《JavaScript 闯关记》之正则表达式

javascript 劼哥stone 10月17日发布

> 由于本课程的核心是 JavaScript，所以本文着重讲解了「正则表达式」在 JavaScript 中的用法，并未深入「正则表达式」的具体细节。如果您尚不了解「正则表达式」，强烈推荐您先学习 **正则表达式30分钟入门教程** **http://deerchao.net/tutorials/regex/regex.htm** 之后，再进行本课程的学习。

正则表达式（regular expression）是一个描述字符模式的对象，使用正则表达式可以进行强大的模式匹配和文本检索与替换功能。JavaScript 的正则表达式语法是 Perl5 的正则表达式语法的大型子集，所以对于有 Perl 编程经验的程序员来说，学习 JavaScript 中的正则表达式是小菜一碟。

正则表达式的定义

JavaScript 中的正则表达式用 `RegExp` 对象表示，可以使用 `RegExp()` 构造函数来创建 `RegExp` 对象，不过 `RegExp` 对象更多是通过字面量的语法来创建，使用下面类似 Perl 的语法，就可以创建一个正则表达式。例如：

```
// 推荐写法
var expression = / pattern / flags ;

// 不推荐写法
var expression = new RegExp(pattern, flags);
```

其中的模式（pattern）部分，是包含在一对斜杠 `/` 之间的字符，可以是任何简单或复杂的正则表达式，可以包含字符类、限定符、分组、向前查找以及反向引用。每个正则表达式都可带有一或多个标志（flags），用以标明正则表达式的行为。正则表达式的匹配模式支持下列3个标志。

- `g`：表示全局（global）模式，即模式将被应用于所有字符串，而非在发现第一个匹配项时立即停止；
- `i`：表示不区分大小写（case-insensitive）模式，即在确定匹配项时忽略模式与字符串的大小写；
- `m`：表示多行（multiline）模式，即在到达一行文本末尾时还会继续查找下一行中是否存在与模式匹配的项。

因此，一个正则表达式就是一个模式与上述3个标志的组合体。不同组合产生不同结果，如下面的例子所示。

```
// 匹配字符串中所有"at"的实例
var pattern1 = /at/g;

// 匹配第一个"bat"或"cat"，不区分大小写
var pattern2 = /[bc]at/i;

// 匹配所有以"at"结尾的3个字符的组合，不区分大小写
var pattern3 = /.at/gi;
```

与其他语言中的正则表达式类似，模式中使用的所有元字符都必须转义。正则表达式中的元字符包括：

```
( [ { \ ^ $ | ) ? * + . ] }
```

这些元字符在正则表达式中都有一或多种特殊用途，因此如果想要匹配字符串中包含的这些字符，就必须对它们进行转义。下面给出几个例子。

```
// 匹配第一个"bat"或"cat"，不区分大小写
var pattern1 = /[bc]at/i;

// 匹配第一个" [bc]at"，不区分大小写
var pattern2 = /\[bc\]at/i;

// 匹配所有以"at"结尾的3个字符的组合，不区分大小写
var pattern3 = /.at/gi;

// 匹配所有".at"，不区分大小写
var pattern4 = /\.at/gi;
```

`RegExp` 实例属性

`RegExp` 的每个实例都具有下列属性，通过这些属性可以取得有关模式的各种信息。

- `global`：布尔值，表示是否设置了 `g` 标志。
- `ignoreCase`：布尔值，表示是否设置了 `i` 标志。
- `lastIndex`：整数，表示开始搜索下一个匹配项的字符位置，从0算起。
- `multiline`：布尔值，表示是否设置了 `m` 标志。
- `source`：正则表达式的字符串表示，按照字面量形式而非传入构造函数中的字符串模式返回。

通过这些属性可以获知一个正则表达式的各方面信息，但却没有多大用处，因为这些信息全都包含在模式声明中。例如：

```
var pattern1 = /\[bc\]at/i;
console.log(pattern1.global);         // false
console.log(pattern1.ignoreCase);     // true
console.log(pattern1.multiline);      // false
console.log(pattern1.lastIndex);      // 0
console.log(pattern1.source);         // "\[bc\]at"

var pattern2 = new RegExp("\\[bc\\]at", "i");
console.log(pattern2.global);         // false
console.log(pattern2.ignoreCase);     // true
console.log(pattern2.multiline);      // false
console.log(pattern2.lastIndex);      // 0
console.log(pattern2.source);         // "\[bc\]at"
```

