相信通过上一课时内容的学习，你应该对前端模块化有了更完整的认识。在上一课时的最后我们提出了对模块化打包方案或工具的设想或者说是诉求：

能够将散落的模块打包到一起；

能够编译代码中的新特性；

能够支持不同种类的前端资源模块。

目前，前端领域有一些工具能够很好的满足以上这 3 个需求，其中最为主流的就是 Webpack、Parcel 和 Rollup，我们以 Webpack 为例：

Webpack 作为一个模块打包工具，本身就可以解决模块化代码打包的问题，将零散的 JavaScript 代码打包到一个 JS 文件中。

对于有环境兼容问题的代码，Webpack 可以在打包过程中通过 Loader 机制对其实现编译转换，然后再进行打包。

对于不同类型的前端模块类型，Webpack 支持在 JavaScript 中以模块化的方式载入任意类型的资源文件，例如，我们可以通过 Webpack 实现在 JavaScript 中加载 CSS 文件，被加载的 CSS 文件将会通过 style 标签的方式工作。

除此之外，Webpack 还具备代码拆分的能力，它能够将应用中所有的模块按照我们的需要分块打包。这样一来，就不用担心全部代码打包到一起，产生单个文件过大，导致加载慢的问题。我们可以把应用初次加载所必需的模块打包到一起，其他的模块再单独打包，等到应用工作过程中实际需要用到某个模块，再异步加载该模块，实现增量加载，或者叫作渐进式加载，非常适合现代化的大型 Web 应用。

当然，除了 Webpack，其他的打包工具也都类似，总之，所有的打包工具都是以实现模块化为目标，让我们可以在开发阶段更好的享受模块化带来的优势，同时又不必担心模块化在生产环境中产生新的问题。

Webpack 快速上手

Webpack 作为目前最主流的前端模块打包器，提供了一整套前端项目模块化方案，而不仅仅局限于对 JavaScript 的模块化。通过 Webpack，我们可以轻松的对前端项目开发过程中涉及的所有资源进行模块化。

因为 Webpack 的设计思想比较先进，起初的使用过程比较烦琐，再加上文档也晦涩难懂，所以在最开始的时候，Webpack 对开发者并不友好，但是随着版本的迭代，官方文档的不断更新，目前 Webpack 对开发者已经非常友好了。此外，随着 React 和 Vue.js 这类框架的普及，Webpack 也随之受到了越来越多的关注，现阶段可以覆盖绝大多数现代 Web 应用的开发过程。

接下来我将通过一个案例，带你快速了解 Webpack 的基本使用，具体操作如下所示：

复制代码
```
└── 02-configuation
    ├── src
    │   ├── heading.js
    │   └── index.js
    └── index.html
```
复制代码
```
// ./src/heading.js
export default () => {
  const element = document.createElement('h2')
  element.textContent = 'Hello webpack'
  element.addEventListener('click', () => alert('Hello webpack'))
  return element
}
```
复制代码
```
// ./src/index.js
import createHeading from './heading.js'
const heading = createHeading()
document.body.append(heading)
```
复制代码
```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Webpack - 快速上手</title>
</head>
<body>
  <script type="module" src="src/index.js"></script>
</body>
</html>
```
P.S. type="module" 这种用法是 ES Modules 中提出的标准, 用来区分加载的是一个普通 JS 脚本还是一个模块。

在上面这个案例中，我们创建了两个 JS 文件，其中 heading.js 中以 ES Modules 的方式导出了一个创建元素的函数，然后在 index.js 中导入 heading.js 并使用了这个模块，最后在 html 文件中通过 script 标签，以模块化的方式引入了 index.js，

按照 ES Modules 的标准，这里的 index.html 可以直接在浏览器中正常工作，但是对于不支持 ES Modules 标准的浏览器，直接使用就会出现错误，所以我们需要使用 Webpack 这样的工具，将我们这里按照模块化方式拆分的 JS 代码再次打包到一起。

