04-第四章 对象的拓展 Symbol数据结构

ECMAscript 6

对象拓展

一、变量属性函数属性

ES6 允许直接写入变量和函数，作为对象的属性和方法。这样的书写更加简洁。

变量属性

```
const foo = 'bar';
    const baz = {foo};
    console.log(baz);//{foo: "bar"}
    //等同于
    const baz = {foo:foo};
```

属性名为变量名, 属性值为变量的值

函数属性

```
function fn(){
        console.log('ok');

    }
    let obj = {fn}
    obj.fn();//ok
```

```
const obj = {
        fn(){
            console.log('ok')
        }
    }
    const obj = {
        fn:function(){
            console.log('ok')
        }
    }
```

二、表达式属性名

javascript定义对象的属性，有两种方式

> 一 使用`标识符`作为属性名
> 二 使用`表达式`作为属性名

```
//方法一
        var obj = {};
            obj.foo = true;
        //方法二
        obj['abc'] = 123;
        obj['hello world'] = 88888;
        console.log(obj['hello world']);//88888
        obj['x'+'y'] = 'yyyyyy';
        console.log(obj['xy']);//yyyyyy
```

对象直接量内部定义属性，在`ES5`只能使用（ `标识符` ）定义属性

```
let obj = {
    foo:true,
    abc:123
}
```

在`ES6`能使用`表达式属性`

```
var  obj = {
    ['bbb']:'abababb',
     ['a'+'b']:'abababb'
};


```

表达式中的字符串没有引起来会被当做`变量`

```
var  obj = {
    [index]:'二狗',//ReferenceError: index is not defined
    ['bbb']:'abababb',

};
```

会变的属性名

```
var index = 'aaa';
var  obj = {
    [index]:'二狗',
    ['bbb']:'abababab',

};

console.log( obj.aaa);//二狗
console.log( obj[index]);//二狗
```

会变的方法名

```
var index = 'ECMA';
var obj = {
    ['a'+'b'](){
        console.log('ok');
    },
    [index](){
        console.log(index);
    }
}
obj.ab();//ok
obj[index]();//ECMA
obj.ECMA();//ECMA
```

> 注意：一

```
var foo = 'hello';
var bar = {[foo]};//SyntaxError
```

> 注意：二

```
let obj1 = {a:111};
    let obj2 = {b:222};

    let myObject = {
        [obj1]:'val1',
        [obj2]:'val2',
    };

    console.log( myObject );//{[object Object]: "val2"}
```

[obj1]和[obj2]得到的都是`[object Object]`，所以`[obj1]`会把`[obj2]`覆盖掉，而myObject最后只有一个`[object Object]`属性

三、方法的name属性

> 函数的name属性，返回函数名。对象方法也是函数，因此也有name属性。

```
function fn(){}
    console.log(fn.name);//fn
//---------------------
    var obj = {
        show(){}
    }
    console.log(obj.show.name);//show
```

> 对象的方法取值函数（ `getter` ）和存值函数（ `setter` ）的name属性不是在该方法上，而是该方法的属性的描述对象的get和set属性上面，返回值是方法名前加上 `get` 和 `set`

```
const obj = {
    get foo(){},
    set foo(val){}
}

console.log( obj.foo.name )//TypeError

const  descriptor = Object.getOwnPropertyDescriptor(obj,'foo');
console.log( descriptor );
console.log(descriptor.get.name);//get foo
console.log(descriptor.set.name);//set foo
```

有两种特殊情况：

> bind方法创造的函数，name属性返回 `bound` 加上原函数的名字；
> `new Function` 构造函数创造的函数，name属性返回 `anonymous`

```
function fn(){}
var r = fn.bind().name;
console.log(r);//'bound fn'
console.log( (new Function).name );//'anonymous'
```

