# 定锚点，去噪点，取数据

## 一、入门：正则字符

关于正则字符，很多文章都会讲到，足足有一篇文章才能描述清楚，我这里就不多说，对于我，平时，常用的有：

### 1. . 匹配不包括换行的任意字符

在php的s修饰符下面可以匹配换行，如$pattern='#<div>(.\*?)</div>#s';就可以匹配div内容有换行的数据。

### 2. \s 空格、tab、换行

[\s\S]表示匹配任意字符，\S是\s的反义。

注意区分[\s\S]与.的区别。

### 3. \* 匹配零个或更多个，即0~n

### 4. + 匹配一个或更多个，即至少一个，1~n

### 5. \ 转义

一个特殊字符前加\就表示转义，说明把它当普通字符用

### 6. [] 单字符取一个，比如[abc]会匹配a或b或c

但是，如果[]里面加上^则会变成排除这个字符，如[^abc]就表示不是a、不是b、也不是c   
另外，在[]里面可以使用-表示一个范围，如[0-9]表示从0到9，类似的还有[a-zA-Z]，如果要包含-字符，可以给它加上转义[\-]。   
关于[]常见的错误用法是：[ab|bc]用来表示ab或bc，实际上，它得到的结果是[abc|]，即a或b或c或|这4个字符（单字符）的任意一个。这里可以改成(ab|bc)。   
总结：[]里面的特殊符有五个：[]-\^，其他字符都是普通字符，包括\*.?等。   
说明：   
\* ^在[^ 的首位时候才有特殊意义   
\* [0-9 -在不是首尾的时候有特殊意义   
\* \[ \] 因为占用[] 本身字符，所以有特殊意义   
\* \本身是转义符，有特殊意义

### 7. ^ 字符串开始

这里的^跟[]里面用的^是同一个字符，但是却不是一个意思，这里它表示整个字符串的开始，比如^www表示以www开头的字符串，注意区分，不在[]里面的是开始符，在里面的排除

### 8. $ 字符串结束

### 9. {1,3} 循环次数

[0-9]{1,3}表示在0-9的范围里面循环1个、2个或者3个，可能结果有5、20、415等。   
如果循环指定次数，如3次，则{3,3}可以简写成{3}。   
如果刚好需要匹配字符{1}，则正则需要给{进行转义，得到\{1}的正则。   
如果{}中间不是数字，则{}本身不需要转义。

### 10. ? 有两个用法

#### （1） 匹配一个或零个

比如https?匹配的https（一个s）或者http（零个s）

#### （2）非贪婪模式

所谓非贪婪模式，就是匹配尽可能少的内容，比如，对于源字符串

1. <div>a</div><div>b</div>

使用<div>(.\*?)</div>会得到2个结果（**注意：如果源字符串有换行，使用[\s\S]替换 .** ）：

1. <div>a</div>

和

1. <div>b</div>

因为，当遇到第一个</div>，非贪婪模式就不会再往后找了。   
而使用<div>(.\*)</div>（贪婪模式）则会得到整个字符串

1. <div>a</div><div>b</div>

，因为它会匹配所有字符直到后面再找不到</div>。

### 11. | 多个数据选一（常用于多字符）

前面提到[]里面的字符有选一个字符功能，但是假如不是一个字符，比如：http|ftp|svn 就需要用|分开，|的作用域是一直往后直到遇到括号，比如，对于源字符串

