深入理解JS对象的深度克隆及多种方式实现

jCodeLife
0.098 2020.08.08 21:33:00 字数 2,285 阅读 17

关于对象的深度克隆网上都会讲到，但看过很多资料、文章觉得讲得不太全、零零散散、漏这漏那，特写此文一来汇总；二来更透彻的掌握深度克隆。
我们知道
JavaScript的数据类型分两大类：

1. 原始值类型，包括：number、string、boolean、null、undefined
2. 引用值类型，包括：对象（object）、函数（function）、数组（array）

当然ES6 引入了一种新的数据类型Symbol，表示独一无二的值

在JS对象中可包含所有数据类型中的任意一些类型，当我们拷贝对象时，必须要考虑到所有的数据类型的拷贝。所以我们的目标就是：**实现JS中所有数据类型的深度拷贝！**包括function，包括symbol

首先，通过typeof判断是原始值、object、function还是symbol；

```
function deepClone(origin, target) {
    //origin:要被拷贝的对象
    var target = target || {};
    for (var prop in origin) {
        if (origin.hasOwnProperty(prop){
            if (typeof (origin[prop]) === "object") {

            } else if (typeof (origin[prop]) === "function") {

            } else if (typeof (origin[prop]) === "symbol") {

            } else {
                //除了object、function、symbol，剩下都是直接赋值的原始值
                target[prop] = origin[prop];
            }
        }
    }
}
```

备注：

- for in循环用来遍历对象属性的，但是会遍历到原型上的
- origin.hasOwnProperty()用于过滤，看看属性到底是不是自己的（除去原型的）
- typeof后除了object、function、symbol，剩下都是直接赋值的原始值，包括number、string、boolean

接着，通过Object.prototype.toString.call()判断object类型是数组、对象还是null

```
function deepClone(origin, target) {
    //origin:要被拷贝的对象
```

jCodeLife
总资产4 (约0.42元)

深入call,apply,bind到手动封装
阅读 40

深入理解JS对象的深度克隆及多种方式实现
阅读 14

区分数组还是对象的四种方法
阅读 1

推荐阅读

27岁女白领，公开夫妻私生活引热议：纵欲上瘾，正在榨干年轻人
阅读 35,404

面试官：小伙子，你连Java集合都讲不清楚，怎么就敢开口要8K呀？
阅读 19,577

名家笔下的绝美爱情句子☁
阅读 747

人间疾苦✨
阅读 1,215

活着就要折腾
阅读 2,475

```
                } else if (Object.prototype.toString.call(origin[prop]) == "[object Object]") {
                    //普通对象
                } else {
                    //null
                }

            } else if (typeof (origin[prop]) === "function") {//函数

            } else if (typeof (origin[prop]) === "symbol") {

            } else {
                //除了object、function、symbol，剩下都是直接赋值的原始值
                target[prop] = origin[prop];
            }
        }
    }
}
```

然后，根据是数组还是对象建立相应的数组或对象；但是因为数组和对象一样，可以存放所以类型的变量，所以这两种数据类型得用到递归，调用本身函数deepClone()

```
function deepClone(origin, target) {
    //origin:要被拷贝的对象
    var target = target || {};
    for (var prop in origin) {
        if (origin.hasOwnProperty(prop)) {
            if (typeof (origin[prop]) === "object") {//对象
                if (Object.prototype.toString.call(origin[prop]) == "[object Array]") {
                    //数组
                    target[prop] = [];
                    deepClone(origin[prop],target[prop]);
                } else if (Object.prototype.toString.call(origin[prop]) == "[object Object]") {
                    //普通对象
                    target[prop] = {};
                    deepClone(origin[prop],target[prop]);
                } else {
                    //null
                    origin[prop] = null;
                }

            } else if (typeof (origin[prop]) === "function") {//函数

            } else if (typeof (origin[prop]) === "symbol") {

            } else {
                //除了object、function、symbol，剩下都是直接赋值的原始值
                target[prop] = origin[prop];
            }
        }
    }
  return target;
}
```

这里有些人会有个疑问，就是当是数组时，我递归调用deepClone(origin[prop],target[prop]);传递的数组能使用for in遍历？答案是肯定的！for in也可以遍历数组，origin[prop]相当于是origin[0]、origin[1]、origin[2]、origin[3]....

