Webpack 的模块热替换做了什么？

更新时间：2019-07-30 10:41:19

> 不安于小成，然后足以成大器；不诱于小利，然后可以立远功。
>
> ——方孝孺

Webpack 的模块热替换（HMR - Hot Module Replacement，又称为热替换、热更新等）是 Webpack 最令人兴奋的特性之一。在没有 HMR 之前，Web 前端开发者使用类似 LiveReload 这类工具配合浏览器插件监听文件变化，然后**重新加载整个页面**。当 Webpack 开启了 HMR 功能之后，我们的代码修改之时，Webpack 会重新打包，并且将修改后的代码发送到浏览器，浏览器替换老的代码，保证了页面状态不会丢失，在不刷新整个页面的前提下进行局部的更新。

所谓保证页面状态，就是在不修改页面当前状态的情况下进行替换。举例来说，当我们打开一个页面，进行了很多步的操作（比如点击了页面的一个按钮），这时候页面弹出一个弹层，我们发现弹层的背景颜色不对，这时候我们可以直接修改代码，HMR 最终会在不刷新页面的前提下直接修改弹层的背景颜色，效果跟我们在 Chrome Devtools 内修改 CSS 样式一样。

本篇文章先带领大家体验一下 HMR 的整个执行过程，然后深入到代码细节来看下 Webpack HMR 的实现原理。

Webpack HMR 的流程

首先我们需要创建一个 HMR 的项目，项目的文件列表如下：

```
├── package.json
├── src
│   ├── index.html
│   ├── index.js
│   └── style.css
└── webpack.config.js
```

项目入口文件是`src/index.js`，其中引入了`src/style.css`，并且给 HTML 页面的`<h1 id="app">`的增加`innerHTML`内容，具体内容如下：

```
// index.js
import './style.css';
const $node = document.getElementById('app');
$node.innerHTML = 'Hi, Webpack HMR';
```

`src/style.css`的内容如下：

```
body {
    background: greenyellow;
}
```

`src/index.html`内容如下：

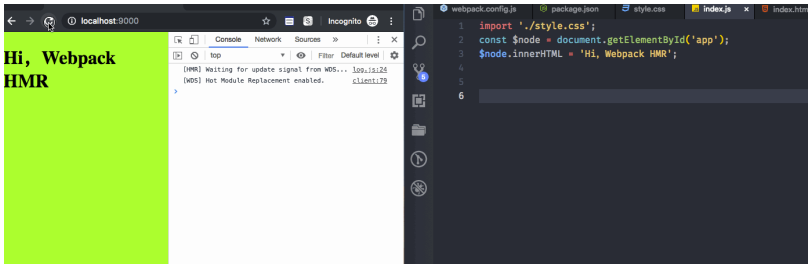
```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8" />
        <meta name="viewport" content="width=device-width, initial-scale=1.0" />
        <meta http-equiv="X-UA-Compatible" content="ie=edge" />
        <title>Dev Server</title>
    </head>
    <body>
        <h1 id="app"></h1>
    </body>
</html>
```

在`webpack.config.js`内容中，增加`html-webpack-plugin`和 CSS 使用到的 loader，并且设置`devServer.hot=true`开启 webpack-dev-server 的 HMR 功能，最后在插件内配上`HotModuleReplacementPlugin`，这样 Webpack 的 HMR 功能就配置完毕了：

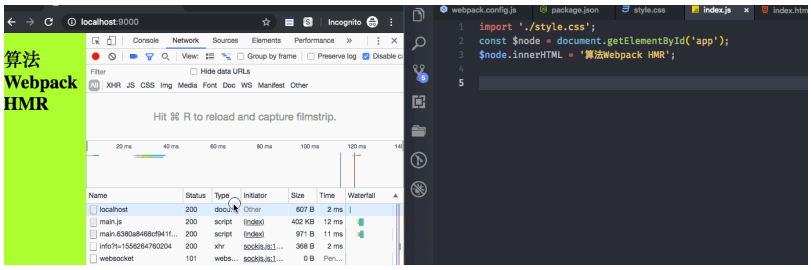
```
// webpack.config.js
const path = require('path');
const webpack = require('webpack');
const HtmlWebPackPlugin = require('html-webpack-plugin');

module.exports = {
    entry: './src/index.js',
    devServer: {
        contentBase: path.join(__dirname, 'dist'),
        port: 9000,
        hot: true
    },
    module: {
        rules: [{test: /\.css$/, use: [{loader: 'style-loader'}, {loader: 'css-loader'}]}]
    },
    plugins: [
        new HtmlWebPackPlugin({
            template: 'src/index.html',
            filename: 'index.html',
            inject: true
        }),
        new webpack.HotModuleReplacementPlugin()
    ]
};
```

