# 4.3　强大的 BeautifulSoup

Beautiful Soup 是一个可以从 HTML 或 XML 文件中提取数据的 Python 库。它能够通过你喜欢的转换器实现惯用的文档导航、查找、修改文档的方式。在 Python 爬虫开发中，我们主要用到的是 Beautiful Soup 的查找提取功能，修改文档的方式很少用到。接下来由浅及深介绍 Beautiful Soup 在 Python 爬虫开发中的使用。

## 4.3.1　安装 BeautifulSoup

对于 Beautiful Soup，我们推荐使用的是 Beautiful Soup 4，已经移植到 BS4 中，Beautiful Soup 3 已经停止开发。安装 Beautiful Soup 4 有三种方式：

* 如果你用的是新版的 Debain 或 ubuntu，那么可以通过系统的软件包管理来安装：apt-get install Python-bs4。
* Beautiful Soup 4 通过 PyPi 发布，可以通过 easy\_install 或 pip 来安装。包的名字是 beautifulsoup4，这个包兼容 Python2 和 Python3。安装命令：easy\_install beautifulsoup4 或者 pip install beautifulsoup4。
* 也可以通过下载源码的方式进行安装，当前最新的版本是4.5.1，源码下载地址为 <https://pypi.python.org/pypi/beautifulsoup4/>。运行下面的命令即可完成安装：python setup.py install。

Beautiful Soup 支持 Python 标准库中的 HTML 解析器，还支持一些第三方的解析器，其中一个是 lxml。由于 lxml 解析速度比标准库中的 HTML 解析器的速度快得多，我们选择安装 lxml 作为新的解析器。根据操作系统不同，可以选择下列方法来安装 lxml：

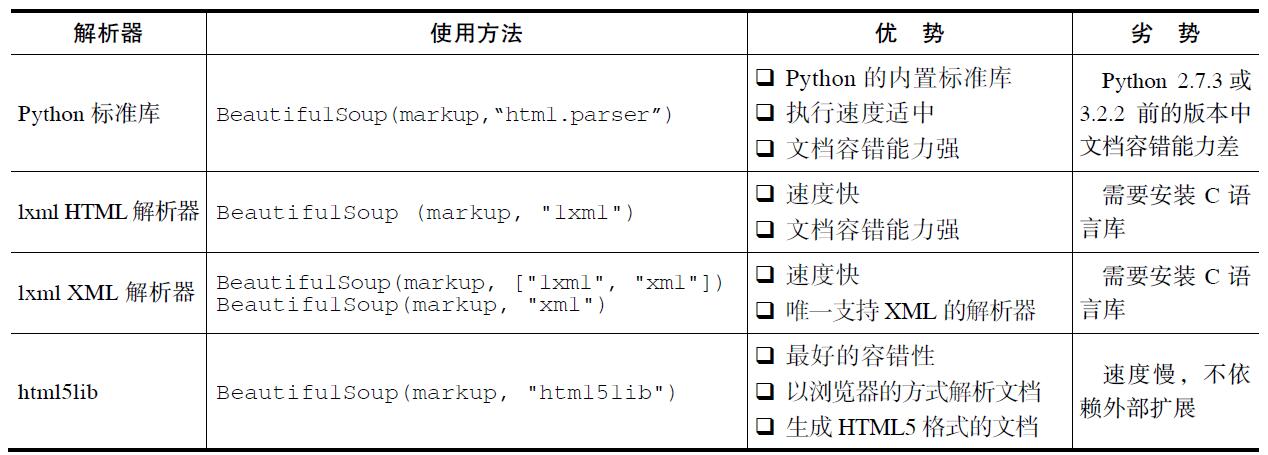
* apt-get install Python-lxml
* easy\_install lxml
* pip install lxml

另一个可供选择的解析器是纯 Python 实现的 html5lib，html5lib 的解析方式与浏览器相同，可以选择下列方法来安装 html5lib：

* apt-get install Python-html5lib
* easy\_install html5lib
* pip install html5lib

表4-9列出了主要的解析器，以及它们的优缺点。

表4-9　解析器比较

[](http://popimage/?src=)

从表4-9中可以看出推荐使用 lxml 作为解析器的原因，因为它效率更高。

## 4.3.2　BeautifulSoup 的使用

安装完 BeautifulSoup，接下来开始讲解 BeautifulSoup 的使用。

1.快速开始

首先导入 bs4 库：from bs4import BeautifulSoup。接着创建包含 HTML 代码的字符串，用来进行解析。字符串如下：

html\_str = """

<html><head><title>The Dormouse's story</title></head>

<body>

<p class="title"><b>The Dormouse's story</b></p>

<p class="story">Once upon a time there were three little sisters; and their names were

