
beautifulsoup Documentation

发布 *4.4.0*

delong

9 月 15, 2016

1	寻求帮助	3
2	快速开始	5
3	安装 Beautiful Soup	9
3.1	安装完成后的问题	9
3.2	安装解析器	10
4	如何使用	13
5	对象的种类	15
5.1	Tag	15
5.2	可以遍历的字符串	17
5.3	BeautifulSoup	18
5.4	注释及特殊字符串	18
6	遍历文档树	21
6.1	子节点	21
6.2	父节点	25
6.3	兄弟节点	26
6.4	回退和前进	28
7	搜索文档树	31
7.1	过滤器	31
7.2	find_all()	34
7.3	像调用 find_all() 一样调用 tag	39
7.4	find()	39
7.5	find_parents() 和 find_parent()	40
7.6	find_next_siblings() 和 find_next_sibling()	40
7.7	find_previous_siblings() 和 find_previous_sibling()	41
7.8	find_all_next() 和 find_next()	41

7.9	<code>find_all_previous()</code> 和 <code>find_previous()</code>	42
7.10	CSS 选择器	43
8	修改文档树	47
8.1	修改 tag 的名称和属性	47
8.2	修改.string	47
8.3	<code>append()</code>	48
8.4	<code>NavigableString()</code> 和 <code>new_tag()</code>	48
8.5	<code>insert()</code>	49
8.6	<code>insert_before()</code> 和 <code>insert_after()</code>	49
8.7	<code>clear()</code>	50
8.8	<code>extract()</code>	50
8.9	<code>decompose()</code>	51
8.10	<code>replace_with()</code>	51
8.11	<code>wrap()</code>	52
8.12	<code>unwrap()</code>	52
9	输出	53
9.1	格式化输出	53
9.2	压缩输出	54
9.3	输出格式	54
9.4	<code>get_text()</code>	54
10	指定文档解析器	57
10.1	解析器之间的区别	57
11	编码	59
11.1	输出编码	60
11.2	Unicode, Dammit! (乱码, 靠!)	62
12	比较对象是否相同	65
13	复制 BeautifulSoup 对象	67
14	解析部分文档	69
14.1	<code>SoupStrainer</code>	69
15	常见问题	71
15.1	代码诊断	71
15.2	文档解析错误	71
15.3	版本错误	72
15.4	解析成 XML	72
15.5	解析器的错误	72
15.6	杂项错误	73
15.7	如何提高效率	73

16 Beautiful Soup 3	75
16.1 迁移到 BS4	75
17 附录	79

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. Beautiful Soup 会帮你节省数小时甚至数天的工作时间.

这篇文档介绍了 BeautifulSoup4 中所有主要特性, 并且有小例子. 让我来向你展示它适合做什么, 如何工作, 怎样使用, 如何达到你想要的效果, 和处理异常情况.

文档中出现的例子在 Python2.7 和 Python3.2 中的执行结果相同

你可能在寻找 Beautiful Soup3 的文档,Beautiful Soup 3 目前已经停止开发, 我们推荐在现在的项目中使用 Beautiful Soup 4, 移植到 BS4

这篇帮助文档已经被翻译成了其它语言:

- 这篇文档当然还有中文版.
- このページは日本語で利用できます (外部リンク)
- . ()

寻求帮助

如果你有关于 BeautifulSoup 的问题, 可以发送邮件到 [讨论组](#) . 如果你的问题包含了一段需要转换的 HTML 代码, 那么确保你提的问题描述中附带这段 HTML 文档的[代码诊断](#)¹

¹ BeautifulSoup 的 google 讨论组不是很活跃, 可能是因为库已经比较完善了吧, 但是作者还是会很热心的尽量帮你解决问题的.

快速开始

下面的一段 HTML 代码将作为例子被多次用到. 这是 爱丽丝梦游仙境的的一段内容 (以后内容中简称为 爱丽丝的文档):

```
html_doc = """
<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>
"""
```

使用 BeautifulSoup 解析这段代码, 能够得到一个 BeautifulSoup 的对象, 并能按照标准的缩进格式的结构输出:

```
from bs4 import BeautifulSoup
soup = BeautifulSoup(html_doc, 'html.parser')

print(soup.prettify())
# <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="link2">
#     Tillie
#   </a>
#   ; and they lived at the bottom of a well.
# </p>
# <p class="story">
#   ...
# </p>
# </body>
# </html>
```

几个简单的浏览结构化数据的方法:

```
soup.title
# <title>The Dormouse's story</title>

soup.title.name
# u'title'

soup.title.string
# u'The Dormouse's story'

soup.title.parent.name
# u'head'

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

soup.p['class']
# u'title'

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

```
soup.find_all('a')
# [Elsie</a>,
#  Lacie</a>,
#  Tillie</a>]

soup.find(id="link3")
# Tillie</a>
```

从文档中找到所有 <a> 标签的链接:

```
for link in soup.find_all('a'):
    print(link.get('href'))
# http://example.com/elsie
# http://example.com/lacie
# http://example.com/tillie
```

从文档中获取所有文字内容:

```
print(soup.get_text())
# The Dormouse's story
#
# The Dormouse's story
#
# Once upon a time there were three little sisters; and their names were
# Elsie,
# Lacie and
# Tillie;
# and they lived at the bottom of a well.
#
# ...
```

这是你想要的吗? 别着急, 还有更好用的

安装 Beautiful Soup

如果你用的是新版的 Debian 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, 所以如果你无法使用系统包管理安装, 那么也可以通过 `easy_install` 或 `pip` 来安装. 包的名字是 `beautifulsoup4`, 这个包兼容 Python2 和 Python3.

```
$ easy_install beautifulsoup4
```

```
$ pip install beautifulsoup4
```

(在 PyPi 中还有一个名字是 `BeautifulSoup` 的包, 但那可能不是你想要的, 那是 `Beautiful Soup3` 的发布版本, 因为很多项目还在使用 BS3, 所以 `BeautifulSoup` 包依然有效. 但是如果你在编写新项目, 那么你应该安装的 `beautifulsoup4`)

如果你没有安装 `easy_install` 或 `pip`, 那你也可以 [下载 BS4 的源码](#), 然后通过 `setup.py` 来安装.

```
$ Python setup.py install
```

如果上述安装方法都行不通, Beautiful Soup 的发布协议允许你将 BS4 的代码打包在你的项目中, 这样无须安装即可使用.

作者在 Python2.7 和 Python3.2 的版本下开发 Beautiful Soup, 理论上 Beautiful Soup 应该在所有当前的 Python 版本中正常工作

3.1 安装完成后的问题

Beautiful Soup 发布时打包成 Python2 版本的代码, 在 Python3 环境下安装时, 会自动转换成 Python3 的代码, 如果没有一个安装的过程, 那么代码就不会被转换.

如果代码抛出了 `ImportError` 的异常: “No module named HTMLParser”, 这是因为你在 Python3 版本中执行 Python2 版本的代码.

如果代码抛出了 `ImportError` 的异常: “No module named `html.parser`”, 这是因为你在 Python2 版本中执行 Python3 版本的代码.

如果遇到上述 2 种情况, 最好的解决方法是重新安装 BeautifulSoup4.

如果在 `ROOT_TAG_NAME = u'[document]'` 代码处遇到 `SyntaxError` “Invalid syntax” 错误, 需要将把 BS4 的 Python 代码版本从 Python2 转换到 Python3. 可以重新安装 BS4:

```
$ Python3 setup.py install
```

或在 bs4 的目录中执行 Python 代码版本转换脚本

```
$ 2to3-3.2 -w bs4
```

3.2 安装解析器

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](#). 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
```

下表列出了主要的解析器, 以及它们的优缺点:

解析器	使用方法	优势	劣势
Python 标准库	<code>BeautifulSoup(markup,"html.parser")</code>	<ul style="list-style-type: none">• Python 的内置标准库• 执行速度适中• 文档容错能力强	<ul style="list-style-type: none">• Python 2.7.3 or 3.2.2) 前的版本中文档容错能力差
lxml HTML 解析器	<code>BeautifulSoup(markup,"lxml")</code>	<ul style="list-style-type: none">• 速度快• 文档容错能力强	<ul style="list-style-type: none">• 需要安装 C 语言库
lxml XML 解析器	<code>BeautifulSoup(markup,["lxml-xml"])</code> <code>BeautifulSoup(markup,"xml")</code>	<ul style="list-style-type: none">• 速度快• 唯一支持 XML 的解析器	<ul style="list-style-type: none">• 需要安装 C 语言库
html5lib	<code>BeautifulSoup(markup,"html5lib")</code>	<ul style="list-style-type: none">• 最好的容错性• 以浏览器的方式解析文档• 生成 HTML5 格式的文档	<ul style="list-style-type: none">• 速度慢• 不依赖外部扩展

推荐使用 lxml 作为解析器, 因为效率更高. 在 Python2.7.3 之前的版本和 Python3 中 3.2.2 之前的版本, 必须安装 lxml 或 html5lib, 因为那些 Python 版本的标准库中内置的 HTML 解析方法不够稳定.

提示: 如果一段 HTML 或 XML 文档格式不正确的话, 那么在不同的解析器中返回的结果可能是不一样的, 查看[解析器之间的区别](#)了解更多细节

如何使用

将一段文档传入 BeautifulSoup 的构造方法, 就能得到一个文档的对象, 可以传入一段字符串或一个文件句柄.

```
from bs4 import BeautifulSoup

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

soup = BeautifulSoup("<html>data</html>")
```

首先, 文档被转换成 Unicode, 并且 HTML 的实例都被转换成 Unicode 编码

```
BeautifulSoup("Sacr&eacute; bleu!")
<html><head></head><body>Sacr   bleu!</body></html>
```

然后, BeautifulSoup 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 BeautifulSoup 会选择指定的解析器来解析文档.(参考[解析成 XML](#)).

对象的种类

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 .

