<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>正则表达式</title>

</head>

<body>

<script type="text/javascript">

//正则表达式是用来处理字符串的可以用来检查一个字符串是否包含某种子串，将匹配的子串替换或者从某个字符串中取出符合某个条件的子串等

// 正则表达式的创建方式

// 1.字面量创建方式

// 2.实例创建方式

// 1.字面量创建方式

// var reg = /pattern/flags

// var reg1 = new RegExp(pattern,flags);

// console.log(reg);

// console.log(reg1);

// pattern:正则表达式

// flags:标识（修饰符）

// 标识主要包括：

// 1.i 忽略大小写

// 2.m 多行匹配，即在到达一行文本末尾时还会继续寻找下一行中是否与正则匹配的项

// 3.g 全局匹配，模式用于所有字符串而不是在找到第一个匹配项时停止

// 字面量创建方式和构造函数创建方式的区别

// 1.字面量创建方式不能进行字符串拼接，实例创建方式可以

// var regParam = 'cm';

// var reg = new RegExp(regParam + '1');

// console.log(reg);

// var reg1 = /regParam/;

// console.log(reg1);

//2.字面量创建方式特殊含义的字符不需要转义，实例创建方式需要转义

// var reg = /\d/;

// var reg1 = new RegExp('\d');

// var reg2 = new RegExp(/\d/);

// var reg3 = new RegExp('\\d');

// console.log(reg);

// console.log(reg1);

// console.log(reg2);

// console.log(reg3);

/\*元字符（具有特殊含义的元字符）

\d: 0-9之间的任意一个数字 \d只占一个位置

\w: 数字、字母、下划线 0-9 a-z A-Z \_

\s: 空格或者空白字符

\D: 除了\d

\W: 除了\w

\S: 除了\s

.: 除了\n之外的任意一个字符

\: 转义字符

|: 或者

(): 分组

\n: 匹配换行符

\b: 匹配边界 字符串的开头和结尾 空格的两边都是边界 不占用字符串位数

^: 限定开始的位置 本身不占位置

$: x限定结束的位置 本身不占位置

[a-z]: 任意字母 []中的任意一个都可以

[^a-z]: 非字母 []中^代表除了

[abc]: abc 三个字母中的任意一个都是可以的 [^abc] 除了这三个字母中的任何一个

\*/

//代表次数的量词元字符

// \*: 0到多个

// +: 1到多个

// ?: 0次或者1次可有可无

// {n}:正好n次

// {n,}:n到多次

// {n,m}:n次道m次

//找出这个字符串中的所有数字

// var str = 'abvvdd23fdfd5fdfdgdfdff6';

// //方法1:

// function findNum(str){

// var tmp = '';

// arr = [];

// for(var i = 0; i<str.length;i++){

// var cur = str[i];

// if(!isNaN(cur)){

// tmp+=cur;

// }else{

// if(tmp){

// arr.push(tmp);

// tmp ='';

// }

// }

// }

// if(tmp){

// arr.push(tmp);

// }

// return arr;

// }

// console.log(findNum(str));

// //方法二：

// // var reg =/\d+/g;

// // console.log(str.match(reg));

/\*量词出现在元字符后面如\d+、\s+限定出现在前面的元字符的次数\*/

// var reg = /\w{2}/g

// console.log(str.match(reg))

//去除一句话的前后空格

// var str =' 我是空格君 ';

// var reg = /^\s+|\s+$/g;

// var res = str.replace(reg,'');

// console.log('('+res+')');

//一般[]中的字符没有特殊含义 如+就表示+

// 但是像\w这样的还是有特殊含义的

// var str1 = 'abc';

// var str2 = 'dbc';

// var str3 = '.bc';

// var reg = /[ad.]bc/;

// console.log(reg.test(str1));

// console.log(reg.test(str2));

// console.log(reg.test(str3));

//[]中不会出现两位数

//例子：匹配18到65岁的人

// var reg = /(18|19)|([2-5]\d)|(6[0-5])/

// console.log(reg.test('76'))

