

30 | JavaScript语法（二）：你知道哪些JavaScript语句？

winter 2019-04-02



00:00

23:33

讲述：winter 大小：21.58M

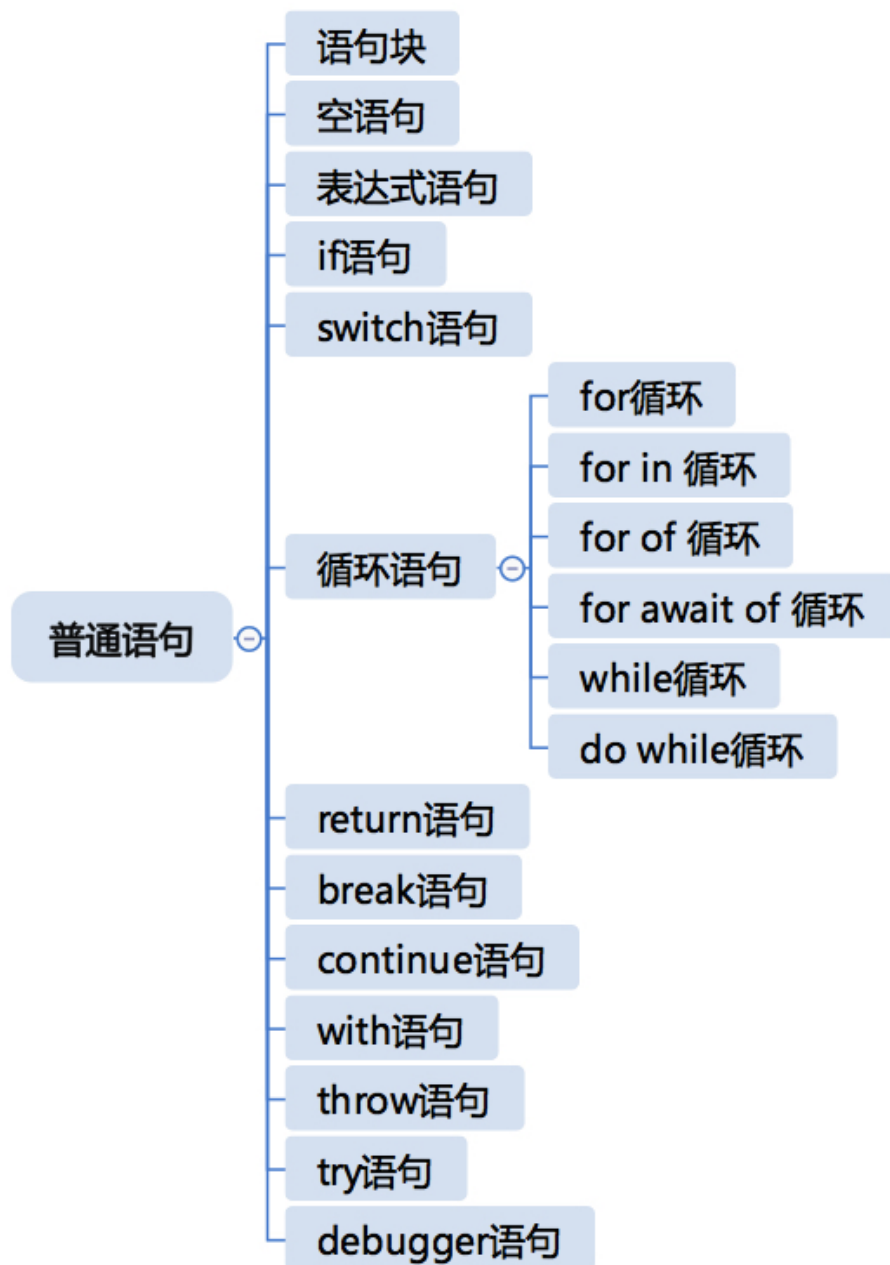
你好，我是 winter。

我们在上一节课中已经讲过了 JavaScript 语法的顶层设计，接下来我们进入到更具体的内容。

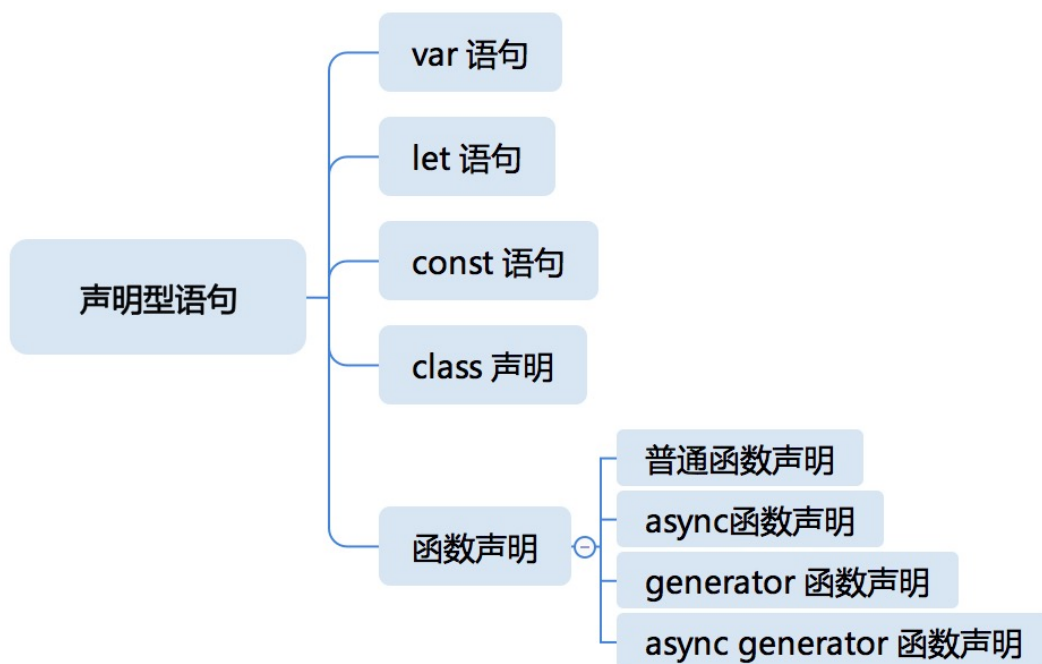
JavaScript 遵循了一般编程语言的‘语句 - 表达式’结构，多数编程语言都是这样设计的。我们在上节课讲的脚本，或者模块都是由语句列表构成的，这一节课，我们就来一起了解一下语句。

在 JavaScript 标准中，把语句分成了两种：声明和语句，不过，这里的区分逻辑比较奇怪，所以，这里我还是按照自己的思路给你整理一下。

普通语句：



声明型语句：




我们根据上面的分类，来遍历学习一下这些语句。

语句块


我们可以这样去简单理解，语句块就是一对大括号。

```
1 {  
2     var x, y;  
3     x = 10;  
4     y = 20;  
5 }  
6
```

 复制代码

语句块的意义和好处在于：让我们可以把多行语句视为同一行语句，这样，if、for 等语句定义起来就比较简单了。不过，我们需要注意的是，语句块会产生作用域，我们看一个例子：

```
1 {  
2     let x = 1;  
3 }  
4 console.log(x); // 报错  
5
```


 复制代码

这里我们的 let 声明，仅仅对语句块作用域生效，于是我们在语句块外试图访问语句块内的变量 x 就会报错。

空语句

空语句就是一个独立的分号，实际上没什么大用。我们来看一下：

```
1 ;  
2
