第一次接触 JavaScript

现在你已经了解了一些关于 JavaScript 的理论以及你可以用它做什么，我们将通过指导你完成一个实用教程，让你了解创建一个简单的 JavaScript 程序的过程是什么样的。在这里，您将逐步构建一个简单的“猜数字”游戏。

先决条件：	基本的计算机知识，对 HTML 和 CSS 的基本理解，对 JavaScript 是什么的理解。
客观的：	拥有编写一些 JavaScript 的初步经验，并至少对编写 JavaScript 程序涉及的内容有基本的了解。

我们想在这里设定非常明确的期望：您不需要在本文结束时学习 JavaScript，甚至不需要理解我们要求您编写的所有代码。相反，我们想让您了解 JavaScript 的功能如何协同工作，以及编写 JavaScript 的感觉。在随后的文章中，您将更详细地回顾此处显示的所有功能，所以如果您不能立即理解它，请不要担心！

> **注意：** 您将在 JavaScript 中看到的许多代码功能与其他编程语言中的相同——函数、循环等。代码语法看起来不同，但概念在很大程度上仍然相同。

像程序员一样思考

编程中最难学的东西之一不是你需要学习的语法，而是如何应用它来解决现实世界的问题。你需要开始像程序员一样思考——这通常包括查看你的程序需要做什么的描述，找出实现这些事情需要哪些代码特性，以及如何让它们协同工作。

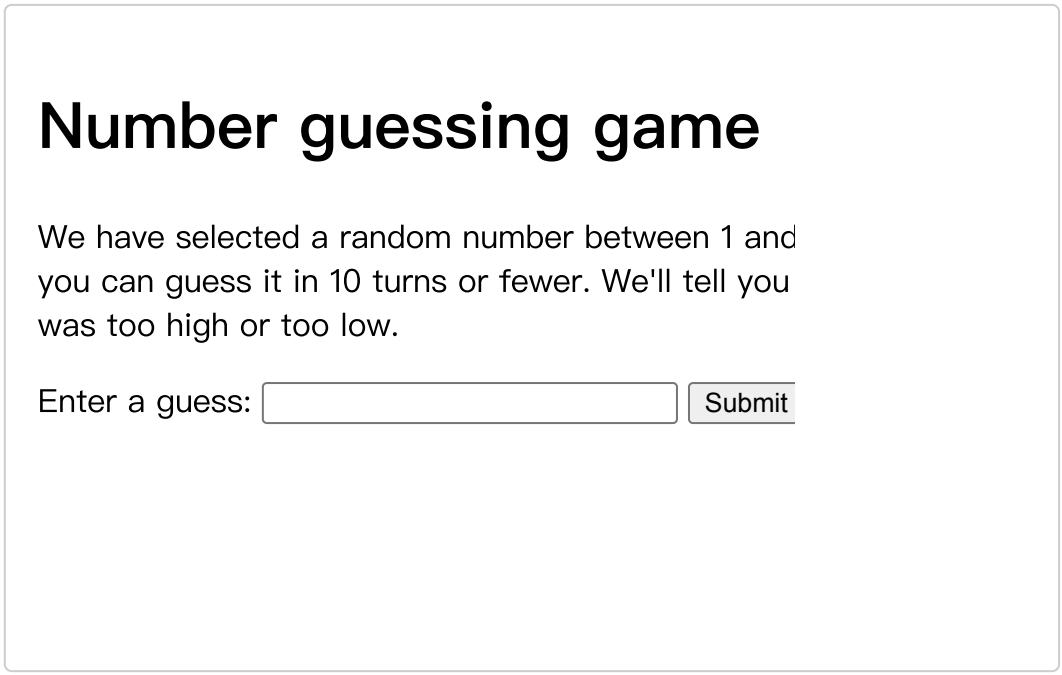
这需要辛勤工作、编程语法经验和实践的结合——再加上一点创造力。你编码的越多，你就会越好。我们不能保证你会在五分钟内开发出“程序员的大

