

# 函数

```
1 function 函数名(参数列表) {  
2     函数体;  
3     return 返回值;  
4 }  
5  
6 function add(x,y){  
7     return x+y;  
8 }  
9 console.log(add(3,5));
```

## 函数表达式

使用表达式来定义函数，表达式中的函数名可以省略，如果这个函数名不省略，也只能用在此函数内部。

```
1 // 匿名函数  
2 const add = function(x, y){  
3     return x + y;  
4 };  
5 console.log(add(4, 6));  
6  
7 // 有名字的函数表达式  
8 const sub = function fn(x, y){  
9     return x - y;  
10 };  
11 console.log(sub(5, 3));  
12 //console.log(fn(3, 2)); // fn只能用在函数内部  
13  
14 // 有名字的函数表达式  
15 const sum = function _sum(n) {  
16     if (n===1) return n;  
17     return n + _sum(--n); // _sum只能内部使用  
18 }  
19 console.log(sum(4));
```

## 函数、匿名函数、函数表达式的差异

函数和匿名函数，本质上都是一样的，都是函数对象，只不过函数有自己的标识符——函数名，匿名函数需要借助其它的标识符而已。

区别在于，函数会**声明提升**，函数表达式不会。function定义函数虽然可以提升，但也请先定义后使用。

```

1 console.log(add(4, 6));
2 // 匿名函数
3 function add (x, y){ // 声明提升
4     return x + y;
5 };
6
7 //console.log(sub(5, 3)); //sub未定义
8 // 有名字的函数表达式
9 const sub = function (x, y){
10     return x - y;
11 };
12 console.log(sub(5, 3));

```

## 高阶函数

高阶函数：函数作为参数或返回一个函数

### 完成一个计数器counter

注意嵌套函数和闭包

```

1 const counter = function (){
2     let c = 0;
3     return function(){
4         return ++c;
5     };
6 };
7
8 const c = counter()
9 console.log(c())
10 console.log(c())
11 console.log(c())

```

另附counter的生成器版本，仅供参考

```

1 const counter = (function * () {
2     let c = 1
3     while (true) {
4         yield c++
5     }
6 })()
7
8 console.log(counter.next())
9 console.log(counter.next())
10 console.log(counter.next())

```

## 练习

完成一个map函数：可以对某一个数组的元素进行某种处理

```

1  const map = function (fn, arr) {
2      let newArr = [];
3      for (let i in arr){
4          newArr[i] = fn(arr[i])
5      }
6      return newArr;
7  }
8
9  console.log(map(function(x){return x+1}, [1,2,3,4])); // 输出什么
10 console.log(map(function(x){return x++}, [1,2,3,4])); // 输出什么
11 console.log(map(function(x){return ++x}, [1,2,3,4])); // 输出什么
12 console.log(map(function(x){return x+=1}, [1,2,3,4])); // 输出什么

```

## 箭头函数

箭头函数就是匿名函数，它是一种更加精简的格式。

将上例中的你们函数更改为箭头函数

```

1  // 以下三行等价
2  console.log(map((x) => {return x * 2}, [1,2,3,4]));
3  console.log(map(x => {return x * 2}, [1,2,3,4]));
4  console.log(map(x => x * 2, [1,2,3,4]));

```

### 箭头函数参数

- 如果一个函数没有参数，使用()
- 如果只有一个参数，参数列表可以省略小括号()
- 多个参数不能省略小括号，且使用逗号间隔

### 箭头函数返回值

如果函数体部分有多行，就需要使用{}，如果有返回值使用return。

如果只有一行语句，可以同时**省略大括号和return**。

只要有return语句，就不能省略大括号。 `console.log(map([1,2,3,4], x => {return ++x}))`，有return必须有大括号。

如果只有一条非return语句，加上大括号，函数就成了无返回值了，例如

`console.log(map([1,2,3,4], x => {x*2}));`加上了大括号，它不等价于 `x => {return x*2}`。因此，记住 `x => x*2` 这种正确的形式就行了。

## 函数参数

### 普通参数

一个参数占一个位置，支持默认参数

```

1  const add = (x,y) => x + y
2  console.log(add(4, 5))
3
4  // 缺省值
5  const add1 = (x, y=5) => x + y
6  console.log(add1(4, 7))
7  console.log(add1(4))

```

那如果有这样一个函数

```
1 | const add2 = (x=6,y) => x+y
```

这可以吗？尝试使用一下

```
1 | console.log(add2())
2 | console.log(add2(1))
3 |
4 | console.log(add2(y=2,z=3)) // 可以吗？
```

上面add2的调用结果分别为

NaN、NaN、5

为什么？

- 1、JS中并没有Python中的关键字传参
- 2、JS只是做参数位置的对应
- 3、JS并不限制默认参数的位置

add2()相当于add(6, undefined)

add2(1)相当于add(1, undefined)

add2(y=2,z=3)相当于add2(2,3)，因为JS没有关键字传参，但是它的赋值表达式有值，y=2就是2，z=3就是3

建议，默认参数写到后面，这是一个好的习惯。

## 可变参数(rest parameters 剩余参数)

JS使用...表示可变参数（Python用\*收集多个参数）

```
1 | const sum = (...args) => {
2 |     let result = 0;
3 |     for (let x of args) {
4 |         result += x;
5 |     }
6 |     return result;
7 | };
8 |
9 | console.log(sum(3,6,9));
```

