

# 0. 学习目标

---

- 能够按照文档安装Node.js
- 能够使用npm安装组件
- 能够使用webpack打包js
- 能够说出es6中let和const变量的区别
- 使用解构表达式处理对象属性
- 能够使用export导出模块文件
- 能够使用import导入模块文件

## 1. Node.js

---

### 1.1. 什么是Node.js

---

简单的说 Node.js 就是运行在服务端的 JavaScript。

Node.js 是一个基于Chrome JavaScript 运行时建立的一个平台。

Node.js是一个事件驱动I/O服务端JavaScript环境，基于Google的V8引擎，V8引擎执行javascript的速度非常快，性能非常好。

### 1.2. Node.js安装

---

1、下载对应你系统的Node.js版本:

<https://nodejs.org/en/download/> 资料 文件夹中已经提供。

2、选安装目录进行安装

推荐下载LTS版本。安装请参考 资料\NodeJS安装说明.pdf

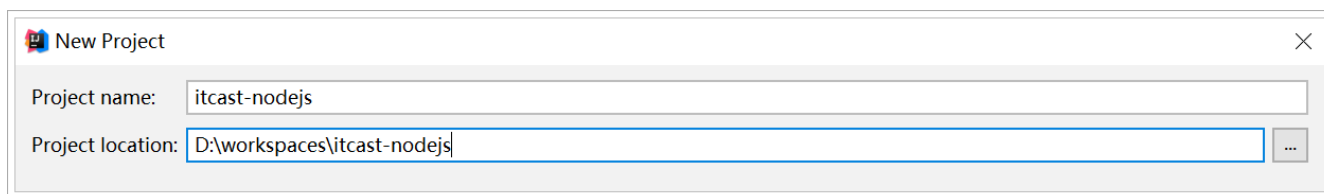
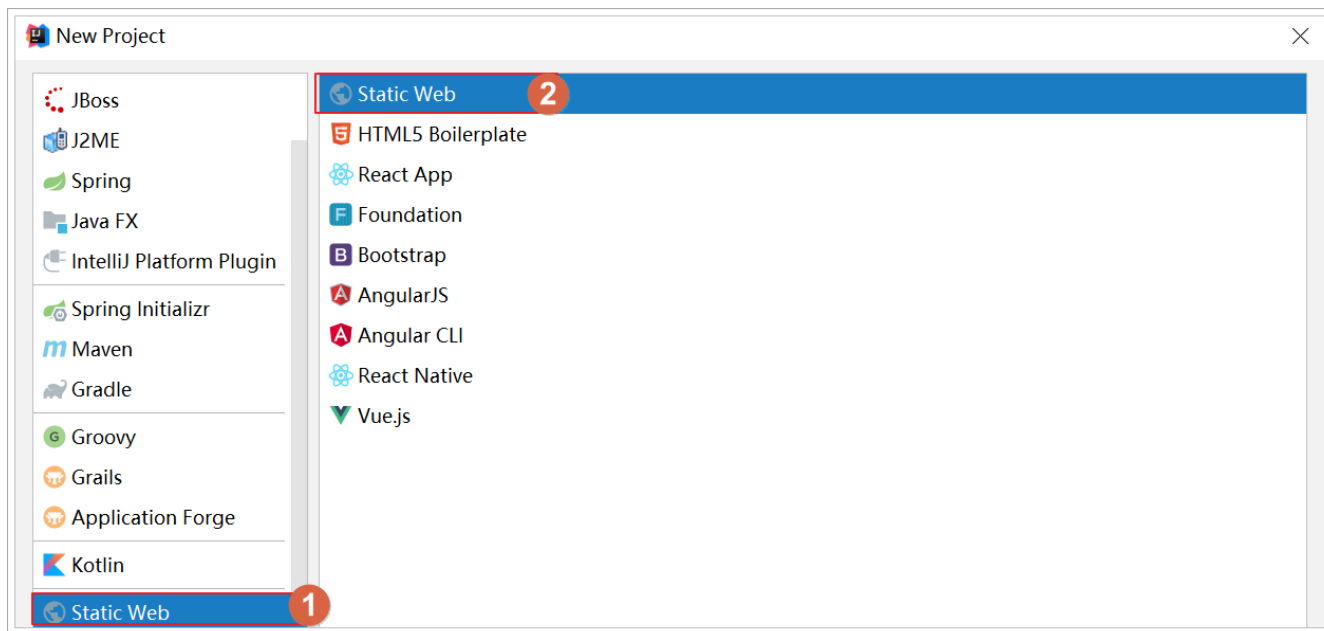
完成以后，在控制台输入：

```
1 # 查看node版本信息
2 node -v
```

### 1.3. 快速入门

---

#### 1.3.1. 创建测试工程



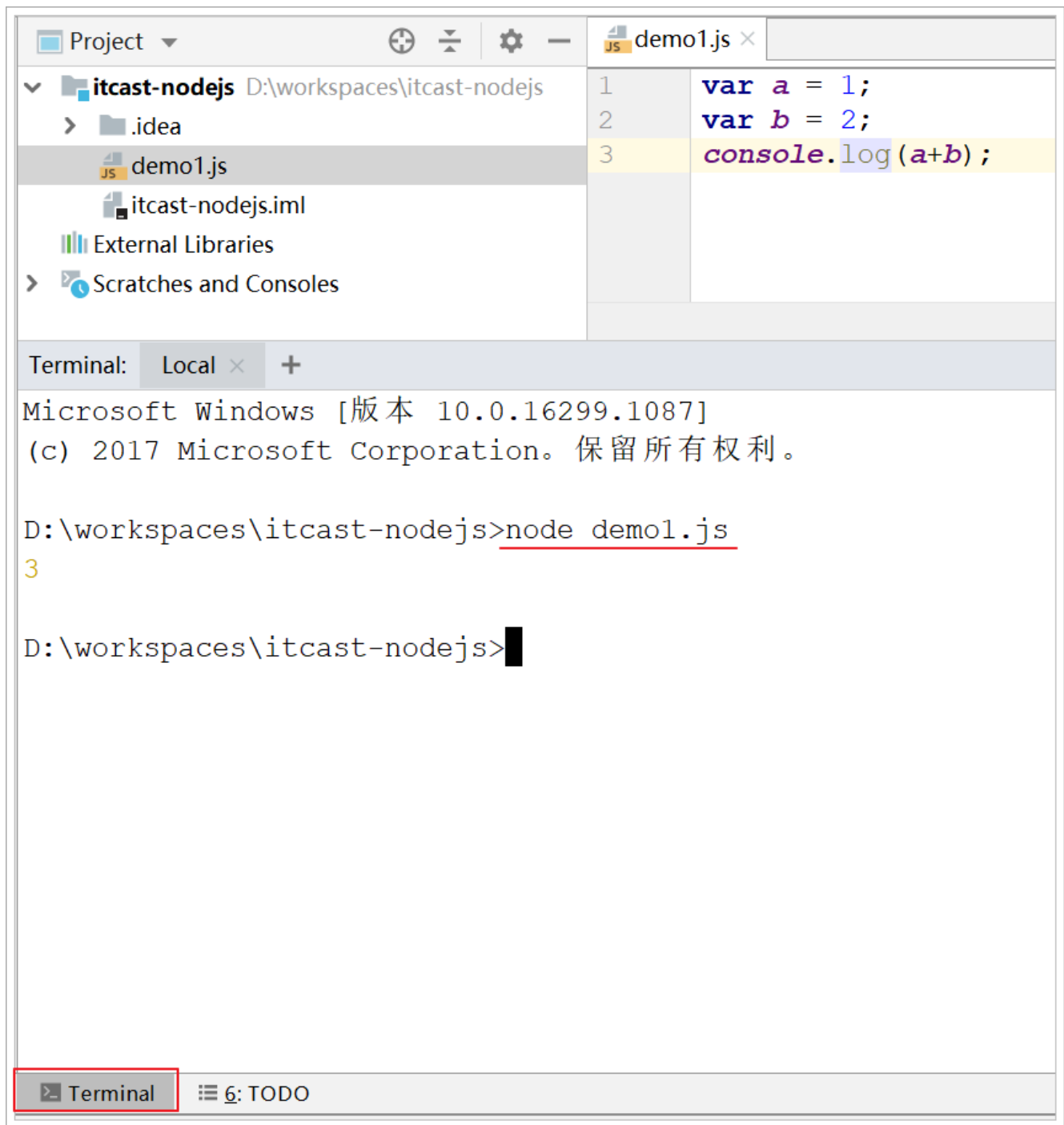
### 1.3.2. 控制台输出

现在做个最简单的小例子，演示如何在控制台输出，创建文本文件demo1.js,代码内容

```
1 var a=1;
2 var b=2;
3 console.log(a+b);
```

在命令提示符下输入命令

```
1 node demo1.js
```



### 1.3.3. 使用函数

创建文本文件demo2.js

```
1 var c=add(100,200);
2 console.log(c);
3 function add(a,b){
4     return a+b;
5 }
```

命令提示符输入命令

```
1 | node demo2.js
```

运行后看到输出结果为300

### 1.3.4. 模块化编程

每个文件就是一个模块，有自己的作用域。在一个文件里面定义的变量、函数、类，都是私有的，对其他文件不可见。

创建文本文件demo3\_1.js

```
1 |
2 | exports.add=function(a,b){
3 |     return a+b;
4 | }
```

每个模块内部，module变量代表当前模块。这个变量是一个对象，它的exports属性（即module.exports）是对外的接口。加载某个模块，其实是加载该模块的module.exports属性。

创建文本文件demo3\_2.js

```
1 | //引入模块demo3_1
2 | var demo= require('./demo3_1');
3 | console.log(demo.add(400,600));
```

在命令提示符下输入命令

```
1 | node demo3_2.js
```

结果为1000

### 1.3.5. 创建web服务器

创建文本文件demo4.js

```
1 | //http是内置模块
2 | var http = require('http');
3 | http.createServer(function (request, response) {
4 |     // 发送 HTTP 头部
5 |     // HTTP 状态值: 200 : OK
6 |     // 内容类型: text/plain
7 |     response.writeHead(200, {'Content-Type': 'text/plain'});
8 |     // 发送响应数据 "Hello world"
9 |     response.end('Hello world\n');
10 | }).listen(8888);
11 | // 终端打印如下信息
12 | console.log('Server running at http://127.0.0.1:8888/');
```

http为node内置的web模块

在命令提示符下输入命令

```
1 | node demo4.js
```

服务启动后，我们打开浏览器，输入网址

□ <http://localhost:8888/>

即可看到网页输出结果Hello World

在命令行中按 `Ctrl+c` 终止运行。

### 1.3.6. 理解服务端渲染

创建demo5.js，将上边的例子写成循环的形式

```
1 | var http = require('http');
2 | http.createServer(function (request, response) {
3 |     // 发送 HTTP 头部
4 |     // HTTP 状态值: 200 : OK
5 |     // 内容类型: text/plain
6 |     response.writeHead(200, {'Content-Type': 'text/plain'});
7 |     // 发送响应数据 "Hello world"
8 |     for(var i=0;i<10;i++){
9 |         response.write('Hello world\n');
10 |     }
11 |     response.end('');
12 | }).listen(8888);
13 | // 终端打印如下信息
14 | console.log('Server running at http://127.0.0.1:8888/');
```

在命令提示符下输入命令启动服务

```
1 | node demo5.js
```

浏览器地址栏输入<http://127.0.0.1:8888>即可看到查询结果。

右键“查看源代码”发现，并没有我们写的for循环语句，而是直接的10条Hello World，这就说明这个循环是在服务端完成的，而非浏览器（客户端）来完成。这与JSP很是相似。

### 1.3.7. 接收参数

创建demo6.js

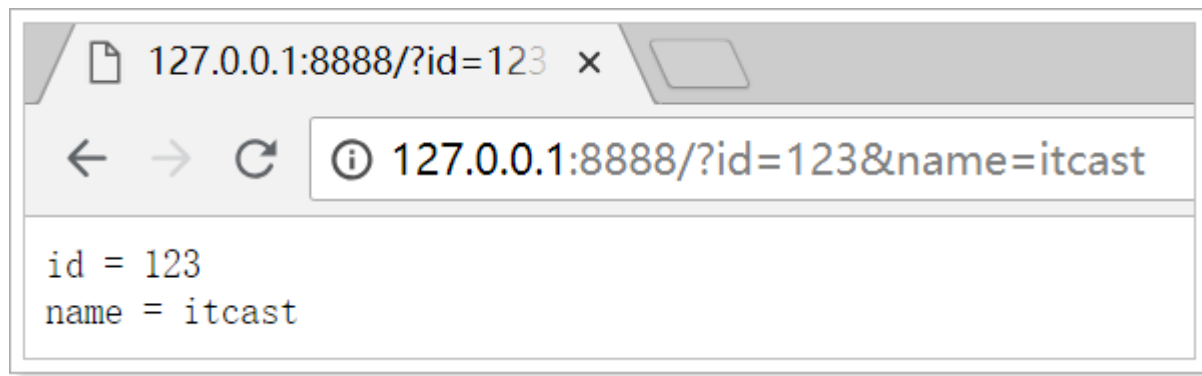
```
1 | //引入http模块
2 | var http = require("http");
3 | var url = require("url");
4 |
5 | //创建服务，监听8888端口
6 | http.createServer(function (request, response) {
```

```
7 //发送http头部
8 //http响应状态200
9 //http响应内容类型为text/plain
10 response.writeHead(200, {"Content-Type":"text/plain"});
11
12 //解析参数
13 //参数1: 请求地址;
14 //参数2: true时query解析参数为一个对象, 默认false
15 var params = url.parse(request.url, true).query;
16 //将所有请求参数输出
17 for (var key in params) {
18     response.write( key + " = " + params[key]);
19     response.write("\n");
20 }
21
22 response.end("");
23 }).listen(8888);
24
25 console.log("Server running at http://127.0.0.1:8888 ")
```

