在第六讲中我们已经简单了解了 Webpack Dev Server 的一些基本用法和特性，它为我们使用 Webpack 构建的项目，提供了一个比较友好的开发环境和一个用于调试的开发服务器。

使用 Webpack Dev Server 就可以让我们在开发过程中专注编码，因为它可以自动监视我们代码的变化然后自动进行打包，最后通过自动刷新的方式同步到浏览器以便于我们即时预览，效果如下：

自动刷新的问题
但是当你实际去使用 Webpack Dev Server 自动刷新的特性去完成具体的开发任务时，你会发现还是有一些不舒服的地方。

例如，这里是一个编辑器应用，我想要即时调试这个编辑器中内容文本的样式。那正常的操作肯定是我先尝试在编辑器里面去添加一些文本，作为展示样例，再回到开发工具中，找到控制编辑器样式的 CSS 文件，然后进行编辑，具体操作如下：

那这时候我们就能够发现问题了：当我们修改完编辑器文本对应的样式过后，原本想着可以即时看到最新的界面效果，但是这时编辑器中的内容却没有了。

这时就不得不再次回到应用中再来添加一些测试文本，查看样式效果。那如果修改后还是对样式不满意的话，你还需要继续调整样式，调整之后又会面临文本内容丢失的问题。那久而久之你就会发现，自动刷新这个功能还是有点鸡肋，并没有想象的那么好用。

出现这个问题的原因，是因为我们每次修改完代码，Webpack 都可以监视到变化，然后自动打包，再通知浏览器自动刷新，一旦页面整体刷新，那页面中的任何操作状态都将会丢失，所以才会出现我们上面所看到的情况。

但是聪明的你总会想到一些应对方法，例如：

回到代码中先写死一个文本内容到编辑器中，这样即便页面刷新，也不会丢失。
通过代码将编辑器中的内容及时保存到 LocalStorage 之类的地方，刷新过后再取回来。
总之就是：你有问题，我有办法。

确实这些都是好办法，但是也存在一些不足，因为它们都是典型的“有漏补漏”的操作，并不能根治自动刷新导致页面状态丢失问题，而且这些方法都需要去编写一些与业务本身无关的辅助代码，不利于维护。

更好的办法自然是能够实现在页面不刷新的情况下，代码也可以及时的更新到浏览器的页面中，重新执行，避免页面状态丢失。针对这个需求，Webpack 同样可以满足。

模块热替换（HMR）

HMR 全称 Hot Module Replacement，翻译过来叫作“模块热替换”或“模块热更新”。

计算机行业经常听到一个叫作热拔插的名词，指的就是我们可以在一个正在运行的机器上随时插拔设备，机器的运行状态不会受插拔的影响，而且插上去的设备可以立即工作，例如我们电脑上的 USB 端口就可以热拔插。

模块热替换中的“热”和这里提到的“热拔插”是相同的意思，都是指在运行过程中的即时变化。

Webpack 中的模块热替换，指的是我们可以在应用运行过程中，实时的去替换掉应用中的某个模块，而应用的运行状态不会因此而改变。例如，我们在应用运行过程中修改了某个模块，通过自动刷新会导致整个应用的整体刷新，那页面中的状态信息都会丢失；而如果使用的是 HMR，就可以实现只将修改的模块实时替换至应用中，不必完全刷新整个应用。

这里我们可以先来对比一下使用热更新和使用自动刷新两种方式之间的体验差异，我们尝试在项目中开启 HMR 特性，具体效果如下图所示：

有了 HMR 支持后，我们同样先在页面中随意添加一些内容，也就是为页面制造一些运行状态，然后我们回到开发工具中，再来尝试修改文本的样式，保存过后页面并没有整体刷新，而且我们能立即看到最新的样式。这种体验相对于自动刷新会友好很多。

HMR 对于项目中其他代码文件的修改，也可以有相同的热更新体验。你可以再去尝试修改一下 JS 文件，保存过后，浏览器中同样不会刷新页面，而是直接执行了你刚刚修改的这个模块，具体效果如下：