## arguments对象

函数的所有参数会被保存在一个arguments的键值对对象中。

```
1 | (function (p1, ...args) {
2 |     console.log(p1)
3 |     console.log(args)
4 |     console.log('-----')
5 |     console.log(arguments) // 对象
6 |     for (let x of arguments) // 该对象可以使用of
7 |         console.log(x);
8 | })('abc', 1,3,5)
```

ES6之前，arguments是唯一可变参数的实现。

ES6开始，不推荐，建议使用可变参数。为了兼容而保留。

注意，使用箭头函数，取到的arguments不是我们想要的，如下

```
1 ((x,...args) => {
2   console.log(args); // 数组
3   console.log(x);
4   console.log(arguments); // 不是传入的值
5 })(...[1,2,3,4]);
```

## 参数解构

和Python类似，Js提供了参数解构，依然使用了...符号来解构。

```
1 const add = (x, y) => {console.log(x,y);return x + y};
2 console.log(add(...[100,200]))
3 console.log(add(...[100,200,300,3,5,3]))
4 console.log(add(...[100]))
```

Js支持参数解构，不需要解构后的值个数和参数个数对应。

## 函数返回值

python 中可以使用 `return 1,2` 返回多值，本质上也是一个值，就是一个元组。Js中呢？

```
1 const add = (x, y) => {return x,y};
2 console.log(add(4,100)); // 返回什么？
```

表达式的值

类C的语言，都有一个概念——表达式的值

赋值表达式的值：等号右边的值。

逗号表达式的值：类C语言，都支持逗号表达式，逗号表达式的值，就是最后一个表达式的值。

```
1 a = (x = 5, y = 6, true);
2 console.log(a);
3
4 b = (123, true, z = 'test')
5 console.log(b)
6
7 function c() {
8   return x = 5, y = 6, true, 'ok';
9 }
10
11 console.log(c());
```

所以，JS的函数返回值依然是单值

---

## 作用域

---

```

1 // 函数中变量的作用域
2 function test(){
3     a = 100;
4     var b = 200;
5     let c = 300;
6 }
7 // 先要运行test函数
8 test()
9
10 console.log(a);
11 console.log(b); // 不可见
12 console.log(c); // 不可见

```

```

1 // 块作用域中变量
2 if (1){
3     a = 100;
4     var b = 200;
5     let c = 300;
6 }
7
8 console.log(a);
9 console.log(b);
10 console.log(c); // 不可见

```

function是函数的定义，是一个独立的作用域，其中定义的变量在函数外不可见。

var a = 100 可以提升声明，但不可以突破函数作用域。

a = 100 隐式声明不能提升声明，在“严格模式”下会出错，但是可以把变量隐式声明为全局变量。建议少用。

let a = 100 不能提升声明，而且不能突破任何的块作用域。推荐使用。

```

1 function show(i, arg) {
2     console.log(i, arg)
3 }
4
5 // 作用域测试
6 x = 500;
7 var j = 'jjjj';
8 var k = 'kkkk';
9
10 function fn(){
11     let z = 400;
12     {
13         var o = 100; // var 作用域当前上下文
14         show(1, x);
15         t = 'free'; // 此语句执行后，t作用域就是全局的，不推荐
16         let p = 200;
17     }
18     var y = 300;
19     show(2, z);
20     show(3, x);
21     show(4, o);
22     show(5, t);
23     //show(6, p); // 异常，let出不来上一个语句块
24     {
25         show(7, y);

```

```

26         show(8,o);
27         show(9,t);
28         {
29             show(10,o);
30             show(11,t);
31             show(12,z);
32         }
33     }
34
35     j = 'aaaa';
36     var k = 'bbbb';
37     show(20, j);
38     show(21, k);
39 }
40
41 // 先执行函数
42 fn()
43
44 show(22, j);
45 show(23, k);
46
47 //show(13,y); // 异常, y只能存在于定义的上下文中, 出不了函数
48 show(14,t); // 全局, 但是严格模式会抛异常
49
50 //show(15,o) // 看不到o, 异常原因同y
51
52 show(16,z); // 变量声明提升, var声明了z, 但是此时还没有赋值
53 var z = 10;
54
55 const m = 1
56 //m = 2 // 常量不可以重新赋值

```

严格模式: 使用"use strict";, 这条语句放到函数的首行, 或者js脚本首行

## 异常

### 抛出异常

Js的异常语法和java相同, 使用throw关键字抛出。

使用throw关键字可以抛出任意对象的异常

```

1  throw new Error('new error');
2  throw new ReferenceError('Ref Error');
3  throw 1;
4  throw 'not ok';
5  throw [1,2,3];
6  throw {'a':1};
7  throw () => {}; // 函数

```

### 捕获异常

try...catch 语句捕获异常。

try...catch...finally 语句捕获异常, finally保证最终一定执行。

注意这里的catch不支持类型, 也就是说至多一个catch语句。可以在catch的语句块内, 自行处理异常。

```
1  try {
2      //throw new Error('new error');
3      //throw new ReferenceError('Ref Error');
4      //throw 1;
5      //throw new Number(100);
6      // throw 'not ok';
7      // throw [1,2,3];
8      // throw {'a':1};
9      throw () => {}; // 函数
10 } catch (error) {
11     console.log(error);
12     console.log(typeof(error));
13     console.log(error.constructor.name);
14     console.log(error.message);
15 } finally {
16     console.log('===end===')
17 }
18
```