package.json for NPM 文件详解

package.json文件描述了一个NPM包的所有相关信息，包括作者、简介、包依赖、构建等信息。格式必须是严格的JSON格式。

通常我们在创建一个NPM程序时，可以使用npm init命令，通过交互式的命令，自动生成一个package.json文件，里面包含了常用的一些字段信息，但远不止这么简单。通过完善package.json文件，我们可以让npm命令更好地为我们服务。

**name**

name和version是package.json中最重要的两个字段，也是发布到NPM平台上的唯一标识，如果没有正确设置这两个字段，包就不能发布和被下载。

下面是官方给出的一些建议：

1. 名字里不要再包含"js"和"node"，因为默认NPM包就是node.js程序，不过你可以通过engines字段来指定。
2. 名字将会被作为url的一部分，所有要符合http url的一般命名规则，不能包含url非法字符，也不能以.和\_开头。
3. 名字也将作为require()命令的参数，所以应该尽量简明。
4. 如果包要发布到NPM平台上的话，最好先检查下有没有重名, 并且字母只能全部小写。

新版本的NPM可以指定scope, 名字可以加前缀标识，如@ijse/mypackage。

**version**

这个字段的取值需要符合[node-semver](https://github.com/isaacs/node-semver)的规则，详细可以见其文档。必须字段。

**description**

包的描述信息，将会在npm search的返回结果中显示，以帮助用户选择合适的包。可选字段，必须是字符串。npm search的时候会用到。

**keywords**

包的关键词信息，是一个字符串数组，同上也将显示在npm search的结果中。可选字段，字符串数组。npm search的时候会用到。

**homepage**

包的主页地址，可选字段，没有[http://](http://blog.csdn.net/woxueliuyun/article/details/39294375" \t "http://my.oschina.net/u/1992917/blog/_blank)等带协议前缀的URL。

**bugs**

可选字段，包的bug跟踪主页地址，应该如下设置：

bugs: {

"url": "http://github.com/ijse/project/issues",

"email": "my@ijser.cn"

}

**license**

包的开源协议名称

可选字段。

如果是使用一个普遍的license，比如BSD-3-Clause或MIT，直接使用：

{ "license" : "BSD-3-Clause" }

**author**

包的作者，可以是字符串或对象：

author: {

"name": "ijse", "email": "my@ijse.cn",

"url": "http://www.ijser.cn"

}

或者:

author: "ijse <my@ijser.cn> (http://www.ijser.cn)"

**contributors, maintainers**

包的贡献者，是一个数组。

**files**

包所包含的所有文件，可以取值为文件夹。

通常我们还是用.npmignore来去除不想包含到包里的文件。

可选字段，项目包含的一组文件。如果是文件夹，文件夹下的文件也会被包含。如果需要把某些文件不包含在项目中，添加一个”.npmignore”文件。这个文件和”gitignore”类似。

**main**

包的入口文件，如index.js

可选字段。这个字段的值是你程序主入口模块的ID。如果其他用户需要你的包，当用户调用require()方法时，返回的就是这个模块的导出（exports）

**bin**

如果你的包里包含可执行文件，通过设置这个字段可以将它们包含到系统的PATH中，这样直接就可以运行，很方便。如：

"bin": { "iapp": "./cli.js" }

当包被安装后，NPM将创建一个cli.js文件的链接到/usr/local/bin/iapp下。

**man**

为系统的man命令提供帮助文档, 如：

"man": "./man/doc.1"

帮助文件的文件名必须以数字结尾，如果是压缩的，需要以.gz结尾。

如果是字符串数组：

"name": "foo",

"man": ["./man/foo.1", "./man/bar.1", "./man/foo.2" ]

则分别可以man foo, man foo-bar, man 2 foo来查看。

**directories**

CommonJS包所要求的目录结构信息，目前除了告诉别人你的程序目录结构，貌似没有别的什么用。下级字段可以是：lib, bin, man, doc, example。 每个都是字符串

**repository**

包的仓库地址。如：

"repository": {

"type": "git",

"url": "http://github.com/ijse/project.git"

}

**scripts**

通过设置这个可以使NPM调用一些命令脚本，封装一些功能。

**config**

添加一些设置，可以供scripts读取用，同时这里的值也会被添加到系统的环境变量中。

"name": "foo",

"config": { "port": "8080" }

npm start的时候会读取到npm\_package\_config\_port环境变量。

同时也可以使用npm config命令来修改设置：

npm config set foo:port 8001

**dependencies**

指定依赖的其它包，这些依赖是指包发布后正常执行时所需要的，如果是开发中依赖的包，可以在devDependencies设置。

通常使用下面命令来安装：

npm install --save otherpackage

形式可以有如下多种：

* version 严格匹配某个版本
* >version 必须大于某个版本
* >=version
* <version
* <=version
* ~version 大概匹配某个版本
* ^version 兼容某个版本
* 1.2.x 可以是1.2.0, 1.2.1等等，但不能是1.3.0
* http://... 指定tarball的url地址
* \* 任何版本都可以
* "" 同上
* version1 - version2 >=version1 <=version2
* range1 || range2 满足range1 或range2
* git://... git地址
* user/repo 同上
* tag 指定某个tag的版本
* path/path 本地包所有文件夹

下面都是可以用的：

{ "dependencies" :

{ "foo" : "1.0.0 - 2.9999.9999" ,

"bar" : ">=1.0.2 <2.1.2" ,

"baz" : ">1.0.2 <=2.3.4" ,

"boo" : "2.0.1" ,

"qux" : "<1.0.0 || >=2.3.1 <2.4.5 || >=2.5.2 <3.0.0" ,

"asd" : "http://asdf.com/asdf.tar.gz" ,

"til" : "~1.2" ,

"elf" : "~1.2.3" ,

"two" : "2.x" ,

"thr" : "3.3.x" ,

"lat" : "latest" ,

"dyl" : "file:../dyl"

}

}

Git URL可以有如下种形式：

git://github.com/user/project.git#commit-ish git+ssh://user@hostname:project.git#commit-ish git+ssh://user@hostname/project.git#commit-ish git+http://user@hostname/project/blah.git#commit-ish git+https://user@hostname/project/blah.git#commit-ish

**devDependencies**

这些依赖只有在开发时候才需要。

npm install --save-dev mypack

**peerDependencies**

相关的依赖，如果你的包是插件，而用户在使用你的包时候，通常也会需要这些依赖（插件），那么可以将依赖列到这里。

举个例子，如karma, 它的package.json中有设置：

"peerDependencies": {

"karma-jasmine": "~0.1.0",

"karma-requirejs": "~0.2.0",

"karma-coffee-preprocessor": "~0.1.0",

"karma-html2js-preprocessor": "~0.1.0",

"karma-chrome-launcher": "~0.1.0",

"karma-firefox-launcher": "~0.1.0",

"karma-phantomjs-launcher": "~0.1.0",

"karma-script-launcher": "~0.1.0"

}

这些都是karma的相关插件，一般使用karma的时候都会需要。

**bundledDependencies**

绑定的依赖包，发布的时候这些绑定包也会被一同发布。

**optionalDependencies**

即使这些依赖没有，也可以正常安装使用

**engines**

指定包运行的环境

"engines": {

"node": ">=0.10.3 < 0.12",

"npm": "~1.0.20"

}

**engineStrict**

设置为true强制限定 engine

**os**

指定你的包可以在哪些系统平台下运行。

"os": [ "darwin", "linux", "!win32" ]

即可以在darwin和linux平台下运行，而不能在win32下。这里设定的取值是来自process.platform的。

**cpu**

可以指定包运行的cpu架构，如

"cpu": [ "x64", "!arm" ]

取值来自process.arch。

**preferGlobal**

如果你的包是命令行运行的，那可以将其设置为true建议用户全局(npm install -g)安装。但它并不强制用户。

**private**

设为true这个包将不会发布到NPM平台下。

**publishConfig**

这个字段用于设置发布时候的一些设定。尤其方便你希望发布前设定指定的tag或registry。

也可以设定其它子字段，但只有tag和registry会影响到发布。

**NPM的一些默认值说明**

"scripts": { "start": "node server.js" }

如果在项目根目录下含有server.js文件，则NPM会自动设置此值。

"scripts": { "preinstall": "node-gyp rebuild" }

如果在项目根目录下含有binding.gyp文件，则NPM会自动设置此值。

"contributors": [...]