配置完毕后，我们使用 npx 启动`webpack-dev-server --open`，这时候浏览器会自动打开我们的 HTML 首页：`http://localhost:9000`。我们修改`index.js`和`style.css`的内容，则浏览器的页面也进行变化，具体效果可以参考下面的 Gif 动图：



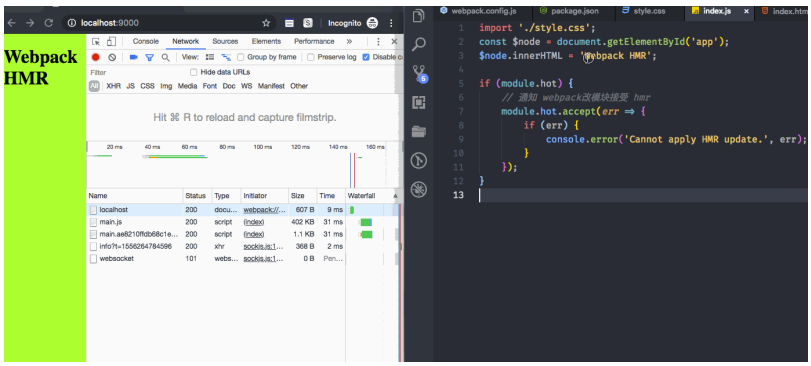
这时候 HMR 并不是真正的局部刷新，而是重新加载，详细请看下面的 Gif 动图。注意观察 Chrome DevTools 的 Network 面板请求是重新加载的，而不是增量加载：



要解决这个问题，就需要我们在`src/index.js`增加一段代码：

```
// index.js
// 忽略之前的内容。。
if (module.hot) {
    // 通知 webpack改模块接受 hmr
    module.hot.accept(err => {
        if (err) {
            console.error('Cannot apply HMR update.', err);
        }
    });
}
```

增加这段代码之后，再来看 HMR 的局部更新效果就实现了，具体看下面的 Gif 动图：



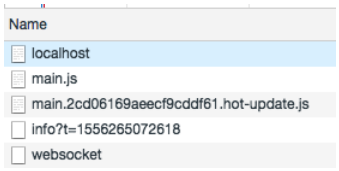
接下来我们来看下 HMR 的具体流程是怎样的。

HMR 流程

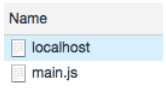
现在我们启动 `webpack-dev-server` 之后，我们先来看下 Webpack 打包的 log ：

```
ℹ ｢wdm｣: Compiled successfully.
ℹ ｢wdm｣: Compiling ...
ℹ ｢wdm｣: Hash: c243117d7cba3d4c4390
Version: webpack 4.30.0
Time: 28ms
Built at: 04/27/2019 3:47:04 PM
                                Asset       Size  Chunks             Chunk Names
   2cd06169aeecf9cddf61.hot-update.json   46 bytes          [emitted]
                           index.html  378 bytes          [emitted]
main.2cd06169aeecf9cddf61.hot-update.js  900 bytes    main  [emitted]  main
                             main.js    402 KiB    main  [emitted]  main
Entrypoint main = main.js main.2cd06169aeecf9cddf61.hot-update.js
```

从 log 中可以看出我们的`entry`打包出来了两个文件：`main.js`和`main.2cd06169aeecf9cddf61.hot-update.js`。这时候我们打开 Chrome 浏览器访问`http://localhost:9000`，看下 Chrome 的 DevTools 的 Network 面板，查看页面的请求：



通过观察发现，跟我们不使用 HMR 功能相比，我们页面请求由`index.html`和`main.js`两个请求（下图是没有 HMR 的正常请求），变成了 5 个请求。



5 个请求的 URL 分别是：

```
1. http://localhost:9000/
2. http://localhost:9000/main.js
3. http://localhost:9000/main.2cd06169aeecf9cddf61.hot-update.js
4. http://localhost:9000/sockjs-node/info?t=1556265072618
5. ws://localhost:9000/sockjs-node/670/rinymwto/websocket
```

5 个请求中`main.js`和`main.2cd06169aeecf9cddf61.hot-update.js`合起来是入口文件打包后的结果，这部分包含了 HMR 的 Runtime 和`src/index.js`的便以结果。其他 3 个请求中包含了一个 WebSocket 请求（`ws://`协议）。这是因为 HMR 是首先使用`sockjs`这个模块来创建一个 WebSocket 长连接来跟 Webpack 的打包服务通信的。

