# 函数

函数是一段可以反复调用的代码块。函数还能接受输入的参数，不同的参数会返回不同的值。

## 概述

### 函数的声明

JavaScript 有三种声明函数的方法。

**（1）function 命令**

`function`命令声明的代码区块，就是一个函数。`function`命令后面是函数名，函数名后面是一对圆括号，里面是传入函数的参数。函数体放在大括号里面。

```javascript
function print(s) {
  console.log(s);
}
```

上面的代码命名了一个`print`函数，以后使用`print()`这种形式，就可以调用相应的代码。这叫做函数的声明（Function Declaration）。

**（2）函数表达式**

除了用`function`命令声明函数，还可以采用变量赋值的写法。

```javascript
var print = function(s) {
  console.log(s);
};
```

这种写法将一个匿名函数赋值给变量。这时，这个匿名函数又称函数表达式（Function Expression），因为赋值语句的等号右侧只能放表达式。

采用函数表达式声明函数时，`function`命令后面不带有函数名。如果加上函数名，该函数名只在函数体内部有效，在函数体外部无效。

```javascript
var print = function x(){
  console.log(typeof x);
};

x
// ReferenceError: x is not defined
```

```
print()
// function
```

上面代码在函数表达式中，加入了函数名`x`。这个`x`只在函数体内部可用，指代函数表达式本身，其他地方都不可用。这种写法的用处有两个，一是可以在函数体内部调用自身，二是方便除错（除错工具显示函数调用栈时，将显示函数名，而不再显示这里是一个匿名函数）。因此，下面的形式声明函数也非常常见。

```javascript
var f = function f() {};
```

需要注意的是，函数的表达式需要在语句的结尾加上分号，表示语句结束。而函数的声明在结尾的大括号后面不用加分号。总的来说，这两种声明函数的方式，差别很细微，可以近似认为是等价的。

**（3）Function 构造函数**

第三种声明函数的方式是`Function`构造函数。

```javascript
var add = new Function(
  'x',
  'y',
  'return x + y'
);

// 等同于
function add(x, y) {
  return x + y;
}
```

上面代码中，`Function`构造函数接受三个参数，除了最后一个参数是`add`函数的“函数体”，其他参数都是`add`函数的参数。

你可以传递任意数量的参数给`Function`构造函数，只有最后一个参数会被当做函数体，如果只有一个参数，该参数就是函数体。

```javascript
var foo = new Function(
  'return "hello world";'
);

// 等同于
function foo() {
  return 'hello world';
```

}
```

`Function`构造函数可以不使用`new`命令，返回结果完全一样。

总的来说，这种声明函数的方式非常不直观，几乎无人使用。

### 函数的重复声明

如果同一个函数被多次声明，后面的声明就会覆盖前面的声明。

```javascript
function f() {
  console.log(1);
}
f() // 2

function f() {
  console.log(2);
}
f() // 2
```

上面代码中，后一次的函数声明覆盖了前面一次。而且，由于函数名的提升（参见下文），前一次声明在任何时候都是无效的，这一点要特别注意。

### 圆括号运算符，return 语句和递归

调用函数时，要使用圆括号运算符。圆括号之中，可以加入函数的参数。

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

add(1, 1) // 2
```

上面代码中，函数名后面紧跟一对圆括号，就会调用这个函数。

函数体内部的`return`语句，表示返回。JavaScript 引擎遇到`return`语句，就直接返回`return`后面的那个表达式的值，后面即使还有语句，也不会得到执行。也就是说，`return`语句所带的那个表达式，就是函数的返回值。`return`语句不是必需的，如果没有的话，该函数就不返回任何值，或者说返回`undefined`。

函数可以调用自身，这就是递归（recursion）。下面就是通过递归，计算斐波那契数列的代码。

```javascript
function fib(num) {