如果项目根目录下含有AUTHORS文件，则NPM会自动将每一行以Name <email> (url)的格式读取并设定此字段。

**一个node package有两种依赖，一种是dependencies一种是devDependencies，其中前者依赖的项该是正常运行该包时所需要的依赖项，而后者则是开发的时候需要的依赖项，像一些进行单元测试之类的包。**

如果你将包下载下来在包的根目录里运行

npm install

默认会安装两种依赖，如果你只是单纯的使用这个包而不需要进行一些改动测试之类的，可以使用

npm install --production

只安装dependencies而不安装devDependencies。

如果你是通过以下命令进行安装

npm install packagename

那么只会安装dependencies，如果想要安装devDependencies，需要输入

npm install packagename --dev

npm命令概述

npm (node package manager)是node模块管理工具，类似与Linux下的yum和apt。

**常用npm命令**

**安装模块**

npm install

安装当前目录package.json文件中配置的dependencies模块

**安装本地的模块文件**

npm install <tarball file>

Example:

npm install ./package.tgz

**安装指定URL的模块**

npm install <tarball url>

Example:

npm install https://github.com/indexzero/forever/tarball/v0.5.6

**安装本地文件系统中指定的目录包含的模块**

npm install <folder>

**安装并更新package.json中的版本配置**

npm install <name> [–save|–save-dev|–save-optional]

其中：

添加–save 参数安装的模块的名字及其版本信息会出现在package.json的dependencies选项中

添加–save-dev 参数安装的模块的名字及其版本信息会出现在package.json的devDependencies选项中

添加–save-optional 参数安装的模块的名字及其版本信息会出现在package.json的optionalDependencies选项中

**安装模块的config的tag配置中含有指定tag的版本**

npm install <name>@<tag>

Example:

npm install sax@latest

**安装模块的指定版本**

npm install <name>@<version>

Example:

npm install underscore@1.5.2

**安装模块指定版本号范围内的某一个版本**

npm install <name>@<version range>

Example:

npm install async@”>=0.2.0 <0.2.9″

–force强制拉取远程资源，即使本地已经安装这个模块

Example:

npm install underscore –force

-g或–global全局安装模块，如果没有这个参数，会安装在当前目录的node\_modules子目录下

Example:

npm install -g express

**显示npm的bin目录**

npm bin

**设置npm配置**

npm config set <key> <value> [–global]

使用–global参数，设置全局配置

Example:

设置代理

npm config set proxy=http://proxy.tencent.com:8080

设置npm的镜像地址

npm config set registry http://npm.oa.com

**获取npm配置**

npm config get <key>

Example:

获取npm当前镜像地址

npm config get registory

**删除npm配置**

npm config delete <key>

Example:

删除代理设置

npm config delete proxy

**在编辑器中打开npm配置文件**

npm config edit

**交互式的创建package.json文件**

npm init

**创建模块的压缩包**

npm pack [<pkg> [<pkg> … ]]

如果没有参数，则npm会打包当前模块目录

**发布模块，发布后可通过名称来安装该模块**

npm publish <tarball>

npm publish <folder>

其中：

<folder>：包含package.json文件的目录

<tarball>：经过gzip压缩并归档的一个URL或文件路径，该压缩包包含单个目录，且该目录内有package.json文件

**删除模块**

npm rm <name>

npm r <name>

npm uninstall <name>

npm un <name>

注意：不会删除package.json文件dependencies选项中对应的依赖配置

**查找模块**

npm search [search terms ..]

npm s [search terms ..]

npm se [search terms ..]

查找匹配查找字符串的模块

**更新模块**

npm update [-g] [<name> [<name> … ]]

更新指定name列表中的模块。-g参数更新全局安装的模块。

如果没有指定name，且不是在某个模块内，会更新当前目录依赖的所有包都会被更新（包括全局和模块内）；如果当前目录在某个模块目录内，会更新该模块依赖的模块，所以不指定name直接运行npm update时，最好在某个模块内运行，以免更新到其他不想更新的模块。

**执行脚本**

npm start [<name>]

npm stop [<name>]

npm test [<name>] 等

运行package的start脚本，该脚本写在package.json文件scripts的start字段中。

该字段的值可以是当前系统控制台可执行的脚本，也可以是当前系统可执行文件的路径。

如果不传name参数，则执行当前目录下package.json文件中定义的脚本。

[nodejs npm常用命令](http://www.cnblogs.com/linjiqin/p/3765772.html)

npm是一个node包管理和分发工具，已经成为了非官方的发布node模块（包）的标准。有了npm，可以很快的找到特定服务要使用的包，进行下载、安装以及管理已经安装的包。

1. **npm install moduleNames：安装Node模块**

安装完毕后会产生一个node\_modules目录，其目录下就是安装的各个node模块。

node的安装分为全局模式和本地模式。

一般情况下会以本地模式运行，包会被安装到和你的应用程序代码的本地node\_modules目录下。

在全局模式下，Node包会被安装到Node的安装目录下的node\_modules下。

全局安装命令为$npm install -g moduleName。

或者使用$npm set global=true来设定安装模式，$npm get global可以查看当前使用的安装模式。

示例：

npm install express

本默认会安装express的最新版本，也可以通过在后面加版本号的方式安装指定版，如npm install express@3.0.6

npm install <name> -g

将包安装到全局环境中

但是代码中，直接通过require()的方式是没有办法调用全局安装的包的。全局的安装是供命令行使用的，就好像全局安装了vmarket后，就可以在命令行中直接运行vm命令

npm install <name> --save

安装的同时，将信息写入package.json中项目路径中如果有package.json文件时，直接使用npm install方法就可以根据dependencies配置安装所有的依赖包，这样代码提交到github时，就不用提交node\_modules这个文件夹了。

1. **npm view moduleNames：查看node模块的package.json文件夹**

注意事项：如果想要查看package.json文件夹下某个标签的内容，可以使用$npm view moduleName labelName

**3、npm list：查看当前目录下已安装的node包**  
 注意事项：Node模块搜索是从代码执行的当前目录开始的，搜索结果取决于当前使用的目录中的node\_modules下的内容。$ npm list parseable=true可以目录的形式来展现当前安装的所有node包

**4、npm help：查看帮助命令**

**5、npm view moudleName dependencies：查看包的依赖关系**

**6、npm view moduleName repository.url：查看包的源文件地址**

**7、npm view moduleName engines：查看包所依赖的Node的版本**

**8、npm help folders：查看npm使用的所有文件夹**

**9、npm rebuild moduleName：用于更改包内容后进行重建**

**10、npm outdated：检查包是否已经过时，此命令会列出所有已经过时的包，可以及时进行包的更新**

**11、npm update moduleName：更新node模块**

**12、npm uninstall moudleName：卸载node模块**

**13、一个npm包是包含了package.json的文件夹，package.json描述了这个文件夹的结构。访问npm的json文件夹的方法如下：**

**$ npm help json**

**此命令会以默认的方式打开一个网页，如果更改了默认打开程序则可能不会以网页的形式打开。**

1. **发布一个npm包的时候，需要检验某个包名是否已存在**

**$ npm search packageName**

1. **npm init：会引导你创建一个package.json文件，包括名称、版本、作者这些信息等**
2. **npm root：查看当前包的安装路径  
    npm root -g：查看全局的包的安装路径**
3. **npm -v：查看npm安装的版本**

webpack入门(译)

**WELCOME**

这份指南始终围绕一个简单例子。通过学习本教程你可以学到：

* 如何安装webpack
* 如何使用webpack
* 如何使用loader
* 如何使用开发服务器

**安装WEBPACK**

首先你需要安装有node，然后执行

npm install webpack -g

*这会使webpack命令生效*

**开始**

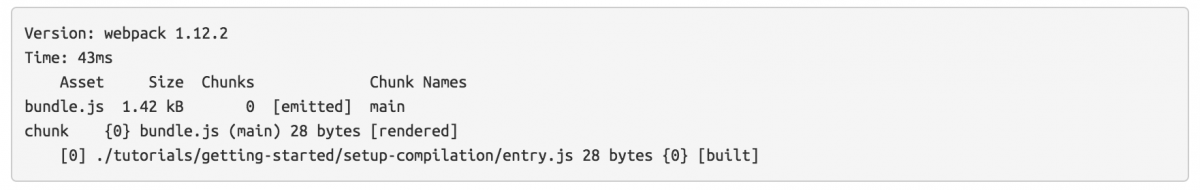
我们从一个空文件夹开始，首先创建以下文件



然后在该文件夹根目录下执行以下命令：

webpack ./entry.js bundle.js

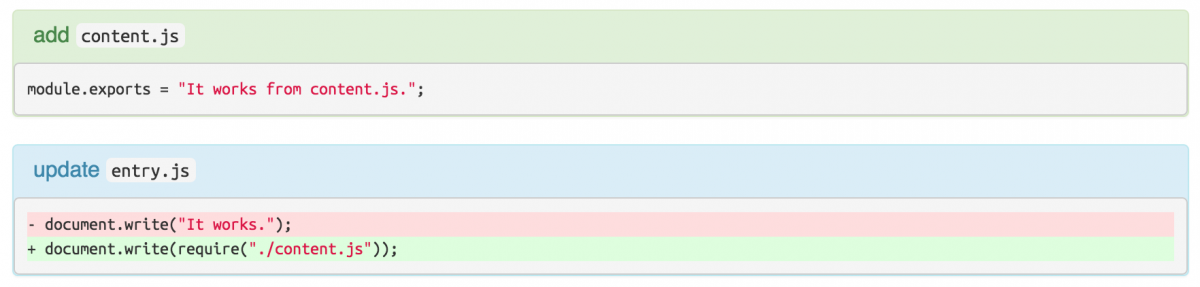
这会编译你的entry.js文件并生成bundle.js。倘若成功，将有以下信息输出：



用浏览器打开index.html，会出现It works.

