03-第三章 ECMAScript6 函数及数组拓展

ECMAscript 6

> # 一、函数的拓展

一、函数参数的默认值

```
function fn(x='hello',y='world'){
        console.log(x,y);
    }
   fn();//hello world
```

参数`变量是默认声明`的，所以不能用`let`或`const`再次声明。

```
function fn(x='hello',y='world'){
        let x = 20;//SyntaxError
        console.log(x,y);
    }
   fn();
```

二、函数参数的位置

如果传入`undefined`，将触发该参数等于默认值

```
function fn(x=1,y){
    console.log(x,y)
}
fn();//1 undefined
fn(2);//2 undefined
fn(,3);//SyntaxError
fn(undefined,1);
```

三、函数的length属性

函数的length属性，返回函数参数个数。指定了默认值后，length属性将失真。

```
var r = (function(){}).length;
console.log(r);//0

var r = (function(a){}).length;
console.log(r);//1

var r = (function(a,b){}).length;
console.log(r);//2
```

指定了默认值后，`length属性将失真`。

```
var r = (function(a,b=10){}).length;
console.log(r);//2
```

且 指定默认值后面的参数不算了length

```
var r = (function(a,c=10,b){}).length;
console.log(r);//1
```

四、默认值参数作用域

函数一旦设置了形参，函数进行声明初始化时，参数会形成一个单独的作用域（context），在不设置参数时，是不会出现的。

```
    function fn(x,y){
        console.log(y,x);
    }
    fn();//undefined undefined
```

没有传递值，相对于声明x,y而没有设置值，所以默认赋值`undefined`;

```
   var x = 20;
    function fn(x,y){
        console.log(y,x);
    }
    fn();//undefined undefined
```

函数参数是局部变量与外部变量没有关系

```
var x = 1;
    function fn(y = x){//此时x为变量
        console.log(x,y);
    }
    fn();//1 1
```

没有传递参数，所以默认值生效了,变量x不是形参，而是变量且值为1 。所以x,y都为1

```
var x = 1;
    function fn(y= x){
        console.log(y,x);
    }
    fn(20);//20 1
```

y传递了20，y 的值就是20，默认值没有生效，打印的x，局部的变量而是全局的,所以写x是1

```
var x = 1;
    function fn(x,y = x){
        console.log(x,y);
    }
    fn(20);//20 20
```

声明了局部变量x,且传递变量x为20, y赋值的也是局部了x,所以x,y都是20

```
var x = 1;
    function fn(y = x,x){
        console.log(y,x);
    }
    fn(20);//20 undefined
```

声明了y和x，且给y传递了20, x默认赋值undefined

作用域进阶

```
let foo = 'outer';
function fn( x = function f(){ return foo;}  ){
        let foo = 'inner';
        console.log( x() );//outer
    }
    fn();
```

函数参数形成的`单独作用域`里面，并没有定义变量foo，所以foo指向外层的全局变量foo，因此输出outer。
单独的作用域?大家可能会想单独的作用域, 那么为什么能够访问x呢？应该访问不到才对啊, 我们可以称这个作用域为`参数作用域`，这个作用域介于全局作用域及函数内部作用域中间

```
var x = 'outer';
    function fn(x,y = function(){ x = 2 } ){
        x = 3;
        y();
        console.log(x);
    }
    fn();//2
    console.log(x);//outer
```

五、rest参数(剩余参数)

> 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 fn(...val){
        console.log(val);//[2, 5, 10]
    }
   fn(2,5,10);

```

```
 function fn(arr,...val){
        console.log(arr);//2
        console.log(val);//[5, 10]
    };
    fn(2,5,10);
```

上面`...val`指代所有的参数

```
function fn(arr,two,three,...val){

        console.log(arr,two,three,val);//2 5 10 []
    };
    fn(2,5,10);

```

…val没有剩余参数则为[]空数组

`arguments`对象不是数组，而是一个类似数组的对象。所以为了使用数组的方法，必须使用`Array.prototype.slice.call`先将其转为数组。`rest`参数就不存在这个问题，它就是一个真正的数组，数组特有的方法都可以使用

`注意！！！`，rest 参数之后不能再有其他参数（即只能是最后一个参数），否则会报错。

```
function fn(arr,...val,last){//SyntaxError
        console.log(val);//[2, 5, 10]
    }
   fn(2,5,10);
```

六、箭头函数

ES6允许使用’`箭头`‘(`=>`)定义函数

- （1）

```
var f = v => v ;
  console.log( f(10) );//10

```

```
上面代码默认会返回v,等同于下面函数
```

```
var f = function(v){
       return v;
     }
  console.log( f(10) );//10
```

- （2）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 fn = (a,b)=> a+b;
     console.log( fn(10,20) );//30
```

