29-JavaScript语法（一）：在script标签写export为什么会抛错？

你好，我是winter，今天我们进入到语法部分的学习。在讲解具体的语法结构之前，这一堂课我首先要给你介绍一下JavaScript语法的一些基本规则。

脚本和模块

首先，JavaScript有两种源文件，一种叫做脚本，一种叫做模块。这个区分是在ES6引入了模块机制开始的，在ES5和之前的版本中，就只有一种源文件类型（就只有脚本）。

脚本是可以由浏览器或者node环境引入执行的，而模块只能由JavaScript代码用import引入执行。

从概念上，我们可以认为脚本具有主动性的JavaScript代码段，是控制宿主完成一定任务的代码；而模块是被动性的JavaScript代码段，是等待被调用的库。

我们对标准中的语法产生式做一些对比，不难发现，实际上模块和脚本之间的区别仅仅在于是否包含import和 export。

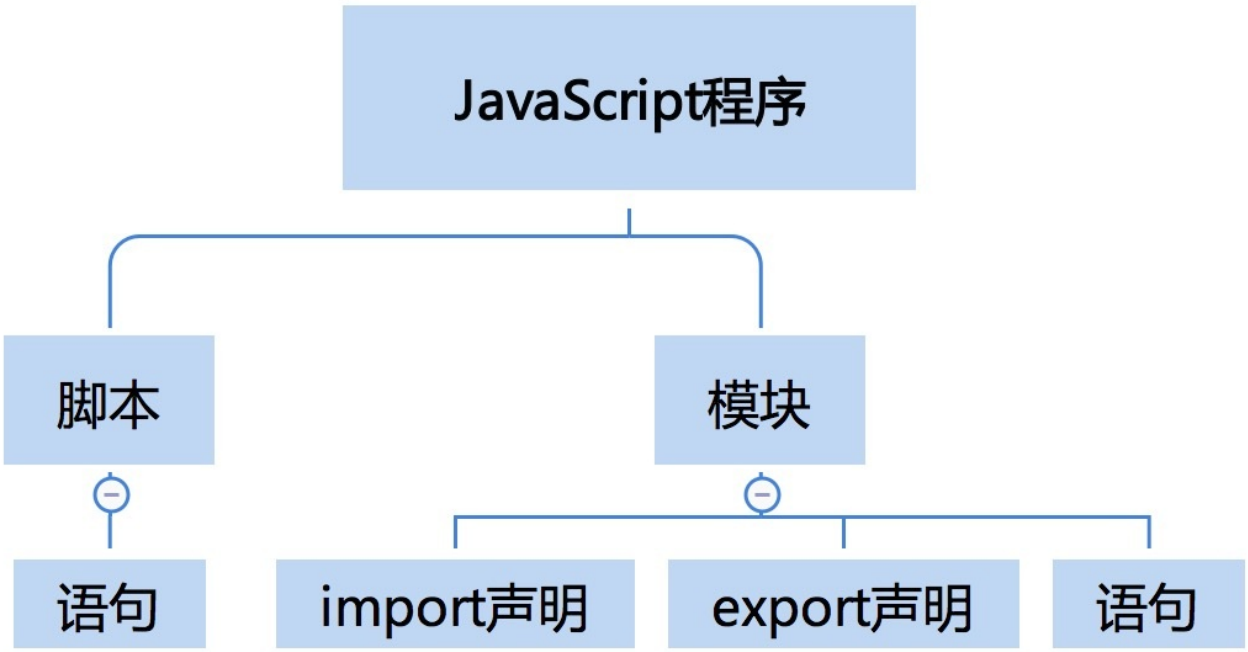
脚本是一种兼容之前的版本的定义，在这个模式下，没有import就不需要处理加载“.js”文件问题。

现代浏览器可以支持用script标签引入模块或者脚本，如果要引入模块，必须给script标签添加type=“module”。如果引入脚本，则不需要type。

```
<script type="module" src="xxxxx.js"></script>
```

这样，就回答了我们标题中的问题，script标签如果不加type=“module”，默认认为我们加载的文件是脚本而非模块，如果我们在脚本中写了export，当然会抛错。

