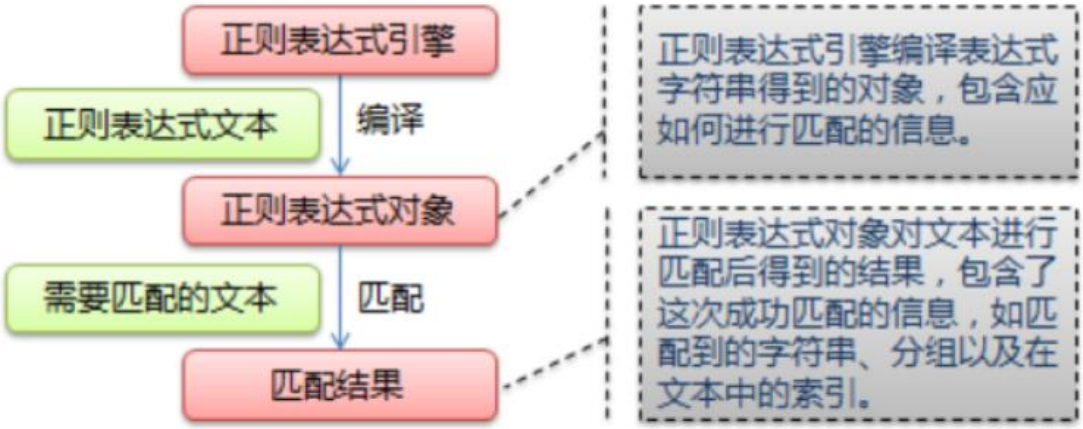
用正则表达式提取数据

正则表达式，又称规则表达式，通常被用来检索、替换那些符合某个模式(规则)的文本。



复习回顾

单字符匹配，多字符匹配，匹配分组，

对文本进行匹配查找的一系列方法

- match 方法：从起始位置开始查找，一次匹配
- search 方法：从任何位置开始查找，一次匹配
- findall 方法：全部匹配，返回列表
- finditer 方法：全部匹配，返回迭代器
- split 方法：分割字符串，返回列表
- sub 方法：替换

数据提取中常用的操作

抓取标签间的内容

案例：抓取 title 标签间的内容

```
from urllib import request
import re

response = request.urlopen("http://fanyi.baidu.com/")
html = response.read()
html = html.decode("utf-8")
#print(html)

pat = r'(?<=<title>).*?(?=</title>)'
ex = re.compile(pat, re.M|re.S)  #只取中间的文字
obj = re.search(ex, html)
title = obj.group(1)
print(title)
```

案例：抓取超链接标签间的内容

```
from urllib import request
import re
import chardet

def down(url):
    head = {}
    #写入 User Agent 信息
    head['User-Agent'] = 'Mozilla/5.0 (Linux; Android 4.1.1; Nexus 7 Build/JRO03D) AppleWebKit/535.19 (KHTML, like Gecko) Chrome/18.0.1025.166  Safari/535.19'
    #创建 Request 对象
    req = request.Request(url, headers=head)
    response = request.urlopen(req)
    html = response.read()
    charset = chardet.detect(html)
    #print(charset)
    print(charset['encoding'])
    html = html.decode(charset['encoding'])
```

```
    return html


if __name__ == "__main__":
    # 获取超链接<a>和</a>之间内容
    url = "https://www.baidu.com/"
    html = down(url)
    pat = re.compile(r'<a .*?>(.*?)</a>',re.M|re.S)
    texts = pat.findall(html)
    for t in texts:
        print(t)
```

案例：抓取 tr\td 标签间的内容

```
if __name__ == "__main__":
    content = '''
    <html>
<head><title>表格</title></head>
<body>
    <table    border=1>
        <tr><th>学号</th><th>姓名</th></tr>
        <tr><td>1001</td><td>李白</td></tr>
        <tr><td>1002</td><td>杜甫</td></tr>
    </table>
</body>
</html>
    '''
    # 获取<tr></tr>间内容
    res = r'<tr>(.*?)</tr>'
    texts = re.findall(res, content, re.S | re.M)
    for m in texts:
        print(m)

    # 获取<th></th>间内容
    for m in texts:
        res_th = r'<th>(.*?)</th>'
        m_th = re.findall(res_th, m, re.S | re.M)
        for t in m_th:
            print(t)

    # 直接获取<td></td>间内容
    res = r'<td>(.*?)</td><td>(.*?)</td>'
```

```
    texts = re.findall(res, content, re.S | re.M)
    for m in texts:
        print(m[0], m[1])
```

