杰普软件科技有限公司
www.briup.com

Tel: (021)55660810
Fax: (021)55660802
Email:training@briup.com
Msn: training.sh@hotmail.com
Home:http://www.briup.com

地址：上海市闸北区万荣路1188弄F栋6号三层-上海服务外包科技园龙软园区
邮编：200436

电话：021-56657112
传真：021-55661523-8003
电邮：training@briup.com
主页：http://www.briup.com

Briup High-End IT Training

第三阶段

企业级开发

Brighten Your Way And Raise You Up.

杰普软件科技有限公司

www.briup.com

Tel: (021)55660810
Fax: (021)55660802
Email:training@briup.com
Msn: training.sh@hotmail.com
Home:http://www.briup.com

地址：上海市闸北区万荣路1188弄F栋6号三层-上海服务外包科技园龙软园区
邮编：200436

电话：021-56657112
传真：021-55661523-8003
电邮：training@briup.com
主页：http://www.briup.com

Briup High-End IT Training

ES6

Brighten Your Way And Raise You Up.

杰普软件科技有限公司
www.briup.com

Tel: (021)55660810
Fax: (021)55660802
Email:training@briup.com
Msn: training.sh@hotmail.com
Home:http://www.briup.com

地址：上海市闸北区万荣路1188弄F栋6号三层-上海服务外包科技园龙软园区
邮编：200436

电话：021-56657112
传真：021-55661523-8003
电邮：training@briup.com
主页：http://www.briup.com

Briup High-End IT Training

第 1 章:环境搭建

Brighten Your Way And Raise You Up.

◆ 下载

Node也叫NodeJS，Node.js，由Ryan-Dahl于2009年5月在GitHub发布了第一版。Node是一个JavaScript运行环境(runtime)。实际上它是对Google V8引擎进行了封装。官网(http://www.nodejs.org) 是这样介绍Node的：一个搭建在ChromeJavaScript运行时上的平台，用于构建高速、可伸缩的网络程序。Node采用的事件驱动、非阻塞I/O模型，使它既轻量又高效，并成为构建运行在分布式设备上的数据密集型实时程序的完美选择。

目前Node在实际开发中主要作为开发基础环境存在，用于进行模块管理，编译es6代码，编译sass文件，运行各种Js脚本。

文档地址：https://nodejs.org/dist/latest-v8.x/docs/api/

Node安装

- 下载

  进入官网https://nodejs.org/en/，选取合适的版本进行下载

- 安装

  - **Linux**

    先将安装包解压

    然后进行环境变量的配置即可

  - **windows**

    直接点击安装即可

基本使用

执行js脚本

使用node可以直接执行js脚本

例如当前有一个js脚本文件为demo.js可以使用node命令来执行js脚本。

$ node demo # 或者 $ node demo.js

REPL环境

在命令行键入node命令，后面没有文件名，就进入一个Node.js的REPL环境（Read–eval–print loop，”读取-求值-输出”循环），可以直接运行各种JavaScript命令。

```
$ node
>1+1
2
>
```

模块化结构

- 模块化

Node.js采用模块化结构，按照CommonJS规范定义和使用模块。在Node中，以模块为单位划分所有功能，并且提供一个完整的模块加载机制，使得我们可以将应用程序划分为各个不同的部分，并且对这些部分进行很好的协同管理。通过将各种可重用的代码编写在各种模块中的方法，我们可以大大减少应用程序的代码量，提高应用程序开发效率以及应用程序的可读性。通过模块加载机制，我们也可以将各种第三方模块引入到我们的应用程序中。

CommonJS

JavaScript是一种功能强大的面向对象语言，具有一些最快速的动态语言解释器。官方JavaScript规范定义了一些用于构建基于浏览器的应用程序的对象的api。但是，规范并没有定义一个用于对于构建更广泛的应用程序的标准库。CommonJS API将通过定义处理许多常见应用程序需求的API来填补这一空白，最终提供与Python、Ruby和Java一样丰富的标准库。其意图是应用程序开发人员能够使用CommonJS API编写应用程序，然后在不同的JavaScript解释器和主机环境中运行该应用程序。在兼容CommonJS的系统中，你可以使用 JavaScript程序开发：

- 服务器端JavaScript应用程序
- 命令行工具
- 图形界面应用程序
- 混合应用程序（如，Titanium或Adobe AIR）

➢ Node应用由模块组成，采用CommonJS模块规范。

模块化结构

◆ 模块作用域

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

- 私有属性

```
// example.js

var x = 5;

var addX = function (value) { return value + x; };
```

上面代码中，变量x和函数addX，是当前文件example.js私有的，其他文件不可见。

- 全局属性

如果想在多个文件分享变量，必须定义为global对象的属性。

```
global.warning = true;
```

◆ 模块交互

CommonJS规范规定，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。

- 定义模块

```
var x = 5;

var addX = function (value) { return value + x; };

module.exports.x = x;

module.exports.addX = addX;
```

- 模块加载

require方法用于加载模块。

```
var example = require('./example.js');
console.log(example.x); // 5
console.log(example.addX(1)); // 6
```

模块化结构

◆ 模块对象

Node内部提供一个Module构建函数。所有模块都是Module的实例。每个模块内部，都有一个module对象，代表当前模块。它有以下属性。

- ✓ module.id 模块的识别符，通常是带有绝对路径的模块文件名。
- ✓ module.filename 模块的文件名，带有绝对路径。
- ✓ module.loaded 返回一个布尔值，表示模块是否已经完成加载。
- ✓ module.parent 返回一个对象，表示调用该模块的模块。
- ✓ module.children 返回一个数组，表示该模块要用到的其他模块。
- ✓ module.exports 表示模块对外输出的值。

◆ exports变量

为了方便，Node为每个模块提供一个exports变量，指向module.exports。这等同在每个模块头部，有一行这样的命令。

```
var exports = module.exports;
```

