

17 变量提升与暂时性死区

更新时间：2020-05-26 15:08:08



“

自信和希望是青年的特权。——大仲马

”

本节我们一起来关注到 ES2015 中新增的 `let` 和 `const` 关键字相关的考点。

从难度上来说，本节很难和前面几节相提并论。但既然是能被笔者选进这个专栏的知识块，它必定有“两把刷子”。事实上，`let` 和 `const` 相关的考题，若要就 `let` 和 `const` 关键字特性本身提问，确实很难考察出候选人的水平。但是，它们背后所牵扯出的变量提升、暂时性死区等知识点，对一些同学来说却具有相当的挑战性——这块知识属于大家多少都听说过、能说上那么两句，但基本没几个人能说清楚的类型。

注：本节所覆盖到的知识点多且琐碎，大家在学习的过程中，需要留心跟随笔者的思路，去尝试建立点与点之间的联系，把细碎的知识点串进一条线里。

从变量提升说起

在 ES2015 之前，JS 引擎用 `var` 这个关键字声明所有的变量。

在 `var` 时代，有一个特别的现象：不管我们的变量声明是写在程序的哪个角落，最后都会被提到作用域的顶端去。我们直接用代码来理解这个情况：

```
console.log(num)
var num = 1
```

这段代码不会报错，反而会输出一个 `undefined`。这就是因为变量的声明被“提升”了，它等价于这样：

```
var num
console.log(num)
num = 1
```

上面这个例子里，我们看到 `num` 作为全局变量会被提升到全局作用域的头部。在函数作用域里，也会有类似的现象发生：

```
function getNum() {
  console.log(num)
  var num = 1
}
```

这里同样会输出 `undefined`，这是因为函数内部的变量声明会被提升至函数作用域的顶端。上面这个例子其实等价于：

```
function getNum() {
  var num
  console.log(num)
  num = 1
}
```

OK，现在大家已经看到了变量提升的效果。那么为啥会有变量提升呢？

变量提升的原理

这和咱们在 `js` 核心部分提到的 `js` 编译过程有关。咱们一起来复习下：

事实上，**JS**也是有编译阶段的，它和传统语言的区别在于，**JS**不会早早地把编译工作做完，而是一边编译一边执行。简单来说，所有的**JS**代码片段在执行之前都会被编译，只是这个编译的过程非常短暂（可能就只有几微妙、或者更短的时间），紧接着这段代码就会被执行。

没错，**JS** 和其他语言一样，都要经历编译和执行阶段。正是在这个短暂的**编译阶段**里，**JS** 引擎会搜集所有的变量声明，并且提前让声明生效。至于剩下的语句，则需要等到执行阶段、等到执行到具体的某一句的时候才会生效。这就是变量提升背后的机制。

被禁用的变量提升

这里我们就聊到了 `let` 和 `const` 区别于 `var` 的一个重要特性——它们不存在变量提升。

现在我们把上面例子里的 `num` 用 `let` 来声明：

```
console.log(num)
let num = 1
```

会发现报错了：

```
> console.log(num)
let num = 1
```

✖ ▶ Uncaught ReferenceError: Cannot access 'num' before initialization
at <anonymous>:1:13

如果改成 `const` 声明，也会是一样的结局——用 `let` 和 `const` 声明的变量，它们的声明生效时机和具体代码的执行时机保持一致。

这样做是因为，早期的声明提升机制，其实容纳了很多程序员的误操作——那些忘记被声明的变量无法在开发阶段被明显地察觉出来，而是以 `undefined` 这样危险的形式藏匿在你的代码里。为了减少运行时错误，防止暗中使坏的 `undefined` 带来不可预知的问题，ES6 特意将“声明前不可用”这一点做了强约束。

块作用域

块作用域是伴随 ES6 而生的一个概念。我们把被一对花括号括起来的代码称为一个代码块：

```
{
  let me = 'xiuyan'
  console.log(me) // 'xiuyan'
}
```

被这个代码块圈起来的变量集，就是块作用域。

let 与 const

`let` 和 `const` 都是 ES6 中用于变量声明的关键字。我们先把它们分开来看：

let 关键字与 var 关键字

大家理解 `let` 的时候可以参考 `var`。`let` 和 `var` 非常相似，`let` 区别于 `var` 的最关键的地方在于：当我们用 `let` 声明变量时，变量会被绑定到块作用域上，而 `var` 是不感知块作用域的。我们先看下 `var` 的表现：

```
{
  var me = 'xiuyan'
  console.log(me) // 'xiuyan'
}

console.log(me) // 'xiuyan'
```

