01-第一章 ECMAScript6 let const 变量解构赋值

ECMAscript 6

ECMAScript 和 JavaScript 到底是什么关系？

> 1996 年 11 月，JavaScript 的创造者 Netscape 公司，决定将 JavaScript 提交给国际标准化组织 ECMA，希望这种语言能够成为国际标准。次年，ECMA 发布 262 号标准文件（ECMA-262）的第一版，规定了浏览器脚本语言的标准，并将这种语言称为 ECMAScript，这个版本就是 1.0 版。
> ECMAScript 和 JavaScript 的关系是，前者是后者的规格，后者是前者的一种实现（另外的 ECMAScript 方言还有 Jscript 和 ActionScript）

一、块级 let 命令

1、 基本用法

> ES6 新增了let命令，用来声明变量。它的用法类似于var，但是所声明的变量，

a、 只在let命令所在的代码块内有效 作用于 块级作用域

```
1. {var a = 10 }
2. console.log( a );//10
3.
4. { let b = 20}
5. console.log( b );/ReferenceError
```

```
for(var i=0;i<5;i++){
   console.log(i);
}
alert(i);//5


for(let i=0;i<5;i++){
   console.log(i);
 }
 alert(i);//ReferenceError
```

b、`不存在变量提升`。提前使用就会报错

> 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。

```
console.log(a);//undefined
    var a = 20;

    console.log(b);//ReferenceError
    let b = 30;
```

c、`暂时性死区`。不可重复定义

> 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`,也不影响外部。

`不可改变let变量值`,修改块级变量的值会报错

```
1.
2.    function fn(){
3.
4.        var a = 10;
5.        let a = 100;//SyntaxError: Identifier 'a' has already been d
   eclared
6.
7.    }
8.
9.    fn();
```

二、块级 常量 const

> `const` 命令 跟 `let` 命令 基本一致，只是const 声明就应赋值否则报错，且不可

```
1. const a;//SyntaxError
```

修改const的值会报错

```
1. const a=20;
2. a = 100;//SyntaxError
```

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

```
  let a = {};
  a.index = 123;
  console.log(a.index);

let a = [];
  a[0] = 123;
  console.log(a.index);
```

三、块级作用域

> `let` 实际上为 JavaScript 新增了块级作用域。
> 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。

```
//块级作用域

(function(){
    var a = 12;
})();

//块级作用域写法

{
    let a = 123;
}
```

四、数组的解构赋值

> ES6 允许按照一定模式，从 `数组和对象` 中提取值，对变量进行赋值，这被称为 `解构` （ Destructuring ）。
>
> 以往赋值

```
let a = 1;
let b = 2;
let c = 3;

  console.log(a);//1
```

解构（1）

```
let [a,b,c] = [1,2,3];
        console.log(a);//1
```

解构（2）

```
let [,,y] = [1,2,3];
    console.log( y );//3
```

解构（3）

```
let [foo,[[bar],baz]] = [1,[[2],3]];
    console.log(bar);//2
```

不完全解构（4）

```
     let [x]=[];
    console.log(x);
    let [a,b] = [1];//undefined;
    console.log(a);//1
    console.log(b);//undefined;
```

以上两种情况都属于解构不成功，foo的值都会等于`undefined`。

如果等号的右边不是数组（或者严格地说，`不是可遍历的结构`），那么将会`报错`。

```
    let [foo]=1;
    let[foo] = false;
    let [foo] = 'hello';
    let [foo] = NaN;
    let [foo] = undefined;
    let [foo] = null;
    let [foo] = {};
```

五、数组解构赋值默认值

- 1

```
        let [foo = 123] = [];
        console.log(foo);//123

        let [foo =123] = ['hello'];
        console.log(foo);//hello
```

- 2

```
let [x,y=100] = [1];
  console.log(x,y)//1 100
   let [a,b=100] = [1,undefined];
   console.log(a,b);//1 100
```

- 3

> 注意，ES6 内部使用严格相等运算符（`===`），判断一个位置是否有值。所以，如果一个数组成员不严格等于`undefined`，默认值是不会生效的。

```
        let [x = 1] = [undefined];
        console.log(x);//1
        let [y = 100] = [null];
        console.log(y);//null
```

上面代码中，如果一个数组成员是null，默认值就不会生效，因为null不严格等于undefined。

- 4

```
function fn(){
    console.log('aaa');
}
let [x = f() ] = [1];
```

上面代码中，因为x能取到值，所以函数f根本不会执行

六、对象的解构赋值

> 解构不仅可以用于数组，还可以用于对象
> 对象的解构与数组有一个重要的不同。数组的元素是按次序排列的，变量的取值由它的位置决定；而对象的属性没有次序，`变量必须与属性同名，才能取到正确的值`。

- a

```
let {f,b} = {foo:'aaa',bar:'bbb'};
        console.log(f);//undefined
```

- b

```


 let {foo,bar} = {foo:'aaa',bar:'bbb'};
    console.log( foo);//'aaa'
    console.log(bar);//'bbb'
```

- C
  `对象的解构本质`是下面形式的简写

```
let {foo:foo,bar:bar} = { foo:'aaa',bar:'bbb'};
        console.log(foo);//'aaa'
        console.log(bar);//'bbb'
```

