# JSCORE01

提前到 FTP 下载 12\_JSCORE/Day01 下的 PDF文件

第三阶段小新老师的微信: 18800108022 没有验证, 直接加即可

开发环境配置视频地址: <https://b23.tv/ASzZwW>

## 前言

* 提问的方式:
* 微信直接问问题就可以, 不要有礼貌
* 关于代码错误
* 发代码截图即可, 一定要截取全屏. 通常是未保存报错
* 问题的回复
* 如果老师没有回复你的问题, 请扣1 提醒下
* 推荐的学习网站
* 标准的参考网站MDN:<https://developer.mozilla.org/zh-CN/docs/Web/JavaScript>
* 视频网站: **哔哩哔哩** -- 质量良莠不齐,需要甄别.
* 本阶段的特色: 理解有难度, 代码简单
* 先听懂, 然后给时间写!

## 课程体系介绍

目前行业要求: 复合型人才 -- 什么都要会一点

培养目标: 大前端工程师-- WEB+App+服务器+UI切图...

|  |  |  |
| --- | --- | --- |
| 阶段 | 内容 | 特点 |
| 1 | 数据库, node.js, js基础, git | 后端基础 |
| 2 | html css bootstrap sass ajax... | 前端入门 |
| 3 | JS高级, BOM/DOM(JS操作html), jQuery(简单的框架),vue(工程化框架) | 开发中真正用到的核心技术 |
| 4 | 扩展(第三方模块), 大数据展示, 微信小程序开发(App)... | 扩充知识, 走向App开发 |
| 5 | 前端3大框架: vue react angular 国内框架 uniapp | 就业竞争力 |
| 6 | webpack, flutter, 设计模式... | 免费赠课, 在线观看 TMOOC |

## 正则表达式

Regular Expression: 简称RegExp

正则表达式是一个对字符串进行逻辑验证的公示, 官方提供了很多元字符 来代表一些模糊的含义!

常见正则表达式:

|  |  |
| --- | --- |
| 正则 | 含义 |
| \d{8} | 8个连续的数字 |
| [a-z]{2,8} | a-z 之间的任意字符, 数量在 2到8个 |
| 1[3-9]\d{9} | 手机号的基础表达方式 |
| [\u4e00-\u9fa5] | \u代表 Unicode编码字典 每个中文 在计算机中 都是一个数字来代表 因为计算机底层是2进制, 只识别数字 [\u4e00-\u9fa5] 中文在计算机中的编码范围 |

