函数——可重用的代码块

编码中的另一个基本概念是**函数**，它允许您在定义的块中存储一段执行单个任务的代码，然后在需要时使用单个短命令调用该代码，而不必输入相同的代码代码多次。在本文中，我们将探讨函数背后的基本概念，例如基本语法、如何调用和定义它们、作用域和参数。

先决条件：	基本的计算机知识，对 HTML 和 CSS 的基本了解，JavaScript 的第一步。
客观的：	了解 JavaScript 函数背后的基本概念。

我在哪里可以找到函数?

在 JavaScript 中，函数无处不在。事实上，到目前为止，我们一直在使用函数。我们只是没有过多地谈论它们。然而，现在是时候让我们开始明确地讨论函数，并真正探索它们的语法了。

几乎任何时候你都在使用带有一对括号的 JavaScript 结构 — () — 而你**没有**使用常见的内置语言结构，如for 循环、while 或 do...while 循环，或者 if... else 语句，你正在使用一个函数。

内置浏览器功能

在本课程中，我们大量使用了浏览器内置的功能。

每次我们操作一个文本字符串，例如：

```
const myText = 'I am a string';
const newString = myText.replace('string', 'sausage');
console.log(newString);
// the replace() string function takes a source string,
```

```
// and a target string and replaces the source string,
// with the target string, and returns the newly formed string
```

或者每次我们操作数组时：

```
const myArray = ['I', 'love', 'chocolate', 'frogs'];
const madeAString = myArray.join(' ');
console.log(madeAString);
// the join() function takes an array, joins
// all the array items together into a single
// string, and returns this new string
```

或者每次我们生成一个随机数：

```
const myNumber = Math.random();
// the random() function generates a random number between
// 0 and up to but not including 1, and returns that number
```

我们正在使用一个*函数*!

> **注意：**如果需要，请随意将这些行输入浏览器的 JavaScript 控制台以重新熟悉它们的功能。

JavaScript 语言有许多内置函数，让您无需自己编写所有代码即可完成有用的事情。事实上，当您调用（运行或执行的花哨词）浏览器内置功能时调用的某些代码不能用 JavaScript 编写——其中许多功能都在调用后台浏览器代码的一部分，这些代码主要使用 C++ 等低级系统语言编写，而不是 JavaScript 等网络语言。

请记住，一些内置浏览器功能不是核心 JavaScript 语言的一部分——有些被定义为浏览器 API 的一部分，它们构建在默认语言之上以提供更多功能（请参阅我们课程的早期部分了解更多说明）。我们将在后面的模块中更详细地介绍如何使用浏览器 API。

函数与方法

作为对象一部分的函数称为**方法**。您还不需要了解结构化 JavaScript 对象的内部工作原理——您可以等到我们后面的模块，它会教您所有关于对象的内部工作原理，以及如何创建您自己的对象。现在，我们只是想澄清方法与函数之间可能存在的任何混淆——当您查看 Web 上可用的相关资源时，您很可能会同时遇到这两个术语。

到目前为止，我们使用的内置代码有两种形式：**函数**和**方法**。您可以在此处查看内置函数的完整列表，以及内置对象及其对应的方法。

到目前为止，您还在课程中看到了很多**自定义函数— 在您的代码中定义的函数，而不是在浏览器中定义的函数**。任何时候当你看到一个自定义名称后面紧跟着一个括号时，你都在使用一个自定义函数。在我们的循环文章中的random-canvas-circles.html 示例（另请参阅完整源代码 ）中，我们包含了一个如下所示的自定义函数： `draw()`

```
function draw() {
  ctx.clearRect(0,0,WIDTH,HEIGHT);
  for (let i = 0; i < 100; i++) {
    ctx.beginPath();
    ctx.fillStyle = 'rgba(255,0,0,0.5)';
    ctx.arc(random(WIDTH), random(HEIGHT), random(50), 0, 2 *
Math.PI);
    ctx.fill();
  }
}
```

此函数在一个元素内绘制 100 个随机圆圈 `<canvas>` 。每次我们想要这样做时，我们都可以用这个调用函数：

```
draw();
```

而不是每次我们想重复它时都必须重新写出所有代码。函数可以包含您喜欢的任何代码——您甚至可以从函数内部调用其他函数。上面的函数例如调用 `random()` 函数三次，定义如下代码：

```
function random(number) {
  return Math.floor(Math.random()*number);
```

```
}
```

我们需要这个函数，因为浏览器内置的Math.random()函数只生成一个介于 0 和 1 之间的随机十进制数。我们想要一个介于 0 和指定数字之间的随机整数。

调用函数

您现在可能已经清楚这一点，但为了以防万一，要在函数定义后实际使用它，您必须运行（或调用）它。这是通过在代码中某处包含函数名称，后跟括号来完成的。

```
function myFunction() {
  alert('hello');
}

myFunction();
// calls the function once
```

