大概算是一份教程吧, 只不过效果肯定不如视频演示之类的好..
Webpack 最近在英文社区上经常看到, 留了心, 但进一步了解是通过下边的视频:
视频: How Instagram.com Works, Peter Hunt
Peter Hunt 也是 React 的传教士, 我由于对 React 的关注因此细看了视频
再后来是出现 React Hot Loader 这样的开发神器, 我认为 Webpack 应该很棒
http://gaearon.github.io/react-hot-loader/
为了解决简聊当中一些问题, 我消耗了很多时间了解 Webpack, 整理在这里

Webpack 是什么

https://github.com/webpack
Webpack 是德国开发者 Tobias Koppers 开发的模块加载器
Instagram 工程师认为这个方案很棒, 似乎还把作者招过去了
在 Webpack 当中, 所有的资源都被当作是模块, js, css, 图片等等..
因此, Webpack 当中 js 可以引用 css, css 中可以嵌入图片 dataUrl

对应各种不同文件类型的资源, Webpack 有对应的模块 loader
比如 CoffeeScript 用的是 `coffee-loader`, 其他还有很多:
http://webpack.github.io/docs/list-of-loaders.html
大致的写法也就这样子:

```
  module: {
    loaders: [
      { test: /\.coffee$/, loader: 'coffee-loader' },
      { test: /\.js$/, loader: 'jsx-loader?harmony' } // loaders can take parameters as a querystring
    ]
  },
```

CommonJS 与 AMD 支持

Webpack 对 CommonJS 的 AMD 的语法做了兼容, 方便迁移代码
不过实际上, 引用模块的规则是依据 CommonJS 来的

```
require('lodash') // 从模块目录查找
require('./file') // 按相对路径查找
```

AMD 语法中, 也要注意, 是按 CommonJS 的方案查找的

```
define (require, exports. module) ->
  require('lodash') # commonjs 当中这样是查找模块的
  require('./file')
```

特殊模块的 Shim

比如某个模块依赖 `window.jQuery`, 需要从 npm 模块中将 `jquery` 挂载到全局
Webpack 有不少的 Shim 的模块, 比如 `expose-loader` 用于解决这个问题
https://github.com/webpack/docs/wiki/shimming-modules
其他比如从模块中导出变量...具体说明有点晦涩..

手头的两个例子, 比如我们用到 Pen 这个模块,
这个模块对依赖一个 `window.jQuery`, 可我手头的 jQuery 是 CommonJS 语法的
而 `Pen` 对象又是生成好了绑在全局的, 可是我又需要通过 `require('pen')` 获取变量
最终的写法就是做 Shim 处理直接提供支持:

```
{test: require.resolve('jquery'), loader: 'expose?jQuery'},
{test: require.resolve('pen'), loader: 'exports?window.Pen'},
```

基本的使用

安装 `webpack` 模块之后, 可是使用 `webpack` 这个命令行工具
可以使用参数, 也可以配置 `webpack.config.js` 文件直接运行 `webpack` 调用
建议按照 Peter Hunt 给的教程走一遍, 基本的功能都会用到了
https://github.com/petehunt/webpack-howto

简单的例子就是这样一个文件, 可以把 `./main.js` 作为入口打包 `bundle.js`:

```
// webpack.config.js
module.exports = {
  entry: './main.js',
  output: {
    filename: 'bundle.js'
  }
};
```

查找依赖

Webpack 是类似 Browserify 那样在本地按目录对依赖进行查找的
可以构造一个例子, 用 `--display-error-details` 查看查找过程,
例子当中 `resolve.extensions` 用于指明程序自动补全识别哪些后缀,
注意一下, `extensions` 第一个是**空字符串**! 对应不需要后缀的情况.

```
// webpack.config.js
module.exports = {
  entry: './a.js',
  output: {
    filename: 'b.js'
  },
  resolve: {
    extensions: ['', '.coffee', '.js']
  }
}
```

```
// a.js
require('./c')
```

```
➤➤ webpack --display-error-details
Hash: e38f7089c39a1cf34032
Version: webpack 1.5.3
Time: 54ms
Asset  Size  Chunks             Chunk Names
 b.js  1646       0  [emitted]  main
   [0] ./a.js 15 {0} [built] [1 error]

ERROR in ./a.js
Module not found: Error: Cannot resolve 'file' or 'directory' ./c in /Users/chen/Drafts/webpack/details
resolve file
  /Users/chen/Drafts/webpack/details/c doesn't exist
  /Users/chen/Drafts/webpack/details/c.coffee doesn't exist
  /Users/chen/Drafts/webpack/details/c.js doesn't exist
resolve directory
  /Users/chen/Drafts/webpack/details/c doesn't exist (directory default file)
  /Users/chen/Drafts/webpack/details/c/package.json doesn't exist (directory description file)
[/Users/chen/Drafts/webpack/details/c]
[/Users/chen/Drafts/webpack/details/c.coffee]
[/Users/chen/Drafts/webpack/details/c.js]
 @ ./a.js 2:0-14
```

