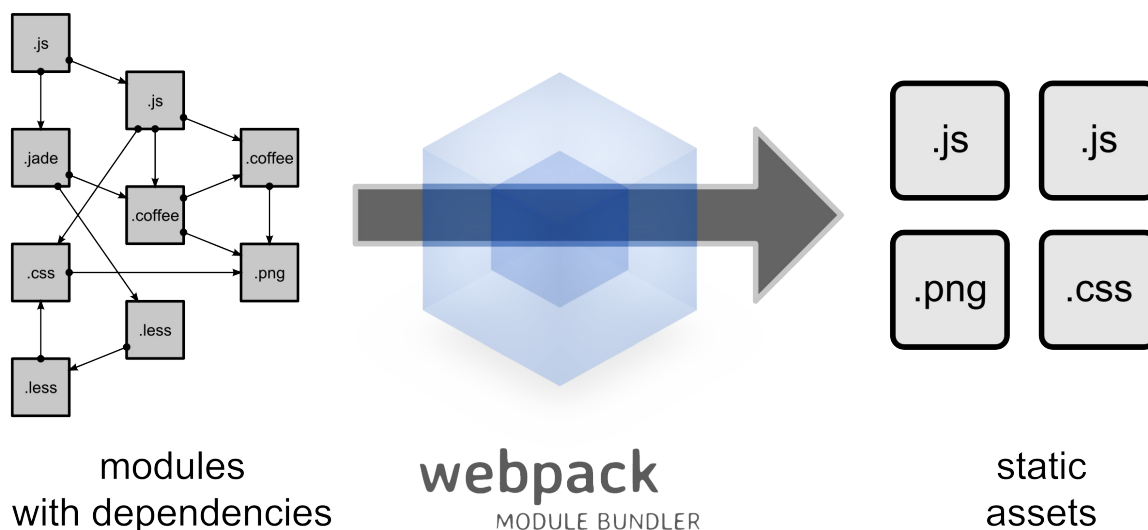

目錄

Introduction	1.1
第一章 模块化	1.2
原始时代	1.2.1
命名空间时代	1.2.2
模块化时代	1.2.3
第二章 webpack入门	1.3
入口文件	1.3.1
非模块化文件打包	1.3.2
AMD模块打包	1.3.3
CommonJS模块打包	1.3.4
Node模块和NPM	1.3.5
UMD模块打包	1.3.6
第三章 webpack进阶	1.4
CLI与API使用模式	1.4.1
基本配置项	1.4.2
分片	1.4.3
CommonChunks插件	1.4.4
SourceMap	1.4.5
高级配置项	1.4.6
第四章 Loader	1.5
使用Loader	1.5.1
常用Loader	1.5.2
bundle-loader	1.5.3
exports-loader	1.5.4
imports-loader	1.5.5
expose-loader	1.5.6
非JS资源管理	1.5.7
生态圈	1.5.8
Loader原理及编写	1.5.9
第五章 集成webpack	1.6
Gulp	1.6.1
第六章 杂谈	1.7
jQuery的引入	1.7.1
TypeScript和Vue	1.7.2

webpack指南

webpack

webpack (<https://webpack.github.io/>) 是一个用于web项目的模块打包工具。在大部分的使用场景中，我们将它看作是一个web前端模块打包工具。



按照官方介绍，webpack可以将前端各种资源（包括CSS及其预编译方案、JS及其预编译方案）统一打包为 .js 文件和资源文件（图片）。

关于本指南

本指南由TooBug编写，所有章节内容均按自己的理解编写，初衷是为了方便国内的前端开发者更好地接触和使用webpack这个强大的前端打包工具。

初次提交诞生于2015年10月20日，彼时webpack还是1.x，国内中文文档相当少，而官方英文文档也语焉不详，饱受诟病，故诞生此指南。

然而由于个人精力方面的原因，本指南未能如期完成。时下webpack的资料已经遍地开花，随着webpack 2的发布，一方面官方文档也趋于完善，另一方面很多配置和用法已不再兼容，因此本指南的部分内容可能存在不准确的情况，请读者自行鉴别。

另外，本指南有可能长期弃坑.....

联系方式：

- 微博 @TooBug
- GitHub @TooBug

本书Github地址<https://github.com/TooBug/webpack-guide>，有任何意见和建议欢迎提出。

第一章 模块化

TODO：模块化简介

原始时代

原始时代场景比较简单，较少碰到命名冲突和复杂依赖的场景，问题不突出。

命名空间时代

复杂度增加之后，命名冲突的问题突显出来，出现了命名空间的模式，即一个Library只占用一个全局变量，所有内容都挂载在命名空间下。

模块化时代

在忍受了命名空间一大串的名字很多年之后，终于有了模块化。促使模块化诞生的另一个因素是依赖管理的问题。

限制

浏览器对资源加载有同源策略限制，也不支持编程化加载资源。（支持加载，不告诉你加载结果，自己猜。）

最终大部分加载器选择通过 `<script>` 标签加载，然后通过各种hack判断是否加载完成。

AMD

require.js将AMD发扬光大，成为AMD事实标准。

模块定义和使用：

```
define(id?, dependencies?, function factory(){  
  
    return moduleContent;  
  
});
```

优点：浏览器直接使用。

Common.js / CMD

Common.js模块定义和使用：

```
var dependency = require('xxx');  
  
// 模块定义  
exports.xxx = xxx;  
  
// 或者  
module.exports = moduleContent;
```

CMD模块定义和使用：

```
define(function(require, exports, module) {  
    var a = require('./a')  
    var b = require('./b') // 依赖可以就近书写  
})
```

UMD

```
(function (root, factory) {
  if (typeof define === 'function' && define.amd) {
    // AMD. Register as an anonymous module.
    define(['b'], factory);
  } else if (typeof module === 'object' && module.exports) {
    // Node. Does not work with strict CommonJS, but
    // only CommonJS-like environments that support module.exports,
    // like Node.
    module.exports = factory(require('b'));
  } else {
    // Browser globals (root is window)
    root.returnExports = factory(root.b);
  }
})(this, function (b) {
  //use b in some fashion.

  // Just return a value to define the module export.
  // This example returns an object, but the module
  // can return a function as the exported value.
  return {};
}));
```

问题

基于运行时，部分实现依赖于hack，没有可靠的基础，虽然有完善的测试，但还是会碰到意外情况。

浏览器端限制多，导致hack多，配置项多。

优化工具不够好用。

无法复用生态圈，需要额外适配。

第二章 webpack入门

与require.js / sea.js不同，webpack是一个在开发阶段进行打包的模块化工具，也就是说它无法不经过打包直接在线上使用。

webpack同时兼容AMD、Common.js以及非模块化的写法。注意这里的同时兼容可不是指你可以任选一种，而是.....你可以同时用三种！

当然你可能会说，谁他X同时用三种模块化方案？执念（就是神经病的意思）吧？

Yes，如果你是一个全新的项目，当然不应该如此规划，但如果是一个已经换了几拨人，跑了好多年的老项目，可能情况就没那么理想了。而这种情况下webpack也能比较从容的应对。简单说，适应性超强的。

而webpack之所以好用，正是因为有诸如此类超出预期的特性，有可能你不需要，但如果你想要的时候就会爽得不要不要的。

webpack基于Node.js编写，在进入下文前，请确保已正确安装Node.js。后文中我们可能会使用全局安装的webpack或者gulp来完成打包，因此建议全局安装好webpack和gulp，后文不再解释。

```
npm install -g webpack gulp
```

由于是全局安装，在Mac/Linux下，如果碰到 EACCES 错误，则可能需要在前面加 sudo 才能安装成功。

由于众所周知的原因，国内使用npm有时候不太顺利，可以考虑使用淘宝的镜像。

```
npm install -g webpack gulp --registry=http://registry.npm.taobao.org
```

闲话不多说，咱们走着！

入口文件

为了方便，这里先解释一下入口文件的概念，这个概念在前端模块化中还是比较重要的，不管是用什么模块化规范，基本都绕不开这个概念。

简单点说，入口文件就是在HTML直接引用的，由浏览器触发执行的JS文件。其它的非入口文件则是由入口文件来直接或间接依赖，由JS互相调用执行。举个简单的例子：

index.html:

```
<html>
  <head>
    <script src="main.js"></script>
  </head>
  <body>
  </body>
</html>
```

main.js

```
var a = require('./a');
var b = require('./b');

a.doSth();
b.doSth();
```

