

此页面由社区从英文翻译而来。了解更多并加入 MDN Web Docs 社区。

箭头函数

箭头函数表达式的语法比函数表达式更简洁，并且没有自己的 `this`，`arguments`，`super` 或 `new.target`。箭头函数表达式更适合作为匿名函数，并且它不能用作构造函数。

尝试一下

JavaScript Demo: Functions =>

```
1 const materials = [  
2   'Hydrogen',  
3   'Helium',  
4   'Lithium',  
5   'Beryllium'  
6 ];  
7  
8 console.log(materials.map(material => material.length));  
9 // Expected output: Array [8, 6, 7, 9]  
10
```

Run >

Reset

语法

基础语法

```
(param1, param2, ..., paramN) => { statements }  
(param1, param2, ..., paramN) => expression  
//相当于: (param1, param2, ..., paramN) =>{ return expression; }  
  
// 当只有一个参数时，圆括号是可选的：  
(singleParam) => { statements }  
singleParam => { statements }  
  
// 没有参数的函数应该写成一对圆括号。  
() => { statements }
```

高级语法

```
//加括号的函数体返回对象字面量表达式:  
params => ({foo: bar})  
  
//支持剩余参数和默认参数  
(param1, param2, ...rest) => { statements }  
(param1 = defaultValue1, param2, ..., paramN = defaultValueN) => {  
  statements }  
  
//同样支持参数列表解构  
let f = ([a, b] = [1, 2], {x: c} = {x: a + b}) => a + b + c;  
f(); // 6
```

描述

参考 ["ES6 In Depth: Arrow functions" on hacks.mozilla.org](https://hacks.mozilla.org/2015/12/es6-in-depth-arrow-functions/) .

引入箭头函数有两个方面的作用：更简短的函数并且不绑定 `this` 。

更短的函数

```
var elements = [  
  'Hydrogen',  
  'Helium',  
  'Lithium',  
  'Beryllium'  
];  
  
elements.map(function(element) {  
  return element.length;  
}); // 返回数组: [8, 6, 7, 9]  
  
// 上面的普通函数可以改写成如下的箭头函数  
elements.map((element) => {  
  return element.length;  
}); // [8, 6, 7, 9]  
  
// 当箭头函数只有一个参数时, 可以省略参数的圆括号  
elements.map(element => {  
  return element.length;  
}); // [8, 6, 7, 9]  
  
// 当箭头函数的函数体只有一个 `return` 语句时, 可以省略 `return` 关键字和方法体的花括号  
elements.map(element => element.length); // [8, 6, 7, 9]  
  
// 在这个例子中, 因为我们只需要 `length` 属性, 所以可以使用参数解构  
// 需要注意的是字符串 ``length`` 是我们想要获得的属性的名称, 而 `lengthFooBarX` 则只是个变量名,  
// 可以替换成任意合法的变量名  
elements.map(({ "length": lengthFooBarX }) => lengthFooBarX); // [8, 6, 7, 9]
```

没有单独的 `this`

在箭头函数出现之前, 每一个新函数根据它是被如何调用的来定义这个函数的 `this` 值:

- 如果该函数是一个构造函数, `this` 指针指向一个新的对象
- 在严格模式下的函数调用下, `this` 指向 `undefined`
- 如果该函数是一个对象的方法, 则它的 `this` 指针指向这个对象
- 等等

This 被证明是令人厌烦的面向对象风格的编程。

```
function Person() {
  // Person() 构造函数定义 `this` 作为它自己的实例。
  this.age = 0;

  setInterval(function growUp() {
    // 在非严格模式, growUp() 函数定义 `this` 作为全局对象,
    // 与在 Person() 构造函数中定义的 `this` 并不相同。
    this.age++;
  }, 1000);
}

var p = new Person();
```

在 ECMAScript 3/5 中, 通过将 this 值分配给封闭的变量, 可以解决 this 问题。

```
function Person() {
  var that = this;
  that.age = 0;

  setInterval(function growUp() {
    // 回调引用的是 `that` 变量, 其值是预期的对象。
    that.age++;
  }, 1000);
}
```

或者, 可以创建[绑定函数](#), 以便将预先分配的 this 值传递到绑定的目标函数 (上述示例中的 growUp() 函数)。

箭头函数不会创建自己的 this, 它只会从自己的作用域链的上一层继承 this。因此, 在下面的代码中, 传递给 setInterval 的函数内的 this 值相同:

```
function Person(){
  this.age = 0;

  setInterval(() => {
    this.age++; // |this| 正确地指向 p 实例
  }, 1000);
}

var p = new Person();
```

