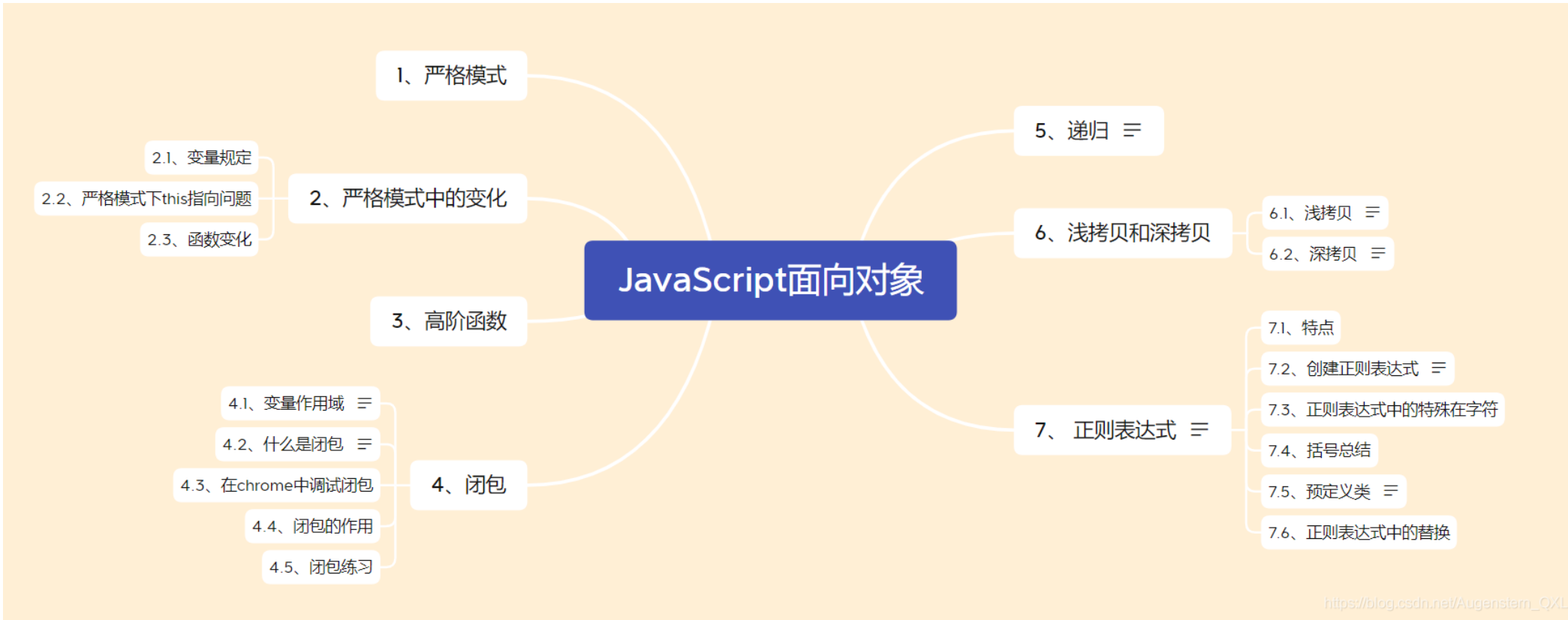
JavaScript提高班之ES6(七)

✍JavaScript帝国之行🔥

内容	地址
JavaScript基础大总结(一) 🔥	https://blog.csdn.net/Augenstern_QXL/article/details/119249534
JavaScript基础之函数与作用域(二) 🔥	https://blog.csdn.net/Augenstern_QXL/article/details/119250991
JavaScript基础之对象与内置对象(三) 🔥	https://blog.csdn.net/Augenstern_QXL/article/details/119250137
JavaScript进阶之DOM技术(四) 🔥	https://blog.csdn.net/Augenstern_QXL/article/details/115416921
JavaScript进阶之BOM技术(五) 🔥	https://blog.csdn.net/Augenstern_QXL/article/details/115406408
JavaScript提高之面向对象(六) 🔥	https://blog.csdn.net/Augenstern_QXL/article/details/115219073
JavaScript提高之ES6(七) 🔥	https://blog.csdn.net/Augenstern_QXL/article/details/115344398