这个例子中，`main.js` 由HTML直接引用，是整个程序逻辑开始的地方，因此是入口文件，而被引用的 `a` 和 `b` 则是由 `main.js` 引入和调用，因此不是入口文件。

一般而言，我们会将一些逻辑提取封装后放到独立的文件中，最后由入口文件引入来调用它们提供的方法完成整个程序逻辑。也就是一般入口文件关注整体流程，而非入口文件关注公共某一部分的实现。

非模块化文件打包

为了演示webpack的强大，我们将首先演示非模块化文件打包。

Demo

首先准备一个HTML（ `example1.1.html` ）：

```
<html>
  <head>
    <title>webpack.toobug.net</title>
    <script src="./bundle1.1.js"></script>
  </head>
  <body>
  </body>
</html>
```

前面解释过，webpack是一个开发时进行打包的工具，因此我们需要准备两份文件，一份是用于开发维护的源码，一份则是由webpack打包生成的文件。

在这个示例中，我们关注源码就好。如果是放到实际项目中，则也需要仔细规划打包后文件的路径。

首先我们使用非模块化的方案，准备我们的源文件 `example1.1.js`：

```
alert('hello world');
```

接下来使用webpack将它打包：

```
cd {index.html所在目录}
webpack example1.1.js bundle1.1.js
```

此时就可以看到目录下生成了一个 `bundle1.1.js`，正是我们在html中引用的JS文件。

访问一下 `example1.1.html`，则可以看到我们的弹出框。



解析

看到这里，你可能会开始怀疑人生了：你真的不是在逗我玩吗？这么简单一个demo，我直接引用源文件不就好了？干嘛要用webpack打包，多此一举？

这个问题，我也无法回答你，非模块化的文件被模块化打包工具支持，本来就是一件很神奇的事情。至于初衷，我也只能想到是为了让开发者能更低成本地迁移到webpack上来。

不过，为了保证我真的没有忽悠你，我们还是可以看一看打包之后的文件：<https://github.com/TooBug/webpack-guide/blob/master/examples/chapter2/non-moduler/bundle1.1.js>。可以看到，一个原本1行的JS文件被打包成了一个50行的文件。而这剩下的49行到底是什么呢？我们简单解析一下：

整体上看，文件被分为两个部分：第1至41行，是一个函数定义，这也就是官方文档中提到的**Runtime**，作用是保证模块的顺利加载和运行。第45至49行则是我们原来写的JS文件，只是被包裹了一个函数，也就是模块。运行的时候模块是作为Runtime的参数被传进去的，也就是这样的形式：

```
(function(modules){  
  // Runtime  
})([  
  // 模块数组  
])
```

这里面特别值得注意的一点是第20行和第45行。

```
// 第20行  
modules[moduleId].call(module.exports, module, module.exports, __webpack_require__);  
  
// 第45行  
function(module, exports) {
```

其中第45行是模块被包裹的函数，给了 module 和 exports 参数。（如果有需要的话，还会给出 require 参数。）而这两个参数本质上都是Runtime中定义的 module.exports，是一个空对象（第14行）。这样就实现了使用Commonjs的方式来定义模块返回值。不过由于我们还在讨论非模块化的文件，就不深入。

真正值得注意的是第20行，使用了 .call，第一个参数是 module.exports，这就导致模块的包裹函数的作用域中的 this 指向了 module.exports。这会带来两个后果：

1. 模块中无法使用 this 指代全局变量（浏览器中就是 window）
2. 模块中可以使用 this.xxx 来指定模块包含的成员，类似Common.js中 exports.xxx 的方式（感觉我们找到了除AMD/Common.js之外的另一种模块化规范，不过因为webpack官方并没有强调这个，我们也只是代过。）

影响

当然，聪明的你肯定早就意识到了另一个更明显的结果，即模块不是暴露在全局作用域下了。也即通过 var xxx 的方式定义的 xxx 变量不再挂在全局对象下。这可能是在非模块化的代码迁移到webpack中碰到的最大的问题，需要手工将 var xxx 的定义方式改为 window.xxx。

同样，由于模块源码是采用非模块化的方案编写的，因此没有通过AMD的 return 或者CommonJS的 exports 或者 this 导出模块本身，这会导致模块被引入的时候只能执行代码而无法将模块引入后赋值给其它模块使用。

例如上面的 example1.1.js，当我们引入的时候（以CommonJS为例），var a = require('example1.1');，此时 a 为 undefined。

你可能觉得这样好像没什么意义是吧？但是事实上大量的模块是用这种方式编写的，包括著名的Angular.js（1.4以下），这会导致无法直接使用 var angular = require('angular') 来引入 angular，需要通过额外的方式来做（exports-loader），后面详述。

AMD模块打包

初试牛刀

webpack对AMD提供了比较完善的支持。我们同样以一个例子开始：

example1.1.html：

```
<html>
<head>
  <title>webpack.toobug.net</title>
  <script src="./bundle1.1.js"></script>
</head>
<body>
</body>
</html>
```

example1.1.js：

```
define([
],function(){
  alert('hello world!');
});
```

同样使用webpack进行编译，用法和前文使用的完全一样：

```
webpack example1.1.js bundle1.1.js
```

同样打开HTML文件，同样弹出“hello world!”的弹窗。（这里就不截图了。）

解析

我们同样打开编译后的文件分析一下，请用力点击<https://github.com/TooBug/webpack-guide/blob/master/examples/chapter2/amd/bundle1.1.js>。

可以看到，编译（就是打包的意思，后文可能会乱入，请习惯）后的文件有53行（非模块化的例子中是50行），结构仍然是一个Runtime + 模块数组，而且1到41行和前面非模块化的例子是完全一样的。

第45行即第一个模块的包裹函数，与前面非模块化的例子相比，多了一个 `__webpack_require__` 参数（也就是前一节说的，如果有必要的话，还有 `require` 参数的意思）。如果在源文件中有使用 `require` 引用其它模块的话，那么使用 `require` 的地方经过编译之后都会变成 `__webpack_require__`。关于这一点，后面会涉及到的时候会有更详细的论述，此处就不再深入。

第45至52行之间即是模块本身了，当然也包含了包裹函数。如果智商正常的话，应该还是会觉得这个模块有点鬼斧神工吧？哦，也就是晦涩难懂的意思。我们稍微整理一下：

```
function(module, exports, __webpack_require__) {

    var __WEBPACK_AMD_DEFINE_ARRAY__, // AMD依赖列表
        __WEBPACK_AMD_DEFINE_RESULT__; // AMD factory函数的返回值，即模块内容

    __WEBPACK_AMD_DEFINE_ARRAY__ = [];

    // 执行factory函数，获取返回值作为模块内容
    // 函数体使用.apply调用，函数体中this为exports
    // 形参则分别对应依赖列表中的各个依赖模块
    __WEBPACK_AMD_DEFINE_RESULT__ = function(){
        alert('hello world!');
    }.apply(exports, __WEBPACK_AMD_DEFINE_ARRAY__);

    // 如果模块内容不为空，则通过module.exports返回
    // 如果为空，则不处理，在Runtime中声明为{}
    if(__WEBPACK_AMD_DEFINE_RESULT__ !== undefined){
        module.exports = __WEBPACK_AMD_DEFINE_RESULT__;
    }

}
```

可以看到，尽管我们的例子中并没有声明什么依赖，但webpack还是为AMD模块做了很多准备工作，比如获取依赖模块内容并传给factory函数之类。主要逻辑在上面代码的例子中已经注释，就不再多解释。

值得注意的是，在模块的最后，仍然是通过 `module.exports` 导出模块内容的，而Runtime中则不管是否AMD都完全一样。这种在不同模块化方案下打包后结构高度统一的方案为webpack带来了诸多好处，因为webpack只需要在模块入口导出这一块处理好不同模块化的差异即可，对后续的模块加载和优化等等都打下了坚实的基础。当然，目前可能还感觉不到这些好处，但是到目前为止，你应该可以非常直观地感觉到，webpack的确能同时支持多种模块化方案混合使用。

为何webpack能同时使用多种模块化（及非模块化）方案的模块而传统的require.js / sea.js之类的方案则不行？这是因为webpack对模块的处理是在编译阶段，针对每一个模块都可以针对性地分析，然后对不同方案加以不同的包裹方案，最后生成可供浏览器运行的代码。而require.js之类的方案必须保证在运行时阶段能正确地引入模块并运行。