这时候我们修改`src/index.js`的内容：

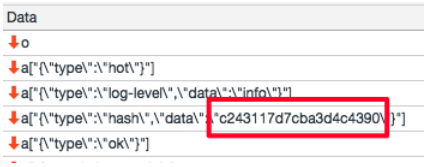
```
// src/index.js
- $node.innerHTML = 'Webpack HMR'; // 将 Webpack HMR 字符串修改为 Hi，Webpack HMR
+ $node.innerHTML = 'Hi，Webpack HMR';
```

在终端中，log 发生了变化：

```
ℹ ｢wdm｣: Compiled successfully.
ℹ ｢wdm｣: Compiling...
ℹ ｢wdm｣: Hash: a0cfc7b18b511cb79db4
Version: webpack 4.30.0
Time: 46ms
Built at: 04/27/2019 3:51:19 PM
                                  Asset       Size  Chunks             Chunk Names
    c243117d7cba3d4c4390.hot-update.json  46 bytes          [emitted]
                              index.html  378 bytes          [emitted]
main.c243117d7cba3d4c4390.hot-update.js  903 bytes    main  [emitted]  main
                                 main.js    402 KiB    main  [emitted]  main
Entrypoint main = main.js main.c243117d7cba3d4c4390.hot-update.js
[./src/index.js] 280 bytes {main} [built]
```

生成了新的打包之后的文件，包括：`main.c243117d7cba3d4c4390.hot-update.js` 和 `c243117d7cba3d4c4390.hot-update.json` 等。

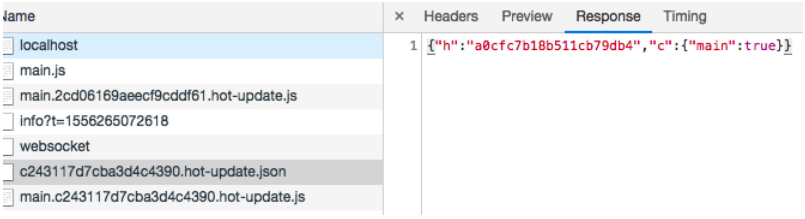
这时候我们观察 Chrome 的请求，首先是 WebSocket 的消息中收到了一条推送内容：




```
{"type":"hash","data":"c243117d7cba3d4c4390"}
```

注意这个 `data` 的值，就是咱们打包后的文件名 **hash**！然后 Chrome 发送了两个请求：

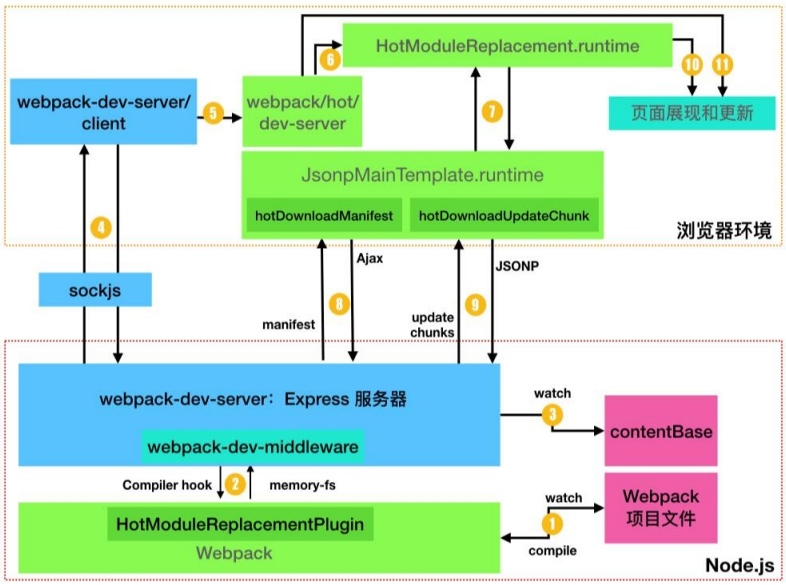
```
http://localhost:9000/c243117d7cba3d4c4390.hot-update.json
http://localhost:9000/main.c243117d7cba3d4c4390.hot-update.js
```



