实战：给 Webpack 项目添加 modern 模式打包

更新时间：2019-07-18 18:39:54

> 学习这件事不在乎有没有人教你，最重要的是在于你自己有没有觉悟和恒心。
>
> —— 法布尔

在 Vue-CLI 3 中有个Modern Mode 的新功能，在现代模式（modern mode）下，打包出来的 JavaScript 代码是ES2015+的，官方的解释是这样的：

> 有了 Babel 我们可以兼顾所有最新的 ES2015+ 语言特性，但也意味着我们需要交付转译和 polyfill 后的包以支持旧浏览器。这些转译后的包通常都比原生的 ES2015+ 代码会更冗长，运行更慢。现如今绝大多数现代浏览器都已经支持了原生的 ES2015，所以因为要支持更老的浏览器而为它们交付笨重的代码是一种浪费。

在支持 ES2015+（ES6+）的浏览器中，我们在 JavaScript 编写的绝大多数 ES 语法都被浏览器原生支持：

- `Class` 语法；
- `async` / `await` 语法；
- 箭头函数、模板等新语法；
- 支持新的 API：`Promise`、`Fetch`、`Map`、`Set`等。

> Tips：ES2015 是 2015 年发布的第六个版本的 ECMAScript 标准，所以 ES2015 和 ES6 是一回事，本文延续 Vue-CLI 文档的称呼。

直接原生的支持 ES2015+语法的好处是：

1. 不需要额外的 polyfill 就可以支持最新的 ES6 语法，减少了代码体积；

2. 体积小了，那么下载速度就加快；
3. JavaScript 是解释执行的，所以更小的代码体积和更现代的代码，能够提升代码的解析速度（parse），运行也更快。

「对于 Vue 的 Hello World 应用（`vue create hello-world`）来说，现代版的包已经小了 16%。在生产环境下，现代版的包通常都会表现出显著的解析速度和运算速度，从而改善应用的加载性能」。由此可见，相对于 Hello World 这样的 Demo 应用都有这么大的体积减少和性能提升，那么对于我们复杂的项目来说使用现代模式的吸引力就更大了。

看到这些好处，我们是不是也想在自己的项目中尝试下现代模式呢？如果我们的项目不是 Vue-CLI 3 作为打包工具，那么我们可以通过本文的实战来给自己的项目添加现代模式。已经使用 Vue-CLI 3 的项目，也可以通过本文学习到现代模式的原理。

Modern Mode 实现原理

Modern Mode 代码通过 Babel 是很容易编译出来的，Modern Mode 实现难点是如何做好浏览器的兼容性，即在不支持 ES2015+ 在浏览器中能够正常执行 JavaScript 代码。为了实现兼容性，Webpack 需要在打包的时候将原始的 JavaScript 代码打包出两份 JavaScript 代码，一份用于老版本的浏览器，一份用于现代浏览器。

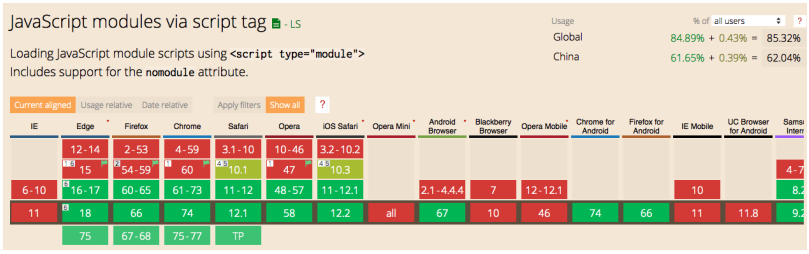
在浏览器中，应该根据所处的 JavaScript 语法特性进行选择，在浏览器环境中进行 ES 语法特性检测没有特比好的解决方案。最终我们可以通过检测`script`标签的`type="module"`的方式进行现代浏览器和老版浏览器的适配。这种方式是参考了 Phillip Walton 的这篇文章 Deploying ES2015+ Code in Production Today（中文版本），其中提出了基于 script 标签的 `type="module"` 和 `nomodule` 属性 区分出当前浏览器对 ES2015+ 的支持程度。具体原理实现可以参考下面的代码：

```
<script type="module" src="main.js"></script>
<script nomodule src="main.legacy.js"></script>
```

