基础知识

正则表达式是用于匹配字符串中字符组合的模式，在 JavaScript 中，正则表达式也是对象。

- 正则表达式是在宿主环境下运行的，如
  js/php/node.js 等
- 本章讲解的知识在其他语言中知识也是可用的，会有些函数使用上的区别

#对比分析

与普通函数操作字符串来比较，正则表达式可以写出更简洁、功能强大的代码。
下面使用获取字符串中的所有数字来比较函数与正则的差异。

```
let hd = "houdunren2200hdcms9988";
let nums = [...hd].filter(a => !Number.isNaN(parseInt(a)));
console.log(nums.join(""));

```

使用正则表达式将简单得多

```
let hd = "houdunren2200hdcms9988";
console.log(hd.match(/\d/g).join(""));

```

#创建正则

JS 提供字面量与对象两种方式创建正则表达式

#字面量创建

使用`//`包裹的字面量创建方式是推荐的作法，但它不能在其中使用变量

```
let hd = "houdunren.com";
console.log(/u/.test(hd));//true

```

下面尝试使用 `a` 变量时将不可以查询

```
let hd = "houdunren.com";
let a = "u";
```

```
console.log(/a/.test(hd)); //false

```

虽然可以使用 `eval` 转换为 js 语法来实现将变量解析到正则中，但是比较麻烦，所以有变量时建议使用下面的对象创建方式

```
let hd = "houdunren.com";
let a = "u";
console.log(eval(`/${a}/`).test(hd)); //true

```

#对象创建

当正则需要动态创建时使用对象方式

```
let hd = "houdunren.com";
let web = "houdunren";
let reg = new RegExp(web);
console.log(reg.test(hd)); //true

```

根据用户输入高亮显示内容，支持用户输入正则表达式

```
<body>
  <div id="content">houdunren.com</div>
</body>
<script>
  const content = prompt("请输入要搜索的内容，支持正则表达式");
  const reg = new RegExp(content, "g");
  let body = document
    .querySelector("#content")
    .innerHTML.replace(reg, str => {
      return `<span style="color:red">${str}</span>`;
    });
  document.body.innerHTML = body;
</script>

```

通过对象创建正则提取标签

```
<body>
  <h1>houdunren.com</h1>
  <h1>hdcms.com</h1>
</body>

<script>
function element(tag) {
  const html = document.body.innerHTML;
  let reg = new RegExp("<(" + tag + ")>.+</\\1>", "g");
  return html.match(reg);
}
console.table(element("h1"));

```

#选择符

`|` 这个符号带表选择修释符，也就是 `|` 左右两侧有一个匹配到就可以。
检测电话是否是上海或北京的坐机

```
let tel = "010-12345678";
//错误结果：只匹配 | 左右两边任一结果
console.log(tel.match(/010|020\-\d{7,8}/));

//正确结果：所以需要放在原子组中使用
console.log(tel.match(/(010|020)\-\d{7,8}/));

```

匹配字符是否包含`houdunren` 或 `hdcms`

```
const hd = "houdunren";
console.log(/houdunren|hdcms/.test(hd)); //true

```

#字符转义

转义用于改变字符的含义，用来对某个字符有多种语义时的处理。
假如有这样的场景，如果我们想通过正则查找`/`符号，但是 `/`在正则中有特殊的意义。如果写成`///`这会造成解析错误，所以要使用转义语法 `/\//`来匹配。

```
const url = "
https://www.houdunren.com
const url = "
console.log(/https:\/\//.test(url)); //true

```

使用 `RegExp` 构建正则时在转义上会有些区别，下面是对象与字面量定义正则时区别

```
let price = 12.23;
//含义1: . 除换行外任何字符          含义2: .普通点
//含义1: d 字母d                                          含义2: \d 数字 0~9
console.log(/\d+\.\d+/.test(price));

//字符串中 \d 与 d 是一样的，所以在 new RegExp 时\d 即为 d
console.log("\d" == "d");

//使用对象定义正则时，可以先把字符串打印一样，结果是字面量一样的定义就对了
console.log("\\d+\\.\\d+");
let reg = new RegExp("\\d+\\.\\d+");
console.log(reg.test(price));

```

下面是网址检测中转义符使用

```
let url = "
https://www.houdunren.com
let url = "
console.log(/https?:\/\/\w+\.\w+\.\w+/.test(url));

```

#字符边界

使用字符边界符用于控制匹配内容的开始与结束约定。

边界符	说明
^	匹配字符串的开始
$	匹配字符串的结束，忽略换行符

匹配内容必须以`www`开始

```
const hd = "www.houdunren.com";
console.log(/^www/.test(hd)); //true

```

匹配内容必须以`.com`结束

```
const hd = "www.houdunren.com";
console.log(/\.com$/.test(hd)); //true