这两个请求的就是更新下来的更新的代码，下面我们来看下 Chrome 一开始加载的 `main.2cd06169aeecf9cddf61.hot-update.js` 文件和后来请求过来的 `main.c243117d7cba3d4c4390.hot-update.js` 内容，比较下 `$node.innerHTML` 内容就是我们刚刚修改 `src/index.js` 的内容了：

```
// http://localhost:9000/main.2cd06169aeecf9cddf61.hot-update.js
webpackHotUpdate('main', {
    './src/index.js': function(module, __webpack_exports__, __webpack_require__) {
        'use strict';
        eval(
            "__webpack_require__.r(__webpack_exports__);\n/* harmony import */ var _style_css__WEBPACK_IMPORTED_MODULE_0__
= __webpack_require__(/*! ./style.css */ \"./src/style.css\");\n/* harmony import */ var _style_css__WEBPACK_IMPORTED_MODUL
E_0___default = /*#__PURE__*/__webpack_require__.n(_style_css__WEBPACK_IMPORTED_MODULE_0__);\n\nconst $node = document.getE
lementById('app');\n$node.innerHTML = 'Webpack HMR';\n\nif (true) {\n    // 通知 webpack改模块接受 hmr\n    module.hot.acce
pt(err => {\n        if (err) {\n            console.error('Cannot apply HMR update.', err);\n        }\n    });\n}\n\n\n//
# sourceURL=webpack:///./src/index.js?"
        );
    }
});
// http://localhost:9000/main.c243117d7cba3d4c4390.hot-update.js
webpackHotUpdate('main', {
    './src/index.js': function(module, __webpack_exports__, __webpack_require__) {
        'use strict';
        eval(
            "__webpack_require__.r(__webpack_exports__);\n/* harmony import */ var _style_css__WEBPACK_IMPORTED_MODULE_0__
= __webpack_require__(/*! ./style.css */ \"./src/style.css\");\n/* harmony import */ var _style_css__WEBPACK_IMPORTED_MODUL
E_0___default = /*#__PURE__*/__webpack_require__.n(_style_css__WEBPACK_IMPORTED_MODULE_0__);\n\nconst $node = document.getE
lementById('app');\n$node.innerHTML = 'Hi, Webpack HMR';\n\nif (true) {\n    // 通知 webpack改模块接受 hmr\n    module.hot.a
ccept(err => {\n        if (err) {\n            console.error('Cannot apply HMR update.', err);\n        }\n    });\n}\n\n\
n//# sourceURL=webpack:///./src/index.js?"
        );
    }
});
```

通过上面的全流程观察，我们可以总结出来 Webpack HMR 的全流程图如下所示：



上面的流程图展现了 HMR 的一个完整周期，整个周期分为两部分：**启动阶段**和**文件监控更新流程**。

在启动阶段，Webpack 和 webpack-dev-server 进行交互。Webpack 和 webpack-dev-server 主要是通过 Express 的中间件 webpack-dev-middleware进行交互，这个阶段可以细分为以下几个步骤：

1. webpack-dev-server 启动 Webpack 打包的 watch 模式，在这种模式下 Webpack 会监听文件的变化，一旦有文件发生变化，则会重新进行打包，watch 模式下 Webpack 打包的结果不会落盘（保存到硬盘上）；
2. webpack-dev-server 通过 webpack-dev-middleware与 Webpack 进行交互，Webpack-dev-middleware 初始化会接收 Webpack 的 `Compiler` 对象，通过`Compiler`的钩子可以监听 Webpack 的打包过程；
3. 如果`devServer.watchContentBase=true`，则 webpack-dev-server 监听文件夹中静态文件的变化，发生变化则通知浏览器刷新页面重新请求新的文件；

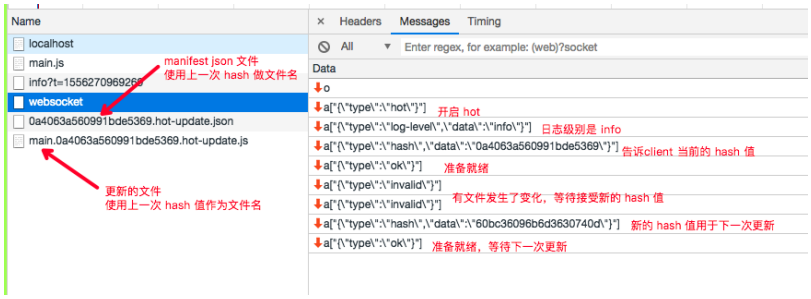
4. 打开浏览器之后，webpack-dev-server 会利用`sockjs`在浏览器和 Server 之间创建一个 WebSocket 长连接，这个长连接是浏览器和 webpack-dev-server 的通信桥梁，它们之间的通信内容主要是传递编译模块的文件信息（hash 值），这时候如果 Webpack 监控的文件发生了修改，`webpack/hot/dev-server`来实现 HMR 更新还是刷新页面。