以require.js为例，在不做包裹的情况下，require.js完全无法感知非模块化JS文件的运行状态和结果，因此无法纳入其管理体系。

此外，require.js必须在全局定义 `require` 和 `define` 方法，而如果此时使用 `<script>` 引入使用UMD定义的模块，则无法正常工作，因为模块会认为当前是AMD环境，而AMD模块无法使用 `<script>` 直接引入。关于这一点，后续在讲解jQuery的时候会有详细说明。

AMD依赖

在上一节中，我们并没有涉及到“依赖声明”这件事情，因为在非模块化的写法下，并不存在依赖声明这回事，如果文件B依赖文件A，则需要手工在HTML中引用A和B，而且必须保证A在B之前引入。

而AMD的诞生初衷之一就是为了解决依赖声明，因此本例中我们也会涉及到依赖声明。

HTML的文件和上例一样，不再重复。本例中的JS文件则变成了两个，一个入口文件 `example2.1.js`，一个被依赖的模块 `example2.2.js`。

`example2.1.js`：

```
define([
    './example2.2'
], function(example2){
    example2.sayHello();
});
```

`example2.2.js`：

```
define([
], function(){
    return {
        sayHello: function(){
            alert('hello world!');
        }
    };
});
```

同样使用webpack打包，只需要指定入口文件即可，webpack会处理好依赖：

```
webpack example2.1.js bundle2.1.js
```

同样的打开HTML文件，同样的“hello world!”，同样的不再截图，也同样的，应该有一段解析。

没错，看这里<https://github.com/TooBug/webpack-guide/blob/master/examples/chapter2/amd/bundle2.1.js>。

同样的前41行是一样的。而这次模块列表数组的部分有了两个模块。这里终于可以说说我们之前一直刻意忽略的第44行了，在这个例子中相似的还有54行。这两行注释了一个数字，代表的是模块的索引，换个更专业的说法，则是（编译后的）“模块ID”。

下面请集中注意力，我们要跳几行代码了。首先请看第1行，形参为 `modules`，而我们前面已经说过，这个就是模块列表，也就是一个数组。再看第20行，从 `modules` 中访问了key为 `moduleId` 的模块，而由于 `modules` 为一个数组，所以这个 `moduleId` 自然就是数字了，正是数组的索引。

特别值得注意的是第40行，`return __webpack_require__(0);`，如果留心的话会发现我们前面的例子中编译出来的这一行全部都是引用写死模块ID `0`，也就是说，模块ID为0的永远是入口。

看完了Runtime之后我们再来看一下模块本身，模块ID为1的没什么好看的，和上面的例子一样，而模块ID为0的入口，和上例略有一点点不一样，即第48行。`__WEBPACK_AMD_DEFINE_ARRAY__` 这个变量中使用 `__webpack_require__(1)` 引入了依赖，对应源文件 `example2.1.js`，也即 `example1.1.js` 中声明的依赖。

至此，我们已经完全清楚了AMD模块的依赖、factory函数在webpack中是如何处理的了。

当然，如果你对AMD很熟悉的话，会发现我们漏掉了对 `define` 的第一个参数 `moduleId` 的讲解。由于AMD中模块ID与寻址直接相关，如果我们直接为模块指定一个模块ID的话，则默认情况下无法找到该依赖模块。这种情况下需要使用 `resolve.alias` 配置来解决。后文再详述。

CommonJS模块打包

经过前面非模块化和AMD打包的例子，本节我们就不再绕圈子，直接看例子。

HTML代码和之前仍然是一样的，不再重复，JS文件如下：

入口文件 `example1.1.js`：

```
var example2 = require('./example1.2');
example2.sayHello();
```

被依赖的 `example1.2.js`：

```
var me = {
  sayHello:function(){
    alert('hello world!');
  }
};
module.exports = me;
```

同样使用webpack编译就能看到相同的弹窗。

我们直接来看编译后的代码<https://github.com/TooBug/webpack-guide/blob/master/examples/chapter2/commonjs/bundle1.1.js>：

呃。是不是惊呆了？前41行Runtime仍然长得那个鸟样，而后面的两个模块代码，几乎和我们编写时完全一样！！唯一的改动是 `require` 被替换成了 `__webpack_require__` 了。

那就假装webpack是天然支持CommonJS的吧。所以通过这个例子也能感觉到，在使用webpack的项目中，使用CommonJS规范会省事很多。

Node和NPM

咦，明明在说前端代码，怎么突然要说Node和NPM？等等，哪里有说我在说前端代码？明明说的是JS代码。

好吧，前面的例子的确是在说前端代码，而且本指南全篇都将聚焦前端代码，但现在一个明显的趋势是前后端越来越趋向于融合，因此了解一些和Node/NPM的知识对于我们日常开发是有好处的。

上面的都是大道理，具体的事实大致有这些：

1. NPM已经不再是node package manager，而只是一个包管理软件，在NPM的愿景中是有前端代码这一块的
2. 越来越多的前端库也选择将代码发布到npm，包括jQuery / Angular / react等
3. 前端开发已经高度依赖于Node带来的生态端，比如各种各样的工具
4. 由于CommonJS在前端开发中应用的成熟，复用Node的代码成为可能（且越来越方便）

引用Node模块和NPM模块

既然webpack支持在前端使用CommonJS模块规范，那么是否意味着我们可以直接使用NPM模块，甚至是Node内置的模块呢？答案是肯定的，我们看一个例子：

首先准备同样的HTML和JS文件，不同的是，这次我们需要使用 `npm init` 准备一个 `package.json` 文件，因为安装npm依赖时需要用到。

```
npm init
```

接下来一顿回车，就生成了一个 `package.json`。然后安装一个npm模块 `cookie`，用于解析和生成cookies串。

```
npm install cookie --save
```

`--save` 会将依赖写入 `package.json`，下次直接使用 `npm install` 即可安装依赖。

接下来编写JS文件 `example1.1.js`：

```
var url = require('url');
var urlResult = url.parse('http://webpack.toobug.net/zh-cn/index.html');
console.log(urlResult);

var cookie = require('cookie');
var cookieResult = cookie.parse('name=toobug; subject=webpack;');
console.log(cookieResult);
```

在这段代码里，我们分别使用Node内置的模块 `url` 和NPM中的模块 `cookie` 来解析字符串并输出结果。我们将这段JS分别在Node和浏览器中（经webpack编译）运行，最终输出如图：

```
→ example1 git:(master) ✗ node example1.1.js
{ protocol: 'http:',
  slashes: true,
  auth: null,
  host: 'webpack.toobug.net',
  port: null,
  hostname: 'webpack.toobug.net',
  hash: null,
  search: null,
  query: null,
  pathname: '/zh-cn/index.html',
  path: '/zh-cn/index.html',
  href: 'http://webpack.toobug.net/zh-cn/index.html' }
```

Node.js输出截图


```
▼ Url {protocol: "http:", slashes: true, auth: null, host: "webpack.toobug.net", port: null...} ⓘ
  auth: null
  hash: null
  host: "webpack.toobug.net"
  hostname: "webpack.toobug.net"
  href: "http://webpack.toobug.net/zh-cn/index.html"
  path: "/zh-cn/index.html"
  pathname: "/zh-cn/index.html"
  port: null
  protocol: "http:"
  query: null
  search: null
  slashes: true
  ► __proto__: Url
```

浏览器输出截图

也就是说，我们可以直接在浏览器端复用Node和NPM的代码了！不知道此时的你是否会感到很兴奋？！这意味着我们的代码可以前后端复用了，而更重要的是，前端也可以使用NPM海量的模块了，就像上例中我们使用了 `cookie` 这个模块，我们手中的工具一下子变强大了好多倍！

Node模块和前端使用的CommonJS

Node使用的模块规范也是CommonJS，所以理想情况下，是可以做到代码在Node和浏览器中复用的。但这里面有几处关键的差异可能导致无法复用：

1. Node的模块在运行时会有一个包裹函数，下面详述
2. 浏览器并不支持所有的Node模块，因此部分使用了Node API的模块无法在浏览器中运行

首先我们来看一下Node模块在运行的时候真相是怎样的，假设我们有如下模块 `example1.1.js`：

```
var me = {};  
module.exports = me;
```

这是一个非常简单的Node模块。不知道大家在写Node模块的时候是否有好奇过，这里的 `module`（以及 `require` / `exports` / `__filename` / `__dirname`）是从哪里来的？因为按照对JS的认知，如果它不是一个局部变量（含函数形参）的话，那么只能是全局变量了。

