webpack：

1 - 它同时支持commonjs和AMD规范（甚至混合的形式）；

2 - 它可以打成一个完整的包，也可以分成多个部分，在运行时异步加载（可以减少第一次加载的时间）；

3 - 依赖在编译时即处理完毕，可以减少运行时包的大小；

4 - Loaders可以使文件在编译时得到预处理，这可以帮我们做很多事情，比如说模板的预编译，图片的base64处理；

5 - 丰富的和可扩展的插件可以适应多变的需求。

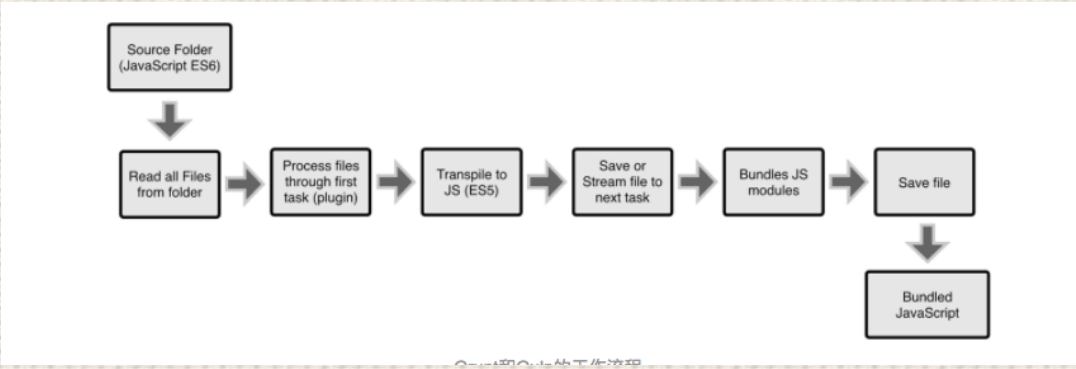
什么是Webpack：

WebPack可以看做是模块打包机：它做的事情是，分析你的项目结构，找到JavaScript模块以及其它的一些浏览器不能直接运行的拓展语言（Scss，TypeScript等），并将其打包为合适的格式以供浏览器使用。

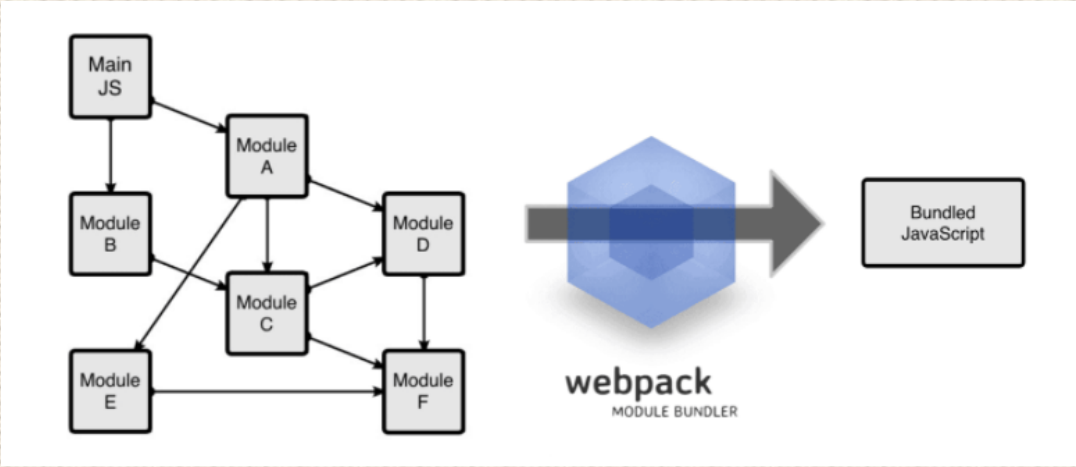
Webpack和Grunt以及Gulp相比有什么特性：

其实Webpack和另外两个并没有太多的可比性，Gulp/Grunt是一种能够优化前端的开发流程的工具，而WebPack是一种模块化的解决方案，不过Webpack的优点使得Webpack可以替代Gulp/Grunt类的工具。

Grunt和Gulp的工作方式是：在一个配置文件中，指明对某些文件进行类似编译，组合，压缩等任务的具体步骤，这个工具之后可以自动替你完成这些任务。



Webpack的工作方式是：把你的项目当做一个整体，通过一个给定的主文件（如：index.js），Webpack将从这个文件开始找到你的项目的所有依赖文件，使用loaders处理它们，最后打包为一个浏览器可识别的JavaScript文件。