```
  if (num === 0) return 0;
  if (num === 1) return 1;
  return fib(num - 2) + fib(num - 1);
}

fib(6) // 8
```

上面代码中，`fib`函数内部又调用了`fib`，计算得到斐波那契数列的第6个元素是8。

### 第一等公民

JavaScript 语言将函数看作一种值，与其它值（数值、字符串、布尔值等等）地位相同。凡是可以使用值的地方，就能使用函数。比如，可以把函数赋值给变量和对象的属性，也可以当作参数传入其他函数，或者作为函数的结果返回。函数只是一个可以执行的值，此外并无特殊之处。

由于函数与其他数据类型地位平等，所以在 JavaScript 语言中又称函数为第一等公民。

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

// 将函数赋值给一个变量
var operator = add;

// 将函数作为参数和返回值
function a(op){
  return op;
}
a(add)(1, 1)
// 2
```

### 函数名的提升

JavaScript 引擎将函数名视同变量名，所以采用`function`命令声明函数时，整个函数会像变量声明一样，被提升到代码头部。所以，下面的代码不会报错。

```javascript
f();

function f() {}
```

表面上，上面代码好像在声明之前就调用了函数`f`。但是实际上，由于“变量提升”，函数`f`被提升到了代码头部，也就是在调用之前已经声明了。但是，如果采用赋值语句定义函数，JavaScript 就会报错。

```javascript
f();
var f = function (){};
// TypeError: undefined is not a function
```

上面的代码等同于下面的形式。

```javascript
var f;
f();
f = function () {};
```

上面代码第二行，调用`f`的时候，`f`只是被声明了，还没有被赋值，等于`undefined`，所以会报错。因此，如果同时采用`function`命令和赋值语句声明同一个函数，最后总是采用赋值语句的定义。

```javascript
var f = function () {
  console.log('1');
}

function f() {
  console.log('2');
}

f() // 1
```

## 函数的属性和方法

### name 属性

函数的`name`属性返回函数的名字。

```javascript
function f1() {}
f1.name // "f1"
```

如果是通过变量赋值定义的函数，那么`name`属性返回变量名。

```javascript
var f2 = function () {};
f2.name // "f2"
```

但是，上面这种情况，只有在变量的值是一个匿名函数时才是如此。如果变量的值是一个具名函数，那么`name`属性返回`function`关键字之后的那个函数名。

```javascript
var f3 = function myName() {};
f3.name // 'myName'
```

上面代码中，`f3.name`返回函数表达式的名字。注意，真正的函数名还是`f3`，而`myName`这个名字只在函数体内部可用。

`name`属性的一个用处，就是获取参数函数的名字。

```javascript
var myFunc = function () {};

function test(f) {
  console.log(f.name);
}

test(myFunc) // myFunc
```

上面代码中，函数`test`内部通过`name`属性，就可以知道传入的参数是什么函数。

### length 属性

函数的`length`属性返回函数预期传入的参数个数，即函数定义之中的参数个数。

```javascript
function f(a, b) {}
f.length // 2
```

上面代码定义了空函数`f`，它的`length`属性就是定义时的参数个数。不管调用时输入了多少个参数，`length`属性始终等于2。

`length`属性提供了一种机制，判断定义时和调用时参数的差异，以便实现面向对象编程的“方法重载”（overload）。

### toString()

函数的`toString`方法返回一个字符串，内容是函数的源码。

```javascript
function f() {
  a();
  b();
  c();
}

f.toString()
```

```
// function f() {
//  a();
//  b();
//  c();
// }
```

对于那些原生的函数，`toString()`方法返回`function (){[native code]}`。

```javascript
Math.sqrt.toString()
// "function sqrt() { [native code] }"
```

上面代码中，`Math.sqrt`是 JavaScript 引擎提供的原生函数，`toString()`方法就返回原生代码的提示。

函数内部的注释也可以返回。

```javascript
function f() {/*
  这是一个
  多行注释
*/}

f.toString()
// "function f(){/*
//   这是一个
//   多行注释
// */}"
```

利用这一点，可以变相实现多行字符串。

```javascript
var multiline = function (fn) {
  var arr = fn.toString().split('\n');
  return arr.slice(1, arr.length - 1).join('\n');
};

function f() {/*
  这是一个
  多行注释
*/}

multiline(f);
// " 这是一个
//   多行注释"
```

