你真的知道 webpack-dev-server 么

很多同学，在配置 webpack 的时候，在开发环境中会使用到 webpack-dev-server（什么是开发模式，不懂的回去看之前的文章——webpack 的开发环境与生产环境），总是去使用它，我们来探究一下，它到底帮助我们解决了哪些问题，有哪些优点？

不得不说，在实际开发中我们需要一些便于开发的功能，比如说：

提供 HTTP 服务而不是使用本地文件预览；
监听文件的变化并自动刷新网页，做到实时预览；
支持 Source Map，以方便调试。

对于这些， Webpack 都为你考虑好了。Webpack 原生支持上述第 2、3 点内容，再结合官方提供的开发工具 DevServer 也可以很方便地做到第 1 点。 DevServer 会启动一个 HTTP 服务器用于服务网页请求，同时会帮助启动 Webpack ，并接收 Webpack 发出的文件更变信号，通过 WebSocket 协议自动刷新网页做到实时预览。

首先需要安装 DevServer：

```
npm i -D webpack-dev-server
```

接下来写一个最最普通的 webpack 配置

webpack.config.js

```
const path = require('path');
module.exports = {
    entry: './src/index.js',
    output: {
        filename: 'bundle.js',
        path: path.resolve(__dirname, 'dist')
    }
}
```

接下来直接通过 webpack-dev-server 启动（webpack-dev-server --config ./webpack.config.js --mode development）：我们看一下结果：

```
$ webpack-dev-server --config webpack.config.js
i ｢wds｣: Project is running at http://localhost:8080/
i ｢wds｣: webpack output is served from /
i ｢wdm｣: Hash: 21a460ed9df9e0257a65
```

这意味着 DevServer 启动的 HTTP 服务器监听在 http://localhost:8080/ ，DevServer 启动后会一直驻留在后台保持运行，访问这个网址你就能获取项目根目录下的 index.html。 用浏览器打开这个地址你会发现页面空白错误原因是 ./dist/bundle.js 加载 404 了。 同时你会发现并没有文件输出到 dist 目录，原因是 DevServer 会把 Webpack 构建出的文件保存在内存中，在要访问输出的文件时，必须通过 HTTP 服务访问。 由于 DevServer 不会理会 webpack.config.js 里配置的 output.publicPath 属性，而不是 path 属性！，所以要获取 bundle.js 的正确 URL 是 http://localhost:8080/bundle.js，对应的 index.html 应该修改为

```
<script src="bundle.js"></script>
```

实时预览

接着上面的步骤，你可以试试修改 index.js 中的任何一个文件，保存后你会发现浏览器会被自动刷新，运行出修改后的效果。

Webpack 在启动时可以开启监听模式，开启监听模式后 Webpack 会监听本地文件系统的变化，发生变化时重新构建出新的结果。Webpack 默认是关闭监听模式的，你可以在启动 Webpack 时通过 webpack --watch 来开启监听模式。

通过 DevServer 启动的 Webpack 会开启监听模式，当发生变化时重新执行完构建后通知 DevServer。 DevServer 会让 Webpack 在构建出的 JavaScript 代码里注入一个代理客户端用于控制网页，网页和 DevServer 之间通过 WebSocket 协议通信， 以方便 DevServer 主动向客户端发送命令。 DevServer 在收到来自 Webpack 的文件变化通知时通过注入的客户端控制网页刷新。

如果尝试修改 index.html 文件并保存，你会发现这并不会触发以上机制，导致这个问题的原因是 Webpack 在启动时会以配置里的 entry 为入口去递归解析出 entry 所依赖的文件，只有 entry 本身和依赖的文件才会被 Webpack 添加到监听列表里。 而 index.html 文件是脱离了 JavaScript 模块化系统的，所以 Webpack 不知道它的存在。

模块热替换

除了通过重新刷新整个网页来实现实时预览，DevServer 还有一种被称作模块热替换的刷新技术。 模块热替换能做到在不重新加载整个网页的情况下，通过将被更新过的模块替换老的模块，再重新执行一次来实现实时预览。 模块热替换相对于默认的刷新机制能提供更快的响应和更好的开发体验。 模块热替换默认是关闭的，要开启模块热替换，你只需在启动 DevServer 时带上 --hot 参数，重启 DevServer 后再去更新文件就能体验到模块热替换的神奇了。

这里说一说 publicPath

webpack 提供一个非常有用的配置，该配置能帮助你为项目中的所有资源指定一个基础路径，它被称为公共路径(publicPath)。

其实这里说的所有资源的基础路径是指项目中引用 css，js，img 等资源时候的一个基础路径，这个基础路径要配合具体资源中指定的路径使用，所以其实打包后资源的访问路径可以用如下公式表示：

静态资源最终访问路径 = output.publicPath + 资源 loader 或插件等配置路径

接下来说说常用的 webpack-dev-server 常用配置

1. contentBase

即 SERVERROOT，如上方示例配置为 “path.join(__dirname, "src/html")”，后续访问 http://localhost:8080/index.html 时，SERVER 会从 src/html 下去查找 index.html 文件。

