## while 和 do-while 语句

### 循环结构

我们将开始接触 JavaScript 中的**循环结构**。顾名思义，循环结构能够在一定条件下循环执行同一段代码，这个过程称之为*迭代*。我们可以使用 *while 语句*来实现循环结构。

while 语句看起来像这样：

while (条件) {  
 执行语句  
}

像 if 语句的条件一样， while 语句的条件是一个表达式，称为*条件*，写在一对小括号中。大括号连同其中的执行语句被称为*循环体*。如果执行语句只有一行，也可以省略包裹它的大括号。

* 对条件进行求值。
* 如果得到的值被看做 true（条件成立），那么执行一次循环。否则不执行。
* 循环体执行完毕后，再次对条件进行求值，如果成立则再次循环执行，否则停止。

用一个简单的示例来演示 while 语句的用法：

let n = 0;  
let x = 0;  
  
while (n < 3) {  
 n += 1;  
 x += n;  
}  
  
alert(n); // 3  
alert(x); // 6

在每次循环中，n 都会自增 1，然后再把 n 加到 x 上。因此，在每轮循环结束后，x 和 n 的值分别是：

* 第一轮后：n = 1，x = 1
* 第二轮后：n = 2，x = 3
* 第三轮后：n = 3，x = 6

当完成第三轮循环后，条件 n< 3 的值不再为真，因此循环终止。

现在我们要考虑用 while 语句进行一些实际应用。我们首先回忆一下高斯小时候那个著名的故事——你一定耳熟能详，对吧？

高斯上小学时，有一天老师出了一道算术难题：计算 1＋2＋3＋……＋100 。这下可难倒了刚学数学的小朋友们，他们按照题目的要求，正把数字一个一个地相加．可这时，却传来了高斯的声音：“老师，我已经算好了！”老师很吃惊，高斯解释道：因为1＋100＝101，2＋99＝101，3＋98＝101，……，49＋52＝101，50＋51＝101，而像这样的等于101的组合一共有50组，所以答案很快就可以求出：101×50＝5050

哦，我们不是高斯，所以我们可以使用 while 语句来进行循环计算，这样就解决了其他小朋友们的痛点。我们要干的事情就像任何一个不懂计算技巧的普通的小朋友那样：

1. 使用一个变量 n，存放初始值 0 。现在什么也没有，计算还没开始。
2. 使用另一个变量 i，用来计算每次应该被加上的值。i 每增加 1，n 就加上 i，直到 i 等于 100。
3. 现在 n 就包含了我们累加的值。

累加的过程使用 while 语句来处理，写成这样：

let n = 0;  
let i = 0;  
while (i < 100) {  
 i += 1;  
 n += i;  
}  
  
alert(n); // 5050

上述代码演示了 while 语句的基本应用。使用循环结构，可以解决我们手工计算的一些常见痛点，避免带来冗余。

实际上，你也可以将这样简单有效的运算推广到更大的范围，例如从 1 加到 10000，或 i 增加的值换成其他。当然，计算量越大，等待计算完成的时间也就越长，如果数字大小超过了 Number.MAX\_VALUE，就会溢出。（还记得第三章中的相关知识吗？:-D）

一个更常见的需求是*阶乘*。在数学上，n 的阶乘是指 1 × 2 × 3 × ... × n 这样的计算过程，其符号是 n! 。

我们已经有了从 1 累加到 100 的经验，而实现阶乘，看起来只是把加法运算改为乘法运算。

let n = 1;  
let i = 1;  
while (i < 5) {  
 i += 1;  
 n \*= i;  
}  
  
alert(n); // 120

这段代码实现了 5 的阶乘。i 作为*计数器*，依然是每次加上 1 ，而作为结果的 n 每次循环中乘以 i ，并作为新值。我们可以将它的步骤展开，以更清楚地观察运行过程：

