# 《ES6》

## # ES6新特性：

\* let const关键字

\* 解构赋值

\* 箭头操作符

\* 参数默认值

\* iterator接口

\* promise，generator，async等异步操作

\* 类的支持

## 1.Let

#### 特点：

\* 块级作用域（let声明的变量只在它所在的代码块有效）

\* 不存在变量提升

\* 暂时性死区

\* 不允许重复声明

## 2.const

#### 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。

\* 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。

\* `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。

\* `const`声明的常量，也与`let`一样不可重复声明。

## 3.ES6 声明变量的六种方法

ES5 只有两种声明变量的方法：`var`命令和`function`命令。

ES6 除了添加`let`和`const`命令，另外两种声明变量的方法：`import`命令和`class`命令

# 《解构赋值》

## 一、数组的解构赋值

#### **1.基本用法**：

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

#### 2.操作特点

**1）解构不成功**

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`。

**2）不完全解构**

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

x // 1

y // 2

**3）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'

```

\*\*注意：\*\*如果解构不成功就使用默认值

\* ES6 内部使用严格相等运算符（`===`），判断一个位置是否有值。所以，只有当一个数组成员严格等于`undefined`，默认值才会生效。

```javascript

let [x = 1] = [undefined];

x // 1

let [x = 1] = [null];

x // null

```

上面代码中，如果一个数组成员是`null`，默认值就不会生效，因为**`null`不严格等于`undefined`。**

**4）惰性求值**

如果默认值是一个表达式，那么这个表达式是惰性求值的，即只有在用到的时候，才会求值。

# 二、对象的解构赋值

**1.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。

# 三、字符串的解构赋值

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。

```javascript

let {length : len} = 'hello';

len // 5

# 四、函数参数的解构赋值

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]

# \*怎么设置默认值（面试）

* 利用if条件判断
* 解构赋值
* 函数参数直接设置

# 五、用途

**（1）交换变量的值**

```javascript

let x = 1;

let y = 2;

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

**（2）从函数返回多个值**

// 返回一个数组

function example() {

return [1, 2, 3];

}

let [a, b, c] = example();

// 返回一个对象

function example() {

return {

foo: 1,

bar: 2

};

}

let { 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。

```javascript

// 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。

let 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

};

```

# 《新增字符串》

### 1.查找字符串：

\*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。

- **includes()：**返回布尔值，表示是否找到了参数字符串。

- **startsWith()**：返回布尔值，表示参数字符串是否在原字符串的\*\*头部\*\*。

- **endsWith()**：返回布尔值，表示参数字符串是否在原字符串的\*\*尾部\*\*。

### 2.重复：repeat()

`repeat`方法返回一个新字符串，表示将原字符串重复`n`次。

```javascript

'x'.repeat(3) // "xxx"

'hello'.repeat(2) // "hellohello"

'na'.repeat(0) // ""

```

### 3.补全：padStart()，padEnd()

`padStart()`用于头部补全，`padEnd()`用于尾部补全。

```javascript

'x'.padStart(5, 'ab') // 'ababx'

'x'.padStart(4, 'ab') // 'abax'

'x'.padEnd(5, 'ab') // 'xabab'

'x'.padEnd(4, 'ab') // 'xaba'

```

如果原字符串的长度，\*\*等于或大于最大长度\*\*，则字符串补全\*\*不生效\*\*，返回原字符串。

```javascript

'xxx'.padStart(2, 'ab') // 'xxx'

'xxx'.padEnd(2, 'ab') // 'xxx'

```

如果用来补全的字符串与原字符串，两者的长度之和\*\*超过了最大长度\*\*，则会\*\*截去\*\*超出位数的补全字符串。

```javascript

'abc'.padStart(10, '0123456789')

// '0123456abc'

```

如果\*\*省略\*\*第二个参数，默认使用\*\*空格补\*\*全长度。

```javascript

'x'.padStart(4) // ' x'

'x'.padEnd(4) // 'x '

### 4.消除空格：trimStart()，trimEnd()

* `trim()` 消除前后空格，
* `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。

const s = ' abc ';

s.trim() // "abc"

s.trimStart() // "abc "

s.trimEnd() // abc"

# 《模板字符串》

模板字符串是增强版的字符串，**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。

如果使用模板字符串表示多行字符串，所有的空格和缩进都会被保留在输出之中。

$('#list').html(**`**

<ul>

<li>first</li>

<li>second</li>

</ul>

**`**);

说明：所有模板字符串的空格和换行，都是被保留的，比如`<ul>`标签前面会有一个换行。如果你不想要这个换行，可以使用`trim`方法消除它。

$('#list').html(`

<ul>

<li>first</li>

<li>second</li>

</ul>

`.trim());

**\* 模板字符串中嵌入变量，需要将变量名写在`${}`之中.\*\***

<body>

<div id="con"></div>

</body>

<script>

var data={"name":"zk","age":"20"}

let str=`

<ul>

<li>${data.name}</li>

<li>${data.age}</li>

</ul>`;

document.getElementById("con").innerHTML=str;

</script>

**\* 大括号内部可以放入任意的 JavaScript 表达式，可以进行运算，以及引用对象属性。\***

let x = 1;

let y = 2;

`${x} + ${y} = ${x + y}`

// "1 + 2 = 3"

`${x} + ${y \* 2} = ${x + y \* 2}`

// "1 + 4 = 5"

let obj = {x: 1, y: 2};

`${obj.x + obj.y}`

// "3"

```

**\*模板字符串之中还能调用函数。\***

function fn() {

return "Hello World";

}

`foo ${fn()} bar`

// foo Hello World bar

**\*如果模板字符串中的变量没有声明，将报错。\*\***

// 变量place没有声明

let msg = `Hello, ${place}`;

// 报错

```

**\* \*\*如果大括号内部是一个字符串，将会原样输出。\*\***

```javascript

`Hello ${'World'}`

// "Hello World"

```

**\* \*\*模板字符串甚至还能嵌套。\*\***

const tmpl = addrs => `

<table>

${addrs.map(addr => `

<tr><td>${addr.first}</td></tr>

<tr><td>${addr.last}</td></tr>

`).join('')}

</table>

`;

