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

更新时间：2019-07-15 10:24:04

> 世界上最快乐的事，莫过于为理想而奋斗。
>
> ——苏格拉底

在原理篇介绍HMR实现的文章中，我们对 Webpack-dev-server 和 HMR 做了深入的剖析。在实际开发中，我们仅仅使用 Webpack-dev-server 功能可能不够，例如：

- 不能监控 Webpack 配置文件更改之后重新启动；
- 对于本身就是 Node.js 做后端服务器的项目来说，webpack-dev-server 反而会因为不能跟原服务器结合显得很鸡肋；
- 只对 Node.js 有支持，如果后端程序是 PHP、Java 写的那么 webpack-dev-server 就束手无策。

webpack-dev-server 本质上是一个 Expressjs 的服务器，而真正跟 Webpack 交互的是它用到的中间件——webpack-dev-middleware，如果我们前端项目本身就是一个 Express 服务器，那么我们可以使用 webpack-dev-middleware 和 webpack-hot-middleware 实现 webpack-dev-server 的功能，webpack-hot-middleware 这个 Express 中间件可以为 Express 服务器提供 LiveReload 功能。

本文将从头带大家零基础用 Express 来实现一个 Webpack-dev-server，总共代码行数带注释不过 150 行左右，却实现了 webpack-dev-server 的主要功能，并且添加 mock 功能，还能够实现 Webpack 配置文件 watch 功能。

> Tips：webpack-dev-middleware 和 webpack-hot-middleware 的功能，还有 Koa 版本和 Hapi 版本，Koa 和 Hapi 都是 Node.js 服务器框架。

Express 核心概念

Express是一个流行的 Node.js 服务端框架，通过 Express 可以创建 Node.js 服务端程序。Express API 简单，功能却很强大，很多流行的 Web 应用开发框架都是基于 Express 来实现的，包括我们公司在内的很多公司 Web 产品后台服务底层也是使用 Express 框架来实现的，所以 Express 是可以用于线上生产环境的。

> Tips：为了后面内容实际操作，可以先创建一个`dev-server`的文件夹，并且执行`npm init -y`准备好 NPM 环境，然后安装 Express 包：`npm i -S express`。后续的代码都是在这个文件夹下编写对应的代码。

Router

路由（Router）是一种将 URL 和 HTTP 方法映射到特定处理回调函数的技术，例如我们希望访问`/hello`这个 URL 地址，就显示`hi, world`文字，那么我们在 Express 中可以这样来实现：

```
const http = require('http');
const express = require('express');
const app = express();
app.get('/hello', (request, response) => {
    response.end('hi, world');
});

http.createServer(app).listen(3000);
```

在上面的代码中，`app.get('url', callback)`形式就是一个`GET`路由，将`callback`和`URL`进行了映射绑定。

在 Express 中封装了多种 HTTP 请求方式，我们主要用到的是 GET 和 POST 两种，即`app.get()`和`app.post()`。它们的第一个参数都是一个请求路径，第二个参数则为处理请求的回调函数。回调函数有两个参数，分别是`request`和`response`，即对应 HTTP 协议中的请求和响应两个概念。

> Tips：Express 的路由除了纯字符串这类，还支持正则、字符串通配符、命名参数等，但是应该注意路由的解析速度，如果一个正则路由写的匹配极低，那么会影响整个 Server 应用速度的。

Request 和 Response

Request 和 Response 分别对应着 HTTP 协议中的请求和响应。在 Express 中，将跟 HTTP 请求相关的变量都放到了 Request 对象中，例如：我们可以从 Request 对象中读取用户的浏览器 UserAgent、用户的 IP 地址、用户携带的 Cookie 信息等。Response 对象则主要提供跟做出 HTTP 响应相关的函数和属性，比如设置响应内容、设置 HTTP 响应头中的设置 Cookie、设置 HTTP 状态码等。