在命令提示符下输入命令

```
1 | node demo6.js
```

在浏览器访问 <http://127.0.0.1:8888?id=123&name=itcast> 测试结果:



## 2. 包资源管理器NPM

### 2.1. 什么是NPM

npm全称Node Package Manager，是node包管理和分发工具。其实我们可以把NPM理解为前端的Maven。

通过npm 可以很方便地下载js库，管理前端工程。

现在的node.js已经集成了npm工具，在命令提示符输入 npm -v 可查看当前npm版本

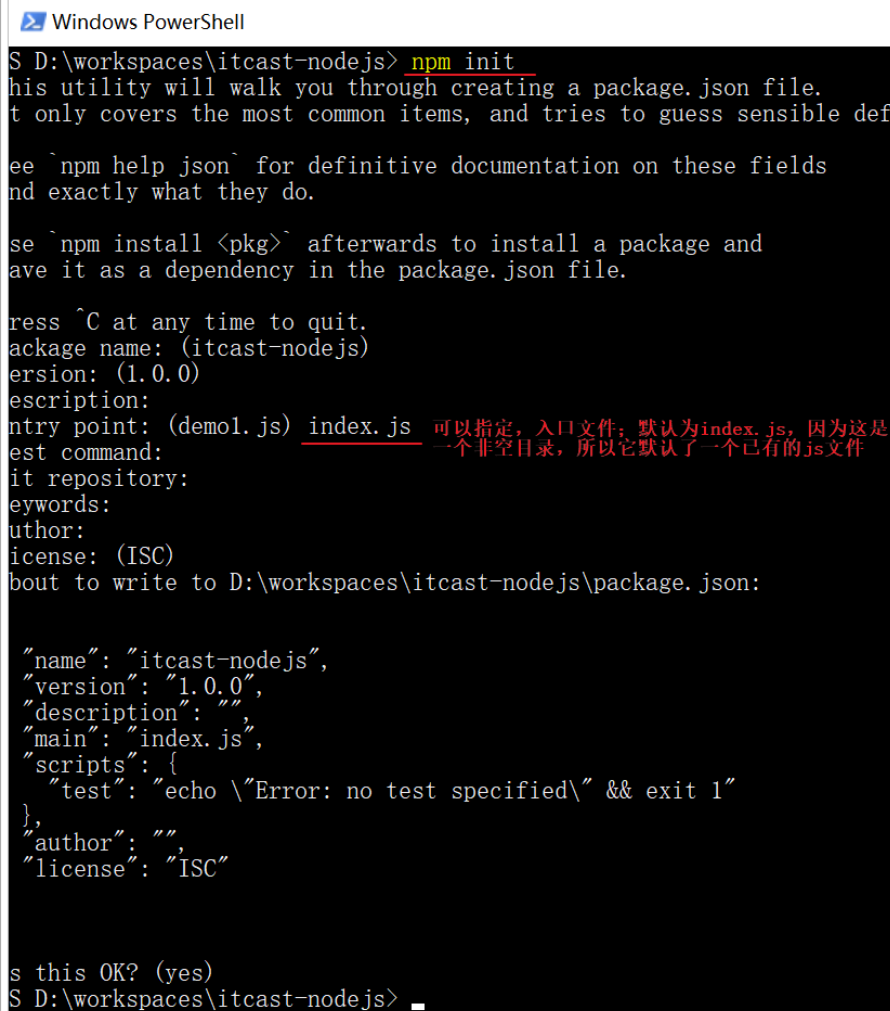
## 2.2. NPM命令

### 2.2.1. 初始化工程

init命令是工程初始化命令。

建立一个空文件夹或者在上述的示例工程中，在命令提示符进入该文件夹 执行命令初始化

```
1 | npm init
```



```
Windows PowerShell
S D:\workspaces\itcast-nodejs> npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Usage: `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (itcast-nodejs)
version: (1.0.0)
description:
entry point: (demo1.js) index.js 可以指定，入口文件；默认为index.js，因为这是一个非空目录，所以它默认了一个已有的js文件
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to D:\workspaces\itcast-nodejs\package.json:

{
  "name": "itcast-nodejs",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}

Is this OK? (yes)
S D:\workspaces\itcast-nodejs> _
```

按照提示输入相关信息，如果是用默认值则直接回车即可。

name: 项目名称

version: 项目版本号

description: 项目描述

keywords: {Array}关键词，便于用户搜索到我们的项目

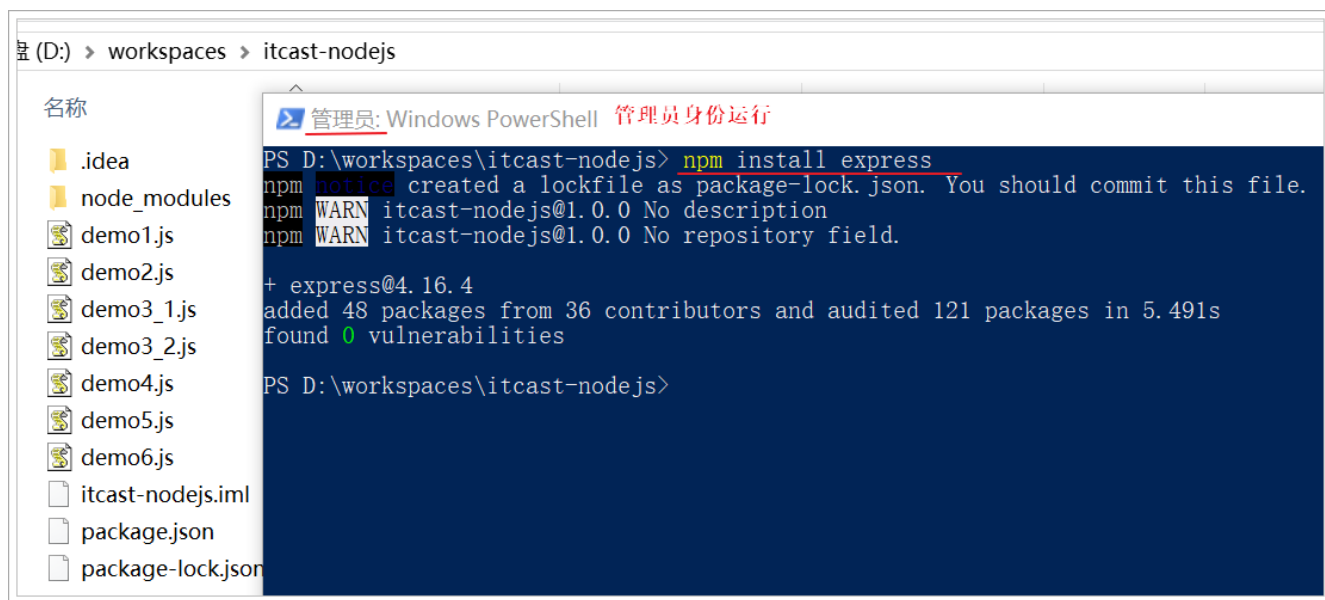
最后会生成 `package.json` 文件，这个是包的配置文件，相当于maven的pom.xml

之后也可以根据需要进行修改。

## 2.2.2. 本地安装

install命令用于安装某个模块，可以通过require引入到项目中使用。如我们想安装express模块（node的web框架），输出命令如下：

```
1 npm install express
```



```
PS D:\workspaces\itcast-nodejs> npm install express
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN itcast-nodejs@1.0.0 No description
npm WARN itcast-nodejs@1.0.0 No repository field.

+ express@4.16.4
added 48 packages from 36 contributors and audited 121 packages in 5.491s
found 0 vulnerabilities

PS D:\workspaces\itcast-nodejs>
```

出现警告信息，可以忽略，请放心，你已经成功执行了该命令。

在该目录下已经出现了一个node\_modules文件夹 和package-lock.json

node\_modules文件夹用于存放下载的js库（相当于maven的本地仓库）

package-lock.json是当 node\_modules 或 package.json 发生变化时自动生成的文件。这个文件主要功能是确定当前安装的包的依赖，以便后续重新安装的时候生成相同的依赖，而忽略项目开发过程中有些依赖已经发生的更新（可能存在切换了不同的镜像源后，同一个大版本号下可能出现兼容问题，package-lock可以保证即使换了源，下载的文件和原来的可以保持一致）。

我们再打开package.json文件，发现刚才下载的express已经添加到依赖列表中了。



```
{
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.16.4"
  }
}
```

关于版本号定义：



- 1 指定版本：比如1.2.2，遵循“大版本.次要版本.小版本”的格式规定，安装时只安装指定版本。
- 2
- 3 波浪号（tilde）+指定版本：比如~1.2.2，表示安装1.2.x的最新版本（不低于1.2.2），但是不安装1.3.x，也就是说安装时不改变大版本号 and 次要版本号。
- 4
- 5 插入号（caret）+指定版本：比如^1.2.2，表示安装1.x.x的最新版本（不低于1.2.2），但是不安装2.x.x，也就是说安装时不改变大版本号。需要注意的是，如果大版本号为0，则插入号的行为与波浪号相同，这是因为此时处于开发阶段，即使是次要版本号变动，也可能带来程序的不兼容。
- 6
- 7 latest：安装最新版本。

### 2.2.3. 全局安装

刚才我们使用的是本地安装，会将js库安装在当前目录，而使用全局安装会将库安装到你的全局目录下。全局安装之后可以在 命令行 使用该安装的模块对应的内容或命令。

如果你不知道你的全局目录在哪里，执行命令查看全局目录路径

```
1 npm root -g
```

默认全局目录在

C:\Users\Administrator\AppData\Roaming\npm\node\_modules

比如全局安装jquery，输入以下命令

```
1 # 安装之后在全局目录下会存在对应的jquery目录，其里面的dist则包含有对应的jquery.js文件
2 npm install jquery -g
```

 管理员: Windows PowerShell

```
PS D:\workspaces\itcast-nodejs> npm install jquery -g
+ jquery@3.4.1
added 1 package from 1 contributor in 1.357s
PS D:\workspaces\itcast-nodejs>
```

### 2.2.4. 批量下载

从网上下载某些代码，发现只有package.json，没有node\_modules文件夹，这时需要通过命令重新下载这些js库。

进入目录（package.json所在的目录）输入命令

```
1 npm install
```

此时，npm会自动下载package.json中依赖的js库。

### 2.2.5. 切换NPM镜像

有时我们使用npm下载资源会很慢，所以可以切换下载的镜像源（如：淘宝镜像）；或者安装一个cnpm(指定淘宝镜像)来加快下载速度。

1、如果使用切换镜像源的方式，可以使用一个工具：nrm

首先安装nrm，这里 -g 代表全局安装

```
1 # 管理员身份 打开cmd执行如下命令
2 npm install nrm -g
```

然后通过 `nrm ls` 命令查看npm的仓库列表,带\*的就是当前选中的镜像仓库：

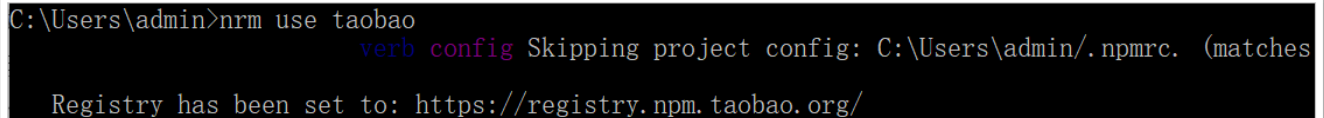


```
管理员: 命令提示符
Microsoft Windows [版本 10.0.16299.1087]
(c) 2017 Microsoft Corporation。保留所有权利。

C:\WINDOWS\system32>npm install nrm -g
npm WARN deprecated coffee-script@1.7.1: CoffeeScript on NPM has moved to "coffeescript" (no hyphen)
C:\Program Files\nodejs\npm_modules\nrm -> C:\Program Files\nodejs\npm_modules\node_modules\nrm\cli.js
+ nrm@1.1.0
added 324 packages from 564 contributors in 25.782s

C:\WINDOWS\system32>nrm ls
* npm ---- https://registry.npmjs.org/
  cnpm --- http://r.cnpmjs.org/
  taobao - https://registry.npm.taobao.org/
  nj ----- https://registry.nodejitsu.com/
  npmMirror https://skimdb.npmjs.com/registry/
  edunpm - http://registry.enpmjs.org/
```

通过 `nrm use taobao` 来指定要使用的镜像源：



```
C:\Users\admin>nrm use taobao
verb config Skipping project config: C:\Users\admin/.npmrc. (matches
Registry has been set to: https://registry.npm.taobao.org/
```

2、如果使用cnpm的方式，则先安装cnpm，输入如下命令

```
1 # 如果不使用nrm 切换，可以在安装cnpm的时候指定镜像仓库
2 npm install -g cnpm --registry=https://registry.npm.taobao.org
```

