正则

字符串方法

```
indexOf('a',5)//从索引5开始从左向右查找
laseIndexOf('a',5)//从索引5开始从右向左查找

serach(reg)//和indexOf很像，优点在于可以使用正则
let str = 'huang0810';
str.search(/\d+/);//返回5

split()//不指定分隔符，默认以逗号分隔

match()//返回值为数组
```

正则：是一个处理字符串的规则

```
正则匹配:验证当前字符串是否符合某个规则
正则捕获:把一个字符串中符合规则的字符获取到
```

创建方式：

```
1，字面量方式
//var reg=/\d+/img;
2，构造函数创建
//var reg=new RegExp('元字符','修饰符')
//var reg=new RegExp('\\d+','img');
```

```
使用构造函数创建和字面量方式创建，最主要的区别是：构造函数创建，用到\需要
写'\\'才可以
使用构造函数方式写验证是否为有效数字的正则
//var reg=new RegExp('^-?([1-9]\\d+|\\d)(\\.\\d+)?$');
```

为什么要用构造函数创建正则?

```
有的时候需要在正则中动态加入一个变量的值，作为正则的一部分
var strClass='String';
```

```
var reg=new RegExp(`^\\[object ${strClass}\\]$`);
reg.test('[object String]');//true
```

正则表达式和字符串相互转换

```
var reg=/\d+/g;
var str=reg.toString();//输出 "/\d+/g"
reg = eval(str);//输出:  /\d+/g  字符串转正则
```

修饰符和元字符

```
let reg = /^\d+$/g
任何一个正则都是由元字符和修饰符组成的;
正则两个斜杠之间包起来的都是"元字符", 斜杠后面\的都是"修饰符"
```

常用的修饰符: `img`

```
i: ignoreCase 忽略大写小匹配
m: multiline 多行匹配
g: global 全局匹配
```

常见的元字符: 包括`量词元字符`、`特殊元字符`、普通元字符
`【量词元字符】:` 让其左边的字符出现多少次`

```
* : 出现零到多次
? : 出现零到一次
+ : 出现一到多次
{n} : 出现N次
{n,} : 出现n到多次
{n,m} : 出现n到m次
```

【特殊元字符】:

```
\d : 0~9之间的一个数字
\D : 非0~9之间的任意字符
\w : "数字、字母、下划线"的任意一个 //=> /[0-9a-zA-Z_]/等价于\w
\s : 匹配任意一个空白字符(包括\t制表符、TAB键)
\b : 匹配单词边界符 //'zhu-feng'(z左边、u右边、f左边、g右边是边界符) '???c'(c的左边和右边是边界符)
\n : 匹配一个换行符
 \ : 转义字符(把一个普通字符转义为特殊的字符,例如:\d, 把有特殊含义的转换为普通
```

意思，例如：\. 此处的点就不是任意字符，而是一个小数点）
. ： 不仅仅是小数点，代表除了\n以外的任意字符
^ ： 以某个元字符开头
$ ： 以某个元字符结尾
x|y ： x或者y中的任意一个(a|z...)，同[ab]
[xyz] ： x或者y或者z中的任意一个
[^xyz] ： 除了x\y\z以外的任意一个字符
[a-z] ： 获取a-z中的任意一个字符([0-9] 等价于\d ...)
[^a-z] ： 除了a-z的任意字符
() ： 正则的小分组，匹配一个小分组，计算是第几个分组的时候，我们从左向右找'('即可
(?:) ： 当前分组只匹配不捕获，不能和分组引用同时使用
(?=) ： 正向预查
(?!) ： 负向预查
...

【普通元字符】： 代表本身意义的元字符

只要在正则中出现的元字符（基于字面方式创建），除了特殊和量词元字符外，其余的都是普通元字符

元字符解读

一个正则设置了^和$，代表匹配的字符串只能是xxx，不加^和$代表字符串中只要包含xxx即可
^或者$只是一个修饰或者声明，不会占据字符串的位置

.

```
var reg=/^2.3$/;
reg.test('2.3')//true
reg.test('2+3')//true

var reg=/^2\.3$/;
reg.test('2.3')//true
reg.test('2+3')//false//使用转义字符把点转换为本身小数点的意思
```

转义字符：\

```
var reg=/^\\d$/;
reg.test('9');//false
reg.test('d');//false
reg.test('\d');//false
```

```
reg.test('\\d');//true
reg.test('\\\d');//true
reg.test('\\\\d');//false
reg.test('\\\\\d');//false
```