1. http abc
2. ftp abc
3. svn abc

http|ftp|svn abc匹配的结果是：

1. http

和

1. ftp

和

1. svn abc

想要匹配 http abc和ftp abc和svn abc就要使用括号把前边的协议括起来，如(http|ftp|svn) abc 可以得到预期的结果。

### 12. () 数据分界和取数据

上面例子(http|ftp|svn) abc就是数据分界的例子，然后，匹配结果会得到一个[1]的子集数据（数组下标1），这里就是子模式的概念，子模式也叫分组，利用子模式，可以得到想要取出来的数据。子模式1、2、3的计算方法为**左括号**的计数，从左到右，从1开始，比如：   
(http|ftp|svn)://([^/]+)，分组1得到的是(http|ftp|svn)里面的数据，分组2得到([^/]+)里面的数据，对于嵌套括号也是点左括号即可。在正则中有很多与括号结合的写法，你在数左括号的时候，一定要注意，非捕获组和[环视](http://www.zjmainstay.cn/deep-regexp#%E6%A6%82%E5%BF%B5%E5%9B%9B%E7%8E%AF%E8%A7%86%E6%96%AD%E8%A8%80%E9%9B%B6%E5%AE%BD%E6%96%AD%E8%A8%80)的左括号都是不需要数的。   
在使用子模式过程中，常见两种写法是：\1 和 $1。   
（1） \1 是在正则表达式本身中引用分组1的内容，如：   
我们要匹配111这样的连续出现3此的数字，我们可以写出正则：(\d)\1\1，(\d)匹配到第一个1，后面再引用这个匹配内容，得到111。   
（2） $1 是在替换中调用分组的内容，如：   
我们要替换链接参数name=Zjmainstay为username=Zjmainstay，我们可以使用正则name=([^&]+)替换为username=$1来实现，这里的$1就引用了分组1的结果Zjmainstay，因此得到我们想要的结果。

### 13. (?:) 非捕获组

上面说到()作为子模式可以得到它里面的数据，但是，有些时候，()只是作为数据分界功能，并不需要取出来，这时候就要用到非捕获组的概念了。比如：(http|ftp|svn)://([^/]+)只想得到域名，也就是[2]，那么(http|ftp|svn)就只是数据分界的功能，这里不需要捕获，因此使用非捕获组功能，(?:http|ftp|svn)屏蔽这部分的数据获取，此时，(?:这个左括号排除[1]计数，也就是(?:http|ftp|svn)://([^/]+)中的([^/]+)变成[1]了。

### 14. 分隔符

在一些语言中，你会发现正则第一个和最后一个字符是相同的，如：   
/\d+/   
这个/ /在PHP中称为分隔符，正则表达式需要由分隔符闭合包裹。在PHP中，分隔符可以使任意非字母数字、非反斜线、非空白字符。这个概念很关键，它能帮助我们简化一些正则的书写，避免错误，如：   
/<div>.\*?</div>/   
这个正则是错误的。   
原因是</div>的/与分隔符相同，但是却没有做转义。   
如下程序：

1. preg\_match('/<div>.\*?</div>/', '<div>abc</div>', $match);

PHP中会收到错误提示：Warning: preg\_match(): Unknown modifier 'd' in regexTest.php on line 2   
对于这种情况，有两种解决方案：   
（1）/<div>.\*?<\/div>/   
（2）#<div>.\*?</div>#   
方案（1）是对正则内部的分隔符做转义，方案（2）是规避了本来的/分隔符，使用#作为分隔符，从而避免/需要转义。   
虽然很多情况下，都是看到前后一致的分隔符，但是，大家需要了解一下，[<div>.\*?</div>]这个表达式在PHP里也是合法的。（不提倡使用，仅了解！）

### 15. 模式修饰符

模式修饰符在许多程序语言中都支持的，比如最常见的是i，不区分大小写，如javascript里的/[a-z0-9]/i，表示匹配字母数字，不区分大小写。

本人在写php正则时常用的模式修饰符主要有i和s，如：   
$pattern = '#[a-z0-9]+#is';

模式修饰符s的作用主要是的.能够匹配换行，在处理换行数据时，通常会用到。

在PHP中，模式修饰符有两种用法，一种是上面的，在分隔符后面的模式修饰符，它的作用范围是全局；另一种是在正则表达式中间的。   
例如：

1. 正则：/((?i)[A-Z]+)c/
2. 测试字符：abcABC
3. 匹配：abc
4. 说明：局部（ab）的大小写被忽略了，(?i)的作用范围在分组1内