接下来我们就尝试引入 Webpack 去处理上述案例中的 JS 模块打包。由于 Webpack 是一个 npm 工具模块，所以我们先初始化一个 package.json 文件，用来管理 npm 依赖版本，完成之后，再来安装 Webpack 的核心模块以及它的 CLI 模块，具体操作如下：

复制代码
```
$ npm init --yes
$ npm i webpack webpack-cli --save-dev
```
P.S. webpack 是 Webpack 的核心模块，webpack-cli 是 Webpack 的 CLI 程序，用来在命令行中调用 Webpack。

安装完成之后，webpack-cli 所提供的 CLI 程序就会出现在 node_modules/.bin 目录当中，我们可以通过 npx 快速找到 CLI 并运行它，具体操作如下：

复制代码
```
$ npx webpack --version
v4.42.1
```
P.S. npx 是 npm 5.2 以后新增的一个命令，可以用来更方便的执行远程模块或者项目 node_modules 中的 CLI 程序。

这里我们使用的 Webpack 版本是 v4.42.1，有了 Webpack 后，就可以直接运行 webpack 命令来打包 JS 模块代码，具体操作如下：

复制代码
```
$ npx webpack
```
这个命令在执行的过程中，Webpack 会自动从 src/index.js 文件开始打包，然后根据代码中的模块导入操作，自动将所有用到的模块代码打包到一起。

完成之后，控制台会提示：顺着 index.js 有两个 JS 文件被打包到了一起。与之对应的就是项目的根目录下多出了一个 dist 目录，我们的打包结果就存放在这个目录下的 main.js 文件中，具体操作如下图所示：

这里我们回到 index.html 中修改引入文件的路径，由于打包后的代码就不会再有 import 和 export 了，所以我们可以删除 type="module"。再次回到浏览器中，查看这个页面，这时我们的代码仍然可以正常工作，index.html 的代码如下所示：

复制代码
```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Webpack - 快速上手</title>
</head>
```

```
<body>
  <script src="dist/main.js"></script>
</body>
</html>
```

我们也可以将 Webpack 命令定义到 npm scripts 中，这样每次使用起来会更加方便，具体如下：

复制代码

```
{
  "name": "01-getting-started",
  "version": "0.1.0",
  "main": "n/a",
  "author": "zce <w@zce.me> (https://zce.me)",
  "license": "MIT",
  "scripts": {
    "build": "webpack"
  },
  "devDependencies": {
    "webpack": "^4.42.1",
    "webpack-cli": "^3.3.11"
  }
}
```

对于 Webpack 最基本的使用，总结下来就是：先安装 webpack 相关的 npm 包，然后使用 webpack-cli 所提供的命令行工具进行打包。

配置 Webpack 的打包过程

Webpack 4 以后的版本支持零配置的方式直接启动打包，整个过程会按照约定将 src/index.js 作为打包入口，最终打包的结果会存放到 dist/main.js 中。

但很多时候我们需要自定义这些路径约定，例如，在下面这个案例中，我需要它的打包入口是 src/main.js，那此时我们通过配置文件的方式修改 Webpack 的默认配置，在项目的根目录下添加一个 webpack.config.js，具体结构如下：

复制代码

```
  └── 02-configuation
      ├── src
      │   ├── heading.js
      │   └── main.js
      ├── index.html
      ├── package.json
+     └── webpack.config.js ····· Webpack 配置文件
```

webpack.config.js 是一个运行在 Node.js 环境中的 JS 文件，也就是说我们需要按照 CommonJS 的方式编写代码，这个文件可以导出一个对象，我们可以通过所导出对象的属性完成相应的配置选项。

这里先尝试添加一个 entry 属性，这个属性的作用就是指定 Webpack 打包的入口文件路径。我们将其设置为 src/main.js，具体代码如下所示：

复制代码
```
// ./webpack.config.js
module.exports = {
  entry: './src/main.js'
}
```
配置完成之后，回到命令行终端重新运行打包命令，此时 Webpack 就会从 src/main.js 文件开始打包。

除了 entry 的配置以外，我们还可以通过 output 属性设置输出文件的位置。output 属性的值必须是一个对象，通过这个对象的 filename 指定输出文件的文件名称，path 指定输出的目录，具体代码如下所示：

