，便于后续的产物分析。

```
{
  "name": "wp",
  "version": "1.0.0",
  "main": "index.js",
  "license": "MIT",
  "devDependencies": {
    "webpack": "^5.70.0",
    "webpack-cli": "^4.9.2"
  },
  "scripts": {
    "build": "webpack ./src/index.js -o ./output --mode=development --devtool=cheap-module-source-map"
  }
}
```

5. 执行命令 `yarn build` ，可见打包出的文件 `output/main.js` 如下：

```
/******/ (() => { // webpackBootstrap
var __webpack_exports__ = {};
/*!**********************!*\
  !*** ./src/index.js ***!
  \**********************/
function test(content) {
  document.querySelector('#app').innerHTML = content;
}

test('something');
/******/ })()
;
//# sourceMappingURL=main.js.map
```

这时候浏览器打开 `index.html` 就可以看到 something 字符了！显然，我们的源码本身就是可直接在浏览器环境运行的 ES5 代码，webpack 处理以后除了将源码包裹到 IIFE 内之外，并没有额外的逻辑。

webpack 从一到二

1. 新建 `webpack.config.js` 文件，将 `build` 命令执行的参数转移到该文件：

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

module.exports = {
  mode: 'development',
  devtool: 'cheap-module-source-map',
  entry: './src/index.js',
  output: {
    path: path.resolve(__dirname, 'output'),
    filename: 'main.js',
    // publicPath: '/' // 导入静态资源时的路径前缀，本教程使用相对路径，可不配置
  },
}
```

2. 修改 `package.json` 命令参数置为缺省值：

```
"scripts": {
  "build": "webpack"
}
```

再次执行命令 `yarn build`，效果一模一样。

3. 增加 ES6 的转换能力

   a. 创建文件 `src/es6.js`：

> webpack 2 支持原生的 ES6 模块语法，意味着你无须额外引入 babel 这样的工具，就可以使用 import 和 export。但是如果使用其他的 ES6+ 特性，需要引入 babel。

```
export default class CountChange {
  count = 1
  increment = () => {
    this.count++
  }
  decrease = () => {
    this.count--;
  }
}
```

b. `src/index.js` 中引入：

```
import CountChange from './es6';

const instance = new CountChange();

function test(content) {
  document.querySelector('#app').innerHTML = content;
}

test(instance.count)
```

c. 打包产物局部如下：

```
...
/* harmony export */ __webpack_require__.d(__webpack_exports__, {
/* harmony export */   "default": () => (/* binding */ CountChange)
/* harmony export */ });
class CountChange {
  count = 1
  increment = () => {
    this.count++
  }
  decrease = () => {
    this.count--;
  }
}
...
(() => {
/*!**********************!*\
  !*** ./src/index.js ***!
  \**********************/
__webpack_require__.r(__webpack_exports__);
/* harmony import */ var _es6__WEBPACK_IMPORTED_MODULE_0__ = __webpack_require__(/*!
./es6 */ "./src/es6.js");

const instance = new _es6__WEBPACK_IMPORTED_MODULE_0__["default"]();

function test(content) {
  document.querySelector('#app').innerHTML = content;
}
```

```
27
28 test(instance.count)
29 })();
```

可见 `class CountChange` 还是原封不动，虽然 chrome 已经支持了类的原生运行，但有些浏览器还是只能使用 ES5 的代码。再如我们使用提案阶段的装饰器语法的话，chrome 也无能为力。

4. 安装 babel 插件

```
1 yarn add @babel/core @babel/preset-env babel-loader -D
```

webpack.config.js 增加配置

```
1 {
2   ...
3   module: {
4     rules: [{
5       test: /\.js$/,
6       use: {
7         loader: 'babel-loader',
8         options: {
9           presets: [
10            '@babel/preset-env'
11          ],
12        }
13      }
14    }]
15  }
16 }
```

此时查看打包结果 output/main.js，截取与上面对应的位置：

```
1 ...
2 var CountChange = /*#__PURE__*/_createClass(function CountChange() {
3   var _this = this;
4
5   _classCallCheck(this, CountChange);
6
7   _defineProperty(this, "count", 1);
8
9   _defineProperty(this, "increment", function () {
10    _this.count++;
11  });
```

```
12
13    _defineProperty(this, "decrease", function () {
14      _this.count--;
15    });
16  });
17  ...
18  var instance = new _es6__WEBPACK_IMPORTED_MODULE_0__["default"]();
19
20  function test(content) {
21    document.querySelector('#app').innerHTML = content;
22  }
23
24  test(instance.count);
```

@babel/preset-env 是一系列插件的预置配置（state-3 之后的提案，不出意料将会成为下阶段的标准特性），包括了类（class）的转译（@babel/plugin-transform-classes）。如果想使用预设之外的插件，需要自行配置 plugins，例如装饰器：