<a href="http://example.com/elsie" class="sister" id="link1">*<!-- Elsie -->*</a>,

<a href="http://example.com/lacie" class="sister" id="link2">*<!-- Lacie -->*</a> and

<a href="http://example.com/tillie" class="sister" id="link3">Tillie</a>;

and they lived at the bottom of a well.</p>

<p class="story">...</p>

"""

接下来的数据解析和提取都是以这个字符串为例子。

然后创建 BeautifulSoup 对象，创建 BeautifulSoup 对象有两种方式。一种直接通过字符串创建：

soup = BeautifulSoup(html\_str,'lxml', from\_encoding='utf-8')

另一种通过文件来创建，假如将 html\_str 字符串保存为 index.html 文件，创建方式如下：

soup = BeautifulSoup(open('index.html'))

文档被转换成 Unicode，并且 HTML 的实例都被转换成 Unicode 编码。打印 soup 对象的内容，格式化输出：

**print** soup.prettify()

输入结果如下：

<html>

<head>

<title>

The Dormouse's story

</title>

</head>

<body>

<p class="title">

<b>

The Dormouse's story

</b>

</p>

<p class="story">

Once upon a time there were three little sisters; and their names were

<a class="sister" href="http://example.com/elsie" id="link1">

*<!--Elsie -->*

</a>

,

<a class="sister" href="http://example.com/lacie" id="link2">

*<!--Lacie-->*

</a>

and

<a class="sister" href="http://example.com/tillie" id="link3">

Tillie

</a>

;

and they lived at the bottom of a well.

</p>

<p class="story">

...

</p>

</body>

</html>

Beautiful Soup 选择最合适的解析器来解析这段文档，如果手动指定解析器那么 Beautiful Soup 会选择指定的解析器来解析文档，使用方法如表4-9所示。

2.对象种类

Beautiful Soup 将复杂 HTML 文档转换成一个复杂的树形结构，每个节点都是 Python 对象，所有对象可以归纳为4种：

* Tag
* NavigableString
* BeautifulSoup
* Comment

1）Tag

首先说一下 Tag 对象，Tag 对象与 XML 或 HTML 原生文档中的 Tag 相同，通俗点说就是标记。比如<title>The Dormouse‘s story</title>或者<a href=“http://example.com/elsie”class=“sister”id=“link1”>Elsie</a>，title 和 a 标记及其里面的内容称为 Tag 对象。怎样从 html\_str 中抽取 Tag 呢？示例如下：

* 抽取 title：print soup.title
* 抽取 a：print soup.a
* 抽取 p：print soup.a

从例子中可以看到利用 soup 加标记名就可以获取这些标记的内容，比之前讲的正则表达式简单多了。不过利用这种方式，查找的是在所有内容中第一个符合要求的标记，如果要查询所有的标记，后面的内容进行讲解。

Tag 中有两个最重要的属性：name 和 attributes。每个 Tag 都有自己的名字，通过 .name 来获取。示例如下：

print soup.name

print soup.title.name

输出结果：

[**document**]

title

soup 对象本身比较特殊，它的 name 为[document]，对于其他内部标记，输出的值便为标记本身的名称。

Tag 不仅可以获取 name，还可以修改 name，改变之后将影响所有通过当前 Beautiful Soup 对象生成的 HTML 文档。示例如下：

soup.title.name = 'mytitle'

**print** soup.title

**print** soup.mytitle

输出结果：

None

<mytitle>The Dormouse's story</mytitle>

这里已经将 title 标记成功修改为 mytitle。

再说一下 Tag 中的属性，<p class=“title”><b>The Dormouse’s story</b></p>有一个“class”属性，值为“title”。Tag 的属性的操作方法与字典相同：

**print** soup.p['class']

**print** soup.p.get('class')

输出结果：

['title']

['title']

也可以直接“点”取属性，比如：.attrs，用于获取 Tag 中所有属性：

print soup.p.attrs

输出结果：

{'class': ['title']}

和 name 一样，我们可以对标记中的这些属性和内容等进行修改，示例如下：

soup.p['class']="myClass"

**print** soup.p

输出结果：

<p **class**="myClass"><b>The Dormouse's story</b></p>

2）NavigableString

我们已经得到了标记的内容，要想获取标记内部的文字怎么办呢？需要用到 .string。

示例如下：

print soup.p.string

print type(soup.p.string)

输出结果：

The Dormouse's story

<class 'bs4.element.NavigableString'>

Beautiful Soup 用 NavigableString 类来包装 Tag 中的字符串，一个 NavigableString 字符串与 Python 中的 Unicode 字符串相同，通过 unicode（） 方法可以直接将 NavigableString 对象转换成 Unicode 字符串：

unicode\_string = unicode(soup.p.**string**)