下面的一段代码，用户访问`http://localhost:3000/`则显示用户的浏览器 UserAgent：

```
const http = require('http');
const express = require('express');
const app = express();
app.get('/', (req, res) => {
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.end(req.headers['user-agent']);
});

http.createServer(app).listen(3000);
```

中间件

中间件是 Express 中最大的特性之一。我们可以将中间件看成由一串回调函数组成的回调栈，每个回调函数都会接受`request`、`response`和`next`参数，我们可以在各个回调函数中做不同的事情，例如专门记录请求日志的回调函数、处理 Cookie 的回调函数、专门处理 HTTP 请求头的回调函数、专门做错误页面展现的回调函数...通过一个 HTTP 请求（request）经过回调栈最终对 HTTP 做出响应（response）。

在 Express 中，可以使用`Express.use`方法给一个 server 添加中间件：

```
const http = require('http');
const cuid = require('cuid');
const express = require('express');
const app = express();
// 日志记录中间件
app.use((request, response, next) => {
    console.log('In comes a ' + request.method + ' to ' + request.url);
    next();
});
// 我们还可以修改request或者response对象，让他们携带每次请求的唯一数据
app.use((request, response, next) => {
    request.id = cuid();
    // 传递至下一个中间件
    next();
});
// 发送实际响应
app.use((request, response) => {
    response.writeHead(200, {'Content-Type': 'text/plain'});
    // 这里最后没有调用next扔给下一个中间件，而是直接输出了响应内容
    response.end('Hello, world! Your ID is ' + request.id);
});

http.createServer(app).listen(3000);
```

> Tips：在编写 Node.js 的 Server 端应用一定要做好内存管理。在浏览器内，每个用户访问同一个页面时都是一个独立的浏览器 Tab 甚至是独立的设备，所以内存使用不当造成内存泄漏的问题也不会特别严重，而在 Node.js Server 应用中，成千上万的用户会同时访问我们的服务，这时候应该注意：1. 不同的用户数据不能使用全局变量管理；2. 小的内存泄漏问题，因为请求多了执行次数增多而造成大的内存泄漏问题，所以开发 Node.js Server 应用一定要转变思想，不能依旧停留在浏览器开发模型中。在 Express 中，用户差异数据可以使用用户自己的`request`对象来存储，参考上面中间件示例中的`request.id`。

使用 Express 及其中间件来实现 Webpack-dev-server

通过之前的功能，我们了解到 Webpack-dev-server 主要功能有以下几点：

1. 静态服务器；
2. 代理服务器，使用 http-proxy-middleware 实现；
3. 实现 Mock Server；
4. 使用 webpack-dev-middleware 实现跟 Webpack 交互；
5. 使用 SockJS 实现跟`webpack/hot/server`通信，实现 HMR 功能。

为了对标 Webpack-dev-server 的配置项，我们先设置一个 Webpack-dev-server 的配置文件，内容如下：

```
// devServer.config.js
module.exports = {
    port: 3000,
    contentBase: '',
    before() {},
    hot: true,
    hotOnly: false,
    proxy: {
        '/api': 'http://localhost:3000'
    }
};
```

下面的 Express 来实现 dev-server 就是使用这个配置文件来做配置。

使用 Express 实现一个静态服务器

在 webpack-dev-server 中可以设置`contentBase`来做静态资源服务器。我们先用 Express 来实现一个静态资源服务器。在 Express 中要实现一个静态资源服务器可以使用 Express 内置的serve-static这个中间件来实现：

```
// static-server.js
const http = require('http');
const path = require('path');
const express = require('express');
const app = express();
// 添加中间件，设置从`public`文件夹中查找文件
app.use(express.static(path.join(__dirname, 'public')));

// 最后，创建server，并且输出地址
const server = http.createServer(app);
server.listen(port, () => {
    console.log('Listening on %j', server.address());
});
```