## 函数作用域

### 定义

作用域（scope）指的是变量存在的范围。在 ES5 的规范中，JavaScript 只有两种作用域：一种是全局作用域，变量在整个程序中一直存在，所有地方都可以读取；另一种是函数作用域，变量只在函数内部存在。ES6 又新增了块级作用域，本教程不涉及。

对于顶层函数来说，函数外部声明的变量就是全局变量（global variable），它可以在函数内部读取。

```javascript
var v = 1;

function f() {
  console.log(v);
}

f()
// 1
```

上面的代码表明，函数`f`内部可以读取全局变量`v`。

在函数内部定义的变量，外部无法读取，称为“局部变量”（local variable）。

```javascript
function f(){
  var v = 1;
}

v // ReferenceError: v is not defined
```

上面代码中，变量`v`在函数内部定义，所以是一个局部变量，函数之外就无法读取。

函数内部定义的变量，会在该作用域内覆盖同名全局变量。

```javascript
var v = 1;

function f(){
  var v = 2;
  console.log(v);
}

f() // 2
v // 1
```

上面代码中，变量`v`同时在函数的外部和内部有定义。结果，在函数内部定义，局部变量`v`覆盖了全局变量`v`。

注意，对于`var`命令来说，局部变量只能在函数内部声明，在其他区块中声明，一律都是全局变量。

```javascript
if (true) {
  var x = 5;
}
console.log(x);  // 5
```

上面代码中，变量`x`在条件判断区块之中声明，结果就是一个全局变量，可以在区块之外读取。

### 函数内部的变量提升

与全局作用域一样，函数作用域内部也会产生“变量提升”现象。`var`命令声明的变量，不管在什么位置，变量声明都会被提升到函数体的头部。

```javascript
function foo(x) {
  if (x > 100) {
    var tmp = x - 100;
  }
}

// 等同于
function foo(x) {
  var tmp;
  if (x > 100) {
    tmp = x - 100;
  };
}
```

### 函数本身的作用域

函数本身也是一个值，也有自己的作用域。它的作用域与变量一样，就是其声明时所在的作用域，与其运行时所在的作用域无关。

```javascript
var a = 1;
var x = function () {
  console.log(a);
};

function f() {
```

```
  var a = 2;
  x();
}

f() // 1
```

上面代码中，函数`x`是在函数`f`的外部声明的，所以它的作用域绑定外层，内部变量`a`不会到函数`f`体内取值，所以输出`1`，而不是`2`。

总之，函数执行时所在的作用域，是定义时的作用域，而不是调用时所在的作用域。

很容易犯错的一点是，如果函数`A`调用函数`B`，却没考虑到函数`B`不会引用函数`A`的内部变量。

```javascript
var x = function () {
  console.log(a);
};

function y(f) {
  var a = 2;
  f();
}

y(x)
// ReferenceError: a is not defined
```

上面代码将函数`x`作为参数，传入函数`y`。但是，函数`x`是在函数`y`体外声明的，作用域绑定外层，因此找不到函数`y`的内部变量`a`，导致报错。

同样的，函数体内部声明的函数，作用域绑定函数体内部。

```javascript
function foo() {
  var x = 1;
  function bar() {
    console.log(x);
  }
  return bar;
}

var x = 2;
var f = foo();
f() // 1
```

上面代码中，函数`foo`内部声明了一个函数`bar`，`bar`的作用域绑定`foo`。当我们在`foo`外部取出`bar`执行时，变量`x`指向的是`foo`内部的`x`，而不是`foo`外部的`x`。正是这种机制，构成了下文要讲解的“闭包”现象。

## 参数

### 概述

函数运行的时候，有时需要提供外部数据，不同的外部数据会得到不同的结果，这种外部数据就叫参数。

```javascript
function square(x) {
  return x * x;
}

square(2) // 4
square(3) // 9
```

上式的`x`就是`square`函数的参数。每次运行的时候，需要提供这个值，否则得不到结果。

### 参数的省略

函数参数不是必需的，JavaScript 允许省略参数。

```javascript
function f(a, b) {
  return a;
}

f(1, 2, 3) // 1
f(1) // 1
f() // undefined

f.length // 2
```

