> 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。

箭头函数表达式

箭头函数表达式的语法比传统的函数表达式更简洁，但在语义上有一些差异，在用法上也有一些限制：

- 箭头函数没有独立的 `this`、`arguments` 和 `super` 绑定，并且不可被用作方法。
- 箭头函数不能用作构造函数。使用 `new` 调用它们会引发 `TypeError`。它们也无法访问 `new.target` 关键字。
- 箭头函数不能在其主体中使用 `yield`，也不能作为生成器函数创建。

尝试一下

JavaScript Demo: Functions ⇒

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

Run ›

Reset

语法

JS

```
() => expression

param => expression

(param) => expression

(param1, paramN) => expression

() => {
  statements
}

param => {
  statements
}
```

```
(param1, paramN) => {
  statements
}
```

参数部分支持剩余参数、默认参数和解构赋值，并且始终需要使用括号：

JS

```
(a, b, ...r) => expression
(a = 400, b = 20, c) => expression
([a, b] = [10, 20]) => expression
({ a, b } = { a: 10, b: 20 }) => expression
```

箭头函数可以是 async 的，方法是在表达式前加上 async 关键字。

JS

```
async param => expression
async (param1, param2, ...paramN) => {
  statements
}
```

描述

让我们逐步将传统的匿名函数分解为最简单的箭头函数。每一步都是一个有效的箭头函数。

> **备注：** 传统函数表达式和箭头函数除了语法上的区别外，还有更多的不同。我们将在接下来的几个小节中详细介绍它们的行为差异。

JS

```
// 传统匿名函数
(function (a) {
  return a + 100;
});

// 1. 移除“function”，并将箭头放置于参数和函数体起始大括号之间
(a) => {
  return a + 100;
};

// 2. 移除代表函数体的大括号和“return”——返回值是隐含的
(a) => a + 100;

// 3. 移除参数周围的括号
a => a + 100;
```

在上面的示例中，参数周围的括号和函数体周围的大括号都可以省略。但是，只有在某些情况下才能省略。

只有当函数只有一个简单参数时，才能省略括号。如果函数有多个参数、无参数、默认参数、重组参数或其余参数，则需要在参数列表周围加上括号。

JS

```
// 传统匿名函数
(function (a, b) {
  return a + b + 100;
});

// 箭头函数
(a, b) => a + b + 100;

const a = 4;
const b = 2;
```

```
// 传统无参匿名函数
(function () {
  return a + b + 100;
});

// 无参箭头函数
() => a + b + 100;
```

只有当函数直接返回表达式时，才可以省略大括号。如果函数体有额外的处理，则大括号是必需的， `return` 关键字也是必需的。箭头函数无法猜测函数体返回什么或何时返回。

JS

```
// 传统匿名函数
(function (a, b) {
  const chuck = 42;
  return a + b + chuck;
});

// 箭头函数
(a, b) => {
  const chuck = 42;
  return a + b + chuck;
};
```

箭头函数总是未命名的。如果箭头函数需要调用自身，请使用具名函数表达式。也可以将箭头函数赋值给一个变量，这样它就有了名字。

JS

```
// 传统函数
function bob(a) {
  return a + 100;
}

// 箭头函数
const bob2 = (a) => a + 100;
```

函数体

箭头函数既可以使用*表达式体*（expression body），也可以使用通常的*块体*（block body）。

在表达式体中，只需指定一个表达式，它将成为隐式返回值。在块体中，必须使用显式的 `return` 语句。

JS

```
const func = (x) => x * x;
// 表达式体语法，隐含返回值

const func2 = (x, y) => {
  return x + y;
};
// 块体语法，需要明确返回值
```

使用表达式体语法 `(params) => { object: literal }` 返回对象字面量时，不能按预期工作。

JS

```
const func = () => { foo: 1 };
// 调用 func() 会返回 undefined!

const func2 = () => { foo: function () {} };
// SyntaxError: function statement requires a name
```

```
const func3 = () => { foo() {} };
// SyntaxError: Unexpected token '{'
```