```

 复制代码


空语句的存在仅仅是从语言设计完备性的角度考虑，允许插入多个分号而不抛出错误。

if 语句

if 语句是条件语句。我想，对多数人来说，if 语句都是熟悉的老朋友了，也没有什么特别需要注意的用法，但是为了我们课程的完备性，这里还是要讲一下。

if 语句示例如下：


```
1 if(a < b)
```

 复制代码

```
2 console.log(a);  
3
```

if 语句的作用是，在满足条件时执行它的内容语句，这个语句可以是一个语句块，这样就可以实现有条件地执行多个语句了。

if 语句还有 else 结构，用于不满足条件时执行，一种常见的用法是，利用语句的嵌套能力，把 if 和 else 连写成多分支条件判断：

 复制代码


```
1 if(a < 10) {  
2     //...  
3 } else if(a < 20) {  
4     //...  
5 } else if(a < 30) {  
6     //...  
7 } else {  
8     //...  
9 }  
10
```

这段代码表示四个互斥的分支，分别在满足 $a < 10$ 、 $a < 20$ 、 $a < 30$ 和其它情况时执行。

switch 语句


switch 语句继承自 Java，Java 中的 switch 语句继承自 C 和 C++，原本 switch 语句是跳转的变形，所以我们如果要用它来实现分支，必须要加上 break。

其实 switch 原本的设计是类似 goto 的思维。我们看一个例子：

 复制代码

```
1 switch(num) {  
2 case 1:  
3     print(1);  
4 case 2:  
5     print 2;  
6 case 3:  
7     print 3;  
8 }  
9
```

