Babel和devServer

王红元 coderwhy

为什么需要babel？

- 事实上，在开发中我们很少直接去接触babel，但是**babel对于前端开发**来说，目前是**不可缺少的一部分**：
  - 开发中，我们想要使用ES6+的语法，想要使用TypeScript，开发React项目，它们都是离不开Babel的；
  - 所以，学习Babel对于我们理解代码从编写到线上的转变过程至关重要；

- **那么，Babel到底是什么呢？**
  - Babel是一个工具链，主要用于旧浏览器或者环境中将ECMAScript 2015+代码转换为向后兼容版本的JavaScript；
  - 包括：语法转换、源代码转换等；

```
[1, 2, 3].map((n) => n + 1);

[1, 2, 3].map(function(n) {
  return n + 1;
});
```

Babel命令行使用

- babel本身可以作为**一个独立的工具**（和postcss一样），不和webpack等构建工具配置来单独使用。
- 如果我们希望在命令行尝试使用babel，需要安装如下库：
  - @babel/core：babel的核心代码，必须安装；
  - @babel/cli：可以让我们在命令行使用babel；

```
npm install @babel/cli @babel/core -D
```

- 使用babel来处理我们的源代码：
  - src：是源文件的目录；
  - --out-dir：指定要输出的文件夹dist；

```
npx babel src --out-dir dist
```

插件的使用

- 比如我们需要转换箭头函数，那么我们就可以使用**箭头函数转换相关的插件**：

```
npm install @babel/plugin-transform-arrow-functions -D
```

```
npx babel src --out-dir dist --plugins=@babel/plugin-transform-arrow-functions
```

- 查看转换后的结果：我们会发现 const 并没有转成 var
  - 这是因为 plugin-transform-arrow-functions，并没有提供这样的功能；
  - 我们需要使用 plugin-transform-block-scoping 来完成这样的功能；

```
npm install @babel/plugin-transform-block-scoping -D
```

```
npx babel src --out-dir dist --plugins=@babel/plugin-transform-block-scoping
                                      ,@babel/plugin-transform-arrow-functions
```

Babel的预设preset

- 但是如果要转换的内容过多，一个个设置是比较麻烦的，我们可以使用预设（preset）：
  - 后面我们再具体来讲预设代表的含义；
- 安装@babel/preset-env预设：

```
npm install @babel/preset-env -D
```

