14 Webpack Dev Server 本地开发服务

更新时间：2019-06-24 09:27:11

> 不想当将军的士兵，不是好士兵。
>
> ——拿破仑

webpack-dev-server是一个基于 Express 的本地开发服务器（看 Roadmap 下个版本内核会从 Express 切换到 Koa）。它使用 webpack-dev-middleware 中间件来为通过 Webpack 打包生成的资源文件提供 Web 服务。它还有一个通过 Socket IO 连接着 webpack-dev-server 服务器的小型运行时程序。webpack-dev-server 发送关于编译状态的消息到客户端，客户端根据消息作出响应。

> Tips：简单来说 webpack-dev-server 就是一个 Express 的小型服务器，它是通过 Express 的中间件 webpack-dev-middleware和 Webpack 进行交互的。所以我们如果自己的项目本身就是个 Express 服务器，那么可以使用 webpack-dev-middleware 和 webpack-hot-middleware 两个中间件来实现 HMR 功能。关于 webpack-dev-middleware 和 webpack-hot-middleware 中间件来实现 webpack-dev-server 的内容，在后续章节还会有介绍。

命令行

webpack-dev-server 安装之后，会提供一个 bin 命令行，通过命令行可以启动对应的服务。

```
# 项目中安装 webpack-dev-server
npm i webpack-dev-server
# 使用 npx 启动
npx webpack-dev-server
```

执行`webpack-dev-server`命令之后，它会读取 Webpack 的配置文件（默认是 webpack.config.js）然后将文件打包到内存中（所以看不到`dist`文件夹的生产，Webpack 会打包到硬盘上），这时候打开 server 的默认地址：`localhost:8080`就可以看到文件目录或者页面（默认是显示 index.html，没有则显示目录）。

跟`webpack-cli`一样，`webpack-dev-server`也有一些选项可以添加：

```
# 修改端口号和 host
webpack-dev-server --port 3000 --host 127.0.0.1
# 启动inline 模式的自动刷新
webpack-dev-server --hot --inline
# 手动指定 webpack config 文件
webpack-dev-server --config webpack.xxx.js
# 指定 webpack 的 mode
webpack-dev-server --mode development
# watch 功能，文件发生变化则触发重新编译
webpack-dev-server --watch
# dev-server默认会将工作目录（当前目录）最为基本目录，可以手动修改它
webpack-dev-server --content-base ./build
```

上面只介绍了常用的并且比较重要的一些命令行选项，要查看全部，可以使用`webpack-dev-server -h`查看帮助。

我们还可以将`webpack-dev-server`放到`package.json`的`scripts`里面，例如下面例子，执行`npm run dev`实际就是执行的对应`webpack-dev-server`命令：

```
{
    "scripts": {
        "dev": "webpack-dev-server --mode development --config webpack.config.dev.js --hot --inline --port 3000"
    }
}
```

自动刷新

在开发中，我们希望边写代码，边看到代码的执行情况，webpack-dev-server 提供自动刷新页面的功能可以满足我们的需求。webpack-dev-server 支持两种模式的自动刷新页面。

- iframe 模式：页面被放到一个 iframe 内，当发生变化时，会重新加载；
- inline 模式：将 webpack-dev-server 的重载代码添加到产出的 bundle 中。

两种模式都支持模块热替换（Hot Module Replacement）。模块热替换的好处是只替换更新的部分，而不是整个页面都重新加载。

使用方式：`webpack-dev-server --hot --inline`是开启`inline`模式的自动刷新。

和 Webpack 配置结合

webpack-dev-server 被 Webpack 作为内置插件对外提供了，这样可以直接在对应的 Webpack 配置文件中通过`devServer`这个属性的配置来配置自己的`webpack-dev-server`。

```
const path = require('path');
module.exports = {
    //...
    devServer: {
        contentBase: path.join(__dirname, 'dist'),
        port: 9000
    }
};
```

其中 `devServer.port` 表示服务器的监听端口，即运行后我们可以通过 `http://localhost:9000` 来访问应用；而 `devServer.contentBase` 表示服务器将从哪个目录去查找内容文件（即页面文件，比如 HTML）。

配置完之后，在项目中执行`webpack-dev-server`就可以看到命令行控制台有输出：

```
[wds]: Project is running at http://localhost:9000/
[wds]: webpack output is served from /
```

我们可以用http://localhost:9000/这个地址来访问本地开发服务了。

> Tips：
>
> 1. 启动 devserver 是通过`webpack-dev-server`命令行来启动的，不是`webpack`命令，执行`webpack`时`devServer`内容会被忽略
> 2. 在使用数组导出配置的方式时，只会使用第一个配置中的 `devServer` 选项，并将其用于数组中的其他所有配置。

Hot Module Replacement

HMR 即模块热替换（Hot Module Replacement）的简称，它可以在应用运行的时候，**不需要刷新页面**，就可以直接替换、增删模块。

Webpack 可以通过配置 `webpack.HotModuleReplacementPlugin` 插件来开启全局的 HMR 能力，开启后 `bundle` 文件会变大一些，因为它加入了一个小型的 HMR 运行时（runtime），当你的应用在运行的时候，Webpack 监听到文件变更并重新打包模块时，HMR 会判断这些模块是否接受 update，若允许，则发信号通知应用进行热替换。