3）BeautifulSoup

BeautifulSoup 对象表示的是一个文档的全部内容。大部分时候，可以把它当作 Tag 对象，是一个特殊的 Tag，因为 BeautifulSoup 对象并不是真正的 HTML 或 XML 的标记，所以它没有 name 和 attribute 属性。但为了将 BeautifulSoup 对象标准化为 Tag 对象，实现接口的统一，我们依然可以分别获取它的 name 和 attribute 属性。示例如下：

**print** **type**(soup.name)

**print** soup.name

**print** soup.attrs

输出结果：

<**type** 'unicode'>

[document]

{}

4）Comment

Tag、NavigableString、BeautifulSoup 几乎覆盖了 HTML 和 XML 中的所有内容，但是还有一些特殊对象。容易让人担心的内容是文档的注释部分：

print soup.a.string

print type(soup.a.string)

输出结果：

Elsie

<**class** '**bs4**.**element**.**Comment**'>

a 标记里的内容实际上是注释，但是如果我们利用 .string 来输出它的内容，会发现它已经把注释符号去掉了。另外如果打印输出它的类型，会发现它是一个 Comment 类型。如果在我们不清楚这个标记 .string 的情况下，可能造成数据提取混乱。因此在提取字符串时，可以判断一下类型：

**if** **type**(soup.a.**string**)==bs4.element.Comment:

print soup.a.**string**

3.遍历文档树

BeautifulSoup 会将 HTML 转化为文档树进行搜索，既然是树形结构，节点的概念必不可少。

1）子节点

首先说一下直接子节点，Tag 中的 .contents 和 .children 是非常重要的。Tag 的 .content 属性可以将 Tag 子节点以列表的方式输出：

print soup.head.contents

输出结果：

[<title>The Dormouse's story</title>]

既然输出方式是列表，我们就可以获取列表的大小，并通过列表索引获取里面的值：

print len(soup.head.contents)

print soup.head.contents[0].string

输出结果：

1

The Dormouse's story

有一点需要注意：字符串没有 .contents 属性，因为字符串没有子节点。

.children 属性返回的是一个生成器，可以对 Tag 的子节点进行循环：

**for** child **in** soup.head.children:

**print**(child)

输出结果：

<title>The Dormouse's story</title>

.contents 和 .children 属性仅包含 Tag 的直接子节点。例如，<head>标记只有一个直接子节点<title>。但是<title>标记也包含一个子节点：字符串“The Dormouse’s story”，这种情况下字符串“The Dormouse’s story”也属于<head>标记的子孙节点。.descendants 属性可以对所有 tag 的子孙节点进行递归循环：

**for** child **in** soup.head.descendants:

**print**(child)

输出结果：

<title>The Dormouse's story</title>

The Dormouse's story

以上都是关于如何获取子节点，接下来说一下如何获取节点的内容，这就涉及 .string、.strings、stripped\_strings 三个属性。

.string 这个属性很有特点：如果一个标记里面没有标记了，那么 .string 就会返回标记里面的内容。如果标记里面只有唯一的一个标记了，那么 .string 也会返回最里面的内容。如果 tag 包含了多个子节点，tag 就无法确定，string 方法应该调用哪个子节点的内容，.string 的输出结果是 None。示例如下：

print soup.head.string

print soup.title.string

print soup.html.string

输出结果：

The Dormouse's story

The Dormouse's story

None

.strings 属性主要应用于 tag 中包含多个字符串的情况，可以进行循环遍历，示例如下：

**for** **string** **in** soup.strings:

print(repr(**string**))

输出结果：

u"The Dormouse's story"

u'\n'

u'\n'

u"The Dormouse's story"

u'\n'

u'Once upon a time there were three little sisters; and their names were\n'

u',\n'

u' and\n'

u'Tillie'

u';\nand they lived at the bottom of a well.'

u'\n'

u'...'

u'\n'

.stripped\_strings 属性可以去掉输出字符串中包含的空格或空行，示例如下：

**for** **string** **in** soup.stripped\_strings:

print(repr(**string**))

输出结果：

u"The Dormouse's story"

u"The Dormouse's story"

u'Once upon a time there were three little sisters; and their names were'

u','

u'and'

u'Tillie'

u';\nand they lived at the bottom of a well.'

u'...'