> **注意：**这种创建函数的形式也称为*函数声明*。它总是被提升，所以你可以在函数定义之上调用函数，它会正常工作。

函数参数

有些函数需要在调用它们时指定**参数——这些是需要包含在函数括号内的值，它需要正确地完成它的工作。**

> **注意：**参数有时称为参数、属性，甚至属性。

例如，浏览器内置的Math.random()函数不需要任何参数。调用时，它总是返回一个介于 0 和 1 之间的随机数：

```
const myNumber = Math.random();
```

然而，浏览器的内置字符串replace()函数需要两个参数——在主字符串中查找的子字符串，以及用以下内容替换该字符串的子字符串：

```
const myText = 'I am a string';
const newString = myText.replace('string', 'sausage');
```

> **注意：**当需要指定多个参数时，以逗号分隔。

可选参数

有时参数是可选的——您不必指定它们。如果不这样做，函数通常会采用某种默认行为。例如，数组join()函数的参数是可选的：

```
const myArray = ['I', 'love', 'chocolate', 'frogs'];
const madeAString = myArray.join(' ');
console.log(madeAString);
// returns 'I love chocolate frogs'

const madeAnotherString = myArray.join();
console.log(madeAnotherString);
// returns 'I,love,chocolate,frogs'
```

如果没有包含参数来指定连接/分隔字符，则默认使用逗号。

默认参数

= 如果您正在编写一个函数并希望支持可选参数，您可以通过在参数名称后添加默认值来指定默认值，后跟默认值：

```
function hello(name = 'Chris') {
  console.log(`Hello ${name}!`);
}

hello('Ari'); // Hello Ari!
hello();      // Hello Chris!
```

匿名函数和箭头函数

到目前为止，我们刚刚创建了一个像这样的函数：

```
function myFunction() {
  alert('hello');
}
```

但是你也可以创建一个没有名字的函数：

```
(function () {
  alert('hello');
})
```

这称为**匿名函数**，因为它没有名称。当一个函数希望接收另一个函数作为参数时，您会经常看到匿名函数。在这种情况下，函数参数通常作为匿名函数传递。

> **注意：**这种创建函数的形式也称为*函数表达式*。与函数声明不同，函数表达式不会提升。

匿名函数示例

例如，假设您希望在用户向文本框输入内容时运行一些代码。为此，您可以调用 `addEventListener()` 文本框的功能。此函数希望您向它传递（至少）两个参数：

- 要监听的事件的名称，在本例中是 `keydown`
- 事件发生时运行的函数。

当用户按下一个键时，浏览器将调用您提供的函数，并将包含有关此事件的信息的参数传递给它，包括用户按下的特定键：

```
function logKey(event) {
  console.log(`You pressed "${event.key}".`);
}

textBox.addEventListener('keydown', logKey);
```

`logKey()` 您可以将匿名函数传递给：而不是定义单独的函数 `addEventListener()`：

```
textBox.addEventListener('keydown', function(event) {
  console.log(`You pressed "${event.key}".`);
});
```

箭头函数

如果你像这样传递一个匿名函数，你可以使用另一种形式，称为**箭头函数**。而不是 `function(event)`，你写 `(event) =>`：

```
textBox.addEventListener('keydown', (event) => {
  console.log(`You pressed "${event.key}".`);
});
```

如果函数在大括号中只有一行，则省略大括号：

```
textBox.addEventListener('keydown', (event) => console.log(`You
pressed "${event.key}".`));
```

如果函数只接受一个参数，也可以省略参数两边的括号：

```
textBox.addEventListener('keydown', event => console.log(`You
pressed "${event.key}".`));
```

最后，如果你的函数需要返回一个值，并且只包含一行，你也可以省略该 `return` 语句。在下面的示例中，我们使用 `map()` 方法 `Array` 将原始数组中的每个值加倍：

```
const originals = [1, 2, 3];

const doubled = originals.map((item) => item * 2);

console.log(doubled); // [2, 4, 6]
```

该 `map()` 方法依次获取数组中的每个项目，并将其传递给给定的函数。然后它获取该函数返回的值并将其添加到一个新数组中。

所以在上面的例子中，`(item) => item * 2` 箭头函数是否等价于：

```
function doubleItem(item) {
  return item * 2;
}
```

我们建议您使用箭头函数，因为它们可以使您的代码更短且更易读。要了解更多信息，请参阅JavaScript 指南中关于箭头函数的部分，以及我们关于箭头函数的参考页面。

