

命令行工具



在正式讲解ES6新特性之前，我们需要了解一些命令行工具，在日后的课程中，我们会经常用到命令行

常用命令行工具有两种

- ① `CMD` 命令行工具
- ② `PowerShell` 命令行工具

CMD命令行

① 打开命令行窗口

- ① win: 左下角开始，找到运行，点击，输入 `cmd`，回车
- ② win: `win+r` 快速打开命令行窗口
- ③ mac: command + 空格，输入 `terminal`

- ② 选择盘符： 盘符名加冒号 `E:`
- ③ 查看盘符及目录下文件与文件夹： `win:dir mac:ls`
- ④ 清空命令行信息： `win:cls mac:clear`
- ⑤ 进入文件夹或目录： `cd 文件夹名称`
- ⑥ 返回到上一级目录： `cd ../`
- ⑦ 快速补全目录或文件夹名称： `tab`
- ⑧ 创建文件夹： `mkdir 文件夹名称`
- ⑨ 查看历史输入过的命令： 上下按键

PowerShell

- ① 打开方式
 - ① 在开始位置搜索 `PowerShell` 打开
 - ② 在对应目录按住 `shift` + 右键，打开
- ② 其他保持一直

实时效果反馈

1. 如何快速打开 `CMD` 命令行工具：

- A `win+R`
- B `win+E`
- C `win+P`
- D `win+L`

2. `CMD` 命令行中进入文件夹或目录：

- A `win+R`

B mkdir 文件夹名称

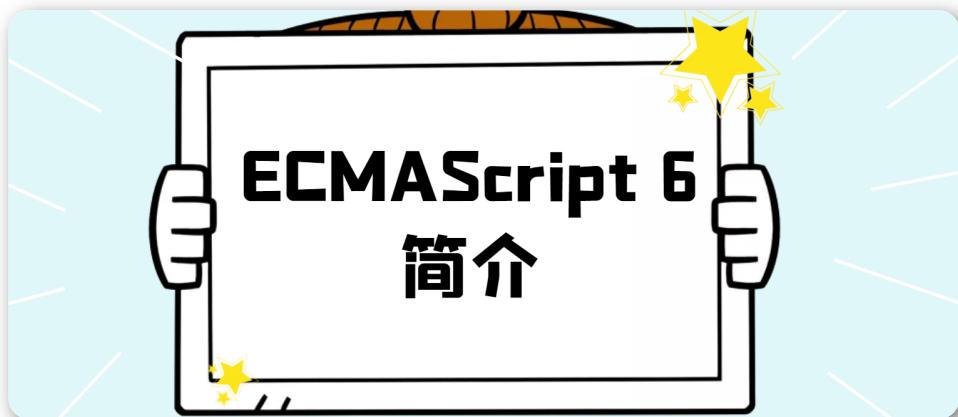
C cd ../

D cd 文件夹名称

答案

1=>A 2=>D

ECMAScript 6 简介



ECMAScript 和 JavaScript 的关系

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，常场合，这两个词是可以互换的。

名称详解

ECMAScript 6 (以下简称 ES6) 是 JavaScript 语言的标准，在 2015 年 6 月发布。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。

版本	官方名称	发布日期
ES1	ECMAScript 1	1997
ES2	ECMAScript 2	1998
ES3	ECMAScript 3	1999
ES4	ECMAScript 4	从未发布过
ES5	ECMAScript 5	2009
ES5.1	ECMAScript 5.1	2011
ES6	ECMAScript 2015 (ECMAScript 6)	2015
ES7	ECMAScript 2016	2016
ES8	ECMAScript 2017	2017
...

因此，ES6 既是一个历史名词，也是一个泛指，含义是 5.1 版以后的 JavaScript 的下一代标准，涵盖了 ES2015、ES2016、ES2017 等等

语法提案的批准流程

任何人都可以向标准委员会（又称 TC39 委员会）提案，要求修改语言标准。

一种新的语法从提案到变成正式标准，需要经历五个阶段。每个阶段的变动都需要由 TC39 委员会批准。

- Stage 0 - Strawman (展示阶段)
- Stage 1 - Proposal (征求意见阶段)
- Stage 2 - Draft (草案阶段)
- Stage 3 - Candidate (候选人阶段)
- Stage 4 - Finished (定案阶段)

一个提案只要能进入 Stage 2，就差不多肯定会包括在以后的正式标准里面。ECMAScript 当前的所有提案，可以在 TC39 的官方网站 [GitHub.com/tc39/ecma262](https://github.com/tc39/ecma262) 查看。

ES6带来的新特性

- ① `let` 和 `const` 命令
- ② 变量的解构赋值
- ③ 字符串扩展
- ④ 函数扩展
- ⑤ 对象扩展
- ⑥ 数组扩展
- ⑦ 运算符扩展
- ⑧ Promise对象
- ⑨ Class
- ⑩ Class 继承
- ⑪ ...

