ES6_01_块级绑定

1.块级声明

块级声明也就是让所声明的变量在指定块的作用域外无法被访问。块级作用域(又被称为词法作用域)在如下情况被创建:
1. 在一个函数内部
2. 在一个代码块(由一对花括号包裹)内部

块级作用域是很多类 C 语言的工作机制, ES6 引入块级声明,是为了给 JS 添加灵活性以及 与其他语言的一致性。

1.let 声明

let 声明的语法与 var 的语法一致。你基本上可以用 let 来代替 var 进行变量声明,但 会将变量的作用域限制在当前代码块中。由于 let 声明并不会被提升到当前代码块的顶部,因此你需要手动将 let 声明放置到顶部,以便让变量在整个 代码块内部可用

```
Users ▸ liuxiangyu ▸ Desktop ▸ JS xxx.js ▸
1   function getValue(condition) {
2       if (condition) {
3           let value = "blue";
4           // 其他代码
5           return value;
6       } else {
7           // value 在此处不可用
8           return null;
9       }
10      // value 在此处不可用
11  }
```

2.禁止重复声明

如果一个标识符已经在代码块内部被定义,那么在此代码块内使用同一个标识符进行 let 声 明就会导致抛出错误。

let 不能 在同一作用域内重复声明一个已有标识符,此处的 let 声明就会抛出错误

```
1   var count = 30;
2   // 语法错误
3   let count = 40;
```

```
let count: number
'count' has already been declared. (E011) jshint(E011)
Quick Fix...  Peek Problem
count = 40;
```

在嵌 套的作用域内使用 let 声明一个同名的新变量,则不会抛出错误

```
var count = 30; // 不会抛出错误
if (condition) {
    let count = 40;
// 其他代码
}
```

3.常量声明

在 ES6 中里也可以使用 const 语法进行声明。使用 const 声明的变量会被认为是常量(constant),意味着它们的值在被设置完成后就不能再被改变。正因为如此,所有的 const 变量都需要在声明时进行初始化

```
// 有效的常量
const maxItems = 30;
// 语法错误:未进行初始化
const name;
```

```
const name: any
const 'name' is initialized to 'undefined'. (E012) jshint(E012)
Quick Fix...  Peek Problem
name;
```

let 与 const 不同于 var 的另一个方面是在全局作用域上的表现

在全局作用域上使 用 var 时,它会创建一个新的全局变量,并成为全局对象(在浏览器中是 window)的一 个属性。这意味着使用 var 可能会无意覆盖一个已有的全局属性

然而若你在全局作用域上使用 let 或 const ,虽然在全局作用域上会创建新的绑定,但不 会有任何属性被添加到全局对象上。这也就意味着你不能使用 let 或 const 来覆盖一个全 局变量,你只能将其屏蔽

```
// 在浏览器中
let RegExp = "Hello!";
console.log(RegExp); // "Hello!"
console.log(window.RegExp ===
RegExp); // false
const ncz = "Hi!";
console.log(ncz); // "Hi!"
console.log("ncz" in window); // false
```

当你不想在全局对象上创建属性时,这种特性会让 let 与 const 在全局作用域中更安全

若想让代码能从全局对象中被访问,你仍然需要使用 var 。在浏览器中跨越帧或窗口去访问代码时,这种做法非常普遍。

4. 对比常量声明与 let 声明

常量声明与 let 声明一样,都是块级声明。这意味着常量在声明它们的语句块外部是无法访 问的,并且声明也不会被提升

```
if (condition) {
    const maxItems = 5;
// 其他代码
}
// maxItems 在此处无法访问
```

与 let 的另一个相似之处,是 const 声明会在同一作用域(全局或是函数作用域)内定义 一个已有变量时会抛出错误,无论是该变量此前是用 var 声明的,还是用 let 声明的

```
var message = "Hello!";
let age = 25;
// 二者均会抛出错误
const message = "Goodbye!";
const age = 30;
```

```
const message: "Goodbye!"
'message' has already been declared. (E011) jshint(E011)
Quick Fix...  Peek Problem
const message = "Goodbye!";
const age = 30;
```

```
const age: 30
'age' has already been declared. (E011) jshint(E011)
Quick Fix...  Peek Problem
const age = 30;
```

与其他语言的常量类似, 不能被再次赋值。然而与其他语言不同, JS 的常量 如果是一个对象,它所包含的值是可以被修改的

```
const person = {
    name: "Nicholas"
};
// 工作正常
person.name = "Greg";
// 抛出错误
person = {
    name: "Greg"
};
```

使用 let 或 const 声明的变量,在达到声明处之前都是无法访问的,试图访问会导致一个 引用错误,即使在通常是安全的操作时(例如使用 typeof 运算符),也是如此

```
if (condition) {
    console.log(typeof value); // 引用错误 let value = "blue";
}
```

使用 let 或 const 声明的变量,若试图在定义位置之前使用它,无论如何都不能避免暂时 性死区。而且正如上例所演示的,这甚至影响了通常安全的 typeof 运算符。然而,你可以 在变量被定义的代码块之外对该变量使用 typeof ,尽管其结果可能并非预期

```
console.log(typeof value); // "undefined"
if (condition) {
    let value = "blue";
}
```

当 typeof 运算符被使用时, value 并没有在暂时性死区内,因为这发生在定义 value 变 量的代码块外部。这意味着此时并没有绑定 value 变量,而 typeof 仅单纯返回了"undefined" 。

暂时性死区只是块级绑定的一个独特表现,而另一个独特表现则是在循环时使用它。

5.暂时性死区

循环中的块级绑定

```
for (var i = 0; i < 10; i++) {
    process(items[i]);
}
// i 在此处仍然可被访问
console.log(i); // 10
```