脑”，但我们会在整个课程中给你大量的机会来练习像程序员一样思考。

考虑到这一点，让我们看看我们将在本文中构建的示例，并回顾将其分解为具体任务的一般过程。

例子——猜数字游戏

在本文中，我们将向您展示如何构建如下所示的简单游戏：



试一试——在继续之前熟悉游戏。

假设您的老板为您提供了以下关于创建此游戏的简介：

> 我希望您创建一个简单的猜数字类型的游戏。它应该在 1 到 100 之间选择一个随机数，然后挑战玩家在 10 回合内猜出这个数字。每轮之后，玩家应该被告知他们是对还是错，如果他们错了，猜测是太低还是太高。它还应该告诉玩家他们之前猜到的数字。一旦玩家猜对了，或者他们用完了回合，游戏就会结束。当游戏结束时，玩家应该可以选择重新开始游戏。

看完这份简报，我们可以做的第一件事就是开始将其分解为简单的可操作任务，尽可能多地以程序员的思维方式：

1. 生成 1 到 100 之间的随机数。
2. 记录玩家所在的回合数。从 1 开始。
3. 为玩家提供一种猜测数字的方法。
4. 一旦提交了猜测，首先将其记录在某个地方，以便用户可以看到他们之前的猜测。
5. 接下来，检查它是否是正确的数字。
6. 如果正确：
    i. 显示祝贺信息。
    ii. 阻止玩家输入更多的猜测（这会使游戏变得混乱）。
    iii. 允许玩家重新开始游戏的显示控件。
7. 如果错误并且玩家左转：
    i. 告诉玩家他们错了，以及他们的猜测是太高还是太低。
    ii. 让他们输入另一个猜测。
    iii. 将轮数增加 1。
8. 如果错误并且玩家没有左转：
    i. 告诉玩家游戏结束。
    ii. 阻止玩家输入更多的猜测（这会使游戏变得混乱）。
    iii. 允许玩家重新开始游戏的显示控件。
9. 游戏重新启动后，确保游戏逻辑和 UI 已完全重置，然后返回步骤 1。

现在让我们继续前进，看看我们如何将这些步骤转化为代码，构建示例，并在我们进行的过程中探索 JavaScript 功能。

初始设置

在开始本教程之前，我们希望您制作number-guessing-game-start.html 文件的本地副本（在此处查看 ）。在文本编辑器和网络浏览器中打开它。目前您会看到一个简单的标题、一段说明和用于输入猜测的表格，但该表格目前不会执行任何操作。

我们将添加所有代码的地方是在 <script> HTML 底部的元素内：

```
<script>
  // Your JavaScript goes here
</script>
```

添加变量来存储我们的数据

让我们开始吧。首先，在您的 <script> 元素中添加以下行：

```
let randomNumber = Math.floor(Math.random() * 100) + 1;

const guesses = document.querySelector('.guesses');
const lastResult = document.querySelector('.lastResult');
const lowOrHi = document.querySelector('.lowOrHi');

const guessSubmit = document.querySelector('.guessSubmit');
const guessField = document.querySelector('.guessField');

let guessCount = 1;
let resetButton;
```

这部分代码设置了我们存储程序将使用的数据所需的变量和常量。

变量基本上是值的名称（例如数字或文本字符串）。您使用关键字创建变量， `let` 后跟变量名称。

常量也用于命名值，但与变量不同的是，值一旦设置就无法更改。在这种情况下，我们使用常量来存储对部分用户界面的引用。其中一些元素中的文本可能会发生变化，但每个常量始终引用与它初始化时相同的 HTML 元素。您可以使用关键字 `const` 后跟常量名称来创建常量。

您可以使用等号（ `=`）后跟您要赋予的值来为变量或常量赋值。

在我们的例子中：

- 第一个变量 — `randomNumber` — 被分配一个介于 1 和 100 之间的随机数，使用数学算法计算得出。

- 前三个常量分别用于在我们的 HTML 中存储对结果段落的引用，并用于稍后在代码中将值插入段落（注意它们如何在元素内，该元素本身用于选择所有 `<div>` 三个稍后进行重置，当我们重新启动游戏时）：

```
<div class="resultParas">
  <p class="guesses"></p>
  <p class="lastResult"></p>
  <p class="lowOrHi"></p>
</div>
```

