实战：手写一个 prefetch-webpack-plugin 插件

更新时间：2019-07-10 14:31:25

> 人生太短，要干的事太多，我要争分夺秒。
>
> ——爱迪生

Webpack 的`plugin`是 Webpack 的核心概念，可以说整个 Webpack 都是由插件组成的。本章节讨论的内容是我们在配置文件配置的 plugin。这个是在整个工作流程的后半部分，Webpack 将整个模块的依赖关系都处理完毕，最终生成 bundle 的时候，然后扔给内置的插件和用户配置的插件依次处理。与`loader`只操作单个模块不同，`plugin`关注得是打包后的 bundle 整体，即所有模块组成的 bundle。所以跟产出相关的都是需要插件来实现的，比如压缩、拆分公共代码。

> 一句话：本质上来说，`plugin`就是通过监听 compiler 的某些 hook 特定时机，然后处理 stats。

在本章节中我们将讨论 Webpack 中不同插件的类型与区别。

插件开发必备的知识

首先我们来看下 Webpack 插件需要包含的几个条件：

- Webapck 的插件必须要是一个类；
- 该类必须包含一个`apply`的函数，该函数接收`compiler`对象参数；
- 该类可以使用 Webpack 的 compiler 和 Compilation 对象的钩子；
- 也可以自定义自己的钩子系统。

比如下面的函数就具备了上面的条件，所以它是可以作为一个 Webpack 插件的，下面是一个例子：

```
class MyPlugin {
    constructor(options) {
        // 自定义配置
        this.options = options;
        console.log(this.options);
    }
    apply(compiler) {
        compiler.hooks.done.tap('my-plugin', () => {
            console.log('Hello World!');
        });
    }
}
module.exports = MyPlugin;
```

我们在 webpack.config.js 中使用刚刚编写的插件，则可以直接`require`进去，然后使用`new`关键字来实例化我们的插件：

```
const MyPlugin = require('./myplugin');
module.exports = {
    // ...
    plugins: [new MyPlugin({options: true})]
};
```

> Tips：webpack 的插件实际是上一个包含`apply`方法的类。

如果我们想在指定 compiler 钩子时机执行某些脚本，自然可以在对应的事件钩子上添加回调方法，在回调里执行你所需的操作。由于 webpack 的钩子都是来自于`Tapable`类，所以一些特殊类型的钩子需要特殊的`tap`方法，例如 compiler 的`emit`钩子是支持`tap`、`tapPromise`和`tapAsync`多种类型的 tap 方式，但是不管哪种方式的 tap，都需要按照`Tapable`的规范来返回对应的值，下面的例子是使用了`emit.tapPromise`，则需要返回一个`Promise`对象。

```
class HelloWorldPlugin {
    apply(compiler) {
        compiler.hooks.emit.tapPromise('HelloAsyncPlugin', compilation => {
            // 返回一个 Promise，在我们的异步任务完成时 resolve……
            return new Promise((resolve, reject) => {
                setTimeout(function() {
                    console.log('异步工作完成……');
                    resolve();
                }, 1000);
            });
        });
    }
}

module.exports = HelloWorldPlugin;
```

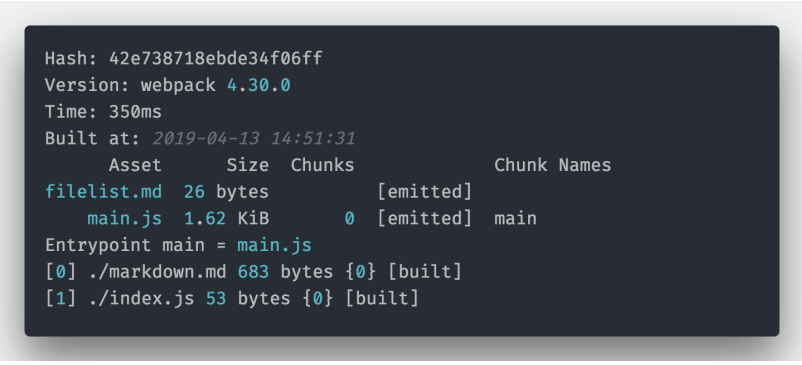
通过上面最简单的 Plugin，结合之前讲解的 Compiler 和 Compilation 对象的部分内容，相信大家已经大概明白了 Plugin 的工作原理。一旦我们深入理解了 webpack compiler 和每个独立的 compilation，我们就能通过 Webpack 引擎本身做到无穷无尽的事情。我们可以重新格式化已有的文件，创建衍生的文件，或者制作全新的生成文件。

