循环代码

编程语言对于快速完成重复性任务非常有用，从多个基本计算到几乎任何其他需要完成大量类似工作的情况。在这里，我们将看看 JavaScript 中可用于处理此类需求的循环结构。

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：	了解如何在 JavaScript 中使用循环。

为什么循环有用?

循环就是一遍又一遍地做同样的事情。通常，每次循环的代码都会略有不同，或者运行相同的代码但使用不同的变量。

循环代码示例

假设我们想在一个元素上绘制 100 个随机圆圈 `<canvas>` （按*更新*按钮一次又一次地运行示例以查看不同的随机集）：

Update

下面是实现此示例的 JavaScript 代码：

```
const btn = document.querySelector('button');
const canvas = document.querySelector('canvas');
const ctx = canvas.getContext('2d');

document.addEventListener('DOMContentLoaded', () => {
  canvas.width = document.documentElement.clientWidth;
  canvas.height = document.documentElement.clientHeight;
})

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

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

```
}

btn.addEventListener('click',draw);
```

有和没有循环

您暂时不必了解所有代码，但让我们看一下实际绘制 100 个圆圈的代码部分：

```
for (let i = 0; i < 100; i++) {
  ctx.beginPath();
  ctx.fillStyle = 'rgba(255,0,0,0.5)';
  ctx.arc(random(canvas.width), random(canvas.height),
random(50), 0, 2 * Math.PI);
  ctx.fill();
}
```

- `random(x)`，在代码前面定义，返回一个介于 `0` 和之间的整数 `x-1` 。

您应该了解基本概念——我们正在使用一个循环来运行此代码的 100 次迭代，每次迭代都会在页面上的随机位置绘制一个圆圈。无论我们绘制 100 个、1000 个或 10,000 个圆圈，所需的代码量都是相同的。只有一个数字必须改变。

如果我们在这里不使用循环，我们将不得不为我们想要绘制的每个圆重复以下代码：

```
ctx.beginPath();
ctx.fillStyle = 'rgba(255,0,0,0.5)';
ctx.arc(random(canvas.width), random(canvas.height), random(50),
0, 2 * Math.PI);
ctx.fill();
```

这会变得非常乏味并且难以维护。

循环遍历一个集合

大多数情况下，当您使用循环时，您会有一个项目集合，并且想要对每个项目做一些事情。

一种类型的集合是 `Array` 我们在本课程的数组一章中遇到的。但是 JavaScript 中还有其他集合，包括 `Set` 和 `Map` 。

for...of 循环

循环遍历集合的基本工具是循环 `for...of`：

```
const cats = ['Leopard', 'Serval', 'Jaguar', 'Tiger', 'Caracal',
'Lion'];

for (const cat of cats) {
  console.log(cat);
}
```

在这个例子中， `for (const cat of cats)` 说：

1. 给定 collection `cats` ，获取集合中的第一项。
2. 将其分配给变量 `cat` ，然后运行大括号之间的代码 `{}` 。
3. 获取下一个项目，然后重复 (2)，直到到达集合的末尾。

地图 () 和过滤器 ()

JavaScript 也有更专门的集合循环，我们将在这里提到其中的两个。

您可以使用 `map()` 对集合中的每个项目执行某些操作并创建一个包含已更改项目的新集合：

```
function toUpper(string) {
  return string.toUpperCase();
}

const cats = ['Leopard', 'Serval', 'Jaguar', 'Tiger', 'Caracal',
'Lion'];

const upperCats = cats.map(toUpper);

console.log(upperCats);
// [ "LEOPARD", "SERVAL", "JAGUAR", "TIGER", "CARACAL", "LION" ]
```

这里我们将一个函数传递给 `cats.map()`，并 `map()` 为数组中的每个项目调用一次该函数，传入该项目。然后它将每个函数调用的返回值添加到一个新数组中，最后返回新数组。在这种情况下，我们提供的函数将项目转换为大写，因此生成的数组包含我们所有大写的猫：

```
[ "LEOPARD", "SERVAL", "JAGUAR", "TIGER", "CARACAL", "LION" ]
```

您可以使用 `filter()` 来测试集合中的每个项目，并创建一个仅包含匹配项的新集合：