# 《指数运算符》

ES2016 新增了一个指数运算符（`\*\*`）。

2 \*\* 2 // 4

2 \*\* 3 // 8

指数运算符可以与等号结合，形成一个新的赋值运算符（`\*\*=`）。

```javascript

let a = 1.5;

a \*\*= 2;

// 等同于 a = a \* a;

let b = 4;

b \*\*= 3;

// 等同于 b = b \* b \* b;

```

\* javascript中的所有数字都被保存成64位浮点数，这决定了整数的精确程度只能到53个二进制。

# 《函数的扩展》

### 1.函数参数的默认值

#### 基本用法：

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);

}

log('Hello') // Hello World

log('Hello', 'China') // Hello China

log('Hello', '') // Hello

优点：

\* 除了简洁，ES6 的写法还有两个好处：

\* 首先，阅读代码的人，可以**\*\*立刻意识\*\***到哪些参数是可以省略的，不用查看函数体或文档；

\* 其次，有利于将来的**\*\*代码优化\*\***，即使未来的版本在对外接口中，彻底拿掉这个参数，也不会导致以前的代码无法运行。

### 2.reset

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

### 3.严格模式

**（1）使用**

* 第一种是设定全局性的严格模式，这是合法的。

'use strict'; //加上这一行代码就可以了

function doSomething(a, b = a) {

// code

* 第二种是把函数包在一个无参数的立即执行函数里面。

function doSomething(a, b) {

'use strict'; //加上这一行代码就可以了

}

\*\*\*注意：\*\*

`ES6做了一点修改，规定只要函数参数使用了默认值，解构赋值，或者扩展运算符，那么函数内部就不能设置严格模式了，否则会报错`。

**（2）name**

函数的`name`属性，返回该函数的函数名。

function foo() {}

foo.name // "foo"

\*注意的是：`ES6 对这个属性的行为做出了一些修改。如果将一个匿名函数赋值给一个变量，ES5 的name属性，会返回空字符串，而 ES6 的name属性会返回实际的函数名。`

### 4.箭头函数

#### 基本用法：

**\* ES6 允许使用“箭头”（`=>`）定义函数**。

**var f = v => v;**

**// 等同于**

**var 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。**

var f = () => 5;

// 等同于

var f = function () { return 5 };

var sum = (num1, num2) => num1 + num2;

// 等同于

var sum = function(num1, num2) {

return num1 + num2;

};

**\* 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**。

var sum = (num1, num2) => { return num1 + num2; }

**\* 如果箭头函数只有一行语句，且不需要返回值，可以采用下面的写法，就不用写大括号了**。

var sum=0;

let nn=()=>{

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

sum+=1;

}

alert(sum);

}

\* 箭头函数可以与\*\*变量解构\*\*结合使用。

const full = ({ first, last }) => first + ' ' + last;

// 等同于

function full(person) {

return person.first + ' ' + person.last;

}

**\* 箭头函数的一个用处是简化回调函数。**

// 正常函数写法

var result = values.sort(function (a, b) {

return a - b;

});

// 箭头函数写法

var result = values.sort((a, b) => a - b);

```

#### \*\*\*特点：

箭头函数有几个使用注意点。

\*（1）函数体内的`this`对象，就是**定义时**所在的对象，而不是**使用时**所在的对象。

（2）`**不可以当作构造函数**`。也就是说，不可以使用`new`命令，否则会抛出一个错误。

（3）**不可以使用`arguments`对象**，该对象在函数体内不存在。如果要用，可以用 rest 参数代替。

（4）**不可以使用`yield`命令**。因此箭头函数不能用作 Generator 函数。

#### \*\*\*不适用场合

由于箭头函数使得`this`从“动态”变成“静态”，下面两个场合不应该使用箭头函数。

\* 第一个场合是定义对象的方法，且该方法内部包括`this`。

```javascript

const cat = {

lives: 9,

jumps: () => {

this.lives--;

}

}

```

\* 第二个场合是需要动态`this`的时候，也不应使用箭头函数。

var button = document.getElementById('press');

button.addEventListener('click', () => {

this.classList.toggle('on');

```

# 《数组的扩展》

### 1.含义

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

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

// 1 2 3 4 5

### 2.扩展运算符的应用：

#### （1）复制数组\*\*

**const a1 = [1, 2];**

**// 写法一**

**const a2 = [...a1];**

**// 写法二**

**const [...a2] = a1;**

#### （2）合并数组\*\*

扩展运算符提供了数组合并的新写法。

const arr1 = ['a', 'b'];

const arr2 = ['c'];

const arr3 = ['d', 'e'];

// ES5 的合并数组

arr1.concat(arr2, arr3);

// [ 'a', 'b', 'c', 'd', 'e' ]

**// ES6 的合并数组**

**[...arr1, ...arr2, ...arr3]**

**// [ 'a', 'b', 'c', 'd', 'e' ]**

#### （3）与解构赋值结合

扩展运算符可以与解构赋值结合起来，用于`**生成数组**`。

[a, ...rest] = list

const [first, ...rest] = [1, 2, 3, 4, 5];

first // 1

rest // [2, 3, 4, 5]

#### （4）字符串转数组

扩展运算符还可以将`字符串转为真正的数组。`

[...'hello']

// [ "h", "e", "l", "l", "o" ]

### 3.Array.from()

* `Array.from`方法用于将\*\***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）。
* `Array.from`还可以\*\***接受第二个参数**\*\*，作用类似于数组的`map`方法，用来对每个元素进行处理，将处理后的值放入返回的数组。

### 4.Array.of()

`Array.of`方法用于将一组\*\***值，转换为数组**\*\*。

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

Array.of(3) // [3]