官方插件分析：FileListPlugin

下面我们来看下 Webpack 官方给的插件 demo 源码，源码很简单。首先这里使用了异步的`emit.tapAsync`钩子，然后在`Compilation`对象上增加了一个`assets`文件`filelist.md`，内容就是我们获取到的`compilation.assets`的文件名（filename）：

```
class FileListPlugin {
    apply(compiler) {
        // emit 是异步 hook，使用 tapAsync 触及它，还可以使用 tapPromise/tap(同步)
        compiler.hooks.emit.tapAsync('FileListPlugin', (compilation, callback) => {
            // 在生成文件中，创建一个头部字符串：
            var filelist = 'In this build:\n\n';

            // 遍历所有编译过的资源文件，
            // 对于每个文件名称，都添加一行内容。
            for (var filename in compilation.assets) {
                filelist += '- ' + filename + '\n';
            }

            // 将这个列表作为一个新的文件资源，插入到 webpack 构建中：
            compilation.assets['filelist.md'] = {
                source: function() {
                    return filelist;
                },
                size: function() {
                    return filelist.length;
                }
            };

            callback();
        });
    }
}

module.exports = FileListPlugin;
```

我们将编写的代码放到了 webpack.config.js 配置中，直接执行后得到的命令行输出内容为：



通过 log 发现多输出了一个`filelist.md`的文件，然后我们打开`dist`文件夹中的这个文件下看内容是否符合我们的预期：

```
In this build:

-   main.js
```

看到内容后，符合我们的预期！

编写一个插件：prefetch-webpack-plugin

下面我们来编写个`prefetch-webpack-plugin`插件，这个插件的作用是将打包中遇见的`import()`或者`require.ensure()`这类异步懒加载的模块使用`<link>`标 签 的 `rel=prefetch`进行预加载，原理参考文档，这里不再就原理做深入介绍，简单来说就是将需要异步加载的模块，提前放到页面的 HTML 中进行预加载（需要浏览器支持）。

在写一个插件之前，我们需要了解这个插件需要用到的钩子有哪些。这里我们其实用到的是`compiler.compilation`和`html-webpack-plugin`的钩子，其实这个插件可以理解成是 html-webpack-plugin的插件，即一个 Webpack 插件的插件，这是因为 html-webpack-plugin 是处理 HTML 文件的插件，而且它本身也提供了钩子，我们可以从这些钩子中得到 HTML 的内容，从而修改 HTML 的页面结构。

在开始之前，继续介绍下 Webpack 的魔法注释，因为这个插件需要依赖魔法注释来标注一个模块是预取模块。

Webpack 的魔法注释 Prefetch

我们之前介绍过使用`/* webpackChunkName: 'name' */`这样的魔法注释给一个异步加载的模块添加名称，其实在 Webpack 4.6+ 版本中如果要实现 Prefetch 或者 Preload 标注，我们只需要使用魔法注释即可标注一个模块是否需要预取/预加载。

假如我们有个`lazy.js`模块需要 Prefetch，那么可以直接使用如下配置：

```
// 下面是魔法注释 （magic comments）
import(/* webpackPrefetch: true */ './lazy');
```