- 接下来的两个常量存储对表单文本输入和提交按钮的引用，用于控制稍后提交猜测。

```
<label for="guessField">Enter a guess: </label>
<input type="number" id="guessField" class="guessField" />
<input type="submit" value="Submit guess"
class="guessSubmit" />
```

- 我们的最后两个变量存储了一个猜测计数 1（用于跟踪玩家的猜测次数），以及对一个尚不存在（但稍后会存在）的重置按钮的引用。

> **注意：**从下一篇文章开始，您将在本课程的后面学到更多关于变量和常量的知识。

功能

接下来，在您之前的 JavaScript 下方添加以下内容：

```
function checkGuess() {
  alert('I am a placeholder');
}
```

函数是可重复使用的代码块，您可以编写一次并一次又一次地运行，从而避免了一直重复代码的需要。这真的很有用。定义函数的方法有很多种，但现在我们只关注一种简单的类型。在这里，我们通过使用关键字定义了一个函数 `function`，后跟名称，并在其后加上括号。之后，我们放上两个大括号（`{ }`）。花括号内是我们调用该函数时要运行的所有代码。

当我们想要运行代码时，我们键入函数的名称，后跟括号。

让我们现在试试看。保存代码并在浏览器中刷新页面。然后进入开发人员工具 JavaScript 控制台，并输入以下行：

```
checkGuess();
```

按`Return`/后`Enter`，您应该会看到一条警告消息`I am a placeholder`：我们在我们的代码中定义了一个函数，每当我们调用它时都会创建一个警报。

> **注意：** 您将在本课程的后面学到更多关于函数的知识。

运营商

JavaScript 运算符允许我们执行测试、做数学运算、将字符串连接在一起，以及其他类似的事情。

如果您还没有这样做，请保存您的代码，在浏览器中刷新页面，然后打开开发人员工具 JavaScript 控制台。然后我们可以尝试输入下面显示的示例 — 完全按照所示输入“示例”列中的每个示例，在每个示例后按`Return`/，然后查看它们返回的结果。`Enter`

首先让我们看一下算术运算符，例如：

操作员	姓名	例子
`+`	添加	`6 + 9`
`-`	减法	`20 - 15`
`*`	乘法	`3 * 7`
`/`	分配	`10 / 5`

您还可以使用 + 运算符将文本字符串连接在一起（在编程中，这称为*连接*）。尝试输入以下行，一次一个：

```
const name = 'Bingo';
name;
const hello = ' says hello!';
hello;
const greeting = name + hello;
greeting;
```

还有一些快捷运算符可用，称为扩充赋值运算符。例如，如果您想向现有文本字符串添加一个新文本字符串并返回结果，您可以这样做：

```
let name1 = 'Bingo';
name1 += ' says hello!';
```

这相当于

```
let name2 = 'Bingo';
name2 = name2 + ' says hello!';
```

当我们运行真/假测试时（例如在条件语句中——见下文）我们使用比较运算符。例如：

操作员	姓名	例子

===	严格相等（是否完全相同？）	`5 === 2 + 4 // false`
		`'Chris' === 'Bob' // false`
		`5 === 2 + 3 // true`
		`2 === '2' // false; number versus string`

!==	不平等（不一样吗？）	`5 !== 2 + 4 // true`
		`'Chris' !== 'Bob' // true`
		`5 !== 2 + 3 // false`
		`2 !== '2' // true; number versus string`

<	少于	`6 < 10 // true` `20 < 10 // false`
>	比...更棒	`6 > 10 // false` `20 > 10 // true`