难道这些变量是全局变量？然而当我们打开Node的命令行去访问的时候又明明白白地告诉我们是 `undefined`：

```
> console.log(global.exports)  
undefined  
  
> console.log(global.__dirname)  
undefined
```

按照Node的文档，`global.require` 确实是存在的，还有 `.cache` / `.resolve()` 等成员，但每个模块中使用的 `require` 仍然是局部变量，并非全局 `require`。

这到底是怎么回事呢？如果在运行的时候去查看它的话，它会变成这样：

```
(function (exports, require, module, __filename, __dirname) {  
  var me = {};  
  module.exports = me;  
});
```

可以使用 `node-inspector` 去远程调试Node代码，于是就能看到模块运行时的真实代码。

可以看到，我们的模块代码在运行的时候是被包裹了一层的，而上面列的这些变量正是在这个包裹函数中作为形参传入的。

其中 `module` 指向模块本身，`module.exports` 和 `exports` 是等价的，表示模块要导出供调用的内容，`__filename` 表示当前模块的文件名，`__dirname` 表示当前模块所在路径。

因此，并不是所有的Node模块都能为浏览器所用。尽管如此，webpack也还是为这些包裹函数带来的新变量（形参）提供了模拟，可以在浏览器端使用的代码中注入这些变量。由于使用得少，而且该选项在CLI中并没有提供，需要另配config文件，故此处不再演示。

UMD模块打包

按理说，分别讲完非模块化、AMD、CommonJS的打包后，并没有必要再专门引入一篇讲UMD打包的。但因为UMD的实现中依赖一些特殊的变量，因此还是提一下。

首先回顾一下UMD的模块定义：

```
(function (root, factory) {
  if (typeof define === 'function' && define.amd) {
    // AMD. Register as an anonymous module.
    define(['b'], factory);
  } else if (typeof module === 'object' && module.exports) {
    // Node. Does not work with strict CommonJS, but
    // only CommonJS-like environments that support module.exports,
    // like Node.
    module.exports = factory(require('b'));
  } else {
    // Browser globals (root is window)
    root.returnExports = factory(root.b);
  }
})(this, function (b) {
  //use b in some fashion.

  // Just return a value to define the module export.
  // This example returns an object, but the module
  // can return a function as the exported value.
  return {};
}));
```

上面的代码来自<https://github.com/umdjs/umd/blob/master/returnExports.js>。我们按这样的方式来定义一个模块 `example1.1.js`（去掉了对 `b` 的依赖），看看webpack会如何处理。

打包后的代码见<https://github.com/TooBug/webpack-guide/blob/master/examples/chapter2/umd/bundle1.1.js>，可以看到，使用UMD声明的模块会默认按照AMD的方式去打包。而核心的一句就是第48行，原本是

```
if (typeof define === 'function' && define.amd) {
```

被webpack直接替换成了

```
if (true) {
```

可见webpack还是相当聪明的！既满足了AMD的判断条件，又没有真的在全局注入 `define` 变量，这样就不会像`require.js`一样，一旦引入就无法再使用 `<script>` 引入UMD脚本。

如果你希望webpack不要使用AMD的方式引入，而是使用CommonJS的方式的话，则需要指定让webpack不按AMD的方式去解析，具体方法则是使用 `imports-loader`。

关于loader，后面会有详细解释，这里只说怎么用。首先 `npm init` 然后 `npm install imports-loader --save` 安装依赖，接下来在打包时加一些参数：

```
webpack example1.1.js bundle1.2.js --module-bind 'js=imports?define=>false'
```

意思是针对 `.js` 文件，使用 `imports-loader`，传给loader的参数是 `define=>false`，即定义 `define` 为 `false`，这样模块就不会使用AMD的方式打包了。`imports-loader`的参数具体含义可参见[文档](#)。

打包后的代码见<https://github.com/TooBug/webpack-guide/blob/master/examples/chapter2/umd/bundle1.2.js>，如我们所愿，AMD的条件判断不再是 `true`。因为没有 `define` 变量，又因为在模块的包裹函数中传了 `module` 且 `module.exports` 存在，所以最终这个模块会按CommonJS的方式被使用。

第三章 webpack进阶

在第二章，我们看到了webpack如何为各种不同的模块化方案编写的模块进行打包。如果webpack的开发者并不是一个负责的开发者的话，那么有可能故事就到此为止了。然而webpack之所以能红遍大江南北，正是因为它给我们带来了太多惊喜。在本章，我们将对这些进阶技巧逐一展开。

首先，我们会大概分析一下webpack的配置文件及其用法，然后会讲解webpack是如何处理开发过程中碰到的各种实际问题的。希望在看完本章之后，你能自豪地说“我掌握webpack了”。

CLI与API使用模式

在进入配置项讲解之前，我们首先看一看webpack为我们提供的使用方式：CLI和API。

CLI

CLI即Command Line Interface，顾名思义，也就是命令行用户界面。到目前为止，我们所有的示例都是以这种方式调用的，例如：

```
webpack example1.1.js bundle1.1.js
```

如果我们直接执行 `webpack`，不加任何参数（且当前目录不存在配置文件），则会显示webpack的帮助信息，里面有非常多的参数可用。事实上，除了我们前面这种出于演示的目的直接在命令行中写参数之外，大部分生产环境下使用时都会需要加上非常多的参数，导致整个命令非常长，既不利于记忆编写，也不利于传播交接等。因此，一般会将配置项写在同目录的 `webpack.config.js` 中，然后执行 `webpack` 即可，webpack会从该配置文件中读取参数，此时不需要在命令行中传入任何参数。

```
# 执行时webpack会去寻找当前目录下的webpack.config.js当作配置文件使用
webpack

# 也可以用参数-c指定配置文件
webpack -c mycofnig.js
```

配置文件 `webpack.config.js` 的写法则是：

```
module.exports = {
  // 配置项
};
```

值得注意的是，配置文件是一个真正的JS文件，因此配置项只要是一个对象即可，并不要求是JSON。也就意味着你可以使用表达式，也可以进行动态计算，或者甚至使用继承的方式生成配置项。

API

API则是指将webpack作为Node.js模块使用，例如：

```
webpack({
  // 配置项
  entry:'main.js',
  ...
},callback);
```

到目前为止，我们所有的示例都还没有用到这种用法。但这种用法并不少见，尤其是在和构建工具（如Gulp）搭配使用的时候，都是使用的这种方式。你也可以自己写一段程序来调用webpack。相比CLI模式而言，使用API模式会更加灵活，因为可以与你的各种工具进行集成，甚至共享一些环境变量然后动态生成打包配置等等，在复杂项目中会很有用。

同理，API模式中的配置项对象也是一个真正的对象。

基本配置项

entry 和 output

首先我们写一个简单的配置文件，将前面我们用到的命令行参数放到配置文件中来。

webpack.config.js ：

```
module.exports = {
  entry: './example1.1',
  output: {
    filename: 'bundle1.1.js'
  }
};
```

示例文件仍然使用CommonJS打包示例中的JS文件：

```
module.exports = {
  sayHello: function() {
    alert('Hello World!');
  }
};
```

然后直接使用 webpack 打包即可：

```
webpack
```

输出

```
Hash: 0cf5b5109a8f4bf34ae5
Version: webpack 1.12.2
Time: 70ms

   Asset      Size  Chunks             Chunk Names
bundle1.1.js  1.48 kB       0  [emitted]  main
   [0] ./example1.1.js 83 bytes {0} [built]
```

同时生成了 bundle1.1.js，可见配置文件的确是生效了的。代码见<https://github.com/TooBug/webpack-guide/blob/master/examples/chapter3/config/example1>。

output.filename 中的占位符

output.filename 除了可以指定具体的文件名以外，还可以使用一些占位符，包括：

- name 模块名称
- hash 模块编译后的（整体）Hash值
- chunkhash 分片的Hash值

使用的方式就是在 output.filename 的中使用 [name].js 或者 my-[name]-[hash].js 之类的，一看就明白。但这三个值具体是从哪里来的呢？

首先，我们一次有可能要打包很多模块，而不止是上例中那样只有一个 example1.1.js，因此会碰到支持多个入口文件（entry）的情况，每一个入口都需要有自己的名字，具体对应 entry 的写法而言，有如下几种情况：

```
entry: './example2.1'
// 或者
entry: ['./example2.1', './example2.2']
```