✍目录总览



1、严格模式

- JavaScript 除了提供正常模式外，还提供了**严格模式**
- ES5 的严格模式是采用具有限制性 JavaScript 变体的一种方式，即在严格的条件下运行 JS 代码
- 严格模式在IE10 以上版本的浏览器才会被支持，旧版本浏览器会被忽略
- 严格模式对正常的JavaScript语义做了一些更改：
  - 消除了Javascript 语法的一些不合理、不严谨之处，减少了一些怪异行为
  - 消除代码运行的一些不安全之处，保证代码运行的安全
  - 提高编译器效率，增加运行速度
  - 禁用了在 ECMAScript 的未来版本中可能会定义的一些语法，为未来新版本的 Javascript 做好铺垫。比如一些保留字如：class, enum, export, extends, import, super 不能做变量名

1.1、开启严格模式

- 严格模式可以应用到**整个脚本**或**个别函数**中。
- 因此在使用时，我们可以将严格模式分为**为脚本开启严格模式**和**为函数开启严格模式两种情况**

1.1.2、为脚本开启严格模式

- 为整个脚本文件开启严格模式，需要**在所有语句之前放一个特定语句**
- `"use strict"` 或 `'use strict'`

```
<script>
    'user strict';
    console.log("这是严格模式。");
</script>
```

因为 "use strict" 加了引号，所以老版本的浏览器会把它当作一行普通字符串而忽略。

有的 script 基本是严格模式，有的 script 脚本是正常模式，这样不利于文件合并，所以可以将整个脚本文件放在一个立即执行的匿名函数之中。这样独立创建一个作用域而不影响其他 script 脚本文件。

```
<script>
    (function (){
        'use strict';
         var num = 10;
         function fn() {}
    })();
</script>
```

1.1.2、为函数开启严格模式

- 若要给某个函数开启严格模式，需要把 "use strict" 或 'use strict' 声明放在函数体所有语句之前

```
<body>
    <!-- 为整个脚本(script标签)开启严格模式 -->
    <script>
        'use strict';
        //    下面的js 代码就会按照严格模式执行代码
    </script>
    <script>
        (function() {
            'use strict';
        })();
    </script>
    <!-- 为某个函数开启严格模式 -->
    <script>
        // 此时只是给fn函数开启严格模式
        function fn() {
            'use strict';
            // 下面的代码按照严格模式执行
        }

        function fun() {
            // 里面的还是按照普通模式执行
        }
    </script>
</body>
```

- 将 "use strict" 放在函数体的第一行，则整个函数以 "严格模式"运行。

2、严格模式中的变化

- 严格模式对JavaScript的语法和行为，都做了一些改变

2.1、变量规定

- 在正常模式中，如果一个变量没有声明就赋值，默认是全局变量
- 严格模式禁止这种用法，变量都必须先用var 命令声明，然后再使用
- 严禁删除已经声明变量，例如，``delete x` 语法是错误的

```
<body>
    <script>
        'use strict';
        // 1. 我们的变量名必须先声明再使用
        // num = 10;
        // console.log(num);
        var num = 10;
        console.log(num);
        // 2.我们不能随意删除已经声明好的变量
        // delete num;

```

```
    </script>
</body>
```

2.2、严格模式下this指向问题

1. 以前在全局作用域函数中的 `this` 指向 `window` 对象
2. 严格模式下全局作用域中函数中的 `this` 是 undefined
3. 以前构造函数时不加 `new` 也可以调用，当普通函数， `this` 指向全局对象
4. 严格模式下，如果构造函数不加 `new` 调用， `this` 指向的是 `undefined` ，如果给它赋值，会报错
5. `new` 实例化的构造函数指向创建的对象实例
6. 定时器 `this` 还是指向 `window`
7. 事件、对象还是指向调用者

```
<body>
    <script>
        'use strict';
        //3. 严格模式下全局作用域中函数中的 this 是 undefined。
        function fn() {
            console.log(this); // undefined。

        }
        fn();
        //4. 严格模式下,如果 构造函数不加new调用, this 指向的是undefined 如果给他赋值则 会报错.
        function Star() {
            this.sex = '男';
        }
        // Star();
        var ldh = new Star();
        console.log(ldh.sex);
        //5. 定时器 this 还是指向 window
        setTimeout(function() {
            console.log(this);

        }, 2000);

    </script>
</body>
```