不仅如此，对于非文本文件同样也可以使用热更新。例如这个案例中显示的 Logo 图片，如果你在外部修改了它，同样也可以及时更新到浏览器中。

那这就是 HMR 的作用和体验，HMR 可以算是 Webpack 中最为强大的特性之一，而且也是最受欢迎的特性，因为它确实极大程度地提高了开发者的工作效率。

开启 HMR

对于 HMR 这种强大的功能而言，使用起来并不算特别复杂。接下来我们就一起了解一下如何去实现项目中的 HMR。

HMR 已经集成在了 webpack 模块中了，所以不需要再单独安装什么模块。

使用这个特性最简单的方式就是，在运行 webpack-dev-server 命令时，通过 --hot 参数去开启这个特性。

或者也可以在配置文件中通过添加对应的配置来开启这个功能。那我们这里打开配置文件，这里需要配置两个地方：

首先需要将 devServer 对象中的 hot 属性设置为 true；
然后需要载入一个插件，这个插件是 webpack 内置的一个插件，所以我们先导入 webpack 模块，有了这个模块过后，这里使用的是一个叫作 HotModuleReplacementPlugin 的插件。
具体配置代码如下：

复制代码

```
// ./webpack.config.js
const webpack = require('webpack')

module.exports = {
  // ...
  devServer: {
    // 开启 HMR 特性，如果资源不支持 HMR 会 fallback 到 live reloading
    hot: true
    // 只使用 HMR，不会 fallback 到 live reloading
    // hotOnly: true
  },
  plugins: [
    // ...
    // HMR 特性所需要的插件
    new webpack.HotModuleReplacementPlugin()
  ]
}
```

配置完成以后，我们打开命令行终端，运行 webpack-dev-server，启动开发服务器。那接下来你就可以来体验 HMR 了。

我们回到开发工具中，这里我们先来尝试修改一下 CSS 文件。样式文件修改保存过后，确实能够以不刷新的形式更新到页面中。

然后我们再来尝试一下修改 JS 文件。保存过后你会发现，这里的页面依然自动刷新了，好像并没有之前所说 HMR 的体验。

为了再次确认，你可以尝试先在页面中的编辑器里随意添加一些文字，然后修改代码，保存过后你就会看到页面自动刷新，页面中的状态也就丢失了，具体效果如下图：

那这是为什么呢？为什么 CSS 文件热替换没出现问题，而到了 JS 这块就不行了呢？我们又该如何去实现其他类型模块的热替换呢？

HMR 的疑问

通过之前的体验我们发现模块热替换确实提供了非常友好的体验，但是当我们自己去尝试开启 HMR 过后，效果却不尽如人意。

很明显：HMR 并不像 Webpack 的其他特性一样可以开箱即用，需要有一些额外的操作。

具体来说，Webpack 中的 HMR 需要我们手动通过代码去处理，当模块更新过后该，如何把更新后的模块替换到页面中。

Q1：可能你会问，为什么我们开启 HMR 过后，样式文件的修改就可以直接热更新呢？我们好像也没有手动处理样式模块的更新啊？

A1：这是因为样式文件是经过 Loader 处理的，在 style-loader 中就已经自动处理了样式文件的热更新，所以就不需要我们额外手动去处理了。

Q2：那你可能会想，凭什么样式就可以自动处理，而我们的脚本就需要自己手动处理呢？

A2：这个原因也很简单，因为样式模块更新过后，只需要把更新后的 CSS 及时替换到页面中，它就可以覆盖掉之前的样式，从而实现更新。

而我们所编写的 JavaScript 模块是没有任何规律的，你可能导出的是一个对象，也可能导出的是一个字符串，还可能导出的是一个函数，使用时也各不相同。所以 Webpack 面对这些毫无规律的 JS 模块，根本不知道该怎么处理更新后的模块，也就无法直接实现一个可以通用所有情况的模块替换方案。

那这就是为什么样式文件可以直接热更新，而 JS 文件更新后页面还是回退到自动刷新的原因。

Q3：那可能还有一些平时使用 vue-cli 或者 create-react-app 这种框架脚手架工具的人会说，“我的项目就没有手动处理，JavaScript 代码照样可以热替换，也没你说的那么麻烦”。

