


函数

es5与es6中函数特性



主要内容

- 关于函数形参默认值
- 不定参数
- 箭头函数
- 其他新特性

函数形参默认值

- 函数形参与实参
- 在es5与es6中形参默认值的设置方法
- es6设置形参默认值的多种方法
- 默认参数值的影响

函数形参 vs 实参

```
1  function fn (url, timeout, cb) {  
2      timeout = timeout || 2000  
3      cb = cb || function () {}  
4  
5      //....  
6  }
```

```
1  function fn (url, timeout, cb) {  
2      timeout = typeof timeout !== 'undefined' ? timeout : 2000  
3      cb = typeof cb !== 'undefined' ? cb : function () {}  
4  
5      //....  
6  }
```

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

```
1  function log (x, y = 'world') {  
2    console.log(x, y)  
3  }  
4  
5  log('Hello')  
6  log('Hello', 'China')  
7  log('Hello', '') // 传入合法的falsy值的时候，不会出现es5中的问题
```

PROBLEMS

OUTPUT

DEBUG CONSOLE

TERMINAL

```
bogon:es6 jane$ node default2.js  
Hello world  
Hello China  
Hello  
bogon:es6 jane$ █
```

默认形参的值可以有以下几种形式：

- 1: 原始值
- 2: 非原始值(原始值经过运算操作得到的值)
- 3: 先定义的参数
- 4: 先传入的参数的基础上做运算求得的值

```
1  function getValue () {  
2    return 5  
3  }  
4  
5  function log (x, y = getValue()) {  
6    console.log(x + y)  
7  }  
8  
9  log(1)    // 6
```

PROBLEMS

OUTPUT

DEBUG CONSOLE

TERMINAL

bogon:es6 jane\$ node default3.js

6

bogon:es6 jane\$

```
1  let value = 5
2
3  function getValue () {
4    return value ++
5  }
6
7  function log (x, y = getValue()) {
8    console.log(x + y)
9  }
10
11 log(1, 1) // 并不会调用getValue()
12 log(1)    // 6
13 log(1)    // 7
14 log(1)    // 8
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

bogon:es6 jane\$ node default3.js

2

6

7

8

bogon:es6 jane\$

用先定义参数传入的值作为后定义的参数的默认值

```
1  function log (x, y = x) {  
2    console.log(x + y)  
3  }  
4  
5  log(1, 1)  // 2
```

PROBLEMS

OUTPUT

DEBUG CONSOLE

TERMINAL

```
bogon:es6 jane$ node default4.js
```

```
2
```

```
bogon:es6 jane$
```

用先定义参数传入的值计算后的结果作为后定义的参数的默认值

```
33  function getValue (value) {  
34  |   return value + 5  
35  | }  
36  
37  function add (first, second = getValue(first)){  
38  |   return first + second  
39  | }  
40  
41  console.log(add(1, 1))    // 2  
42  console.log(add(1))      // 7  
43  
44  
45
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

bogon:es6 jane\$ node default

2

7

bogon:es6 jane\$

默认参数的位置问题

- 1: 通常设置了默认值的参数应该是函数的尾参数，方便最简传值
- 2: 参数的默认值必须是undefined才能触发，传null之类的会取null值，非尾参数的不传会报错

```
1  function f (x, y = 5, z) {  
2    console.log([x, y, z])  
3  }  
4  
5  f()           // [ undefined, 5, undefined ]  
6  f(1)          // [ 1, 5, undefined ]  
7  f(1, ,2)      // 报错  
8  f(1, undefined, 4) // [1, 5, 4]  
9  f(1, null, 4)   // [1, null, 4]
```

结论： 位于中间的设置了默认值的参数在调用的时候，必须显示传undefined才能触发默认值
尾参数可以传undefined，也可以直接不传任何值，都可以触发默认值

设置了默认形参值之后，对函数的一些影响

- 1: 函数的length 属性
- 2: 函数的arguments 对象

fn.length = 函数参数中第一个默认参数之前的所有命名参数个数

```
44  function a (x, y) {  
45  
46  }  
47  function b (x, y = 0) {  
48  
49  }  
50  function c (x, y = 0, z, m) {  
51  
52  }  
53  
54  console.log(a.length) // 2  
55  console.log(b.length) // 1  
56  console.log(c.length) // 1  
57
```

PROBLEMS

OUTPUT

DEBUG CONSOLE

TERMINAL

```
bogon:es6 jane$ node default.js
```

```
2
```

```
1
```

```
1
```

```
bogon:es6 jane$
```

