Printable
概念

本质上，**webpack** 是一个用于现代 JavaScript 应用程序的_静态模块打包工具_。当 webpack 处理应用程序时，它会在内部构建一个 依赖图(dependency graph)，此依赖图对应映射到项目所需的每个模块，并生成一个或多个 *bundle*。

> **Tip**
>
> 可以在 这里 了解更多关于 JavaScript 模块和 webpack 模块的信息。

从 v4.0.0 开始，__webpack 可以不用再引入一个配置文件__来打包项目，然而，它仍然有着 高度可配置性，可以很好满足你的需求。

在开始前你需要先理解一些__核心概念__：

- 入口(entry)
- 输出(output)
- loader
- 插件(plugin)
- 模式(mode)
- 浏览器兼容性(browser compatibility)
- 环境(environment)

本文档旨在给出这些概念的__高度__概述，同时提供具体概念的详尽相关用例的链接。

为了更好地理解模块打包工具背后的理念，以及在底层它们是如何运作的，请参考以下资源：

- 手动打包一个应用程序
- 实时创建一个简单打包工具
- 一个简单打包工具的详细说明

入口(entry)

__入口起点(entry point)__指示 webpack 应该使用哪个模块，来作为构建其内部 依赖图(dependency graph) 的开始。进入入口起点后，webpack 会找出有哪些模块和库是入口起点（直接和间接）依赖的。

默认值是 `./src/index.js`，但你可以通过在 webpack configuration 中配置 `entry` 属性，来指定一个（或多个）不同的入口起点。例如：

webpack.config.js

```
module.exports = {
  entry: './path/to/my/entry/file.js'
};
```

> **Tip**
>
> 在 入口起点 章节可以了解更多信息。

输出(output)

output 属性告诉 webpack 在哪里输出它所创建的 *bundle*，以及如何命名这些文件。主要输出文件的默认值是 `./dist/main.js`，其他生成文件默认放置在 `./dist` 文件夹中。

你可以通过在配置中指定一个 `output` 字段，来配置这些处理过程：

webpack.config.js

```
const path = require('path');

module.exports = {
  entry: './path/to/my/entry/file.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'my-first-webpack.bundle.js'
  }
};
```

在上面的示例中，我们通过 `output.filename` 和 `output.path` 属性，来告诉 webpack bundle 的名称，以及我们想要 bundle 生成(emit)到哪里。可能你想要了解在代码最上面导入的 path 模块是什么，它是一个 Node.js 核心模块，用于操作文件路径。

> **Tip**
>
> `output` 属性还有 更多可配置的特性，如果你想要了解更多关于 `output` 属性的概念，可以通过阅读 输出章节 来了解更多。

loader

webpack 只能理解 JavaScript 和 JSON 文件，这是 webpack 开箱可用的自带能力。**loader** 让 webpack 能够去处理其他类型的文件，并将它们转换为有效 模块，以供应用程序使用，以及被添加到依赖图中。

> ### Warning
>
> 注意，loader 能够 `import` 导入任何类型的模块（例如 `.css` 文件），这是 webpack 特有的功能，其他打包程序或任务执行器的可能并不支持。我们认为这种语言扩展是很有必要的，因为这可以使开发人员创建出更准确的依赖关系图。

在更高层面，在 webpack 的配置中，**loader** 有两个属性：

1. `test` 属性，识别出哪些文件会被转换。
2. `use` 属性，定义出在进行转换时，应该使用哪个 loader。

webpack.config.js

```
const path = require('path');

module.exports = {
  output: {
    filename: 'my-first-webpack.bundle.js'
  },
  module: {
    rules: [
      { test: /\.txt$/, use: 'raw-loader' }
    ]
  }
};
```

以上配置中，对一个单独的 module 对象定义了 `rules` 属性，里面包含两个必须属性： `test` 和 `use` 。这告诉 webpack 编译器(compiler) 如下信息：

> “嘿，*webpack* 编译器，当你碰到「在 *`require()` / `import`* 语句中被解析为 '*.txt*' 的路径」时，在你对它打包之前，先 ***use(使用)*** *`raw-loader`* 转换一下。”

> ### Warning
>
> 重要的是要记住，在 webpack 配置中定义 rules 时，要定义在 `module.rules` 而不是 `rules` 中。为了使你便于理解，如果没有按照正确方式去做，webpack 会给出警告。

>

> **Warning**
>
> 请记住，使用正则表达式匹配文件时，你不要为它添加引号。也就是说， `/\.txt$/` 与 `'/\.txt$/'` 或 `"/\.txt$/"` 不一样。前者指示 webpack 匹配任何以 .txt 结尾的文件，后者指示 webpack 匹配具有绝对路径 '.txt' 的单个文件; 这可能不符合你的意图。

在使用 loader 时，可以阅读 loader 章节 查看更深入的自定义配置。

插件(plugin)

loader 用于转换某些类型的模块，而插件则可以用于执行范围更广的任务。包括：打包优化，资源管理，注入环境变量。

> **Tip**
>
> 查看 插件接口(plugin interface)，学习如何使用它来扩展 webpack 能力。

想要使用一个插件，你只需要 `require()` 它，然后把它添加到 `plugins` 数组中。多数插件可以通过选项(option)自定义。你也可以在一个配置文件中因为不同目的而多次使用同一个插件，这时需要通过使用 `new` 操作符来创建一个插件实例。

webpack.config.js

```
const HtmlWebpackPlugin = require('html-webpack-plugin'); // 通过 npm 安装
const webpack = require('webpack'); // 用于访问内置插件

module.exports = {
  module: {
    rules: [
      { test: /\.txt$/, use: 'raw-loader' }
    ]
  },
  plugins: [
    new HtmlWebpackPlugin({template: './src/index.html'})
  ]
};
```

在上面的示例中， `html-webpack-plugin` 为应用程序生成一个 HTML 文件，并自动注入所有生成的 bundle。

> **Tip**
>
> webpack 提供许多开箱可用的插件！查阅 插件列表 获取更多。

在 webpack 配置中使用插件是简单直接的。然而，也有很多值得我们进一步探讨的用例。查看这里了解更多。

模式(mode)

通过选择 `development`，`production` 或 `none` 之中的一个，来设置 `mode` 参数，你可以启用 webpack 内置在相应环境下的优化。其默认值为 `production`。

```
module.exports = {
  mode: 'production'
};
```

想要了解更多，请查阅 mode 配置，这里有具体每个值相应的优化行为。

浏览器兼容性(browser compatibility)

webpack 支持所有符合 ES5 标准 的浏览器（不支持 IE8 及以下版本）。webpack 的 `import()` 和 `require.ensure()` 需要 `Promise`。如果你想要支持旧版本浏览器，在使用这些表达式之前，还需要 提前加载 polyfill。

环境(environment)

webpack 5 运行于 Node.js v10.13.0+ 的版本。

入口起点(entry points)

正如我们在 起步 中提到的，在 webpack 配置中有多种方式定义 `entry` 属性。除了解释为什么它可能非常有用，我们还将向你展示__如何去__配置 `entry` 属性。