Array.of(3).length // 1

Array.of() // []

```

`Array.of`总是返回参数值组成的数组。如果没有参数，就返回一个空数组。

### 5.copyWithin()

Array.prototype.copyWithin(target, start = 0, end = this.length)

**例：**

[1, 2, 3, 4, 5].copyWithin(0, 3)

// [4, 5, 3, 4, 5]

它接受三个参数。

* target（必需）：从该位置开始替换数据。如果为负值，表示倒数。
* start（可选）：从该位置开始读取数据，默认为 0。如果为负值，表示从末尾开始计算。
* end（可选）：到该位置前停止读取数据，默认等于数组长度。如果为负值，表示从末尾开始计算。

这三个参数都应该是数值，如果不是，会自动转为数值。

### 6.find() 和 findIndex()

#### 区别：

* **find返回的是具体的值**
* **findindex返回的是符合条件的数组索引值，如果所有成员都不符合条件，则返回`-1`**

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

// -5

### 7.fill()填充数组

**\* `fill`方法使用给定值，填充一个数组。**t

['a', 'b', 'c'].fill(7) // [7, 7, 7]

new Array(3).fill(7) // [7, 7, 7]

\* `fill`方法还可以接受第二个和第三个参数,用于指定填充的\*\*起始位置\*\*和\*\*结束位置\*\*。(不包含结束位置)

['a', 'b', 'c'].fill(7, 1, 2) // ['a', 7, 'c']

上面代码表示，`fill`方法从 1 号位开始，向原数组填充 7，到 2 号位之前结束。

\* 注意，如果`**填充的类型为对象`，那么被赋值的是同一个内存地址的对象，而不是深拷贝对象。**

let arr = new Array(3).fill({name: "Mike"});

arr[0].name = "Ben";

arr

// [{name: "Ben"}, {name: "Ben"}, {name: "Ben"}]

### 8.entries()，keys() 和 values()

`entries()`，`keys()`和`values()`——用于遍历数组。它们都返回一个\*\*遍历器对象\*\*（详见《Iterator》一章），可以用`for...of`循环进行遍历，唯一的区别是:

#### 区**别:**

* **`keys()`是对键名的遍历、**
* **`values()`是对键值的遍历，**
* **`entries()`是对键值对的遍历。**

### 9. includes()

`Array.prototype.includes`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

### 10.flat()拉平

ES6拉平：

* **\*多维数组转一维**

function test(arr){

var newarr=arr.flat(infinity)

return newarr

}

\* `Array.prototype.flat()`用于将嵌套的数组“拉平”，变成一维的数组。该方法返回一个新数组，对原数据没有影响。

[1, 2, [3, 4]].flat() // [1, 2, 3, 4]

* **`flat()`默认只会“拉平”一层，**如果想要“拉平”多层的嵌套数组，可以将`flat()`方法的参数写成一个整数，表示想要拉平的层数，**\*\*默认为1\*\*。**

[1, 2, [3, [4, 5]]].flat()

// [1, 2, 3, [4, 5]]

[1, 2, [3, [4, 5]]].flat(2)

// [1, 2, 3, 4, 5]

* 如果不管有多少层嵌套，都要转成一维数组，可以用**`Infinity`**关键字作为参数。

[1, [2, [3]]].flat(Infinity) // [1, 2, 3]

* 如果原数组有空位，`flat()`方法会跳过空位。

[1, 2, , 4, 5].flat() // [1, 2, 4, 5]

#### flatMap()

\* `flatMap()`方法对原数组的每个成员执行一个函数（相当于执行`Array.prototype.map()`），然后对返回值组成的数组执行`flat()`方法。该方法返回一个新数组，不改变原数组。

// 相当于 [[2, 4], [3, 6], [4, 8]].flat()

[2, 3, 4].flatMap((x) => [x, x \* 2])

// [2, 4, 3, 6, 4, 8]

```

\* `flatMap()`只能展开一层数组。

// 相当于 [[[2]], [[4]], [[6]], [[8]]].flat()

[1, 2, 3, 4].flatMap(x => [[x \* 2]])

// [[2], [4], [6], [8]]

\*\*注意：属性名表达式与简洁表示法，不能同时使用，会报错。

### 11.数组的空位

数组的空位指，数组的某一个位置没有任何值。比如，`Array`构造函数返回的数组都是空位。

Array(3) // [, , ,]

```

\* ES5 对空位的处理，已经很不一致了，大多数情况下会忽略空位。

- `forEach()`, `filter()`, `reduce()`, `every()` 和`some()`都会跳过空位。

- `map()`会跳过空位，但会保留这个值

- `join()`和`toString()`会将空位视为`undefined`，而`undefined`和`null`会被处理成空字符串。

**\* ES6 则是明确将空位转为`undefined`**。

# 《对象的扩展》

### 1.属性简洁表示法：

\* 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。

const foo = 'bar';

const baz = {foo};

baz // {foo: "bar"}

// 等同于

const baz = {foo: foo};

上面代码中，变量`foo`直接写在大括号里面。这时，属性名就是变量名, 属性值就是变量值。下面是另一个例子。

function f(x, y) {

return {x, y};

}

// 等同于

function f(x, y) {

return {x: x, y: y};

}

f(1, 2) // Object {x: 1, y: 2}

### 1.2方法简洁表示法：

const o = {

method() {

return "Hello!";

}

};

// 等同于

const o = {

method: function() {

return "Hello!";

}

};

### 2.属性名表达式

JavaScript 定义对象的属性，有两种方法。

// 方法一

obj.foo = true;

// 方法二

obj['a' + 'bc'] = 123;

**\* 注意，属性名表达式与简洁表示法，不能同时使用，会报错。**

// 报错

const foo = 'bar';

const bar = 'abc';

const baz = { [foo] };

// 正确

const foo = 'bar';

const baz = { [foo]: 'abc'};

```

### 3.可枚举性和遍历

**\*可枚举性**:

* 对象的每个属性都有一个描述对象，用来控制该属性的行为。
* 描述对象的`enumerable`属性，称为“可枚举性”，如果该属性为`false`，就表示某些操作会忽略当前属性。

\* `Object.getOwnPropertyDescriptor`方法可以获取该属性的描述对象。

let obj = { foo: 123 };

Object.getOwnPropertyDescriptor(obj, 'foo')