2）父节点

继续分析文档树，每个 Tag 或字符串都有父节点：被包含在某个 Tag 中。

通过 .parent 属性来获取某个元素的父节点。在 html\_str 中，<head>标记是<title>标记的父节点：

print soup.title

print soup.title.parent

输出结果：

<title>The Dormouse's story</title>

<head><title>The Dormouse's story</title></head>

通过元素的 .parents 属性可以递归得到元素的所有父辈节点，下面的例子使用了 .parents 方法遍历了<a>标记到根节点的所有节点：

**print** soup.a

**for** **parent** in soup.a.parents:

**if** **parent** is None:

**print**(**parent**)

**else**:

**print**(**parent**.name)

输出结果：

<a **class**="sister" href="http://example.com/elsie" id="link1">*<!-- Elsie -->*</a>

p

body

html

[**document**]

3）兄弟节点

从 soup.prettify（） 的输出结果中，我们可以看到<a>有很多兄弟节点。兄弟节点可以理解为和本节点处在同一级的节点，.next\_sibling 属性可以获取该节点的下一个兄弟节点，.previous\_sibling 则与之相反，如果节点不存在，则返回 None。示例如下：

print soup.p.next\_sibling

print soup.p.prev\_sibling

print soup.p.next\_sibling.next\_sibling

输出结果：

None

<p class="story">Once upon a time there were three little sisters; and their names were

<a class="sister" href="http://example.com/elsie" id="link1">*<!-- Elsie -->*</a>,

<a class="sister" href="http://example.com/lacie" id="link2">*<!-- Lacie -->*</a> and

<a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>;

and they lived at the bottom of a well.</p>

第一个输出结果为空白，因为空白或者换行也可以被视作一个节点，所以得到的结果可能是空白或者换行。

通过 .next\_siblings 和 .previous\_siblings 属性可以对当前节点的兄弟节点迭代输出：

**for** sibling **in** soup.a.next\_siblings:

**print**(repr(sibling))

输出结果：

u',\n'

<a class="sister" href="http://example.com/lacie" id="link2">*<!-- Lacie -->*</a>

u' and\n'

<a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>

u';\nand they lived at the bottom of a well.'

4）前后节点

前后节点需要使用 .next\_element、.previous\_element 这两个属性，与 .next\_sibling.previous\_sibling 不同，它并不是针对于兄弟节点，而是针对所有节点，不分层次，例如<head><title>The Dormouse‘s story</title></head>中的下一个节点就是 title：

print soup.head

print soup.head.next\_element

输出结果：

<head><title>The Dormouse's story</title></head>

<title>The Dormouse's story</title>

如果想遍历所有的前节点或者后节点，通过 .next\_elements 和 .previous\_elements 的迭代器就可以向前或向后访问文档的解析内容，就好像文档正在被解析一样：

**for** element **in** soup.a.next\_elements:

**print**(repr(element))

输出结果：

u' Elsie '

u',\n'

<a class="sister" href="http://example.com/lacie" id="link2">*<!-- Lacie -->*</a>

u' Lacie '

u' and\n'

<a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>

u'Tillie'

u';\nand they lived at the bottom of a well.'

u'\n'

<p class="story">...</p>

u'...'

u'\n'

以上就是遍历文档树的用法，接下来开始讲解比较核心的内容：搜索文档树。

4.搜索文档树

Beautiful Soup 定义了很多搜索方法，这里着重介绍 find\_all（） 方法，其他方法的参数和用法类似，请大家举一反三。

首先看一下 find\_all 方法，用于搜索当前 Tag 的所有 Tag 子节点，并判断是否符合过滤器的条件，函数原型如下：

find\_all( name , attrs , recursive , text , \*\*kwargs )

接下来分析函数中各个参数，不过需要打乱函数参数顺序，这样方便例子的讲解。

1）name 参数

name 参数可以查找所有名字为 name 的标记，字符串对象会被自动忽略掉。name 参数取值可以是字符串、正则表达式、列表、True 和方法。

最简单的过滤器是字符串。在搜索方法中传入一个字符串参数，Beautiful Soup 会查找与字符串完整匹配的内容，下面的例子用于查找文档中所有的<b>标记，返回值为列表：

**print** soup.find\_all('b')

输出结果：

[<b>The Dormouse's story</b>]

如果传入正则表达式作为参数，Beautiful Soup 会通过正则表达式的 match（） 来匹配内容。下面的例子中找出所有以 b 开头的标记，这表示<body>和<b>标记都应该被找到：

**import** re

**for** tag **in** soup.find\_all(re.compile("^b")):

**print**(tag.name)

输出结果：

body

b

如果传入列表参数，Beautiful Soup 会将与列表中任一元素匹配的内容返回。下面的代码找到文档中所有<a>标记和<b>标记：

**print** soup.find\_all(["a", "b"])

输出结果：

[<b>The Dormouse's story</b>, <a class="sister" href="http://example.com/elsie" id="link1">*<!-- Elsie -->*</a>, <a class="sister" href="http://example.com/lacie" id="link2">*<!-- Lacie -->*</a>, <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]

如果传入的参数是 True，True 可以匹配任何值，下面代码查找到所有的 tag，但是不会返回字符串节点：

**for** tag **in** soup.find\_all(**True**):

print(tag.name)