有了这个注释，在获取 chunk 对象的时候，就可以拿到它的这个标注，从而根据这个注释给页面增加`<link rel="prefetch">`标签。

> Tips：
>
> - `/* webpackPrefetch: true */`：把主加载流程加载完毕，在**空闲时**在加载其它，等再点击其他时，只需要从缓存中读取即可，性能更好，**推荐使用**；能够提高代码利用率，把一些交互后才能用到的代码写到异步组件里，通过懒加载的形式，去把这块的代码逻辑加载进来，性能提升，页面访问速度更快；
> - `/* webpackPreload: true */`: 和主加载流程一起并行加载。

prefetch-webpack-plugin 代码实现

下面简单说下原理和实现步骤：

1. 首先我们应该利用`compiler.compilation`这个钩子，得到`Compilation`对象；
2. 然后在`Compilation`对象中监听 html-webpack-plugin 的钩子，拿到 HTML 对象，这里需要区分 html-webpack-plugin 的版本：
    1. 在`3.x`版本，html-webpack-plugin 的钩子是直接挂在 Compilation 对象上的，我们使用的是`compilation.hooks.htmlWebpackPluginAfterHtmlProcessing`；
    2. 在`4.x`版本（截稿最新版本是 4.0-beta.3）中，html-webpack-plugin 自己使用`Tapable`实现了自定义钩子，需要使用`HtmlWebpackPlugin.getHooks(compilation)`的方式获取自定义的钩子。
3. 然后我们从`Compilation`对象中读取当前 HTML 页面的所有`chunks`，筛选异步加载的 chunk 模块，这里有两种情况：
    1. 生成多个 HTML 页面，那么 html-webpack-plugin 插件会设置`chunks`选项，我们需要从`Compilation.chunks`来选取 HTML 页面真正用到的 chunks，然后在从 chunks 中过滤出 Prefetch chunk；
    2. 如果是单页应用，那么不存在`chunks`选项，这时候默认`chunks='all'`，我们需要从全部`Compilation.chunks`中过滤出 Prefetch chunk。
4. 最后结合 Webpack 配置的`publicPath`得到异步 chunk 的实际线上地址，然后修改 html-webpack-plugin 钩子得

到的 HTML 对象，给 HTML 的`<head>`添加`<link rel="prefetch">`内容。

首先我们创建一个具有`apply`方法的类作为插件的结构，在`apply`中我们 tap `compiler`的`compilation`钩子获取`Compilation`对象：

```
class PrefetchPlugin {
    constructor() {
        this.name = 'prefetch-plugin';
    }
    apply(compiler) {
        compiler.hooks.compilation.tap(this.name, compilation => {
            // 得到 Compilation 对象了!
            console.log(compilation);
        });
    }
}
```

接下来我们需要结合html-webpack-plugin文档，获取页面 HTML 数据对象，这里我们根据步骤二，编写代码如下：

```
apply(compiler) {
    compiler.hooks.compilation.tap(this.name, compilation => {
        const run = this.run.bind(this, compilation);
        if (compilation.hooks.htmlWebpackPluginAfterHtmlProcessing) {
            // html-webpack-plugin v3 插件
            compilation.hooks.htmlWebpackPluginAfterHtmlProcessing.tapAsync(this.name, run);
        } else {
            // html-webpack-plugin v4
            HtmlWebpackPlugin.getHooks(compilation).beforeEmit.tapAsync(this.name, run);
        }
    });
}
```

上面代码，我们将实际处理的 HTML 数据的逻辑，扔给了 PreloadPlugin 这个类的`run`方法，在这里我使用了`bind`方式，保证了`this`的指向和第一个`compilation`参数的传入。而 html-webpack-plugin 的`htmlWebpackPluginAfterHtmlProcessing`和`beforeEmit`钩子实际是个`AsyncSeriesWaterfallHook`类型的钩子，所以需要使用`tapAsync`来绑定，然后需要执行异步回调的`callback`。下面来看下`run`函数的代码，在`run`函数中主要做了三件事情：