安装后，我们可以使用以下命令来查看cnpm的版本

```
1 cnpm -v
```

使用cnpm

```
1 cnpm install 需要下载的js库；一般只有在下载模块的时候才使用cnpm，其它情况还是一样使用npm；
```

## 2.2.6. 运行工程说明

如果我们想运行某个工程，则使用run命令

如果package.json中定义的脚本中有：

- dev是开发阶段测试运行
- build是构建编译工程
- lint 是运行js代码检测

运行时命令格式：

```
1 | npm run dev或者build或者lint
```

## 2.2.7. 编译工程说明

编译后的代码会放在dist文件夹中，进入命令提示符输入命令

```
1 | npm run build
```

生成后会发现只有个静态页面，和一个static文件夹

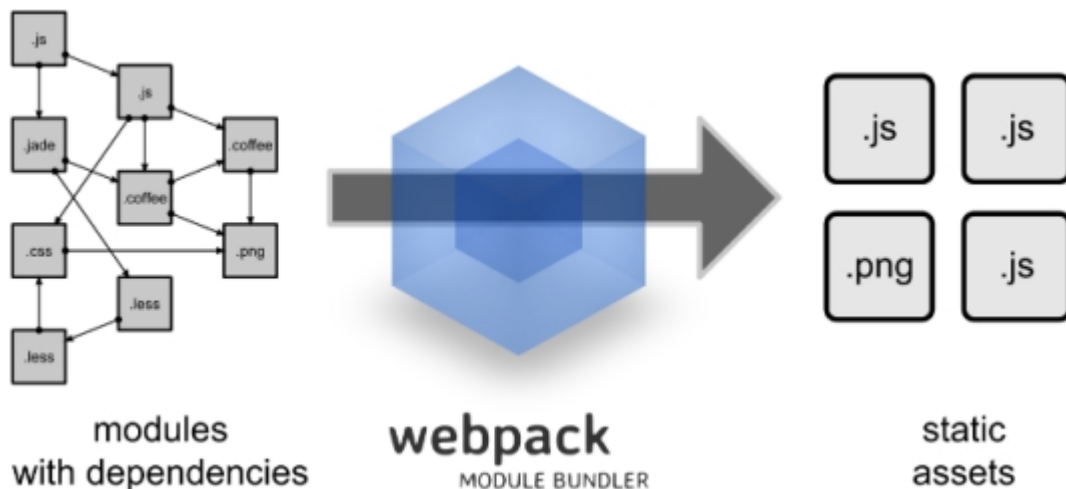
这种工程我们称之为单页Web应用（single page web application，SPA），就是只有一张Web页面的应用，是加载单个HTML 页面并在用户与应用程序交互时动态更新该页面的Web应用程序。

这里其实是调用了webpack来实现打包的，关于webpack下面的章节将进行介绍。

# 3. Webpack入门

## 3.1. 什么是Webpack

Webpack 是一个前端资源加载/打包工具。它将根据模块的依赖关系进行静态分析，然后将这些模块按照指定的规则生成对应的静态资源。[webpackjs](http://webpack.js.org)



从图中我们可以看出，Webpack 可以将多种静态资源 js、css 等转换成一个静态文件，减少了页面的请求。接下来简单为大家介绍 Webpack 的安装与使用。

## 3.2. Webpack安装

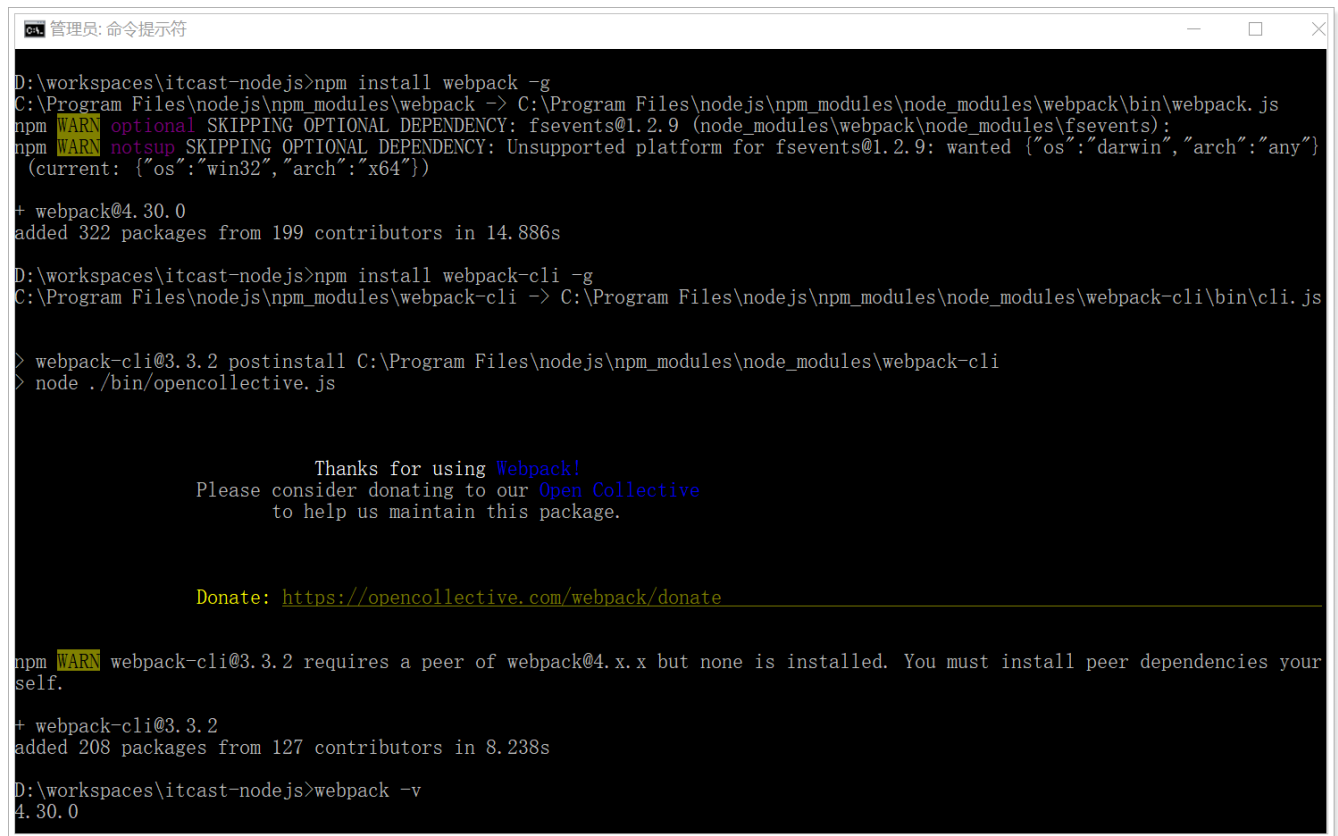
全局安装

```
1 | npm install webpack -g
2 | npm install webpack-cli -g
```

如果安装失败；则将全局目录下的webpack的相关文件夹删除再执行上述命令

安装后查看版本号

```
1 | webpack -v
```



```
管理员: 命令提示符
D:\workspaces\itcast-nodejs>npm install webpack -g
C:\Program Files\nodejs\npm_modules\webpack -> C:\Program Files\nodejs\npm_modules\node_modules\webpack\bin\webpack.js
npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@1.2.9 (node_modules\webpack\node_modules\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@1.2.9: wanted {"os":"darwin","arch":"any"}
(current: {"os":"win32","arch":"x64"})
+ webpack@4.30.0
added 322 packages from 199 contributors in 14.886s
D:\workspaces\itcast-nodejs>npm install webpack-cli -g
C:\Program Files\nodejs\npm_modules\webpack-cli -> C:\Program Files\nodejs\npm_modules\node_modules\webpack-cli\bin\cli.js
> webpack-cli@3.3.2 postinstall C:\Program Files\nodejs\npm_modules\node_modules\webpack-cli
> node ./bin/opencollective.js

      Thanks for using Webpack!
      Please consider donating to our Open Collective
      to help us maintain this package.

      Donate: https://opencollective.com/webpack/donate

npm WARN webpack-cli@3.3.2 requires a peer of webpack@4.x.x but none is installed. You must install peer dependencies your
self.
+ webpack-cli@3.3.2
added 208 packages from 127 contributors in 8.238s
D:\workspaces\itcast-nodejs>webpack -v
4.30.0
```

## 3.3. 快速入门

### 3.3.1. JS打包

(1) 创建src文件夹，创建bar.js

```
1 exports.info=function(str){
2     document.write(str);
3 }
```

(2) src下创建logic.js

```
1 exports.add=function(a,b){
2     return a+b;
3 }
```

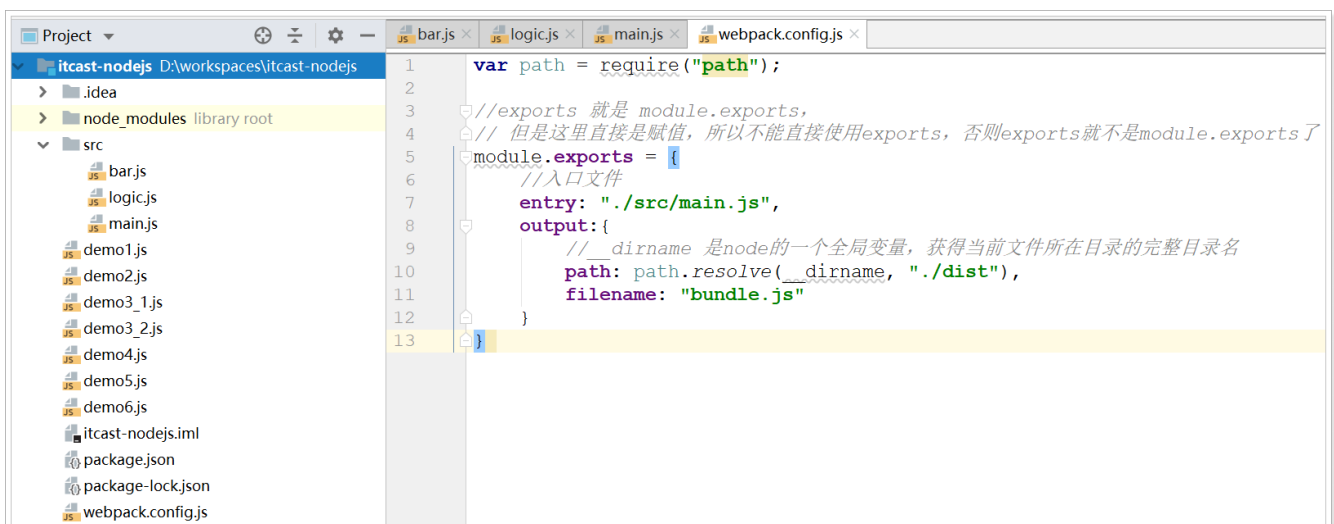
(3) src下创建main.js

```
1 var bar= require('./bar');
2 var logic= require('./logic');
3 bar.info( '100 + 200 = ' + logic.add(100,200));
```

(4) 创建配置文件webpack.config.js，该文件与src处于同级目录

```
1 var path = require("path");
2
3 //exports 就是 module.exports,
4 // 但是这里直接是赋值，所以不能直接使用exports，否则exports就不是module.exports了
5 module.exports = {
6     //入口文件
7     entry: "./src/main.js",
8     output:{
9         //__dirname 是node的一个全局变量，获得当前文件所在目录的完整目录名
10        path: path.resolve(__dirname, "./dist"),
11        filename: "bundle.js"
12    }
13 }
```

以上代码的意思是：读取当前目录下src文件夹中的main.js（入口文件）内容，把对应的js文件打包，打包后的文件放入当前目录的dist文件夹下，打包后的js文件名为bundle.js



(5) 执行编译命令

执行后查看bundle.js 会发现里面包含了上面两个js文件的内容

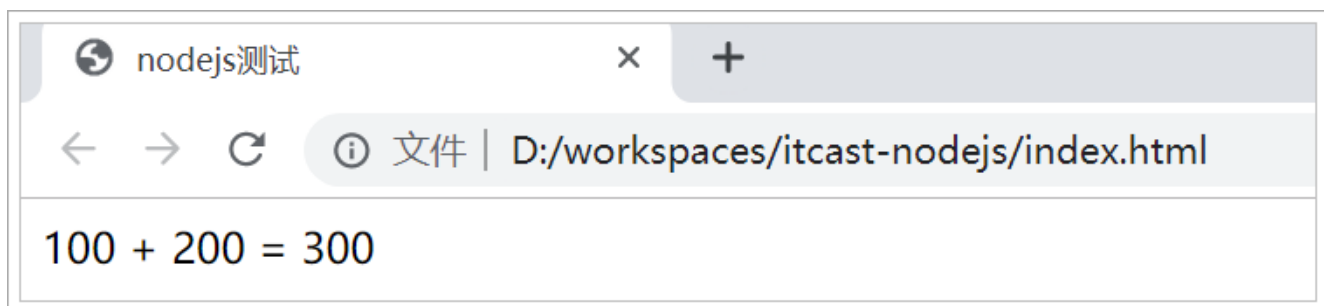
```
Windows PowerShell
PS D:\workspaces\itcast-nodejs> webpack
Hash: 47a2f4e659857f749757
Version: webpack 4.30.0
Time: 113ms
Built at: 2019-05-07 16:19:23
    Asset      Size  Chunks             Chunk Names
bundle.js  1.06 KiB       0  [emitted]  main
Entrypoint main = bundle.js
[0] ./src/main.js 111 bytes {0} [built]
[1] ./src/bar.js 57 bytes {0} [built]
[2] ./src/logic.js 52 bytes {0} [built]

WARNING in configuration
The 'mode' option has not been set, webpack will fallback to 'production' for this value. Set 'mode'
option to 'development' or 'production' to enable defaults for each environment.
You can also set it to 'none' to disable any default behavior. Learn more: https://webpack.js.org/co
ncepts/mode/
PS D:\workspaces\itcast-nodejs>
```