在 JS 中,循环结束后 i 仍然可被访问,因为 var 声明导致了变量提升。 若像如下代码那样换为使用 let ,则会看到预期行为:

```
for (let i = 0; i < 10; i++) {
    process(items[i]);
}
// i 在此处不可访问,抛出错误
console.log(i);
```

循环内的函数

长期以来, var 的特点使得循环变量在循环作用域之外仍然可被访问,于是在循环内创建函 数就变得很有问题。

```
var funcs = [];
for (var i = 0; i < 10; i++) {
    funcs.push(function()
{ console.log(i); });
}
funcs.forEach(function(func) {
    func(); // 输出数值 "10" 十次
});
```

你原本可能预期这段代码会输出 0 到 9 的数值,但它却在同一行将数值 10 输出了十次。 这是 因为变量 i 在循环的每次迭代中都被共享了,意味着循环内创建的那些函数都拥有对于同一 变量的引用。在循环结束后,变量 i 的值会是 10 ,因此当 console.log() 被调用时, 每次都打印出 10 。

为了修正这个问题,开发者在循环内使用立即调用函数表达式(IIFEs),以便在每次迭代中 强制创建变量的一个新副本,示例如下:

```
var funcs = [];
for (var i = 0; i < 10; i++) {
    funcs.push((function(value) {
        return function() {
            console.log(value);
        }
    }(i)));
}
funcs.forEach(function(func) {
    func(); // 从 0 到 9 依次输出
});
```

幸运的是,使用 let 与 const 的块级绑定可以在 ES6 中为你简化这个循环

let 声明通过有效模仿上例中 IIFE 的作用而简化了循环。在每次迭代中,都会创建一个新的 同名变量并对其进行初始化。这意味着你可以完全省略 IIFE 而获得预期的结果,就像这样:

```
var funcs = [];
for (let i = 0; i < 10; i++) {
    funcs.push(function() {
        console.log(i);
    });
}
funcs.forEach(function(func) {
    func(); // 从 0 到 9 依次输出
});
```

与使用 var 声明以及 IIFE 相比,这里代码能达到相同效果,但无疑更加简洁。在循环中 let 声明每次都创建了一个新的 i 变量,因此在循环内部创建的函数获得了各自的 i 副
本,而每个 i 副本的值都在每次循环迭代声明变量的时候被确定了。这种方式在 for-in 和 for-of 循环中同样适用,如下所示:

```
var funcs = [],
    object = {
        a: true,
        b: true,
        c: true
    };
for (let key in object) {
    funcs.push(function() {
        console.log(key);
    });
}
funcs.forEach(function(func) {
    func(); // 依次输出 "a"、 "b"、 "c"
});
```

本例中的 for-in 循环体现出了与 for 循环相同的行为。每次循环,一个新的 key 变量绑 定就被创建,因此每个函数都能够拥有它自身的 key 变量副本,结果每个函数都输出了一个 不同的值。而如果使用 var 来声明 key ,则所有函数都只会输出 "c"

重点: let 声明在循环内部的行为是在规范中特别定义的,而与不提升变 量声明的特征没有必然联系。事实上,在早期 let 的实现中并没有这种行为,它是后来 才添加的。

循环内的常量声明

ES6 规范没有明确禁止在循环中使用 const 声明,然而它会根据循环方式的不同而有不同行 为。在常规的 for 循环中,你可以在初始化时使用 const ,但循环会在你试图改变该变量 的值时抛出错误

```
var funcs = [];
// 在一次迭代后抛出错误
for (const i = 0; i < 10; i++) {
    funcs.push(function() {
        console.log(i);
    });
}
```

在此代码中, i 被声明为一个常量。循环的第一次迭代成功执行,此时 i 的值为 0 。在 i++ 执行时,一个错误会被抛出,因为该语句试图更改常量的值。因此,在循环中你只能使用 const 来声明一个不会被更改的变量

另一方面, const 变量在 for-in 或 for-of 循环中使用时,与 let 变量效果相同。因
此下面代码不会导致出错:

```
var funcs = [],
    object = {
        a: true,
        b: true,
        c: true
    };
// 不会导致错误
for (const key in object) {
    funcs.push(function() {
        console.log(key);
    });
}
funcs.forEach(function(func) {
    func(); // 依次输出 "a"、 "b"、 "c"
});
```

总结

let 与 const 块级绑定将词法作用域引入了 JS 。这两种声明方式都不会进行提升,并且 只会在声明它们的代码块内部存在。由于变量能够在必要位置被准确声明,其表现更加接近 其他语言,并且能减少无心错误的产生。作为一个副作用,你不能在变量声明位置之前访问它们,即便使用的是 typeof 这样的安全运算符。由于块级绑定存在暂时性死区(TDZ), 试图在声明位置之前访问它就会导致错误。

let 与 const 的表现在很多情况下都相似于 var ,然而在循环中就不是这样。在 for-in 与 for-of 循环中, let 与 const 都能在每一次迭代时创建一个新的绑定,这意味着在循 环体内创建的函数可以使用当前迭代所绑定的循环变量值(而不是像使用 var 那样,统一使 用循环结束时的变量值)。这一点在 for 循环中使用 let 声明时也成立,不过在 for 循 环中使用 const 声明则会导致错误。

块级绑定当前的最佳实践就是:在默认情况下使用 const ,而只在你知道变量值需要被更改 的情况下才使用 let 。这在代码中能确保基本层次的不可变性,有助于防止某些类型的错 误。