Webpack

1. 简介

1.1 特性

- Webpack 当中 js 可以引用 css, css 中可以嵌入图片 dataUrl

2. 知识点

- `url-loader` - 实际上是对 `file-loader` 的封装

```
loaders: [
    {test: /\.(png|jpg)$/, loader: 'url-loader?limit=8192'} // inline base64 URLs for <=8k images, direct URLs for the rest
]
```

- > Webpack 有两种组织模块依赖的方式，同步和异步。异步依赖作为分割点，形成一个新的块。在优化了**依赖树**后，每一个异步区块都作为一个文件被打包。

- 允许使用动态表达式：

```
require("./templates/" + name + ".jade")
```

- > Webpack 使用异步 I/O 和多级缓存提高运行效率，这使得 Webpack 能够以令人难以置信的速度快速增量编译。

- > Webpack 本身只能处理 JavaScript 模块，如果要处理其他类型的文件，就需要使用 loader 进行转换。
  >
  > Loader 可以理解为是模块和资源的转换器，它本身是一个函数，接受源文件作为参数，返回转换的结果。

  - Loader 可以通过管道方式链式调用，每个 loader 可以把资源转换成任意格式并传递给下一个 loader，但是最后一个 loader 必须返回 JavaScript
  - 在引用 loader 的时候可以使用全名 `json-loader`，或者使用短名 `json`，这个命名规则和搜索优先级顺序在 webpack 的 `resolveLoader.moduleTemplates` api 中定义：

    ```
    Default: ["*-webpack-loader", "*-web-loader", "*-loader", "*"]
    ```

- loader应用：

```
require("!style-loader!css-loader!./style.css") // 载入 style.css
```

```
$ webpack entry.js bundle.js --module-bind 'css=style-loader!css-loader'

# 有些环境下可能需要使用双引号
$ webpack entry.js bundle.js --module-bind "css=style-loader!css-loader"
```

- 编译结果：展示编译进度、颜色

```
webpack --progress --colors
```

- 监听模式 - 增量编译，提高编译速度

```
webpack --progress --colors --watch
```

- **webpack-dev-server** - 将在 localhost:8080 启动一个 express 静态资源 web 服务器，并且会以监听模式自动运行 webpack，在浏览器打开 http://localhost:8080/ 或 http://localhost:8080/webpack-dev-server/ 可以浏览项目中的页面和编译后的资源输出，并且通过一个 socket.io 服务实时监听它们的变化并自动刷新页面

```
# 运行
$ webpack-dev-server --progress --colors
```

- `--display-error-details` - 打印错误详情

- > Webpack 的配置提供了 `resolve` 和 `resolveLoader` 参数来设置模块解析的处理细节，`resolve` 用来配置应用层的模块（要被打包的模块）解析，`resolveLoader` 用来配置 `loader` 模块的解析

- > Node.js 模块的依赖解析算法很简单，是通过查看模块的每一层父目录中的 `node_modules` 文件夹来查询依赖的。当出现 Node.js 模块依赖查找失败的时候，可以尝试设置 `resolve.fallback` 和 `resolveLoader.fallback` 来解决问题。

- 默认配置：

  webpack 会假定项目的入口起点为 `src/index`（*./src/index.js*），然后会在 `dist/main.js`（*./dist/main.js*） 输出结果，并且在生产环境开启压缩和优化；

- 默认模式是 `production`：

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

- vendor： 卖主；小贩；供应商；[贸易] 自动售货机

  —— 理解为：第三方库

- 模式：

  > 记住，只设置 `NODE_ENV` 时，不会自动设置 `mode`。

  - **development**

    启用 `NamedChunksPlugin` 和 `NamedModulesPlugin`

  - **production**

    启用 `FlagDependencyUsagePlugin`，`FlagIncludedChunksPlugin`，`ModuleConcatenationPlugin`，`NoEmitOnErrorsPlugin`，`OccurrenceOrderPlugin`，`SideEffectsFlagPlugin` 和 `UglifyJsPlugin`

  - **none**

    不选用任何默认优化选项

- 如果你想要根据 *webpack.config.js* 中的 **mode** 变量去影响编译行为，那你必须将导出对象，改为导出一个函数：

```
var config = {
  entry: './app.js'
  //...
};

module.exports = (env, argv) => {

  if (argv.mode === 'development') {
    config.devtool = 'source-map';
  }

  if (argv.mode === 'production') {
    //...
  }

  return config;
};
```

- webpack 1 需要特定的 loader 来转换 ES 2015 `import`，然而通过 webpack 2 可以开箱即用
- 模块解析

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

当指定的路径是文件夹时，解析步骤：

> - 如果文件夹中包含 `package.json` 文件，则按照顺序查找 `resolve.mainFields` 配置选项中指定的字段。并且 `package.json` 中的第一个这样的字段确定文件路径。
> - 如果 `package.json` 文件不存在或者 `package.json` 文件中的 main 字段没有返回一个有效路径，则按照顺序查找 `resolve.mainFiles` 配置选项中指定的文件名，看是否能在 import/require 目录下匹配到一个存在的文件名。
> - 文件扩展名通过 `resolve.extensions` 选项采用类似的方法进行解析。