抓取标签中的参数

案例：抓取超链接标签的 URL

```
import re

content = '''
<a href="http://news.baidu.com" name="tj_trnews" class="mnav">新闻</a>
<a href="http://www.hao123.com" name="tj_trhao123" class="mnav">hao123</a>
<a href="http://map.baidu.com" name="tj_trmap" class="mnav">地图</a>
<a href="http://v.baidu.com" name="tj_trvideo" class="mnav">视频</a>
'''

res = r"(?<=href=\").+?(?=\")|(?<=href=\').+?(?=\')"
urls = re.findall(res, content, re.I | re.S | re.M)
for url in urls:
    print(url)
```

案例：抓取图片超链接标签的 URL

```
import re

content = '''<img alt="Python" src="http://www..csdn.net/eastmount.jpg" />'''
urls = re.findall('src="(.*?)"', content, re.I|re.S|re.M)
print(urls)
```

案例：获取 URL 中最后一个参数

```
import re

content = '''<img alt="Python" src="http://www..csdn.net/eastmount.jpg" />'''
urls = 'http://www..csdn.net/eastmount.jpg'
name = urls.split('/')[-1]
print(name)
```

字符串处理

案例：查找子字符串

```
import re
content = '''<div><a href="#">锚点</a></div>'''
start = content.find(r'<a href="#">') + len(r'<a href="#">') #起点位置
end = content.find(r'</a>')                    #重点点位置
infobox = content[start:end]
print(infobox)
```

案例：字符串替换

爬取过程中可能会爬取到无关变量，此时需要对无关内容进行过滤，这里推荐使用 replace

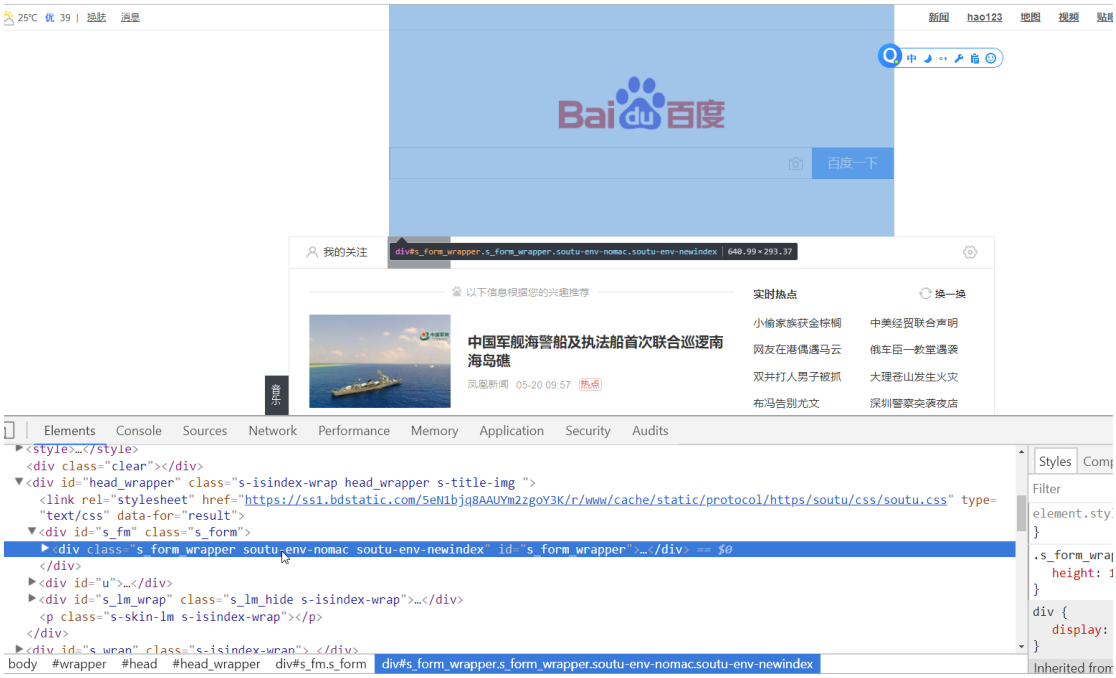
函数和正则表达式进行处理。

```
import re

content = '''
<tr><td>1001</td><td>杨秀璋<br /></td></tr>
<tr><td>1002</td><td>颜  娜</td></tr>
<tr><td>1003</td><td><B>Python</B></td></tr>
'''

res = r'<td>(.*?)</td><td>(.*?)</td>'
texts = re.findall(res, content, re.S|re.M)
for m in texts:
    value0 = m[0].replace('<br />', '').replace(' ', '')
    value1 = m[1].replace('<br />', '').replace(' ', '')
    if '<B>' in    value1:
        m_value = re.findall(r'<B>(.*?)</B>', value1, re.S|re.M)
        print(value0, m_value[0])
    else:
        print(value0, value1)
```

