你好，我是汪磊，通过前面几个课时的学习，相信你已经了解了 Webpack 的相关概念、基本用法，以及核心工作原理，看似好像已经掌握了 Webpack，但是如果以目前的认知状态去应对日常的开发工作，其实还远远不够。

因为“编写源代码 → Webpack 打包 → 运行应用 → 浏览器查看”这种周而复始的开发方式过于原始，在实际开发过程中，如果你还是按照这种方式来工作，开发效率必然会十分低下。

那究竟该如何提高我们的开发效率呢？

这里我先对一个较为理想的开发环境做出设想：

首先，它必须能够使用 HTTP 服务运行而不是文件形式预览。这样的话，一来更接近生产环境状态，二来我们的项目可能需要使用 AJAX 之类的 API，以文件形式访问会产生诸多问题。
其次，在我们修改完代码过后，Webpack 能够自动完成构建，然后浏览器可以即时显示最新的运行结果，这样就大大减少了开发过程中额外的重复操作，同时也会让我们更加专注，效率自然得到提升。
最后，它还需要能提供 Source Map 支持。这样一来，运行过程中出现的错误就可以快速定位到源代码中的位置，而不是打包后结果中的位置，更便于我们快速定位错误、调试应用。
对于以上的这些需求 Webpack 都已经提供了相对应的功能，其中部分功能需要用到一些周边的工具，具体效果如下：

所以，今天我们的主题就是：学习如何增强使用 Webpack 的开发体验。

Webpack 自动编译
正如前面所讲，如果我们每次修改完代码，都是通过命令行手动重复运行 Webpack 命令，从而得到最新的打包结果，那么这样的操作过程根本没有任何开发体验可言。

针对上述这个问题，我们可以使用 Webpack CLI 提供的另外一种 watch 工作模式来解决。

如果你之前了解过其它的一些构建工具，你应该对 watch 模式并不陌生。在这种模式下，Webpack 完成初次构建过后，项目中的源文件会被监视，一旦发生任何改动，Webpack 都会自动重新运行打包任务。

具体的用法也非常简单，就是在启动 Webpack 时，添加一个 --watch 的 CLI 参数，这样的话，Webpack 就会以监视模式启动运行。在打包完成过后，CLI 不会立即退出，它会等待文件变化再次工作，直到我们手动结束它或是出现不可控的异常。

在 watch 模式下我们就只需专注编码，不必再去手动完成编译工作了，相比于原始手动操作的方式，有了很明显的进步。

我们还可以再开启另外一个命令行终端，同时以 HTTP 形式运行我们的应用，然后打开浏

览器去预览应用。

我们可以将浏览器移至屏幕的左侧，然后将编辑器移至右侧，此时我们尝试修改源代码，保存过后，以 watch 模式工作的 Webpack 就会自动重新打包，然后我们就可以在浏览器中刷新页面查看最新的结果，具体效果如下图所示：

那此时我们的开发体验就是：修改代码 → Webpack 自动打包 → 手动刷新浏览器 → 预览运行结果。

P.S. 这里我使用的静态文件服务器是一个 npm 模块，叫作 serve。

此时距离我们的目标状态还差一点，如果浏览器能够在 Webpack 打包过后自动刷新，那我们的开发体验将会更好一些。

如果你已经了解过一个叫作 BrowserSync 的工具，你应该知道 BrowserSync 就可以帮我们实现文件变化过后浏览器自动刷新的功能。

所以，我们就可以使用 BrowserSync 工具替换 serve 工具，启动 HTTP 服务，这里还需要同时监听 dist 目录下文件的变化，具体命令如下：

复制代码
```
# 可以先通过 npm 全局安装 browser-sync 模块，然后再使用这个模块
$ npm install browser-sync --global
$ browser-sync dist --watch

# 或者也可以使用 npx 直接使用远端模块
$ npx browser-sync dist --watch
```
启动过后，我们回到编辑器，然后尝试修改源文件，保存完成以后浏览器就会自动刷新，显示最新结果。

它的原理就是 Webpack 监视源代码变化，自动打包源代码到 dist 中，而 dist 中文件的变化又被 BrowserSync 监听了，从而实现自动编译并且自动刷新浏览器的功能，整个过程由两个工具分别监视不同的内容。

这种 watch 模式 + BrowserSync 虽然也实现了我们的需求，但是这种方法有很多弊端：

操作烦琐，我们需要同时使用两个工具，那么需要了解的内容就会更多，学习成本大大提高；
效率低下，因为整个过程中， Webpack 会将文件写入磁盘，BrowserSync 再进行读取。过程中涉及大量磁盘读写操作，必然会导致效率低下。
所以这只能算是“曲线救国”，并不完美，我们仍然需要继续改善。