在上面的代码中，如果浏览器支持 ES6 的`module`语法，那么就可以执行`main.js`的代码，从而忽略下面的`nomodule`代码，这样现代浏览器就可以通过这种方式执行我们的 ES2015+ 语法的 JavaScript 文件。而对于不支持`module`语法的浏览器，那么`type="module"`不被识别，而会执行后面的`main.legacy.js`代码。

这就是我们要实现 Modern Mode 的基本原理。在caniuse.com网站上我们可以看到`<script type="module">`的支持情况：

`China 61.65% + 0.39% = 62.04%`。这个数据来看，在国内已经有 60%+的浏览器支持 ES2015+语法了，我们可以让着 60%+的用户使用更快更好的 JavaScript 加载、解析和执行体验，并且我们不需要修改任何代码，就可以让 Webpack 来做这个优化，今后随着浏览器升级，我们甚至可以全部使用 ES2015+语法。



在使用`<script type="module">`之前，我们还需要修复 Safari 10.1 和 iOS Safari 10.3 已经支持`module`语法，但是不支持 script 标签`nomodule`属性的一个问题，具体的代码来自A polyfill is available for Safari 10.1/iOS Safari 10.3。

```
(function() {
    var check = document.createElement('script');
    if (!('noModule' in check) && 'onbeforeload' in check) {
        var support = false;
        document.addEventListener(
            'beforeload',
            function(e) {
                if (e.target === check) {
                    support = true;
                } else if (!e.target.hasAttribute('nomodule') || !support) {
                    return;
                }
                e.preventDefault();
            },
            true
        );

        check.type = 'module';
        check.src = '.';
        document.head.appendChild(check);
        check.remove();
    }
})();
```

Vue-CLI 3 Modern Mode 实现分析

上面的原理介绍完了，我们来看下具体 Vue-CLI 3 在 Modern Mode 模式下的产出：

```
<!--预取-->
<link href=/css/app.e2713bb0.css rel=preload as=style>
<link href=/js/app.962c146a.js rel=modulepreload as=script>
<link href=/js/chunk-vendors.ab5b1059.js rel=modulepreload as=script>
<link href=/css/app.e2713bb0.css rel=stylesheet>

<!--type=module-->
<script type=module src=/js/chunk-vendors.ab5b1059.js></script>
<script type=module src=/js/app.962c146a.js></script>
<script>
// 兼容 Safari 10
!(function() {
  var e = document,
    t = e.createElement('script');
  if (!('noModule' in t) && 'onbeforeload' in t) {
    var n = !1;
    e.addEventListener(
      'beforeload',
      function(e) {
        if (e.target === t) n = !0;
        else if (!e.target.hasAttribute('nomodule') || !n) return;
        e.preventDefault();
      },
      !0
    ),
      (t.type = 'module'),
      (t.src = '.'),
      e.head.appendChild(t),
      t.remove();
  }
})();
</script>
<!--老浏览器代码-->
<script src=/js/chunk-vendors-legacy.ecd76ec1.js nomodule></script>
<script src=/js/app-legacy.7c8e48ce.js nomodule></script>
```

在实战部分实现`prefetch`插件的时候，已经介绍过`link`标签的`prefetch`和`preload`，在 Vue 的产出中，使用了`preload`，并且配合了`as`和`modulepreload`。使用`as`属性，可以明确的告诉浏览器预加载的资源类型，从而使浏览器能够更加精确的去优化加载资源。Chrome 从 64 版本后 开始 「实验性的支持这个特征`modulepreload`」，这个属性值，`<link rel="modulepreload">`是 `<link rel="preload">` 的特定模块（module）版本，可以针对 ES Modules 进行特定的优化和处理。

最后在产出物的最后在使用`<script type="module">`加载现代浏览器执行的 JavaScript 代码，使用`<script nomodule>`加载不支持 ES6 语法的 polyfill 代码。从上面的产出来看，我们要实现 Modern Mode 模式代码，需要做的事情是：