这段代码当 num 为 1 时输出 1 2 3，当 num 为 2 时输出 2 3，当 num 为 3 时输出 3。如果我们把它变成分支型，则需要在每个 case 后加上 break。

 复制代码

```
1 switch(num) {  
2 case 1:  
3     print 1;  
4     break;
```

```
5 case 2:
6     print 2;
7     break;
8 case 3:
9     print 3;
10    break;
11 }
12
```

在 C 时代，switch 生成的汇编代码性能是略优于 if else 的，但是对 JavaScript 来说，则无本质区别。我个人的看法是，现在 switch 已经完全没有必要使用了，应该用 if else 结构代替。

循环语句

循环语句应该也是你所熟悉的语句了，这里我们把重点放在一些新用法上。

while 循环和 do while 循环

这两个都是历史悠久的 JavaScript 语法了，示例大概如下：

```
1 let a = 100
2 while(a--) {
3     console.log("*");
4 }
5
```

[📄 复制代码](#)

```
1 let a = 101;
2 do {
3     console.log(a);
4 } while(a < 100)
5
```

[📄 复制代码](#)

注意，这里 do while 循环无论如何至少会执行一次。

普通 for 循环

首先我们来看看普通的 for 循环。

```
1
2 for(i = 0; i < 100; i++)
3     console.log(i);
4
5 for(var i = 0; i < 100; i++)
6     console.log(i);
7
8 for(let i = 0; i < 100; i++)
9     console.log(i);
```

[📄 复制代码](#)


```
10
11 var j = 0;
12 for(const i = 0; j < 100; j++)
13     console.log(i);
14
15
```

这里为了配合新语法，加入了允许 `let` 和 `const`，实际上，`const` 在这里是非常奇葩的东西，因为这里声明和初始化的变量，按惯例是用于控制循环的，但是它如果是 `const` 就没法改了。

我想，这一点可能是从保持 `let` 和 `const` 一致性的角度考虑的吧。

for in 循环

`for in` 循环枚举对象的属性，这里体现了属性的 `enumerable` 特征。

 复制代码

```
1 let o = { a: 10, b: 20}
2 Object.defineProperty(o, "c", {enumerable:false, value:30})
3
4 for(let p in o)
5     console.log(p);
6
7
```


这段代码中，我们定义了一个对象 `o`，给它添加了不可枚举的属性 `c`，之后我们用 `for in` 循环枚举它的属性，我们会发现，输出时得到的只有 `a` 和 `b`。

如果我们定义 `c` 这个属性时，`enumerable` 为 `true`，则 `for in` 循环中也能枚举到它。

for of 循环和 for await of 循环

`for of` 循环是非常棒的语法特性。


我们先看下基本用法，它可以用于数组：

 复制代码

```
1 for(let e of [1, 2, 3, 4, 5])
2     console.log(e);
3
```

但是实际上，它背后的机制是 `iterator` 机制。

我们可以给任何一个对象添加 `iterator`，使它可以用于 `for of` 语句，看下示例：

 复制代码

```
1 let o = {
```

```
2     [Symbol.iterator]:() => ({
3         _value: 0,
4         next(){
5             if(this._value == 10)
6                 return {
7                     done: true
8                 }
9             else return {
10                 value: this._value++,
11                 done: false
12             };
13         }
14     })
15 }
16 for(let e of o)
17     console.log(e);
18
19
```

这段代码展示了如何为一个对象添加 iterator。但是，在实际操作中，我们一般不需要这样定义 iterator，我们可以使用 generator function。

[📄 复制代码](#)

```
1 function* foo(){
2     yield 0;
3     yield 1;
4     yield 2;
5     yield 3;
6 }
7 for(let e of foo())
8     console.log(e);
9
```

这段代码展示了 generator function 和 foo 的配合。

此外，JavaScript 还为异步生成器函数配备了异步的 for of，我们来看一个例子：

[📄 复制代码](#)

```
1 function sleep(duration) {
2     return new Promise(function(resolve, reject) {
3         setTimeout(resolve,duration);
4     })
5 }
6 async function* foo(){
7     i = 0;
8     while(true) {
9         await sleep(1000);
10        yield i++;
11    }
12 }
13 }
14 for await(let e of foo())
15     console.log(e);
16
```