```

检测用户名长度为 3~6 位，且只能为字母。如果不使用 `^与$` 限制将得不到正确结果

```
<body>
  <input type="text" name="username" />
</body>

<script>
  document
    .querySelector(`[name="username"]`)
    .addEventListener("keyup", function() {
      let res = this.value.match(/^[a-z]{3,6}$/i);
      console.log(res);
      console.log(res ? "正确" : "失败");
    });
</script>

```

#元子字符

元字符是正则表达式中的最小元素，只代表单一（一个）字符

#字符列表

元字符	说明	示例
\d	匹配任意一个数字	[0-9]
\D	与除了数字以外的任何一个字符匹配	[^0-9]

\w	与任意一个英文字母,数字或下划线匹配	[a-zA-Z_]
\W	除了字母,数字或下划线外与任何字符匹配	[^a-za-z_]
\s	任意一个空白字符匹配，如空格，制表符 \t ，换行符 \n	[\n\f\r\t\v]
\S	除了空白符外任意一个字符匹配	[^\n\f\r\t\v]
.	匹配除换行符外的任意字符	

#使用体验

匹配任意数字

```
let hd = "houdunren 2010";
console.log(hd.match(/\d/g)); //["2", "0", "1", "0"]

```

匹配所有电话号码

```
let hd = `
 张三:010-99999999,李四:020-88888888
`;

let res = hd.match(/\d{3}-\d{7,8}/g);
console.log(res);

```

获取所有用户名

```
let hd = `
张三:010-99999999,李四:020-88888888`;
let res = hd.match(/[^:\d-,]+/g);
console.log(res);

```

匹配任意非数字

```
console.log(/\D/.test(2029)); //false

```

匹配字母数字下划线

```
let hd = "hdcms@";
console.log(hd.match(/\w/g)); //["h", "d", "c", "m", "s"]

```

匹配除了字母,数字或下划线外与任何字符匹配

```
console.log(/\W/.test("@")); //true

```

匹配与任意一个空白字符匹配

```
console.log(/\s/.test(" ")); //true
console.log(/\s/.test("\n")); //true

```

匹配除了空白符外任意一个字符匹配

```
let hd = "hdcms@";
console.log(hd.match(/\S/g)); //["2", "0", "1", "0","@"]

```

如果要匹配点则需要转义

```
let hd = `houdunren@com`;
console.log(/houdunren.com/i.test(hd)); //true
console.log(/houdunren\.com/i.test(hd)); //false

```

使用`.`匹配除换行符外任意字符，下面匹配不到`hdcms.com`因为有换行符

```
const url = `

```

```
  https://www.houdunren.com

  hdcms.com
`;
console.log(url.match(/.+/)[0]);

```

使用`/s`视为单行模式（忽略换行）时，`.` 可以匹配所有

```
let hd = `
  <span>
    houdunren
    hdcms
  </span>
`;
let res = hd.match(/<span>.*<\/span>/s);
console.log(res[0]);

```

正则中空格会按普通字符对待

```
let tel = `010 - 999999`;
console.log(/\d+-\d+/.test(tel)); //false
console.log(/\d+ - \d+/.test(tel)); //true

```

#所有字符

可以使用 `[\s\S]` 或 `[\d\D]` 来匹配所有字符

```
let hd = `
  <span>
    houdunren
    hdcms
  </span>
`;
let res = hd.match(/<span>[\s\S]+<\/span>/);
console.log(res[0]);

```

#模式修饰

正则表达式在执行时会按他们的默认执行方式进行，但有时候默认的处理方式总不能满足我们的需求，所以可以使用模式修正符更改默认方式。

修饰符	说明
i	不区分大小写字母的匹配
g	全局搜索所有匹配内容
m	视为多行
s	视为单行忽略换行符，使用 . 可以匹配所有字符
y	从 regexp.lastIndex 开始匹配
u	正确处理四个字符的 UTF-16 编码

#i

将所有`houdunren.com`统一为小写

```
let hd = "houdunren.com HOUDUNREN.COM";
hd = hd.replace(/houdunren\.com/gi, "houdunren.com");
console.log(hd);

```

#g

使用`g`修饰符可以全局操作内容

```
let hd = "houdunren";
hd = hd.replace(/u/, "@");
console.log(hd); //没有使用 g 修饰符是，只替换了第一个

let hd = "houdunren";
hd = hd.replace(/u/g, "@");
console.log(hd); //使用全局修饰符后替换了全部的 u
```

```
8
```