2.3、函数变化

1. 函数不能有重名的**参数**
2. 函数必须声明在顶层，新版本的JavaScript会引入“块级作用域”（ES6中已引入）。为了与新版本接轨，**不允许在非函数的代码块内声明函数**

```
<body>
    <script>
        'use strict';
        // 6. 严格模式下函数里面的参数不允许有重名
        function fn(a, a) {
           console.log(a + a);

        };
        // fn(1, 2);
        function fn() {}
    </script>
</body>
```

3、高阶函数

- 高阶函数是对其他函数进行操作的函数，它接收函数作为参数或将函数作为返回值输出

接收函数作为参数

```
<body>
    <div></div>
    <script>
        // 高阶函数- 函数可以作为参数传递
        function fn(a, b, callback) {
            console.log(a + b);
            callback && callback();
```

```
        }
        fn(1, 2, function() {
            console.log('我是最后调用的');

        });

    </script>
</body>
```

将函数作为返回值

```
<script>
    function fn(){
        return function() {}
    }
</script>
```

- 此时 fn 就是一个高阶函数
- 函数也是一种数据类型，同样可以作为参数，传递给另外一个参数使用。最典型的就是作为回调函数
- 同理函数也可以作为返回值传递回来

4、闭包

4.1、变量作用域

变量根据作用域的不同分为两种：全局变量和局部变量

1. 函数内部可以使用全局变量
2. 函数外部不可以使用局部变量
3. 当函数执行完毕，本作用域内的局部变量会销毁。

4.2、什么是闭包

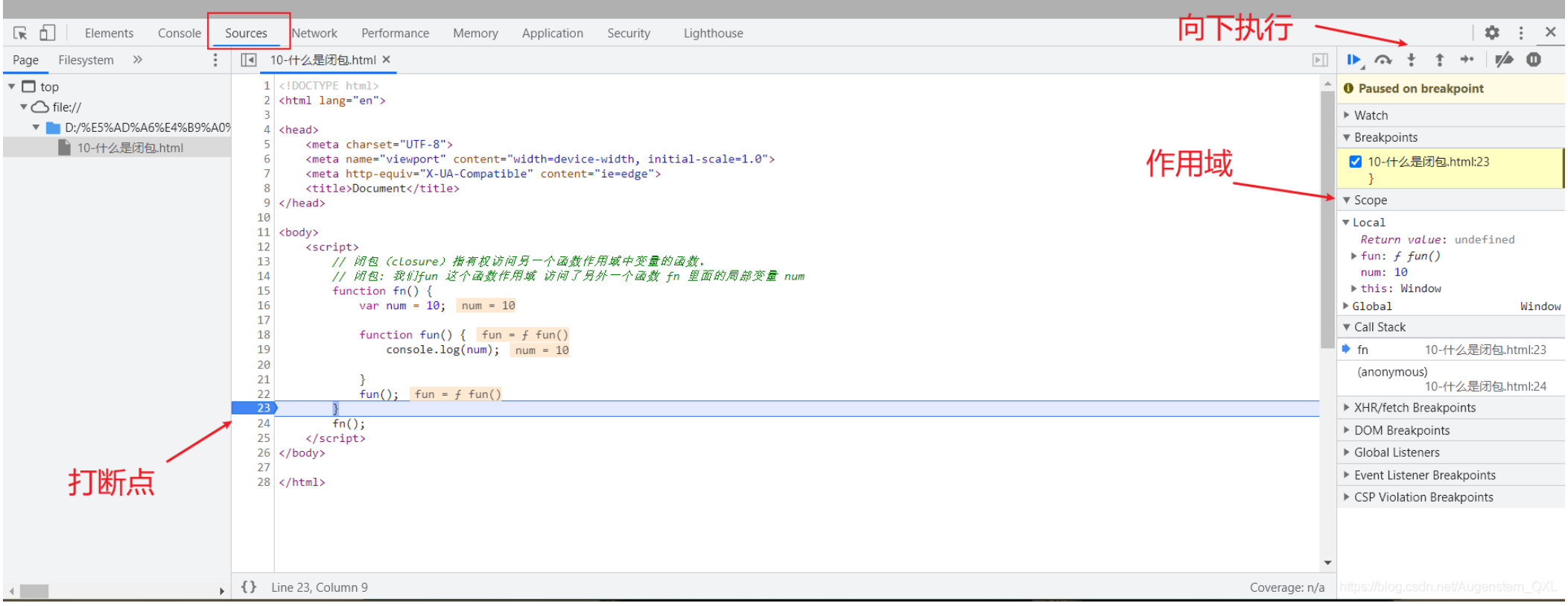
闭包指有权访问另一个函数作用域中的变量的函数

简单理解：一个作用域可以访问另外一个函数内部的局部变量

```
<body>
    <script>
        // 闭包（closure）指有权访问另一个函数作用域中变量的函数。
        // 闭包：我们fn2 这个函数作用域 访问了另外一个函数 fn1 里面的局部变量 num
        function fn1() {        // fn1就是闭包函数
            var num = 10;
            function fn2() {
                console.log(num);   //10
            }
            fn2();
        }
        fn1();
    </script>
</body>
```

4.3、在chrome中调试闭包

1. 打开浏览器，按 F12 键启动 chrome 调试工具。
2. 设置断点。