Nodejs环境安装



本节课为前置课程，在接下来的ES6课程中，我们需要先安装 Nodejs环境

Nodejs简介

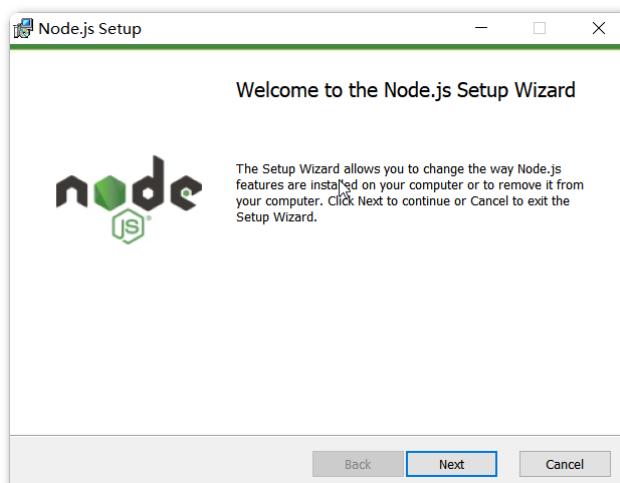
Nodejs诞生于2009年，主攻服务器方向，使得利用 `JavaScript` 也可以完成服务器代码的编写

Nodejs安装

Nodejs官网

<https://nodejs.org/en/>

Nodejs的安装与一般软件一样



大量的库

在安装 Node.js 的同时，会附带一个 npm 命令，npm 是 Node 的包管理工具，这样正是接下来我们要用到的

npm 的简单结构有助于 Node.js 生态系统的激增，现在 npm 仓库托管了超过 1,000,000 个可以自由使用的开源库包

npm 镜像

由于服务器在国外，所以下载速度比较慢，我们可以用国内的镜像

阿里镜像地址

<https://npmmirror.com/>

在命令行运行如下命令即可

```
1 | npm install -g cnpm --  
registry=https://registry.npmmirror.com
```

看到如下信息，代表安装成功

```
C:\Windows\system32\cmd.exe  
Microsoft Windows [版本 10.0.19043.1466]  
(c) Microsoft Corporation。保留所有权利。  
C:\Users\A>npm install -g cnpm --registry=https://registry.npmmirror.com  
npm WARN deprecated uuid@3.4.0: Please upgrade to version 7 or higher. Older versions may use Math.random() in certain  
circumstances, which is known to be problematic. See https://v8.dev/blog/math-random for details.  
npm WARN deprecated request@2.88.2: request has been deprecated, see https://github.com/request/request/issues/3142  
npm WARN deprecated har-validator@5.1.5: this library is no longer supported  
added 11 packages, removed 384 packages, and changed 379 packages in 14s  
npm notice New minor version of npm available! 8.1.2 => 8.4.1  
npm notice Changelog: https://github.com/npm/cli/releases/tag/v8.4.1  
npm notice Run npm install -g npm@8.4.1 to update!  
npm notice  
C:\Users\A>
```

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，从而在老版本的浏览器执行。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。

浏览器支持性查看

<https://caniuse.com/>

Babel官网

<https://babeljs.io/>

转码示例

原始代码用了箭头函数，Babel 将其转为普通函数，就能在不支持箭头函数的 JavaScript 环境执行了

```
1 // 转码前
2 input.map(item => item + 1);
3
4 // 转码后
5 input.map(function (item) {
6   return item + 1;
7 }) ;
```

Babel安装流程

第一步：安装 Babel

```
1 npm install --save-dev @babel/core
```

第二步：配置文件 `.babelrc`

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

```
1 {
2   "presets": [],
3   "plugins": []
4 }
```

第三步：转码规则

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

```
1 | npm install --save-dev @babel/preset-env
```

第四步：将规则加入 .babelrc

```
1 | {  
2 |     "presets": [  
3 |         "@babel/env"  
4 |     ],  
5 |     "plugins": []  
6 | }
```

Babel命令行转码

Babel 提供命令行工具 `@babel/cli`，用于命令行转码

```
1 | npm install --save-dev @babel/cli
```

基本用法如下

```
1 | # 转码结果输出到标准输出  
2 | $ npx babel example.js  
3 |  
4 | # 转码结果写入一个文件  
5 | # --out-file 或 -o 参数指定输出文件  
6 | $ npx babel example.js --out-file  
    compiled.js  
7 | # 或者  
8 | $ npx babel example.js -o compiled.js
```

```
9  
10 # 整个目录转码  
11 # --out-dir 或 -d 参数指定输出目录  
12 $ npx babel src --out-dir lib  
13 # 或者  
14 $ npx babel src -d lib
```