> Tips：
>
> 1. webpack-dev-server 的 contentBase 可以理解为静态资源服务器的目录文件夹，启动 server 之后，可以通过`网址+电脑中文件路径`的方式访问到具体文件，这个文件跟 Webpack 打包出来的路径并不一样；
> 2. 这里有两个文件变化的监控，第一步中 Webpack 监控整个依赖模块的文件变化，发生变化则重新出发 Webpack 编译；第三步中 webpack-dev-server 自己监控`contentBase`的文件变化，文件发生变化则通知浏览器刷新页面，这里是刷新页面并不是 HMR，这是因为`contentBase`内容是非 Webpack 打包的依赖文件。
> 3. WebSocket 需要服务端和浏览器端都有对应的创建连接代码（`new WebSocket`），webpack-dev-server 在浏览器中通过在 chunks 中插入`webpack-dev-server/client`这个文件来创建 WebSocket 通信。

到此启动阶段结束，当 Webpack 监控的文件发生变化之后，这时候就进入了文件监控更新流程，当 Webpack 监控的依赖图中的某个文件修改之后：

1. Webpack 会重新编译文件，这时候我们在`webpack.config.js`中添加的插件`HotModuleReplacementPlugin`会生成两次编译之间差异文件列表（manifest）文件`[hash].hot-update.json`，这个 manifest JSON 文件包含了变化文件的 **Update** 内容，即`[id].[hash].hot-update.js`。webpack-dev-server 中的 webpack-dev-middler 会通过 Webpack 的`Compiler`钩子监听打包进程，然后通知 webpack-dev-server 使用 WebSocket 长连接推送编译之后的 hash 值；
2. 除了发送编译后 Hash 值之外，webpack-dev-server 还会通过长连接告诉浏览器当前的页面代码是**`invalid`**状态的，需要更新新的代码；
3. 浏览器拿到 Hash 之后，会首先发起一个 Ajax 请求 manifest 文件`[hash].hot-update.json`文件内容；
4. manifest 列表文件内容拿到之后，会告诉 HMR 的 Runtime 请求那些变化的 JavaScript 文件，这时候会 Runtime 会按照清单列表发起 JSONP 请求，将两次编译的差异文件`[id].[hash].hot-update.js`获取下来，插到页面`head`标签的`script`中执行，最终完成了更新的全流程。

WebSocket 消息的含义如下图所示：



> Tips：
>
> 1. 这里的 Hash 值为 Compilation 的 hash 值，获取 manifest 和更新文件时用的是上一次的 hash 值；
> 2. manifest 和 update 文件名可以通过 Webpack 的`output.hotUpdateMainFilename`和`output.hotUpdateChunkFilename`来设置。

小结一下

整个过程虽然步骤比较多，而且涉及模块比较复杂，这里先小结下：

- webpack-dev-server：启动一个 Express Server，整合 webpack-dev-middleware 中间件、WebSocket 长连接、proxy、静态资源服务器等功能；
  - webpack-dev-middleware：跟 Webpack 进行交互，通过`Compiler`的 Hook 来监控打包流程，保证文件修改后打包结束后请求新的文件，上线一个内存型的文件系统，文件直接从内存读取可以提升 webpack-dev-server 的速度。
- `HotModuleReplacementPlugin`：插件是用来生成 HMR 的文件清单列表和差异文件的：
  - manifest 文件：JSON 文件，文件名格式为`[hash].hot-update.json`，包含所有需要更新的文件信息；
  - update 文件：需要更新的 JavaScript 文件，文件名格式为`[id].[hash].hot-update.js`，包含 HMR 的差异化执行代码。