## 正则元字符

|  |  |
| --- | --- |
| 字符 | 含义 |
| [\](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions#special-backslash) | 依照下列规则匹配：在非特殊字符之前的反斜杠表示下一个字符是特殊字符，不能按照字面理解。例如，前面没有 "" 的 "b" 通常匹配小写字母 "b"，即字符会被作为字面理解，无论它出现在哪里。但如果前面加了 ""，它将不再匹配任何字符，而是表示一个[字符边界](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions#note)。在特殊字符之前的反斜杠表示下一个字符不是特殊字符，应该按照字面理解。详情请参阅下文中的 "转义（Escaping）" 部分。如果你想将字符串传递给 RegExp 构造函数，不要忘记在字符串字面量中反斜杠是转义字符。所以为了在模式中添加一个反斜杠，你需要在字符串字面量中转义它。/[a-z]\s/i 和 new RegExp("[a-z]\\s", "i") 创建了相同的正则表达式：一个用于搜索后面紧跟着空白字符（\s 可看后文）并且在 a-z 范围内的任意字符的表达式。为了通过字符串字面量给 RegExp 构造函数创建包含反斜杠的表达式，你需要在字符串级别和正则表达式级别都对它进行转义。例如 /[a-z]:\\/i 和 new RegExp("[a-z]:\\\\","i") 会创建相同的表达式，即匹配类似 "C:" 字符串。 |
| [^](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions#special-caret) | 匹配输入的开始。如果多行标志被设置为 true，那么也匹配换行符后紧跟的位置。例如，/^A/ 并不会匹配 "an A" 中的 'A'，但是会匹配 "An E" 中的 'A'。当 '^' 作为第一个字符出现在一个字符集合模式时，它将会有不同的含义。[反向字符集合](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions#special-negated-character-set) 一节有详细介绍和示例。 |
| [$](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions#special-dollar) | 匹配输入的结束。如果多行标志被设置为 true，那么也匹配换行符前的位置。例如，/t$/ 并不会匹配 "eater" 中的 't'，但是会匹配 "eat" 中的 't'。 |
| [\*](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions#special-asterisk) | 匹配前一个表达式 0 次或多次。等价于 {0,}。例如，/bo\*/ 会匹配 "A ghost boooooed" 中的 'booooo' 和 "A bird warbled" 中的 'b'，但是在 "A goat grunted" 中不会匹配任何内容。 |
| [+](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions#special-plus) | 匹配前面一个表达式 1 次或者多次。等价于 {1,}。例如，/a+/ 会匹配 "candy" 中的 'a' 和 "caaaaaaandy" 中所有的 'a'，但是在 "cndy" 中不会匹配任何内容。 |
| [?](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions#special-questionmark) | 匹配前面一个表达式 0 次或者 1 次。等价于 {0,1}。例如，/e?le?/ 匹配 "angel" 中的 'el'、"angle" 中的 'le' 以及 "oslo' 中的 'l'。如果**紧跟在任何量词 \*、 +、? 或 {} 的后面**，将会使量词变为**非贪婪**（匹配尽量少的字符），和缺省使用的**贪婪模式**（匹配尽可能多的字符）正好相反。例如，对 "123abc" 使用 /\d+/ 将会匹配 "123"，而使用 /\d+?/ 则只会匹配到 "1"。还用于先行断言中，如本表的 x(?=y) 和 x(?!y) 条目所述。 |
| [.](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions#special-dot) | （小数点）默认匹配除换行符之外的任何单个字符。例如，/.n/ 将会匹配 "nay, an apple is on the tree" 中的 'an' 和 'on'，但是不会匹配 'nay'。如果 s ("dotAll") 标志位被设为 true，它也会匹配换行符。 |
| [(x)](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions#special-capturing-parentheses) | 像下面的例子展示的那样，它会匹配 'x' 并且记住匹配项。其中括号被称为*捕获括号*。模式 /(foo) (bar) \1 \2/ 中的 '(foo)' 和 '(bar)' 匹配并记住字符串 "foo bar foo bar" 中前两个单词。模式中的 \1 和 \2 表示第一个和第二个被捕获括号匹配的子字符串，即 foo 和 bar，匹配了原字符串中的后两个单词。注意 \1、\2、...、\n 是用在正则表达式的匹配环节，详情可以参阅后文的 [\n](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions$edit#special-backreference) 条目。而在正则表达式的替换环节，则要使用像 $1、$2、...、$n 这样的语法，例如，'bar foo'.replace(/(...) (...)/, '$2 $1')。$& 表示整个用于匹配的原字符串。 |
| [(?:x)](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions#special-non-capturing-parentheses) | 匹配 'x' 但是不记住匹配项。这种括号叫作*非捕获括号*，使得你能够定义与正则表达式运算符一起使用的子表达式。看看这个例子 /(?:foo){1,2}/。如果表达式是 /foo{1,2}/，{1,2} 将只应用于 'foo' 的最后一个字符 'o'。如果使用非捕获括号，则 {1,2} 会应用于整个 'foo' 单词。更多信息，可以参阅下文的 [Using parentheses](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions#Using_parentheses) 条目. |
| [x(?=y)](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions#special-lookahead) | 匹配'x'仅仅当'x'后面跟着'y'.这种叫做先行断言。例如，/Jack(?=Sprat)/会匹配到'Jack'仅当它后面跟着'Sprat'。/Jack(?=Sprat|Frost)/匹配‘Jack’仅当它后面跟着'Sprat'或者是‘Frost’。但是‘Sprat’和‘Frost’都不是匹配结果的一部分。 |
| [(?<=y)](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions#special-lookahead)x | 匹配'x'仅当'x'前面是'y'.这种叫做后行断言。例如，/(?<=Jack)Sprat/会匹配到' Sprat '仅仅当它前面是' Jack '。/(?<=Jack|Tom)Sprat/匹配‘ Sprat ’仅仅当它前面是'Jack'或者是‘Tom’。但是‘Jack’和‘Tom’都不是匹配结果的一部分。 |
| [x(?!y)](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions#special-negated-look-ahead) | 仅仅当'x'后面不跟着'y'时匹配'x'，这被称为正向否定查找。例如，仅仅当这个数字后面没有跟小数点的时候，/\d+(?!.)/ 匹配一个数字。正则表达式/\d+(?!.)/.exec("3.141")匹配‘141’而不是‘3.141’ |
| (?<!\*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。例如, 仅仅当这个数字前面没有负号的时候，/(?<!-)\d+/ 匹配一个数字。 /(?<!-)\d+/.exec('3') 匹配到 "3". /(?<!-)\d+/.exec('-3') 因为这个数字前有负号，所以没有匹配到。 |
| [x|y](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions#special-or) | 匹配‘x’或者‘y’。例如，/green|red/匹配“green apple”中的‘green’和“red apple”中的‘red’ |
| [{n}](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions#special-quantifier) | n 是一个正整数，匹配了前面一个字符刚好出现了 n 次。 比如， /a{2}/ 不会匹配“candy”中的'a',但是会匹配“caandy”中所有的 a，以及“caaandy”中的前两个'a'。 |
| [{n,}](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions#special-quantifier) | n是一个正整数，匹配前一个字符至少出现了n次。例如, /a{2,}/ 匹配 "aa", "aaaa" 和 "aaaaa" 但是不匹配 "a"。 |
