3_21

1. ECMAScript 6

1.1. 上节回顾

1.1.1. Node.js基础

1.1.2. 掌握基本模块的使用以及开发

1.1.3. NPM的基本操作

1.2. 学习目标

1.2.1. 了解ECMAScript6

1.2.2. ECMAScript6变量声明

1.2.3. ECMAScript6解构赋值

1.3. 教学过程描述

1.3.1. 1

ECMAScript是什么

1996年11月，JavaScript 的创造者 Netscape 公司，决定将 JavaScript 提交给国际标准化组织ECMA，希望这种语言能够成为国际标准。次年，ECMA 发布262号标准文件（ECMA-

262)的第一版，规定了浏览器脚本语言的标准，并将这种语言称为 ECMAScript，这个版本就是1.0版。

该标准从一开始就是针对 JavaScript 语言制定的，但是之所以不叫 JavaScript，有两个原因。一是商标，Java 是 Sun 公司的商标，根据授权协议，只有 Netscape 公司可以合法地使用 JavaScript 这个名字，且 JavaScript 本身也已经被 Netscape 公司注册为商标。二是想体现这门语言的制定者是 ECMA，不是 Netscape，这样有利于保证这门语言的开放性和中立性。

因此，ECMAScript 和 JavaScript 的关系是，前者是后者的规格，后者是前者的一种实现（另外的 ECMAScript 方言还有 Jscript 和 ActionScript）。日常场合，这两个词是可以互换的。

ECMAScript 6简介

ECMAScript 6.0（以下简称 ES6）是 JavaScript 语言的新一代标准，已经在2015年6月正式发布了。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。

浏览器ES6功能测试

http://kangax.github.io/compat-table/es6/

http://ruanyf.github.io/es-checker/

Babel 转码器

Babel 是一个广泛使用的 ES6 转码器，可以将 ES6 代码转为 ES5 代码，从而在现有环境执行。这意味着，你可以用 ES6 的方式编写程序，又不用担心现有环境是否支持。

Babel配置

Babel 是一个广泛使用的 ES6 转码器，可以将 ES6 代码转为 ES5 代码，从而在现有环境执行。这意味着，你可以用

ES6

的方式编写程序，又不用担心现有环境是否支持。

1、全局安装 babel-cli

```
$ npm install -g babel-cli
```

2、创建Babel的配置文件.babelrc，存放在项目的根目录下。

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

3、安装对应的编译版本到项目开发环境下

```
# ES2015转码规则
$ npm install --save-dev babel-preset-es2015

# react转码规则
$ npm install --save-dev babel-preset-react
```

#

ES7不同阶段语法提案的转码规则（共有4个阶段），选装一个

```
$ npm install --save-dev babel-preset-stage-0
$ npm install --save-dev babel-preset-stage-1
$ npm install --save-dev babel-preset-stage-2
$ npm install --save-dev babel-preset-stage-3
```

4、配置文件.babelrc

```
{
    "presets": [
      "es2015",
      "react",
      "stage-2"
    ],
    "plugins": []
}
```

5、修改package.json文件

```
"scripts": {
    "build": "babel es6_src -d es5_lib"
```

```
},
```

1.3.2. 2

let变量声明

ES6新增了let命令，用来声明变量。

它的用法类似于var，但是所声明的变量，只在let命令所在的代码块内有效。

```
{
  let a = 10;
  var b = 1;
}

a // ReferenceError: a is not defined.
b // 1
```

for循环的计数器，就很合适使用let命令。

```
for (let i = 0; i < 10; i++) {}

console.log(i);
//ReferenceError: i is not defined
```

块级作用域

ES5

只有全局作用域和函数作用域，没有块级**作用域，**这带**来很多不合理的**场景。

第一种场景，**内**层变量**可能会覆盖外**层变量。

```
var tmp = new Date();

function f() {
  console.log(tmp);
  if (false) {
    var tmp = 'hello world';
  }
}

f(); // undefined
```

上面代码的原意是，if代码块的外部使用外层的tmp变量，内部使用内层的tmp变量。但是，函数f执行后，输出结果为undefined，原因在于变量提升，导致内层的tmp变量覆盖了外层的tmp变量。

第二种场景，用来计数的循环变量泄露为全局变量。

```
var s = 'hello';

for (var i = 0; i < s.length; i++) {
  console.log(s[i]);
}

console.log(i); // 5
```

上面代码中，变量i只用来控制循环，但是循环结束后，它并没有消失，泄露成了全局变量。

不存在变量提升

var命令会发生”变量提升“现象，即变量可以在声明之前使用，值为undefined。这种现象多多少少是

有些奇怪的，按照一般的逻辑，变量应该在声明语句之后才可以使用。

为了纠正这种现象，let命令改变了语法行为，它所声明的变量一定要在声明后使用，否则报错。

```
// var 的情况
console.log(foo); // 输出undefined
var foo = 2;

// let 的情况
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。这在语法上，称为“暂时性死区”（temporal dead zone，简称 TDZ）

只要块级作用域内存在let命令，它所声明的变量就“绑定”（binding）这个区域，不再受外部的影响。

```
var tmp = 123;

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

上面代码中，存在全局变量tmp，但是块级作用域内let又声明了一个局部变量tmp，导致后者绑定这个块级作用域，所以在let声明变量前，对tmp赋值会报错。

不允许重复声明

let不允许在相同作用域内，重复声明同一个变量。

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

// 报错
function () {
```

```
  let a = 10;
  let a = 1;
}
```

因此，不能在函数内部当前块域中重新声明参数。

```
function func(arg) {
  let arg; // 报错
}

function func(arg) {
  {
    let arg; // 不报错
  }
}
```

1.3.3. 3

const常量声明

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 // 3.1415
```