- 执行如下命令：

```
npx babel src --out-dir dist --presets=@babel/preset-env
```

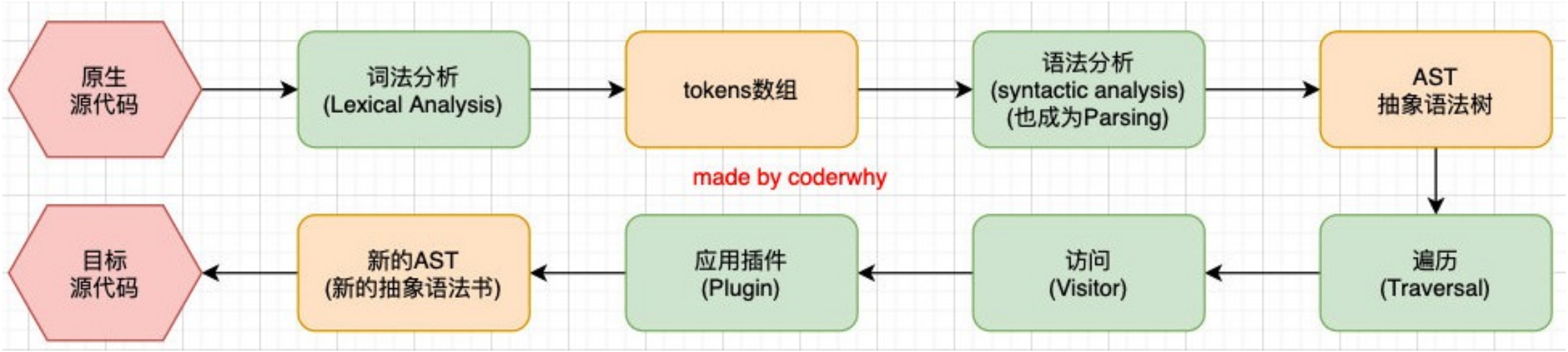
Babel的底层原理

- babel是如何做到将我们的**一段代码（ES6、TypeScript、React）**转成**另外一段代码（ES5）**的呢？
  - 从一种源代码（原生语言）转换成另一种源代码（目标语言），这是什么的工作呢？
  - 就是**编译器**，事实上我们可以将babel看成就是一个编译器。
  - Babel编译器的作用就是将我们的源代码，转换成浏览器可以直接识别的另外一段源代码；
- **Babel也拥有编译器的工作流程：**
  - 解析阶段（Parsing）
  - 转换阶段（Transformation）
  - 生成阶段（Code Generation）
- https://github.com/jamiebuilds/the-super-tiny-compiler

Babel编译器执行原理

- Babel的执行阶段

- 当然，这只是一个简化版的编译器工具流程，在每个阶段又会有自己具体的工作：



babel-loader

- 在实际开发中，我们通常会在构建工具中通过配置babel来对其进行使用的，比如在webpack中。
- 那么我们就需要去安装相关的依赖：
  - 如果之前已经安装了@babel/core，那么这里不需要再次安装；

```
npm install babel-loader @babel/core
```

- 我们可以设置一个规则，在加载js文件时，使用我们的babel：

```
module: {
  rules: [
    {
      test: /\.m?js$/,
      use: {
        loader: "babel-loader"
      }
    }
  ]
},
```

指定使用的插件

- 我们必须指定使用的插件才会生效

```
{
  test: /\.m?js$/,
    use: {
      loader: "babel-loader",
      options: {
        plugins: [
          "@babel/plugin-transform-block-scoping",
          "@babel/plugin-transform-arrow-functions"
        ]
      }
  }
}
```

babel-preset

- 如果我们一个个去安装使用插件，那么需要手动来管理大量的babel插件，我们可以直接给webpack提供一个preset，webpack会根据我们的预设来加载对应的插件列表，并且将其传递给babel。
- 比如常见的预设有三个：
  - env
  - react
  - TypeScript
- 安装preset-env：

```
npm install @babel/preset-env
```

```
{
  test: /\.m?js$/,
  use: {
    loader: "babel-loader",
    options: {
      presets: [
        ["@babel/preset-env"]
      ]
    }
  }
}
```

Babel的配置文件

- 像之前一样，我们可以将babel的配置信息放到一个独立的文件中，babel给我们提供了两种配置文件的编写：
  - babel.config.json（或者.js，.cjs，.mjs）文件；
  - .babelrc.json（或者.babelrc，.js，.cjs，.mjs）文件；
- 它们两个有什么区别呢？目前很多的项目都采用了多包管理的方式（babel本身、element-plus、umi等）；
  - .babelrc.json：早期使用较多的配置方式，但是对于配置Monorepos项目是比较麻烦的；
  - babel.config.json（babel7）：可以直接作用于Monorepos项目的子包，更加推荐；

```
module.exports = {
  presets: [
    ["@babel/preset-env"]
  ]
}
```

Vue源码的打包

- 我们课程主要是学习Vue的，那么我们应该包含Vue相关的代码：

```
import { createApp } from "vue";

// Vue代码
createApp({
  template: '#my-app',
  data() {
    return {
      title: "我是标题",
      content: "我是内容，哈哈哈"
    }
  }
}).mount("#app");
```

- 界面上是没有效果的：
  - 并且我们查看运行的控制台，会发现如下的警告信息；

```
▸ [Vue warn]: Component provided template option but runtime compilation is not supported in this build of Vue. Configure your bundler to alias "vue" to "vue/dist/vue.esm-bundler.js".
  at <App>
```

Vue打包后不同版本解析

- **vue(.runtime).global(.prod).js：**
  - 通过浏览器中的 <script src="..." > 直接使用；
  - 我们之前通过CDN引入和下载的Vue版本就是这个版本；
  - 会暴露一个全局的Vue来使用；
- **vue(.runtime).esm-browser(.prod).js：**
  - 用于通过原生 ES 模块导入使用 (在浏览器中通过 <script type="module"> 来使用)。
- **vue(.runtime).esm-bundler.js：**
  - 用于 webpack，rollup 和 parcel 等构建工具；
  - 构建工具中默认是vue.runtime.esm-bundler.js；
  - 如果我们需要解析模板template，那么需要手动指定vue.esm-bundler.js；
- **vue.cjs(.prod).js：**
  - 服务器端渲染使用；
  - 通过require()在Node.js中使用；