**再添加一个文件**

接下来，我们会添加一个新文件，并加入如下代码。



再执行以下命令

webpack ./entry.js bundle.js

刷新浏览器，此时你会看到文字变为"It works from content.js."

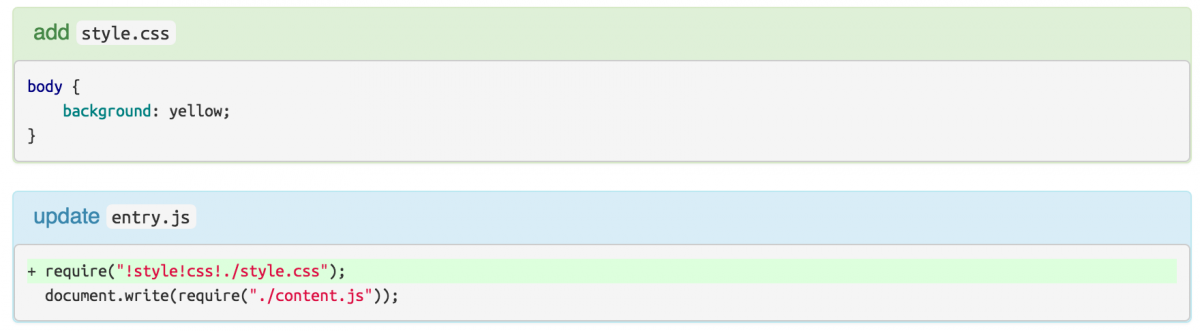
***webpack会分析你入口文件对于其他文件的依赖，这些文件（通常称为模块）也会被包含在bundle.js中。***

**第一个LOADER**

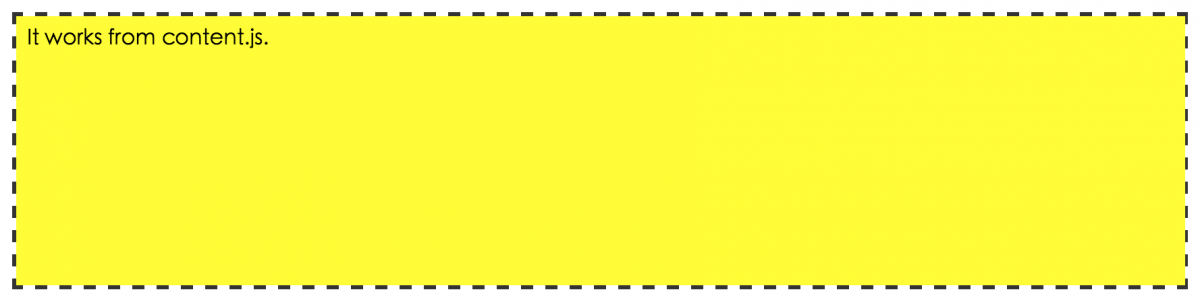
我们希望为我们的应用添加css文件。webpack原生仅支持JavaScript，所以我们需要css-loader来处理css文件，同时我们也需要style-loader来应用这些样式。

运行npm install css-loader style-loader来安装这些loader（此处使用局部安装而非全局安装），这会在你的目录下生成node\_modules文件夹。

需要改动的文件如下：

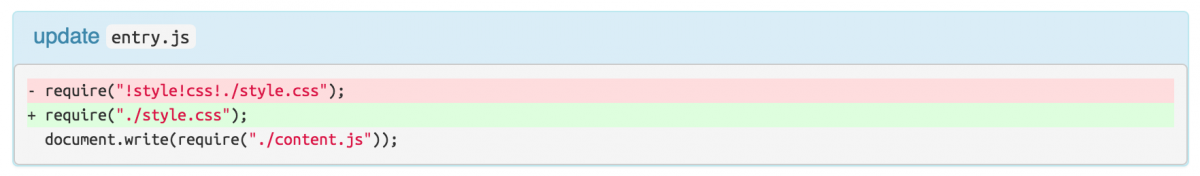


再次编译并刷新浏览器你会看到应用的背景变为黄色。



**绑定LOADER**

我们不希望总是敲下如此冗长的require("!style!css!./style.css");，所以我们可以绑定文件的扩展名以简化写法为require("./style.css")。



执行以下的编译命令：

webpack ./entry.js bundle.js --module-bind "css=style!css"

你将看到同样的结果。

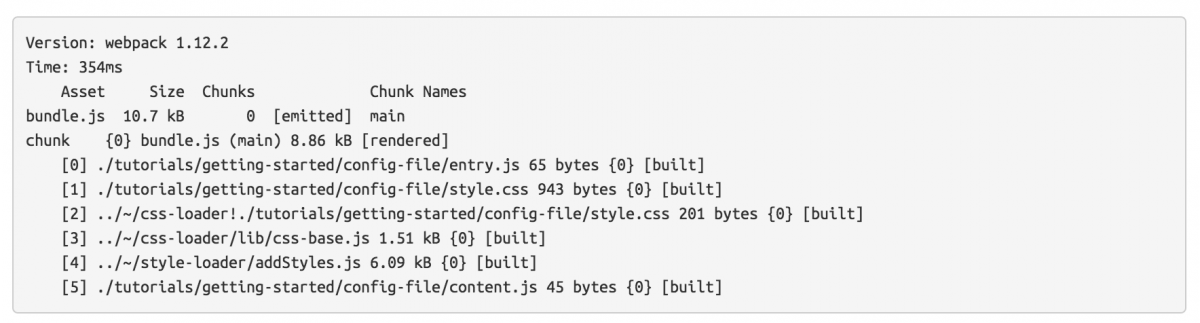
**配置文件**

我们可以将配置选项写入配置文件中。



此后，仅需执行：

webpack



webpack命令总会尝试载入当前目录的webpack.config.js文件

**更友好地输出**

随着项目的增长，编译过程可能会越来越长，所以我们可以展示一些进度条以及增加配色来实现更友好地输出。我们可以通过以下命令达到目的：

webpack --progress --colors

**开发服务器**

提供开发服务器是非常好的一项服务，可以替换python -m SimpleHTTPServer启用HTTP静态服务器

通过以下命令全局安装

npm install webpack-dev-server -g

启动服务器

webpack-dev-server --progress --colors

这会绑定一个小型express服务器到localhost:8080，来为你的静态资源及bundle（自动编译）服务。通过访问http://localhost:8080/webpack-dev-server/bundle，bundle每次重编译后浏览器页面都会自动更新。

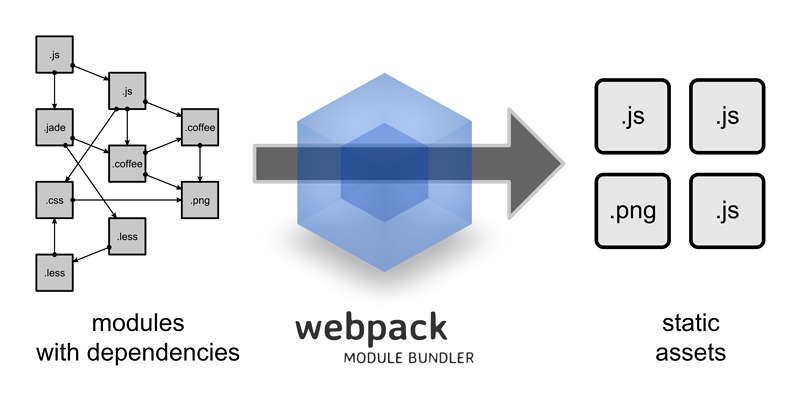
[详解前端模块化工具-Webpack](https://segmentfault.com/a/1190000003970448)

React自发布以来吸引了越来越多的开发者，React开发和模块管理的主流工具webpack也被大家所熟知。那么webpack有哪些优势，可以成为最主流的React开发工具呢？

**webpack是什么**

CommonJS和AMD是用于JavaScript模块管理的两大规范，前者定义的是模块的同步加载，主要用于NodeJS；而后者则是异步加载，通过requirejs等工具适用于前端。随着npm成为主流的JavaScript组件发布平台，越来越多的前端项目也依赖于npm上的项目，或者自身就会发布到npm平台。因此，让前端项目更方便的使用npm上的资源成为一大需求。

web开发中常用到的静态资源主要有JavaScript、CSS、图片、Jade等文件，webpack中将静态资源文件称之为模块。webpack是一个module bundler(模块打包工具)，其可以兼容多种js书写规范，且可以处理模块间的依赖关系，具有更强大的js模块化的功能。Webpack对它们进行统一的管理以及打包发布，其官方主页用下面这张图来说明Webpack的作用：



**webpack特性**

webpack具有requireJs和browserify的功能，但仍有很多自己的新特性：

1. 对 CommonJS 、 AMD 、ES6的语法做了兼容
2. 对js、css、图片等资源文件都支持打包
3. 串联式模块加载器以及插件机制，让其具有更好的灵活性和扩展性，例如提供对CoffeeScript、ES6的支持
4. 有独立的配置文件webpack.config.js
5. 可以将代码切割成不同的chunk，实现按需加载，降低了初始化时间
6. 支持 SourceUrls 和 SourceMaps，易于调试
7. 具有强大的Plugin接口，大多是内部插件，使用起来比较灵活
8. webpack 使用异步 IO 并具有多级缓存。这使得 webpack 很快且在增量编译上更加快