1. 让 Webpack 打包两份代码，一份是支持现代浏览器的代码，一份是不支持 ES6 语法的`legacy`代码；
2. 处理 HTML 中对代码的引入，增加`modulepreload`、添加第一步打出的两份 bundle 地址，并且插入对应的`script`属性和 Safari 10 的 polyfill 代码。

在第一步打出两份代码的方式，Vue-CLI 3 是执行两次打包，两次打包通过不同的 Babel 配置产生不同的代码，具体来说，在 Vue-CLI 3 的`babel-preset-app`中，设置`@babel/preset-env`的`targets={esmodule:true}`，不转换所有的语法也不添加 polyfill，生成 ES6 的能被现代浏览器执行的代码：

```
// https://github.com/vuejs/vue-cli/blob/dev/packages/%40vue/babel-preset-app/index.js#L89
let targets;
if (process.env.VUE_CLI_BABEL_TARGET_NODE) {
    // running tests in Node.js
    targets = {node: 'current'};
} else if (process.env.VUE_CLI_BUILD_TARGET === 'wc' || process.env.VUE_CLI_BUILD_TARGET === 'wc-async') {
    // targeting browsers that at least support ES2015 classes
    // https://github.com/babel/babel/blob/master/packages/babel-preset-env/data/plugins.json#L52-L61
    targets = {
        browsers: ['Chrome >= 49', 'Firefox >= 45', 'Safari >= 10', 'Edge >= 13', 'iOS >= 10', 'Electron >= 0.36']
    };
} else if (process.env.VUE_CLI_MODERN_BUILD) {
    // targeting browsers that support <script type="module">
    targets = {esmodules: true};
} else {
    targets = rawTargets;
}

//...
const envOptions = {
    corejs: 3,
    spec,
    loose,
    debug,
    modules,
    targets,
    useBuiltIns,
    ignoreBrowserslistConfig,
    configPath,
    include,
    exclude: polyfills.concat(exclude || []),
    shippedProposals,
    forceAllTransforms
};

// ...
// pass options along to babel-preset-env
presets.unshift([require('@babel/preset-env'), envOptions]);
```

处理 HTML 代码，添加 Modern Mode 代码支持则是在`cli-service`中的`lib/webpack/ModernModePlugin.js`通过编写一个`html-webpack-plugin`的插件而实现的，这个跟我们之前[TODO 插件实战文章链接]实战编写 Webpack 插件一样的「套路」。

如何打包出来 Modern Mode 的 JavaScript 代码

通过上面的介绍，我们知道了：要实现 Webpack 打包出支持 Modern Mode 的代码，需要设置`@babel/preset-env`的`targets={esmodule: true}`，而要打包出两份代码，则需要 Webpack 打包两次。那么在我们实际项目中，可以通过我们首先可以使用 Webpack 的 API 来打包，并且将 API 改写成 Promise 的方式，具体代码示例如下：

```
const webpack = require('webpack');
const webpackConfig = require('./webpack.config.js');

const webpackPromise = webpackConfig => {
    return new Promise((resolve, reject) => {
        webpack(webpackConfig, (err, stats) => {
            if (err) {
                console.error(err);
                reject();
                return;
            }
            if (stats.hasErrors()) {
                const info = stats.toJson();
                console.error(info.errors);
                reject();
                return;
            }
            resolve(stats);
        });
    });
};
// 下面是使用
webpackPromise(webpackConfig)
    .then(stats => {
        console.log(stats.toString());
    })
    .catch(e => {
        console.log(e);
    });
```

上面的`webpackPromise`我们已经将 Webpack 的 API 调用改成了 Promise 的方式，那么下面是我们怎么将非现代浏览器的 Webpack 配置修改成现代浏览器的，即设置`@babel/preset-env`的`targets={esmodule: true}`，这里我们有很多办法，下面我推荐两种方式：