这时候我们在当前路径下创建个`public`文件夹，并且在里面添加个`hello.js`文件，然后执行`node static-server.js`，打开浏览器访问http://localhost:3000/hello.js，就会看到 hello.js 的内容了！

就这么简单，6 行代码实现了一个静态资源服务器！

添加代理服务器中间件

现在我们已经知道 Webpack-dev-server 中使用的`devServer.proxy`实际是使用 Express 的中间件http-proxy-middleware来实现的了，那么我们首先在`package.json`中添加这个依赖：

```
npm install -S http-proxy-middleware
```

然后按照 http-proxy-middleware 的文档，给我们刚才的代码添加 http-proxy-middleware 中间件：

```
Object.keys(proxy).forEach(router => {
    // 添加http-proxy-middleware
    app.use(router, httpProxyMiddleware(proxy[router]));
});
```

实现 mock server 功能

对于前后端开发的项目，我们在绝大多数情况前后端开发人员会先约定接口格式，Web 前端可以按照约定接口格式使用本地的 Mock 数据进行开发，等后端接口准备就绪之后再进行使用后端接口进行联调。webpack-dev-server 提供了 proxy 配置。我们可以在开发中将接口代理到本地服务。一般我们的接口都是 JSON 格式的接口，但是 webpack-dev-server 使用的 http-proxy-middleware，已经不支持本地 JSON 文件的代理，这个 issue说明了原因。我们在开发中的 Mock 需要自己使用中间件来实现。

在这里我们使用mocker-api，我们先添加`mocker-api`：

```
npm install -S mocker-api
```

然后按照 mocker-api 的文档添加中间件：

```
// 2. mocker-api
if (mock) {
    // 这里的mock是mocker-api配置文件路径
    apiMockerMiddleware(app, path.resolve(mock));
}
```

整合 Webpack

经过上面三步，我们的 Express 服务已经可以访问静态文件、做代理服务器、可以使用本地数据模拟 API 接口，下面要做的就是整合 Webpack 进我们的 Express 服务器，并且使用 webpack-dev-middleware 和 webpack-hot-middleware 来实现 HMR。

webpack-hot-middleware 通信机制

webpack-hot-middleware 的通信机制用的是EventSource，EventsSource 的官方名称应该是 Server-sent events（缩写 SSE）服务端派发事件，EventSource 基于 HTTP 协议，它通过 HTTP 连接到一个服务器，以 text/event-stream 格式接收事件，只是简单的单向通信，实现了服务端推送消息到客户端，而不能实现客户端发送数据到服务端。 虽然 EventSource 不能实现双向通信，但是在功能设计上它也有一些优点，比如：

1. 可以自动重连；
2. 基于 HTTP 协议，可以引入polyfill支持低版本浏览器；
3. 相对于 WebSocket 需要基于 TCP 设计一种新的通信协议，EventSource 更加轻量级一些。

在日常应用中，EventSource 因为受单向通信的限制只能用来实现像股票报价、新闻推送、实时天气这些只需要服务器发送消息给客户端场景中。

使用示例：

首先支持 EventSource 的浏览器，存在`window.EventSource`，我们可以直接使用 EventSource 的 server 路径来实例化一个 EventSource 对象，然后可以绑定`open`、`message`、`error`等消息，还可以通过`addEventListener`方式监听具名消息：

```
// client.js
// 实例化 EventSource 参数是服务端监听的路由
const source = new EventSource('/event-source-demo');
source.onopen = event => {
    // 与服务器连接成功回调
    console.log('成功与服务器连接');
};
// 监听从服务器发送来的所有没有指定事件类型的消息(没有event字段的消息)
source.onmessage = event => {
    // 监听未命名事件
    console.log('未命名事件', event.data);
};
source.onerror = error => {
    // 监听错误
    console.log('错误');
};
// 监听指定类型的事件（可以监听多个）
source.addEventListener('ping', event => {
    console.log('ping', JSON.parse(event.data));
});
```