- **path模块**

path 模块提供了一些工具函数，用于处理文件与目录的路径，使用如下方法引用：

var path = require('path');

path.basename()	该方法返回一个参数路径的最后一部分
path.dirname()	该方法返回一个 path 的目录名
path.extname()	该方法返回 path 的扩展名，即从 path 的最后一部分中的最后一个 .（句号）字符到字符串结束。
path.isAbsolute()	该方法会判定 path 是否为一个绝对路径。
path.join()	该方法使用平台特定的分隔符把全部给定的 path 片段连接到一起，并规范化生成的路径
path.normalize()	该方法会规范化给定的 path，并解析 '..' 和 '.' 片段
path.delimiter	该属性提供平台特定的路径分隔符

另外：

- __filename：指向当前运行的脚本文件名。
- __dirname：指向当前运行的脚本所在的目录。

- **querystring模块**

querystring 模块提供了一些实用函数，用于解析与格式化 URL 查询字符串。使用如下方法引用

var querystring = require('querystring');

querystring.stringify(obj[, sep[, eq]]) 将对象转换为查询字符串

obj 要序列化成 URL 查询字符串的对象。

sep 用于界定查询字符串中的键值对的子字符串。默认为 '&'。

eq 用于界定查询字符串中的键与值的子字符串。默认为 '='。

querystring.parse(str[, sep[, eq]]) 将查询字符串转换为对象

- **url模块**

  url模块提供了一些实用函数，用于 URL 处理与解析。 可以通过以下方式使用

  var url = require('url');

  url.parse()　　将一个url地址转换为一个对象

  url.resolve()　　该方法会以一种 Web 浏览器解析超链接的方式把一个目标 URL 解析成相对于一个基础 URL

```
url.resolve('/one/two/three', 'four'); // '/one/two/four'
url.resolve('http://example.com/', '/one'); // 'http://example.com/one'
url.resolve('http://example.com/one', '/two'); // 'http://example.com/two'
```

◆ 介绍

Npm包管理工具，JS开发者能够更方便的分享和复用以及更新代码，被复用的代码被称为包或者模块，一个模块中包含了一到多个js文件。在模块中一般还会包含一个package.json的文件，该文件中包含了该模块的配置信息。一个完整的项目，需要依赖很多个模块。一个完整的npm包含三部分

- **npm网站**

  用于预览npm管理的包

- **注册机制**

  用于上传包，使用数据库来维护包与上传者的信息。

- **客户端**

  用于安装包

文档地址：https://docs.npmjs.com/

◆ 创建一个模块

Node.js的模块是一种能够被发布到npm上的包。创建模块从创建package.json文件开始，package.json是模块的配置文件。

- 可以使用**npm init**命令来初始化**package.json**文件

  $ npm init

  - ✓ **name** 模块名称　　**version** 模块版本
  - ✓ **description** 描述信息　　**main** 指定模块入口文件
  - ✓ **Dependencies** 依赖关系　　**engines** 指定**node**版本
  - ✓ **devDependencies** 环境依赖或测试依赖
  - ✓ **optionalDependencies**可选择依赖
  - ✓ **script** 定义当前模块脚本，使用**npm run**来运行所定义的脚本

- 使用**-y**参数创建默认**package.json**文件

  $ npm init -y

安装npm

npm会随着node一起被安装到本地。可以使用以下命令来更新npm

$ npm install npm@latest -g

安装淘宝镜像

由于默认npm的仓库在国外，下载起来很慢，可以使用淘宝镜像来加快下载速度。

$ npm install -g cnpm --registry=https://registry.npm.taobao.org

- Registry注册中心

  官方：https://registry.npmjs.org

  淘宝：https://registry.npm.taobao.org

  私有：http://localIP:port

修改npm权限

执行npm的时候有时会遇到权限不足的情况，可以通过以下方式进行修正。

$ npm config get prefix

$ sudo chown -R $(whoami) $(npm config get prefix)/{lib/node_modules,bin,share}

◆ 模块安装

如果想要仅在当前模块中使用某个第三方模块，就可以使用**npm install**的默认安装，默认安装即是本地安装；如果想要在命令行中使用模块，就需要进行全局安装。安装时，如果当前目录中没有**node_modules**目录，**npm**就会创建一个该目录。

$ npm install <package_name>

- **$ npm install**

  安装所有项目依赖的模块，依赖的模块定义在package.json中

- **$ npm install**

  安装模块时，默认会将所安装的模块写入到package.json中的dependencies属性，通过添加一些参数改变这个特性。

  -S, –save

  -D, --save-dev: Package will appear in your devDependencies.

  -O, --save-optional: Package will appear in your optionalDependencies.

  --no-save: Prevents saving to dependencies

  -E, --save-exact: Saved dependencies will be configured with an exact version rather than using npm's default semver range operator.

- 模块更新

  全局更新依赖的模块

  $ npm update <module_name>

模块删除

从node_modules中删除不需要的模块

$ npm uninstall -g <package_name>

不仅删除node_modules中的依赖，还需要删除package.json中的信息，可以使用—save参数

$ npm uninstall –save -g <package_name>

◆ 搭建本地npm仓库（sinopia）

1. 安装 **$ npm install -g sinopia**
2. 配置 **$ npm set registry http://localhost:4873/**
3. 添加用户 **$ npm adduser --registry http://localhost:4873/**
4. 发布模块 **$ npm publish <module_name>**
5. 启动 **$ sinopia**