输出结果：

html

head

title

body

p

b

p

a

a

a

p

如果没有合适过滤器，那么还可以定义一个方法，方法只接受一个元素参数 Tag 节点，如果这个方法返回 True 表示当前元素匹配并且被找到，如果不是则返回 False。比如过滤包含 class 属性，也包含 id 属性的元素，程序如下：

**def** **hasClass\_Id**(tag):

**return** tag.has\_attr('class') **and** tag.has\_attr('id')

**print** soup.find\_all(hasClass\_Id)

输出结果：

[<a class="sister" href="http://example.com/elsie" id="link1">*<!-- Elsie -->*</a>, <a class="sister" href="http://example.com/lacie" id="link2">*<!-- Lacie -->*</a>, <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]

2）kwargs 参数

kwargs 参数在 Python 中表示为 keyword 参数。如果一个指定名字的参数不是搜索内置的参数名，搜索时会把该参数当作指定名字 Tag 的属性来搜索。搜索指定名字的属性时可以使用的参数值包括字符串、正则表达式、列表、True。

如果包含 id 参数，Beautiful Soup 会搜索每个 tag 的“id”属性。示例如下：

**print** soup.find\_all(id='link2')

输出结果：

[<a class="sister" href="http://example.com/lacie" id="link2">*<!-- Lacie -->*</a>]

如果传入 href 参数，Beautiful Soup 会搜索每个 Tag 的“href”属性。比如查找 href 属性中含有“elsie”的 tag：

**import** re

**print** soup.find\_all(href=re.compile("elsie"))

输出结果：

[<a class="sister" href="http://example.com/elsie" id="link1">*<!-- Elsie -->*</a>]

下面的代码在文档树中查找所有包含 id 属性的 Tag，无论 id 的值是什么：

**print** soup.find\_all(id=**True**)

输出结果：

[<a class="sister" href="http://example.com/elsie" id="link1">*<!-- Elsie -->*</a>, <a class="sister" href="http://example.com/lacie" id="link2">*<!-- Lacie -->*</a>, <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]

如果我们想用 class 过滤，但是 class 是 python 的关键字，需要在 class 后面加个下划线：

**print** soup.find\_all("a", class\_="sister")

输出结果：

[<a class="sister" href="http://example.com/elsie" id="link1">*<!-- Elsie -->*</a>, <a class="sister" href="http://example.com/lacie" id="link2">*<!-- Lacie -->*</a>, <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]

使用多个指定名字的参数可以同时过滤 tag 的多个属性：

**print** soup.find\_all(href=re.compile("elsie"), id='link1')

输出结果：

[<a class="sister" href="http://example.com/elsie" id="link1">*<!-- Elsie -->*</a>]

有些 tag 属性在搜索不能使用，比如 HTML5 中的 data-\* 属性：

data\_soup = BeautifulSoup('<div data-foo="value">foo!</div>')

data\_soup.find\_all(data-foo="value")

这样的代码在 Python 中是不合法的，但是可以通过 find\_all（） 方法的 attrs 参数定义一个字典参数来搜索包含特殊属性的 tag，示例代码如下：

data\_soup = BeautifulSoup('<div data-foo="value">foo!</div>')

data\_soup.find\_all(attrs={"data-foo": "value"})

输出结果：

[<div data-foo="value">foo!</div>]

3）text 参数

通过 text 参数可以搜索文档中的字符串内容。与 name 参数的可选值一样，text 参数接受字符串、正则表达式、列表、True。示例如下：

**print** soup.find\_all(text="Elsie")

**print** soup.find\_all(text=["Tillie", "Elsie", "Lacie"])

**print** soup.find\_all(text=re.compile("Dormouse"))

输出结果：

[u'Elsie']

[u'Elsie', u'Lacie', u'Tillie']

[u"The Dormouse's story", u"The Dormouse's story"]

虽然 text 参数用于搜索字符串，还可以与其他参数混合使用来过滤 tag。Beautiful Soup 会找到 .string 方法与 text 参数值相符的 tag。下面的代码用来搜索内容里面包含“Elsie”的<a>标记：

**print** soup.find\_all("a", text="Elsie")

输出结果：

[<a class="sister" href="http://example.com/elsie" id="link1">*<!--Elsie-->*</a>]

4）limit 参数

find\_all（） 方法返回全部的搜索结构，如果文档树很大那么搜索会很慢。如果我们不需要全部结果，可以使用 limit 参数限制返回结果的数量。效果与 SQL 中的 limit 关键字类似，当搜索到的结果数量达到 limit 的限制时，就停止搜索返回结果。下面的例子中，文档树中有3个 tag 符合搜索条件，但结果只返回了2个，因为我们限制了返回数量。