3. 找到 Scope 选项（Scope 作用域的意思）。
4. 当我们重新刷新页面，会进入断点调试，Scope 里面会有两个参数（global 全局作用域、local 局部作用域）。
5. 当执行到 fn2() 时，Scope 里面会多一个 Closure 参数，这就表明产生了闭包。



4.4、闭包的作用

- 延伸变量的作用范围

```
<body>
    <script>
        // 闭包（closure）指有权访问另一个函数作用域中变量的函数。
        // 一个作用域可以访问另外一个函数的局部变量
        // 我们fn 外面的作用域可以访问fn 内部的局部变量
        // 闭包的主要作用：延伸了变量的作用范围
        function fn() {
            var num = 10;
            return function() {
                console.log(num);
            }
        }
        var f = fn();
        f();
    </script>
</body>
```

4.5、闭包练习

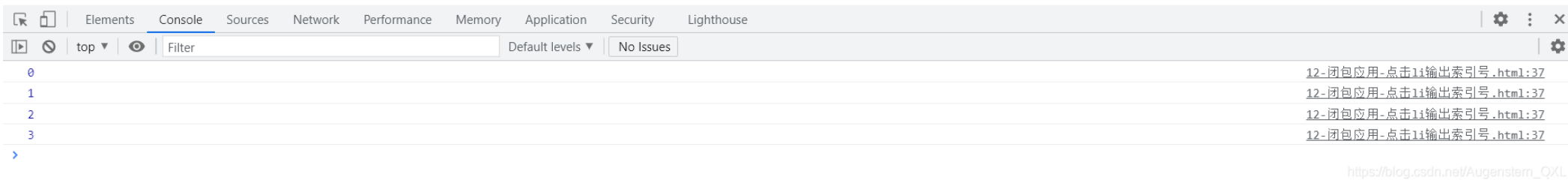
4.5.1、点击li输出索引号

```
<body>
    <ul class="nav">
        <li>榴莲</li>
        <li>臭豆腐</li>
        <li>鲱鱼罐头</li>
        <li>大猪蹄子</li>
    </ul>
    <script>
        // 闭包应用-点击li输出当前li的索引号
        // 1. 我们可以利用动态添加属性的方式
        var lis = document.querySelector('.nav').querySelectorAll('li');
        for (var i = 0; i < lis.length; i++) {
            lis[i].index = i;
            lis[i].onclick = function() {
                // console.log(i);
                console.log(this.index);

            }
        }
        // 2. 利用闭包的方式得到当前小li 的索引号 面试重点 but这里的闭包有内存泄漏，效率不如动态添加属性的方式高
        for (var i = 0; i < lis.length; i++) {
            // 利用for循环创建了4个立即执行函数
            // 立即执行函数也成为小闭包因为立即执行函数里面的任何一个函数都可以使用它的i这变量
            (function(i) {
                // console.log(i);
                lis[i].onclick = function() {
                    console.log(i);

                }
```

