30-JavaScript语法（二）：你知道哪些JavaScript语句？

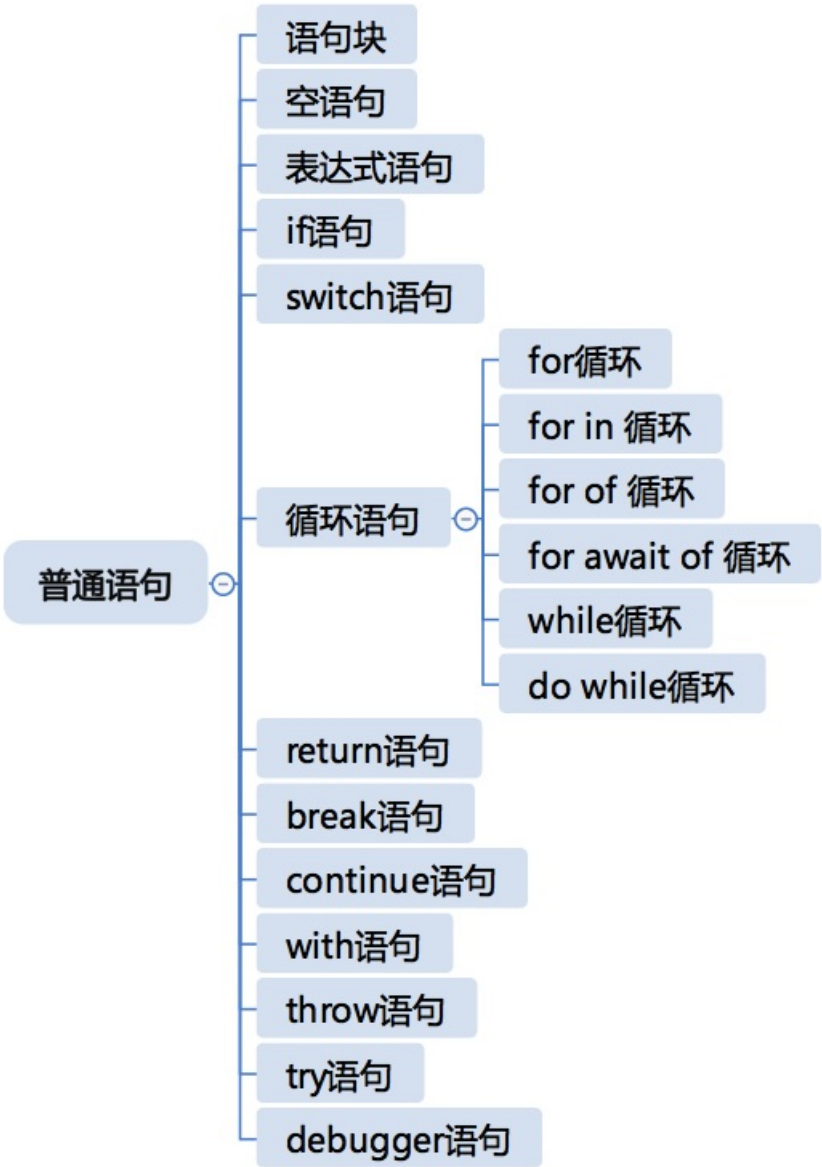
你好，我是winter。

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。

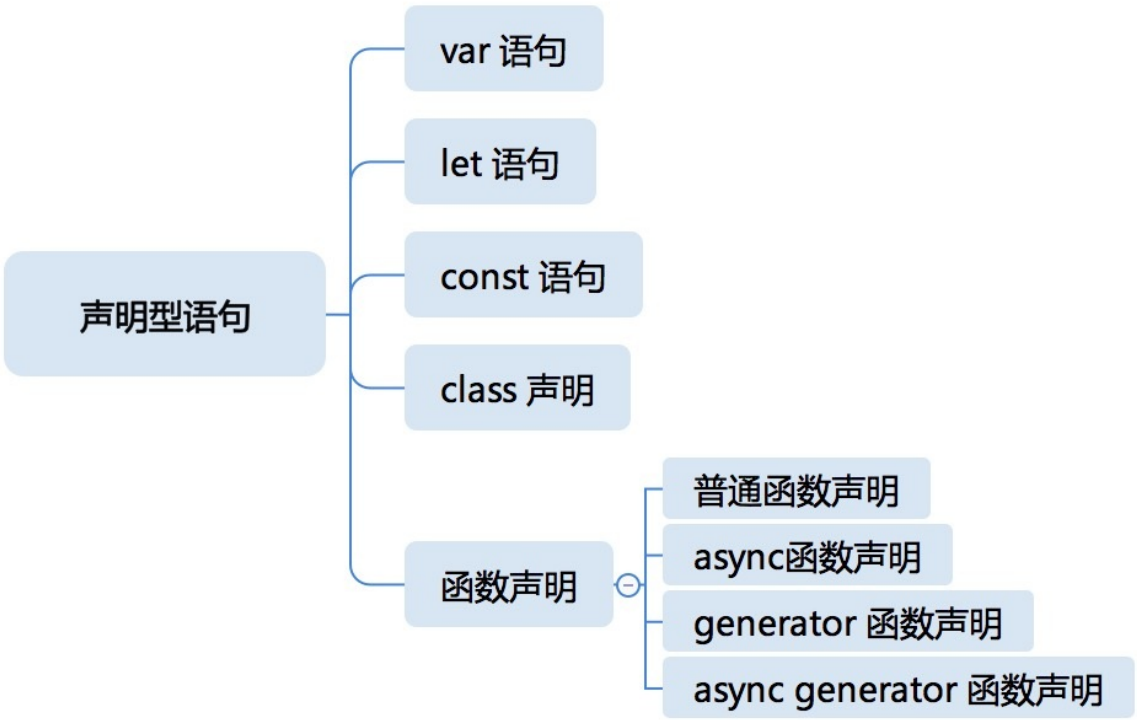
JavaScript遵循了一般编程语言的‘语句-表达式’结构，多数编程语言都是这样设计的。我们在上节课讲的脚本，或者模块都是由语句列表构成的，这一节课，我们就来一起了解一下语句。

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。

```
{
    var x, y;
    x = 10;
    y = 20;
}
```

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：

```
{
    let x = 1;
}
console.log(x); // 报错
```

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：

```
;
```

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：

```
if(a < b)
    console.log(a);
```

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：

```
if(a < 10) {
    //...
} else if(a < 20) {
    //...
} else if(a < 30) {
    //...
} else {
    //...
}
```

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：

```
switch(num) {
case 1:
    print(1);
```

```
case 2:
    print 2;
case 3:
    print 3;
}
```

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。如果我们要把它变成分支型，则需要在每个case后加上break。

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

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：

```
let a = 100
while(a--) {
    console.log("*");
}
```

```
let a = 101;
do {
    console.log(a);
} while(a < 100)
```

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。

```
for(i = 0; i < 100; i++)
    console.log(i);

for(var i = 0; i < 100; i++)
    console.log(i);

for(let i = 0; i < 100; i++)
    console.log(i);

var j = 0;
for(const i = 0; j < 100; j++)
    console.log(i);
```

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。

```
let o = { a: 10, b: 20}
Object.defineProperty(o, "c", {enumerable:false, value:30})

for(let p in o)
    console.log(p);
```

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：

```
for(let e of [1, 2, 3, 4, 5])
```

```
    console.log(e);
```

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：

```
let o = {
    [Symbol.iterator]:() => ({
        _value: 0,
        next(){
            if(this._value == 10)
                return {
                    done: true
                }
            else return {
                value: this._value++,
                done: false
            };
        }
    })
}
for(let e of o)
    console.log(e);
```

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。

```
function* foo(){
    yield 0;
    yield 1;
    yield 2;
    yield 3;
}
for(let e of foo())
    console.log(e);
```

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：

```
function sleep(duration) {
    return new Promise(function(resolve, reject) {
        setTimeout(resolve,duration);
    })
}
async function* foo(){
    i = 0;
```

```
    while(true) {
        await sleep(1000);
        yield i++;
    }

}
for await(let e of foo())
    console.log(e);
```