我们注意到，尽管第一个模式使用的是字面量，第二个模式使用了 `RegExp` 构造函数，但它们的 `source` 属性是相同的。可见，`source` 属性保存的是规范形式的字符串，即字面量形式所用的字符串。

`RegExp` 实例方法

`RegExp` 对象的主要方法是 `exec()`，该方法是专门为捕获组而设计的。`exec()` 接受一个参数，即要应用模式的字符串，然后返回包含第一个匹配项信息的数组；或者在没有匹配项的情况下返回 `null`。返回的数组虽然是 `Array` 的实例，但包含两个额外的属性：`index` 和 `input`。其中，`index` 表示匹配项在字符串中的位置，而 `input` 表示应用正则表达式的字符串。在数组中，第一项是与整个模式匹配的字符串，其他项是与模式中的捕获组匹配的字符串（如果模式中没有捕获组，则该数组只包含一项）。请看下面的例子。

```
var text = "mom and dad and baby";
var pattern = /mom( and dad( and baby)?)?/gi;

var matches = pattern.exec(text);
console.log(matches.index);      // 0
console.log(matches.input);      // "mom and dad and baby"
console.log(matches[0]);         // "mom and dad and baby"
console.log(matches[1]);         // " and dad and baby"
console.log(matches[2]);         // " and baby"
```

这个例子中的模式包含两个捕获组。最内部的捕获组匹配 `"and baby"`，而包含它的捕获组匹配 `"and dad"` 或者 `"and dad and baby"`。当把字符串传入 `exec()` 方法中之后，发现了一个匹配项。因为整个字符串本身与模式匹配，所以返回的数组 `matchs` 的 `index` 属性值为 `0`。数组中的第一项是匹配的整个字符串，第二项包含与第一个捕获组匹配的内容，第三项包含与第二个捕获组匹配的内容。

对于 `exec()` 方法而言，即使在模式中设置了全局标志 `g`，它每次也只会返回一个匹配项。在不设置全局标志的情况下，在同一个字符串上多次调用 `exec()` 将始终返回第一个匹配项的信息。而在设置全局标志的情况下，每次调用 `exec()` 则都会在字符串中继续查找新匹配项，如下面的例子所

示。

```
var text = "cat, bat, sat, fat";

var pattern1 = /.at/;

// 非全局模式，第一次匹配
var matches = pattern1.exec(text);
console.log(matches.index);         // 0
console.log(matches[0]);            // cat
console.log(pattern1.lastIndex);    // 0

// 非全局模式，第二次匹配
matches = pattern1.exec(text);
console.log(matches.index);         // 0
console.log(matches[0]);            // cat
console.log(pattern1.lastIndex);    // 0

var pattern2 = /.at/g;

// 全局模式，第一次匹配
var matches = pattern2.exec(text);
console.log(matches.index);         // 0
console.log(matches[0]);            // cat
console.log(pattern2.lastIndex);    // 0

// 全局模式，第二次匹配
matches = pattern2.exec(text);
console.log(matches.index);         // 5
console.log(matches[0]);            // bat
```

这个例子中的第一个模式 `pattern1` 不是全局模式，因此每次调用 `exec()` 返回的都是第一个匹配项 `"cat"`。而第二个模式 `pattern2` 是全局模式，因此每次调用 `exec()` 都会返回字符串中的下一个匹配项，直至搜索到字符串末尾为止。此外，还应该注意模式的 `lastIndex` 属性的变化情况。在全局匹配模式下，`lastIndex` 的值在每次调用 `exec()` 后都会增加，而在非全局模式下则始终保持不变。

> IE 的 JavaScript 实现在 `lastIndex` 属性上存在偏差，即使在非全局模式下，`lastIndex` 属性每次也会变化。

正则表达式的第二个方法是 `test()`，它接受一个字符串参数。在模式与该参数匹配的情况下返回 `true`；否则，返回 `false`。在只想知道目标字符串与某个模式是否匹配，但不需要知道其文本内容的情况下，使用这个方法非常方便。因此，`test()` 方法经常被用在 `if` 语句中，如下面的例子所示。

```
var text = "000-00-0000";
var pattern = /\d{3}-\d{2}-\d{4}/;

if (pattern.test(text)){
    console.log("The pattern was matched.");
}
```