1. 使用webpack-merge；
2. 零件配置方式。

Webpack-merge

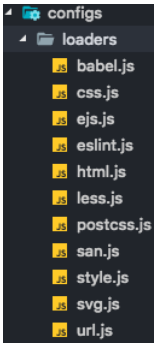
webpack-merge 是我们常用的将 Webpack 的 Object 类型配置进行合并（merge）的方法，在之前《Webpack 环境相关配置及配置文件拆分[TODO 链接]》中提到过 webpack-merge 的使用，可以将多个配置文件的配置进行合并，我们在这里也是使用这个插件的，具体代码示例如下：

```
// 首先我们假设下面是 webpack.config.js 内容
module.exports = {
    mode: 'production',
    entry: {
        main: './src/index.js'
    },
    output: {
        filename: '[name]-legacy-[chunkhash].js'
    },
    module: {
        rules: [
            {
                test: /\.js$/,
                loader: 'babel-loader',
                options: {
                    presets: [
                        [
                            '@babel/preset-env',
                            {
                                useBuiltIns: 'usage',
                                corejs: 3
                            }
                        ]
                    ],
                    plugins: ['babel-plugin-transform-dynamic-import-default']
                }
            }
        ]
    }
};
// 那么我们在打包脚本中可以使用 merge 来替换 preset 的配置
const merge = require('webpack-merge');
const webpackConfig = require('./webpack.config');

// 这里使用 merge.smart 方法
const modernConfig = merge.smart(webpackConfig, {
    output: {filename: '[name]-modern-[chunkhash].js'},
    module: {
        rules: [
            {
                test: /\.js$/,
                loader: 'babel-loader',
                options: {
                    presets: [
                        [
                            '@babel/preset-env',
                            {
                                targets: {esmodules: true}
                            }
                        ]
                    ]
                }
            }
        ]
    }
});
// 下面是合并后的 webpack config
console.log(modernConfig);
```

零件配置方式

如果项目的 webpack 配置文件按照在《Webpack 环境相关配置及配置文件拆分[TODO webpack]》文章中方法，将 loader 相关的配置拆成了函数，例如我们项目中的 loader 配置是如下拆分的：



这样每个 loader 配置都是一个类似下面的函数，通过传入的参数最终生成对应 loader 的配置：

```
// babel.js
// 通过传入参数获取 babel loader 的配置
module.exports = options => {
    const plugins = (options && options.plugins) || [];
    let targets = options.browserslist;
    // 是 modern 模式，但不是 modern 打包，那么 js 加上 legacy
    const isModernBundle = options.modernBuild;
    if (isModernBundle) {
        // 这个是 modern 打包
        targets = {esmodules: true};
    }

    return {
        name: 'babel-loader',
        loader: require.resolve('babel-loader'),
        options: {
            cacheDirectory: true,
            presets: [
                [
                    require('@babel/preset-env'),
                    {
                        debug: false,
                        useBuiltIns: 'usage',
                        corejs: 3,
                        targets,
                        modules: false
                    }
                ]
            ],
            plugins: [
                require('@babel/plugin-syntax-dynamic-import'),
                require('@babel/plugin-syntax-import-meta'),
                require('@babel/plugin-proposal-class-properties'),
                require('@babel/plugin-transform-new-target'),
                require('@babel/plugin-transform-modules-commonjs'),
                [
                    require('@babel/plugin-transform-runtime'),
                    {
                        // corejs: false, // 默认值，可以不写
                        regenerator: false, // 通过 preset-env 已经使用了全局的 regeneratorRuntime，不再需要 transform-runti
me 提供的 不污染全局的 regeneratorRuntime
                        helpers: true, // 默认，可以不写
                        useESModules: false, // 不使用 es modules helpers，减少 commonJS 语法代码
                        absoluteRuntime: path.dirname(require.resolve('@babel/runtime/package.json'))
                    }
                ],
                ...plugins
            ]
        }
    };
};
```

上面的代码中，那么我们可以通过传入参数的方式得到现代浏览器的 Babel 配置：

```
// webpack config 类文件中
const babelOptions = getBabelLoader({modernBuild: true});
module.exports = {
    module: {
        rules: [
            {
                test: /\.js$/,
                ...babelOptions
            }
        ]
    }
};
```

