**揭秘变量提升和 ES6 的暂时性死区**

DEARPORK [技术漫谈](javascript:void(0);) 7月7日

引用 ES6 规范作者 Allen Wirfs-Brock 一条最近的推特：

变量提升是一个陈旧且令人困惑的术语。甚至在 ES6 之前：变量提升的意思究竟是“提升至当前作用域顶部”还是“从嵌套的代码块中提升到最近的函数或脚本作用域中”？还是两者都有？

受 Allen 启发，本文提出了一种不同的方法来描述变量声明。

**声明：作用域与激活**

我建议将声明分为两个方面：

* 作用域：在哪可以看到一个声明的实体？这是一个静态特征。
* 激活：我何时可以访问实体？这是一个动态特征：有的实体在我们进入他们作用域的时候就可以被访问，其余的我们必须等待代码执行到它们的声明。

下面的表格总结了不同的声明如何处理这两个方面。“Duplicates”表示一个变量名是否允许在同一作用域声明两次。“Global prop.”表示一个在 **script** 标签（模块的前身）中的声明，在全局作用域中被执行时，是否会向全局对象添加属性。**TDZ** 意思是**暂时死区**（我们稍后解释）。函数声明在严格模式下是块作用域（例如在模块内部），但在非严格模式下是函数作用域。

以下部分更加详细地描述了其中一些结构的行为。

**const 和 let：暂时死区**

对于 JavaScript，TC39 需要决定如果在声明之前访问其直接作用域中的常量会发生什么：

{  
  console.log(x); *// 这里会发生什么？*  
  const x;  
}

一些可能的方案是：

1. 该变量名在包围当前作用域的作用域中解析。
2. 你会得到 undefined。
3. 报错。

方案（1）被否决，因为这种方案在该语言中没有先例。因此这对于 JavaScript 程序员并不直观。

方案（2）被否决，因为这样 x 将不是一个常量 —— 在声明前和声明后它将拥有不同的值。

let 与 const 一样使用了方案（3），所以它们工作方式相似并且很容易在它们之间切换。

进入变量作用域与执行声明之间的这段时间被称为该变量的**暂时死区**（TDZ）：

* 在此期间，该变量被认为是未初始化的（就好像它有一个特殊的值）。
* 如果你访问一个未初始化的变量，你会得到一个 ReferenceError。
* 一旦你执行到了变量声明，这个变量将被设置为初始化的值（通过赋值符号指定）或者 undefined —— 如果没有初始化的话。

以下代码阐释了暂时死区：

if (true) { *// 进入 `tmp` 的作用域，TDZ 开始*  
  *// `tmp` 未被初始化：*  
  assert.throws(() => (tmp = 'abc'), ReferenceError);  
  assert.throws(() => console.log(tmp), ReferenceError);  
  
  let tmp; *// TDZ 结束*  
  assert.equal(tmp, undefined);  
}

下一个例子表明暂时死区是真的暂时的（与时间有关）：

if (true) { *// 进入 `myVar` 作用域，TDZ 开始*  
  const func = () => {  
    console.log(myVar); *// 稍后执行*  
  };  
  
  *// 我们在 TDZ 中：*  
  *// 访问 `myVar` 造成 `ReferenceError`*  
  
  let myVar = 3; *// TDZ 结束*  
  func(); *// OK，在 TDZ 外调用*  
}

即使 func() 声明位于 myVar 声明之前且使用了该变量，我们仍然可以调用 func()。但是我们必须等到 myVar 的暂时死区结束之后。

**函数声明与提前激活**

无论一个函数声明在作用域内的什么位置，它都会在进入作用域时执行。这使你可以在foo() 函数声明前调用它。

assert.equal(foo(), 123); *// OK*  
function foo() { return 123; }

提前激活 foo() 意味着上述代码相当于：

function foo() { return 123; }  
assert.equal(foo(), 123);

如果你用 const 或 let 声明一个函数，那么它就不会被提前激活：在下面的例子中，你只能在 bar() 声明后调用它。

assert.throws(  
  () => bar(), *// 声明前*  
  ReferenceError);  
const bar = () => { return 123; };  
assert.equal(bar(), 123); *// 声明后*

**未提前激活的提前调用**

即使函数 g() 并未提前激活，它仍可以被前面的函数 f()（在同一作用域中）调用 —— 只要我们遵守以下规则：f() 必须在声明 g() 之后调用。

const f = () => g();  
const g = () => 123;  
*// 我们在 g() 声明后调用 f()：*  
assert.equal(f(), 123);

模块中的函数通常在模块体执行完后才被调用。所以在模块中，你很少需要担心函数的顺序。

最后，注意提前激活是怎样自动执行以维持上述规则的：当进入一个作用域时，在任何函数被调用前，所有的函数声明都会被先执行。

**提前激活的一个陷阱**

如果你依赖提前激活在声明前调用一个函数，那你需要注意它并不能访问未提前激活的数据。

funcDecl();  
const MY\_STR = 'abc';  
function funcDecl() {  
  assert.throws(  
    () => MY\_STR,  
    ReferenceError);  
}

如果你在 MY\_STR 声明之后调用 funcDecl() 问题将不复存在。

**提前激活的利弊**

我们已经看到提前激活有一个陷阱，你可以在不使用它的情况下获得大部分好处。因此，最好避免提前激活。但我对此说法并非十分认同，如前所述，我经常使用函数声明，因为我喜欢它们的语法。

**类声明不会提前激活**

类声明不会提前激活：

assert.throws(  
  () => new MyClass(),  
  ReferenceError);  
  
class MyClass {}  
  
assert.equal(new MyClass() instanceof MyClass, true);

为什么呢？看看下面的类声明：

class MyClass extends Object {}

extends 是可选的。它的操作数是个表达式。因此，你可以这样做：

const identity = x => x;  
class MyClass extends identity(Object) {}

计算这样的表达式必须在它被引用的地方完成。其它行为都会使人困惑。这解释了为什么类声明不提前激活。

**var：变量提升（部分提前激活）**

var 是 const 和 let（现在更建议使用这两种方式）之前一种更老的声明变量的方式。考虑以下 var 声明。

var x = 123;

这个声明包含两个部分：

* 声明 var x：用 var 声明的变量的作用域是最里面的包围函数，而不是最里面的包围块，就像大多数其他声明一样。这样的变量在它作用域的开头就已经激活并以 undefined初始化。
* 赋值 x = 123：赋值总是在适当位置执行。

以下代码演示了 var：

function f() {  
  *// 部分提前激活：*  
  assert.equal(x, undefined);  
  if (true) {  
    var x = 123;  
    *// 赋值已经执行*  
    assert.equal(x, 123);  
  }  
  *// 作用域为函数作用域，非块级作用域。*  
  assert.equal(x, 123);  
}

* 原文地址：https://2ality.com/2019/05/unpacking-hoisting.html
* 原文作者：Dr. Axel Rauschmayer
* 本文译者：DEARPORK
* 本文校对：csming1995, Jalan