实时效果反馈

1. Babel的作用是什么：

- A Babel是ES6的一部分，是ES6的新特性
- B Babel是ES6的转码器，可以将 ES6 代码转为 ES5 代码
- C Babel是配置文件，配置ES6环境
- D Babel是ES5的基础知识

2. 下列哪个是安装Babel命令转码工具：

- A `npm install --save-dev @babel/core`
- B `npm install --save-dev @babel/preset-env`
- C `npm install --save-dev @babel/cli`
- D `npx babel example.js`

答案

1=>B 2=>C

Let 命令



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块级作用域

```
1 {
2   let itbaizhan = 10;
3   var sxt = 1;
4 }
5
6 itbaizhan // ReferenceError: itbaizhan is not
7 defined.
8 sxt // 1
```

`for` 循环的计数器，就很合适使用 `let` 命令

```
1 for (let i = 0; i < 10; i++) {  
2     // ...  
3 }  
4  
5 console.log(i);  
6 // ReferenceError: i is not defined
```

对比 `var` 和 `let` 在循环中的应用

```
1 var a = [];  
2 for (var i = 0; i < 10; i++) {  
3     a[i] = function () {  
4         console.log(i);  
5     };  
6 }  
7 a[6](); // 10
```

上面代码，输出的 `10`，而我们期待的是 `6`

```
1 var a = [];  
2 for (let i = 0; i < 10; i++) {  
3     a[i] = function () {  
4         console.log(i);  
5     };  
6 }  
7 a[6](); // 6
```

上面代码，输出的 `6`

let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。

```
1 // var 的情况
2 console.log(foo); // 输出undefined
3 var foo = 2;
4
5 // let 的情况
6 console.log(bar); // 报错ReferenceError
7 let bar = 2;
```

let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。

```
1 // 报错
2 function func() {
3     let a = 10;
4     var a = 1;
5 }
6
7 // 报错
8 function func() {
9     let a = 10;
10    let a = 1;
11 }
```

实时效果反馈

1. 下列那个不是Let的特性：

- A Let是块级作用域
- B Let不存在变量提升
- C Let不允许重复声明
- D Let和Var一样并没有区别

答案

1=>D

Const 命令



`const` 声明一个只读的常量。一旦声明，常量的值就不能改变

```
1 const PI = 3.1415;
2 PI // 3.1415
3
4 PI = 3;
5 // 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

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

`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

```
1 if (true) {
2   const MAX = 5;
3 }
4
5 MAX // Uncaught ReferenceError: MAX is not
defined
```

`const` 命令声明的常量也是不存在提升

```
1 if (true) {
2   console.log(MAX); // ReferenceError
3   const MAX = 5;
4 }
```

`const` 声明的常量，也与 `let` 一样不可重复声明

```
1 var message = "Hello!";
2 let age = 25;
3
4 // 以下两行都会报错
5 const message = "Goodbye!";
6 const age = 30;
```

实时效果反馈

1. 下列 `const` 特性描述错误的是：

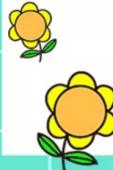
- A `const` 命令声明的常量不提升
- B `const` 命令不可重复声明
- C `const` 命令是块级作用域
- D `const` 命令声明可以改变

答案

1=>D

变量的解构赋值

>>对象解构赋值<<



解构可以用于对象

```
1 | let {name, age} = {name: "iwen", age: 20};
```

温馨提示

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

```
1 | let {age, name} = {name: "iwen", age: 20};  
2 | age // 20
```

```
1 | let {sex, age, name} = {name: "iwen", age: 20};  
2 | sex // undefined
```

对象的解构赋值，可以很方便地将现有对象的方法，赋值到某个变量

```
1 | let { random, floor } = Math;  
2 | let { log } = console;
```

注意事项，如果要将一个已经声明的变量用于解构赋值，必须非常小心

```
1 let hello = "Hello";
2 let { hello } = {hello:"hello"}; // 报错
3
4 let hello = "Hello";
5 ({ hello } = {hello:"hello"}); // 正确
```

实时效果反馈

1. 下列对象解构赋值的代码，输出结果是多少：