接下来，深度克隆function
有两种方法克隆function

- 通过eval()
- 通过new Function()

```
var a = function(){alert(1)}
var b = eval("0,"+a);//方法一
var c = new Function("return "+a)();//方法二
```

但是如果函数上面附有许多静态属性，我们可以封装一个专门的函数来实现函数的深度拷贝：

```
var copyFn = function (fn) {
    var result = eval("0," + fn);
    for (var i in fn) {
        result[i] = fn[i]
    }
    return result
}
```


最后，实现symbol类型的拷贝：

温馨提示：不考虑拷贝symbol类型的可以跳过这部分，这部分也比较长

先来了解一下symbol吧

Symbol是ES6 引入了一种新的数据类型，用于表示独一无二的值

在ES6出来之前，对象属性名都是字符串，这容易造成属性名的冲突。比如，你使用了一个他人提供的对象，但又想为这个对象添加新的方法（mixin 模式），新方法的名字就有可能与现有方法产生冲突。如果有一种机制，保证每个属性的名字都是独一无二的就好了，这样就从根本上防止属性名的冲突。这就是 ES6 引入Symbol的原因。

Symbol 值通过Symbol函数生成。

```
let s = Symbol();
typeof s;// "symbol"
```

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。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。

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函数可以接受一个字符串作为参数，表示对 Symbol 实例的描述，主要是为了在控制台显示，或者转为字符串时，比较容易区分。

```
let s1 = Symbol('foo');
let s2 = Symbol('bar');

console.log(s1);// Symbol(foo)
console.log(s2);// Symbol(bar)

console.log(s1.toString()); // "Symbol(foo)"
console.log(s2.toString()); // "Symbol(bar)"
```

如果 Symbol 的参数是一个对象，就会调用该对象的toString方法（没有就找原型上的toString方法），将其转为字符串，然后才生成一个 Symbol 值。

```
const obj = {
  toString() {
    return 'abc';
  }
};
const sym = Symbol(obj);
console.log(sym); // Symbol(abc)
```

```
var a = Symbol("a");
var b = Symbol("a");
console.log(a == b);//false

b = a;
console.log(a == b);//true
```

那我们如何拷贝 Symol 类型呢？
有文章提到有个方法可以实现：

- 方法：Object.getOwnPropertySymbols(obj)
  Object.getOwnPropertySymbols(obj)，用于返回在给定对象自身上找到的所有 Symbol 类型的属性的数组。
  注意：因为所有的对象在初始化的时候不会包含任何的 Symbol，除非你在对象上赋值了 Symbol 否则Object.getOwnPropertySymbols()只会返回一个空的数组。

```
var obj = {};
var a = Symbol("a");
var b = Symbol.for("b");

obj[a] = "localSymbol";
obj[b] = "globalSymbol";

var objectSymbols = Object.getOwnPropertySymbols(obj);

console.log(objectSymbols.length); // 2
console.log(objectSymbols)         // [Symbol(a), Symbol(b)]
console.log(objectSymbols[0])      // Symbol(a)
```

但是我有点不解，这方法不就是找到对象身上的 Symbol 类型的属性吗，怎么就能实现symbol类型的拷贝呢？单纯查找区分的话，我在typeof的时候不是也能找到吗？不解！在写个代码测try try

```
var obj = {
    c: Symbol("cc"),
    fn:function(){
        // console.log(typeof this.c)
    }
};
for (var temp in obj){
    console.log(temp,typeof(obj[temp]))
}
```

打印：'c symbol”和'fn function'
证实for in确实可以遍历出对象中定义symbol类型的属性。
那我们猜测一下，通过obj.xx =Symbol()方式后面添加的属性，能不能通过forin拿出来

```
var obj = {
    c: Symbol("cc"),
    fn:function(){
        // console.log(typeof this.c)
    }
};
var a = Symbol("aa");
var b = Symbol.for("bb");
obj[a] = "localSymbol";
obj[b] = "globalSymbol";
for (var temp in obj){
    console.log(temp,typeof(obj[temp]))//'c symbol' 和'fn function'
}

console.log(obj[a],obj[b]) //localSymbol globalSymbol
console.log(typeof(obj[a]),typeof(obj[b]))//string string
```

再回过头来试试getOwnPropertySymbols代替forin找出对象的symbol属性

```
var obj = {
    c: Symbol("cc"),
    fn:function(){}
};
var a = Symbol("aa");
var b = Symbol.for("bb");
obj[a] = "localSymbol";
obj[b] = "globalSymbol";

var objectSymbols = Object.getOwnPropertySymbols(obj);
console.log(objectSymbols.length); // 2
console.log(objectSymbols)         // [Symbol(aa), Symbol(bb)]
console.log(typeof(objectSymbols[0]))      // symbol
```