Webpack HMR 原理

HMR 的全流程我们已经梳理清楚了，下面就是 HMR 实现的技术细节了，要理解 HMR 的技术细节，只需要回答下面 4 个问题即可：

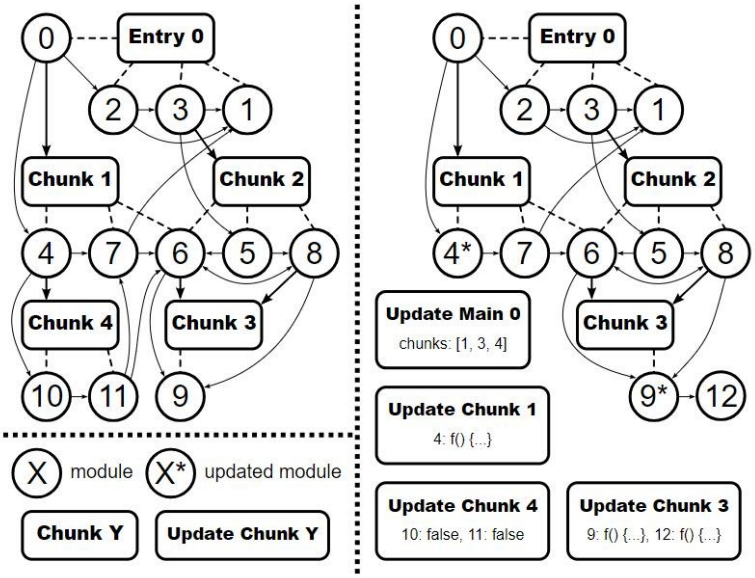
1. webpack-dev-middleware 是怎么让 Webpack 打包出来的文件存入内存的？

在 webpack-dev-middleware 中，使用了memory-fs这个内存文件系统模块，然后将 Webpack 的`Compiler`对象的`Compiler.outputFileSystem`替换掉，`Compiler.outputFileSystem`就是输出打包结果文件的：

```
// webpack-dev-middleware/lib/fs.js
const MemoryFileSystem = require('memory-fs');
module.exports = {
    setFs(context, compiler) {
        //.. 忽略
        fileSystem = new MemoryFileSystem();
        compiler.outputFileSystem = fileSystem;
    }
};
```

2. HotModuleReplacementPlugin 怎么计算两次编译差量文件的？

这里 Webpack wiki 中有一张比较形象的图：



上图中，右边的 moduleID 为`4`和`9`的模块发生了变化，那么依赖两个模块的 4 个 Chunk 文件就需要更新，这 4 个 chunks 被放入了`manifest` JSON 文件中：

- `moduleID=9`模块修改后：引入了新的依赖`12`，所以`chunk=3`生成了包含`9`和`12`的 update 文件；
- `moduleID=4`模块修改后：
  - 去掉了`chunk=4`的依赖，所以`chunk=4`中的`10`和`11`没有其他的模块使用，所以删除掉；
  - 依赖`moduleID=4`模块的`chunk=1`需要更新`4`更新的内容；
- 最后是入口文件`entry=0`文件依赖`chunks=[1,3,4]`需要更新。

> Tips：每个模块（module）都可以通过它的`parents`和`children`属性找到自己被谁依赖和依赖谁。

3. HMR Runtime 是根据怎样的规则去拉取新代码的？

HMR Runtime 指的是`HotModuleReplacementPlugin`插件中，通过`Compilation.mainTemplate`的`bootstrap`钩子注入的`lib/HotModuleReplacement.runtime.js`文件：

```
// webpack 4.30.0
// lib/HotModuleReplacementPlugin.js
const hotInitCode = Template.getFunctionContent(
    // lib/HotModuleReplacement.runtime.js内容
    require('./HotModuleReplacement.runtime')
);
// hook 钩子注入
mainTemplate.hooks.bootstrap.tap('HotModuleReplacementPlugin', (source, chunk, hash) => {
    source = mainTemplate.hooks.hotBootstrap.call(source, chunk, hash);
    return Template.asString([
        source,
        '',
        hotInitCode
            .replace(/\$require\$/g, mainTemplate.requireFn)
            .replace(/\$hash\$/g, JSON.stringify(hash))
            .replace(/\$requestTimeout\$/g, requestTimeout)
            .replace(
                /\/\*foreachInstalledChunks\*\//g,
                needChunkLoadingCode(chunk)
                    ? 'for(var chunkId in installedChunks)'
                    : `var chunkId = ${JSON.stringify(chunk.id)};`
            )
    ]);
});
```

这个 Runtime 中会调用`mainTemplate`根据不同环境注入的`runtime.js`。在浏览器环境下，注入的是`lib/web/JsonpMainTemplate.runtime.js`，这里有 2 个跟更新有关的函数，分别是：

- `hotDownloadManifest`：发起 Ajax 请求 mainifest JSON 文件；
- `hotDownloadUpdateChunk`：创建 JSONP 请求 update 文件。

详细代码可以参见`lib/web/JsonpMainTemplate.runtime.js`内容，其他 Node.js 执行环境、WebWorker 执行环境类似。

4. 如何让自己的代码支持 HMR?