只要是在立即执行函数里面的，就可以使用立即执行函数里面的i

```
            })(i);
        }
    </script>
</body>
```

- 榴莲
- 臭豆腐
- 鲱鱼罐头
- 大猪蹄子



4.5.2、定时器中的闭包

```
<body>
    <ul class="nav">
        <li>榴莲</li>
        <li>臭豆腐</li>
        <li>鲱鱼罐头</li>
        <li>大猪蹄子</li>
    </ul>
    <script>
        // 闭包应用-3秒钟之后,打印所有li元素的内容
        var lis = document.querySelector('.nav').querySelectorAll('li');
        for (var i = 0; i < lis.length; i++) {
            (function(i) {
                setTimeout(function() {
                    console.log(lis[i].innerHTML);
                }, 3000)
            })(i);
        }
    </script>
</body>
```

- 榴莲
- 臭豆腐
- 鲱鱼罐头
- 大猪蹄子

5、递归

如果一个函数在内部可以调用其本身，那么这个函数就是递归函数

简单理解： 函数内部自己调用自己，这个函数就是递归函数

回调调的是其它的函数，递归是自己调自己并且要有出口，不然就是死循环、栈溢出

由于递归很容易发生"栈溢出"错误，所以必须要加退出条件 return

```

<body>
    <script>
        // 递归函数 : 函数内部自己调用自己, 这个函数就是递归函数
        var num = 1;

        function fn() {
            console.log('我要打印6句话');

            if (num == 6) {
                return; // 递归里面必须加退出条件
            }
            num++;
            fn();
        }
        fn();

```

```
    </script>
</body>
```

6、浅拷贝 和深拷贝

1. 浅拷贝只是拷贝一层，更深层次对象级别的只拷贝引用
2. 深拷贝拷贝多层，每一级别的数据都会拷贝
3. `Object.assign(target,....sources)` ES6新增方法可以浅拷贝

前面就是你要拷贝谁，后面是你要拷贝给哪个对象

浅拷贝：只是把地址拷贝给了对象
深拷贝：重新开辟一个空间

6.1、浅拷贝

```
// 浅拷贝只是拷贝一层，更深层次对象级别的只拷贝引用
var obj = {
    id: 1,
    name: 'andy',
    msg: {
        age: 18
    }
};
var o = {}
for(var k in obj){
    // k是属性名，obj[k]是属性值
    o[k] = obj.[k];
}
console.log(o);
// 浅拷贝语法糖
Object.assign(o,obj);
```

6.2、深拷贝

```
// 深拷贝拷贝多层，每一级别的数据都会拷贝
var obj = {
    id: 1,
    name: 'andy',
    msg: {
        age: 18
    }
    color: ['pink','red']
};
var o = {};
// 封装函数
function deepCopy(newobj,oldobj){
    for(var k in oldobj){
        // 判断属性值属于简单数据类型还是复杂数据类型
        // 1.获取属性值    oldobj[k]
        var item = obldobj[k];
        // 2.判断这个值是否是数组
        if(item instanceof Array){
            newobj[k] = [];
            deepCopy(newobj[k],item)
        }else if (item instanceof Object){
            // 3.判断这个值是否是对象
            newobj[k] = {};
            deepCopy(newobj[k],item)
        }else {
            // 4.属于简单数据类型
            newobj[k] = item;

        }
    }
}
deepCopy(o,obj);
```