如果把正则改成：/([A-Z]+)c/i，则匹配结果将是：abcABC   
示例地址：[PHP正则表达式中间的模式修饰符](https://regex101.com/r/tS8pP3/1) （选择Version 1/2切换版本查看结果区别）

关于PHP模式修饰符的讲解，请查看[PHP手册](http://www.zjmainstay.cn/php-manual)中的《[PHP模式修饰符](http://php.net/manual/zh/reference.pcre.pattern.modifiers.php)》。

关于常用字符的使用差不多到这里，还有更多的请参考[正则表达式30分钟入门教程](http://deerchao.net/tutorials/regex/regex.htm)，这是我看过比较全面的正则入门资料。

## 二、 操作：定锚点

注：这里的锚点区分于正则原本关于锚点的定义，此处是确定的参照文本的意思，如a标签里的<a

每一个正则都是有针对性的，只有这样正则才有意义。因此，写正则之前，先观察你要解析的数据，找准唯一的锚点，比如，你要解析一个页面的title标签，得到title内容，那么这个title就是锚点。有时候，所要取的数据确实无法定位一个唯一的锚点，那么，你可以分解数据，先通过一个唯一锚点锁定你的数据块，取出来之后，再对这个数据块取数据即可。比如，有这么一段源字符串：

1. <div id="module\_1">
2. <div class="content">
3. content 1
4. </div>
5. </div>
6. <div id="module\_2">
7. <div class="content">
8. content 2
9. </div>
10. </div>

你直接通过class="content"来匹配数据的话很明显会得到两个，那么，你可以扩展它的数据域，先以id="module\_1"作为锚点，获取整个

1. <div id="module\_1">
2. <div class="content">
3. content 1
4. </div>
5. </div>

然后在对这个数据块的数据处理，得到class="contents"的内容即可。   
因此，这里用到2个正则：

（1）<div id="module\_1">(.\*?)</div>\s\*<div id="module\_2">   
（2）<div class="content">(.\*?)</div>

当然，这个正则可以改进为：   
<div id="module\_1">\s\*<div class="content">(.\*?)</div>

**注：为了更清晰查看，前面源码做了换行，匹配失败的朋友，可以修改 .\*? 为 [\s\S]\*? 修正正则。**

### 总结：锚点，就是能唯一定位你数据的标识

## 三、 操作：去噪点

所谓去噪点，就是把无关的东西都当浮云，用通配符过掉它，只关心我们想要的数据，比如：   
<meta content="text/html; charset=utf-8" http-equiv="content-type">   
要从这里得到字符集utf-8，我们需要怎么做？   
首先，定位锚点，有<meta 、charset=和utf-8后面的"，其他都是浮云~   
因此得到正则：   
<meta[^>]\*?charset=([^"]+)"   
即可，用子模式取数据[1]就能得到utf-8

### 总结：关心的留下，不关心的都是浮云

## 四、 操作：取数据

关于取数据，上面一大篇下来大家应该有概念了，就是利用子模式来获取，这里不再赘述。

### 总结：子模式计数，数左括号从1开始，排除非捕获组的左括号

## 五 正则表达式高级用法

1. 概念一：按单字符匹配

2. 概念二：匹配优先和不匹配优先

3. 概念三：贪婪模式与非贪婪模式

4. 概念四：环视（断言）

5. 概念五：平衡组

6. 概念六：模式修饰符

7. 附：正则三段论应用

概念一：按单字符匹配

正则里面的数据都是按照单个字符来进行匹配的，这个通过数值区间的例子最容易体现出来，比如，示例一：

我要匹配0-15的数值区间，用正则来写的话，便是[0-9]|1[0-5]，这里，便是把0-9这种单字符的情况，和10-15这种多字符的情况拆分开了，使用分支|来区分开，表示要么是0-9，要么是10-15。   
上面是两位数值的情况，现在延伸至1-65535，我个人的处理思想是从大到小，一块块分解：