这种情况下，模块是没有名字的，webpack会使用 `main` 作为模块名字，因此像下面这种用数组来指定入口的情况，模块名会重复，而此时webpack会将它们的代码合并打包！

```
Hash: 1af82371840be6a233d2
Version: webpack 1.12.2
Time: 101ms

   Asset      Size  Chunks             Chunk Names
main.js  1.76 kB      0  [emitted]  main
    [0] multi main 40 bytes {0} [built]
    [1] ./example2.1.js 83 bytes {0} [built]
    [2] ./example2.2.js 84 bytes {0} [built]
```

注意上方的 `multi main` 字样，表示有多个模块名为 `main`。这种情况下的代码见<https://github.com/TooBug/webpack-guide/blob/master/examples/chapter3/config/example2>，里面包含了使用 `[name]` 和 `[hash]` 打包出来的代码，均包含了 `example2.1` 和 `example2.2` 的代码。

另一种是webpack比较推荐的多入口写法：

```
entry:{
  'example2.1':'example2.1.js',
  'example2.2':'example2.2.js'
}
```

这种写法中，名字和模块文件名一一对应，每个模块都有独立的名字。因此这里的 `[name]` 可以理解成模块名字。

事实上，在webpack的文档中，这个name是指“chunk name”，即分片的名字，这里需要先剧透一下后面要说的“分片”的概念。所谓分片就是指一个入口模块的代码有可能会被分成多个文件，还有一些文件可能是来自模块的公共代码，而不是入口模块。因此这里的 `[name]` 并非严格与入口模块一一对应。

了解了这些情况之后，`[hash]` 和 `[chunkhash]` 就自然好理解了，一个是指本次打包相关的整体的hash，一个是指分片的hash。

看示例：

```
module.exports = {
  entry:{
    'example3.1':'./example3.1',
    'example3.2':'./example3.2'
  },
  output:{
    //这里分别用hash和chunkhash，结果不一样
    filename:'[name]-[hash].js'
    //filename:'[name]-[chunkhash].js'
  }
};
```

为了让两个模块不一样，我们将 `example3.1` 和 `example3.2` 中 `alert` 的内容做了修改，保证它们的内容是不一样的。用 `[hash]` 时打包的结果：

```
Hash: 6f17d6321f8580500bc9
Version: webpack 1.12.2
Time: 73ms

   Asset      Size  Chunks             Chunk Names
example3.1-6f17d6321f8580500bc9.js  1.48 kB      0  [emitted]  example3.1
example3.2-6f17d6321f8580500bc9.js  1.48 kB      1  [emitted]  example3.2
    [0] ./example3.1.js 88 bytes {0} [built]
    [0] ./example3.2.js 88 bytes {1} [built]
```

可以看到两个文件hash值是一样的，这就是所谓整体hash的意思（官方文档叫作 `bundle` 的hash值）。

用 `[chunkhash]` 时打包的结果：


```
Hash: 87ac4d29062084750a92
Version: webpack 1.12.2
Time: 75ms

      Asset      Size  Chunks             Chunk Names
example3.1-ef6b40ba3b9335fc2551.js  1.48 kB      0  [emitted]  example3.1
example3.2-b92fc07f9784897342c5.js  1.48 kB      1  [emitted]  example3.2
   [0] ./example3.1.js 88 bytes {0} [built]
   [0] ./example3.2.js 88 bytes {1} [built]
```

两个文件hash值是不同的，也就是每一个分片的hash都不一样。

代码见<https://github.com/TooBug/webpack-guide/blob/master/examples/chapter3/config/example3>。

output.path

有时候我们希望输出的文件不在当前目录（其实大部分时候都是这样），比如源码在 `src` 目录，输出的文件在 `dist` 目录，此时就需要用到 `output.path` 来指定输出路径。

`output.path` 也可以使用占位符。

```
entry:{
  'example4.1': 'src/example4.1'
},
output:{
  filename: '[name].js',
  path: './dist'
}
```

你肯定能猜到，文件会打包到 `dist/example4.1.js`，这并没有什么好惊奇的，对，我是说没什么好演示的。

真正值得注意的是，如果你的模块是存放在子目录中的，而你又想保持这种目录结构到打包后的 `dist` 中，怎么办？没听懂是吧，就是这种情况：

```
src/
  example4.1.js
  hello/
    example4.2.js
```

希望打包之后是这样：

```
dist/
  example4.1.js
  hello/
    example4.2.js
```

这种情况下，子目录并不能由 `output.path` 配置而来，而应该将目录写到模块名上，配置文件变成这样：

```
entry:{
  'example4.1': './src/example4.1',
  'hello/example4.2': './src/hello/example4.2'
},
output:{
  filename: '[name].js',
  path: './dist'
}
```

注意这里的 `filename` 一定要包含 `[name]` 才行，因为路径信息是带在模块名上的。

代码见<https://github.com/TooBug/webpack-guide/blob/master/examples/chapter3/config/example4>。

分片

状态：草稿

随着项目开发过程中越来越大，我们的代码体积也会越来越大，而将所有的脚本都打包到同一个JS文件中显然会带来性能方面的问题（无法并发，首次加载时间过长等）。

webpack也提供了代码分片机制，使我们能够将代码拆分后进行异步加载。

值得注意的是，webpack对代码拆分的定位仅仅是为了解决文件过大，无法并发加载，加载时间过长等问题，并不包括公共代码提取和复用的功能。对公共代码的提取将由CommonChunks插件来完成。

要使用webpack的分片功能，首先需要定义“分割点”，即代码从哪里分割成两个文件。

分割点

分割点表示代码在此处被分割成两个独立的文件。具体的方式有两种。

使用 `require.ensure`：

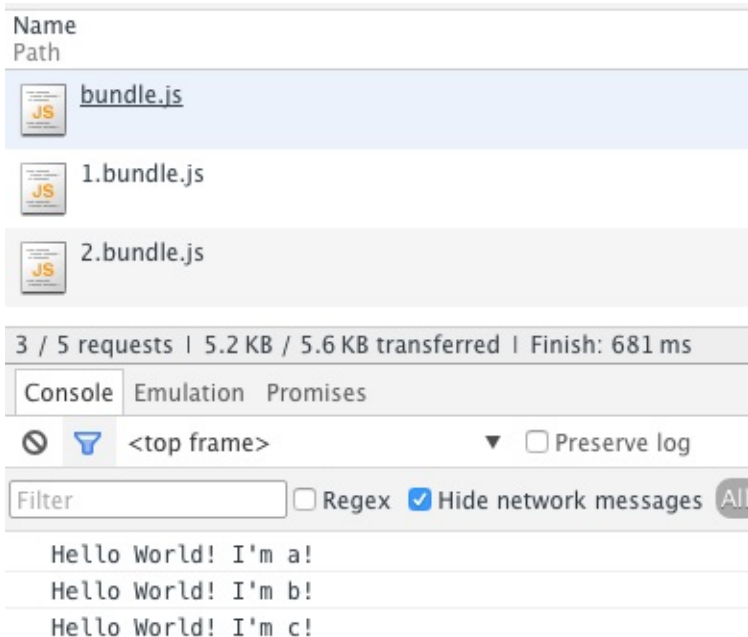
```
require.ensure(["module-a", "module-b"], function(require) {  
  var a = require("module-a");  
  // ...  
});
```

使用AMD的动态 `require`：

```
require(["module-a", "module-b"], function(a, b) {  
  // ...  
});
```

上面的例子中，`module-a` 和 `module-b` 就会被分割到独立的文件中去，而不会和入口文件打包在同一个文件中。

TODO：module-a和module-b何时会在同一个文件，何时不会在同一个文件？



example1中使用了 `require.ensure` 和AMD动态 `require` 两种方式，来建立分割点，代码在此处被分片。

```
var a=require('./a');
a.sayHello();

require.ensure(['./b'], function(require){
  var b = require('./b');
  b.sayHello();
});

require(['./c'], function(c){
  c.sayHello();
});
```

打包后的代码：

- bundle.js -> main.js + a.js
- 1.bundle.js -> b.js
- 2.bundle.js -> c.js

多入口

多入口的情况下：

- 入口1 bundle.main1.js -> main.js + a.js
- 入口2 bundle.main2.js -> main2.js + a.js
- 1.bundle.1.js -> b.js
- 2.bundle.2.js -> c.js