#m

用于将内容视为多行匹配，主要是对 `^`和 `$` 的修饰
将下面是将以 `#数字`开始的课程解析为对象结构，学习过后面讲到的原子组可以让代码简单些

```
let hd = `
  #1 js,200元 #
  #2 php,300元 #
  #9 houdunren.com # 后盾人
  #3 node.js,180元 #
`;
// [{name:'js',price:'200元'}]
let lessons = hd.match(/^\s*#\d+\s+.+\s+#$/gm).map(v => {
  v = v.replace(/\s*#\d+\s*/, "").replace(/\s+#/, "");
  [name, price] = v.split(",");
  return { name, price };
});
console.log(JSON.stringify(lessons, null, 2));

```

#u

每个字符都有属性，如`L`属性表示是字母，`P` 表示标点符号，需要结合 `u` 模式才有效。其他属性简写可以访问 属性的别名 (opens new window)网站查看。

```
//使用\p{L}属性匹配字母
let hd = "houdunren2010.不断发布教程，加油！";
console.log(hd.match(/\p{L}+/u));

//使用\p{P}属性匹配标点
console.log(hd.match(/\p{P}+/gu));

```

字符也有 unicode 文字系统属性 `Script=文字系统`，下面是使用 `\p{sc=Han}` 获取中文字符 `han`为中文系统，其他语言请查看 文字语言表(opens new window)

```
let hd = `
张三:010-99999999,李四:020-88888888`;
let res = hd.match(/\p{sc=Han}+/gu);
console.log(res);

```

使用 `u` 模式可以正确处理四个字符的 UTF-16 字节编码

```
let str = "𝒳𝒴";
console.table(str.match(/[𝒳𝒴]/)); //结果为乱字符"🤷"

console.table(str.match(/[𝒳𝒴]/u)); //结果正确 "𝒳"

```

#lastIndex

RegExp 对象`lastIndex` 属性可以返回或者设置正则表达式开始匹配的位置

- 必须结合

g 修饰符使用

- 对

exec 方法有效

- 匹配完成时，

lastIndex 会被重置为 0

```
let hd = `后盾人不断分享视频教程，后盾人网址是 houdunren.com`;
let reg = /后盾人(.{2})/g;
reg.lastIndex = 10; //从索引10开始搜索
console.log(reg.exec(hd));
console.log(reg.lastIndex);

reg = /\p{sc=Han}/gu;
while ((res = reg.exec(hd))) {
  console.log(res[0]);
}

```

#y

我们来对比使用 `y` 与`g` 模式，使用 `g` 模式会一直匹配字符串

```
let hd = "udunren";
let reg = /u/g;
console.log(reg.exec(hd));
console.log(reg.lastIndex); //3
console.log(reg.exec(hd));
console.log(reg.lastIndex); //3
console.log(reg.exec(hd)); //null
console.log(reg.lastIndex); //0

```

但使用`y` 模式后如果从 `lastIndex` 开始匹配不成功就不继续匹配了

```
let hd = "udunren";
let reg = /u/y;
console.log(reg.exec(hd));
console.log(reg.lastIndex); //1
console.log(reg.exec(hd)); //null
console.log(reg.lastIndex); //0

```

因为使用 `y` 模式可以在匹配不到时停止匹配，在匹配下面字符中的 qq 时可以提高匹配效率

```
let hd = `后盾人QQ群:11111111,999999999,88888888
后盾人不断分享视频教程，后盾人网址是 houdunren.com`;

let reg = /(\d+),?/y;
reg.lastIndex = 7;
while ((res = reg.exec(hd))) console.log(res[1]);

```

#原子表

在一组字符中匹配某个元字符，在正则表达式中通过元字符表来完成，就是放到`[]` (方括号)中。

#使用语法

原子表	说明
[]	只匹配其中的一个原子
[^]	只匹配"除了"其中字符的任意一个原子
[0-9]	匹配 0-9 任何一个数字
[a-z]	匹配小写 a-z 任何一个字母
[A-Z]	匹配大写 A-Z 任何一个字母

#实例操作

使用`[]`匹配其中任意字符即成功，下例中匹配`ue`任何一个字符，而不会当成一个整体来对待

```
const url = "houdunren.com";
console.log(/ue/.test(url)); //false
console.log(/[ue]/.test(url)); //true

```

日期的匹配

```
let tel = "2022-02-23";
console.log(tel.match(/\d{4}([-\/])\d{2}\1\d{2}/));

```

获取`0~3`间的任意数字

```
const num = "2";
console.log(/[0-3]/.test(num)); //true

```

匹配`a~f`间的任意字符

```
const hd = "e";
console.log(/[a-f]/.test(hd)); //true

```

顺序为升序否则将报错

```
const num = "2";
console.log(/[3-0]/.test(num)); //SyntaxError

```

字母也要升序否则也报错

```
const hd = "houdunren.com";
console.log(/[f-a]/.test(hd)); //SyntaxError

```

获取所有用户名

```
let hd = `
张三:010-99999999,李四:020-88888888`;
let res = hd.match(/[^:\d-,]+/g);
console.log(res);

```

原子表中有些正则字符不需要转义，如果转义也是没问题的，可以理解为在原子表中`.`就是小数点

```
let str = "(houdunren.com)+";
console.table(str.match(/[().+]/g));

//使用转义也没有问题
console.table(str.match(/[\(\)\.\+]/g));

```

可以使用`[\s\S]`或`[\d\D]`匹配到所有字符包括换行符

```
...
const reg = /[\s\S]+/g;
...