单个入口（简写）语法

用法：`entry: string | [string]`

webpack.config.js

```
module.exports = {
  entry: './path/to/my/entry/file.js'
};
```

`entry` 属性的单个入口语法，参考下面的简写：

webpack.config.js

```
module.exports = {
  entry: {
    main: './path/to/my/entry/file.js'
  }
};
```

我们也可以将一个文件路径数组传递给 `entry` 属性，这将创建一个所谓的 **"multi-main entry"**。在你想要一次注入多个依赖文件，并且将它们的依赖关系绘制在一个 "chunk" 中时，这种方式就很有用。

webpack.config.js

```
module.exports = {
  entry: [
    './src/file_1.js',
    './src/file_2.js'
  ],
  output: {
    filename: 'bundle.js'
  }
};
```

当你希望通过一个入口（例如一个库）为应用程序或工具快速设置 webpack 配置时，单一入口的语法方式是不错的选择。然而，使用这种语法方式来扩展或调整配置的灵活性不大。

对象语法

用法： `entry: { <entryChunkName> string | [string] } | {}`

webpack.config.js

```
module.exports = {
  entry: {
    app: './src/app.js',
    adminApp: './src/adminApp.js'
  }
};
```

对象语法会比较繁琐。然而，这是应用程序中定义入口的最可扩展的方式。

> **Tip**
>
> __“webpack 配置的可扩展”__是指，这些配置可以重复使用，并且可以与其他配置组合使用。这是一种流行的技术，用于将关注点从环境(environment)、构建目标(build target)、运行时(runtime)中分离。然后使用专门的工具（如 webpack-merge）将它们合并起来。

> **Tip**
>
> 当你通过插件生成入口时，你可以传递空对象 `{}` 给 `entry` 。

描述入口的对象

用于描述入口的对象。你可以使用如下属性：

- `dependOn` ：当前入口所依赖的入口。它们必须在该入口被加载前被加载。
- `filename` ：指定要输出的文件名称。
- `import` ：启动时需加载的模块。
- `library` ：library 的相关选项。
- `runtime` ：运行时 chunk 的名字。如果设置了，就会创建一个以这个名字命名的运行时 chunk，否则将使用现有的入口作为运行时。

webpack.config.js

```
module.exports = {
  entry: {
    a2: 'dependingfile.js',
    b2: {
      dependOn: 'a2',
      import: './src/app.js'
    }
  }
};
```

`runtime` 和 `dependOn` 不应在同一个入口上同时使用，所以如下配置无效，并且会抛出错误：

webpack.config.js

```
module.exports = {
  entry: {
    a2: './a',
    b2: {
      runtime: 'x2',
      dependOn: 'a2',
```

```
      import: './b'
    }
  }
};
```

确保 `runtime` 不能指向已存在的入口名称，例如下面配置会抛出一个错误：

```
module.exports = {
  entry: {
    a1: './a',
    b1: {
      runtime: 'a1',
      import: './b'
    }
  }
};
```

另外 `dependOn` 不能是循环引用的，下面的例子也会出现错误：

```
module.exports = {
  entry: {
    a3: {
      import: './a',
      dependOn: 'b3'
    },
    b3: {
      import: './b',
      dependOn: 'a3'
    }
  }
};
```

常见场景

以下列出一些入口配置和它们的实际用例：

分离 **app(**应用程序**)** 和 **vendor(**第三方库**)** 入口

webpack.config.js

```
module.exports = {
  entry: {
    main: './src/app.js',
    vendor: './src/vendor.js'
```

```
  }
};
```

webpack.prod.js

```
module.exports = {
  output: {
    filename: '[name].[contenthash].bundle.js'
  }
};
```

webpack.dev.js

```
module.exports = {
  output: {
    filename: '[name].bundle.js'
  }
};
```

__这是什么？__这是告诉 webpack 我们想要配置 2 个单独的入口点（例如上面的示例）。

__为什么？__这样你就可以在 `vendor.js` 中存入未做修改的必要 library 或文件（例如 Bootstrap, jQuery, 图片等），然后将它们打包在一起成为单独的 chunk。内容哈希保持不变，这使浏览器可以独立地缓存它们，从而减少了加载时间。

> **Tip**
>
> 在 webpack < 4 的版本中，通常将 vendor 作为一个单独的入口起点添加到 entry 选项中，以将其编译为一个单独的文件（与 `CommonsChunkPlugin` 结合使用）。
>
> 而在 webpack 4 中不鼓励这样做。而是使用 `optimization.splitChunks` 选项，将 vendor 和 app(应用程序) 模块分开，并为其创建一个单独的文件。**不要** 为 vendor 或其他不是执行起点创建 entry。

多页面应用程序

webpack.config.js

```
module.exports = {
  entry: {
    pageOne: './src/pageOne/index.js',
    pageTwo: './src/pageTwo/index.js',
    pageThree: './src/pageThree/index.js'
  }
};
```

_这是什么？_我们告诉 webpack 需要三个独立分离的依赖图（如上面的示例）。

_为什么？_在多页面应用程序中，server 会拉取一个新的 HTML 文档给你的客户端。页面重新加载此新文档，并且资源被重新下载。然而，这给了我们特殊的机会去做很多事，例如使用 `optimization.splitChunks` 为页面间共享的应用程序代码创建 bundle。由于入口起点数量的增多，多页应用能够复用多个入口起点之间的大量代码/模块，从而可以极大地从这些技术中受益。

> **Tip**
>
> 根据经验：每个 HTML 文档只使用一个入口起点。具体原因请参阅此 issue。

输出(output)

可以通过配置 `output` 选项，告知 webpack 如何向硬盘写入编译文件。注意，即使可以存在多个 `entry` 起点，但只能指定一个 `output` 配置。

用法

在 webpack 配置中， `output` 属性的最低要求是，将它的值设置为一个对象，然后为将输出文件的文件名配置为一个 `output.filename` ：

webpack.config.js

```
module.exports = {
  output: {
    filename: 'bundle.js',
  }
};
```

此配置将一个单独的 `bundle.js` 文件输出到 `dist` 目录中。

多个入口起点

如果配置中创建出多于一个 "chunk"（例如，使用多个入口起点或使用像 CommonsChunkPlugin 这样的插件），则应该使用 占位符(substitutions) 来确保每个文件具有唯一的名称。

```
module.exports = {
  entry: {
    app: './src/app.js',
    search: './src/search.js'
  },
```

```
  output: {
    filename: '[name].js',
    path: __dirname + '/dist'
  }
};

// 写入到硬盘：./dist/app.js, ./dist/search.js
```

高级进阶

以下是对资源使用 CDN 和 hash 的复杂示例：

config.js

```
module.exports = {
  //...
  output: {
    path: '/home/proj/cdn/assets/[fullhash]',
    publicPath: 'https://cdn.example.com/assets/[fullhash]/'
  }
};
```