如下会报错

```
    var fn = f(a,b)=> a+b;//报错

```

上面等同于

```
function fn(a,b){
        return a+b;
    }
     console.log( fn(10,20) );
```

- （3）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,且使用`return`;
  *

```
var fn = (a,b,c)=>{
        var res = 0;
        res+=a;
        res+=b;
        res+=c;
        return res;
    };

    console.log( fn(10,20,30) );//60
```

- （4）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

```
 var fn = (x,y)=> {name:x,age:y};//SyntaxError

/--------------------

var fn = (x,y)=> ({name:x,age:y});//SyntaxError
console.log( fn('二狗',20) );//{name: "二狗", age: 20}


```

- （5）箭头函数可以与变量解构结合使用

```
var fn = ({x,y})=>x+y;
     console.log( fn({x:10,y:20}) );30
```

- （6）箭头函数与 `rest` 参数结合使用

```
var fn = (...val)=>val[0];
     console.log( fn(10,20) );//10
```

使用注意事项

（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。

```
function fn(){
        setTimeout(()=>{
            console.log(this);//{name: "二狗"}
        },100)

    }
    fn.call({name:'二狗'});
```

```
var obj = {
        show:{
            abc:()=>{ console.log(this); }//window
        }
    }
    obj.show.abc();
```

函数的`length`属性，不包括 rest 参数。

```
var r = function fn(arr,...val){
        console.log(val);
    }.length;
    console.log(r);//1
```

七、函数name属性

函数的`name`属性，返回该函数的函数名。

```
function fn(){}
    console.log( fn.name );//fn
```

这个属性早就被浏览器广泛支持，但是直到 ES6，才将其写入了标准。

八、严格模式

‘`use strict`‘; 定义javascript严格模式 ，要定义在作用域的顶部

```
 a = 20;
console.log(a );//20
```

非严格模式，变量可以不定义直接使用

```
    'use strict';
    a = 20;//ReferenceError
    console.log(a );
```

定义了严格模式，就会报错

```
'use strict';
  function fn(x,y){

    console.log(x,y);/
  }
  fn(10,20);
```

> 从 ES5 开始，函数内部可以设定为 `严格模式` 。

```
function fn(){
    'use strict';
      a = 20;//ReferenceError
      console.log(a );

  }
  fn();
```

ES2016 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显式设定为严格模式，否则会报错。

```
function fn(x = 10){
    'use strict';//SyntaxError

  }
  fn();

//---------------------------------
function fn({x,y}){
    'use strict';//SyntaxError

  }
  fn({x:10,y:20});
//---------------------------------


function fn(...val){
    'use strict';//SyntaxError
    console.log(val);
  }
  fn(10,20);
```

两种方法可以规避这种限制。第一种是设定全局性的严格模式，这是合法的。

```
'use strict';
  function fn(...val){

    console.log(val);//[10, 20]
  }
  fn(10,20);
```

一、数组的拓展

一、拓展运算符

> 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。
> rest是为数据加 `[]`
> spread是为数据去 `[]`

```
1.      console.log( ['a','b','c']);// ["a", "b", "c"]
2.      console.log( ...['a','b','c']);// a b c
3.      console.log( 1,...['a','b','c'],3);//1 "a" "b" "c" 3
4.
5.      console.log( document.querySelectorAll('li'));//[li, li, li,
    li]
6.       console.log( ...document.querySelectorAll('li'));//<li> <li>
    <li>
```

该运算符主要用于函数调用

```
1.  var arr = [];
2.      function fn(...val){
3.          arr.push(val);
4.      }
5.      fn(1,2,3);
6.      console.log( arr );//[Array[3]]
7.
```

```
1.  var arr = [];
2.      function fn(...val){
3.          arr.push(...val);
4.      }
5.      fn(1,2,3);
6.      console.log( arr );//[1, 2, 3]
```

二、拓展运算符运用

- （1）复制数组 扩展运算符提供了复制数组的简便写法。

```
    const a1 = [1,2];
    const a2 = [...a1];
    a2.push('ok');
    console.log(a2);//[1, 2, "ok"]
    console.log(a1);//[1, 2]
```

- （2）合并数组

```
var arr1 = [1,2,3];
   var arr2 = ['a',...arr1,'b'];
   console.log( arr2);//["a", 1, 2, 3, "b"]
```

- （3）与解构赋值结合

```
const [first,...rest] = [1,2,3,4,5];
   console.log( first );//1
   console.log( rest );//[2,3,4,5]
```

- （4）与字符串结合

```
var str = 'hello';
   console.log( [...str] );//["h", "e", "l", "l", "o"]
```