| [{n,m}](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions#special-quantifier-range) | n 和 m 都是整数。匹配前面的字符至少n次，最多m次。如果 n 或者 m 的值是0， 这个值被忽略。例如，/a{1, 3}/ 并不匹配“cndy”中的任意字符，匹配“candy”中的a，匹配“caandy”中的前两个a，也匹配“caaaaaaandy”中的前三个a。注意，当匹配”caaaaaaandy“时，匹配的值是“aaa”，即使原始的字符串中有更多的a。 |
| [[xyz\]](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions#special-character-set) | 一个字符集合。匹配方括号中的任意字符，包括[转义序列](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Grammar_and_types)。你可以使用破折号（-）来指定一个字符范围。对于点（.）和星号（\*）这样的特殊符号在一个字符集中没有特殊的意义。他们不必进行转义，不过转义也是起作用的。 例如，[abcd] 和[a-d]是一样的。他们都匹配"brisket"中的‘b’,也都匹配“city”中的‘c’。/[a-z.]+/ 和/[\w.]+/与字符串“test.i.ng”匹配。 |
| [[^xyz\]](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions#special-negated-character-set) | 一个反向字符集。也就是说， 它匹配任何没有包含在方括号中的字符。你可以使用破折号（-）来指定一个字符范围。任何普通字符在这里都是起作用的。例如，abc 和 a-c 是一样的。他们匹配"brisket"中的‘r’，也匹配“chop”中的‘h’。 |
| [[\b\]](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions#special-backspace) | 匹配一个退格(U+0008)。（不要和\b混淆了。） |
| [\b](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions#special-word-boundary) | 匹配一个词的边界。一个词的边界就是一个词不被另外一个“字”字符跟随的位置或者前面跟其他“字”字符的位置，例如在字母和空格之间。注意，匹配中不包括匹配的字边界。换句话说，一个匹配的词的边界的内容的长度是0。（不要和[\b]混淆了）使用"moon"举例： /\bm/匹配“moon”中的‘m’； /oo\b/并不匹配"moon"中的'oo'，因为'oo'被一个“字”字符'n'紧跟着。 /oon\b/匹配"moon"中的'oon'，因为'oon'是这个字符串的结束部分。这样他没有被一个“字”字符紧跟着。 /\w\b\w/将不能匹配任何字符串，因为在一个单词中间的字符永远也不可能同时满足没有“字”字符跟随和有“字”字符跟随两种情况。**注意:** JavaScript的正则表达式引擎将[特定的字符集](https://www.ecma-international.org/ecma-262/5.1/#sec-15.10.2.6)定义为“字”字符。不在该集合中的任何字符都被认为是一个断词。这组字符相当有限：它只包括大写和小写的罗马字母，十进制数字和下划线字符。不幸的是，重要的字符，例如“é”或“ü”，被视为断词。 |
| [\B](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions#special-non-word-boundary) | 匹配一个非单词边界。匹配如下几种情况：字符串第一个字符为非“字”字符字符串最后一个字符为非“字”字符两个单词字符之间两个非单词字符之间空字符串例如，/\B../匹配"noonday"中的'oo', 而/y\B../匹配"possibly yesterday"中的’yes‘ |
| [\c\*X\*](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions#special-control) | 当X是处于A到Z之间的字符的时候，匹配字符串中的一个控制符。例如，/\cM/ 匹配字符串中的 control-M (U+000D)。 |
| [\d](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions#special-digit) | 匹配一个数字。``等价于[0-9]。例如， /\d/ 或者 /[0-9]/ 匹配"B2 is the suite number."中的'2'。 |
| [\D](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions#special-non-digit) | 匹配一个非数字字符。``等价于[^0-9]。例如， /\D/ 或者 /[^0-9]/ 匹配"B2 is the suite number."中的'B' 。 |
| [\f](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions#special-form-feed) | 匹配一个换页符 (U+000C)。 |
| [\n](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions#special-line-feed) | 匹配一个换行符 (U+000A)。 |
| [\r](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions#special-carriage-return) | 匹配一个回车符 (U+000D)。 |
| [\s](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions#special-white-space) | 匹配一个空白字符，包括空格、制表符、换页符和换行符。等价于[ \f\n\r\t\v\u00a0\u1680\u180e\u2000-\u200a\u2028\u2029\u202f\u205f\u3000\ufeff]。例如, /\s\w\*/ 匹配"foo bar."中的' bar'。经测试，\s不匹配"[\u180e](https://unicode-table.com/cn/180E/)"，在当前版本Chrome(v80.0.3987.122)和Firefox(76.0.1)控制台输入/\s/.test("\u180e")均返回false。 |
| [\S](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions#special-non-white-space) | 匹配一个非空白字符。等价于 [^\f\n\r\t\v\u00a0\u1680\u180e\u2000-\u200a\u2028\u2029\u202f\u205f\u3000\ufeff]。例如，/\S\w\*/ 匹配"foo bar."中的'foo'。 |
| [\t](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions#special-tab) | 匹配一个水平制表符 (U+0009)。 |
| [\v](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions#special-vertical-tab) | 匹配一个垂直制表符 (U+000B)。 |
| [\w](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions#special-word) | 匹配一个单字字符（字母、数字或者下划线）。等价于 [A-Za-z0-9\_]。例如, /\w/ 匹配 "apple," 中的 'a'，"$5.28,"中的 '5' 和 "3D." 中的 '3'。 |
| [\W](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions#special-non-word) | 匹配一个非单字字符。等价于 [^A-Za-z0-9\_]。例如, /\W/ 或者 /[^A-Za-z0-9\_]/ 匹配 "50%." 中的 '%'。 |
| [\\*n\*](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions#special-backreference) | 在正则表达式中，它返回最后的第n个子捕获匹配的子字符串(捕获的数目以左括号计数)。比如 /apple(,)\sorange\1/ 匹配"apple, orange, cherry, peach."中的'apple, orange,' 。 |
| [\0](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions#special-null) | 匹配 NULL（U+0000）字符， 不要在这后面跟其它小数，因为 \0<digits> 是一个八进制转义序列。 |
| [\xhh](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions#special-hex-escape) | 匹配一个两位十六进制数（\x00-\xFF）表示的字符。 |
| [\uhhhh](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions#special-unicode-escape) | 匹配一个四位十六进制数表示的 UTF-16 代码单元。 |
| \u{hhhh}或\u{hhhhh} | （仅当设置了u标志时）匹配一个十六进制数表示的 Unicode 字符。 |