**print** soup.find\_all("a", **limit**=2)

输出结果：

[<a class="sister" href="http://example.com/elsie" id="link1">*<!--Elsie-->*</a>, <a **class**="sister" href="http://example.com/lacie" id="link2">*<!--Lacie-->*</a>]

5）recursive 参数

调用 tag 的 find\_all（） 方法时，Beautiful Soup 会检索当前 tag 的所有子孙节点，如果只想搜索 tag 的直接子节点，可以使用参数 recursive=False。示例如下：

**print** soup.find\_all("title")

**print** soup.find\_all("title", recursive=**False**)

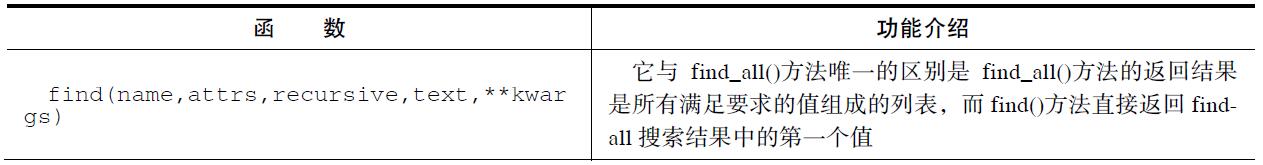
输出结果：

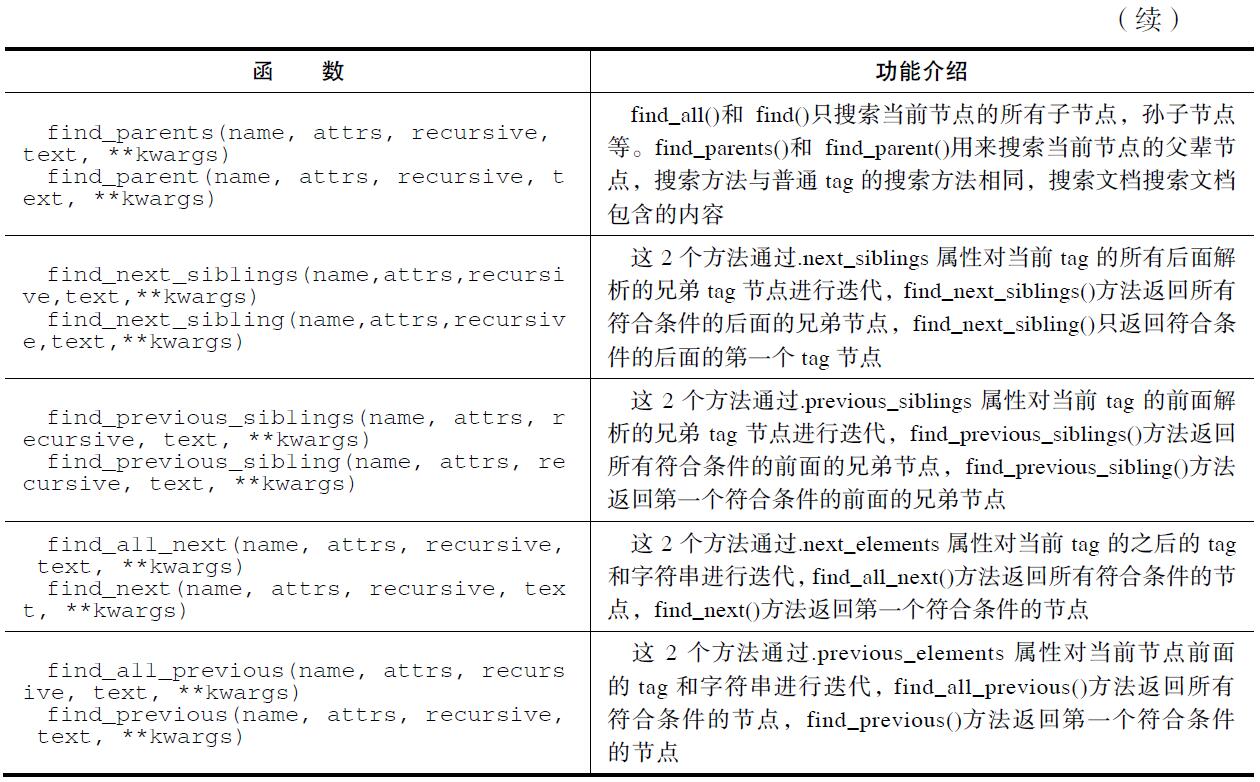
[<title>The Dormouse's story</title>]

[]

以上将 find\_all 函数的各个参数基本上讲解完毕，其他函数的使用方法和这个类似，表4-10列举了其他函数。

表4-10　搜索函数

[](http://popimage/?src=)

[](http://popimage/?src=)