```
function lCat(cat) {
  return cat.startsWith('L');
}

const cats = ['Leopard', 'Serval', 'Jaguar', 'Tiger', 'Caracal',
'Lion'];

const filtered = cats.filter(lCat);

console.log(filtered);
// [ "Leopard", "Lion" ]
```

这看起来很像 `map()`，除了我们传入的函数返回一个布尔值：如果它返回 `true`，则该项目包含在新数组中。我们的函数测试项目是否以字母“L”开头，因此结果是一个仅包含名称以“L”开头的猫的数组：

```
[ "Leopard", "Lion" ]
```

请注意， `map()` 和 `filter()` 都经常与*函数表达式一起使用，我们将在*函数模块中学习。使用函数表达式，我们可以重写上面的示例，使其更加紧凑：

```
const cats = ['Leopard', 'Serval', 'Jaguar', 'Tiger', 'Caracal',
'Lion'];

const filtered = cats.filter((cat) => cat.startsWith('L'));
console.log(filtered);
// [ "Leopard", "Lion" ]
```

for循环的标准

在上面的"绘制圆圈"示例中，您没有要循环的项目集合：您实际上只想运行相同的代码 100 次。在这种情况下，您应该使用循环 `for` 。这具有以下语法：

```
for (initializer; condition; final-expression) {
  // code to run
}
```

在这里我们有：

1. 关键字 `for` ，后跟一些括号。
2. 在括号内我们有三个项目，用分号分隔：
    i. 初始化**器**——这通常是一个设置为数字的变量，它会递增以计算循环运行的次数。它有时也称为**计数器变量**。
    ii. 条件——定义**循环**何时停止循环。这通常是一个具有比较运算符的表达式，用于查看是否满足退出条件的测试。
    iii. final**表达式**——每次循环完成一次完整迭代时，它总是被评估（或运行）。它通常用于增加（或在某些情况下减少）计数器变量，使其更接近条件不再为的点 `true` 。
3. 一些包含代码块的大括号——每次循环迭代时都会运行这段代码。

计算平方

让我们看一个真实的例子，这样我们就可以更清楚地想象它们的作用。

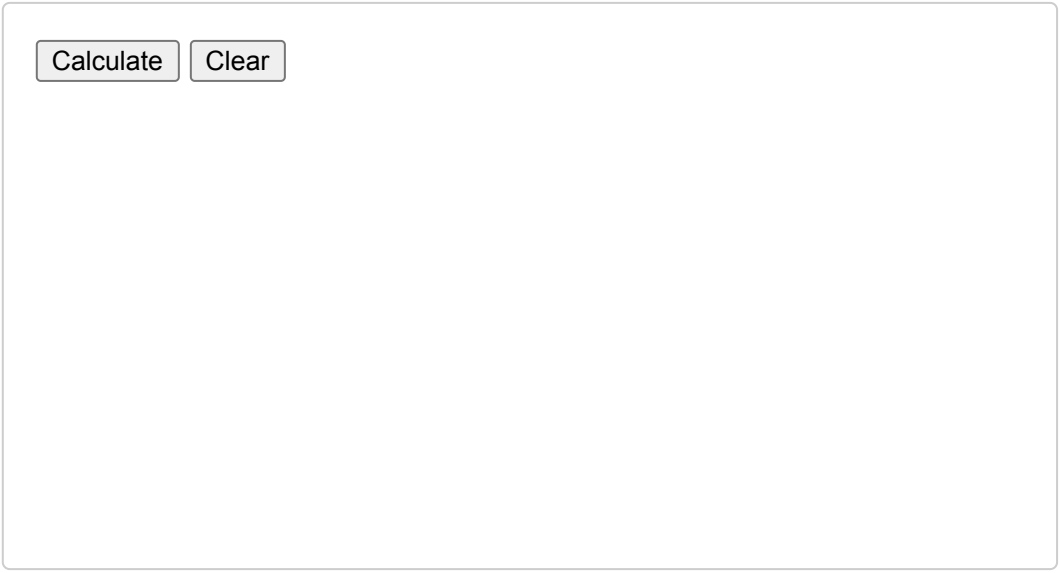
```
const results = document.querySelector('#results');

function calculate() {
  for (let i = 1; i < 10; i++) {
    const newResult = `${i} x ${i} = ${i * i}`;
    results.textContent += `${newResult}\n`;
  }
  results.textContent += '\nFinished!';
}
```