```

下面是使用原子表知识删除所有标题

```
<body>
  <p>后盾人</p>
```

```
  <h1>houdunren.com</h1>
  <h2>hdcms.com</h2>
</body>
<script>
  const body = document.body;
  const reg = /<(h[1-6])>[\s\S]*<\/\1>*/g;
  let content = body.innerHTML.replace(reg, "");
  document.body.innerHTML = content;
</script>
```

#原子组

- 如果一次要匹配多个元子，可以通过元子组完成
- 原子组与原子表的差别在于原子组一次匹配多个元子，而原子表则是匹配任意一个字符
- 元字符组用

() 包裹

下面使用原子组匹配 `h1` 标签，如果想匹配 `h2` 只需要把前面原子组改为 `h2` 即可。

```
const hd = `<h1>houdunren.com</h1>`;
console.log(/<(h1)>.+<\/\1>/.test(hd)); //true
```

#基本使用

没有添加 `g` 模式修正符时只匹配到第一个，匹配到的信息包含以下数据

变量	说明
0	匹配到的完整内容
1,2....	匹配到的原子组
index	原字符串中的位置
input	原字符串
groups	命名分组

在`match`中使用原子组匹配，会将每个组数据返回到结果中

- 0 为匹配到的完成内容
- 1/2 等 为原子级内容

- index 匹配的开始位置
- input 原始数据
- groups 组别名

```
let hd = "houdunren.com";
console.log(hd.match(/houdun(ren)\.(com)/));
//["houdunren.com", "ren", "com", index: 0, input: "houdunren.com", groups: undefined]

```

下面使用原子组匹配标题元素

```
let hd = `
  <h1>houdunren</h1>
  <span>后盾人</span>
  <h2>hdcms</h2>
`;

console.table(hd.match(/<(h[1-6])[\s\S]*<\/\1>/g));

```

检测 `0~100` 的数值，使用 `parseInt` 将数值转为 10 进制

```
console.log(/^(\d{1,2}|100)$/.test(parseInt(09, 10)));

```

#邮箱匹配

下面使用原子组匹配邮箱

```
let hd = "2300071698@qq.com";
let reg = /^[\w\-]+@[\w\-]+\.(com|org|cn|cc|net)$/i;
console.dir(hd.match(reg));

```

如果邮箱是以下格式 `houdunren@hd.com.cn` 上面规则将无效，需要定义以下方式

```
let hd = `admin@houdunren.com.cn`;
```

```
let reg = /^[\w-]+@([\w-]+\.)+(org|com|cc|cn)$/;
console.log(hd.match(reg));

```

#引用分组

`\n` 在匹配时引用原子组， `$n` 指在替换时使用匹配的组数据。下面将标签替换为`p`标签

```
let hd = `
  <h1>houdunren</h1>
  <span>后盾人</span>
  <h2>hdcms</h2>
`;

let reg = /<(h[1-6])>([\s\S]*)<\/\1>/gi;
console.log(hd.replace(reg, `<p>$2</p>`));

```

如果只希望组参与匹配，便不希望返回到结果中使用 `(?:` 处理。下面是获取所有域名的示例

```
let hd = `
https://www.houdunren.com

http://houdunwang.com

https://hdcms.com

`;

let reg = /https?:\/\/((?:\w+\.)?\w+\.(?:com|org|cn))/gi;
while ((v = reg.exec(hd))) {
  console.dir(v);
}

```

#分组别名

如果希望返回的组数据更清晰，可以为原子组编号，结果将保存在返回的 `groups`字段中

```
let hd = "<h1>houdunren.com</h1>";
console.dir(hd.match(/<(?<tag>h[1-6])[\s\S]*<\/\1>/));

```

组别名使用 `?<>` 形式定义，下面将标签替换为`p`标签

```
let hd = `
  <h1>houdunren</h1>
  <span>后盾人</span>
  <h2>hdcms</h2>
`;
let reg = /<(?<tag>h[1-6])>(?<con>[\s\S]*)<\/\1>/gi;
console.log(hd.replace(reg, `<p>$<con></p>`));