四、Object.is() 判断2个值是否相等

ES5 只有两个值判断是否相等

> 相等运算符（ `==` ）
> 严格相等运算符（ `===` ）
> 它们都有缺点

ES6 提出“Same-value equality”（ 同值相等 ）算法，Object.is就是部署这个算法的新方法。与严格比较运算符（ === ）的行为基本一致

> 不同之处只有两个：一是 +0 不等于 -0 ，二是 NaN 等于自身。

```
console.log( +0===-0 );//true
    console.log( NaN===NaN );//false
```

五、Object.assign() 拷贝对象属性

> Object.assign 方法用于对象的合并，将源对象（ source ）的可枚举属性（不拷贝继承属性），复制到目标对象（ target ）。并且返回目标对象

```
const target = {a:1};
const source2 = {b:2};
const source3 = {c:3};

Object.assign(target,source2,source3);
console.log( target );//{a: 1, b: 2, c: 3}
```

注意，如果目标对象与源对象有同名属性，或多个源对象有同名属性，则后面的属性会覆盖前面的属性

- （1）浅拷贝

> Object.assign方法实行的是 浅拷贝 ，而不是深拷贝。如果源对象某个属性的值是对象，那么目标对象拷贝得到的是这个对象的引用。

```
const obj1 = {a:{
        name:'二狗'
    }};

    const obj2 = Object.assign({},obj1);
    console.log( obj2);//{a: Object}
    obj2.a.name = '旺财';
    console.log( obj1.a.name );//旺财
```

- （2）取值器处理

> Object.assign只能进行值的复制，如果要复制的值是一个取值函数，那么将求值后再复制

```
const obj = {
        get foo(){ return 1;}
    }
    const target = {};
    Object.assign(target,obj);
    console.log( target );//{foo: 1}
```

- （3）运用

```
 function fn(name){ this.name = name;}
    fn.prototype={
            show(){ console.log(this.name) }
    }
    function fn2(name){fn.call(this,name) }
    Object.assign(fn2.prototype,fn.prototype);
    var obj = new fn2('二狗');
    obj.show();//二狗
```

六、对象属性的遍历方法

ES6 一共有 5 种方法可以遍历对象的属性。

> - （1）`for in`

or…in循环遍历对象自身的和继承的可枚举属性（不含 Symbol 属性）

- （2）`Object.keys(obj)`

返回一个数组，包括对象自身的（不含继承的）所有`可枚举属性`（不含 Symbol 属性）的键名。

- （3）`Object.getOwnPropertyNames(obj)`

返回一个数组，包含对象`自身的所有属性`（不含 Symbol 属性，但是`包括不可枚举属性`）的键名。

- （4）`Object.getOwnPropertySymbols(obj)`
  返回一个数组，包含对象自身的所有 Symbol 属性的键名
- （5）`Reflect.ownKeys(obj)`
- Reflect.ownKeys返回一个数组，包含对象自身的所有`键名`，不管`键名`是`Symbol`或字符串，`不管是否可枚举`

七、Object.setPrototypeOf(),Ojbect.setPrototypeOf()

> Object.setPrototypeOf()（写操作）、
> Object.getPrototypeOf()（读操作）、
> Object.create()（生成操作）

`Object.setPrototypeOf(` object, prototype `)` 设置对象prototype

> Object.setPrototypeOf({},null);

```
var obj = {age:20 }
        var obj2 = Object.setPrototypeOf(obj,{name:'hello'});
        console.log(obj2.age);//20
        console.log(obj2.name);//hello
```

```
let proto = {};
        var obj = {x:20 }
        Object.setPrototypeOf(obj,proto);
        proto.y=40;
        proto.z=60;
        console.log(obj.x);//20
        console.log(obj.y);//40
        console.log(obj.z);//60
```

`Ojbect.getPrototypeOf(` obj `)` 获取对象原型

```
function fn(){}
        var obj = new fn();
        var proto = Object.getPrototypeOf(obj) === fn.prototype;
        console.log( proto );//true
```