这段代码定义了一个异步生成器函数，异步生成器函数每隔一秒生成一个数字，这是一个无限的生成器。

接下来，我们使用 `for await of` 来访问这个异步生成器函数的结果，我们可以看到，这形成了一个每隔一秒打印一个数字的无限循环。

但是因为我们这个循环是异步的，并且有时间延迟，所以，这个无限循环的代码可以用于显示时钟等有意义的操作。

return

`return` 语句用于函数中，它终止函数的执行，并且指定函数的返回值，这是大家非常熟悉语句了，也没有什么特殊之处。

 复制代码


```
1 function squire(x){
2     return x * x;
3 }
4
```

这段代码展示了 `return` 的基本用法。它后面可以跟一个表达式，计算结果就是函数返回值。

break 语句和 continue 语句

`break` 语句用于跳出循环语句或者 `switch` 语句，`continue` 语句用于结束本次循环并继续循环。

这两个语句都属于控制型语句，用法也比较相似，所以我们就一起讲了。需要注意的是，它们都有带标签的用法。

 复制代码

```
1 outer:for(let i = 0; i < 100; i++)
2     inner:for(let j = 0; j < 100; j++)
3         if( i == 50 && j == 50)
4             break outer;
5 outer:for(let i = 0; i < 100; i++)
6     inner:for(let j = 0; j < 100; j++)
7         if( i >= 50 && j == 50)
8             continue outer;
9
```

带标签的 `break` 和 `continue` 可以控制自己被外层的哪个语句结构消费，这可以跳出复杂的语句结构。

with 语句

`with` 语句是个非常巧妙的设计，但它把 JS 的变量引用关系变得不可分析，所以一般都认为这种语句都属于糟粕。

但是历史无法改写，现在已经无法去除 with 了。我们来了解一下它的基本用法即可。

[📄 复制代码](#)

```
1 let o = {a:1, b:2}
2 with(o){
3     console.log(a, b);
4 }
5
```

with 语句把对象的属性在它内部的作用域内变成变量。

try 语句和 throw 语句

try 语句和 throw 语句用于处理异常。它们是配合使用的，所以我们就放在一起讲了。在大型应用中，异常机制非常重要。

[📄 复制代码](#)

```
1 try {
2     throw new Error("error");
3 } catch(e) {
4     console.log(e);
5 } finally {
6     console.log("finally");
7 }
8
9
```

一般来说，throw 用于抛出异常，但是单纯从语言的角度，我们可以抛出任何值，也不一定是异常逻辑，但是为了保证语义清晰，不建议用 throw 表达任何非异常逻辑。

try 语句用于捕获异常，用 throw 抛出的异常，可以在 try 语句的结构中被处理掉：try 部分用于标识捕获异常的代码段，catch 部分则用于捕获异常后做一些处理，而 finally 则是用于执行后做一些必须执行的清理工作。

catch 结构会创建一个局部的作用域，并且把一个变量写入其中，需要注意，在这个作用域，不能再声明变量 e 了，否则会出错。

在 catch 中重新抛出错误的情况非常常见，在设计比较底层的函数时，常常会这样做，保证抛出的错误能被理解。

finally 语句一般用于释放资源，它一定会被执行，我们在前面的课程中已经讨论过一些 finally 的特征，即使在 try 中出现了 return，finally 中的语句也一定要被执行。（你可以参考第 19 讲）

debugger 语句

debugger 语句的作用是：通知调试器在此断点。在没有调试器挂载时，它不产生任何效果。

介绍完普通语句，我们再来看看声明型语句。声明型语句跟普通语句最大区别就是声明型语句响应预处理过程，普通语句只有执行过程。

var

var 声明语句是古典的 JavaScript 中声明变量的方式。而现在，在绝大多数情况下，let 和 const 都是更好的选择。

我们在上一节课已经讲解了 var 声明对全局作用域的影响，它是一种预处理机制。

如果我们仍然想要使用 var，我的个人建议是，把它当做一种“保障变量是局部”的逻辑，遵循以下三条规则：


声明同时必定初始化；

尽可能在离使用的位置近处声明；

不要在意重复声明。

例如：


```
1 var x = 1, y = 2;
2 doSth(x, y);
3
4 for(var x = 0; x < 10; x++)
5     doSth2(x);
6
```

 复制代码

这个例子中，两次声明了变量 x，完成了两段逻辑，这两个 x 意义上可能不一定相关，这样，不论我们把代码复制粘贴在哪里，都不会出错。

当然，更好的办法是使用 let 改造，我们看看如何改造：

```
1 {
2     let x = 1, y = 2;
3     doSth(x, y);
4 }
5
6 for(let x = 0; x < 10; x++)
7     doSth2(x);
8
```