`x|y`

```
var reg=/^18|19$/;
//匹配的规则：以18开头或者以19结尾
//可以匹配的字符串为：'18' '19' '181' '189' '819' '119'
var reg=/^(18|19)$/;
//只能是18或者19
```

小括号`()`

```
正则中的小分组
作用：
1. 改变默认的优先级
2. 分组引用
3. 分组捕获
前两种情况，可以在小括号里加上?:，代表只匹配，不捕获
```

```
分组引用：\1、\2、\3...出现和第N个分组一模一样的内容
var reg=/^([a-z])([a-z])\2([a-z])$/;
//可以匹配的字符串为'food' 'week'等，分组引用必须和小分组同时使用时才有效果
var reg=/^[a-z][a-z]\2[a-z]$/;
reg.test('ab\2d');//true

var reg=/(\d)\1{2}/;//匹配三个一样的数字
reg.test('222');//true

var reg=/\d\1{2}/
reg.test('3\\1\\1')//false
reg.test('3\1\1')//true
```

中括号：`[]`

- 具有消磁作用，`单字元字符`出现在`[]`中，比如：`*`、`+`、`.`、`^`、`$`、`?`等，不再表示元字符的含义，而是表示这个字符本身。

```
let reg = /^[\d]+$/; //注意：\d在这里依然是0~9中的一个数字
```

```
let reg = /^[12-65]$/;
//这个正则的意思是 1或者2~6或者5
```

```
[xyz]:匹配x或y或z中的 "任意一个字符"
var reg=/^[.?+&]$/;//只能是这四个字符中的其中一个
reg.test('.?');//false
reg.test('&');//true
//只要在ASCII码表里连续出现的字符，都可以用这种方法表示

reg = /^[!-z]$/; //会匹配ASCII码表中从字符'!'到字符'z'之间的任意一个字符

也可以用十六进制表示上面的正则，写成reg = /^[\u0021-\u007a]$/
```

优先级

正则表达式`运算符优先级`：从高到低

```
1. \
2. (), (?:), (?=), []
3. *, +, ?, {n}, {n,}, {n,m}
4. ^, $, \任何元字符、任何字符
5. |
//字符具有高于替换运算符的优先级，使得"m|food"匹配"m"或"food"。若要匹配"mood"或"food"，请使用括号创建子表达式，从而产生"(m|f)ood"
```

正则实例

匹配有效数字

```
规则：
符号：可有可无 // -?
整数部分：如果多位，不能以0开头；如果一位可以是任意值，按照一位和多位来考虑//([1-9]\d+|\d)
小数部分：可有可无，有小数点的话后面至少要有一位数字//(\.\d+)?
var reg=/^-?([1-9]\d+|\d)(\.\d+)?$/;
```

匹配QQ邮箱

```
规则：
最少6位，最多11位
不能0开头
```

```
var reg=/^[1-9]\d{5,10}@qq\.com$/;
```

验证年龄

```
需求：验证18-65之间的年龄
分段：
18-19：1[89]
20-59：[2-5]\d
60-65：6[0-5]
var reg=/^(1[89])|([2-5]\d)|(6[0-5])$/;
//最外面的()不能省略，若省略以18或19开头的任意数字都可以匹配成功
```

匹配用户真实姓名

```
中文汉字的正则：/^[\u4E00-\u9FA5]$/
匹配中文名：部分少数民族名字中有"·"
var reg=/^[\u4E00-\u9FA5]{2,10}(\·[\u4E00-\u9FA5]){2,10}$/;
```

匹配邮箱

```
规则：邮箱名@域名
第一部分：数字、字母、下划线、-、. ，以多个字母或数字开头，-和 .不能连续出现，出现一次后后面必须跟\w+
第二部分：xxx.xx.xx  xxx.xx  xxx.xx.xx.xx  xxx-xxx-xx.xx.xx
//@163.com.cn;@zhu-feng-pei-xun.com.cn
var reg=/^[0-9a-zA-Z]+([-.]\w+)*@[0-9a-zA-Z]+((\.|-)[0-9a-zA-Z]+)*\.[0-9a-zA-Z]+$/;
解析：
邮箱名：以多个字母或数字开头，后面的.xxx或-xxx有*个
域名：@后必须是字母数字，后面的.xxx或-xxx有*个，必须以.xxx结尾
其中，xxx是指字母或数字
```