复制代码
```
// ./webpack.config.js
const path = require('path')

module.exports = {
  entry: './src/main.js',
  output: {
    filename: 'bundle.js',
    path: path.join(__dirname, 'output')
  }
}
```
TIPS：webpack.config.js 是运行在 Node.js 环境中的代码，所以直接可以使用 path 之类的 Node.js 内置模块。

由于 Webpack 支持的配置有很多，篇幅的关系，这里我们就不一一介绍了，详细的文档你可以在 Webpack 的官网中找到：https://webpack.js.org/configuration/#options

让配置文件支持智能提示
在这里，我想跟你分享我在编写 Webpack 配置文件时用过的一个小技巧，因为 Webpack 的配置项比较多，而且很多选项都支持不同类型的配置方式。如果你刚刚接触 Webpack 的配置，这些配置选项一定会让你感到头大。如果开发工具能够为 Webpack 配置文件提供智能提示的话，这种痛苦就会减小很多，配置起来，效率和准确度也会大大提高。

我们知道， VSCode 对于代码的自动提示是根据成员的类型推断出来的，换句话说，如果 VSCode 知道当前变量的类型，就可以给出正确的智能提示。即便你没有使用 TypeScript 这种类型友好的语言，也可以通过类型注释的方式去标注变量的类型。

默认 VSCode 并不知道 Webpack 配置对象的类型，我们通过 import 的方式导入

Webpack 模块中的 Configuration 类型，然后根据类型注释的方式将变量标注为这个类型，这样我们在编写这个对象的内部结构时就可以有正确的智能提示了，具体代码如下所示：

复制代码

```
// ./webpack.config.js
import { Configuration } from 'webpack'

/**
 * @type {Configuration}
 */
const config = {
  entry: './src/index.js',
  output: {
    filename: 'bundle.js'
  }
}

module.exports = config
```

需要注意的是：我们添加的 import 语句只是为了导入 Webpack 配置对象的类型，这样做的目的是为了标注 config 对象的类型，从而实现智能提示。在配置完成后一定要记得注释掉这段辅助代码，因为在 Node.js 环境中默认还不支持 import 语句，如果执行这段代码会出现错误。

复制代码

```
// ./webpack.config.js

// 一定记得运行 Webpack 前先注释掉这里。
// import { Configuration } from 'webpack'

/**
 * @type {Configuration}
 */
const config = {
  entry: './src/index.js',
  output: {
    filename: 'bundle.js'
  }
}

module.exports = config
```

没有智能提示的效果，如下所示：

加上类型标注实现智能提示的效果，如下所示：

使用 import 语句导入 Configuration 类型的方式固然好理解，但是在不同的环境中还是会有各种各样的问题，例如我们这里在 Node.js 环境中，就必须要额外注释掉这个导入类型的语句，才能正常工作。

所以我一般的做法是直接在类型注释中使用 import 动态导入类型，具体代码如下：

复制代码

```
// ./webpack.config.js
/** @type {import('webpack').Configuration} */
const config = {
  entry: './src/index.js',
  output: {
    filename: 'bundle.js'
  }
}
module.exports = config
```

这种方式同样也可以实现载入类型，而且相比于在代码中通过 import 语句导入类型更为方便，也更为合理。

不过需要注意一点，这种导入类型的方式并不是 ES Modules 中的 Dynamic Imports，而是 TypeScript 中提供特性。虽然我们这里只是一个 JavaScript 文件，但是在 VSCode 中的类型系统都是基于 TypeScript 的，所以可以直接按照这种方式使用，详细信息你可以参考这种 import-types 的文档。