上面代码的函数`f`定义了两个参数，但是运行时无论提供多少个参数（或者不提供参数），JavaScript 都不会报错。省略的参数的值就变为`undefined`。需要注意的是，函数的`length`属性与实际传入的参数个数无关，只反映函数预期传入的参数个数。

但是，没有办法只省略靠前的参数，而保留靠后的参数。如果一定要省略靠前的参数，只有显式传入`undefined`。

```javascript
function f(a, b) {
  return a;
}

f( , 1) // SyntaxError: Unexpected token ,(…)
f(undefined, 1) // undefined
```

```

上面代码中，如果省略第一个参数，就会报错。

### 传递方式

函数参数如果是原始类型的值（数值、字符串、布尔值），传递方式是传值传递（passes by value）。这意味着，在函数体内修改参数值，不会影响到函数外部。

```javascript
var p = 2;

function f(p) {
  p = 3;
}
f(p);

p // 2
```

上面代码中，变量`p`是一个原始类型的值，传入函数`f`的方式是传值传递。因此，在函数内部，`p`的值是原始值的拷贝，无论怎么修改，都不会影响到原始值。

但是，如果函数参数是复合类型的值（数组、对象、其他函数），传递方式是传址传递（pass by reference）。也就是说，传入函数的原始值的地址，因此在函数内部修改参数，将会影响到原始值。

```javascript
var obj = { p: 1 };

function f(o) {
  o.p = 2;
}
f(obj);

obj.p // 2
```

上面代码中，传入函数`f`的是参数对象`obj`的地址。因此，在函数内部修改`obj`的属性`p`，会影响到原始值。

注意，如果函数内部修改的，不是参数对象的某个属性，而是替换掉整个参数，这时不会影响到原始值。

```javascript
var obj = [1, 2, 3];

function f(o) {
  o = [2, 3, 4];
```
```

```
}
f(obj);

obj // [1, 2, 3]
```

上面代码中，在函数`f`内部，参数对象`obj`被整个替换成另一个值。这时不会影响到原始值。这是因为，形式参数（`o`）的值实际是参数`obj`的地址，重新对`o`赋值导致`o`指向另一个地址，保存在原地址上的值当然不受影响。

### 同名参数

如果有同名的参数，则取最后出现的那个值。

```javascript
function f(a, a) {
  console.log(a);
}

f(1, 2) // 2
```

上面代码中，函数`f`有两个参数，且参数名都是`a`。取值的时候，以后面的`a`为准，即使后面的`a`没有值或被省略，也是以其为准。

```javascript
function f(a, a) {
  console.log(a);
}

f(1) // undefined
```

调用函数`f`的时候，没有提供第二个参数，`a`的取值就变成了`undefined`。这时，如果要获得第一个`a`的值，可以使用`arguments`对象。

```javascript
function f(a, a) {
  console.log(arguments[0]);
}

f(1) // 1
```

### arguments 对象

**（1）定义**

由于 JavaScript 允许函数有不定数目的参数，所以需要一种机制，可以在函数体内部读取所有参数。这就是`arguments`对象的由来。

`arguments`对象包含了函数运行时的所有参数，`arguments[0]`就是第一个参数，`arguments[1]`就是第二个参数，以此类推。这个对象只有在函数体内部，才可以使用。

```javascript
var f = function (one) {
  console.log(arguments[0]);
  console.log(arguments[1]);
  console.log(arguments[2]);
}

f(1, 2, 3)
// 1
// 2
// 3
```

正常模式下，`arguments`对象可以在运行时修改。

```javascript
var f = function(a, b) {
  arguments[0] = 3;
  arguments[1] = 2;
  return a + b;
}

f(1, 1) // 5
```

上面代码中，函数`f`调用时传入的参数，在函数内部被修改成`3`和`2`。

严格模式下，`arguments`对象与函数参数不具有联动关系。也就是说，修改`arguments`对象不会影响到实际的函数参数。

```javascript
var f = function(a, b) {
  'use strict'; // 开启严格模式
  arguments[0] = 3;
  arguments[1] = 2;
  return a + b;
}

f(1, 1) // 2
```