◆ 命令行转码babel-cli

全局环境下进行 Babel 转码。这意味着，如果项目要运行，全局环境必须有 Babel，也就是说项目产生了对环境的依赖。

- 安装

  $ npm install --global babel-cli

- 安装预设并且添加配置文件配置.babelrc

  在当前项目的根目录下创建该文件

  $ npm install --save-dev babel-preset-es2015

  { "presets": ["es2015"]}

- 使用

  转码结果输出到标准输出

  $ babel example.js

  转码结果写入一个文件，--out-file 或 -o 参数指定输出文件

  $ babel example.js --out-file compiled.js

  整个目录转码 --out-dir 或 -d 参数指定输出目录

  $ babel src --out-dir lib

◆ 配置文件

Babel 的配置文件是.babelrc，存放在项目的根目录下。使用 Babel 的第一步，就是配置这个文件。该文件用来设置转码规则和插件，基本格式如下。

{ "presets": [], "plugins": [] }

- presets字段设定转码规则，官方提供以下的规则集，你可以根据需要安装。

ES2015转码规则

$ npm install --save-dev babel-preset-es2015 =>es2015

最新转码规则

$ npm install --save-dev babel-preset-latest =>latest

不会过时的转码规则

$ npm install --save-dev babel-preset-env =>env

- 然后，将这些规则加入.babelrc。

{ “presets”: ["es2015"], "plugins": [] , "ignore":[]}

1.全局安装babel-cli

cnpm install -g babel-cli

babel --version 查看版本号

2.局部安装babel-preset-latest，安装到当前目录底下

cnpm install -g babel-preset-latest

3.项目根目录下创建一个配置文件

.babelrc

{"presets":['latest']}

4.开始转换

babel a.js 将转换后的代码输出到终端

babel a.js --out-file b.js 将转换后的代码输出到b.js文件中

babel src --out-dir dist 将src目录中的所有的文件转换成ES5的代码，输出到dist目录中。文件名一致

1.将当前目录初始化，成为一个Node项目， 在项目中有package.json文件，文件中可以声明该项目需要的包-模块。其他人拿到项目之后，使用cnpm install ,可以直接下载需要的依赖

cnpm init -y 快速初始化一个项目

2.在项目开发阶段安装babel-cli

cnpm install --save-dev babel-cli

3.在项目开发阶段安装babel-preset-latest

cnpm install --save-dev babel-preset-latest

--save-dev是在开发阶段依赖的包

--save 是在打包之后依然依赖的包，

将安装的记录，需要的依赖，记录到package.json文件中

4.配置文件.babelrc

{'presets':['latest']}

5.在package.json中编写脚本，执行转换

'build':'babel src --out-dir dist'

6.执行脚本

cnpm run build

- 将**babel-cli**安装到项目中
  - 安装babel-cli以及预设

    $ npm install --save-dev babel-cli

    $ npm install –save-dev babel-preset-env
  - 配置文件.babelrc

    $ vim .babelrc

    {“presets”:[“env”]}
  - 在package.json中添加脚本

```
{      // ...
       "scripts": { "build": "babel src -d lib" }
}
```

  - 运行npm脚本，执行转码操作

    $ npm run build

◆ babel-polyfill

Babel 默认只转换新的 JavaScript 句法（syntax），而不转换新的 API，比如Iterator、Generator、Set、Maps、Proxy、Reflect、Symbol、Promise等全局对象，以及一些定义在全局对象上的方法（比如Object.assign）都不会转码。举例来说，ES6在Array对象上新增了Array.from方法。Babel 就不会转码这个方法。如果想让这个方法运行，必须使用babel-polyfill，为当前环境提供一个垫片

- 安装

  $ npm install --save babel-polyfill

- 在js文件中引用并且使用

  import 'babel-polyfill'; // 或者 require('babel-polyfill');

杰普软件科技有限公司
www.briup.com

Tel: (021)55660810
Fax: (021)55660802
Email:training@briup.com
Msn: training.sh@hotmail.com
Home:http://www.briup.com

地址：上海市闸北区万荣路1188弄F栋6号三层-上海服务外包科技园龙软园区
邮编：200436

电话：021-56657112
传真：021-55661523-8003
电邮：training@briup.com
主页：http://www.briup.com

Briup High-End IT Training

第 2 章:基础知识

Brighten Your Way And Raise You Up.

◆ 基本用法

ES6 新增了let命令，用来声明变量。它的用法类似于var，但是也存在新的特性。

- let所声明的变量，只在let命令所在的代码块内有效。适用于for循环

```
{ let a = 10; var b = 1; } a // ReferenceError: a is not defined. b // 1
```

- 不存在变量提升

```
console.log(bar); // 报错ReferenceError
let bar = 2;
```

- 暂时性死区

在代码块内，使用let命令声明变量之前，该变量都是不可用的。

```
var tmp = 123;
if (true) {
        tmp = 'abc'; // ReferenceError
        let tmp;
}
```

- 不允许重复声明。

```
function () { let a = 10; let a = 1; }          // 报错
```

◆ 块级作用域

Let实际上为Javascript新增了块级作用域。外层作用域无法读取内层作用域的变量，内层作用域可以定义外层作用域的同名变量。

```
let foo = 1;
{
  let foo =2 ;   //定义同名变量
  let bar ="one";
}
console.log(foo);//1
console.log(bar);//报错
```

块级作用域的出现，实际上使得获得广泛应用的立即执行函数表达式（IIFE）不再必要了。

◆ 基本用法

const声明一个只读的常量。一旦声明，常量的值就不能改变。

```
const PI = 3.1415; PI = 3; // TypeError: Assignment to constant variable.
```

- const声明的变量不得改变值，这意味着，const一旦声明变量，就必须立即初始化，不能留到以后赋值。

```
const foo; // SyntaxError: Missing initializer in const declaration
```

- 块级作用域

只在声明所在的块级作用域内有效。

- 暂时性死区。

在代码块内，使用let命令声明变量之前，该变量都是不可用的。

```
if (true) {      console.log(MAX); // ReferenceError   const MAX = 5; }
```

- 不允许重复声明。

```
var message = "Hello!"; let age = 25;
// 以下两行都会报错
const message = "Goodbye!";
const age = 30;
```

解构赋值

◆ 解构

ES6 允许按照一定模式，从数组和对象中提取值，对变量进行赋值，这被称为解构（Destructuring）。例如：

```
let [a, b, c] = [1, 2, 3];
```

本质上，这种写法属于“模式匹配”，只要等号两边的模式相同，左边的变量就会被赋予对应的值。如果解构不成功，变量的值就等于undefined。另一种情况是不完全解构，即等号左边的模式，只匹配一部分的等号右边的数组。这种情况下，解构依然可以成功。