1. 我们需要获取 html-webpack-plugin 的配置，然后根据`chunks`的值从`Compilation.chunks`筛选当前 HTML 页面真正用到的`chunks`；
2. 从当前页面获取`chunks`中需要预取的 chunk；
3. 生成 prefetch link 标签，添加到 HTML 片段。

在`tapAsync` Hook 的 `run`中会得到三个参数：

- `compilation`：本次编译的 Compilation 对象；
- `data`：是 html-webpack-plugin 创建的一个给其插件使用的对象，里面包含页面的 HTML 判断和 html-webpack-plugin 插件实例化后的实例本身；
    - `data.html`：这个是生成 HTML 页面的 HTML 片段字符串；
    - `data.plugin`：这个是 html-webpack-plugin 的实例，可以从`data.plugin.options`读取 html-webpack-plugin 插件的参数。
- `callback`：`tapAsync`的回调函数，应该将`data`处理后的结果通过`callback`传递给下一个处理回调。

首先第一步，获取当前 HTML 页面真正用到的`chunks`：

```
run(compilation, data, callback) {
    // 获取 chunks，默认不指定就是 all
    const chunkNames = data.plugin.options.chunks || 'all';
    // 排除需要排除的 chunks
    const excludeChunkNames = data.plugin.options.excludeChunks || [];

    // 所有 chunks 的 Map，用于根据 ID 查找 chunk
    const chunks = new Map();
    // 预取的 id
    const prefetchIds = new Set();
    const curPageChunks = compilation.chunks
        .filter(chunk => {
            const {id, name} = chunk;
            // 添加到 map
            chunks.set(id, chunk);
            if (chunkNames === 'all') {
                // 全部的 chunks 都要过滤
                // 按照 exclude 过滤
                return excludeChunkNames.indexOf(name) === -1;
            }
            // 过滤想要的chunks
            return chunkNames.indexOf(name) !== -1 && excludeChunkNames.indexOf(name) === -1;
        });
    console.log(curPageChunks);
}
```

然后第二步是将 chunks 遍历，获取每个 chunk 的子模块（children），根据`chunk.getChildIdsByOrders`得到的`childIdByOrder`对象中的`prefetch`来判断有没有预取的模块，如果 chunk 中存在`/*webpackPrefetch: true*/`的模块，则可以得到`childIdByOrder.prefetch`数组，该数组中包含 chunk 中包含的 prefetch 的 chunkId，具体 run 部分的代码实现如下：

```
run(compilation, data, callback) {
    // 获取 chunks，默认不指定就是 all
    const chunkNames = data.plugin.options.chunks || 'all';
    // 排除需要排除的 chunks
    const excludeChunkNames = data.plugin.options.excludeChunks || [];

    // 所有 chunks 的 Map，用于根据 ID 查找 chunk
    const chunks = new Map();
    // 预取的 id
    const prefetchIds = new Set();
    compilation.chunks
        .filter(chunk => {
            const {id, name} = chunk;
            // 添加到 map
            chunks.set(id, chunk);
            if (chunkNames === 'all') {
                // 全部的 chunks 都要过滤
                // 按照 exclude 过滤
                return excludeChunkNames.indexOf(name) === -1;
            }
            // 过滤想要的chunks
            return chunkNames.indexOf(name) !== -1 && excludeChunkNames.indexOf(name) === -1;
        })
        .map(chunk => {
            const children = new Set();
            // 预取的内容只存在 children 内，不能 entry 就预取吧
            const childIdByOrder = chunk.getChildIdsByOrders();
            for (const chunkGroup of chunk.groupsIterable) {
                for (const childGroup of chunkGroup.childrenIterable) {
                    for (const chunk of childGroup.chunks) {
                        children.add(chunk.id);
                    }
                }
            }
            if (Array.isArray(childIdByOrder.prefetch) && childIdByOrder.prefetch.length) {
                prefetchIds.add(...childIdByOrder.prefetch);
            }
        });
    // 这里就是获取的 prefetch id 了
    console.log(prefetchIds)
}
```