条件句

回到我们的 `checkGuess()` 函数，我认为可以肯定地说我们不希望它只是吐出一个占位符消息。我们希望它检查玩家的猜测是否正确，并做出适当的回应。

此时， `checkGuess()` 用这个版本替换你当前的函数：

```
function checkGuess() {
  const userGuess = Number(guessField.value);
  if (guessCount === 1) {
    guesses.textContent = 'Previous guesses: ';
  }
  guesses.textContent += `${userGuess} `;

  if (userGuess === randomNumber) {
    lastResult.textContent = 'Congratulations! You got it
right!';
    lastResult.style.backgroundColor = 'green';
    lowOrHi.textContent = '';
    setGameOver();
```

```
  } else if (guessCount === 10) {
    lastResult.textContent = '!!!GAME OVER!!!';
    lowOrHi.textContent = '';
    setGameOver();
  } else {
    lastResult.textContent = 'Wrong!';
    lastResult.style.backgroundColor = 'red';
    if (userGuess < randomNumber) {
      lowOrHi.textContent = 'Last guess was too low!';
    } else if (userGuess > randomNumber) {
      lowOrHi.textContent = 'Last guess was too high!';
    }
  }

  guessCount++;
  guessField.value = '';
  guessField.focus();
}
```

这是很多代码——呸！让我们浏览每个部分并解释它的作用。

- 第一行声明一个名为 call 的变量 `userGuess` ，并将其值设置为在文本字段中输入的当前值。我们还通过内置 `Number()` 构造函数运行此值，只是为了确保该值绝对是一个数字。由于我们不更改此变量，因此我们将使用 `const` .
- 接下来，我们遇到第一个条件代码块。条件代码块允许您有选择地运行代码，具体取决于特定条件是否为真。它看起来有点像一个函数，但它不是。最简单的条件块形式以关键字开始 `if` ，然后是一些圆括号，然后是一些大括号。在括号内，我们包含一个测试。如果测试返回 `true` ，我们将运行大括号内的代码。如果没有，我们不这样做，然后继续下一段代码。在这种情况下，测试正在测试 `guessCount` 变量是否等于 `1` （即这是否是玩家的第一次尝试）：

  ```
  guessCount === 1
  ```

  如果是，我们让猜测段落的文本内容等于 `Previous guesses:` 。如果没有，我们就不会。
- 第 6 行将当前 `userGuess` 值附加到段落的末尾 `guesses` ，加上一个空格，以便在显示的每个猜测之间有一个空格。
- 下一个块做一些检查：

  - 第一个 `if (){ }` 检查用户的猜测是否等于 `randomNumber` 我们 JavaScript 顶部的设置。如果是，则玩家猜对了游戏获胜，因此我们向玩家显示一条带有漂亮绿色的祝贺消息，清除低/高猜测信息框的内容，并运行一个 `setGameOver()` 名为稍后再讨论。
  - 现在我们已经使用结构将另一个测试链接到最后一个测试的末尾 `else if (){ }` 。这个检查这个回合是否是用户的最后一个回合。如果是，程序将执行与前一个块相同的操作，除了使用游戏结束消息而不是祝贺消息。
  - 链接到此代码末尾的最后一个块（the `else { }`）包含仅在其他两个测试均未返回 true 时运行的代码（即玩家没有猜对，但他们还有更多猜测）。在这种情况下，我们告诉他们他们错了，然后我们执行另一个条件测试来检查猜测是高于还是低于答案，并根据需要显示进一步的消息来告诉他们更高或更低。
- 函数中的最后三行（上面的第 26-28 行）让我们准备好提交下一个猜测。我们将变量加 1， `guessCount` 以便玩家用完他们的轮次（ `++` 这是一个递增操作——递增 1），然后清空表单文本字段中的值并再次聚焦它，为输入下一个猜测做好准备。

事件

在这一点上，我们有一个很好地实现的 `checkGuess()` 函数，但它不会做任何事情，因为我们还没有调用它。理想情况下，我们希望在按下“提交猜测”按钮时调用它，为此我们需要使用一个**事件**。事件是浏览器中发生的事情——单击按钮、加载页面、播放视频等——作为响应，我们可以运行代码块。**事件侦听器**观察特定事件并调用**事件处理程序**，事件处理程序是响应事件触发而运行的代码块。

在您的函数下方添加以下行 `checkGuess()`：

```
guessSubmit.addEventListener('click', checkGuess);
```

在这里，我们向按钮添加了一个事件侦听器 `guessSubmit` 。这是一个采用两个输入值（称为*参数*）的方法——我们正在侦听的事件类型（在本例中 `click` ）作为字符串，以及我们希望在事件发生时运行的代码（在本例中为函数 `checkGuess()` ）。注意我们在里面写的时候不需要指定括号 `addEventListener()` 。