解构赋值

数组的解构赋值

```
let [a, b, c] = [1, 2, 3];
```

- 不完全解构

```
let [a, [b], d] = [1, [2, 3], 4];          //a = 1; b = 2; d = 4
```

- 集合解构

```
let [head, ...tail] = [1, 2, 3, 4];       //head = 1; tail = [2, 3, 4]
```

- 默认值（当匹配值严格等于undefined时，默认值生效）

```
let [x, y = 'b'] = ['a'];      // x='a', y='b'
```

- 默认值也可以为函数

```
function f() { console.log('aaa'); }
let [x = f()] = [1];
```

解构赋值

◆ 对象的解构赋值

- 对象的属性没有次序，变量必须与属性同名，才能取到正确的值

```
let { foo, bar } = { foo: “aaa”, bar: “bbb” }; // foo = "aaa”; bar = "bbb”
```

- 如果变量名与属性名不一致，必须写成下面这样。重命名

```
var { foo: baz } = { foo: 'aaa', bar: 'bbb' };  //baz = "aaa”
```

- 这实际上说明，对象的解构赋值是下面形式的简写。

```
let { foo: foo, bar: bar } = { foo: "aaa", bar: "bbb" };
```

- 嵌套解构

```
let obj = { p: [ 'Hello', { y: 'World' } ] };
let { p: [x, { y }] } = obj;                    //x = "Hello”; y = "World”
```

- 默认值（默认值生效的条件是，对象的属性值严格等于undefined）

```
var {x: y = 3} = {}; //y=3
```

解构赋值

- 字符串的解构赋值
  - 解构时，字符串被转换成了一个类似数组的对象。

```
const [a, b, c, d, e] = 'hello';       //a=h;b=e;c=l;d=l;e=o
```

  - 也可以对length属性解构

```
let {length : len} = 'hello';          //len = 5
```

- 数值和布尔值解构赋值
  - 解构时，如果等号右边是数值和布尔值，则会先转为对象

```
let {toString: s} = 123;  //函数 s === Number.prototype.toString true
let {toString: s} = true; //函数 s === Boolean.prototype.toString true
```

解构赋值

函数参数的解构赋值

- 基本语法。

```
function add([x, y]){ return x + y; }

add([1, 2]);
```

- 默认值

```
function move({x = 0, y = 0}) {
      return [x, y];
}
move({x: 3, y: 8}); // [3, 8]
move({x: 3}); // [3, 0]
move({});     // [0, 0]
move();       //报错  Cannot destructure property `x` of 'undefined' or 'null'.
```

解构赋值

常见用途

- 交换变量的值

```
let x = 1; let y = 2; [x, y] = [y, x];
```

- 从函数返回多个值

```
function example() { return [1, 2, 3]; }
let [a, b, c] = example();
```

- 函数参数的定义

```
function f([x, y, z]) { ... }
f([1, 2, 3]);
```

- 提取数据

```
let obj= { id: 42, status: "OK", data: [867, 5309] };
let { id, status, data: number } = obj;
```

- 输入模块的指定方法

```
const { SourceMapConsumer, SourceNode } = require("source-map");
```

解构赋值

常见用途

- 函数参数的默认值

```
jQuery.ajax = function (url, {
    async = true, cache = true, global = true,
    beforeSend = function () {},
    complete = function () {},
    // ... more config
}) { // ... do stuff };
```

指定参数的默认值，就避免了在函数体内部再写var foo = config.foo || 'default foo';这样的语句

- 遍历map结构

```
var map = new Map();
map.set('first', 'hello');
map.set('second', 'world');
for (let [key, value] of map) {
    console.log(key + " is " + value);
}
```

杰普软件科技有限公司

www.briup.com

Tel: (021)55660810
Fax: (021)55660802
Email:training@briup.com
Msn: training.sh@hotmail.com
Home:http://www.briup.com

地址：上海市闸北区万荣路1188弄F栋6号三层-上海服务外包科技园龙软园区
邮编：200436

电话：021-56657112
传真：021-55661523-8003
电邮：training@briup.com
主页：http://www.briup.com

Briup High-End IT Training

第 3 章：对象、函数、数组的扩展

Brighten Your Way And Raise You Up.

学习目标

- 属性简写方式
- 方法简写方式
- Object方法的扩展
- 函数默认值
- 箭头函数
- 扩展运算符
- Array.from()
- Array.of()
- 数组实例的find(), findIndex()
- 数组实例的fill()
- 数组实例的entries(),keys(), values()
- 数组实例的includes()

对象扩展

- 属性简写

  ES6允许直接写入变量和函数，作为对象的属性和方法。这时，属性名为变量名, 属性值为变量的值。

  ```
  var foo = 'bar';

  var baz = {foo}; // 等同于 var baz = {foo: foo};
  ```

- 方法简写

  ```
  var o = { method() { return "Hello!"; } };

  // 等同于

  var o = { method: function() { return "Hello!"; } };
  ```

- 属性名表达式

  ES6 允许字面量定义对象时，可以把表达式放在方括号内。

  ```
  let propKey = 'foo';

  let obj = { [propKey]: true, ['a' + 'bc']: 123 };
  ```

对象扩展

- 方法的name属性

  函数的name属性，返回函数名。

  ```
  const person = { sayName() { console.log('hello!'); }, };

  person.sayName.name // "sayName"
  ```

- Object.is(value1,value2)

  同值相等，与===类似，不同之处在于：一是+0不等于-0，二是NaN等于自身。

  ```
  Object.is('foo', 'foo') // true

  Object.is({}, {}) // false
  ```

对象扩展

Object.assign(target,o1,o2…)

用于对象的合并，将源对象（source）的所有可枚举属性，复制到目标对象（target）。Object.assign方法实行的是浅拷贝，而不是深拷贝。也就是说，如果源对象某个属性的值是对象，那么目标对象拷贝得到的是这个对象的引用。具有以下作用：

- 为对象添加属性和方法

```
Object.assign(SomeClass.prototype, {
    someMethod(arg1, arg2) { ··· },
    anotherMethod() { ··· }
});
```