5.1 Tag

Tag 对象与 XML 或 HTML 原生文档中的 tag 相同:

```
soup = BeautifulSoup('<b class="boldest">Extremely bold</b>')
tag = soup.b
type(tag)
# <class 'bs4.element.Tag'>
```

Tag 有很多方法和属性, 在[遍历文档树](#)和[搜索文档树](#)中有详细解释. 现在介绍一下 tag 中最重要的属性: name 和 attributes

5.1.1 Name

每个 tag 都有自己的名字, 通过 .name 来获取:

```
tag.name
# u'b'
```

如果改变了 tag 的 name, 那将影响所有通过当前 BeautifulSoup 对象生成的 HTML 文档:

```
tag.name = "blockquote"
tag
# <blockquote class="boldest">Extremely bold</blockquote>
```

5.1.2 Attributes

一个 tag 可能有很多个属性. tag `<b class="boldest">` 有一个 “class” 的属性, 值为 “boldest”. tag 的属性的操作方法与字典相同:

```
tag['class']
# u'boldest'
```

也可以直接”点”取属性, 比如: `.attrs`:

```
tag.attrs
# {u'class': u'boldest'}
```

tag 的属性可以被添加, 删除或修改. 再说一次, tag 的属性操作方法与字典一样

```
tag['class'] = 'verybold'
tag['id'] = 1
tag
# <blockquote class="verybold" id="1">Extremely bold</blockquote>

del tag['class']
del tag['id']
tag
# <blockquote>Extremely bold</blockquote>

tag['class']
# KeyError: 'class'
print(tag.get('class'))
# None
```

多值属性

HTML 4 定义了一系列可以包含多个值的属性. 在 HTML5 中移除了一些, 却增加更多. 最常见的多值的属性是 class (一个 tag 可以有多个 CSS 的 class). 还有一些属性 rel, rev, accept-charset, headers, accesskey. 在 Beautiful Soup 中多值属性的返回类型是 list:

```
css_soup = BeautifulSoup('<p class="body strikeout"></p>')
css_soup.p['class']
# ["body", "strikeout"]

css_soup = BeautifulSoup('<p class="body"></p>')
css_soup.p['class']
# ["body"]
```

如果某个属性看起来好像有多个值,但在任何版本的 HTML 定义中都没有被定义为多值属性,那么 BeautifulSoup 会将这个属性作为字符串返回

```
id_soup = BeautifulSoup('<p id="my id"></p>')
id_soup.p['id']
# 'my id'
```

将 tag 转换成字符串时,多值属性会合并为一个值

```
rel_soup = BeautifulSoup('<p>Back to the <a rel="index">homepage</a></p>')
rel_soup.a['rel']
# ['index']
rel_soup.a['rel'] = ['index', 'contents']
print(rel_soup.p)
# <p>Back to the <a rel="index contents">homepage</a></p>
```

如果转换的文档是 XML 格式,那么 tag 中不包含多值属性

```
xml_soup = BeautifulSoup('<p class="body strikeout"></p>', 'xml')
xml_soup.p['class']
# u'body strikeout'
```

5.2 可以遍历的字符串

字符串常被包含在 tag 内.Beautiful Soup 用 NavigableString 类来包装 tag 中的字符串:

```
tag.string
# u'Extremely bold'
type(tag.string)
# <class 'bs4.element.NavigableString'>
```

一个 NavigableString 字符串与 Python 中的 Unicode 字符串相同,并且还支持包含在遍历文档树和搜索文档树中的一些特性.通过 unicode() 方法可以直接将 NavigableString 对象转换成 Unicode 字符串:

```
unicode_string = unicode(tag.string)
unicode_string
# u'Extremely bold'
type(unicode_string)
# <type 'unicode'>
```

tag 中包含的字符串不能编辑,但是可以被替换成其它的字符串,用 `replace_with()` 方法:

```
tag.string.replace_with("No longer bold")
tag
# <blockquote>No longer bold</blockquote>
```

`NavigableString` 对象支持遍历文档树 和 搜索文档树 中定义的大部分属性, 并非全部. 尤其是, 一个字符串不能包含其它内容 (tag 能够包含字符串或是其它 tag), 字符串不支持 `.contents` 或 `.string` 属性或 `find()` 方法.

如果想在 Beautiful Soup 之外使用 `NavigableString` 对象, 需要调用 `unicode()` 方法, 将该对象转换成普通的 Unicode 字符串, 否则就算 Beautiful Soup 已方法已经执行结束, 该对象的输出也会带有对象的引用地址. 这样会浪费内存.

5.3 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, 它支持遍历文档树 和 搜索文档树 中描述的大部分的方法.

因为 `BeautifulSoup` 对象并不是真正的 HTML 或 XML 的 tag, 所以它没有 `name` 和 `attribute` 属性. 但有时查看它的 `.name` 属性是很方便的, 所以 `BeautifulSoup` 对象包含了一个值为 "[document]" 的特殊属性 `.name`

```
soup.name
# u'[document]'
```

5.4 注释及特殊字符串

`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:

```
markup = "<b><!--Hey, buddy. Want to buy a used parser?--></b>"
soup = BeautifulSoup(markup)
comment = soup.b.string
type(comment)
# <class 'bs4.element.Comment'>
```

`Comment` 对象是一个特殊类型的 `NavigableString` 对象:

```
comment
# u'Hey, buddy. Want to buy a used parser'
```

但是当它出现在 HTML 文档中时, `Comment` 对象会使用特殊的格式输出:


```
print(soup.b.prettify())
# <b>
#   <!--Hey, buddy. Want to buy a used parser?-->
# </b>
```

Beautiful Soup 中定义的其它类型都可能会出现在 XML 的文档中: `CData` , `ProcessingInstruction` , `Declaration` , `Doctype` . 与 `Comment` 对象类似, 这些类都是 `NavigableString` 的子类, 只是添加了一些额外的方法的字符串独享. 下面是用 `CDATA` 来替代注释的例子:

```
from bs4 import CData
cdata = CData("A CDATA block")
comment.replace_with(cdata)

print(soup.b.prettify())
# <b>
#   <![CDATA[A CDATA block]]>
# </b>
```

遍历文档树

还拿“爱丽丝梦游仙境”的文档来做例子:

```
html_doc = """
<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>
"""

from bs4 import BeautifulSoup
soup = BeautifulSoup(html_doc, 'html.parser')
```

通过这段例子来演示怎样从文档的一段内容找到另一段内容

6.1 子节点

一个 Tag 可能包含多个字符串或其它的 Tag, 这些都是这个 Tag 的子节点. BeautifulSoup 提供了许多操作和遍历子节点的属性.

注意: BeautifulSoup 中字符串节点不支持这些属性, 因为字符串没有子节点

6.1.1 tag 的名字

操作文档树最简单的方法就是告诉它你想获取的 tag 的 name. 如果想获取 <head> 标签, 只要用 `soup.head` :

```
soup.head
# <head><title>The Dormouse's story</title></head>

soup.title
# <title>The Dormouse's story</title>
```

这是个获取 tag 的小窍门, 可以在文档树的 tag 中多次调用这个方法. 下面的代码可以获取 <body> 标签中的第一个 标签:

```
soup.body.b
# <b>The Dormouse's story</b>
```

通过点取属性的方式只能获得当前名字的第一个 tag:

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

如果想要得到所有的 <a> 标签, 或是通过名字得到比一个 tag 更多的内容的时候, 就需要用到 *Searching the tree* 中描述的方法, 比如: `find_all()`

```
soup.find_all('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>]
```

6.1.2 .contents 和 .children

tag 的 `.contents` 属性可以将 tag 的子节点以列表的方式输出:

```
head_tag = soup.head
head_tag
# <head><title>The Dormouse's story</title></head>

head_tag.contents
[<title>The Dormouse's story</title>]

title_tag = head_tag.contents[0]
title_tag
# <title>The Dormouse's story</title>
title_tag.contents
# [u'The Dormouse's story']
```

BeautifulSoup 对象本身一定会包含子节点, 也就是说 `<html>` 标签也是 BeautifulSoup 对象的子节点:

```
len(soup.contents)
# 1
soup.contents[0].name
# u'html'
```

字符串没有 `.contents` 属性, 因为字符串没有子节点:

```
text = title_tag.contents[0]
text.contents
# AttributeError: 'NavigableString' object has no attribute 'contents'
```

通过 tag 的 `.children` 生成器, 可以对 tag 的子节点进行循环:

```
for child in title_tag.children:
    print(child)
# The Dormouse's story
```

6.1.3 .descendants

`.contents` 和 `.children` 属性仅包含 tag 的直接子节点. 例如, `<head>` 标签只有一个直接子节点 `<title>`

```
head_tag.contents
# [<title>The Dormouse's story</title>]
```

但是 `<title>` 标签也包含一个子节点: 字符串 “The Dormouse’s story”, 这种情况下字符串 “The Dormouse’s story” 也属于 `<head>` 标签的子孙节点. `.descendants` 属性可以对所有 tag 的子孙节点进行递归循环⁵:

```
for child in head_tag.descendants:
    print(child)
# <title>The Dormouse's story</title>
# The Dormouse's story
```

上面的例子中, `<head>` 标签只有一个子节点, 但是有 2 个子孙节点: `<head>` 节点和 `<head>` 的子节点, BeautifulSoup 有一个直接子节点 (`<html>` 节点), 却有很多子孙节点:

```
len(list(soup.children))
# 1
```

⁵ 采用先序遍历方式

```
len(list(soup.descendants))
# 25
```

6.1.4 .string

如果 tag 只有一个 NavigableString 类型子节点, 那么这个 tag 可以使用 .string 得到子节点:

```
title_tag.string
# u'The Dormouse's story'
```

如果一个 tag 仅有一个子节点, 那么这个 tag 也可以使用 .string 方法, 输出结果与当前唯一子节点的 .string 结果相同:

```
head_tag.contents
# [<title>The Dormouse's story</title>]

head_tag.string
# u'The Dormouse's story'
```

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 :

```
print(soup.html.string)
# None
```

6.1.5 .strings 和 stripped_strings

如果 tag 中包含多个字符串², 可以使用 .strings 来循环获取:

```
for string in soup.strings:
    print(repr(string))
    # u'The Dormouse's story'
    # u'\n\n'
    # u'The Dormouse's story'
    # u'\n\n'
    # u'Once upon a time there were three little sisters; and their names were\n'
    # u'Elsie'
    # u',\n'
    # u'Lacie'
    # u' and\n'
    # u'Tillie'
```

² 文档被解析成树形结构, 所以下一步解析过程应该是当前节点的子节点

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

输出的字符串中可能包含了很多空格或空行, 使用 `.stripped_strings` 可以去除多余空白内容:

```
for string in soup.stripped_strings:
    print(repr(string))
# 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'Elsie'
# u','
# u'Lacie'
# u'and'
# u'Tillie'
# u';\nand they lived at the bottom of a well.'
# u'...'

```

全部是空格的行会被忽略掉, 段首和段末的空白会被删除

6.2 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

6.2.1 .parent

通过 `.parent` 属性来获取某个元素的父节点. 在例子“爱丽丝”的文档中, `<head>` 标签是 `<title>` 标签的父节点:

```
title_tag = soup.title
title_tag
# <title>The Dormouse's story</title>
title_tag.parent
# <head><title>The Dormouse's story</title></head>
```

文档 `title` 的字符串也有父节点: `<title>` 标签

```
title_tag.string.parent
# <title>The Dormouse's story</title>
```

文档的顶层节点比如 `<html>` 的父节点是 BeautifulSoup 对象:

```
html_tag = soup.html
type(html_tag.parent)
# <class 'bs4.BeautifulSoup'>
```

BeautifulSoup 对象的 `.parent` 是 `None`:

```
print(soup.parent)
# None
```

6.2.2 .parents

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>` 标签到根节点的所有节点.

```
link = soup.a
link
# <a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>
for parent in link.parents:
    if parent is None:
        print(parent)
    else:
        print(parent.name)
# p
# body
# html
# [document]
# None
```

6.3 兄弟节点

看一段简单的例子:

```
sibling_soup = BeautifulSoup("<a><b>text1</b><c>text2</c></b></a>")
print(sibling_soup.prettify())
# <html>
# <body>
# <a>
# <b>
# text1
# </b>
# <c>
```



```
#      text2
#      </c>
#      </a>
# </body>
# </html>
```

因为 `` 标签和 `<c>` 标签是同一层: 他们是同一个元素的子节点, 所以 `` 和 `<c>` 可以被称为兄弟节点. 一段文档以标准格式输出时, 兄弟节点有相同的缩进级别. 在代码中也可以使用这种关系.