```

获取链接与网站名称组成数组集合

```
<body>
  <a href="
https://www.houdunren.com
  <a href="
  <a href="
https://www.hdcms.com
  <a href="
  <a href="
https://www.sina.com.cn
  <a href="
</body>

<script>
  let body = document.body.innerHTML;
  let reg = /<a\s*.+?(?<link>https?:\/\/(\w+\.)+(com|org|cc|cn)).*>(?<title>.+)
<\/a>/gi;
  const links = [];
  for (const iterator of body.matchAll(reg)) {
    links.push(iterator["groups"]);
  }
  console.log(links);
</script>
```

```
16
```

#重复匹配

#基本使用

如果要重复匹配一些内容时我们要使用重复匹配修饰符，包括以下几种。

符号	说明
*	重复零次或更多次
+	重复一次或更多次
?	重复零次或一次
{n}	重复 n 次
{n,}	重复 n 次或更多次
{n,m}	重复 n 到 m 次

> 因为正则最小单位是元字符，而我们很少只匹配一个元字符如 a、b 所以基本上重复匹配在每条正则语句中都是必用到的内容。

默认情况下重复选项对单个字符进行重复匹配，即不是贪婪匹配

```
let hd = "hdddd";
console.log(hd.match(/hd+/i)); //hddd

```

使用原子组后则对整个组重复匹配

```
let hd = "hdddd";
console.log(hd.match(/(hd)+/i)); //hd

```

下面是验证坐机号的正则

```
let hd = "010-12345678";
console.log(/0\d{2,3}-\d{7,8}/.exec(hd));

```

验证用户名只能为 3~8 位的字母或数字，并以字母开始

```
<body>
  <input type="text" name="username" />
</body>
<script>
  let input = document.querySelector(`[name="username"]`);
  input.addEventListener("keyup", e => {
    const value = e.target.value;
    let state = /^[a-z][\w]{2,7}$/i.test(value);
    console.log(
      state ? "正确！" : "用户名只能为3~8位的字母或数字，并以字母开始"
    );
  });
</script>

```

验证密码必须包含大写字母并在 5~10 位之间

```
<body>
<input type="text" name="password" />
</body>
<script>
let input = document.querySelector(`[name="password"]`);
input.addEventListener("keyup", e => {
  const value = e.target.value.trim();
  const regs = [/^[a-zA-Z0-9]{5,10}$/, /[A-Z]/];
  let state = regs.every(v => v.test(value));
  console.log(state ? "正确！" : "密码必须包含大写字母并在5~10位之间");
});
</script>

```

#禁止贪婪

正则表达式在进行重复匹配时，默认是贪婪匹配模式，也就是说会尽量匹配更多内容，但是有的时候我们并不希望他匹配更多内容，这时可以通过?进行修饰来禁止重复匹配

使用	说明
*?	重复任意次，但尽可能少重复

+?	重复 1 次或更多次，但尽可能少重复
??	重复 0 次或 1 次，但尽可能少重复
{n,m}?	重复 n 到 m 次，但尽可能少重复
{n,}?	重复 n 次以上，但尽可能少重复

下面是禁止贪婪的语法例子

```
let str = "aaa";
console.log(str.match(/a+/)); //aaa
console.log(str.match(/a+?/)); //a
console.log(str.match(/a{2,3}?/)); //aa
console.log(str.match(/a{2,}?/)); //aa

```

将所有 span 更换为`h4` 并描红，并在内容前加上 `后盾人-`

```
<body>
  <main>
    <span>houdunwang</span>
    <span>hdcms.com</span>
    <span>houdunren.com</span>
  </main>
</body>
<script>
  const main = document.querySelector("main");
  const reg = /<span>([\s\S]+?)<\/span>/gi;
  main.innerHTML = main.innerHTML.replace(reg, (v, p1) => {
    console.log(p1);
    return `<h4 style="color:red">后盾人-${p1}</h4>`;
  });
</script>

```

下面是使用禁止贪婪查找页面中的标题元素

```
<body>
  <h1>
```

```
    houdunren.com
  </h1>
  <h2>hdcms.com</h2>
  <h3></H3>
  <H1></H1>
</body>

<script>
  let body = document.body.innerHTML;
  let reg = /<(h[1-6])>[\s\S]*?<\/\1>/gi;
  console.table(body.match(reg));
</script>

```

#全局匹配

#问题分析

下面是使用`match` 全局获取页面中标签内容，但并不会返回匹配细节

```
<body>
  <h1>houdunren.com</h1>
  <h2>hdcms.com</h2>
  <h1>后盾人</h1>
</body>

<script>
  function elem(tag) {
    const reg = new RegExp("<(" + tag + ")>.+?<\.\\1>", "g");
    return document.body.innerHTML.match(reg);
  }
  console.table(elem("h1"));
</script>

