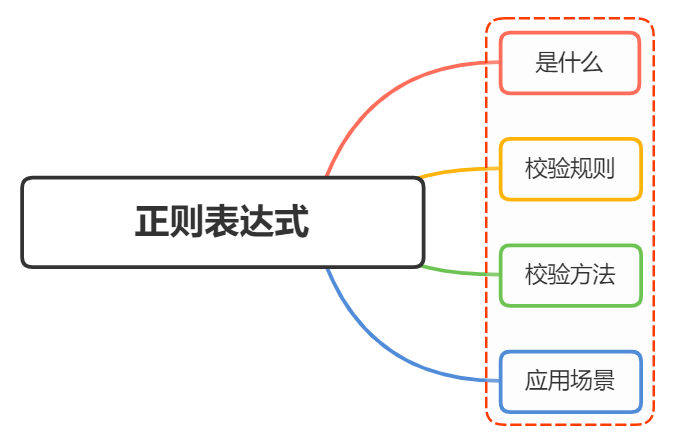
# 面试官：说说你对正则表达式的理解？应用场景？



## 一、是什么

正则表达式是一种用来匹配字符串的强有力的武器

它的设计思想是用一种描述性的语言定义一个规则，凡是符合规则的字符串，我们就认为它“匹配”了，否则，该字符串就是不合法的

在 JavaScript中，正则表达式也是对象，构建正则表达式有两种方式：

1. 字面量创建，其由包含在斜杠之间的模式组成

const re = /\d+/g;

1. 调用RegExp对象的构造函数

const re = new RegExp("\\d+","g");  
  
const rul = "\\d+"  
const re1 = new RegExp(rul,"g");

使用构建函数创建，第一个参数可以是一个变量，遇到特殊字符\需要使用\\进行转义

## 二、匹配规则

常见的校验规则如下：

|  |  |
| --- | --- |
| 规则 | 描述 |
|  | 转义 |
| ^ | 匹配输入的开始 |
| $ | 匹配输入的结束 |
| \* | 匹配前一个表达式 0 次或多次 |
| + | 匹配前面一个表达式 1 次或者多次。等价于 {1,} |
| ? | 匹配前面一个表达式 0 次或者 1 次。等价于{0,1} |
| . | 默认匹配除换行符之外的任何单个字符 |
| x(?=y) | 匹配'x'仅仅当'x'后面跟着'y'。这种叫做先行断言 |
| (?<=y)x | 匹配'x'仅当'x'前面是'y'.这种叫做后行断言 |
| x(?!y) | 仅仅当'x'后面不跟着'y'时匹配'x'，这被称为正向否定查找 |
| (?<!*y*)*x* | 仅仅当'x'前面不是'y'时匹配'x'，这被称为反向否定查找 |
| x|y | 匹配‘x’或者‘y’ |
| {n} | n 是一个正整数，匹配了前面一个字符刚好出现了 n 次 |
| {n,} | n是一个正整数，匹配前一个字符至少出现了n次 |
| {n,m} | n 和 m 都是整数。匹配前面的字符至少n次，最多m次 |
| [xyz] | 一个字符集合。匹配方括号中的任意字符 |
| [^xyz\] | 匹配任何没有包含在方括号中的字符 |
|  | 匹配一个词的边界，例如在字母和空格之间 |
|  | 匹配一个非单词边界 |
|  | 匹配一个数字 |
|  | 匹配一个非数字字符 |
|  | 匹配一个换页符 |
|  | 匹配一个换行符 |
|  | 匹配一个回车符 |
|  | 匹配一个空白字符，包括空格、制表符、换页符和换行符 |
|  | 匹配一个非空白字符 |
|  | 匹配一个单字字符（字母、数字或者下划线） |
|  | 匹配一个非单字字符 |

### 正则表达式标记

|  |  |
| --- | --- |
| 标志 | 描述 |
| g | 全局搜索。 |
| i | 不区分大小写搜索。 |
| m | 多行搜索。 |
| s | 允许 . 匹配换行符。 |
| u | 使用unicode码的模式进行匹配。 |
| y | 执行“粘性(sticky)”搜索,匹配从目标字符串的当前位置开始。 |

使用方法如下：

var re = /pattern/flags;  
var re = new RegExp("pattern", "flags");