## 正则表达式

有两种声明正则对象的方式

1. 字面量: 适合正则不变, 内容变

* 例如: 查看用户输入内容中是否有 中文

1. 构造方式: 适合正则会变的场景

* 例如: 一些专业网站, 让用户录入正则进行测试

正则验证: 正则对象的test方法, 用于验证字符串是否符合正则的格式要求

切忌: 正则必须添加 ^ 和 $ 来表达开头结尾

预习的资料 可以看 FTP上 往期班级的笔记, 例如 06 05

下午内容: 函数

### 正则字面量

<!DOCTYPE html>
  
<html lang="en">
  
 <head>
  
 <meta charset="UTF-8" />
  
 <meta http-equiv="X-UA-Compatible" content="IE=edge" />
  
 <meta name="viewport" content="width=, initial-scale=1.0" />
  
 <title>Document</title>
  
 </head>
  
 <body>
  
 <script>
  
 // JS的变量都有两种创建方式:
  
 // 1.字面量: 4 5 true false {} []
  
 // 2.构造函数: new String(); new Boolean()..
  
   
 // 字面量方式: 简单易写, 效率高
  
 // 正则表达式不属于JS, 所以JS要想使用正则, 需要专门的正则对象, 来对正则表达式进行处理
  
 // 字符串用 引号"" '' `` 包围
  
 // 正则 用 // 包围
  
   
 // 正则表达式的 修饰符!
  
 // i ignore 忽略大小写
  
 // g global 全局匹配
  
   
 var reg = /[\u4e00-\u9fa5]/g; //1个中文
  
   
 // 英文:
  
 reg = /[a-z]/gi;
  
   
 //字符串:
  
 var words = "ABCDE abcde 123456 亮亮欠我伍佰元!";
  
 // 从字符串中, 抓取中文字符
  
 // 字符串提供了 match 的方法, 专门用正则来抓取内容
  
 var result = words.match(reg);
  
   
 // clg
  
 console.log("匹配的结果:", result);
  
 </script>
  
 </body>
  
</html>

### 正则构造函数

<!DOCTYPE html>
  
<html lang="en">
  
 <head>
  
 <meta charset="UTF-8" />
  
 <meta http-equiv="X-UA-Compatible" content="IE=edge" />
  
 <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  
 <title>Document</title>
  
 </head>
  
 <body>
  
 <script>
  
 // 构造函数方式: 适合正则会变化的场景
  
   
 var words = "ABCDE abcde 123456 亮亮欠我伍佰元!";
  
 // new Number(); new String(); new Date(); new Boolean();
  
 // 参数1: 正则表达式
  
 // 参数2: 修饰符
  
 var reg = new RegExp("[\u4e00-\u9fa5]", "ig");
  
   
 // match: 从字符串中找到符合 参数正则的 内容
  
 var result = words.match(reg);
  
 console.log(result);
  
 </script>
  
 </body>
  
</html>

### 字面量使用场景

<!DOCTYPE html>
  
<html lang="en">
  
 <head>
  
 <meta charset="UTF-8" />
  
 <meta http-equiv="X-UA-Compatible" content="IE=edge" />
  
 <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  
 <title>Document</title>
  
 </head>
  
 <body>
  
 <script>
  
 // 字面量方式: 适合 正则不变 内容变
  
 // 让用户输入一段内容, 来判断是否有中文!
  
 var msg = prompt("请输入一段内容:");
  
 var reg = /[\u4e00-\u9fa5]/;
  
 // 如果找不到匹配的项目, 则返回null
  
 var result = msg.match(reg);
  
 console.log("用户录入的信息: ", msg);
  
 console.log("查询结果:", result);
  
 // 结果为null 说明没找到中文
  
 if (result == null) {
  
 console.log("密码格式正确!");
  
 } else {
  
 console.log("密码不允许有中文!");
  
 }
  
 </script>
  
 </body>
  
</html>

### 构造方法使用场景

<!DOCTYPE html>
  
<html lang="en">
  
 <head>
  
 <meta charset="UTF-8" />
  
 <meta http-equiv="X-UA-Compatible" content="IE=edge" />
  
 <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  
 <title>Document</title>
  
 </head>
  
 <body>
  
 <script>
  
 // 构造方式: 适合正则表达式会变化的场景
  
 var msg = prompt(
  
 "请输入正则表达式, 从: 'ABCDE abcde 132345 亮亮欠我伍佰元!' 中找到你想要的内容"
  
 );
  
 console.log("msg:", msg);
  
 // 把用户录入的正则字符串, 实时封装为 正则对象
  
 var reg = new RegExp(msg, "ig");
  
 var words = "ABCDE abcde 132345 亮亮欠我伍佰元!";
  
 var result = words.match(reg);
  
 console.log("结果:", result);
  
   
 // \d 数字; [a-z] 英文 [\u4e00-\u9fa5] 中文
  
 </script>
  
 </body>
  
</html>

### 正则格式验证

<!DOCTYPE html>
  
<html lang="en">
  
 <head>
  
 <meta charset="UTF-8" />
  
 <meta http-equiv="X-UA-Compatible" content="IE=edge" />
  
 <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  
 <title>Document</title>
  