```

#matchAll

在新浏览器中支持使用 `matchAll` 操作，并返回迭代对象

> 需要添加`g`修饰符

```
let str = "houdunren";
let reg = /[a-z]/ig;
for (const iterator of str.matchAll(reg)) {
  console.log(iterator);
}

```

在原型定义 `matchAll`方法，用于在旧浏览器中工作，不需要添加`g` 模式运行

```
String.prototype.matchAll = function(reg) {
  let res = this.match(reg);
  if (res) {
    let str = this.replace(res[0], "^".repeat(res[0].length));
    let match = str.matchAll(reg) || [];
    return [res, ...match];
  }
};
let str = "houdunren";
console.dir(str.matchAll(/(U)/i));

```

#exec

使用 `g` 模式修正符并结合 `exec` 循环操作可以获取结果和匹配细节

```
<body>
  <h1>houdunren.com</h1>
  <h2>hdcms.com</h2>
  <h1>后盾人</h1>
</body>
<script>
  function search(string, reg) {
    const matchs = [];
    while ((data = reg.exec( string))) {
      matchs.push(data);
    }
    return matchs;
  }
```

```
  console.log(search(document.body.innerHTML, /<(h[1-6])>[\s\S]+?<\/\1>/gi));
</script>

```

使用上面定义的函数来检索字符串中的网址

```
let hd = `
https://hdcms.com
let hd = `

https://www.sina.com.cn


https://www.houdunren.com


let res = search(hd, /https?:\/\/(\w+\.)?(\w+\.)+(com|cn)/gi);
console.dir(res);

```

#字符方法

下面介绍的方法是 `String` 提供的支持正则表达式的方法

#search

search() 方法用于检索字符串中指定的子字符串，也可以使用正则表达式搜索，返回值为索引位置

```
let str = "houdunren.com";
console.log(str.search("com"));

```

使用正则表达式搜索

```
console.log(str.search(/\.com/i));

```

#match

直接使用字符串搜索

```

```

```
let str = "houdunren.com";
console.log(str.match("com"));

```

使用正则获取内容，下面是简单的搜索字符串

```
let hd = "houdunren";
let res = hd.match(/u/);
console.log(res);
console.log(res[0]); //匹配的结果
console.log(res[index]); //出现的位置

```

如果使用 `g` 修饰符时，就不会有结果的详细信息了（可以使用 exec），下面是获取所有 h1~6 的标题元素

```
let body = document.body.innerHTML;
let result = body.match(/<(h[1-6])>[\s\S]+?<\/\1>/g);
console.table(result);

```

#matchAll

在新浏览器中支持使用 `matchAll` 操作，并返回迭代对象

```
let str = "houdunren";
let reg = /[a-z]/ig;
for (const iterator of str.matchAll(reg)) {
  console.log(iterator);
}

```

#split

用于使用字符串或正则表达式分隔字符串，下面是使用字符串分隔日期

```
let str = "2023-02-12";
console.log(str.split("-")); //["2023", "02", "12"]
```

```
3
```

如果日期的连接符不确定，那就要使用正则操作了

```
let str = "2023/02-12";
console.log(str.split(/-|\//));

```

#replace

`replace` 方法不仅可以执行基本字符替换，也可以进行正则替换，下面替换日期连接符

```
let str = "2023/02/12";
console.log(str.replace(/\//g, "-")); //2023-02-12

```

替换字符串可以插入下面的特殊变量名：

变量	说明
$$	插入一个 "$"。
$&	插入匹配的子串。
$`	插入当前匹配的子串左边的内容。
$'	插入当前匹配的子串右边的内容。
$n	假如第一个参数是 RegExp 对象，并且 n 是个小于 100 的非负整数，那么插入第 n 个括号匹配的字符串。提示：索引是从 1 开始

在后盾人前后添加三个`=`

```
let hd = "=后盾人=";
console.log(hd.replace(/后盾人/g, "$`$`$&$'$'"));

```

把电话号用 `-` 连接

```
let hd = "(010)99999999 (020)8888888";
```

```
console.log(hd.replace(/\((\d{3,4})\)(\d{7,8})/g, "$1-$2"));

```

把所有`教育`汉字加上链接 https://www.houdunren.com

```
<body>
  在线教育是一种高效的学习方式，教育是一生的事业
</body>
<script>
  const body = document.body;
  body.innerHTML = body.innerHTML.replace(
    /教育/g,
    `<a href="
https://www.houdunren.com
    `<a href="
  );
</script>