A3：这是因为你使用的是框架，使用框架开发时，我们项目中的每个文件就有了规律，例如 React 中要求每个模块导出的必须是一个函数或者类，那这样就可以有通用的替换办法，所以这些工具内部都已经帮你实现了通用的替换操作，自然就不需要手动处理了。

当然如果你之前没有接触过这样的工具，那你可以忽略这一条，这也并不影响后面的理解。

综上所述，我们还是需要自己手动通过代码来处理，当 JavaScript 模块更新过后，该如何将更新后的模块替换到页面中。

HMR APIs

HotModuleReplacementPlugin 为我们的 JavaScript 提供了一套用于处理 HMR 的 API，我们需要在我们自己的代码中，使用这套 API 将更新后的模块替换到正在运行的页面中。

接下来我们回到代码中，尝试通过 HMR 的 API 手动处理模块更新后的热替换。

这里我们打开 main.js，具体代码如下：

复制代码

```
// ./src/main.js
import createEditor from './editor'
import logo from './icon.png'
import './global.css'

const img = new Image()
img.src = logo
document.body.appendChild(img)

const editor = createEditor()
document.body.appendChild(editor)
```

这是 Webpack 打包的入口文件，正常情况下，在这个文件中会加载一些其他模块。正是因为在 main.js 中使用了这些模块，所以一旦这些模块更新了过后，我们在 main.js 中就必须重新使用更新后的模块。

所以说，我们需要在这个文件中添加一些额外的代码，去处理它所依赖的这些模块更新后的热替换逻辑。

对于开启 HMR 特性的环境中，我们可以访问到全局的 module 对象中的 hot 成员，这个成员是一个对象，这个对象就是 HMR API 的核心对象，它提供了一个 accept 方法，用于注册当某个模块更新后的处理函数。accept 方法第一个参数接收的就是所监视的依赖模块路径，第二个参数就是依赖模块更新后的处理函数。

那我们这里先尝试注册 ./editor 模块更新过后的处理函数，第一个参数就是 editor 模块的路径，第二个参数则需要我们传入一个函数，然后在这个函数中打印一个消息，具体代码如下：

复制代码

```
// ./main.js

// ... 原本的业务代码

module.hot.accept('./editor', () => {
  // 当 ./editor.js 更新，自动执行此函数
  console.log('editor 更新了～～')
})
```

完成过后，我们打开命令行终端再次启动 webpack-dev-server 命令，然后回到浏览器，打开开发人员工具。

此时，如果我们修改了 editor 模块，保存过后，浏览器的控制台中就会自动打印我们上面

在代码中添加的消息，而且浏览器也不会自动刷新了。

那也就是说一旦这个模块的更新被我们手动处理了，就不会触发自动刷新；反之，如果没有手动处理，热替换会自动 fallback（回退）到自动刷新。

JS 模块热替换

了解了这个 HMR API 的作用过后，接下来需要考虑的就是：具体如何实现 editor 模块的热替换。

这个模块导出的是一个 createEditor 函数，我们先正常把它打印到控制台，然后在模块更新后的处理函数中再打印一次，具体代码如下：

复制代码

```
// ./main.js
import createEditor from './editor'

// ... 原本的业务代码

console.log(createEditor)
module.hot.accept('./editor', () => {
  console.log(createEditor)
})
```

这个时候如果你再次修改 editor 模块，保存过后，你就会发现当模块更新后，我们这里拿到的 createEditor 函数也就更新为了最新的结果，具体结果如下图所示：

既然模块文件更新后 createEditor 函数可以自动更新，那剩下的就好办了。我们这里使用 createEditor 函数是用来创建一个界面元素的，那模块一旦更新了，这个元素也就需要重新创建，所以我们这里先移除原来的元素，然后再调用更新后的 createEditor 函数，创建一个新的元素追加到页面中，具体代码如下：