它可以是单个或多个地址的形式：

```
contentBase: path.join(__dirname, "public")
//多个：
contentBase: [path.join(__dirname, "public"), path.join(__dirname, "assets")]
```

若不填写该项，默认为项目根目录。

2. port

即监听端口，默认为 8080。

3. compress

传入一个 boolean 值，通知 SERVER 是否启用 gzip。

4. hot

传入一个 boolean 值，通知 SERVER 是否启用 HMR。（什么是 HMR 后面的文章会说）

5. https

可以传入 true 来支持 https 访问，也支持传入自定义的证书：

webpack.config.js

```
https: true
//也可以传入一个对象，来支持自定义证书
https: {
  key: fs.readFileSync("/path/to/server.key"),
  cert: fs.readFileSync("/path/to/server.crt"),
  ca: fs.readFileSync("/path/to/ca.pem"),
}
```

6. proxy

代理配置，适用场景是，除了 webpack-dev-server 的 SERVER（SERVER A） 之外，还有另一个在运行的 SERVER（SERVER B），而我们希望能通过 SERVER A 的相对路径来访问到 SERVER B 上的东西。

举个例子：

webpack.config.js

```
devServer: {
        contentBase: path.join(__dirname, "src/html"),
        port: 3333,
        hot: true,
        proxy: {
            "/api": "http://localhost:5050"
        }
    }
```

运行 webpack-dev-server 后，你若访问 http://localhost:3333/api/user，则相当于访问 http://localhost:5050/api/user。

这种代理在很多情况下是很重要的，比如可以把一些静态文件通过本地的服务器加载，而一些 API 请求全部通过一个远程的服务器来完成。还有一个情景就是在两个独立的服务器之间进行请求分割，如一个服务器负责授权而另外一个服务器负责应用本身。

下面给出日常开发中遇到的一个例子：
（1）有一个请求是通过相对路径来完成的，比如地址是 "/msg/show.htm"。但是，在日常和生产环境下前面会加上不同的域名，如日常是 you.test.com 而生产环境是 you.inc.com。

（2）那么比如现在想在本地启动一个 webpack-dev-server，然后通过 webpack-dev-server 来访问日常的服务器，而且日常的服务器地址是 11.160.119.131，所以会通过如下的配置来完成：

```
devServer: {
    port: 8000,
    proxy: {
      "/msg/show.htm": {
        target: "http://11.160.119.131/",
        secure: false
      }
    }
  }
```

此时当请求 "/msg/show.htm" 的时候，其实请求的真实 URL 地址为 "http//11.160.119.131/msg/show.htm"。

（3）在开发环境中遇到一个问题，那就是：如果本地的 devServer 启动的地址为："http://30.11.160.255:8000/" 或者常见的 "http://0.0.0.0:8000/" ，那么真实的服务器会返回一个 URL 要求登录，但是，将本地 devServer 启动到 localhost 上就不存在这个问题了（一个可能的原因在于 localhost 种上了后端需要的 cookie，而其他的域名没有种上 cookie，导致代理服务器访问日常服务器的时候没有相应的 cookie，从而要求权限验证）。其中指定 localhost 的方式可以通过

(4)

其实正向代理和反向代理用一句话来概括就是：“正向代理隐藏了真实的客户端，而反向代理隐藏了真实的服务器”。而 webpack-dev-server 其实扮演了一个代理服务器的角色，服务器之间通信不会存在前端常见的同源策略，这样当请求 webpack-dev-server 的时候，它会从真实的服务器中请求数据，然后将数据发送给你的浏览器。

```
 browser => localhost:8080(webpack-dev-server 无代理) =>
http://you.test.com
 browser => localhost:8080(webpack-dev-server 有代理) =>
http://you.test.com
```

上面的第一种情况就是没有代理的情况，在 localhost:8080 的页面通过前端策略去访问 http://you.test.com 会存在同源策略，即第二步是通过前端策略去访问另外一个地址的。但是对于第二种情况，第二步其实是通过代理去完成的，即服务器之间的通信，不存在同源策略问题。而我们变成了直接访问代理服务器，代理服务器返回一个页面，对于页面中某些满足特定条件前端请求（proxy、rewrite 配置）全部由代理服务器来完成，这样同源问题就通过代理服务器的方式得到了解决。

7. setup

webpack-dev-server 的服务应用层使用了 express，故可以通过 express app 的能力来模拟数据回包，devServer.setup 方法就是干这事的：

```
devServer: {
    contentBase: path.join(__dirname, "src/html"),
    port: 3333,
    hot: true,
    setup(app){  //模拟数据
        app.get('/getJSON', function(req, res) {
            res.json({ name: 'value' });
        });
    }
}
```

然后我们可以通过请求 http://localhost:3333/getJSON 来取得模拟的数据

在下篇文章，我们来探讨一下 webpack-dev-server 的底层知识。