(7) 创建index.html,引用bundle.js

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <title>nodejs测试</title>
6   <script src="dist/bundle.js"></script>
7 </head>
8 <body>
9
10 </body>
11 </html>
```

浏览器上访问index.html文件, 会有内容输出。



### 3.3.2. CSS打包

(1) 安装style-loader和 css-loader

Webpack 本身只能处理 JavaScript 模块, 如果要处理其他类型的文件, 就需要使用 loader 进行转换。

Loader 可以理解为是模块和资源的转换器，它本身是一个函数，接受源文件作为参数，返回转换的结果。这样，我们就可以通过 require 来加载任何类型的模块或文件，比如 CoffeeScript、JSX、LESS 或图片。首先我们需要安装相关Loader插件，css-loader 是将 css 装载到 javascript；style-loader 是让 javascript 认识css

```
1 | cnpm install style-loader css-loader --save-dev
```

```
PS D:\workspaces\itcast-nodejs> cnpm install style-loader css-loader --save-dev
√ Installed 2 packages
√ Linked 36 latest versions
√ Run 0 scripts
peerDependencies WARNING css-loader@* requires a peer of webpack@^4.0.0 but none was installed
Recently updated (since 2019-04-30): 1 packages (detail see file D:\workspaces\itcast-nodejs\node_modules\recently_updates.txt)
√ All packages installed (39 packages installed from npm registry, used 968ms(network 944ms), speed 403.04kB/s, json 38(95.4kB), tarball 285.06kB)
PS D:\workspaces\itcast-nodejs>
```

-save 的意思是将模块安装到项目目录下，并在package文件的dependencies节点写入依赖。运行npm install --production或者注明NODE\_ENV变量值为production时，会自动下载模块到node\_modules目录中。

-save-dev 的意思是将模块安装到项目目录下，并在package文件的devDependencies节点写入依赖。运行 npm install --production或者注明NODE\_ENV变量值为production时，不会自动下载模块到node\_modules目录中。

```
1 | cnpm install less less-loader --save-dev
```

```
PS D:\workspaces\itcast-nodejs> cnpm install less less-loader --save-dev
√ Installed 2 packages
√ Linked 63 latest versions
√ Run 0 scripts
peerDependencies WARNING less-loader@* requires a peer of webpack@^2.0.0 || ^3.0.0 || ^4.0.0 but none was installed
Recently updated (since 2019-06-17): 1 packages (detail see file D:\workspaces\itcast-nodejs\node_modules\recently_updates.txt)
√ All packages installed (67 packages installed from npm registry, used 7s(network 7s), speed 139.7kB/s, json 65(143.3kB), tarball 849.63kB)
```

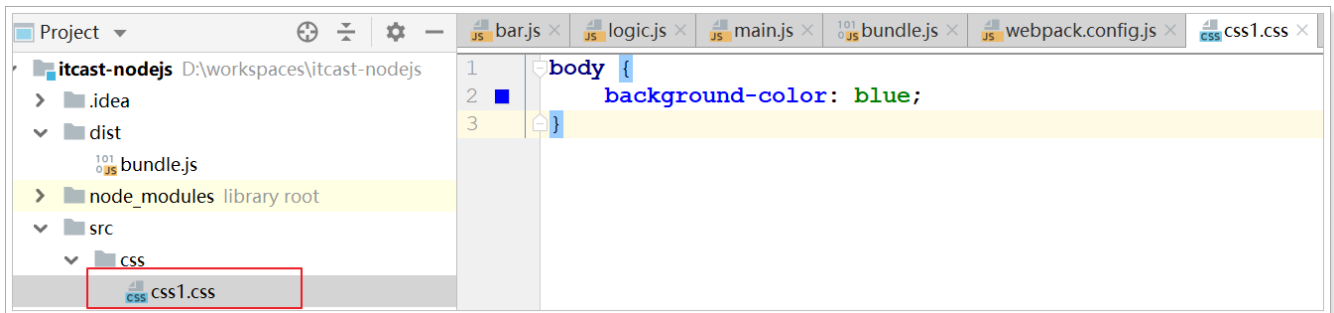
## (2) 修改webpack.config.js

```
1 | var path = require("path");
2 |
3 | //exports 就是 module.exports,
4 | // 但是这里直接是赋值，所以不能直接使用exports，否则exports就不是module.exports了
5 | module.exports = {
6 |   //入口文件
7 |   entry: "./src/main.js",
8 |   output:{
9 |     //__dirname 是node的一个全局变量，获得当前文件所在目录的完整目录名
10 |    path: path.resolve(__dirname, "./dist"),
11 |    filename: "bundle.js"
12 |  },
13 |   module:{
14 |     rules:[
15 |       {
16 |         test: /\.css$/,
17 |         use: ["style-loader", "css-loader"]
18 |       }
19 |     ]
20 |   }
21 | }
```

```
20 }
21 }
```

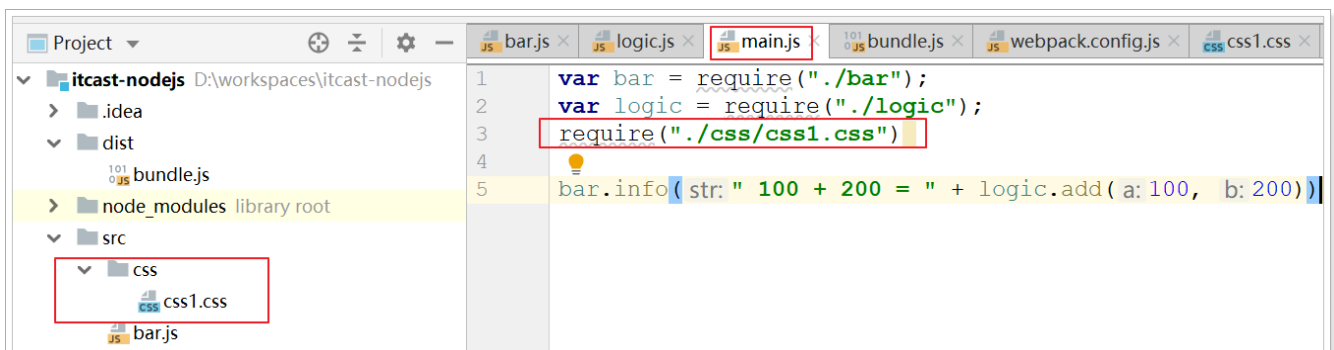
(3) 在src文件夹创建css文件夹,css文件夹下创建css1.css

```
1 body {
2     background-color: blue;
3 }
```

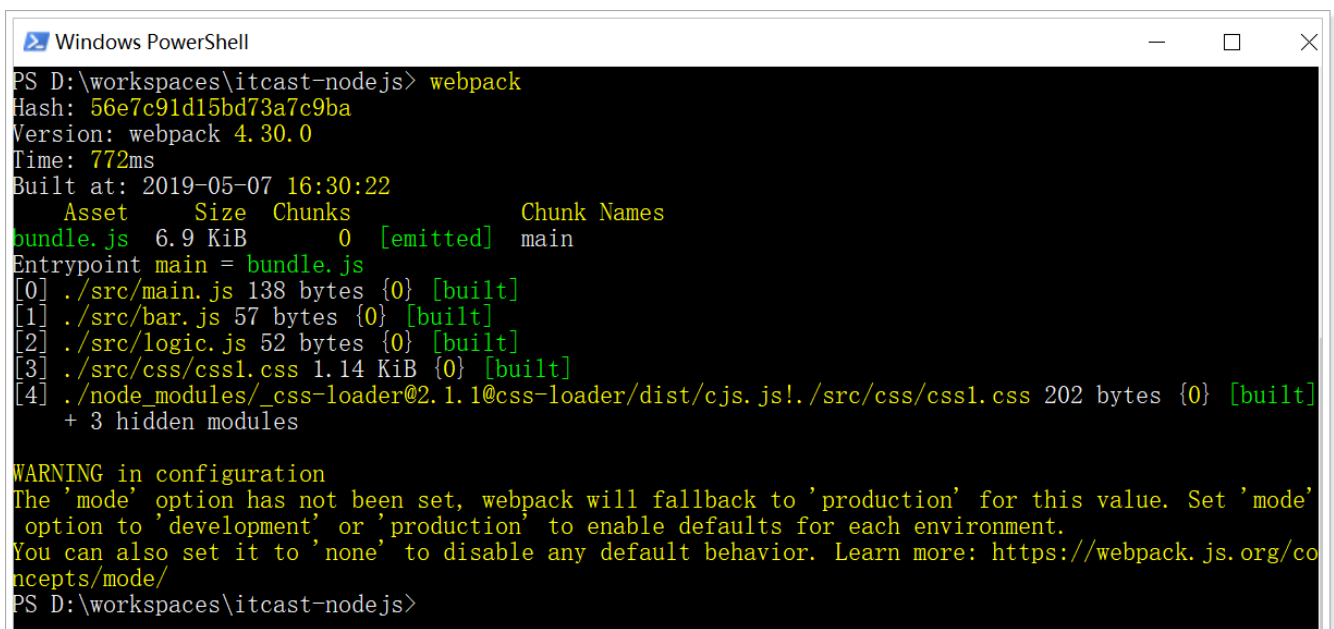


(4) 修改main.js , 引入css1.css