现在尝试保存并刷新您的代码，您的示例应该可以工作 — 在一定程度上。现在唯一的问题是，如果您猜对了答案或猜完了，游戏就会中断，因为我们还没有定义 `setGameOver()` 游戏结束后应该运行的函数。现在让我们添加缺少的代码并完成示例功能。

完成游戏功能

让我们将该 `setGameOver()` 函数添加到我们代码的底部，然后遍历它。现在将其添加到其余 JavaScript 的下方：

```
function setGameOver() {
  guessField.disabled = true;
  guessSubmit.disabled = true;
  resetButton = document.createElement('button');
  resetButton.textContent = 'Start new game';
  document.body.append(resetButton);
  resetButton.addEventListener('click', resetGame);
}
```

- 前两行通过将它们的禁用属性设置为来禁用表单文本输入和按钮 `true` 。这是必要的，因为如果我们不这样做，用户可能会在游戏结束后提交更多的猜测，这会把事情搞砸。
- 接下来的三行生成一个新 `<button>` 元素，将其文本标签设置为“开始新游戏”，并将其添加到我们现有 HTML 的底部。
- 最后一行在我们的新按钮上设置一个事件侦听器，以便在单击它时 `resetGame()` 运行一个调用的函数。

现在我们也需要定义这个函数！再次将以下代码添加到 JavaScript 的底部：

```
function resetGame() {
  guessCount = 1;

  const resetParas = document.querySelectorAll('.resultParas
p');
  for (const resetPara of resetParas) {
    resetPara.textContent = '';
  }
```

```
  resetButton.parentNode.removeChild(resetButton);

  guessField.disabled = false;
  guessSubmit.disabled = false;
  guessField.value = '';
  guessField.focus();

  lastResult.style.backgroundColor = 'white';

  randomNumber = Math.floor(Math.random() * 100) + 1;
}
```

这个相当长的代码块将所有内容完全重置为游戏开始时的状态，因此玩家可以再试一次。它：

- 将 `guessCount` back 降为 1。
- 清空信息段落中的所有文本。我们选择里面的所有段落 `<div class="resultParas"></div>`，然后遍历每个段落，将它们设置 `textContent` 为 `''`（一个空字符串）。
- 从我们的代码中删除重置按钮。
- 启用表单元素，清空并聚焦文本字段，为输入新的猜测做好准备。
- 从段落中删除背景颜色 `lastResult`。
- 生成一个新的随机数，这样您就不会再次猜测同一个数字！

在这一点上，您应该有一个完全工作的（简单的）游戏——恭喜！

在本文中，我们现在要做的就是讨论您已经看到的其他一些重要的代码功能，尽管您可能没有意识到。

循环

我们需要更详细地了解上面代码的一部分是for...of循环。循环是编程中一个非常重要的概念，它可以让你一遍又一遍地运行一段代码，直到满足某个条件。

首先，再次转到您的浏览器开发人员工具 JavaScript 控制台，并输入以下内容：

```
const fruits = ['apples', 'bananas', 'cherries'];
for (const fruit of fruits) {
  console.log(fruit);
}
```

发生了什么？字符串 `'apples', 'bananas', 'cherries'` 在您的控制台中打印出来。

这是因为循环。该行 `const fruits = ['apples', 'bananas', 'cherries'];` 创建一个数组。我们将在本模块稍后部分完成完整的数组指南，但现在：数组是项目的集合（在本例中为字符串）。

循环 `for...of` 为您提供了一种获取数组中的每个项目并在其上运行一些 JavaScript 的方法。该行 `for (const fruit of fruits)` 说：

1. 获取中的第一项 `fruits` 。
2. 将变量设置 `fruit` 为该项目，然后运行括号之间的代码 `{}` 。
3. 获取 中的下一项 `fruits` ，并重复 2，直到到达 的末尾 `fruits` 。

在这种情况下，括号内的代码正在写出 `fruit` 到控制台。

现在让我们看一下猜数游戏中的循环——可以在函数内部找到以下内容 `resetGame()` ：

```
const resetParas = document.querySelectorAll('.resultParas p');
for (const resetPara of resetParas) {
  resetPara.textContent = '';
}
```

`<div class="resultParas">` 此代码使用该方法创建一个包含所有段落列表的变量 `querySelectorAll()` ，然后遍历每个段落，删除每个段落的文本内容。

请注意，即使 `resetPara` 是常量，我们也可以更改其内部属性，例如 `textContent` .