Webpack Dev Server

webpack-dev-server 是 Webpack 官方推出的一款开发工具，根据它的名字我们就应该知道，它提供了一个开发服务器，并且将自动编译和自动刷新浏览器等一系列对开发友好的功能全部集成在了一起。

Webpack 官方推出 webpack-dev-server 这款工具的初衷，就是为了提高开发者日常的开发效率，使用这个工具就可以解决我在开头所提出的问题。而且它是一个高度集成的工具，使用起来十分的方便。

webpack-dev-server 同样也是一个独立的 npm 模块，所以我们需要通过 npm 将 webpack-dev-server 作为项目的开发依赖安装。安装完成过后，这个模块为我们提供了一个叫作 webpack-dev-server 的 CLI 程序，我们同样可以直接通过 npx 直接去运行这个 CLI，或者把它定义到 npm scripts 中，具体操作如下：

复制代码
```
# 安装 webpack-dev-server
$ npm install webpack-dev-server --save-dev
# 运行 webpack-dev-server
$ npx webpack-dev-server
```
运行 webpack-dev-server 这个命令时，它内部会启动一个 HTTP Server，为打包的结果提供静态文件服务，并且自动使用 Webpack 打包我们的应用，然后监听源代码的变化，一旦文件发生变化，它会立即重新打包，大致流程如下：

不过这里需要注意的是，webpack-dev-server 为了提高工作速率，它并没有将打包结果写入到磁盘中，而是暂时存放在内存中，内部的 HTTP Server 也是从内存中读取这些文件的。这样一来，就会减少很多不必要的磁盘读写操作，大大提高了整体的构建效率。

我们还可以为 webpack-dev-server 命令传入一个 --open 的参数，用于自动唤起浏览器打开我们的应用。打开浏览器过后，此时如果你有两块屏幕，就可以把浏览器放到另外一块屏幕上，然后体验一边编码，一边即时预览的开发环境了。

配置选项

Webpack 配置对象中可以有一个叫作 devServer 的属性，专门用来为 webpack-dev-server 提供配置，具体如下：

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

module.exports = {
  // ...
```

```
  devServer: {
    contentBase: path.join(__dirname, 'dist'),
    compress: true,
    port: 9000
    // ...
    // 详细配置文档：https://webpack.js.org/configuration/dev-server/
  }
}
```

具体有哪些配置我们就不在这里一一介绍了，你可以在它的官网中找到相对应的说明文档。

接下来我们来看几个 webpack-dev-server 的常用功能。

静态资源访问

webpack-dev-server 默认会将构建结果和输出文件全部作为开发服务器的资源文件，也就是说，只要通过 Webpack 打包能够输出的文件都可以直接被访问到。但是如果你还有一些没有参与打包的静态文件也需要作为开发服务器的资源被访问，那你就需要额外通过配置“告诉” webpack-dev-server。

具体的方法就是在 webpack-dev-server 的配置对象中添加一个对应的配置。我们回到配置文件中，找到 devServer 属性，它的类型是一个对象，我们可以通过这个 devServer 对象的 contentBase 属性指定额外的静态资源路径。这个 contentBase 属性可以是一个字符串或者数组，也就是说你可以配置一个或者多个路径。具体配置如下：

复制代码