```
1  yarn add @babel/plugin-proposal-decorators -D
```

src/es6.js

```
1  const decorator = (target, key, descriptor) => {
2    target[key] = function (...args) {
3      console.log(this.count);
4      return descriptor.value.apply(this, args);
5    };
6    return target[key];
7  }
8
9  export default class CountChange {
10   count = 1
11
12   @decorator
13   increment() { // 注意这里不用箭头函数
14     this.count++;
15   }
16   ...
17 }
```

webpack.config.js

```
// module.rules
[{
  test: /\.js$/,
  use: {
    loader: 'babel-loader',
    options: {
      presets: ['@babel/preset-env'],
      plugins: [
        ["@babel/plugin-proposal-decorators", { "legacy": true }],
      ]
    }
  }
}]
```

再次观察 output/main.js：

```
...
var CountChange = (_class = /*#__PURE__*/function () {
  function CountChange() {
    var _this = this;

    _classCallCheck(this, CountChange);

    _defineProperty(this, "count", 1);

    _defineProperty(this, "decrease", function () {
      _this.count--;
    });
  }

  _createClass(CountChange, [{
    key: "increment",
    value: function increment() {
      this.count++;
    }
  }]);

  return CountChange;
}(), (
  _applyDecoratedDescriptor(
```

```
    _class.prototype, "increment", [decorator],
    Object.getOwnPropertyDescriptor(_class.prototype, "increment"),
    _class.prototype
  )
), _class);
...
```

注意，装饰器函数源码的 target 是类的原型对象，本示例中方法的装饰等于重写，现在成功通过 babel 插件打包了新的语法。

可用于生产的 react 脚手架

- 安装模块

```
yarn add react react-dom -S
yarn add @babel/preset-react -D
```

- 创建文件 src/react.js

```
import React from 'react';
import { render } from 'react-dom';

const App = () => <div>App</div>;

render(<App />,
document.querySelector('#app'));
```

- 补充配置，webpack.config.js：

```
{
  entry: './src/react.js', // 修改文件入口
  module: {
    rules: [{
      test: /\.js$/,
      use: {
        loader: 'babel-loader',
        options: {
          presets: [
            '@babel/preset-env', '@babel/preset-react'
          ],
          ... // 不使用装饰器的话可以去掉 @babel/plugin-proposal-decorators 配置
        }
      }
    }]
```

```
  }
}
```

现在打包得到一个比较大文件 output/main.js，访问 index.html 也成功了。如果把较为固定的模块抽离出来，便于缓存，增加配置：

```
{
  optimization: {
    splitChunks: {
      cacheGroups: {
        vendor: {
          filename: 'vendor.js',
          chunks: 'all',
          test: /[\\/]node_modules[\\/](react|react-dom)[\\/]/
        },
      }
    }
  }
}
```

打包会多出一个文件 vendor.js，其实就是把 react 和 react-dom 的代码抽离出来，原先的 main.js 的体积减小。optimization 是 webpack5 优化打包产物的一个通用配置，官方文档有更详细的说明。现在多出来的产物我们需要手动引入到 HTML 文件的开头才能正常访问。

- 处理样式的 loader 和 plugin，先安装：

```
yarn add style-loader css-loader mini-css-extract-plugin -D
```

再增加配置，webpack.config.js：

```
const MiniCssExtractPlugin = require('mini-css-extract-plugin');

module.exports = {
  ...
  plugins: [
    new MiniCssExtractPlugin({
      filename: "[name].css",
      chunkFilename: "[id].css"
    })
  ],
  module: {
    rules: [
      ..., {
```

```
      test: /\.css$/,
      use: [MiniCssExtractPlugin.loader, 'css-loader']
    }]
  }
}
```

在 react.js 文件导入一份样式表随意加入一些样式，现在打包会多出来一个样式表文件 `output/main.css` ，但是刷新 `index.html` 是不成效的，需要我们手动把这个样式表引入文档。至此，打包样式表的能力已经有了，但是样式表是动态生成的，尤其是名称加入 hash 后手动引入是不现实的。借助另一个插件[html-webpack-plugin]：

```
yarn add html-webpack-plugin -D
```

补充配置 webpack.config.js（多页面时可以多次配置该插件）：

```
const HtmlWebpackPlugin = require('html-webpack-plugin');
{
  ...
  plugins:[
    ...
    new HtmlWebpackPlugin()
  ],
}
```

这样打包会多输出一个文件 `output/index.html` ，内容为

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Webpack App</title>
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <script defer src="vendor.js"></script>
    <script defer src="main.js"></script>
    <link href="main.css" rel="stylesheet">
  </head>
  <body>
  </body>
</html>
```

可见生成的 js、css 文件都自动引入了，唯一差的就是 body 标签里没有 div#app 元素，所以渲染页面是空白的。最简单的方式是我们提供一份 HTML 模板，使插件使用模板并向里面插入链接。增

加文件 template.html，内容至少包含 div#app 元素，当然可以根据业务需求加入其它 SDK。配置上调整 webpack.config.js：

```
plugins:[
  ...
  new HtmlWebpackPlugin({
    template: './template.html'
  })
]
```

