# 资源

空格： /^[\s\uFEFF\xA0]+|[\s\uFEFF\xA0]+$/g (匹配双字符)

# 正则表达式

**js的正则引擎是NFA(非确定型有限自动机)**

**Nondeterministic Finite Automata**

**匹配慢，但编译快。**

## 几种js正则表达式相关方法

**1.1 String方法**

**1.1.1str.search()**

1.返回第一个匹配子串，找不到返回-1。不支持全局匹配

2.如果参数是字符串，就字符串构造为正则

**1.1.2 str.replace()**

“abcaaabbcc”.replace(/a/gi,’d’);

1. 支持全局匹配。如果regexp带有g，则替换所有的匹配字串。

2. 参数中$1匹配子串。

3.第二个参数可以是函数。\

4.如果第一个参数是字符串，不会转为正则，会直接搜索这个字符串。

**1.1.3 str.match()**

‘aaabbbcc’.match(/a/g)

1. 支持全局。如果有g，会返回匹配字符数组。如果没有g，也返回一个数组。arr[0 ]为匹配的字符；arr[1]为第一个子串.........
2. 非全局返回的数组有两个属性 input（原str）0 index（匹配字符位置）

第一项为 匹配字符串

后面的项则是 匹配各个表达式的字符串

3.如果参数是字符串，转为正则。

**1.1.4 str.split()**

‘a,b,c’.split(“，”)；

‘a, b , c’.split(/\s\*,\*\s/);

**1.2 RegExp方法**

**1.2.1 regExp.exec()**

1.非全局是与match结果一致。

2.全局时每一个匹配都会更新lastIndex

**1.2.2 regExp.test()**

1. .与exec表现一致
2. 返回一个Boolean

## 正则深究

**2.1 RegExp对象的属性(不能被枚举)**

source lastIndex global multiline ignoreCase sticky unicode

**2.2 几个组成元素**

**直接量**

1. a b 1 2
2. \uxxx [\u4e00-\u9fa5]匹配中文
3. \s \S :空白符 \w \W:ASCII字符[a-zA-Z0-9\_] \d\D:数字[0-9]

[\b]空格直接量（特例） \b\B：单词边界（就是/w 与/W之间,/w与^或$之间），而不匹配具体的字符。 后者为字符与字符边界

4. . 通配符

**类**

[]

简单类 [abc]

负向类 [^abc]

范围类 [a-z]

组合类 [a-d1-4A-B]

**重复**

{n,m} {n,} {n} ? \* +

上述为贪婪匹配

{n,m}? ?? +? \*？为懒惰匹配

懒惰匹配的规则是：如‘aaab’.match(/(a+?)b/)

先匹配a,由于a是懒惰模式，只要匹配到a,就立即到下一个字符匹配b，没有b就继续匹配a,一直到匹配到b。

所以上述匹配的是aaab;

**开始结束**

^ 开始 $ 结束

**表达式**

（）可以嵌套 （（））\2匹配是左括号的位置对应的匹配字符

（？：）不记忆这个分组。

分支

（p1 | p2 | p3） 是懒惰模式

**匹配位置**

**^ $**

**锚点条件**

先行断言——接下来有x时匹配

（？=” x”）

负向先行断言

(？！ x)

**修饰符**

i g m

**细节**

在ES5 后的规范中，程序每次遇到RegExp直接量都会创建新的对象。所以不要正则定义（字面量或new RegExp()）放在循环中

通过new一个正则是，注意let reg = RegExp(`\\B(?=(\\d{${n}})+(?!\\d))`,'g')

\b这种要转义

正则表达式式匹配模式：要么匹配字符，要么匹配位置。

（应该翻译成字符串模板。。）

1. 两种模糊匹配

横向模糊匹配

： 一个可匹配的字符串的长度不是固定的。

纵向模糊匹配

：具体到某一个字符，他可以不是某个确定的字符

1. 匹配位置

^ $ \b \B (?=p) (?!p)

^ $的开始和结尾是行的概念，也就是说匹配一行的开始和结尾

在（）表达式可以写在前面或者后面 ^() ()$ ()^ $()

\b 包括单词与$ ,单词与^之间的位置

\B为\b取反

\B (?=p)

匹配p前面的位置

(?!p)

匹配非p前面的位置

1. 括号

一、分组与分支

1.1分组

（ab）+

1.2 分支

（p1 | p2）

1. 引用分组

2.1提取数据

正则操作可以使得

RegExp.$1 | $2 | $3..... 中附加这个正则操作中匹配的表达式对应的内容

2.2替换

var replacStr='2012-10-10';

var replaceRex=/(\d{4})-(\d{2})-(\d{2})/;

var r=replacStr.replace(replaceRex,function () {

return RegExp.$1+'/'+RegExp.$2+'/'+RegExp.$3

})

1. 内部引用

var regex = /\d{4}(-|\/|\.)\d{2}(-|\/|\.)\d{2}/;

可以匹配‘2010-01-10’ ‘2010/01/10’ ‘2010.01.10’

但也会匹配‘2010-10.10’