6.3.1 .next_sibling 和 .previous_sibling

在文档树中, 使用 `.next_sibling` 和 `.previous_sibling` 属性来查询兄弟节点:

```
sibling_soup.b.next_sibling
# <c>text2</c>

sibling_soup.c.previous_sibling
# <b>text1</b>
```

`` 标签有 `.next_sibling` 属性, 但是没有 `.previous_sibling` 属性, 因为 `` 标签在同级节点中是第一个. 同理, `<c>` 标签有 `.previous_sibling` 属性, 却没有 `.next_sibling` 属性:

```
print(sibling_soup.b.previous_sibling)
# None
print(sibling_soup.c.next_sibling)
# None
```

例子中的字符串 “text1” 和 “text2” 不是兄弟节点, 因为它们的父节点不同:

```
sibling_soup.b.string
# u'text1'

print(sibling_soup.b.string.next_sibling)
# None
```

实际文档中的 tag 的 `.next_sibling` 和 `.previous_sibling` 属性通常是字符串或空白. 看看 “爱丽丝” 文档:

```
<a href="http://example.com/elsie" class="sister" id="link1">Elsie</a>
<a href="http://example.com/lacie" class="sister" id="link2">Lacie</a>
<a href="http://example.com/tillie" class="sister" id="link3">Tillie</a>
```

如果以为第一个 `<a>` 标签的 `.next_sibling` 结果是第二个 `<a>` 标签, 那就错了, 真实结果是第一个 `<a>` 标签和第二个 `<a>` 标签之间的顿号和换行符:

```
link = soup.a
link
# <a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>

link.next_sibling
# u',\n'
```

第二个 <a> 标签是顿号的 .next_sibling 属性:

```
link.next_sibling.next_sibling
# <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>
```

6.3.2 .next_siblings 和 .previous_siblings

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))
    # 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'; and they lived at the bottom of a well.'
    # None

for sibling in soup.find(id="link3").previous_siblings:
    print(repr(sibling))
    # ' and\n'
    # <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>
    # u',\n'
    # <a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>
    # u'Once upon a time there were three little sisters; and their names were\n'
    # None
```

6.4 回退和前进

看一下“爱丽丝”文档:

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

HTML 解析器把这段字符串转换成一连串的事件: “打开 <html> 标签”, “打开一个 <head> 标签”, “打开一个 <title> 标签”, “添加一段字符串”, “关闭 <title> 标签”, “打开 <p> 标签”, 等等. Beautiful Soup 提供了重现解析器初始化过程的方法.

6.4.1 .next_element 和 .previous_element

.next_element 属性指向解析过程中下一个被解析的对象 (字符串或 tag), 结果可能与 .next_sibling 相同, 但通常是不一样的.

这是“爱丽丝”文档中最后一个 <a> 标签, 它的 .next_sibling 结果是一个字符串, 因为当前的解析过程² 因为当前的解析过程因为遇到了 <a> 标签而中断了:

```
last_a_tag = soup.find("a", id="link3")
last_a_tag
# <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>

last_a_tag.next_sibling
# '; and they lived at the bottom of a well.'
```

但这个 <a> 标签的 .next_element 属性结果是在 <a> 标签被解析之后的解析内容, 不是 <a> 标签后的句子部分, 应该是字符串“Tillie”:

```
last_a_tag.next_element
# u'Tillie'
```

这是因为在原始文档中, 字符串“Tillie”在分号前出现, 解析器先进入 <a> 标签, 然后是字符串“Tillie”, 然后关闭 标签, 然后是分号和剩余部分. 分号与 <a> 标签在同一层级, 但是字符串“Tillie”会被先解析.

.previous_element 属性刚好与 .next_element 相反, 它指向当前被解析的对象的前一个解析对象:

```
last_a_tag.previous_element
# u' and\n'
last_a_tag.previous_element.next_element
# <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>
```

6.4.2 .next_elements 和 .previous_elements

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 last_a_tag.next_elements:
    print(repr(element))
# u'Tillie'
```

```
# u';\nand they lived at the bottom of a well.'  
# u'\n\n'  
# <p class="story">...</p>  
# u'...'  
# u'\n'  
# None
```

搜索文档树

Beautiful Soup 定义了很多搜索方法, 这里着重介绍 2 个: `find()` 和 `find_all()`。其它方法的参数和用法类似, 请读者举一反三。

再以“爱丽丝”文档作为例子:

```
html_doc = """
<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>
"""

from bs4 import BeautifulSoup
soup = BeautifulSoup(html_doc, 'html.parser')
```

使用 `find_all()` 类似的方法可以查找到想要查找的文档内容

7.1 过滤器

介绍 `find_all()` 方法前, 先介绍一下过滤器的类型³, 这些过滤器贯穿整个搜索的 API。过滤器可以被用在 tag 的 name 中, 节点的属性中, 字符串中或他们的混合中。

³ 过滤器只能作为搜索文档的参数, 或者说应该叫参数类型更为贴切, 原文中用了 `filter` 因此翻译为过滤器

7.1.1 字符串

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 `` 标签:

```
soup.find_all('b')
# [The Dormouse's story]
```

如果传入字节码参数,Beautiful Soup 会当作 UTF-8 编码, 可以传入一段 Unicode 编码来避免 BeautifulSoup 解析编码出错

7.1.2 正则表达式

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. 下面例子中找出所有以 `b` 开头的标签, 这表示 `<body>` 和 `` 标签都应该被找到:

```
import re
for tag in soup.find_all(re.compile("^b")):
    print(tag.name)
# body
# b
```

下面代码找出所有名字中包含“t”的标签:

```
for tag in soup.find_all(re.compile("t")):
    print(tag.name)
# html
# title
```

7.1.3 列表

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. 下面代码找到文档中所有 `<a>` 标签和 `` 标签:

```
soup.find_all(["a", "b"])
# [The Dormouse's story,
# <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>]
```

7.1.4 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)
# html
# head
# title
# body
# p
# b
# p
# a
# a
# a
# p
```

7.1.5 方法

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⁴, 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

下面方法校验了当前元素, 如果包含 class 属性却不包含 id 属性, 那么将返回 True:

```
def has_class_but_no_id(tag):
    return tag.has_attr('class') and not tag.has_attr('id')
```

将这个方法作为参数传入 find_all() 方法, 将得到所有 <p> 标签:

```
soup.find_all(has_class_but_no_id)
# [

<b>The Dormouse's story</b></p>,
#  <p class="story">Once upon a time there were...</p>,
#  <p class="story">...</p>]


```

返回结果中只有 <p> 标签没有 <a> 标签, 因为 <a> 标签还定义了”id”, 没有返回 <html> 和 <head>, 因为 <html> 和 <head> 中没有定义”class”属性.

通过一个方法来过滤一类标签属性的时候, 这个方法的参数是要被过滤的属性的值, 而不是这个标签. 下面的例子是找出 href 属性不符合指定正则的 a 标签.

```
def not_lacie(href):
    return href and not re.compile("lacie").search(href)
soup.find_all(href=not_lacie)
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]
```

标签过滤方法可以使用复杂方法. 下面的例子可以过滤出前后都有文字的标签.

⁴ 元素参数, HTML 文档中的一个 tag 节点, 不能是文本节点

```
from bs4 import NavigableString
def surrounded_by_strings(tag):
    return (isinstance(tag.next_element, NavigableString)
            and isinstance(tag.previous_element, NavigableString))

for tag in soup.find_all(surrounded_by_strings):
    print tag.name
# p
# a
# a
# a
# p
```

现在来了解一下搜索方法的细节

7.2 find_all()

`find_all(name , attrs , recursive , string , **kwargs)`

`find_all()` 方法搜索当前 tag 的所有 tag 子节点, 并判断是否符合过滤器的条件. 这里有几个例子:

```
soup.find_all("title")
# [<title>The Dormouse's story</title>]

soup.find_all("p", "title")
# [<p class="title"><b>The Dormouse's story</b></p>]

soup.find_all("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>]

soup.find_all(id="link2")
# [<a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>]

import re
soup.find(string=re.compile("sisters"))
# u'Once upon a time there were three little sisters; and their names were\n'
```

有几个方法很相似, 还有几个方法是新的, 参数中的 `string` 和 `id` 是什么含义? 为什么 `find_all("p", "title")` 返回的是 CSS Class 为 "title" 的 `<p>` 标签? 我们来仔细看一下 `find_all()` 的参数

7.2.1 name 参数

`name` 参数可以查找所有名字为 `name` 的 tag, 字符串对象会被自动忽略掉。

简单的用法如下:

```
soup.find_all("title")
# [<title>The Dormouse's story</title>]
```

重申: 搜索 `name` 参数的值可以使任一类型的过滤器, 字符串, 正则表达式, 列表, 方法或是 `True` .

7.2.2 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, 如果包含一个名字为 `id` 的参数, Beautiful Soup 会搜索每个 tag 的 `"id"` 属性.

```
soup.find_all(id='link2')
# [<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:

```
soup.find_all(href=re.compile("elsie"))
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>]
```

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` .