- （5）实现了 `Iterator` 接口的对象
  任何 Iterator 接口的对象，都可以用扩展运算符转为真正的数组。

```
var aLi = document.getElementsByTagName('li');
  console.log( [...aLi] );
```

document.getElementsByTagName方法返回的是一个nodeList对象。它不是数组，而是一个类似数组的对象。这时，扩展运算符可以将其转为真正的数组，原因就在于NodeList对象实现了 Iterator 。

对于那些没有部署 Iterator 接口的类似数组的对象，扩展运算符就无法将其转为真正的数组。

使用Array.from方法也可以转为真正的数组

三、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）。

```
// arguments对象
function foo() {
  const args = [...arguments];
}

// NodeList对象
[...document.querySelectorAll('div')]
```

四、Array.of() 方法用于将一组值，转换为数组。

```
var r = Array.of(3,1,10);
    console.log(r);//[3, 1, 10]
    var r = Array.of(3);
    console.log(r);//[3]
    var r = Array.of();
    console.log(r);//[]
```

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() // []
Array(3) // [, , ,]
Array(3, 11, 8) // [3, 11, 8]
```

五、copyWithin() 复制填充

> 数组实例的copyWithin方法，在当前数组内部，将指定位置的成员复制到其他位置（会覆盖原有成员），然后返回当前数组。也就是说，使用这个方法，会修改当前数组。

语法

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

`target`（必需）：从该位置开始替换数据。
`start`（可选）：从该位置开始读取数据，默认为 0。如果为负值，表示倒数。
`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，表示倒数。

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


/-------------------

var r = [1,2,3,4,5].copyWithin(0,3);
    console.log(r);//[4, 5, 3, 4, 5]

/----------------

var r = [1,2,3,4,5].copyWithin(0,-2,-1);
    console.log(r);//[4, 2, 3, 4, 5]
```

六、find() 与 findIndex() 查找数组值及下标

> 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，返回 第一个值为`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` 。

---

arr.`find(` function(){} `)`;

`find( function( value ,index,arry){ })` 的回调函数可以接受三个参数，依次为
`value` 当前的值、
`index` 当前的位置
`arry` 原数组

```
let r = ['a',1,'b','c'].find((n)=>n == 'c');
    console.log(r);
```

```
 let r = [1,5,10,15].find((val,index,arr)=>{
        return val > 9;

    });
    console.log( r );//10
```

---

arr.`findIndex(` function(){} `)`;

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

```
 let r = [1,5,10,15].findIndex((val,index,arr)=>{
        return val > 9;

    });
    console.log( r );//2
```

七、fill() 填充数组

`fill(` content,start,end `)` 填充一个数组

`content` 填充内容
`start` 用于指定填充的起始位置
`end` 截止位置，不包含此位

```
    var r = ['a','b','c'].fill(7);
     console.log( r );//[7,7,7]
```

```
var r = ['a','b','c'].fill(7,1,2);
     console.log( r );//["a", 7, "c"]
```

八、 keys() values() entries() 遍历数组

ES6 提供三个新的方法——`entries()`，`keys()`和`values()`——用于遍历数组。它们都返回一个`遍历器对象`

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、`values()`是对`键值`的遍历，`entries()`是对`键值对`的遍历

```
let arr = ['a', 'b', 'c'];
    let key = arr.keys();
    console.log(key);//ArrayIterator {}
```

不能直接使用

```
for(key of ['a','b'].keys() ){
        console.log( key );// 0 1
    }
//------------------------

for(key of ['a','b'].values() ){
        console.log( key );// 报错
    }
//-------------------------

 for(key of ['a','b'].entries() ){
        console.log( key );
        //[0, "a"]
         //[1, "b"]
    }


```

八、includes() 是否包含

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

```
includes(value,start);
```

```
1.      var r = [1,2,3].includes(5);
2.       console.log(r);//false
3.       var r = [1,2,3].includes(2);
4.       console.log(r);//true
5.       var r = [1,2,NaN].includes(NaN);
6.       console.log(r);//true
```

该方法的第二个参数表示搜索的起始位置，默认为`0`。
如果第二个参数为负数，则表示倒数的位置，如果这时它大于数组长度（比如第二个参数为-4，但数组长度为3），则会重置为从`0`开始。

```
1. var r = ['a','b','c'].includes('b',-4);
2.       console.log(r);//true
```

`includes()`填补了`indexOf`方法不能判断`NaN`缺陷

```
1. var r = [NaN].indexOf(NaN);
2. console.log(r);//-1
3.
4. //-------------------------
5.
6. var r = [NaN].includes(NaN);
7. console.log(r);//true
```