ES 6

momo

LET 命令

1 基本用法

- 只在所在的代码块内有效

```
{
  let a = 10;
  var b = 1;
}

a // ReferenceError: a is not defined.
b // 1
```

```
for (let i = 0; i < 10; i++) {
  // ...
}

console.log(i);
// ReferenceError: i is not defined
```

```
var a = [];
for (var i = 0; i < 10; i++) {
  a[i] = function () {
    console.log(i);
  };
}
a[6](); // 10
```

```
var a = [];
for (let i = 0; i < 10; i++) {
  a[i] = function () {
    console.log(i);
  };
}
a[6](); // 6
```

- for循环还有一个特别之处，就是设置循环变量的那部分是一个父作用域，而循环体内部是一个单独的子作用域。

```
for (let i = 0; i < 3; i++) {
  let i = 'abc';
  console.log(i);
}
// abc
// abc
// abc
```

2 不存在变量提升

- var命令会发生”变量提升“现象，即变量可以在声明之前使用，值为undefined。这种现象多多少少是有些奇怪的，按照一般的逻辑，变量应该在声明语句之后才可以使用。为了纠正这种现象，let命令改变了语法行为，它所声明的变量一定要在声明后使用，否则报错。

```
// var 的情况
console.log(foo); // 输出undefined
var foo = 2;

// let 的情况
console.log(bar); // 报错ReferenceError
let bar = 2;
```

3 暂时性死区

- 只要块级作用域内存在let命令，它所声明的变量就“绑定”（binding）这个区域，不再受外部的影响。
- 总之，在代码块内，使用let命令声明变量之前，该变量都是不可用的。这在语法上，称为“暂时性死区”（temporal dead zone，简称 TDZ）。

```
var tmp = 123;

if (true) {
  tmp = 'abc'; // ReferenceError
  let tmp;
}
```

```
if (true) {
  // TDZ开始
  tmp = 'abc'; // ReferenceError
  console.log(tmp); // ReferenceError

  let tmp; // TDZ结束
  console.log(tmp); // undefined

  tmp = 123;
  console.log(tmp); // 123
}
```

```
typeof x; // ReferenceError
let x;
```

```
typeof undeclared_variable // "undefined"
```

```
function bar(x = y, y = 2) {
  return [x, y];
}

bar(); // 报错
```

```
function bar(x = 2, y = x) {
  return [x, y];
}
bar(); // [2, 2]
```

```
// 不报错
var x = x;

// 报错
let x = x;
// ReferenceError: x is not defined
```

- ES6 规定暂时性死区和let、const语句不出现变量提升，主要是为了减少运行时错误，防止在变量声明前就使用这个变量，从而导致意料之外的行为。这样的错误在 ES5 是很常见的，现在有了这种规定，避免此类错误就很容易了。
- 总之，暂时性死区的本质就是，只要一进入当前作用域，所要使用的变量就已经存在了，但是不可获取，只有等到声明变量的那一行代码出现，才可以获取和使用该变量。

4 不允许重复声明

```
// 报错
function func() {
  let a = 10;
  var a = 1;
}

// 报错
function func() {
  let a = 10;
  let a = 1;
}
```

```
function func(arg) {
  let arg; // 报错
}

function func(arg) {
  {
    let arg; // 不报错
  }
}
```

块级作用域

1为什么需要块级作用域？

- ES5 只有全局作用域和函数作用域，没有块级作用域，这带来很多不合理的场景。
- 第一种场景，内层变量可能会覆盖外层变量。

```
var tmp = new Date();

function f() {
  console.log(tmp);
  if (false) {
    var tmp = 'hello world';
  }
}

f(); // undefined
```

- 第二种场景，用来计数的循环变量泄露为全局变量。

```
var s = 'hello';

for (var i = 0; i < s.length; i++) {
  console.log(s[i]);
}

console.log(i); // 5
```

2 ES6 的块级作用域

```
function f1() {
  let n = 5;
  if (true) {
    let n = 10;
  }
  console.log(n); // 5
}
```

- ES6 允许块级作用域的任意嵌套。

```
{{{{{let insane = 'Hello World'}}}}};
```

- 外层作用域无法读取内层作用域的变量。

```
{{{{
  {let insane = 'Hello World'}
  console.log(insane); // 报错
}}}};
```

- 内层作用域可以定义外层作用域的同名变量。

```
{{{{
  let insane = 'Hello World';
  {let insane = 'Hello World'}
}}}};
```

- 块级作用域的出现，实际上使得获得广泛应用的立即执行函数表达式（IIFE）不再必要了。

```
// IIFE 写法
(function () {
  var tmp = ...;
  ...
}());

// 块级作用域写法
{
  let tmp = ...;
  ...
}
```

块级作用域与函数声明

- ES6 规定，块级作用域之中，函数声明语句的行为类似于let，在块级作用域之外不可引用。

```
function f() { console.log('I am outside!'); }

(function () {
  if (false) {
    // 重复声明一次函数f
    function f() { console.log('I am inside!'); }
  }

  f();
}());
```