```
1 require('./css/css1.css');
```

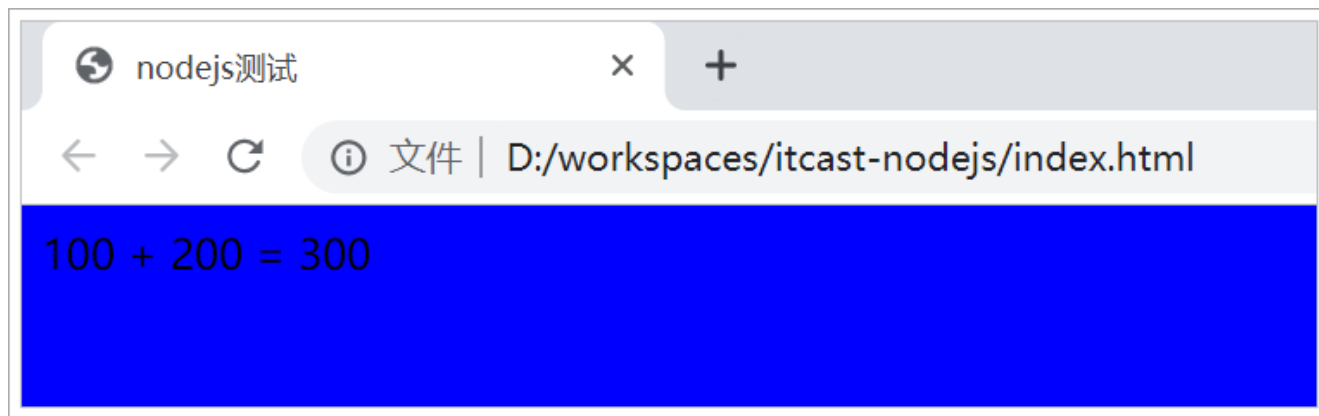


(5) 重新运行webpack





(6) 运行index.html看看背景是不是变成蓝色？



## 4. ES6

ES6? 就是ECMAScript第6版标准。

### 4.1. 什么是ECMAScript?

编程语言JavaScript是ECMAScript的实现和扩展。ECMAScript是由ECMA（一个类似W3C的标准组织）参与进行标准化的语法规范。ECMAScript定义了：

[语言语法](#) – 语法解析规则、关键字、语句、声明、运算符等。

[类型](#) – 布尔型、数字、字符串、对象等。

[原型和继承](#)

内建对象和函数的[标准库](#) – [JSON](#)、[Math](#)、[数组方法](#)、[对象自省方法](#)等。

ECMAScript标准不定义HTML或CSS的相关功能，也不定义类似DOM（文档对象模型）的[Web API](#)，这些都在独立的标准中进行定义。ECMAScript涵盖了各种环境中JS的使用场景，无论是浏览器环境还是类似[node.js](#)的非浏览器环境。

ECMAScript标准的历史版本分别是1、2、3、5。

那么为什么没有第4版？其实，在过去确实曾计划发布提出巨量新特性的第4版，但最终却因想法太过激进而惨遭废除（这一版标准中曾经有一个极其复杂的支持泛型和类型推断的内建静态类型系统）。

ES4饱受争议，当标准委员会最终停止开发ES4时，其成员同意发布一个相对谦和的ES5版本，随后继续制定一些更具实质性的新特性。这一明确的协商协议最终命名为“Harmony”，因此，ES5规范中包含这样两句话

ECMAScript是一门充满活力的语言，并在不断进化中。

未来版本的规范中将持续进行重要的技术改进

2009年发布的改进版本ES5，引入了[Object.create\(\)](#)、[Object.defineProperty\(\)](#)、[getters](#)和[setters](#)、[严格模式](#)以及[JSON](#)对象。

ECMAScript 6.0（以下简称ES6）是JavaScript语言的下一代标准，2015年6月正式发布。它的目标，是使得JavaScript语言可以用来编写复杂的大型应用程序，成为企业级开发语言。

## 4.2. ECMAScript的快速发展

而后，ECMAScript就进入了快速发展期。

- 1998年6月，ECMAScript 2.0 发布。
- 1999年12月，ECMAScript 3.0 发布。这时，ECMAScript 规范本身也相对比较完善和稳定了，但是接下来的事情，就比较悲剧了。
- 2007年10月。。。ECMAScript 4.0 草案发布。

这次的新规范，历时颇久，规范的新内容也有了很多人争议。在制定ES4的时候，是分成了两个工作组同时工作的。

- 一边是以 Adobe, Mozilla, Opera 和 Google为主的 ECMAScript 4 工作组。
- 一边是以 Microsoft 和 Yahoo 为主的 ECMAScript 3.1 工作组。

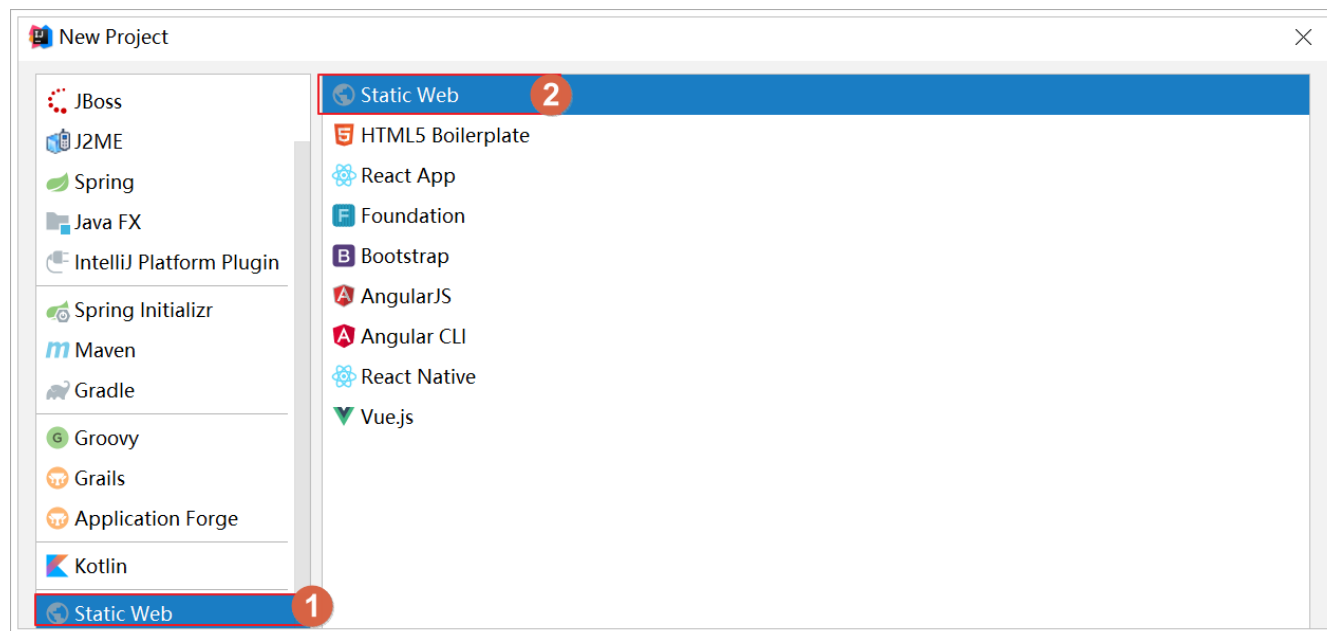
ECMAScript 4 的很多主张比较激进，改动较大。而 ECMAScript 3.1 则主张小幅更新。最终经过 TC39 的会议，决定将一部分不那么激进的改动保留发布为 ECMAScript 3.1，而ES4的内容，则延续到了后来的ECMAScript5和6版本中

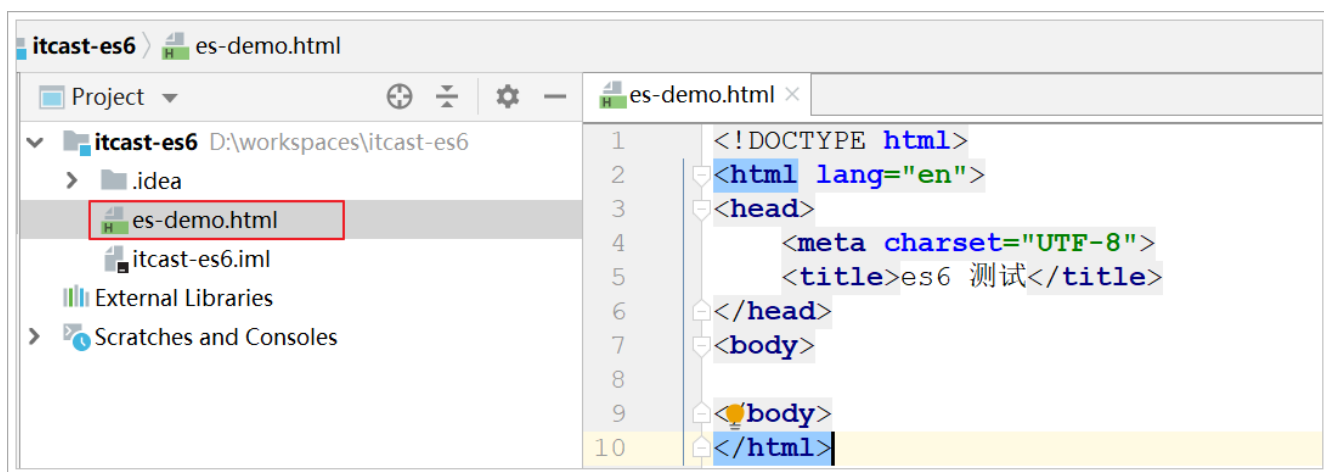
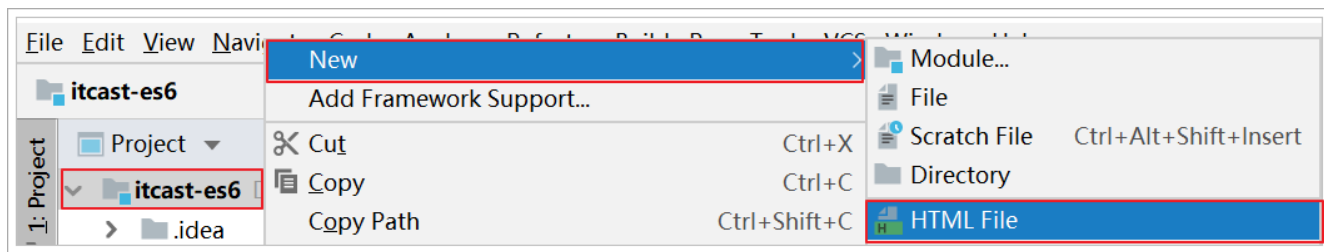
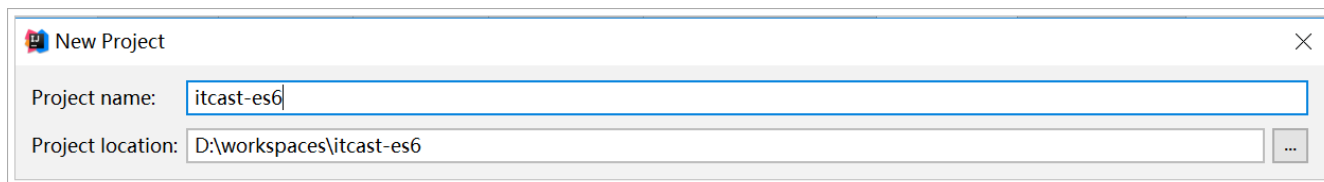
- 2009年12月，ECMAScript 5 发布。
- 2011年6月，ECMAScript 5.1 发布。
- 2015年6月，ECMAScript 6，也就是 ECMAScript 2015 发布了。并且从 ECMAScript 6 开始，开始采用年号来做版本。即 ECMAScript 2015，就是ECMAScript6。

## 4.3. ES6的一些新特性

这里只把一些常用的进行学习，更详细的大家参考：[阮一峰的ES6教程](#)

### 4.3.1. 创建测试工程





### 4.3.2. let 和 const 命令

var

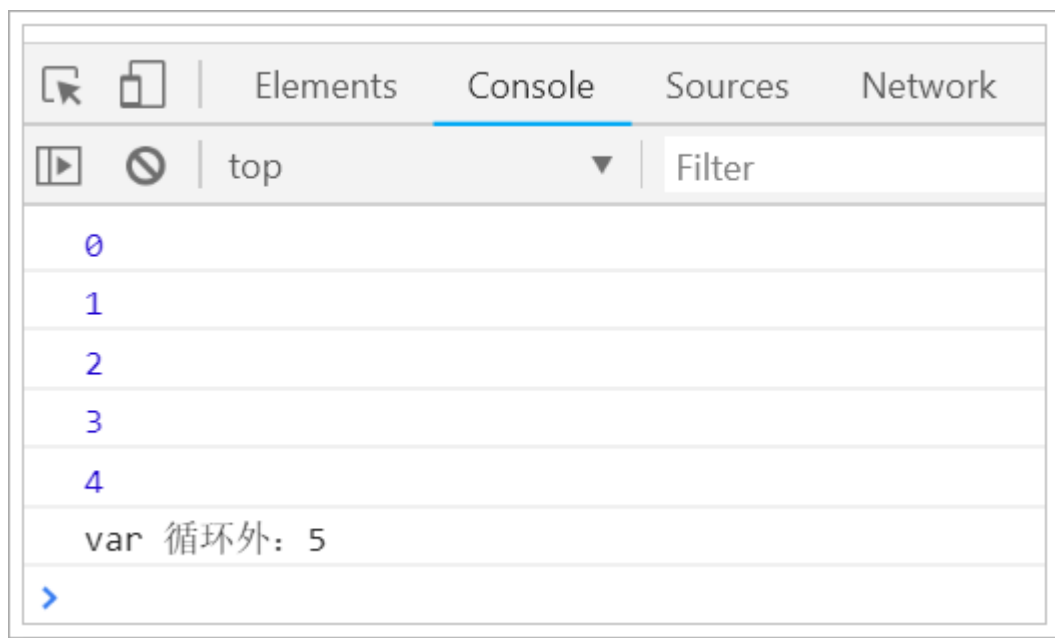
之前, js定义变量只有一个关键字: var

var 有一个问题, 就是定义的变量有时会莫名其妙的成为全局变量。

例如这样的代码:



打印的结果：



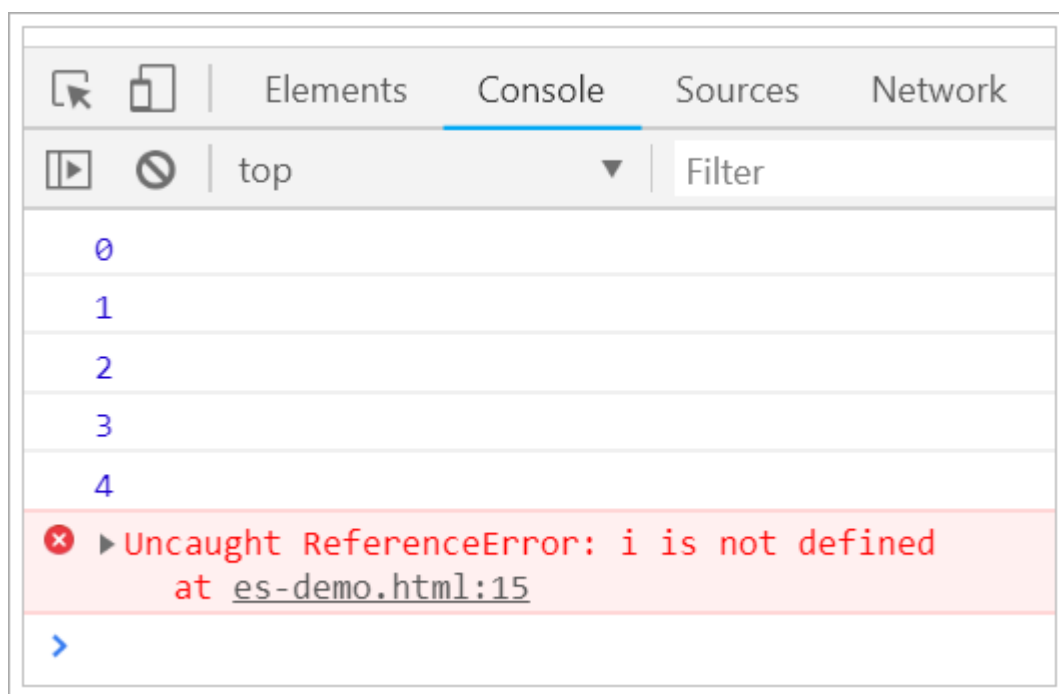
let

let 所声明的变量，只在 let 命令所在的代码块内有效。

把刚才的 var 改成 let 试试：

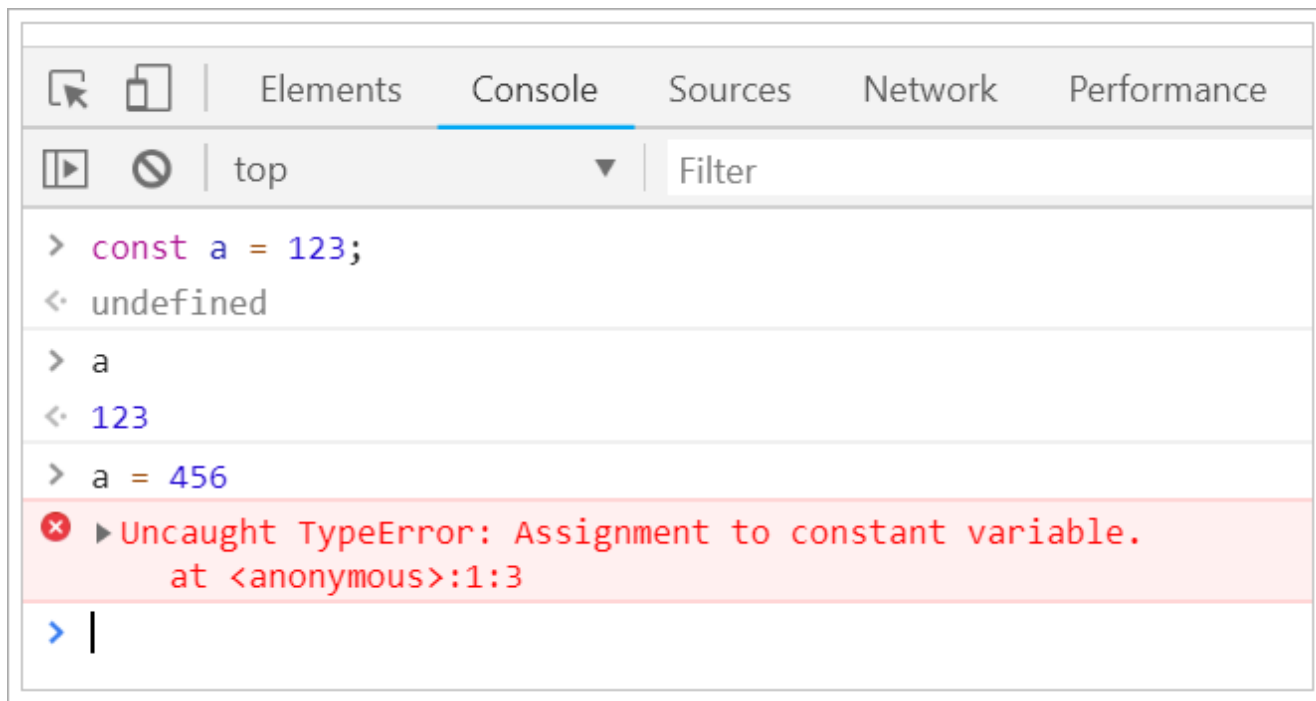
```
1 for(let i = 0; i < 5; i++){  
2   console.log(i);  
3 }  
4 console.log("let 循环外: " + i)
```

结果：



const

`const` 声明的变量是常量，不能被修改；在浏览器控制台进行如下操作：



### 4.3.3. 模板字符串

第一个用途，基本的字符串格式化。将表达式嵌入字符串中进行拼接。用`${}`来界定。