如果参数不是对象，会被自动转为对象。

```
Object.getPrototypeOf(1);
    //等同于Object.getPrototypeOf( new Number(1) );
    Object.getPrototypeOf('hello');
    //等同于Object.getPrototypeOf( new String('hello') );
    Object.getPrototypeOf(true);
    //等同于Object.getPrototypeOf(new Boolean(true) );
```

如果参数是`undefined`或`null`，它们无法转为对象，所以会报错。

```
Object.getPrototypeOf(null);//TypeError
Object.getPrototypeOf(undefined);//TypeError
```

八、`super`关键字，对象原型

```
const proto = {
        foo:'hello'
    }
    const obj = {
        find(){
            console.log( super.foo);
        }
    }
    Object.setPrototypeOf(obj,proto);
    obj.find();//hello
```

```
 const obj = {
        set super_foo(val){

            super.foo = val;
            console.log(super.foo);
        },
        get super_foo(){
            console.log(super.foo);
        }
    }
    var r = Object.setPrototypeOf(obj,proto);
    obj.super_foo = 888;
    console.log(r === proto);//false
```

注意，super关键字表示原型对象时，只能用在对象的方法之中，否则都会报错。

```
const obj = {//报错
        foo:super.foo
    };

//----------------------------
const obj = {//报错
        find:function(){
            console.log( super.foo);
        }
    }
//-------------------------
const obj = {//报错
        foo:()=>super.foo;
    }

```

super对象方法中`this`指向对象;

```
const proto = {
        name:'proto',
        foo:function(){
            console.log(this);
        }
    }
    const obj = {
        name:'二狗',
        find(){
            //super.foo();//{name: "二狗"}
             //Object.getPrototypeOf(obj).foo.call(this);//{name: "二狗"}
            this.foo();//{name: "二狗"}

        }
    };

    Object.setPrototypeOf(obj,proto);
    obj.find();//
```

JavaScript 引擎内部，super.foo等同于`Object.getPrototypeOf(this).foo.call(this)`（方法）。

九、Object.keys() Object.values() Object.entries()

`Object.keys` 返回对象`键名`列表数组

> ES5 引入了Object.keys方法，返回一个数组，成员是参数对象自身的（不含继承的）`所有可遍历`（enumerable）`属性`的`键名`

```
var obj = {
        index:888,
        name:'hello'
    }

    var r = Object.keys(obj);
    console.log(r);//["index", "name"]
```

`Object.values` 返回对象`键值`列表数组

> 返回一个数组，成员是参数对象自身的（不含继承的）`所有可遍历`（enumerable）`属性`的`键名`

```
var obj = {
        index:888,
        name:'hello'
    }

    var r = Object.values(obj);
    console.log(r);//[888, "hello"]
```

`Object.entries` 返回对象`键值对`列表数组

> 返回一个数组，成员是参数对象自身的（不含继承的）所有可遍历（enumerable）属性的`键值对数组`

```
var obj = {
        index:888,
        name:'hello'
    }

    var r = Object.entries(obj);
    console.log(r);//[['index',888],['name','hello']]
```

Object.entries的基本用途是遍历对象的属性。

```
for ([key,val] of Object.entries(obj)) {
       console.log( [key,val]);
       //["index", 888]
       //["name", "hello"]
    }
```

十、对象的拓展运算符

> `ES2017` 将这个运算符引入了对象。
> 《数组的扩展》一章中，已经介绍过扩展运算符（`...`）；

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

- （1）解构赋值中的 `...`

```
 let {x,y,...z} = {x:100,y:200,a:300,b:400};
    console.log(x,y,z);//100 200 {a: 300, b: 400}
```

由于解构赋值要求等号右边是一个对象，所以如果等号右边是`undefined`或`null`，就会报错，因为它们无法转为对象。

```
let { x, y, ...z } = null; // 运行时错误
let { x, y, ...z } = undefined; // 运行时错误
```