Webpack的处理速度更快更直接，能打包更多不同类型的文件。

安装

Webpack可以使用npm安装，新建一个空的练习文件夹（此处命名为webpack sample project），在终端中转到该文件夹后执行下述指令就可以完成安装。

```
//全局安装
npm install -g webpack
//安装到你的项目目录
npm install --save-dev webpack
```

正式使用Webpack前的准备

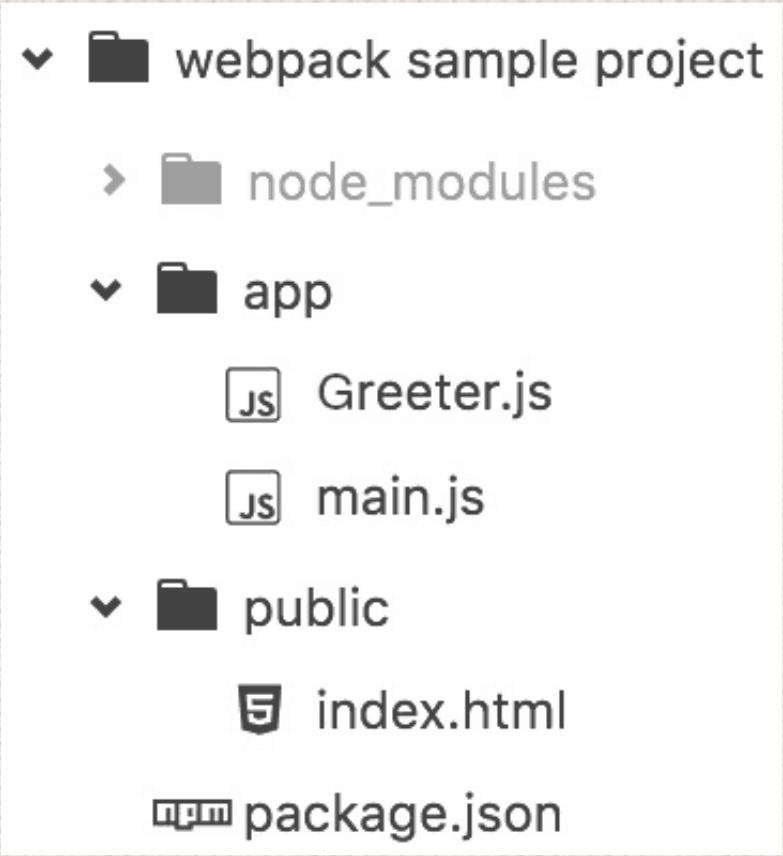
在上述练习文件夹中创建一个package.json文件，这是一个标准的npm说明文件，里面蕴含了丰富的信息，包括当前项目的依赖模块，自定义的脚本任务等等。在终端中使用`npm init`命令可以自动创建这个package.json文件

```
npm init
```

1、package.json文件已经就绪，我们在本项目中安装Webpack作为依赖包

```
// 安装Webpack
npm install --save-dev webpack
```

2、回到之前的空文件夹，并在里面创建两个文件夹,app文件夹和public文件夹，app文件夹用来存放原始数据和我们将写的JavaScript模块，public文件夹用来存放准备给浏览器读取的数据（包括使用webpack生成的打包后的js文件以及一个index.html文件）。在这里还需要创建三个文件，index.html 文件放在public文件夹中，两个js文件（Greeter.js和main.js）放在app文件夹中，此时项目结构如下图所示



index.html文件只有最基础的html代码，它唯一的目的就是加载打包后的js文件

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Webpack Sample Project</title>
  </head>
  <body>
    <div id='root'>
    </div>
    <script src="bundle.js"></script>
  </body>
</html>
```

Greeter.js只包括一个用来返回包含问候信息的html元素的函数。

```
// Greeter.js
module.exports = function() {
  var greet = document.createElement('div');
  greet.textContent = "Hi there and greetings!";
  return greet;
};
```

main.js用来把Greeter模块返回的节点插入页面。