审查元素

可以从浏览器中查看网页代码，例如谷歌浏览器，在任意界面单击右键选择检查，也就是审

查元素(不是所有页面都可以审查元素的，例如起点中文网付费章节就不行.)，以百度界面为例，截图如下：



案例：爬取内涵吧，正则表达式提取数据

url 规律分析

第一页 url: https://www.neihan8.com/article/index.html

第二页 url: https://www.neihan8.com/article/index_2.html

第三页 url: https://www.neihan8.com/article/index_3.html

第四页 url: https://www.neihan8.com/article/index_4.html

url 最后一个数字不同，它表示页码，只需要修改该值即可爬取所有段子

案例：爬取中超联赛新闻

附录：正则表达式匹配规则

语法	说明	表达式实例	完整匹配的字符串
	字符		
一般字符	匹配自身	abc	abc
.	匹配任意除换行符"\n"外的字符。 在DOTALL模式中也能匹配换行符。	a.c	abc
\	转义字符，使后一个字符改变原来的意思。 如果字符串中有字符*需要匹配，可以使用*或者字符集[*]。	a\.c a\\c	a.c a\c
[...]	字符集（字符类）。对应的位置可以是字符集中任意字符。 字符集中的字符可以逐个列出，也可以给出范围，如[abc]或[a-c]。第一个字符如果是^则表示取反，如[^abc]表示不是abc的其他字符。 所有的特殊字符在字符集中都失去其原有的特殊含义。在字符集中如果要使用]、-或^，可以在前面加上反斜杠，或把]、-放在第一个字符，把^放在非第一个字符。	a[bcd]e	abe ace ade

语法	说明	表达式实例	完整匹配的字符串
	预定义字符集（可以写在字符集[...]中）		
\d	数字：[0-9]	a\dc	a1c
\D	非数字：[^\d]	a\Dc	abc
\s	空白字符：[<空格>\t\r\n\f\v]	a\sc	a c
\S	非空白字符：[^\s]	a\Sc	abc
\w	单词字符：[A-Za-z0-9_]	a\wc	abc
\W	非单词字符：[^\w]	a\Wc	a c

语法	说明	表达式实例	完整匹配的字符串
	数量词（用在字符或(...)之后）		
*	匹配前一个字符0或无限次。	abc*	ab abccc
+	匹配前一个字符1次或无限次。	abc+	abc abccc
?	匹配前一个字符0次或1次。	abc?	ab abc
{m}	匹配前一个字符m次。	ab{2}c	abbc
{m,n}	匹配前一个字符m至n次。 m和n可以省略：若省略m，则匹配0至n次；若省略n，则匹配m至无限次。	ab{1,2}c	abc abbc
*? +? ?? {m,n}?	使 * + ? {m,n}变成非贪婪模式。	示例将在下文中介绍。	

语法	说明	表达式实例	完整匹配的字符串
边界匹配（不消耗待匹配字符串中的字符）			
^	匹配字符串开头。 在多行模式中匹配每一行的开头。	^abc	abc
$	匹配字符串末尾。 在多行模式中匹配每一行的末尾。	abc$	abc
\A	仅匹配字符串开头。	\Aabc	abc
\Z	仅匹配字符串末尾。	abc\Z	abc
\b	匹配\w和\W之间。	a\b!bc	a!bc
\B	[^\b]	a\Bbc	abc