这是因为只有当箭头后面的标记不是左括号时，JavaScript 才会将箭头函数视为表达式体，因此括号（{}）内的代码会被解析为一系列语句，其中 `foo` 是标签，而不是对象文字中的键。

要解决这个问题，可以用括号将对象字面量包装起来：

JS

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

不能用作方法

箭头函数表达式只能用于非方法函数，因为它们没有自己的 `this`。让我们看看将它们用作方法时会发生什么：

JS

```
"use strict";

const obj = {
  i: 10,
  b: () => console.log(this.i, this),
  c() {
    console.log(this.i, this);
  },
};

obj.b(); // 输出 undefined, Window { /* … */ }（或全局对象）
obj.c(); // 输出 10, Object { /* … */ }
```

另外一个示例涉及到 `Object.defineProperty()`：

JS

```
"use strict";

const obj = {
  a: 10,
};

Object.defineProperty(obj, "b", {
  get: () => {
    console.log(this.a, typeof this.a, this); // undefined 'undefined' Window { /* … */ }（或全局对象）
    return this.a + 10; // 代表全局对象 'Window'，故 `this.a' 返回 'undefined'
  },
});
```

由于类体具有 `this` 上下文，因此作为类字段的箭头函数会关闭类的 `this` 上下文，箭头函数体中的 `this` 将正确指向实例（对于静态字段来说是类本身）。但是，由于它是一个闭包，而不是函数本身的绑定，因此 `this` 的值不会根据执行上下文而改变。

JS

```
class C {
  a = 1;
  autoBoundMethod = () => {
    console.log(this.a);
  };
}

const c = new C();
c.autoBoundMethod(); // 1
const { autoBoundMethod } = c;
```

```
autoBoundMethod(); // 1
// 如果这是普通方法，此时应该是 undefined
```

箭头函数属性通常被称作"自动绑定方法"，因为它与普通方法的等价性相同：

JS

```
class C {
  a = 1;
  constructor() {
    this.method = this.method.bind(this);
  }
  method() {
    console.log(this.a);
  }
}
```

> **备注：** 类字段是在*实例*（instance）上定义的，而不是在*原型*（prototype）上定义的，因此每次创建实例都会创建一个新的函数引用并分配一个新的闭包，这可能会导致比普通非绑定方法更多的内存使用。

出于类似原因，`call()`、`apply()` 和 `bind()` 方法在箭头函数上调用时不起作用，因为箭头函数是根据箭头函数定义的作用域来建立 `this` 的，而 `this` 值不会根据函数的调用方式而改变。

没有参数绑定

箭头函数没有自己的 `arguments` 对象。因此，在本例中，`arguments` 是对外层作用域参数的引用：

JS

```
function foo(n) {
  const f = () => arguments[0] + n; // foo 的隐式参数绑定。arguments[0] 为 n
  return f();
}

foo(3); // 3 + 3 = 6
```

> **备注：** 在严格模式下不能声明名为 `arguments` 的变量，因此上面的代码会出现语法错误。这使得 `arguments` 的范围效应更容易理解。

在大多数情况下，使用剩余参数是比使用 `arguments` 对象更好的选择。

JS

```
function foo(n) {
  const f = (...args) => args[0] + n;
  return f(10);
}

foo(1); // 11
```

不能用作构造函数

箭头函数不能用作构造函数，当使用 `new` 调用时会出错。它们也没有 `prototype` 属性。

JS

```
const Foo = () => {};
const foo = new Foo(); // TypeError: Foo is not a constructor
console.log("prototype" in Foo); // false
```

不能用作生成器

箭头函数的主体中不能使用 `yield` 关键字（除非在箭头函数进一步嵌套的生成器函数中使用）。因此，箭头函数不能用作生成器。

箭头前换行

箭头函数的参数和箭头之间不能换行。

```
JS
```

```
const func = (a, b, c)
  => 1;
// SyntaxError: Unexpected token '=>'
```

