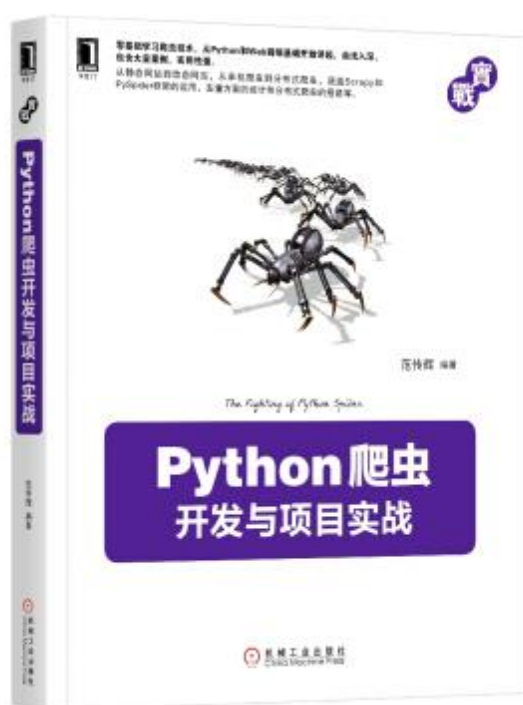


---

# 《Python 爬虫开发与项目实战》

## 试读章节



购买链接:<https://item.jd.com/12206762.html>

## 第 4 章 HTML 解析大法

HTML 网页数据解析提取是 Python 爬虫开发中非常关键的一步。HTML 网页的解析提取有很多种方式，本章主要从三个方面进行讲解，分别为 Firebug 工具的使用、正则表达式和 Beautiful soup，基本上涵盖了 HTML 网页数据解析提取的方方面面。

### 4.1 初识 Firebug

Firebug 是一个用于 Web 前端开发的工具，它是 FireFox 浏览器的一个扩展插件。它可以用于调试 JavaScript、查看 DOM、分析 CSS、监控网络流量以及进行 Ajax 交互等。它提供了几乎前端开发需要的全部功能，因此在 Python 爬虫开发中非常有用，尤其是在分析协议和分析动态网站的时候，之后我们所有的分析场景都是基于这个工具，基于

Firefox 浏览器。Firebug 面板如图 4.1 所示：



图 4.1 Firebug 面板

大家如果之前用过 Firebug 会发现在面板上，多了一个 FirePath 的选项。FirePath 是 Firebug 上的一个扩展插件，它的功能主要是帮助我们精确定位网页中的元素，生成 XPath 或者是 CSS 查找路径表达式，在 Python 爬虫开发中抽取网页元素非常便利，省去了手写 XPath 和 CSS 路径表达式的麻烦。FirePath 选项面板内容如图 4.2 所示：

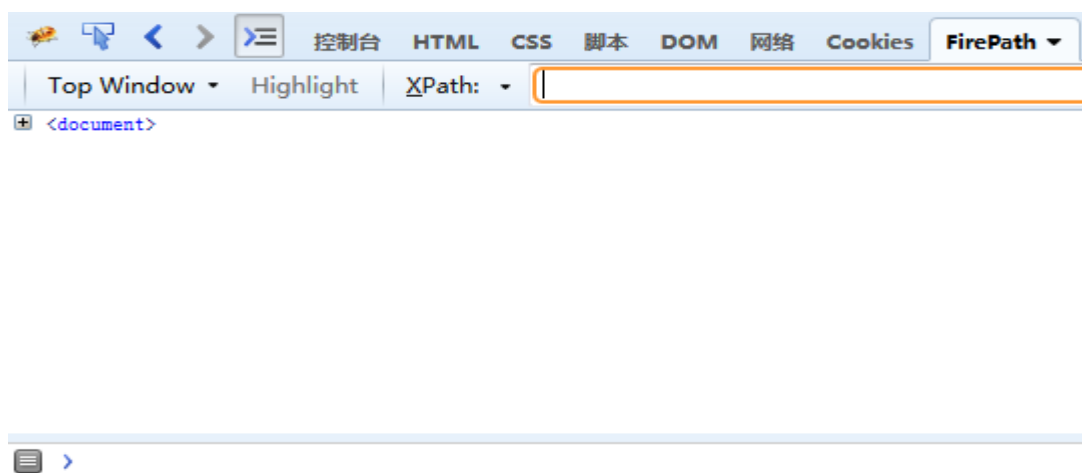



图 4.2 FirePath 面板

### 4.1.1 安装 Firebug

由于 Firebug 是 Firefox 浏览器的一个扩展插件，所以首先需要下载 Firefox 浏览器。读者可以访问 [www.mozilla.com](http://www.mozilla.com) 下载并安装 Firefox 浏览器。安装完成后用它访问 <https://addons.mozilla.org/zh-CN/firefox/collections/mozilla/webdeveloper/> 进入如图 4.3 所示页面。点击"添加到 Firefox"，然后点击"立即安装"，最后重新启动 Firefox 浏览器即可完成安装。

此收藏集中有 12 个附加组件

排序依照： 热门度 ▼

**Firebug**  
作者：Joe Hewitt, Jan Odvarko, 等

Firebug 为你的 Firefox 集成了浏览网页的同时随手可得丰富开发工具。你可以对任何网页的 CSS、HTML 和 JavaScript 进行实时编辑、调试和监控...  
Firebug 1.4 仅支持 Firefox 3.0 或更高版本。

[+ 添加到 Firefox](#)

精选  
★★★★★ 1,843 条评论  
2,003,645 个用户

图 4.3 Firebug 下载页面

Firebug 安装完成后，为了扩展 Firebug 在路径选择上的功能，还需要安装 Firebug 的插件 FirePath。打开火狐浏览器->设置->附件组件->搜索->输入 firepath，如图 4.4 所示：

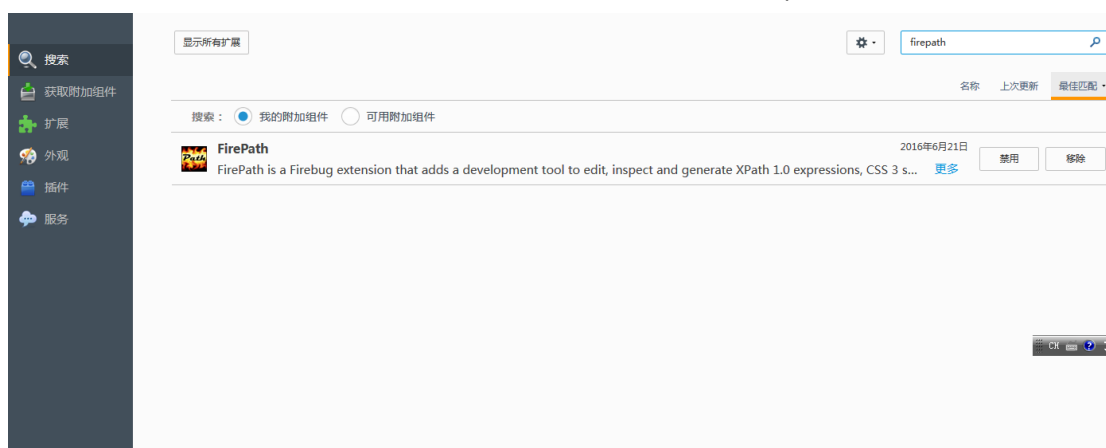


图 4.4 FirePath 安装

当然也可以使用这种方式安装 Firebug.

## 4.1.2 强大的功能

安装完 Firebug 后，开始讲解 Firebug 的强大的功能。接下来按照主面板、子面板的顺序说明 Firebug 的功能，相信大家会被 Firebug 所吸引。

### 1. 主面板

安装完成之后，在 Firefox 浏览器的地址后方就会有一个小虫子的图标，这是 Firebug 启动开关，如图 4.5 所示：



图 4.5 Firebug 启动开关

单击该图标后即可展开 Firebug 的控制台，也可以通过快捷键<F12>来打开控制台，默认位于 Firefox 浏览器底部。使用 Ctrl+F12 快捷键可以使 Firebug 独立打开一个窗口而不占用 Firefox 页面底部的空间，如图 4.6 所示：

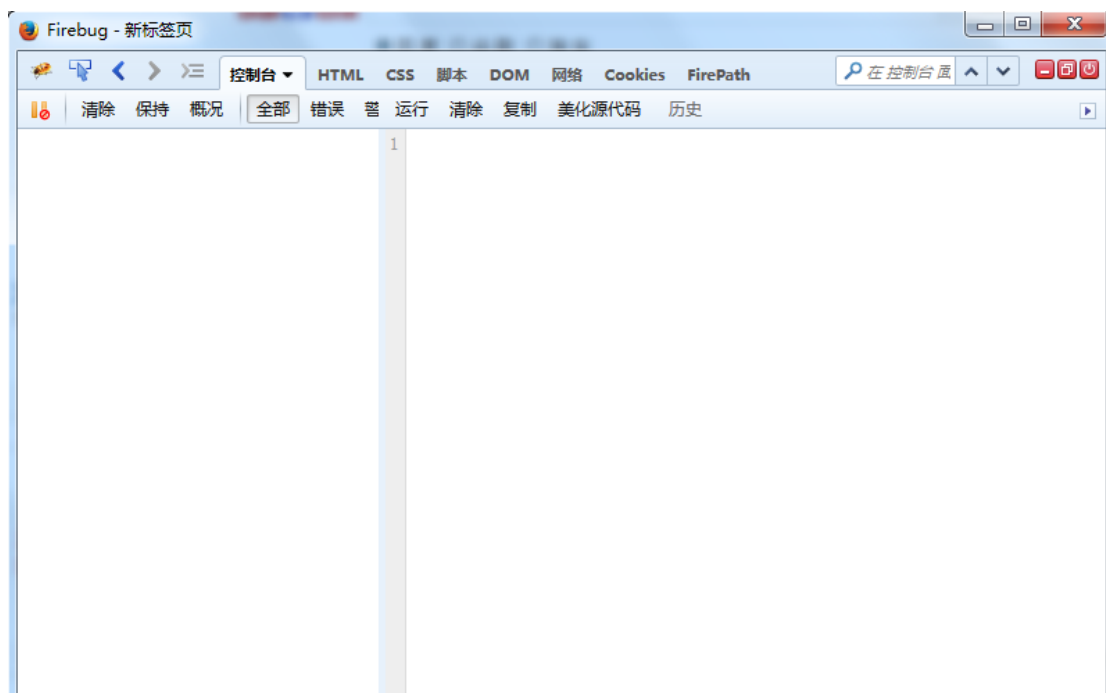


图 4.6 Firebug 主面板

从上图中可以看出，Firebug 包括 8 个子面板：

- 控制台面板：用于记录日志、概览、错误提示和执行命令行，同时也用于 Ajax 的调试
- HTML 面板：用于查看 HTML 元素，可以实时地编辑 HTML 和改变 CSS 样式，它包括 3 个子面板，分别是样式、计算出的样式、布局、DOM 和事件面板
- CSS 面板：用于查看所有页面上的 CSS 文件，可以动态地修改 CSS 样式，由于 HTML 面板中已经包含了样式面板，因此该面板将很少用到
- 脚本面板：用于显示 Javascript 文件及其所在的页面，也可以用来显示 Javascript 的 Debug 调试，包含 3 个子面板，分别是监控、堆栈和断点
- DOM 面板：用于显示页面上的所有对象
- 网络面板：用于监视网络活动，可以帮助查看一个页面的载入情况，包括文件下载所占用的时间和文件下载出错等信息，也可以用于监视 Ajax 行为。在分析网络请求和动态网站加载时非常有用。
- Cookies 面板：用于查看和调整 cookie
- FirePath 面板：用于精确定位网页中的元素，生成 XPath 或者是 CSS 查找路径表达式

## 2. 控制台面板

控制台面板面板可以用于记录日志，也可以用于输入脚本的命令行。Firebug 提供如下几个常用的记录日志的函数：

- `console.log`：简单的记录日志
- `console.debug`：记录调试信息，并且附上行号的超链接
- `console.error`：在消息前显示错误图标，并且附上行号的超链接

- `console.info`: 在消息前显示消息图标，并且附上行号的超链接
  - `console.warn`: 在纤细钱显示警告图标，并且附行号的超链接
- 新建一个 `html` 页面中，向`<body>`标签中加入`<script>`标签，代码如下：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
    "http://www.w3.org/TR/html4/strict.dtd">