- 上面代码在 ES5 中运行，会得到“I am inside!”，因为在if内声明的函数f会被提升到函数头部，实际运行的代码如下。

```
// ES5 环境
function f() { console.log('I am outside!'); }

(function () {
  function f() { console.log('I am inside!'); }
  if (false) {
  }
  f();
}());
```

- ES6 就完全不一样了，理论上会得到“I am outside!”。因为块级作用域内声明的函数类似于let，对作用域之外没有影响。但是，如果你真的在 ES6 浏览器中运行一下上面的代码，是会报错的，这是为什么呢？
- 原来，如果改变了块级作用域内声明的函数的处理规则，显然会对老代码产生很大影响。为了减轻因此产生的不兼容问题，ES6 在附录 B里面规定，浏览器的实现可以不遵守上面的规定，有自己的行为方式。
- 允许在块级作用域内声明函数。
- 函数声明类似于var，即会提升到全局作用域或函数作用域的头部。
- 同时，函数声明还会提升到所在的块级作用域的头部。
- 注意，上面三条规则只对 ES6 的浏览器实现有效，其他环境的实现不用遵守，还是将块级作用域的函数声明当作let处理。

- 根据这三条规则，在浏览器的 ES6 环境中，块级作用域内声明的函数，行为类似于var声明的变量。

```
// 浏览器的 ES6 环境
function f() { console.log('I am outside!'); }

(function () {
  if (false) {
    // 重复声明一次函数f
    function f() { console.log('I am inside!'); }
  }

  f();
}());
// Uncaught TypeError: f is not a function
```

- 上面的代码在符合 ES6 的浏览器中，都会报错，因为实际运行的是下面的代码。

```
// 浏览器的 ES6 环境
function f() { console.log('I am outside!'); }
(function () {
  var f = undefined;
  if (false) {
    function f() { console.log('I am inside!'); }
  }

  f();
}());
// Uncaught TypeError: f is not a function
```

- 考虑到环境导致的行为差异太大，应该避免在块级作用域内声明函数。如果确实需要，也应该写成函数表达式，而不是函数声明语句。

```
// 函数声明语句
{
  let a = 'secret';
  function f() {
    return a;
  }
}

// 函数表达式
{
  let a = 'secret';
  let f = function () {
    return a;
  };
}
```

- 另外，还有一个需要注意的地方。ES6 的块级作用域允许声明函数的规则，只在使用大括号的情况下成立，如果没有使用大括号，就会报错。

```
// 不报错
'use strict';
if (true) {
  function f() {}
}

// 报错
'use strict';
if (true)
  function f() {}
```

CONST命令

1 基本用法

- const声明一个只读的常量。一旦声明，常量的值就不能改变。

```
const PI = 3.1415;
PI // 3.1415

PI = 3;
// TypeError: Assignment to constant variable.
```

- const声明的变量不得改变值，这意味着，const一旦声明变量，就必须立即初始化，不能留到以后赋值。

```
const foo;
// SyntaxError: Missing initializer in const declaration
```

- const的作用域与let命令相同：只在声明所在的块级作用域内有效。

```
if (true) {
  const MAX = 5;
}

MAX // Uncaught ReferenceError: MAX is not defined
```

- const命令声明的常量也是不提升，同样存在暂时性死区，只能在声明的位置后面使用。

```
if (true) {
  console.log(MAX); // ReferenceError
  const MAX = 5;
}
```

- const声明的常量，也与let一样不可重复声明。

```
var message = "Hello!";
let age = 25;

// 以下两行都会报错
const message = "Goodbye!";
const age = 30;
```

2 本质

- const实际上保证的，并不是变量的值不得改动，而是变量指向的那个内存地址不得改动。对于简单类型的数据（数值、字符串、布尔值），值就保存在变量指向的那个内存地址，因此等同于常量。但对于复合类型的数据（主要是对象和数组），变量指向的内存地址，保存的只是一个指针，const只能保证这个指针是固定的，至于它指向的数据结构是不是可变的，就完全不能控制了。

- 常量foo储存的是一个地址，这个地址指向一个对象。不可变的只是这个地址，即不能把foo指向另一个地址，但对象本身是可变的，所以依然可以为其添加新属性。

```
const foo = {};

// 为 foo 添加一个属性，可以成功
foo.prop = 123;
foo.prop // 123

// 将 foo 指向另一个对象，就会报错
foo = {}; // TypeError: "foo" is read-only
```

- 常量a是一个数组，这个数组本身是可写的，但是如果将另一个数组赋值给a，就会报错。

```
const a = [];
a.push('Hello'); // 可执行
a.length = 0;    // 可执行
a = ['Dave'];    // 报错
```

3 ES6声明变量的六种方法

- Var (es5)
- Function (es5)

- Let
- Const
- Import
- Class

待续