要开启 HMR 功能，需要三步：

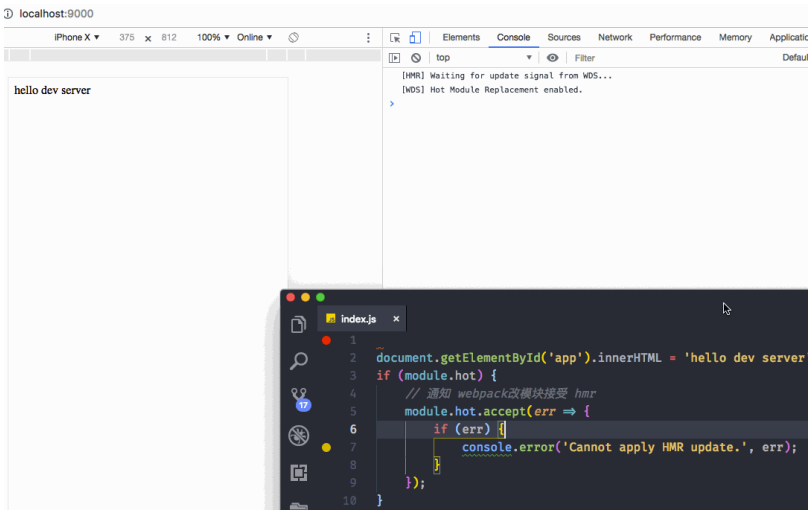
1. 设置`devServer.hot=true`， `devServer.inline=true`（默认）；
   - `devServer.hot=true`：会给 entry 添加`webpack/hot/dev-serve`或者`webpack/hot/only-dev-serve`（`devServer.hotOnly=true`），这个是实现 HMR 的服务端代码；
   - `devServer.inline=true`：会给 entry 添加`webpack-dev-server/client`，这是通信客户端；
2. 在`webpack.config.js`中添加 plugins： `new webpack.HotModuleReplacementPlugin()`；
3. 修改入口文件添加 HMR 支持代码：

```
// 在入口文件index.js最后添加如下代码
if (module.hot) {
    // 通知 webpack 该模块接受 hmr
    module.hot.accept(err => {
        if (err) {
            console.error('Cannot apply HMR update.', err);
        }
    });
}
```

最终修改后的`webpack.config.js`内容如下：

```
const path = require('path');
module.exports = {
    entry: './src/index.js',
    devServer: {
        contentBase: path.join(__dirname, 'dist'),
        port: 9000,
        // 开启 hmr 支持
        hot: true
    },
    plugins: [
        // 添加 hmr plugin
        new webpack.HotModuleReplacementPlugin()
    ]
};
```

经过上面配置之后，再次执行`webpack-dev-server`，打开`http://localhost:9000`，然后修改`index.js`内容，就能看到效果了



> Tips：使用 webpack-dev-server 的 CLI 功能只需要命令行中添加`--hot`，webpack-dev-server 会自动将`webpack.HotModuleReplacementPlugin`这个插件添加到 Webpack 的配置中去，所以开启 HotModuleReplacementPlugin 最简单的方式就是使用 inline 模式（命令行添加`--inline`）。

proxy

在实际开发中，本地开发服务器是不能直接请求线上数据接口的，这是因为浏览器的同源安全策略导致的跨域问题，我们可以使用`devServer.proxy`来解决本地开发跨域的问题。

下面的配置是将页面访问的`/api`所有请求都转发到了`baidu.com`上：

```
module.exports = {
    //...
    devServer: {
        proxy: {
            '/api': 'http://baidu.com'
        }
    }
};
```

那么，我们请求`/api/users`则会被转发到`http://baidu.com/api/users`线上地址。

`devServer.proxy`的值还支持高级属性，通过高级属性我们可以做更多的事情，如上面的需求变成，将`/api/users`转发到`http://baidu.com/users`，那么配置就需要改成：

```
module.exports = {
    //...
    devServer: {
        proxy: {
            '/api': {
                target: 'http://baidu.com',
                pathRewrite: {'^/api': ''}
            }
        }
    }
};
```

如果我们需要转发的网站是支持 https 的，那么需要增加`secure=false`，来防止转发失败：

```
module.exports = {
    //...
    devServer: {
        proxy: {
            '/api': {
                target: 'https://baidu.com',
                secure: false,
                pathRewrite: {'^/api': ''}
            }
        }
    }
};
```

又有新的需求了，这时候只能代理`json`接口的数据，对于`html`文件，还是使用打包后 dist 文件夹中文件，那么我们使用`bypass`来实现这个需求：

```
module.exports = {
    //...
    devServer: {
        proxy: {
            '/api': {
                target: 'http://baidu.com',
                bypass(req, res, proxyOptions) {
                    // 判断请求头中的 accept 值
                    if (req.headers.accept.indexOf('html') !== -1) {
                        console.log('Skipping proxy for browser request.');
                        // 返回的是 contentBase 的路径
                        return '/index.html';
                    }
                }
            }
        }
    }
};
```