在服务端，需要设置一个 EventSource 的路由，然后修改 HTTP 请求头的`'Content-Type': 'text/event-stream'`，最后按照 EventSource 规范发送 Response 响应内容即可：

```
// server.js
// 使用Express框架，忽略之前代码
// 监听event-source-demo路由服务端返回事件流
app.get('/event-source-demo', (ewq, res) => {
    // 根据 EventSource 规范设置报头
    res.writeHead(200, {
        'Content-Type': 'text/event-stream', // 规定把报头设置为 text/event-stream
        'Cache-Control': 'no-cache' // 设置不对页面进行缓存
    });
    // 用write返回事件流，事件流仅仅是一个简单的文本数据流，每条消息以一个空行(\n)作为分割。
    res.write(':注释' + '\n\n'); // 注释行
    res.write('data:' + '消息内容1' + '\n\n'); // 未命名事件

    res.write(
        // 命名事件
        'event: ping' + '\n' + 'data:' + '消息内容2' + '\n' + 'retry:' + '2000' + '\n' + 'id:' + '12345' + '\n\n'
    );

    setInterval(() => {
        // 定时事件
        res.write('data:' + '定时消息' + '\n\n');
    }, 2000);
});
```

EventSource 事件流格式为普通的 UTF-8 字符串即可，每条消息后面跟着一个`\n`来做分隔符，然后按照下面规范来规定字符串内容：

1. 注释行：以`:`开头的内容是注释行，会被忽略，定时发送一条注释行可以用来防止连接超时；
2. 字段：字段由字段名和字段值按照`字段名: 字段值`的规范组成，规范中的字段有：`event`、`data`、`id`和`retry`。

举例来说明：

下面是未命名事件：

```
: this is a test stream

data: some text

data: another message
data: with two lines
```

命名事件：

```
event: ping
data: {"username": "bobby", "time": "02:33:48"}
```

也可以在一个事件流中同时使用命名事件和未命名事件：

```
event: userconnect
data: {"username": "bobby", "time": "02:33:48"}

data: Here's a system message of some kind that will get used
data: to accomplish some task.
```

给 contenBase 增加 watch 功能

先来看下 webpack-dev-server 是怎么实现的。首先在 Server 端使用chokidar添加 contentBase 路径的监听，发生`change`事件则触发`sockWrite`发送一个`content-changed`事件，这部分代码在`webpack-dev-server/lib/Server.js`中：

```
// webpack-dev-server/lib/Server.js
const watcher = chokidar.watch(watchPath, options);

watcher.on('change', () => {
    this.sockWrite(this.sockets, 'content-changed');
});
// 下面是sockWrite的实现
sockWrite(sockets, type, data) {
    sockets.forEach((socket) => {
        // socket来自于sockjs.createServer之后的connection实例
        socket.write(JSON.stringify({ type, data }));
    });
}
```

webpack-dev-server 的 client 端，通过`sock = new SockJS(url)`之后，添加`sock.onmessage`监听，监听 Server 端消息，最终接收到的消息处理扔给了`webpack-dev-server/client-src/default/index.js`的`onSocketMsg`对象针对不同的消息进行处理。下面是`content-changed`消息的处理，在里面我们看到了执行的是`self.location.reload()`最终导致页面重载：

```
'content-changed': function contentChanged() {
    log.info('[WDS] Content base changed. Reloading...');
    self.location.reload();
  }
```

所以要在 webpack-hot-middleware 中实现 contentBase 的文件变化监听，然后通知 client 端页面重载，需要我们做两件事情：

1. 使用 chokidar 监听文件变化；
2. 然后通过 webpack-hot-middleware 推送消息到 client 端，client 端接收到消息重新加载页面。