> **注意：** 箭头函数和普通函数之间有一些细微的差别。它们超出了本介绍性指南的范围，并且不太可能对我们在此讨论的案例产生影响。要了解更多信息，请参阅箭头函数参考文档。

箭头函数实例

这是我们上面讨论的“keydown”示例的完整工作示例：

HTML：

```
<input id="textBox" type="text">
<div id="output"></div>
```

JavaScript：

```
const textBox = document.querySelector("#textBox");
const output = document.querySelector("#output");

textBox.addEventListener('keydown', (event) =>
output.textContent = `You pressed "${event.key}".`);
```

结果 - 尝试在文本框中输入并查看输出：

功能范围和冲突

让我们谈谈作用域——处理函数时一个非常重要的概念。当你创建一个函数时，函数内部定义的变量和其他东西都在它们自己独立的**范围**内，这意味着它们被锁定在它们自己独立的隔间中，无法从函数外部的代码访问。

所有功能之外的顶层称为**全局范围**。在代码的任何地方都可以访问在全局范围内定义的值。

出于各种原因，JavaScript 是这样设置的——但主要是因为安全性和组织性。有时您不希望代码中的任何地方都可以访问变量——您从其他地方调用的外部脚本可能会开始扰乱您的代码并导致问题，因为它们碰巧使用与代码其他部分相同的变量名，引起冲突。这可能是恶意的，也可能是偶然的。

例如，假设您有一个调用两个外部 JavaScript 文件的 HTML 文件，并且它们都定义了一个使用相同名称的变量和函数：

```
<!-- Excerpt from my HTML -->
<script src="first.js"></script>
<script src="second.js"></script>
<script>
  greeting();
</script>
```

```
// first.js
const name = 'Chris';
function greeting() {
  alert(`Hello ${name}: welcome to our company.`);
}
```

```
// second.js
const name = 'Zaptec';
```

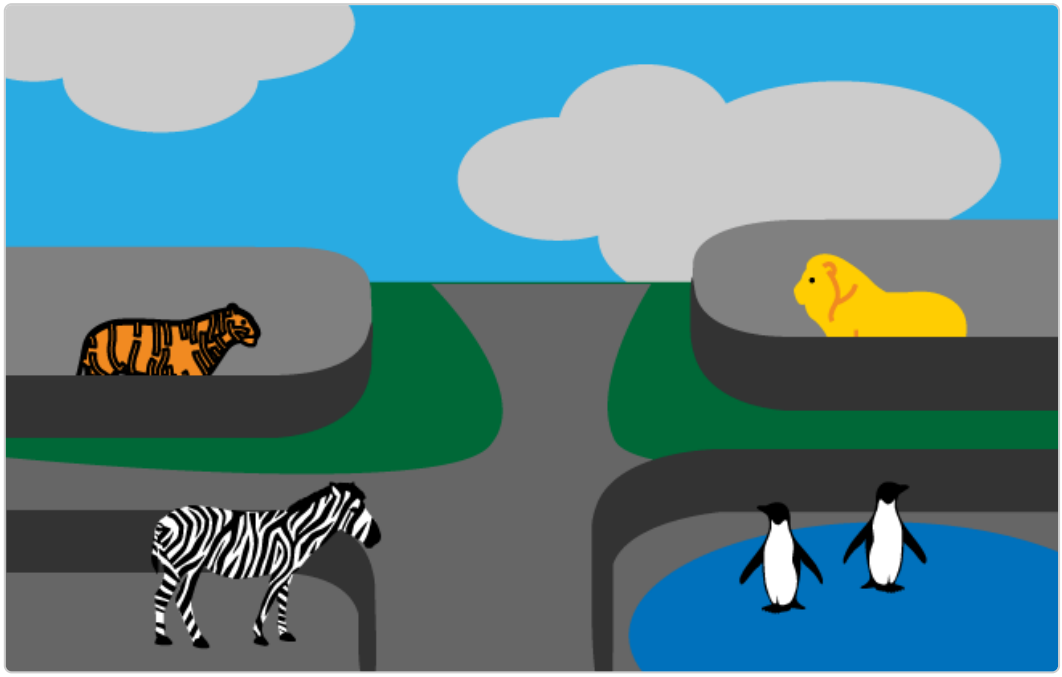
```
function greeting() {
  alert(`Our company is called ${name}.`);
}
```

您要调用的两个函数都被调用了 `greeting()`，但您只能访问 `first.js` 文件的 `greeting()` 函数（第二个被忽略）。 `second.js` 此外，尝试（在文件中）为变量分配新值时会导致错误 `name` ——因为它已经用 声明 `const`，因此无法重新分配。