脚本中可以包含语句。模块中可以包含三种内容：import声明，export声明和语句。普通语句我们会在下一课专门给你讲解，下面我们就来讲讲import声明和export声明。



import声明

我们首先来介绍一下import声明，import声明有两种用法，一个是直接import一个模块，另一个是带from的import，它能引入模块里的一些信息。

```
import "mod"; //引入一个模块
import v from "mod";  //把模块默认的导出值放入变量v
```

直接import一个模块，只是保证了这个模块代码被执行，引用它的模块是无法获得它的任何信息的。

带from的import意思是引入模块中的一部分信息，可以把它们变成本地的变量。

带from的import细分又有三种用法，我们可以分别看下例子：

- `import x from "./a.js"` 引入模块中导出的默认值。
- `import {a as x, modify} from "./a.js";` 引入模块中的变量。
- `import * as x from "./a.js"` 把模块中所有的变量以类似对象属性的方式引入。

第一种方式还可以跟后两种组合使用。

- `import d, {a as x, modify} from "./a.js"`
- `import d, * as x from "./a.js"`

语法要求不带as的默认值永远在最前。注意，这里的变量实际上仍然可以受到原来模块的控制。

我们看一个例子，假设有两个模块a和b。我们在模块a中声明了变量和一个修改变量的函数，并且把它们导出。我们用b模块导入了变量和修改变量的函数。

模块a：

```
export var a = 1;

export function modify(){
    a = 2;
}
```

模块b：

```
import {a, modify} from "./a.js";

console.log(a);

modify();

console.log(a);
```

当我们调用修改变量的函数后，b模块变量也跟着发生了改变。这说明导入与一般的赋值不同，导入后的变量只是改变了名字，它仍然与原来的变量是同一个。

export声明

我们再来说说export声明。与import相对，export声明承担的是导出的任务。

模块中导出变量的方式有两种，一种是独立使用export声明，另一种是直接在声明型语句前添加export关键字。

独立使用export声明就是一个export关键字加上变量名列表，例如：

```
export {a, b, c};
```

我们也可以直接在声明型语句前添加export关键字，这里的export可以加在任何声明性质的语句之前，整理如下：

- var
- function (含async和generator)
- class
- let
- const

export还有一种特殊的用法，就是跟default联合使用。export default 表示导出一个默认变量值，它可以用于function和class。这里导出的变量是没有名称的，可以使用`import x from "./a.js"`这样的语法，在模块中引入。

export default 还支持一种语法，后面跟一个表达式，例如：

```
var a = {};
export default a;
```

但是，这里的行为跟导出变量是不一致的，这里导出的是值，导出的就是普通变量a的值，以后a的变化与导出的值就无关了，修改变量a，不会使得其他模块中引入的default值发生改变。

在import语句前无法加入export，但是我们可以直接使用export from语法。

```
export a from "a.js"
```

JavaScript引擎除了执行脚本和模块之外，还可以执行函数。而函数体跟脚本和模块有一定的相似之处，所以接下来，给你讲讲函数体的相关知识。

函数体

执行函数的行为通常是在JavaScript代码执行时，注册宿主环境的某些事件触发的，而执行的过程，就是执行函数体（函数的花括号中间的部分）。

我们先看一个例子，感性地理解一下：

```
setTimeout(function(){
    console.log("go go go");
}, 10000)
```

这段代码通过setTimeout函数注册了一个函数给宿主，当一定时间之后，宿主就会执行这个函数。

你还记得吗，我们前面已经在运行时这部分讲过，宿主会为这样的函数创建宏任务。

当我们学习了语法之后，我们可以认为，宏任务中可能会执行的代码包括“脚本(script)”“模块（module）”和“函数体（function body）”。正因为这样的相似性，我们把函数体也放到本课来讲解。