- 克隆对象

```
function clone(origin) { return Object.assign({}, origin); }
```

- 为属性提供默认值

```
function processContent(options) {
    options = Object.assign({}, DEFAULTS, options);
    //...
}
```

上面代码中，DEFAULTS对象是默认值，options对象是用户提供的参数。

对象扩展

- **__proto__属性**

  本质上属于内部属性，指向当前对象的prototype对象，一般不直接使用。

- **Object. setPrototypeOf(obj,prototype)**

  用来设置一个对象的prototype对象，返回参数对象本身。它是 ES6 正式推荐的设置原型对象的方法。该方法等同如下写法：

```
function (obj, proto) {
    obj.__proto__ = proto;
    return obj;
}
```

- **Object. getPrototypeOf(obj)**

  用于读取一个对象的原型对象。

对象扩展

- **Object.keys(obj)**

  返回一个数组，成员是参数对象自身的（不含继承的）所有可遍历（enumerable）属性的键名。

- **Object.values(obj)**

  返回一个数组，成员是参数对象自身的（不含继承的）所有可遍历（enumerable）属性的键值

- **Object. entries(obj)**

  返回一个数组，成员是参数对象自身的（不含继承的）所有可遍历（enumerable）属性的键值对数组。

函数的扩展

◆ 函数参数的默认值

ES6 允许为函数的参数设置默认值，即直接写在参数定义的后面。

```
function log(x, y = 'World') {
    console.log(x, y);
}
```

- 通常情况下，定义了默认值的参数，应该是函数的尾参数
- 函数的length属性，将返回没有指定默认值的参数个数。如果遇到有默认值的参数就停止。

与解构赋值默认值结合使用

参数默认值可以与解构赋值的默认值，结合起来使用。

```
function foo({x, y = 5}) {
    console.log(x, y);
}
foo({}) // undefined 5
foo({x: 1}) // 1 5
foo({x: 1, y: 2}) // 1 2
```

rest参数

ES6 引入 rest 参数（形式为...变量名），用于获取函数的多余参数，这样就不需要使用arguments对象了。rest 参数搭配的变量是一个数组，该变量将多余的参数放入数组中

```
function add(...values) {
    let sum = 0;
    for (var val of values) {
        sum += val;
    }
    return sum;
}
add(2, 5, 3) // 10
```

箭头函数

ES6 允许使用“箭头”（=>）定义函数

- 基本用法

```
let f = v => v;
```

等价于

```
let f = function(v) {
    return v;
};
```

- 如果箭头函数不需要参数或需要多个参数，就使用一个圆括号代表参数部分。
- 如果箭头函数的代码块部分多于一条语句，就要使用大括号将它们括起来，并且使用return语句返回。

箭头函数

- **this**

箭头函数里面没有自己的this，而是引用外层的this。

```
// ES6
function foo() { setTimeout(() => { console.log('id:', this.id); }, 100); }
// ES5
function foo() {
    var _this = this;
    setTimeout(function () { console.log('id:', _this.id); }, 100);
}
```

- 不能作为构造函数
- 有内部属性arguments，不保存实参

数组的扩展

◆ 扩展运算符

扩展运算符（spread）是三个点（...）。它好比 rest 参数的逆运算，将一个数组转为用逗号分隔的参数序列

```
console.log(...[1, 2, 3]) // 1 2 3
```

- 函数的调用

```
function add(x, y) { return x + y; }

add(…[1,3])

Math.max(...[14, 3, 77])
```

- 将字符串转换为数组

```
[...'hello']
```

数组的扩展

Array.from

用于将两类对象转为真正的数组：类似数组的对象（array-like object）和可遍历（iterable）的对象（包括ES6新增的数据结构Set和Map）

```
let arrayLike = { '0': 'a', '1': 'b', '2': 'c', length: 3 };

// ES6的写法

let arr2 = Array.from(arrayLike); // ['a', 'b', 'c']
```

- 只要是部署了Iterator接口的数据结构，Array.from都能将其转为数组。

```
Array.from('hello')                    //将字符串转换为数组 ['h', 'e', 'l', 'l', 'o']

let namesSet = new Set(['a', 'b'])

Array.from(namesSet)                   // ['a', 'b']
```

数组的扩展

Array.of()

用于将一组值，转换为数组。这个方法的主要目的，是弥补数组构造函数Array()的不足。因为参数个数的不同，会导致Array()的行为有差异。

Array.of(3, 11, 8) // [3,11,8]

new Array(10)　[] length=10

数组的扩展

数组实例的 find() 和 findIndex()

- 数组实例的find方法，用于找出第一个符合条件的数组成员。它的参数是一个回调函数，所有数组成员依次执行该回调函数，直到找出第一个返回值为true的成员，然后返回该成员。如果没有符合条件的成员，则返回undefined 。find方法的回调函数可以接受三个参数，依次为当前的值、当前的位置和原数组。

```
[1, 4, -5, 10].find((n) => n < 0) // -5
```

- 数组实例的findIndex方法的用法与find方法非常类似，返回第一个符合条件的数组成员的位置，如果所有成员都不符合条件，则返回-1。

```
[1, 5, 10, 15].findIndex(function(value, index, arr) { return value > 9; }) // 2
```

数组的扩展

- **数组实例的fill()**

  fill方法使用给定值，填充一个数组。

```
['a', 'b', 'c'].fill(7)          // [7, 7, 7]
new Array(3).fill(7)             // [7, 7, 7]
```

数组的扩展

- 数组实例的 entries()，keys()

这两个方法用于遍历数组。它们都返回一个遍历器对象（详见第四章中的Iterator迭代器），可以用for...of循环进行遍历，唯一的区别是keys()是对键名的遍历，entries()是对键值对的遍历

```
for (let index of ['a', 'b'].keys()) {
    console.log(index);
}
for (let [index, elem] of ['a', 'b'].entries()) {
    console.log(index, elem);
 }
```