- Loader 解析遵循与文件解析器指定的规则相同的规则。但是 `resolveLoader` 配置选项可以用来为 Loader 提供独立的解析规则
- Manifest
- 当需要自己开发类库（lib）的时候，需要了解 `output.library`
- `Rule.loader` 是 `Rule.use: [ { loader } ]` 的简写
- `module.target` - 运行环境
- 更多的配置形式

多个配置对象：

```
module.exports = [{
  output: {
    filename: './dist-amd.js',
    libraryTarget: 'amd'
  },
  entry: './app.js',
  mode: 'production',
}, {
  output: {
    filename: './dist-commonjs.js',
    libraryTarget: 'commonjs'
```

```
  },
  entry: './app.js',
  mode: 'production',
}];
```

3. 配置

3.1 output

3.1.1 path

编译输出目录，默认当前目录：

3.1.2 filename

- 编译输出文件名，多模块入口时，可以有多个区分变量：
  - name - 入口名称
  - id - 内部chunk的id
  - hash - 每次构建过程中，唯一生成的 hash
  - chunkhash - 每个 chunk 内容的 hash
  - query - 模块的 query，例如，文件名 `?` 后面的字符串

  例：[name].js

- `[hash]` 和 `[chunkhash]` 的长度可以使用 `[hash:16]`（默认为20）来指定。或者，通过指定 `output.hashDigestLength` 在全局配置长度。
- 虽然属性名叫“文件名（filename）”，但是也可以定义成文件路径：

  如：/[name]/[id].js

- 此选项不会影响那些「按需加载 chunk」的输出文件。对于这些文件，请使用 `output.chunkFilename` 选项来控制输出。
- 通过 loader 创建的文件也不受影响。在这种情况下，你必须尝试 loader 特定的可用选项。
- 在使用 `ExtractTextWebpackPlugin` 时，可以用 `[contenthash]` 来获取提取文件的 hash（既不是 `[hash]` 也不是 `[chunkhash]`）。

3.1.3 hashDigest

- 在生成 hash 时使用的编码方式，默认为 `'hex'`。支持 Node.js `hash.digest` 的所有编码。
  - 对文件名使用 `'base64'`，可能会出现问题，因为 base64 字母表中具有 `/` 这个字符(character)。

3.1.4 hashFunction

散列算法，默认为 `'md5'`。支持 Node.JS `crypto.createHash` 的所有功能。

- 从 `4.0.0-alpha2` 开始，`hashFunction` 现在可以是一个返回自定义 hash 的构造函数。

3.1.5 hashSalt

加盐值，通过 Node.JS `hash.update` 来更新哈希。

3.1.6 publicPath

此选项指定在浏览器中所引用的「此输出目录对应的**公开 URL**」

```
publicPath: 'https://cdn.example.com/assets/'
```

- 默认值是一个空字符串 `""`；
- 该选项的值是以 runtime(运行时) 或 loader(载入时) 所创建的每个 URL 为前缀。因此，在多数情况下，**此选项的值都会以/结束**。

3.2 resolve

3.2.1 extensions

自动解析确定的扩展，能够使用户在引入模块时不带扩展。默认值为：

```
module.exports = {
  //...
  resolve: {
    extensions: ['.wasm', '.mjs', '.js', '.json']
  }
};
```

> 使用此选项，会**覆盖默认数组**，这就意味着 webpack 将不再尝试使用默认扩展来解析模块。对于使用其扩展导入的模块，例如，`import SomeFile from "./somefile.ext"`，要想正确的解析，一个包含“*”的字符串必须包含在数组中。

3.2.2 alias

```
resolve: {
  extensions: ['.js', '.vue', '.json'],
  alias: {
    '@': resolve('src')  // 别名，创建 import 或 require 的别名，来确保模块引入变得更简单。
  }
}
```

3.3 module

3.3.1 rules

创建模块时，匹配请求的规则数组。这些规则能够修改模块的创建方式。这些规则能够对模块(module)应用 loader，或者修改解析器(parser)。

3.3.1.1 Rule

每个规则可以分为三部分 - 条件(condition)，结果(result)和嵌套规则(nested rule)

3.3.1.1.1 条件

- 条件有两种输入值：
  1. resource：请求文件的绝对路径。它已经根据 `resolve` 规则解析。
  2. issuer: 被请求资源(requested the resource)的模块文件的绝对路径。是导入时的位置。

例如：从 `app.js` 导入 `'./style.css'`，resource 是 `/path/to/style.css`. issuer 是 `/path/to/app.js`。

- 在规则中，属性 `test`, `include`, `exclude` 和 `resource` 对 resource 匹配，并且属性 `issuer` 对 issuer 匹配。