解构赋值必须是`最后一个参数`，否则会报错。

```
let {...z,x,y} = {x:100,y:200,a:300,b:400};//SyntaxError
let {x,...z,y} = {x:100,y:200,a:300,b:400};//SyntaxError
```

函数对象参数中的...拓展参数

```
function fn({x,y,...z}){
        console.log(x,y);//10 30
        console.log(z);//{z: 20, f: 40}

    }

    fn( {x:10,z:20,y:30,f:40} );
```

解构赋值的拷贝是`浅拷贝`

```
let obj =  {x:100,y:200,b:{index:1} };
    let {...x} = obj;
    console.log(x);//{x: 100, y: 200, b: {…}}
    obj.b.index = 888;
    console.log(x.b.index);888
```

另外，扩展运算符的解构赋值，`不能复制继承自原型对象的属性`

```
let obj_1 = {a:1};
    let obj_2 = {b:2};
    obj_2.__proto__ = obj_1;
    let {...obj_3} = obj_2;
    console.log(obj_3);//{b: 2}
     console.log(obj_3.a);//undefined
```

- （2）拓展运算符
  扩展运算符（`...`）用于取出参数对象的所有可遍历属性，拷贝到当前对象之中

```
let a = {
        x:1,
        y:2,
        z:3
    }
    console.log( {...a});//{x: 1, y: 2, z: 3}
    let {x} = {...a};
    console.log(x);//1
```

拷贝对象及其原型

```
var proto = {
        age:25,
    }
    var obj = {
        name:'二狗'
    }

    const clone = Object.create(//方法三
        Object.getPrototypeOf(obj),
        Object.getOwnPropertyDescriptors(obj)
    );

    Object.setPrototypeOf(obj,proto);
    // const clone = {//方法一
    //     __proto__:Object.getPrototypeOf(obj),
    //     ...obj
    // }

    // const clone = Object.assign(//方法二
    //     Object.getPrototypeOf(obj),
    //     obj
    // );



    console.log(clone.age);//25
    console.log(clone.name);//二狗
```

合并对象

```
var obj ={
        age:20
    }
    var obj2 = {
        name:'二狗'
    }

    var obj3 = { ...obj,...obj2 };
    console.log(obj3);//{age: 20, name: "二狗"}
```

```
var obj = {
        age:25
    }
    var obj2 = {
        name:'二狗'
    }
    var obj3 ={...obj,x:888,...obj2,...{y:4,x:5}};
    console.log(obj3);//{age: 25, x: 5, name: "二狗", y: 4}
```

二、 Symbol

ES5 的对象属性名都是字符串，这容易造成属性名的冲突，ES6 引入了一种新的原始数据类型 `Symbol` ，表示 `独一无二` 的值
它是 JavaScript 语言的第 `七` 种数据类型，前六种是： `undefined` 、 `null` 、布尔值（ `Boolean` ）、字符串（ `String` ）、数值（ `Number` ）、对象（ `Object` ）

一、Symbol概述

Symbol 值通过 `Symbol函数` 生成,这就是说，对象的属性名现在可以有两种类型，一种是原来就有的字符串，另一种就是新增的 Symbol 类型。

```
let s = Symbol();
console.log( typeof s);//symbol
```

凡是属性名属于 Symbol 类型，就都是 `独一无二` 的，可以保证不会与其他属性名产生冲突

```
       var symbol1 = Symbol();
        var symbol2 = Symbol();
        console.log( symbol1 === symbol2 );//false
```

`注意` ，Symbol函数前不能使用 `new` 命令，否则会报错。这是因为生成的 Symbol 是一个原始类型的值， `不是对象` 。也就是说，由于 Symbol 值不是对象，所以不能添加属性。基本上，它是一种类似于字符串的数据类型
Symbol函数可以接受一个 `参数` ，表示对 Symbol 实例的描述，主要是为了在控制台显示

```
    var symbol1 = Symbol('ok');
       var symbol2 = Symbol('ok');
       console.log( symbol1 === symbol2 );//false
```