```
//main.js
var greeter = require('./Greeter.js');
document.getElementById('root').appendChild(greeter());
```

通过配置文件来使用Webpack

Webpack拥有很多其它的比较高级的功能（比如说本文后面会介绍的loaders和plugins），这些功能其实都可以通过命令行模式实现，但是正如已经提到的，这样不太方便且容易出错的，一个更好的办法是定义一个配置文件，这个配置文件其实也是一个简单的JavaScript模块，可以把所有的与构建相关的信息放在里面。

还是继续上面的例子来说明如何写这个配置文件，在当前练习文件夹的根目录下新建一个名为webpack.config.js的文件，并在其中进行最最简单的配置，如下所示，它包含入口文件路径和存放打包后文件的地方的路径。

```
module.exports = {
  entry:  __dirname + "/app/main.js",//已多次提及的唯一入口文件
  output: {
    path: __dirname + "/public",//打包后的文件存放的地方
    filename: "bundle.js"//打包后输出文件的文件名
  }
}
```

注：“__dirname”是node.js中的一个全局变量，它指向当前执行脚本所在的目录。

现在如果你需要打包文件只需要在终端里你运行webpack(非全局安装需使用node_modules/.bin/webpack)命令就可以了，这条命令会自动参考webpack.config.js文件中的配置选项打包你的项目。

更快捷的执行打包任务

执行类似于`node_modules/.bin/webpack`这样的命令其实是比较烦人且容易出错的，不过值得庆幸的是npm可以引导任务执行，对其进行配置后可以使用简单的`npm start`命令来代替这些繁琐的命令。在package.json中对npm的脚本部分进行相关设置即可，设置方法如下。

```
{
  "name": "webpack-sample-project",
  "version": "1.0.0",
  "description": "Sample webpack project",
  "scripts": {
    "start": "webpack" //配置的地方就是这里啦，相当于把npm的start命令指向webpack命令
  },
  "author": "zhang",
  "license": "ISC",
  "devDependencies": {
    "webpack": "^1.12.9"
  }
}
```

> 注：package.json中的脚本部分已经默认在命令前添加了 `node_modules/.bin` 路径，所以无论是全局还是局部安装的Webpack，你都不需要写前面那指明详细的路径了。

在命令行中使用npm start就可以执行相关命令。

Webpack的强大功能

生成Source Maps（使调试更容易）

开发总是离不开调试，如果可以更加方便的调试当然就能提高开发效率，不过打包后的文件有时候你是不容易找到出错了的地方对应的源代码的位置的，Source Maps就是来帮我们解决这个问题的。

通过简单的配置后，Webpack在打包时可以为我们生成的source maps，这为我们提供了一种对应编译文件和源文件的方法，使得编译后的代码可读性更高，也更容易调试。

在webpack的配置文件中配置source maps，需要配置`devtool`，它有以下四种不同的配置选项，各具优缺点，描述如下：

devtool 选项	配置结果
source-map	在一个单独的文件中产生一个完整且功能完全的文件。这个文件具有最好的source map，但是它会减慢打包文件的构建速度；
cheap-module-source-map	在一个单独的文件中生成一个不带列映射的map，不带列映射提高项目构建速度，但是也使得浏览器开发者工具只能对应到具体的行，不能对应到具体的列（符号），会对调试造成不便；
eval-source-map	使用eval打包源文件模块，在同一个文件中生成干净的完整的source map。这个选项可以在不影响构建速度的前提下生成完整的sourcemap，但是对打包后输出的JS文件的执行具有性能和安全的隐患。不过在开发阶段这是一个非常好的选项，但是在生产阶段一定不要用这个选项；
cheap-module-eval-source-map	这是在打包文件时最快的生成source map的方法，生成的Source Map 会和打包后的JavaScript文件同行显示，没有列映射，和 `eval-source-map` 选项具有相似的缺点；

正如上表所述，上述选项由上到下打包速度越来越快，不过同时也具有越来越多的负面作用，较快的构建速度的后果就是对打包后的文件的的执行有一定影响。在学习阶段以及在小到中性的项目上，`eval-source-map`是一个很好的选项，不过记得只在开发阶段使用它，继续上面的例子，进行如下配置

```
module.exports = {
  devtool: 'eval-source-map',//配置生成Source Maps，选择合适的选项
  entry:  __dirname + "/app/main.js",
  output: {
    path: __dirname + "/public",
    filename: "bundle.js"
  }
}
```

> `cheap-module-eval-source-map` 方法构建速度更快，但是不利于调试，推荐在大型项目考虑da时间成本是使用。

使用webpack构建本地服务器

想不想让你的浏览器监测你的代码的修改，并自动刷新修改后的结果，其实Webpack提供一个可选的本地开发服务器，这个本地服务器基于node.js构建，可以实现你想要的这些功能，不过它是一个单独的组件，在webpack中进行配置之前需要单独安装它作为项目依赖