与严格模式的关系

鉴于 this 是词法层面上的, [严格模式](#)中与 this 相关的规则都将被忽略。

```
var f = () => { 'use strict'; return this; };
f() === window; // 或者 global
```

严格模式的其他规则依然不变。

通过 call 或 apply 调用

由于 箭头函数没有自己的 this 指针, 通过 call() 或 apply() 方法调用一个函数时, 只能传递参数 (不能绑定 this---译者注) 数会被忽略。(这种现象对于 bind 方法同样成立 --- 译者注)

```
var adder = {
  base : 1,

  add : function(a) {
```

```

    var f = v => v + this.base;
    return f(a);
  },

  addThruCall: function(a) {
    var f = v => v + this.base;
    var b = {
      base : 2
    };

    return f.call(b, a);
  }
};

console.log(adder.add(1));           // 输出 2
console.log(adder.addThruCall(1)); // 仍然输出 2

```

不绑定 arguments

箭头函数不绑定[Arguments 对象](#)。因此，在本示例中，arguments 只是引用了封闭作用域内的 arguments：

```

var arguments = [1, 2, 3];
var arr = () => arguments[0];

arr(); // 1

function foo(n) {
  var f = () => arguments[0] + n; // 隐式绑定 foo 函数的 arguments 对象。arguments[0] 是 n，即传给 foo 函数的第一个参数
  return f();
}

foo(1); // 2
foo(2); // 4
foo(3); // 6
foo(3,2);//6

```

在大多数情况下，使用[剩余参数](#)是相较使用 arguments 对象的更好选择。

```

function foo(arg) {
  var f = (...args) => args[0];
  return f(arg);
}
foo(1); // 1

function foo(arg1,arg2) {
  var f = (...args) => args[1];
  return f(arg1,arg2);
}
foo(1,2); //2

```

使用箭头函数作为方法

如上所述，箭头函数表达式对非方法函数是最合适的。让我们看看当我们试着把它们作为方法时发生了什么。

```

'use strict';
var obj = {
  i: 10,
  b: () => console.log(this.i, this),
  c: function() {
    console.log( this.i, this)
  }
}
obj.b();
// undefined, Window{...}

```

```
obj.c();  
// 10, Object {...}
```

箭头函数没有定义 this 绑定。另一个涉及 [Object.defineProperty\(\)](#) 的示例：

```
'use strict';  
var obj = {  
  a: 10  
};  
  
Object.defineProperty(obj, "b", {  
  get: () => {  
    console.log(this.a, typeof this.a, this);  
    return this.a+10;  
    // 代表全局对象 'Window'，因此 'this.a' 返回 'undefined'  
  }  
});  
  
obj.b; // undefined    "undefined"    Window {postMessage: f, blur: f, focus: f, close: f, frames: Window, ...}
```

使用 new 操作符

箭头函数不能用作构造器，和 new 一起用会抛出错误。

```
var Foo = () => {};  
var foo = new Foo(); // TypeError: Foo is not a constructor
```

使用 prototype 属性

箭头函数没有 prototype 属性。

```
var Foo = () => {};  
console.log(Foo.prototype); // undefined
```

使用 yield 关键字

[yield](#) 关键字通常不能在箭头函数中使用（除非是嵌套在允许使用的函数内）。因此，箭头函数不能用作函数生成器。

函数体

箭头函数可以有一个“简写体”或常见的“块体”。

在一个简写体中，只需要一个表达式，并附加一个隐式的返回值。在块体中，必须使用明确的 return 语句。

```
var func = x => x * x;  
// 简写函数 省略 return  
  
var func = (x, y) => { return x + y; };  
// 常规编写 明确的返回值
```

返回对象字面量

记住用 params => {object:literal} 这种简单的语法返回对象字面量是行不通的。

```
var func = () => { foo: 1 };  
// Calling func() returns undefined!  
  
var func = () => { foo: function() {} };  
// SyntaxError: function statement requires a name
```

这是因为花括号（`{}`）里面的代码被解析为一系列语句（即 `foo` 被认为是一个标签，而非对象字面量的组成部分）。

所以，记得用圆括号把对象字面量包起来：

```
var func = () => ({foo: 1});
```

换行

箭头函数在参数和箭头之间不能换行。

```
var func = ()
    => 1;
// SyntaxError: expected expression, got '=>'
```