为便于格式化，可在箭头后换行，或在函数体周围使用括号/花括号，如下图所示。也可以在参数之间换行。

```
JS
```

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

const func2 = (a, b, c) => (
  1
);

const func3 = (a, b, c) => {
  return 1;
};

const func4 = (
  a,
  b,
  c,
) => 1;
```

箭头的优先级

虽然箭头函数中的箭头不是运算符，但与普通函数相比，箭头函数具有特殊的解析规则，与运算符优先级的交互方式不同。

```
JS
```

```
let callback;

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

由于 `=>` 的优先级低于大多数运算符，因此需要使用括号来避免 `callback || ()` 被解析为箭头函数的参数列表。

```
JS
```

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

示例

使用箭头函数

```
JS
```

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

(() => "foobar")();
// 返回 "foobar"
// 这是一个立即执行函数表达式
```

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

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

// 更方便进行数组的过滤、映射等工作
const arr = [5, 6, 13, 0, 1, 18, 23];

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

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

const 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("我发生更早");
  setTimeout(() => {
    // 深层次代码
    console.log("我发生更晚");
  }, 1);
}, 1);
```

使用 call、bind 和 apply

call() 、 apply() 和 bind() 方法与传统函数一样按照预期工作，因为我们为每个方法建立了作用域：

JS

```
const obj = {
  num: 100,
};

// 在 globalThis 上设置“num”，以显示它如何没有被使用到。
globalThis.num = 42;

// 对“this”进行操作的简单传统函数
const add = function (a, b, c) {
  return this.num + a + b + c;
};

console.log(add.call(obj, 1, 2, 3)); // 106
console.log(add.apply(obj, [1, 2, 3])); // 106
const boundAdd = add.bind(obj);
console.log(boundAdd(1, 2, 3)); // 106
```

对于箭头函数，由于我们的 add 函数基本上是在 globalThis （全局）作用域上创建的，因此它会假定 this 就是 globalThis 。

JS

```
const obj = {
  num: 100,
```

```
};

// 在 globalThis 上设置“num”，以显示它是如何被接收到的。
globalThis.num = 42;

// 箭头函数
const add = (a, b, c) => this.num + a + b + c;

console.log(add.call(obj, 1, 2, 3)); // 48
console.log(add.apply(obj, [1, 2, 3])); // 48
const boundAdd = add.bind(obj);
console.log(boundAdd(1, 2, 3)); // 48
```

使用箭头函数的最大好处可能是在使用 `setTimeout()` 和 `EventTarget.prototype.addEventListener()` 等方法时，这些方法通常需要某种闭包、`call()`、`apply()` 或 `bind()`，以确保函数在适当的作用域中执行。

对于传统的函数表达式，类似这样的代码并不能像预期的那样工作：

JS

```
const obj = {
  count: 10,
  doSomethingLater() {
    setTimeout(function () {
      // 此函数在 window 作用域下执行
      this.count++;
      console.log(this.count);
    }, 300);
  },
};

obj.doSomethingLater(); // 输出“NaN”，因为“count”属性不在 window 作用域下。
```

有了箭头函数，`this` 作用域更容易被保留：

JS

```
const obj = {
  count: 10,
  doSomethingLater() {
    // 该方法语法将“this”与“obj”上下文绑定。
    setTimeout(() => {
      // 由于箭头函数没有自己的绑定，
      // 而 setTimeout（作为函数调用）本身也不创建绑定，
      // 因此使用了外部方法的“obj”上下文。
      this.count++;
      console.log(this.count);
    }, 300);
  },
};

obj.doSomethingLater(); // 输出 11
```

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

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

Full support See implementation notes.

参见

- 函数指南
- 函数参考
- `function`
- `function` 表达式
- 深入了解 ES6：箭头函数 ，载于 hacks.mozilla.org（2015）