```
1 let hello = "Hello";
2 let { hello } = {hello:"hello"};
3 console.log(hello);
```

- A hello
- B Hello
- C 报错
- D undefined

答案

1=>C

字符串扩展



字符串Unicode 表示法

ES6 加强了对 Unicode 的支持，允许采用 `\uxxxx` 形式表示一个字符，其中 `xxxx` 表示字符的 Unicode 码点。

Unicode

统一码（Unicode），也叫万国码、单一码，是计算机科学领域里的一项业界标准，包括字符集、编码方案等。Unicode 是为了解决传统的字符编码方案的局限而产生的，它为每种语言中的每个字符设定了统一并且唯一的二进制编码，以满足跨语言、跨平台进行文本转换、处理的要求。

```
1  "\u0061"
```

```
2  // "a"
```

字符串遍历器接口

for...of 循环遍历

```
1 for (let i of 'itbaizhan') {  
2   console.log(i);  
3 }
```

模板字符串

模板字符串 (template string) 是增强版的字符串，用反引号 (`) 标识。它可以当作普通字符串使用，也可以用来定义多行字符串，或者在字符串中嵌入变量。

```
1 let url = "www.itbaizhan.com"  
2 let h1 = "<a href='"+ url +"'>itbaizhan</a>"  
3 let h2 = `<a href='${url}'>itbaizhan</a>`
```

实时效果反馈

1. 下列字符串模板，表示正确的是：

- A ``
- B ``
- C ``
- D ``

答案

1=>B

字符串新增方法



includes(), startsWith(), endsWith()

传统上，JavaScript 只有 `indexOf` 方法，可以用来确定一个字符串是否包含在另一个字符串中。ES6 又提供了三种新方法。

- ① `includes()`: 返回布尔值，表示是否找到了参数字符串
- ② `startsWith()`: 返回布尔值，表示参数字符串是否在原字符串的头部
- ③ `endsWith()`: 返回布尔值，表示参数字符串是否在原字符串的尾部

```
1 let s = 'Hello world!';  
2  
3 s.startsWith('Hello') // true  
4 s.endsWith('!') // true  
5 s.includes('o') // true
```

这三个方法都支持第二个参数，表示开始搜索的位置

```
1 let s = 'Hello world!';
2
3 s.startsWith('world', 6) // true
4 s.endsWith('Hello', 5) // true
5 s.includes('Hello', 6) // false
```

repeat()

`repeat` 方法返回一个新字符串，表示将原字符串重复 `n` 次。

```
1 'x'.repeat(3) // "xxx"
2 'hello'.repeat(2) // "hellohello"
3 'na'.repeat(0) // ""
```

padStart(), padEnd()

ES2017 引入了字符串补全长度的功能。如果某个字符串不够指定长度，会在头部或尾部补全。`padStart()` 用于头部补全，`padEnd()` 用于尾部补全。

```
1 'x'.padStart(5, 'ab') // 'ababx'
2 'x'.padStart(4, 'ab') // 'abax'
3
4 'x'.padEnd(5, 'ab') // 'xabab'
5 'x'.padEnd(4, 'ab') // 'xaba'
```

trimStart(), trimEnd()

ES2019对字符串实例新增了 `trimStart()` 和 `trimEnd()` 这两个方法。它们的行为与 `trim()` 一致，`trimStart()` 消除字符串头部的空格，`trimEnd()` 消除尾部的空格。它们返回的都是新字符串，不会修改原始字符串。

```
1 const s = ' itbaizhan ';  
2  
3 s.trim() // "itbaizhan"  
4 s.trimStart() // "itbaizhan "  
5 s.trimEnd() // " itbaizhan"
```

at()

`at()` 方法接受一个整数作为参数，返回参数指定位置的字符，支持负索引（即倒数的位置）。

```
1 const str = 'hello';  
2 str.at(1) // "e"  
3 str.at(-1) // "o"
```

温馨提示

如果参数位置超出了字符串范围，`at()` 返回 `undefined`

实时效果反馈

1. 下列字符串方法中，那个可以判断是否包含某个字符串：

- A `includes()`
- B `repeat()`
- C `padStart()`

D

at()

答案

1=>A

数组扩展_扩展运算符



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

```
1 console.log(...[1, 2, 3])
2 // 1 2 3
3
4 console.log(1, ...[2, 3, 4], 5)
5 // 1 2 3 4 5
```

替代函数的 apply 方法

由于扩展运算符可以展开数组，所以不再需要 `apply` 方法，将数组转为函数的参数了

```
1 // ES5 的写法
2 Math.max.apply(null, [14, 3, 77])
3
4 // ES6 的写法
5 Math.max(...[14, 3, 77])
6
7 // 等同于
8 Math.max(14, 3, 77);
```

合并数组

扩展运算符提供了数组合并的新写法