语法	说明	表达式实例	完整匹配的字符串
逻辑、分组			
|	|代表左右表达式任意匹配一个。 它总是先尝试匹配左边的表达式，一旦成功匹配则跳过匹配右边的表达式。 如果|没有被包括在()中，则它的范围是整个正则表达式。	abc|def	abc def
(...)	被括起来的表达式将作为分组，从表达式左边开始每遇到一个分组的左括号'('，编号+1。 另外，分组表达式作为一个整体，可以后接数量词。表达式中的|仅在该组中有效。	(abc){2} a(123|456)c	abcabc a456c
(?P<name>...)	分组，除了原有的编号外再指定一个额外的别名。	(?P<id>abc){2}	abcabc
\<number>	引用编号为<number>的分组匹配到的字符串。	(\d)abc\1	1abc1 5abc5
(?P=name)	引用别名为<name>的分组匹配到的字符串。	(?P<id>\d)abc(?P=id)	1abc1 5abc5

语法	说明	表达式实例	完整匹配的字符串
特殊构造（不作为分组）			
(?:...)	(...)的不分组版本，用于使用'|'或后接数量词。	(?:abc){2}	abcabc
(?iLmsux)	iLmsux的每个字符代表一个匹配模式，只能用在正则表达式的开头，可选多个。匹配模式将在下文中介绍。	(?i)abc	AbC
(?#...)	#后的内容将作为注释被忽略。	abc(?#comment)123	abc123
(?=...)	之后的字符串内容需要匹配表达式才能成功匹配。 不消耗字符串内容。	a(?=\d)	后面是数字的a
(?!...)	之后的字符串内容需要不匹配表达式才能成功匹配。 不消耗字符串内容。	a(?!\d)	后面不是数字的a
(?<=...)	之前的字符串内容需要匹配表达式才能成功匹配。 不消耗字符串内容。	(?<=\d)a	前面是数字的a
(?<!...)	之前的字符串内容需要不匹配表达式才能成功匹配。 不消耗字符串内容。	(?<!\d)a	前面不是数字的a
(?(id/name) yes-pattern |no-pattern)	如果编号为id/别名为name的组匹配到字符，则需要匹配yes-pattern，否则需要匹配no-pattern。 |no-patern可以省略。	(\d)abc(?(1)\d|abc)	1abc2 abcabc

re 基本用法（选）

Python re 模块导入：

import re

单字符匹配

.匹配任意字符(除了\n)

[]匹配列举的字符

\d 匹配数字

\w 匹配单词字符

范例

```
# .匹配任意字符(除了\n)
ret = re.match(".","abc")
print(ret.group())

#匹配列举的字符
ret = re.match("[hH]","hello Python")
ret.group()

#匹配数字
ret = re.match("嫦娥\d 号","嫦娥 3 号发射成功")
print(ret.group())
ret = re.match("[0-9]","7Hello Python")
print(ret.group())

#匹配单词字符
ret = re.match('\w\w\w\w\w','hello world')
ret.group()
```

多字符匹配

*匹配一个字符 0 到多次

+匹配前一个元字符 1 到多次

？匹配前一个元字符 0 到 1 次

{m,n} 匹配前一个元字符 m 到 n 次

范例

```
#匹配一个字符 0 到多次
ret = re.match("[A-Z][a-z]*","China")
ret.group()

#匹配前一个元字符 1 到多次
ret = re.match("[a-zA-Z_]+","__init__")
ret.group()

#匹配前一个元字符 0 到 1 次
ret = re.match("[1-9]?[0-9]","777")
ret.group()

#匹配前一个元字符 m 到 n 次
ret = re.match("[a-zA-Z0-9_-]{8,20}","2018-07-01")
ret.group()
```

表示边界

```
ret = re.match("^[\w]{4,20}@163\.com$", "xiaoWang@163.com")
ret.group()
```

匹配分组

```
#引用分组 num 匹配到的字符串
ret = re.match(r"<(\w+)><(\w+)>.+</\2></\1>", "<html><h1>www.baidu.com</h1></html>")
ret.group()

# 命名分组，引用别名为 name 的分组匹配到的字符串
ret = re.match("<(?P<name1>\w*)><(?P<name2>\w*)>.*</(?P=name2)></(?P=name1)>",
"<html><h1>www.baidu.com</h1></html>")
ret.group()
```