// {

// value: 123,

// writable: true,

// enumerable: true, 可枚举性

// configurable: true

// }

### \*\*\*哪些操作会忽略可枚举性：

* `for...in`循环：只遍历对象自身的和继承的可枚举的属性。
* `Object.keys()`：返回对象自身的所有可枚举的属性的键名。
* `JSON.stringify()`：只串行化对象自身的可枚举的属性。
* `Object.assign()`： 忽略`enumerable`为`false`的属性，只拷贝对象自身的可枚举的属性。

### 4.属性的遍历

#### （1）for...in

`for...in`循环遍历对象自身的和继承的可枚举属性（不含 Symbol 属性）。

#### （2）Object.keys(obj)

Object.keys`返回一个数组，包括对象自身的（不含继承的）所有可枚举属性（不含 Symbol 属性）的键名。

#### （3）Object.getOwnPropertyNames(obj)

`Object.getOwnPropertyNames`返回一个数组，包含对象自身的所有属性（不含 Symbol 属性，但是包括不可枚举属性）的键名。

4）Object.getOwnPropertySymbols(obj)

`Object.getOwnPropertySymbols`返回一个数组，包含对象自身的所有 Symbol 属性的键名。

#### （5）Reflect.ownKeys(obj)\*\*

`Reflect.ownKeys`返回一个数组，包含对象自身的所有键名，不管键名是 Symbol 或字符串，也不管是否可枚举。

### 5.super 关键字

* 我们知道，`this`关键字总是指向函数所在的当前对象，
* ES6 又新增了另一个类似的关键字**`super`**，指向**当前对象的原型对象**
* \*\*注意:\*\*`super`关键字表示原型对象时，**只能用在\*\*对象的方法\*\*之中**，用在其他地方都会报错。
* 目前，只有对象方法的\*\***简写法**\*\*可以让 JavaScript 引擎确认，定义的是对象的方法。

const proto = {

foo: 'hello'

};

const obj = {

foo: 'world',

find() {

return super.foo;

}

};

Object.setPrototypeOf(obj, proto);//把proto变成obj的原型对象

obj.find() // "hello"

### 6.对象的扩展运算符

#### 作用：

* 对象的解构赋值，用来接收剩余参数
* 实现拷贝对象
* 合并对象

# 《对象的新增方法》\*

### \*1.Object.is()比较

#### 与严格全等的区别：

\* 它用来比较两个值是否严格相等，与严格比较运算符（===）的行为基本一致。

\* 不同之处只有两个：**一是`+0`不等于`-0`，二是`NaN`等于自身。**

+0 === -0 //true

NaN === NaN // false

Object.is(+0, -0) // false

Object.is(NaN, NaN) // true

### 2.Object.assign() 合并

基本用法

`Object.assign`方法用于对象的合并，将源对象（source）的所有可枚举属性，复制到目标对象（target）。

const target = { a: 1 };

const source1 = { b: 2 };

const source2 = { c: 3 };

Object.assign(target, source1, source2);

target // {a:1, b:2, c:3}

**\*\*注意\*\*：**

* `Object.assign`方法的\*\*第一个参数是目标对象\*\*，后面的参数都是源对象。
* 如果目标对象与源对象有\*\***同名属性**\*\*，或多个源对象有同名属性，则后面的属性会\*\***覆盖**\*\*前面的属性。
* `Object.assign`拷贝的属性是有\*\***限制**\*\*的，只拷贝源对象的\*\***自身属性**\*\*（不拷贝继承属性），也不拷贝不可枚举的属性（`enumerable: false`）。

Object.assign({b: 'c'},

Object.defineProperty({}, 'invisible', {

enumerable: false,

value: 'hello'

})

)