```
1 const arr1 = ['a', 'b'];
2 const arr2 = ['c'];
3 const arr3 = ['d', 'e'];
4
5 // ES5 的合并数组
6 arr1.concat(arr2, arr3);
7 // [ 'a', 'b', 'c', 'd', 'e' ]
8
9 // ES6 的合并数组
10 [...arr1, ...arr2, ...arr3]
11 // [ 'a', 'b', 'c', 'd', 'e' ]
```

实时效果反馈

1. 下列代码，获取数组的最大值，划横线处填写的代码是：

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

A apply

B call

C ...

D concat

答案

1=>C

数组扩展_新增方法



Array.from()

`Array.from` 方法用于将类数组转为真正的数组

温馨提示

常见的类数组有三类：

- ① arguments
- ② 元素集合
- ③ 类似数组的对象

arguments

```
1 function add(){
2     let collect = Array.from(arguments);
3     collect.push(40);
4     console.log(collect);
5 }
6 add(10, 20, 30)
```

元素集合

```
1 let divs = document.querySelectorAll('div');
2 console.log(Array.from(divs));
```

类似数组的对象

```
1 let arrayLike = {  
2     '0': 'a',  
3     '1': 'b',  
4     '2': 'c',  
5     length: 3  
6 };  
7 let arr = Array.from(arrayLike);  
8 console.log(arr);
```

Array.of()

`Array.of()` 方法用于将一组值，转换为数组

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

实时效果反馈

1. 下列代码输出结果是什么：

```
1 | Array.of(3)  
2 | Array(3)
```

A [3] [3]

B [..] [..]

C [..] [3]

D [3] [..]

答案

1=>D

对象的扩展



属性的简洁表示法

ES6 允许在大括号里面，直接写入变量和函数，作为对象的属性和方法。这样的书写更加简洁。

```
1 let name = "iwen"
2 const user = {
3     name,
4     age:20
5 }
```

除了属性简写，方法也可以简写

```
1 const o = {  
2     method() {  
3         return "Hello!";  
4     }  
5 };  
6  
7 // 等同于  
8  
9 const o = {  
10    method: function() {  
11        return "Hello!";  
12    }  
13};
```

这种写法用于函数的返回值，将会非常方便

```
1 function getPoint() {  
2     const x = 1;  
3     const y = 10;  
4     return {x, y};  
5 }  
6  
7 getPoint() // {x:1, y:10}
```

属性名表达式

ES6 允许字面量定义对象时，用表达式作为对象的属性名，即把表达式放在方括号内

```
1 let propKey = 'itbaizhan';
2
3 let obj = {
4     [propKey]: true,
5     ['a' + 'bc']: 123
6 };
```

对象的扩展运算符

ES2018 将这个运算符引入了对象

```
1 let z = { a: 3, b: 4 };
2 let n = { ...z };
3 console.log(n);
4
5 {...{}, a: 1}
6 // { a: 1 }
```

实时效果反馈

1. 下列代码输出结果是什么：

```
1 let propKey = 'itbaizhan';
2 let obj = {
3     [propKey]: true,
4     ['a' + 'bc']: 123
5 };
6 console.log(obj.propKey);
```

A

true

B 报错

C undefined

D propKey

答案

1=>C

函数的扩展_箭头函数



基本用法

ES6 允许使用“箭头” (`=>`) 定义函数

```
1 var add = (x) => x;  
2  
3 // 等同于  
4 var add = function (x) {  
5     return x;  
6 };
```

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

```
1 var add = (x,y) => x+y;  
2 // 等同于  
3 var add = function (x,y) {  
4     return x+y;  
5 };  
6  
7 var add = () => 100;  
8 // 等同于  
9 var add = function () {  
10    return 100;  
11 };
```

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

```
1 var add = (x,y) => {
2     var z = 10;
3     return x+y+z
4 };
5
6 // 等同于
7 var add = function (x,y) {
8     var z = 100
9     return x+y+z
10};
```

由于大括号被解释为代码块，所以如果箭头函数直接返回一个对象，必须在对象外面加上括号，否则会报错。

```
1 var add = (x,y) => ({x:10,y:20});
```

箭头函数的一个用处是简化回调函数（匿名函数）

```
1 var arr = [10,20,30]
2 arr.map(item =>{
3     console.log(item);
4 })
```

使用注意点

对于普通函数来说，内部的 `this` 指向函数运行时所在的对象，但是这一点对箭头函数不成立。它没有自己的 `this` 对象，内部的 `this` 就是定义时上层作用域中的 `this`

```
1 var name = "itbaizhan"
2 var user = {
3     name:"iwen",
4     getName(){
5         setTimeout(() =>{
6             console.log(this.name); // iwen
7         })
8     }
9 }
10 user.getName()
```

温馨提示

箭头函数里面根本没有自己的 this，而是引用外层的 this

实时效果反馈

1. 下列代码运行结果是多少：