运行时+编译器 vs 仅运行时

- 在Vue的开发过程中我们有**三种方式**来编写DOM元素：
  - 方式一：template模板的方式（之前经常使用的方式）；
  - 方式二：render函数的方式，使用h函数来编写渲染的内容；
  - 方式三：通过.vue文件中的template来编写模板；
- **它们的模板分别是如何处理的呢？**
  - 方式二中的h函数可以直接返回一个**虚拟节点**，也就是**Vnode节点**；
  - 方式一和方式三的template都需要有特定的代码来对其进行解析：
    - ✓ 方式三.vue文件中的template可以通过在vue-loader对其进行编译和处理；
    - ✓ 方式一种的template我们必须要通过源码中一部分代码来进行编译；
- 所以，Vue在让我们选择版本的时候分为 **运行时+编译器** vs **仅运行时**
  - 运行时+编译器包含了对template模板的编译代码，更加完整，但是也更大一些；
  - 仅运行时没有包含对template版本的编译代码，相对更小一些；

全局标识的配置

- 我们会发现控制台还有另外的一个警告：

```
▸You are running the esm-bundler build of Vue. It is recommended to configure your bundler to explicitly replace feature   runtime-core.esm-bu
flag globals with boolean literals to get proper tree-shaking in the final bundle. See http://link.vuejs.org/feature-flags for more details.
```

- 在GitHub上的文档中我们可以找到说明：

Bundler Build Feature Flags

Starting with 3.0.0-rc.3, `esm-bundler` builds now exposes global feature flags that can be overwritten at compile time:

- `__VUE_OPTIONS_API__` (enable/disable Options API support, default: `true`)
- `__VUE_PROD_DEVTOOLS__` (enable/disable devtools support in production, default: `false`)

The build will work without configuring these flags, however it is **strongly recommended** to properly configure them in order to get proper tree-shaking in the final bundle. To configure these flags:

- 这是两个特性的标识，一个是使用Vue的Options，一个是Production模式下是否支持devtools工具；
- 虽然他们都有默认值，但是强烈建议我们手动对他们进行配置；

VSCode对SFC文件的支持

- 在前面我们提到过，真实开发中多数情况下我们都是使用SFC（ **single-file components (单文件组件)** ）。

- 我们先说一下VSCode对SFC的支持：
  - 插件一：Vetur，从Vue2开发就一直在使用的VSCode支持Vue的插件；
  - 插件二：Volar，官方推荐的插件（后续会基于Volar开发官方的VSCode插件）；

编写App.vue代码

- 接下来我们编写自己的App.vue代码：

```vue
<template>
  <h2>{{title}}</h2>
  <p>{{content}}</p>
</template>

<script>
export default {
  data() {
    return {
      title: "我是App标题",
      content: "我是App的内容，哈哈哈"
    }
  }
}
</script>

<style>
  h2 {
    color: red;
  }
  p {
    color: blue;
  }
</style>
```

```js
import { createApp } from "vue/dist/vue.esm-bundler";
import App from './vue/App.vue';

// Vue代码
createApp(App).mount("#app");
```

App.vue的打包过程

- 我们对代码打包会报错：我们需要合适的Loader来处理文件。

```
ERROR in ./src/vue/App.vue 1:0
Module parse failed: Unexpected token (1:0)
You may need an appropriate loader to handle this file type,
ttps://webpack.js.org/concepts#loaders
```

- 这个时候我们需要使用vue-loader：

```
npm install vue-loader -D
```

- 在webpack的模板规则中进行配置：

```
{
  test: /\.vue$/,
  loader: "vue-loader"
}
```

@vue/compiler-sfc

- 打包依然会报错，这是因为我们必须添加@vue/compiler-sfc来对template进行解析：

```
npm install @vue/compiler-sfc -D
```