```
1 //es5
2 let name = 'itcast'
3 console.log('hello ' + name)
4 //es6
5 const name = 'itcast'
6 console.log(`hello ${name}`) //hello itcast
```

第二个用途，在ES5时我们通过反斜杠`()`来做多行字符串或者字符串一行行拼接。ES6反引号```直接搞定。

```
1 // es5
2 var msg = "Hi \
3 man!"
4 // es6
5 const template = `

6   <span>hello world</span>
7 </div>`


```

### 4.3.4. 对象初始化简写

ES5对于对象都是以键值对的形式书写，是有可能出现键值对重名的。例如

```
1 function person5(name, age) {
2     return {name:name, age:age};
3 }
4 console.log("es5 => " + JSON.stringify(person5("itcast", 13)))
5
```

以上代码可以简写为

```
1 function person6(name, age) {
2     //属性和值是同名的则可以省略为如下:
3     return {name, age}
4 }
5 console.log("es6 => " + JSON.stringify(person6("itcast", 13)))
```

## 4.3.5. 解构表达式

### 1) 数组解构

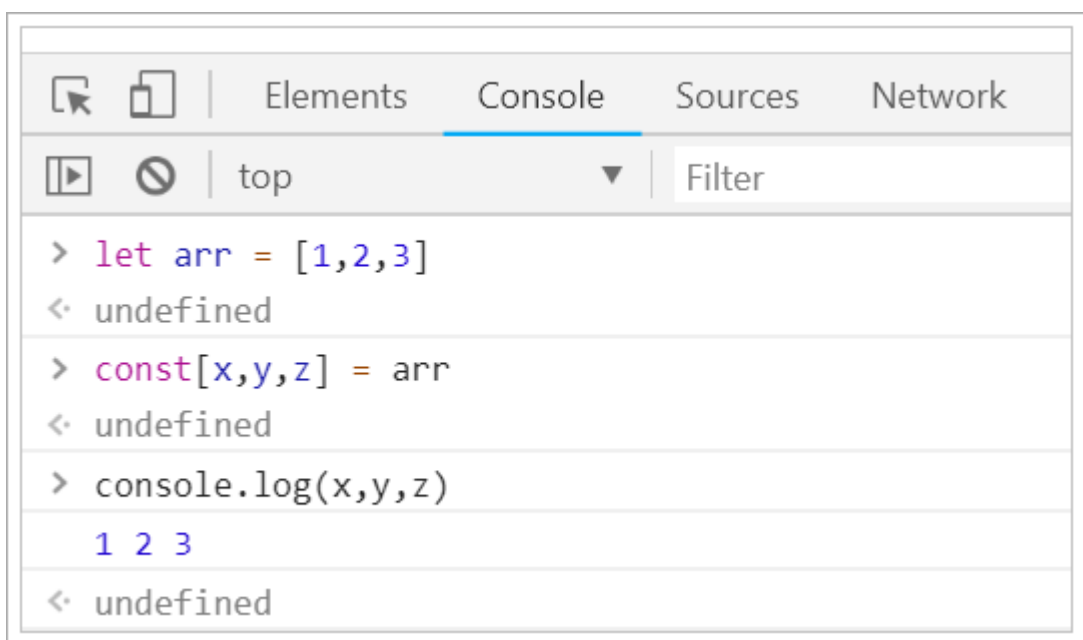
比如有一个数组:

```
1 let arr = [1,2,3]
```

想获取其中的值, 只能通过角标。ES6可以这样:

```
1 const [x,y,z] = arr; // x, y, z将与arr中的每个位置对应来取值
2 // 然后打印
3 console.log(x,y,z);
```

结果:



## 2) 对象解构

例如有个person对象：

```
1  const person = {  
2    name: "jack",  
3    age: 21,  
4    language: ['java', 'js', 'css']  
5  }
```

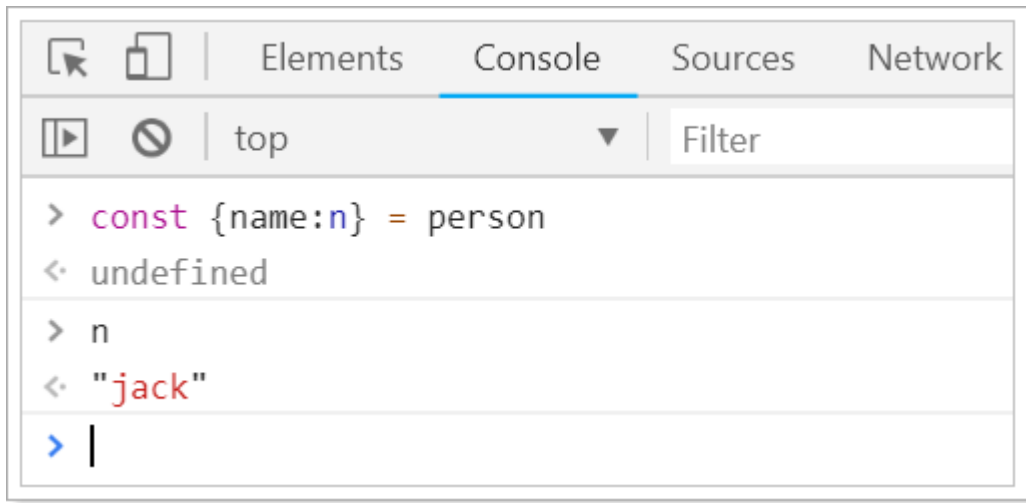
可以这么做：

```
1  // 解构表达式获取值  
2  const {name, age, language} = person;  
3  // 打印  
4  console.log(name);  
5  console.log(age);  
6  console.log(language);
```

结果：



如果想要用其它变量接收，需要额外指定别名：



- `{name:n}`: `name`是`person`中的属性名，冒号后面的`n`是解构后要赋值给的变量。

### 3) 拷贝对象属性

比如有一个`person`对象：

```
1  const person = {  
2    name: "jack",  
3    age: 21,  
4    language: ['java', 'js', 'css']  
5  }
```

想获取它的 `name` 和 `age` 属性，封装到新的对象，该怎么办？



在解构表达式中，通过 `language` 接收到语言，剩下的所有属性用 `... obj` 方式，可以一起接收，这样 `obj` 就是一个新的对象，包含了 `person` 中，除了 `language` 外的所有其它属性



数组也可以采用类似操作。

## 4.3.6. 函数优化

### 1) 箭头函数

ES6中定义函数的简写方式：

在es-demo.html中测试；一个参数时：

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <title>es6 测试</title>
6      <script type="text/javascript">
7          /*for(var i = 0; i < 5; i++){
8              console.log(i);
9          }
10         console.log("var 循环外: " + i)*/
11     /*
12         for(let i = 0; i < 5; i++){
13             console.log(i);
14         }
15         console.log("let 循环外: " + i)
16     */
17     /*
18         let name = "itcast";
19         console.log("hello " + name);
20
21         //es 6
22         console.log(`es6 hello ${name}`);
23     */
24     /*
25         //es 5
26         const str = "hello " +
27             "itcast";
28         //es 6
29         const template = `<div>
30             <span>hello itcast</span>
31             </div>`
32     */
33     /*
34         function person5(name, age) {
35             return {name:name, age:age};
36         }
37         console.log("es5 => " + JSON.stringify(person5("itcast", 13)))
38
39         function person6(name, age) {
40             return {name, age}
41         }
42         console.log("es6 => " + JSON.stringify(person6("itcast", 13)))
43     */
```

```

44     var print = function (obj) {
45         console.log(obj);
46     };
47     print("print");
48
49     var print2 = obj => console.log(obj);
50     print2("print2");
51
52     </script>
53 </head>
54 <body>
55
56 </body>
57 </html>

```

多个参数:

```

1 // 两个参数的情况:
2 var sum = function (a , b) {
3     return a + b;
4 }
5 console.log(sum(1, 2));
6 // 简写为:
7 var sum2 = (a,b) => a+b;
8 console.log(sum2(1, 2));

```

代码不止一行, 可以用 {} 括起来

```

1 var sum3 = (a,b) => {
2     console.log(a+b)
3     return a + b;
4 }
5 console.log(sum3(1, 2));

```

## 2) 对象的函数属性简写

比如一个Person对象, 里面有eat方法:

```

1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4     <meta charset="UTF-8">
5     <title>es6 测试</title>
6     <script type="text/javascript">
7         /*for(var i = 0; i < 5; i++){
8             console.log(i);
9         }
10         console.log("var 循环外: " + i)*/
11     /*
12         for(let i = 0; i < 5; i++){

```

```
13         console.log(i);
14     }
15     console.log("let 循环外: " + i)
16 */
17 /*
18     let name = "itcast";
19     console.log("hello " + name);
20
21     //es 6
22     console.log(`es6 hello ${name}`);
23 */
24 /*
25     //es 5
26     const str = "hello " +
27         "itcast";
28     //es 6
29     const template = `<div>
30     <span>hello itcast</span>
31     </div>`
32 */
33 /*
34     function person5(name, age) {
35         return {name:name, age:age};
36     }
37     console.log("es5 => " + JSON.stringify(person5("itcast", 13)))
38
39     function person6(name, age) {
40         return {name, age}
41     }
42     console.log("es6 => " + JSON.stringify(person6("itcast", 13)))
43 */
44 /*
45     var print = function (obj) {
46         console.log(obj);
47     };
48     print("print");
49
50     var print2 = obj => console.log(obj);
51     print2("print2");
52 */
53     // 两个参数的情况:
54     var sum = function (a , b) {
55         return a + b;
56     }
57     console.log(sum(1,1))
58
59     // 简写为:
60     var sum2 = (a,b) => a+b;
61     console.log(sum2(1,2))
62
63     var sum3 = (a,b) => {
64         console.log(a+b)
65         return a + b;
```

```

66     }
67     console.log(sum3(1,3))
68
69     let person = {
70         "name": "jack",
71         //以前, 也可以给food默认值
72         eat1: function (food="肉") {
73             console.log(this.name + "在吃" + food);
74         },
75         //箭头函数版
76         eat2: (food) => console.log(person.name + "在吃" + food), //不能使用this
77         //简版
78         eat3(food){
79             console.log(this.name + "在吃" + food)
80         }
81     }
82     person.eat1("111")
83     person.eat2("222")
84     person.eat3("333")
85 </script>
86 </head>
87 <body>
88
89 </body>
90 </html>

```

### 3) 箭头函数结合解构表达式

比如有一个函数：

```

1     const person = {
2         name: "jack",
3         age: 21,
4         language: ['java', 'js', 'css']
5     }
6
7     function hello(person) {
8         console.log("hello," + person.name)
9     }
10
11     hello(person);

```

如果用箭头函数和解构表达式

```

1     var hello2 = ({name}) => console.log("hello2," + name);
2
3     hello2(person);