<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Firefox 测试</title>
</head>
<body>
<script type="text/javascript">
var a = "Python";
var b = "爬虫开发";
document.write(a,b);//网页上输出内容
console.log(a + b);
console.debug(a + b);
console.error(a + b);
console.info(a + b);
console.warn(a + b);
</script>
</body>
</html>
```

在 `Firefox` 浏览器中开启 `Firebug` 并运行此 `HTML` 文档，效果如图 4.7 所示：

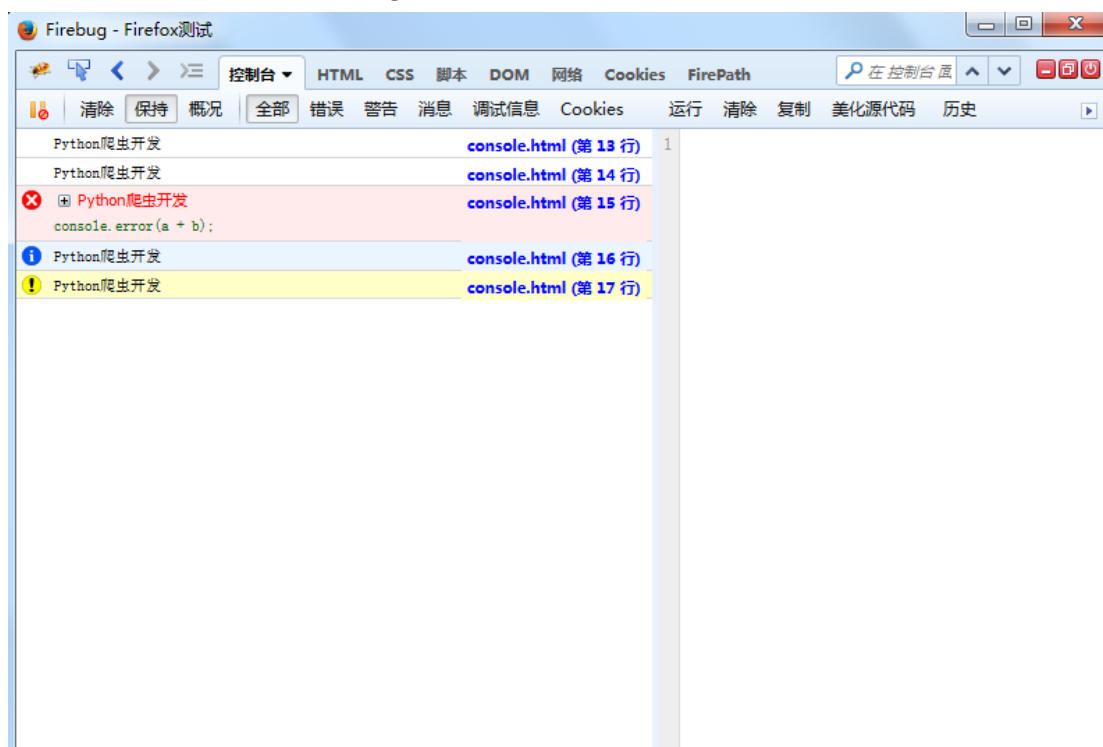


图 4.7 Firebug 控制台输出结果

也可以直接在右侧输入 `javascript` 代码执行，同时可以对输入的源代码格式进行美化，示例如图 4.8 所示：

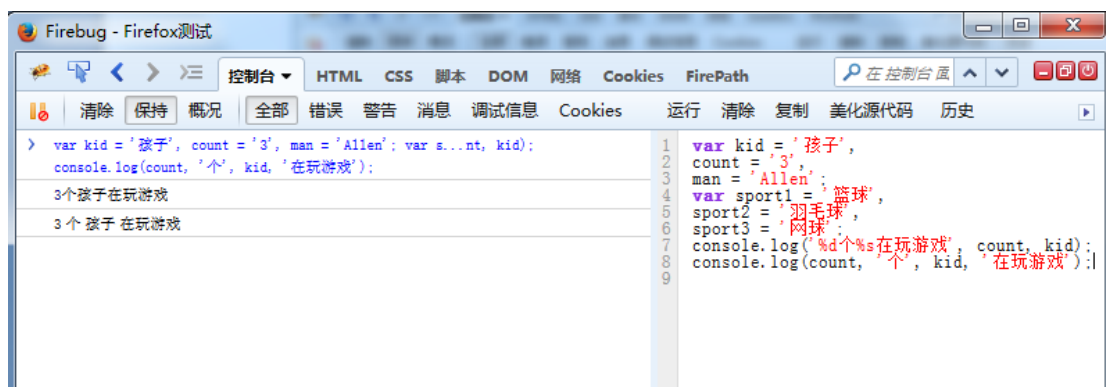


图 4.8 Firebug js 脚本执行

控制台面板内有一排子菜单，分别是清除、保持、概况、全部等。

- “清除”用于清除控制台中的内容。
- “保持”则是把控制台中的内容保存，即使刷新了依然存在。
- “全部”则是显示全部的信息。
- “概况”菜单用于查看函数的性能。
- 后面的“错误”、“警告”、“消息”、“调试信息”、“Cookies”菜单则是对所有的信息进行了分类。

控制台面板还可以进行 Ajax 调试。例如我打开一个页面，可以在 Firebug 控制台查看到本次 Ajax 的 Http 请求头信息和服务器响应头信息。首先在 Firefox 浏览器中开启 Firebug，并访问百度的首页，可以看到图 4.9 的效果：

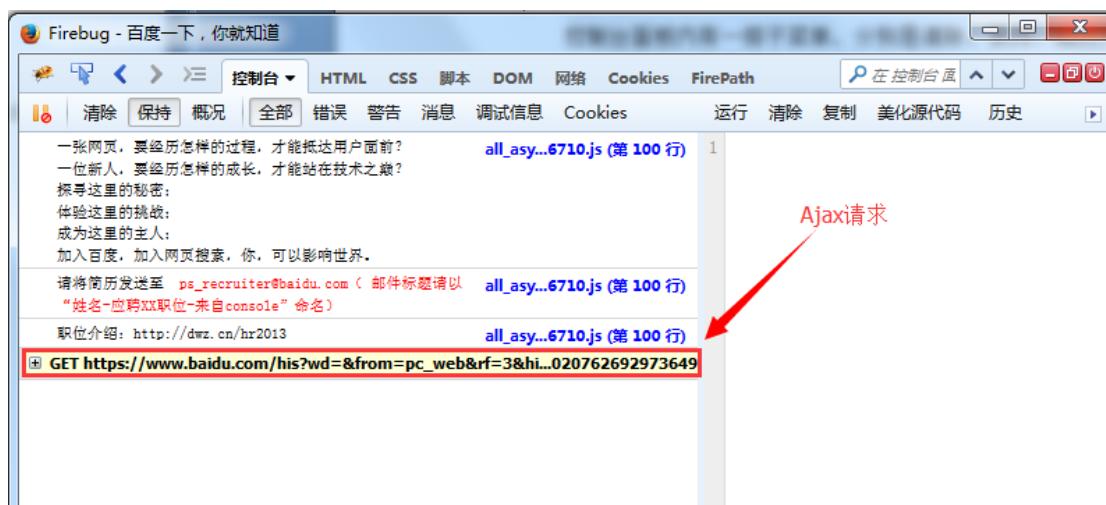


图 4.9 Ajax 请求

如果你没有上图的效果，可以在控制台的下拉菜单中，选中显示 XMLHttpRequests，如图 4.10 所示：

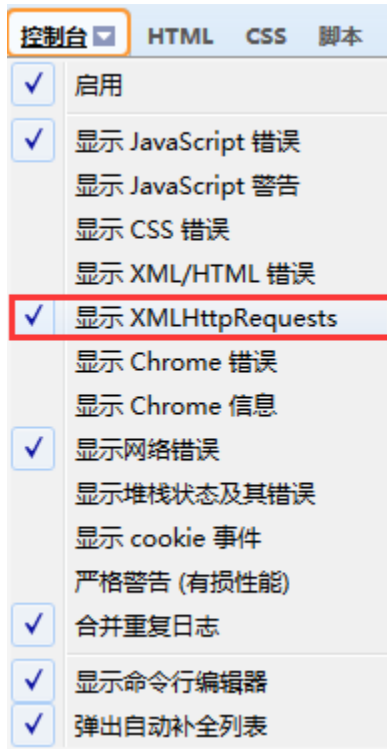


图 4.10 显示 XMLHttpRequests

### 3. HTML 面板

Html 面板的强大之处就是能查看和修改 HTML 代码，而且这些代码都是经过格式化的。以百度首页为例子进行讲解，在 HTML 控制台的左侧可以看到整个页面当前的文档结构，可以通过单击“+”来展开。当单击相应的元素时，右侧面板中就会显示出当前元素的样式、布局以及 DOM 信息。效果如图 4.11 所示：

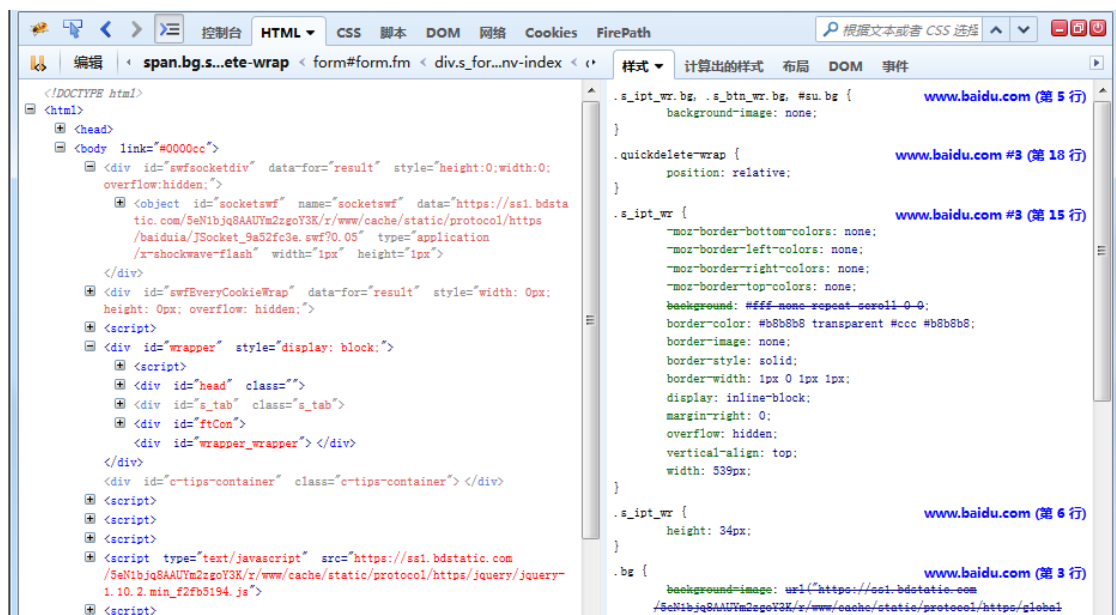


图 4.11 百度首页 HTML 结构

而当光标移动到 HTML 树中相应元素上时,上面页面中相应的元素将会被高亮显示,高亮部分我用红框圈起来了。如图 4.12 所示:

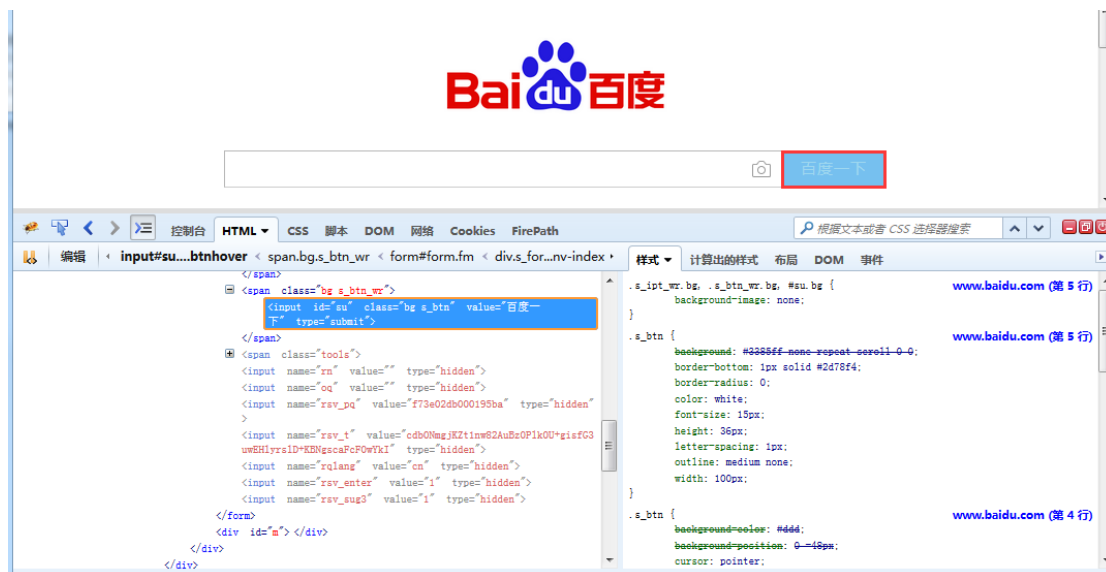


图 4.12 百度一下高亮显示

还有一种更快更常用的查找 HTML 元素的方法。利用查看(Inspect)功能,可以快速地寻找到某个元素的 HTML 结构,红色框圈起来的就是 Inspect 按钮。如图 4.13 所示:



图 4.13 Inspect 按钮查看元素

当单击"Inspect"按钮后,用鼠标在网页上选中一个元素时,元素会被一个蓝色的框框住,同时下面的 HTML 面板中相应的 HTML 树也会展开并且高亮显示,再次单击后即可退出该模式。通过这个功能,可以快速寻找页面内的元素,调试和查找相应代码非常



方便。