- 另外我们需要配置对应的Vue插件：

```
const { VueLoaderPlugin } = require('vue-loader/dist/index');

new VueLoaderPlugin()
```

- 重新打包即可支持App.vue的写法

- 另外，我们也可以编写其他的.vue文件来编写自己的组件；

为什么要搭建本地服务器？

- 目前我们开发的代码，为了运行需要有两个操作：
  - 操作一：npm run build，编译相关的代码；
  - 操作二：通过live server或者直接通过浏览器，打开index.html代码，查看效果；
- 这个过程经常操作会影响我们的开发效率，我们希望可以做到，当文件发生变化时，可以自动的完成 编译 和 展示；
- 为了完成自动编译，webpack提供了几种可选的方式：
  - webpack watch mode；
  - webpack-dev-server（常用）；
  - webpack-dev-middleware；

Webpack watch

- webpack给我们提供了watch模式：
  - 在该模式下，webpack依赖图中的所有文件，只要有一个发生了更新，那么代码将被重新编译；
  - 我们不需要手动去运行 npm run build指令了；
- 如何开启watch呢？两种方式：
  - 方式一：在导出的配置中，添加 watch: true；
  - 方式二：在启动webpack的命令中，添加 --watch的标识；
- 这里我们选择方式二，在package.json的 scripts 中添加一个 watch 的脚本：

```
"scripts": {
  "build": "webpack --config wk.config.js",
  "watch": "webpack --watch",
  "type-check": "tsc --noEmit",
  "type-check-watch": "npm run type-check -- --watch"
},
```

webpack-dev-server

- 上面的方式可以监听到文件的变化，但是事实上它本身是没有自动刷新浏览器的功能的：
  - 当然，目前我们可以在VSCode中使用live-server来完成这样的功能；
  - 但是，我们希望在不适用live-server的情况下，可以具备live reloading（实时重新加载）的功能；
- 安装webpack-dev-server

```
npm install webpack-dev-server -D
```

- 修改配置文件，告知 dev server，从什么位置查找文件：

```
devServer: {
  contentBase: "./build"
},
```

```
target: "web",
```

```
"serve": "webpack serve --config wk.config.js",
```

- webpack-dev-server 在编译之后不会写入到任何输出文件。而是将 bundle 文件保留在内存中：
  - 事实上webpack-dev-server使用了一个库叫memfs（memory-fs webpack自己写的）

认识模块热替换（HMR）

- 什么是HMR呢？
  - HMR的全称是Hot Module Replacement，翻译为模块热替换；
  - 模块热替换是指在 应用程序运行过程中，替换、添加、删除模块，而无需重新刷新整个页面；
- HMR通过如下几种方式，来提高开发的速度：
  - 不重新加载整个页面，这样可以保留某些应用程序的状态不丢失；
  - 只更新需要变化的内容，节省开发的时间；
  - 修改了css、js源代码，会立即在浏览器更新，相当于直接在浏览器的devtools中直接修改样式；
- 如何使用HMR呢？
  - 默认情况下，webpack-dev-server已经支持HMR，我们只需要开启即可；
  - 在不开启HMR的情况下，当我们修改了源代码之后，整个页面会自动刷新，使用的是live reloading；

开启HMR

- 修改webpack的配置：

```
devServer: {
  hot: true
},
```

- 浏览器可以看到如下效果：

```
[HMR] Waiting for update signal from WDS...
[WDS] Hot Module Replacement enabled.
[WDS] Live Reloading enabled.
```

- 但是你会发现，当我们修改了某一个模块的代码时，依然是刷新的整个页面：
  - 这是因为我们需要去指定哪些模块发生更新时，进行HMR；

```
if (module.hot) {
  module.hot.accept("./util.js", () => {
    console.log("util更新了");
  })
}
```