或者，我们需要代理`http://baidu.com`下面的`/api`和`/auth`两个地址，其他的地址都放行，这时候可以使用`context`：

```
module.exports = {
    //...
    devServer: {
        proxy: [
            {
                context: ['/auth', '/api'],
                target: 'http://baidu.com'
            }
        ]
    }
};
```

原理上来说，webpack-dev-server 使用了 http-proxy-middleware中间件来实现的 proxy 功能，所以更多配置项及其实现可以直接参考 http-proxy-middleware的文档

自定义中间件

在 webpack-dev-server 中有两个时机可以插入自己实现的中间件，分别是在`devServer.before`和`devServer.after`两个时机，即 webpack-dev-server 加载所有内部中间件之前和之后两个时机。

```
module.exports = {
    //...
    devServer: {
        before(app, server) {
            app.get('/some/path', (req, res) => {
                res.json({custom: 'response'});
            });
        }
    }
};
```

自定义中间件在开发中常常被用来做 mock server 使用。

mock server

现在的前端团队一般都会采取前后端分离的开发模式，这样可以做到前后端同时并行开发，而前端同学开发的时候需要依赖后端同学提供的数据接口，后端同学的数据接口没有开发完成的时候，前端不能傻傻得等着，所以这就需要一个 mock server 来根据前后端接口的约定格式伪造一些假数据，这样前端开发就可以继续下去，加快开发进度。

webpack-dev-server 提供了自定义中间件的 Hook，所以我们可以很简单的实现自己的 mock server。下面代码是在`devServer.before`插入一个接口`/api/mock.json`的接口响应：

```
module.exports = {
    //...
    devServer: {
        port: 9000,
        before(app, server) {
            app.get('/api/mock.json', (req, res) => {
                res.json({hello: 'world'});
            });
        }
    }
};
```

启动 dev server，访问 http://localhost:9000/api/mock.json就可以看到这个接口返回的数据了。

https://juejin.im/post/5afba2746fb9a07aaf356327

- `devServer.compress`：服务开启 **Gzip** 压缩；

> Tips：`devServer.setup`也可以用于设置 mock server，但是已经被废弃了，将来版本肯定会被删除，不过使用`devServer.before`和`devServer.after`这俩 Hook 已经够用了。

Webpack Dev Server 常用配置

- `devServer.historyApiFallback`：配置如果找不到页面就默认显示的页面；
- `devServer.compress`：启用 gzip 压缩；
- `devServer.hotOnly`：构建失败的时候是否不允许回退到使用刷新网页；
- `devServer.inline`：模式切换，默认为内联模式，使用`false`切换到 iframe 模式；
- `devServer.open`：启动后，是否自动使用浏览器打开首页；
- `devServer.openPage`：启动后，自动使用浏览器打开设置的页面；
- `devServer.overlay`：是否允许使用全屏覆盖的方式显示编译错误，默认不允许；
- `devServer.port`：监听端口号，默认 8080；
- `devServer.host`：指定 host，使用`0.0.0.0`可以让局域网内可访问；
- `devServer.contentBase`：告诉服务器从哪里提供内容，只有在你想要提供静态文件时才需要；
- `devServer.publicPath`：设置内存中的打包文件的虚拟路径映射，区别于`output.publicPath`；
- `devServer.staticOptions`：为 Expressjs 的 `express.static`配置参数，参考文档：http://expressjs.com/en/4x/api.html#express.static
- `devServer.clientLogLevel`：在 inline 模式下用于控制在浏览器中打印的 log 级别，如`error`, `warning`, `info` or `none`；
- `devServer.quiet`：静默模式，设置为`true`则不在控制台输出 log；
- `devServer.noInfo`：不输出启动 log；
- `devServer.lazy`：不监听文件变化，而是当请求来的时候再重新编译；
- `devServer.watchOptions`：watch 相关配置，可以用于控制间隔多少秒检测文件的变化；
- `devServer.headers`：自定义请求头，例如自定义 userAgent 等；
- `devServer.https`：https 需要的证书签名等配置。

小结

Webpack 的 webpack-dev-server 是 Webpack 生态链上很重要的一环，在我们日常的开发环境，我们可以使用 webpack-dev-server 启动本地服务器，而且能够实现 API 接口代理、静态资源服务器、HMR，甚至还能够通过编写 Express 中间件的方式来扩展功能。

但是 webpack-dev-server 本身也有它的局限性，比如我们项目本身就有一个 Node.js 的业务服务，那么在使用 webpack-dev-server 来模拟接口数据就显得多此一举了。后面实战章节我们将介绍使用 Express 的中间件来实现一个自己的 dev-server！

> 本小节 Webpack 相关面试题：
>
> 1. webpack-dev-server 的 inline 模式和 iframe 模式有何异同？
> 2. webpack-dev-server 怎么配置 HMR？
> 3. webpack-dev-server 怎么使用 Express 中间件？
> 4. 能够说下你对 webpack-dev-server 理解吗？原理吗？

13 Webpack 中打包 HTML 和多页面配置

15 Webpack 中配置React和Vue开发环境