我们看到在代码块里用 `var` 定义的变量，在代码块之外也能访问到。这时的花括号压根创建不出啥块作用域。

```
{
  let me = 'xiuyan'
  console.log(me) // 'xiuyan'
}

console.log(me) // 报错
```

而当我们用 `let` 声明变量时，变量被绑定到了它被声明的那个代码块里。这时块作用域生效了，它表现出了和函数作用域相似的特征——出了块作用域，你就访问不到里面的变量。

const 关键字

const 关键字和 let 具备相同的生命周期特性——用 const 声明的变量，也会被绑定到块作用域上。像这样：

```
{
  const me = 'xiuyan'
  console.log(me) // 'xiuyan'
}

console.log(me) // 报错
```

const 与 let、var 之间的区别，大家需要引起重视：const 声明的变量，必须在声明同时被初始化，否则会报错：

```
const a // 这样就会报错
```

报错形式如下：

```
> const a
```

```
✖ Uncaught SyntaxError: Missing initializer in const declaration
```

const 声明的变量，在赋值过后，值不可以再被更改。否则同样会报错：

```
const me = 'xiuyan'
me = 'Bear'
```

报错形式如下：

```
> const me = 'xiuyan'
```

```
< undefined
```

```
> me = 'Bear'
```

```
✖ ▶ Uncaught TypeError: Assignment to constant variable.
    at <anonymous>:1:4
```

值得注意的是，这个规则在声明引用类型时有点不同——引用类型的属性值（包括数组的元素）可以被更改，只要你不修改引用的指向。比如这样：

```
const me = {
  name: 'xiuyan'
}

me.name = 'Bear' // 没问题
```

像这样修改 name 这个属性值，而 me 对象的引用仍然指向原有的内存地址，这种更改就可以被接受。而类似这种：

```
const me = {
  name: 'xiuyan'
}

me = {
  name: 'Bear'
} // 报错
```

这样就相当于重新给 `me` 赋值了，是在尝试把 `me` 的引用指向一个全新的对象、指向另一块内存空间，这种做法就是不被接受的。

总之，牢记一点——`const` 是用来被声明常量的，它的内存空间在哪个位置，这一点一开始就锁死了，不要尝试把 `const` 定义的变量指向新的内存空间。

暂时性死区

现在大家对块作用域、`let`&`const` 特性以及变量提升都有了自己的理解和把握，在这个基础上，我们来理解暂时性死区就是一件非常容易的事情了。

我们来看这样一段代码：

```
var me = 'xiuyan';

{
  me = 'bear';
  let me;
}
```

这样的代码也经常作为面试题来出。面试官会问你：这段代码的运行结果是啥？事实上，这段代码啥也运行不出来，它会报错：

```
> var me = 'xiuyan';

{
  me = 'bear';
  let me;
}
```

✖ ▶ Uncaught ReferenceError: Cannot access 'me' before initialization
at <anonymous>:4:5

一些同学可能会有点摸不着头脑：这个块作用域和函数作用域都是局部作用域。你说 `let` 声明前不可用，我理解。但是这个例子中，明明全局作用域也有一个 `me` 变量。代码块第一行这个 `me`，按照作用域规则，难道不能引用父级作用域里的 `me` 吗？为啥会报错呢？

这是因为 **ES6** 中有明确的规定：如果区块中存在 `let` 和 `const` 命令，这个区块对这些命令声明的变量，从一开始就形成了封闭作用域。假如我们尝试在声明前去使用这类变量，就会报错。

这一段会报错的危险区域，有一个专属的名字，叫“暂时性死区”。在我们的 **demo** 中，以红线为界，上面的区域就是暂时性死区：

```
{  
  me = 'bear';  
  let me;  
}
```

这个 demo 里，如果我们想成功引用全局的 `me` 变量，需要把 `let` 声明给去掉：

```
var me = 'xiuyan';  
  
{  
  me = 'bear';  
}
```

这时程序就能运行无阻了：

```
var me = 'xiuyan';  
  
{  
  me = 'bear';  
}  
  
"bear"
```

是不是觉得这个“死区”非常鸡贼？它并不意味着引擎感知不到 `me` 这个变量的存在，恰恰相反，它感知到了，而且它清楚地知道 `me` 是用 `let` 声明在当前块里的——正因如此，它才会给这个变量加上暂时性死区的限制。一旦我们把 `let` 关键字撤走了，它反而也不吭声了。

其实这也就是暂时性死区的本质：当我们进入当前作用域时，`let` 或者 `const` 声明的变量已经存在了——它们只是不允许被获取而已。要想获取它们，必须得等到代码执行到声明处。

```
}
```