这里就可以用到表达式的内部引用

var regex= / \d{4}(-|\/|\.)\d{2}\1\d{2}/

\ 1 \ 2对应的是左括号，不算断言的括号

如果\n不存在 ，就是匹配n本身

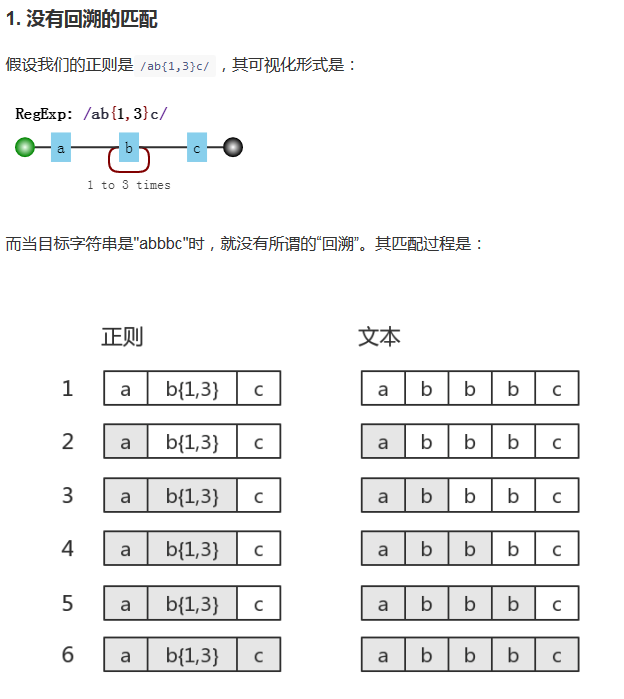
非捕获引用

如果不想使得括号被引用

可以用(?:) 非捕获分组

1. 正则表达式回溯原理

4.1没有回溯的匹配

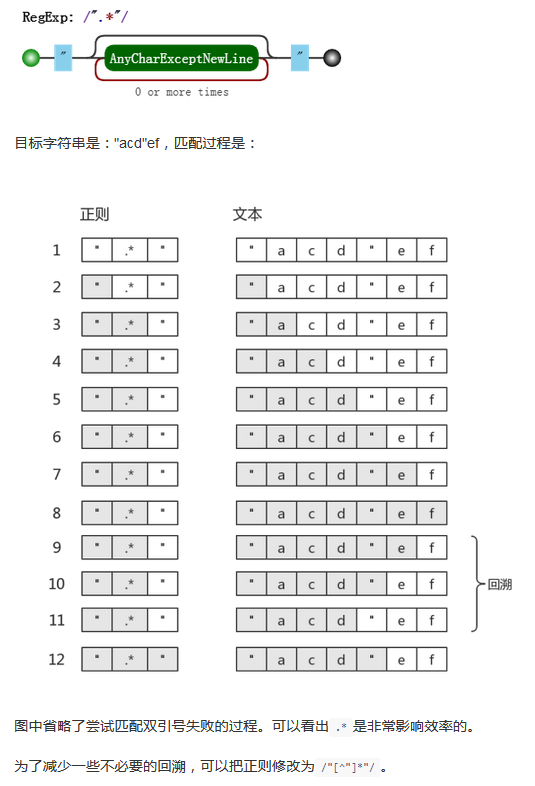


4.2有回溯的匹配



abb 到c 不匹配b，回溯到4 ，b匹配完成，在匹配c





4.3常见的回溯形式

回溯法 也被称为试探法

基本思想：

从问题的某一个状态出发，搜索这种状态出发所能达到的所有‘状态’，当一条路走到‘尽头’（不能再前进），再退后一步或若干步，从另一种可能‘状态’出发，继续搜索。直到所有得到正确解