**Webpack安装和配置**

**安装**

webpack 可以作为全局的npm模块安装，也可以在当前项目中安装。

npm install -g webpack

npm install --save-dev webpack

对于全局安装的webpack，直接执行此命令会默认使用当前目录的webpack.config.js作为配置文件。如果要指定另外的配置文件，可以执行：

webpack —config webpack.custom.config.js

**配置**

每个项目下都必须配置有一个 webpack.config.js ，它的作用如同常规的 gulpfile.js/Gruntfile.js ，就是一个配置项，告诉 webpack 它需要做什么。

前文说了，webpack.config.js文件通常放在项目的根目录中，它本身也是一个标准的Commonjs规范的模块。在导出的配置对象中有几个关键的参数：

**entry**

entry参数定义了打包后的入口文件，可以是个字符串或数组或者是对象；如果是数组，数组中的所有文件会打包生成一个filename文件；如果是对象，可以将不同的文件构建成不同的文件:

{

entry: {

page1: "./page1",

//支持数组形式，将加载数组中的所有模块，但以最后一个模块作为输出

page2: ["./entry1", "./entry2"]

},

output: {

path: "dist/js/page",

publicPath: "/output/",

filename: "[name].bundle.js"

}

}

该段代码最终会生成一个 page1.bundle.js 和 page2.bundle.js，并存放到 ./dist/js/page 文件夹下

**output**

output参数是个对象，定义了输出文件的位置及名字：

output: {

path: "dist/js/page",

publicPath: "/output/",

filename: "[name].bundle.js"

}

path:打包文件存放的绝对路径

publicPath:网站运行时的访问路径

filename:打包后的文件名

当我们在entry中定义构建多个文件时，filename可以对应的更改为[name].js用于定义不同文件构建后的名字。

**module**

在webpack中JavaScript，CSS，LESS，TypeScript，JSX，CoffeeScript，图片等静态文件都是模块，不同模块的加载是通过模块加载器（webpack-loader）来统一管理的。loaders之间是可以串联的，一个加载器的输出可以作为下一个加载器的输入，最终返回到JavaScript上：

module: {

//加载器配置

loaders: [

//.css 文件使用 style-loader 和 css-loader 来处理

{ test: /\.css$/, loader: 'style-loader!css-loader' },

//.js 文件使用 jsx-loader 来编译处理

{ test: /\.js$/, loader: 'jsx-loader?harmony' },

//.scss 文件使用 style-loader、css-loader 和 sass-loader 来编译处理

{ test: /\.scss$/, loader: 'style!css!sass?sourceMap'},

//图片文件使用 url-loader 来处理，小于8kb的直接转为base64

{ test: /\.(png|jpg)$/, loader: 'url-loader?limit=8192'}

]

}

test项表示匹配的资源类型，loader或loaders项表示用来加载这种类型的资源的loader。

！用来定义loader的串联关系，"-loader"是可以省略不写的，多个loader之间用“!”连接起来，但所有的加载器都需要通过 npm 来加载。

此外，还可以添加用来定义png、jpg这样的图片资源在小于10k时自动处理为base64图片的加载器：

{ test: /\.(png|jpg)$/,loader: 'url-loader?limit=10000'}

给css和less还有图片添加了loader之后，我们不仅可以像在node中那样require js文件了，我们还可以require css、less甚至图片文件：

require('./bootstrap.css');

require('./myapp.less');

var img = document.createElement('img');

img.src = require('./glyph.png');

注意，require()还支持在资源path前面指定loader，即require(![loaders list]![source path])形式：

require("!style!css!less!bootstrap/less/bootstrap.less");

// “bootstrap.less”这个资源会先被"less-loader"处理，

// 其结果又会被"css-loader"处理，接着是"style-loader"

// 可类比pipe操作

require()时指定的loader会覆盖配置文件里对应的loader配置项。

**resolve**

webpack在构建包的时候会按目录的进行文件的查找，resolve属性中的extensions数组中用于配置程序可以自行补全哪些文件后缀：

resolve: {

//查找module的话从这里开始查找

root: '/pomy/github/flux-example/src', //绝对路径

//自动扩展文件后缀名，意味着我们require模块可以省略不写后缀名

extensions: ['', '.js', '.json', '.scss'],

//模块别名定义，方便后续直接引用别名，无须多写长长的地址

alias: {

AppStore : 'js/stores/AppStores.js',

//后续直接 require('AppStore') 即可

ActionType : 'js/actions/ActionType.js',

AppAction : 'js/actions/AppAction.js'

}

}

然后我们想要加载一个js文件时，只要require('common')就可以加载common.js文件了。

注意一下, extensions 第一个是空字符串! 对应不需要后缀的情况.

**plugin**

webpack提供了[丰富的组件]用来满足不同的需求，当然我们也可以自行实现一个组件来满足自己的需求：

plugins: [

//your plugins list

]

在webpack中编写js文件时，可以通过require的方式引入其他的静态资源，可通过loader对文件自动解析并打包文件。通常会将js文件打包合并，css文件会在页面的header中嵌入style的方式载入页面。但开发过程中我们并不想将样式打在脚本中，最好可以独立生成css文件，以外链的形式加载。这时extract-text-webpack-plugin插件可以帮我们达到想要的效果。需要使用npm的方式加载插件，然后参见下面的配置，就可以将js中的css文件提取，并以指定的文件名来进行加载。

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

plugins: [

new ExtractTextPlugin('styles.css')

]

**externals**

当我们想在项目中require一些其他的类库或者API，而又不想让这些类库的源码被构建到运行时文件中，这在实际开发中很有必要。此时我们就可以通过配置externals参数来解决这个问题：

externals: {

"jquery": "jQuery"

}

这样我们就可以放心的在项目中使用这些API了：var jQuery = require("jquery");

**context**

当我们在require一个模块的时候，如果在require中包含变量，像这样：

require("./mods/" + name + ".js");

那么在编译的时候我们是不能知道具体的模块的。但这个时候，webpack也会为我们做些分析工作：

1. 分析目录：'./mods'；
2. 提取正则表达式：'/^.\*\.js$/'；

于是这个时候为了更好地配合wenpack进行编译，我们可以给它指明路径，像在cake-webpack-config中所做的那样（我们在这里先忽略abcoption的作用）：

var currentBase = process.cwd();

var context = abcOptions.options.context ? abcOptions.options.context :

path.isAbsolute(entryDir) ? entryDir : path.join(currentBase, entryDir);

**webpack常用命令**

webpack的使用通常有三种方式：

1. 命令行使用：webpack <entry.js> <result.js> 其中entry.js是入口文件，result.js是打包后的输出文件
2. node.js API使用：

var webpack = require('webpack');

webpack({//configuration

}, function(err, stats){});

1. 默认使用当前目录的webpack.config.js作为配置文件。如果要指定另外的配置文件，可以执行：webpack --config webpack.custom.config.js

webpack 的执行也很简单，直接执行

$ webpack --display-error-details

即可，后面的参数“--display-error-details”是推荐加上的，方便出错时能查阅更详尽的信息（比如 webpack 寻找模块的过程），从而更好定位到问题。

**常用命令**

webpack的使用和browserify有些类似，下面列举几个常用命令：

webpack 最基本的启动webpack命令

webpack -w 提供watch方法，实时进行打包更新

webpack -p 对打包后的文件进行压缩

webpack -d 提供SourceMaps，方便调试

webpack --colors 输出结果带彩色，比如：会用红色显示耗时较长的步骤

webpack --profile 输出性能数据，可以看到每一步的耗时

webpack --display-modules 默认情况下 node\_modules 下的模块会被隐藏，加上这个参数可以显示这些被隐藏的模块

前面的四个命令比较基础，使用频率会比较大，后面的命令主要是用来定位打包时间较长的原因，方便改进配置文件，提高打包效率。

**图片打包和静态资源服务器**

**图片打包**

webpack中对于图片的处理，可以通过url-loader来实现图片的压缩。

div.img{

background: url(../image/xxx.jpg)

}

//或者

var img = document.createElement("img");

img.src = require("../image/xxx.jpg");document.body.appendChild(img);

针对上面的两种使用方式，loader可以自动识别并处理。根据loader中的设置，webpack会将小于指点大小的文件转化成 base64 格式的 dataUrl，其他图片会做适当的压缩并存放在指定目录中。

module: {

{

test: /\.(png|jpg)$/,

loader: 'url-loader?limit=10000&name=build/[name].[ext]'

}]

}

对于上面的配置，如果图片资源小于10kb就会转化成 base64 格式的 dataUrl，其他的图片会存放在build/文件夹下。

**静态资源服务器**

除了提供模块打包功能，Webpack还提供了一个基于Node.js Express框架的开发服务器，它是一个静态资源Web服务器，对于简单静态页面或者仅依赖于独立服务的前端页面，都可以直接使用这个开发服务器进行开发。在开发过程中，开发服务器会监听每一个文件的变化，进行实时打包，并且可以推送通知前端页面代码发生了变化，从而可以实现页面的自动刷新。