如果在编译时，不知道最终输出文件的 `publicPath` 是什么地址，则可以将其留空，并且在运行时通过入口起点文件中的 `__webpack_public_path__` 动态设置。

```
__webpack_public_path__ = myRuntimePublicPath;

// 应用程序入口的其余部分
```

loader

loader 用于对模块的源代码进行转换。loader 可以使你在 `import` 或 "load(加载)" 模块时预处理文件。因此，loader 类似于其他构建工具中“任务(task)”，并提供了处理前端构建步骤的得力方式。loader 可以将文件从不同的语言（如 TypeScript）转换为 JavaScript 或将内联图像转换为 data URL。loader 甚至允许你直接在 JavaScript 模块中 `import` CSS文件！

示例

例如，你可以使用 loader 告诉 webpack 加载 CSS 文件，或者将 TypeScript 转为 JavaScript。为此，首先安装相对应的 loader：

```
npm install --save-dev css-loader ts-loader
```

然后指示 webpack 对每个 `.css` 使用 `css-loader`，以及对所有 `.ts` 文件使用 `ts-loader`：

webpack.config.js

```
module.exports = {
  module: {
    rules: [
      { test: /\.css$/, use: 'css-loader' },
      { test: /\.ts$/, use: 'ts-loader' }
    ]
  }
};
```

使用 loader

在你的应用程序中，有三种使用 loader 的方式：

- 配置方式（推荐）：在 **webpack.config.js** 文件中指定 loader。
- 内联方式：在每个 `import` 语句中显式指定 loader。
- CLI 方式：在 shell 命令中指定它们。

配置方式

`module.rules` 允许你在 webpack 配置中指定多个 loader。 这种方式是展示 loader 的一种简明方式，并且有助于使代码变得简洁和易于维护。同时让你对各个 loader 有个全局概览：

loader 从右到左（或从下到上）地取值(evaluate)/执行(execute)。在下面的示例中，从 sass-loader 开始执行，然后继续执行 css-loader，最后以 style-loader 为结束。查看 loader 功能 章节，了解有关 loader 顺序的更多信息。

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.css$/,
        use: [
          // [style-loader](/loaders/style-loader)
          { loader: 'style-loader' },
          // [css-loader](/loaders/css-loader)
          {
            loader: 'css-loader',
            options: {
              modules: true
```

```
            }
          },
          // [sass-loader](/loaders/sass-loader)
          { loader: 'sass-loader' }
        ]
      }
    ]
  }
};
```

内联方式

可以在 `import` 语句或任何 与 "import" 方法同等的引用方式 中指定 loader。使用 `!` 将资源中的 loader 分开。每个部分都会相对于当前目录解析。

```
import Styles from 'style-loader!css-loader?modules!./styles.css';
```

通过为内联 `import` 语句添加前缀，可以覆盖 配置 中的所有 loader, preLoader 和 postLoader：

- 使用 `!` 前缀，将禁用所有已配置的 normal loader(普通 loader)

```
import Styles from '!style-loader!css-loader?modules!./styles.css';
```

- 使用 `!!` 前缀，将禁用所有已配置的 loader（preLoader, loader, postLoader）

```
import Styles from '!!style-loader!css-loader?modules!./styles.css';
```

- 使用 `-!` 前缀，将禁用所有已配置的 preLoader 和 loader，但是不禁用 postLoaders

```
import Styles from '-!style-loader!css-loader?modules!./styles.css';
```

选项可以传递查询参数，例如 `?key=value&foo=bar` ，或者一个 JSON 对象，例如 `?{"key":"value","foo":"bar"}` 。

> **Tip**
>
> 尽可能使用 `module.rules` ，因为这样可以减少源码中样板文件的代码量，并且可以在出错时，更快地调试和定位 loader 中的问题。

CLI 方式

还可以通过 CLI 使用 loader：

```
webpack --module-bind pug-loader --module-bind 'css=style-loader!css-loader'
```

这会对 `.jade` 文件使用 `pug-loader`，以及对 `.css` 文件使用 `style-loader` 和 `css-loader`。

loader 特性

- loader 支持链式调用。链中的每个 loader 会将转换应用在已处理过的资源上。一组链式的 loader 将按照相反的顺序执行。链中的第一个 loader 将其结果（也就是应用过转换后的资源）传递给下一个 loader，依此类推。最后，链中的最后一个 loader，返回 webpack 所期望的 JavaScript。
- loader 可以是同步的，也可以是异步的。
- loader 运行在 Node.js 中，并且能够执行任何操作。
- loader 可以通过 `options` 对象配置（仍然支持使用 `query` 参数来设置选项，但是这种方式已被废弃）。
- 除了常见的通过 `package.json` 的 `main` 来将一个 npm 模块导出为 loader，还可以在 module.rules 中使用 `loader` 字段直接引用一个模块。
- 插件(plugin)可以为 loader 带来更多特性。
- loader 能够产生额外的任意文件。

可以通过 loader 的预处理函数，为 JavaScript 生态系统提供更多能力。用户现在可以更加灵活地引入细粒度逻辑，例如：压缩、打包、语言翻译和 更多其他特性。

解析 loader

loader 遵循标准 模块解析 规则。多数情况下，loader 将从 模块路径 加载（通常是从 `npm install`, `node_modules` 进行加载）。

我们预期 loader 模块导出为一个函数，并且编写为 Node.js 兼容的 JavaScript。通常使用 npm 进行管理 loader，但是也可以将应用程序中的文件作为自定义 loader。按照约定，loader 通常被命名为 `xxx-loader` （例如 `json-loader` ）。更多详细信息，请查看 编写一个 loader。

plugin

__插件__是 webpack 的 支柱 功能。webpack 自身也是构建于你在 webpack 配置中用到的__相同的插件系统__之上！

插件目的在于解决 loader 无法实现的__其他事__。

> **Tip**

> 如果在插件中使用了 `webpack-sources` 的 package，请使用 `require('webpack').sources` 替代 `require('webpack-sources')`，以避免持久缓存的版本冲突。

剖析

webpack _插件_是一个具有 `apply` 方法的 JavaScript 对象。`apply` 方法会被 webpack compiler 调用，并且在_整个_编译生命周期都可以访问 compiler 对象。

ConsoleLogOnBuildWebpackPlugin.js

```
const pluginName = 'ConsoleLogOnBuildWebpackPlugin';

class ConsoleLogOnBuildWebpackPlugin {
  apply(compiler) {
    compiler.hooks.run.tap(pluginName, compilation => {
      console.log('webpack 构建过程开始! ');
    });
  }
}

module.exports = ConsoleLogOnBuildWebpackPlugin;
```

compiler hook 的 tap 方法的第一个参数，应该是驼峰式命名的插件名称。建议为此使用一个常量，以便它可以在所有 hook 中重复使用。

用法

由于_插件_可以携带参数/选项，你必须在 webpack 配置中，向 `plugins` 属性传入一个 `new` 实例。

取决于你的 webpack 用法，对应有多种使用插件的方式。

配置方式

webpack.config.js

