webpack-dev-middleware 基本使用与扩展

webpack-dev-middleware 是一个容器(wrapper)， 它可以把 webpack 处理后的文件传递给一个服务器(server)。webpack-dev-server 在内部使用了它， 同时， 它也可以作为一个单独的包来使用， 以便进行更多自定义设置来实现更多的需求。

作用

webpack-dev-middleware,作用就是，生成一个与 webpack 的 compiler 绑定的中间件，然后在 express 启动的服务 app 中调用这个中间件。
这个中间件的作用呢，简单总结为以下三点：通过 watch mode，监听资源的变更，然后自动打包，快速编译，走内存；返回中间件，支持 express 的 use 格式。特别注明：webpack 明明可以用 watch mode，可以实现一样的效果，但是为什么还需要这个中间件呢？
答案就是，第二点所提到的，采用了内存方式。如果，只依赖 webpack 的 watch mode 来监听文件变更，自动打包，每次变更，都将新文件打包到本地，就会很慢。

如何使用

接下来看看 webpack-dev-middleware 是如何配置的。

1. 配置 publicPath.

publicPath,熟悉 webpack 的同学都知道，这是生成的新文件所指向的路径，可以模拟 CDN 资源引用。那么跟此处的主角 webpack-dev-middleware 什么关系呢，关系就是，此处采用内存的方式，内存中采用的文件存储 write path 就是此处的 publicPath，因此，这里的配置 publicPath 需要使用相对路径。

webpack.config.js

```
module.exports = {
//...
    entry: './app.js',
    output: {
        publicPath: "/assets/",
        filename: 'bundle.js',
        path: '/'
    },
//...
```

};

在使用 webpack-dev-middleware *（或其它走内存的工具）*的情况下，publicPath 只建议配置相对路径 —— 因为 webpack-dev-middleware 在使用的时候，也需要再配置一个 publicPath *（见下文 server.js 的配置）*，用于标记从内存的哪个路径去存放和查找资源，这意味着 webpack-dev-middleware 的 publicPath 必须是相对路径。

而如果 webpack.config.js 里的 publicPath 跟 webpack-dev-middleware 的 publicPath 不一致的话*（比如前者配置了 http 的路径）*，会导致资源请求到了内存外的地方去了*（本地也没这个文件，也没法走 Fiddler 代理来解决）*，从而返回 404~

对于上面的 path 与 publicPath， 说说他们的区别

publicPath

1. 不设值，那么资源文件会从相对的根目录加载，是 html 文件的同级，网页的话则是/
2. 通过 file://打开网页，是通过绝对根目录/往下寻找路径
3. 通过 http(s)://打开网页，是通过网页的/往下寻找路径
4. 值是 http(s)://这样的 URL 路径，会直接去该路径下加载文件
   (主要用作模拟 CDN 访问资源)
5. webpack-dev-middleware 配置项里的 publicPath 要与 webpack.config 里的 output.publicPath 保持一致（并且只能是相对路径），不然会出现问题；
6. 使用 webpack-dev-middleware 的时候，其实可以完全无视 webpack.config 里的 output.path，甚至不写也可以，因为走的纯内存，output.publicPath 才是实际的 controller；
7. publicPath 配置的相对路径，实际是相对于 html 文件的访问路径。

path

就是 webpack 打包的指定物理存储地址，bundle.js 的存放位置.