Webpack开发服务器需要单独安装，同样是通过npm进行：

npm install -g webpack-dev-server

可以使用webpack-dev-server直接启动，也可以增加参数来获取更多的功能，具体配置可以参见[官方文档](http://webpack.github.io/docs/webpack-dev-server.html" \t "https://segmentfault.com/a/_blank)。默认启动端口8080，通过localhost:8080/webpack-dev-server/可以访问页面，文件修改后保存时会在页面头部看到sever的状态变化，并且会进行热替换，实现页面的自动刷新。

[webpack 入门指南](http://www.cnblogs.com/vajoy/p/4650467.html)

**什么是 webpack？**

webpack是近期最火的一款模块加载器兼打包工具，它能把各种资源，例如JS（含JSX）、coffee、样式（含less/sass）、图片等都作为模块来使用和处理。

我们可以直接使用 require(XXX) 的形式来引入各模块，即使它们可能需要经过编译（比如JSX和sass），但我们无须在上面花费太多心思，因为 webpack 有着各种健全的加载器（loader）在默默处理这些事情，这块我们后续会提到。

**webpack 的优势**

其优势主要可以归类为如下几个：

1. webpack 是以 commonJS 的形式来书写脚本滴，但对 AMD/CMD 的支持也很全面，方便旧项目进行代码迁移。

2. 能被模块化的不仅仅是 JS 了。

3. 开发便捷，能替代部分 grunt/gulp 的工作，比如打包、压缩混淆、图片转base64等。

4. 扩展性强，插件机制完善，特别是支持 React 热插拔（见 [react-hot-loader](https://github.com/gaearon/react-hot-loader" \t "http://www.cnblogs.com/vajoy/p/_blank) ）的功能让人眼前一亮。

**安装和配置**

**一. 安装**

我们常规直接使用 npm 的形式来安装：

$ npm install webpack -g

当然如果常规项目还是把依赖写入 package.json 包去更人性化：

$ npm init

$ npm install webpack --save-dev

**二. 配置**

每个项目下都必须配置有一个 webpack.config.js ，它的作用如同常规的 gulpfile.js/Gruntfile.js ，就是一个配置项，告诉 webpack 它需要做什么。

var webpack = require('webpack');

var commonsPlugin = new webpack.optimize.CommonsChunkPlugin('common.js');

module.exports = {

//插件项

plugins: [commonsPlugin],

//页面入口文件配置

entry: {

index : './src/js/page/index.js'

},

//入口文件输出配置

output: {

path: 'dist/js/page',

filename: '[name].js'

},

module: {

//加载器配置

loaders: [

{ test: /\.css$/, loader: 'style-loader!css-loader' },

{ test: /\.js$/, loader: 'jsx-loader?harmony' },

{ test: /\.scss$/, loader: 'style!css!sass?sourceMap'},

{ test: /\.(png|jpg)$/, loader: 'url-loader?limit=8192'}

]

},

//其它解决方案配置

resolve: {

root: 'E:/github/flux-example/src',

//绝对路径

extensions: ['', '.js', '.json', '.scss'],

alias: {

AppStore : 'js/stores/AppStores.js',

ActionType : 'js/actions/ActionType.js',

AppAction : 'js/actions/AppAction.js'

}

}

};

⑴ plugins 是插件项，这里我们使用了一个 CommonsChunkPlugin 的插件，它用于提取多个入口文件的公共脚本部分，然后生成一个 common.js 来方便多页面之间进行复用。

⑵ entry 是页面入口文件配置，output 是对应输出项配置（即入口文件最终要生成什么名字的文件、存放到哪里），其语法大致为：

{

entry: {

page1: "./page1",

//支持数组形式，将加载数组中的所有模块，但以最后一个模块作为输出

page2: ["./entry1", "./entry2"]

},

output: {

path: "dist/js/page",

filename: "[name].bundle.js"

}

}

该段代码最终会生成一个 page1.bundle.js 和 page2.bundle.js，并存放到 ./dist/js/page 文件夹下。

⑶ module.loaders 是最关键的一块配置。它告知 webpack 每一种文件都需要使用什么加载器来处理：

module: {

//加载器配置

loaders: [

//.css 文件使用 style-loader 和 css-loader 来处理

{ test: /\.css$/, loader: 'style-loader!css-loader' },

//.js 文件使用 jsx-loader 来编译处理

{ test: /\.js$/, loader: 'jsx-loader?harmony' },

//.scss 文件使用 style-loader、css-loader 和 sass-loader 来编译处理

{ test: /\.scss$/, loader: 'style!css!sass?sourceMap'},

//图片文件使用 url-loader 来处理，小于8kb的直接转为base64

{ test: /\.(png|jpg)$/, loader: 'url-loader?limit=8192'}

]

}

如上，"-loader"其实是可以省略不写的，多个loader之间用“!”连接起来。

注意所有的加载器都需要通过 npm 来加载，并建议查阅它们对应的 readme 来看看如何使用。

拿最后一个 [url-loader](https://github.com/webpack/url-loader" \t "http://www.cnblogs.com/vajoy/p/_blank) 来说，它会将样式中引用到的图片转为模块来处理，使用该加载器需要先进行安装：

npm install url-loader -save-dev

配置信息的参数“?limit=8192”表示将所有小于8kb的图片都转为base64形式（其实应该说超过8kb的才使用 url-loader 来映射到文件，否则转为data url形式）。

⑷ 最后是 resolve 配置，这块很好理解，直接写注释了：

resolve: {

//查找module的话从这里开始查找

root: 'E:/github/flux-example/src', //绝对路径

//自动扩展文件后缀名，意味着我们require模块可以省略不写后缀名

extensions: ['', '.js', '.json', '.scss'],

//模块别名定义，方便后续直接引用别名，无须多写长长的地址

alias: {

AppStore : 'js/stores/AppStores.js',

//后续直接 require('AppStore') 即可

ActionType : 'js/actions/ActionType.js',

AppAction : 'js/actions/AppAction.js'

}

}

**运行 webpack**

webpack 的执行也很简单，直接执行

$ webpack --display-error-details

即可，后面的参数“--display-error-details”是推荐加上的，方便出错时能查阅更详尽的信息（比如 webpack 寻找模块的过程），从而更好定位到问题。

其他主要的参数有：

$ webpack --config XXX.js

//使用另一份配置文件（比如webpack.config2.js）来打包

$ webpack --watch //监听变动并自动打包

$ webpack -p //压缩混淆脚本，这个非常非常重要！

$ webpack -d //生成map映射文件，告知哪些模块被最终打包到哪里了

其中的 -p 是很重要的参数，曾经一个未压缩的 700kb 的文件，压缩后直接降到 180kb（主要是样式这块一句就独占一行脚本，导致未压缩脚本变得很大）。

**模块引入**

**一. HTML**

直接在页面引入 webpack 最终生成的页面脚本即可，不用再写什么 data-main 或 seajs.use 了：

<!DOCTYPE html>

<html>

<head lang="en">

<meta charset="UTF-8">

<title>demo</title>

</head>

<body>

<script src="dist/js/page/common.js"></script>

<script src="dist/js/page/index.js"></script>

</body>

</html>

可以看到我们连样式都不用引入，毕竟脚本执行时会动态生成<style>并标签打到head里。

**二. JS**

各脚本模块可以直接使用 commonJS 来书写，并可以直接引入未经编译的模块，比如 JSX、sass、coffee等（只要你在 webpack.config.js 里配置好了对应的加载器）。

我们再看看编译前的页面入口文件（index.js）：

require('../../css/reset.scss'); //加载初始化样式

require('../../css/allComponent.scss'); //加载组件样式

var React = require('react');

var AppWrap = require('../component/AppWrap'); //加载组件

var createRedux = require('redux').createRedux;

var Provider = require('redux/react').Provider;

var stores = require('AppStore');

var redux = createRedux(stores);

var App = React.createClass({

render: function() {

return (

<Provider redux={redux}>

{function() { return <AppWrap />; }}

</Provider>

);

}

});

React.render(

<App />, document.body

);

一切就是这么简单，后续各种有的没的，webpack 都会帮你进行处理。

**其他**

至此我们已经基本上手了 webpack 的使用，下面是补充一些有用的技巧。

**一. shimming**

在 AMD/CMD 中，我们需要对不符合规范的模块（比如一些直接返回全局变量的插件）进行 shim 处理，这时候我们需要使用 [exports-loader](https://github.com/webpack/exports-loader" \t "http://www.cnblogs.com/vajoy/p/_blank) 来帮忙：

{

test: require.resolve("./src/js/tool/swipe.js"),

loader: "exports?swipe"

}

之后在脚本中需要引用该模块的时候，这么简单地来使用就可以了：

require('./tool/swipe.js');

swipe();