匹配身份证号

```
规则：
18位，前6位省市县；接下来8位出生年月日(1950-2018)；倒数第二位性别，奇男偶女；最后一位校验，数字或X
不细分出生年月日的情况下：
var reg=/^\d{6}\d{4}\d{2}\d{2}\d{2}\d(\d|X)$/;
细分
年：1950-1999；2000-2009；2010-2018
/^(19[5-9]\d)|(20(0\d|1[0-8]))$/
月：0\d|1[012]
```

```
日：([0-2]\d)|3[01]
```

获取一个人的出生年月日，和性别

```
要求输出：这个人的出生年月日是  年  月  日；性别是
var str = '130425199110124246';
var reg = /\d{6}(\d{4})(\d{2})(\d{2})\d{2}(\d)\w/;
var ary = reg.exec(str);
console.log(`这个人的出生年月日是${ary[1]}年${ary[2]}月${ary[3]}日；性别是${ary[4]%2?'女':'男'}`)

//["130425199110124246", "1991", "10", "12", "4", index: 0, input: "130425199110124246", groups: undefined]
```

queryUrlParameter

```
var str = 'http://bd.kuwo.cn/yinyue/870685?from=baidu&a=123&b=555';
var reg=/([^?=&]+)=([^?=&]+)/g;
// var ary=str.match(reg);
var obj={};
str.replace(reg,function (cur, lit1, lit2) {
    obj[lit1]=lit2;
})
console.log(obj);
//{from: "baidu", a: "123", b: "555"}
```

日期格式化

```
//执行：str.format(template);
//输出：今天是2018年12月18日 18时00分00秒
var str='2018-12-18 18:00:00';
var template = '今天是{1}年{2}月{3}日 {4}时{5}分{6}秒';
String.prototype.format = function format(template) {
    var ary=this.split(/[-: ]/);
    var i=0;
    var temp=arguments[0].replace(/\{\d\}/g,function () {
        return ary[i++];
    });
    return temp;
};
console.log(str.format(template));
//今天是2018年12月18日 18时00分00秒
```

去除字符串首尾空格

```
let str = '  q w   e r  ';
let str1 = str.replace(/^ +| +$/g,'');
//"q w   e r"
```

验证是否包含类名

```
let str1 = 'box box1 box2';
let str = 'box'
let reg = new RegExp('\\b' + str + '\\b');
reg.test(str)
```

正则捕获

所有支持正则的方法都可以实现正则的捕获

当正则捕获的时候：
1，先去验证当前字符串和正则是否匹配，如果不匹配返回的结果是null
2，如果匹配，先从字符串最左边开始，向右查找到匹配的内容，并且把匹配的内容返回

exec

```
//["3", index: 6, input: "srgwse34", groups: undefined]
```

1，捕获的结果是一个数组
2，数组中的第一项是当前本次大正则在字符串中匹配到的结果
3，index：记录了当前本次捕获到结果的起始索引
4，input：当前正则操作的原始字符串
5，如果当前正则当中有分组，获取的数组中，从第二项开始都是每个小分组本次匹配到的结果(通过exec可以把分组中的内容捕获到)
6，无论是否加g，执行一次exec只能把符合正则规则条件中的一个内容捕获到，如果还有其他符合规则的，需要再次执行exec才有可能捕获到
7. 如果正则表达式加g修饰符，可以解决正则捕获懒惰性的问题

```
var reg=/\d/;
reg.exec('srgwse34');
//["3", index: 6, input: "srgwse34", groups: undefined]
var reg=/\d+/;//捕获结果：["34", index: 6, input: "srgwse34", groups: undefined]
var reg=/\d*/;//捕获结果：["", index: 0, input: "srgwse34", groups: undefined]
```

```
var reg=/\d/g;
reg.lastIndex//0
reg.exec('珠峰2018')[0]//"2"
reg.lastIndex//3
reg.exec('珠峰2018')[0]//"0"
reg.lastIndex//4
reg.exec('珠峰2018')[0]//"1"
reg.lastIndex//5
reg.exec('珠峰2018')[0]//"8"
reg.lastIndex//6
reg.exec('珠峰2018')//null
reg.lastIndex//0
reg.exec('珠峰2018')[0]//"2
```