```

为链接添加上`https`，并补全 `www.`

```
<body>
  <main>
    <a style="color:red" href="
http://www.hdcms.com
    <a style="color:red" href="
      开源系统
    </a>
    <a id="l1" href="
http://houdunren.com
    <a id="l1" href="
    <a href="
http://yahoo.com
    <a href="
    <h4>
http://www.hdcms.com
    <h4>
  </main>
</body>
<script>
  const main = document.querySelector("body main");
  const reg = /(<a.*href=['"])(http)(:\/\/)(www\.)?(hdcms|houdunren)/gi;
```

```
  main.innerHTML = main.innerHTML.replace(reg, (v, ...args) => {
    args[1] += "s";
    args[3] = args[3] || "www.";
    return args.splice(0, 5).join("");
  });
</script>
```

将标题标签全部替换为 `p` 标签

```
<body>
  <h1>houdunren.com</h1>
  <h2>hdcms.com</h2>
  <h1>后盾人</h1>
</body>

<script>
  const reg = /<(h[1-6])>(.*?)<\/\1>/g;
  const body = document.body.innerHTML;
  const html = body.replace(reg, function(str, tag, content) {
    return `<p>${content}</p>`;
  });
  document.body.innerHTML = html;
</script>
```

删除页面中的 `h1~h6` 标签

```
<body>
  <h1>houdunren.com</h1>
  <h2>hdcms.com</h2>
  <h1>后盾人</h1>
</body>
<script>
  const reg = /<(h[1-6])>(.*?)<\/\1>/g;
  const body = document.body.innerHTML;
  const html = body.replace(reg, "");
  document.body.innerHTML = html;
</script>
```

```
12
```

回调函数

replace 支持回调函数操作，用于处理复杂的替换逻辑

变量名	代表的值
match	匹配的子串。（对应于上述的$&。）
p1,p2, ...	假如 replace()方法的第一个参数是一个 RegExp 对象，则代表第 n 个括号匹配的字符串。（对应于上述的$1，$2 等。）例如，如果是用 /(\a+)(\b+)/ 这个来匹配， p1 就是匹配的 \a+ ， p2 就是匹配的 \b+ 。
offset	匹配到的子字符串在原字符串中的偏移量。（比如，如果原字符串是 'abcd' ，匹配到的子字符串是 'bc' ，那么这个参数将会是 1）
string	被匹配的原字符串。
NamedCaptureGroup	命名捕获组匹配的对象

使用回调函数将 `后盾人` 添加上链接

```
<body>
  <div class="content">
    后盾人不断更新优质视频教程
  </div>
</body>

<script>
  let content = document.querySelector(".content");
  content.innerHTML = content.innerHTML.replace("后盾人", function(
```

```
    search,
    pos,
    source
  ) {
    return `<a href="
https://www.houdunren.com
    return `<a href="
  });
</script>

```

为所有标题添加上 `hot` 类

```
<body>
  <div class="content">
    <h1>后盾人</h1>
    <h2>houdunren.com</h2>
    <h1>后盾人</h1>
  </div>
</body>
<script>
  let content = document.querySelector(".content");
  let reg = /<(h[1-6])>([\s\S]*?)<\/\1>/gi;
  content.innerHTML = content.innerHTML.replace(
    reg,
    (
      search, //匹配到的字符
      p1, //第一个原子组
      p2, //第二个原子组
      index, //索引位置
      source //原字符
    ) => {
      return `
    <${p1} class="hot">${p2}</${p1}>
    `;
    }
  );
</script>

```

#正则方法

下面是 `RegExp` 正则对象提供的操作方法

#test

检测输入的邮箱是否合法

```
<body>
  <input type="text" name="email" />
</body>

<script>
  let email = document.querySelector(`[name="email"]`);
  email.addEventListener("keyup", e => {
    console.log(/^\w+@\w+\.\w+$/.test(e.target.value));
  });
</script>

```

#exec

不使用 `g` 修饰符时与 `match` 方法使用相似，使用 `g` 修饰符后可以循环调用直到全部匹配完。

- 使用

g 修饰符多次操作时使用同一个正则，即把正则定义为变量使用

- 使用

g 修饰符最后匹配不到时返回

null

计算内容中后盾人出现的次数

```
<body>
  <div class="content">
    后盾人不断分享视频教程，后盾人网址是 houdunren.com
  </div>
</body>

<script>
  let content = document.querySelector(".content");
  let reg = /(?<tag>后盾)人/g;
```

```
  let num = 0;
  while ((result = reg.exec(content.innerHTML))) {
    num++;
  }
  console.log(`后盾人共出现${num}次`);
</script>
```

#断言匹配

断言虽然写在扩号中但它不是组，所以不会在匹配结果中保存，可以将断言理解为正则中的条件。

#(?=exp)