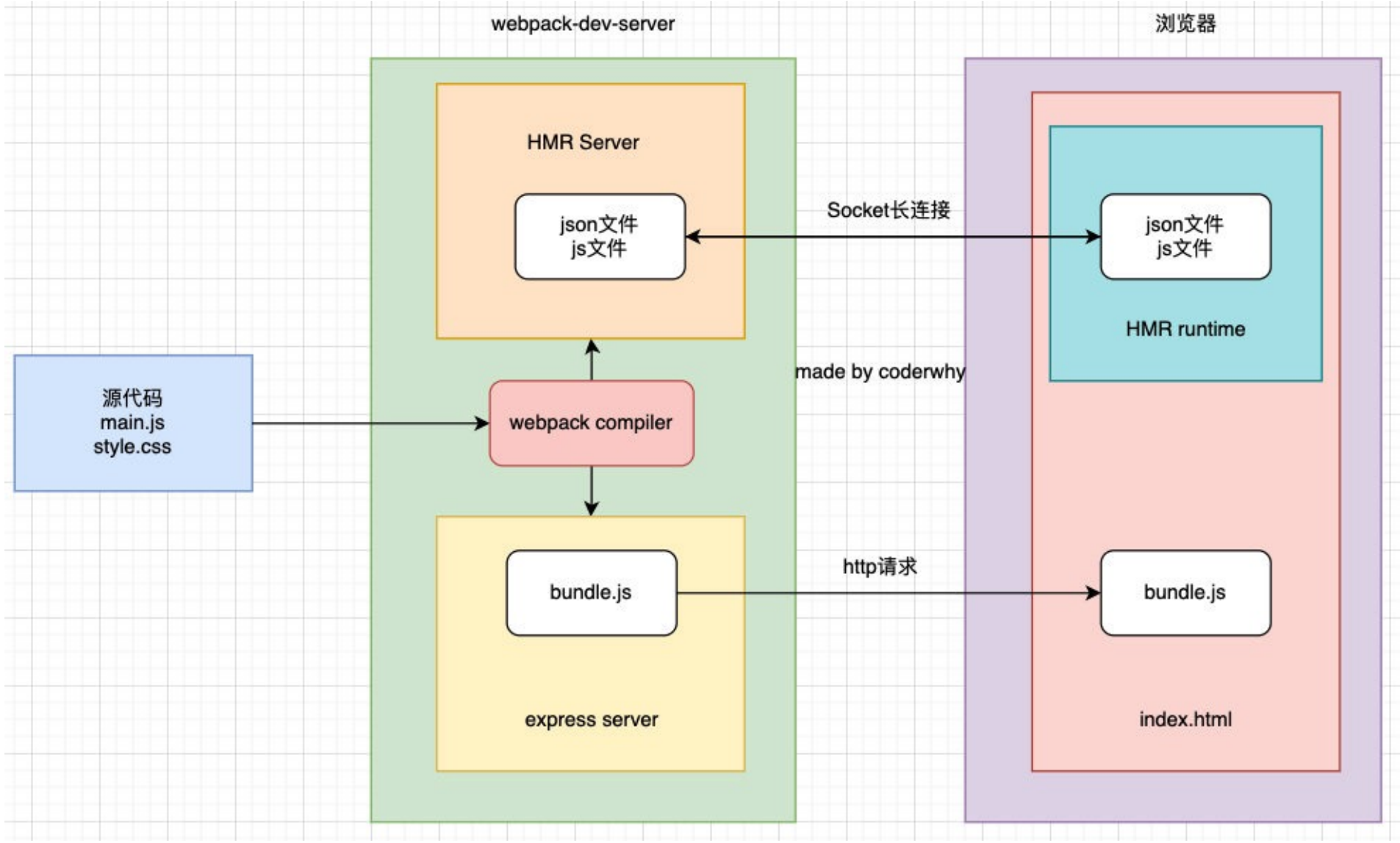
框架的HMR

- 有一个问题：在开发其他项目时，我们是否需要经常手动去写入 module.hot.accpet相关的API呢？
  - 比如开发Vue、React项目，我们修改了组件，希望进行热更新，这个时候应该如何去操作呢？
  - 事实上社区已经针对这些有很成熟的解决方案了：
  - 比如vue开发中，我们使用vue-loader，此loader支持vue组件的HMR，提供开箱即用的体验；
  - 比如react开发中，有React Hot Loader，实时调整react组件（目前React官方已经弃用了，改成使用react-refresh）；
- 接下来我们来演示一下Vue实现一下HMR功能。