2 在服务器端（node 端中引入中间件）

server.js

```
const path = require('path');
const express = require("express");
const app = express();
const webpack = require('webpack');
const webpackMiddleware = require("webpack-dev-middleware");
```

```
const webpackConf = require('./webpack.config.js');
const compiler = webpack(webpackConf);
app.use(webpackMiddleware(compiler, {
    publicPath: webpackConf.output.publicPath,
}));
app.use(express.static('dist'))
app.listen(12306);
```

其实我们能用代理服务都是因为我们有 node 帮我们进行发送服务的，node 中的 http 库可以发起服务，express 库类似 http 库，所以正在发起服务的就是 express 和 node 原生 http 库。
而我们一般用的 webpack-dev-server 中就引用 http 库，通过 npm 安装的 webpack-dev-server，你可以看他的 lib 目录下的 server.js 文件中就有应用 http 库，它用的大都是 node 的库，来进行文件操作，URL 操作，和发起服务等。

扩展

机智的小伙伴们在看完 webpack-dev-middleware 的介绍后，会洞悉出它的一处弱点 —— 虽然 webpack-dev-middleware 会在文件变更后快速地重新打包，但是每次都得手动刷新客户端页面来访问新的内容，还是略为麻烦。这是因为 webpack-dev-middleware 在应用执行的时候，没办法感知到模块的变化。

那么是否有办法可以让页面也能自动更新呢？webpack-hot-middleware 便是帮忙填这个坑的人，所以我在前文称之为 —— webpack-dev-middleware 的好朋友。

webpack-hot-middleware 提供的这种能力称为 HMR，所以在介绍 webpack-hot-middleware 之前，我们先来科普一下 HMR。

Hot Module Replacement（以下简称 HMR）是 webpack 发展至今引入的最令人兴奋的特性之一 ，当你对代码进行修改并保存后，webpack 将对代码重新打包，并将新的模块发送到浏览器端，浏览器通过新的模块替换老的模块，这样在不刷新浏览器的前提下就能够对应用进行更新。例如，在开发 Web 页面过程中，当你点击按钮，出现一个弹窗的时候，发现弹窗标题没有对齐，这时候你修改 CSS 样式，然后保存，在浏览器没有刷新的前提下，标题样式发生了改变。感觉就像在 Chrome 的开发者工具中直接修改元素样式一样。

为什么需要 HMR

在 webpack HMR 功能之前，已经有很多 live reload 的工具或库，比如 live-server，这些库监控文件的变化，然后通知浏览器端刷新页面，那么我们为什么还需要 HMR 呢？答案其实在上文中已经提及一些。

- live reload 工具并不能够保存应用的状态（states），当刷新页面后，应用之前状态丢失，还是上文中的例子，点击按钮出现弹窗，当浏览器刷新后，弹窗也随即消失，要恢复到之前状态，还需再次点击按钮。而 webapck HMR 则不会刷新浏览器，而是运行时对模块进行热替换，保证了应用状态不会丢失，提升了开发效率。

- 在古老的开发流程中，我们可能需要手动运行命令对代码进行打包，并且打包后再手动刷新浏览器页面，而这一系列重复的工作都可以通过 HMR 工作流来自动化完成，让更多的精力投入到业务中，而不是把时间浪费在重复的工作上。
- HMR 兼容市面上大多前端框架或库，比如 React Hot Loader，Vue-loader，能够监听 React 或者 Vue 组件的变化，实时将最新的组件更新到浏览器端。Elm Hot Loader 支持通过 webpack 对 Elm 语言代码进行转译并打包，当然它也实现了 HMR 功能。

