JS语法

函数

```
function 函数名(参数列表) {
    函数体;
    return 返回值;
}

function add(x,y){
    return x+y;
}
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。

```
// 匿名函数
const add = function(x, y){
    return x + y;
};
console.log(add(4, 6));

// 有名字的函数表达式
const sub = function fn(x, y){
    return x - y;
};
console.log(sub(5, 3));
//console.log(fn(3, 2)); // fn只能用在函数内部

// 有名字的函数表达式
const sum = function _sum(n) {
    if (n===1) return n;
    return n + _sum(--n) // _sum只能内部使用
}
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。

```
console.log(add(4, 6));
// 匿名函数
function add (x, y){ // 声明提升
    return x + y;
};

//console.log(sub(5, 3)); //sub未定义
// 有名字的函数表达式
const sub = function (x, y){
    return x - y;
};
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

```
const counter = function (){
    let c = 0;
    return function(){
        return ++c;
    };
};

const c = counter()
console.log(c())
console.log(c())
console.log(c())
```


完成一个map函数：可以对某一个数组的元素进行某种处理

```
const map = function(arr,fn) {
    let newarr = [];
    for (let i in arr){
        newarr[i] = fn(arr[i]);
    }
    return newarr
}
console.log(map([1,2,3,4], function(x){return ++x}));
```

另附counter的生成器版本，仅供参考

```
const counter = (function * () {
    let count = 1;
    while (true)
        yield count++;
})();

console.log(counter.next())
console.log(counter.next())
```

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

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

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

```
const add = (x,y) => x+y;
console.log(add(4,5));

// 缺省值
const add1 = (x,y=5) => x+y;
console.log(add1(4,6));
console.log(add1(4));
```

那如果有这样一个函数

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

这可以吗？尝试使用一下

```
console.log(add2());
console.log(add2(1));

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）

```
const sum = function (...args){
    let result = 0;
    for (let x in args){
        result += args[x];
    }
    return result;
};

console.log(sum(3,6,9))
```

arguments对象

函数的所有参数会被保存在一个arguments的键值对字典对象中。

```
(function (p1, ...args) {
    console.log(p1);
    console.log(args);
    console.log('----------------');
    console.log(arguments); // 对象
    for (let x of arguments)
        console.log(x);
})('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

```
((x,...args) => {
    console.log(args); // 数组
    console.log(x);
    console.log(arguments); // 不是传入的值
})(...[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。

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

```
const add = (x, y) => {return x,y};
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。

```
a = (x = 5, y = 6, true);
console.log(a); // true

b = (123, true, z = 'test')
console.log(b)

function c() {
    return x = 5, y = 6, true, 'ok';
}

console.log(c()); // ok
```

所以，JS的函数返回值依然是单值

作用域

```
// 函数中变量的作用域
function test(){
    a = 100;
    var b = 200;
    let c = 300;
}
// 先要运行test函数
test()

console.log(a);
console.log(b); // 不可见
console.log(c); // 不可见
```

```
// 块作用域中变量
if (1){
    a = 100;
    var b = 200;
    let c = 300;
}

console.log(a);
console.log(b);
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，也可以突破非函数的块作用域。
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。

```
function show(i, arg) {
    console.log(i, arg)
}

// 作用域测试
x = 500;
var j = 'jjjj';
var k = 'kkkk';

function fn(){
    let z = 400;
    {
        var o = 100; // var 作用域当前上下文
        show(1, x);
        t = 'free'; // 此语句执行后，t作用域就是全局的，不推荐
        let p = 200;
    }
    var y = 300;
```

```
    show(2,z);
    show(3,x);
    show(4,o);
    show(5,t);
    //show(6,p); // 异常，let出不来上一个语句块
    {
        show(7,y);
        show(8,o);
        show(9,t);
        {
            show(10,o);
            show(11,t);
            show(12,z);
        }
    }

    j = 'aaaa';
    var k = 'bbbb';
    show(20, j);
    show(21, k);
}

// 先执行函数
fn()

show(22, j);
show(23, k);

//show(13,y); // 异常，y只能存在于定义的上下文中，出不了函数
show(14,t); // 全局，但是严格模式会抛异常

//show(15,o) // 看不到o，异常原因同y

show(16,z); // 变量声明提升，var声明了z，但是此时还没有赋值
var z = 10;

const m = 1
//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

```
throw new Error('new error');
throw new ReferenceError('Ref Error');
throw 1;
throw 'not ok';
throw [1,2,3];
throw {'a':1};
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。

```
try {
    //throw new Error('new error');
    //throw new ReferenceError('Ref Error');
    //throw 1;
    //throw new Number(100);
    // throw 'not ok';
    // throw [1,2,3];
    // throw {'a':1};
    throw () => {}; // 函数
} catch (error) {
    console.log(error);
    console.log(typeof(error));
    console.log(error.constructor.name);
} finally {
    console.log('===end===')
}
```