把数组放在上面是因为array也是一个object，如果把数组放在下面就不会执行array那一块了

7、 正则表达式

正则表达式是用于匹配字符串中字符组合的模式。在JavaScript中，正则表达式也是对象。

正则表通常被用来检索、替换那些符合某个模式（规则）的文本，例如验证表单：用户名表单只能输入英文字母、数字或者下划线， 昵称输入框中可以输入中文(匹配)。此外，正则表达式还常用于过滤掉页面内容中的一些敏感词(替换)，或从字符串中获取我们想要的特定部分(提取)等 。

7.1、特点

- 实际开发，一般都是直接复制写好的正则表达式
- 但是要求会使用正则表达式并且根据自身实际情况修改正则表达式

7.2、创建正则表达式

在JavaScript中，可以通过两种方式创建正则表达式

1. 通过调用 RegExp 对象的构造函数创建　正则表达式：Regular Express

2. 通过字面量创建

7.2.1、通过调用 RegExp 对象的构造函数创建

通过调用 RegExp 对象的构造函数创建

```
var 变量名 = new RegExp(/表达式/);
```

7.2.2、通过字面量创建

通过字面量创建

```
var 变量名 = /表达式/;
```

注释中间放表达式就是正则字面量

7.2.3、测试正则表达式 test

- `test()` 正则对象方法，用于检测字符串是否符合该规则，该对象会返回 `true` 或 `false` ,其参数是测试字符串

```
regexObj.test(str)
```

- `regexObj` 写的是正则表达式
- `str` 我们要测试的文本
- 就是检测 `str` 文本是否符合我们写的正则表达式规范

示例

```
<body>
    <script>
        // 正则表达式在js中的使用

        // 1. 利用 RegExp对象来创建 正则表达式
        var regexp = new RegExp(/123/);
        console.log(regexp);

        // 2. 利用字面量创建 正则表达式
        var rg = /123/;
        // 3.test 方法用来检测字符串是否符合正则表达式要求的规范
        console.log(rg.test(123));
        console.log(rg.test('abc'));
    </script>
</body>
```

7.3、正则表达式中的特殊在字符

7.3.1、边界符

正则表达式中的边界符(位置符)用来提示字符所处的位置，主要有两个字符

边界符	说明
^	表示匹配行首的文本(以谁开始)
$	表示匹配行尾的文本(以谁结束)

如果^ 和 $ 在一起，表示必须是精确匹配

```
// 边界符 ^ $
var rg = /abc/;   //正则表达式里面不需要加引号，不管是数字型还是字符串型
```

```
// /abc/只要包含有abc这个字符串返回的都是true
console.log(rg.test('abc'));
console.log(rg.test('abcd'));
console.log(rg.test('aabcd'));

var reg = /^abc/;
console.log(reg.test('abc'));   //true
console.log(reg.test('abcd'));  // true
console.log(reg.test('aabcd')); // false

var reg1 = /^abc$/
// 以abc开头，以abc结尾，必须是abc
```