上面代码中，函数体内是严格模式，这时修改`arguments`对象，不会影响到真实参数`a`和`b`。

通过`arguments`对象的`length`属性，可以判断函数调用时到底带几个参数。

```javascript
function f() {
  return arguments.length;
}

f(1, 2, 3) // 3
f(1) // 1
f() // 0
```

**（2）与数组的关系**

需要注意的是，虽然`arguments`很像数组，但它是一个对象。数组专有的方法（比如`slice`和`forEach`），不能在`arguments`对象上直接使用。

如果要让`arguments`对象使用数组方法，真正的解决方法是将`arguments`转为真正的数组。下面是两种常用的转换方法：`slice`方法和逐一填入新数组。

```javascript
var args = Array.prototype.slice.call(arguments);

// 或者
var args = [];
for (var i = 0; i < arguments.length; i++) {
  args.push(arguments[i]);
}
```

**（3）callee 属性**

`arguments`对象带有一个`callee`属性，返回它所对应的原函数。

```javascript
var f = function () {
  console.log(arguments.callee === f);
}

f() // true
```

可以通过`arguments.callee`，达到调用函数自身的目的。这个属性在严格模式里面是禁用的，因此不建议使用。

## 函数的其他知识点

### 闭包

闭包（closure）是 JavaScript 语言的一个难点，也是它的特色，很多高级应用都要依靠闭包实现。

理解闭包，首先必须理解变量作用域。前面提到，JavaScript 有两种作用域：全局作用域和函数作用域。函数内部可以直接读取全局变量。

```javascript
var n = 999;

function f1() {
  console.log(n);
}
f1() // 999
```

上面代码中，函数`f1`可以读取全局变量`n`。

但是，函数外部无法读取函数内部声明的变量。

```javascript
function f1() {
  var n = 999;
}

console.log(n)
// Uncaught ReferenceError: n is not defined(
```

上面代码中，函数`f1`内部声明的变量`n`，函数外是无法读取的。

如果出于种种原因，需要得到函数内的局部变量。正常情况下，这是办不到的，只有通过变通方法才能实现。那就是在函数的内部，再定义一个函数。

```javascript
function f1() {
  var n = 999;
  function f2() {
      console.log(n); // 999
  }
}
```

上面代码中，函数`f2`就在函数`f1`内部，这时`f1`内部的所有局部变量，对`f2`都是可见的。但是反过来就不行，`f2`内部的局部变量，对`f1`就是不可见的。这就是 JavaScript 语言特有的"链式作用域"结构（chain scope），子对象会一级一级地向上寻找所有父对象的变量。所以，父对象的所有变量，对子对象都是可见的，反之则不成立。

既然`f2`可以读取`f1`的局部变量，那么只要把`f2`作为返回值，我们不就可以在`f1`外部读取它的内部变量了吗！

```javascript
function f1() {
  var n = 999;
  function f2() {
    console.log(n);
  }
  return f2;
}

var result = f1();
result(); // 999
```

上面代码中，函数`f1`的返回值就是函数`f2`，由于`f2`可以读取`f1`的内部变量，所以就可以在外部获得`f1`的内部变量了。

闭包就是函数`f2`，即能够读取其他函数内部变量的函数。由于在 JavaScript 语言中，只有函数内部的子函数才能读取内部变量，因此可以把闭包简单理解成“定义在一个函数内部的函数”。闭包最大的特点，就是它可以“记住”诞生的环境，比如`f2`记住了它诞生的环境`f1`，所以从`f2`可以得到`f1`的内部变量。在本质上，闭包就是将函数内部和函数外部连接起来的一座桥梁。

闭包的最大用处有两个，一个是可以读取函数内部的变量，另一个就是让这些变量始终保持在内存中，即闭包可以使得它诞生环境一直存在。请看下面的例子，闭包使得内部变量记住上一次调用时的运算结果。

```javascript
function createIncrementor(start) {
  return function () {
    return start++;
  };
}

var inc = createIncrementor(5);

inc() // 5
inc() // 6
inc() // 7
```