复制代码

```
// ./main.js
import createEditor from './editor'

const editor = createEditor()
document.body.appendChild(editor)

// ... 原本的业务代码

// HMR ----------------------------------
module.hot.accept('./editor', () => {
  document.body.removeChild(editor) // 移除之前创建的元素
```

```
    const newEditor = createEditor() // 用新模块创建新元素
    document.body.appendChild(newEditor)
})
```

但如果只是这样实现的话，一次热替换结束后，第二次就没法再实现热替换了。因为第二次执行这个函数的时候，editor 变量指向的元素已经在上一次执行时被移除了，所以我们这里还应该记录下来每次热替换创建的新元素，以便于下一次热替换时的操作，具体代码如下：

复制代码

```
// ./main.js
import createEditor from './editor'

const editor = createEditor()
document.body.appendChild(editor)

// ... 原本的业务代码

// HMR ----------------------------------
let lastEditor = editor
module.hot.accept('./editor', () => {
    document.body.removeChild(lastEditor) // 移除之前创建的元素
    lastEditor = createEditor() // 用新模块创建新元素
    document.body.appendChild(lastEditor)
})
```

完成以后，我们再来尝试修改 editor 模块，此时就应该是正常的热替换效果了，具体效果如下图：

热替换的状态保持

此时，如果我们尝试在界面上输入一些内容（形成页面操作状态），然后回到代码中再次修改 editor 模块。那此时你仍然会发现问题，由于热替换时，把界面上之前的编辑器元素移除了，替换成了一个新的元素，所以页面上之前的状态同样会丢失。

这也就证明我们的热替换操作还需要改进，我们必须在替换时把状态保留下来。

我们回到 main.js 中，要想保留这个状态也很简单，就是在替换前先拿到编辑器中的内容，然后替换后在放回去就行了。那因为我这里使用的是可编辑元素，而不是文本框，所以我们需要通过 innerHTML 拿到之前编辑的内容，然后设置到更新后创建的新元素中，具体代码如下：

复制代码

```
// ./main.js
import createEditor from './editor'
```

```
const editor = createEditor()
document.body.appendChild(editor)

// ... 原本的业务代码

// HMR --------------------------------
let lastEditor = editor
module.hot.accept('./editor', () => {
  // 当 editor.js 更新，自动执行此函数
  // 临时记录更新前编辑器内容
  const value = lastEditor.innerHTML
  // 移除更新前的元素
  document.body.removeChild(lastEditor)
  // 创建新的编辑器
  // 此时 createEditor 已经是更新过后的函数了
  lastEditor = createEditor()
  // 还原编辑器内容
  lastEditor.innerHTML = value
  // 追加到页面
  document.body.appendChild(lastEditor)
})
```

这样就可以解决界面状态保存的问题了。

至此，对于 editor 模块的热替换逻辑就算是全部实现了。通过这个过程你应该能够发现，为什么 Webpack 需要我们自己处理 JS 模块的热更新了：因为不同的模块有不同的情况，不同的情况，在这里处理时肯定也是不同的。就好像，我们这里是一个文本编辑器应用，所以需要保留状态，如果不是这种类型那就不需要这样做。所以说 Webpack 没法提供一个通用的 JS 模块替换方案。

图片模块热替换

相比于 JavaScript 模块热替换，图片的热替换逻辑就简单多了，这里我们快速来看一下。

我们同样通过 module.hot.accept 注册这个图片模块的热替换处理函数，在这个函数中，我们只需要重新给图片元素的 src 设置更新后的图片路径就可以了。因为图片修改过后图片的文件名会发生变化，而这里我们就可以直接得到更新后的路径，所以重新设置图片的 src 就能实现图片热替换，具体代码如下：

复制代码

```
// ./src/main.js
import logo from './icon.png'
// ... 其他代码
module.hot.accept('./icon.png', () => {
  // 当 icon.png 更新后执行
  // 重写设置 src 会触发图片元素重新加载，从而局部更新图片
```

```
  img.src = logo
})
```