HMR 的工作原理讲解

- 第一步，在 webpack 的 watch 模式下，文件系统中某一个文件发生修改，webpack 监听到文件变化，根据配置文件对模块重新编译打包，并将打包后的代码通过简单的 JavaScript 对象保存在内存中。
- 第二步是 webpack-dev-server 和 webpack 之间的接口交互，而在这一步，主要是 dev-server 的中间件 webpack-dev-middleware 和 webpack 之间的交互，webpack-dev-middleware 调用 webpack 暴露的 API 对代码变化进行监控，并且告诉 webpack，将代码打包到内存中。
- 第三步是 webpack-dev-server 对文件变化的一个监控，这一步不同于第一步，并不是监控代码变化重新打包。当我们在配置文件中配置了 devServer.watchContentBase 为 true 的时候，Server 会监听这些配置文件夹中静态文件的变化，变化后会通知浏览器端对应用进行 live reload。注意，这儿是浏览器刷新，和 HMR 是两个概念。
- 第四步也是 webpack-dev-server 代码的工作，该步骤主要是通过 sockjs（webpack-dev-server 的依赖）在浏览器端和服务端之间建立一个 websocket 长连接，将 webpack 编译打包的各个阶段的状态信息告知浏览器端，同时也包括第三步中 Server 监听静态文件变化的信息。浏览器端根据这些 socket 消息进行不同的操作。当然服务端传递的最主要信息还是新模块的 hash 值，后面的步骤根据这一 hash 值来进行模块热替换。
- webpack-dev-server/client 端并不能够请求更新的代码，也不会执行热更模块操作，而把这些工作又交回给了 webpack，webpack/hot/dev-server 的工作就是根据 webpack-dev-server/client 传给它的信息以及 dev-server 的配置决定是刷新浏览器呢还是进行模块热更新。当然如果仅仅是刷新浏览器，也就没有后面那些步骤了。
- HotModuleReplacement.runtime 是客户端 HMR 的中枢，它接收到上一步传递给他的新模块的 hash 值，它通过 JsonpMainTemplate.runtime 向 server 端发送 Ajax 请求，服务端返回一个 json，该 json 包含了所有要更新的模块的 hash 值，获取到更新列表后，该模块再次通过 jsonp 请求，获取到最新的模块代码
- 接下来决定 HMR 成功与否的关键步骤，在该步骤中，HotModulePlugin 将会对新旧模块进行对比，决定是否更新模块，在决定更新模块后，检查模块之间的依赖关系，更新模块的同时更新模块间的依赖引用。
- 最后一步，当 HMR 失败后，回退到 live reload 操作，也就是进行浏览器刷新来获取最新打包代码。

我们试着对前文使用的项目来做一番改造 —— 引入 webpack-hot-middleware 来提升开发体验。

首先往 server.js 加上一小段代码：

server.js

```
const path = require('path');
const express = require("express");
const app = express();
const webpack = require('webpack');
const webpackMiddleware = require("webpack-dev-middleware");
const webpackConf = require('./webpack.config.js');
const compiler = webpack(webpackConf);
app.use(webpackMiddleware(compiler, {
    publicPath: webpackConf.output.publicPath,
}));
//添加的代码段，引入和使用 webpack-hot-middleware
app.use(require("webpack-hot-middleware")(compiler, {
    path: '/__webpack_hmr'
}));
app.use(express.static('dist'))
app.listen(12306);
```

即在原先的基础上引入了 webpack-hot-middleware：

```
app.use(require("webpack-hot-middleware")(webpackCompiler,
options));
```

这里的 options 是 webpack-hot-middleware 的配置项，详细见官方文档，这里咱们只填一个必要的 path —— 它表示 webpack-hot-middleware 会在哪个路径生成热更新的事件流服务

然后是 webpack.config.js 文件：

webpack.config.js

```
module.exports = {
//...
     // entry: './app.js',
    entry: ['webpack-hot-middleware/client', './app.js'],
    output: {
        publicPath: "/assets/",
        filename: 'bundle.js',
```

```
        path: '/'
    },
    plugins: [
                new webpack.HotModuleReplacementPlugin()
        ]
//...
};
```

首先是 entry 里要多加上 'webpack-hot-middleware/client'，此举是与 server 创建连接。

虽然 webpack-dev-middleware + webpack-hot-middleware 的组合为开发过程提供了便利，但它们仅适用于服务侧开发的场景。

很多时候我们仅仅对客户端页面做开发，没有直接的 server 来提供支持，这时候就需要 webpack-dev-server 来进行帮助了。所以这篇文章让大家认识一下 webpack-dev-server 更底层的知识，也就是 webpack-dev-middleware 以及 webpack-hot-middleware