`lastIndex属性规定了每一次开始查找的起始索引，不加g，每一次的lastIndex都是0；加上g之后，每一次的exec都会改变lastIndex这个属性`

exec方法的`局限性`：

> 即使正则设置了g修饰符，exec也不会进行全文查找，只会修改正则对象的lastIndex属性，执行一次exec只能捕获到一个和正则匹配的结果，如果需要都捕获到，我们需要执行N次exec方法才可以(g修饰符在这里的意义仅限于解决懒惰性)

`封装execAll方法`

```
如何一次取出下面字符串中的的2018和2019?
var str='珠峰2018培训2019'

RegExp.prototype.execAll = function (str) {
    let _this = this;//不能直接给this赋值，所以要先把this用其他的变量存起来
    if(!_this.global){
        _this = eval(_this.toString() + 'g');
    }
    let arr = [],
        res = null;
    while(res = _this.exec(str)){
        arr.push(res[0]);
    }
    return arr;
}
let reg = /\d+/;
let str = '珠峰2018培训2019';
console.log(reg.execAll(str))//["2018", "2019"]
```

match

1，加了修饰符g，执行一次**match**会把所有匹配的内容捕获到，但是得不到小分组
2，没有加修饰符g，执行一次**match**只能把第一个匹配的结果捕获到，会得到小分组(此时和**exec**方法没有区别，返回一个数组)

```
var str='zhufeng2018zf2019';
var reg1=/\d+/g;
str.match(reg1);// ["2018", "2019"]
var reg2=/20(\d+)/g;
str.match(reg2);//["2018", "2019"]

var reg3=/\d+/;
str.match(reg3);//["2018", index: 7, input: "zhufeng2018zf2019", gr
oups: undefined]
var reg3=/20(\d+)/;
str.match(reg3);//["2018", "18", index: 7, input: "zhufeng2018zf201
9", groups: undefined]
```

正则中replace的三种用法

1. str.replace(正则，newStr)
2. 第二个参数为回调函数
3. $1、$2...小分组

用法1

str.erplace(正则，newStr)
如果正则中有修饰符g，将替换所有匹配的子字符串，否则只替换第一个匹配的子字符串

```
var str = 'zhufeng2018peixun2019';
str.replace(/\d+/g,'');//"zhufengpeixun"
```

用法2

str = str.**replace**(**reg**,callback)
首先用**reg**到字符串中进行查找匹配，每次匹配到符合规则的，就把回调函数执行一次，并且还把正则本次捕获的结果当作实参传递给这个回调函数，此处**replace**捕获的结果和exec捕获的结果类似(包括大正则内容的索引和原字符串)，并且每一次执行回调函数，函数中**return**的结果，都相当于把本次大正则匹配的内容替换掉

```
var str = 'zhufeng2018peixun2019';
var str1 = str.replace(/(\d)+/,function () {
    console.log(arguments);
    //Arguments(4) ["2018", "8", 7, "zhufeng2018peixun2019", calle
e: f, Symbol(Symbol.iterator): f]
    return '';
})
此时str1 = "zhufengpeixun2019"
```

```
var str = 'zhufeng2018peixun2019';
var reg3=/20(\d+)/g;
str = str.replace(reg3,function(bit,little){
    console.log(bit,little)
})
//2018 18
//2019 19
注意：此时str="zhufengundefinedpeixunundefined"，因为回调函数没有return，默认返回undefined，此外如果reg3=/20(\d)+/g，那么输出的结果就变为
//2018 8
//2019 9，这是因为小分组会获取最后一个符合匹配规则的元素
```

用法3

```
name = "Hello , World";
neme = name.replace(/(\w+)\s*,\s*(\w+)/, "$2 $1");
//"World Hello"
```

总结

```
    对于replace(参数1，参数2)来说，如果参数1是字符串或者是不带全局标志g的正则表达式，那么只进行一次匹配；如果参数1中带有全局标志g，那么会进行多次匹配，不管参数2是字符串还是函数，都会进行多次替换（如果是函数的话是用函数的返回值进行多次替换）；
    如果参数1中带有捕获组的话，那么一般参数2是回调函数或者一些特殊变量
参数2如果是字符串的话，$1,$2...等特殊变量对应的就是各分组捕获到的内容
参数2如果是回调函数，那么回调函数的参数对应的分别就是大正则和各分组匹配到的内容，
```

split

`split方法中的正则`

```
var str='we.fds,agaf,fsr,d+gfs-ers';
var ary=str.split(/[.,+-]/);
```

```
console.log(ary);
//["we", "fds", "agaf", "fsr", "d", "gfs", "ers"]
```