// { b: 'c' }

```

#### #注意点

（1）浅拷贝\*\*

`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。

const obj1 = {a: {b: 1}};

const obj2 = Object.assign({}, obj1);

obj1.a.b = 2;

obj2.a.b // 2

（2）同名属性的替换\*\*（同上）

（3）数组的处理\*\*

`Object.assign`可以用来处理数组，但是会把`数组`视为`对象`。

Object.assign([1, 2, 3], [4, 5])

// [4, 5, 3]

（4）取值函数的处理\*\*

`Object.assign`只能进行值的复制，如果要复制的值是一个取值函数，那么将`求值后再复制`。

const source = {

get foo() { return 1 }

};

const target = {};

Object.assign(target, source)

// { foo: 1 }

#### # 常见用途

`Object.assign`方法有很多用处。

#### （1）为对象添加属性\*\*

class Point {

constructor(x, y) {

Object.assign(this, {x, y});

}

}

上面方法通过`Object.assign`方法，将`x`属性和`y`属性添加到`Point`类的对象实例。

#### （2）为对象添加方法\*\*

Object.assign(SomeClass.prototype, {

someMethod(arg1, arg2) {

···

},

anotherMethod() {

···

}

});

// 等同于下面的写法

SomeClass.prototype.someMethod = function (arg1, arg2) {

···

};

SomeClass.prototype.anotherMethod = function () {

···

};

上面代码使用了对象属性的简洁表示法，直接将两个函数放在大括号中，再使用`assign`方法添加到`SomeClass.prototype`之中。

#### （3）克隆对象\*\*

function clone(origin) {

return Object.assign({}, origin);

}

上面代码将原始对象拷贝到一个空对象，就得到了原始对象的克隆。

不过，采用这种方法克隆，只能克隆原始对象自身的值，不能克隆它继承的值。如果想要保持继承链，可以采用下面的代码。

function clone(origin) {

let originProto = Object.getPrototypeOf(origin);

return Object.assign(Object.create(originProto), origin);

}

#### （4）合并多个对象\*\*

将多个对象合并到某个对象。

const merge =

(target, ...sources) => Object.assign(target, ...sources);

如果希望合并后返回一个新对象，可以改写上面函数，对一个空对象合并。

const merge =

(...sources) => Object.assign({}, ...sources);

#### （5）为属性指定默认值\*\*

const DEFAULTS = {

logLevel: 0,

outputFormat: 'html'

};

function processContent(options) {

options = Object.assign({}, DEFAULTS, options);

console.log(options);

// ...

}

### 3.Object.getOwnPropertyDescriptors()

ES5 的`Object.getOwnPropertyDescriptor()`方法会返回某个对象属性的描述对象

### 4.

#### --proto-- 读取或设置

--proto--`属性（前后各两个下划线），用来读取或设置当前对象的`prototype`对象

#### Object.setPrototypeOf()

`Object.setPrototypeOf`方法的作用与`\_\_proto\_\_`相同，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

function setPrototypeOf(obj, proto) {

obj.\_\_proto\_\_ = proto;

return obj;

}

Object.getPrototypeOf()

该方法与`Object.setPrototypeOf`方法配套，用于读取一个对象的原型对象。

如果参数不是对象，会被自动转为对象。

如果参数是`undefined`或`null`，它们无法转为对象，所以会报错。

### \*5.

#### Object.keys()

ES5 引入了`Object.keys`方法，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。

var obj = { foo: 'bar', baz: 42 };

Object.keys(obj)

// ["foo", "baz"]

#### Object.values()

`Object.values`方法返回一个数组，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。

const obj = { foo: 'bar', baz: 42 };

Object.values(obj)

// ["bar", 42]

#### Object.entries()

`Object.entries()`方法返回一个数组，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。

const obj = { foo: 'bar', baz: 42 };

Object.entries(obj)

// [ ["foo", "bar"], ["baz", 42] ]

### 6.Object.fromEntries()

Object.fromEntries()`方法是**`Object.entries()`的逆操作**，用于将一个键值对数组转为对象。

Object.fromEntries([

['foo', 'bar'],

['baz', 42]

])

// { foo: "bar", baz: 42 }

该方法的主要目的，是将键值对的数据结构还原为对象，因此特别适合将 Map 结构转为对象。

# 《Symbol》

### 1.概述：

* 一种新的原始数据类型`Symbol`，表示**独一无二**的值。
* 它是 JavaScript 语言的第七种数据类型，前六种是：`undefined`、`null`、布尔值（Boolean）、字符串（String）、数值（Number）、对象（Object）。
* 凡是属性名属于 Symbol 类型，就都是独一无二的，可以保证不会与其他属性名产生冲突。

### 2.注意：

* `Symbol`函数前不能使用`new`命令，否则会报错。Symbol 值不是对象，所以不能添加属性。基本上，它是一种类似于字符串的数据类型。
* `Symbol`函数的参数只是表示对当前 Symbol 值的描述，因此相同参数的`Symbol`函数的返回值是不相等的。
* Symbol 值可以显式转为字符串。Symbol 值也可以转为布尔值，但是`不能转为数值`。

### 3.用法：

* `Symbol`函数可以接受一个字符串作为参数，表示对 Symbol 实例的描述，主要是为了在控制台显示，或者转为字符串时，比较容易区分。

let s1 = Symbol('foo');

let s2 = Symbol('bar');

s1 // Symbol(foo)

s2 // Symbol(bar)

s1.toString() // "Symbol(foo)"

s2.toString() // "Symbol(bar)"

* 如果 Symbol 的参数是一个对象，就会调用该对象的`toString`方法，将其转为字符串，然后才生成一个 Symbol 值。

const obj = {

toString() {

return 'abc';

}

};

const sym = Symbol(obj);

sym // Symbol(abc)

### 4.Symbol.description (2019新增)

提供了一个实例属性`description`，直接返回 Symbol 的描述。

const sym = Symbol('foo');

sym.description // "foo"

### 5.作为属性名的 Symbol

由于每一个 Symbol 值都是不相等的，这意味着 Symbol 值可以作为标识符，用于对象的属性名，就能保证不会出现同名的属性。这对于一个对象由多个模块构成的情况非常有用，能防止某一个键被不小心改写或覆盖。

### ### 作用：

**（1）用于对象的属性名，就能保证不会出现同名的属性\*\***

let mySymbol = Symbol();

// 第一种写法

let a = {};

a[mySymbol] = 'Hello!';

// 第二种写法

let a = {

[mySymbol]: 'Hello!'

};

// 以上写法都得到同样结果

a[mySymbol] // "Hello!"

**注意:**

* Symbol 值作为对象属性名时，不能用点运算符。
* 在对象的内部，使用 Symbol 值定义属性时，Symbol 值必须放在方括号之中。

let s = Symbol();

let obj = {

[s]: function (arg) { ... }

};

obj[s](123);

上面代码中，如果`s`不放在方括号中，该属性的键名就是字符串`s`，而不是`s`所代表的那个 Symbol 值。

**（2）消除魔术字符串\*\***

魔术字符串指的是，在代码之中多次出现、与代码形成强耦合的某一个具体的字符串或者数值。风格良好的代码，应该尽量消除魔术字符串，改由含义清晰的变量代替。

\* 常用的消除魔术字符串的方法，就是把它写成一个变量。

const shapeType = {

triangle: 'Triangle'

};

function getArea(shape, options) {

let area = 0;

switch (shape) {

case shapeType.triangle: //魔术字符串

area = .5 \* options.width \* options.height;

break;

}

return area;

}

getArea(shapeType.triangle, { width: 100, height: 100 });