1. 1. 65530-65535 ==> 6553[0-5] 末位区间0-5
2. 2. 65500-65529 ==> 655[0-2][0-9] 第四位区间0-2，末位区间0-9
3. 3. 65000-65499 ==> 65[0-4][0-9]{2} 第三位区间0-4，后两位0-9
4. 4. 60000-64999 ==> 6[0-4][0-9]{3} 第二位区间0-4，后三位0-9
5. 5. 10000-59999 ==> [1-5][0-9]{4} 第一位区间1-5，后四位0-9
6. 6. 1-9999 ==> [1-9][0-9]{0,3} 第一位只能是1-9，后三位可有可无

最后组合起来：   
(6553[0-5]|655[0-2][0-9]|65[0-4][0-9]{2}|6[0-4][0-9]{3}|[1-5][0-9]{4}|[1-9][0-9]{0,3})   
便得到1-65535匹配正则。

根据数据处理需求，可以在正则前后加上^和$，以匹配整个数据串，或者前后加入\b，把它当做单词边界处理。**没有限定字符的边界往往是js正则判断中常见的错误之一**。

概念二：匹配优先和不匹配优先

匹配优先和不匹配优先从字面理解也是比较容易的，所谓匹配优先，就是，能匹配我就先匹配；不匹配优先就是，能不匹配我就先不匹配，这段匹配先跳过，先看看后面的匹配能不能通过。

概念三：贪婪模式与非贪婪模式

正则的贪婪模式和非贪婪模式是一个比较容易混淆的概念，不过，我们可以这么理解，一个人很贪婪，所以他会能拿多少拿多少，换过来，那就是贪婪模式下的正则表达式，能匹配多少就匹配多少，尽可能最多。而非贪婪模式，则是能不匹配就不匹配，尽可能最少。

下面举个例子，示例二：

需求：匹配1后面跟任意个0

源串：10001

使用贪婪模式：10\* 结果：1000 和 1

使用非贪婪模式：10\*? 结果：1 和 1

首先，\*是匹配0个或多个的意思。

1. 贪婪模式下，它表示，首先匹配一个1，然后匹配1后面的0，最多可以匹配3个0，因此得到1000，然后第二次又匹配到一个1，但是后面没有0，因此得到1；
2. 非贪婪模式下，它表示，首先匹配一个1，然后1后面的0，能不匹配就不匹配了，所以，它每次都只是匹配了一个1。

看到这里，也许，有些人觉得，哎呀，我懂了！   
那么，下来我们改改，看看你是不是真懂了。

示例三：

需求：匹配1后面跟任意个0，再跟一个1

源串：10001

使用贪婪模式：10\*1 结果：10001

使用非贪婪模式：10\*?1 结果：10001

为什么这两次的结果一样了？

因为，正则表达式要判断完这整个正则才算成功，这种情况下，

1. 贪婪模式，首先匹配一个1，然后匹配1后面尽可能多的0，发现3个，再匹配0后面的一个1，正则表达式匹配完，完成匹配，得到10001；
2. 非贪婪模式，首先匹配一个1，然后，0\*?是非贪婪模式，它不想匹配了（非贪婪模式不匹配优先），看看后面是不是1了？然后发现哎妈呀，后面是个0啊，然后它回头，不能再偷懒了，用0\*?匹配一个0吧，然后匹配到10，又不想匹配了，看看后面有没有1了，还是没有，又回去用0\*?匹配掉一个0，得到100，继续偷懒，但是发现后面还不是1，然后又用0\*?匹配得到1000，最后，发现真不容易啊，终于看到1了，匹配得到10001，正则表达式匹配完，完成匹配。

看到这里，是不是懂了？   
**那究竟哪个模式好呢？**

什么时候使用贪婪模式，什么时候使用非贪婪模式，哪个性能好，哪个性能不好，不能一概而论，要根据情况分析。   
下面我举个例子：   
源码：

1. <a href="http://www.zjmainstay.cn/my-regexp" target="\_blank" title="我眼里的正则表达式（入门）">我眼里的正则表达式（入门）</a>
2. <a title="我眼里的正则表达式（入门）" href="http://www.zjmainstay.cn/my-regexp" target="\_blank">我眼里的正则表达式（入门）</a>