 </head>
  
 <body>
  
 <script>
  
 // 格式验证: 手机号
  
 // 手机号特征: 11位 1开头 第二位3-9
  
 // ^: 字符串开头, 暴力理解为 字符串左边的"
  
 // $: 字符串结尾, 暴力理解为 字符串右边的"
  
 var reg = /^1[3-9]\d{9}$/;
  
 // 记住: 验证格式操作, 永远都添加 ^ 和 $
  
   
 var phone = prompt("请输入手机号:");
  
 // 验证: 查看字符串是否符合正则的格式要求
  
 // 正则对象的 test方法: 返回值是 boolean 类型 true/false
  
 // 坑: 正则验证, 只要找到符合条件的 就认为通过
  
 // A13658888981 也正确!
  
 var result = reg.test(phone);
  
 console.log("手机号:", phone);
  
 console.log("验证结果:", result);
  
 </script>
  
 </body>
  
</html>

### 正则替换

<!DOCTYPE html>
  
<html lang="en">
  
 <head>
  
 <meta charset="UTF-8" />
  
 <meta http-equiv="X-UA-Compatible" content="IE=edge" />
  
 <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  
 <title>Document</title>
  
 </head>
  
 <body>
  
 <script>
  
 // 手机号 136\*\*\*\*1232
  
 var phone = "13658874555";
  
   
 // (): 正则的捕获组
  
 // \d: 一个数字
  
 // {n}: 代表有n个
  
 var reg = /(\d{3})(\d{4})(\d{4})/g;
  
 //捕获组序号 1 2 3
  
   
 // 正则替换: replace
  
 // 把参数1正则找到的内容, 替换成 参数2
  
 // $n : 代表第n个捕获组捕捉的值
  
 var result = phone.replace(reg, "$1-$2-$3");
  
 var result = phone.replace(reg, "$1\*\*\*\*$3");
  
 // 练习: 转化为 手机号 136\*\*\*\*\*\*32
  
 reg = /(\d{3})(\d{6})(\d{2})/g;
  
 var result = phone.replace(reg, "$1\*\*\*\*\*\*$3");
  
   
 console.log("替换后的结果:", result);
  
 </script>
  
 </body>
  
</html>

### 正则万能方法

<!DOCTYPE html>
  
<html lang="en">
  
 <head>
  
 <meta charset="UTF-8" />
  
 <meta http-equiv="X-UA-Compatible" content="IE=edge" />
  
 <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  
 <title>Document</title>
  
 </head>
  
 <body>
  
 <script>
  
 // 正则万能方法 exec
  
 // 之前所有学到的正则方法 本质上都是利用 exec 方法 封装出来的
  
 var words = "ABCDEFGHIJKLMN";
  
   
 // 两位英文
  
 var reg = /([a-z])([a-z])/gi;
  
 // exec: 正则对象的, 对字符串参数使用 正则的方法
  
 var result = reg.exec(words); //同字符串的match方法
  
 console.log(result);
  
 // exec: 是分解操作.. 在全局匹配模式下, 每调用一次, 就会向下查找一次.
  
 var result = reg.exec(words);
  
 console.log(result);
  
   
 var result = reg.exec(words);
  
 console.log(result);
  
   
 var result = reg.exec(words);
  
 console.log(result);
  
   
 var result = reg.exec(words);
  
 console.log(result);
  
   
 var result = reg.exec(words);
  
 console.log(result);
  
   
 var result = reg.exec(words);
  
 console.log(result);
  
   
 var result = reg.exec(words);
  
 console.log(result);
  
   
 // 同样的代码, 多次反复执行: 用循环
  
 // 循环语句分两种: for while
  
 // for: 执行固定次数的循环
  
 // while: 执行不固定次数的循环 while(true){}
  
   
 // 此处在正则匹配之前, 并不知道有多少个符合条件的, 所以用while循环更合适.
  
 // while 分两种写法: while(){} do{}while()
  
 // do..while.. 不论条件真假, 都会先执行一次
  
 // 当前场景: 先匹配一次 再决定要不要继续匹配, 适合do..while
  
 // exec() 返回值是null 代表匹配到结尾
  
   
 var reg = /[a-z]{3}/gi; //匹配3个英文
  
   
 do {
  
 // exec() 返回值是null 代表匹配到结尾
  
 var result = reg.exec(words);
  
 console.log("匹配结果:", result);
  
 // 最常见报错: null
  
 // 当使用一个对象之前, 一定要确保对象不是 null
  
 if (result != null) {
  
 console.log(`在序号${result["index"]}找到了${result[0]}`);
  
 }
  
 // 如果结果不是null, 说明还可以继续匹配
  
 } while (result != null);
  
   
 // 使用场景: 当想要封装一个类似于 match test replace 具有特定功能的正则方法, 其中底层使用的就是exec
  
 </script>
  
 </body>
  
</html>

## 函数

### 函数的声明

<!DOCTYPE html>
  
<html lang="en">
  
 <head>
  
 <meta charset="UTF-8" />
  
 <meta http-equiv="X-UA-Compatible" content="IE=edge" />
  
 <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  
 <title>Document</title>
  
 </head>
  
 <body>
  
 <script>
  
 // 函数: 函数就是把一大段重复的代码, 封装在{}中, 起个名字. 之后通过名字来调用这个大段代码.
  