```

上面代码中，我们把`Triangle`写成`shapeType`对象的`triangle`属性，这样就消除了强耦合。

`shapeType.triangle`等于哪个值\*\*并不重要\*\*，只要确保不会跟其他`shapeType`属性的值冲突即可。因此，这里就很适合改用 Symbol 值。

```javascript

const shapeType = {

triangle: Symbol()

};

### 6.属性名的遍历

\* Symbol 作为属性名，遍历对象的时候，该属性不会出现在`for...in`、`for...of`循环中，也不会被`Object.keys()`、`Object.getOwnPropertyNames()`、`JSON.stringify()`返回。

\* `Object.getOwnPropertySymbols()`方法，可以获取指定对象的所有 Symbol 属性名。该方法返回一个数组，成员是当前对象的所有用作属性名的 Symbol 值。

const obj = {};

let a = Symbol('a');

let b = Symbol('b');

obj[a] = 'Hello';

obj[b] = 'World';

const objectSymbols = Object.getOwnPropertySymbols(obj);

objectSymbols

// [Symbol(a), Symbol(b)]

# 《Set 和 Map 数据结构》

## 《Set》

#### 1.概述与基本用法

* **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.**

#### 2.数组去重的方法。

var ss=new Set([1,2,2,4,4,5])

console.log(Array.form(ss))

#### 3.字符串去重

[...new Set('ababbc')].join('')

// "abc"

\* 它类似于精确相等运算符（`===`），主要的区别是向 Set 加入值时认为`NaN`等于自身，而精确相等运算符认为`NaN`不等于自身。

let set = new Set();

let a = NaN;

let b = NaN;

set.add(a);

set.add(b);

set // Set {NaN}

## 《WeakSet》

与set区别：

\* WeakSet 的成员只能是\*\*对象\*\*，而不能是其他类型的值。

\* WeakSet 中的对象都是\*\*弱引用\*\*，即\*\*垃圾回收机制不考虑\*\* WeakSet 对该对象的引用。也就是说，如果其他对象都不再引用该对象，那么垃圾回收机制会自动回收该对象所占用的内存，不考虑该对象还存在于 WeakSet 之中。

## 《Map》

#### 1.概述

与普通对象区别：

* JavaScript 的对象（Object），本质上是键值对的集合（Hash 结构），但是传统上只能用字符串当作键。这给它的使用带来了很大的限制。
* ES6 提供了 `**Map 数据结构**`。它`**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。

#### 2.创建方法：

\* 作为构造函数，Map 也可以接受一个数组作为参数。该数组的成员是一个个表示\*\*键值对的数组\*\*。

function test(){}

const map = new Map([

['name', '张三'],

['title', 'Author']，

[test,"abc"]

]);

map.size // 2

map.has('name') // true

map.get('name') // "张三"

map.has('title') // true

map.get('title') // "Author"

map.get(test)//"abc"

```

#### 3.属性和操作方法

**（1）size 属性\*\***

**（2）Map.prototype.set(key, value)\*\***

const m = new Map();

m.set('edition', 6) // 键是字符串

m.set(262, 'standard') // 键是数值

m.set(undefined, 'nah') // 键是 undefined

`set`方法返回的是当前的`Map`对象，因此可以采用链式写法。

let map = new Map()

.set(1, 'a')

.set(2, 'b')

.set(3, 'c');

**（3）Map.prototype.get(key)\*\***

`get`方法读取`key`对应的键值，如果找不到`key`，返回`undefined`。

const m = new Map();

const hello = function() {console.log('hello');};

m.set(hello, 'Hello ES6!') // 键是函数

m.get(hello) // Hello ES6!