最后，我们就需要处理 `data.html`，在 HTML 页面 `<head>` 标签添加 `link` 标签了：

```
run(compilation, data, callback) {
    // ... 忽略上面部分代码
    console.log(prefetchIds)
    // 获取 publicPath，保证路径正确
    const publicPath = compilation.outputOptions.publicPath || '';

    if (prefetchIds.size) {
        const prefetchTags = [];
        for (let id of prefetchIds) {
            const chunk = chunks.get(id);
            const files = chunk.files;
            files.forEach(filename => {
                prefetchTags.push(`<link rel="prefetch" href="${publicPath}${filename}">`);
            });
        }
        // 开始生成 prefetch html片段
        const prefetchTagHtml = prefetchTags.join('\n');

        if (data.html.indexOf('</head>') !== -1) {
            // 有 head，就在 head 结束前添加 prefetch link
            data.html = data.html.replace('</head>', prefetchTagHtml + '</head>');
        } else {
            // 没有 head 就加上个 head
            data.html = data.html.replace('<body>', '<head>' + prefetchTagHtml + '</head><body>');
        }
    }

    callback(null, data);
}
```

整个`PrefetchPlugin`的代码如下：

```
const HtmlWebpackPlugin = require('html-webpack-plugin');

class PrefetchPlugin {
    constructor() {
        this.name = 'prefetch-plugin';
    }
    apply(compiler) {
        compiler.hooks.compilation.tap(this.name, compilation => {
            const run = this.run.bind(this, compilation);
            if (compilation.hooks.htmlWebpackPluginAfterHtmlProcessing) {
                // html-webpack-plugin v3 插件
                compilation.hooks.htmlWebpackPluginAfterHtmlProcessing.tapAsync(this.name, run);
            } else {
                // html-webpack-plugin v4
                HtmlWebpackPlugin.getHooks(compilation).beforeEmit.tapAsync(this.name, run);
            }
        });
    }
    run(compilation, data, callback) {
        // 获取 chunks，默认不指定就是 all
        const chunkNames = data.plugin.options.chunks || 'all';
        // 排除需要排除的 chunks
        const excludeChunkNames = data.plugin.options.excludeChunks || [];

        // 所有 chunks 的 Map，用于根据 ID 查找 chunk
        const chunks = new Map();
        // 预取的 id
        const prefetchIds = new Set();
        compilation.chunks
            .filter(chunk => {
                const {id, name} = chunk;
                // 添加到 map
                chunks.set(id, chunk);
                if (chunkNames === 'all') {
                    // 全部的 chunks 都要过滤
                    // 按照 exclude 过滤
                    return excludeChunkNames.indexOf(name) === -1;
                }
                // 过滤想要的chunks
```

```
                return chunkNames.indexOf(name) !== -1 && excludeChunkNames.indexOf(name) === -1;
            })
            .map(chunk => {
                const children = new Set();
                // 预取的内容只存在 children 内，不能 entry 就预取吧
                const childIdByOrder = chunk.getChildIdsByOrders();
                for (const chunkGroup of chunk.groupsIterable) {
                    for (const childGroup of chunkGroup.childrenIterable) {
                        for (const chunk of childGroup.chunks) {
                            children.add(chunk.id);
                        }
                    }
                }
                if (Array.isArray(childIdByOrder.prefetch) && childIdByOrder.prefetch.length) {
                    prefetchIds.add(...childIdByOrder.prefetch);
                }
            });

        // 获取 publicPath，保证路径正确
        const publicPath = compilation.outputOptions.publicPath || '';

        if (prefetchIds.size) {
            const prefetchTags = [];
            for (let id of prefetchIds) {
                const chunk = chunks.get(id);
                const files = chunk.files;
                files.forEach(filename => {
                    prefetchTags.push(`<link rel="prefetch" href="${publicPath}${filename}">`);
                });
            }
            // 开始生成 prefetch html片段
            const prefetchTagHtml = prefetchTags.join('\n');

            if (data.html.indexOf('</head>') !== -1) {
                // 有 head，就在 head 结束前添加 prefetch link
                data.html = data.html.replace('</head>', prefetchTagHtml + '</head>');
            } else {
                // 没有 head 就加上个 head
                data.html = data.html.replace('<body>', '<head>' + prefetchTagHtml + '</head><body>');
            }
        }

        callback(null, data);
    }
}

module.exports = PrefetchPlugin;
```