正则1：<a [^>]\*?href="([^"]\*?)"[^>]\*?>([^<]\*?)</a>（238次）   
正则2：<a [^>]\*?href="([^"]\*)"[^>]\*>([^<]\*)</a>（65次）   
正则3：<a [^>]\*href="([^"]\*)"[^>]\*>([^<]\*)</a>（136次）   
附：执行次数的获取请下载正则表达式测试工具：[RegexBuddy 4.1.0-正则测试工具.rar](http://www.zjmainstay.cn/download/category/4-regexp?download=21%3aregexbuddy)，使用里面的Debug功能。

正则1是通用写法，正则2是在确定字符不会溢出的情况下消除非贪婪模式，正则3是证明并不是全部消除非贪婪模式就是最优。

因此，关于贪婪模式好还是非贪婪模式好的讨论，只能说根据需求而定，不过，在平时的时候用，一般使用非贪婪模式较多，因为贪婪模式经常会由于元字符范围限制不严谨而导致匹配越界，得到非预期结果。

在确定的数据结构里，可以尝试使用[^>]\*>这样的排除字符贪婪模式替换非贪婪模式，提升匹配的效率。注意，贪婪部分（[^>]\*）的匹配，最好不要越过其后面的字符（>），否则会导致贪婪模式下的回溯，如正则3，[^>]\*的匹配越过了href，一直匹配到>为止，而这时候再匹配href，会匹配不到而导致多次回溯处理，直到回溯到href前的位置，后面才继续了下去。

另外，需要注意一点，无论使用贪婪模式还是非贪婪模式，在不同语言需要注意回溯次数和嵌套次数的限制，比如在PHP中，pcre.backtrack\_limit=100000，pcre.recursion\_limit=100000。

概念四：环视（断言/零宽断言）

环视，在不同的地方又称之为零宽断言，简称断言。   
用一句通俗的话解释：   
环视，就是先从全局环顾一遍正则，（然后断定结果，）再做进一步匹配处理。   
断言，就是先从全局环顾一遍正则，然后断定结果，再做进一步匹配处理。

两个虽然字面不一样，意思却是同一个，都是做全局观望，再做进一步处理。

环视的作用相当于**对其所在位置加了一个附加条件，只有满足这个条件，环视子表达式才能匹配成功**。

环视主要有以下4个用法：   
(?<=exp) 匹配前面是exp的数据   
(?<!exp) 匹配前面不是exp的数据   
(?=exp) 匹配后面是exp的数据   
(?!exp) 匹配后面不是exp的数据

示例四：   
(?<=B)AAA 匹配前面是B的数据，即BAAA匹配，而CAAA不匹配   
(?<!B)AAA 匹配前面不是B的数据，即CAAA匹配，而BAAA不匹配   
AAA(?=B) 匹配后面是B的数据，即AAAB匹配，而AAAC不匹配   
AAA(?!B) 匹配后面不是B的数据，即AAAC能匹配，而AAAB不能匹配

另外，还会看到(?!B)[A-Z]这种写法，其实它是[A-Z]范围里，排除B的意思，前置的(?!B)只是对后面数据的一个限定，从而达到过滤匹配的效果。

因此，环视做排除处理是比较实用的，比如，示例五：

需求：字母、数字组合，不区分大小写，不能纯数字或者纯字母，6-16个字符。

通用正则：^[a-z0-9]{6,16}$ 字母数字组合，6-16个字符

排除纯字母：(?!^[a-z]+$)

排除纯数字：(?!^[0-9]+$)

组合起来：(?!^[a-z]+$)(?!^[0-9]+$)^[a-z0-9]{6,16}$

注意，**环视部分是不占宽度的**，所以有零宽断言的叫法。   
所谓不占宽度，可以分成两部分理解：   
1、环视的匹配结果不纳入数据结果   
2、环视它匹配过的地方，下次还能用它继续匹配。