```
const calculateBtn = document.querySelector('#calculate');
const clearBtn = document.querySelector('#clear');

calculateBtn.addEventListener('click', calculate);
clearBtn.addEventListener('click', () => results.textContent =
'');
```

这为我们提供了以下输出：



此代码计算从 1 到 9 的数字的平方，并写出结果。代码的核心是 `for` 执行计算的循环。

让我们把这条线分解 `for (let i = 1; i < 10; i++)` 成三个部分：

1. `let i = 1`：计数器变量， `i` 从 开始 `1` 。请注意，我们必须使用 `let` for 计数器，因为每次循环时我们都会重新分配它。
2. `i < 10` `i`：只要小于 ， 就继续循环 `10` 。
3. `i++` `i`： 每次循环加一。

在循环内部，我们计算 的当前值的平方 `i` ，即： `i * i` 。我们创建一个字符串来表示我们所做的计算和结果，并将此字符串添加到输出文本中。我们还添加了 `\n` ，因此我们添加的下一个字符串将在新行开始。所以：

1. 在第一次运行期间， `i = 1` ，因此我们将添加 `1 x 1 = 1` 。
2. 在第二次运行期间， `i = 2` ，因此我们将添加 `2 x 2 = 4` 。

3. 等等...
4. 当 `i` 等于时， `10` 我们将停止运行循环并直接移动到循环下面的下一段代码， `Finished!` 在新行上打印出消息。

使用 for 循环遍历集合

您可以使用 `for` 循环来遍历集合，而不是 `for...of` 循环。

让我们再看看 `for...of` 上面的例子：

```
const cats = ['Leopard', 'Serval', 'Jaguar', 'Tiger', 'Caracal',
'Lion'];

for (const cat of cats) {
  console.log(cat);
}
```

我们可以像这样重写该代码：

```
const cats = ['Leopard', 'Serval', 'Jaguar', 'Tiger', 'Caracal',
'Lion'];

for (let i = 0; i < cats.length; i++) {
  console.log(cats[i]);
}
```

在此循环中，我们从 开始 `i` ， `0` 并在 `i` 达到数组长度时停止。然后在循环内，我们依次 `i` 访问数组中的每个项目。

这工作得很好，并且在 JavaScript 的早期版本中 `for...of` 不存在，所以这是遍历数组的标准方法。但是，它提供了更多将错误引入代码的机会。例如：

- 您可能会从 开始 `i` ， `1` 忘记第一个数组索引是零，而不是 1。
- 您可能会停在 `i <= cats.length` ，而忘记最后一个数组索引位于 `length - 1` 。

`for...of` 出于这样的原因，如果可以的话，通常最好使用。

有时您仍然需要使用 `for` 循环来遍历数组。例如，在下面的代码中，我们要记录一条列出我们的猫的消息：

```
const cats = ['Pete', 'Biggles', 'Jasmine'];

let myFavoriteCats = 'My cats are called ';

for (const cat of cats) {
  myFavoriteCats += `${cat}, `
}

console.log(myFavoriteCats); // "My cats are called Pete,
Biggles, Jasmine, "
```

最终输出句子的格式不是很好：

```
My cats are called Pete, Biggles, Jasmine,
```

我们希望它以不同的方式处理最后一只猫，如下所示：

```
My cats are called Pete, Biggles, and Jasmine.
```

但要做到这一点，我们需要知道何时进行最后的循环迭代，为此我们可以使用循环 `for` 并检查 的值 `i`：

```
const cats = ['Pete', 'Biggles', 'Jasmine'];

let myFavoriteCats = 'My cats are called ';

for (let i = 0; i < cats.length; i++) {
  if (i === cats.length - 1) {   // We are at the end of the
array
    myFavoriteCats += `and ${cats[i]}.`
  } else {
    myFavoriteCats += `${cats[i]}, `
  }
}
```

```
console.log(myFavoriteCats);     // "My cats are called Pete,
Biggles, and Jasmine."
```

使用 break 退出循环

如果要在所有迭代完成之前退出循环，可以使用break语句。我们在上一篇文章中查看switch 语句时已经遇到过这一点——当在与输入表达式匹配的 switch 语句中遇到 case 时，该 break 语句立即退出 switch 语句并继续执行它之后的代码。

循环也是如此——一条 break 语句将立即退出循环并使浏览器继续执行它后面的任何代码。

假设我们想要搜索一组联系人和电话号码并只返回我们想要查找的号码？首先，一些简单的 HTML——一个 <input> 允许我们输入要搜索的名称的文本，一个 <button> 用于提交搜索的元素，以及一个 <p> 用于显示结果的元素：

```
<label for="search">Search by contact name: </label>
<input id="search" type="text" />
<button>Search</button>