3.3.1.1.2 结果

规则结果只在规则条件匹配时使用。

3.3.1.1.3 test

`Rule.test` 是 `Rule.resource.test` 的简写。如果你提供了一个 `Rule.test` 选项，就不能再提供 `Rule.resource`。

3.3.1.1.4 loaders

`Rule.loaders` 是 `Rule.use` 的别名。

3.3.1.1.5 loader

`Rule.loader` 是 `Rule.use: [ { loader } ]` 的简写。

3.3.1.1.6 use

传递字符串（如：`use: [ "style-loader" ]`）是 loader 属性的简写方式（如：`use: [ { loader: "style-loader "} ]`）。

- 由于需要支持 `Rule.options` 和 `UseEntry.options`，`Rule.use`，`Rule.query` 已废弃。

3.3.1.1.7 options

字符串或对象。值可以传递到 loader 中，将其理解为 loader 选项。

- 由于兼容性原因，也可能有 `query` 属性，它是 `options` 属性的别名。使用 `options` 属性替代。

注意，webpack 需要生成资源和所有 loader 的独立模块标识，包括选项。它尝试对选项对象使用 `JSON.stringify`。 #

4. 踩坑

- webpack-dev-server 进行打包时，它默认打包到根目录下

5. 方案

5.1 Vuejs中使用Awesome图标 - ✈

6. 插件

6.1 html-webpack-plugin

`HtmlWebpackPlugin` 简化了HTML文件的创建，以便为你的webpack包提供服务。

> 这对于在文件名中包含每次会随着编译而发生变化哈希的 webpack bundle 尤其有用。 你可以让插件为你生成一个HTML文件，使用lodash模板提供你自己的模板，或使用你自己的loader。

官方文档

6.2 portfinder

6.2.1 使用

方式一：

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

portfinder.getPort(function (err, port) {
    //
    // `port` is guaranteed to be a free port
    // in this scope.
    //
});
```

方式二：

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

  portfinder.getPortPromise()
    .then((port) => {
        //
        // `port` is guaranteed to be a free port
        // in this scope.
        //
    })
    .catch((err) => {
        //
        // Could not get a free port, `err` contains the reason.
        //
    });
```

注：如果在不支持promise的nodejs版本中调用了 `portfinder.getPortPromise()` 函数，程序会抛出错误，除非对promise进行了扩展支持；

6.2.2 配置

6.2.2.1 portfinder.basePort

端口查询其实端口号

比如：如果设置 `portfinder.basePort` =8080，那么portfinder就会从8080端口开始检查端口是否被占用，知道找到一个闲置的端口，否则，抛出错误；

7. 疑点

- 高级进阶
- ？记住，只设置 `NODE_ENV` 时，不会自动设置 `mode` 。
- 插件目的在于解决 loader 无法实现的**其他事**

8优化

8.1 CommonJS

- > commonJS用同步的方式加载模块。在服务端，模块文件都存在本地磁盘，读取非常快，所以这样做不会有问题。但是在浏览器端，限于网络原因，更合理的方案是使用异步加载

8.2 AMD

- AMD规范采用异步方式加载模块，模块的加载不影响它后面语句的运行

8.3 CMD

- CMD是另一种js模块化方案，它与AMD很类似，不同点在于：AMD 推崇依赖前置、提前执行，CMD推崇依赖就近、延迟执行。此规范其实是在sea.js推广过程中产生的

8.4 ES6引入

- ES6的模块不是对象，`import`命令会被 JavaScript 引擎静态分析，在编译时就引入模块代码，而不是在代码运行时加载，所以无法实现条件加载。也正因为这个，使得静态分析成为可能

- CommonJS 模块输出的是值的拷贝，也就是说，一旦输出一个值，模块内部的变化就影响不到这个值。

  ES6 模块的运行机制与 CommonJS 不一样。JS 引擎对脚本静态分析的时候，遇到模块加载命令`import`，就会生成一个只读引用。等到脚本真正执行时，再根据这个只读引用，到被加载的那个模块里面去取值。换句话说，ES6 的`import`有点像 Unix 系统的“符号连接”，原始值变了，`import`加载的值也会跟着变。因此，ES6 模块是动态引用，并且不会缓存值，模块里面的变量绑定其所在的模块。

- **CommonJS 模块是运行时加载，ES6 模块是编译时输出接口**
  - 运行时加载: CommonJS 模块就是**对象**；即在输入时是先加载整个模块，生成一个对象，然后再从这个对象上面读取方法，这种加载称为“运行时加载”。
  - 编译时加载: ES6 模块**不是对象**，而是通过`export`命令显式指定输出的代码，`import`时采用静态命令的形式。即在`import`时可以指定加载某个输出值，而不是加载整个模块，这种加载称为“编译时加载”

9. 相关博客

- webpack主要配置介绍：http://web.jobbole.com/84847/

10. 扩展

10.1 自由变量

查看博客

11. 跳过的步骤总是要还的

- 手动打包一个应用程序
- 实时编写一个简单的模块打包工具
- 简单模块打包工具的细节说明