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

上面代码表明改变常量的值会报错。

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
```

上面代码表示，对于const来说，只声明不赋值，就会报错。

const的作用域与let命令相同：只在声明所在的块级作用域内有效。

```
if (true) {
  const MAX = 5;
}

MAX // Uncaught ReferenceError: MAX is not defined
```

const命令声明的常量也是不提升，同样存在暂时性死区，只能在声明的位置后面使用。

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

上面代码在常量MAX声明之前就调用，结果报错。

const声明的常量，也与let一样不可重复声明。

```
var message = "Hello!";
let age = 25;

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

解构赋值

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）。

数组解构

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

上面代码表示，可以从数组中提取值，按照对应位置，对变量赋值。

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。下面是一些使用嵌套数组进行解构的例子。

```
let [foo, [[bar], baz]] = [1, [[2], 3]];
foo // 1
bar // 2
baz // 3

let [ , , third] = ["foo", "bar", "baz"];
third // "baz"

let [x, , y] = [1, 2, 3];
x // 1
y // 3

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

```
let [x, y, ...z] = ['a'];
x // "a"
y // undefined
z // []
```

如果解构不成功，变量的值就等于undefined。

```
let [foo] = [];
let [bar, foo] = [1];
```

以上两种情况都属于解构不成功，foo的值都会等于undefined。、

解构赋值**允**许**指定默**认值。

```
let [foo = true] = [];
foo // true

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

默认值

解构赋值**允**许**指定默**认值。

```
let [foo = true] = [];
```

```
foo // true

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

默认值可以引用解构赋值的其他变量，但该变量必须已经声明。

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

1.3.4. 4

解构赋值

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）。

对象解构

解构不仅可以用于数组，还可以用于对象。

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

```
foo // "aaa"
bar // "bbb"
```

对象的解构与数组有一个重要的不同。数组的元素是按次序排列的，变量的取值由它的位置决定；而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。

如果变量名与属性名不一致，必须写成下面这样。

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

let obj = { first: 'hello', last: 'world' };
let { first: f, last: l } = obj;
f // 'hello'
l // 'world'
```

也就是说，对象的解构赋值的内部机制，是先找到同名属性，然后再赋给对应的变量。真正被赋值的是后者，而不是前者。

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

```
baz // "aaa"
foo // error: foo is not defined
```

上面代码中，foo是匹配的模式，baz才是变量。真正被赋值的是变量baz，而不是模式foo。

嵌套

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

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

注意，这时p是模式，不是变量，x和y是变量，因此不会被赋值。

字符串解构

字符串也可以解构赋值。这是因为此时，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"
```

类似数组的对象都有一个length属性，因此还可以对这个属性解构赋值。

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

函数参数解构

函数的参数也可以使用解构赋值。

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

add([1, 2]); // 3
```

函数参数的解构也可以使用默认值。

```
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(); // [0, 0]
```

注意，下面的写法会得到不一样的结果。

```
function move({x, y} = { x: 0, y: 0 }) {
  return [x, y];
}

move({x: 3, y: 8}); // [3, 8]
move({x: 3}); // [3, undefined]
move({}); // [undefined, undefined]
move(); // [0, 0]
```

上面代码是为函数move的参数指定默认值，而不是为变量x和y指定默认值，所以会得到与前一种写法不同的结果。

用途

(1)交换变量的值

```
[x, y] = [y, x];
```

上面代码交换变量x和y的值，这样的写法不仅简洁，而且易读，语义非常清晰。

(2)从函数返回多个值

函数只能返回一个值，如果要返回多个值，只能将它们放在数组或对象里返回。有了解构赋值，取出这些值就非常方便。

```
// 返回一个数组

function example() {
  return [1, 2, 3];
}
```

```
var [a, b, c] = example();

// 返回一个对象

function example() {
  return {
    foo: 1,
    bar: 2
  };
}
var { foo, bar } = example();
```

(3)函数参数的定义

解构赋值可以方便地将一组参数与变量名对应起来。

```
// 参数是一组有次序的值
function f([x, y, z]) { ... }
f([1, 2, 3]);

// 参数是一组无次序的值
function f({x, y, z}) { ... }
f({z: 3, y: 2, x: 1});
```

(4)提取JSON数据

解构赋值对提取JSON对象中的数据，尤其有用。

```
var jsonData = {
  id: 42,
  status: "OK",
  data: [867, 5309]
};

let { id, status, data: number } = jsonData;

console.log(id, status, number);
// 42, "OK", [867, 5309]
```

上面代码可以快速提取JSON数据的值。

(5)函数参数的默认值

```
jQuery.ajax = function (url, {
  async = true,
  beforeSend = function () {},
  cache = true,
  complete = function () {},
```

```
  crossDomain = false,
  global = true,
  // ... 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。

(6)获取对象方法

```
let { log, sin, cos } = Math;
```

(7)获取数组的首位和末尾

```
let arr = [1, 2, 3];
let {0 : first, [arr.length - 1] : last} = arr;
first // 1
last // 3
```

1.4. 练习和作业

1.4.1. 搭建好babel转换环境

1.4.2. 练习并熟悉let变量声明操作

1.4.3. 熟悉解构赋值操作

1.5. 当天小结

1.5.1. 了解ECMAScript6

1.5.2. ECMAScript6变量声明

1.5.3. ECMAScript6解构赋值