这样自动引入链接的需求就完成了，更多模板细节可以在官网上了解。

HMR（hot module replacement）

到目前为止，我们一直在使用打包 + 刷新的模式查看代码效果，显然十分繁琐低效。现代前端开发脚手架都会借助本地开发服务器来解决这个问题：

```
yarn add webpack-dev-server -D
```

增加命令 package.json：

```
"scripts": {
  "start": "webpack serve"
 }
```

`yarn start` 启动项目，访问 8080 端口可以看到页面正常渲染了，修改 App 组件也能实时得到反馈，但页面的更新是刷新而非局部渲染！继续完善配置 webpack.config.js，第一步热更新配置（顺便指定一下端口，后面记得访问 8000 端口查看页面）：

```
module.exports = {
  ...
  devServer: {
    port: 8000,
    hot: true
  }
}
```

第二步修改入口文件，文件底部增加新内容：

```
if (module.hot) {
  module.hot.accept(App, () => {
    render(<App />, document.querySelector('#app'));
  });
}
```

```
// 或者， App 组件是外部文件创建的，通常写作（与 import 导入的路径一致）：
if (module.hot) {
  module.hot.accept('./App', () => {
    render(<App />, document.querySelector('#app'));
  });
}

```

至此，一个基于 webpack 的功能满足基本需求的脚手架就搭建完成了！至于配置状态管理、路由管理、代码风格管理，参照官方文档便可以轻松解决。

异步组件的原理

- react 官方异步组件。新建组件 Async.js：

```
import React from 'react';

export default function Async() {
  return (
    <div className='async'>
      Async
    </div>
  );
}
```

根组件导入它（react.js）：

```
import React, { lazy, Suspense } from 'react';
import { render } from 'react-dom';

const Async = lazy(() => import('./Async'));
const App = () => <div className="red">
  App:
  <Suspense fallback={<>loading</>}>
    <Async />
  </Suspense>
</div>;

render(<App />, document.querySelector('#app'));
...
```

现在页面上已经出现 Async 组件的内容了，打包后会有新的文件 output/src_Async_js.main.js （文件名以当前配置的默认规则生成，未来可能会不同，也可以通过配置修改）产生。所以异步组件的明显特征之一，就是独立于其他代码加载，便于主体逻辑能第一时间呈现给用户。强制刷新页面时，异步组件的位置会有占位的内容一闪而过，上面的 fallback 就是占位的 JSX。占位的内容体积较小，以同步的方式加载，直到异步组件真正下载完成，才会被替换。

- 自定义异步组件，创建 loadComponent.js 文件：

```
import React, { Component } from 'react';

export default function (loadComponent) {

  return class extends Component {
    state = {
      C: () => <>loading</>
    }
    async componentDidMount() {
      const { default: C } = await loadComponent();
      this.setState({ C });
    }
    render() {
      const C = this.state.C;
      return <C />;
    }
  }
}
```

根组件使用（react.js）：

```
import loadComponent from './loadComponent';

// 注意现在没有使用 react 的 lazy, Suspense
const Async = loadComponent(() => import('./Async'));
const App = () => <div className="red">
  App:<Async />
</div>;
...
```

在这儿使用了 async/await ，报错 regeneratorRuntime is not defined ，因为 babel 默认只转换新的 JavaScript 语法（syntax），如箭头函数、class 等，而不转换新的 API，比如 Iterator、Generator、Set、Maps、Proxy、Reflect、Symbol、Promise 等全局对象，此时需要一些辅助函数

（babel 6.x以下版本借助 polyfill，需要在 entry 之前或根文件头部引入，本课程均以 babel 7 之后的标准讲解）：

```
yarn add @babel/plugin-transform-runtime -D
```

现在我们移除 webpack.config.js 的 babel-loader 配置的 options 属性，在项目根目录建立 .babelrc 文件（不使用装饰器时可以不写 decorators 插件）：

```
{
  presets: [
    "@babel/preset-env",
    "@babel/preset-react"
  ],
  plugins: [
    "@babel/plugin-transform-runtime",
    ["@babel/plugin-proposal-decorators", { "legacy": true }]
  ]
}
```

现在打包依然单独抽取出了 output/src_Async_js.main.js 文件，达到了使用异步组件的目的。异步组件从本质上，解决的是 SPA 用户体验的问题。它为webpack 提供了代码分割的依据，使得使用率高或者加载时间长的组件代码独立出去，同时通过低成本的过渡交互，保证了网站的流畅性。

- 异步路由，曾几何时，我们在 react-router 中配置路由组件是这样的：

```
{
  path: '/home',
  getComponent(location, callback) {
    require.ensure([], () => {
      const Page = require('./Home').default;
      callback(null, Page);
    });
  }
}
```

require.ensure 函数是 webpack 特有的作为代码分割点的依据，已被 import() 取代。其语法了解即可：

```
require.ensure(
  dependencies: String[],
  callback: function(require),
  errorCallback: function(error),
  chunkName: String
```

```
)
```

下一部分：webpack HMR 原理、plugin & loader 的实现