```

### 4.3.7. map和reduce

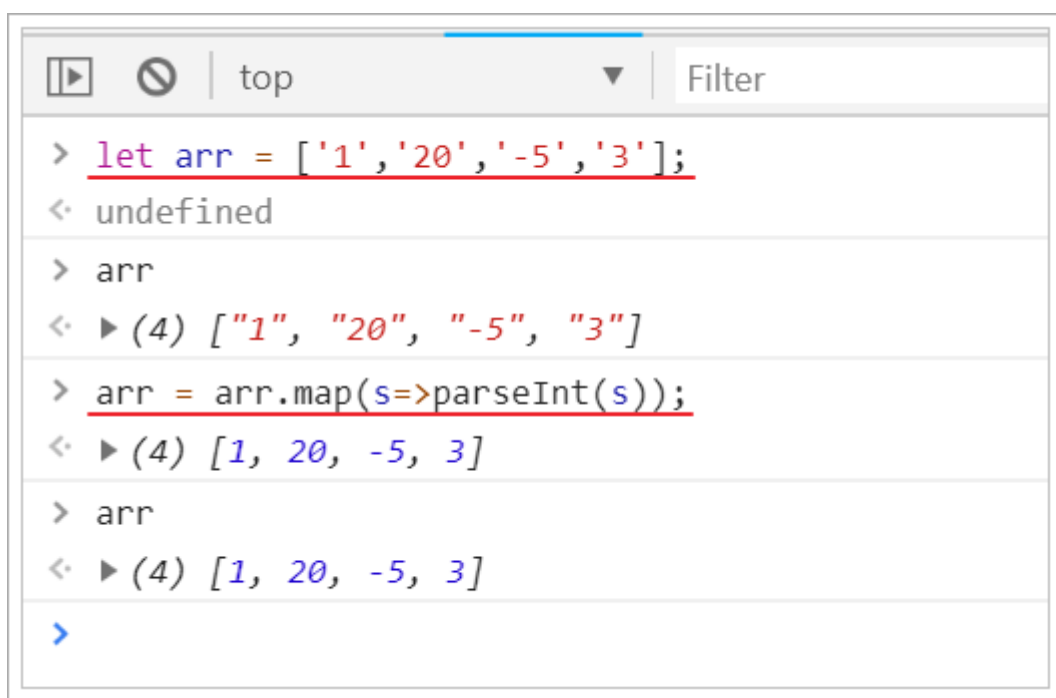
数组中新增了map和reduce方法。

## map

`map()`：接收一个函数，将原数组中的所有元素用这个函数处理后放入新数组返回。

举例：有一个字符串数组，希望转为int数组

```
1 let arr = ['1', '20', '-5', '3'];
2 console.log(arr)
3
4 arr = arr.map(s => parseInt(s));
5
6 console.log(arr)
```



## reduce

`reduce(function(), 初始值 (可选))`：接收一个函数（必须）和一个初始值（可选），该函数接收两个参数：

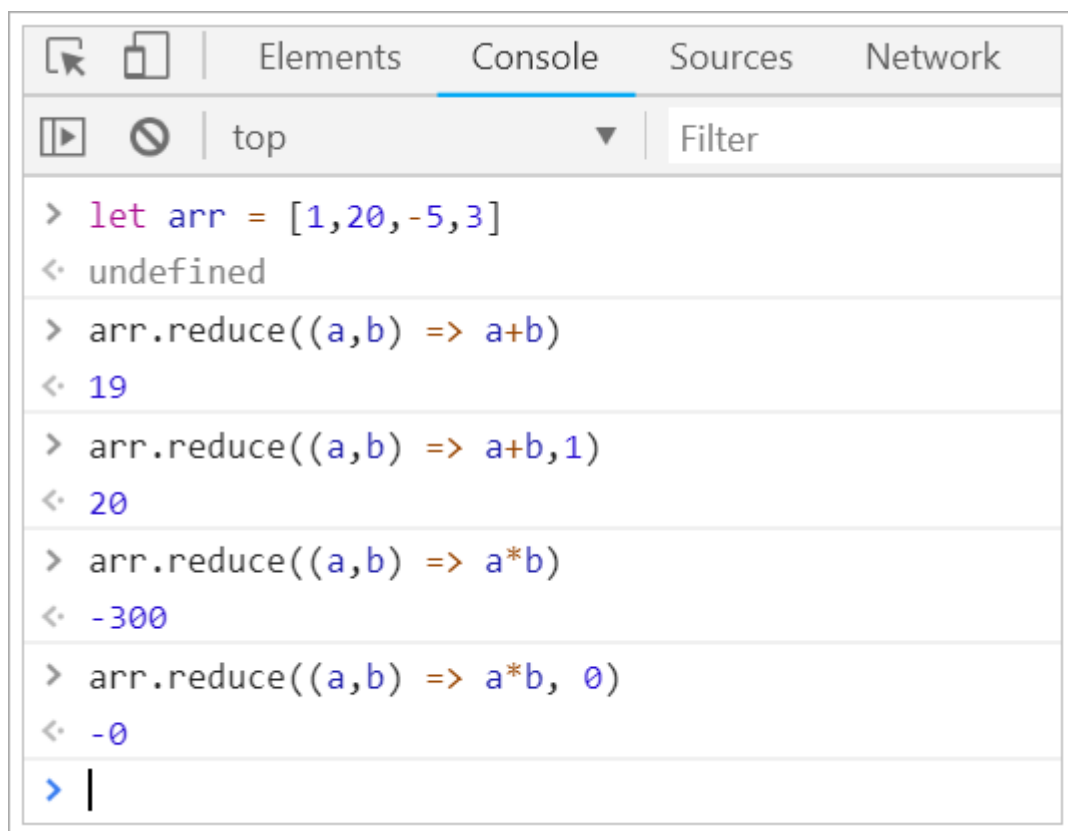
- 第一个参数是上一次reduce处理的结果
- 第二个参数是数组中要处理的下一个元素

`reduce()` 会从左到右依次把数组中的元素用reduce处理，并把处理的结果作为下次reduce的第一个参数。如果是第一次，会把**前两个元素**作为计算参数，或者把用户指定的初始值作为起始参数

举例：

```
1 let arr = [1,20,-5,3]
2
3 # 没有初始值
4 arr.reduce((a, b) => a+b)
5 # 设置初始值为1
6 arr.reduce((a, b) => a+b, 1)
7
8 arr.reduce((a, b) => a*b)
9
10 arr.reduce((a,b) => a*b, 0)
```

执行结果：



### 4.3.8. promise

所谓Promise，简单说就是一个容器，里面保存着某个未来才会结束的事件（通常是一个异步操作）的结果。从语法上说，Promise 是一个对象，从它可以获取异步操作的消息。Promise 提供统一的 API，各种异步操作都可以用同样的方法进行处理。

可以通过Promise的构造函数来创建Promise对象，并在内部封装一个异步执行的结果。

语法：

```

1  const promise = new Promise(function(resolve, reject) {
2    // ... 执行异步操作
3
4    if (/* 异步操作成功 */) {
5      resolve(value); // 调用resolve, 代表Promise将返回成功的结果
6    } else {
7      reject(error); // 调用reject, 代表Promise会返回失败结果
8    }
9  });

```

这样，在promise中就封装了一段异步执行的结果。

如果我们想要等待异步执行完成，做一些事情，我们可以通过promise的then方法来实现,语法：

```

1  promise.then(function(value) {
2    // 异步执行成功后的回调
3  });

```

如果想要处理promise异步执行失败的事件，还可以跟上catch：

```

1  promise.then(function(value) {
2    // 异步执行成功后的回调
3  }).catch(function(error) {
4    // 异步执行失败后的回调
5  })

```

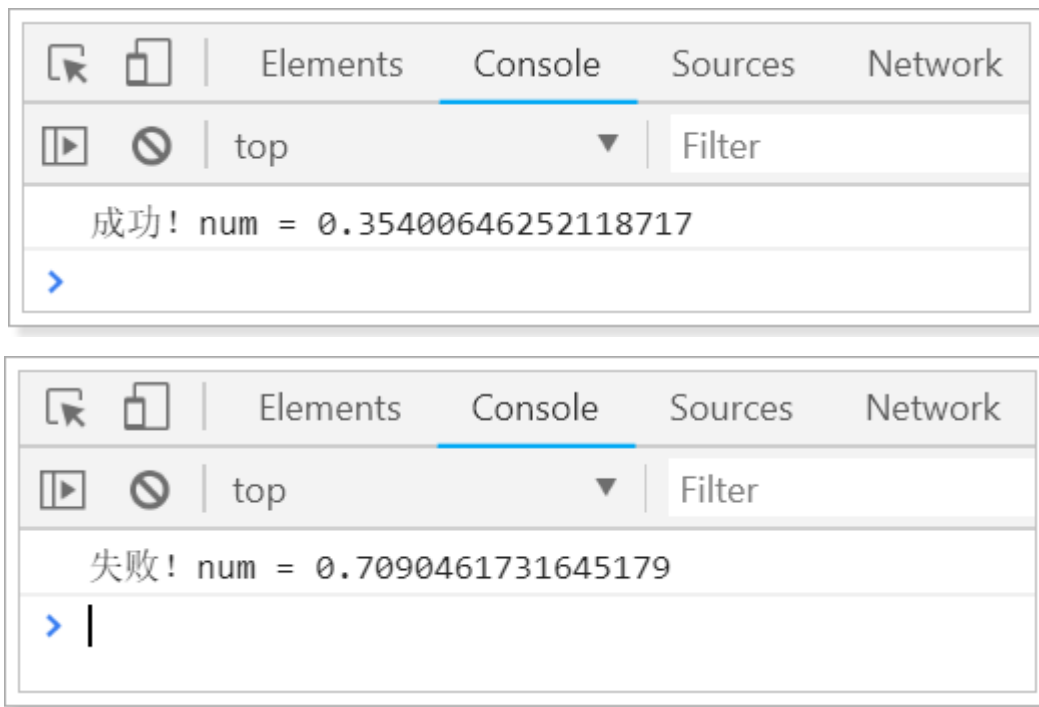
示例：

```

1  const promise = new Promise(function(resolve, reject){
2    setTimeout(()=>{
3      let num = Math.random();
4      if (num < 0.5) {
5        resolve("成功! num = " + num)
6      } else {
7        reject("失败! num = " + num)
8      }
9    }, 300)
10  });
11
12  promise.then(function (msg) {
13    console.log(msg)
14  }).catch(function (msg) {
15    console.log(msg)
16  })

```

结果：



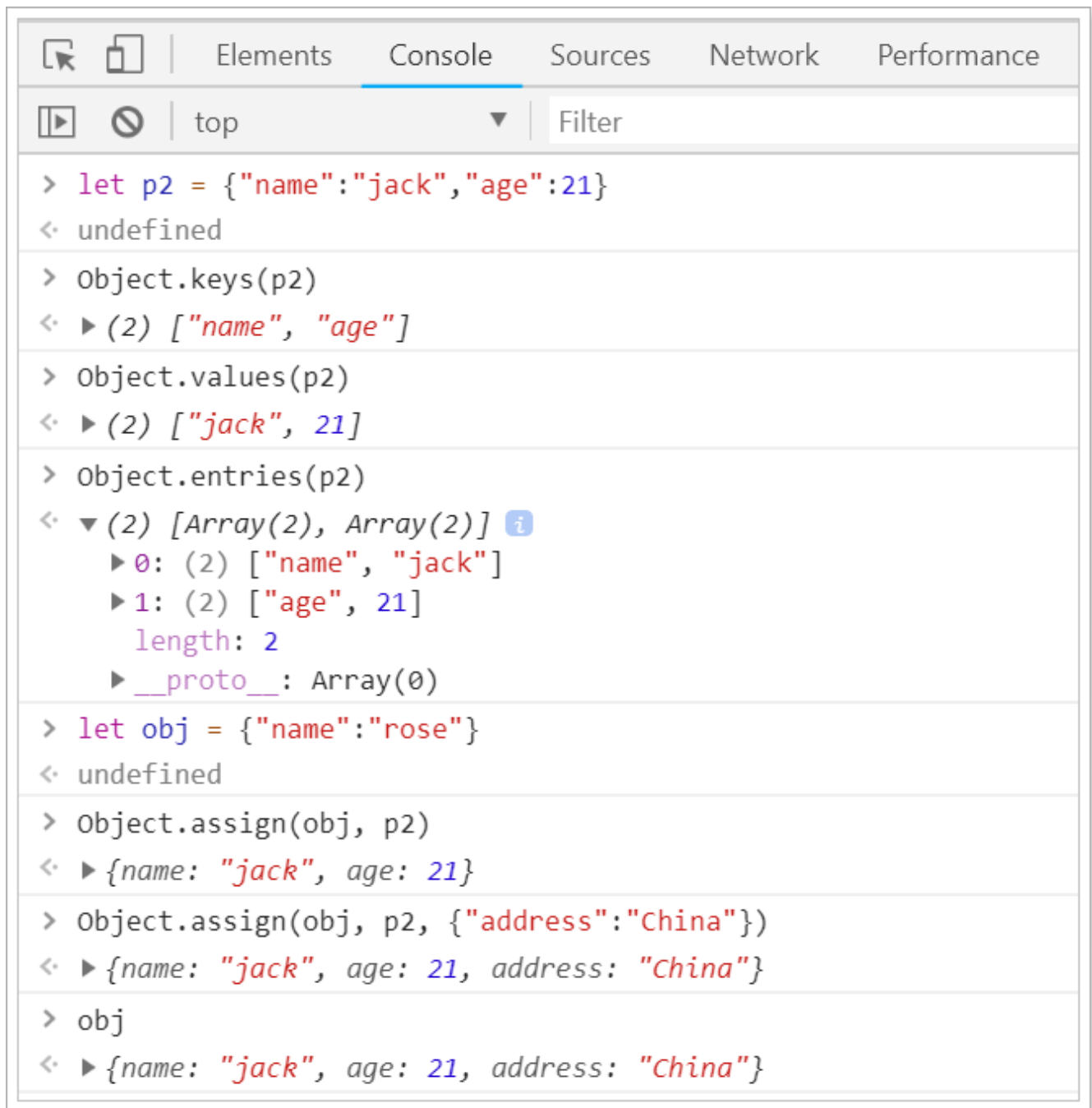
### 4.3.9. 对象扩展

ES6给Object拓展了许多新的方法，如：

- `keys(obj)`：获取对象的所有key形成的数组
- `values(obj)`：获取对象的所有value形成的数组
- `entries(obj)`：获取对象的所有key和value形成的二维数组。格式： `[[k1,v1],[k2,v2],...]`
- `assign(dest, ...src)`：将多个src对象的值 拷贝到 dest中（浅拷贝）。

```
1  let p2 = {"name": "jack", "age": 21}
2  Object.keys(p2)
3  Object.values(p2)
4  Object.entries(p2)
5
6  //测试覆盖问题
7  let obj = {"name": "rose"}
8  Object.assign(obj, p2)
9
10 Object.assign(obj, p2, {"address": "China"})
```