HMR的原理

- 那么HMR的原理是什么呢？如何可以做到只更新一个模块中的内容呢？
  - webpack-dev-server会创建两个服务：提供静态资源的服务（express）和Socket服务（net.Socket）；
  - express server负责直接提供静态资源的服务（打包后的资源直接被浏览器请求和解析）；
- HMR Socket Server，是一个socket的长连接：
  - 长连接有一个最好的好处是建立连接后双方可以通信（服务器可以直接发送文件到客户端）；
  - 当服务器监听到对应的模块发生变化时，会生成两个文件.json（manifest文件）和.js文件（update chunk）；
  - 通过长连接，可以直接将这两个文件主动发送给客户端（浏览器）；
  - 浏览器拿到两个新的文件后，通过HMR runtime机制，加载这两个文件，并且针对修改的模块进行更新；

HMR的原理图

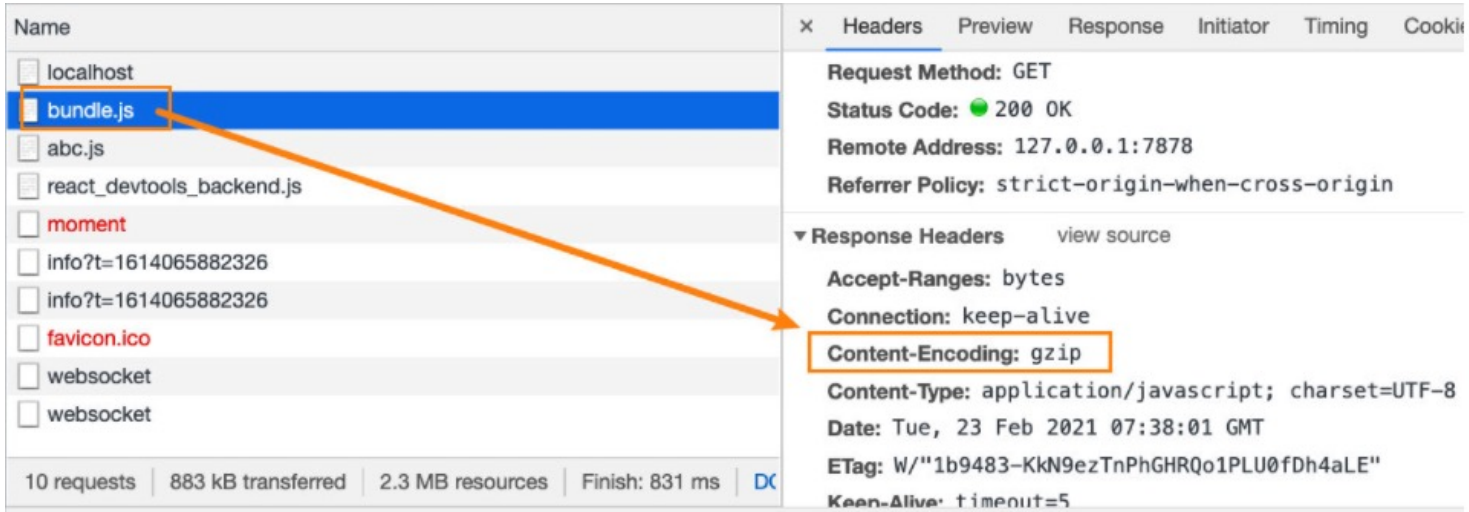

webpack-dev-server
浏览器
HMR Server
json文件
js文件
Socket长连接
json文件
js文件
HMR runtime
源代码
main.js
style.css
webpack compiler
made by coderwhy
bundle.js
http请求
bundle.js
express server
index.html

hotOnly、host配置

- host设置主机地址：
  - 默认值是localhost；
  - 如果希望其他地方也可以访问，可以设置为 0.0.0.0；
- localhost 和 0.0.0.0 的区别：
  - localhost：本质上是一个域名，通常情况下会被解析成127.0.0.1;
  - 127.0.0.1：回环地址(Loop Back Address)，表达的意思其实是我们主机自己发出去的包，直接被自己接收;
    - ✓ 正常的数据库包经常 应用层 - 传输层 - 网络层 - 数据链路层 - 物理层；
    - ✓ 而回环地址，是在网络层直接就被获取到了，是不会经常数据链路层和物理层的;
    - ✓ 比如我们监听 127.0.0.1时，在同一个网段下的主机中，通过ip地址是不能访问的;
  - 0.0.0.0：监听IPV4上所有的地址，再根据端口找到不同的应用程序;
    - ✓ 比如我们监听 0.0.0.0时，在同一个网段下的主机中，通过ip地址是可以访问的;

port、open、compress

- port设置监听的端口，默认情况下是8080
- open是否打开浏览器：
  - 默认值是false，设置为true会打开浏览器；
  - 也可以设置为类似于 Google Chrome等值；
- compress是否为静态文件开启gzip compression：
  - 默认值是false，可以设置为true；