常见问题

如果你刚开始使用 Webpack 的 HMR 特性，肯定会遇到一些问题，接下来我分享几个最容易发生的问题。

第一个问题，如果处理热替换的代码（处理函数）中有错误，结果也会导致自动刷新。例如我们这里在处理函数中故意加入一个运行时错误，代码如下：

复制代码

```
// ./src/main.js
// ... 其他代码
module.hot.accept('./editor', () => {
  // 刻意造成运行异常
  undefined.foo()
})
```

直接测试你会发现 HMR 不会正常工作，而且根本看不到异常，效果如下图：

这是因为 HMR 过程报错导致 HMR 失败，HMR 失败过后，会自动回退到自动刷新，页面一旦自动刷新，控制台中的错误信息就会被清除，这样的话，如果不是很明显的错误，就很难被发现。

在这种情况下，我们可以使用 hotOnly 的方式来解决，因为现在使用的 hot 方式，如果热替换失败就会自动回退使用自动刷新，而 hotOnly 的情况下并不会使用自动刷新。

我们回到配置文件中，这里我们将 devServer 中的 hot 等于 true 修改为 hotOnly 等于 true，具体代码如下：

复制代码

```
// ./webpack.config.js
const webpack = require('webpack')

module.exports = {
  // ...
  devServer: {
    // 只使用 HMR，不会 fallback 到 live reloading
    hotOnly: true
  },
  plugins: [
    // ...
    // HMR 特性所需要的插件
    new webpack.HotModuleReplacementPlugin()
```

```
  ]
}
```

配置完成以后，重新启动 webpack-dev-server。此时我们再去修改代码，无论是否处理了这个代码模块的热替换逻辑，浏览器都不会自动刷新了，这样的话，热替换逻辑中的错误信息就可以直接看到了，具体效果如下图：

第二个问题，对于使用了 HMR API 的代码，如果我们在没有开启 HMR 功能的情况下运行 Webpack 打包，此时运行环境中就会报出 Cannot read property 'accept' of undefined 的错误，具体错误信息如下：

原因是 module.hot 是 HMR 插件提供的成员，没有开启这个插件，自然也就没有这个对象。

解决办法也很简单，与我们在业务代码中判断 API 兼容一样，我们先判断是否存在这个对象，然后再去使用就可以了，具体代码如下：

复制代码

```
// HMR ----------------------------------
if (module.hot) { // 确保有 HMR API 对象
  module.hot.accept('./editor', () => {
    // ...
  })
}
```

除此之外，可能你还有一个问题：我们在代码中写了很多与业务功能本身无关的代码，会不会对生产环境有影响？

那这个问题的答案很简单，我通过一个简单的操作来帮你解答，我们回到配置文件中，确保已经将热替换特性关闭，并且移除掉了 HotModuleReplacementPlugin 插件，然后打开命令行终端，正常运行一下 Webpack 打包，打包过后，我们找到打包生成的 bundle.js 文件，然后找到里面 main.js 对应的模块，具体结果如下图：

你会发现之前我们编写的处理热替换的代码都被移除掉了，只剩下一个 if (false) 的空判断，这种没有意义的判断，在压缩过后也会自动去掉，所以根本不会对生产环境有任何影响。

写在最后

以上就是我们对 Webpack 模块热替换特性做的一些探索，整体下来可能你会觉得 HMR 比较麻烦，需要写一些额外的代码，甚至觉得不如不用。

我个人的看法是利大于弊，这个道理就像是为什么现在的开发者都愿意写单元测试一样，对

于长期开发的项目而言，这点额外的工作不算什么，而且如果你能为自己的代码设计出一些规律，那你也可以实现一个通用替换方案。

那当然，如果你是使用 React 或者 Vue.js 这类的框架开发，那么使用 HMR 功能会更加简单，因为大部分框架都有成熟的 HMR 方案，你只需要使用就可以了。但是如果你是使用纯原生 JavaScript 开发，那 HMR 功能使用起来相对就会麻烦一点。这也正是为什么大部分人都喜欢选择集成式框架的原因。

关于框架的 HMR，因为在大多数情况下是开箱即用的，所以这里不做过多介绍，详细可以参考：

React HMR 方案；
Vue.js HMR 方案。