7.3.2、字符类

- 字符类表示有一系列字符可供选择，只要匹配其中一个就可以了
- 所有可供选择的字符都放在方括号内

①[] 方括号

```
/[abc]/.test('andy');     // true
```

后面的字符串只要包含 `abc` 中任意一个字符,都返回 `true`

②[-]方括号内部 范围符

任何一个数字则可以输入0-9

```
/^[a-z]$/.test()
```

方括号内部加上 - 表示范围，这里表示 a - z 26个英文字母都可以

③[^] 方括号内部 取反符 ^

放在[]外面的就是边界符，放在[]里面的是取反，取反即不能包含什么东西

```
/[^abc]/.test('andy')   // false
```

方括号内部加上 ^ 表示取反，只要包含方括号内的字符，都返回 `false`

注意和边界符 ^ 区别，边界符写到方括号外面

④字符组合

```
/[a-z1-9]/.test('andy')    // true
```

方括号内部可以使用字符组合，这里表示包含 a 到 z的26个英文字母和1到9的数字都可以

```
<body>
    <script>
        //var rg = /abc/;  只要包含abc就可以
        // 字符类: [] 表示有一系列字符可供选择，只要匹配其中一个就可以了
        var rg = /[abc]/; // 只要包含有a 或者 包含有b 或者包含有c 都返回为true
        console.log(rg.test('andy'));
        console.log(rg.test('baby'));
        console.log(rg.test('color'));
        console.log(rg.test('red'));
        var rg1 = /^[abc]$/; // 三选一 只有是a 或者是 b  或者是c 这三个字母才返回 true
        console.log(rg1.test('aa'));
        console.log(rg1.test('a'));
        console.log(rg1.test('b'));
        console.log(rg1.test('c'));
        console.log(rg1.test('abc'));
        console.log('------------------');

        var reg = /^[a-z]$/; // 26个英文字母任何一个字母返回 true  - 表示的是a 到z 的范围
        console.log(reg.test('a'));
        console.log(reg.test('z'));
        console.log(reg.test(1));
        console.log(reg.test('A'));
        // 字符组合
        var reg1 = /^[a-zA-Z0-9_-]$/; // 26个英文字母(大写和小写都可以)任何一个字母返回 true
        console.log(reg1.test('a'));
        console.log(reg1.test('B'));
        console.log(reg1.test(8));
        console.log(reg1.test('-'));
        console.log(reg1.test('_'));
```

```
        console.log(reg1.test('!'));
        console.log('----------------');
        // 如果中括号里面有^ 表示取反的意思 千万和 我们边界符 ^ 别混淆
        var reg2 = /^[^a-zA-Z0-9_-]$/;
        console.log(reg2.test('a'));
        console.log(reg2.test('B'));
        console.log(reg2.test(8));
        console.log(reg2.test('-'));
        console.log(reg2.test('_'));
        console.log(reg2.test('!'));
    </script>
</body>
```

7.3.3、量词符

量词符用来设定某个模式出现的次数

量词	说明
*	重复零次或更多次
+	重复一次或更多次
?	重复零次或一次
{n}	重复n次
{n,}	重复n次或更多次
{n,m}	重复n到m次

```
<body>
    <script>
        // 量词符：用来设定某个模式出现的次数
        // 简单理解：就是让下面的a这个字符重复多少次
        // var reg = /^a$/;


        //  * 相当于 >= 0 可以出现0次或者很多次
        // var reg = /^a*$/;
        // console.log(reg.test(''));
        // console.log(reg.test('a'));
        // console.log(reg.test('aaaa'));



        //  + 相当于 >= 1 可以出现1次或者很多次
        // var reg = /^a+$/;
        // console.log(reg.test('')); // false
        // console.log(reg.test('a')); // true
        // console.log(reg.test('aaaa')); // true

        //  ?  相当于 1 || 0
        // var reg = /^a?$/;
        // console.log(reg.test('')); // true
        // console.log(reg.test('a')); // true
        // console.log(reg.test('aaaa')); // false

        //  {3 } 就是重复3次
        // var reg = /^a{3}$/;
        // console.log(reg.test('')); // false
        // console.log(reg.test('a')); // false
        // console.log(reg.test('aaaa')); // false
        // console.log(reg.test('aaa')); // true
        //  {3, }  大于等于3
        var reg = /^a{3,}$/;
        console.log(reg.test('')); // false
        console.log(reg.test('a')); // false
        console.log(reg.test('aaaa')); // true
        console.log(reg.test('aaa')); // true
        //  {3,16}  大于等于3 并且 小于等于16
        var reg = /^a{3,6}$/;
        console.log(reg.test('')); // false
        console.log(reg.test('a')); // false
        console.log(reg.test('aaaa')); // true
        console.log(reg.test('aaa')); // true
```

```
        console.log(reg.test('aaaaaaa')); // false
    </script>
</body>
```

7.3.4、用户名验证

功能需求：

1. 如果用户名输入合法, 则后面提示信息为 : 用户名合法,并且颜色为绿色
2. 如果用户名输入不合法, 则后面提示信息为: 用户名不符合规范, 并且颜色为绿色