写完了插件之后，我们写个 entry 和 webpack.config.js 来测试下插件：

```
// entry.js
import('./lazy').then(name => {
    console.log(name);
});
// webpack.config.js
const PrefetchPlugin = require('../PrefetchPlugin');
const HTMLWebpackPlugin = require('html-webpack-plugin');
module.exports = {
    mode: 'development',
    entry: './index.js',
    plugins: [new HTMLWebpackPlugin(), new PrefetchPlugin()]
};
```

打包之后，我们看到 log 输出了`0.js`这个异步加载的 chunk 文件，然后打开`index.html`看到内容中添加了`prefetch`内容：

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Webpack App</title>
  <link rel="prefetch" href="0.js">
</head>
  <body>
  <script type="text/javascript" src="main.js"></script></body>
</html>
```

为了验证我们的插件可用，我们通过注释给`import()`异步加载进来的模块命名为`lazy`，然后修改 webpack.config.js 的`output.publicPath`，看下输出的 HTML 中 prefetch 地址是否是正确的地址：

```
// index.js
import(/* webpackChunkName: "lazy", webpackPrefetch: true */ './lazy').then(name => {
    console.log(name);
});
// test.js，为了对比我们添加一个test entry，用于多页面的对比
console.log('test file');

//webpack.config.js
module.exports = {
    // ...
    // 多 entry 入口
    entry: {
        index: './index.js',
        test: './test.js'
    },
    output: {
        publicPath: 'http://www.example.com/js/'
    },
    plugins: [
        // 使用 chunks，index 中有引入 index -> prefetch lazy.js
        new HTMLWebpackPlugin({chunks: ['index'], filename: 'index.html'}),
        // 没有 chunk 测试，chunks='all'，test 和 index 都会被包含，index -> prefetch lazy.js
        new HTMLWebpackPlugin({filename: 'no-chunk.html'}),
        // chunks=[test]，test 中没有 prefetch 的内容，所以 html 应该不会包含 prefetch link
        new HTMLWebpackPlugin({chunks: ['test'], filename: 'test.html'}),
        new PrefetchPlugin({options: true})
    ]
};
```

最后我们发现修改之后的代码，得到正确的内容：`<link rel="prefetch" href="http://www.example.com/js/lazy.js">`，说明我们的插件没有问题。

> Tips：这里插件只处理了`/*webpackPrefetch:true*/`的情况，`/*webpackPreload:true*/`的情况请读者自己动手来实现吧！

总结

本小节主要介绍了 Webpack 的插件编写时候涉及到的知识点，最后剖析了 Webpack 官方的 FilelistPlugin 的插件，最后我们动手实现了一个将异步加载模块给 HTML 添加`prefetch`实现预加载内容的插件。编写插件之前应该先理解插件需要做的事情，然后根据前边介绍的 Compiler 和 Compilation 对象的钩子章节，寻找合适的事件注入时机，然后得到对应的钩子回调参数，最后处理数据。

> 本小节 Webpack 相关面试题：
>
> 1. 编写过 Webpack 插件吗？

实战：手写一个 markdown-loader

实战：使用 Express 和中间件来实现 Webpack-dev-server