   
 // 计算 1 - 100 的总和
  
 var sum = 0;
  
 for (var i = 1; i <= 100; i++) {
  
 sum += i;
  
 }
  
 console.log(sum); //5050
  
   
 // 场景: 如果计算 1-200, 则需要大段复制, 只修改 100-> 200 即可
  
 var sum = 0;
  
 for (var i = 1; i <= 200; i++) {
  
 sum += i;
  
 }
  
 console.log(sum); //5050
  
   
 // 场景: 如果计算 1-300, 则需要大段复制, 只修改 100-> 300 即可
  
 var sum = 0;
  
 for (var i = 1; i <= 300; i++) {
  
 sum += i;
  
 }
  
 console.log(sum); //5050
  
   
 // 利用函数封装: 把代码放 {} 里, 然后起个名字
  
 function getSum() {
  
 var sum = 0;
  
 for (var i = 1; i <= 100; i++) {
  
 sum += i;
  
 }
  
 return sum; //返回结果给调用者
  
 }
  
 // 通过 函数名(); 就可以多次使用{}中的代码
  
 console.log(getSum());
  
 console.log(getSum());
  
   
 // 参数: 函数的参数,可以传递变化量
  
 function getSum1(end) {
  
 //参数称为形参--形式参数
  
 var sum = 0;
  
 for (var i = 1; i <= end; i++) {
  
 sum += i;
  
 }
  
 return sum; //返回结果给调用者
  
 }
  
 console.log(getSum1(200)); //200 是实参 -> 实际参数
  
 console.log(getSum1(300));
  
 console.log(getSum1(400));
  
   
 // 多个参数
  
 function getSum2(start, end) {
  
 var sum = 0;
  
 for (var i = start; i <= end; i++) {
  
 sum += i;
  
 }
  
 return sum;
  
 }
  
   
 console.log(getSum2(1, 100));
  
 console.log(getSum2(40, 100));
  
 </script>
  
 </body>
  
</html>

### 不固定数量参数:arguments

<!DOCTYPE html>
  
<html lang="en">
  
 <head>
  
 <meta charset="UTF-8" />
  
 <meta http-equiv="X-UA-Compatible" content="IE=edge" />
  
 <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  
 <title>Document</title>
  
 </head>
  
 <body>
  
 <script>
  
 // 神奇max方法
  
 // 方法: 对象中的函数, 称为方法.
  
 console.log(Math.max(12, 3, 4));
  
 console.log(Math.max(12, 3, 4, 4, 45, 34534, 345));
  
 console.log(Math.max(12, 3, 4, 234, 435, 345));
  
   
 function max() {
  
 // 函数内部自带一个 arguments 属性, 其中保存了所有传入的参数
  
 // console.log("arguments:", arguments);
  
 //临时认为 第一个是最大值
  
 var num\_max = arguments[0];
  
   
 for (var i = 0; i < arguments.length; i++) {
  
 // 数组下标取值: arr[0]
  
 // console.log(`参数${i}的值${arguments[i]}`);
  
 // 如果循环的新值, 比临时最大值大, 则临时最大值易主
  
 if (arguments[i] > num\_max) {
  
 num\_max = arguments[i];
  
 }
  
 }
  
   
 return num\_max;
  
 }
  
 // 编程: 把人类的思维 转化成 计算机语言, 然后计算机按照你的思维进行工作!
  
   
 console.log(max(12, 3, 34));
  
 console.log(max(12, 3, 34, 34, 45));
  
 console.log(max(12, 3, 34, 234, 234));
  
 console.log(max(12, 3, 34, 345, 546));
  
 </script>
  
 </body>
  
</html>

### 函数重载

<!DOCTYPE html>
  
<html lang="en">
  
 <head>
  
 <meta charset="UTF-8" />
  
 <meta http-equiv="X-UA-Compatible" content="IE=edge" />
  
 <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  
 <title>Document</title>
  
 </head>
  
 <body>
  
 <script>
  
 // 函数重载 overload
  
 // 此概念来自于 C 语言, 如果一个函数接受不同数量/类型的参数, 则执行不同的操作
  
 // 在JS中官方不提供专业的函数重载写法, 需要我们自己来实现
  
   
 // 期望:
  
 console.log(getSum()); // 得到 1-100的和
  
 console.log(getSum(200)); //得到 1-200的和
  
 console.log(getSum(40, 200)); //得到 40 - 200 的和
  
   
 function getSum() {
  
 console.log("arguments:", arguments);
  
 // 判断参数的个数, 来决定不同的逻辑操作
  
 if (arguments.length == 0) {
  
 // 计算 1 - 100 的和
  
 var sum = 0;
  
 for (var i = 0; i <= 100; i++) {
  
 sum += i;
  
 }
  
 return sum;
  
 }
  
   
 if (arguments.length == 1) {
  
 // 计算 1 - 参数0 的和
  
 var sum = 0;
  
 for (var i = 0; i <= arguments[0]; i++) {
  
 sum += i;
  
 }
  
 return sum;
  
 }
  
   
 if (arguments.length == 2) {
  
 // 计算 参数0 - 参数1 的和
  
 var sum = 0;
  
 for (var i = arguments[0]; i <= arguments[1]; i++) {
  
 sum += i;
  
 }
  
 return sum;
  
 }
  
 }
  
   
 // 函数重载的优点:
  
 // 1. 可以把多个执行相同任何的函数 整合在一起.
  