测试 **modern mode** 打包效果

通过上面的配置，我们可以完成 modern mode 模式打包了，下面编写个测试项目：

```
// src/index.js
import {dep1} from './dep-1.js';
import {dep2} from './dep-2.js';

const main = async () => {
    console.log('Dependency 1 value:', dep1);
    console.log('Dependency 2 value:', dep2);

    const {import1} = await import(
        /* webpackChunkName: "import1" */
        './import-1.js'
    );
    console.log('Dynamic Import 1 value:', import1);

    const {import2} = await import(
        /* webpackChunkName: "import2" */
        './import-2.js'
    );
    console.log('Dynamic Import 2 value:', import2);

    console.log('Fetching data, awaiting response...');
    const response = await fetch('http://jsonplaceholder.typicode.com/users');
    const json = await response.json();

    console.log('Response:', json);
};

main();

// src/dep-1.js
export const dep1 = 'dep-1';

// src/dep-2.js
export const dep2 = 'dep-2';

// src/import-1.js
import {dep1} from './dep-1';
export const import1 = `imported: ${dep1}`;

// src/import-2.js
import {dep2} from './dep-2';
export const import2 = `imported: ${dep2}`;
```

对应`webpack.config.js`的配置如下：

```
module.exports = {
    mode: 'production',
    entry: {
        main: './src/index.js'
    },
    output: {
        filename: '[name]-legacy-[chunkhash].js'
    },
    module: {
        rules: [
            {
                test: /\.js$/,
                loader: 'babel-loader',
                options: {
                    presets: [
                        [
                            '@babel/preset-env',
                            {
                                useBuiltIns: 'usage',
                                corejs: 3
                            }
                        ]
                    ],
                    plugins: ['babel-plugin-transform-dynamic-import-default']
                }
            }
        ]
    }
};
```

打包脚本`build.js`内容如下：

```
// build.js
const webpack = require('webpack');
const merge = require('webpack-merge');
const webpackConfig = require('./webpack.config');

const webpackPromise = webpackConfig => {
    return new Promise((resolve, reject) => {
        webpack(webpackConfig, (err, stats) => {
            if (err) {
                console.error(err);
                reject();
                return;
            }
            if (stats.hasErrors()) {
                const info = stats.toJson();
                console.error(info.errors);
                reject();
                return;
            }
            resolve(stats);
        });
    });
};

const modernConfig = merge.smart(webpackConfig, {
    output: {filename: '[name]-modern-[chunkhash].js'},
    module: {
        rules: [
            {
                test: /\.js$/,
                loader: 'babel-loader',
                options: {
                    presets: [
                        [
                            '@babel/preset-env',
                            {
                                targets: {esmodules: true}
                            }
                        ]
                    ],
                    plugins: ['babel-plugin-transform-dynamic-import-default']
                }
            }
        ]
    }
});

Promise.all([webpackPromise(webpackConfig), webpackPromise(modernConfig)])
    .then(([legacyStats, modernStats]) => {
        // 输出老版本打包
        console.log(legacyStats.toString({chunks: false, modules: false, colors: true}));
        // 输出 modern 版本打包
        console.log(modernStats.toString({chunks: false, modules: false, colors: true}));
    })
    .catch(e => console.log(e));
```

最终执行`node build.js`，结果如下：

上面的 log 显示，我们的老版本浏览器的文件`main-legacy-f2fb0978e775b1b7d7e9.js`为 53K，而 Modern Mode 打包出来的文件`main-modern-4cfa0263ac7971396ede.js`才 3K！

如何通过`script`标签处理 Modern Mode 代码兼容性

打包出来 ES2015+（modern） 和 ES2015-（legacy） 两个 bundle 文件还不是结束，还需要通过 html-webpack-plugin 插件将这两个 bundle 文件按照之前的介绍使用 `script` 标签的 `type="module"` 和 `nomodule` 添加到 HTML 中，最终得到如下的 HTML：

```
<script type="module" src="main-modern-4cfa0263ac7971396ede.js"></script>
<script>
// 下面是修复 safari 10 的 polyfill
!(function() {
    var e = document,
        t = e.createElement('script');
    if (!('noModule' in t) && 'onbeforeload' in t) {
        var n = !1;
        e.addEventListener(
            'beforeload',
            function(e) {
                if (e.target === t) n = !0;
                else if (!e.target.hasAttribute('nomodule') || !n) return;
                e.preventDefault();
            },
            !0
        ),
            (t.type = 'module'),
            (t.src = '.'),
            e.head.appendChild(t),
            t.remove();
    }
})();</script>
<script type="text/javascript" src="main-legacy-f2fb0978e775b1b7d7e9.js" nomodule></script>
```