使用test也可实现正则捕获

不管是正则的匹配还是正则的捕获，处理的原理是没区别的：
从字符串的第一个字符向后查找，找到符合正则规则的字符，如果可以找到，说明正则和字符串匹配（test检测返回true，exec捕获返回捕获的内容），如果找到末尾都没有匹配的，说明正则和字符串不匹配（test检测返回false，exec捕获返回null）；
此外，如果正则设置了修饰符g，不管使用test还是exec，都会修改lastIndex值(下一次查找是基于上一次匹配结果的末尾开始查找的)

test可以实现正则捕获，但是每一次只能获取到小分组捕获到的内容
实现原理：
RegExp类下有$1 $2...$9等属性，分别表示第n个小分组匹配的内容

```
var reg=/\{(\d+)\}/g;
var str='my name is {0},i am {1} years old';
//reg.test(str);//true
//console.log(RegExp.$1);//0
//console.log(RegExp.$2);//1

//使用test
var result = [];
while(reg.test(str)){
    result.push(RegExp.$1);
}
console.log(result);// ["0", "1"]

//使用exec
var result = [];
var ary=reg.exec(str);
while(ary){
    result.push(ary[0]);
    ary=reg.exec(str);
}
console.log(result);// ["0", "1"]
```

正则的懒惰性和贪婪性

正则捕获的懒惰性

不加修饰符g时，执行一次exec捕获到第一个符合规则的内容，第二次执行exec，捕获到的依然是第一个匹配的内容，后面符合匹配规则的内容不管执行多少次exec都无法捕获到

解决正则捕获的懒惰性：

> 加修饰符g*(全局匹配)*

正则为什么会存在懒惰性?

> 1. 正则本身有一个属性：lastIndex(下一次正则在字符串中开始匹配查找的位置)
> 2. 默认值：0，从字符串第一个字符开始查找匹配的内容
> 3. 并且默认不管执行多少遍exec方法，正则的lastIndex值都不会变
> 4. 当我们手动把lsatIndx进行修改的时候，不会起到任何的作用

为什么加修饰符g就解决了懒惰性?

> 加了修饰符g，每一次exec结束后，浏览器默认会把lastIndex值进行修改，下一次从上一次结束的位置开始查找，所以可以得到后面匹配的内容了。

正则捕获的贪婪性

> 只要在合法的情况下，正则表达式会尽量多去匹配字符

解决正则捕获的贪婪性：

> 在量词元字符后面加上?，表示非贪婪匹配

```
var reg1=/d(\w+)d/;
str='dffffdfffffd';
reg1.exec(str);
//["dffffdfffffd", "ffffdffff", index: 0, input: "dffffdfffffd", grou
ps: undefined]

str='dffffdfffffd';
var reg2=/d(\w+?)d/;
//["dfffffd", "ffff", index: 0, input: "dffffdfffffd", groups: undefi
ned]
```

```
var reg=/\d+?/
reg.exec('2018');
//["2", index: 0, input: "2018", groups: undefined]
//非贪婪匹配，只拿到2
```

tips

```
var reg = /\d?/
reg.test('efeer234sdsf');// true
reg.test('www')// true
```

```
// 以1 开头 以1结尾  中间无所谓
var reg = /^1(.|\n)*1$/;

reg = /^[1-13]$/;//从1-1或者3，即1或3
reg = /[^]/;//匹配所有非空字符串
(/\b/).test('???c')//true, \b是单词边界符
```

```
//?:不能和分组引用同时使用
(/(\w)\1/).exec('hello');
//["ll", "l", index: 2, input: "hello", groups: undefined]
(/(?:\w)\1/).exec('hello'); //null
```

*运算符的问题

```
var reg=/\d*/g
reg.exec('1234agafg3f2');
//["1234", index: 0, input: "1234agafg3f2", groups: undefined]
reg.exec('1234agafg3f2');
//["", index: 4, input: "1234agafg3f2", groups: undefined]
reg.exec('1234agafg3f2');
//["", index: 4, input: "1234agafg3f2", groups: undefined]
...
```

exec分组捕获，小分组获取最后一个符合匹配规则的元素

```
//以下输出省略部分内容

reg1=/([a-z])+(\d)+/;
reg1.exec('zhufeng2018');
//["zhufeng2018", "g", "8"]
reg1=/([a-z]+)(\d+)/;
reg1.exec('zhufeng2018');
//["zhufeng2018", "zhufeng", "2018"]
```