The screenshot shows a web browser's developer console with the 'Console' tab selected. The console displays the following code and its output:

```
> let p2 = {"name": "jack", "age": 21}
< undefined

> Object.keys(p2)
< ▶ (2) ["name", "age"]

> Object.values(p2)
< ▶ (2) ["jack", 21]

> Object.entries(p2)
< ▼ (2) [Array(2), Array(2)] ⓘ
  ▶ 0: (2) ["name", "jack"]
  ▶ 1: (2) ["age", 21]
  length: 2
  ▶ __proto__: Array(0)

> let obj = {"name": "rose"}
< undefined

> Object.assign(obj, p2)
< ▶ {name: "jack", age: 21}

> Object.assign(obj, p2, {"address": "China"})
< ▶ {name: "jack", age: 21, address: "China"}

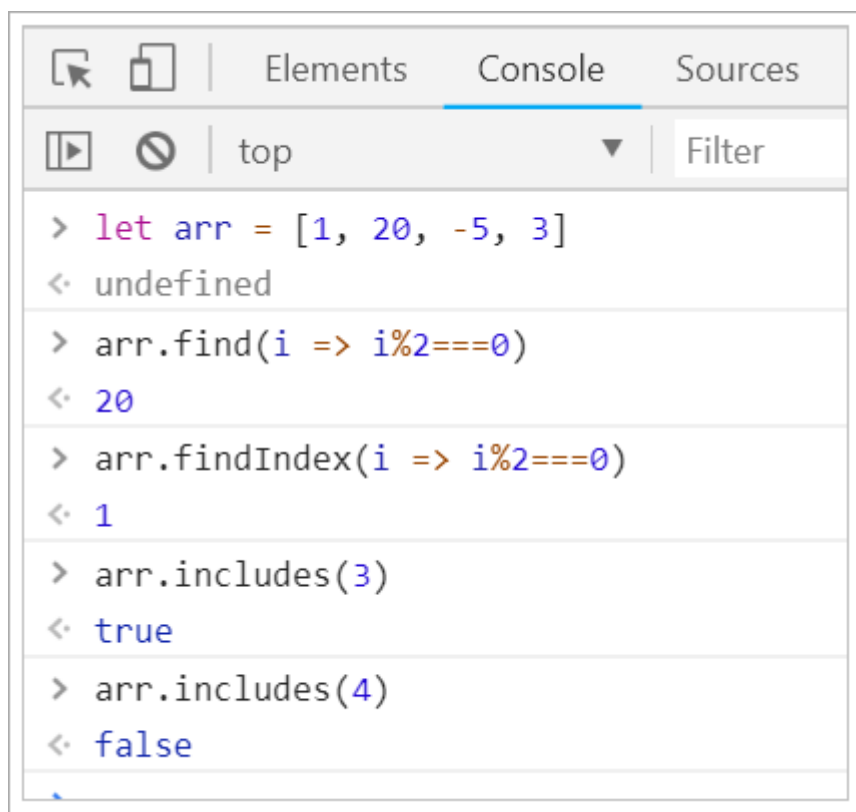
> obj
< ▶ {name: "jack", age: 21, address: "China"}
```

### 4.3.10. 数组扩展

ES6给数组新增了许多方法：

- `find(callback)`：把数组中的元素逐个传递给函数`callback`执行，如果返回`true`，则返回该元素
- `findIndex(callback)`：与`find`类似，不过返回的是匹配到的元素的索引
- `includes (element)`：判断指定元素是否存在

```
1 let arr = [1, 20, -5, 3]
2
3 //元素能整除2的
4 arr.find(i => i%2===0)
5
6 //查询元素能整除2的索引号
7 arr.findIndex(i => i%2===0)
8
9 arr.includes(3)
10 arr.includes(4)
```



### 4.3.11. export和import

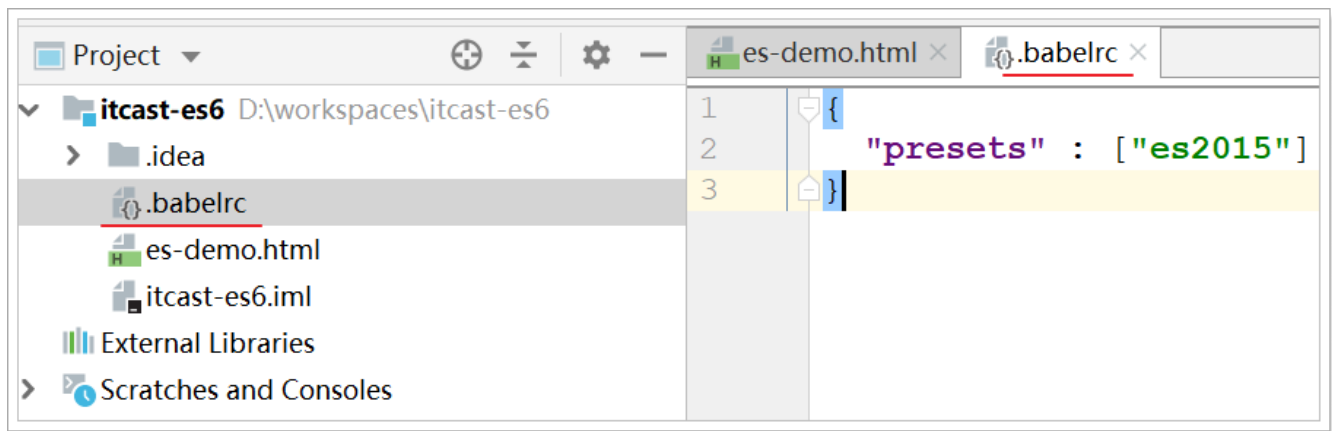
ES6 在语言标准的层面上，实现了模块功能。ES6 模块不是对象，而是通过export命令显式指定输出的代码，再通过import命令输入。遗憾的是export和import命令不能在浏览器直接使用，不过可以通过babel转换为es5再运行。import导入模块、export导出模块。

#### 1) 安装babel

babel是JavaScript 语法的编译器。

(1) babel转换配置,项目根目录添加 `.babelrc` 文件; 注意不要忘记了这个文件中的 `.`

```
1 {
2   "presets" : ["es2015"]
3 }
```



## (2) 安装es6转换模块

```
1 | cnpm install babel-preset-es2015 --save-dev
```

```
PS D:\workspaces\itcast-es6> cnpm install babel-preset-es2015 --save-dev
√ Installed 1 packages
√ Linked 65 latest versions
√ Run 0 scripts
deprecate babel-preset-es2015@* ????: Thanks for using Babel: we recommend using babel-preset-env no
w: please read babeljs.io/env to update!
√ All packages installed (67 packages installed from npm registry, used 2s(network 2s), speed 208.1
5kB/s, json 66(206.5kB), tarball 277.86kB)
```

## (3) 全局安装命令行工具

```
1 | # 管理员身份 执行
2 | cnpm install babel-cli -g
```

```
管理员: Windows PowerShell
PS D:\workspaces\itcast-es6> cnpm install babel-cli -g
Downloading babel-cli to C:\Program Files\nodejs\npm_modules\node_modules\babel-cli_tmp
Copying C:\Program Files\nodejs\npm_modules\node_modules\babel-cli_tmp\babel-cli@6.26.0@babel-cli to C:\Program Files\nodejs\npm_modules\node_modules\babel-cli
Installing babel-cli's dependencies to C:\Program Files\nodejs\npm_modules\node_modules\babel-cli\node_modules
[1/15] fs-readdir-recursive@ 1.0.0 installed at node_modules\fs-readdir-recursive@1.1.0@fs-readdir-recursive
[2/15] commander@ 2.11.0 installed at node_modules\commander@2.20.0@commander
[3/15] path-is-absolute@ 1.0.1 existed at node_modules\path-is-absolute@1.0.1@path-is-absolute
[4/15] convert-source-map@ 1.5.0 installed at node_modules\convert-source-map@1.6.0@convert-source-map
[5/15] slash@ 1.0.0 installed at node_modules\slash@1.0.0@slash
[6/15] output-file-sync@ 1.1.2 installed at node_modules\output-file-sync@1.1.2@output-file-sync
[7/15] source-map@ 0.5.6 installed at node_modules\source-map@0.5.7@source-map
[8/15] lodash@ 4.17.4 installed at node_modules\lodash@4.17.11@lodash
[9/15] glob@ 7.1.2 installed at node_modules\glob@7.1.3@glob
[10/15] v8flags@ 2.1.1 installed at node_modules\v8flags@2.1.1@v8flags
[11/15] babel-polyfill@ 6.26.0 installed at node_modules\babel-polyfill@6.26.0@babel-polyfill
[12/15] babel-runtime@ 6.26.0 installed at node_modules\babel-runtime@6.26.0@babel-runtime
[13/15] babel-register@ 6.26.0 installed at node_modules\babel-register@6.26.0@babel-register
fsevents@1.2.9 download from binary mirror: {"module_name": "fse", "module_path": ".\\lib\\binding\\{configuration}\\{node_abi}-{platform}-{arch}", "remote_path": ".\\v{version}\\", "package_name": "{module_name}-v{version}-{node_abi}-{platform}-{arch}.tar.gz", "host": "https://cdn.npm.taobao.org/dist/fsevents"}
chokidar@1.7.0 > fsevents@ 1.0.0 Package require os(darwin) not compatible with your platform(win32)
[fsevents@1.0.0] optional install error: Package require os(darwin) not compatible with your platform(win32)
[14/15] babel-core@ 6.26.0 installed at node_modules\babel-core@6.26.3@babel-core
[15/15] chokidar@ 1.6.1 installed at node_modules\chokidar@1.7.0@chokidar
All packages installed (188 packages installed from npm registry, used 3s(network 3s), speed 942.27kB/s, json 163(303.7kB), tarball 2.73MB)
[babel-cli@6.26.0] link @ -> C:\Program Files\nodejs\npm_modules\node_modules\babel-cli\bin\babel-doctor.js
[babel-cli@6.26.0] link @ -> C:\Program Files\nodejs\npm_modules\node_modules\babel-cli\bin\babel.js
[babel-cli@6.26.0] link @ -> C:\Program Files\nodejs\npm_modules\node_modules\babel-cli\bin\babel-node.js
[babel-cli@6.26.0] link @ -> C:\Program Files\nodejs\npm_modules\node_modules\babel-cli\bin\babel-external-helpers.js
PS D:\workspaces\itcast-es6>
```

#### (4) 使用

1 | babel-node js文件名

## 2) 命名导出 (Named exports)

此方式每一个需要输出的数据类型都要有一个name，统一输入一定要带有 {}，即便只有一个需要输出的数据类型。

export导出模块：编写export1.js

```
1 //方式一
2 export let name = "itcast";
3 export let age = 13;
4 export let gender = "男";
5 export let say = function(str){
6     console.log(str);
7 }
8
9 //方式二; 更推荐
10 let name = "itcast";
11 let age = 13;
12 let gender = "男";
13 let say = function(str){
14     console.log(str);
15 }
16 export {name, age, gender, say}
```

import导入模块：编写import1.js

```
1 import {name, age, gender, say} from "./export1"
2
3 console.log(name, age, gender)
4
5 //import命令输入的变量都是只读的，因为它的本质是输入接口。也就是说，不允许在加载模块的脚本里面，改写接口。
6 //name = "heima" //会报错 "name" is read-only
7
8 //上面代码中，脚本加载了变量name，对其重新赋值就会报错，因为name是一个只读的接口。但是，如果name是一个对象，改写name的属性是允许的。也就是name.abc = xxx 这样是可以的
9
10 say("hello itcast")
11
12 //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。
13 import {name as abc, say as hello} from "./export1"
14 console.log(abc)
15
16 hello("hello heima")
17
```

```
PS D:\workspaces\itcast-es6> babel-node import1.js
itcast 13 男
hello itcast
itcast
hello heima
PS D:\workspaces\itcast-es6>
```

### 3) 默认导出 (Default exports)

默认输出不需要name，但是一个js文件中只能有一个export default。

export导出模块：编写export2.js

```
1 //方式一
2 export default function(str){
3   console.log(str)
4 }
5
6 //方式二；一个文件只能一个export default，不能1个以上同时存在
7 /*
8 export default {
9   eat(sth){
10     console.log("eat " + sth)
11   },
12   drink(sth) {
```

```
13     console.log("drink " + sth)
14   }
15 }
16 */
17
```

import导入模块: 编写import2.js

```
1 //导入default的模块文件不需要使用{}
2 import itcast1 from "./export2"
3
4 itcast1("hello itcast");
5
6 /*
7 import abc from "./export2"
8
9 abc.eat("meat")
10 abc.drink("water")
11 */
```

需要注意 `export default` 在一个模块（文件）中只能使用一次；并且不能放置在函数或条件代码里面