```
const HtmlWebpackPlugin = require('html-webpack-plugin'); // 通过 npm 安装
const webpack = require('webpack'); // 访问内置的插件
const path = require('path');

module.exports = {
  entry: './path/to/my/entry/file.js',
```

```
  output: {
    filename: 'my-first-webpack.bundle.js',
    path: path.resolve(__dirname, 'dist')
  },
  module: {
    rules: [
      {
        test: /\.(js|jsx)$/,
        use: 'babel-loader'
      }
    ]
  },
  plugins: [
    new webpack.ProgressPlugin(),
    new HtmlWebpackPlugin({template: './src/index.html'})
  ]
};
```

`ProgressPlugin` 用于自定义编译过程中的进度报告， `HtmlWebpackPlugin` 将生成一个 HTML 文件，并在其中使用 `script` 引入一个名为 `my-first-webpack.bundle.js` 的 JS 文件。

Node API 方式

在使用 Node API 时，还可以通过配置中的 `plugins` 属性传入插件。

some-node-script.js

```
const webpack = require('webpack'); // 访问 webpack 运行时(runtime)
const configuration = require('./webpack.config.js');

let compiler = webpack(configuration);

new webpack.ProgressPlugin().apply(compiler);

compiler.run(function(err, stats) {
  // ...
});
```

> **Tip**
>
> 你知道吗：以上看到的示例和 webpack 运行时(runtime)本身 极其类似。webpack 源码 中隐藏有大量使用示例，你可以将其应用在自己的配置和脚本中。

配置（Configuration）

你可能已经注意到，很少有 webpack 配置看起来完全相同。这是因为 **webpack 的配置文件是 JavaScript 文件，文件内导出了一个 webpack 配置的对象。** webpack 会根据该配置定义的属性进行处理。

由于 webpack 遵循 CommonJS 模块规范，因此，你__可以在配置中使用__：

- 通过 `require(...)` 引入其他文件
- 通过 `require(...)` 使用 npm 下载的工具函数
- 使用 JavaScript 控制流表达式，例如 `?:` 操作符
- 对 value 使用常量或变量赋值
- 编写并执行函数，生成部分配置

请在合适的场景，使用这些功能。

虽然技术上可行，**但还是应避免如下操作**：

- 当使用 webpack CLI 工具时，访问 CLI 参数（应编写自己的 CLI 工具替代，或者使用 `--env`）
- 导出不确定的结果（两次调用 webpack 应产生相同的输出文件）
- 编写超长的配置（应将配置文件拆分成多个）

> **Tip**
>
> 此文档中得出最重要的结论是，webpack 的配置可以有许多不同的样式和风格。关键在于，为了易于维护和理解这些配置，需要在团队内部保证一致。

接下来的示例中，展示了 webpack 配置如何实现既可表达，又可灵活配置，这主要得益于_配置即为代码_：

基本配置

webpack.config.js

```
var path = require('path');

module.exports = {
  mode: 'development',
  entry: './foo.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'foo.bundle.js'
```

```
  }
};
```

查看：配置章节中所有支持的配置选项。

多个 target

除了可以将单个配置导出为 object，function 或 Promise 以外，还可以将其导出为多个配置。

查看：导出多个配置

使用其它配置语言

webpack 支持由多种编程和数据语言编写的配置文件。

查看：配置语言

模块（Modules）

在模块化编程中，开发者将程序分解为功能离散的 chunk，并称之为 **模块**。

每个模块都拥有小于完整程序的体积，使得验证、调试及测试变得轻而易举。 精心编写的 _模块_提供了可靠的抽象和封装界限，使得应用程序中每个模块都具备了条理清晰的设计和明确的目的。

Node.js 从一开始就支持模块化编程。 然而，web 的_模块化_正在缓慢支持中。 在 web 界存在多种支持 JavaScript 模块化的工具，这些工具各有优势和限制。 webpack 从这些系统中汲取了经验和教训，并将_模块_的概念应用到项目的任何文件中。

何为 webpack 模块

与 Node.js 模块相比，webpack _模块_能以各种方式表达它们的依赖关系。下面是一些示例：

- ES2015 `import` 语句
- CommonJS `require()` 语句
- AMD `define` 和 `require` 语句
- css/sass/less 文件中的 `@import` 语句。
- stylesheet `url(...)` 或者 HTML `<img src=...>` 文件中的图片链接。

支持的模块类型

webpack 天生支持如下模块类型：

- ECMAScript 模块
- CommonJS 模块
- AMD 模块
- Assets
- WebAssembly 模块

通过 **loader** 可以使 webpack 支持多种语言和预处理器语法编写的模块。**loader** 向 webpack 描述了如何处理非原生模块，并将相关**依赖**引入到你的 **bundles**中。 webpack 社区已经为各种流行的语言和预处理器创建了 *loader*，其中包括：

- CoffeeScript
- TypeScript
- ESNext (Babel)
- Sass
- Less
- Stylus
- Elm

当然还有更多！总得来说，webpack 提供了可定制，强大且丰富的 API，允许在 **任何技术栈** 中使用，同时支持在开发、测试和生产环境的工作流中做到 **无侵入性**。

关于 loader 的相关信息，请参考 **loader 列表** 或 **自定义** loader。

模块解析（Module Resolution）

resolver 是一个帮助寻找模块绝对路径的库。 一个模块可以作为另一个模块的依赖模块，然后被后者引用，如下：

```
import foo from 'path/to/module';
// 或者
require('path/to/module');
```

所依赖的模块可以是来自应用程序的代码或第三方库。 resolver 帮助 webpack 从每个 `require / import` 语句中，找到需要引入到 bundle 中的模块代码。 当打包模块时，webpack 使用 enhanced-resolve 来解析文件路径。

webpack 中的解析规则

使用 `enhanced-resolve`，webpack 能解析三种文件路径：

绝对路径

```
import '/home/me/file';

import 'C:\\Users\\me\\file';
```

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

```
import '../src/file1';
import './file2';
```

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