发现通过getOwnPropertySymbols可以找出后面添加的symbol值
小结：

- 通过typeof找出symbol类型是直接在里面定义的，但是个人感觉这种值没多大意义，只是用于作一个对象的标识。
- 通过getOwnPropertySymbols找出对象声明之后通过对象[prop]=Symbol("xxx")的形式添加的属性。

所以分两个部分来处理symbol：

1. 对象声明时的symbol值：
   特点：for in遍历的到，并可以直接用typeof判断为symbol
2. 对象声明之后的symbol值：
   特点：通过for in遍历不出来，而是通过getOwnPropertySymbols获取这类symbol值。

到这完成了一大半，剩下就是怎么赋值拷贝了。
我的观点是，反正拷贝的结果就是看上去和功能都是一模一样，但是彼此独立不关联。
symbol的特性就是独一无二，所以天生不关联，我们只要考虑生成的时候传递的一样就行，即Symbol(context)中的context值。
上面我们讲过toString方法，我们深入试试传入不同类型时，返回的值有什么区别：

```
//Symbol传入不同类型的值
var _string = 'aa';
var _number = 1;
var _boolean = true;
var _undefined = undefined;
var _null= null;
var _array = [1,2];
var _fn = function(){a:1};
var _obj = {name:"alice"};
var _symbol = Symbol("s");
console.log(Symbol(_string).toString()); //Symbol(aa)
console.log(Symbol(_number).toString()); //Symbol(1)
console.log(Symbol(_boolean).toString()); //Symbol(true)
console.log(Symbol(_undefined).toString()); //Symbol()
console.log(Symbol(_null).toString()); //Symbol(null)
console.log(Symbol(_fn).toString()); //Symbol(function(){a:1})
console.log(Symbol(_obj).toString()); //Symbol([object Object])
console.log(Symbol(_symbol).toString()); //test.html:431 Uncaught TypeError: Cannot convert a Sym
```

我们通过Symbol传入不同类型的值，发现

`Symbol传入的值必须是字符串，如果不是，会调用String()发生隐式类型转换，将其转成字符串。特殊的Symbol(undefined).toString())返回是Symbol()`

那么可以写个方法：

```
    var tempArr = str.split("(");
    var arr = tempArr[1].split(")")[0];
    return Symbol(arr);
}
```

我们先来解决对象声明时的symbol：

```
function deepClone(origin, target) {
    //origin:要被拷贝的对象
    var target = target || {};
    for (var prop in origin) {
        if (origin.hasOwnProperty(prop)) {
            if (typeof (origin[prop]) === "object") {//对象
                if (Object.prototype.toString.call(origin[prop]) == "[object Array]") {
                    //数组
                    target[prop] = [];
                    deepClone(origin[prop], target[prop]);
                } else if (Object.prototype.toString.call(origin[prop]) == "[object Object]") {
                    //普通对象
                    target[prop] = {};
                    deepClone(origin[prop], target[prop]);
                } else {
                    //null
                    console.log(11111111111)
                    target[prop] = null;
                }

            } else if (typeof (origin[prop]) === "function") {//函数
                var _copyFn = function (fn) {
                    var result = new Function("return " + fn)();
                    for (var i in fn) {
                        result[i] = fn[i]
                    }
                    return result
                }
                target[prop] = _copyFn(origin[prop]);
            } else if (typeof (origin[prop]) === "symbol") {//里面的symbol
                target[prop] = _copySymbol(origin[prop]);
                console.log(target[prop])
            } else {
                    //除了object、function、symbol，剩下都是直接赋值的原始值number,string,boolean,ur
                target[prop] = origin[prop];


                //但有个特殊的是，string中可能有symbol，
                //那我可以通过getOwnPropertySymbols先找出来
                // var _tempArr = Object.getOwnPropertySymbols(origin);
                // console.log(_tempArr)
                // for (var i = 0;i<_tempArr.length;i++){
                //     _tempArr[i] ==
                // }
                // if (typeof (origin[prop] === "string"  )) {

                // } else {
                // }

            }
        }
    }
    function _copySymbol(val) {
        var str = val.toString();
        var tempArr = str.split("(");
        var arr = tempArr[1].split(")")[0];
        return Symbol(arr);
    }
    return target;
}

var student = {
    name: "alice",
    age: 12,
    isOldPerson: false,
    sex: undefined,
    money: null,
    grader: [{
```

```
    },
    key: Symbol("s1-key"),
    book: {
        English: true
    }
}
var a = Symbol('a')
var b = Symbol.for("b")
student[a] = "aaa"
student[b] = "bbb"

var res = deepClone(student);
```