使用 chokidar 监听文件变化

这部分代码使用 chokidar 实现的，比较简单，直接放实现代码：

```
if (devServerConfig.watchContentBase) {
    // 判断watchoptions
    const {poll, ignored} = devServerConfig.watchOptions;
    const usePolling = poll ? true : undefined;
    const interval = typeof poll === 'number' ? poll : undefined;
    // 这个是chokidar的options，参考它的文档
    const options = {
        ignoreInitial: true,
        persistent: true,
        followSymlinks: false,
        atomic: false,
        alwaysStat: true,
        ignorePermissionErrors: true,
        ignored,
        usePolling,
        interval
    };
    function _watch(watchPath) {
        const watcher = chokidar.watch(watchPath, options);
        watcher.on('change', () => {
            // TODO 这里需要我们新一步编写代码实现发送Server消息
            // ...
        });
    }
    if (Array.isArray(contentBase)) {
        contentBase.forEach(item => {
            _watch(item);
        });
    } else if (contentBase && typeof contentBase === 'string') {
        _watch(contentBase);
    }
}
```

利用 webpack-hot-middleware 实现页面重载

在 webpack-hot-middleware 中是使用 EventSource 来通信的，同时它在客户端和 Server 端都暴漏了接口可以让我们实现 Server 端到客户端的通信。我们判断出来 contentBase 内容发生变化之后，应该像 webpack-dev-server 实现一样，发送个让页面重新加载（reload）的事件。

可以在`webpack-hot-middleware/middleware.js`中找到 webpack-hot-middleware 实际返回的是一个`middleware`的函数，而这个函数有个`publish`方法，这个方法就是利用`eventStream`对象发送 Server 消息给 client 的：

```
// webpack-hot-middleware/middleware.js
middleware.publish = function(payload) {
    if (closed) return;
    eventStream.publish(payload);
};
```

这样，我们只需将 webpack-hot-middleware 的返回赋值给一个`webpackHotMiddlewareInstance`对象，后面就可以在上面的 chokidar 代码中的`_watch`函数内直接使用`webpackHotMiddlewareInstance.publish`发送消息了：

```
const webpackHotMiddlewareInstance = webpackHotMiddleware(compiler, {
    log: console.log,
    path: '/__webpack_hmr',
    heartbeat: 10 * 1000
});
app.use(webpackHotMiddlewareInstance);

// 忽略其他代码
function _watch(watchPath) {
    const watcher = chokidar.watch(watchPath, options);
    watcher.on('change', () => {
        // 使用webpackHotMiddlewareInstance发送reload消息给client
        // client接收消息后reload页面
        webpackHotMiddlewareInstance.publish({action: 'reload'});
    });
}
```

上面的代码发送了一个`{action: 'reload'}`的消息，那么我们接下来需要做的就是让 client 端页面处理这个消息，我们可以在`webpack-hot-middleware/client.js`中找到下面的代码：

```
// webpack-hot-middleware/client.js
module.exports = {
    subscribeAll: function subscribeAll(handler) {
        subscribeAllHandler = handler;
    },
    subscribe: function subscribe(handler) {
        customHandler = handler;
    },
    useCustomOverlay: function useCustomOverlay(customOverlay) {
        if (reporter) reporter.useCustomOverlay(customOverlay);
    },
    setOptionsAndConnect: setOptionsAndConnect
};
```

说明这个`client.js`提供了`subscribeAll`和`subscribe`这两个跟消息订阅相关的接口函数，那么我们再顺藤摸瓜看下`subscribeAllHandler`和`customHandler`究竟是什么，继续在`client.js`中查找，找到了`processMessage`这个方法：

```
// webpack-hot-middleware/client.js
var customHandler;
var subscribeAllHandler;
function processMessage(obj) {
    switch (obj.action) {
        //....忽略其他的action
        default:
            if (customHandler) {
                customHandler(obj);
            }
    }

    if (subscribeAllHandler) {
        subscribeAllHandler(obj);
    }
}
```