```
import 'module';
import 'module/lib/file';
```

在 `resolve.modules` 中指定的所有目录检索模块。 你可以通过配置别名的方式来替换初始模块路径，具体请参照 `resolve.alias` 配置选项。

- 如果 package 中包含 `package.json` 文件，那么在 `resolve.exportsFields` 配置选项中指定的字段会被依次查找， `package.json` 中的第一个字段会根据 package 导出指南确定 package 中可用的 export。

一旦根据上述规则解析路径后，resolver 将会检查路径是指向文件还是文件夹。如果路径指向文件：

- 如果文件具有扩展名，则直接将文件打包。
- 否则，将使用 `resolve.extensions` 选项作为文件扩展名来解析，此选项会告诉解析器在解析中能够接受那些扩展名（例如 `.js`， `.jsx`）。

如果路径指向一个文件夹，则进行如下步骤寻找具有正确扩展名的文件：

- 如果文件夹中包含 `package.json` 文件，则会根据 `resolve.mainFields` 配置中的字段顺序查找，并根据 `package.json` 中的符合配置要求的第一个字段来确定文件路径。
- 如果不存在 `package.json` 文件或 `resolve.mainFields` 没有返回有效路径，则会根据 `resolve.mainFiles` 配置选项中指定的文件名顺序查找，看是否能在 import/require 的目录下匹配到一个存在的文件名。
- 然后使用 `resolve.extensions` 选项，以类似的方式解析文件扩展名。

webpack 会根据构建目标，为这些选项提供合理的默认配置。

解析 loader

loader 的解析规则也遵循特定的规范。但是 `resolveLoader` 配置项可以为 loader 设置独立的解析规则。

缓存

每次文件系统访问文件都会被缓存，以便于更快触发对同一文件的多个并行或串行请求。在 watch 模式 下，只有修改过的文件会被从缓存中移出。如果关闭 watch 模式，则会在每次编译前清理缓存。

欲了解更多上述配置信息，请查阅 Resolve API。

Module Federation

动机

多个独立的构建可以组成一个应用程序，这些独立的构建之间不应该存在依赖关系，因此可以单独开发和部署它们。

这通常被称作微前端，但并不仅限于此。

底层概念

我们区分本地模块和远程模块。本地模块即为普通模块，是当前构建的一部分。远程模块不属于当前构建，并在运行时从所谓的容器加载。

加载远程模块被认为是异步操作。当使用远程模块时，这些异步操作将被放置在远程模块和入口之间的下一个 chunk 的加载操作中。如果没有 chunk 加载操作，就不能使用远程模块。

chunk 的加载操作通常是通过调用 `import()` 实现的，但也支持像 `require.ensure` 或 `require([...])` 之类的旧语法。

容器是由容器入口创建的，该入口暴露了对特定模块的异步访问。暴露的访问分为两个步骤：

1. 加载模块（异步的）
2. 执行模块（同步的）

步骤 1 将在 chunk 加载期间完成。步骤 2 将在与其他（本地和远程）的模块交错执行期间完成。这样一来，执行顺序不受模块从本地转换为远程或从远程转为本地的影响。

容器可以嵌套使用，容器可以使用来自其他容器的模块。容器之间也可以循环依赖。

重载（Overriding）

容器能够将选定的本地模块标记为“可重载”。容器的使用者能够提供“重载”，即替换容器中的一个“可重载”的模块。当使用者提供重载模块时，容器的所有模块将使用替换模块而非本地模块。当使用者不提供替换模块时，容器的所有模块将使用本地模块。

容器管理可重载模块的方式为：当使用者已经重写它们后，就不需要下载了。这通常是通过将它们放在单独的 chunk 中来实现的。

另一方面，替换模块的提供者，将只提供异步加载函数。它允许容器仅在需要替换模块时才去加载。提供者管理替换模块的方式为：当容器不请求替换模块时，则无需下载。这通常是通过将它们放在单独的 chunk 中来实现的。

"name" 用于标识容器中可重载的模块。

重载（Overrides）的提供和容器暴露模块类似，它分为两个步骤:

1. 加载（异步）
2. 执行（同步）

> **Warning**
>
> 当嵌套使用时，向容器提供重载将自动覆盖嵌套容器中具有相同 "name" 的模块。

必须在容器模块加载之前提供重载。在初始 chunk 中使用的重载只能被不使用 Promise 的同步模块重载。一旦执行，就不可再次被重载。

高级概念

每个构建都充当一个容器，也可将其他构建作为容器。通过这种方式，每个构建都能够通过从对应容器中加载模块来访问其他容器暴露出来的模块。

共享模块是指既可重写的又可作为向嵌套容器提供重写的模块。它们通常指向每个构建中的相同模块，例如相同的库。

packageName 选项允许通过设置包名来查找所需的版本。默认情况下，它会自动推断模块请求，当想禁用自动推断时，请将 requiredVersion 设置为 false 。

构建块(Building blocks)

`OverridablesPlugin` (底层 API)

这个插件使得特定模块"可重载"。一个本地 API (`__webpack_override__`) 允许提供重载。

webpack.config.js

```
const OverridablesPlugin = require('webpack/lib/container/OverridablesPlugin');
module.exports = {
  plugins: [
    new OverridablesPlugin([
      {
        // 通过 OverridablesPlugin 定义一个可重载的模块
        test1: './src/test1.js',
      },
    ]),
  ],
};
```

src/index.js

```
__webpack_override__({
  // 这里我们重写 test1 模块
  test1: () => 'I will override test1 module under src',
});
```

`ContainerPlugin` (底层 API)

该插件使用指定的公开模块来创建一个额外的容器入口。它还会在内部使用 OverridablesPlugin，并向容器的使用者暴露 `override` API。

`ContainerReferencePlugin`（底层 API)

该插件将特定的引用添加到作为外部资源（externals）的容器中，并允许从这些容器中导入远程模块。它还会调用这些容器的 `override` API 来为它们提供重载。本地的重载（当构建也是一个容器时，通过 `__webpack_override__` 或 `override` API）和指定的重载被提供给所有引用的容器。

`ModuleFederationPlugin`（高级 API）

该插件组合了 `ContainerPlugin` 和 `ContainerReferencePlugin` 。重载（overrides）和可重载（overridables）被合并到指定共享模块的单个列表中。

概念目标

- 它既可以暴露，又可以使用 webpack 支持的任何模块类型
- 代码块加载应该并行加载所需的所有内容(web:到服务器的单次往返)
- 从使用者到容器的控制
  - 重写模块是一种单向操作
  - 同级容器不能重写彼此的模块。
- 概念适用于独立于环境
  - 可用于 web、Node.js 等
- 共享中的相对和绝对请求
  - 会一直提供，即使不使用
  - 会将相对路径解析到 `config.context`
  - 默认不会使用 `requiredVersion`
- 共享中的模块请求
  - 只在使用时提供
  - 会匹配构建中所有使用的相等模块请求
  - 将提供所有匹配模块
  - 将从图中这个位置的 package.json 提取 `requiredVersion`
  - 当你有嵌套的 node_modules 时，可以提供和使用多个不同的版本
- 共享中尾部带有 `/` 的模块请求将匹配所有具有这个前缀的模块请求

用例

每个页面单独构建

单页应用的每个页面都是在单独的构建中从容器暴露出来的。主体应用程序(application shell)也是独立构建，会将所有页面作为远程模块来引用。通过这种方式，可以单独部署每个页面。在更新路由或添加新路由时部署主体应用程序。主体应用程序将常用库定义为共享模块，以避免在页面构建中出现重复。

将组件库作为容器

许多应用程序共享一个通用的组件库，可以将其构建成暴露所有组件的容器。每个应用程序使用来自组件库容器的组件。可以单独部署对组件库的更改，而不需要重新部署所有应用程序。应用程序自动使用组件库的最新版本。

动态远程容器

该容器接口支持 `get` 和 `init` 方法。 `init` 是一个兼容 `async` 的方法，调用时，只含有一个参数：共享作用域对象(shared scope object)。此对象在远程容器中用作共享作用域，并由 host 提供的模块填充。 可以利用它在运行时动态地将远程容器连接到 host 容器。

init.js