下面的例子在文档树中查找所有包含 `id` 属性的 tag, 无论 `id` 的值是什么:

```
soup.find_all(id=True)
# [<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:

```
soup.find_all(href=re.compile("elsie"), id='link1')
# [<a class="sister" href="http://example.com/elsie" id="link1">three</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")
# SyntaxError: keyword can't be an expression
```

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:

```
data_soup.find_all(attrs={"data-foo": "value"})
# [<div data-foo="value">foo!</div>]
```

7.2.3 按 CSS 搜索

按照 CSS 类名搜索 tag 的功能非常实用, 但标识 CSS 类名的关键字 `class` 在 Python 中是保留字, 使用 `class` 做参数会导致语法错误. 从 Beautiful Soup 的 4.1.1 版本开始, 可以通过 `class_` 参数搜索有指定 CSS 类名的 tag:

```
soup.find_all("a", class_="sister")
# [<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>]
```

`class_` 参数同样接受不同类型的 过滤器, 字符串, 正则表达式, 方法或 `True` :

```
soup.find_all(class_=re.compile("itl"))
# [<p class="title"><b>The Dormouse's story</b></p>]

def has_six_characters(css_class):
    return css_class is not None and len(css_class) == 6

soup.find_all(class_=has_six_characters)
# [<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>]
```

tag 的 `class` 属性是多值属性. 按照 CSS 类名搜索 tag 时, 可以分别搜索 tag 中的每个 CSS 类名:

```
css_soup = BeautifulSoup('<p class="body strikeout"></p>')
css_soup.find_all("p", class_="strikeout")
# [<p class="body strikeout"></p>]

css_soup.find_all("p", class_="body")
# [<p class="body strikeout"></p>]
```

搜索 `class` 属性时也可以通过 CSS 值完全匹配:

```
css_soup.find_all("p", class_="body strikeout")
# [<p class="body strikeout"></p>]
```

完全匹配 `class` 的值时, 如果 CSS 类名的顺序与实际不符, 将搜索不到结果:

```
soup.find_all("a", attrs={"class": "sister"})
# [<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>]
```

7.2.4 string 参数

通过 `string` 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, `string` 参数接受字符串, 正则表达式, 列表, `True`. 看例子:

```
soup.find_all(string="Elsie")
# [u'Elsie']

soup.find_all(string=["Tillie", "Elsie", "Lacie"])
# [u'Elsie', u'Lacie', u'Tillie']

soup.find_all(string=re.compile("Dormouse"))
[u"The Dormouse's story", u"The Dormouse's story"]

def is_the_only_string_within_a_tag(s):
    """Return True if this string is the only child of its parent tag."""
    return (s == s.parent.string)

soup.find_all(string=is_the_only_string_within_a_tag)
# [u"The Dormouse's story", u"The Dormouse's story", u'Elsie', u'Lacie', u'Tillie
#  ↳ ', u'...']
```

虽然 `string` 参数用于搜索字符串, 还可以与其它参数混合使用来过滤 `tag`. BeautifulSoup 会找到 `.string` 方法与 `string` 参数值相符的 `tag`. 下面代码用来搜索内容里面包含“Elsie”的 `<a>` 标签:

```
soup.find_all("a", string="Elsie")
# [<a href="http://example.com/elsie" class="sister" id="link1">Elsie</a>]
```

7.2.5 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.

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:

```
soup.find_all("a", limit=2)
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>]
```

7.2.6 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` .

一段简单的文档:

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

是否使用 `recursive` 参数的搜索结果:

```
soup.html.find_all("title")
# [<title>The Dormouse's story</title>]

soup.html.find_all("title", recursive=False)
# []
```

这是文档片段

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

`<title>` 标签在 `<html>` 标签下,但并不是直接子节点,`<head>` 标签才是直接子节点. 在允许查询所有后代节点时 Beautiful Soup 能够查找到 `<title>` 标签. 但是使用了 `recursive=False` 参数之后,只能查找直接子节点,这样就查不到 `<title>` 标签了.

Beautiful Soup 提供了多种 DOM 树搜索方法. 这些方法都使用了类似的参数定义. 比如这些方法: `find_all()`: `name`, `attrs`, `text`, `limit`. 但是只有 `find_all()` 和 `find()` 支持 `recursive` 参数.

7.3 像调用 `find_all()` 一样调用 `tag`

`find_all()` 几乎是 BeautifulSoup 中最常用的搜索方法, 所以我们定义了它的简写方法. BeautifulSoup 对象和 tag 对象可以被当作一个方法来使用, 这个方法的执行结果与调用这个对象的 `find_all()` 方法相同, 下面两行代码是等价的:

```
soup.find_all("a")
soup("a")
```

这两行代码也是等价的:

```
soup.title.find_all(string=True)
soup.title(string=True)
```

7.4 `find()`

`find(name , attrs , recursive , string , **kwargs)`

`find_all()` 方法将返回文档中符合条件的所有 tag, 尽管有时候我们只想得到一个结果. 比如文档中只有一个 `<body>` 标签, 那么使用 `find_all()` 方法来查找 `<body>` 标签就不太合适, 使用 `find_all` 方法并设置 `limit=1` 参数不如直接使用 `find()` 方法. 下面两行代码是等价的:

```
soup.find_all('title', limit=1)
# [<title>The Dormouse's story</title>]

soup.find('title')
# <title>The Dormouse's story</title>
```

唯一的区别是 `find_all()` 方法的返回结果是值包含一个元素的列表, 而 `find()` 方法直接返回结果.

`find_all()` 方法没有找到目标是返回空列表, `find()` 方法找不到目标时, 返回 `None`.

```
print(soup.find("nosuchtag"))
# None
```

`soup.head.title` 是 `tag` 的名字 方法的简写. 这个简写的原理就是多次调用当前 tag 的 `find()` 方法:

```
soup.head.title
# <title>The Dormouse's story</title>

soup.find("head").find("title")
# <title>The Dormouse's story</title>
```

7.5 find_parents() 和 find_parent()

```
find_parents( name , attrs , recursive , string , **kwargs )
```

```
find_parent( name , attrs , recursive , string , **kwargs )
```

我们已经用了很大篇幅来介绍 `find_all()` 和 `find()` 方法,Beautiful Soup 中还有 10 个用于搜索的 API. 它们中的五个用的是与 `find_all()` 相同的搜索参数,另外 5 个与 `find()` 方法的搜索参数类似. 区别仅是它们搜索文档的不同部分.

记住: `find_all()` 和 `find()` 只搜索当前节点的所有子节点,孙子节点等.`find_parents()` 和 `find_parent()` 用来搜索当前节点的父辈节点,搜索方法与普通 tag 的搜索方法相同,搜索文档搜索文档包含的内容. 我们从一个文档中的一个叶子节点开始:

```
a_string = soup.find(string="Lacie")
a_string
# u'Lacie'

a_string.find_parents("a")
# [<a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>]

a_string.find_parent("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>

a_string.find_parents("p", class="title")
# []
```

文档中的一个 `<a>` 标签是是当前叶子节点的直接父节点,所以可以被找到. 还有一个 `<p>` 标签,是目标叶子节点的间接父辈节点,所以也可以被找到. 包含 `class` 值为“title”的 `<p>` 标签不是不是目标叶子节点的父辈节点,所以通过 `find_parents()` 方法搜索不到.

`find_parent()` 和 `find_parents()` 方法会让人联想到 `.parent` 和 `.parents` 属性. 它们之间的联系非常紧密. 搜索父辈节点的方法实际上就是对 `.parents` 属性的迭代搜索.

7.6 find_next_siblings() 合 find_next_sibling()

```
find_next_siblings( name , attrs , recursive , string , **kwargs )
```

```
find_next_sibling( name , attrs , recursive , string , **kwargs )
```

这 2 个方法通过 `.next_siblings` 属性对当 tag 的所有后面解析⁵ 的兄弟 tag 节

点进行迭代, `find_next_siblings()` 方法返回所有符合条件的后面的兄弟节点, `find_next_sibling()` 只返回符合条件的后面的第一个 tag 节点.

```
first_link = soup.a
first_link
# <a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>

first_link.find_next_siblings("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>]

first_story_paragraph = soup.find("p", "story")
first_story_paragraph.find_next_sibling("p")
# <p class="story">...</p>
```

7.7 `find_previous_siblings()` 和 `find_previous_sibling()`

`find_previous_siblings(name , attrs , recursive , string , **kwargs)`

`find_previous_sibling(name , attrs , recursive , string , **kwargs)`

这 2 个方法通过 `previous_siblings` 属性对当前 tag 的前面解析⁵ 的兄弟 tag 节点进行迭代, `find_previous_siblings()` 方法返回所有符合条件的前面的兄弟节点, `find_previous_sibling()` 方法返回第一个符合条件的前面的兄弟节点:

```
last_link = soup.find("a", id="link3")
last_link
# <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>

last_link.find_previous_siblings("a")
# [<a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>]

first_story_paragraph = soup.find("p", "story")
first_story_paragraph.find_previous_sibling("p")
# <p class="title"><b>The Dormouse's story</b></p>
```

7.8 `find_all_next()` 和 `find_next()`

`find_all_next(name , attrs , recursive , string , **kwargs)`

`find_next(name , attrs , recursive , string , **kwargs)`

这 2 个方法通过 `.next_elements` 属性对当前 tag 的之后的 ⁵ tag 和字符串进行迭代, `find_all_next()` 方法返回所有符合条件的节点, `find_next()` 方法返回第一个符合条件的节点:

```
first_link = soup.a
first_link
# <a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>

first_link.find_all_next(string=True)
# [u'Elsie', u',\n', u'Lacie', u' and\n', u'Tillie',
# u';\nand they lived at the bottom of a well.', u'\n\n', u'...', u'\n']

first_link.find_next("p")
# <p class="story">...</p>
```

第一个例子中, 字符串 “Elsie” 也被显示出来, 尽管它被包含在我们开始查找的 `<a>` 标签的里面. 第二个例子中, 最后一个 `<p>` 标签也被显示出来, 尽管它与我们开始查找位置的 `<a>` 标签不属于同一部分. 例子中, 搜索的重点是要匹配过滤器的条件, 并且在文档中出现的顺序而不是开始查找的元素的位置.

7.9 find_all_previous() 和 find_previous()

`find_all_previous(name , attrs , recursive , string , **kwargs)`

`find_previous(name , attrs , recursive , string , **kwargs)`

这 2 个方法通过 `.previous_elements` 属性对当前节点前面 ⁵ 的 tag 和字符串进行迭代, `find_all_previous()` 方法返回所有符合条件的节点, `find_previous()` 方法返回第一个符合条件的节点.

```
first_link = soup.a
first_link
# <a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>

first_link.find_all_previous("p")
# [<p class="story">Once upon a time there were three little sisters; ...</p>,
# <p class="title"><b>The Dormouse's story</b></p>]

first_link.find_previous("title")
# <title>The Dormouse's story</title>
```

`find_all_previous("p")` 返回了文档中的第一段 (`class="title"` 的那段), 但还返回了第二段, `<p>` 标签包含了我们开始查找的 `<a>` 标签. 不要惊讶, 这段代码的功能是查找所有出现在指定 `<a>` 标签之前的 `<p>` 标签, 因为这个 `<p>` 标签包含了开始的 `<a>` 标签, 所以 `<p>` 标签一定是在 `<a>` 之前出现的.

7.10 CSS 选择器

Beautiful Soup 支持大部分的 CSS 选择器 <http://www.w3.org/TR/CSS2/selector.html>⁶, 在 Tag 或 BeautifulSoup 对象的 `.select()` 方法中传入字符串参数, 即可使用 CSS 选择器的语法找到 tag:

```
soup.select("title")
# [<title>The Dormouse's story</title>]

soup.select("p nth-of-type(3)")
# [<p class="story">...</p>]
```

通过 tag 标签逐层查找:

```
soup.select("body 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>]

soup.select("html head title")
# [<title>The Dormouse's story</title>]
```

找到某个 tag 标签下的直接子标签⁶:

```
soup.select("head > title")
# [<title>The Dormouse's story</title>]

soup.select("p > 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>]

soup.select("p > a:nth-of-type(2)")
# [<a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>]

soup.select("p > #link1")
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>]

soup.select("body > a")
# []
```

找到兄弟节点标签:

```
soup.select("#link1 ~ .sister")
# [<a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>,
```

⁶ CSS 选择器是一种单独的文档搜索语法, 参考 http://www.w3school.com.cn/css/css_selector_type.asp

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

soup.select("#link1 + .sister")
# [<a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>]
```

通过 CSS 的类名查找:

```
soup.select(".sister")
# [<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>]

soup.select("[class~=sister]")
# [<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>]
```

通过 tag 的 id 查找:

```
soup.select("#link1")
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>]

soup.select("a#link2")
# [<a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>]
```

同时用多种 CSS 选择器查询元素:

```
soup.select("#link1,#link2")
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
# <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>]
```

通过是否存在某个属性来查找:

```
soup.select('a[href]')
# [<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>]
```

通过属性的值来查找:

```
soup.select('a[href="http://example.com/elsie"]')
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>]

soup.select('a[href^="http://example.com/"]')
# [<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>]
```

```
soup.select('a[href$="tillie"]')
# [<a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]

soup.select('a[href*=".com/el"]')
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>]
```

通过语言设置来查找:

```
multilingual_markup = """
<p lang="en">Hello</p>
<p lang="en-us">Howdy, y'all</p>
<p lang="en-gb">Pip-pip, old fruit</p>
<p lang="fr">Bonjour mes amis</p>
"""
multilingual_soup = BeautifulSoup(multilingual_markup)
multilingual_soup.select('p[lang=en]')
# [<p lang="en">Hello</p>,
#  <p lang="en-us">Howdy, y'all</p>,
#  <p lang="en-gb">Pip-pip, old fruit</p>]
```

返回查找到的元素的第一个

```
soup.select_one(".sister")
# <a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>
```

对于熟悉 CSS 选择器语法的人来说这是个非常方便的方法。Beautiful Soup 也支持 CSS 选择器 API, 如果你仅仅需要 CSS 选择器的功能, 那么直接使用 `lxml` 也可以, 而且速度更快, 支持更多的 CSS 选择器语法, 但 Beautiful Soup 整合了 CSS 选择器的语法和自身方便使用 API。

修改文档树

Beautiful Soup 的强项是文档树的搜索, 但同时也可以方便的修改文档树

8.1 修改 tag 的名称和属性

在 *Attributes* 的章节中已经介绍过这个功能, 但是再看一遍也无妨. 重命名一个 tag, 改变属性的值, 添加或删除属性:

```
soup = BeautifulSoup('<b class="boldest">Extremely bold</b>')
tag = soup.b

tag.name = "blockquote"
tag['class'] = 'verybold'
tag['id'] = 1
tag
# <blockquote class="verybold" id="1">Extremely bold</blockquote>

del tag['class']
del tag['id']
tag
# <blockquote>Extremely bold</blockquote>
```

8.2 修改.string

给 tag 的 .string 属性赋值, 就相当于用当前的内容替代了原来的内容:

```
markup = '<a href="http://example.com/">I linked to <i>example.com</i></a>'
soup = BeautifulSoup(markup)
```

```
tag = soup.a
tag.string = "New link text."
tag
# <a href="http://example.com/">New link text.</a>
```

注意: 如果当前的 tag 包含了其它 tag, 那么给它的 .string 属性赋值会覆盖掉原有的所有内容包括子 tag

8.3 append()

Tag.append() 方法想 tag 中添加内容, 就好像 Python 的列表的 .append() 方法:

```
soup = BeautifulSoup("<a>Foo</a>")
soup.a.append("Bar")

soup
# <html><head></head><body><a>FooBar</a></body></html>
soup.a.contents
# [u'Foo', u'Bar']
```

8.4 NavigableString() 和.new_tag()

如果想添加一段文本内容到文档中也没问题, 可以调用 Python 的 append() 方法或调用 NavigableString 的构造方法:

```
soup = BeautifulSoup("<b></b>")
tag = soup.b
tag.append("Hello")
new_string = NavigableString(" there")
tag.append(new_string)
tag
# <b>Hello there.</b>
tag.contents
# [u'Hello', u' there']
```

如果想要创建一段注释, 或 NavigableString 的任何子类, 只要调用 NavigableString 的构造方法:

```
from bs4 import Comment
new_comment = soup.new_string("Nice to see you.", Comment)
tag.append(new_comment)
tag
```

```
# <b>Hello there<!--Nice to see you.--></b>
tag.contents
# [u'Hello', u' there', u'Nice to see you.']
```

这是 BeautifulSoup 4.2.1 中新增的方法

创建一个 tag 最好的方法是调用工厂方法 BeautifulSoup.new_tag() :

```
soup = BeautifulSoup("<b></b>")
original_tag = soup.b

new_tag = soup.new_tag("a", href="http://www.example.com")
original_tag.append(new_tag)
original_tag
# <b><a href="http://www.example.com"></a></b>

new_tag.string = "Link text."
original_tag
# <b><a href="http://www.example.com">Link text.</a></b>
```

第一个参数作为 tag 的 name, 是必填, 其它参数选填

8.5 insert()

Tag.insert() 方法与 Tag.append() 方法类似, 区别是不会把新元素添加到父节点 .contents 属性的最后, 而是把元素插入到指定的位置. 与 Python 列表总的 .insert() 方法的用法下同:

```
markup = '<a href="http://example.com/">I linked to <i>example.com</i></a>'
soup = BeautifulSoup(markup)
tag = soup.a

tag.insert(1, "but did not endorse ")
tag
# <a href="http://example.com/">I linked to but did not endorse <i>example.com</i>
↪</a>
tag.contents
# [u'I linked to ', u'but did not endorse', <i>example.com</i>]
```

8.6 insert_before() 和 insert_after()

insert_before() 方法在当前 tag 或文本节点前插入内容:

```
soup = BeautifulSoup("<b>stop</b>")
tag = soup.new_tag("i")
tag.string = "Don't"
soup.b.string.insert_before(tag)
soup.b
# <b><i>Don't</i>stop</b>
```

`insert_after()` 方法在当前 tag 或文本节点后插入内容:

```
soup.b.i.insert_after(soup.new_string(" ever "))
soup.b
# <b><i>Don't</i> ever stop</b>
soup.b.contents
# [<i>Don't</i>, u' ever ', u'stop']
```

8.7 clear()

`Tag.clear()` 方法移除当前 tag 的内容:

```
markup = '<a href="http://example.com/">I linked to <i>example.com</i></a>'
soup = BeautifulSoup(markup)
tag = soup.a

tag.clear()
tag
# <a href="http://example.com/"></a>
```

8.8 extract()

`PageElement.extract()` 方法将当前 tag 移除文档树, 并作为方法结果返回:

```
markup = '<a href="http://example.com/">I linked to <i>example.com</i></a>'
soup = BeautifulSoup(markup)
a_tag = soup.a

i_tag = soup.i.extract()

a_tag
# <a href="http://example.com/">I linked to</a>

i_tag
# <i>example.com</i>
```



```
print(i_tag.parent)
None
```

这个方法实际上产生了 2 个文档树: 一个是用来解析原始文档的 BeautifulSoup 对象, 另一个是被移除并且返回的 tag. 被移除并返回的 tag 可以继续调用 `extract` 方法:

```
my_string = i_tag.string.extract()
my_string
# u'example.com'

print(my_string.parent)
# None
i_tag
# <i></i>
```

8.9 decompose()

`Tag.decompose()` 方法将当前节点移除文档树并完全销毁:

```
markup = '<a href="http://example.com/">I linked to <i>example.com</i></a>'
soup = BeautifulSoup(markup)
a_tag = soup.a

soup.i.decompose()

a_tag
# <a href="http://example.com/">I linked to</a>
```

8.10 replace_with()

`PageElement.replace_with()` 方法移除文档树中的某段内容, 并用新 tag 或文本节点替代它:

```
markup = '<a href="http://example.com/">I linked to <i>example.com</i></a>'
soup = BeautifulSoup(markup)
a_tag = soup.a

new_tag = soup.new_tag("b")
new_tag.string = "example.net"
a_tag.i.replace_with(new_tag)
```

```
a_tag
# <a href="http://example.com/">I linked to <b>example.net</b></a>
```

`replace_with()` 方法返回被替代的 tag 或文本节点, 可以用来浏览或添加到文档树其它地方

8.11 wrap()

`PageElement.wrap()` 方法可以对指定的 tag 元素进行包装⁸, 并返回包装后的结果:

```
soup = BeautifulSoup("<p>I wish I was bold.</p>")
soup.p.string.wrap(soup.new_tag("b"))
# <b>I wish I was bold.</b>

soup.p.wrap(soup.new_tag("div"))
# <div><p><b>I wish I was bold.</b></p></div>
```

该方法在 Beautiful Soup 4.0.5 中添加

8.12 unwrap()

`Tag.unwrap()` 方法与 `wrap()` 方法相反. 将移除 tag 内的所有 tag 标签, 该方法常被用来进行标记的解包:

```
markup = '<a href="http://example.com/">I linked to <i>example.com</i></a>'
soup = BeautifulSoup(markup)
a_tag = soup.a

a_tag.i.unwrap()
a_tag
# <a href="http://example.com/">I linked to example.com</a>
```

与 `replace_with()` 方法相同, `unwrap()` 方法返回被移除的 tag

⁸ `wrap` 含有包装, 打包的意思, 但是这里的包装不是在外包装而是将当前 tag 的内部内容包装在一个 tag 里. 包装原来内容的新 tag 依然在执行 `wrap()` 方法的 tag 内

输出

9.1 格式化输出

`prettify()` 方法将 BeautifulSoup 的文档树格式化后以 Unicode 编码输出, 每个 XML/HTML 标签都独占一行

```
markup = '<a href="http://example.com/">I linked to <i>example.com</i></a>'
soup = BeautifulSoup(markup)
soup.prettify()
# '<html>\n <head>\n </head>\n <body>\n  <a href="http://example.com/">\n...'

print(soup.prettify())
# <html>
# <head>
# </head>
# <body>
#   <a href="http://example.com/">
#     I linked to
#     <i>
#       example.com
#     </i>
#   </a>
# </body>
# </html>
```

BeautifulSoup 对象和它的 tag 节点都可以调用 `prettify()` 方法:

```
print(soup.a.prettify())
# <a href="http://example.com/">
#   I linked to
#   <i>
#     example.com
#   </i>
```

```
# </a>
```

9.2 压缩输出

如果只想得到结果字符串, 不重视格式, 那么可以对一个 BeautifulSoup 对象或 Tag 对象使用 Python 的 `unicode()` 或 `str()` 方法:

```
str(soup)
# '<html><head></head><body><a href="http://example.com/">I linked to <i>example.
  ↳com</i></a></body></html>'

unicode(soup.a)
# u'<a href="http://example.com/">I linked to <i>example.com</i></a>'
```

`str()` 方法返回 UTF-8 编码的字符串, 可以指定编码的设置.

还可以调用 `encode()` 方法获得字节码或调用 `decode()` 方法获得 Unicode.