实现上，可以采用 Vue-CLI 的 Webpack 插件 ModernModePlugin 实现。在TODO 手写 plugin的时候介绍过 html-webpack-plugin 这个插件，这个 webpack 的 HTML 插件，html-webpack-plugin 会在 `compilation` 对象上增加一些 Hook。

> Tips：Vue-CLI 的 ModernModePlugin 是基于 html-webpack-plugin 3.2 版本编写的，html-webpack-plugin 3.x 和最新的 html-webpack-plugin 4.x 使用 Hook 是不一样的，但是基本 Hook 的流程点是可以对上的，这部分包括 html-webpack-plugin 的 hook 使用方法在《TODO 手写 plugin》小节有更加详细的介绍。

在 ModernModePlugin 中，我们使用了 html-webpack-plugin 的 v3 版本的两个 Hook：

- `compilation.hooks.htmlWebpackPluginAlterAssetTags`
- `compilation.hooks.htmlWebpackPluginAfterHtmlProcessing`

主要 Hook 是 `htmlWebpackPluginAlterAssetTags`，在这个 Hook 中得到的 `data` 数据可以直接操作 `data.head` 和 `data.body` 这俩数组包含了 html-webpack-plugin 生成的 HTML 中在 `<head>` 和 `<body>` 的所有静态资源，包括 JavaScript 和 CSS 文件等。

在 ModernModePlugin 中，需要区分开是 legacy 打包（非现代模式）还是 modern 打包：

- 在 legacy 打包时：
  - 这时候在 `htmlWebpackPluginAlterAssetTags` 中需要记录下来对应的 bundle 到 `legacy-assets-${htmlName}.json`；
- 在 modern 打包时：
  - 将 modern bundle 添加到 `html-webpack-plugin` 的 `data.head` 中添加 `modulepreload` 的 `link`；

- 添加 Safari 10 的 polyfill 到 `data.body`；
- 读取 legacy 打包时产生的 `legacy-assets-${htmlName}.json` 得到 legacy 的 bundle 内容，然后添加到 `data.body`；

两者的关系连接是通过生成一个中间产物 `legacy-assets-${htmlName}.json` 来实现的！下面我们来详细解释下代码，首先是一个 Webpack 的插件的类结构是这样的：

```
class ModernModePlugin {
    constructor({targetDir, isModernBuild}) {
        // 接受Plugin 实例化时候传入的 options
        this.targetDir = targetDir;
        this.isModernBuild = isModernBuild;
    }

    apply(compiler) {
        // 这是 Webpack Plugin 的核心apply函数
        // 接收的是 comipler 对象
        if (!this.isModernBuild) {
            // 根据参数，进入不同的打包逻辑
            this.applyLegacy(compiler);
        } else {
            this.applyModern(compiler);
        }
    }
    applyLegacy(compiler) {}
    applyModern(compiler) {}
}
```

上面代码看到了，ModernModePlugin 是通过在使用的时候传入不同的 Options（isModernBuild） 而区分是 legacy 还是 modern 打包，如果是 legacy 打包则执行 `this.applyLegacy` 方法，如果是 modern 打包则执行 `this.applyModern` 方法。 `options.targetDir` 是用来存储 `legacy-assets-${htmlName}.json` 文件夹，这里直接使用 output 文件夹即可。