> **注意：**您可以在 GitHub 上看到这个实时运行的 示例（另请参阅源代码 ）。

将部分代码锁定在函数中可以避免此类问题，这被认为是最佳实践。

它有点像动物园。狮子、斑马、老虎和企鹅都被关在自己的围栏里，并且只能访问围栏内的东西——与函数作用域的方式相同。如果他们能够进入其他围栏，就会出现问题。充其量，不同的动物在不熟悉的栖息地里会感到非常不舒服——狮子或老虎在企鹅的水域、冰冷的领地里会感觉很糟糕。最坏的情况是，狮子和老虎可能会试图吃掉企鹅！



动物园管理员就像全球范围——他们拥有进入每个围栏、补充食物、照顾生病动物等的钥匙。

主动学习：发挥作用域

让我们看一个真实的例子来演示作用域。

1. 首先，制作我们的function-scope.html 示例的本地副本。这包含两个名为 `a()` and的函数 `b()`，以及三个变量 — `x`、`y` 和 `z` — 其中两个在函数内部定义，一个在全局范围内定义。它还包含第三个函数，称为 `output()`，它接受一个参数并将其输出到页面的一个段落中。
2. 在浏览器和文本编辑器中打开示例。
3. 在浏览器开发人员工具中打开 JavaScript 控制台。在 JavaScript 控制台中，输入以下命令：

   ```
   output(x);
   ```

   `x` 您应该看到打印到浏览器视口 的变量值。
4. 现在尝试在您的控制台中输入以下内容

   ```
   output(y);
   output(z);
   ```

   这两个都应该沿着“ ReferenceError: y is not defined ”的行向控制台抛出错误。这是为什么？由于函数范围 — `y` 并且 `z` 被锁定在 `a()` 和 `b()` 函数中，因此 `output()` 在从全局范围调用时无法访问它们。
5. 但是，当它从另一个函数内部调用时呢？尝试编辑 `a()`， `b()` 它们看起来像这样：

   ```
   function a() {
     const y = 2;
     output(y);
   }

   function b() {
     const z = 3;
     output(z);
   }
   ```

保存代码并在浏览器中重新加载它，然后尝试从 JavaScript 控制台 调用 `a()` 和函数： `b()`

```
a();
b();
```

您应该会在浏览器视口中看到打印的 `y` 和值。 `z` 这很好用，因为该 `output()` 函数在其他函数内部被调用——在每种情况下都在与定义它正在打印的变量相同的范围内。 `output()` 本身在任何地方都可用，因为它是在全局范围内定义的。

6. 现在尝试像这样更新您的代码：

```
function a() {
  const y = 2;
  output(x);
}

function b() {
  const z = 3;
  output(x);
}
```

7. 保存并重新加载，然后在您的 JavaScript 控制台中再次尝试：

```
a();
b();
```

`a()` 和调用都应 `b()` 将 `x` 的值打印到浏览器视口。这些工作正常，因为即使调用不在定义的 `output()` 同一范围内，它也是一个全局变量，因此在所有代码中随处可用。 `x x`

8. 最后，尝试像这样更新您的代码：

```
function a() {
  const y = 2;
  output(z);
}

function b() {
  const z = 3;
```

9. 保存并重新加载，然后在您的 JavaScript 控制台中再次尝试：

```
a();
b();
```

这次 `a()` 和调用将在控制台中 `b()` 抛出烦人的ReferenceError: *variable name* is not defined `output()` 错误——这是因为它们试图打印的调用和变量不在相同的函数范围内——变量实际上对那些函数不可见电话。

> **注意：**相同的范围规则不适用于循环（例如 `for() { }`）和条件块（例如 `if () { }`）——它们看起来非常相似，但它们不是一回事！注意不要混淆这些。

> **注意：** ReferenceError : "x" is not defined错误是您会遇到的最常见错误之一。如果你得到这个错误并且你确定你已经定义了有问题的变量，请检查它在什么范围内。

测试你的技能！

您已读完本文，但您还记得最重要的信息吗？在继续之前，您可以找到一些进一步的测试来验证您是否记住了这些信息——请参阅测试您的技能：函数。这些测试需要接下来两篇文章中介绍的技能，因此您可能希望在尝试之前先阅读这些内容。

结论

本文探索了函数背后的基本概念，为下一篇文章铺平了道路，在下一篇文章中我们将带您逐步构建您自己的自定义函数。

也可以看看

- 功能详细指南— 涵盖此处未包含的一些高级功能。
- 函数参考
- 默认参数，箭头函数——高级概念参考

此页面最后修改于 2023 年 3 月 29 日由MDN 贡献者提供。