**二. 自定义公共模块提取**

在文章开始我们使用了 CommonsChunkPlugin 插件来提取多个页面之间的公共模块，并将该模块打包为 common.js 。

但有时候我们希望能更加个性化一些，我们可以这样配置：

var CommonsChunkPlugin = require("webpack/lib/optimize/CommonsChunkPlugin");

module.exports = {

entry: {

p1: "./page1",

p2: "./page2",

p3: "./page3",

ap1: "./admin/page1",

ap2: "./admin/page2"

},

output: {

filename: "[name].js"

},

plugins: [

new CommonsChunkPlugin("admin-commons.js", ["ap1", "ap2"]),

new CommonsChunkPlugin("commons.js", ["p1", "p2", "admin-commons.js"])

]

};

// <script>s required:

// page1.html: commons.js, p1.js

// page2.html: commons.js, p2.js

// page3.html: p3.js

// admin-page1.html: commons.js, admin-commons.js, ap1.js

// admin-page2.html: commons.js, admin-commons.js, ap2.js

**三. 独立打包样式文件**

有时候可能希望项目的样式能不要被打包到脚本中，而是独立出来作为.css，然后在页面中以<link>标签引入。这时候我们需要 [extract-text-webpack-plugin](https://github.com/webpack/extract-text-webpack-plugin" \t "http://www.cnblogs.com/vajoy/p/_blank) 来帮忙：

var webpack = require('webpack');

var commonsPlugin = new webpack.optimize.CommonsChunkPlugin('common.js');

var ExtractTextPlugin = require("extract-text-webpack-plugin");

module.exports = {

plugins: [commonsPlugin, new ExtractTextPlugin("[name].css")],

entry: {

//...省略其它配置

最终 webpack 执行后会乖乖地把样式文件提取出来：

**四. 使用CDN/远程文件**

有时候我们希望某些模块走CDN并以<script>的形式挂载到页面上来加载，但又希望能在 webpack 的模块中使用上。

这时候我们可以在配置文件里使用 externals 属性来帮忙：

{

externals: {

// require("jquery") 是引用自外部模块的

// 对应全局变量 jQuery

"jquery": "jQuery"

}

}

需要留意的是，得确保 CDN 文件必须在 webpack 打包文件引入之前先引入。

我们倒也可以使用 [script.js](https://github.com/ded/script.js" \t "http://www.cnblogs.com/vajoy/p/_blank) 在脚本中来加载我们的模块：

var $script = require("scriptjs");

$script("//ajax.googleapis.com/ajax/libs/jquery/2.0.0/jquery.min.js", function() {

$('body').html('It works!')

});

**五. 与 grunt/gulp 配合**

以 gulp 为示例，我们可以这样混搭：

gulp.task("webpack", function(callback) {

// run webpack

webpack({

// configuration

}, function(err, stats) {

if(err) throw new gutil.PluginError("webpack", err);

gutil.log("[webpack]", stats.toString({

// output options }));

callback();

});

});

当然我们只需要把配置写到 webpack({ ... }) 中去即可，无须再写 webpack.config.js 了。

# Module

ES6的Class只是面向对象编程的语法糖，升级了ES5的构造函数的原型链继承的写法，并没有解决模块化问题。Module功能就是为了解决这个问题而提出的。

历史上，JavaScript一直没有模块（module）体系，无法将一个大程序拆分成互相依赖的小文件，再用简单的方法拼装起来。其他语言都有这项功能，比如Ruby的require、Python的import，甚至就连CSS都有@import，但是JavaScript任何这方面的支持都没有，这对开发大型的、复杂的项目形成了巨大障碍。

在ES6之前，社区制定了一些模块加载方案，最主要的有CommonJS和AMD两种。前者用于服务器，后者用于浏览器。ES6在语言规格的层面上，实现了模块功能，而且实现得相当简单，完全可以取代现有的CommonJS和AMD规范，成为浏览器和服务器通用的模块解决方案。

ES6模块的设计思想，是尽量的静态化，使得编译时就能确定模块的依赖关系，以及输入和输出的变量。CommonJS和AMD模块，都只能在运行时确定这些东西。比如，CommonJS模块就是对象，输入时必须查找对象属性。

// CommonJS模块

let { stat, exists, readFile } = require('fs');

// 等同于

let \_fs = require('fs');

let stat = \_fs.stat, exists = \_fs.exists, readfile = \_fs.readfile;

上面代码的实质是整体加载fs模块（即加载fs的所有方法），生成一个对象（\_fs），然后再从这个对象上面读取3个方法。这种加载称为“运行时加载”，因为只有运行时才能得到这个对象，导致完全没办法在编译时做“静态优化”。

ES6模块不是对象，而是通过export命令显式指定输出的代码，输入时也采用静态命令的形式。

// ES6模块

import { stat, exists, readFile } from 'fs';

上面代码的实质是从fs模块加载3个方法，其他方法不加载。这种加载称为“编译时加载”，即ES6可以在编译时就完成模块加载，效率要比CommonJS模块的加载方式高。当然，这也导致了没法引用ES6模块本身，因为它不是对象。

由于ES6模块是编译时加载，使得静态分析成为可能。有了它，就能进一步拓宽JavaScript的语法，比如引入宏（macro）和类型检验（type system）这些只能靠静态分析实现的功能。

除了静态加载带来的各种好处，ES6模块还有以下好处。

1. 不再需要UMD模块格式了，将来服务器和浏览器都会支持ES6模块格式。目前，通过各种工具库，其实已经做到了这一点。
2. 将来浏览器的新API就能用模块格式提供，不再必要做成全局变量或者navigator对象的属性。
3. 不再需要对象作为命名空间（比如Math对象），未来这些功能可以通过模块提供。

**严格模式**

ES6的模块自动采用严格模式，不管你有没有在模块头部加上"use strict";。

严格模式主要有以下限制。

1. 变量必须声明后再使用
2. 函数的参数不能有同名属性，否则报错
3. 不能使用with语句
4. 不能对只读属性赋值，否则报错
5. 不能使用前缀0表示八进制数，否则报错
6. 不能删除不可删除的属性，否则报错
7. 不能删除变量delete prop，会报错，只能删除属性delete global[prop]
8. eval不会在它的外层作用域引入变量
9. eval和arguments不能被重新赋值
10. arguments不会自动反映函数参数的变化
11. 不能使用arguments.callee
12. 不能使用arguments.caller
13. 禁止this指向全局对象
14. 不能使用fn.caller和fn.arguments获取函数调用的堆栈
15. 增加了保留字（比如protected、static和interface）

上面这些限制，模块都必须遵守。由于严格模式是ES5引入的，不属于ES6，所以请参阅相关ES5书籍，本书不再详细介绍了。

**export命令**

模块功能主要由两个命令构成：export和import。export命令用于规定模块的对外接口，import命令用于输入其他模块提供的功能。

一个模块就是一个独立的文件。该文件内部的所有变量，外部无法获取。如果你希望外部能够读取模块内部的某个变量，就必须使用export关键字输出该变量。下面是一个JS文件，里面使用export命令输出变量。

// profile.js

export var firstName = 'Michael';

export var lastName = 'Jackson';

export var year = 1958;

上面代码是profile.js文件，保存了用户信息。ES6将其视为一个模块，里面用export命令对外部输出了三个变量。

export的写法，除了像上面这样，还有另外一种。

// profile.js

var firstName = 'Michael';

var lastName = 'Jackson';

var year = 1958;

export {firstName, lastName, year};

上面代码在export命令后面，使用大括号指定所要输出的一组变量。它与前一种写法（直接放置在var语句前）是等价的，但是应该优先考虑使用这种写法。因为这样就可以在脚本尾部，一眼看清楚输出了哪些变量。

export命令除了输出变量，还可以输出函数或类（class）。

export function multiply (x, y) {

return x \* y;

};

上面代码对外输出一个函数multiply。

通常情况下，export输出的变量就是本来的名字，但是可以使用as关键字重命名。

function v1() { ... }

function v2() { ... }

export {

v1 as streamV1,

v2 as streamV2,

v2 as streamLatestVersion

};

上面代码使用as关键字，重命名了函数v1和v2的对外接口。重命名后，v2可以用不同的名字输出两次。

需要特别注意的是，export命令规定的是对外的接口，必须与模块内部的变量建立一一对应关系。

// 报错

export 1;

// 报错

var m = 1;

export m;

上面两种写法都会报错，因为没有提供对外的接口。第一种写法直接输出1，第二种写法通过变量m，还是直接输出1。1只是一个值，不是接口。正确的写法是下面这样。

// 写法一

export var m = 1;

// 写法二

var m = 1;

export {m};

// 写法三

var n = 1;

export {n as m};

上面三种写法都是正确的，规定了对外的接口m。其他脚本可以通过这个接口，取到值1。它们的实质是，在接口名与模块内部变量之间，建立了一一对应的关系。

同样的，function和class的输出，也必须遵守这样的写法。

// 报错

function f() {}

export f;

// 正确

export function f() {};

// 正确

function f() {}

export {f};

另外，export语句输出的接口，与其对应的值是动态绑定关系，即通过该接口，可以取到模块内部实时的值。

export var foo = 'bar';

setTimeout(() => foo = 'baz', 500);