5.CSS 选择器

在之前 Web 前端的章节中，我们讲到了 CSS 的语法，通过 CSS 也可以定位元素的位置。在写 CSS 时，标记名不加任何修饰，类名前加点“.”，id 名前加“#”，在这里我们也可以利用类似的方法来筛选元素，用到的方法是 soup.select（），返回类型是 list。

1）通过标记名称进行查找

通过标记名称可以直接查找、逐层查找，也可以找到某个标记下的直接子标记和兄弟节点标记。示例如下：

*# 直接查找 title 标记*

print soup.select("title")

*# 逐层查找 title 标记*

print soup.select("html head title")

*# 查找直接子节点*

*# 查找 head 下的 title 标记*

print soup.select("head > title")

*# 查找 p 下的 id="link1" 的标记*

print soup.select("p > # link1")

*# 查找兄弟节点*

*# 查找 id="link1" 之后 class=sisiter 的所有兄弟标记*

print soup.select("# link1 ~ .sister")

*# 查找紧跟着 id="link1" 之后 class=sisiter 的子标记*

print soup.select("# link1 + .sister")

输出结果：

[<title>The Dormouse's story</title>]

[<title>The Dormouse's story</title>]

[<title>The Dormouse's story</title>]

[<a class="sister" href="http://example.com/elsie" id="link1">*<!--Elsie-->*</a>]

[<a class="sister" href="http://example.com/lacie" id="link2">*<!--Lacie-->*</a>, <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]

[<a class="sister" href="http://example.com/lacie" id="link2">*<!--Lacie-->*</a>]

2）通过 CSS 的类名查找

示例如下：

print soup.select(".sister")

print soup.select("[class~=sister]")

输出结果：

[<a class="sister" href="http://example.com/elsie" id="link1">*<!--Elsie-->*</a>, <a class="sister" href="http://example.com/lacie" id="link2">*<!--Lacie-->*</a>, <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]

[<a class="sister" href="http://example.com/elsie" id="link1">*<!--Elsie-->*</a>, <a class="sister" href="http://example.com/lacie" id="link2">*<!--Lacie-->*</a>, <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]

3）通过 tag 的 id 查找

示例如下：

print soup.select("# link1")

print soup.select("a# link2")

输出结果：

[<a class="sister" href="http://example.com/elsie" id="link1">*<!--Elsie-->*</a>]

[<a class="sister" href="http://example.com/lacie" id="link2">*<!--Lacie-->*</a>]

4）通过是否存在某个属性来查找

示例如下：

print soup.select('a[href]')

输出结果：

[<a class="sister" href="http://example.com/elsie" id="link1">*<!--Elsie-->*</a>, <a class="sister" href="http://example.com/lacie" id="link2">*<!--Lacie-->*</a>, <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]

5）通过属性值来查找

示例如下：

print soup.select('a[href="http://example.com/elsie"]')

print soup.select('a[href^="http://example.com/"]')

print soup.select('a[href$="tillie"]')

print soup.select('a[href\*=".com/el"]')

输出结果：

[<a class="sister" href="http://example.com/elsie" id="link1">*<!--Elsie-->*</a>]

[<a class="sister" href="http://example.com/elsie" id="link1">*<!--Elsie-->*</a>, <a class="sister" href="http://example.com/lacie" id="link2">*<!--Lacie-->*</a>, <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]

[<a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]

[<a class="sister" href="http://example.com/elsie" id="link1">*<!--Elsie-->*</a>]

以上就是 CSS 选择器的查找方式，如果大家对 CSS 选择器的写法不是很熟悉，可以搜索一下 W3CSchool 的 CSS 选择器参考手册进行学习。除此之外，还可以使用 Firebug 中的 FirePath 功能自动获取网页元素的 CSS 选择器表达式，如图4-25所示。