可以看到，剩下最后一个问题了，就是就是后面添加的symbol，即在forin外头实现。

也是手写对象深度克隆的最终代码

```
function deepClone(origin, target) {
    //origin:要被拷贝的对象
    var target = target || {};
    for (var prop in origin) {
        if (origin.hasOwnProperty(prop)) {
            if (typeof (origin[prop]) === "object") {//对象
                if (Object.prototype.toString.call(origin[prop]) == "[object Array]") {
                    //数组
                    target[prop] = [];
                    deepClone(origin[prop], target[prop]);
                } else if (Object.prototype.toString.call(origin[prop]) == "[object Object]") {
                    //普通对象
                    target[prop] = {};
                    deepClone(origin[prop], target[prop]);
                } else {
                    //null
                    target[prop] = null;
                }

            } else if (typeof (origin[prop]) === "function") {//函数
```

```
                }
                target[prop] = _copyFn(origin[prop]);
            } else if (typeof (origin[prop]) === "symbol") {//里面的symbol
                target[prop] = _copySymbol(origin[prop]);
            } else {
                //除了object、function、symbol，剩下都是直接赋值的原始值number,string,boolean,undefi
                target[prop] = origin[prop];
            }
        }
    }
    function _copySymbol(val) {
        var str = val.toString();
        var tempArr = str.split("(");
        var arr = tempArr[1].split(")")[0];
        return Symbol(arr);
    }
    //通过getOwnPropertySymbols找出来的symbol
    var _symArr = Object.getOwnPropertySymbols(origin);
    if (_symArr.length) {//查找成功
        _symArr.forEach(symKey => {
                target[symKey] = origin[symKey];
        });
    }
    return target;
}
```

试试

```
var student = {
    name: "alice",
    age: 12,
    isOldPerson: false,
    sex: undefined,
    money: null,
    grader: [{
        English: 120,
        math: 80
    }, 100],
    study: function () {
        console.log("I am a student,I hava to study every day!")
    },
    key: Symbol("s1-key"),
    book: {
        English: true
    }
}
var a = Symbol('a')
var b = Symbol.for("b")
student[a] = "1111111111111"
student[b] = "222222222222222"
var res = deepClone(student);
```

测试结果

完美！
另外还有一方法，通过Reflect.ownKeys(origin),用于返回一个由目标对象自身的属性键组成的数组，这也能找到后面的symbol类型。你可以自己试试怎么实现。

其他方式实现深度克隆

除了上面手写实现以外，还有以下几个方法：

- 通过扩展运算符...

```
var a = {name:"aaa"}
var b = {...a}
```

- 通过合并对象的方法Object.assign

```
var newObj = Object.assign([],oldObj);
```

缺点：Object.assign只对顶层属性做了赋值，完全没有继续做递归把下层属性做深度拷贝。简而言之，只实现第一层深度拷贝，后续层次还是浅拷贝。

- 通过json的方法实现

```
var obj = {a:1};
var str = JSON.stringify(obj); //序列化对象
var newobj = JSON.parse(str); //还原
```

缺点：
如果obj里面有时间对象，则JSON.stringify后再JSON.parse的结果，时间将只是字符串的形式。而不是时间对象；
如果obj里有RegExp、Error对象，则序列化的结果将只得到空对象；
如果obj里有function，Symbol 类型，undefined，则序列化的结果会把函数或 undefined丢失；
如果obj里有NaN、Infinity和-Infinity，则序列化的结果会变成null

- jquery中的$.extend

```
var bob = {
    name: "Bob",
    age: 32
};

var bill = $.extend(true, {}, bob);
bill.name = "Bill";

console.log(bob.name);//Bob
console.log(bill.name);//Bill
```

参考文献：
深度克隆对象（不考虑function）-腾讯课堂渡一教育
JS对象深度克隆的实现—作者：兔子juan
js对象的深度克隆的三种方法（深拷贝）—作者：web全栈入门

如何实现一个深拷贝（考虑循环引用对象、和symbol类型）— 作者：Dream_Lee_1997

Object.getOwnPropertySymbols--MDN web docs

3人点赞 >

渡一领跑计划

"小礼物走一走，来简书关注我"

赞赏支持

还没有人赞赏，支持一下

jCodeLife 书山有路勤为径，学海无涯苦作舟

总资产4 (约0.42元) 共写了8.4W字 获得97个赞 共19个粉丝

被以下专题收入，发现更多相似内容

投稿管理

+收入我的专题

渡一领跑计划笔记

写下你的评论...

评论0 赞3