<p></p>
```

现在进入 JavaScript：

```
const contacts = ['Chris:2232322', 'Sarah:3453456',
'Bill:7654322', 'Mary:9998769', 'Dianne:9384975'];
const para = document.querySelector('p');
const input = document.querySelector('input');
const btn = document.querySelector('button');

btn.addEventListener('click', () => {
  const searchName = input.value.toLowerCase();
  input.value = '';
  input.focus();
  para.textContent = '';
  for (const contact of contacts) {
    const splitContact = contact.split(':');
```

```
    if (splitContact[0].toLowerCase() === searchName) {
      para.textContent = `${splitContact[0]}'s number is
${splitContact[1]}.`;
      break;
    }
  }
  if (para.textContent === '') {
   para.textContent = 'Contact not found.';
 }
});
```

Search by contact name:
Search

1. 首先，我们有一些变量定义——我们有一个联系信息数组，每个项目都是一个字符串，其中包含用冒号分隔的姓名和电话号码。
2. 接下来，我们将事件侦听器附加到按钮（ `btn` ），以便在按下按钮时运行一些代码来执行搜索并返回结果。
3. 我们将输入到文本输入中的值存储在一个名为 的变量中 `searchName` ，然后清空文本输入并再次聚焦它，为下一次搜索做好准备。 `toLowerCase()` 请注意，我们还在字符串上 运行该方法，因此搜索将不区分大小写。
4. 现在进入有趣的部分，循环 `for...of` ：
   i. 在循环内，我们首先在冒号字符处拆分当前联系人，并将生成的两个值存储在名为 的数组中 `splitContact` 。
   ii. 然后我们使用条件语句来测试 `splitContact[0]` （联系人的姓名，再次小写 `toLowerCase()` ）是否等于输入的 `searchName` 。如果是，我们在段落中输入一个字符串来报告联系人的号码，并使用它 `break` 来结束循环。
5. 循环后，我们检查是否设置了联系人，如果没有，我们将段落文本设置为“Contact not found.”。

> **注意：您也可以**在 GitHub 上 查看完整的源代码（也可以查看实时运行 ）。

使用 continue 跳过迭代

continue语句的工作方式与 类似 `break` ，但它不会完全跳出循环，而是跳到循环的下一次迭代。让我们看另一个例子，它以数字作为输入，只返回整数的平方数（整数）。

HTML 与上一个示例基本相同——一个简单的文本输入和一个用于输出的段落。

```
<label for="number">Enter number: </label>
<input id="number" type="text" />
<button>Generate integer squares</button>

<p>Output:</p>
```

JavaScript 也基本相同，尽管循环本身有点不同：

```
const para = document.querySelector('p');
const input = document.querySelector('input');
const btn = document.querySelector('button');

btn.addEventListener('click', () => {
  para.textContent = 'Output: ';
  const num = input.value;
  input.value = '';
  input.focus();
  for (let i = 1; i <= num; i++) {
    let sqRoot = Math.sqrt(i);
    if (Math.floor(sqRoot) !== sqRoot) {
      continue;
    }
    para.textContent += `${i} `;
  }
});
```

这是输出：

Enter number:
Generate integer squares

Output:

1. 在这种情况下，输入应该是一个数字（ `num` ）。循环 `for` 被赋予一个从 1 开始的计数器（因为在这种情况下我们对 0 不感兴趣），一个退出条件，表示当计数器变得大于输入时循环将停止，以及一个迭代器，每次将计数器加 `num` 1时间。
2. 在循环中，我们使用Math.sqrt(i)找到每个数字的平方根，然后通过四舍五入到最接近的整数（this是Math.floor()对它传递的数字所做的）。
3. 如果平方根和四舍五入后的平方根不相等（ `!==` ），则表示平方根不是整数，因此我们对此不感兴趣。在这种情况下，我们使用 `continue` 语句跳到下一个循环迭代而不在任何地方记录数字。
4. 如果平方根是一个整数，我们就跳过 `if` 整个块，所以 `continue` 不执行该语句； `i` 相反，我们在段落内容的末尾连接当前值和一个空格。

> **注意：您也可以**在 GitHub 上 查看完整的源代码（也可以查看实时运行 ）。

一边做...一边

`for` 不是 JavaScript 中唯一可用的循环类型。实际上还有很多其他的，虽然您现在不需要了解所有这些，但值得看看其他几个的结构，这样您就可以以略有不同的方式识别工作中的相同功能。

首先，让我们看一下while循环。该循环的语法如下所示：