> 也就是说，对象的解构赋值的内部机制，是先找到同名属性，然后再 `赋给对应的变量` ！！！。真正被赋值的是后者，而不是前者。

```
let {foo:baz,} = {foo:'aaa',bar:'bbb'};
        //console.log(foo);//ReferenceError
        console.log(baz);//'aaa'
```

- D

> 如果变量名与属性名不一致，必须写成这样

```
let {foo:baz} = {foo:'aaa',bar:'bbb'};
console.log(baz);

let obj = {first:'hello',last:'word'};
let {first:f,last:l} = obj;
console.log(f);//hello
console.log(l);//world
```

- e `嵌套赋值`

```
        let obj = {};
        let arr = [];

        ({foo:obj.prop,bar:arr[0]} = {foo:123,bar:true});
        console.log(obj);//{prop: 123}
        console.log(arr);//[true]
```

七、对象的解构赋值`默认值`

```
var {x = 3} = {};
    console.log(x);//2
    var {x,y = 5 } = {x:1};
    console.log(x);//1
    console.log(y);//5
    var { message: msg = 'hello'} = {};
    console.log(msg);//'hello'
```

`注意!!!`默认值使用`{x = 3}`而不是{x:3} ,中间是等号

–

默认值生效的条件是，对象的属性值严格等于undefined。

```
        var {x = 3} = {x: undefined};
         console.log( x );// 3
        var {x = 5} = {x:null};
        console.log(x);//null
```

八、函数的解构赋值

- 1

```
function add([x,y]){

    return x + y;
}

var re = add([1,2]);
console.log( re );//3
```

```
function add({x,y}){

    return x + y;
}

var re = add({x:1,y:2});
console.log( re );//3
```

九、函数解构中数组默认值

- 数组 `默认值`

```
function add([x=1,y=2]){ //

        return [x,y];
    }

    var re = add([5,5]);
    console.log( re );//[5, 5]
    var re = add([]);
    console.log( re );//[1, 2]
    var re = add();
    console.log( re );//报错
```

- 数组 `传递undefined`

```

function add([x=1,y=2] = []){//默认实参为 []

        return [x,y];
    }

    var re = add([5,5]);
    console.log( re );//[5, 5]
    var re = add([]);
    console.log( re );//[1, 2]
    var re = add();
    console.log( re );//[1, 2]

```

- 指定 `传递有值`

```
function add([x,y] = [0,1]){

        return [x,y];
    }

    var re = add([1,2]);
    console.log( re );//[1, 2]
    var re = add([]);
    console.log( re );//[undefined, undefined]
    var re = add();
    console.log( re );//[0, 1]
```

这里是为参数指定默认值，而不是为变量x和y指定默认值

```
function add([x=1,y=2] = []){

        return [x,y];
    }

    var re = add([1,2]);
    console.log( re );//[1, 2]
    var re = add([]);
    console.log( re );//[undefined, undefined]
    var re = add();
    console.log( re );//[0, 1]
```

十、对象默认值的函数解构

- 对象指定`默认形参`

```
function move({x = 0,y=0}){
        return [x,y];
    }

    var re = move({x:3,y:8});
    console.log(re);//[3, 8]
    var re = move({x:3});
    console.log(re);//[3, 0]
    var re = move({});
    console.log(re);//[0, 0]
    var re = move();//TypeError
```

注意这里是`{x = 0}`不是 {x:0}

- 对象指定`默认实参`

```
function move({x = 0,y=0} = {}){//{}是默认实参
        return [x,y];
    }

     var re = move();
     console.log(re)//[0,0]
```

- 注意，下面的写法会得到不一样的结果

```
 function move({x,y}={x:0,y:0}){
        return [x,y];

    }
    var r = move({x:3,y:8});
    console.log(r);//[3, 8]
    var r = move({x:3});
    console.log(r);//[3,undefined]
    var r = move({});
    console.log(r);//[undefind]
    var r = move();
    console.log(r);//[0,0]
```

总结

```
let 命令
    let 作用域作用
            ---作用域代码块
    不存在变量提升
            ---先声明在使用
    暂时性死区
            --不可重复定义
const 常量，跟let基本一致，
    let 作用域作用
            ---作用域代码块
    需定义即赋值,不可修改值
    不存在变量提升
            ---先声明在使用
    暂时性死区

            --不可重复定义
块作用域代替自执行函数

数组的解构赋值
    let [x = 20] = [];
    let [x = fn()] = [];
数组解构默认值
    --严格的（===）undefined,才能赋值默认值

对象的解构赋值
    a let {age} = {age:123};
    b let {age,index} = {age:123,name:'二狗'};

    C 对象解构赋值本质
    D 变量名与属性名不一致情况
    let {age:bar,index} = {age:888,index:666};

    e 嵌套赋值

对象解构默认值
    let {index,age} = {};
    let {index = 123,age =20} = {}
    let {message:msg = 123} = {};


函数参数的解构赋值
    数组参数
        function fn([x,y])

    对象参数
        function fn({index,age})

```

```
49.
50.                 默认值
51.
52.                     数组变量默认值
53.                     数组传递undefined默认值
54.                     数组传递有值默认值
55.
56.                     对象变量默认值
57.                     对象传递undefined 默认值
58.                     对象传递有值默认值
```