上面代码中，`start`是函数`createIncrementor`的内部变量。通过闭包，`start`的状态被保留了，每一次调用都是在上一次调用的基础上进行计算。从中可以看到，闭包`inc`使得函数`createIncrementor`的内部环境，一直存在。所以，闭包可以看作是函数内部作用域的一个接口。

为什么会这样呢？原因就在于`inc`始终在内存中，而`inc`的存在依赖于`createIncrementor`，因此也始终在内存中，不会在调用结束后，被垃圾回收机制回收。

闭包的另一个用处，是封装对象的私有属性和私有方法。

```javascript
function Person(name) {
  var _age;
  function setAge(n) {
    _age = n;
  }
  function getAge() {
    return _age;
  }

  return {
    name: name,
    getAge: getAge,
    setAge: setAge
  };
}

var p1 = Person('张三');
p1.setAge(25);
p1.getAge() // 25
```

上面代码中，函数`Person`的内部变量`_age`，通过闭包`getAge`和`setAge`，变成了返回对象`p1`的私有变量。

注意，外层函数每次运行，都会生成一个新的闭包，而这个闭包又会保留外层函数的内部变量，所以内存消耗很大。因此不能滥用闭包，否则会造成网页的性能问题。

### 立即调用的函数表达式（IIFE）

在 JavaScript 中，圆括号`()`是一种运算符，跟在函数名之后，表示调用该函数。比如，`print()`就表示调用`print`函数。

有时，我们需要在定义函数之后，立即调用该函数。这时，你不能在函数的定义之后加上圆括号，这会产生语法错误。

```javascript
function(){ /* code */ }();
// SyntaxError: Unexpected token (
```

产生这个错误的原因是，`function`这个关键字即可以当作语句，也可以当作表达式。

```javascript
```

```
// 语句
function f() {}

// 表达式
var f = function f() {}
```

为了避免解析上的歧义，JavaScript 引擎规定，如果`function`关键字出现在行首，一律解释成语句。因此，JavaScript 引擎看到行首是`function`关键字之后，认为这一段都是函数的定义，不应该以圆括号结尾，所以就报错了。

解决方法就是不要让`function`出现在行首，让引擎将其理解成一个表达式。最简单的处理，就是将其放在一个圆括号里面。

```javascript
(function(){ /* code */ }());
// 或者
(function(){ /* code */ })();
```

上面两种写法都是以圆括号开头，引擎就会认为后面跟的是一个表示式，而不是函数定义语句，所以就避免了错误。这就叫做“立即调用的函数表达式”（Immediately-Invoked Function Expression），简称 IIFE。

注意，上面两种写法最后的分号都是必须的。如果省略分号，遇到连着两个 IIFE，可能就会报错。

```javascript
// 报错
(function(){ /* code */ }())
(function(){ /* code */ }())
```

上面代码的两行之间没有分号，JavaScript 会将它们连在一起解释，将第二行解释为第一行的参数。

推而广之，任何让解释器以表达式来处理函数定义的方法，都能产生同样的效果，比如下面三种写法。

```javascript
var i = function(){ return 10; }();
true && function(){ /* code */ }();
0, function(){ /* code */ }();
```

甚至像下面这样写，也是可以的。

```javascript
!function () { /* code */ }();
~function () { /* code */ }();
-function () { /* code */ }();
+function () { /* code */ }();
```

通常情况下，只对匿名函数使用这种“立即执行的函数表达式”。它的目的有两个：一是不必为函数命名，避免了污染全局变量；二是 IIFE 内部形成了一个单独的作用域，可以封装一些外部无法读取的私有变量。

```javascript
// 写法一
var tmp = newData;
processData(tmp);
storeData(tmp);

// 写法二
(function () {
  var tmp = newData;
  processData(tmp);
  storeData(tmp);
}());
```

上面代码中，写法二比写法一更好，因为完全避免了污染全局变量。

## eval 命令

### 基本用法

`eval`命令接受一个字符串作为参数，并将这个字符串当作语句执行。

```javascript
eval('var a = 1;');
a // 1
```

上面代码将字符串当作语句运行，生成了变量`a`。

如果参数字符串无法当作语句运行，那么就会报错。

```javascript
eval('3x') // Uncaught SyntaxError: Invalid or unexpected token
```