默认参数值对arguments对象的影响

es5非严格模式下

```
1  function changeArgs (first, second) {  
2      // 'use strict'  
3      console.log(arguments.length)  
4  
5      console.log(first == arguments[0])  
6      console.log(second == arguments[1])  
7  
8      first = 'c'  
9      second = 'd'  
10  
11     console.log(first == arguments[0])  
12     console.log(second == arguments[1])  
13 }  
14  
15 changeArgs('a', 'b')
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
bogon:es6 jane$ node arguments  
2  
true  
true  
true  
true  
bogon:es6 jane$
```

es5严格模式下

```
1  function changeArgs (first, second) {  
2      'use strict'  
3      console.log(arguments.length)  
4  
5      console.log(first == arguments[0])  
6      console.log(second == arguments[1])  
7  
8      first = 'c'  
9      second = 'd'  
10  
11     console.log(first == arguments[0])  
12     console.log(second == arguments[1])  
13 }  
14  
15 changeArgs('a', 'b')
```

PROBLEMS

OUTPUT

DEBUG CONSOLE

TERMINAL

bogon:es6 jane\$ node arguments

2

true

true

false

false

bogon:es6 jane\$

```
1  function changeArgs (first, second = 'm') {
2    // 'use strict'
3    console.log(arguments.length)
4
5    console.log(first == arguments[0])
6    console.log(second == arguments[1])
7
8    first = 'c'
9    second = 'd'
10
11    console.log(first == arguments[0])
12    console.log(second == arguments[1])
13  }
14
15  changeArgs('a')
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
bogon:es6 jane$ node arguments
1
true
false
false
false
bogon:es6 jane$
```

es6中一个函数使用了默认参数值，无论是否显式指定了严格模式，arguments的行为都将与es5中的严格模式下保持一致

不定参数

- 不定参数出现的意义
- 不定参数的限制
- 不定参数对arguments的影响

不定参数

```
1  function fn () {  
2      let args = Array.prototype.slice.call(arguments, 0)  
3      let args = [].slice.call(arguments, 0)  
4  
5      //....  
6  }
```

```
1  function fn (...args) {  
2    console.log(Object.prototype.toString.call(args) == '[object Array]')  
3    console.log(args)  
4  }  
5  
6  fn(1, 2, 3)  
7  fn(1, 'a', 6, 9)
```

PROBLEMS

OUTPUT

DEBUG CONSOLE

TERMINAL

```
bogon:es6 jane$ node args  
true  
[ 1, 2, 3 ]  
true  
[ 1, 'a', 6, 9 ]  
bogon:es6 jane$
```

- 1: 直接是一个真正的数组
- 2: 处理的参数名固定
- 3: 简单明了

- 1: 每个函数只能声明一个不定参数，并且必须放在所有参数的末尾
- 2: 不定参数不能用在对象字面量的setter之中，setter参数有且只能有一个

不定参数的位置必须在最后，
函数的length，不会计算不定参数

length = 不定参数之前的所有命名参数的个数

tips:

不定参数与展开运算符比较相似

```
1  let arr = [1, 25, 9, 34]
2  let a = Math.max(...arr)
3  let b = Math.max.apply(Math, arr)
4
5  let c = Math.max(...arr, 6)
6  let d = Math.max(...arr, 89)
7
8  console.log(a, b)
9  console.log(c, d)
10
11 let crr = [1, 2, 3, 5]
12 let drr = [...crr]
13 drr.push(6)
14
15 console.log(crr)
16 console.log(drr)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
bogon:es6 jane$ node args
34 34
34 89
[ 1, 2, 3, 5 ]
[ 1, 2, 3, 5, 6 ]
bogon:es6 jane$
```

- 1: 大多数使用apply的地方，都可以使用展开运算符
- 2: 解决数组引用问题

箭头函数

- 箭头函数出现的意义
- 与传统函数的不同
- 语法
- 应用场景

this的困扰

函数中this的指向是js程序中常见的错误来源

箭头函数

- 1: 在箭头函数声明的时候就指定了this的指向，避免了麻烦
- 2: 便于优化（尾调用优化）

- 箭头函数中的this、arguments由外围最近的一层非箭头函数决定
- 没有原型，不能通过new关键字调用
- 不可以改变this的绑定
- 不支持arguments对象，对实参的操作通过命名参数和不定参数实现
- 不支持重复的命名