其次，这种 @type 类型注释的方式是基于 JSDoc 实现的。JSDoc 中类型注释的用法还有很多，详细可以参考官方文档中对 @type 标签的介绍。

Webpack 工作模式

Webpack 4 新增了一个工作模式的用法，这种用法大大简化了 Webpack 配置的复杂程度。你可以把它理解为针对不同环境的几组预设配置：

production 模式下，启动内置优化插件，自动优化打包结果，打包速度偏慢；

development 模式下，自动优化打包速度，添加一些调试过程中的辅助插件；

none 模式下，运行最原始的打包，不做任何额外处理。

针对工作模式的选项，如果你没有配置一个明确的值，打包过程中命令行终端会打印一个对应的配置警告。在这种情况下 Webpack 将默认使用 production 模式去工作。

production 模式下 Webpack 内部会自动启动一些优化插件，例如，自动压缩打包后的代码。这对实际生产环境是非常友好的，但是打包的结果就无法阅读了。

修改 Webpack 工作模式的方式有两种：

通过 CLI --mode 参数传入；

通过配置文件设置 mode 属性。

上述三种 Webpack 工作模式的详细差异我们不再赘述了，你可以在官方文档中查看：https://webpack.js.org/configuration/mode/

打包结果运行原理
最后，我们来一起学习 Webpack 打包后生成的 bundle.js 文件，深入了解 Webpack 是如何把这些模块合并到一起，而且还能正常工作的。

为了更好的理解打包后的代码，我们先将 Webpack 工作模式设置为 none，这样 Webpack 就会按照最原始的状态进行打包，所得到的结果更容易理解和阅读。

按照 none 模式打包完成后，我们打开最终生成的 bundle.js 文件，如下图所示：

我们可以先把代码全部折叠起来，以便于了解整体的结构，如下图所示：

TIPS：
-VSCode 中折叠代码的快捷键是 Ctrl + K，Ctrl + 0 （macOS：Command + K，Command + 0）

整体生成的代码其实就是一个立即执行函数，这个函数是 Webpack 工作入口（webpackBootstrap），它接收一个 modules 参数，调用时传入了一个数组。

展开这个数组，里面的元素均是参数列表相同的函数。这里的函数对应的就是我们源代码中的模块，也就是说每个模块最终被包裹到了这样一个函数中，从而实现模块私有作用域，如下图所示：

我们再来展开 Webpack 工作入口函数，如下图所示：

这个函数内部并不复杂，而且注释也很清晰，最开始定义了一个 installedModules 对象用于存放或者缓存加载过的模块。紧接着定义了一个 require 函数，顾名思义，这个函数是用

来加载模块的。再往后就是在 require 函数上挂载了一些其他的数据和工具函数，这些暂时不用关心。

这个函数执行到最后调用了 require 函数，传入的模块 id 为 0，开始加载模块。模块 id 实际上就是模块数组的元素下标，也就是说这里开始加载源代码中所谓的入口模块，如下图所示：

为了更好的理解 bundle.js 的执行过程，你可以把它运行到浏览器中，然后通过 Chrome 的 Devtools 单步调试一下。调试过程我单独录制了一个视频，详情见视频（19 分 11 秒）。

写在最后

整体上对于 Webpack 的基本使用其实并不复杂，特别是在 Webpack 4 以后，很多配置都已经被简化了，在这种配置并不复杂的前提下，开发人员对它的掌握程度主要就体现在了是否能够理解它的工作机制和原理上了。

就拿 Webpack 打包过后的结果来说，大多数的开发者其实根本不会关心它内部的结构是怎样的，又是如何运行起来的，总觉得不需要关心，但是当这种“不用关心”的事情越积越多，整个开发过程不可控的点也会随之增多，当出现问题时，也就很难定位问题的根源了。

其实通过我们的探索你会发现，当你打开“黑盒子”后，里面的东西并没有想象的那么复杂，很多时候你离成功就只有一步之遥，而驱使你走向成功的其实是你的好奇心。在我看来，好奇心应该是一个优秀开发者的基本素质，对待未知的好奇就是我们进步的源泉，与君共勉。

总结

最后我来总结一下本课时的重点，你也可以通过这几个重点反思一下掌握与否：

Webpack 是如何满足模块化打包需求的。

Webpack 打包的配置方式以及一个可以实现配置文件智能提示的小技巧。

Webpack 工作模式特性的作用。

通过 Webpack 打包后的结果是如何运行起来的？