在了解下正则表达式基本的之外，还可以掌握几个正则表达式的特性：

### 贪婪模式

在了解贪婪模式前，首先举个例子：

const reg = /ab{1,3}c/

在匹配过程中，尝试可能的顺序是从多往少的方向去尝试。首先会尝试bbb，然后再看整个正则是否能匹配。不能匹配时，吐出一个b，即在bb的基础上，再继续尝试，以此重复

如果多个贪婪量词挨着，则深度优先搜索

const string = "12345";  
const regx = /(\d{1,3})(\d{1,3})/;  
console.log( string.match(reg) );  
// => ["12345", "123", "45", index: 0, input: "12345"]

其中，前面的\d{1,3}匹配的是"123"，后面的\d{1,3}匹配的是"45"

### 懒惰模式

惰性量词就是在贪婪量词后面加个问号。表示尽可能少的匹配

var string = "12345";  
var regex = /(\d{1,3}?)(\d{1,3})/;  
console.log( string.match(regex) );  
// => ["1234", "1", "234", index: 0, input: "12345"]

其中\d{1,3}?只匹配到一个字符"1"，而后面的\d{1,3}匹配了"234"

### 分组

分组主要是用过()进行实现，比如beyond{3}，是匹配d字母3次。而(beyond){3}是匹配beyond三次

在()内使用|达到或的效果，如(abc | xxx)可以匹配abc或者xxx

反向引用，巧用$分组捕获

let str = "John Smith";  
  
// 交换名字和姓氏  
console.log(str.replace(/(john) (smith)/i, '$2, $1')) // Smith, John

## 三、匹配方法

正则表达式常被用于某些方法，我们可以分成两类：

* 字符串（str）方法：match、matchAll、search、replace、split
* 正则对象下（regexp）的方法：test、exec

|  |  |
| --- | --- |
| 方法 | 描述 |
| exec | 一个在字符串中执行查找匹配的RegExp方法，它返回一个数组（未匹配到则返回 null）。 |
| test | 一个在字符串中测试是否匹配的RegExp方法，它返回 true 或 false。 |
| match | 一个在字符串中执行查找匹配的String方法，它返回一个数组，在未匹配到时会返回 null。 |
| matchAll | 一个在字符串中执行查找所有匹配的String方法，它返回一个迭代器（iterator）。 |
| search | 一个在字符串中测试匹配的String方法，它返回匹配到的位置索引，或者在失败时返回-1。 |
| replace | 一个在字符串中执行查找匹配的String方法，并且使用替换字符串替换掉匹配到的子字符串。 |
| split | 一个使用正则表达式或者一个固定字符串分隔一个字符串，并将分隔后的子字符串存储到数组中的 String 方法。 |

### str.match(regexp)

str.match(regexp) 方法在字符串 str 中找到匹配 regexp 的字符

如果 regexp 不带有 g 标记，则它以数组的形式返回第一个匹配项，其中包含分组和属性 index（匹配项的位置）、input（输入字符串，等于 str）

let str = "I love JavaScript";  
  
let result = str.match(/Java(Script)/);  
  
console.log( result[0] ); // JavaScript（完全匹配）  
console.log( result[1] ); // Script（第一个分组）  
console.log( result.length ); // 2  
  
// 其他信息：  
console.log( result.index ); // 7（匹配位置）  
console.log( result.input ); // I love JavaScript（源字符串）

如果 regexp 带有 g 标记，则它将所有匹配项的数组作为字符串返回，而不包含分组和其他详细信息

let str = "I love JavaScript";  
  
let result = str.match(/Java(Script)/g);  
  
console.log( result[0] ); // JavaScript  
console.log( result.length ); // 1

如果没有匹配项，则无论是否带有标记 g ，都将返回 null

let str = "I love JavaScript";  
  
let result = str.match(/HTML/);  
  
console.log(result); // null

### str.matchAll(regexp)

返回一个包含所有匹配正则表达式的结果及分组捕获组的迭代器

const regexp = /t(e)(st(\d?))/g;  
const str = 'test1test2';  
  
const array = [...str.matchAll(regexp)];  
  
console.log(array[0]);  
// expected output: Array ["test1", "e", "st1", "1"]  
  
console.log(array[1]);  
// expected output: Array ["test2", "e", "st2", "2"]