**（4）Map.prototype.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。

```javascript

const m = new Map();

m.set('edition', 6);

m.set(262, 'standard');

m.set(undefined, 'nah');

m.has('edition') // true

m.has('years') // false

m.has(262) // true

m.has(undefined) // true

**（5）Map.prototype.delete(key)\*\***

`delete`方法删除某个键，返回`true`。如果删除失败，返回`false`。

const m = new Map();

m.set(undefined, 'nah');

m.has(undefined) // true

m.delete(undefined)

m.has(undefined) // false

**（6）Map.prototype.clear()\*\***

`clear`方法清除所有成员，没有返回值。

let map = new Map();

map.set('foo', true);

map.set('bar', false);

map.size // 2

map.clear()

map.size // 0

#### 4.遍历方法

Map 结构原生提供三个遍历器生成函数和一个遍历方法。

- `Map.prototype.keys()`：返回键名的遍历器。

- `Map.prototype.values()`：返回键值的遍历器。

- `Map.prototype.entries()`：返回所有成员的遍历器。

- `Map.prototype.forEach()`：遍历 Map 的所有成员。

# # 创建一个没有原型的对象(面试题)

**let obj = Object.create(null);**

## 《WeakMap》

`WeakMap`与`Map`的区别：

\* `WeakMap`只接受对象作为键名（`null`除外），不接受其他类型的值作为键名。

\* `WeakMap`的键名所指向的对象，不计入垃圾回收机制。

# 《同步异步》

### \*\*<JS运行机制>与<事件轮询>：

1. 所有的任务都在主线程上执行，形成一个执行栈。
2. 主线程之外还有一个任务队列，所有异步任务都放在其中。
3. \*\*执行栈中的所有同步任务执行完毕，系统就会读取“任务队列”里已经有运行结果的异步任务放入执行栈中开始执行。\*\*
4. 主线程不断重复上面的第三步。
5. \*\*只要主线程空了，就会去读取任务队列，这就是Js的运行机制。这个过程会不断重复叫做事件轮询。\*\*

### \*\*处理异步的方法：

1. async
2. AJAX
3. 事件监听
4. 发布订阅
5. promise对象
6. callback

function f2(){

console.log("3")

}

function f1(callback){

console.log("1")

setTimeout(function(){

callback();

},0);

console.log("2")

}

f1(f2)

//运行结果：1 2 3

### \*任务队列：

#### \*\*微任务先执行，然后再执行宏任务\*\*

#### 微任务：promise.then(),cash(),finally(),all(),race()等方法

#### 宏任务：setTimeout,setinterval,I/O,绑定的事件

# 《Promise 对象》\*\*

### \*1.含义：

\* 所谓`Promise，简单说就是一个**封装异步操作的容器**。

\* 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 AP**I，各种异步操作都可以用同样的方法进行处理。

### \*2.特点：

`Promise`对象有以下两个特点。

（1）对象的状态不受外界影响。`Promise`对象代表一个异步操作，有三种状态：`pending`（进行中）、`fulfilled`（已成功）和`rejected`（已失败）。只有异步操作的结果，可以决定当前是哪一种状态，任何其他操作都无法改变这个状态。这也是`Promise`这个名字的由来，它的英语意思就是“承诺”，表示其他手段无法改变。

（2）一旦状态改变，就不会再变，任何时候都可以得到这个结果。`Promise`对象的状态改变，只有两种可能：从`pending`变为`fulfilled`和从`pending`变为`rejected`。只要这两种情况发生，状态就凝固了，不会再变了，会一直保持这个结果，这时就称为 resolved（已定型）。如果改变已经发生了，你再对`Promise`对象添加回调函数，也会立即得到这个结果。这与事件（Event）完全不同，事件的特点是，如果你错过了它，再去监听，是得不到结果的。

### 3.缺点：

* 首先，无法取消`Promise`，一旦新建它就会立即执行，\*\*无法中途取消\*\*。
* 其次，如果不设置回调函数，`Promise`内部抛出的错误，不会反应到外部。
* 第三，当处于`pending`状态时，无法得知目前进展到哪一个阶段（刚刚开始还是即将完成）。

### 4.基本用法

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

**const promise = new Promise(function(resolve, reject) {**

**// ... some code**

**if (/\* 异步操作成功 \*/){**

**resolve(value);**

**} else {**

**reject(error);**

**}**

**});**

#### `#\*<异步加载图片>

function loadImageAsync(url) {

return new Promise(function(resolve, reject) {

const image = new Image();

image.onload = function() {

resolve(image); //回调函数，返回成功的事件

};

image.onerror = function() {

reject(new Error('Could not load image at ' + url)); //回调函数，返回失败的事件

};

image.src = url;

});

}

#### # <`Promise`对象实现的 Ajax>

function newtitle(){

const promise=new Promise(funxtion(resolve,reject){

let xmlhttp=new XMLHttpREquest();

xmlhttp.open("post","http://www.qhdlink-student.top/student/banner.php");

xmlhttp.setRequestHeader("content-type","application/x-www-form-urlencoded");

xmlhttp.onreadystate=function(){

if(this.readyState!=4){

return;

}

if(this.status==200){

resolve(response)

}else{

reject(new error("数据获取失败"))

}

}

xmlhttp.send("username=张柯&userpwd=123456&userclass=56&type=4")

})

return promise()

}

## \*5.方法：

#### （1）.then()\*\*

1. `then`方法是定义在原型对象`Promise.prototype`上的。
2. then`方法返回的是一个新的`Promise`实例
3. **\*\*`它的作用是为 Promise 实例添加状态改变时的回调函数。\*\***
4. `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。

data.then(function(value){

//。。。请求成功的数据

},function(error){

//。。。请求失败的数据

})

**链式写法**：

`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。

getJSON("/posts.json").then(function(json) {

return json.post;

}).then(function(post) {

// ...

});

上面的代码使用`then`方法，依次指定了两个回调函数。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。

#### （2）.catch()\*\*

**\* 是then方法第二个参数的别名，用于指定发生错误时的回调函数。\***

\* 与then的区别：\*

\*then方法指定的回调函数，如果运行中抛出错误，也会被`catch`方法捕获。

getJSON('/posts.json').then(function(posts) {

// ...

}).catch(function(error) {

// 处理 getJSON 和 前一个回调函数运行时发生的错误

console.log('发生错误！', error);

});

#### （3）.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**

**\* 服务器使用 Promise 处理请求，然后使用`finally`方法关掉服务器**。

server.listen(port)

.then(function () {

// ...

})

.finally(server.stop);

`

\* `finally`方法里面的操作，应该是与状态无关的，不依赖于 Promise 的执行结果。

#### （4）.all()\*\*

#作用：

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

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

\*\*p的状态由`p1`、`p2`、`p3`决定，分成两种情况:\*\*

（1）只有`p1`、`p2`、`p3`的状态都变成`fulfilled`，`p`的状态才会变成`fulfilled`，此时`p1`、`p2`、`p3`的返回值组成一个\*\*数组\*\*，传递给`p`的回调函数。

（2）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。

#### （5）.race()\*\*

**`Promise.race()`方法同样是将多个 Promise 实例，包装成一个新的 Promise 实例。**

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

\*与all区别：

上面代码中，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。

# 《Iterator》

### 1.概念及作用：

* 它是一种**遍历器接口**，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）。
* 通过**Symbol.iterator**来判断是否有遍历器接口。
* Iterator主要使用**next()方法**和一种新的遍历命令**`for...of**`循环

#### \*原理：

**`创建一个指针对象，指向当前数据结构的起始位置。+next方法`**

#### #next方法：

* 每一次调用`next`方法，返回一个包含`value`和`done`两个属性的对象。
* \* 其中，`value`属性是当前成员的值，`done`属性是一个布尔值，表示遍历是否结束(true) 。

下面是一个模拟`next`方法返回值的例子。

var it = makeIterator(['a', 'b']);

it.next() // { value: "a", done: false }

it.next() // { value: "b", done: false }

it.next() // { value: undefined, done: true }

### 2.默认 Iterator 接口

\*原生具备Iterator 接口的数据结构如下：\*

* **Array**
* **Map**
* **Set**
* **String**
* **TypedArray（类数组）**
* **函数的 arguments 对象**
* **NodeList 对象**

#### \*为对象添加 Iterator 接口的例子：\*\*

let obj = {

data: [ 'hello', 'world' ],

[Symbol.iterator]() {

const self = this;

let index = 0;

return {

next() {

if (index < self.data.length) {

return {

value: self.data[index++],

done: false

};

} else {

return { value: undefined, done: true };

}

}

};

}

};

#### \*\*类似数组的对象调用数组的`Symbol.iterator`方法：\*\*

let iterable = {

0: 'a',

1: 'b',

2: 'c',

length: 3,

[Symbol.iterator]: Array.prototype[Symbol.iterator] //直接引用数组的 Iterator 接口

};

for (let item of iterable) {

console.log(item); // 'a', 'b', 'c'

}

```

### 3.

### for...of与for...in区别：

**`for...in`循环读取键名，`for...of`循环读取键值。**

#### #for...in()缺点：

* + 数组的键名是数字，但是`for...in`循环是以**字符串作为键名** “0”、“1”、“2”等等。
  + `for...in`循环不仅遍历数字键名，还会遍历手动\*\***添加的其他键**\*\*，甚至包括原型链上的键。
  + 某些情况下，`for...in`循环会以\*\***任意顺序**\*\*遍历键名。