关于对象的小讨论

在我们开始讨论之前，让我们再添加一个最后的改进。 `let resetButton;` 在 JavaScript 顶部附近的行下方添加以下行，然后保存文件：

```
guessField.focus();
```

此行使用该方法在页面加载后 `focus()` 立即将文本光标自动放入文本字段，这意味着用户可以立即开始输入他们的第一个猜测，而无需先单击表单字段。 `<input>` 这只是一个小的添加，但它提高了可用性——给用户一个很好的视觉线索，让他们知道他们必须做什么才能玩游戏。

让我们更详细地分析一下这里发生了什么。在 JavaScript 中，您将在代码中操作的大多数项目都是对象。对象是存储在单个分组中的相关功能的集合。您可以创建自己的对象，但这是非常高级的，我们将在本课程的后面部分进行介绍。现在，我们将简要讨论浏览器包含的内置对象，它们可以让您做很多有用的事情。

以下行：

```
const guessField = document.querySelector('.guessField');
```

为了获得这个引用，我们使用了对象 `querySelector()` 的方法 `document` 。 `querySelector()` 获取一条信息——一个CSS 选择器，用于选择您想要引用的元素。

因为 `guessField` 现在包含对元素的引用 `<input>` ，所以它现在可以访问许多属性（基本上是存储在对象内部的变量，其中一些不能更改其值）和方法（基本上是存储在对象内部的函数）。输入元素可用的一种方法是 `focus()` ，所以我们现在可以使用这一行来聚焦文本输入：

```
guessField.focus();
```

不包含对表单元素的引用的变量将无法 `focus()` 使用。例如， `guesses` 常量包含对元素的引用 `<p>` ， 而 `guessCount` 变量包含一个数字。

玩转浏览器对象

让我们稍微玩一下浏览器对象。

1. 首先，在浏览器中打开您的程序。
2. 接下来，打开您的浏览器开发人员工具，并确保 JavaScript 控制台选项卡已打开。
3. `guessField` 在控制台中键入，控制台会显示该变量包含一个 `<input>` 元素。您还会注意到控制台会自动完成执行环境中存在的对象的名称，包括您的变量！
4. 现在输入以下内容：

   ```
   guessField.value = 2;
   ```

   该 `value` 属性表示输入到文本字段中的当前值。您会看到，通过输入此命令，我们已经更改了文本字段中的文本！
5. 现在尝试 `guesses` 在控制台中输入并按回车键。控制台显示该变量包含一个 `<p>` 元素。
6. 现在尝试输入以下行：

   ```
   guesses.value
   ```

   浏览器返回 `undefined` ， 因为段落没有该 `value` 属性。
7. 要更改段落内的文本，您需要该 `textContent` 属性。尝试这个：

   ```
   guesses.textContent = 'Where is my paragraph?';
   ```

8. 现在来一些有趣的东西。尝试逐行输入以下几行：

   ```
   guesses.style.backgroundColor = 'yellow';
   guesses.style.fontSize = '200%';
   guesses.style.padding = '10px';
   guesses.style.boxShadow = '3px 3px 6px black';
   ```

页面上的每个元素都有一个 `style` 属性，该属性本身包含一个对象，该对象的属性包含应用于该元素的所有内联 CSS 样式。这允许我们使用 JavaScript 在元素上动态设置新的 CSS 样式。

暂时完成...

这就是构建示例的全部内容。你走到了尽头——干得好！试用您的最终代码，或在此处试用我们的最终版本 。如果您无法使该示例运行，请对照源代码 检查它。

此页面最后修改于 2023 年 2 月 26 日由MDN 贡献者提供。