之前讲的都是查看 HTML，还可以修改 HTML 内容和样式。例如，修改百度首页的百度一下按钮文字为搜索一下，将 input 标签中的 value 值改为搜索一下，如图 4.14 所示：

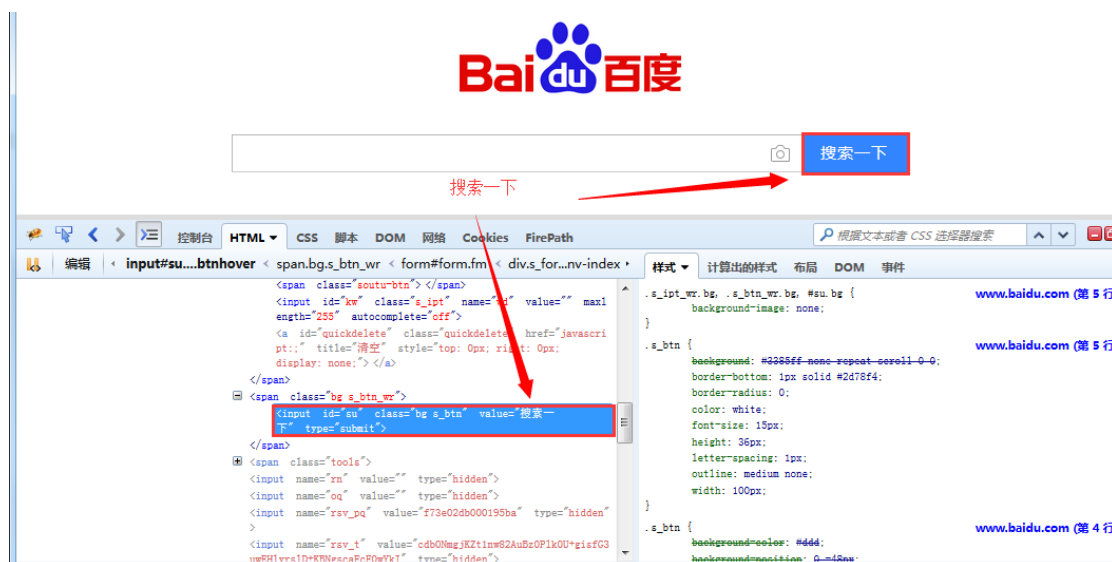


图 4.14 修改 HTML 元素值

在这个基础上，修改一下样式，将 background 值改为 red,搜索一下的背景立即变成了红色，效果如图 4.15 所示：

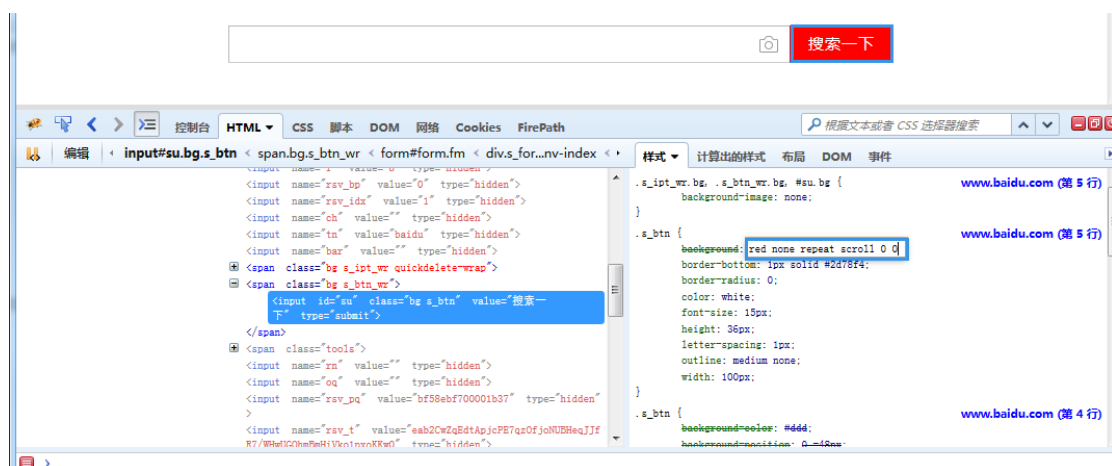


图 4.15 修改 HTML 样式

## 4. 网络面板

在 Python 爬虫开发中，网络面板比较常用，能够监听网络访问请求与响应，在分析异步加载请求时非常有用。例如访问百度首页，效果如图 4.16 所示：

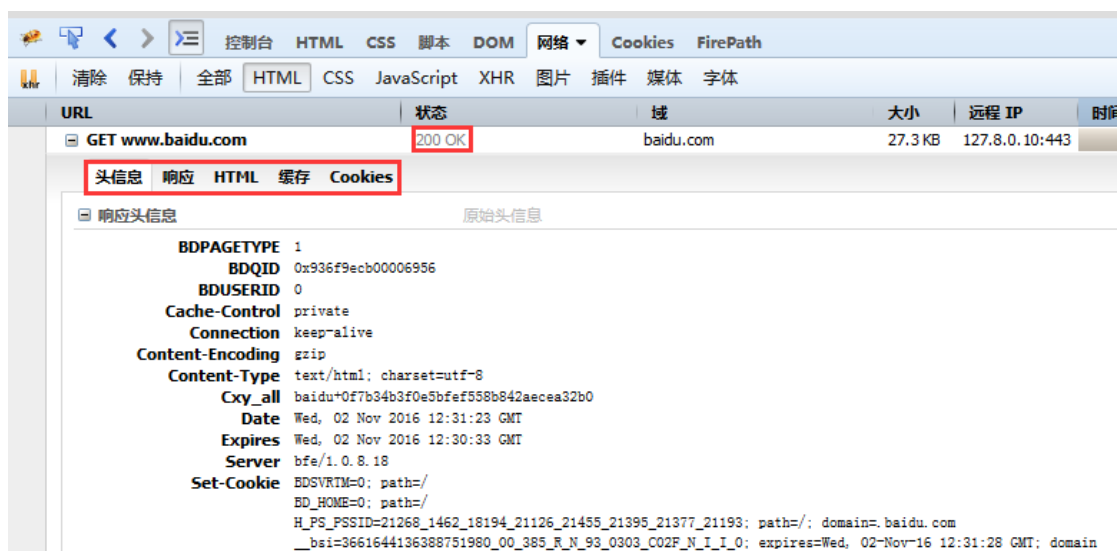


图 4.16 网络请求

红色的框中可以看到网络访问的头信息、响应码、响应内容和 Cookies 都能得到有效的记录。

在网络面板的子菜单中又分为 HTML、CSS、JavaScript、XHR、图片等选项，其实只是将所有的网络访问进行了分类划分。

## 5. 脚本面板

脚本面板不仅可以查看页面内的脚本，而且还有强大的调试功能。在脚本面板的右侧有“监控”、“堆栈”和“断点”三个面板，利用 Firebug 提供的设置断点的功能，可以很方便地调试程序，还可以将 javascript 脚本格式化，方便阅读源码分析。如图 4.17 所示：

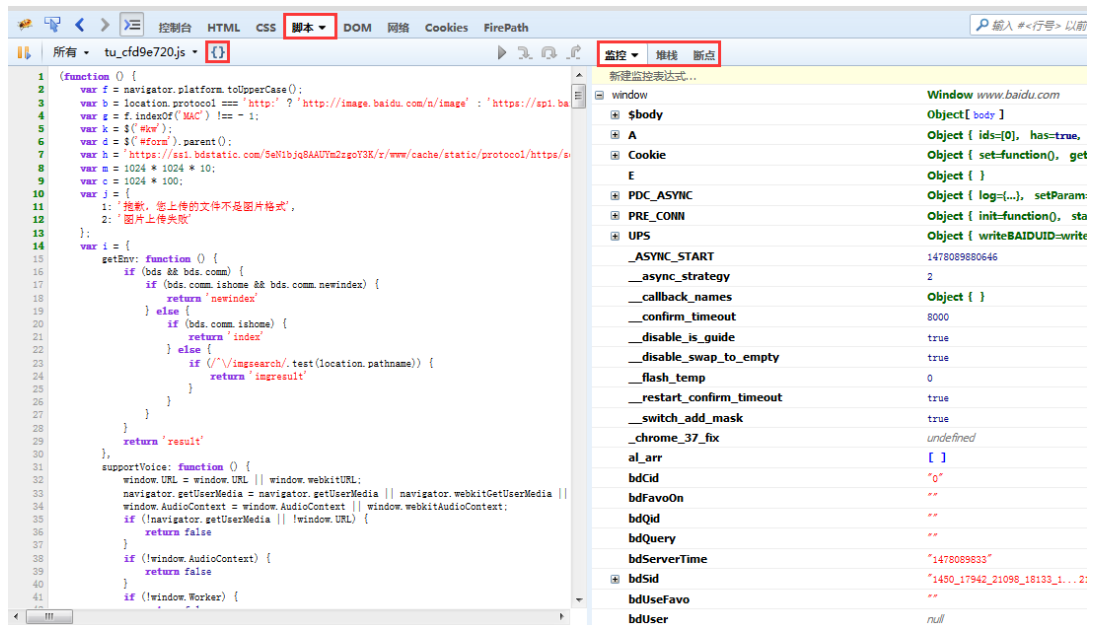


图 4.17 脚本面板

接下来测试一下脚本面板的断点调试功能，以 jsTest.html 文件为例，代码如下：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8">
    <script type="text/javascript">
        function doLogin(){
            var msg = document.getElementById('message');
            var username = document.getElementById('username');
            var password = document.getElementById('password');
            arrs=[1,2,3,4,5,6,7,8,9];
            for(var arr in arrs){
                msg.innerHTML+=arr+"<br />"
                msg.innerHTML+="username->"+username.value
                    +"password->"+password.value+"<br />"
            }
        }
    </script>
</head>
<body>
    <div>
        <input id="username" type="text" placeholder="用户名" value=""/>
        <br/>
        <input id="password" type="text" placeholder="密码" value=""/>
        <br/>
        <input type="button" value="login" onClick="doLogin();"/>
        <br/>
        <div id="message"></div>
    </div>
</body>
</html>
```