```
1 var name = "itbaizhan"
2 var user = {
3     name:"iwen",
4     getName(){
5         setTimeout(() =>{
6             console.log(this.name);
7         })
8     }
9 }
10 user.getName()
```

A itbaizhan

B iwen

C null

D 报错

答案

1=>B

Set 数据结构



基本用法

ES6 提供了新的数据结构 Set。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。

```
1 const s = new Set();
2
3 [2, 3, 5, 4, 5, 2, 2].forEach(x => s.add(x));
4
5 for (let i of s) {
6   console.log(i);
7 }
8 // 2 3 5 4
```

通过 `add()` 方法向 Set 结构加入成员，结果表明 Set 结构不会添加重复的值。

`Set` 函数可以接受一个数组作为参数

```
1 const set = new Set([1, 2, 3, 4, 4]);
2 [...set]
3 // [1, 2, 3, 4]
```

数组去除重复成员的方法

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

字符串去除重复字符

```
1 [...new Set('ababbc')].join('')
2 // "abc"
```

向 Set 加入值的时候，不会发生类型转换，所以 5 和 "5" 是两个不同的值。

```
1 var mySet = new Set();
2 mySet.add("5")
3 mySet.add(5)
4 console.log(mySet); // Set(2) {'5', 5}
```

size 属性

返回 Set 实例的成员总数

```
1 const items = new Set([1, 2, 3, 4, 5, 5, 5,
2 5]);
2 items.size // 5
```

实时效果反馈

1. 下列代码输出结果是多少：

```
1 var mySet = new Set();
2 mySet.add("5")
3 mySet.add(5)
4 mySet.add(5)
5 console.log(mySet);
```

A "5" 5 5

B "5" 5

C "5"

D

null

答案

1=>B

Set 数据结构方法



add()

set 添加方法

```
1 var mySet = new Set();
2 mySet.add("5")
3 console.log(mySet);
```

delete()

删除某个值，返回一个布尔值，表示删除是否

```
1 var mySet = new Set();
2 mySet.add("5")
3 var flag = mySet.delete("5");
4 console.log(flag); // true
```

has()

返回一个布尔值，表示该值是否为 `Set` 的成员

```
1 var mySet = new Set();
2 mySet.add("5")
3 var flag = mySet.has("5");
4 console.log(flag); // true
```

clear()

清除所有成员，没有返回值

```
1 var mySet = new Set();
2 mySet.add("5")
3 mySet.clear();
4 console.log(mySet); // Set(0) {size: 0}
```

实时效果反馈

1. 下列那个方法可以清空 `Set` 数据结构中所有的数据：

A add()

B delete()

C has()

D clear()

答案

1=>D

Promise 对象



基本概念

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

ES6 规定，`Promise` 对象是一个构造函数，用来生成 `Promise` 实例

```
1 const promise = new Promise(function(resolve, reject) {  
2     // ... some code  
3  
4     if (/* 异步操作成功 */){  
5         resolve(value);  
6     } else {  
7         reject(error);  
8     }  
9 });
```

`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

`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。

```
1 promise.then(function(value) {  
2     // success  
3 }, function(error) {  
4     // failure  
5 });
```

加载图片资源例子

```
1 <!DOCTYPE html>  
2 <html lang="en">  
3 <head>  
4     <meta charset="UTF-8">  
5     <meta http-equiv="X-UA-Compatible"  
content="IE=edge">  
6     <meta name="viewport"  
content="width=device-width, initial-  
scale=1.0">  
7     <title>Document</title>  
8 </head>  
9 <body>  
10    <div></div>  
11    <script>  
12        function loadImageAsync(url) {  
13            var promise = new  
Promise(function (resolve, reject) {  
14                const image = new Image();  
15                image.onload = function () {  
16                    resolve(image);  
17                };  
18                image.onerror = function () {  
19                };
```