```
initializer
while (condition) {
  // code to run

  final-expression
}
```

这与循环的工作方式非常相似 `for`，除了初始化变量设置在循环之前，并且 final-expression 包含在要运行的代码之后的循环内，而不是这两项包含在括号内。条件包含在括号内，前面是关键字 `while` 而不是 `for`。

相同的三个项目仍然存在，并且它们的定义顺序与它们在 for 循环中的顺序相同。这是因为您必须先定义一个初始值设定项，然后才能检查条件是否为真。final-expression 然后在循环内的代码运行（迭代已完成）之后运行，这只有在条件仍然为真时才会发生。

让我们再次看一下我们的猫列表示例，但是重写为使用 while 循环：

```
const cats = ['Pete', 'Biggles', 'Jasmine'];

let myFavoriteCats = 'My cats are called ';

let i = 0;

while (i < cats.length) {
  if (i === cats.length - 1) {
    myFavoriteCats += `and ${cats[i]}.`;
  } else {
    myFavoriteCats += `${cats[i]}, `;
  }

  i++;
}

console.log(myFavoriteCats);     // "My cats are called Pete,
Biggles, and Jasmine."
```

> **注意：**这仍然和预期的一样有效——看看它在 GitHub 上的实时运行情况（也可以查看完整的源代码）。

do ...while循环非常相似，但提供了 while 结构的变体：

```
initializer
do {
  // code to run
```

```
  final-expression
} while (condition)
```

在这种情况下，初始化器再次出现在循环开始之前。关键字直接位于包含要运行的代码和最终表达式的花括号之前。

`do...while` 循环和循环之间的主要区别 `while` 是循环*内的代码 `do...while` 总是至少执行一次*。那是因为条件出现在循环内的代码之后。所以我们总是运行该代码，然后检查是否需要再次运行它。在 `while` 和 `for` 循环中，首先检查，因此代码可能永远不会执行。

让我们再次重写我们的 cat 列表示例以使用 `do...while` 循环：

```
const cats = ['Pete', 'Biggles', 'Jasmine'];

let myFavoriteCats = 'My cats are called ';

let i = 0;

do {
  if (i === cats.length - 1) {
    myFavoriteCats += `and ${cats[i]}.`;
  } else {
    myFavoriteCats += `${cats[i]}, `;
  }

  i++;
} while (i < cats.length);

console.log(myFavoriteCats);     // "My cats are called Pete,
Biggles, and Jasmine."
```

> **注意：**同样，这与预期的一样有效——看看它在 GitHub 上的实时运行情况 （也可以查看完整的源代码 ）。

> **警告：**对于 while 和 do...while — 与所有循环一样 — 您必须确保初始化程序递增，或者根据情况递减，以便条件最终变为假。

否则，循环将永远持续下去，浏览器将强制它停止，或者它会崩溃。这称为**无限循环**。

主动学习：启动倒计时

在本练习中，我们希望您在输出框中打印一个简单的发射倒计时，从 10 倒计时到 Blastoff。具体来说，我们希望您：

- 从 10 向下循环到 0。我们为您提供了一个初始值设定项 — `let i = 10;` 。
- 对于每次迭代，创建一个新段落并将其附加到 `<div>` 我们使用选择的输出中 `const output = document.querySelector('.output');` 。在注释中，我们为您提供了需要在循环内某处使用的三个代码行：
  - `const para = document.createElement('p');` — 创建一个新段落。
  - `output.appendChild(para);` — 将段落附加到输出中 `<div>` 。
  - `para.textContent =` — 使段落内的文本等于您放在右侧等号后的任何内容。
- 不同的迭代次数需要在该迭代的段落中放置不同的文本（您需要一个条件语句和多行 `para.textContent =` ）：
  - 如果数字是 10，则在段落中打印“倒计时 10”。
  - 如果数字为 0，则打印“Blast off!” 到段落。
  - 对于任何其他数字，只打印该段落的数字。
- 记得包含一个迭代器！然而，在这个例子中，我们在每次迭代后倒计时，而不是向上倒计时，所以你不想——— `i++` 你如何向下迭代？

注意：如果你开始输入循环（例如 (while(i>=0)），浏览器可能会卡住，因为你还没有输入结束条件。所以要小心。你可以开始写你的代码评论来处理这个问题，完成后删除评论。

如果您犯了错误，您可以随时使用“重置”按钮重置示例。如果您真的卡住了，请按“显示解决方案”以查看解决方案。

```
html {
  font-family: sans-serif;
}

h2 {
  font-size: 16px;
}

.a11y-label {
  margin: 0;
  text-align: right;
  font-size: 0.7rem;
  width: 98%;
}

body {
  margin: 10px;
  background: #f5f9fa;
}
```

Live output

Editable code

Press Esc to move focus away from the code area (Tab inserts a tab character).