不使用数组实例的values()方法，返回的是迭代器对象，还需继续遍历。

数组实例的 includes()

该方法返回一个布尔值，表示某个数组是否包含给定的值，与字符串的includes方法类似。ES2016 引入了该方法。

```
[1, 2, 3].includes(2)       // true
[1, 2, 3].includes(4)       // false
[1, 2, NaN].includes(NaN)   // true
```

杰普软件科技有限公司
www.briup.com

Tel: (021)55660810
Fax: (021)55660802
Email:training@briup.com
Msn: training.sh@hotmail.com
Home:http://www.briup.com

地址：上海市闸北区万荣路1188弄F栋6号三层-上海服务外包科技园龙软园区
邮编：200436

电话：021-56657112
传真：021-55661523-8003
电邮：training@briup.com
主页：http://www.briup.com

Briup High-End IT Training

第 4 章:

Set和Map数据结构以及Promise

Brighten Your Way And Raise You Up.

学习目标

- set
- map
- Iterator
- Promise介绍
- Promise基本用法

Set实例的创建

它类似于数组，但是成员的值都是唯一的，没有重复的值。Set 本身是一个构造函数，用来生成 Set 数据结构。

```
const s = new Set();

[2, 3, 5, 4, 5, 2, 2].forEach(x => s.add(x));

console.log(s);            // 2 3 5 4
```

- Set 函数可以接受一个数组（或者具有 iterable 接口的其他数据结构）作为参数，用来初始化。

```
[...new Set(array)] // 去除数组的重复成员
```

数组的扩展

◆ Set实例的属性和方法

Set 结构的实例有以下属性。

- ✓ **Set.prototype.constructor**：构造函数，默认就是Set函数。
- ✓ **Set.prototype.size**：返回Set实例的成员总数。

Set 结构的实例有以下方法。

- ✓ **add(value)**：添加某个值，返回Set结构本身
- ✓ **delete(value)**：删除某个值，返回一个布尔值，表示删除是否成功。
- ✓ **has(value)**：返回一个布尔值，表示该值是否为Set的成员。
- ✓ **clear()**：清除所有成员，没有返回值。
- ✓ **keys()**：返回键名的遍历器
- ✓ **values()**：返回键值的遍历器
- ✓ **entries()**：返回键值对的遍历器
- ✓ **forEach()**：使用回调函数遍历每个成员

Map实例的属性和方法

Map类似于对象，也是键值对的集合，但是“键”的范围不限于字符串，各种类型的值（包括对象）都可以当作键。也就是说，Object 结构提供了“字符串—值”的对应，Map结构提供了“值—值”的对应，是一种更完善的 Hash 结构实现。如果你需要“键值对”的数据结构，Map 比 Object 更合适。

Map 可以接受一个数组作为参数。该数组的成员是一个表示键值对的数组。

```
const map = new Map([ ['name', '张三'], ['title', 'Author'] ]);
```

◆ Map实例的属性和方法

Map 结构的实例有以下属性。

- ✓ **Map.prototype.size：返回 Map 结构的成员总数。**

Map 结构的实例有以下方法。

- ✓ **set(key, value)：set方法设置键名key对应的键值为value，然后返回整个 Map 结构。如果key已经有值，则键值会被更新，否则就新生成该键。**
- ✓ **get(key)：get方法读取key对应的键值，如果找不到key，返回undefined。**
- ✓ **has(key)：has方法返回一个布尔值，表示某个键是否在当前 Map 对象之中。**
- ✓ **delete(key)：delete方法删除某个键，返回true。如果删除失败，返回false。**
- ✓ **clear():清除所有成员，没有返回值**
- ✓ **keys()：返回键名的遍历器**
- ✓ **values()：返回键值的遍历器**
- ✓ **entries()：返回键值对的遍历器**
- ✓ **forEach()：使用回调函数遍历每个成员**

◆ Iterator（遍历器）的概念

遍历器（Iterator）就是这样一种机制。它是一种接口，为各种不同的数据结构提供统一的访问机制。任何数据结构只要部署Iterator接口，就可以完成遍历操作（即依次处理该数据结构的所有成员）

Iterator 的作用有三个：一是为各种数据结构，提供一个统一的、简便的访问接口；二是使得数据结构的成员能够按某种次序排列；三是ES6创造了一种新的遍历命令for...of循环，Iterator接口主要供for...of消费。

- Iterator 的遍历过程是这样的。
  - ✓ **创建一个指针对象，指向当前数据结构的起始位置。也就是说，遍历器对象本质上，就是一个指针对象。**
  - ✓ **第一次调用指针对象的next方法，可以将指针指向数据结构的第一个成员。**
  - ✓ **第二次调用指针对象的next方法，指针就指向数据结构的第二个成员。**
  - ✓ **不断调用指针对象的next方法，直到它指向数据结构的结束位置。**

默认Iterator接口

Iterator 接口的目的，就是为所有数据结构，提供了一种统一的访问机制，即for...of循环（详见下文）。当使用for...of循环遍历某种数据结构时，该循环会自动去寻找Iterator 接口。一种数据结构只要部署了 Iterator 接口，我们就称这种数据结构是“可遍历的”（iterable）。可以通过如下方法访问Iterator对象

```
var iterator = iterObj[Symbol.iterator]();
```

- 原生具备 Iterator 接口的数据结构如下
  - ✓ Array
  - ✓ Map
  - ✓ Set
  - ✓ String
  - ✓ TypedArray 二进制数据缓存区的一个对象
  - ✓ 函数的 arguments 对象
  - ✓ NodeList 对象

◆ Promise介绍

Promise 是异步编程的一种解决方案，比传统的解决方案（回调函数和事件）更合理和更强大。它由社区最早提出和实现，ES6 将其写进了语言标准，统一了用法，原生提供了Promise对象。