这个方法就是处理接收到的 Server 端消息。最终所有的消息都扔给了`subscribeAllHandler`。而存在`obj.action`内的消息，如果`action`值不在`switch`分支，最后会进入`default`交给`customHandler`处理。`switch`这里只处理了`building`、`built`、`sync`3 个`action`，知道这些之后，我们就可以在 Webpack 的入口文件中，增加下面订阅 Server 消息的代码，然后`reload`页面：

```
import webpackHotMiddleware from 'webpack-hot-middleware/client?path=/__webpack_hmr&timeout=20000';
// 通过subscribe方法来添加订阅
webpackHotMiddleware.subscribe(({action}) => {
    // 判断action===reload，则reload页面
    if (action === 'reload') {
        location.reload();
    }
});
```

实现 **Webpack** 配置文件监控自动重启 **dev-server**

最后我们在一开始配置 Webpack 配置文件的时候，每次想看效果都要重启 dev-server，如果手动重启还是比较麻烦的，这里在介绍一个使用nodemon的方式来监控`webpack.config.js`文件变化、自动重启 dev-server 的方式。

我们可以在`package.json`中添加一个 NPM scripts：

```
{
    "scripts": {
        "start": "nodemon --watch webpack.config.js --exec node ./dev-server.js"
    }
}
```

> Tips：除了这种方式还可以通过 Node.js 的`child_process.fork`来 fork 一个进程执行`dev-server.js`，同时使用 chokidar 来监控`webpack.config.js`变化，一旦变化则 kill 掉进程重新 fork 新的进程执行`dev-server.js`。

收尾整理

好了，到这里我们的 dev-server 就已经完成了，为了更好地被使用，我们也可以封装成一个 Node.js 模块，通过传入`webpack.config.js`的`devServer`配置来启动我们的 server。我们还可以`return`一个 Promise 对象，方便绑定回调，同时我们还需要使用`Compiler`的`done`hook 来监听 Webpack 打包的进展和结果，通过拿到的`stats`来展现打包结果。最后整个代码如下：

```
// server.js
const path = require('path');
const http = require('http');
const express = require('express');
// http-proxy-middleware
const httpProxyMiddleware = require('http-proxy-middleware');
// mock-middleware
const apiMockerMiddleware = require('mocker-api');
// hot
const webpackHotMiddleware = require('webpack-hot-middleware');
// dev
const webpackDevMiddleware = require('webpack-dev-middleware');
// watch file change
const chokidar = require('chokidar');

module.exports = webpackConfig => {
    const devServerConfig = webpackConfig.devServer;
    const {port = 3000, host = '0.0.0.0', contentBase, proxy, mock} = devServerConfig;
    const app = express();

    // 0. 实现static服务器
    // 判断类型，如果是数组，那么就循环添加
    if (Array.isArray(contentBase)) {
        contentBase.forEach(item => {
            app.get('*', express.static(path.resolve(item)));
        });
    } else if (contentBase && typeof contentBase === 'string') {
        app.get('*', express.static(path.resolve(contentBase)));
    }
```

```
}

// 1. proxy
Object.keys(proxy).forEach(router => {
    app.use(router, httpProxyMiddleware(proxy[router]));
});

// 2. mocker-api
if (mock) {
    // 这里的mock是mocker-api配置文件路径
    apiMockerMiddleware(app, path.resolve(mock));
}

// 3. webpack

// 3.1. 读取webpack.config.js文件
const webpack = require('webpack');
// 3.2. 创建compiler对象
const compiler = webpack(webpackConfig);

// 3.3. 添加dev-middleware，使用compiler参数
app.use(
    webpackDevMiddleware(compiler, {
        logLevel: 'warn',
        publicPath: webpackConfig.output.publicPath
    })
);

// 3.4. 添加hot-middleware，使用compiler参数
const webpackHotMiddlewareInstance = webpackHotMiddleware(compiler, {
    log: console.log,
    path: '/__webpack_hmr',
    heartbeat: 10 * 1000
});
app.use(webpackHotMiddlewareInstance);

// 4. watchContentBase
if (devServerConfig.watchContentBase) {
    // 判断watchoptions
    const {poll, ignored} = devServerConfig.watchOptions;
    const usePolling = poll ? true : undefined;
    const interval = typeof poll === 'number' ? poll : undefined;
    // 这个是chokidar的options
    const options = {
        ignoreInitial: true,
        persistent: true,
        followSymlinks: false,
        atomic: false,
        alwaysStat: true,
        ignorePermissionErrors: true,
        ignored,
        usePolling,
        interval
    };
    function _watch(watchPath) {
        const watcher = chokidar.watch(watchPath, options);
        watcher.on('change', () => {
            // 使用webpackHotMiddlewareInstance发送reload消息给client
            // client接收消息后reload页面
            webpackHotMiddlewareInstance.publish({action: 'reload'});
        });
    }
    if (Array.isArray(contentBase)) {
        contentBase.forEach(item => {
            _watch(item);
        });
    } else if (contentBase && typeof contentBase === 'string') {
        _watch(contentBase);
    }
}

// 6. 增加before支持
if (typeof devServerConfig.before === 'function') {
```

```
        devServerConfig.before(app);
    }

    // 最后，创建server，并且输出地址
    // const server = http.createServer(app);
    // server.listen(port, () => {
    //     console.log(`Listening on http://localhost:${port}`);
    // });

    const ID = 'diy-dev-server';
    return new Promise((resolve, reject) => {
        const server = http.createServer(app);
        const urlForBrowser = `http://${host}:${port}`;
        // 第一次flag，第一次编译就输出log
        let isFirstCompile = true;
        compiler.hooks.done.tap(ID, stats => {
            if (stats.hasErrors()) {
                const info = stats.toJson();
                console.error(info.errors);
                reject(stats);
                return;
            }
            console.log(stats.toString({children: false, modules: false, chunks: false, colors: true}));
            if (isFirstCompile) {
                console.log();
                console.log(`    DevServer running at: ${urlForBrowser}`);
                console.log();
                resolve({
                    server,
                    url: urlForBrowser
                });
            }
        });

        server.listen(port, host, err => {
            if (err) {
                reject(err);
            }
        });
    });
};
```