可见公共代码a.js并没有被提取出来。

因此分片只是分片，并没有自动提取公共模块的作用。

CommonChunks插件

问题

顾名思义，Common Chunks 插件的作用就是提取代码中的公共模块，然后将公共模块打包到一个独立的文件中，以便在其它的入口和模块中使用。

main.js :

```
var a=require('./a');
a.sayHello();

var b = require('./b');
b.sayHello();

var c = require('./c');
c.sayHello();
```

main.2.js :

```
var a=require('./a');
a.sayHello();

var b = require('./b');
b.sayHello();
```

a 、 b 、 c 和之前一样，只有一个 sayHello() 方法。

打包后看到 bundle.main1.js 和 bundle.main2.js 中分别包含了 a 、 b 、 c 三个模块。其中的 a 和 b 正是我们要使用 CommonChunksLoader提取出来的公共模块。

小试牛刀

接下来，我们在配置项中添加CommonChunksLoader的配置，使用它来提取公共模块。

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

module.exports = {
  entry:{
    main1: './main',
    main2: './main.2'
  },
  output:{
    filename: 'bundle.[name].js'
  },
  plugins: [
    new webpack.optimize.CommonsChunkPlugin('common.js', ['main1', 'main2'])
  ]
};
```

注意第1行，添加了webpack的引用（同时也要在项目目录下安装webpack），然后添加了 plugins 选项，引用了 webpack.optimize.CommonsChunkPlugin 来提取公共模块，参数 common.js 表示公共模块的文件名，后面的数组元素与 entry 一一对应，表示要提取这些模块中的公共模块。

重新使用 webpack 打包后，看到生成的文件中多了一个 common.js :

webpack

Hash: 2eaa6808a94e7ed42693

Version: webpack 1.12.8

Time: 99ms

	Asset	Size	Chunks	Chunk Names
bundle.main1.js	417 bytes	0 [emitted]	main1	
bundle.main2.js	193 bytes	1 [emitted]	main2	
common.js	3.9 kB	2 [emitted]	common.js	
[0]	./main.js	117 bytes {0} [built]		
[0]	./main.2.js	77 bytes {1} [built]		
[1]	./a.js	97 bytes {2} [built]		
[2]	./b.js	97 bytes {2} [built]		
[3]	./c.js	97 bytes {0} [built]		

这个 `common.js` 正是公共部分 `a` 和 `b` 两个模块。而生成的 `bundle.main1.js` 中只包含了 `c` 模块，`bundle.main2.js` 中则没有包含任何其它模块。生成后的代码可参见<https://github.com/TooBug/webpack-guide/blob/master/examples/chapter3/common-chunks-plugin>。

当然，还有一步不能少，就是在HTML中加入公共部分 `common.js`：

```
<script src="./common.js"></script>
<script src="./bundle.main1.js"></script>
```

运行时截图如下：



另一个问题

上面的代码成功地将公共模块 `a` 和 `b` 提取出来了。但不知道看到此处，你的心里是否会有点疑问。

如果你还没有的话，我们一起来看一下下面这些东东：

1. 什么是入口文件？如果你还记得前面我们给出的定义的话，就会发现，这个地方有两个入口文件！
2. 模块化方案应该允许多个入口文件吗？应该吗？不应该吗？应该吗？不应该吗？
3. 如果模块规划调整了，`common.js` 消失了，或者增加了一个 `common.2.js`，我应该修改每一个HTML引入的部分吗？

其实写到这里，我是有点迷茫的，按照我对模块化方案的认知，我的观点是：

1. 入口文件应该只有一个，其它的逻辑全部由入口文件处理，因此不应该允许多入口文件
2. 模块规划应该在模块化方案内部完成，不应该还需要时时调整模块调用入口

高级配置项

状态：草稿

output.publicPath

一般情况下，用上面的配置就能搞定了。但如果你的网站稍大一点，可能需要引入CDN，而且很可能CDN还有一些很古怪的前缀，这个时候可以通过 `output.publicPath` 来搞定。

例如，在前面的例子中，输出的脚本路径是 `dist/example4.1.js`，而在生产环境中访问的时候却有可能是 `http://cdn.toobug.net/scripts/webpack_guide/dist/example4.1.js`。这种情况下，就需要使用 `output.publicPath` 来将前面的路径补全：

```
output:{
  publicPath:'http://cdn.toobug.net/scripts/webpack_guide/'
}
```

这样发布到生产环境以后，就会到CDN上对应的路径去加载脚本了。

这个选项适用于各种非入口文件的场景，包括分片后的文件、loader加载文件时的路径、css中引入的图片资源文件路径等等。

output.library 、 output.libraryTarget 和 output.umdNamedDefine

如我们在前面很多例子中看到的那样，webpack在打包的时候会将 `entry` 中配置的文件打包成一个在浏览器中直接运行的文件，也即我们前面定义过的“入口文件”。而如果我们想要的并不是入口文件，而是一个可供其它脚本再引用的类库要怎么办呢？此时就可以用 `output.library` 和 `output.libraryTarget` 来告诉webpack，我们想要的打包结果是一个类库。

todo:介绍一下入口文件和类库一般有什么区别

首先看 `output.library`，它被用来指定模块的名字，就像jQuery，它的模块名叫 `jquery`，至于这个名字有什么用，马上就会看到。我们在后文中都以`{Library}`来代替它的值。

`output.libraryTarget`，它的意思是指定打包后的脚本如何导出模块。前面我们介绍过模块化规范，有AMD、CommonJS、UMD以及直接定义变量等等。这个选项即是指定用哪种方式来对模块内容进行导出。

它的可选项如下：

- 'var' 通过 `var {Library} = xxx;` 的方式导出模块，即定义（隐性）全局变量
- 'this' 通过 `this[{Library}] = xxx;` 的方式导出模块，其中this一般指向window，也是定义全局变量
- 'commonjs' 通过指定 `exports` 属性的方式导出模块，例如 `exports[{Library}] = xxx;`，引用的时候需要 `require('xxx.js')[{Library}]` 才能引用到 `xxx`。
- 'commonjs2' 通过指定 `module.exports` 的方式导出模块，例如 `module.exports = xxx;`，引用的时候直接使用 `require('xxx.js')` 即可引用到 `xxx`。

使用loader

在webpack中，可以使用 `require('./a')` 的方式来引入 `a.js`，如果你是从前文一路看过来的话会发现这并没有什么新奇的。那loader是做什么的呢？

loader是webpack中一个重要的概念，它是指用来将一段代码转换成另一段代码的webpack插件。晕了没？为什么需要将一段代码转换成另一段代码呢？这是因为webpack实际上只能处理JS文件，如果需要使用一些非JS文件（比如CoffeeScript），就需要将它转换成JS再 `require` 进来。当然，这个代码转换的过程能做的远不止是Coffee->JS这么简单，稍后将看到它的无穷魅力。

虽然本质上说，loader也是插件，但因为webpack的体系中还有一个专门的名词就叫插件（plugins），为避免混淆，后面不再将loader与插件混淆说，后文中这将是两个相互独立的概念。

用途

webpack处理的主要对象是JS文件，而我们知道JS文件是可以做很多事情的，比如编译Less/SASS/CoffeeScript，比如在页面中插入一段HTML，比如修改替换文本文件等等。既然如此，我们能不能将这些能力集中到webpack上来呢，比如我在 `require('a.coffee')` 的时候，webpack自动帮我把它编译成JS文件，再当成JS文件引入进来？或者更极端一点，`require('a.css')` 的时候，直接将CSS变成一段JS，用这段JS将样式插入DOM中呢？

这些，正是loader要做的事情。

用法

为了言之有物，以下皆以 `coffee-loader` 为例来说明。

coffee就是指coffee script啦，可以算是JS的一种方言，深受ruby/python党喜爱。虽然笔者并不使用coffee，也不提倡使用，但这个loader相对比较典型，容易理解。在生产环境中，更提倡使用ES6编写代码，然后使用babel-loader编译成ES5代码使用，但因为babel 6改成了插件式的，配置项略多，为了简单起见，先不使用这个。

`coffee-loader` 的作用就是在引用 `.coffee` 文件时，自动转换成JS文件，这样可以省去额外的将 `.coffee` 变成 `.js` 的过程。

首先我们准备一个 `example1.1.js`：

```
var hello = require('coffee!./example1.2.coffee');
hello.sayHello();
```

值得注意的是这里在 `require` 参数最前面加了 `coffee!`，这个表示使用 `coffee-loader` 来处理文件内容。详细的机制稍后说明。

接下来，准备 `example1.2.coffee`：