我们在学习 DOM 事件的时候知道，DOM 事件会顺着 DOM 树进行冒泡。如果当前的节点不停止冒泡，则事件会往其父节点继续冒泡，一直到根节点。在 Webpack 的 HMR 处理上，也是有这个冒泡过程的。Webpack 的模块实际也有一个 module 树，module 树的根节点就是入口文件（entry），当我们一个模块代码发生了更改，就需要执行 update 事件，如果当前模块处理不了这个事件，即不知道怎么实现 HMR，那么会冒泡到父依赖节点，直到有模块可以处理 HMR 更新代码，如果到了根节点（entry）都没有处理 update 事件，就会刷新页面。

因为我们的代码究竟能不能实现 HMR 只有编写者知道，比如我们一个 Vue 应用，修改了 Vue 组件的`template`和`state`肯定是不同的处理方式，所以不能一概而论地都给所有的 JavaScript 模块添加`module.hot.accept()`，要视情况而定。

幸运的是大多数框架，像 React、Vue、Angular 都有自己的 HMR 工具。这些工具有的是通过 loader 的方式来实现，有的是通过 Babel 插件方式来实现的。另外要实现 Less、Sass、CSS 文件的热加载，我们可以直接使用 style-loader 来完成（生产环境打包模式下不建议使用），比如在开发模式下对于 CSS 的加载可以配置如下的 loader：

```
module.exports = {
    module: {
        rules: [
            {
                test: /\.css$/,
                use: ['style-loader', 'css-loader']
            }
        ]
    }
};
```

接下来介绍下如果要自己编写 HMR 的代码，那么需要用到的几个 Webpack 的扩展函数方法：

- `module.hot.accept()`
- `module.hot.decline()`
- `module.hot.dispose()`
- `module.hot.status()`
- `module.hot.apply()`

`module.hot.accept()`函数

`module.accept()`有两种使用方式，第一种是在使用的时候，声明对应依赖模块，当声明的依赖模块的更新后才会执行对应的回调，这种方式的使用方式如下：

```
module.hot.accept(
    dependencies, // 可以是一个字符串或字符串数组
    callback // 用于在模块更新后触发的函数
);
```

这里的`callback`会拿到更新的依赖模块，即`callback(updatedDependencies)`。

`accept`的第二种用法是自身更新：

```
module.hot.accept(
    errorHandler // 在计算新版本时处理错误的函数
);
```

在这种用法中，当前模块**所有**依赖的模块代码会被更新，而且这种更新不会冒泡到父级模块中去。如果此模块没有导出（`export`）的情况下有用（这是因为没有导出，所以也就没有父级调用它）。

`module.hot.decline()`函数

`module.hot.decline()`函数接受一个模块依赖列表，表示拒绝这些依赖模块的更新：

```
module.hot.decline(
    dependencies // 可以是一个字符串或字符串数组
);
```

这时候我们在父模块可以获取到错误代码`decline`。通常这种情况下，我们没法处理 HMR 模块，只能重新加载页面。

`module.hot.dispose()` 函数

`module.hot.dispose()`是给当前模块代码添加一个处理函数，当前模块代码被替换时会执行对应的处理函数回调。被添加的函数应该用于移除你声明或创建的任何持久资源。如果要将状态传入到更新过的模块，请添加给定 data 参数。更新后，此对象在更新之后可通过 module.hot.data 调用：

```
module.hot.dispose(data => {
    // 清理并将 data 传递到更新后的模块……
});
```

举个例子说明，我们有个模块中存在一个全局变量`globalId`。模块更新的时候，我们希望重新定义这个变量，或者将这个变量放到 `data` 中，下次可以通过`module.hot.data`进行访问，那么我们可以写如下代码：

```
if (module.hot) {
    module.hot.accept();
    // dispose handler
    module.hot.dispose(data => {
        // 使用，这时候 globalId 是更新之前的，而不是更新之后的变量值
        console.log(globalId);
        // 重新定义
        globalId = 999;
        // 放到 data
        data.globalId = globalId;
    });
}
```

> Tips：`dispose`是`addDisposeHandler`的 alias，所以`module.hot.dispose`效果等同于`module.hot.addDisposeHandler()`，而要移除这个回调，可以使用`module.hot.removeDisposeHandler(callback)`。