运行代码后可以看到如图 4.18 所示的效果。图中加粗并有绿色的行号表示此处为 JavaScript 代码，可以在此处设置断点。比如在第 8 行这句代码前面单击一下，那它前面就会出现一个红褐色的圆点，表示此处已经被设置了断点。此时，在右侧断点面板的断点列表中就出现了刚才设置的断点。如果想暂时禁用某个断点，可以在断点列表中去掉某个断点的前面的复选框中的勾，那么此时左侧面板中相应的断点就从红褐色变成了红灰褐色了。

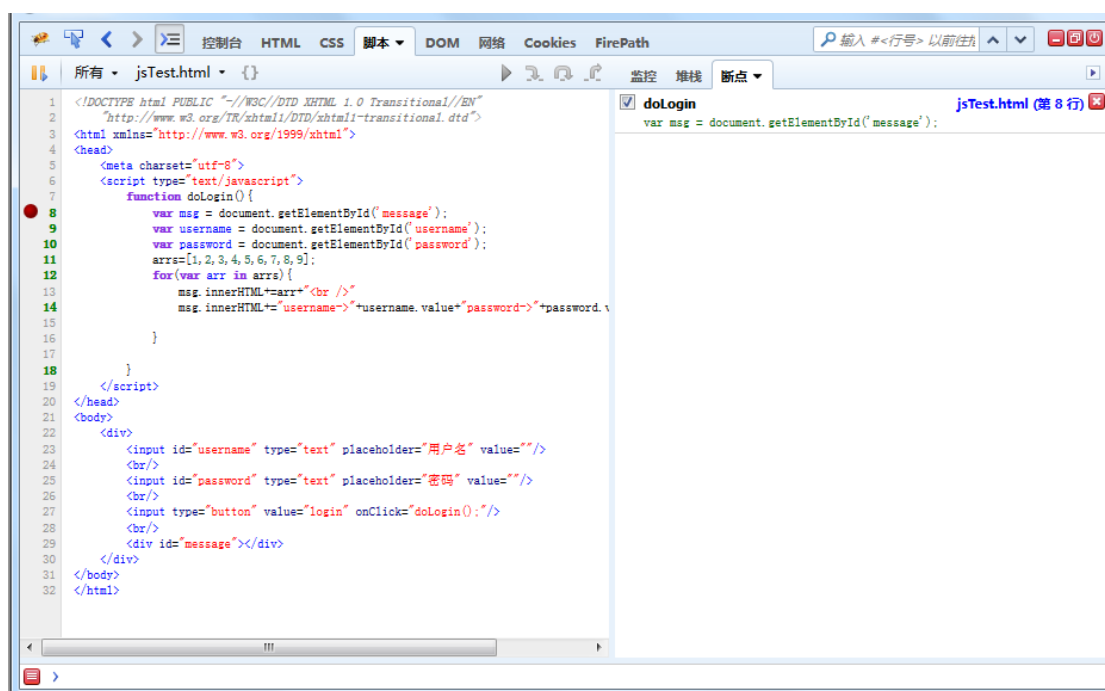


图 4.18 断点设置

设置完断点之后，我们就可以调试程序了。单击页面中的“login”按钮，可以看到脚本停止在用淡黄色底色标出的那一行上。此时用鼠标移动到某个变量上即可显示此时这个变量的 value。显示效果如图 4.19 所示：

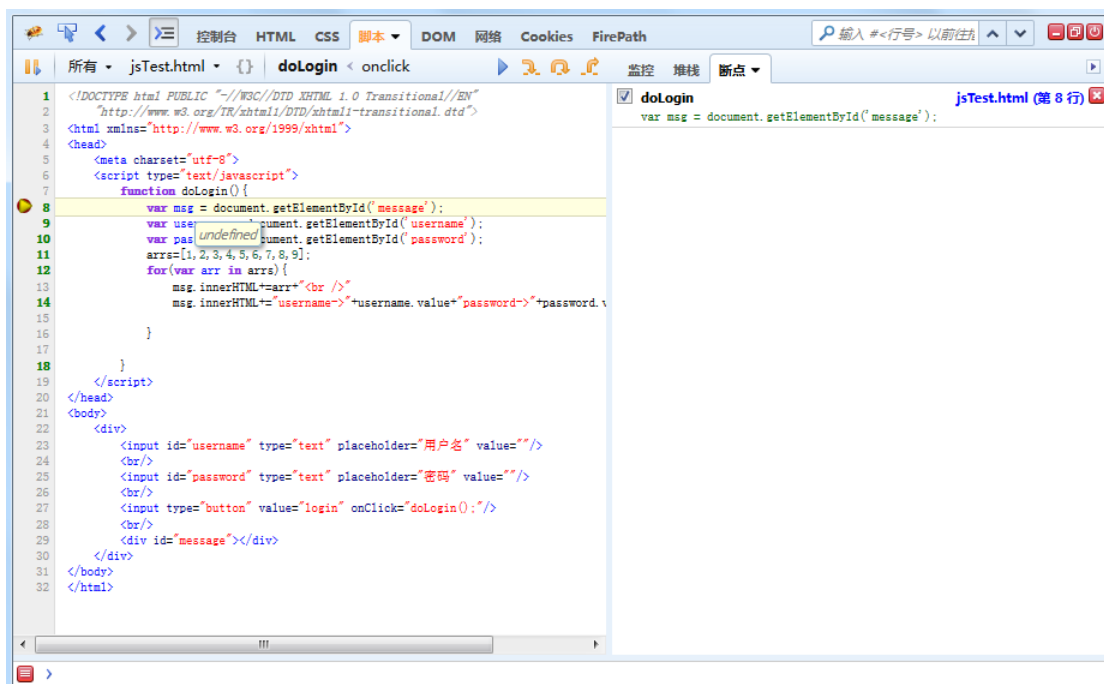



图 4.19 断点调试

此时 JavaScript 内容上方的  四个按钮已经变得可用了。它们分别代表“继续执行”、“单步进入”、“单步跳过”和“单步退出”。可以使用快捷键进行操作：

- 继续执行<F8>：当通过断点来停止执行脚本时，单击<F8>就会恢复执行脚本
- 单步进入<F11>：允许跳到页面中的其他函数内部
- 单步跳过<F10>：单击<F10>来直接跳过函数的调用即跳到 return 之后
- 单步退出<shift+F11>：允许恢复脚本的执行，直到下一个断点为止

单击“单步进入”按钮，代码会跳到下一行，当鼠标移动到“msg”变量上时，就可以显示出它的内容是一个 DOM 元素，即“div#message”。将右侧面板切换到“监控”面板，这里列出了几个变量，包括“this”指针的指向以及“msg”变量。单击“+”可以看到详细的信息。如图 4.20 所示：

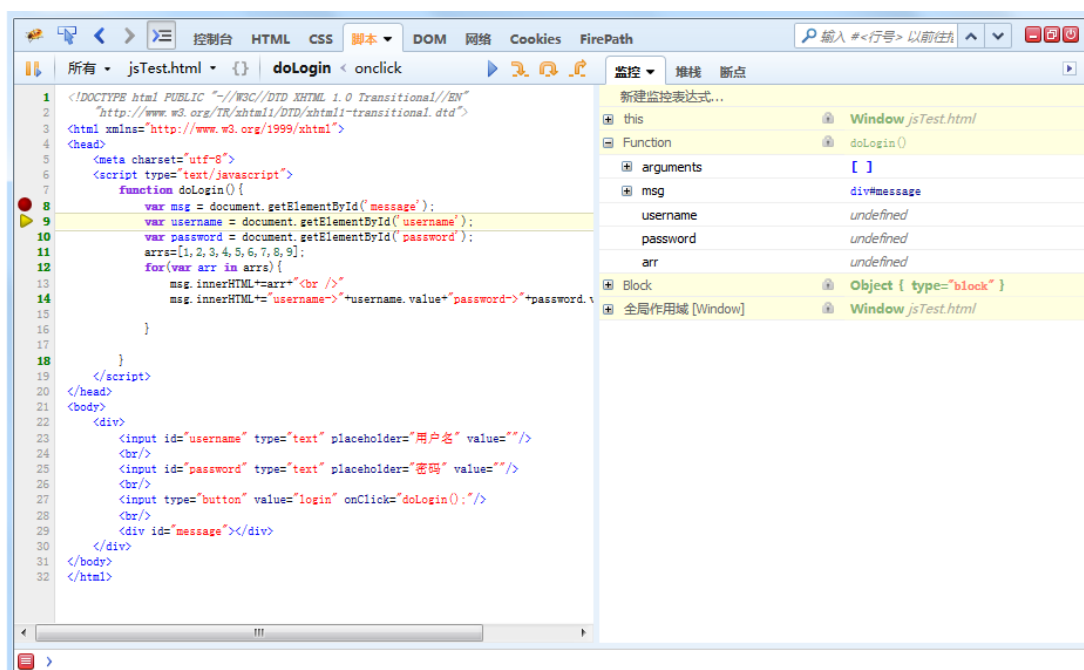


图 4.20 单步调试

以上设置的都是静态断点，脚本面板还提供了条件断点的高级功能。在要调试的代码前面的序号上单击鼠标右键，就可以出现设置条件断点的输入框。在该框内输入“arr==5”，然后回车确认，显示效果如图 4.21 所示：

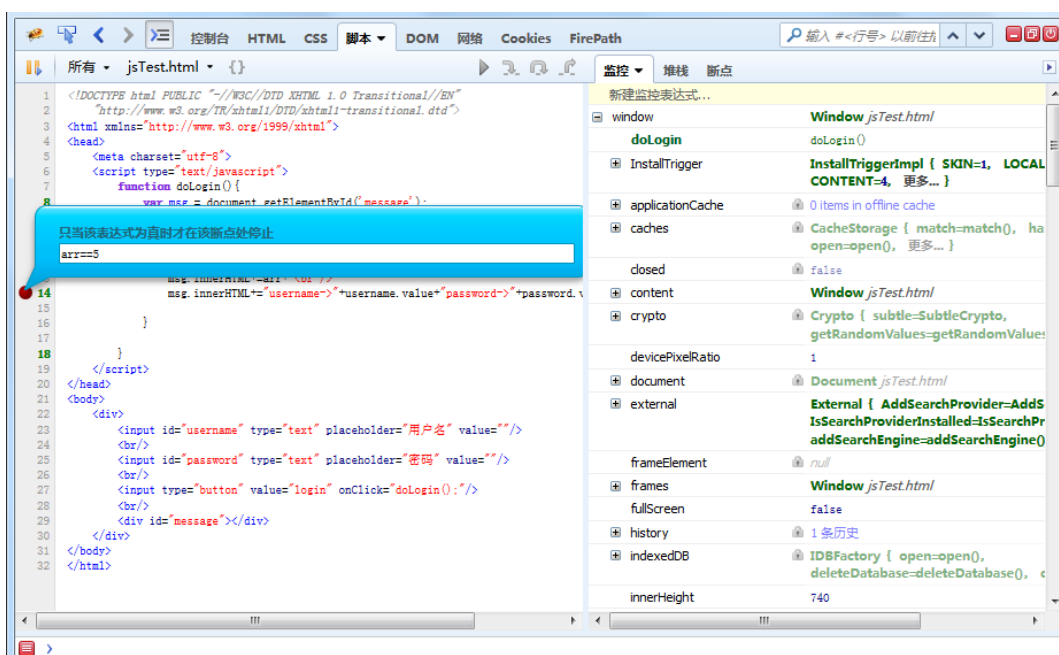


图 4.20 条件调试

最后单击页面的“login”按钮。可以发现，脚本在“arr==5”这个表达式为真时停下了。

## 6. FirePath 面板

切换到 FirePath 面板，通过查看(Inspect)按钮，点击百度一下按钮，XPath 后面的输

出框中出现 XPath 路径表达式，这在 Python 爬虫开发中非常有用。如图 4.21 所示：



图 4.21 FirePath 面板

## 4.2 正则表达式

在编写处理网页文本的程序时，经常会有查找符合某些复杂规则的字符串的需要。正则表达式就是用于描述这些规则的工具。正则表达式是由普通字符（例如字符 a 到 z）以及特殊字符（称为“元字符”）组成的文字模式。模式描述在搜索文本时要匹配的一个或多个字符串。正则表达式作为一个模板，将某个字符模式与所搜索的字符串进行匹配。

### 4.2.1 基本语法与使用

正则表达式功能非常强大，但是学好并不是很困难。一些初学者总是感觉到正则表达式很抽象，看到稍微长的表达式直接选择放弃。接下来从一个新手的角度，由浅及深，配合各种示例来讲解正则表达式的用法。

#### 1. 入门小例子

学习正则表达式最好的办法就是通过例子。在不断解决问题的过程中，就会不断理解正则表达式构造方法的灵活多变。

例如我们想找到一篇英文文献中所有的 **we** 单词，你可以使用正则表达式：**we**，这是最简单的正则表达式，可以精确匹配英文文献中的 **we** 单词。正则表达式工具一般可以设置为忽略大小写，那 **we** 这个正则表达式可以将文献中的 **We**、**wE**、**we** 和 **WE** 都匹配出来。如果仅仅使用 **we** 来匹配，会发现得出来的结果和预想的不一样，匹配多了，类似于 **well**、**welcome** 这样的单词也会被匹配出来，因为这些单词中也包含 **we**。如何仅

仅是将 **we** 单词匹配出来呢？我们需要使用这样的正则表达式：**\bwe\b**。

**\b** 是正则表达式规定的一个特殊代码，被称为元字符，代表着单词的开头或结尾，也就是单词的分界处，它不代表着英语中空格、标点符号、换行等单词分隔符，只是用来匹配一个位置，这种理解方式很关键。

假如我们看到 **we** 单词不远处有一个 **work** 单词，想把 **we**、**work** 和它们之间的所有内容都匹配出来，那么我们需要了解另外两个元字符 **.** 和 **\***，正则表达式可以写为 **\bwe\b.\*\bwork\b**。**.** 这个元字符的含义是匹配除了换行符的任意字符，**\*** 元字符不是代表着字符，而是代表着数量，含义是 **\*** 前面的内容可以连续重复任意次使得整个表达式被匹配。**.\***整体的意思就非常明显了，代表着可以匹配任意数量不换行的字符，