# 《Generator 》

### 1.概念：

**Generator 函数是一个状态机，封装了多个内部状态。**

### \*2.特点：

* 一是，`function`关键字与函数名之间有一个**星号**；
* 二是，函数体内部使用`**yield`**表达式，定义不同的内部状态

### 3.yield 表达式

* \*`yield`表达式**本身没有返回值**，或者说总是返回**`undefined`**。
* **`next`方法可以带一个参数**，该参数就会被当作上一个`yield`表达式的**返回值**。

#### # next()方法：

遍历器对象的`next`方法的运行逻辑如下。

（1）遇到**`yield`表达式，就暂停执行后面的操作**，并将紧跟在`yield`后面的那个表达式的值，作为**返回的对象的`value`属性值**。

（2）下一次调用`next`方法时，再继续往下执行，直到遇到下一个`yield`表达式。

（3）如果没有再遇到新的`yield`表达式，就一直运行到函数结束，直到return语句为止，并将**`return`语句**后面的表达式的值，作为返回的对象的`value`属性值。

（4）如果该函数没有`return`语句，则返回的对象的`value`属性值为`undefined`。

#### \*\*需要注意的是:

**yield`表达式后面的表达式，只有当调用`next`方法、内部指针指向该语句时才会执行**，因此等于为 JavaScript 提供了手动的“惰性求值”（Lazy Evaluation）的语法功能。

## 4.yield与return区别：

* 区别在于每次遇到`yield`，函数暂停执行，下一次再从该位置\*\***继续**\*\*向后执行，而`return`语句**不具备位置记忆的功能**。
* 一个函数里面，只能\*\***执行一次**（或者说一个）`return`语句，但是可以\*\***执行多次**\*\*（或者说多个）`yield`表达式。正常函数只能返回一个值，因为只能执行一次`return`；Generator 函数可以返回一系列的值，因为可以有任意多个`yield`。

## 5.\*\*\*应用：

自定义函数用for..of实现遍历

function\* objectEntries() {

let Keys = Object.keys(this);

for (let Key of Keys) {

yield [Key, this[Key]];

}

}

let jane = { first: 'Jane', last: 'Doe' };

jane[Symbol.iterator] = objectEntries; //将 Generator 函数加到对象的Symbol.iterator属性上面。给对象添加接口

for (let [key, value] of jane) {

console.log(`${key}: ${value}`);

}

// first: Jane

// last: Doe

## 6.作用：

**（1）异步操作的同步化表达**

**（2）控制流管理**

**（3）部署 Iterator 接口**

**（4）作为数据结构**

## 7.generator封装AJAX

function\* main(){

var result=yield request("http://www.qhdlink-student.top/student/coach.php")

console.log(result);

}

functiom request(url){

makeAjaxCall(url,function(response){

it.next(response);

});

}

var it=main()

it.next()//调用实现数据处理

function makeAjaxCall(url,callback){

$.ajax({

url:url,

type:"post",

datatype:"json",

data:{

"username":"张柯",

"userpwd":"123456",

"userclass":"56",

"type":4

},

success:function(data){

callback(data) //与callback结合使用

}

})

}

# 《async 函数》

### \*1.特点：

`async`函数对 Generator 函数的优化：

（1）内置执行器。

（2）更好的语义。

（3）更广的适用性。

（4）返回值是 Promise。

### 2.await 命令

防止出错的方法，也是将其放在`**try...catch`**代码块之中。

如果有多个`await`命令，可以统一放在`try...catch`结构中。

async function main() {

try {

const val1 = await firstStep();

const val2 = await secondStep(val1);

const val3 = await thirdStep(val1, val2);

console.log('Final: ', val3);

}

catch (err) {

console.error(err);

}

}

使用注意点

\* 第一点，前面已经说过，`await`命令后面的`Promise`对象，运行结果可能是`rejected`，所以最好把`await`命令放在`try...catch`代码块中。

\* 第二点，多个`await`命令后面的异步操作，如果不存在继发关系，最好让它们同时触发

\* 第三点，`await`命令只能用在`async`函数之中，如果用在普通函数，就会报错。

\* 第四点，async 函数可以保留运行堆栈。

# 《Class》

### 1.\*类的创建：

class person{

constructor(name,age){

this.name=name;

this.age=age; //属性

}

sayName(x){console.log(this.name+x)} //方法

}

var obj=new person("zk","20")

obj.sayName("hahaha")

### 2.\*this指向：

class Logger {

constructor() {

this.printName = this.printName.bind(this); //改变this指向，改为logger

}

printName(name = 'there') {

this.print(`Hello ${name}`); //模板字符串

}

print(text) {

console.log(text);

}

}

const logger = new Logger();

const { printName } = logger; //对象的解构赋值

printName(); //This当前指向为window会报错，调用上边方法改变指向

### 3.\*Class 的继承

**1.简介：**

Class 可以通过extends关键字实现继承。

class father {  
    constructor(name,age){  
        this.name=name;  
        this.age=age;    
    }  
   sayName(){console.log(this.name)}  
}  
​  
class children extends father {  
   constructor(name,age,color) {  
   super(name, age);    // 调用父类的constructor(name, age)  
   this.color = color;                                                                                                            
}  
   x(){alert(this.color)}  
}   
​  
var obj=new children("zk","20",123)  
obj.sayName();

**注意:**

super关键字，它在这里表示父类的构造函数，用来新建父类的this对象。

必须先调用super方法，然后再用子类的构造函数修改this。

**3.super 关键字**

* 第一种情况，super作为函数调用时，代表父类的构造函数。ES6 要求，子类的构造函数必须执行一次super函数.

作为函数时，super()只能用在子类的构造函数之中，用在其他地方就会报错。

* 第二种情况，super作为对象时，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。

### 4.类的 prototype 属性和--proto--属性 ：

（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。

# 《Module》

### 1.export 命令

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。

### 2.加载规则

浏览器加载 ES6 模块，也使用<script>标签，但是要加入type="module"属性。

<script type="module" src="./foo.js"></script>

### 3.defer与async的区别是：

* defer是“渲染完再执行”，async是“下载完就执行”。