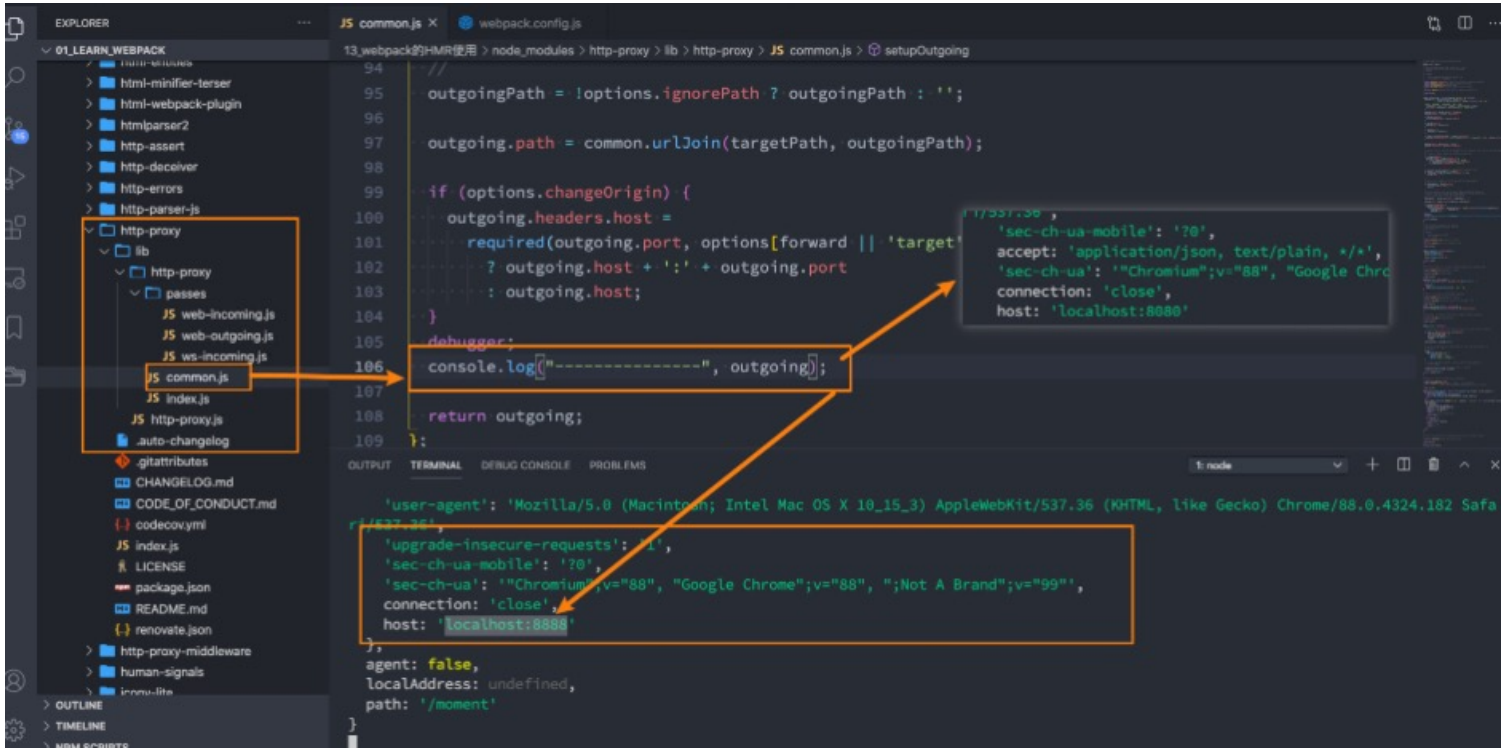


Proxy

- proxy是我们开发中非常常用的一个配置选项，它的目的设置代理来解决跨域访问的问题：
  - 比如我们的一个api请求是 http://localhost:8888，但是本地启动服务器的域名是 http://localhost:8000，这个时候发送网络请求就会出现跨域的问题；
  - 那么我们可以将请求先发送到一个代理服务器，代理服务器和API服务器没有跨域的问题，就可以解决我们的跨域问题了；
- 我们可以进行如下的设置：
  - target：表示的是代理到的目标地址，比如 /api-hy/moment会被代理到 http://localhost:8888/api-hy/moment；
  - pathRewrite：默认情况下，我们的 /api-hy 也会被写入到URL中，如果希望删除，可以使用pathRewrite；
  - secure：默认情况下不接收转发到https的服务器上，如果希望支持，可以设置为false；
  - changeOrigin：它表示是否更新代理后请求的headers中host地址；

changeOrigin的解析

- 这个 changeOrigin官方说的非常模糊，通过查看源码我发现其实是要修改代理请求中的headers中的host属性：
  - 因为我们真实的请求，其实是需要通过 http://localhost:8888来请求的；
  - 但是因为使用了代码，默认情况下它的值时 http://localhost:8000；
  - 如果我们需要修改，那么可以将changeOrigin设置为true即可；



historyApiFallback

- historyApiFallback是开发中一个非常常见的属性，它主要的作用是解决SPA页面在路由跳转之后，进行页面刷新时，返回404的错误。
- boolean值：默认是false
  - 如果设置为true，那么在刷新时，返回404错误时，会自动返回 index.html 的内容；
- object类型的值，可以配置rewrites属性：
  - 可以配置from来匹配路径，决定要跳转到哪一个页面；

- 事实上devServer中实现historyApiFallback功能是通过connect-history-api-fallback库的：
  - 可以查看connect-history-api-fallback 文档

resolve模块解析

- resolve用于设置模块如何被解析：