语法

参数 => 函数体

参数：

无参数 () => {}

1个参数 arg => {}

多个参数 (arg1, arg2, arg3...) => {}

函数体：

1个表达式 () => a + b

多个表达式 () => { 函数体, 除了不能用arguments之外, 与传统函数一样 }

Tips: 返回对象字面量的话, 需要用圆括号包起来

(id, name) => ({id, name})

创建iife

```
1  let person = ((name) => {  
2    return {  
3      getName: function () {  
4        return name  
5      }  
6    }  
7  })('Jhon')  
8  
9  console.log(person.getName()) // 'Jhon'
```

箭头函数与数组

```
let arr = [1, 3, 9, 2, 6]
```

```
// 常规写法
```

```
let order = arr.sort(function () {  
    return a - b  
})
```

```
// 箭头函数写法
```

```
let result = arr.sort(() => a - b)
```

简化了回调函数，代码更加简洁

尾调用优化

```
1  // 阶乘函数
2  function factorial (n) {
3      if(n === 1){
4          return 1
5      }
6
7      return n * factorial(n - 1)
8  }
9
10 // 递归函数 斐波那契数列
11 function fabnacci (n) {
12     if(n <= 1){
13         return 1
14     }
15
16     return fabnacci(n - 1) + fabnacci(n - 2)
17 }
```

尾调用优化的条件

- 尾调用不访问当前栈的变量
- 在函数内部，尾调用是最后一条数据
- 尾调用的结果作为函数值返回

尾调用： 某个函数的最后一步是调用另一个函数

利用尾调用优化改写阶乘函数

```
1  function factorial (n, total) {  
2      if(n === 1){  
3          return total  
4      }  
5      return factorial(n-1, n * total)  
6  }
```

```
8  function factorial (n, total = 1) {  
9      if(n === 1){  
10         return total  
11     }  
12     return factorial(n-1, n * total)  
13 }  
14
```

利用尾调用改造递归函数

```
7  
8  function fibonacchi (n, ac1 = 1, ac2 = 1) {  
9      if(n <= 1){  
10         return ac2  
11     }  
12  
13     return fibonacchi(n-1, ac2, ac1 + ac2)  
14 }
```


其他新特性

- 增强的Function构造函数
- Name属性
- 明确函数的多重用途
- 元属性
- 块级函数

定义函数的三种方式

- 1: 函数声明
- 2: 函数表达式
- 3: Function构造函数

使用Function构造函数

```
var sum = new Function(arg1, arg2, ..... 函数体字符串)
```

增强点：

- 1: 支持设定默认参数
- 2: 支持使用不定参数

```
1  let add = new Function('first', 'second=first', 'return first + second')
2
3  console.log(add(1, 1))           // 2
4  console.log(add(1))              // 2
5
6  let pickFirst = new Function('...args', 'return args[0]')
7  console.log(pickFirst(1, 2, 3))  // 1
8  console.log(pickFirst(2, 3))     // 2
```

PROBLEMS

OUTPUT

DEBUG CONSOLE

TERMINAL

bogon:es6 jane\$ node quote

2

2

1

2

bogon:es6 jane\$

在es6之前，Firxfox、Chrome、Safari、Opera给函数定义了一个非标准的name属性，针对函数声明和函数表达式有不同的值。

使用函数声明定义的函数的name属性值，就是在function关键字之后的标识；
使用函数表达式定义的函数的name属性值，是空字符串。

```
function fn () {  
  
}  
  
var fn2 = function () {  
  
}  
  
console.log(fn.name)  
console.log(fn2.name)
```

在es6中，所有的形式的函数都可以获取到name值，
函数声明是function关键字之后的标识符
函数表达式是被赋值为该匿名函数的变量的名称

```
1  function fn () {  
2  
3  }  
4  
5  var fn2 = function () {  
6  
7  }  
8  
9  console.log(fn.name)  
10 console.log(fn2.name)  
11
```

PROBLEMS

OUTPUT

DEBUG CONSOLE

TERMINAL

```
bogon:es6 jane$ node functionname.js  
fn  
fn2  
bogon:es6 jane$
```

特殊情况

- 1: 命名函数赋值给变量，name值为原函数name
- 2: getter、setter函数会有前缀
- 3: bind绑定的函数会有bound前缀
- 4: Function构造函数创建的函数name值是anonymous