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。

```
function squre(x){
    return x * x;
}
```

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。

```
outer:for(let i = 0; i < 100; i++)
    inner:for(let j = 0; j < 100; j++)
        if( i == 50 && j == 50)
            break outer;
outer:for(let i = 0; i < 100; i++)
    inner:for(let j = 0; j < 100; j++)
        if( i >= 50 && j == 50)
            continue outer;
```

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。

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

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。

```
try {
    throw new Error("error");
} catch(e) {
    console.log(e);
} finally {
    console.log("finally");
}
```

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：

```
var x = 1, y = 2;
doSth(x, y);

for(var x = 0; x < 10; x++)
    doSth2(x);
```

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：

```
{
    let x = 1, y = 2;
    doSth(x, y);
}

for(let x = 0; x < 10; x++)
    doSth2(x);
```

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：

```
const a = 2;
if(true){
    const a = 1;
    console.log(a);
}
console.log(a);
```

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：

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

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：

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

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：

```
class a {
```

```
}
```

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。

```
const a = 2;
if(true){
    console.log(a); //抛错
    class a {

    }
}
```

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。

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
  // Getter
  get area() {
    return this.calcArea();
  }
  // Method
  calcArea() {
    return this.height * this.width;
  }
}
```

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

```
function foo(){
```

```
}

function* foo(){
    yield 1;
    yield 2;
    yield 3;
}

async function foo(){
    await sleep(3000);

}

async function* foo(){
    await sleep(3000);
    yield 1;
}
```

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：

```
function foo(a = 1, ...other) {
    console.log(a, other)
}
```

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。

精选留言：

- 阿成 2019-04-02 08:38:34
  大概就这些？
  Array, Map, Set, String, Float32Array, Float64Array, Int8Array, Int16Array, Int32Array, Uint8Array, Uint16Array, Uint32Array, Uint8ClampedArray [5赞]

- mfist 2019-04-04 07:21:26
  遍历了下window上面的全局对象，上面有Symbol.iterator的原生属性有15个，主要Array Set Map String相关的。当然还有很多宿主环境提供的全局对象有Symbol.iterator属性，他们有个共同的特征：都是些集合性质的数据结构。
  0: "Array"
  1: "String"
  2: "Uint8Array"
  3: "Int8Array"
  4: "Uint16Array"
  5: "Int16Array"
  6: "Uint32Array"
  7: "Int32Array"
  8: "Float32Array"
  9: "Float64Array"
  10: "Uint8ClampedArray"
  11: "BigUint64Array"
  12: "BigInt64Array"
  13: "Map"
  14: "Set" [3赞]

- 许童童 2019-04-02 14:38:31
  Some built-in types have a default iteration behavior, while other types (such as Object) do not. The built-in types with a @@iterator method are:
  Array.prototype[@@iterator]()
  TypedArray.prototype[@@iterator]()

String.prototype[@@iterator]()
Map.prototype[@@iterator]()
Set.prototype[@@iterator]() [1赞]

- 让时间说真话 2019-04-12 17:06:52
  Map，set，arguments

- 翰弟 2019-04-04 16:16:11
  Array、Map、Set、String、TypedArray、函数的arguments、NodeList对象

- qqq 2019-04-03 10:15:51
  catch 中可以使用 var 重新声明

- Format 2019-04-02 19:29:19
  请问老师后面可以讲讲，例如手淘购物车这种较复杂的功能吗？详细的那种，封装方法，兼容，处理特殊情况之类的

- K4SHIFZ 2019-04-02 16:55:02
  请问老师，规范中的Statement和Declaration到底有什么区别？不都是声明的意思吗？

  作者回复2019-04-09 18:24:48
  Statement是语句，Declaration是声明，但是我觉得这个分类不好，因为语句里还有var语句也是声明性质的。

- K4SHIFZ 2019-04-02 16:12:30
  为什么和第十九节中的分类不一样啊？