  - 在开发中我们会有各种各样的模块依赖，这些模块可能来自于自己编写的代码，也可能来自第三方库；
  - resolve可以帮助webpack从每个 require/import 语句中，找到需要引入到合适的模块代码；
  - webpack 使用 enhanced-resolve 来解析文件路径；
- **webpack能解析三种文件路径：**
- 绝对路径
  - 由于已经获得文件的绝对路径，因此不需要再做进一步解析。
- 相对路径
  - 在这种情况下，使用 import 或 require 的资源文件所处的目录，被认为是上下文目录；
  - 在 import/require 中给定的相对路径，会拼接此上下文路径，来生成模块的绝对路径；
- 模块路径
  - 在 resolve.modules中指定的所有目录检索模块；
    - ✓ 默认值是 ['node_modules']，所以默认会从node_modules中查找文件；
  - 我们可以通过设置别名的方式来替换初识模块路径，具体后面讲解alias的配置；

确实文件还是文件夹

- 如果是一个文件：
  - 如果文件具有扩展名，则直接打包文件；
  - 否则，将使用 resolve.extensions选项作为文件扩展名解析；
- 如果是一个文件夹：
  - 会在文件夹中根据 resolve.mainFiles配置选项中指定的文件顺序查找；
    - ✓ resolve.mainFiles的默认值是 ['index']；
    - ✓ 再根据 resolve.extensions来解析扩展名；

extensions和alias配置

- extensions是解析到文件时自动添加扩展名：
  - 默认值是 ['.wasm', '.mjs', '.js', '.json']；
  - 所以如果我们代码中想要添加加载 .vue 或者 jsx 或者 ts 等文件时，我们必须自己写上扩展名；
- 另一个非常好用的功能是配置别名alias：
  - 特别是当我们项目的目录结构比较深的时候，或者一个文件的路径可能需要 ../../../这种路径片段；
  - 我们可以给某些常见的路径起一个别名；

```
resolve: {
  extensions: [".wasm", ".mjs", ".js", ".json", ".jsx", ".ts", ".vue"],
  alias: {
    "@": resolveApp("./src"),
    pages: resolveApp("./src/pages"),
  },
},
```