# ES6

1. 声明变量let和const
2. let

（1）作用域：块级作用域，由花括号组成的 {........}。

块级作用域定义的变量，外部是无法进行访问的。

|  |
| --- |
| if (true) **{**  let a = 110;  **}**  **{**  let b = 120;  **}** |

1. let声明的变量不存在预解析

|  |
| --- |
| if (true) {  console.log(a);  let a = 110;  }  1 |

（3）let声明的变量不允许重复（同一个作用域内，在不同作用域下可以）

|  |
| --- |
| 同个作用域下  if (true) {  let a = '11';  let a = '22';  console.log(a);  }  1 |
| 不同作用域下，则没有问题。  if (true) {  let a = '11';  console.log(a);  }  {  let a = '22';  console.log(a);  } |

（4）for循环括号中声明的变量只能在循环体中使用

|  |
| --- |
| for (let i=0; i < 5; i++) {  console.log(i);  }  console.log(i); // 报错 i 没有定义 |

1. 在块级作用域内部，变量只能先声明再使用（即不能进行预解析，不能进行变量的提升）

（6）声明let变量之前的那部分区域，是不允许访问的，那部分就叫做暂时性死区，蓝色区域

|  |
| --- |
| if (true) {  console.log(a);  let a = 110;  } |

2、const声明常量

（1）跟let的规则是一样的，除了for循环的那个变量，它是需要变化的

（2）const声明的常量不允许重新赋值

（3）const声明的常量必须初始化

如果不初始化，那就是空，空就没有什么意义，又不能给它重新赋值。

|  |
| --- |
| if (true) {  const a;  a = 1;  }  1 |

二、变量的解构赋值

1. 引入

　　我们经常定义许多对象和数组，然后有组织地从中提取相关的信息片段。在ES6中添加了可以简化这种任务的新特性：解构。解构是一种打破数据结构，将其拆分为更小部分的过程。

在ES5中，开发者们为了从对象和数组中获取特定数据并赋值给变量，编写了许多看起来同质化的代码。

|  |
| --- |
| let options = {  repeat: true,  save: false  };  // 从对象中提取数据  let repeat = options.repeat,  save = options.save; |

这段代码从options对象中提取repeat和save的值，并将其存储为同名局部变量，提取的过程极为相似。

如果要提取更多变量，则必须依次编写类似的代码来为变量赋值，如果其中还包含嵌套结构，只靠遍历是找不到真实信息的，必须要深入挖掘整个数据结构才能找到所需数据。

所以ES6添加了解构功能，将数据结构打散的过程变得更加简单，可以从打散后更小的部分中获取所需信息。

2、数组解构赋值

　　它使用的是数组字面量，且解构操作全部在数组内完成，而不是像对象字面量语法一样使用对象的命名属性。a、b、c是变量名。

|  |
| --- |
| let [a,b,c] = [1,2,3];  console.log(a,b,c); |
| 等价于  let a = 1, b = 2, c = 3;  console.log(a,b,c); |

　　在数组解构语法中，我们通过值在**数组中的位置**进行选取，且可以将其存储在任意变量中，未显式声明的元素都会直接被忽略

（1）在解构模式中，也可以直接省略元素，只为感兴趣的元素提供变量名

|  |
| --- |
| let [, , blue] = ['red','green','blue'];  console.log(blue); |
| 占位符，　这段代码使用解构赋值语法从 [] 中获取第3个元素，blue前的逗号是前方元素的无论数组中的元素有多少个，都可以通过这种方法提取想要的元素，不需要为每一个元素都指定变量名。 |

|  |
| --- |
| let [**red = 'red'**, , blue] = [ ,'green','blue'];  console.log(red, blue); |

3、对象的解构赋值

（1）使用解构赋值表达式时，如果指定的**局部变量名称**在对象中不存在，那么这个局部变量会被赋值为undefined。

|  |
| --- |
| let people = {  name: 'Mike',  age: 25,  }  let {name, age} = people;  console.log(name,); |

（2）当指定的属性不存在时，可以随意定义一个默认值，在属性名称后添加一个等号(=)和相应的默认值即可

|  |
| --- |
| let people = {  name: 'Mike',  age: 25,  }  let {name, age, friend='Jon'} = people;  console.log(name,age, friend); |

（3）可以获取内置对象的函数

|  |
| --- |
| let {sin, cos, random} = **Math**;  console.log(typeof sin);  console.log(typeof cos);  console.log(**random()**); |
| 1 |

（4）对象的属性如果给了别名，那么它在使用的时候，只能用别名。

|  |
| --- |
| let people = {  name: 'Mike',  age: 25,  sex: 'gg'  }  let {name, age, **sex: boy**} = people;  console.log(name,age, **boy**);  如果是sex，会报错，sex未定义 |

4、字符串的解构赋值

|  |
| --- |
| let [a, b, c, d] = 'people';  console.log(a, b, c, d);  1 |

1. 如果想获取字符串的length属性，就得把字符串当成字符串对象来处理，因为length本身就是它的属性

|  |
| --- |
| let **{**length**}** = 'people';  console.log(length);  输出的结果为 6 |

5、字符串相关扩展（API）

（1）includes()

判断字符串中有没有特定的字符串，返回 true、false

第二个参数是从哪个索引开始匹配

1. startsWith()