1. n = 1, i = 1
2. i = 2, n = n \* i = 1 \* 2 = 2
3. i = 3, n = n \* i = 2 \* 3 = 6
4. i = 4, n = n \* i = 6 \* 4 = 24
5. i = 5, n = n \* i = 24 \* 5 = 120

我们可以从用户那里得到一个数字，并求出它的阶乘值。

let n = 1;  
let i = 1;  
let value = parseInt(prompt("请输入一个整数："));  
while (i < value) {  
 i += 1;  
 n \*= i;  
}  
  
if (!isFinite(n)) {  
 alert("数字太大了！");  
} else {  
 alert(value + "的阶乘是：" + n);  
}

我们可以输入一些数字来进行测试。如果我们输入的数字的阶乘值太大，超出了 JavaScript 的表示范围（得到 Infinity），那么我们会得到一个贴心的提示。或者，得到这个阶乘值（或其约数）。看起来一切正常。

但是！当我们输入负数呢？如果我们输入的内容无法解析为整数以至于得到 NaN 呢？我们会得出错误的结果。

value = 0; // 1，正确  
value = -1; // 1，错误  
value = -2; // 1，错误  
value = "Hello"; // 1，错误

关于“负数是否具有阶乘”等数学概念不在这里讨论范围内，我们应当设立明确的界限，对得到的值进行检查。如果它不符合要求，就通知用户，并不进行后续计算。

let n = 1;  
let i = 1;  
let value = parseInt(prompt("请输入一个正整数："));  
  
if (isNaN(value) || value < 0) {  
 alert("无法进行计算！")  
} else {  
   
 while (i < value) {  
 i += 1;  
 n \*= i;  
 }  
  
 if (!isFinite(n)) {  
 alert("数字太大了！");  
 } else {  
 alert(value + "的阶乘是：" + n);  
 }  
}

这样，我们可以保证：只有当得到一个正确的值的时候，才会进行计算。

我们还可以开动脑筋，将这个程序赋予更多创意：

let n = 1;  
let i = 1;  
let value = parseInt(prompt("请输入一个正整数："));  
  
while (!isNaN(value) && value >= 0) {   
 while (i < value) {  
 i += 1;  
 n \*= i;  
 }  
  
 if (!isFinite(n)) {  
 alert("数字太大了！");  
 } else {  
 alert(value + "的阶乘是：" + n);  
 }  
 value = parseInt(prompt("请输入一个正整数："));  
}

这个程序将我们已经学习的诸多概念融合在了一起，如果你一时没看明白这个程序究竟在做什么，可以多花点时间仔细看一看。这里解释一下这个程序所干的事情：

1. 得到一个用户输入的值（value）。
2. 如果这个值符合我们的要求，就开始执行后续过程。
3. 进行常规阶乘计算。
4. 通知用户关于阶乘计算的情况。
5. 再次要求用户输入一个值。
6. 回到步骤 2。

这个程序演示了 while 语句作为控制结构的一般应用。它控制了整个程序的运行流程，这类流程被称为*控制流*。我们将在本章后续了解到如何进一步完善控制流。在这个程序中，当你输入一个不符合要求的值时，控制流便会终止。当我们学习*异常处理*的概念之后，我们可以使用更加优雅的方式来处理异常和终止的情况。

### do-while 语句

while 语句有一种变体，称为 *do-while 语句*。它的形式如下：

do {  
 执行语句  
} while (条件)

与通常的 while 语句不同的是，它的循环体写在了 do 关键字后面，而循环条件则放在了循环体的后面。

在第一次查看条件之前，do-while 语句无论如何都会先执行循环体，然后再查看条件，判断是否进行下一次循环。除此以外与通常的 while 语句是完全等价的。

let n = 1;  
let i = 1;  
do {  
 i += 1;  
 n \*= i;  
} while (n < 5)  
  
alert(n); // 120

do-while 语句适用于需要先执行一遍，再进行条件判断的情况。