模式

re.I 忽略大小写的匹配模式

re.S '.' 可匹配任何字符，包括换行符

re.X 冗余模式，忽略正则表达式中的空白和#号的注释

re.M 多行模式

范例

```
#忽略大小写
s = 'hello World!'
regex = re.compile("hello world!", re.I)
print(regex.match(s).group())

#匹配换行
s = '''first line
second line
third line'''
#
regex = re.compile(".+")
print(regex.findall(s))

# re.S
regex_dotall = re.compile(".+", re.S)
print(regex_dotall.findall(s))
```

r 前缀

正则表达式使用\对特殊字符进行转义，所以如果我们要使用原始字符串，只需加一个 r 前缀，示例：

```
r'qiku\t\.\tpython'
```

编译模式（选）

使用 compile() 函数将正则表达式的字符串形式编译为一个 Pattern 对象，然后调用正则表达式对象的相应方法。

推荐使用编译模式，正则对象可以多次使用，可以大大地提高搜索的效率

compile 函数

compile 函数用于编译正则表达式，生成一个 Pattern 对象

```
import re

# 将正则表达式编译成 Pattern 对象
pattern = re.compile(r'\d+')
```

Pattern 对象

提供了对文本进行匹配查找的一系列方法

- match 方法：从起始位置开始查找，一次匹配
- search 方法：从任何位置开始查找，一次匹配
- findall 方法：全部匹配，返回列表
- finditer 方法：全部匹配，返回迭代器
- split 方法：分割字符串，返回列表
- sub 方法：替换

match 方法

从字符串的头部或者指定位置开始查找一次匹配，只要找到了一个匹配的结果就返回

```
match(string[, pos[, endpos]])
```

string 是待匹配的字符串

pos 和 **endpos** 可选参数，指定字符串的起始和终点位置，默认值分别是 0 和 len (字符串长度)。

当匹配成功时，返回一个 Match 对象，如果没有匹配上，则返回 None。

```
import re
pattern = re.compile(r'\d+')  # 用于匹配至少一个数字

m = pattern.match('one12twothree34four')  # 查找头部，没有匹配
print(m)

m = pattern.match('one12twothree34four', 2, 10) # 从'e'的位置开始匹配，没有匹配
print(m)

m = pattern.match('one12twothree34four', 3, 10) # 从'1'的位置开始匹配，正好匹配
print(m)                                        # 返回一个 Match 对象

print(m.group(0))   # 可省略 0
print(m.start(0))   # 可省略 0
print(m.end(0))     # 可省略 0
print(m.span(0))    # 可省略 0
```

Match 对象

group([group1, ...])获得一个或多个分组匹配的字符串

group() 或 group(0) 获得整个匹配的子串

start([group]) 获取分组匹配的子串在整个字符串中的起始位置（子串第一个字符的索引）

end([group]) /获取分组匹配的子串在整个字符串中的结束位置（子串最后一个字符的索引 +1）

span([group]) 方法返回 (start(group), end(group))。

范例

```
import re
pattern = re.compile(r'([a-z]+) ([a-z]+)', re.I)  # re.I 表示忽略大小写
m = pattern.match('Hello World Wide Web')

print(m)     # 匹配成功，返回一个 Match 对象
print(m.group(0))  # 返回匹配成功的整个子串
print(m.span(0))   # 返回匹配成功的整个子串的索引
print(m.group(1))  # 返回第一个分组匹配成功的子串
print(m.span(1))   # 返回第一个分组匹配成功的子串的索引
print(m.group(2))  # 返回第二个分组匹配成功的子串
print(m.span(2))   # 返回第二个分组匹配成功的子串
print(m.groups())  # 等价于 (m.group(1), m.group(2), ...)
print(m.group(3))   # 不存在第三个分组 IndexError: no such group
```

search 方法

查找字符串的任何位置，只匹配一次，只要找到了一个匹配的结果就返回

`search(string[, pos[, endpos]])`

string 是待匹配的字符串

pos 和 endpos 可选参数，指定字符串的起始和终点位置

当匹配成功时，返回一个 Match 对象，如果没有匹配上，则返回 None。

范例

```
import re
pattern = re.compile('\d+')
m = pattern.search('one12twothree34four')  # 这里如果使用 match 方法则不匹配
print(m.group())
m = pattern.search('one12twothree34four', 10, 30)  # 指定字符串区间
print(m.group())
```

```
print(m.span())
```

findall 方法

以列表形式返回全部能匹配的子串，如果没有匹配，则返回一个空列表。

```
findall(string[, pos[, endpos]])
```

string 待匹配的字符串

pos 和 endpos 可选参数，指定字符串的起始和终点位置。

范例

```
import re
pattern = re.compile(r'\d+')   # 查找数字

result1 = pattern.findall('hello 123456 789')
result2 = pattern.findall('one1two2three3four4', 0, 10)

print(result1)
print(result2)
```