```
(async () => {
  // 初始化共享作用域（shared scope）用提供的已知此构建和所有远程的模块填充它
  await __webpack_init_sharing__('default');
  const container = window.someContainer; // 或从其他地方获取容器
  // 初始化容器 它可能提供共享模块
  await container.init(__webpack_share_scopes__.default);
  const module = await container.get('./module');
})();
```

容器尝试提供共享模块，但是如果共享模块已经被使用，则会发出警告，并忽略所提供的共享模块。容器仍能将其作为降级模块。

你可以通过动态加载的方式，提供一个共享模块的不同版本，从而实现 A/B 测试。

> **Tip**
>
> 在尝试动态连接远程容器之前，确保已加载容器。

例子：

init.js

```
function loadComponent(scope, module) {
  return async () => {
    // 初始化共享作用域（shared scope）用提供的已知此构建和所有远程的模块填充它
    await __webpack_init_sharing__('default');
    const container = window[scope]; // 或从其他地方获取容器
    // 初始化容器 它可能提供共享模块
    await container.init(__webpack_share_scopes__.default);
    const factory = await window[scope].get(module);
    const Module = factory();
    return Module;
  };
}

loadComponent('abtests', 'test123');
```

查看完整实现

故障排除

Uncaught Error: Shared module is not available for eager consumption

应用程序正急切地执行一个作为全局主机运行的应用程序。有如下选项可供选择:

你可以在模块联邦的高级 API 中将依赖设置为即时依赖，此 API 不会将模块放在异步 chunk 中，而是同步地提供它们。这使得我们在初始块中可以直接使用这些共享模块。但是要注意，由于所有提供的和降级模块是要异步下载的，因此，建议只在应用程序的某个地方提供它，例如 shell。

我们强烈建议使用异步边界(asynchronous boundary)。它将把初始化代码分割成更大的块，以避免任何额外的开销，以提高总体性能。

例如，你的入口看起来是这样的：

index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';
ReactDOM.render(<App />, document.getElementById('root'));
```

让我们创建 bootstrap.js 文件，并将入口文件的内容放到里面，然后将 bootstrap 引入到入口文件中:

index.js

```
+ import('./bootstrap');
- import React from 'react';
```

```
- import ReactDOM from 'react-dom';
- import App from './App';
- ReactDOM.render(<App />, document.getElementById('root'));
```

bootstrap.js

```
+ import React from 'react';
+ import ReactDOM from 'react-dom';
+ import App from './App';
+ ReactDOM.render(<App />, document.getElementById('root'));
```

这种方法有效，但存在局限性或缺点。

通过 `ModuleFederationPlugin` 将依赖的 `eager` 属性设置为 `true`

webpack.config.js

```
// ...
new ModuleFederationPlugin({
  shared: {
    ...deps,
    react: {
      eager: true,
    }
  }
});
```

```
Uncaught Error: Module "./Button" does not exist in container.
```

错误提示中可能不会显示 `"./Button"` ，但是信息看起来差不多。这个问题通常会出现在将 webpack beta.16 升级到 webpack beta.17 中。

在 ModuleFederationPlugin 里，更改 exposes:

```
new ModuleFederationPlugin({
  exposes: {
-   'Button': './src/Button'
+   './Button':'./src/Button'
  }
});
```

```
Uncaught TypeError: fn is not a function
```

此处错误可能是丢失了远程容器，请确保在使用前添加它。 如果已为试图使用远程服务器的容器加载了容器，但仍然看到此错误，则需将主机容器的远程容器文件也添加到 HTML 中。

依赖图(dependency graph)

每当一个文件依赖另一个文件时，webpack 都会将文件视为直接存在 依赖关系。这使得 webpack 可以获取非代码资源，如 images 或 web 字体等。并会把它们作为 依赖 提供给应用程序。

当 webpack 处理应用程序时，它会根据命令行参数中或配置文件中定义的模块列表开始处理。 从入口 开始，webpack 会递归的构建一个_依赖关系图_，这个依赖图包含着应用程序中所需的每个模块，然后将所有模块打包为少量的 *bundle* —— 通常只有一个 —— 可由浏览器加载。

> **Tip**
>
> 对于 *HTTP/1.1* 的应用程序来说，由 webpack 构建的 bundle 非常强大。当浏览器发起请求时，它能最大程度的减少应用的等待时间。而对于 *HTTP/2* 来说，你还可以使用代码分割进行进一步优化。

target

由于 JavaScript 即可以编写服务端代码也可以编写浏览器代码，所以 webpack 提供了多种部署 *target*，你可以在 webpack 的配置选项中进行设置。

> **Warning**
>
> webpack 的 `target` 属性，不要和 `output.libraryTarget` 属性混淆。有关 `output` 属性的更多信息，请参阅 output 指南

用法

想设置 `target` 属性，只需在 webpack 配置中设置 target 字段：

webpack.config.js

```
module.exports = {
  target: 'node'
};
```

在上述示例中，target 设置为 `node` ，webpack 将在类 Node.js 环境编译代码。(使用 Node.js 的 `require` 加载 chunk，而不加载任何内置模块，如 `fs` 或 `path`)。

每个 *target* 都包含各种 deployment（部署）/environment（环境）特定的附加项，以满足其需求。具体请参阅 target 可用值。

> **Todo**
>
> 后续会进一步扩展受欢迎的 target。

多 target

虽然 webpack **不支持** 向 `target` 属性传入多个字符串，但是可以通过设置两个独立配置，来构建对 library 进行同构：

webpack.config.js

```
const path = require('path');
const serverConfig = {
  target: 'node',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'lib.node.js'
  }
  //…
};

const clientConfig = {
  target: 'web', // <=== 默认为 'web'，可省略
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'lib.js'
  }
  //…
};

module.exports = [ serverConfig, clientConfig ];
```

上述示例中，将会在 `dist` 文件夹下创建 `lib.js` 和 `lib.node.js` 文件。

资源

从上面选项可以看出，你可以选择部署不同的 *target*。下面是可以参考的示例和资源：

- compare-webpack-target-bundles：测试并查看 webpack *target* 的绝佳资源。同样包含错误上报。
- Boilerplate of Electron-React Application: 一个关于 electron 主进程和渲染进程构建过程的优秀示例。

> **Todo**
>
> 需要你查看在线代码或样本中 webpack 配置中使用的 target 示例。

manifest

在使用 webpack 构建的典型应用程序或站点中，有三种主要的代码类型：

1. 你或你的团队编写的源码。
2. 你的源码会依赖的任何第三方的 library 或 "vendor" 代码。
3. webpack 的 runtime 和 **manifest**，管理所有模块的交互。