如果不是环视，则匹配过的地方，不能再匹配第二次了。

上面示例四体现了：**环视的匹配结果不纳入数据结果**，它的结果：

1. (?<=B)AAA 源串：BAAA 结果：AAA
2. (?<!B)AAA 源串：CAAA 结果：AAA
3. AAA(?=B) 源串：AAAB 结果：AAA
4. AAA(?!B) 源串：AAAC 结果：AAA

而示例五体现了：**环视它匹配过的地方，下次还能用它继续匹配**   
因为，整个匹配过程中，正则表达式一共走了3次字符串匹配，第一次匹配不全部是字母，第二次匹配不全部是数字，第三次匹配全部是字母数字组合，6-16个字符。

1. 扩展部分：
2. `[A-Z](?<=B)` [A-Z]范围等于B
3. `[A-Z](?<!B)` [A-Z]范围排除B
4. `(?!B)[A-Z]` [A-Z]范围排除B

附： js不支持(?<=exp) 和 (?<!exp) 语法

概念五：平衡组

平衡组并不是所有程序语言都支持，而我本人使用的PHP语言就不支持，所以平时接触也是比较少的。

平衡组主要用到下面四个语法：

1. (?'group') 把捕获的内容命名为group,并压入堆栈(Stack)
2. (?'-group') 从堆栈上弹出最后压入堆栈的名为group的捕获内容，如果堆栈本来为空，则本分组的匹配失败
3. (?(group)yes|no) 如果堆栈上存在以名为group的捕获内容的话，继续匹配yes部分的表达式，否则继续匹配no部分
4. (?!) 零宽负向先行断言，由于没有后缀表达式，如没有(?!B)的B，试图匹配总是失败

在PHP中是支持(?(group)yes|no)语法的，这里的group是分组编号，即子模式编号，如(A)?(?(1)yes|no) ，匹配Ayes 和 no

下面这里引用《[正则表达式30分钟入门教程#平衡组](http://deerchao.net/tutorials/regex/regex.htm#balancedgroup)》关于<>配对匹配的例子，展示平衡组用法，

1. < #最外层的左括号
2. [^<>]\* #最外层的左括号后面的不是括号的内容
3. (
4. (
5. (?'Open'<) #碰到了左括号，在黑板上写一个"Open"
6. [^<>]\* #匹配左括号后面的不是括号的内容
7. )+
8. (
9. (?'-Open'>) #碰到了右括号，擦掉一个"Open"
10. [^<>]\* #匹配右括号后面不是括号的内容
11. )+
12. )\*
13. (?(Open)(?!)) #在遇到最外层的右括号时，判断黑板上还有没有没擦掉的"Open"；如果还有，则匹配失败
14. > #最外层的右括号
15. 平衡组的一个最常见的应用就是匹配HTML,下面这个例子可以匹配嵌套的<div>标签：
16. <div[^>]\*>[^<>]\*(((?'Open'<div[^>]\*>)[^<>]\*)+((?'-Open'</div>)[^<>]\*)+)\*(?(Open)(?!))</div>

概念六：模式修饰符

模式修饰符在许多程序语言中都支持的，比如最常见的是i，不区分大小写，如javascript里的/[a-z0-9]/i，表示匹配字母数字，不区分大小写。

本人在写php正则时常用的模式修饰符主要有i和s，如：   
$pattern = '#[a-z0-9]+#is';

模式修饰符s的作用主要是的.能够匹配换行，在处理换行数据时，通常会用到。

在PHP中，模式修饰符有两种用法，一种是上面的，在分隔符后面的模式修饰符，它的作用范围是全局；另一种是在正则表达式中间的。   
例如：

1. 正则：/((?i)[A-Z]+)c/
2. 测试字符：abcABC
3. 匹配：abc
4. 说明：局部（ab）的大小写被忽略了，(?i)的作用范围在分组1内

如果把正则改成：/([A-Z]+)c/i，则匹配结果将是：abcABC   
示例地址：[PHP正则表达式中间的模式修饰符](https://regex101.com/r/tS8pP3/1)