`./c` 是不存在, 从这个错误信息当中我们大致能了解 Webpack 是怎样查找的
大概就是会尝试各种文件名, 会尝试作为模块, 等等
一般模块就是查找 `node_modules`, 但这个也是能被配置的:
http://webpack.github.io/docs/configuration.html#resolve-modulesdirectories

CSS 及图片的引用

英文的教程上有明确的例子:
https://github.com/petehunt/webpack-howto#5-stylesheets-and-images

```
require('./bootstrap.css');
require('./myapp.less');

var img = document.createElement('img');
img.src = require('./glyph.png');
```

上边的是 JavaScript 代码, CSS 跟 LESS, 还有图片, 被直接引用了
实际上 CSS 被转化为 `<style>` 标签, 而图片可能被转化成 base64 格式的 dataUrl
但是要主要在 `webpack.config.js` 文件写好对应的 `loader`:

```
// webpack.config.js
module.exports = {
  entry: './main.js',
  output: {
    path: './build', // This is where images AND js will go
    publicPath: 'http://mycdn.com/', // This is used to generate URLs to e.g. images
    filename: 'bundle.js'
  },
  module: {
    loaders: [
      { test: /\.less$/, loader: 'style-loader!css-loader!less-loader' }, // use ! to chain loaders
      { test: /\.css$/, loader: 'style-loader!css-loader' },
      {test: /\.(png|jpg)$/, loader: 'url-loader?limit=8192'} // inline base64 URLs for <=8k images, direct URLs
 for the rest
    ]
  }
};
```

`url-loader`

稍微啰嗦一下这个 loader, 这个 loader 实际上是对 `file-loader` 的封装
https://github.com/webpack/url-loader
比如 CSS 文件当中有这样的引用:

```
.demo {
  background-image: url('a.png');
}
```

那么对应这样的 loader 配置就能把 `a.png` 抓出来,
并且按照文件大小, 或者转化为 base64, 或者单独作为文件:

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

上边 `?` 后边的 query 有两种写法, 可以看下文档:
http://webpack.github.io/docs/using-loaders.html#query-parameters

- `file-loader`

由于 `url-loader` 是对 `file-loader` 的一个封装, 以因此带有后者一些功能:
https://github.com/webpack/file-loader
比如说, `file-loader` 有不弱的定义文件名的功能

```
require("file?name=[path][name].[ext]?[hash]!./dir/file.png")
```

对应 `url-loader` 当中如果文件超出体积, 就给一个这样的文件名..

打成多个包

有时考虑类库代码的缓存, 我们会考虑打成多个包, 这样不难
比如下边的配置, 首先 `entry` 有多个属性, 对应多个 JavaScript 包,
然后 `commonsPlugin` 可以用于分析模块的共用代码, 单独打一个包出来:
https://github.com/petehunt/webpack-howto#8-optimizing-common-code
https://github.com/webpack/docs/wiki/optimization#multi-page-app

```
// webpack.config.js
var webpack = require('webpack');
var commonsPlugin = new webpack.optimize.CommonsChunkPlugin('common.js');

module.exports = {
  entry: {
    Profile: './profile.js',
    Feed: './feed.js'
  },
  output: {
    path: 'build',
    filename: '[name].js' // Template based on keys in entry above
  },
  plugins: [commonsPlugin]
};
```

对文件做 revision

这个在文档上做了说明, 可以自动生成 js 文件的 Hash:
http://webpack.github.io/docs/long-term-caching.html

```
output: { chunkFilename: "[chunkhash].bundle.js" }
```

```
plugins: [
  function() {
    this.plugin("done", function(stats) {
      require("fs").writeFileSync(
        path.join(__dirname, "...", "stats.json"),
        JSON.stringify(stats.toJson()));
    });
  }
]
```

同时, 可以注册事件, 拿到生成的带 Hash 的文件的一个表
但是拿到那个表之后, 就需要自己写代码进行替换了.. 这有点麻烦
官网的 Issue 里提到个办法是生成 HTML 时引用 `stats.json` 的数据,
我此前的方案是生成 HTML 之后再进行替换, 相对赖上生成时写入更好一些