```
let output =
document.querySelector('.output');
output.innerHTML = '';

// let i = 10;

// const para =
document.createElement('p');
// para.textContent = ;
// output.appendChild(para);
```

Reset Show solution

主动学习：填写宾客名单

在本练习中，我们希望您将存储在数组中的姓名列表放入客人列表中。但这并不那么容易——我们不想让 Phil 和 Lola 进来，因为他们既贪婪又粗鲁，而且总是吃光所有的食物！我们有两份名单，一份让客人承认，一份让客人拒绝。

具体来说，我们希望您：

- 编写一个循环遍历数组 `people` 。
- 在每次循环迭代期间，使用条件语句检查当前数组项是否等于“Phil”或“Lola”：
  - `refused` 如果是，则将数组项连接到段落的末尾 `textContent` ，后跟一个逗号和一个空格。
  - 如果不是，则将数组项连接到 `admitted` 段落的末尾 `textContent` ，后跟一个逗号和一个空格。

我们已经为您提供：

- `refused.textContent +=` — 一行的开头将连接 `refused.textContent` .
- `admitted.textContent +=` — 一行的开头将连接 `admitted.textContent` .

额外的奖励问题——成功完成上述任务后，你会得到两个名字列表，用逗号分隔，但它们会不整齐——每个列表的末尾都有一个逗号。你能想出如何写出在每种情况下切掉最后一个逗号并在末尾添加句号的行吗？查看有用的字符串方法一文以获得帮助。

如果您犯了错误，您可以随时使用“重置”按钮重置示例。如果您真的卡住了，请按“显示解决方案”以查看解决方案。

mdn web docs_

Live output

Admit:

Refuse:

Editable code

Press Esc to move focus away from the code area (Tab inserts a tab character).

```
const people = ['Chris', 'Anne', 'Colin',
'Terri', 'Phil', 'Lola', 'Sam', 'Kay',
'Bruce'];

const admitted =
document.querySelector('.admitted');
const refused =
document.querySelector('.refused');
admitted.textContent = 'Admit: ';
refused.textContent = 'Refuse: ';

// loop starts here

// refused.textContent += ;
// admitted.textContent += ;
```

Reset Show solution

您应该使用哪种循环类型?

如果您正在遍历数组或其他支持它的对象，并且不需要访问每个项目的索引位置，那么 `for...of` 是最佳选择。它更容易阅读，而且出错的可能性也更少。

对于其他用途， `for` 、 `while` 和 `do...while` 循环在很大程度上是可以互换的。它们都可以用来解决相同的问题，您使用哪一个在很大程度上取决于您的个人喜好——您觉得哪一个最容易记住或最直观。我们建议 `for` 至少从一开始就使用 ， 因为它可能最容易记住所有内容 — 初始化程序、条件和最

终表达式都必须整齐地放在括号中，因此很容易看到它们的位置并检查你没有想念他们。

让我们再看看它们。

第一 `for...of`：

```
for (const item of array) {
  // code to run
}
```

`for`：

```
for (initializer; condition; final-expression) {
  // code to run
}
```

`while`：

```
initializer
while (condition) {
  // code to run

  final-expression
}
```

最后 `do...while`：

```
initializer
do {
  // code to run

  final-expression
} while (condition)
```

> **注意：**还有其他循环类型/功能，它们在高级/专业情况下很有用，超出了本文的范围。如果您想进一步学习循环，请阅读我们

> 的高级循环和迭代指南。

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：循环。

结论

本文向您揭示了背后的基本概念，以及在 JavaScript 中循环代码时可用的不同选项。您现在应该清楚为什么循环是处理重复代码的好机制并且很少在您自己的示例中使用它们！

如果您有任何不明白的地方，请随时重新阅读文章，或联系我们寻求帮助。

也可以看看

- 循环和迭代的详细信息
- 供……参考
- 声明参考
- while和do...while引用
- 中断并继续引用

此页面最后修改于 2023 年 2 月 23 日由MDN 贡献者提供。