关于PHP模式修饰符的讲解，请查看[PHP手册](http://www.zjmainstay.cn/php-manual)中的《[PHP模式修饰符](http://php.net/manual/zh/reference.pcre.pattern.modifiers.php)》。

概念七：正则三段论应用

《[我眼里的正则表达式（入门）](http://www.zjmainstay.cn/my-regexp)》 和 本文《[深入正则表达式应用](http://www.zjmainstay.cn/deep-regexp)》几乎倾尽本人正则学习全部思想，但是很多读者反馈，看晕了！看到如此点评，实属无奈，因此，有必要追加本节，来个整体统筹运用，希望能让大家犹如拨云见月，洞悉其中的精义。

正则三段论：定锚点，去噪点，取数据

两篇文章中，最重要的部分当属**正则三段论**：**定锚点，去噪点，取数据**，它是整个正则处理过程中的灵魂，它贯穿整个正则撰写过程。

下面举例说明它的思想，示例六：

1. 源数据：标题：深入正则表达式应用，作者：Zjmainstay
2. 需求：匹配作者名字

我要从源数据取到**Zjmainstay**这个作者名，那么，在这里，**作者：**就是我们所说的锚点，因为在上面这段数据中它能够唯一定位到我们的数据**Zjmainstay**（就在它后面），因此，我们得到

(1) 定锚点：**作者：**

而在这里，我们不需要关心标题什么的，因此，**标题：深入正则表达式应用，**就是我们的噪点，因此，我们得到

(2) 去噪点

最后，我们确定**作者：**后面就是我们的数据，这个数据可以是任意字符，因此，我们得到正则：   
作者：(.\*)

而噪点部分，因为不会对数据取值造成干扰，直接去掉，不需要引入正则中。

下面再举一个噪点干扰的例子，示例七：

1. 源数据： <a href="http://www.zjmainstay.cn/my-regexp" class="demo8" title="正则三段论应用举例">正则表达式入门教程</a>
2. 需求：提取链接和标题，还有a标签的文字

看到这个源数据和需求，我们必须定位好锚点，主要有：   
1. <a //必须是a标签   
2. href=" 和 " //href=""的内容得到链接   
3. title=" 和 " //title=""的内容得到标题   
4. > 和 </a> //>和</a>的内容得到标签文字

然后，其他的都是噪点，使用.\*?替代，需要提取的数据部分使用括号获取子模式，得到分组数据，因此得到正则：  
<a href="(.\*?)".\*?title="(.\*?)">(.\*?)</a>

看到这里，也许有朋友觉得，我还是不会写，那么，再来一个更简单的构建方法，细化步骤，从源串逐步得到正则，示例八：

1. 1. 直接拷贝源串，特殊字符处理转义（本例没特殊字符）
2. <a href="http://www.zjmainstay.cn/my-regexp" class="demo8" title="正则三段论应用举例">正则表达式入门教程</a>
3. 2. 从左到右，一段段转化
4. 2.1 <a href="(.\*?)" class="demo8" title="正则三段论应用举例">正则表达式入门教程</a>
5. 2.2 <a href="(.\*?)".\*?title="正则三段论应用举例">正则表达式入门教程</a>
6. 2.3 <a href="(.\*?)".\*?title="(.\*?)">正则表达式入门教程</a>
7. 2.4 <a href="(.\*?)".\*?title="(.\*?)">(.\*?)</a>
8. 3. 得到最终的正则
9. <a href="(.\*?)".\*?title="(.\*?)">(.\*?)</a>

至此，正则三段论的基本思想已经展示完毕，大家还有什么不解请评论留言，本人看到会第一时间给予回复。

熟悉正则三段论处理思想，剩下的便是基本语义的熟练程度了，这个通过多用多练可以达到熟能生巧的境界。

最后留下一句**至尊提醒**：.是**万能字符**，大家看着用，遇到换行使用[\s\S]替换.即可。

（完）