那么**\bwe\b.\*\bwork\b** 作用就是先匹配出 **we** 单词，接着在匹配任意的字符(非换行)，直到匹配到 **work** 单词结束。通过上面的例子，我们看到元字符在正则表达式中非常关键，元字符的组合能构造出强大的功能，接下来咱们开始讲解常用的元字符。在讲解之前，需要介绍一个正则表达式的测试工具 **Match Tracer**，这个工具可以将写的正则表达式生成树状结构，描述并高亮每一部分的语法，同时可以检验正则表达式写的是否正确。如图 4.22 所示：

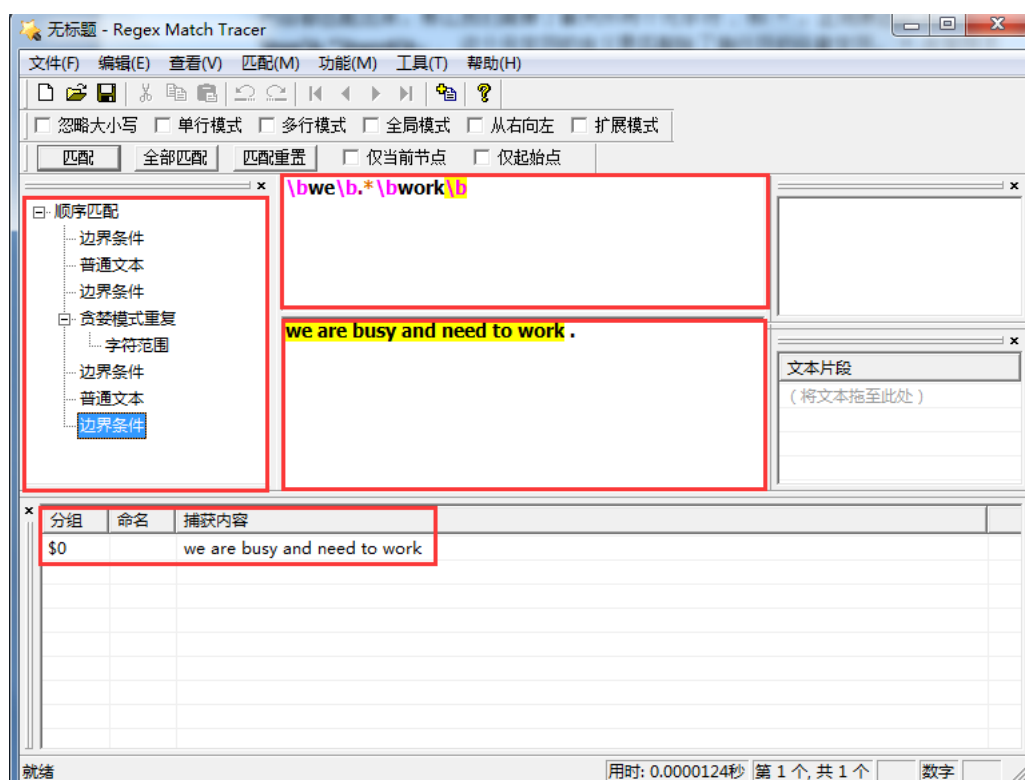


图 4.22 Match Tracer

## 2. 常用元字符

元字符主要有四种作用：有的用来匹配字符，有的用来匹配位置，有的用来匹配数量，有的用来匹配模式。在上面的例子中，我们讲到了 **.** **\*** 这两个元字符，还有其他



的元字符，如表 4-1 所示：

表 4-1 常见元字符

元字符	含义
.	匹配除换行符以外的任意字符
\b	匹配单词的开始或结束
\d	匹配数字
\w	匹配字母或数字或下划线或汉字
\s	匹配任意空白符，包括 空格，制表符(Tab)，换行符，中文全角空格等
^	匹配字符串的开始
\$	匹配字符串的结束

上面的元字符是用来匹配字符和位置的，接下来讲解其他功能时，会依次列出匹配数量和模式的元字符。对上面列出的元字符使用一些小例子来进行一下练习。

假如一行文本为: **we are still studying and so busy**，我们想匹配出所有以 s 开头的单词，那么正则表达式可以写为: **\bs\w\*\b**。**\bs\w\*\b** 的匹配顺序：先是某个单词开始处 (\b)，然后是字母 s,然后是任意数量的字母或数字(\w\*)，最后是单词结束处(\b)。同理，如果匹配 **s100** 这样的字符串（不是单词），需要用到^和\$，一个匹配开头，一个匹配结束，可以写为**^s\d\*\$**。

### 3.字符转义

如果你想查找元字符本身的话，比如你查找 . 或者 \* 就会出现出了问题，因为它们具有特定功能，没办法把它们指定为普通字符。这个时候就需要用到转义，使用\来取消这些字符的特殊意义。因此如果查找 . 、 \或者 \* 时，必须写成\. 、\\ 和 \\*。例如匹配 **www.google.com** 这个网址时，可以表达式可以写为 **www\\.google\\.com**。

### 4.重复

首先将匹配重复的限定符（指定数量的代码）列举一下，如表 4-2 所示：

表 4-2 常用限定符

限定符	含义
*	重复零次或更多次
+	重复一次或更多次
?	重复零次或一次

{n}	重复 n 次
{n,}	重复 n 次或更多次
{n,m}	重复 n 到 m 次

下面是一些重复的例子：

- **hello\d+** :匹配 hello 后面跟 1 个或更多数字，例如可以匹配 hello1、hello10 等情况。
- **^\d{5,12}\$** :匹配 5 到 12 个数字的字符串，例如 QQ 号符合要求。
- **we\d?** : 匹配 we 后面跟 0 个或者一个数字，例如 we、we0 符合情况。

## 5. 字符集合

通过上面介绍的元字符，可以看到查找数字，字母或数字，空白是很简单的，因为已经有了对应这些字符集合，但是如果匹配没有预定义元字符的字符集合，例如匹配 a、b、c、d 和 e 中任意一个字符，这时候就需要自定义字符集合。正则表达式是通过 [] 来实现，[abcde] 就是匹配 abcde 中的任意一个字符，[.?!] 匹配标点符号(.或?或!)。

除了将需要自定义的字符都写入[]中，还可以指定一个字符范围。**[0-9]**代表的含义与\d 是完全一致的，代表一位数字；**[a-zA-Z\_]**也完全等同于\w（只考虑英文），代表着 26 个字母中的大小写、0-9 数字和下划线中的任一个字符。

## 6. 分枝条件

正则表达式里的分枝条件指的是有几种匹配规则，如果满足其中任意一种规则都应该当成匹配，具体方法是用 | 把不同的规则分隔开。例如匹配电话号码，电话号码中一种是 3 位区号，8 位本地号，形如 010-11223344，另一种是 4 位区号，7 位本地号，形如 0321-1234567。如果想把电话号码匹配出来，就需要用到分支条件：**0\d{2}-\d{8}|0\d{3}-\d{7}**。在分支条件中有一点需要注意，匹配分枝条件时，将会从左到右地测试每个条件，如果满足了某个分枝的话，就不会再去管其它的条件了，是一种或的关系，例如从 1234567890 匹配出连续的 4 个数字或者连续 8 个数字，如果写成**\d{4}|\d{8}**，其实\d{8}是失效的，既然能匹配出来 8 位数字，肯定就能匹配出 4 位数字。

## 7. 分组

先以简单的 IP 地址匹配为例子，想匹配类似 192.168.1.1 这样的 IP 地址，可以这样写正则表达式**((\d{1,3})\.)\d{1,3}**。下面分析一下这个正则表达式：**\d{1,3}**代表着 1-3 位的数字，**((\d{1,3})\.)\d{1,3}**代表着将 1-3 位数字加上一个 . 重复 3 次，匹配出类似 192.168.1.这部分，之后再加上**\d{1,3}** 1-3 位的数字。但是上述的正则表达式会匹配出

类似 333.444.555.666 这些不可能存在的 IP 地址，因为 IP 地址中每个数字都不能大于 255，所以要写出一个完整的 IP 地址匹配表达式，还需要关注一下细节，下面给出一个使用分组完整的 IP 表达式：((25[0-5]|2[0-4]\d|[0-1]\d{2}|[1-9]?\d)\.){3}((25[0-5]|2[0-4]\d|[0-1]\d{2}|[1-9]?\d))。其中关键是(25[0-5]|2[0-4]\d|[0-1]\d{2}|[1-9]?\d)部分，大家应该有能力分析出来。

8. 反义

有时需要查找除某一类字符集合之外的字符。比如想查找除了数字以外，其它任意字符都行的情况，这时需要用到反义，如表 4-3 所示：

表 4-3 常用的反义

代码	含义
\W	匹配任意不是字母，数字，下划线，汉字的字符
\S	匹配任意不是空白符的字符
\D	匹配任意非数字的字符
\B	匹配不是单词开头或结束的位置
[^a]	匹配除了 a 以外的任意字符
[^abcde]	匹配除了 abcde 这几个字母以外的任意字符
^(123 abc)]	匹配除了 123 或者 abc 这几个字符以外的任意字符

例如\D+匹配非数字的一个或者多个字符。

9. 后向引用

前面我们讲到了分组，使用小括号指定一个表达式就可以看做是一个分组。默认情况下，每个分组会自动拥有一个组号，规则是：从左向右，以分组的左括号为标志，第一个出现的分组的组号为 1，第二个为 2，以此类推。还是以简单的 IP 匹配表达式((\d{1,3})\.){3}\d{1,3}为例，这里面有两个分组 1,2，使用 Match Tracer 这个工具可以很明显的看出来。如图 4.23 所示：

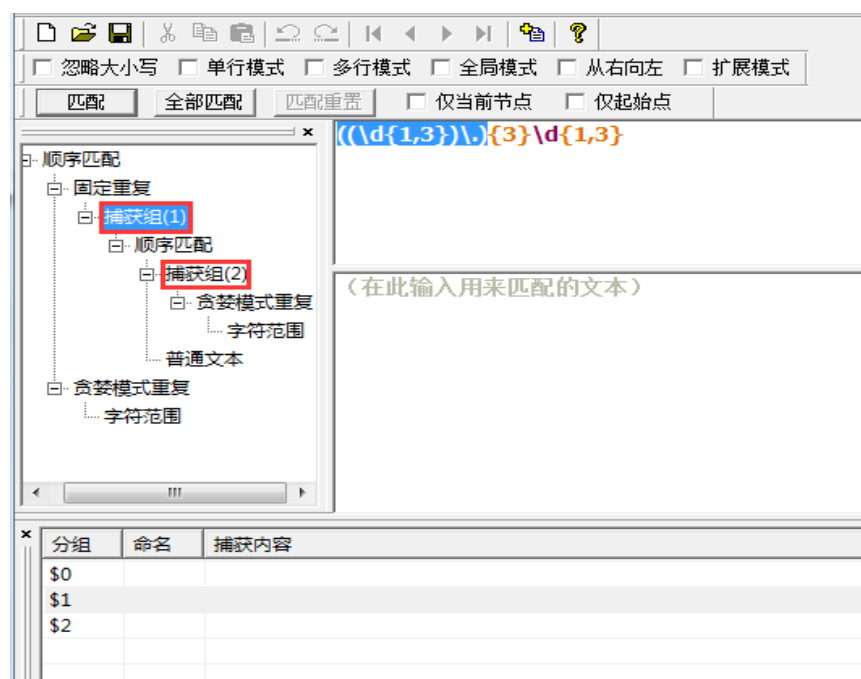


图 4.23 捕获组

所以上面的表达式可以改写成`((\d{1,3})\\.){3}\2`。

你也可以自己指定子表达式的组名。要指定一个子表达式的组名，使用这样的语法：`(?<Digit>\d+)`或者`(?'Word'\d+)`，这样就把`\d+`的组名指定为 `Digit` 了。要反向引用这个分组捕获的内容，你可以使用 `\k<Digit>`，所以上面的 IP 匹配表达式写成 `((?<Digit>\d{1,3})\\.){3}\k<Digit>`。使用小括号的地方很多，主要是用来分组，在表 4-4 中列出一些常用的形式。