### str.search(regexp)

返回第一个匹配项的位置，如果未找到，则返回 -1

let str = "A drop of ink may make a million think";  
  
console.log( str.search( /ink/i ) ); // 10（第一个匹配位置）

这里需要注意的是，search 仅查找第一个匹配项

## str.replace(regexp)

替换与正则表达式匹配的子串，并返回替换后的字符串。在不设置全局匹配g的时候，只替换第一个匹配成功的字符串片段

const reg1=/javascript/i;  
const reg2=/javascript/ig;  
console.log('hello Javascript Javascript Javascript'.replace(reg1,'js'));  
//hello js Javascript Javascript  
console.log('hello Javascript Javascript Javascript'.replace(reg2,'js'));  
//hello js js js

### str.split(regexp)

使用正则表达式（或子字符串）作为分隔符来分割字符串

console.log('12, 34, 56'.split(/,\s\*/)) // 数组 ['12', '34', '56']

### regexp.exec(str)

regexp.exec(str) 方法返回字符串 str 中的 regexp 匹配项，与以前的方法不同，它是在正则表达式而不是字符串上调用的

根据正则表达式是否带有标志 g，它的行为有所不同

如果没有 g，那么 regexp.exec(str) 返回的第一个匹配与 str.match(regexp) 完全相同

如果有标记 g，调用 regexp.exec(str) 会返回第一个匹配项，并将紧随其后的位置保存在属性regexp.lastIndex 中。 下一次同样的调用会从位置 regexp.lastIndex 开始搜索，返回下一个匹配项，并将其后的位置保存在 regexp.lastIndex 中

let str = 'More about JavaScript at https://javascript.info';  
let regexp = /javascript/ig;  
  
let result;  
  
while (result = regexp.exec(str)) {  
 console.log( `Found ${result[0]} at position ${result.index}` );  
 // Found JavaScript at position 11  
 // Found javascript at position 33  
}

### regexp.test(str)

查找匹配项，然后返回 true/false 表示是否存在

let str = "I love JavaScript";  
  
// 这两个测试相同  
console.log( /love/i.test(str) ); // true

## 四、应用场景

通过上面的学习，我们对正则表达式有了一定的了解

下面再来看看正则表达式一些案例场景：

验证QQ合法性（5~15位、全是数字、不以0开头）：

const reg = /^[1-9][0-9]{4,14}$/  
const isvalid = patrn.exec(s)

校验用户账号合法性（只能输入5-20个以字母开头、可带数字、“\_”、“.”的字串）：

var patrn=/^[a-zA-Z]{1}([a-zA-Z0-9]|[.\_]){4,19}$/;  
const isvalid = patrn.exec(s)

将url参数解析为对象

const protocol = '(?<protocol>https?:)';  
const host = '(?<host>(?<hostname>[^/#?:]+)(?::(?<port>\\d+))?)';  
const path = '(?<pathname>(?:\\/[^/#?]+)\*\\/?)';  
const search = '(?<search>(?:\\?[^#]\*)?)';  
const hash = '(?<hash>(?:#.\*)?)';  
const reg = new RegExp(`^${protocol}\/\/${host}${path}${search}${hash}$`);  
function execURL(url){  
 const result = reg.exec(url);  
 if(result){  
 result.groups.port = result.groups.port || '';  
 return result.groups;  
 }  
 return {  
 protocol:'',host:'',hostname:'',port:'',  
 pathname:'',search:'',hash:'',  
 };  
}  
  
console.log(execURL('https://localhost:8080/?a=b#xxxx'));  
protocol: "https:"  
host: "localhost:8080"  
hostname: "localhost"  
port: "8080"  
pathname: "/"  
search: "?a=b"  
hash: "#xxxx"

再将上面的search和hash进行解析

function execUrlParams(str){  
 str = str.replace(/^[#?&]/,'');  
 const result = {};  
 if(!str){ //如果正则可能配到空字符串，极有可能造成死循环，判断很重要  
 return result;   
 }  
 const reg = /(?:^|&)([^&=]\*)=?([^&]\*?)(?=&|$)/y  
 let exec = reg.exec(str);  
 while(exec){  
 result[exec[1]] = exec[2];  
 exec = reg.exec(str);  
 }  
 return result;  
}  
console.log(execUrlParams('#'));// {}  
console.log(execUrlParams('##'));//{'#':''}  
console.log(execUrlParams('?q=3606&src=srp')); //{q: "3606", src: "srp"}  
console.log(execUrlParams('test=a=b=c&&==&a='));//{test: "a=b=c", "": "=", a: ""}

## 参考文献

* https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular\_Expressions