9.3 输出格式

Beautiful Soup 输出是会将 HTML 中的特殊字符转换成 Unicode, 比如 ““”: :

```
soup = BeautifulSoup("&ldquo;Dammit!&rdquo; he said.")
unicode(soup)
# u'<html><head></head><body>\u201cDammit!\u201d he said.</body></html>'
```

如果将文档转换成字符串, Unicode 编码会被编码成 UTF-8. 这样就无法正确显示 HTML 特殊字符了:

```
str(soup)
# '<html><head></head><body>\xe2\x80\x9cDammit!\xe2\x80\x9d he said.</body></
  ↳html>'
```

9.4 get_text()

如果只想得到 tag 中包含的文本内容, 那么可以调用 `get_text()` 方法, 这个方法获取到 tag 中包含的所有文版内容包括子孙 tag 中的内容, 并将结果作为 Unicode 字符串返回:

```
markup = '<a href="http://example.com/">\nI linked to <i>example.com</i>\n</a>'
soup = BeautifulSoup(markup)

soup.get_text()
u'\nI linked to example.com\n'
soup.i.get_text()
u'example.com'
```

可以通过参数指定 tag 的文本内容的分隔符:

```
# soup.get_text("/")
u'\nI linked to |example.com|\n'
```

还可以去除获得文本内容的前后空白:

```
# soup.get_text("/", strip=True)
u'I linked to|example.com'
```

或者使用 `.stripped_strings` 生成器, 获得文本列表后手动处理列表:

```
[text for text in soup.stripped_strings]
# [u'I linked to', u'example.com']
```

指定文档解析器

如果仅是想要解析 HTML 文档, 只要用文档创建 `BeautifulSoup` 对象就可以了. `Beautiful Soup` 会自动选择一个解析器来解析文档. 但是还可以通过参数指定使用那种解析器来解析当前文档.

`BeautifulSoup` 第一个参数应该是要被解析的文档字符串或是文件句柄, 第二个参数用来标识怎样解析文档. 如果第二个参数为空, 那么 `Beautiful Soup` 根据当前系统安装的库自动选择解析器, 解析器的优先数序: `lxml`, `html5lib`, Python 标准库. 在下面两种条件下解析器优先顺序会变化:

- 要解析的文档是什么类型: 目前支持, “html”, “xml”, 和 “html5”
- 指定使用哪种解析器: 目前支持, “lxml”, “html5lib”, 和 “html.parser”

安装解析器 章节介绍了可以使用哪种解析器, 以及如何安装.

如果指定的解析器没有安装, `Beautiful Soup` 会自动选择其它方案. 目前只有 `lxml` 解析器支持 XML 文档的解析, 在没有安装 `lxml` 库的情况下, 创建 `beautifulsoup` 对象时无论是否指定使用 `lxml`, 都无法得到解析后的对象

10.1 解析器之间的区别

`Beautiful Soup` 为不同的解析器提供了相同的接口, 但解析器本身时有区别的. 同一篇文档被不同的解析器解析后可能会生成不同结构的树型文档. 区别最大的是 HTML 解析器和 XML 解析器, 看下面片段被解析成 HTML 结构:

```
BeautifulSoup("<a><b /></a>")
# <html><head></head><body><a><b></b></a></body></html>
```

因为空标签 `` 不符合 HTML 标准, 所以解析器把它解析成 ``

同样的文档使用 XML 解析如下 (解析 XML 需要安装 `lxml` 库). 注意, 空标签 `` 依然被保留, 并且文档前添加了 XML 头, 而不是被包含在 `<html>` 标签内:

```
BeautifulSoup("<a><b /></a>", "xml")
# <?xml version="1.0" encoding="utf-8"?>
# <a><b/></a>
```

HTML 解析器之间也有区别, 如果被解析的 HTML 文档是标准格式, 那么解析器之间没有任何差别, 只是解析速度不同, 结果都会返回正确的文档树。

但是如果被解析文档不是标准格式, 那么不同的解析器返回结果可能不同。下面例子中, 使用 lxml 解析错误格式的文档, 结果 `</p>` 标签被直接忽略掉了:

```
BeautifulSoup("<a></p>", "lxml")
# <html><body><a></a></body></html>
```

使用 html5lib 库解析相同文档会得到不同的结果:

```
BeautifulSoup("<a></p>", "html5lib")
# <html><head></head><body><a><p></p></a></body></html>
```

html5lib 库没有忽略掉 `</p>` 标签, 而是自动补全了标签, 还给文档树添加了 `<head>` 标签。

使用 python 内置库解析结果如下:

```
BeautifulSoup("<a></p>", "html.parser")
# <a></a>
```

与 lxml⁷ 库类似的, Python 内置库忽略掉了 `</p>` 标签, 与 html5lib 库不同的是标准库没有尝试创建符合标准的文档格式或将文档片段包含在 `<body>` 标签内, 与 lxml 不同的是标准库甚至连 `<html>` 标签都没有尝试去添加。

因为文档片段“`<a></p>`”是错误格式, 所以以上解析方式都能算作“正确”, html5lib 库使用的是 HTML5 的部分标准, 所以最接近“正确”。不过所有解析器的结构都能够被认为是“正常”的。

不同的解析器可能影响代码执行结果, 如果在分发给别人的代码中使用了 BeautifulSoup, 那么最好注明使用了哪种解析器, 以减少不必要的麻烦。

⁷ 原文写的是 html5lib, 译者觉得这是原文档的一个笔误

编码

任何 HTML 或 XML 文档都有自己的编码方式, 比如 ASCII 或 UTF-8, 但是使用 BeautifulSoup 解析后, 文档都被转换成了 Unicode:

```
markup = "<h1>Sacré bleu!</h1>"
soup = BeautifulSoup(markup)
soup.h1
# <h1>Sacré bleu!</h1>
soup.h1.string
# u'Sacr   bleu!'
```

这不是魔术 (但很神奇), BeautifulSoup 用了[编码自动检测](#) 子库来识别当前文档编码并转换成 Unicode 编码. BeautifulSoup 对象的 `.original_encoding` 属性记录了自动识别编码的结果:

```
soup.original_encoding
'utf-8'
```

[编码自动检测](#) 功能大部分时候都能猜对编码格式, 但有时候也会出错. 有时候即使猜测正确, 也是在逐个字节的遍历整个文档后才猜对的, 这样很慢. 如果预先知道文档编码, 可以设置编码参数来减少自动检查编码出错的概率并且提高文档解析速度. 在创建 BeautifulSoup 对象的时候设置 `from_encoding` 参数.

下面一段文档用了 ISO-8859-8 编码方式, 这段文档太短, 结果 BeautifulSoup 以为文档是用 ISO-8859-7 编码:

```
markup = b"<h1>\xed\xe5\xec\xf9</h1>"
soup = BeautifulSoup(markup)
soup.h1
<h1>νϵμω</h1>
soup.original_encoding
'ISO-8859-7'
```

通过传入 `from_encoding` 参数来指定编码方式:

```
soup = BeautifulSoup(markup, from_encoding="iso-8859-8")
soup.h1
<h1> </h1>
soup.original_encoding
'iso8859-8'
```

如果仅知道文档采用了 Unicode 编码, 但不知道具体编码. 可以先自己猜测, 猜测错误 (依旧是乱码) 时, 可以把错误编码作为 `exclude_encodings` 参数, 这样文档就不会尝试使用这种编码了解码了. 译者备注: 在没有指定编码的情况下, BS 会自己猜测编码, 把不正确的编码排除掉, BS 就更容易猜到正确编码.

```
soup = BeautifulSoup(markup, exclude_encodings=["ISO-8859-7"])
soup.h1
<h1> </h1>
soup.original_encoding
'WINDOWS-1255'
```

猜测结果是 Windows-1255 编码, 猜测结果可能不够准确, 但是 Windows-1255 编码是 ISO-8859-8 的扩展集, 所以猜测结果已经十分接近了, 并且不影响使用. (`exclude_encodings` 参数是 4.4.0 版本的新功能)

少数情况下 (通常是 UTF-8 编码的文档中包含了其它编码格式的文件), 想获得正确的 Unicode 编码就不得不将文档中少数特殊编码字符替换成特殊 Unicode 编码, “REPLACEMENT CHARACTER” (U+FFFD,)⁹. 如果 Beautiful Soup 猜测文档编码时作了特殊字符的替换, 那么 BeautifulSoup 会把 `UnicodeDammit` 或 `BeautifulSoup` 对象的 `.contains_replacement_characters` 属性标记为 `True`. 这样就可以知道当前文档进行 Unicode 编码后丢失了一部分特殊内容字符. 如果文档中包含 而 `.contains_replacement_characters` 属性是 `False`, 则表示 就是文档中原来的字符, 不是转码失败.

11.1 输出编码

通过 Beautiful Soup 输出文档时, 不管输入文档是什么编码方式, 输出编码均为 UTF-8 编码, 下面例子输入文档是 Latin-1 编码:

```
markup = b'''
<html>
  <head>
    <meta content="text/html; charset=ISO-Latin-1" http-equiv="Content-type" />
  </head>
  <body>
    <p>Sacr\xe9 bleu!</p>
```

⁹ 文档中特殊编码字符被替换成特殊字符 (通常是) 的过程是 Beautiful Soup 自动实现的, 如果想要多种编码格式的文档被完全转换正确, 那么, 只好, 预先手动处理, 统一编码格式

```

    </body>
</html>
'''

soup = BeautifulSoup(markup)
print(soup.prettify())
# <html>
# <head>
# <meta content="text/html; charset=utf-8" http-equiv="Content-type" />
# </head>
# <body>
# <p>
#   Sacré bleu!
# </p>
# </body>
# </html>

```

注意, 输出文档中的 `<meta>` 标签的编码设置已经修改成了与输出编码一致的 UTF-8.

如果不想用 UTF-8 编码输出, 可以将编码方式传入 `prettify()` 方法:

```

print(soup.prettify("latin-1"))
# <html>
# <head>
# <meta content="text/html; charset=latin-1" http-equiv="Content-type" />
# ...

```

还可以调用 BeautifulSoup 对象或任意节点的 `encode()` 方法, 就像 Python 的字符串调用 `encode()` 方法一样:

```

soup.p.encode("latin-1")
# '<p>Sacré bleu!</p>'

soup.p.encode("utf-8")
# '<p>Sacré bleu!</p>'

```

如果文档中包含当前编码不支持的字符, 那么这些字符将呗转换成一系列 XML 特殊字符引用, 下面例子中包含了 Unicode 编码字符 SNOWMAN:

```

markup = u"<b>\N{SNOWMAN}</b>"
snowman_soup = BeautifulSoup(markup)
tag = snowman_soup.b

```

SNOWMAN 字符在 UTF-8 编码中可以正常显示 (看上去像是), 但有些编码不支持 SNOWMAN 字符, 比如 ISO-Latin-1 或 ASCII, 那么在这些编码中 SNOWMAN 字符会被转换成 “☃”:

```
print(tag.encode("utf-8"))
# <b></b>

print tag.encode("latin-1")
# <b>Ë#9731;</b>

print tag.encode("ascii")
# <b>Ë#9731;</b>
```

11.2 Unicode, Dammit! (乱码, 靠!)

译者备注: UnicodeDammit 是 BS 内置库, 主要用来猜测文档编码.