分析：

1. 用户名只能为英文字母,数字,下划线或者短横线组成, 并且用户名长度为 6~16位.
2. 首先准备好这种正则表达式模式 /$[a-zA-Z0-9-_]{6,16}^/
3. 当表单失去焦点就开始验证.
4. 如果符合正则规范, 则让后面的span标签添加 right 类.
5. 如果不符合正则规范, 则让后面的span标签添加 wrong 类.

```
<body>
    <input type="text" class="uname"> <span>请输入用户名</span>
    <script>
        //  量词是设定某个模式出现的次数
        var reg = /^[a-zA-Z0-9_-]{6,16}$/; // 这个模式用户只能输入英文字母 数字 下划线 短横线但是有边界符和[ ] 这就限定了只能多选1
        // {6,16}  中间不要有空格
        // console.log(reg.test('a'));
        // console.log(reg.test('8'));
        // console.log(reg.test('18'));
        // console.log(reg.test('aa'));
        // console.log('-------------');
        // console.log(reg.test('andy-red'));
        // console.log(reg.test('andy_red'));
        // console.log(reg.test('andy007'));
        // console.log(reg.test('andy!007'));
        var uname = document.querySelector('.uname');
        var span = document.querySelector('span');
        uname.onblur = function() {
            if (reg.test(this.value)) {
                console.log('正确的');
                span.className = 'right';
                span.innerHTML = '用户名格式输入正确';
            } else {
                console.log('错误的');
                span.className = 'wrong';
                span.innerHTML = '用户名格式输入不正确';
            }
        }
    </script>
</body>
```

7.4、括号总结

1. 大括号 量词符 里面面表示重复次数
2. 中括号 字符集合 匹配方括号中的任意字符
3. 小括号 表示优先级

```
// 中括号 字符集合 匹配方括号中的任意字符
var reg = /^[abc]$/;
// a || b || c
// 大括号 量词符 里面表示重复次数
var reg = /^abc{3}$/;   // 它只是让c 重复3次 abccc
// 小括号 表示优先级
var reg = /^(abc){3}$/;  //它是让 abc 重复3次
```

在线测试正则表达式：https://c.runoob.com/

7.5、预定义类

预定义类指的是 某些常见模式的简写写法

预定类	说明
\d	匹配0-9之间的任一数字，相当于[0-9]
\D	匹配所有0-9以外的字符，相当于[^ 0-9]
\w	匹配任意的字母、数字和下划线,相当于[A-Za-z0-9_]
\W	除所有字母、数字、和下划线以外的字符，相当于[^A-Za-z0-9_]
\s	匹配空格（包括换行符，制表符，空格符等），相当于[\t\t\n\v\f]
\S	匹配非空格的字符，相当于[^ \t\r\n\v\f]

7.5.1、表单验证

分析：

1.手机号码: `/^1[3|4|5|7|8][0-9]{9}$/`

2.QQ: `[1-9][0-9]{4,}` (腾讯QQ号从10000开始)

3.昵称是中文: `^[\u4e00-\u9fa5]{2,8}$`

```
<body>
    <script>
        // 座机号码验证:  全国座机号码  两种格式:   010-12345678  或者  0530-1234567
        // 正则里面的或者 符号  |
        // var reg = /^\d{3}-\d{8}|\d{4}-\d{7}$/;
        var reg = /^\d{3,4}-\d{7,8}$/;
    </script>
</body>
```

7.6、正则表达式中的替换

7.6.1、replace 替换

`replace()` 方法可以实现替换字符串操作，用来替换的参数可以是一个字符串或是一个正则表达式

```
stringObject.replace(regexp/substr,replacement)
```

1. 第一个参数: 被替换的字符串或者正则表达式
2. 第二个参数：替换为的字符串
3. 返回值是一个替换完毕的新字符串

```
// 替换 replace
var str = 'andy和red';
var newStr = str.replace('andy','baby');
var newStr = str.replace(/andy/,'baby');
```

7.6.2、正则表达式参数

```
/表达式/[switch]
```

`switch` 按照什么样的模式来匹配，有三种

- `g` : 全局匹配 global ignore
- `i` :忽略大小写
- `gi` : 全局匹配 + 忽略大小写