```
19             reject(new Error('Could
  not load image at ' + url));
20         };
21         image.src = url;
22     });
23     return promise;
24 }
25
26 loadImageAsync("http://iwenwiki.com/api/vue
-data/vue-data-1.png")
27     .then(function(data){
28         console.log(data);
29         $("div").append(data)
30     },function(error){
31         $("div").html(error)
32     })
33 </script>
34 </body>
35 </html>
```

实时效果反馈

1. **Promise** 的作用是什么，下列描述正确的是：

- A **Promise** 是异步编程的一种解决方案，可以将异步操作以同步操作的流程表达出来
- B **Promise** 是同步编程的一种解决方案，可以将同步操作以异步操作的流程表达出来
- C **Promise** 使得控制同步操作更加容易
- D **Promise** 还不是ES6的标准，目前是社区版本

答案

1=>A

Promise对象_Ajax实操



Promise封装Ajax，让网络请求的异步操作变得更简单

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta http-equiv="X-UA-Compatible"
content="IE=edge">
```

```
6      <meta name="viewport"  
7          content="width=device-width, initial-  
8          scale=1.0">  
9      <title>Document</title>  
10     </head>  
11     <body>  
12         <script>  
13             const getJSON = function (url) {  
14                 const promise = new  
15                 Promise(function (resolve, reject) {  
16                     const handler = function ()  
17                     {  
18                         if (this.readyState !==  
19                             4) {  
20                             return;  
21                         }  
22                         if (this.status === 200)  
23                         {  
24                             resolve(this.response);  
25                         } else {  
26                             reject(new  
Error(this.statusText));  
27                         }  
28                     };  
29                     const client = new  
30                     XMLHttpRequest();  
31                     client.open("GET", url);  
32                     client.onreadystatechange =  
33                     handler;  
34                     client.responseText =  
35                     "json";  
36                 });  
37             };  
38         </script>  
39     </body>  
40 </html>
```

27

```
    client.setRequestHeader("Accept",
"application/json");
            client.send();
        });
        return promise;
    };
}

getJSON("http://iwenwiki.com/api/blueberrypa
i/getIndexBanner.php").then(function (json)
{
    console.log(json);
}, function (error) {
    console.error('出错了', error);
});
</script>
</body>
</html>
```

Async 函数



ES2017 标准引入了 `async` 函数，使得异步操作变得更加方便

`async` 函数可以将异步操作变为同步操作

示例代码

```
1 function print(){
2     setTimeout(() =>{
3         console.log("定时器");
4     ,1000)
5         console.log("Hello");
6     }
7
8     print()
```

基本语法

```
1 function timeout(ms) {
2     return new Promise((resolve) => {
3         setTimeout(resolve, ms);
4     });
5 }
6
7 async function asyncPrint(value, ms) {
8     await timeout(ms);
9     console.log(value);
10}
11
12 asyncPrint('hello world', 50);
```

异步应用

```
1 function ajax(url){
2     return new
3     Promise(function(resolve,reject){
4         $.getJSON(url,function(result){
5             resolve(result)
6         },function(error){
7             reject(error)
8         })
9     })
10
11 async function getInfo(){
12     let ids = await
13     ajax("http://iwenwiki.com/api/generator/list
14     .php")
```

```
13 let names = await
  ajax("http://iwenwiki.com/api/generator/id.p
  hp?id="+ids[0])
14 let infos = await
  ajax("http://iwenwiki.com/api/generator/name
  .php?name=" + names.name)
15 console.log(infos);
16 }
17
18 getInfo();
```

实时效果反馈

1. **Async** 是什么：

- A Async是完成网络请求，如Ajax一样
- B Async的作用是完成异步网络请求，如Ajax一样
- C Async使得异步操作变得更加方便
- D Async是新的网络请求解决方案

答案

1=>C

Class 的基本语法



类的由来

JavaScript 语言中，生成实例对象的传统方法是通过构造函数

```
1 function Point(x, y) {
2     this.x = x;
3     this.y = y;
4 }
5
6 Point.prototype.toString = function () {
7     return '(' + this.x + ', ' + this.y + ')';
8 };
9
10 var p = new Point(1, 2);
```

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

```
1 class Point {  
2     constructor(x, y) {  
3         this.x = x;  
4         this.y = y;  
5     }  
6  
7     toString() {  
8         return '(' + this.x + ', ' + this.y +  
9             ')';  
10    }  
11 }
```

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

```
1 class Point {  
2 }  
3  
4 // 等同于  
5 class Point {  
6     constructor() {}  
7 }
```

类的实例

生成类的实例的写法，与 ES5 完全一样，也是使用 `new` 命令

```
1 class Point {  
2     // ...  
3 }  
4  
5 // 报错  
6 var point = Point(2, 3);  
7  
8 // 正确  
9 var point = new Point(2, 3);
```

注意点

不存在提升

类不存在变量提升 (hoist)，这一点与 ES5 完全不同

```
1 new Foo(); // ReferenceError  
2 class Foo {}
```

实时效果反馈

1. 下列代码，划横线处要填写的代码时：

```
1 class Person {  
2     constructor(name) {  
3         this.name = name;  
4     }  
5 }  
6  
7 var p = ___.Person("sxt")
```

- A
- B new
- C this
- D class

答案

1=>B

Class属性与方法

Class 属性与方法

实例方法

通过类的实例对象调用方法

```
1 class People{  
2     say(){  
3         console.log("Hello");  
4     }  
5 }  
6 var p = new People();  
7 p.say()
```