在这个例子中，我们使用正则表达式来测试了一个数字序列。如果输入的文本与模式匹配，则显示一条消息。这种用法经常出现在验证用户输入的情况下，因为我们只想知道输入是不是有效，至于它为什么无效就无关紧要了。

`RegExp` 实例继承的 `toLocaleString()` 和 `toString()` 方法都会返回正则表达式的字面量，与创建正则表达式的方式无关。例如：

```
var pattern = new RegExp("\\[bc\\]at", "gi");
console.log(pattern.toString());            // /\[bc\]at/gi
console.log(pattern.toLocaleString());      // /\[bc\]at/gi
```

即使上例中的模式是通过调用 `RegExp` 构造函数创建的，但 `toLocaleString()` 和 `toString()` 方法仍然会像它是以字面量形式创建的一样显示其字符串表示。

`RegExp` 构造函数属性

`RegExp` 构造函数包含一些属性（这些属性在其他语言中被看成是静态属性）。这些属性适用于作用域中的所有正则表达式，并且基于所执行的最近一次正则表达式操作而变化。关于这些属性的另一个独特之处，就是可以通过两种方式访问它们。换句话说，这些属性分别有一个长属性名和一个短属性名（Opera是例外，它不支持短属性名）。下表列出了RegExp构造函数的属性。

长属性名	短属性名	说明

input	$_	最近一次要匹配的字符串。Opera未实现此属性。
lastMatch	$&	最近一次的匹配项。Opera未实现此属性。
lastParen	$+	最近一次匹配的捕获组。Opera未实现此属性。
leftContext	$`	input字符串中lastMatch之前的文本。
multiline	$*	布尔值，表示是否所有表达式都使用多行模式。IE和Opera未实现此属性。
rightContext	$'	Input字符串中lastMatch之后的文本。

使用这些属性可以从 `exec()` 或 `test()` 执行的操作中提取出更具体的信息。请看下面的例子。

```
var text = "this has been a short summer";
var pattern = /(.)hort/g;

/*
 * 注意: Internet Explorer 不支持 multiline 属性
 * Opera 不支持 input、lastMatch、lastParen 和 multiline 属性
 */
if (pattern.test(text)){
    console.log(RegExp.input);          // this has been a short summer
    console.log(RegExp.leftContext);    // this has been a
    console.log(RegExp.rightContext);   // summer
    console.log(RegExp.lastMatch);      // short
    console.log(RegExp.lastParen);      // s
    console.log(RegExp.multiline);      // false
}
```

如前所述，例子使用的长属性名都可以用相应的短属性名来代替。只不过，由于这些短属性名大都不是有效的 JavaScript 标识符，因此必须通过方括号语法来访问它们，如下所示。

```
var text = "this has been a short summer";
var pattern = /(.)hort/g;

/*
 * 注意: Internet Explorer 不支持 multiline 属性
 * Opera 不支持 input、lastMatch、lastParen 和 multiline 属性
 */
if (pattern.test(text)){
    console.log(RegExp.$_);      // this has been a short summer
    console.log(RegExp["$`"]);   // this has been a
    console.log(RegExp["$'"]);   // summer
    console.log(RegExp["$&"]);   // short
    console.log(RegExp["$+"]);   // s
    console.log(RegExp["$*"]);   // false
}
```

除了上面介绍的几个属性之外，还有多达9个用于存储捕获组的构造函数属性。访问这些属性的语法是 `RegExp.$1`、`RegExp.$2`...`RegExp.$9`，分别用于存储第一、第二...第九个匹配的捕获组。在调用 `exec()` 或 `test()` 方法时，这些属性会被自动填充。然后，我们就可以像下面这样来使用它们。

```
var text = "this has been a short summer";
var pattern = /(..)or(.)/g;

if (pattern.test(text)){
    console.log(RegExp.$1);  // sh
    console.log(RegExp.$2);  // t
}
```

这里创建了一个包含两个捕获组的模式，并用该模式测试了一个字符串。即使 `test()` 方法只返回一个布尔值，但 `RegExp` 构造函数的属性 `$1` 和 `$2` 也会被匹配相应捕获组的字符串自动填充。

模式的局限性

尽管 JavaScript 中的正则表达式功能还是比较完备的，但仍然缺少某些语言（特别是 Perl）所支持的高级正则表达式特性。下面列出了 JavaScript 正则表达式所不支持的特性。