// console.log(reg.test('50'))

//()有提高优先级功能：凡是出现的时候，我们一定要注意是否有必要加上（）来提高优先级：

// 例子：求出这个字符串'abAAbcBCCccdaACBDDabcccddddaab'中出现最多的字母，并求出出现多少次（忽略大小写）

// var str ='abAAbcBCCccdaACBDDabcccddddaab';

//先通过split这个数组方法将字符串转化为数组并且通过toLowerCase()方法转换为小写然后按照本地特定的顺序进行相关排序

// str = str.toLowerCase().split('').sort(function(a,b){

// return a.localeCompare(b)

// }).join('');

// var reg = /(\w)\1+/ig

// var maxStr = '';

// var maxLen = 0;

// str.replace(reg,function($0,$1){

// var regLen = $0.length;

// // console.log(regLen);

// // console.log($1);

// // console.log($0)

// // console.log($1.length)

// if(regLen>maxLen){

// maxLen = regLen;

// maxStr = $1;

// }else if(maxLen == regLen){

// maxStr+=$1;

// }

// })

// // console.log('${maxStr},${maxLen}')

// console.log('出现最多的字母是'+maxStr,'共出现了'+maxLen+'次');

//当我们加（）只是为了提高优先级而不想捕获小分组时，可以在括号中加?:来取消分组的捕获

// var str = 'aaabbb';

// var reg = /(a+)(?:b+)/;

// var res = reg.exec(str);

// console.log(res);

/\*正则运算符的优先级

1.正则表达式从左到右进行计算，并遵循优先级顺序，这与算术表达式类

2.相同的优先级会从左到右进行计算，不同优先级的运算先高后低

\:转义符

(),(?:),(?=),[]==>圆括号和方括号

\*，+，？，{n},{n,},{n,m}==>量词限定符

^,$,\任何元字符、任何字符

| ==> 替换，“或"操作

\*/

/\*

正则的特性

1.贪婪性

正则在捕获时，每一次尽可能多的去捕获符合条件的内容

如果我们想要尽可能少的去捕获符合条件的字符串的话，可以在量词元字符后加？

2.懒惰性

正则在成功捕获一次后不管后边的字符串有没有符合条件的都不再捕获

如果想捕获目标中所有符合条件的字符串的话，我们可以用标识符g来标明是全局捕获

\*/

// var str = '123aaa456';

// var reg = /\d+/;//只捕获一次，一次尽可能多的捕获

// console.log(str.match(reg));

// reg = /\d+?/g//解决贪婪性（加？号）、懒惰性（加g）

// console.log(str.match(reg))

/\*

和正则相关的一些方法

这里我们只介绍test、exec、match和replace方法

\*/

// 1.reg.test(str) 用来验证字符串是否符合正则 符合返回true 否则返回false

// var str = 'abc';

// var reg = /\w+/;

// console.log(reg.test(str))

// 2.reg.exec() 用来捕获符合规则的字符串

// var str = 'abc2133gdfgfdf809';

// var reg = /\d+/;

// console.log(reg.exec(str));

// console.log(reg.lastIndex);

/\*

reg.exec()捕获的数组中

[0:"2133",index:3,input:"abc2133gdfgfdf809"]

0:表示捕获到的字符串

index:3 表示捕获开始的位置的索引

input: 表示原有的字符串

每次调用exec方法时，捕获到的字符串都是不相同的

lastIndex:这个属性记录的是下一次捕获从哪个索引开始。当未开始捕获时，这个值为0;

如果当前次捕获的结果为null时，那么lastIndex的值会被修改为0，下次从头开始捕获

\*/

// var str = 'abc123cba45676aaa789';

// var reg = /\d+/g;

// console.log(reg.lastIndex);

// console.log(reg.exec(str));

// console.log(reg.lastIndex);

// console.log(reg.exec(str));

// console.log(reg.lastIndex);

// console.log(reg.exec(str));

// console.log(reg.lastIndex);

// console.log(reg.exec(str))