编码自动检测 功能可以在 BeautifulSoup 以外使用,检测某段未知编码时,可以使用这个方法:

```
from bs4 import UnicodeDammit
dammit = UnicodeDammit("Sacré bleu!")
print(dammit.unicode_markup)
# Sacré bleu!
dammit.original_encoding
# 'utf-8'
```

如果 Python 中安装了 `chardet` 或 `cchardet` 那么编码检测功能的准确率将大大提高。输入的字符越多, 检测结果越精确, 如果事先猜测到一些可能编码, 那么可以将猜测的编码作为参数, 这样将优先检测这些编码:

```
dammit = UnicodeDammit("Sacré bleu!", ["latin-1", "iso-8859-1"])
print(dammit.unicode_markup)
# Sacré bleu!
dammit.original_encoding
# 'latin-1'
```

编码自动检测 功能中有 2 项功能是 Beautiful Soup 库中用不到的

11.2.1 智能引号

使用 Unicode 时,Beautiful Soup 还会智能的把引号 ¹⁰ 转换成 HTML 或 XML 中的特殊字符:

¹⁰ 智能引号, 常出现在 microsoft 的 word 软件中, 即在某一段落中按引号出现的顺序每个引号都被自动转换为左引号, 或右引号.

```
markup = b"<p>I just \x93love\x94 Microsoft Word\x92s smart quotes</p>"

UnicodeDammit(markup, ["windows-1252"], smart_quotes_to="html").unicode_markup
# u'<p>I just &lquote;love&rdquo; Microsoft Word&rsquo;s smart quotes</p>'

UnicodeDammit(markup, ["windows-1252"], smart_quotes_to="xml").unicode_markup
# u'<p>I just &#x201C;love&#x201D; Microsoft Word&#x2019;s smart quotes</p>'
```

也可以把引号转换为 ASCII 码:

```
UnicodeDammit(markup, ["windows-1252"], smart_quotes_to="ascii").unicode_markup
# u'<p>I just "love" Microsoft Word\'s smart quotes</p>'
```

很有用的功能, 但是 Beautiful Soup 没有使用这种方式. 默认情况下, Beautiful Soup 把引号转换成 Unicode:

```
UnicodeDammit(markup, ["windows-1252"]).unicode_markup
# u'<p>I just \u201clove\u201d Microsoft Word\u2019s smart quotes</p>'
```

11.2.2 矛盾的编码

有时文档的大部分都是用 UTF-8, 但同时还包含了 Windows-1252 编码的字符, 就像微软的智能引号¹⁰一样. 一些包含多个信息的来源网站容易出现这种情况. UnicodeDammit.detwingle() 方法可以把这类文档转换成纯 UTF-8 编码格式, 看个简单的例子:

```
snowmen = (u"\N{SNOWMAN}" * 3)
quote = (u"\N{LEFT DOUBLE QUOTATION MARK}I like snowmen!\N{RIGHT DOUBLE QUOTATION MARK}")
doc = snowmen.encode("utf8") + quote.encode("windows_1252")
```

这段文档很杂乱, snowmen 是 UTF-8 编码, 引号是 Windows-1252 编码, 直接输出时不能同时显示 snowmen 和引号, 因为它们编码不同:

```
print(doc)
# I like snowmen!

print(doc.decode("windows-1252"))
# â~fâ~fâ~f “I like snowmen!”
```

如果对这段文档用 UTF-8 解码就会得到 UnicodeDecodeError 异常, 如果用 Windows-1252 解码就回得到一堆乱码. 幸好, UnicodeDammit.detwingle() 方法会吧这段字符串转换成 UTF-8 编码, 允许我们同时显示出文档中的 snowmen 和引号:

```
new_doc = UnicodeDammit.detwingle(doc)
print(new_doc.decode("utf8"))
#    "I like snowmen!"
```

`UnicodeDammit.detwingle()` 方法只能解码包含在 UTF-8 编码中的 Windows-1252 编码内容, 但这解决了最常见的一类问题.

在创建 `BeautifulSoup` 或 `UnicodeDammit` 对象前一定要先对文档调用 `UnicodeDammit.detwingle()` 确保文档的编码方式正确. 如果尝试去解析一段包含 Windows-1252 编码的 UTF-8 文档, 就会得到一堆乱码, 比如: `â~fâ~fâ~f "I like snowmen!"`

`UnicodeDammit.detwingle()` 方法在 Beautiful Soup 4.1.0 版本中新增

比较对象是否相同

两个 `NavigableString` 或 `Tag` 对象具有相同的 HTML 或 XML 结构时, `Beautiful Soup` 就判断这两个对象相同. 这个例子中, 2 个 `` 标签在 BS 中是相同的, 尽管他们在文档树的不同位置, 但是具有相同的表象: “`pizza`”

```
markup = "<p>I want <b>pizza</b> and more <b>pizza</b>!</p>"
soup = BeautifulSoup(markup, 'html.parser')
first_b, second_b = soup.find_all('b')
print first_b == second_b
# True

print first_b.previous_element == second_b.previous_element
# False
```

如果想判断两个对象是否严格的指向同一个对象可以通过 `is` 来判断

```
print first_b is second_b
# False
```

复制 BeautifulSoup 对象

`copy.copy()` 方法可以复制任意 Tag 或 `NavigableString` 对象

```
import copy
p_copy = copy.copy(soup.p)
print p_copy
# <p>I want <b>pizza</b> and more <b>pizza</b>!</p>
```

复制后的对象跟与对象是相等的, 但指向不同的内存地址

```
print soup.p == p_copy
# True

print soup.p is p_copy
# False
```

源对象和复制对象的区别是源对象在文档树中, 而复制后的对象是独立的还没有添加到文档树中. 复制后对象的效果跟调用了 `extract()` 方法相同.

```
print p_copy.parent
# None
```

这是因为相等的对象不能同时插入相同的位置

解析部分文档

如果仅仅因为想要查找文档中的 `<a>` 标签而将整片文档进行解析, 实在是浪费内存和时间. 最快的方法是从一开始就把 `<a>` 标签以外的东西都忽略掉. `SoupStrainer` 类可以定义文档的某段内容, 这样搜索文档时就不必先解析整篇文档, 只会解析在 `SoupStrainer` 中定义过的文档. 创建一个 `SoupStrainer` 对象并作为 `parse_only` 参数给 `BeautifulSoup` 的构造方法即可.

14.1 SoupStrainer

`SoupStrainer` 类接受与典型搜索方法相同的参数: `name`, `attrs`, `recursive`, `string`, `**kwargs`。下面举例说明三种 `SoupStrainer` 对象:

```
from bs4 import SoupStrainer

only_a_tags = SoupStrainer("a")

only_tags_with_id_link2 = SoupStrainer(id="link2")

def is_short_string(string):
    return len(string) < 10

only_short_strings = SoupStrainer(string=is_short_string)
```

再拿“爱丽丝”文档来举例, 来看看使用三种 `SoupStrainer` 对象做参数会有什么不同:

```
html_doc = """
<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>
"""

print(BeautifulSoup(html_doc, "html.parser", parse_only=only_a_tags).prettify())
# <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>

print(BeautifulSoup(html_doc, "html.parser", parse_only=only_tags_with_id_link2).
    .prettify())
# <a class="sister" href="http://example.com/lacie" id="link2">
# Lacie
# </a>

print(BeautifulSoup(html_doc, "html.parser", parse_only=only_short_strings).
    .prettify())
# Elsie
# ,
# Lacie
# and
# Tillie
# ...
#
```

还可以将 `SoupStrainer` 作为参数传入搜索文档树中提到的方法. 这可能不是个常用用法, 所以还是提一下:

```
soup = BeautifulSoup(html_doc)
soup.find_all(only_short_strings)
# [u'\n\n', u'\n\n', u'Elsie', u',\n', u'Lacie', u' and\n', u'Tillie',
#  u'\n\n', u'...', u'\n']
```

常见问题

15.1 代码诊断

如果想知道 BeautifulSoup 到底怎样处理一份文档, 可以将文档传入 `diagnose()` 方法 (Beautiful Soup 4.2.0 中新增), BeautifulSoup 会输出一份报告, 说明不同的解析器会怎样处理这段文档, 并标出当前的解析过程会使用哪种解析器:

```
from bs4.diagnose import diagnose
data = open("bad.html").read()
diagnose(data)

# Diagnostic running on BeautifulSoup 4.2.0
# Python version 2.7.3 (default, Aug 1 2012, 05:16:07)
# I noticed that html5lib is not installed. Installing it may help.
# Found lxml version 2.3.2.0
#
# Trying to parse your data with html.parser
# Here's what html.parser did with the document:
# ...
```

`diagnose()` 方法的输出结果可能帮助你找到问题的原因, 如果不行, 还可以把结果复制出来以便寻求他人的帮助

15.2 文档解析错误

文档解析错误有两种. 一种是崩溃, BeautifulSoup 尝试解析一段文档结果却抛除了异常, 通常是 `HTMLParser.HTMLParseError`. 还有一种异常情况, 是 BeautifulSoup 解析后的文档树看起来与原来的内容相差很多.

这些错误几乎都不是 BeautifulSoup 的原因, 这不会是因为 BeautifulSoup 的代码写的太优秀, 而是因为 BeautifulSoup 没有包含任何文档解析代码. 异常产生自被依赖的解析

器, 如果解析器不能很好的解析出当前的文档, 那么最好的办法是换一个解析器. 更多细节查看[安装解析器](#) 章节.

最常见的解析错误是 `HTMLParser.HTMLParseError: malformed start tag` 和 `HTMLParser.HTMLParseError: bad end tag`. 这都是由 Python 内置的解析器引起的, 解决方法是安装 [lxml](#) 或 [html5lib](#)

最常见的异常现象是当前文档找不到指定的 Tag, 而这个 Tag 光是用眼睛就足够发现的了. `find_all()` 方法返回 `[]`, 而 `find()` 方法返回 `None`. 这是 Python 内置解析器的又一个问题: 解析器会跳过那些它不知道的 tag. 解决方法还是安装 [lxml](#) 或 [html5lib](#)

15.3 版本错误

- `SyntaxError: Invalid syntax` (异常位置在代码行: `ROOT_TAG_NAME = u'[document]'`), 因为 Python2 版本的代码没有经过迁移就在 Python3 中窒息感
- `ImportError: No module named HTMLParser` 因为在 Python3 中执行 Python2 版本的 Beautiful Soup
- `ImportError: No module named html.parser` 因为在 Python2 中执行 Python3 版本的 Beautiful Soup
- `ImportError: No module named BeautifulSoup` 因为在没有安装 BeautifulSoup3 库的 Python 环境下执行代码, 或忘记了 BeautifulSoup4 的代码需要从 `bs4` 包中引入
- `ImportError: No module named bs4` 因为当前 Python 环境下还没有安装 BeautifulSoup4

15.4 解析成 XML

默认情况下,Beautiful Soup 会将当前文档作为 HTML 格式解析, 如果要解析 XML 文档, 要在 BeautifulSoup 构造方法中加入第二个参数 “xml”:

```
soup = BeautifulSoup(markup, "xml")
```

当然, 还需要安装 [lxml](#)

15.5 解析器的错误

- 如果同样的代码在不同环境下结果不同, 可能是因为两个环境下使用不同的解析器造成的. 例如这个环境中安装了 `lxml`, 而另一个环境中只有 `html5lib`, [解析器之间的区别](#) 中说明了原因. 修复方法是在 BeautifulSoup 的构造方法中指定解析器

- 因为 HTML 标签是 [大小写敏感](#) 的, 所以 3 种解析器再出来文档时都将 tag 和属性转换成小写. 例如文档中的 `<TAG></TAG>` 会被转换为 `<tag></tag>`. 如果想要保留 tag 的大写的话, 那么应该将文档解析成 [XML](#).