函数体其实也是一个语句的列表。跟脚本和模块比起来，函数体中的语句列表中多了return语句可以用。

函数体实际上有四种，下面，我来分别介绍一下。

- 普通函数体，例如：

```
function foo(){
    //Function body
}
```

- 异步函数体，例如：

```
async function foo(){
    //Function body
}
```

- 生成器函数体，例如：

```
function *foo(){
    //Function body
}
```

- 异步生成器函数体，例如：

```
async function *foo(){
    //Function body
}
```

上面四种函数体的区别在于：能否使用await或者yield语句。

关于函数体、模块和脚本能使用的语句，我整理了一个表格，你可以参考一下：

类型	yield	await	return	import & export
普通函数体	×	×	√	×
异步函数体	×	√	√	×
生成器函数体	√	×	√	×
异步生成器函数体	√	√	√	×
脚本	×	×	×	×
模块	×	×	×	√

讲完了三种语法结构，我再来介绍两个JavaScript语法的全局机制：预处理和指令序言。

这两个机制对于我们解释一些JavaScript的语法现象非常重要。不理解预处理机制我们就无法理解var等声明类语句的行为，而不理解指令序言，我们就无法解释严格模式。

预处理

JavaScript执行前，会对脚本、模块和函数体中的语句进行预处理。预处理过程将会提前处理var、函数声明、class、const和let这些语句，以确定其中变量的意义。

因为一些历史包袱，这一部分内容非常复杂，首先我们看一下var声明。

var声明

var声明永远作用于脚本、模块和函数体这个级别，在预处理阶段，不关心赋值的部分，只管在当前作用域声明这个变量。

我们还是从实例来进行学习。

```
var a = 1;

function foo() {
    console.log(a);
    var a = 2;
}

foo();
```

这段代码声明了一个脚本级别的a，又声明了foo函数体级别的a，我们注意到，函数体级的var出现在console.log语句之后。

但是预处理过程在执行之前，所以有函数体级的变量a，就不会去访问外层作用域中的变量a了，而函数体

级的变量a此时还没有赋值，所以是undefined。我们再看一个情况：

```
var a = 1;

function foo() {
    console.log(a);
    if(false) {
        var a = 2;
    }
}

foo();
```

这段代码比上一段代码在`var a = 2`之外多了一段if，我们知道if(false)中的代码永远不会被执行，但是预处理阶段并不管这个，var的作用能够穿透一切语句结构，它只认脚本、模块和函数体三种语法结构。所以这里结果跟前一段代码完全一样，我们会得到undefined。

我们看下一个例子，我们在运行时部分讲过类似的例子。

```
var a = 1;

function foo() {
    var o= {a:3}
    with(o) {
        var a = 2;
    }
    console.log(o.a);
    console.log(a);
}

foo();
```

在这个例子中，我们引入了with语句，我们用with(o)创建了一个作用域，并把o对象加入词法环境，在其中使用了`var a = 2;`语句。

在预处理阶段，只认`var`中声明的变量，所以同样为foo的作用域创建了a这个变量，但是没有赋值。

在执行阶段，当执行到`var a = 2`时，作用域变成了with语句内，这时候的a被认为访问到了对象o的属性a，所以最终执行的结果，我们得到了2和undefined。

这个行为是JavaScript公认的设计失误之一，一个语句中的a在预处理阶段和执行阶段被当做两个不同的变量，严重违背了直觉，但是今天，在JavaScript设计原则“don’t break the web”之下，已经无法修正了，所以你需要特别注意。

因为早年JavaScript没有let和const，只能用var，又因为var除了脚本和函数体都会穿透，人民群众发明了“立即执行的函数表达式（IIFE）”这一用法，用来产生作用域，例如：

```
for(var i = 0; i < 20; i ++) {
    void function(i){
        var div = document.createElement("div");
        div.innerHTML = i;
        div.onclick = function(){
            console.log(i);
        }
        document.body.appendChild(div);
    }(i);
}
```

这段代码非常经典，常常在实际开发中见到，也经常被用作面试题，为文档添加了20个div元素，并且绑定了点击事件，打印它们的序号。

我们通过IIFE在循环内构造了作用域，每次循环都产生一个新的环境记录，这样，每个div都能访问到环境中的i。

如果我们不用IIFE：

```
for(var i = 0; i < 20; i ++) {
    var div = document.createElement("div");
    div.innerHTML = i;
    div.onclick = function(){
        console.log(i);
    }
    document.body.appendChild(div);
}
```