本质上：就是深度优先搜索算法，其中退到之前某一步这一过程，称为‘回溯’。

可能引发回溯

贪婪量词

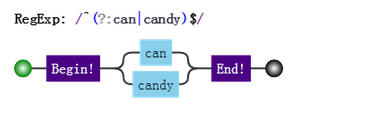
如{1,5}

由于是深度优先搜索，前面的贪婪量词优先于后面的。

惰性量词

后加？

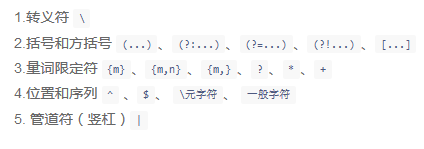
分支结构



1. 正则表达式的拆分

5.1结构和操作符

操作符优先级



由高到低

5.2注意要点

1. 由于 | 优先级最低





要注意 ^ $优先级比 | 高

2.注意量词的连缀

会报错，因为 + 前面没有可以重复的

需要改为/([abc]{3})+/

1. 元字符转义

需要转义

1. 匹配’[abc]’ ‘{1,0}’这样的字符串

/\[abc\]/ 或者 /\[abc]/

6正则表达式的构建

1. 平衡法则
   1. 匹配预期的字符串
   2. 不匹配非预期的字符串
   3. 可读性和可维护性
   4. 效率
2. 构建正则前提

2.1是否能用正则

像 1010010001就不能使用正则

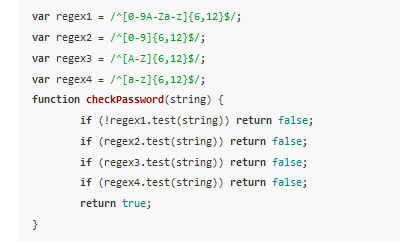
2.2是否有必要使用正则

2.3是否有必要构建一个复杂的正则

如密码问题，需要一个6-12位，有数字，大小写字母，但必须包含两种字符



可以切割成多个小正则来做



1. 准确性
2. 效率

正则表达式的运行分为下面几个步骤

1. 编译
2. 设定起始位置
3. 尝试匹配
4. 匹配失败，从下一位开始继续第三步
5. 最终结果

主要效率比较低下的是3 ，4 步

可以采用方法

1使用具体字符来替代通配符，消除回溯

一般不要使用通配符 . ,而使用[^”]这种类型的匹配模式。

由于回溯的存在，需要引擎保存多种可能未尝试的状态，以便以后回溯时使用。注定要占用一定内存。  
 2.使用非捕获型分组

括号的分组需要内存来保存 捕获分组和分支里的数据。

1. 独立出确定字符

/a+/ 优化为 /aa\*/

1. 提取分支中的公共部分
2. 减少分支数量，缩小它们的范围（回溯成本更低）
3. 正则表达式编程

先匹配，在操作

操作分为：

1.验证

str.search(reg)

reg.test(str)

str.match(reg)

reg.exec(str)

1. 切分

str.split(reg)

1. 提取（匹配表达式字符串）

str.match()

reg.exec()

reg.test()

str.search()

str.replace(reg,function(match, year,month,day){})

1. 替换

str.replace()

1. 相关API

正则两个

①test

②exec

①②当时全局匹配时，每一次匹配，会修改实例的lastIndex属性，而字符串方法不会

字符串4个

①search

②split

③match

④replace

①③会把字符串转为正则

③的返回对象格式与是否有g相关，

有：返回匹配所有内容数组

无：返回（数组第一个为匹配内容，接下来是分组捕获内容，最后是目标字符串）

②str.spllic(/,/,2)可以有第二个参数，表示数组最大长度，正则使用分组，结果数组中是包含分隔符的

如： ‘html,css,js’.split(/(,)/);

[‘html’ , ’,’ , ‘css’ , ‘,’ , ‘javascript’ ]

④强大的replace

第二个参数：

一、是字符串时

$1 , $2 ,......$99匹配捕获数组

$&匹配到的子串文本。

$` 匹配子串左边文本

$’ 匹配子串右边文本

1. 是函数

"1234 2345 3456".replace(/(\d)\d{2}(\d)/g, function(match, $1, $2, index, input) {

console.log([match, $1, $2, index, input]);

});// => ["1234", "1", "4", 0, "1234 2345 3456"]// => ["2345", "2", "5", 5, "1234 2345 3456"]// => ["3456", "3", "6", 10, "1234 2345 3456"]

7正则实例属性

gloable,ingnoreCase , multiline ,lastIndex , source，lastIndex

source: 正则的构成

var utils = {};"Boolean|Number|String|Function|Array|Date|RegExp|Object|Error".split("|").forEach(function(item) {

utils["is" + item] = function(obj) {

return {}.toString.call(obj) == "[object " + item + "]";

};

});console.log( utils.isArray([1, 2, 3]) );

1. 正则类属性

RegExp.Input :最近一次目标字符串 RegExp["$\_"]

RegExp.lastMatch :最近一次匹配字符串 RegExp["$&"]

RegExp.lastParen :最近一次捕获字符串 RegExp["$+"]

RegExp.leftContext 目标字符串中lastMatch之前的文本，简写成RegExp["$`"]

RegExp.rightContext 目标字符串中lastMatch之后的文本，简写成RegExp["$'"]

## 常用的正则表达式

**3.1 电话号码**

**3.2 隔n个字符加空格**

'12345678900'.replace(/\B(?=(?:\d{3})+\b)/g, ',')

'1234567890001245012'.replace(/(\d{4})(?=\d)/g, "$1 ")

3.3 两边去空格

' test '.replace(/^\s+|\s+$/gm,'')

/^[\s\uFEFF\xA0]+|[\s\uFEFF\xA0]+$/g 考虑到全角空格

3.4 邮箱

/^[a-z0-9]+([.\_\\-]\*[a-z0-9])\*@([a-z0-9]+[-a-z0-9]\*[a-z0-9]+.){1,63}[a-z0-9]+$/

3.5必须包含数字

(?=.\*[0-9])^

不全是数字

(?!^[0-9]{6,12}$)^

3.6 转驼峰

function camelize(str) {

var a=str.replace(/[^^][-\_\s]+(.)?/g, function(match, c) {

return c ? c.toUpperCase() : '';

})

return a.replace(/^[-\_\s]/,'')

}

相关连接

E:\大前端技术栈\前端侧\Common JS and ES\RegExp

E:\大前端技术栈\读书笔记\js\_ninja