放在`eval`中的字符串，应该有独自存在的意义，不能用来与`eval`以外的命令配合使用。举例来说，下面的代码将会报错。

```javascript
eval('return;'); // Uncaught SyntaxError: Illegal return statement
```

上面代码会报错，因为`return`不能单独使用，必须在函数中使用。

如果`eval`的参数不是字符串，那么会原样返回。

```javascript
eval(123) // 123
```

`eval`没有自己的作用域，都在当前作用域内执行，因此可能会修改当前作用域的变量的值，造成安全问题。

```javascript
var a = 1;
eval('a = 2');

a // 2
```

上面代码中，`eval`命令修改了外部变量`a`的值。由于这个原因，`eval`有安全风险。

为了防止这种风险，JavaScript 规定，如果使用严格模式，`eval`内部声明的变量，不会影响到外部作用域。

```javascript
(function f() {
  'use strict';
  eval('var foo = 123');
  console.log(foo);  // ReferenceError: foo is not defined
})()
```

上面代码中，函数`f`内部是严格模式，这时`eval`内部声明的`foo`变量，就不会影响到外部。

不过，即使在严格模式下，`eval`依然可以读写当前作用域的变量。

```javascript
(function f() {
  'use strict';
  var foo = 1;
  eval('foo = 2');
  console.log(foo);  // 2
})()
```

上面代码中，严格模式下，`eval`内部还是改写了外部变量，可见安全风险依然存在。

总之，`eval`的本质是在当前作用域之中，注入代码。由于安全风险和不利于 JavaScript 引擎优化执行速度，所以一般不推荐使用。通常情况下，`eval`最常见的场合是解析 JSON 数据的字符串，不过正确的做法应该是使用原生的`JSON.parse`方法。

### eval 的别名调用

前面说过`eval`不利于引擎优化执行速度。更麻烦的是，还有下面这种情况，引擎在静态代码分析的阶段，根本无法分辨执行的是`eval`。

```javascript
var m = eval;
m('var x = 1');
x // 1
```

上面代码中，变量`m`是`eval`的别名。静态代码分析阶段，引擎分辨不出`m('var x = 1')`执行的是`eval`命令。

为了保证`eval`的别名不影响代码优化，JavaScript 的标准规定，凡是使用别名执行`eval`，`eval`内部一律是全局作用域。

```javascript
var a = 1;

function f() {
  var a = 2;
  var e = eval;
  e('console.log(a)');
}

f() // 1
```

上面代码中，`eval`是别名调用，所以即使它是在函数中，它的作用域还是全局作用域，因此输出的`a`为全局变量。这样的话，引擎就能确认`e()`不会对当前的函数作用域产生影响，优化的时候就可以把这一行排除掉。

`eval`的别名调用的形式五花八门，只要不是直接调用，都属于别名调用，因为引擎只能分辨`eval()`这一种形式是直接调用。

```javascript
eval.call(null, '...')
window.eval('...')
(1, eval)('...')
(eval, eval)('...')
```

上面这些形式都是`eval`的别名调用，作用域都是全局作用域。

## 参考链接

- Ben Alman, [Immediately-Invoked Function Expression (IIFE)](http://benalman.com/news/2010/11/immediately-invoked-function-expression/)
- Mark Daggett, [Functions Explained](http://markdaggett.com/blog/2013/02/15/functions-explained/)
- Juriy Zaytsev, [Named function expressions demystified](http://kangax.github.com/nfe/)
- Marco Rogers polotek, [What is the arguments object?](http://docs.nodejitsu.com/articles/javascript-conventions/what-is-the-arguments-object)
- Juriy Zaytsev, [Global eval. What are the options?](http://perfectionkills.com/global-eval-what-are-the-options/)
- Axel Rauschmayer, [Evaluating JavaScript code via eval() and new Function()](http://www.2ality.com/2014/01/eval.html)