上线

- 另一份配置文件

用 `webpack --config webpack.min.js` 指定另一个名字的配置文件
这个文件当中可以写不一样配置, 专门用于代码上线时的操作

- 压缩 JavaScript

因为代码都是 JavaScript, 所以压缩就很简单了, 加上一行 plugin 就好了
http://webpack.github.io/docs/list-of-plugins.html#uglifyjsplugin

```
plugins: [
   new webpack.optimize.MinChunkSizePlugin(minSize)
]
```

- 压缩 React

React 官方提供的代码是已经合并的, 这个是 Webpack 不推荐的用法,
在合并话的代码上进行定制有点麻烦, Webpack 提供了设置环境变量来优化代码的方案:

```
new webpack.DefinePlugin({
  "process.env": {
    NODE_ENV: JSON.stringify("production")
  }
})
```

https://github.com/webpack/webpack/issues/292#issuecomment-44804366

- CDN

替换 CDN 这个工作, Webpack 也内置了, 设置 `output.publicPath` 即可
http://webpack.github.io/docs/configuration.html#output-publicpath

代码热替换

虽然文档上写得挺复杂的, 但如果只是简单的功能还是很容易的

第一步, 把 `'webpack/hot/dev-server'` 加入到打包的代码当中,
这个是对应 `node_modules/webpack/` 目录当中的文件的:

```
  entry: {
    main: ['webpack/hot/dev-server', './main'],
    vendor: ['lodash', './styles']
  },
```

启动服务器, 比如我是这样子的

```
webpack-dev-server --hot --quiet
```

正常可以看到提示说服务器已经起来了
http://localhost:8080/webpack-dev-server/
如果有 `index.html` 的话, 直接访问网址应该就能开始调试了

React Hot Replace

调试 React 的话, 有这样的工具简直是神器了, 甚至不用刷新页面!
http://gaearon.github.io/react-hot-loader/getstarted/

```
entry: [
  'webpack-dev-server/client?http://0.0.0.0:8080', // WebpackDevServer host and port
  'webpack/hot/only-dev-server',
  './scripts/index' // Your app’s entry point
]
```

我特意问了下作者为什么上边配置看起来不一样..
https://github.com/gaearon/react-hot-loader/issues/73#issuecomment-73679446
回复大致说是为了避免自动的强制刷新他用了特别的写法..
关于这项功能具体如何实现, 我没有深入了解过...

hot replace 非静态的网页

上边 `localhost:8080` 的方案并不适合复杂的页面,
于是文档上给出了一套稍微复杂一些的方案, 用来配合其他的服务器调试
大致的思路是这样的:

Webpack 打包生成的那些静态资源用服务器 A 进行 serve
这里说的 A 就是上边说的这个:

```
webpack-dev-server --hot --quiet
```

我们的 HTML 由 B 渲染, B 会引用 A serve 的静态资源
B 生成的页面当中加上类似这样的代码:

```
<script src="http://<A 的地址>/assets/bundle.js">
```

还可能要设置一下 `output.publicPath`, 把所有静态资源指向 A
3. 文件修改时, `webpack-dev-server` 通过 `socket.io` 通知客户端更新

这个步骤在文档上写得有点难懂, 大概要多尝试几次才行, 我也弄错很多次
http://webpack.github.io/docs/webpack-dev-server.html

单独打包 CSS

因为公司里有这个需要求, 强制把 CSS 从 js 文件当中独立出来.
官方文档是以插件的形式做的:
http://webpack.github.io/docs/stylesheets.html#separate-css-bundle
参考文档但是注意一下函数参数, 第一第二个参数是有区别的, 比如这样用:

```
ExtractTextPlugin.extract('style-loader', 'css!less')
```

第一个参数是编译好以后的代码用的, 第二个参数是编译到源代码用的.. 有点难懂..

感想

Webpack 的报错挺不友好的, 最初的时候我看着模块找不到没法搞明白
这种时候把中间过程打印出来看是不错的选择:

```
webpack --display-error-details
```

另一个报错是没有对应 loader 的提示.. log 可能很长找不到重点
我建议是先自己去想想什么地方需要考虑 loader 吧... 可能就知道了
我还遇到就是源码里有使用 dataUrl 导致报错... 确实奇怪了

不说这些坑的话, Webpack 我认为是我目前接触到最好的前端开发方案
很多功能之前 FIS 文档上看到过, 但 FIS 相对重一些我始终没上手
而 Webpack 一上来就绕过了此前公司用 RequireJS 打包时遇到的各种问题

如果去扫 Webpack 的文档的话, 还有很多功能我完全没涉及到..
http://webpack.github.io/docs/