表 4-4 常用分组形式

分类	语法	含义
捕获	<code>(exp)</code>	匹配 <code>exp</code> ,并捕获文本到自动命名的组里
	<code>(?&lt;name&gt;exp)</code>	匹配 <code>exp</code> ,并捕获文本到名称为 <code>name</code> 的组里，也可以写成 <code>(?'name'exp)</code>
	<code>(?:exp)</code>	匹配 <code>exp</code> ,不捕获匹配的文本，也不给此分组分配组号
零宽断言	<code>(?=exp)</code>	匹配 <code>exp</code> 前面的位置
	<code>(?&lt;=exp)</code>	匹配 <code>exp</code> 后面的位置
	<code>(?!exp)</code>	匹配后面跟的不是 <code>exp</code> 的位置
	<code>(?&lt;!exp)</code>	匹配前面不是 <code>exp</code> 的位置
注释	<code>(?#comment)</code>	这种类型的分组不对正则表达式的处理产生任何影响，用于提供注释让人阅读

在捕获这个表项里，我们讲解了前两种用法，还有`(?:exp)`没有进行讲解。第三个`(?:exp)`不会改变正则表达式的处理方式，只是这样的组匹配的内容不会像前两种那样被捕获到某个组里面，也不会拥有组号，这样做有什么意义？一般来说是为了节省资

---

源，提高效率。比如说验证输入是否为整数，可以这样写`^([1-9][0-9]*|0)$`。这时候我们需要用到`()`来限制“|”表示“或”关系的范围，但我们只是要判断规则，没必要把`exp`匹配的内容保存到组里，这时就可以用非捕获组了`^(?:[1-9][0-9]*|0)$`。

## 10. 零宽断言

在表 4-4 中，零宽断言总共有四种形式。前两种是正向零宽断言，后两种是负向零宽断言。什么是零宽断言呢？我们知道元字符`\b`、`^`匹配的是一个位置，而且这个位置需要满足一定的条件，我们把这个条件称为断言或零宽度断言。断言用来声明一个应该为真的事实，正则表达式中只有当断言为真时才会继续进行匹配。可能大家感到有些抽象，下面通过一些例子进行讲解。

首先说一下正向零宽断言的两种形式：

- **(?=exp)**叫零宽度正预测先行断言，它断言此位置的后面能匹配表达式 `exp`。比如`[a-z]*(?=ing)` 匹配以 `ing` 结尾的单词的前面部分(除了 `ing` 以外的部分)，查找 **I love cooking and singing** 时会匹配出中的 **cook** 与 **sing**。先行断言的执行步骤应该是从要匹配字符的最右端找到第一个“`ing`”，在匹配前面的表达式，如无法匹配则查找第二个“`ing`”。
- **(?<=exp)**叫零宽度正回顾后发断言，它断言此位置的前面能匹配表达式 `exp`。比如`(?<=abc).*` 匹配以 `abc` 开头的字符串的后面部分，可以匹配 `abcdefgabc` 中的 **defgabc** 而不是 `abcdefg`。通过比较很容易看出后发断言和先行断言正好相反：它先从要匹配的字符串的最左端进行查找断言表达式，之后再匹配后面的字符串，如果无法匹配则继续查找第二个断言表达式，如此反复。

再说一下负向零宽断言的两种形式：

- **(?!exp)**叫零宽度负预测先行断言，断言**此位置**的后面不能匹配表达式 `exp`。比如`\b((?!abc)\w)+\b`匹配不包含连续字符串 `abc` 的单词，查找 **abc123,ade123** 这个字符串，可以匹配出 **ade123**，可以使用 **Match Tracer** 进行查看分析。
- **(?<!exp)**叫零宽度负回顾后发断言，断言此位置的前面不能匹配表达式 `exp`。比如`(?<![a-z])\d{7}`匹配前面不是小写字母的七位数字。还有一个复杂的例子：`(?<=<(\w+)>).*(?=<\1>)`，用于匹配不包含属性的简单 HTML 标签内里的内容。该表达式可以从`<div>python 爬虫</div>`中提取出 **python 爬虫**，这在 Python 爬虫开发中常用到。大家可以思考一下怎么提取出来包含属性的 HTML 标签内里的内容。

## 11. 注释

正则表达式可以包含注释进行解释说明，通过语法`(?#comment)`来实现，例如`\b\w+(?#字符串)\b`。要包含注释的话，最好是启用“忽略模式里的空白符”选项，这样在编写表达式时能任意的添加空格，Tab，换行，而实际使用时这些都被忽略。

## 12. 贪婪与懒惰

当正则表达式中包含能接受重复的限定符时，通常的行为是（在使整个表达式能得到匹配的前提下）匹配尽可能多的字符，这就是贪婪模式。以表达式 `a\w+b` 所示，如果搜索 `a12b34b`，会尽可能匹配更多的个数，最后就将 `a12b34b` 整个匹配到，而不是 `a12b`。但是如果想匹配出 `a12b` 怎么办呢？这时候就需要使用懒惰模式，尽可能匹配个数较少的情况，因此需要将上面的 `a\w+b` 表达式改为 `a\w+?b`，使用 `?` 来启用懒惰模式。表 4-5 列举了懒惰限定符的使用。

表 4-5 懒惰限定符的使用

语法	含义
<code>*?</code>	重复任意次，但尽可能少重复
<code>+?</code>	重复 1 次或更多次，但尽可能少重复
<code>??</code>	重复 0 次或 1 次，但尽可能少重复
<code>{n,m}?</code>	重复 n 到 m 次，但尽可能少重复
<code>{n,}? </code>	重复 n 次以上，但尽可能少重复

## 13. 处理选项

一般正则表达式的实现库都提供了用来改变处理正则表达式的方式，表 4-6 提供了常用的处理选项。

表 4-6 常用的处理选项

名称	含义
忽略大小写	匹配时不区分大小写。
多行模式	更改 <code>^</code> 和 <code>\$</code> 的含义，使它们分别在任意一行的行首和行尾匹配，而不仅仅在整个字符串的开头和结尾匹配。（在此模式下， <code>\$</code> 的精确含意是：匹配 <code>\n</code> 之前的位置以及字符串结束前的位置。）
单行模式	更改 <code>.</code> 的含义，使它与每一个字符匹配（包括换行符 <code>\n</code> ）。
忽略空白	忽略表达式中的非转义空白并启用由 <code>#</code> 标记的注释。
显式捕获	仅捕获已被显式命名的组。

正则表达式中还有平衡组/递归匹配的概念，对于初学者来说，一般用不了这么复杂，此处不进行讲解。

### 4.2.2 Python 与正则

上一节讲解了正则表达式的语法和应用，对于不同的编程语言来说，对正则表达式的语法绝大部分都是支持的，但是还是略有不同，每种编程语言都有一些独特的匹配规则，Python 也不例外。下面通过表 4-7 列出一些 Python 的匹配规则。

表 4-7 Python 的匹配规则

语法	含义	表达式示例	完整匹配的字符串
\A	仅匹配字符串开头	\Aabc	abc
\Z	仅匹配字符串末尾	abc\Z	abc
(?P<name>)	分组，除了原有编号外再指定一个额外的别名	(?P<word>abc){2}	abcbabc
(?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

在讲 Python 对正则表达式的实现之前，首先让说一下反斜杠问题。正则表达式里使用“\”作为转义字符，这就可能造成反斜杠困扰。假如你需要匹配文本中的字符“\”，那么使用编程语言表示的正则表达式里将需要 4 个反斜杠“\\”：前两个和后两个分别用于在编程语言里转义成反斜杠，转换成两个反斜杠后再在正则表达式里转义成一个反斜杠。但是 Python 提供了对原生字符串的支持，从而解决了这个问题。匹配一个“\”的正则表达式可以写为 r'\'，同样，匹配一个数字的“\d”可以写成 r'\d'，

Python 通过 re 模块提供对正则表达式的支持。使用 re 的一般步骤是先将正则表达式的字符串形式编译为 Pattern 实例，然后使用 Pattern 实例处理文本并获得匹配结果，最后使用 Match 实例获得信息，进行其他的操作。主要用到的方法列举如下：

- re.compile(string[,flag])
- re.match(pattern, string[, flags])
- re.search(pattern, string[, flags])
- re.split(pattern, string[, maxsplit])
- re.findall(pattern, string[, flags])
- re.finditer(pattern, string[, flags])
- re.sub(pattern, repl, string[, count])
- re.subn(pattern, repl, string[, count])

首先说一下 re 中 compile 函数，是将一个正则表达式的字符串转化为 pattern 匹配对象。示例如下：

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

```

这是生成一个匹配数字的 pattern 对象，用来给接下来的函数作为参数，进行进一步的搜索操作。

大家发现其他几个函数中，还有一个 flags 参数。参数 flag 是匹配模式，取值可以使用按位或运算符“|”表示同时生效，比如 re.I | re.M。flags 的可选值如下：

- re.I: 忽略大小写
- re.M: 多行模式，改变'^'和'\$'的行为

- 
- re.S: 点任意匹配模式, 改变'.'的行为
  - re.L: 使预定字符类 `\w \W \b \B \s \S` 取决于当前区域设定
  - re.U: 使预定字符类 `\w \W \b \B \s \S \d \D` 取决于 `unicode` 定义的字符属性
  - re.X: 详细模式。这个模式下正则表达式可以是多行, 忽略空白字符, 并可以加入注释

## 1. re.match(pattern, string[, flags])

这个函数是从输入参数 `string` (匹配的字符串) 的开头开始, 尝试匹配 `pattern`, 一直向后匹配, 如果遇到无法匹配的字符或者已经到达 `string` 的末尾, 立即返回 `None`, 反之获取匹配的结果。示例如下:

```
#coding:utf-8
import re
# 将正则表达式编译成 Pattern 对象
pattern = re.compile(r'\d+')
# 使用 re.match 匹配文本, 获得匹配结果, 无法匹配时将返回 None
result1 = re.match(pattern, '192abc')
if result1:
    print result1.group()
else:
    print '匹配失败 1'
result2 = re.match(pattern, 'abc192')
if result2:
    print result2.group()
else:
    print '匹配失败 2'
```

运行结果如下:

192

匹配失败 2

匹配 `192abc` 字符串时, `match` 函数是从字符串开头进行匹配, 匹配到 `192` 立即返回值, 通过 `group()` 就获取捕获的值。同样, 匹配 `abc192` 字符串时, 字符串开头不符合正则表达式, 立即返回 `None`。

## 2. re.search (pattern, string[, flags])

`search` 方法与 `match` 方法极其类似, 区别在于 `match()` 函数只从 `string` 的开始位置匹配, `search()` 会扫描整个 `string` 查找匹配, `match()` 只有在 `string` 起始位置匹配成功的时候才有返回, 如果不是开始位置匹配成功的话, `match()` 就返回 `None`。同样, `search` 方法的返回对象和 `match()` 返回对象方法和属性是一致的。示例如下:

```
import re
# 将正则表达式编译成 Pattern 对象
pattern = re.compile(r'\d+')
result = re.search(pattern, 'abc192')
```



---

*# 使用 re.match 匹配文本，获得匹配结果，无法匹配时将返回 None*

```
result1 = re.search(pattern,'abc192edf')
```

```
if result1:
```

```
    print result1.group()
```

```
else:
```

```
    print '匹配失败 1'
```

输出结果为：

```
192
```

### 3. re.split(pattern, string[, maxsplit])

按照能够匹配的子串将 string 分割后返回列表。maxsplit 用于指定最大分割次数，不指定将全部分割。示例如下：

```
import re
```

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

```
print re.split(pattern,'A1B2C3D4')
```

输出结果为：

```
['A', 'B', 'C', 'D', '']
```

### 4. re.findall (pattern, string[, flags])

搜索整个 string，结果以列表形式返回全部能匹配的子串。示例如下：

```
import re
```

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

```
print re.findall(pattern,'A1B2C3D4')
```

输出结果为：

```
['1', '2', '3', '4']
```

### 5. re.finditer (pattern, string[, flags])

搜索整个 string，结果以迭代器形式返回全部能匹配的 Match 对象。示例如下：

```
import re
```

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

```
matchiter = re.finditer(pattern,'A1B2C3D4')
```

```
for match in matchiter:
```

```
    print match.group()
```

输出结果为：

```
1
```

```
2
```

```
3
```

```
4
```

---

## 6. re.sub(pattern, repl, string[, count])

使用 `repl` 替换 `string` 中每一个匹配的子串后返回替换后的字符串。当 `repl` 是一个字符串时，可以使用 `\id` 或 `\g<id>`、`\g<name>` 引用分组，但不能使用编号 0。当 `repl` 是一个方法时，这个方法应当只接受一个参数（`Match` 对象），并返回一个字符串用于替换（返回的字符串中不能再引用分组）。`count` 用于指定最多替换次数，不指定时全部替换。示例如下：