这段代码的结果将会是点每个div都打印20，因为全局只有一个i，执行完循环后，i变成了20。

function声明

function声明的行为原本跟var非常相似，但是在最新的JavaScript标准中，对它进行了一定的修改，这让情况变得更加复杂了。

在全局（脚本、模块和函数体），function声明表现跟var相似，不同之处在于，function声明不但在作用域中加入变量，还会给它赋值。

我们看一下function声明的例子

```
console.log(foo);
function foo(){

}
```

这里声明了函数foo，在声明之前，我们用console.log打印函数foo，我们可以发现，已经是函数foo的值了。

function声明出现在if等语句中的情况有点复杂，它仍然作用于脚本、模块和函数体级别，在预处理阶段，仍然会产生变量，它不再被提前赋值：

```
console.log(foo);
if(true) {
    function foo(){

    }
}
```

这段代码得到undefined。如果没有函数声明，则会抛出错误。

这说明function在预处理阶段仍然发生了作用，在作用域中产生了变量，没有产生赋值，赋值行为发生在了执行阶段。

出现在if等语句中的function，在if创建的作用域中仍然会被提前，产生赋值效果，我们会在下一节课继续讨论。

class声明

class声明在全局的行为跟function和var都不一样。

在class声明之前使用class名，会抛错：

```
console.log(c);
class c{

}
```

这段代码我们试图在class前打印变量c，我们得到了个错误，这个行为很像是class没有预处理，但是实际上并非如此。

我们看个复杂一点的例子：

```
var c = 1;
function foo(){
    console.log(c);
    class c {}
}
foo();
```

这个例子中，我们把class放进了一个函数体中，在外层作用域中有变量c。然后试图在class之前打印c。

执行后，我们看到，仍然抛出了错误，如果去掉class声明，则会正常打印出1，也就是说，出现在后面的class声明影响了前面语句的结果。

这说明，class声明也是会被预处理的，它会在作用域中创建变量，并且要求访问它时抛出错误。

class的声明作用不会穿透if等语句结构，所以只有写在全局环境才会有声明作用，这部分我们将会在下一节课讲解。

这样的class设计比function和var更符合直觉，而且在遇到一些比较奇怪的用法时，倾向于抛出错误。

按照现代语言设计的评价标准，及早抛错是好事，它能够帮助我们尽量在开发阶段就发现代码的可能问题。

指令序言机制

脚本和模块都支持一种特别的语法，叫做指令序言（Directive Prologs）。

这里的指令序言最早是为了use strict设计的，它规定了一种给JavaScript代码添加元信息的方式。

```
"use strict";
function f(){
    console.log(this);
};
f.call(null);
```

这段代码展示了严格模式的用法，我这里定义了函数f，f中打印this值，然后用call的方法调用f，传入null作为this值，我们可以看到最终结果是null原封不动地被当做this值打印了出来，这是严格模式的特征。

如果我们去掉严格模式的指令需要，打印的结果将会变成global。

"use strict"是JavaScript标准中规定的唯一一种指令序言，但是设计指令序言的目的是，留给JS的引擎和实现者一些统一的表达方式，在静态扫描时指定JS代码的一些特性。

例如，假设我们要设计一种声明本文件不需要进行lint检查的指令，我们可以这样设计：

```
"no lint";
"use strict";
function doSth(){
    //......
}
//......
```

JavaScript的指令序言是只有一个字符串直接量的表达式语句，它只能出现在脚本、模块和函数体的最前

面。

我们看两个例子：

```
function doSth(){
    //......
}
"use strict";
var a = 1;
//......
```

这个例子中，"use strict"没有出现在最前，所以不是指令序言。

```
'use strict';
function doSth(){
    //......
}
var a = 1;
//......
```