零宽先行断言 `?=exp` 匹配后面为 `exp` 的内容
把后面是`教程` 的后盾人汉字加上链接

```
<body>
  <main>
    后盾人不断分享视频教程，学习后盾人教程提升编程能力。
  </main>
</body>

<script>
  const main = document.querySelector("main");
  const reg = /后盾人(?=教程)/gi;
  main.innerHTML = main.innerHTML.replace(
    reg,
    v => `<a href="
https://houdunren.com
    v => `<a href="
  );
</script>
```

下面是将价格后面 添加上 `.00`

```
<script>
  let lessons = `
    js,200元,300次
```

```
    php,300.00元,100次
    node.js,180元,260次
  `;
  let reg = /(\d+)(.00)?(?=元)/gi;
  lessons = lessons.replace(reg, (v, ...args) => {
    args[1] = args[1] || ".00";
    return args.splice(0, 2).join("");
  });
  console.log(lessons);
</script>

```

使用断言验证用户名必须为五位，下面正则体现断言是不是组，并且不在匹配结果中记录

```
<body>
  <input type="text" name="username" />
</body>

<script>
  document
    .querySelector(`[name="username"]`)
    .addEventListener("keyup", function() {
      let reg = /^(?=[a-z]{5}$)/i;
      console.log(reg.test(this.value));
    });
</script>

```

#(?<=exp)

零宽后行断言 `?<=exp` 匹配前面为 `exp` 的内容

匹配前面是`houdunren` 的数字

```
let hd = "houdunren789hdcms666";
let reg = /(?<=houdunren)\d+/i;
console.log(hd.match(reg)); //789

```

匹配前后都是数字的内容

```
let hd = "houdunren789hdcms666";
let reg = /(?<=\d)[a-z]+(?=\d{3})/i;
console.log(hd.match(reg));

```

所有超链接替换为`houdunren.com`

```
<body>
  <a href="https://baidu.com
  <a href="
  <a href="https://yahoo.com
  <a href="
</body>
<script>
  const body = document.body;
  let reg = /(?<=<a.*href=(['"])).+?(?=\1)/gi;
  // console.log(body.innerHTML.match(reg));
  body.innerHTML = body.innerHTML.replace(reg, "https://houdunren.com
  body.innerHTML = body.innerHTML.replace(reg, "
</script>

```

下例中将 `后盾人` 后面的视频添加上链接

```
<body>
  <h1>后盾人视频不断录制案例丰富的视频教程</h1>
</body>

<script>
  let h1 = document.querySelector("h1");
  let reg = /(?<=后盾人)视频/;
  h1.innerHTML = h1.innerHTML.replace(reg, str => {
    return `<a href="https://www.houdunren.com
    return `<a href="
  });
</script>
```

```
12
```

将电话的后四位模糊处理

```
let users = `
  向军电话: 12345678901
  后盾人电话: 98745675603
`;

let reg = /(?<=\d{7})\d+\s*/g;
users = users.replace(reg, str => {
  return "*".repeat(4);
});
console.log(users); //向军电话: 1234567****后盾人电话: 9874567****

```

获取标题中的内容

```
let hd = `<h1>后盾人视频不断录制案例丰富的视频教程</h1>`;
let reg = /(?<=<h1>).*(?=<\/h1>)/g;
console.log(hd.match(reg));

```

#(?!exp)

零宽负向先行断言 后面不能出现 `exp` 指定的内容

使用 `(?!exp)` 字母后面不能为两位数字

```
let hd = "houdunren12";
let reg = /[a-z]+(?!\d{2})$/i;
console.log(reg.exec(hd));

```

下例为用户名中不能出现`向军`

```
<body>
  <main>
    <input type="text" name="username" />
  </main>
```

```
</body>
<script>
  const input = document.querySelector(`[name="username"]`);
  input.addEventListener("keyup", function() {
    const reg = /^(?!.*向军.*)[a-z]{5,6}$/i;
    console.log(this.value.match(reg));
  });
</script>

```

#(?<!exp)

零宽负向后行断言 前面不能出现 exp 指定的内容

获取前面不是数字的字符

```
let hd = "hdcms99houdunren";
let reg = /(?<!\d+)[a-z]+/i;
console.log(reg.exec(hd)); //hdcms

```

把所有不是以 https://oss.houdunren.com 开始的静态资源替换为新网址

```
<body>
  <main>
    <a href="https://www.houdunren.com/1.jpg
    <a href="https://oss.houdunren.com/2.jpg
    <a href="https://cdn.houdunren.com/2.jpg
    <a href="https://houdunren.com/2.jpg
    <a href="
  </main>
</body>
<script>
  const main = document.querySelector("main");
  const reg = /https:\/\/(\w+)?(?<!oss)\..+?(?=\/)/gi;
```

```
  main.innerHTML = main.innerHTML.replace(reg, v => {
    console.log(v);
    return "
https://oss.houdunren.com
    return "
  });
</script>
```