实例属性

实例属性指的是类的实例对象可调用的属性

```
1 class People{  
2     constructor(name,age){  
3         this.name = name;  
4         this.age = age;  
5     }  
6     say(){  
7         console.log(this.name,this.age);  
8     }  
9 }  
10 var p = new People("iwen",20);  
11 p.say()  
12 console.log(p.name,p.age);
```

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”

```
1 class Person {  
2     static classMethod() {  
3         console.log("Hello");  
4     }  
5 }  
6  
7 Person.classMethod() // Hello  
8  
9 var p = new Person();  
10 p.classMethod() // p.classMethod is not a  
function
```

温馨提示

注意，如果静态方法包含 `this` 关键字，这个 `this` 指的是类，而不是实例。

```
1 class People {  
2     static getSay() {  
3         this.say();  
4     }  
5     static say() {  
6         console.log('hello');  
7     }  
8     say() {  
9         console.log('world');  
10    }  
11 }  
12 People.getSay() // hello
```

静态属性

静态属性指的是 Class 本身的属性，即 `Class.propName`

```
1 class People{}  
2 People.status = "等待"  
3 console.log(People.status);
```

实时效果反馈

1. 下列代码，运行结果是什么：

```
1 class People {
2     static getSay() {
3         this.say();
4     }
5     static say() {
6         console.log('hello');
7     }
8     say() {
9         console.log('world');
10    }
11 }
12 People.getSay()
```

- A hello
- B world
- C null
- D undefined

答案

1=>A

Class 的继承



基础用法

Class 可以通过 `extends` 关键字实现继承，让子类继承父类的属性和方法。`extends` 的写法比 ES5 的原型链继承，要清晰和方便很多

```
1 class Point {  
2 }  
3  
4 class ColorPoint extends Point {  
5 }
```

ES6 规定，子类必须在 `constructor()` 方法中调用 `super()`，否则就会报错，这是因为子类自己的 `this` 对象，必须先通过父类的构造函数完成塑造，得到与父类同样的实例属性和方法，然后再对其进行加工，添加子类自己的实例属性和方法。如果不调用 `super()` 方法，子类就得不到自己的 `this` 对象

```
1 class Point {  
2     constructor(x,y){  
3         this.x = x;  
4         this.y = y;  
5     }  
6     getPoint(){  
7         console.log(this.x,this.y);  
8     }  
9 }
```

```
8    }
9 }
10 class ColorPoint extends Point {
11     constructor(x,y,z){
12         super(x,y)
13         this.z = z;
14     }
15 }
16 let cp = new ColorPoint(10,20,30)
17 cp.getPoint();
```

Module 的语法



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。

ES6 模块是通过 `export` 命令显式指定输出的代码，再通过 `import` 命令输入。

```
1 | export var Hello = "hello" // hello.js文件
2 | import { Hello } from "./hello.js" //
   | index.js文件
```

测试方式

我们采用Nodejs方式进行测试Module语法

但是nodejs采用的是CommonJS的模块化规范，使用require引入模块；而import是ES6的模块化规范关键字。想要使用import，必须引入babel转义支持，通过babel进行编译，使其变成node的模块化代码。

疑惑：为啥不用前端方式测试，前端方式测试会更加麻烦

第一步：全局安装babel-cli `npm install -g babel-cli`

第二步：安装 babel-preset-env `npm install -D babel-preset-env`

第三步：运行代码 `babel-node --presets env index.js`

export 命令

export命令导出变量

```
1 export var firstName = 'sxt';
2 export var lastName = 'itbaizhan';
3 export var year = 2000;
```

export命令导出函数

```
1 export function add(x, y) {
2     return x + y;
3 };
```

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

```
1 // name.js
2 export var firstName = 'sxt';
3 export var lastName = 'itbaizhan';
4 export var year = 2000;
5
6 // main.js
7 import { firstName, lastName, year } from
'./profile.js';
```

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

```
1 // value.js
2 export var value = 1;
3
4 // main.js
5 import { value as val } from './value.js';
```

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

```
1 // circle.js
2 export function area(radius) {
3     return Math.PI * radius * radius;
4 }
5 export function circumference(radius) {
6     return 2 * Math.PI * radius;
7 }
8
9
10 // main.js
11 import { area, circumference } from
12     './circle';
13 // 可以修改如下
14 import * as circle from './circle';
```

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。

```
1 // export-default.js
2 export default function () {
3     console.log('foo');
4 }
```

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

```
1 // import-default.js
2 import customName from './export-default';
3 customName(); // 'foo'
```

实时效果反馈

1. 下列那个不属于ES6的Module语法的关键字：

- A `export`
- B `import`
- C `export default`
- D `require`

答案

1=>D