```
class Hello
  constructor: (@name) ->

  sayHello: () ->
    alert "hello #{@name}"

module.exports = new Hello 'world'
```

接下来，我们需要安装 `coffee-loader`。webpack的每一个loader命名都是 `xxx-loader`，在安装的时候需要将对应的loader装到项目目录下：

```
npm install coffee-loader --save
```


然后编译：

```
webpack example1.1.js bundle.js
```

打HTML，即可看到我们写的代码生效了。



example1.2.coffee 编译后的代码也可以拿出来看一下：

```
function(module, exports) {  
  
  var Hello;  
  
  Hello = (function() {  
    function Hello(name) {  
      this.name = name;  
    }  
  
    Hello.prototype.sayHello = function() {  
      return alert("hello " + this.name);  
    };  
  
    return Hello;  
  })();  
  
  module.exports = new Hello('world');  
  
  /**/  
}
```

完整的代码见<https://github.com/TooBug/webpack-guide/tree/master/examples/chapter4/using-loaders/example1>

进阶

知道了loader的作用之后，可以再来多看一看loader的用法了。

首先，除了npm安装模块的时候以外，在任何场景下，loader名字都是可以简写的，coffee-loader 和 coffee 是等价的，这意味着 require('coffee!./a.coffee') 和 require('coffee-loader!./a.coffee') 是等价的。这一点同样适用于下面讲到的其它用法，不再单独说明。

串联

loader是可以串联使用的，也就是说，一个文件可以先经过A-loader再经过B-loader最后再经过C-loader处理。而在经过所有的loader处理之前，webpack会先取到文件内容交给第一个loader。以我们后面经常会涉及的一个例子说明：

```
require('style!css!./style.css');
```

这个例子的意思是将 `style.css` 文件内容先经过 `css-loader` 处理（路径处理、`import` 处理等），然后经过 `style-loader` 处理（包装成JS文件，运行的时候直接将样式插入DOM中。）

参数

`loader`还可以接受参数，不同的参数可以让`loader`有不同的行为（前提是`loader`确实支持不同的行为），具体每个`loader`支持什么样的参数可以参考`loader`的文档。同样以 `coffee-loader` 为例，它可以指定一个名为 `literate` 的参数，意思是.....我也不知道，猜测可能是编译非完整的`coffee script`文件（例如内嵌在`markdown`中的代码片段等），如果你知道是什么意思，请告诉我。

在上面的用法中，参数的指定方式和`url`很像，要通过 `?` 来指定，例如指定 `literate` 参数需要这样写：

```
require('coffee?literate=1!./a.coffee');
```

这段代码中指定了 `literate` 参数为 `1`，如果不写 `=1` 的话，默认值为 `true`。（`coffee-loader` 官方并没有说可以为`1`，而是只写了 `literate`，没有值）。

loader使用方法

`loader`的使用有三种方法，分别是：

- 在 `require` 中显式指定，即上面看到的用法
- 在配置项（`webpack.config.js`）中指定
- 在命令行中指定

第一种我们已经看过了，不再说明。

第二种，在配置项中指定是最灵活的方式，它的指定方式是这样：

```
module: {
  // loaders是一个数组，每个元素都用来指定loader
  loaders: [{
    test: /\.jade$/,      //test值为正则表达式，当文件路径匹配时启用
    loader: 'jade',       //指定使用什么loader，可以用字符串，也可以用数组
    exclude: /regex/,     //可以使用exclude来排除一部分文件

    //可以使用query来指定参数，也可以在loader中用和require一样的用法指定参数，如`jade?p1=1`
    query: {
      p1: '1'
    }
  },
  {
    test: /\.css$/,
    loader: 'style!css'    //loader可以和require用法一样串联
  },
  {
    test: /\.css$/,
    loaders: ['style', 'css'] //也可以用数组指定loader
  }]
}
```

在命令行中指定参数的用法用得较少，可以这样写：

```
webpack --module-bind jade --module-bind 'css=style!css'
```

使用 `--module-bind` 指定`loader`，如果后缀和`loader`一样，直接写就好了，比如 `jade` 表示 `.jade` 文件用 `jade-loader` 处理，如果不一样，则需要显示指定，如 `css=style!css` 表示分别使用 `css-loader` 和 `style-loader` 处理 `.css` 文件。

更多

loader本质上做的是一个anything to JS的转换，因此想象空间极大，从目前[官方列出的loader列表](#)来看，大致有这样一些用途：

1. 其它语言编译到JS，包括JSON、纯文本、coffee、CSS等，也包括比较复杂的整体方案处理，比如 vue-loader 、 polymer-loader 等
2. “微处理”类，比如为非模块化文件添加一些模块化导入导出代码，对模块细节（代码）微调等，例如 exports-loader 、 expose-loader 等
3. 校验和压缩类，这一类其实最终并不生成代码，检验类如果报错就阻止构建进行，压缩类只是替换一下图片资源等
4. 纯打包类，比如 file-loader ，将资源文件一并纳入路径管理

具体的可以对照官方的loader列表一一查看。

bundle-loader

bundle-loader是一个用来在运行时异步加载模块的loader。

```
// 在require bundle时，浏览器会加载它
var waitForChunk = require("bundle!./file.js");

// 等待加载，在回调中使用
waitForChunk(function(file) {
  // 这里可以使用file，就像是用下面的代码require进来一样
  // var file = require("./file.js");
});

// todo：这句注释说的是？
// wraps the require in a require.ensure block
```

上面的例子中，在 `require` 时文件就会加载，如果你想在使用的时候再加载的话，可以这样：

```
var load = require("bundle?lazy!./file.js");

// 只有在调用load的时候才会真正加载
load(function(file) {

});
```

但前面我们已经在“分片”相关的内容中知道，将代码分片之后，本身就是异步加载的。为什么要使用bundle-loader呢？

这是因为webpack在进行分片的时候，会根据“分割点”，分片配置项等各种情况选择性地将各个模块打包在一起。例如：

```
require(["./f1", "./f2"], function(f1, f2) { /*...*/ });
```

在打包的时候，`f1` 和 `f2` 会被打包到同一个文件中（也可能还有其它的文件一起），这样加载的时候 `f1` 和 `f2` 是一起加载的。

而如果使用bundle-loader的话，则是独立的文件和独立的请求：

```
var f1 = require("bundle!./f1");
var f2 = require("bundle!./f2");
```

`f1` 和 `f2` 是独立打包和请求的。

这有什么意义呢？是的，如果 `f1` 和 `f2` 都是确定的，当然是打包到一起更好，能很好地减少浏览器发起的请求，加快页面加载速度。但如果模块名是动态的呢？

```
function loadPage(pageName, callback) {
  try {
    require(["./pages/" + pageName], function(page) {
      callback(null, page);
    });
  } catch(e) {
    callback(e);
  }
}
```

这种情况下webpack为了保证不管 `pageName` 传什么都能正确加载，会将 `./pages/` 目录下的所有文件打包到一个文件中，而很多时候，我们更希望每传一个 `pageName` 就动态加载一个文件，而不是全部加载。此时就可以用bundle-loader解决：

```
function loadPage(pageName, callback) {  
  try {  
    var pageBundle = require("bundle!./pages/" + pageName)  
  } catch(e) {  
    return callback(e);  
  }  
  pageBundle(function(page) { callback(null, page); })  
}
```

每个模块会被打包到一个独立的文件中，加载时也只加载一个模块的内容。

exports-loader

exports，导出的意思，顾名思义，这个**loader**是用来将（模块中的）某些内容导出的。之所以为“模块中的”加上括号，是因为实际上只要在模块中能被访问到的成员（变量）都可以被导出，当然也包括全局变量。

export导出在这里有特定含义，指将模块中的内容暴露出来，供使用方使用。例如CommonJS中的 `exports.xxx=xxx` 或者 `module.exports=xxx`，以及AMD中的 `return xxx` 都叫导出，导出后外部模块引用时就可以使用被导出的部分。没导出的部分将作为模块私有部分，外部无法访问。

导出全局变量

在实际使用中，用 **exports-loader** 最多的场景是将某些不支持模块化规范的模块所声明的全局变量作为模块内容导出。以 Angular.js 为例（1.3.14以下版本），它不支持模块化引用，因此如果直接使用以下代码引用 **angular** 是无效的：

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

此时，**angular** 仍然是一个挂载在 **window** 上的全局变量，而 **require** 返回的却是 **undefined**，导致上述 **angular** 变量为空。此时就需要我们使用 **exports-loader** 将全局的 **window.angular** 作为 **angular** 模块的返回值暴露出来。

看一个例子，在这个例子中，我们没有真的引用 **angular.js**，而是在 **example1.2.js** 中声明了一个全局变量 **HELLO**：

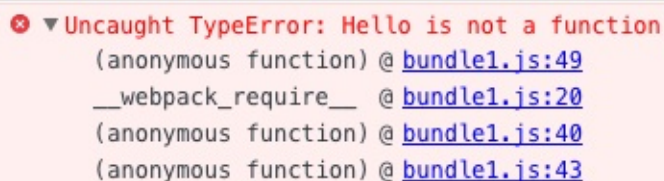
```
window.Hello = function(){
  console.log('hello from global Hello() function');
};
```

接下来，在 **example1.1.js** 中引用这个模块：