下面再来看第一步， `applyLegacy` 的实现：

```
const fs = require('fs-extra');

class ModernModePlugin {
    constructor({targetDir, isModernBuild}) {
        // 接受Plugin 实例化时候传入的 options
        this.targetDir = targetDir;
        this.isModernBuild = isModernBuild;
    }
    applyLegacy(compiler) {
        const ID = 'html-legacy-bundle';
        // 添加 compiler 对象的 Hook compilation，可以得到 compilation 对象
        compiler.hooks.compilation.tap(ID, compilation => {
            // 按照 v3 版本的 API 绑定htmlWebpackPluginAlterAssetTags Hook
            compilation.hooks.htmlWebpackPluginAlterAssetTags.tapAsync(ID, async (data, cb) => {
                // 使用 fs-extra 的 ensureDir 方法，如果不存在路径则创建，类似 mkdir -p 方法
                await fs.ensureDir(this.targetDir);
                // data.plugin 是 html-webpack-plugin 插件的实例，options 可以得到对应的配置项
                // 得到 html 的 name
                const htmlName = path.basename(data.plugin.options.filename);
                // 得到html 文件路径
                const htmlPath = path.dirname(data.plugin.options.filename);
                // 拼接临时文件的路径
                const tempFilename = path.join(this.targetDir, htmlPath, `legacy-assets-${htmlName}.json`);
                // 调用 fs-extra 的 mkdirp 方法，先创建目录结构，相当于 mkdir -p
                await fs.mkdirp(path.dirname(tempFilename));
                // 将 data.body 内容格式化写到tempFilename文件
                await fs.writeFile(tempFilename, JSON.stringify(data.body));
                cb();
            });
        });
    }
}
```

经过 `applyLegacy` 处理后，生成的 `legacy-assets-index.html.json` 内容包含了 `data.body` 的内容，格式如下：

```
[
    {
        "tagName": "script",
        "closeTag": true,
        "attributes": {"type": "text/javascript", "src": "main-legacy-f2fb0978e775b1b7d7e9.js"}
    }
    //...
]
```

`data.body` 和 `data.head` 最终经过createHtmlTag函数生成对应的 HTML String 片段，然后生成 HTML 页面！

legacy 打包首先打包，结束后就是 modern 的打包，这时候调用的是 `applyModern` 方法：

```
const fs = require('fs-extra');

// 这个是 Safari 10 的 polyfill
// 来自 https://gist.github.com/samthor/64b114e4a4f539915a95b91ffd340acc
const safariFix = `!function(){var e=document,t=e.createElement("script");if(!("noModule"in t)&&"onbeforeload"in t){var n=!
1;e.addEventListener("beforeload",function(e){if(e.target===t)n=!0;else if(!e.target.hasAttribute("nomodule")||!n)return;e.
preventDefault()},!0),t.type="module",t.src=".",e.head.appendChild(t),t.remove()}}();`;

class ModernModePlugin {
    constructor({targetDir, isModernBuild}) {
        // 接受Plugin 实例化时候传入的 options

        this.targetDir = targetDir;
        this.isModernBuild = isModernBuild;
    }
    applyModern(compiler) {
        const ID = 'html-modern-bundle';
        // 添加 compiler 对象的 Hook compilation，可以得到 compilation 对象
        compiler.hooks.compilation.tap(ID, compilation => {
            // 按照 v3 版本的 API 绑定htmlWebpackPluginAlterAssetTags Hook
            compilation.hooks.htmlWebpackPluginAlterAssetTags.tapAsync(ID, async (data, cb) => {
                // 首先将 data.body 中的 js 添加上 <script type="module"> 用于 modern 浏览器识别使用
                data.body.forEach(tag => {
                    if (tag.tagName === 'script' && tag.attributes) {
                        tag.attributes.type = 'module';
                    }
                });

                // 将 head 中的 link preload 资源更换成 modulepreload <link rel="modulepreload">
                data.head.forEach(tag => {
                    if (tag.tagName === 'link' && tag.attributes.rel === 'preload' && tag.attributes.as === 'script') {
                        tag.attributes.rel = 'modulepreload';
                    }
                });

                // 得到 htmlName，实际是为了得到legacy 打包阶段生成的临时文件的路径
                const htmlName = path.basename(data.plugin.options.filename);
                // 得到 html 路径 ，实际是为了得到legacy 打包阶段生成的临时文件的路径
                const htmlPath = path.dirname(data.plugin.options.filename);
                // 拼接得到 legacy 打包阶段生成的临时文件
                const tempFilename = path.join(this.targetDir, htmlPath, `legacy-assets-${htmlName}.json`);
                // 读取 legacy 打包阶段生成的临时文件
                const legacyAssets = JSON.parse(await fs.readFile(tempFilename, 'utf-8')).filter(
                    a => a.tagName === 'script' && a.attributes
                );
                // 给 legacyAssets 的script 标签加上 nomodule 属性，保证 modern 浏览器不能用
                legacyAssets.forEach(a => {
                    a.attributes.nomodule = '';
                });

                // 插入 Safari 10 nomodule polyfill
                data.body.push({
                    tagName: 'script',
                    closeTag: true,
                    innerHTML: safariFix
                });
                // 将 legacyAssets添加到 modern 阶段生成的 data.body 上
                data.body.push(...legacyAssets);
                // 删除临时文件
                await fs.remove(tempFilename);
                cb();
            });
            // 租后是替换掉空的`nodemodule=''`属性
            compilation.hooks.htmlWebpackPluginAfterHtmlProcessing.tap(ID, data => {
                data.html = data.html.replace(/\snomodule="">/g, ' nomodule>');
            });
        });
    }
}
```