上面代码输出变量foo，值为bar，500毫秒之后变成baz。

这一点与CommonJS规范完全不同。CommonJS模块输出的是值的缓存，不存在动态更新，详见下文《ES6模块加载的实质》一节。

最后，export命令可以出现在模块的任何位置，只要处于模块顶层就可以。如果处于块级作用域内，就会报错，下一节的import命令也是如此。这是因为处于条件代码块之中，就没法做静态优化了，违背了ES6模块的设计初衷。

function foo() {

export default 'bar' // SyntaxError

}

foo()

上面代码中，export语句放在函数之中，结果报错。

**import命令**

使用export命令定义了模块的对外接口以后，其他JS文件就可以通过import命令加载这个模块（文件）。

// main.js

import {firstName, lastName, year} from './profile';

function setName(element) {

element.textContent = firstName + ' ' + lastName;

}

上面代码的import命令，就用于加载profile.js文件，并从中输入变量。import命令接受一个对象（用大括号表示），里面指定要从其他模块导入的变量名。大括号里面的变量名，必须与被导入模块（profile.js）对外接口的名称相同。

如果想为输入的变量重新取一个名字，import命令要使用as关键字，将输入的变量重命名。

import { lastName as surname } from './profile';

注意，import命令具有提升效果，会提升到整个模块的头部，首先执行。

foo();

import { foo } from 'my\_module';

上面的代码不会报错，因为import的执行早于foo的调用。

如果在一个模块之中，先输入后输出同一个模块，import语句可以与export语句写在一起。

export { es6 as default } from './someModule';

// 等同于

import { es6 } from './someModule';

export default es6;

上面代码中，export和import语句可以结合在一起，写成一行。但是从可读性考虑，不建议采用这种写法，而应该采用标准写法。

另外，ES7有一个提案，简化先输入后输出的写法，拿掉输出时的大括号。

// 提案的写法

export v from 'mod';

// 现行的写法

export {v} from 'mod';

import语句会执行所加载的模块，因此可以有下面的写法。

import 'lodash';

上面代码仅仅执行lodash模块，但是不输入任何值。

**模块的整体加载**

除了指定加载某个输出值，还可以使用整体加载，即用星号（\*）指定一个对象，所有输出值都加载在这个对象上面。

下面是一个circle.js文件，它输出两个方法area和circumference。

// circle.js

export function area(radius) {

return Math.PI \* radius \* radius;

}

export function circumference(radius) {

return 2 \* Math.PI \* radius;

}

现在，加载这个模块。

// main.js

import { area, circumference } from './circle';

console.log('圆面积：' + area(4));

console.log('圆周长：' + circumference(14));

上面写法是逐一指定要加载的方法，整体加载的写法如下。

import \* as circle from './circle';

console.log('圆面积：' + circle.area(4));

console.log('圆周长：' + circle.circumference(14));

**export default命令**

从前面的例子可以看出，使用import命令的时候，用户需要知道所要加载的变量名或函数名，否则无法加载。但是，用户肯定希望快速上手，未必愿意阅读文档，去了解模块有哪些属性和方法。

为了给用户提供方便，让他们不用阅读文档就能加载模块，就要用到export default命令，为模块指定默认输出。

// export-default.js

export default function () {

console.log('foo');

}

上面代码是一个模块文件export-default.js，它的默认输出是一个函数。

其他模块加载该模块时，import命令可以为该匿名函数指定任意名字。

// import-default.js

import customName from './export-default';

customName(); // 'foo'

上面代码的import命令，可以用任意名称指向export-default.js输出的方法，这时就不需要知道原模块输出的函数名。需要注意的是，这时import命令后面，不使用大括号。

export default命令用在非匿名函数前，也是可以的。

// export-default.js

export default function foo() {

console.log('foo');

}

// 或者写成

function foo() {

console.log('foo');

}

export default foo;

上面代码中，foo函数的函数名foo，在模块外部是无效的。加载的时候，视同匿名函数加载。

下面比较一下默认输出和正常输出。

// 输出

export default function crc32() {

// ...

}

// 输入

import crc32 from 'crc32';

// 输出

export function crc32() {

// ...

};

// 输入

import {crc32} from 'crc32';

上面代码的两组写法，第一组是使用export default时，对应的import语句不需要使用大括号；第二组是不使用export default时，对应的import语句需要使用大括号。

export default命令用于指定模块的默认输出。显然，一个模块只能有一个默认输出，因此export deault命令只能使用一次。所以，import命令后面才不用加大括号，因为只可能对应一个方法。

本质上，export default就是输出一个叫做default的变量或方法，然后系统允许你为它取任意名字。所以，下面的写法是有效的。

// modules.js

function add(x, y) {

return x \* y;

};

export {add as default};

// 等同于

// export default add;

// app.js

import { default as xxx } from 'modules';

// 等同于

// import xxx from 'modules';

正是因为export default命令其实只是输出一个叫做default的变量，所以它后面不能跟变量声明语句。

// 正确

export var a = 1;

// 正确

var a = 1;

export default a;

// 错误

export default var a = 1;

上面代码中，export default a的含义是将变量a的值赋给变量default。所以，最后一种写法会报错。

有了export default命令，输入模块时就非常直观了，以输入jQuery模块为例。

import $ from 'jquery';

如果想在一条import语句中，同时输入默认方法和其他变量，可以写成下面这样。

import customName, { otherMethod } from './export-default';

如果要输出默认的值，只需将值跟在export default之后即可。

export default 42;

export default也可以用来输出类。

// MyClass.js

export default class { ... }

// main.js

import MyClass from 'MyClass'

let o = new MyClass();

**模块的继承**

模块之间也可以继承。

假设有一个circleplus模块，继承了circle模块。

// circleplus.js

export \* from 'circle';

export var e = 2.71828182846;

export default function(x) {

return Math.exp(x);

}

上面代码中的export \*，表示再输出circle模块的所有属性和方法。注意，export \*命令会忽略circle模块的default方法。然后，上面代码又输出了自定义的e变量和默认方法。

这时，也可以将circle的属性或方法，改名后再输出。

// circleplus.js

export { area as circleArea } from 'circle';

上面代码表示，只输出circle模块的area方法，且将其改名为circleArea。

加载上面模块的写法如下。

// main.js

import \* as math from 'circleplus';

import exp from 'circleplus';

console.log(exp(math.e));

上面代码中的import exp表示，将circleplus模块的默认方法加载为exp方法。

**ES6模块加载的实质**

ES6模块加载的机制，与CommonJS模块完全不同。CommonJS模块输出的是一个值的拷贝，而ES6模块输出的是值的引用。

CommonJS模块输出的是被输出值的拷贝，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。请看下面这个模块文件lib.js的例子。

// lib.js

var counter = 3;

function incCounter() {

counter++;

}

module.exports = {

counter: counter,

incCounter: incCounter,

};

上面代码输出内部变量counter和改写这个变量的内部方法incCounter。然后，在main.js里面加载这个模块。

// main.js

var mod = require('./lib');

console.log(mod.counter); // 3

mod.incCounter();

console.log(mod.counter); // 3

上面代码说明，lib.js模块加载以后，它的内部变化就影响不到输出的mod.counter了。这是因为mod.counter是一个原始类型的值，会被缓存。除非写成一个函数，才能得到内部变动后的值。

// lib.js

var counter = 3;

function incCounter() {

counter++;

}

module.exports = {

get counter() {

return counter

},

incCounter: incCounter,

};

上面代码中，输出的counter属性实际上是一个取值器函数。现在再执行main.js，就可以正确读取内部变量counter的变动了。

$ node main.js

3

4

ES6模块的运行机制与CommonJS不一样，它遇到模块加载命令import时，不会去执行模块，而是只生成一个动态的只读引用。等到真的需要用到时，再到模块里面去取值，换句话说，ES6的输入有点像Unix系统的”符号连接“，原始值变了，import输入的值也会跟着变。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。

上面代码说明，ES6模块输入的变量counter是活的，完全反应其所在模块lib.js内部的变化。

再举一个出现在export一节中的例子。

// m1.js

export var foo = 'bar';

setTimeout(() => foo = 'baz', 500);

// m2.js

import {foo} from './m1.js';

console.log(foo);

setTimeout(() => console.log(foo), 500);

上面代码中，m1.js的变量foo，在刚加载时等于bar，过了500毫秒，又变为等于baz。

让我们看看，m2.js能否正确读取这个变化。

$ babel-node m2.js

bar

baz

上面代码表明，ES6模块不会缓存运行结果，而是动态地去被加载的模块取值，并且变量总是绑定其所在的模块。

由于ES6输入的模块变量，只是一个”符号连接“，所以这个变量是只读的，对它进行重新赋值会报错。

// lib.js

export let obj = {};

// main.js

import { obj } from './lib';

obj.prop = 123; // OK

obj = {}; // TypeError

上面代码中，main.js从lib.js输入变量obj，可以对obj添加属性，但是重新赋值就会报错。因为变量obj指向的地址是只读的，不能重新赋值，这就好比main.js创造了一个名为obj的const变量。