```
npm install --save-dev webpack-dev-server
```

devserver作为webpack配置选项中的一项，具有以下配置选项

devserver配置选项	功能描述
contentBase	默认webpack-dev-server会为根文件夹提供本地服务器，如果想为另外一个目录下的文件提供本地服务器，应该在这里设置其所在目录（本例设置到“public"目录）
port	设置默认监听端口，如果省略，默认为”8080“
inline	设置为 `true`，当源文件改变时会自动刷新页面
colors	设置为 `true`，使终端输出的文件为彩色的
historyApiFallback	在开发单页应用时非常有用，它依赖于HTML5 history API，如果设置为 `true`，所有的跳转将指向index.html

继续把这些命令加到webpack的配置文件中，现在的配置文件如下所示

```
module.exports = {
  devtool: 'eval-source-map',

  entry:  __dirname + "/app/main.js",
  output: {
    path: __dirname + "/public",
    filename: "bundle.js"
  },

  devServer: {
    contentBase: "./public",//本地服务器所加载的页面所在的目录
    colors: true,//终端中输出结果为彩色
    historyApiFallback: true,//不跳转
    inline: true//实时刷新
  }
}
```

Loaders

鼎鼎大名的Loaders登场了！

Loaders是webpack中最让人激动人心的功能之一了。通过使用不同的loader，webpack通过调用外部的脚本或工具可以对各种各样的格式的文件进行处理，比如说分析JSON文件并把它转换为JavaScript文件，或者说把下一代的JS文件（ES6，ES7)转换为现代浏览器可以识别的JS文件。或者说对React的开发而言，合适的Loaders可以把React的JSX文件转换为JS文件。

Loaders需要单独安装并且需要在webpack.config.js下的`modules`关键字下进行配置，Loaders的配置选项包括以下几方面：

- `test`：一个匹配loaders所处理的文件的拓展名的正则表达式（必须）
- `loader`：loader的名称（必须）
- `include/exclude`:手动添加必须处理的文件（文件夹）或屏蔽不需要处理的文件（文件夹）（可选）；
- `query`：为loaders提供额外的设置选项（可选）

继续上面的例子，我们把Greeter.js里的问候消息放在一个单独的JSON文件里，并通过合适的配置使Greeter.js可以读取该JSON文件的值，配置方法如下

```
//安装可以装换JSON的loader
npm install --save-dev json-loader
```

```
module.exports = {
  devtool: 'eval-source-map',

  entry:  __dirname + "/app/main.js",
  output: {
    path: __dirname + "/public",
    filename: "bundle.js"
  },

  module: {//在配置文件里添加JSON loader
    loaders: [
      {
        test: /\.json$/,
        loader: "json"
      }
    ]
  },

  devServer: {
    contentBase: "./public",
    colors: true,
    historyApiFallback: true,
    inline: true
  }
}
```

创建带有问候信息的JSON文件(命名为config.json)

```
//config.json
{
  "greetText": "Hi there and greetings from JSON!"
}
```

更新后的Greeter.js

```
var config = require('./config.json');

module.exports = function() {
  var greet = document.createElement('div');
  greet.textContent = config.greetText;
  return greet;
};
```

Loaders很好，不过有的Loaders使用起来比较复杂，比如说Babel。

Babel

Babel其实是一个编译JavaScript的平台，它的强大之处表现在可以通过编译帮你达到以下目的：

- 下一代的JavaScript标准（ES6，ES7），这些标准目前并未被当前的浏览器完全的支持；
- 使用基于JavaScript进行了拓展的语言，比如React的JSX

Babel的安装与配置

Babel其实是几个模块化的包，其核心功能位于称为babel-core的npm包中，不过webpack把它们整合在一起使用，但是对于每一个你需要的功能或拓展，你都需要安装单独的包（用得最多的是解析Es6的babel-preset-es2015包和解析JSX的babel-preset-react包）。

我们先来一次性安装这些依赖包

```
// npm一次性安装多个依赖模块，模块之间用空格隔开
npm install --save-dev babel-core babel-loader babel-preset-es2015 babel-preset-react
```

在webpack中配置Babel的方法如下