但是，可以通过在`'=>'`之后换行，或者用`()`、`{}`来实现换行，如下：

```
var func = (a, b, c) =>
    1;
```

```
var func = (a, b, c) => (
    1
);
```

```
var func = (a, b, c) => {
    return 1
};
```

```
var func = (
    a,
    b,
    c
) => 1;
```

```
// 不会有语法错误
```

解析顺序

虽然箭头函数中的箭头不是运算符，但箭头函数具有与常规函数不同的特殊[运算符优先级](#)解析规则。

```
let callback;

callback = callback || function() {}; // ok

callback = callback || () => {};
// SyntaxError: invalid arrow-function arguments

callback = callback || (() => {});    // ok
```

更多示例

```
// 空的箭头函数返回 undefined
let empty = () => {};

(() => 'foobar')();
// Returns "foobar"
// （这是一个立即执行函数表达式，可参阅 'IIFE' 术语表）

var simple = a => a > 15 ? 15 : a;
simple(16); // 15
simple(10); // 10
```

```

let max = (a, b) => a > b ? a : b;

// Easy array filtering, mapping, ...

var arr = [5, 6, 13, 0, 1, 18, 23];

var sum = arr.reduce((a, b) => a + b);
// 66

var even = arr.filter(v => v % 2 == 0);
// [6, 0, 18]

var double = arr.map(v => v * 2);
// [10, 12, 26, 0, 2, 36, 46]

// 更简明的 promise 链
promise.then(a => {
  // ...
}).then(b => {
  // ...
});

// 无参数箭头函数在视觉上容易分析
setTimeout( () => {
  console.log('I happen sooner');
  setTimeout( () => {
    // deeper code
    console.log('I happen later');
  }, 1);
}, 1);

```

箭头函数也可以使用条件（三元）运算符：

```

var simple = a => a > 15 ? 15 : a;
simple(16); // 15
simple(10); // 10

let max = (a, b) => a > b ? a : b;

```

箭头函数内定义的变量及其作用域

```

// 常规写法
var greeting = () => {let now = new Date(); return ("Good" + ((now.getHours() > 17) ? " evening." : " day."));}
greeting();           //"Good day."
console.log(now);     // ReferenceError: now is not defined 标准的 let 作用域

// 参数括号内定义的变量是局部变量（默认参数）
var greeting = (now=new Date()) => "Good" + (now.getHours() > 17 ? " evening." : " day.");
greeting();           //"Good day."
console.log(now);     // ReferenceError: now is not defined

// 对比：函数体内{}不使用 var 定义的变量是全局变量
var greeting = () => {now = new Date(); return ("Good" + ((now.getHours() > 17) ? " evening." : " day."));}
greeting();           //"Good day."
console.log(now);     // Fri Dec 22 2017 10:01:00 GMT+0800 (中国标准时间)

// 对比：函数体内{} 用 var 定义的变量是局部变量
var greeting = () => {var now = new Date(); return ("Good" + ((now.getHours() > 17) ? " evening." : " day."));}
greeting();           //"Good day."
console.log(now);     // ReferenceError: now is not defined

```

箭头函数也可以使用闭包：

```
// 标准的闭包函数
function A(){
    var i=0;
    return function b(){
        return (++i);
    };
};

var v=A();
v();    //1
v();    //2

//箭头函数体的闭包 (i=0 是默认参数)
var Add = (i=0) => {return (() => (++i) )};
var v = Add();
v();    //1
v();    //2

//因为仅有一个返回, return 及括号 () 也可以省略
var Add = (i=0)=> ()=> (++i);
```

箭头函数递归

```
var fact = (x) => ( x==0 ? 1 : x*fact(x-1) );
fact(5);    // 120
```


规范

Specification
ECMAScript Language Specification
sec-arrow-function-definitions

浏览器兼容性

[Report problems with this compatibility data on GitHub](#)

	Chrome	Edge	Firefox	Opera	Safari	Chrome Android	Firefox for Android
Arrow functions	Chrome 45	Edge 12	Firefox 22	Opera 32	Safari 10	Chrome 45 Android	Firefox for Android
Trailing comma in parameters	Chrome 58	Edge 12	Firefox 52	Opera 45	Safari 10	Chrome 58 Android	Firefox Andro



Master React 18 and build scalable apps with a complete React learning path.

Mozilla ads

Don't want to see ads?

	ne		x			ne Android	x for Android

Tip: you can click/tap on a cell for more information.

Full support Partial support No support See implementation notes. User must explicitly enable this feature.

Has more compatibility info.

相关链接

- ["ES6 In Depth: Arrow functions" on hacks.mozilla.org](#)

This page was last modified on 2023年4月7日 by [MDN contributors](#).