> Tips：测试的代码这里不再做讲解了，直接打开本篇文章的代码然后执行`npm start`查看我们的 dev-server 效果吧：
>
> 1. HMR：修改`index.js`内容，就可以看到 HMR 效果了，比如把`app.style.background = '#99d';`修改成其它颜色；
> 2. contentBase 和 watch：修改`public/hello.js`内容，保存则会直接接收到 reload 消息，重载页面；
> 3. proxy：可以访问`http://localhost:3000/users/`查看效果；
> 4. mock：可以按照`mock/index.js`的配置访问各接口试下，比如：`http://localhost:3000/repos/hello`；
> 5. webpack.config.js watch：修改下配置文件，服务就重新启动了。

总结

对于后端服务已经是 Express 或者有自己的后端逻辑需要 Express 来实现的时候，例如在我们项目中，后端服务器实际为 PHP+Smarty 模板的，我们就是用自定义的 Express 服务器实现一个 dev-server。它支持 Webpack-dev-server 的全部功能（本文内容就是其中一部功能），还能够利用 PHP `bin`命令来做 Smarty 模板数据 Mock 和模板渲染。本篇文章就是最简单地实现了 Webpack-dev-server 功能，并且我们还可以继续在本节源码技术上扩展自己的 Express 服务器功能。通过本文的内容，可以让我们更好地理解 Webpack-dev-server 和 HMR 内核实现，并且在实际项目中得到应用。

实战：手写一个 prefetch-webpack-plugin 插件

实战：使用 Stats 数据结构生成 Webpack 构建报告

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