```
module.exports = {
  devtool: 'eval-source-map',

  entry:  __dirname + "/app/main.js",
  output: {
    path: __dirname + "/public",
    filename: "bundle.js"
  },

  module: {
    loaders: [
      {
        test: /\.json$/,
        loader: "json"
      },
      {
        test: /\.js$/,
        exclude: /node_modules/,
        loader: 'babel',//在webpack的module部分的loaders里进行配置即可
        query: {
          presets: ['es2015','react']
        }
      }
    ]
  },

  devServer: {
    contentBase: "./public",
    colors: true,
    historyApiFallback: true,
    inline: true
  }
}
```

现在你的webpack的配置已经允许你使用ES6以及JSX的语法了。继续用上面的例子进行测试，不过这次我们会使用React，记得先安装 React 和 React-DOM

```
npm install --save react react-dom
```

使用ES6的语法，更新Greeter.js并返回一个React组件

```
//Greeter.js
import React, {Component} from 'react'
import config from './config.json';

class Greeter extends Component{
  render() {
    return (
      <div>
        {config.greetText}
      </div>
    );
  }
}

export default Greeter
```

使用ES6的模块定义和渲染Greeter模块

main.js

```
import React from 'react';
import {render} from 'react-dom';
import Greeter from './Greeter';

render(<Greeter />, document.getElementById('root'));
```

Babel的配置选项

Babel其实可以完全在webpack.config.js中进行配置，但是考虑到babel具有非常多的配置选项，在单一的webpack.config.js文件中进行配置往往使得这个文件显得太复杂，因此一些开发者支持把babel的配置选项放在一个单独的名为".babelrc" 的配置文件中。我们现在的babel的配置并不算复杂，不过之后我们会再加一些东西，因此现在我们就提取出相关部分，分两个配置文件进行配置（webpack会自动调用`.babelrc`里的babel配置选项），如下：

```
// webpack.config.js
module.exports = {
  devtool: 'eval-source-map',

  entry:  __dirname + "/app/main.js",
  output: {
    path: __dirname + "/public",
    filename: "bundle.js"
  },

  module: {
    loaders: [
      {
        test: /\.json$/,
        loader: "json"
      },
      {
        test: /\.js$/,
        exclude: /node_modules/,
        loader: 'babel'
      }
    ]
  },

  devServer: {...} // Omitted for brevity
}
```

```
//.babelrc
{
  "presets": ["react", "es2015"]
}
```

CSS

webpack提供两个工具处理样式表，`css-loader` 和 `style-loader`，二者处理的任务不同，`css-loader`使你能够使用类似`@import` 和 `url(...)`的方法实现 `require()`的功能,`style-loader`将所有的计算后的样式加入页面中，二者组合在一起使你能够把样式表嵌入webpack打包后的JS文件中。

继续上面的例子

```
//安装
npm install --save-dev style-loader css-loader
```

```
//使用
module.exports = {
  devtool: 'eval-source-map',

  entry:  __dirname + "/app/main.js",
  output: {
    path: __dirname + "/build",
    filename: "bundle.js"
  },

  module: {
    loaders: [
      {
        test: /\.json$/,
        loader: "json"
      },
      {
        test: /\.js$/,
        exclude: /node_modules/,
        loader: 'babel'
      },
      {
        test: /\.css$/,
        loader: 'style!css'//添加对样式表的处理
      }
    ]
  },

  devServer: {...}
}
```

> **注：感叹号的作用在于使同一文件能够使用不同类型的loader**

接下来，在app文件夹里创建一个名字为"main.css"的文件，对一些元素设置样式

```
html {
  box-sizing: border-box;
  -ms-text-size-adjust: 100%;
  -webkit-text-size-adjust: 100%;
}

*, *:before, *:after {
  box-sizing: inherit;
}

body {
  margin: 0;
  font-family: 'Helvetica Neue', Helvetica, Arial, sans-serif;
}

h1, h2, h3, h4, h5, h6, p, ul {
  margin: 0;
  padding: 0;
}
```

你还记得吗？webpack只有单一的入口，其它的模块需要通过 import, require, url等导入相关位置，为了让webpack能找到”main.css“文件，我们把它导入”main.js “中，如下