`module.hot.status()`获取 HMR 状态

在 HMR 中，我们可以使用`module.hot.status()`获取 HMR 的状态，状态有以下几种：

- `idle`：当前 HMR 处于空闲状态，可以调用 `module.hot.check`方法检测更新情况，调用`module.hot.check`后状态为 `check`；
- `check`： HMR 正在检查模块更新，如果没有模块更新，那么重新回到 `idle` 状态。如果有更新那么会依次经过 `prepare`、`dispose`、`apply` 然后重新回到 `idle` 状态；
- `watch`和`watch-delay`：`watch`表明 HMR 当前处于监听模式，可以自动接收到更新。如果接受到更新，那么就会进入 `watch-delay` 模式，然后等待机会开始更新操作。如果开始更新，那么会依次经过 `prepare`、`dispose`、`apply`状态。如果在更新的时候又监听到文件更新，那么重新回到 `watch` 或者 `watch-delay`状态；
- `prepare`：表明 HMR 在准备更新。比如正在下载 Webpack 更新后的一些资源用于更新；
- `ready`： 可以开始更新了，需要手动调用 `apply` 方法去继续更新操作；
- `dispose`： HMR 在调用模块自己的 `dispose` 方法，并开始更新后的模块替换操作；
- `apply`： HMR 在调用被替换后（`dispose`）的模块的父级模块的 `accept` 方法，当然模块自己必须能够被 `dispose`；
- `abort`： 更新无法被进一步 `apply`，但是文件处于更新之前的一致状态；
- `fail`： 在更新过程中抛出了异常，当前的文件状态处于不一致状态，系统需要重启。

我们还可以通过注入`status`的监听回调对 HMR 的`status`进行监听：

```
module.hot.addStatusHandler(status => {
    // 响应当前状态……
});
// 移除监听
module.hot.removeStatusHandler(callback);
```

`module.hot.apply()`函数

`module.hot.apply()`函数用于主动触发更新流程，调用方式如下：

```
module.hot
    .apply(options)
    .then(outdatedModules => {
        // 过期的模块……
    })
    .catch(error => {
        // 捕获错误
    });
```

其中 `options` 的参数有：

- `ignoreUnaccepted`：调用 `accept` 时没有指定的模块。如果 accpet 没有参数，接受任何模块更新；
- `ignoreDeclined` ：调用 `decline` 明确指定不需要检查的模块；
- `ignoreErrored` ：忽略在调用`accept`` 时抛出的错误；
- `onDeclined`：接收 `decline` 指定的模块的回调函数；
- `onUnaccepted`：接收 `accept` 中没有指定的模块的回调；
- `onAccepted`：接收 `accept` 中指定的模块的回调；
- `onDisposed`：接收被 `dispose` 的模块的回调；
- `onErrored`：接收出错的模块回调。

每一个函数接受到的参数为如下类型：

```
{
    type: "self-declined" | "declined" |
        "unaccepted" | "accepted" |
        "disposed" | "accept-errored" |
        "self-accept-errored" | "self-accept-error-handler-errored",
    moduleId: 4,
    // The module in question.
    dependencyId: 3,
    // For errors: the module id owning the accept handler.
    chain: [1, 2, 3, 4],
    // For declined/accepted/unaccepted: the chain from where the update was propagated.
    // 这个 chain 表示更新冒泡的顺序
    parentId: 5,
    // For declined: the module id of the declining parent
    outdatedModules: [1, 2, 3, 4],
    // For accepted: the modules that are outdated and will be disposed
    outdatedDependencies: {
    // For accepted: The location of accept handlers that will handle the update
    5: [4]
    },
    error: new Error(...),
    // For errors: the thrown error
    originalError: new Error(...)
    // For self-accept-error-handler-errored:
    // the error thrown by the module before the error handler tried to handle it.
}
```

总结

本文主要讲解 Webpack 的 Hot Module Replacement 流程和实现原理。先从实际项目复习了 HMR 的用法和体验了 HMR 的流程，然后详细讲解了 HMR 的整个流程中各个模块做的事情，最后在原理上通过解答 3 个问题来加深对 HMR 原理的理解。webpack-dev-server 虽然可以直接来启动 HMR，但是真正核心的是 webpack-dev-middleware。webpack-dev-server 除了这个中间件之外主要功能就是个静态服务器，而后面实战部分我会使用 Express、webpack-dev-middleware 自己来实现 「webpack-dev-server」，通过实战加深 HMR 原理 和 webpack-dev-server 功能理解。

本小节 Webpack 相关面试题：

1. 怎么配置 Webpack 的热替换更新？
2. Webpack 热替换是怎么实现的？

从 Webpack 的产出代码来看 Webpack 是怎么执行的

实战：使用 PostCSS 打造移动适配方案

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