本文将重点介绍这三个部分中的最后部分：runtime 和 manifest，特别是 manifest。

runtime

runtime，以及伴随的 manifest 数据，主要是指：在浏览器运行过程中，webpack 用来连接模块化应用程序所需的所有代码。它包含：在模块交互时，连接模块所需的加载和解析逻辑。包括：已经加载到浏览器中的连接模块逻辑，以及尚未加载模块的延迟加载逻辑。

manifest

一旦你的应用在浏览器中以 `index.html` 文件的形式被打开，一些 bundle 和应用需要的各种资源都需要用某种方式被加载与链接起来。在经过打包、压缩、为延迟加载而拆分为细小的 chunk 这些 webpack 优化 之后，你精心安排的 `/src` 目录的文件结构都已经不再存在。所以 webpack 如何管理所有所需模块之间的交互呢？这就是 manifest 数据用途的由来……

当 compiler 开始执行、解析和映射应用程序时，它会保留所有模块的详细要点。这个数据集合称为 "manifest"，当完成打包并发送到浏览器时，runtime 会通过 manifest 来解析和加载模块。无论你选择哪种 模块语法，那些 `import` 或 `require` 语句现在都已经转换为 `__webpack_require__` 方法，此方法指向模块标识符(module identifier)。通过使用 manifest 中的数据，runtime 将能够检索这些标识符，找出每个标识符背后对应的模块。

问题

所以，现在你应该对 webpack 在幕后工作有一点了解。“但是，这对我有什么影响呢？”，你可能会问。答案是大多数情况下没有。runtime 做完成这些工作：一旦你的应用程序加载到浏览器中，使用 manifest，然后所有内容将展现出魔幻般运行结果。然而，如果你决定通过使用浏览器缓存来改善项目的性能，理解这一过程将突然变得极为重要。

通过使用内容散列(content hash)作为 bundle 文件的名称，这样在文件内容修改时，会计算出新的 hash，浏览器会使用新的名称加载文件，从而使缓存无效。一旦你开始这样做，你会立即注意

到一些有趣的行为。即使某些内容明显没有修改，某些 hash 还是会改变。这是因为，注入的 runtime 和 manifest 在每次构建后都会发生变化。

查看_管理输出_指南的 manifest 部分，了解如何提取 manifest，并阅读下面的指南，以了解更多长效缓存错综复杂之处。

模块热替换(hot module replacement)

模块热替换(HMR - hot module replacement)功能会在应用程序运行过程中，替换、添加或删除模块，而无需重新加载整个页面。主要是通过以下几种方式，来显著加快开发速度：

- 保留在完全重新加载页面期间丢失的应用程序状态。
- 只更新变更内容，以节省宝贵的开发时间。
- 在源代码中 CSS/JS 产生修改时，会立刻在浏览器中进行更新，这几乎相当于在浏览器 devtools 直接更改样式。

这一切是如何运行的？

让我们从一些不同的角度观察，以了解 HMR 的工作原理……

在应用程序中

通过以下步骤，可以做到在应用程序中置换(swap in and out)模块：

1. 应用程序要求 HMR runtime 检查更新。
2. HMR runtime 异步地下载更新，然后通知应用程序。
3. 应用程序要求 HMR runtime 应用更新。
4. HMR runtime 同步地应用更新。

你可以设置 HMR，以使此进程自动触发更新，或者你可以选择要求在用户交互时进行更新。

在 compiler 中

除了普通资源，compiler 需要发出 "update"，将之前的版本更新到新的版本。"update" 由两部分组成：

1. 更新后的 manifest (JSON)
2. 一个或多个 updated chunk (JavaScript)

manifest 包括新的 compilation hash 和所有的 updated chunk 列表。每个 chunk 都包含着全部更新模块的最新代码（或一个 flag 用于表明此模块需要被移除）。

compiler 会确保在这些构建之间的模块 ID 和 chunk ID 保持一致。通常将这些 ID 存储在内存中（例如，使用 webpack-dev-server 时），但是也可能会将它们存储在一个 JSON 文件中。

在模块中

HMR 是可选功能，只会影响包含 HMR 代码的模块。举个例子，通过 `style-loader` 为 style 追加补丁。为了运行追加补丁， `style-loader` 实现了 HMR 接口；当它通过 HMR 接收到更新，它会使用新的样式替换旧的样式。

类似的，当在一个模块中实现了 HMR 接口，你可以描述出当模块被更新后发生了什么。然而在多数情况下，不需要在每个模块中强行写入 HMR 代码。如果一个模块没有 HMR 处理函数，更新就会冒泡(bubble up)。这意味着某个单独处理函数能够更新整个模块树。如果在模块树的一个单独模块被更新，那么整组依赖模块都会被重新加载。

有关 `module.hot` 接口的详细信息，请查看 HMR API 页面。

在 **runtime** 中

这件事情比较有技术性……如果你对其内部不感兴趣，可以随时跳到 HMR API 页面 或 HMR 指南。

对于模块系统运行时(module system runtime)，会发出额外代码，来跟踪模块 `parents` 和 `children` 关系。在管理方面，runtime 支持两个方法 `check` 和 `apply` 。

`check` 方法，发送一个 HTTP 请求来更新 manifest。如果请求失败，说明没有可用更新。如果请求成功，会将 updated chunk 列表与当前的 loaded chunk 列表进行比较。每个 loaded chunk 都会下载相应的 updated chunk。当所有更新 chunk 完成下载，runtime 就会切换到 `ready` 状态。

`apply` 方法，将所有 updated module 标记为无效。对于每个无效 module，都需要在模块中有一个 update handler，或者在此模块的父级模块中有 update handler。否则，会进行无效标记冒泡，并且父级也会被标记为无效。继续每个冒泡，直到到达应用程序入口起点，或者到达带有 update handler 的 module（以最先到达为准，冒泡停止）。如果它从入口起点开始冒泡，则此过程失败。

之后，所有无效 module 都会被（通过 dispose handler）处理和解除加载。然后更新当前 hash，并且调用所有 `accept` handler。runtime 切换回 `idle` 状态，一切照常继续。

起步

在开发环境，可以将 HMR 作为 LiveReload 的替代。webpack-dev-server 支持 `hot` 模式，在试图重新加载整个页面之前， `hot` 模式会尝试使用 HMR 来更新。更多细节请查看 模块热替换 指南。

> **Tip**
>
> 与许多其他功能一样，webpack 的强大之处在于它的可定制化。取决于特定项目需求，会有_许多方式_来配置 HMR。然而，对于多数项目的实现目的来说， `webpack-dev-server` 都能够很好适应，可以帮助你在项目中快速应用 HMR。

为什么选择 webpack

想要理解为什么要使用 webpack，我们先回顾下历史，在打包工具出现之前，我们是如何在 web 中使用 JavaScript 的。

在浏览器中运行 JavaScript 有两种方法。第一种方式，引用一些脚本来存放每个功能；此解决方案很难扩展，因为加载太多脚本会导致网络瓶颈。第二种方式，使用一个包含所有项目代码的大型 `.js` 文件，但是这会导致作用域、文件大小、可读性和可维护性方面的问题。

立即调用函数表达式(IIFE) - Immediately invoked function expressions

IIFE 解决大型项目的作用域问题；当脚本文件被封装在 IIFE 内部时，你可以安全地拼接或安全地组合所有文件，而不必担心作用域冲突。

IIFE 使用方式产生出 Make, Gulp, Grunt, Broccoli 或 Brunch 等工具。这些工具称为任务执行器，它们将所有项目文件拼接在一起。

但是，修改一个文件意味着必须重新构建整个文件。拼接可以做到很容易地跨文件重用脚本，但是却使构建结果的优化变得更加困难。如何判断代码是否实际被使用?