这个例子中，'use strict'是单引号，这不妨碍它仍然是指令序言。

总结

今天，我们一起进入了JavaScript的语法部分，在开始学习之前，我先介绍了一部分语法的基本规则。

我们首先介绍了JavaScript语法的全局结构，JavaScript有两种源文件，一种叫做脚本，一种叫做模块。介绍完脚本和模块的基础概念，我们再来把它们往下分，脚本中可以包含语句。模块中可以包含三种内容：import声明，export声明和语句。

最后，我介绍了两个JavaScript语法的全局机制：预处理和指令序言。

最后，给你留一个小任务，我们试着用babel，分析一段JavaScript的模块代码，并且找出它中间的所有export的变量。

精选留言：

- Leo 2019-03-28 14:20:03
  当你认为你已经掌握了JS，JS会反手给你一巴掌。 [21赞]

- 以勒 2019-04-01 09:19:11
  前面学的宏观任务和微观人物 还记得的同学举个手，点个赞 [3赞]

- 让时间说真话 2019-03-30 16:42:46
  首先讲了脚本和模块，而这次老师讲的模块补缺我近段时间用模块时的一些疑问，Js的预处理语法让我更加理解了以前经常用到的作用域。感谢winter！！！[2赞]

- 许童童 2019-03-28 15:35:09
  通过@babel/parser解析模块文件，然后通过遍历ExportNamedDeclaration，找出所有export的变量，spec参考：https://github.com/babel/babel/blob/master/packages/babel-parser/ast/spec.md#exports [2赞]

- 无羡 2019-03-29 09:44:09
  能否讲讲为什么导出的无论是基本类型还是引用类型，都会和原模块的变量有绑定关系？ [1赞]

- 阿成 2019-03-28 12:37:49
  https://github.com/aimergenge/get-exported-names-via-babel [1赞]

- K4SHIFZ 2019-03-28 11:53:27
  相反，我看了老师的文章后，以前觉得是语言bug的地方，现在觉得不是bug了 [1赞]

- _CountingStars 2019-03-28 08:36:20
  作为一个非前段程序员 看了老师的专栏发现 js 坑真多 各种奇怪的语法和表现 感觉像语言的 bug 一样 [1赞]

- 马儿 2019-04-15 18:14:03

把var的那个例子改写成let，报错a is not defined 。不明白let是怎么进行预处理的

- 马儿 2019-04-15 18:02:01
  真不亏为大神！长见识了

- xwchris 2019-04-08 10:28:44
  console.log(foo);

  if (true) {
  function foo() {}
  }

  为什么这段代码 我在chrome73中执行得到的是ƒ foo() {}

  作者回复2019-04-09 18:07:53
  老内核和新内核不一样

- favorlm 2019-03-29 08:16:37
  准备用babel进行分析

- 有铭 2019-03-28 14:03:00
  看了老师的文章，越来越理解为啥TS出现的地方越来越多了

- 翰弟 2019-03-28 13:06:20
  君子承诺 老师出的课 继续买反复看

- 阿成 2019-03-28 13:03:50
  * 预处理机制让我对 js 中的声明有了更全面的认识，很多文章中提到的一个词是“提升”，与这里提到的预处理机制不无关联。
  * 关于声明这块儿，这篇文章讲得也有点意思，不知道winter老师怎么看：
  https://zhuanlan.zhihu.com/p/28140450
  * 在我看来，if中的function声明在预处理阶段的”赋值“行为好像被if形成的块级作用域”拦截“了，导致这个赋值行为推迟到if语句块执行开始之前。（这里只是一种隐喻，并不准确）。
  * let,const,class这些在js中的”后来者“由于没有历史包袱，行为大多更加正常（符合直觉，及早抛错）。这让我想到了一篇文章中介绍的temporal dead zone机制：http://es6.ruanyifeng.com/#docs/let#%E6%9A%82%E6%97%B6%E6%80%A7%E6%AD%BB%E5%8C%BA

- 阿成 2019-03-28 09:35:05
  想问一个问题：import 进来的引用为什么可以获取到最新的值，是类似于 getter 的机制吗？