 // 可以少写很多函数名, 可以节省 内存空间
  
 // 网页上所有的 JS变量 /函数, 都是存在 window 对象中
  
 </script>
  
 </body>
  
</html>

### 声明提升

<!DOCTYPE html>
  
<html lang="en">
  
 <head>
  
 <meta charset="UTF-8" />
  
 <meta http-equiv="X-UA-Compatible" content="IE=edge" />
  
 <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  
 <title>Document</title>
  
 </head>
  
 <body>
  
 <script>
  
 // 声明提升: 其中var就带有声明提升特征
  
   
 console.log(a); //没有声明过的变量, 使用会报错!
  
 // 变量提升特征: JS文件实际上是执行了两次
  
 // 第一次执行: JS引擎会查看整个文件 , 加载所有声明的变量 和 函数
  
 var a;
  
   
 show();
  
   
 function show() {
  
 console.log(123123);
  
 }
  
 </script>
  
 </body>
  
</html>

### 函数的三种声明方式

<!DOCTYPE html>
  
<html lang="en">
  
 <head>
  
 <meta charset="UTF-8" />
  
 <meta http-equiv="X-UA-Compatible" content="IE=edge" />
  
 <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  
 <title>Document</title>
  
 </head>
  
 <body>
  
 <script>
  
 // 函数的三种声明方式
  
 function show(name) {
  
 console.log("命名函数:", name);
  
 }
  
 show("亮亮");
  
   
 // 匿名函数
  
 var show1 = function (name) {
  
 console.log("匿名函数:", name);
  
 };
  
 show1("亮亮2");
  
   
 // 构造函数创建: 适合底层框架使用
  
 // 最后一个参数是 函数体: 要求用字符串书写
  
 // 除了最后一个参数, 其他都是函数的参数, 用字符串依次书写
  
 var show2 = new Function("name", `console.log("构造函数:", name)`);
  
 show2("亮亮3");
  
 </script>
  
 </body>
  
</html>

### 函数声明方式的场景

<!DOCTYPE html>
  
<html lang="en">
  
 <head>
  
 <meta charset="UTF-8" />
  
 <meta http-equiv="X-UA-Compatible" content="IE=edge" />
  
 <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  
 <title>Document</title>
  
 </head>
  
 <body>
  
 <script>
  
 // 声明提升: JS引擎会预读代码, 找到所有的 变量声明 和 函数声明
  
 // JS对于重名变量/函数的解决方案: 后来的 替代 先来的
  
 // 先读取 打印123123, 后读取 打印456456, 最后456的生效
  
 function show() {
  
 console.log(123123);
  
 }
  
 show();
  
   
 function show() {
  
 console.log(456456);
  
 }
  
 show();
  
   
 // 匿名写法:
  
 console.log("=============================");
  
 //
  
 var a = 5; //这是合写格式
  
 var a; //声明 --- 声明提升
  
 a = 5; //赋值 --- 赋值不会提升
  
   
 var show1;
  
 // 赋值操作是普通优先级, 会顺序依次执行
  
 show1 = function () {
  
 console.log(123123);
  
 };
  
 show1();
  
   
 var show1 = function () {
  
 console.log(456456);
  
 };
  
 show1();
  
   
 // 推荐使用 匿名函数 + 变量的方式保存函数
  
 // 可以解决 小概率出现的问题: 同名 命名函数, 覆盖的情况
  
 // 面试常考题!
  
 </script>
  
 </body>
  
</html>

### 变量作用域 与 作用域链

<!DOCTYPE html>
  
<html lang="en">
  
 <head>
  
 <meta charset="UTF-8" />
  
 <meta http-equiv="X-UA-Compatible" content="IE=edge" />
  
 <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  
 <title>Document</title>
  
 </head>
  
 <body>
  
 <script>
  
 // 变量作用域: 共有两个位置
  
 // 1.全局区域: -- window内部, 公共的
  
 // 2.函数区域: -- 函数内部, 私有的
  
   
 // 直接在 script 范围中声明的都在公共区域
  
 var a = 10;
  
   
 function show() {
  
 // 函数内部具有自身的作用域
  
 // 当一个变量, 在全局区域存在, 函数内部也存在, 则优先使用函数内部的 --- 就近原则
  
 var a = 5;
  
 console.log(a);
  
 }
  
   
 show();
  
   
 // 作用域链: 就近原则 一层层向上找
  
 function show2() {
  
 var a = "show2";
  
 function show3() {
  
 // var a = "show3";
  
 function show4() {
  
 // var a = "show4";
  
 function show5() {
  
 // var a = "show5";
  
 console.log(a);
  
 }
  
 show5();
  
 }
  
 show4();
  
 }
  
 show3();
  
 }
  
 show2();
  
 </script>
  
 </body>
  
</html>

## 闭包

闭包: 防止变量全局污染导致的问题, 改用非全局方式存储变量

即 函数作用域

闭包是一个古老的设计, 现在已经被淘汰了!

但是面试基本 必考题! 因为可以考的知识点太多了!