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

有了Promise对象，就可以将异步操作以同步操作的流程表达出来，避免了层层嵌套的回调函数。此外，Promise对象提供统一的接口，使得控制异步操作更加容易。

◆ 基本用法

Promise构造函数接受一个函数作为参数，该函数的两个参数分别是resolve和reject。它们是两个函数，由 JavaScript 引擎提供，不用自己部署。

resolve函数的作用是，将Promise对象的状态从“未完成”变为“成功”（即从 Pending 变为 Resolved），在异步操作成功时调用，并将异步操作的结果，作为参数传递出去；reject函数的作用是，将Promise对象的状态从“未完成”变为“失败”（即从 Pending 变为 Rejected），在异步操作失败时调用，并将异步操作报出的错误，作为参数传递出去。如果调用resolve函数和reject函数时带有参数，那么它们的参数会被传递给回调函数。

Promise实例生成以后，可以用then方法分别指定Resolved状态和Rejected状态的回调函数。.then(function(){//success},functions(){//error});

```
function timeout(ms) {
    return new Promise((resolve, reject) => {
            setTimeout(resolve, ms, 'done');
    });
}
timeout(100).then((value) => { console.log(value); });
```

Promise.prototype.then()

Promise 实例具有then方法，也就是说，then方法是定义在原型对象Promise.prototype上的。它的作用是为 Promise 实例添加状态改变时的回调函数。then方法的第一个参数是Resolved状态的回调函数，第二个参数（可选）是Rejected状态的回调函数。then方法返回的是一个新的Promise实例（注意，不是原来那个Promise实例）。因此可以采用链式写法，即then方法后面再调用另一个then方法。如果使用两个then方法，第一个回调函数完成以后，会将返回结果作为参数，传入第二个回调函数。

```
getJSON("/post/1.json").then(
    post => getJSON(post.commentURL)
 ).then(
    comments => console.log("Resolved: ", comments),
    err => console.log("Rejected: ", err)
);
```

上面代码中，第一个then方法指定的回调函数，返回的是另一个Promise对象。这时，第二个then方法指定的回调函数，就会等待这个新的Promise对象状态发生变化。如果变为resolved，就调用funcA，如果状态变为rejected，就调用funcB。

Promise.prototype.catch()

Promise.prototype.catch方法是.then(null, rejection)的别名，用于指定发生错误时的回调函数，般来说，不要在then方法里面定义Reject状态的回调函数（即then的第二个参数），总是使用catch方法。

```
var promise = new Promise(function(resolve, reject) {
    reject(new Error('test'));
});
 promise.catch(function(error) {
    console.log(error);
 });
```

Promise 对象的错误具有“冒泡”性质，会一直向后传递，直到被捕获为止。也就是说，错误总是会被下一个catch语句捕获。

Promise.all()

Promise.all方法用于将多个 Promise 实例，包装成一个新的 Promise 实例。

var p = Promise.all([p1, p2, p3]);

上面代码中，Promise.all方法接受一个数组作为参数，p1、p2、p3都是 Promise 实例，p的状态由p1、p2、p3决定，分成两种情况。

- ✓ 只有p1、p2、p3的状态都变成resolved，p的状态才会变成resolved，此时p1、p2、p3的返回值组成一个数组，传递给p的回调函数。
- ✓ 只要p1、p2、p3之中有一个被rejected，p的状态就变成rejected，此时第一个被reject的实例的返回值，会传递给p的回调函数。

Promise.race()

Promise.race方法同样是将多个Promise实例，包装成一个新的Promise实例。下面代码中，只要p1、p2、p3之中有一个实例率先改变状态，p的状态就跟着改变。那个率先改变的 Promise 实例的返回值，就传递给p的回调函数。

var p = Promise.race([p1, p2, p3]);

上面代码中，Promise.all方法接受一个数组作为参数，p1、p2、p3都是 Promise 实例，p的状态由p1、p2、p3决定，分成两种情况。

赛跑，谁快用谁的结果。

Promise. resolve()

Promise.resolve()方法将现有对象转为Promise对象，例如：

```
var jsPromise = Promise.resolve($.ajax('/whatever.json'));
```

- ✓ **参数是一个Promise实例**

  Promise.resolve将不做任何修改、原封不动地返回这个实例。

- ✓ **参数是一个thenable对象**

  thenable对象指的是具有then方法的对象，Promise.resolve方法会将这个对象转为Promise对象，然后就立即执行thenable对象的then方法。

- ✓ **参数不是具有then方法的对象，或根本就不是对象**

  如果参数是一个原始值，或者是一个不具有then方法的对象，则Promise.resolve方法返回一个新的Promise对象，状态为Resolved。

- ✓ **不带有任何参数**

  直接返回一个Resolved状态的Promise对象。需要注意的是，立即resolve的Promise对象，是在本轮“事件循环”（event loop）的结束时，而不是在下一轮“事件循环”的开始时。

Promise. reject()

Promise.reject(reason)方法也会返回一个新的 Promise 实例，该实例的状态为 rejected 。

```
var p = Promise.reject('出错了');

// 等同于

var p = new Promise((resolve, reject) => reject('出错了'))
```

finally()

finally方法用于指定不管Promise对象最后状态如何，都会执行的操作。它接受一个普通的回调函数作为参数，该函数不管怎样都必须执行。

下面是一个例子，服务器使用Promise处理请求，然后使用finally方法关掉服务器

```
server.listen(0) .then(function () {
        // run test
 }) .finally(server.stop);
```

杰普软件科技有限公司

www.briup.com

Tel: (021)55660810
Fax: (021)55660802
Email:training@briup.com
Msn: training.sh@hotmail.com
Home:http://www.briup.com

地址：上海市闸北区万荣路1188弄F栋6号三层-上海服务外包科技园龙软园区
邮编：200436

电话：021-56657112
传真：021-55661523-8003
电邮：training@briup.com
主页：http://www.briup.com

Briup High-End IT Training

第 5 章：ES6模块

Brighten Your Way And Raise You Up.