```
//main.js
import React from 'react';
import {render} from 'react-dom';
import Greeter from './Greeter';

import './main.css';//使用require导入css文件

render(<Greeter />, document.getElementById('root'));
```

> 通常情况下，css会和js打包到同一个文件中，并不会打包为一个单独的css文件，不过通过合适的配置webpack也可以把css打包为单独的文件的。
> 不过这也只是webpack把css当做模块而已，咱们继续看看一个真的CSS模块的实践。

CSS module

在过去的一些年里，JavaScript通过一些新的语言特性，更好的工具以及更好的实践方法（比如说模块化）发展得非常迅速。模块使得开发者把复杂的代码转化为小的，干净的，依赖声明明确的单元，且基于优化工具，依赖管理和加载管理可以自动完成。

不过前端的另外一部分，CSS发展就相对慢一些，大多的样式表却依旧是巨大且

充满了全局类名，这使得维护和修改都非常困难和复杂。
最近有一个叫做 CSS modules 的技术就意在把JS的模块化思想带入CSS中来，通过CSS模块，所有的类名，动画名默认都只作用于当前模块。Webpack从一开始就对CSS模块化提供了支持，在CSS loader中进行配置后，你所需要做的一切就是把”modules“传递到所需要的地方，然后就可以直接把CSS的类名传递到组件的代码中，且这样做只对当前组件有效，不必担心在不同的模块中具有相同的类名可能会造成的问题。具体的代码如下

```
module.exports = {
  devtool: 'eval-source-map',

  entry:  __dirname + "/app/main.js",
  output: {...},

  module: {
    loaders: [
      {
        test: /\.json$/,
        loader: "json"
      },
      {
        test: /\.js$/,
        exclude: /node_modules/,
        loader: 'babel'
      },
      {
        test: /\.css$/,
        loader: 'style!css?modules'//跟前面相比就在后面加上了?modules
      }
    ]
  },

  devServer: {...}
}
```

创建一个Greeter.css文件

```
.root {
  background-color: #eee;
  padding: 10px;
  border: 3px solid #ccc;
}
```

导入.root到Greeter.js中

```
import React, {Component} from 'react';
import config from './config.json';
import styles from './Greeter.css';//导入

class Greeter extends Component{
  render() {
    return (
      <div className={styles.root}>//添加类名
        {config.greetText}
      </div>
    );
  }
}

export default Greeter
```

放心使用把，相同的类名也不会造成不同组件之间的污染。

CSS modules 也是一个很大的主题，有兴趣的话可以去官方文档查看更多消息

CSS预处理器

Sass 和 Less之类的预处理器是对原生CSS的拓展，它们允许你使用类似于variables, nesting, mixins, inheritance等不存在于CSS中的特性来写CSS，CSS预处理器可以这些特殊类型的语句转化为浏览器可识别的CSS语句，

插件（Plugins）

插件（Plugins）是用来拓展Webpack功能的，它们会在整个构建过程中生效，执行相关的任务。

Loaders和Plugins常常被弄混，但是他们其实是完全不同的东西，可以这么来说，loaders是在打包构建过程中用来处理源文件的（JSX，Scss，Less..），一次处理一个，插件并不直接操作单个文件，它直接对整个构建过程其作用。

Webpack有很多内置插件，同时也有很多第三方插件，可以让我们完成更加丰富的功能。

使用插件的方法

要使用某个插件，我们需要通过npm安装它，然后要做的就是在webpack配置中的plugins关键字部分添加该插件的一个实例（plugins是一个数组）继续看例子，我们添加了一个实现版权声明的插件。

```
//webpack.config.js
var webpack = require('webpack');

module.exports = {
  devtool: 'eval-source-map',
  entry:  __dirname + "/app/main.js",
  output: {...},

  module: {
    loaders: [
      { test: /\.json$/, loader: "json" },
      { test: /\.js$/, exclude: /node_modules/, loader: 'babel' },
      { test: /\.css$/, loader: 'style!css?modules!postcss' }//这里添加PostCSS
    ]
  },
  postcss: [
    require('autoprefixer')
  ],

  plugins: [
    new webpack.BannerPlugin("Copyright Flying Unicorns inc.")//在这个数组中new一个就可
  ],

  devServer: {...}
}
```

产品阶段的构建

目前为止，我们已经使用webpack构建了一个完整的开发环境。但是在产品阶段，可能还需要对打包的文件进行额外的处理，比如说优化，压缩，缓存以及分离CSS和JS。

对于复杂的项目来说，需要复杂的配置，这时候分解配置文件为多个小的文件可以使得事情井井有条，以上面的例子来说，我们创建一个“webpack.production.config.js”的文件，在里面加上基本的配置,它和原始的webpack.config.js很像，如下