到此，最后 html-webpack-plugin 就会生成对应的 HTML 代码了！

测试

最后在来测试下我们的 Modern 模式打包全流程，继续修改`build.js`内容，让它先打包 legacy 包，成功之后在打包 modern 包，具体的代码如下：

```
// 添加各自的 ModernModePlugin，传入不同的isModernBuild
webpackConfig.plugins.push(new ModernModePlugin({targetDir: __dirname + '/dist', isModernBuild: false}));
modernConfig.plugins.push(new ModernModePlugin({targetDir: __dirname + '/dist', isModernBuild: true}));
// 首先打包 legacy
// 成功后打包 modern
webpackPromise(webpackConfig)
    .then(stats => Promise.all([Promise.resolve(stats), webpackPromise(modernConfig)]))
    .then(([legacyStats, modernStats]) => {
        // 输出老版本打包
        console.log(legacyStats.toString({chunks: false, modules: false, colors: true}));
        // 输出 modern 版本打包
        console.log(modernStats.toString({chunks: false, modules: false, colors: true}));
    })
    .catch(e => console.log(e));
```

总结

本小节的实战内容主要讲解了怎么来实现 Vue-CLI 的 Modern Mode 模式打包，首先我们认识了如何利用 Babel 的配置打出现代浏览器执行的代码，然后我们讲到可以通过使用`<script type="module">`的方式来让现代浏览器执行 Modern 模式打包出来的文件，然后利用`<script nomodule>`来让现代浏览器忽略 legacy 的代码，这时候需要注意到 Safari 10 中不支持 script 标签的`nomodule`属性，需要添加 polyfill 代码。Legacy 和 Modern 代码打出来之后，需要做的是利用 html-webpack-plugin 的插件，将两次打包的 bundle 文件合并到一个 HTML 页面中，modern 的代码使用`<script type="module">`加载，legacy 的代码使用`<script nomodule>`方式加载，同时给 Safari 10 添加 polyfill。在插件实现上，我们直接使用了 ModernModePlugin，在 legacy 打包时，将`data.body`内容生成 JSON 存储起来，在 Modern 打包的时候，读取 JSON 内容，合并两次打包的 Bundle 资源，生成 HTML。本小节的实战内容可以直接在项目中实践，如果你现在的项目不支持 Modern 模式打包，你可以尝试使用本小节的代码给项目添加 Modern 模式打包。

> 戳此访问本小节源码：webpack-tutorial/packages/charpter-05/06-modern

> 本小节 Webpack 相关面试题：
>
> 1. Vue-CLI 3 中的 modern mode 是怎么实现的？
> 2. 如何让自己的项目在浏览器中直接执行 ES2015+ 代码？
> 3. 你能够说出 Vue-CLI 3 一个印象深刻的功能吗？

实战：使用 Stats 数据结构生成 Webpack 构建报告

Webpack 5.0

精选留言 0

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