- 匹配字符串开始和结尾的A和Z锚
- 向后查找（lookbehind）
- 并集和交集类
- 原子组（atomic grouping）
- Unicode支持（单个字符除外，如uFFFF）
- 命名的捕获组
- s（single，单行）和x（free-spacing，无间隔）匹配模式
- 条件匹配
- 正则表达式注释

即使存在这些限制，JavaScript 正则表达式仍然是非常强大的，能够帮我们完成绝大多数模式匹配任务。

关卡

按要求完成下列常用的正则表达式。

```
// 挑战一：数字
var pattern1 = null;  // 补全该正则表达式
console.log(pattern1.test('123')); // true
console.log(pattern1.test('abc')); // false
```

```
// 挑战二：3位的数字
var pattern2 = null;  // 补全该正则表达式
console.log(pattern2.test('123'));  // true
console.log(pattern2.test('1234')); // false
```

```
// 挑战三：至少3位的数字
var pattern3 = null;  // 补全该正则表达式
console.log(pattern3.test('1234')); // true
console.log(pattern3.test('12'));   // false
```

```
// 挑战四：3-5位的数字
var pattern4 = null;  // 补全该正则表达式
console.log(pattern4.test('1234')); // true
console.log(pattern4.test('1'));    // false
```

```
// 挑战五：由26个英文字母组成的字符串
var pattern5 = null;  // 补全该正则表达式
console.log(pattern5.test('abc'));  // true
console.log(pattern5.test('1abc')); // false
```

```
// 挑战六：由数字和26个英文字母组成的字符串
var pattern6 = null;  // 补全该正则表达式
console.log(pattern6.test('1abc'));  // true
console.log(pattern6.test('_abc'));  // false
```

```
// 挑战七：日期格式：年-月-日
var pattern7 = null;  // 补全该正则表达式
console.log(pattern7.test('2016-08-20'));  // true
console.log(pattern7.test('2016/08/20'));  // false
```

```
// 挑战八：时间格式：小时:分钟，24小时制
var pattern8 = null;  // 补全该正则表达式
console.log(pattern8.test('13:45'));  // true
console.log(pattern8.test('13点45')); // false
```

```
// 挑战九：中国大陆身份证号，15位或18位
var pattern9 = null;  // 补全该正则表达式
console.log(pattern9.test('422322219990109003 3'));  // true
console.log(pattern9.test('asdfasdfasfasdf1234'));  // false
```

更多

> 关注微信公众号「劼哥舍」回复「答案」，获取关卡详解。
> 关注 https://github.com/stone0090/javascript-lessons，获取最新动态。

10月17日发布　更多

2 推荐　收藏　¥ 赞赏

你可能感兴趣的文章

javascript javascript javascript javascript　1 收藏，1k 浏览

Javascript 数组　318 浏览

Strict Mode - Javascript语法基础 - Javascript核心　8 收藏，3.5k 浏览

讨论区

关卡中的第二题怎么做？

Ninnka · 10月23日

知道了，要加上起始和结束符

Ninnka · 10月23日

使用评论询问更多信息或提出修改意见，请不要在评论里回答问题　提交评论

评论支持部分 Markdown 语法：`**bold**` `_italic_` `[link](http://example.com)` `> 引用` `` `code` `` `- 列表`。
同时，被你 @ 的用户也会收到通知

本文隶属于专栏

劼哥舍

欢迎来到「劼哥舍」，您非要念成「劫个色」也行。这里坚持原创，分享随笔和学习心得，主要涉及 前端 / .NET / Java 等方面的内容。欢迎交流，欢迎提问，欢迎转载，但需注明出处。

劼哥stone
作者　关注作者

关注专栏

系列文章

《JavaScript 闯关记》之对象 6 收藏， 164 浏览

《JavaScript 闯关记》之数组 6 收藏， 232 浏览

《JavaScript 闯关记》之函数 3 收藏， 161 浏览

《JavaScript 闯关记》之基本包装类型 18 收藏， 862 浏览

《JavaScript 闯关记》之单体内置对象 11 收藏， 420 浏览

《JavaScript 闯关记》之 BOM 14 收藏， 525 浏览

《JavaScript 闯关记》之 DOM（上） 3 收藏， 232 浏览

相关收藏夹

换一组

啦啦啦啦啦啦

8 个条目 | 0 人关注

RxJS

8 个条目 | 1 人关注

mark

4 个条目 | 0 人关注

分享扩散：