```
var Hello = require('./example1.2');

Hello();
```

然后直接使用**webpack**打包，生成 **bundle1.js**，运行截图如下：



```
Uncaught TypeError: Hello is not a function
    (anonymous function) @ bundle1.js:49
    __webpack_require__ @ bundle1.js:20
    (anonymous function) @ bundle1.js:40
    (anonymous function) @ bundle1.js:43
```

可以看到，由于 **example1.2** 并没有导出模块，导致 **example1.1** 在引用的时候报错，找不到 **Hello** 这个函数。

此时就轮到 **exports-loader** 上场了。我们建立一个 **webpack.config.js**：

```
module.exports = {
  module: {
    loaders: [
      { test: require.resolve('./example1.2'), loader: "exports?Hello" }
    ]
  }
};
```

其中的 **test** 表示使用 **loader** 的条件，这里使用了 **require.resolve** 解析出来的路径，表示只处理 **example1.2.js**，**loader** 中指定了 **exports-loader**，参数为 **Hello**，意为将 **Hello** 作为模块导出值。

重新打包生成 **bundle2.js**，运行截图如下：

```
hello from global Hello() function
```

可见 `Hello` 函数已被正确调用。

查看 `bundle2.js` 中的代码可以清楚地看到 `HELLO` 被导出的过程：

```
/**/ function(module, exports) {  
  
    window.Hello = function(){  
        console.log('hello from global Hello() function');  
    };  
  
    /** EXPORTS FROM exports-loader **/  
    module.exports = Hello;  
  
/**/ }
```

注：这里其实有两个示例，分别生成了 `bundle1.js` 和 `bundle2.js`，由于 `bundle1.js` 需要在 `webpack.config.js` 生效前创建，因此如果你克隆代码后自己运行的话，在生成 `bundle1.js` 前请将 `webpack.config.js` 改名，否则其中的配置会生效，最终效果和第二例一样。

导出局部变量

如前文所述，`exports-loader` 可以导出一切在模块内可以访问的变量，因此如果模块内定义了一个局部变量，也是可以导出的，方法和上面的例子几乎完全一样，这里就不再演示。

更多用法

上面例子中的配置文件是通过 `webpack.config.js` 来指定的，你也可以选择用其它方法来指定，详情见[使用loader](#)。

此外，上面例子中我们只导出了一个变量，此时该变量就是模块被引用时的值（通过 `module.exports` 导出）。事实上，`exports-loader` 还可以支持同时导出多个变量，例如 `exports?HELLO,WORLD` 会生成如下代码：

```
exports.HELLO = HELLO;  
exports.WORLD = WORLD;
```

此时模块的导出值是一个对象，其中的各个键值对我们指定的变量，引用时需要使用类似 `require('example1.2').HELLO` 的形式。

`exports-loader` 还支持变量名和导出名不同，例如 `exports?HELLO=obj.hello,WORLD=shijie` 会生成如下代码：

```
exports.HELLO = obj.hello;  
exports.WORLD = shijie;
```

一目了然，不再解释。

本文全部示例代码见<https://github.com/TooBug/webpack-guide/tree/master/examples/chapter4/exports-loader/example1>。

imports-loader

状态：草稿

`imports`，顾名思义是导入的意思，这个loader的作用与`exports-loader`刚好相反，是用来将其它模块导入到当前模块中。这个loader常常在处理一些非模块化规范编写的文件时被用到。

背景

举个简单的例子，有模块 `b`，依赖全局变量 `window.a`，那么在没有使用模块化规范的情况下，有可能它的代码中会直接写 `a`：

```
// 模块b直接依赖全局变量a
a.xxx();
```

而如果 `a` 经过webpack打包，则很有可能不会暴露全局变量，此时就需要使用`imports-loader`来对 `a` 进行导入。

使用

todo:例子

```
require('imports?jQuery!./example.js');

require('imports?jQuery,angular,config=>{size:50}!./file.js');
```

典型使用场景

- 为jQuery插件注入\$变量 `imports?jQuery`
- 自定义angular `imports?angular`
- 让umd模块检测不到`define`方法从而避免使用amd规范 `imports?define=>false`
- 将模块中的 `this` 还原为 `window` `imports?this=window`

一目了然，不再解释。

本文全部示例代码见<https://github.com/TooBug/webpack-guide/tree/master/examples/chapter4/exports-loader/example1>。

expose-loader

状态：草稿

`expose`是暴露的意思，那么`expose-loader`的作用即是将变量暴露到`window`对象下成为全局变量。

使用

```
require('expose?GLOBAL_VAR!example.js');
```

在使用`expose-loader`之后，模块原本要导出的内容（`module.exports`）会被暴露到全局变量中，而全局变量的名字则是上面示例中的 `GLOBAL_VAR`。这个全局变量的名字是可以自己定义的。

例如官方有一个示例，将 `React` 暴露到全局变量：

```
require('expose?React!react');
```

要将模块导出的内容暴露给多个全局变量的话，可以使用多次`expose-loader`：

```
expose?${!expose?jQuery!
```

todo: 与 `exports-loader` 结合使用

TypeScript与Vue

本文适用于webpack 2+

TypeScript（ts-loader）

要使用TypeScript的话，只需要将文件名后缀改成 `.ts`，并引入 `ts-loader` 进行处理即可。

例如我们有一个TS文件：

```
function a(num: number){
  return num+1;
}

console.log(a(5));
```

只需要在 `webpack.config.js` 中配置一下（注意这是webpack 2+的配置）：

```
module:{
  rules:[{
    test: /\.ts$/,
    loader:'ts-loader',
  }]
}
```

即可。

Vue单文件组件（vue-loader）

Vue为我们提供了单文件组件的写法，例如下面的 `test.vue`：

```
<style>
  div{
    color:red;
  }
</style>
<template>
  <div>{{message}}</div>
</template>
<script>
import Vue from 'vue/dist/vue.esm.js';
new Vue({
  data: function(){
    return {
      message: 'hello'
    }
  }
});
</script>
```

这种写法需要使用 `vue-loader` 转换成纯JS文件才可以正常在浏览器中运行。和使用 `ts-loader` 类似，只要使用 `vue-loader` 处理即可，这里就不再演示。唯一值得注意的是 `vue-loader` 会需要同时安装几个模块，如果弄不清楚的话，安装 `vue-loader` 的时候留意一下npm的输出，把需要的模块都装上就可以了。

TypeScript + Vue

基本用法就是同时加上 `ts-loader` 和 `vue-loader`。但是 `ts-loader` 需要加上两个选项：

```
options: {  
  transpileOnly: true,  
  appendTsSuffixTo: [/\.vue$/],  
}
```

`transpileOnly` 的含义是指让 `ts-loader` 只做转译。什么意思呢？就是不加这个选项的话，它会把转义的结果写入到文件中，而不是在内存中由 `webpack` 来处理，这会导致后续 `loader` 无法处理 `ts-loader` 的结果。所以加上 `transpileOnly` 让它按 `webpack` 的操作来，这样后续 `loader` 就可以继续处理。

`appendTsSuffixTo` 的含义是碰到 `.vue` 结尾的文件时，加上 `.ts` 的后缀，这样 `ts-loader` 就会去处理 `.vue` 文件中的 `ts` 代码。

另外在使用 `TypeScript` 编写 `Vue` 代码时，可能会碰到一些类型上的问题，可以参见 `Vue` 的官方文档：

<https://vuejs.org/v2/guide/typescript.html>