// console.log(reg.lastIndex);

// exec的捕获还受分组的影响

// 例题：

// var str = '2018-8-8';

// var reg = /-(\d+)/g;

// console.log(reg.exec(str));

// "-8":正则捕获到的内容

// "8":捕获到的字符串中小分组中的内容

// 3.str.match(reg)如果匹配成功就返回匹配成功的数组，如果匹配不成功，就返回null

//数组去重

// var str = 'aaabbbccc';

// var reg = /(\w)\1+/g;

// var res = str.replace(reg,'$1');

// console.log(res);

//match方法和exec方法差不多当我们进行全局匹配时，二者的不同就体现出来了，当全局匹配时，match方法会一次性把符合匹配条件的字符串全部捕获到数组中而exec只会捕获一次

//用exec方法来简单模拟下match方法

// String.prototype.myMatch = function(reg){

// var arr = [];

// var res = reg.exec(this);

// //如果出现了g也就是全局匹配

// if(reg.global){

// while(res){

// arr.push(res[0]);

// res = reg.exec(this);

// }

// }

// else{

// arr.push(res[0]);

// }

// return arr;

// }

// var str = 'avcdffdfghfh134fhdhf345fh89';

// var reg = /\d+/;

// console.log(str.myMatch(reg));

// //["134"]

// var str = 'avcdffdfghfh134fhdhf345fh89';

// var reg = /\d+/g;

// console.log(str.myMatch(reg));

/\*match和exec都可以受到分组()的影响，不过match只在没有标识符的情况下才可以显示小分组的内容，

如果有全局g，则match会一次性全部捕获放到数组中\*/

// 4.str.replace()方法

// 正则去匹配字符串，匹配成功的字符去替换成新的字符串

//写法：str.replace(reg,newStr);

// var str = 'akk345ffg4566fg';

// var reg = /\d/g;

// var res = str.replace(reg,'Q');

// console.log(res);

// replace的第三个参数也可以是一个函数

// str.replace(reg,fn)

// var str = '2018-08-08';

// str= str.replace(/-\d+/,function(){

// console.log(arguments);

// })

// 通过replace方法获取url中的参数的方法

// (function(pro){

// function queryString(){

// var obj = {};

// reg = /([^?&#+]+)=([^?&#+]+)/g;

// this.replace(reg,function($0,$1,$2){

// obj[$1]=$2;

// })

// return obj;

// }

// pro.queryString = queryString;

// }(String.prototype));

// 零宽断言

// 用于查找某些内容（但不包括这些内容）之前或者之后的东西，如\b,^,$那样用于指定一个位置，这个位置应该满足一定的条件（即断言），因此我们也称为零宽断言

// 1.零宽断言正预测先行断言（?=exp)

// 2.零宽断言负预测先行断言(?!exp)

// 3.零宽断言正回顾后发断言(?<=exp)

// 4.零宽断言负回顾后发断言(?<!exp)

// 例子

//1.零宽断言正预测先行断言（(?=exp)字符出现位置的右边必须匹配到exp这个表达式）

// var str = "i'm singing and dancing";

// var reg = /\b(\w+(?=ing))/g

// console.log(str.match(reg));

//要注意，这里说的是位置，而不是字符

// var str = 'abc';

// var reg = /a(?=b)c/;

// console.log(str.match(reg));

//2.零宽断言负预测先行断言(?!exp)这个就是说字符的位置的右边不能是exp这个表达式

// var str = 'nodejs';

// var reg = /node(?!js)/;

// console.log(reg.test(str));

//3.零宽断言正回顾后发断言(?<=exp)就是说字符出现的位置的前边是exp这个表达式

// var str = '￥998$888';

// var reg = /(?<=\$)\d+/;

// console.log(reg.exec(str));

//4.零宽断言负回顾后发断言(?<?exp)就是说字符出现的位置的前边不能是exp这个表达式

// var str = '￥998$888';

// var reg = /(?<!=\$)\d+/;

// console.log(reg.exec(str));

</script>

</body>

</html>