函数：
普通函数
构造函数

函数内部新增了两个方法： **[[Call]]** 和 **[[Construct]]**。

是否使用new 关键字调用，区分不同的函数。

Tips： 不是所有的函数都有**[[Construct]]**，比如箭头函数就没有

es5中instanceof运算符用来判断，实例与构造函数之间的关系。

```
1  function Person (name) {
2    if(this instanceof Person){
3      this.name = name
4    }else{
5      throw new Error('必须通过new调用Person')
6    }
7  }
8
9  let person = new Person('Jhon')
10 let aperson = Person('Jhon')
11
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
bogon:es6 jane$ node metaProperty.js
/Users/jane/Desktop/learning/es6/metaProperty.js:5
  throw new Error('必须通过new调用Person')
  ^
```

```
Error: 必须通过new调用Person
    at Person (/Users/jane/Desktop/learning/es6/metaProperty.js:5:11)
    at Object.<anonymous> (/Users/jane/Desktop/learning/es6/metaProperty.js:10:15)
    at Module._compile (module.js:570:32)
    at Object.Module._extensions..js (module.js:579:10)
    at Module.load (module.js:487:32)
    at tryModuleLoad (module.js:446:12)
    at Function.Module._load (module.js:438:3)
    at Module.runMain (module.js:604:10)
    at run (bootstrap_node.js:389:7)
    at startup (bootstrap_node.js:149:9)
bogon:es6 jane$
```


但是这种方法不可靠

```
1  function Person (name) {  
2    if(this instanceof Person){  
3      this.name = name  
4    }else{  
5      throw new Error('必须通过new调用Person')  
6    }  
7  }  
8  
9  let person = new Person('Jhon')  
10 let aperson = Person.call(person, 'Jhon') // 可以正常通过  
11
```

PROBLEMS

OUTPUT

DEBUG CONSOLE

TERMINAL

bogon:es6 jane\$ node metaProperty.js

bogon:es6 jane\$

es6引入了new.target元属性，提供非对象目标的补充信息（比如new）

```
1  function Person (name) {
2    if(typeof new.target !== 'undefined'){
3      this.name = name
4    }else{
5      throw new Error('必须通过new调用Person')
6    }
7  }
8
9  let person = new Person('Jhon')
10 let aperson = Person.call(person, 'Jhon') // 可以正常通过
11
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
bogon:es6 jane$ node metaProperty.js
/Users/jane/Desktop/learning/es6/metaProperty.js:5
  throw new Error('必须通过new调用Person')
  ^

Error: 必须通过new调用Person
    at new Person (/Users/jane/Desktop/learning/es6/metaProperty.js:5:11)
    at Object.<anonymous> (/Users/jane/Desktop/learning/es6/metaProperty.js:10:22)
    at Module._compile (module.js:570:32)
    at Object.Module._extensions..js (module.js:579:10)
    at Module.load (module.js:487:32)
    at tryModuleLoad (module.js:446:12)
    at Function.Module._load (module.js:438:3)
    at Module.runMain (module.js:604:10)
    at run (bootstrap_node.js:389:7)
    at startup (bootstrap_node.js:149:9)
bogon:es6 jane$
```

之前版本的js中，不同浏览器对于块级函数的支持不一致。

```
1  var condition = true
2  var sayHi
3  if (condition) {
4      function sayHi () {
5          console.log('hi')
6      }
7  } else {
8      function sayHi () {
9          console.log('Yo')
10     }
11 }
12
13 sayHi()
```

在es6中,支持了块级函数

```
1  'use strict'
2
3  var condition = true
4  if (condition) {
5      console.log(typeof sayHi) // function
6      function sayHi () {
7          console.log('hi')
8      }
9  } else {
10     console.log(typeof sayHi) // 使用let报错、用var输出undefined
11     let sayHi = function () {
12         console.log('Yo')
13     }
14 }
15
16 console.log('----->', typeof sayHi) // 只在块内有效
```

在非严格模式下，与严格模式下，稍有不同，声明提升位置会变成外围函数全局的顶部

```
1  var condition = true
2  if (condition) {
3      console.log(typeof sayHi) // function
4      function sayHi () {
5          console.log('hi')
6      }
7  } else {
8      console.log(typeof sayHi) // 使用let报错、用var输出undefined
9      let sayHi = function () {
10         console.log('Yo')
11     }
12 }
13
14 console.log('----->', typeof sayHi) // 非严格模式下，在块外也有效
```

PROBLEMS

OUTPUT

DEBUG CONSOLE

TERMINAL

```
bogon:es6 jane$ node blockFn.js
function
-----> function
bogon:es6 jane$
```