介绍

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。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 在语言标准的层面上，实现了模块功能，而且实现得相当简单，完全可以取代 CommonJS 和 AMD 规范，成为浏览器和服务器通用的模块解决方案。

◆ 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。

```
var firstName = 'Michael';
var lastName = 'Jackson';
var year = 1958;
function multiply(x, y) { return x * y; };
export {firstName, lastName, year, multiply};
```

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，不能直接导出一个值

```
export var m = 1;
```

或 `var m = 1; export {m};`

或 `var n = 1; export {n as m};`

在一个模块中，export可以调用多次

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。

- 解构导入

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

- 重命名变量

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

- 重复导入

```
import {name} from './module1';
import {age} from './module1';
```

如果多次重复执行同一句import语句，那么只会执行一次模块代码。

- 模块的整体加载

```
import * as person from './module1'
```

export default 命令

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 function () {
    console.log('foo');
}
```

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 customName from './export-default';
customName(); // 'foo'
```

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 default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或者对象。

- **export 与 import 的复合写法**

  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 { foo, bar } from 'my_module';
  // 等同于
  import { foo, bar } from 'my_module';
  export { foo, bar }; 。

杰普软件科技有限公司
www.briup.com

Tel: (021)55660810
Fax: (021)55660802
Email:training@briup.com
Msn: training.sh@hotmail.com
Home:http://www.briup.com

地址：上海市闸北区万荣路1188弄F栋6号三层-上海服务外包科技园龙软园区
邮编：200436

电话：021-56657112
传真：021-55661523-8003
电邮：training@briup.com
主页：http://www.briup.com

Briup High-End IT Training

第 6 章：Class

Brighten Your Way And Raise You Up.

介绍

JavaScript 语言中，生成实例对象的传统方法是通过构造函数。 ES6 提供了更接近传统语言的写法，引入了 Class（类）这个概念，作为对象的模板。通过class关键字，可以定义类。

基本上，ES6 的class可以看作只是一个语法糖，它的绝大部分功能，ES5 都可以做到，新的class写法只是让对象原型的写法更加清晰更像面向对象编程的语法而已。所以ES6 的类，完全可以看作构造函数的另一种写法。

```
class Point {
    constructor(x, y) {
        this.x = x; this.y = y;
    }
    toString() {
        return '(' + this.x + ', ' + this.y + ')';
    }
}
```

方法

在类中可以直接定义方法，实际上类的所有方法都定义在类的prototype属性上面。在类的实例上面调用方法，其实就是调用原型上的方法。

```
class Point {
    constructor() { // ... }
    toString() { // ... }
    toValue() { // ... }
}
```

由于类的方法都定义在prototype对象上面，所以类的新方法可以添加在prototype对象上面。Object.assign方法可以很方便地一次向类添加多个方法。

```
class Point {
    constructor(){ // ... }
 }
Object.assign(Point.prototype, {
    toString(){},
    toValue(){}
});
```

◆ constructor 方法

constructor方法是类的默认方法，通过new命令生成对象实例时，自动调用该方法。一个类必须有constructor方法，如果没有显式定义，一个空的constructor方法会被默认添加。

```
class Point { }
// 等同于
class Point { constructor() {} }
```

类必须使用new调用，否则会报错。这是它跟普通构造函数的一个主要区别，后者不用new也可以执行。

◆ 静态方法

类相当于实例的原型，所有在类中定义的方法，都会被实例继承。如果在一个方法前，加上static关键字，就表示该方法不会被实例继承，而是直接通过类来调用，这就称为“静态方法”。

```
class Foo {
    static classMethod() { return 'hello'; }
}
 Foo.classMethod() // 'hello'
```

如果静态方法包含this关键字，这个this指的是类，而不是实例。

实例属性

类的实例属性可以定义在构造函数中。

```
class Person{
    constructor(id,name,age) {
        this.id = id;
        this.name = name;
        this.age = age;
    }
}
```

◆ 静态属性

直接在类上定义的属性成为是静态属性。

```
class Foo {
}
Foo.prop = 1;
Foo.prop // 1
```

目前，只有这种写法可行，因为 ES6 明确规定，Class 内部只有静态方法，没有静态属性。

◆ 继承

class 可以通过extends关键字实现继承，这比 ES5 的通过修改原型链实现继承，要清晰和方便很多。

```
class Animal {
    constructor(name){
        this.name = name;
    }
    sayName(){
        console.log("my name is ",this.name);
    }
}
class Dog extends Animal{
}
```

子类必须在constructor方法中调用super方法，否则新建实例时会报错。这是因为子类没有自己的this对象，而是继承父类的this对象，然后对其进行加工。如果不调用super方法，子类就得不到this对象。子类构造函数可以省略。在子类的构造函数中，只有调用super之后，才可以使用this关键字，否则会报错。

◆ super

super这个关键字，既可以当作函数使用，也可以当作对象使用。在这两种情况下，它的用法完全不同。

- 函数

子类B的构造函数之中的super()，代表调用父类的构造函数。 super虽然代表了父类A的构造函数，但是返回的是子类B的实例，即super内部的this指的是B，因此super()在这里相当于A.prototype.constructor.call(this)。

- 对象

在普通方法中，指向父类的原型对象；在静态方法中，指向父类。由于super指向父类的原型对象，所以定义在父类实例上的方法或属性，是无法通过super调用的。

ES6 规定，通过super调用父类的方法时，super会绑定子类的this。

super.print();

=>

super.print.call(this);

不能直接打印super，因为无法得知super到底是函数还是对象

◆ 类的 prototype 属性和__proto__属性

class 作为构造函数的语法糖，同时有prototype属性和__proto__属性，因此同时存在两条继承链

1. 子类的__proto__属性，表示构造函数的继承，总是指向父类。
2. 子类prototype属性的__proto__属性，表示方法的继承，总是指向父类的prototype属性。

```
class A { }
class B extends A { }
B.__proto__ === A // true
B.prototype.__proto__ === A.prototype // true
```

类的继承是按照下面的模式实现的。

```
class A { }
class B { }
// B 的实例继承 A 的实例
Object.setPrototypeOf(B.prototype, A.prototype);
// B 的实例继承 A 的静态属性
Object.setPrototypeOf(B, A); const b = new B();
```