finditer 方法

搜索所有匹配的结果。返回一个迭代器，可顺序访问每一个匹配的结果（**Match 对象**）

范例

```
# -*- coding: utf-8 -*-

import re
pattern = re.compile(r'\d+')

result_iter1 = pattern.finditer('hello 123456 789')
result_iter2 = pattern.finditer('one1two2three3four4', 0, 10)

print(type(result_iter1))
print(type(result_iter2))
```

```
print('result1...')
for m1 in result_iter1:   # m1 是 Match 对象
    print('matching string: {}, position: {}'.format(m1.group(), m1.span()))

print('result2...')
for m2 in result_iter2:
    print('matching string: {}, position: {}'.format(m2.group(), m2.span()))
```

split 方法

按照能够匹配的子串将字符串分割后返回列表

`split(string[, maxsplit])`

maxsplit 用于指定最大分割次数，不指定将全部分割。

范例

```
import re
p = re.compile(r'[\s,;]+')
print(p.split('a,b;; c   d'))
```

sub 方法

方法用于替换。

`sub(repl, string[, count])`

epl 可以是字符串也可以是一个函数:

- ✧ 如果 repl 是字符串，则会使用 repl 去替换字符串每一个匹配的子串
- ✧ 如果 repl 是函数，方法只接受一个参数（Match 对象），并返回一个字符串用于替换。

count 用于指定最多替换次数，不指定时全部替换。

范例

```
import re
p = re.compile(r'(\w+) (\w+)') # \w = [A-Za-z0-9]
s = 'hello 123, hello 456'
```

```
print(p.sub(r'hello world', s)) # 使用 'hello world' 替换 'hello 123' 和
'hello 456'
print(p.sub(r'\2 \1', s))      # 引用分组

def func(m):
    return 'hi' + ' ' + m.group(2)

print p.sub(func, s)
print p.sub(func, s, 1)         # 最多替换一次
```

贪婪模式与非贪婪模式

贪婪模式：在整个表达式匹配成功的前提下，尽可能多的匹配（*）；

非贪婪模式：在整个表达式匹配成功的前提下，尽可能少的匹配（?）；

Python 里数量词默认是贪婪的。

示例一

```
import re


str = 'abbbc'
# 贪婪模式
pattern = re.compile(r'ab*') # * 决定了尽可能多匹配 b,结果是 abbb
result = pattern.match(str)
print(result.group())

# 非贪婪模式
pattern = re.compile(r'ab*?') # *? 决定了尽可能少匹配 b，结果是 a
result = pattern.match(str)
print(result.group())

pattern = re.compile(r'ab+?') # *? 决定了尽可能少匹配 b，结果是 ab
result = pattern.match(str)
print(result.group())
```

示例二

```
import re
# 贪婪模式
```

```
str = "aa<div>test1</div>bb<div>test2</div>cc"
pattern = re.compile(r'<div>.*</div>') #* 决定了尽可能多匹配 b, 结果是<div>test1</div>bb<div>test2</div>
result = pattern.search(str)
print(result.group())

# 非贪婪模式
str = "aa<div>test1</div>bb<div>test2</div>cc"
pattern = re.compile(r'<div>.*?</div>') # *? 决定了尽可能少匹配 b，结果是<div>test1</div>
result = pattern.search(str)
print(result.group())
```

正则表达式测试网址

http://tool.oschina.net/regex/

特殊构造（不作为分组）

(?=...) 之后的字符串内容需要匹配表达式才能成功匹配 a(?=\d) 后面是数则的 a

```
import re
pattern = re.compile(r'a(?=\d)')
matchObj = pattern.match('ba123',1)
print(matchObj.group())
```

结果

a

(?<=...) 之前的字符串内容需要匹配表达式才能成功匹配 (?<=\d)a 前面是数字的 a

```
pattern = re.compile(r'(?<=\d)a')
matchObj = pattern.match('2a')
print(matchObj)
```

结果

a

匹配中文

中文的 unicode 编码范围 主要在 [u4e00-u9fa5] （全角（中文）标点等除外），不过，在大部分情况下，应该是够用的。

假设现在想把字符串 title = u'你好，hello，世界' 中的中文提取出来，可以这么做：

```
import re

title = '你好，hello，世界'
pattern = re.compile(r'[\u4e00-\u9fa5]+')
result = pattern.findall(title)
print(result)
```

思考问题

1、 match 方法和 search 方法的区别？
2、 如何理解贪婪模式和费贪婪模式