```
// ./webpack.config.js
module.exports = {
  // ...
  devServer: {
    contentBase: 'public'
  }
}
```

我们这里将这个路径设置为项目中的 public 目录。可能有人会有疑问，之前我们在使用插件的时候已经将这个目录通过 copy-webpack-plugin 输出到了输出目录，按照刚刚的说法，所有输出的文件都可以直接被 serve，也就是能直接访问到，按道理应该不需要再作为开发服务器的静态资源路径了。

确实是这样的，而且如果你能想到这一点，也就证明你真正理解了 webpack-dev-server 的文件加载规则。

但是在实际使用 Webpack 时，我们一般都会把 copy-webpack-plugin 这种插件留在上线前的那一次打包中使用，而开发过程中一般不会用它。因为在开发过程中，我们会频繁重复执行打包任务，假设这个目录下需要拷贝的文件比较多，如果每次都需要执行这个插件，那打包过程开销就会比较大，每次构建的速度也就自然会降低。

至于如何实现某些插件只在生产模式打包时使用，是额外的话题，所以具体的操作方式会在 10 课时中详细介绍。这里我们先移除 CopyWebpackPlugin，确保这里的打包不会输出 public 目录中的静态资源文件，然后回到命令行再次执行 webpack-dev-server。

启动过后，我们打开浏览器，这里我们访问的页面文件和 bundle.js 文件均来自于打包结果。我们再尝试访问 favicon.ico，因为这个文件已经没有参与打包了，所以这个文件必然来源于 contentBase 中配置的目录了。

Proxy 代理

由于 webpack-dev-server 是一个本地开发服务器，所以我们的应用在开发阶段是独立运行在 localhost 的一个端口上，而后端服务又是运行在另外一个地址上。但是最终上线过后，我们的应用一般又会和后端服务部署到同源地址下。

那这样就会出现一个非常常见的问题：在实际生产环境中能够直接访问的 API，回到我们的开发环境后，再次访问这些 API 就会产生跨域请求问题。

可能有人会说，我们可以用跨域资源共享（CORS）解决这个问题。确实如此，如果我们请求的后端 API 支持 CORS，那这个问题就不成立了。但是并不是每种情况下服务端的 API 都支持 CORS。如果前后端应用是同源部署，也就是协议 / 域名 / 端口一致，那这种情况下，根本没必要开启 CORS，所以跨域请求的问题仍然是不可避免的。

那解决这种开发阶段跨域请求问题最好的办法，就是在开发服务器中配置一个后端 API 的代理服务，也就是把后端接口服务代理到本地的开发服务地址。

webpack-dev-server 就支持直接通过配置的方式，添加代理服务。接下来，我们来看一下它的具体用法。

这里我们假定 GitHub 的 API 就是我们应用的后端服务，那我们的目标就是将 GitHub API 代理到本地开发服务器中。

我们可以先在浏览器中尝试访问其中的一个接口，具体结果如下图：

GitHub API 的 Endpoint 都是在根目录下，也就是说不同的 Endpoint 只是 URL 中的路径部分不同，例如 https://api.github.com/users 和 https://api.github.com/events。

知道 API 地址的规则过后，我们回到配置文件中，在 devServer 配置属性中添加一个 proxy 属性，这个属性值需要是一个对象，对象中的每个属性就是一个代理规则配置。

属性的名称是需要被代理的请求路径前缀，一般为了辨别，我都会设置为 /api。值是所对应的代理规则配置，我们将代理目标地址设置为 https://api.github.com，具体代码如下：

复制代码

```
// ./webpack.config.js
module.exports = {
  // ...
  devServer: {
    proxy: {
      '/api': {
        target: 'https://api.github.com'
      }
    }
  }
}
```

那此时我们请求 http://localhost:8080/api/users ，就相当于请求了 https://api.github.com/api/users。

而我们真正希望请求的地址是 https://api.github.com/users，所以对于代理路径开头的 /api 我们要重写掉。我们可以添加一个 pathRewrite 属性来实现代理路径重写，重写规则就是把路径中开头的 /api 替换为空，pathRewrite 最终会以正则的方式来替换请求路径。

复制代码

```
// ./webpack.config.js
module.exports = {
  // ...
  devServer: {
    proxy: {
      '/api': {
        target: 'https://api.github.com',
        pathRewrite: {
          '^/api': '' // 替换掉代理地址中的 /api
        }
      }
    }
  }
}
```

这样我们代理的地址就正常了。

除此之外，我们还需设置一个 changeOrigin 属性为 true。这是因为默认代理服务器会以我们实际在浏览器中请求的主机名，也就是 localhost:8080 作为代理请求中的主机名。而一般服务器需要根据请求的主机名判断是哪个网站的请求，那 localhost:8080 这个主机名，对于

GitHub 的服务器来说，肯定无法正常请求，所以需要修改。

将代理规则配置的 changeOrigin 属性设置为 true，就会以实际代理请求地址中的主机名去请求，也就是我们正常请求这个地址的主机名是什么，实际请求 GitHub 时就会设置成什么。

复制代码

```
// ./webpack.config.js
module.exports = {
  // ...
  devServer: {
    proxy: {
      '/api': {
        target: 'https://api.github.com',
        pathRewrite: {
          '^/api': '' // 替换掉代理地址中的 /api
        },
        changeOrigin: true // 确保请求 GitHub 的主机名就是：api.github.com
      }
    }
  }
}
```

完成以后，打开命令行终端，运行 webpack-dev-server。然后打开浏览器，这里我们直接尝试请求 http://localhost:8080/api/users，得到的就是 GitHub 的用户数据。因为这个地址已经被代理到了 GitHub 的用户数据接口。

此时，我们就可以回到代码中使用代理后的本地同源地址去请求后端接口，而不必担心出现跨域问题了。

写在最后

最后再来总结一下今天的内容，今天跟你分享了一个叫作 webpack-dev-server 的工具，它是 Webpack 周边工具中最重要的一个，作用就是提升开发者的开发体验，帮助开发者更快更高效的完成开发工作。

当然 webpack-dev-server 提供的体验还不止如此，它还可以提供一种叫作“模块热替换”的开发体验，这一块内容相对复杂一些，我会在 08 课时中详细介绍。

另外我想说，现代化的前端开发过程已经非常方便了，如果你还在使用原始“刀耕火种”的方式进行开发，就一定要尝试一下这些现代化的工具。