即使你只用到 lodash 中的某个函数，也必须在构建结果中加入整个库，然后将它们压缩在一起。如何 treeshake 代码依赖？难以大规模地实现延迟加载代码块，这需要开发人员手动地进行大量工作。

感谢 Node.js，JavaScript 模块诞生了

Node.js 是一个 JavaScript 运行时，可以在浏览器环境之外的计算机和服务器中使用。webpack 运行在 Node.js 中。

当 Node.js 发布时，一个新的时代开始了，它带来了新的挑战。既然不是在浏览器中运行 JavaScript，现在已经没有了可以添加到浏览器中的 html 文件和 script 标签。那么 Node.js 应用程序要如何加载新的代码 chunk 呢?

CommonJS 问世并引入了 `require` 机制，它允许你在当前文件中加载和使用某个模块。导入需要的每个模块，这一开箱即用的功能，帮助我们解决了作用域问题。

npm + Node.js + modules - 大规模分发模块

JavaScript 已经成为一种语言、一个平台和一种快速开发和创建快速应用程序的方式，接管了整个 JavaScript 世界。

但 CommonJS 没有浏览器支持。没有 live binding(实时绑定)。循环引用存在问题。同步执行的模块解析加载器速度很慢。虽然 CommonJS 是 Node.js 项目的绝佳解决方案，但浏览器不支持模块，因而产生了 Browserify, RequireJS 和 SystemJS 等打包工具，允许我们编写能够在浏览器中运行的 CommonJS 模块。

ESM - ECMAScript 模块

来自 Web 项目的好消息是，模块正在成为 ECMAScript 标准的官方功能。然而，浏览器支持不完整，版本迭代速度也不够快，目前还是推荐上面那些早期模块实现。

依赖自动收集

传统的任务构建工具基于 Google 的 Closure 编译器都要求你手动在顶部声明所有的依赖。然而像 webpack 一类的打包工具自动构建并基于你所引用或导出的内容推断出依赖的图谱。这个特性与其它的如插件 and 加载器一道让开发者的体验更好。

看起来都不是很好.....

是否可以有一种方式，不仅可以让我们编写模块，而且还支持任何模块格式（至少在我们到达 ESM 之前），并且可以同时处理资源和资产?

这就是 webpack 存在的原因。它是一个工具，可以打包你的 JavaScript 应用程序（支持 ESM 和 CommonJS），可以扩展为支持许多不同的资产，例如：images, fonts 和 stylesheets。

webpack 关心性能和加载时间；它始终在改进或添加新功能，例如：异步地加载 chunk 和预取，以便为你的项目和用户提供最佳体验。

揭示内部原理

> 此章节描述 *webpack* 内部实现，对于插件开发人员可能会提供帮助

打包，是指处理某些文件并将其输出为其他文件的能力。

但是，在输入和输出之间，还包括有 模块, 入口起点, chunk, chunk 组和许多其他中间部分。

主要部分

项目中使用的每个文件都是一个 模块

./index.js

```
import app from './app.js';
```

./app.js

```
export default 'the app';
```

通过互相引用，这些模块会形成一个图(`ModuleGraph`)数据结构。

在打包过程中，模块会被合并成 chunk。 chunk 合并成 chunk 组，并形成一个通过模块互相连接的图(`ModuleGraph`)。 那么如何通过以上来描述一个入口起点：在其内部，会创建一个只有一个 chunk 的 chunk 组。

./webpack.config.js

```
module.exports = {
  entry: './index.js'
};
```

这会创建出一个名为 `main` 的 chunk 组（ `main` 是入口起点的默认名称）。 此 chunk 组包含 `./index.js` 模块。随着 parser 处理 `./index.js` 内部的 import 时， 新模块就会被添加到此 chunk 中。

另外的一个示例：

./webpack.config.js

```
module.exports = {
  entry: {
    home: './home.js',
```

```
    about: './about.js'
  }
};
```

这会创建出两个名为 `home` 和 `about` 的 chunk 组。 每个 chunk 组都有一个包含一个模块的 chunk： `./home.js` 对应 `home` ， `./about.js` 对应 `about`

> 一个 *chunk* 组中可能有多个 *chunk*。例如，*SplitChunksPlugin* 会将一个 *chunk* 拆分为一个或多个 *chunk*。

chunk

chunk 有两种形式：

- `initial(初始化)` 是入口起点的 main chunk。此 chunk 包含为入口起点指定的所有模块及其依赖项。
- `non-initial` 是可以延迟加载的块。可能会出现在使用 动态导入(dynamic imports) 或者 SplitChunksPlugin 时。

每个 chunk 都有对应的 asset(**资源**)。资源，是指输出文件（即打包结果）。

webpack.config.js

```
module.exports = {
  entry: './src/index.jsx'
};
```

./src/index.jsx

```
import React from 'react';
import ReactDOM from 'react-dom';

import('./app.jsx').then(App => {
  ReactDOM.render(<App />, root);
});
```

这会创建出一个名为 `main` 的 initial chunk。其中包含：

- `./src/index.jsx`
- `react`
- `react-dom`

以及除 `./app.jsx` 外的所有依赖

然后会为 `./app.jsx` 创建 non-initial chunk，这是因为 `./app.jsx` 是动态导入的。

Output:

- `/dist/main.js` - 一个 `initial` chunk
- `/dist/394.js` - `non-initial` chunk

默认情况下，这些 `non-initial` chunk 没有名称，因此会使用唯一 ID 来替代名称。 在使用动态导入时，我们可以通过使用 magic comment(魔术注释) 来显式指定 chunk 名称：

```
import(
  /* webpackChunkName: "app" */
  './app.jsx'
).then(App => {
  ReactDOM.render(<App />, root);
});
```

Output:

- `/dist/main.js` - 一个 `initial` chunk
- `/dist/app.js` - `non-initial` chunk

output(输出)

输出文件的名称会受配置中的两个字段的影响：

- `output.filename` - 用于 `initial` chunk 文件
- `output.chunkFilename` - 用于 `non-initial` chunk 文件
- 在某些情况下，使用 `initial` 和 `non-initial` 的 chunk 时，可以使用 `output.filename` 。

这些字段中会有一些 占位符。常用的占位符如下：

- `[id]` - chunk id（例如 `[id].js -> 485.js` ）
- `[name]` - chunk name（例如 `[name].js -> app.js` ）。如果 chunk 没有名称，则会使用其 id 作为名称
- `[contenthash]` - 输出文件内容的 md4-hash（例如 `[contenthash].js -> 4ea6ff1de66c537eb9b2.js` ）