需要思考的问题:

1. 放在全局作用域的同名变量会覆盖, 那就不能都放在全局, 所以放哪?
2. 脚本中 只能调用 window中存储的变量/函数,

* window.xxx 就可以把任意内容 放在window中

// window: 是每个网页自带的一个对象, 其中保存了 JS 能调用的所有资源, 称为全局区域
  
// alert("11111");
  
// 本质是
  
// window.alert(2222);
  
   
// 在脚本中直接声明的变量或函数 都是存储在 window中
  
var aa = "AAA";
  
function aaShow() {
  
console.log(123123);
  
var aa = "AAAAAAAAAAAA";
  
   
window.aabb = "AABB";
  
window.abc = aa;
  
   
window.abcShow = function () {
  
 console.log(aa);
  
};
  
}
  
aaShow();
  
window.aaShow();
  
abcShow();

//最简化的闭包
  
var aa = "AAA";
  
function ashow() {
  
var aa = "AAABBB";
  
window.aaShow = function () {
  
 console.log(aa);
  
};
  
}
  
//此处执行, 是为了激发函数内的代码, 实现aaShow的赋值操作
  
ashow();
  
aaShow(); // 相当于 window.aaShow()

//最简化的闭包
  
var aa = "AAA";
  
// 目前: 声明了函数 -> 然后调用了一次, 单纯为了触发内部的 aaShow 的赋值
  
// 匿名函数自调用: 可以少声明一个函数 ashow, 节省空间
  
// 闭包的本质: 提供了一个独立的函数作用域
  
   
// aaShow = xxx; //相当于 window.aaShow = xxx
  
var aaShow = (function () {
  
var aa = "AAABBB";
  
// 不应该手动调用window
  
// window.aaShow = function () {
  
return function () {
  
 console.log(aa);
  
};
  
})();
  
//此处执行, 是为了激发函数内的代码, 实现aaShow的赋值操作
  
// ashow();
  
aaShow(); // 相当于 window.aaShow()

// function demo() {
  
// window.awords = "亮亮啥都没有讲??!!";
  
// }
  
function demo() {
  
return "亮亮啥都没有讲??!!";
  
}
  
var awords = demo();
  
// window.awords = demo();
  
// demo();
  
   
// 匿名函数自调用: 节省代码, 少声明一个函数, 节省内存空间
  
var awords = (function () {
  
return "亮亮啥都没有讲??!!";
  
})();

//闭包练习:
  
var name = "全局范围";
  
// 1.声明闭包 2.闭包内声明变量 3.闭包返回一个函数,函数中使用变量 4.用变量存储这个返回的函数
  
var aShow = (function () {
  
var name = "闭包内的name";
  
return function () {
  
 console.log(name);
  
};
  
})(); //自调用是为了完成 aShow的赋值过程
  
aShow();

不使用闭包, 会出现全局污染

01.js

// 亮亮书写的代码:
  
// JS一共就两个作用域:
  
// 共享的window全局作用域 和 私有的函数作用域
  
// 考虑把变量放在函数中声明, 就可以避免全局声明的尴尬--覆盖
  
// 此函数没有特殊作用, 就是为了要一个作用域: 所以他连起名的资格都没有
  
// 用匿名函数: 函数必须调用 才能执行内部代码, 下方是 匿名还是的自调用
  
// (匿名函数)()
  
var name = "3333333333";
  
(function () {
  
 // JS引擎非常现实: 只有有用的东西 才会保存下来, 否则会统统删除
  
 var name = "亮亮";
  
   
 function show() {
  
 console.log(name);
  
 }
  
 // window一直存活, 把show保存给window, show就能活下来
  
 // show函数中 使用了name 变量, name有用, 就会活下来
  
 window.lalala = show;
  
 // 不保存到window 就无法调用show
  
 // 因为只能调用 window 中的内容
  
})();
  
   
// show();

02.js

// 亚楠写的代码
  
var name = "亚楠";
  
   
function talk() {
  
 console.log(name);
  
}
  
// talk();

demo.html

<!DOCTYPE html>
  
<html lang="en">
  
 <head>
  
 <meta charset="UTF-8" />
  
 <meta http-equiv="X-UA-Compatible" content="IE=edge" />
  
 <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  
 <title>Document</title>
  
 </head>
  
 <body>
  
 <script src="./01.js"></script>
  
 <script src="./02.js"></script>
  
 <script>
  
 // JS早期的设计理念: 同名变量, 会偷偷的覆盖
  
 // 现在版本中, 采用let声明, 可以直接报错, 防止隐式修改!
  
 // 所以现行版本中, 都使用let 代替var, var被淘汰了!
  
 //
  
 // 在没有let 的年代, 改如何保障变量不被覆盖呢??
  
 // 覆盖原因: 变量name是存储在 同一个对象 window 中
  
   
 // var name = "亚楠";
  
 // var name = "亮亮";
  
   
 // 此处只能调用window中的内容
  
   
 lalala(); //本质是
  
 window.lalala();
  
 // talk(); //应该是 亚楠
  
 // show(); //应该是 亮亮
  
 </script>
  
 </body>
  
</html>

## 回顾

### 正则

两种声明方式:

* 字面量: /正则表达式/ 使用 // 包围
  + 用途: 适合字符串变化, 正则不变的场景
* 构造方式: new RegExp(正则表达式, 修饰词)
  + 适合正则会变动的场景

几个方法:

* 字符串match: 查找符合正则要求的子字符串
* 字符串replace: 正则替换, 新的内容:$1 $2 使用 捕获组()
* 正则的 test: 验证格式, 必须搭配 ^$ 来代表开头与结尾
* 万能方法: exec 是所有正则方法的根本, 适合封装自己的正则函数时使用

### 函数

* arguments: 接受传入的所有参数, 函数自带的变量
* 函数重载: 根据传入参数的数量不同, 类型不同, 来执行不同的逻辑操作
  + 好处: 可以合并 相似功能的函数
* 声明提升, 作用域
* 闭包: 面试必考, 实际不用--被淘汰了!

预习: 可以看 06的笔记 JSCORE day02