最后，export通过接口，输出的是同一个值。不同的脚本加载这个接口，得到的都是同样的实例。

// mod.js

function C() {

this.sum = 0;

this.add = function () {

this.sum += 1;

};

this.show = function () {

console.log(this.sum);

}

}

export let c = new C();

上面的脚本mod.js，输出的是一个C的实例。不同的脚本加载这个模块，得到的都是同一个实例。

// x.js

import {c} from './mod';

c.add();

// y.js

import {c} from './mod';

c.show();

// main.js

import './x';

import './y';

现在执行main.js，输出的是1。

$ babel-node main.js

1

这就证明了x.js和y.js加载的都是C的同一个实例。

**循环加载**

“循环加载”（circular dependency）指的是，a脚本的执行依赖b脚本，而b脚本的执行又依赖a脚本。

// a.js

var b = require('b');

// b.js

var a = require('a');

通常，“循环加载”表示存在强耦合，如果处理不好，还可能导致递归加载，使得程序无法执行，因此应该避免出现。

但是实际上，这是很难避免的，尤其是依赖关系复杂的大项目，很容易出现a依赖b，b依赖c，c又依赖a这样的情况。这意味着，模块加载机制必须考虑“循环加载”的情况。

对于JavaScript语言来说，目前最常见的两种模块格式CommonJS和ES6，处理“循环加载”的方法是不一样的，返回的结果也不一样。

**CommonJS模块的加载原理**

介绍ES6如何处理"循环加载"之前，先介绍目前最流行的CommonJS模块格式的加载原理。

CommonJS的一个模块，就是一个脚本文件。require命令第一次加载该脚本，就会执行整个脚本，然后在内存生成一个对象。

{

id: '...',

exports: { ... },

loaded: true,

...

}

上面代码中，该对象的id属性是模块名，exports属性是模块输出的各个接口，loaded属性是一个布尔值，表示该模块的脚本是否执行完毕。其他还有很多属性，这里都省略了。

以后需要用到这个模块的时候，就会到exports属性上面取值。即使再次执行require命令，也不会再次执行该模块，而是到缓存之中取值。也就是说，CommonJS模块无论加载多少次，都只会在第一次加载时运行一次，以后再加载，就返回第一次运行的结果，除非手动清除系统缓存。

**CommonJS模块的循环加载**

CommonJS模块的重要特性是加载时执行，即脚本代码在require的时候，就会全部执行。一旦出现某个模块被"循环加载"，就只输出已经执行的部分，还未执行的部分不会输出。

让我们来看，Node官方文档里面的例子。脚本文件a.js代码如下。

exports.done = false;

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

console.log('在 a.js 之中，b.done = %j', b.done);

exports.done = true;

console.log('a.js 执行完毕');

上面代码之中，a.js脚本先输出一个done变量，然后加载另一个脚本文件b.js。注意，此时a.js代码就停在这里，等待b.js执行完毕，再往下执行。

再看b.js的代码。

exports.done = false;

var a = require('./a.js');

console.log('在 b.js 之中，a.done = %j', a.done);

exports.done = true;

console.log('b.js 执行完毕');

上面代码之中，b.js执行到第二行，就会去加载a.js，这时，就发生了“循环加载”。系统会去a.js模块对应对象的exports属性取值，可是因为a.js还没有执行完，从exports属性只能取回已经执行的部分，而不是最后的值。

a.js已经执行的部分，只有一行。

exports.done = false;

因此，对于b.js来说，它从a.js只输入一个变量done，值为false。

然后，b.js接着往下执行，等到全部执行完毕，再把执行权交还给a.js。于是，a.js接着往下执行，直到执行完毕。我们写一个脚本main.js，验证这个过程。

var a = require('./a.js');

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

console.log('在 main.js 之中, a.done=%j, b.done=%j', a.done, b.done);

执行main.js，运行结果如下。

$ node main.js

在 b.js 之中，a.done = false

b.js 执行完毕

在 a.js 之中，b.done = true

a.js 执行完毕

在 main.js 之中, a.done=true, b.done=true

上面的代码证明了两件事。一是，在b.js之中，a.js没有执行完毕，只执行了第一行。二是，main.js执行到第二行时，不会再次执行b.js，而是输出缓存的b.js的执行结果，即它的第四行。

exports.done = true;

总之，CommonJS输入的是被输出值的拷贝，不是引用。

另外，由于CommonJS模块遇到循环加载时，返回的是当前已经执行的部分的值，而不是代码全部执行后的值，两者可能会有差异。所以，输入变量的时候，必须非常小心。

var a = require('a'); // 安全的写法

var foo = require('a').foo; // 危险的写法

exports.good = function (arg) {

return a.foo('good', arg); // 使用的是 a.foo 的最新值

};

exports.bad = function (arg) {

return foo('bad', arg); // 使用的是一个部分加载时的值

};

上面代码中，如果发生循环加载，require('a').foo的值很可能后面会被改写，改用require('a')会更保险一点。

**ES6模块的循环加载**

ES6处理“循环加载”与CommonJS有本质的不同。ES6模块是动态引用，遇到模块加载命令import时，不会去执行模块，只是生成一个指向被加载模块的引用，需要开发者自己保证，真正取值的时候能够取到值。

请看下面的例子（摘自 Dr. Axel Rauschmayer 的《Exploring ES6》）。

// a.js

import {bar} from './b.js';

export function foo() {

bar();

console.log('执行完毕');

}

foo();

// b.js

import {foo} from './a.js';

export function bar() {

if (Math.random() > 0.5) {

foo();

}

}

按照CommonJS规范，上面的代码是没法执行的。a先加载b，然后b又加载a，这时a还没有任何执行结果，所以输出结果为null，即对于b.js来说，变量foo的值等于null，后面的foo()就会报错。

但是，ES6可以执行上面的代码。

$ babel-node a.js

执行完毕

a.js之所以能够执行，原因就在于ES6加载的变量，都是动态引用其所在的模块。只要引用是存在的，代码就能执行。

我们再来看ES6模块加载器SystemJS给出的一个例子。

// even.js

import { odd } from './odd'

export var counter = 0;

export function even(n) {

counter++;

return n == 0 || odd(n - 1);

}

// odd.js

import { even } from './even';

export function odd(n) {

return n != 0 && even(n - 1);

}

上面代码中，even.js里面的函数even有一个参数n，只要不等于0，就会减去1，传入加载的odd()。odd.js也会做类似操作。

运行上面这段代码，结果如下。

$ babel-node

> import \* as m from './even.js';

> m.even(10);

true

> m.counter

6

> m.even(20)

true

> m.counter

17

上面代码中，参数n从10变为0的过程中，even()一共会执行6次，所以变量counter等于6。第二次调用even()时，参数n从20变为0，even()一共会执行11次，加上前面的6次，所以变量counter等于17。

这个例子要是改写成CommonJS，就根本无法执行，会报错。

// even.js

var odd = require('./odd');

var counter = 0;

exports.counter = counter;

exports.even = function(n) {

counter++;

return n == 0 || odd(n - 1);

}

// odd.js

var even = require('./even').even;

module.exports = function(n) {

return n != 0 && even(n - 1);

}

上面代码中，even.js加载odd.js，而odd.js又去加载even.js，形成“循环加载”。这时，执行引擎就会输出even.js已经执行的部分（不存在任何结果），所以在odd.js之中，变量even等于null，等到后面调用even(n-1)就会报错。

$ node

> var m = require('./even');

> m.even(10)

TypeError: even is not a function

**ES6模块的转码**

浏览器目前还不支持ES6模块，为了现在就能使用，可以将转为ES5的写法。除了Babel可以用来转码之外，还有以下两个方法，也可以用来转码。

**ES6 module transpiler**

ES6 module transpiler是square公司开源的一个转码器，可以将ES6模块转为CommonJS模块或AMD模块的写法，从而在浏览器中使用。

首先，安装这个转玛器。

$ npm install -g es6-module-transpiler

然后，使用compile-modules convert命令，将ES6模块文件转码。

$ compile-modules convert file1.js file2.js

-o参数可以指定转码后的文件名。

$ compile-modules convert -o out.js file1.js

**SystemJS**

另一种解决方法是使用SystemJS。它是一个垫片库（polyfill），可以在浏览器内加载ES6模块、AMD模块和CommonJS模块，将其转为ES5格式。它在后台调用的是Google的Traceur转码器。

使用时，先在网页内载入system.js文件。

<script src="system.js"></script>

然后，使用System.import方法加载模块文件。

<script>

System.import('./app');

</script>

上面代码中的./app，指的是当前目录下的app.js文件。它可以是ES6模块文件，System.import会自动将其转码。

需要注意的是，System.import使用异步加载，返回一个Promise对象，可以针对这个对象编程。下面是一个模块文件。

然后，在网页内加载这个模块文件。

<script>

System.import('app/es6-file').then(function(m) {

console.log(new m.q().es6); // hello

});

</script>

上面代码中，System.import方法返回的是一个Promise对象，所以可以用then方法指定回调函数。