```
var webpack = require('webpack');
var HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {
  entry: __dirname + "/app/main.js",
  output: {
    path: __dirname + "/build",
    filename: "bundle.js"
  },

  module: {
    loaders: [
      {
        test: /\.json$/,
        loader: "json"
      },
      {
        test: /\.js$/,
        exclude: /node_modules/,
        loader: 'babel'
      },
      {
        test: /\.css$/,
        loader: 'style!css?modules!postcss'
      }
    ]
  },
  postcss: [
    require('autoprefixer')
  ],

  plugins: [
    new HtmlWebpackPlugin({
      template: __dirname + "/app/index.tmpl.html"
    }),
  ],

}
```

```
//package.json
{
  "name": "webpack-sample-project",
  "version": "1.0.0",
  "description": "Sample webpack project",
  "scripts": {
    "start": "webpack-dev-server --progress",
    "build": "NODE_ENV=production webpack --config ./webpack.production.config.js --
  },
  "author": "Cássio Zen",
  "license": "ISC",
  "devDependencies": {...},
  "dependencies": {...}
}
```

优化插件

webpack提供了一些在发布阶段非常有用的优化插件，它们大多来自于webpack社区，可以通过npm安装，通过以下插件可以完成产品发布阶段所需的功能

- OccurenceOrderPlugin :为组件分配ID，通过这个插件webpack可以分析和优先考虑使用最多的模块，并为它们分配最小的ID
- UglifyJsPlugin：压缩JS代码；
- ExtractTextPlugin：分离CSS和JS文件

我们继续用例子来看看如何添加它们，OccurenceOrder 和 UglifyJS plugins 都是内置插件，你需要做的只是安装它们

```
npm install --save-dev extract-text-webpack-plugin
```

在配置文件的plugins后引用它们

```
var webpack = require('webpack');
var HtmlWebpackPlugin = require('html-webpack-plugin');
var ExtractTextPlugin = require('extract-text-webpack-plugin');

module.exports = {
  entry: __dirname + "/app/main.js",
  output: {
    path: __dirname + "/build",
    filename: "bundle.js"
  },

  module: {
    loaders: [
      {
        test: /\.json$/,
        loader: "json"
      },
      {
        test: /\.js$/,
        exclude: /node_modules/,
        loader: 'babel'
      },
      {
        test: /\.css$/,
        loader: ExtractTextPlugin.extract('style', 'css?modules!postcss')
      }
    ]
  },
  postcss: [
    require('autoprefixer')
  ],

  plugins: [
    new HtmlWebpackPlugin({
      template: __dirname + "/app/index.tmpl.html"
    }),
    new webpack.optimize.OccurenceOrderPlugin(),
    new webpack.optimize.UglifyJsPlugin(),
    new ExtractTextPlugin("style.css")
  ]
}
```

缓存

缓存无处不在，使用缓存的最好方法是保证你的文件名和文件内容是匹配的（内容改变，名称相应改变）

webpack可以把一个哈希值添加到打包的文件名中，使用方法如下,添加特殊的字符串混合体（[name], [id] and [hash]）到输出文件名前

```
var webpack = require('webpack');
var HtmlWebpackPlugin = require('html-webpack-plugin');
var ExtractTextPlugin = require('extract-text-webpack-plugin');

module.exports = {
  entry: __dirname + "/app/main.js",
  output: {
    path: __dirname + "/build",
    filename: "[name]-[hash].js"
  },

  module: {
    loaders: [
      {
        test: /\.json$/,
        loader: "json"
      },
      {
        test: /\.js$/,
        exclude: /node_modules/,
        loader: 'babel'
      },
      {
        test: /\.css$/,
        loader: ExtractTextPlugin.extract('style', 'css?modules!postcss')
      }
    ]
  },
  postcss: [
    require('autoprefixer')
  ],

  plugins: [
    new HtmlWebpackPlugin({
      template: __dirname + "/app/index.tmpl.html"
    }),
    new webpack.optimize.OccurenceOrderPlugin(),
    new webpack.optimize.UglifyJsPlugin(),
    new ExtractTextPlugin("[name]-[hash].css")
  ]
}
```