15.6 杂项错误

- `UnicodeEncodeError: 'charmap' codec can't encode character u'\xfoo' in position bar` (或其它类型的 `UnicodeEncodeError`) 的错误, 主要是两方面的错误 (都不是 Beautiful Soup 的原因), 第一种是正在使用的终端 (console) 无法显示部分 Unicode, 参考 [Python wiki](#), 第二种是向文件写入时, 被写入文件不支持部分 Unicode, 这时只要用 `u.encode("utf8")` 方法将编码转换为 UTF-8.
- `KeyError: [attr]` 因为调用 `tag['attr']` 方法而引起, 因为这个 tag 没有定义该属性. 出错最多的是 `KeyError: 'href'` 和 `KeyError: 'class'`. 如果不确定某个属性是否存在时, 用 `tag.get('attr')` 方法去获取它, 跟获取 Python 字典的 key 一样
- `AttributeError: 'ResultSet' object has no attribute 'foo'` 错误通常是因为把 `find_all()` 的返回结果当作一个 tag 或文本节点使用, 实际上返回结果是一个列表或 `ResultSet` 对象的字符串, 需要对结果进行循环才能得到每个节点的 `.foo` 属性. 或者使用 `find()` 方法仅获取到一个节点
- `AttributeError: 'NoneType' object has no attribute 'foo'` 这个错误通常是在调用了 `find()` 方法后直接取某个属性 `.foo` 但是 `find()` 方法并没有找到任何结果, 所以它的返回值是 `None`. 需要找出为什么 `find()` 的返回值是 `None`.

15.7 如何提高效率

Beautiful Soup 对文档的解析速度不会比它所依赖的解析器更快, 如果对计算时间要求很高或者计算机的时间比程序员的时间更值钱, 那么就应该直接使用 [lxml](#).

换句话说, 还有提高 Beautiful Soup 效率的办法, 使用 [lxml](#) 作为解析器. Beautiful Soup 用 [lxml](#) 做解析器比用 [html5lib](#) 或 Python 内置解析器速度快很多.

安装 [cchardet](#) 后文档的解码的编码检测会速度更快

解析部分文档 不会节省多少解析时间, 但是会节省很多内存, 并且搜索时也会变得更快.

Beautiful Soup 3

Beautiful Soup 3 是上一个发布版本, 目前已经停止维护. Beautiful Soup 3 库目前已经被几个主要的 linux 平台添加到源里:

```
$ apt-get install Python-beautifulsoup
```

在 PyPi 中分发的包名字是 BeautifulSoup :

```
$ easy_install BeautifulSoup
```

```
$ pip install BeautifulSoup
```

或通过 [Beautiful Soup 3.2.0 源码包](#) 安装

Beautiful Soup 3 的在线文档查看 [这里](#) .

16.1 迁移到 BS4

只要一个小变动就能让大部分的 BeautifulSoup 3 代码使用 BeautifulSoup 4 的库和方法 — 修改 BeautifulSoup 对象的引入方式:

```
from BeautifulSoup import BeautifulSoup
```

修改为:

```
from bs4 import BeautifulSoup
```

- 如果代码抛出 `ImportError` 异常 “No module named BeautifulSoup”, 原因可能是尝试执行 BeautifulSoup 3, 但环境中只安装了 BeautifulSoup 4 库
- 如果代码跑出 `ImportError` 异常 “No module named bs4”, 原因可能是尝试运行 BeautifulSoup 4 的代码, 但环境中只安装了 BeautifulSoup 3.

虽然 BS4 兼容绝大部分 BS3 的功能, 但 BS3 中的大部分方法已经不推荐使用了, 就方法按照 [PEP8 标准](#) 重新定义了方法名. 很多方法都重新定义了方法名, 但只有少数几个方法没有向下兼容.

上述内容就是 BS3 迁移到 BS4 的注意事项

16.1.1 需要的解析器

Beautiful Soup 3 曾使用 Python 的 `SGMLParser` 解析器, 这个模块在 Python3 中已经被移除了. Beautiful Soup 4 默认使用系统的 `html.parser`, 也可以使用 `lxml` 或 `html5lib` 扩展库代替. 查看[安装解析器](#) 章节

因为解析器 `html.parser` 与 `SGMLParser` 不同. BS4 和 BS3 处理相同的文档会产生不同的对象结构. 使用 `lxml` 或 `html5lib` 解析文档的时候, 如果添加了 `html.parser` 参数, 解析的对象又回发生变化. 如果发生了这种情况, 只能修改对应的处文档结果处理代码了.

16.1.2 方法名的变化

- `renderContents` -> `encode_contents`
- `replaceWith` -> `replace_with`
- `replaceWithChildren` -> `unwrap`
- `findAll` -> `find_all`
- `findAllNext` -> `find_all_next`
- `findAllPrevious` -> `find_all_previous`
- `findNext` -> `find_next`
- `findNextSibling` -> `find_next_sibling`
- `findNextSiblings` -> `find_next_siblings`
- `findParent` -> `find_parent`
- `findParents` -> `find_parents`
- `findPrevious` -> `find_previous`
- `findPreviousSibling` -> `find_previous_sibling`
- `findPreviousSiblings` -> `find_previous_siblings`
- `nextSibling` -> `next_sibling`
- `previousSibling` -> `previous_sibling`

Beautiful Soup 构造方法的参数部分也有名字变化:

- BeautifulSoup(parseOnlyThese=...) -> BeautifulSoup(parse_only=...)
- BeautifulSoup(fromEncoding=...) -> BeautifulSoup(from_encoding=...)

为了适配 Python3, 修改了一个方法名:

- Tag.has_key() -> Tag.has_attr()

修改了一个属性名, 让它看起来更专业点:

- Tag.isSelfClosing -> Tag.is_empty_element

修改了下面 3 个属性的名字, 以免和 Python 保留字冲突. 这些变动不是向下兼容的, 如果在 BS3 中使用了这些属性, 那么在 BS4 中这些代码无法执行.

- UnicodeDammit.Unicode -> UnicodeDammit.Unicode_markup“
- Tag.next -> Tag.next_element
- Tag.previous -> Tag.previous_element

16.1.3 生成器

将下列生成器按照 PEP8 标准重新命名, 并转换成对象的属性:

- childGenerator() -> children
- nextGenerator() -> next_elements
- nextSiblingGenerator() -> next_siblings
- previousGenerator() -> previous_elements
- previousSiblingGenerator() -> previous_siblings
- recursiveChildGenerator() -> descendants
- parentGenerator() -> parents

所以迁移到 BS4 版本时要替换这些代码:

```
for parent in tag.parentGenerator():
    ...
```

替换为:

```
for parent in tag.parents:
    ...
```

(两种调用方法现在都能使用)

BS3 中有的生成器循环结束后会返回 None 然后结束. 这是个 bug. 新版生成器不再返回 None .

BS4 中增加了 2 个新的生成器, `.strings` 和 `stripped_strings`. `.strings` 生成器返回 `NavigableString` 对象, `.stripped_strings` 方法返回去除前后空白的 Python 的 `string` 对象.

16.1.4 XML

BS4 中移除了解析 XML 的 `BeautifulStoneSoup` 类. 如果要解析一段 XML 文档, 使用 `BeautifulSoup` 构造方法并在第二个参数设置为 “xml”. 同时 `BeautifulSoup` 构造方法也不再识别 `isHTML` 参数.

`Beautiful Soup` 处理 XML 空标签的方法升级了. 旧版本中解析 XML 时必须指明哪个标签是空标签. 构造方法的 `selfClosingTags` 参数已经不再使用. 新版 `Beautiful Soup` 将所有空标签解析为空元素, 如果向空元素中添加子节点, 那么这个元素就不再是空元素了.

16.1.5 实体

HTML 或 XML 实体都会被解析成 `Unicode` 字符, `Beautiful Soup 3` 版本中有很多处理实体的方法, 在新版中都被移除了. `BeautifulSoup` 构造方法也不再接受 `smartQuotesTo` 或 `convertEntities` 参数. `编码自动检测` 方法依然有 `smart_quotes_to` 参数, 但是默认会将引号转换成 `Unicode`. 内容配置项 `HTML_ENTITIES`, `XML_ENTITIES` 和 `XHTML_ENTITIES` 在新版中被移除. 因为它们代表的特性已经不再被支持.

如果在输出文档时想把 `Unicode` 字符转换成 HTML 实体, 而不是输出成 `UTF-8` 编码, 那就需要用到 `输出格式` 的方法.

16.1.6 迁移杂项

`Tag.string` 属性现在是一个递归操作. 如果 A 标签只包含了一个 B 标签, 那么 A 标签的 `.string` 属性值与 B 标签的 `.string` 属性值相同.

多值属性 比如 `class` 属性包含一个他们的值的列表, 而不是一个字符串. 这可能会影响到如何按照 `CSS` 类名搜索 `tag`.

如果使用 `find*` 方法时同时传入了 `string` 参数 和 `name` 参数. `Beautiful Soup` 会搜索指定 `name` 的 `tag`, 并且这个 `tag` 的 `Tag.string` 属性包含 `text` 参数的内容. 结果中不会包含字符串本身. 旧版本中 `Beautiful Soup` 会忽略掉 `tag` 参数, 只搜索 `text` 参数.

`BeautifulSoup` 构造方法不再支持 `markupMassage` 参数. 现在由解析器负责文档的解析正确性.

很少被用到的几个解析器方法在新版中被移除, 比如 `ICantBelieveItsBeautifulSoup` 和 `BeautifulSOAP`. 现在由解析器完全负责如何解释模糊不清的文档标记.

`prettify()` 方法在新版中返回 `Unicode` 字符串, 不再返回字节流.

附录

原文: <http://www.crummy.com/software/BeautifulSoup/bs4/doc/>

翻译: delong

查看 BeautifulSoup3 文档