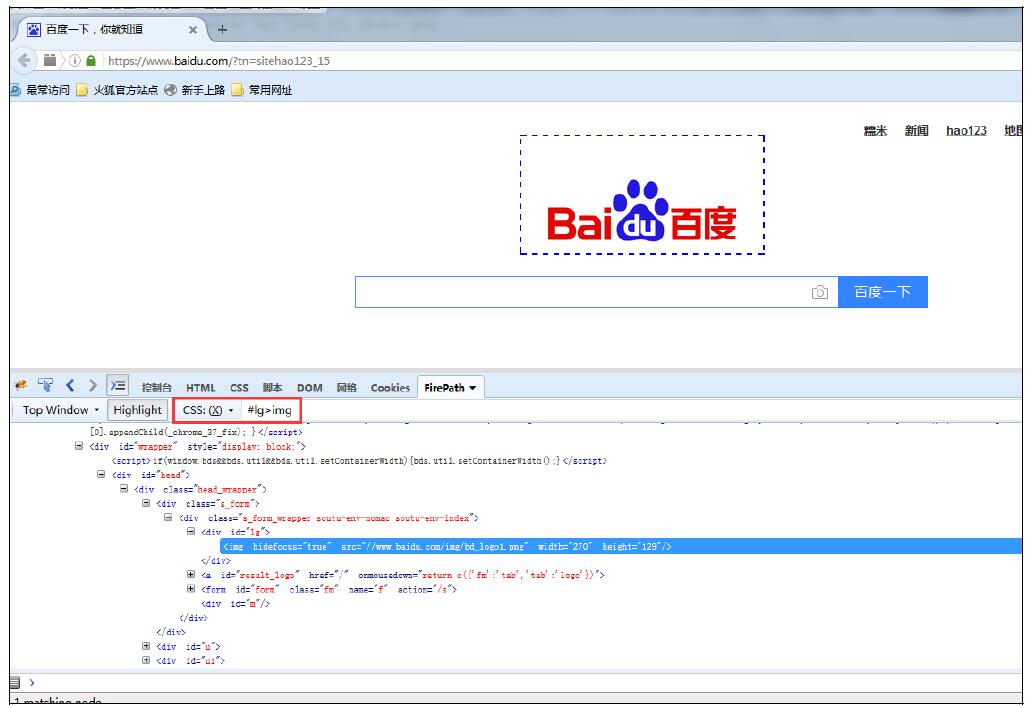
[](http://popimage/?src=)

图4-25　FirePath CSS 选择器

## 4.3.3　lxml 的 XPath 解析

BeautifulSoup 可以将 lxml 作为默认的解析器使用，同样 lxml 可以单独使用。下面比较一下这两者之间的优缺点：

* BeautifulSoup 和 lxml 的原理不一样，BeautifulSoup 是基于 DOM 的，会载入整个文档，解析整个 DOM 树，因此时间和内存开销都会大很多。而 lxml 是使用 XPath 技术查询和处理 HTML/XML 文档的库，只会局部遍历，所以速度会快一些。幸好现在 BeautifulSoup 可以使用 lxml 作为默认解析库。
* BeautifulSoup 用起来比较简单，API 非常人性化，支持 CSS 选择器，适合新手。lxml 的 XPath 写起来麻烦，开发效率不如 BeautifulSoup，当然这也是因人而异，如果你能熟练使用 XPath，那么使用 lxml 是更好的选择，况且现在又有了 FirePath 这样的自动生成 XPath 表达式的利器。

第2章已经讲过了 XPath 的用法，所以现在直接介绍如何使用 lxml 库来解析网页。示例如下：

from lxml import etree

html\_str = """

<html><head><title>The Dormouse's story</title></head>

<body>

<p class="title"><b>The Dormouse's story</b></p>

<p class="story">Once upon a time there were three little sisters; and their names were

<a href="http://example.com/elsie" class="sister" id="link1">Elsie</a>,

<a href="http://example.com/lacie" class="sister" id="link2">Lacie</a> and

<a href="http://example.com/tillie" class="sister" id="link3">Tillie</a>;

and they lived at the bottom of a well.</p>

<p class="story">...</p>

"""

html = etree.HTML(html\_str)

result = etree.tostring(html)

print(result)

输出结果：

<html><head><title>The Dormouse's story</title></head>

<body>

<p class="title"><b>The Dormouse's story</b></p>

<p class="story">Once upon a time there were three little sisters; and their names were

<a href="http://example.com/elsie" class="sister" id="link1">*<!--Elsie-->*</a>,

<a href="http://example.com/lacie" class="sister" id="link2">*<!--Lacie-->*</a> and

<a href="http://example.com/tillie" class="sister" id="link3">Tillie</a>;

and they lived at the bottom of a well.</p>

<p class="story">...</p>

</body></html>

大家看到 html\_str 最后是没有</html>和</body>标签的，没有进行闭合，但是通过输出结果我们可以看到 lxml 的一个非常实用的功能就是自动修正 html 代码。

除了读取字符串之外，lxml 还可以直接读取 html 文件。假如将 html\_str 存储为 index.html 文件，利用 parse 方法进行解析，示例如下：

**from** lxml **import** etree

html = etree.parse('index.html')

result = etree.tostring(html, pretty\_print=**True**)

print(result)

接下来使用 XPath 语法抽取出其中所有的 URL，示例如下：

html = etree.HTML(html\_str)

urls = html.xpath(".// \*[@class='sister']/@href")

print urls

输出结果：

['http://example.com/elsie', 'http://example.com/lacie', 'http://example.com/tillie']

使用 lxml 的关键是构造 XPath 表达式，如果大家对 XPath 不熟悉，可以复习一下第2章中 XPath 内容。