判读字符串是否以指定的字符串开始

在判断请求url参数的开头

1. endsWith()

判读字符串是否以指定的字符串结束

判断一些后缀

|  |
| --- |
| let url = 'http://index.html';  console.log(url.**includes**('index', 7));  console.log(url.**startsWith**('http'));  console.log(url.**endsWith**('.html')); |

三、模板字符串

一些其他的知识，可在红宝书上有。跟一些普通的字符串用法还是有差别的，例如换行。

在拼接html标签的时候，比较有用，维护比较方便，有html那种结构

**反引号**表示模板，模板中的内容可以有格式，通过 **${...}** 方式填充数据。{...}里边可以是**函数调用**，或者一些**表达式**如 1+1。

|  |
| --- |
| let people = {  name: 'Mike',  age: '25',  sex: 'male'  }  let tag = '<div><span>'+people.name+'</span><span>'+people.age+'</span><span>'+people.sex+'</span></div>';  console.log(tag);  function sayHello() {  return 'hello';  }  let templateStr = `  <div>  <span>${people.name}</span>  <span>${people.age}</span>  <span>${people.sex}</span>  <span>${1+1}</span>  <span>${sayHello()}</span>  </div>  `;  console.log(templateStr); |
| 1 |

四、函数拓展

1、参数默认值

|  |
| --- |
| function say(name) {  var str = name || 'Mike';  console.log(str);  }  say('king'); |
| function say(**name = 'Mike'**) {  console.log(name);  }  say('Hi'); |

2、参数解构赋值

|  |
| --- |
| function info(**{name, age} = {}**) {  console.log(name, age);  }  let people = {  name: 'Mike',  age: 35,  }  info(people); |

3、rest参数（剩余参数...parma），把剩余的参数组成一个数组

|  |
| --- |
| function say(a, **...parma**) {  console.log(a);  console.log(Object.prototype.toString.call(parma));  console.log(parma);  }  say(1,2,3,4,5,6); |
| 1 |

4、...扩展运算符（三个点），把数组拆散成单个参数

|  |
| --- |
| function add(a, b, c, d) {  var sum = a + b + c + d;  console.log(sum);  }  let arr = [1,2,3,4];  // add.apply(null, arr);  add(**...arr**); |

（1）它的作用，来合并数组

|  |
| --- |
| let arr1 = [1,2,3];  let arr2 = ['a', 'b', 'c'];  let result = **[...arr1, ...arr2];**  console.log(result); |
| 1 |

5、箭头函数，在nodejs环境中是不会指向window的

跟函数表达式比较类似。

（1）当只有一个参数且直接返回时，可以这样写

|  |
| --- |
| function fn(**v**) {  **return v;**  }  console.log(fn('hello')); |
| let fn = v => v;  console.log(fn('Hi')); |

|  |
| --- |
| // function add(a,b) {  // let result = a + b;  // console.log(result);  // }  let add = (a, b) => {  let result = a + b;  console.log(result);  }  add(10,10); |

（2）箭头函数作为箭头匿名函数

|  |
| --- |
| let arr = [1,2,3,4];  arr.forEach(**(element, index) => {**  **console.log(element, index);**  **}**); |

（3）箭头函数的注意事项

函数中的this取决于函数的定义，在哪里定义就是指谁，而不是调用。

箭头函数体内的this对象，就是**定义该函数时所在的作用域指向的对象**，而不是使用时所在的作用域指向的对象。

|  |
| --- |
| 在全局定义指向了 {} ？？？在命令行是能直接输出顶级对象的，暂时无解。  let people = {  name: 'Mike',  age: '24'  }  let add = () => {  console.log(this);  }  add.call(people);  输出 {} |

在js的全局箭头函数中的this，以及全局作用下的this指向**这个模块导出的对象exports，其他两种导出方式不会影响。**

|  |
| --- |
| console.log(this);  1 |
| let a = 1;  exports.a = a;  console.log(this);  1 |

|  |
| --- |
| let people = {  name: 'Mike',  age: '24'  }  function time() {  console.log(this);  setTimeout(() => {  console.log(this); // 如果是普通匿名函数，this指向的是node.js的运行环境。  console.log(this.age);  }, 1000);  }  time.call(people);  1 |

箭头函数不可以new

箭头函数不可以使用arguments获取参数列表，可以使用rest参数代替

|  |
| --- |
| let add = (**...parma**) => {  console.log(**parma**);  }  add(11,22,33,44); |
| 在箭头函数中的arguments指的是另外的东西  1 |

五、类与继承

1、class，类

静态方法（它只能通过类名去调用，不可以使用实例对象的调用）

|  |
| --- |
| **class** Animal {  **constructor**(name) {  this.name = name;  }  sayName() {  console.log(this.name);  }  // 静态方法 有static关键字修饰  **static** eat() {  console.log('eating');  }  }  let tom = new Animal('Tom');  tom.sayName();  Animal.eat(); |

1. extends，类的继承

|  |
| --- |
| class Dog **extends** Animal {  constructor(name, color) {  // super 用来调用父类的构造函数  **super**(name);  this.color = color;  }  hobby() {  console.log(this.name + ' like playing');  }  }  let tom = new Dog('Tom', 'white');  Dog.eat();  tom.sayName();  tom.hobby();  1 |