 复制代码

这里我用代码块限制了第一个 x 的作用域，这样就更难发生变量命名冲突引起的错误了。

let 和 const

`let` 和 `const` 是都是变量的声明，它们的特性非常相似，所以我们放在一起讲了。`let` 和 `const` 是新设计的语法，所以没有什么硬伤，非常地符合直觉。`let` 和 `const` 的作用范围是 `if`、`for` 等结构型语句。

我们看下基本用法：

[📄 复制代码](#)

```
1 const a = 2;
2 if(true){
3     const a = 1;
4     console.log(a);
5 }
6 console.log(a);
7
```

这里的代码先在全局声明了变量 `a`，接下来又在 `if` 内声明了 `a`，`if` 内构成了一个独立的作用域。

`const` 和 `let` 语句在重复声明时会抛错，这能够有效地避免变量名无意中冲突：

[📄 复制代码](#)

```
1 let a = 2
2 const a = 1;
3
```

这段代码中，先用 `let` 声明了 `a`，接下来又试图使用 `const` 声明变量 `a`，这时，就会产生错误。

`let` 和 `const` 声明虽然看上去是执行到了才会生效，但是实际上，它们还是会被预处理。如果当前作用域内有声明，就无法访问到外部的变量。我们来看这段代码：

[📄 复制代码](#)

```
1 const a = 2;
2 if(true){
3     console.log(a); // 抛错
4     const a = 1;
5 }
6
```

这里在 `if` 的作用域中，变量 `a` 声明执行到之前，我们访问了变量 `a`，这时会抛出一个错误，这说明 `const` 声明仍然是有预处理机制的。

在执行到 `const` 语句前，我们的 JavaScript 引擎就已经知道后面的代码将会声明变量 `a`，从而不允许我们访问外层作用域中的 `a`。

class 声明

我们在之前的课程中，已经了解过 `class` 相关的用法。这里我们再从语法的角度来看一遍：

```
1 class a {  
2  
3 }  
4
```

class 最基本的用法只需要 class 关键字、名称和一对大括号。它的声明特征跟 const 和 let 类似，都是作用于块级作用域，预处理阶段则会屏蔽外部变量。

```
1 const a = 2;  
2 if(true){  
3     console.log(a); // 抛错  
4     class a {  
5  
6     }  
7 }  
8
```

class 内部，可以使用 constructor 关键字来定义构造函数。还能定义 getter/setter 和方法。

```
1 class Rectangle {  
2     constructor(height, width) {  
3         this.height = height;  
4         this.width = width;  
5     }  
6     // Getter  
7     get area() {  
8         return this.calcArea();  
9     }  
10    // Method  
11    calcArea() {  
12        return this.height * this.width;  
13    }  
14 }  
15
```

这个例子来自 MDN，它展示了构造函数、getter 和方法的定义。

以目前的兼容性，class 中的属性只能写在构造函数中，相关标准正在 TC39 讨论。

需要注意，class 默认内部的函数定义都是 strict 模式的。

函数声明

函数声明使用 function 关键字。

在上一节课中，我们已经讨论过函数声明对全局作用域的影响了。这一节课，我们来看看函数声明具体的内容，我们先看一下函数声明的几种类型

[📄 复制代码](#)

```
1
2 function foo(){
3
4 }
5
6 function* foo(){
7     yield 1;
8     yield 2;
9     yield 3;
10 }
11
12 async function foo(){
13     await sleep(3000);
14
15 }
16
17 async function* foo(){
18     await sleep(3000);
19     yield 1;
20 }
21
22
```

带 * 的函数是 generator，我们在前面的部分已经见过它了。生成器函数可以理解为返回一个序列的函数，它的底层是 iterator 机制。

async 函数是可以暂停执行，等待异步操作的函数，它的底层是 Promise 机制。异步生成器函数则是二者的结合。

函数的参数，可以只写形参名，现在还可以写默认参数和指定多个参数，看下例子：

[📄 复制代码](#)

```
1
2 function foo(a = 1, ...other) {
3     console.log(a, other)
4 }
5
```

这个形式可以代替一些对参数的处理代码，表意会更加清楚。

结语

今天我们一起学习了语句家族，语句分成了普通语句和声明型语句。

普通语句部分，建议你把重点放在循环语句上面。声明型语句我觉得都很重要，尤其是它们的行为。熟练掌握了它们，我们就可以在工作中去综合运用它们，从而减少代码中的错误。新特性大多可以帮助我们发现代码中的错误。

最后留一个小作业，请你找出所有具有 Symbol.iterator 的原生对象，并且看看它们的 for of 遍历行为。

©



由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。

Ctrl + Enter 发表

0/2000字

提交留言

精选留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。