Symbol 值 `不能进行运算` ，会报错

```
var s = Symbol('hello world');
        'abc'+s;//Cannot convert a Symbol value to a string
        123+s;//Cannot convert a Symbol value to a string
        var val =  `you ${s}`;//Cannot convert a Symbol value to a string
```

Symbol 值可以显式转为 `字符串` , `布尔值`

```
var s = Symbol('hello world');
          var r = s.toString(s);
          console.log( r);//Symbol(hello world)
          console.log( typeof r);//string
//--------------------
var s = Symbol('ok');
          var r = String(s);
          console.log(r);//Symbol(ok)

//--------------------------
var s = Symbol('ok');
          var r = Boolean(s);
          console.log(r);//true
```

但是不能转为`数值`

```
var s = Symbol('ok');
          var r = Number(s);//Cannot convert a Symbol value to a number
```

二、Symbol作为属性名

> 由于每一个 Symbol 值都是不相等的，这意味着 Symbol 值可以作为标识符，用于对象的属性名，就能保证不会出现同名的属性。防止某一个键被不小心改写或覆盖。

```
var symbol = Symbol();


         //一
          let a ={};
          a[symbol] = 'hello';
          console.log(a[symbol]);//hello

         //二
          let b = {
             [symbol]:'world'
          };
          console.log( b[symbol]);//world

          //三
          let c = {};
          Object.defineProperty(c,symbol,{
             value:'hello world'
          });
          console.log( c);//{Symbol(): "hello world"}
```

使用 Symbol 值定义属性时，Symbol 值必须放`在方括号之中`

```
    var symbol = Symbol();
        var symbol2 = Symbol();

        let obj = {
            [symbol](){
                console.log('ok')
            },
            [symbol2](){
                console.log('symbol')
            }
        }
        obj[symbol]();//ok
        obj[symbol2]();//symbol
```

三、Symbol属性名遍历

> Symbol 作为属性名，该属性不会出现在`for...in`、`for...of`循环中，也不会被`Object.keys()`、`Object.getOwnPropertyNames()`、`JSON.stringify()`返回。但是，它 也不是私有属性

`Object.getOwnPropertySymbols()` 获取指定对象的所有 Symbol 属性名。

返回一个数组，当前对象的所有用 Symbol的属性名

```
var symbol = Symbol('ok1');
        var symbol2 = Symbol('ok2');

        let obj = {
            [symbol](){
                console.log('ok')
            },
            [symbol2](){
                console.log('symbol')
            }
        }

        var r = Object.getOwnPropertySymbols(obj);
        console.log(r);//[Symbol(ok1), Symbol(ok2)]
```

`Reflect.ownKeys`方法可以返回所有类型的键名，包括常规键名和`Symbol`键名。

四、Symbol.for() Symbol.keyFor()

> 想重复使用同一个 Symbol 值，`Symbol.for()`它接受 一个字符串 作为参数，然后全局中搜索有没有以该参数作为名称的 Symbol 值。如果有，就返回这个 Symbol 值，否则在被登记在全局环境中创建一个 Symbol 值并返回.供全局搜索

如果你调用Symbol.for(“cat”)30 次，每次都会返回`同一个 Symbol 值`，但是调用Symbol(“cat”)30 次，会返回 30 个`不同的 Symbol 值`

```
1.          console.log( Symbol.for('ok') === Symbol.for('ok') );//true
2.         console.log( Symbol('ok') === Symbol('ok') );//false
```

由于`Symbol()没有登记机`制，所以每次调用都会返回一个不同的值

Symbol.keyFor() Symbol 类型值的是否被登记; 是返回`key`，否则`undefined`

```
1. var sym = Symbol.for('ok');
2.         console.log(Symbol.keyFor(sym) );//ok 已登记
3.
4.         var sym = Symbol('ok');
5.         console.log( Symbol.keyFor(sym) );//undefined
```