```
import re
p = re.compile(r'(?P<word1>\w+) (?P<word2>\w+)')#使用名称引用
s = 'i say, hello world!'
print p.sub(r'\g<word2> \g<word1>', s)
p = re.compile(r'(\w+) (\w+)')#使用编号
print p.sub(r'\2 \1', s)
def func(m):
    return m.group(1).title() + ' ' + m.group(2).title()
print p.sub(func, s)
```

输出结果为：

```
say i, world hello!
say i, world hello!
I Say, Hello World!
```

## 7. re.subn(pattern, repl, string[, count])

返回 `(sub(repl, string[, count]), 替换次数)`。示例如下：

```
import re
s = 'i say, hello world!'
p = re.compile(r'(\w+) (\w+)')
print p.subn(r'\2 \1', s)
def func(m):
    return m.group(1).title() + ' ' + m.group(2).title()
print p.subn(func, s)
```

输出结果为：

```
('say i, world hello!', 2)
('I Say, Hello World!', 2)
```

以上 7 个函数在 `re` 模块中进行搜索匹配，如何将捕获到的值提取出来呢？这就需要介绍 `Match` 对象，之前已经使用了 `Match` 中的 `groups` 方法，现在介绍一下 `Match` 对象的属性和方法。

`Match` 对象的属性：

- `string`: 匹配时使用的文本。
- `re`: 匹配时使用的 `Pattern` 对象。

- 
- **pos**: 文本中正则表达式开始搜索的索引。值与 `Pattern.match()` 和 `Pattern.seach()` 方法的同名参数相同。
  - **endpos**: 文本中正则表达式结束搜索的索引。值与 `Pattern.match()` 和 `Pattern.seach()` 方法的同名参数相同。
  - **lastindex**: 最后一个被捕获的分组在文本中的索引。如果没有被捕获的分组, 将为 `None`。
  - **lastgroup**: 最后一个被捕获的分组的别名。如果这个分组没有别名或者没有被捕获的分组, 将为 `None`。

**Match 对象的方法:**

- **group([group1, ...])**: 获得一个或多个分组截获的字符串; 指定多个参数时将以元组形式返回。**group1** 可以使用编号也可以使用别名; 编号 0 代表整个匹配的子串; 不填写参数时, 返回 `group(0)`; 没有截获字符串的组返回 `None`; 截获了多次的组返回最后一次截获的子串。
- **groups([default])**: 以元组形式返回全部分组截获的字符串。相当于调用 `group(1,2,...last)`。**default** 表示没有截获字符串的组以这个值替代, 默认为 `None`。
- **groupdict([default])**: 返回以有别名的组的别名为键、以该组截获的子串为值的字典, 没有别名的组不包含在内。**default** 含义同上。
- **start([group])**: 返回指定的组截获的子串在 `string` 中的起始索引 (子串第一个字符的索引)。**group** 默认值为 0。
- **end([group])**: 返回指定的组截获的子串在 `string` 中的结束索引 (子串最后一个字符的索引+1)。**group** 默认值为 0。
- **span([group])**: 返回(`start(group)`, `end(group)`)。
- **expand(template)**: 将匹配到的分组代入 `template` 中然后返回。`template` 中可以使用 `\id` 或 `\g<id>`、`\g<name>` 引用分组, 但不能使用编号 0。`\id` 与 `\g<id>` 是等价的; 但 `\10` 将被认为是第 10 个分组, 如果你想表达 `\1` 之后是字符 '0', 只能使用 `\g<1>0`。

示例如下:

```
import re
pattern = re.compile(r'(\w+) (\w+) (?P<word>.*')
match = pattern.match( 'I love you!')

print "match.string:", match.string
print "match.re:", match.re
print "match.pos:", match.pos
print "match.endpos:", match.endpos
print "match.lastindex:", match.lastindex
print "match.lastgroup:", match.lastgroup

print "match.group(1,2):", match.group(1, 2)
print "match.groups():", match.groups()
print "match.groupdict():", match.groupdict()
print "match.start(2):", match.start(2)
print "match.end(2):", match.end(2)
print "match.span(2):", match.span(2)
print r"match.expand(r'\2 \1 \3'):", match.expand(r'\2 \1 \3')
```

输出结果:

```
match.string: I love you!  
match.re: <_sre.SRE_Pattern object at 0x003F47A0>  
match.pos: 0  
match.endpos: 11  
match.lastindex: 3  
match.lastgroup: word  
match.group(1,2): ('I', 'love')  
match.groups(): ('I', 'love', 'you!')  
match.groupdict(): {'word': 'you!'}  
match.start(2): 2  
match.end(2): 6  
match.span(2): (2, 6)  
match.expand(r'\2 \1 \3'): love I you!
```

以上介绍的 7 种方法的调用方式大都是 `re.match`,`re.search` 之类的。其实还可以使用 `re.compile` 方法产生的 `pattern` 对象直接调用这些函数，类似 `pattern.match`，`pattern.search`，只不过不用将 `pattern` 作为第一个参数传入。函数对比如表 4-8 所示：

表 4-8 函数调用方式

Re 调用	Pattern 调用
<code>re.match(pattern, string[, flags])</code>	<code>pattern.match(string[, flags])</code>
<code>re.search(pattern, string[, flags])</code>	<code>pattern.search(string[, flags])</code>
<code>re.split(pattern, string[, maxsplit])</code>	<code>pattern.split(string[, maxsplit])</code>
<code>re.findall(pattern, string[, flags])</code>	<code>pattern.findall(string[, flags])</code>
<code>re.finditer(pattern, string[, flags])</code>	<code>pattern.finditer(string[, flags])</code>
<code>re.sub(pattern, repl, string[, count])</code>	<code>pattern.sub(repl, string[, count])</code>
<code>re.subn(pattern, repl, string[, count])</code>	<code>pattern.subn(repl, string[, count])</code>

### 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 BeautifulSoup 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 BeautifulSoup 在 Python 爬虫开发中的使用。

#### 4.3.1 安装 BeautifulSoup

对于 BeautifulSoup，我们推荐使用的是 BeautifulSoup 4，已经移植到 BS4 中，Beautiful Soup 3 已经停止开发了。安装 BeautifulSoup 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`。
- BeautifulSoup 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

如表 4-9 所示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

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

这也是为什么推荐使用 lxml 作为解析器的原因，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 bs4 import BeautifulSoup`。接着创建一下包含 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按照这个字符串为例子。

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。打印一下 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>
```

---

```
</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，使用方法在表 4-9 中。

## 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，通俗点说就是标签。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 标签以及它们里面的内容被称作 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.p`

从例子中可以看到利用 `soup` 加标签名就可以获取这些标签的内容，比之前讲的正则表达式简单多了。不过利用这种方式，它查找的是在所有内容中的第一个符合要求的标签，如果要查询所有的标签，之后会依次进行讲解。

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]，对于其他内部标签，输出的值便为标签本身的名称。

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 BeautifulSoup 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>
```

已经将 title 标签成功修改为 mytitle。

再说一下 Tag 中的 attributes。一个 tag 可能有很多个属性，<p class="title"><b>The Dormouse's story</b></p> 有一个 “class” 的属性,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，获取 Tag 中所有属性：

```
print soup.p.attrs
```

输出结果：

```
{'class': ['title']}
```

和 name 一样，我们可以对标签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

我们已经得到了标签的内容，要想获取标签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 的标签，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 标签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，我们发现它已经把注释符号去掉了。另外我们打印输出下它的类型，发现它是一个 Comment 类型。如果在我们不清楚这个标签.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>` 标签只有一个直接子节点 `<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：

```
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：如果一个标签里面没有标签了，那么 `.string` 就会返回标签里面的内容。如果标签里面只有唯一的一个标签了，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，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'
```



---

### (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统一级的节点，`.next_sibling` 属性获取了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

前后节点需要讲解 `.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，而是在所有节点，不分层次，其中 `<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，其它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演示：

##### (1)name 参数

`name` 参数可以查找所有名字为 `name` 的标签，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>**标签，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。下面例子中找出所有以 `b` 开头的标签,这表示**<body>**和**<b>**标签都应该被找到：

```
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。下面代码找到文档中所有**<a>**标签和**<b>**标签：

```
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参数值包括 [字符串](#)、[正则表达式](#)、[列表](#)、[True](#)。

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。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>]
```

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`：

```
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，还可以与其它参数混合使用来过滤 `tag`。`BeautifulSoup` 会找到 `.string` 方法与 `text` 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：

```
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。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, BeautifulSoup 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, 通过表 4-10 将其他函数列举出来:

表 4-10 搜索函数

函数	功能介绍
find(name,attrs,recursive,text,**kwargs)	它与 find_all() 方法唯一的区别是 find_all() 方法的返回结果是所有满足要求的值组成的列表, 而 find() 方法直接返回 find_all 搜索结果中的第一个值。
find_parents( <a href="#">name</a> , <a href="#">attrs</a> , <a href="#">recursive</a> , <a href="#">text</a> , <a href="#">**kwargs</a> ) find_parent( <a href="#">name</a> , <a href="#">attrs</a> , <a href="#">recursive</a> , <a href="#">text</a> , <a href="#">**kwargs</a> )	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。
find_next_siblings(name,attrs,recursive,text,**kwargs) find_next_sibling(name,attrs,recursive,text,**kwargs)	这 2 个方法通过 .next_siblings 属性对当前 tag 的所有后面解析的兄弟 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。
find_previous_siblings( <a href="#">name</a> , <a href="#">attrs</a> , <a href="#">recursive</a> , <a href="#">text</a> , <a href="#">**kwargs</a> ) find_previous_sibling( <a href="#">name</a> , <a href="#">attrs</a> , <a href="#">recursive</a> , <a href="#">text</a> , <a href="#">**kwargs</a> )	这 2 个方法通过 .previous_siblings 属性对当前 tag 的前面解析的兄弟 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。
find_all_next( <a href="#">name</a> , <a href="#">attrs</a> , <a href="#">recursive</a> , <a href="#">text</a> , <a href="#">**kwargs</a> ) find_next( <a href="#">name</a> , <a href="#">attrs</a> , <a href="#">recursive</a> , <a href="#">text</a> , <a href="#">**kwargs</a> )	这 2 个方法通过 .next_elements 属性对当前 tag 的之后的 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。
find_all_previous( <a href="#">name</a> , <a href="#">attrs</a> , <a href="#">recursive</a> , <a href="#">text</a> , <a href="#">**kwargs</a> ) find_previous( <a href="#">name</a> , <a href="#">attrs</a> , <a href="#">recursive</a> , <a href="#">text</a> , <a href="#">**kwargs</a> )	这 2 个方法通过 .previous_elements 属性对当前节点前面的 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。

## 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 也是可以定位元素的位置。在写 CSS 时, 标签名不加任何修饰, 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) 通过标签名称进行查找

通过标签名称可以直接查找、逐层查找，也可以找到某个 tag 标签下的直接子标签和兄弟节点标签。示例如下：

```
#直接查找 title 标签
print soup.select("title")
#逐层查找 title 标签
print soup.select("html head title")
#查找直接子节点
#查找 head 下的 title 标签
print soup.select("head > title")
#查找 p 下的 id="link1" 的标签
print soup.select("p > #link1")
#查找兄弟节点
#查找 id="link1" 之后 class=sister 的所有兄弟标签
print soup.select("#link1 ~ .sister")
#查找紧跟着 id="link1" 之后 class=sister 的子标签
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 4.24 所示:

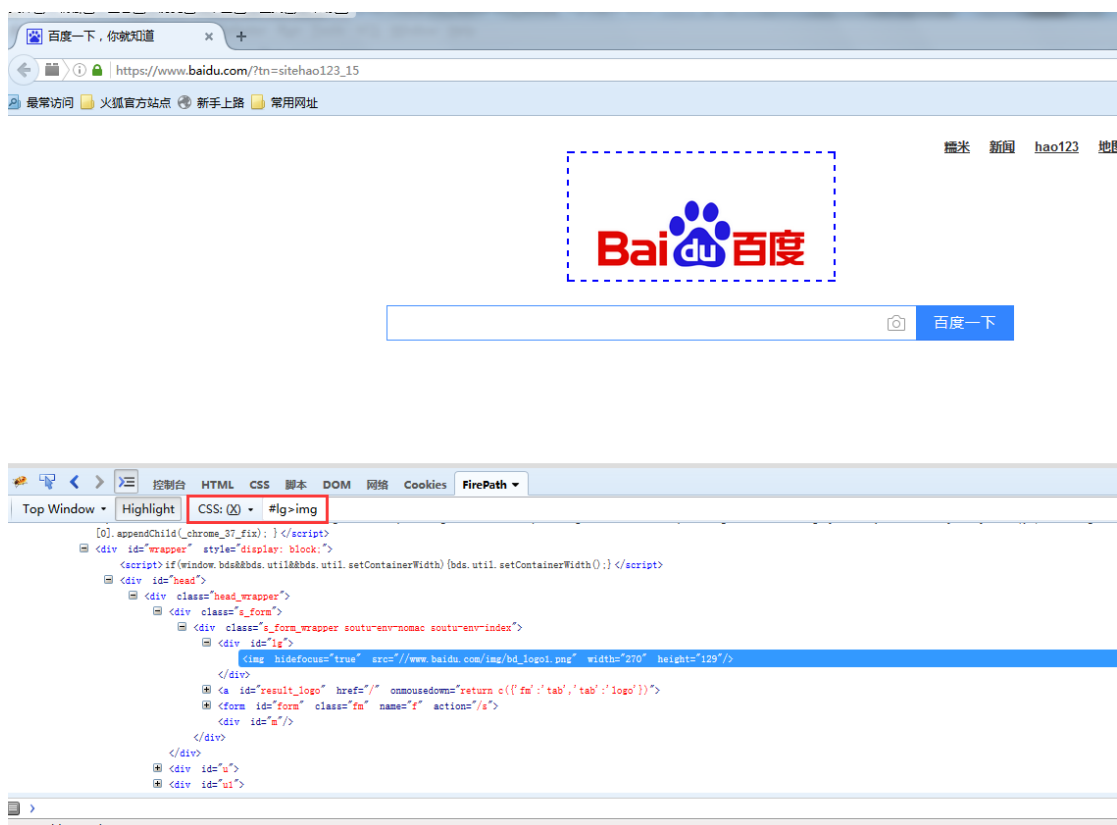


图 4.24 FirePath CSS 选择器

---

### 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，挺适合新手。lxml 的 XPath 写起来麻烦，开发效率不如 BeautifulSoup，当然这也是因人而异，如果你熟练 XPath，使用 lxml 是更好地选择，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']
```

lxml 的使用讲到这里，使用 lxml 的关键是构造 XPath 表达式，如果大家对 XPath 不熟悉，可以复习一下第二章中 XPath 内容。

## 4.4 小结

本章主要讲解了 HTML 解析的各种方式，这也是提取网页数据非常关键的环节。希望大家把正则表达式、Beautiful Soup 和 XPath 的知识做到灵活运用。同时还要注意 Firebug、FirePath 和 Match Tracer 的配合使用，将会使开发达到事半功倍的效果。

购买链接：<https://item.jd.com/12206762.html>

# 第 6 章 实战项目:基础爬虫

本章开始讲解第一个实战项目：基础爬虫。为什么叫基础爬虫呢？首先这个爬虫项目功能简单，仅仅考虑功能实现，未涉及优化和稳健性的考虑。再者爬虫虽小，但是五脏俱全，大型爬虫有的基础模块，这个爬虫也都有，只不过实现方式、优化方式，大型爬虫做的更加全面、多样。本次实战项目的需求是爬取 100 个百度百科网络爬虫词条以及相关词条的标题、摘要和链接等信息。如图 6.1 所示：



图 6.1 网络爬虫词条

## 6.1 基础爬虫架构及运行流程

首先讲解一下基础爬虫的架构。一个基础爬虫包含哪些模块，各个模块之间的关系是什么？如图 6.2 所示：

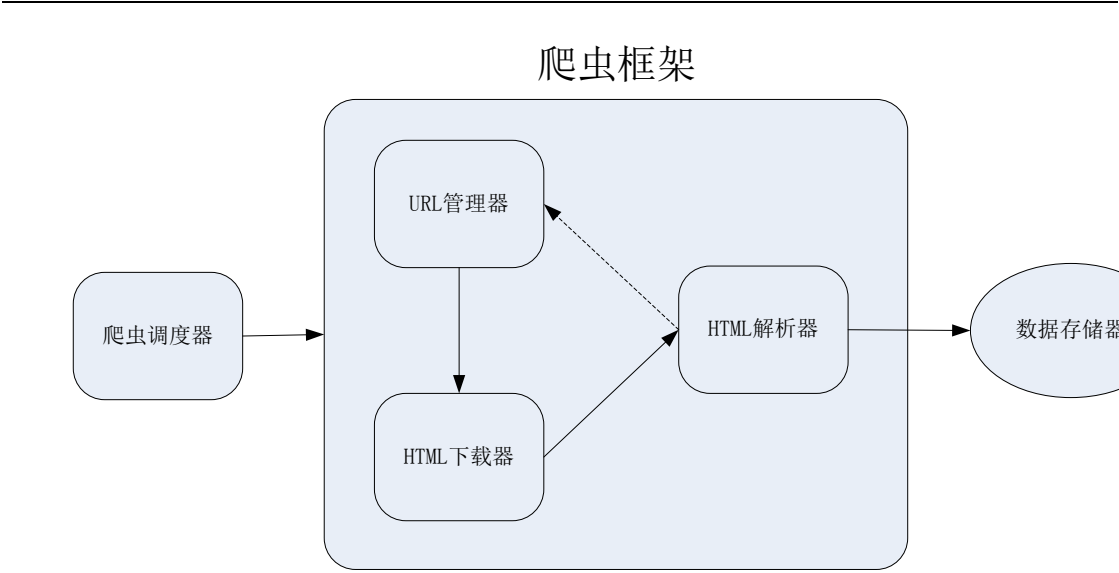


图 6.2 爬虫框架

从图 6.2 可以看到，基础爬虫框架主要包括五大模块，分别为爬虫调度器、URL 管理器、HTML 下载器、HTML 解析器、数据存储器。功能分析如下：

- 爬虫调度器主要负责统筹其他四个模块的协调工作。
- URL 管理器负责管理 URL 链接，维护已经爬取的 URL 集合和未爬取的 URL 集合，提供获取新 URL 链接的接口。
- HTML 下载器用于从 URL 管理器中获取未爬取的 URL 链接并下载 HTML 网页。
- HTML 解析器用于从 HTML 下载器中获取已经下载的 HTML 网页，并从中解析出新的 URL 链接交给 URL 管理器，解析出有效数据交给数据存储器。
- 数据存储器用于将 HTML 解析器解析出来的数据通过文件或者数据库的形式存储起来。

下面通过图 6.3 展示一下爬虫框架的动态运行流程，方便大家理解。

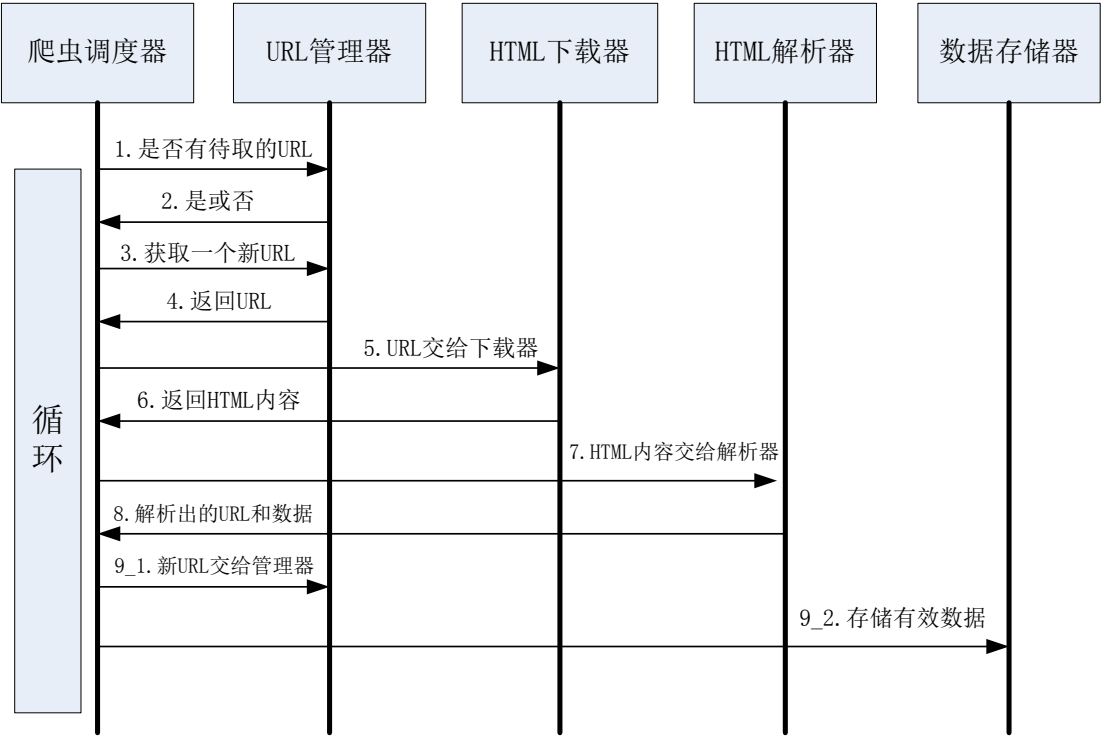


图 6.3 运行流程

## 6.2 URL 管理器

URL 管理器主要包括两个变量，一个是已爬取 URL 的集合，另一个是未爬取 URL 的集合，采用的 Python 中 set 类型，主要是使用 set 的去重复功能，防止链接重复爬取。因为爬取链接重复时候，容易造成死循环。链接去重复在 Python 爬虫开发中是必备的功能，解决方案主要有三种方式：1. 内存去重 2. 关系数据库去重 3. 缓存数据库去重。大型成熟的爬虫基本上采用缓存数据库的去重方案，尽可能避免了内存大小的限制，又比关系型数据库去重性能高很多。由于我们基础爬虫爬取数量较小，因此我们使用 Python 中 set 这个内存去重方式。

URL 管理器除了具有两个 URL 集合，还需要提供以下接口，用于配合其他模块使用，接口如下：

- 判断是否有待取的 URL，方法定义为 has\_new\_url()。
- 添加新的 URL 到未爬取集合中，方法定义为 add\_new\_url(url), add\_new\_urls(urls)。
- 获取一个未爬取的 URL，方法定义为 get\_new\_url()。
- 获取未爬取 URL 集合的大小，方法定义为 new\_url\_size()。
- 获取已经爬取的 URL 集合的大小，方法定义为 old\_url\_size()。

程序 URLManager.py 完整代码如下：

```
#coding:utf-8
class UrlManager(object):
    def __init__(self):
        self.new_urls = set()#未爬取 URL 集合
        self.old_urls = set()#已爬取 URL 集合

    def has_new_url(self):
        """
        判断是否有未爬取的 URL
        :return:
        """
        return self.new_url_size() != 0

    def get_new_url(self):
        """
        获取一个未爬取的 URL
        :return:
        """
        new_url = self.new_urls.pop()
        self.old_urls.add(new_url)
        return new_url

    def add_new_url(self, url):
        """
```

---

```

        将新的 URL 添加到未爬取的 URL 集合中
    :param url: 单个 URL
    :return:
    """
    if url is None:
        return
    if url not in self.new_urls and url not in self.old_urls:
        self.new_urls.add(url)

def add_new_urls(self,urls):
    """
    将新的 URLs 添加到未爬取的 URL 集合中
    :param urls:url 集合
    :return:
    """
    if urls is None or len(urls)==0:
        return
    for url in urls:
        self.add_new_url(url)

def new_url_size(self):
    """
    获取未爬取 URL 集合的s 大小
    :return:
    """
    return len(self.new_urls)

def old_url_size(self):
    """
    获取已经爬取 URL 集合的大小
    :return:
    """
    return len(self.old_urls)

```

## 6.3HTML 下载器

HTML 下载器用来下载网页，这时候需要注意网页的编码，保证下载到网页没有乱码。下载器需要用到 requests 模块，里面只需要实现一个接口即可：download(url)。程序 HtmlDownloader.py 代码如下：

```

#coding:utf-8
import requests
class HtmlDownloader(object):

    def download(self,url):
        if url is None:
            return None
        user_agent = 'Mozilla/4.0 (compatible; MSIE 5.5; Windows NT)'
        headers={'User-Agent':user_agent}

```



```

r = requests.get(url,headers=headers)
if r.status_code==200:
    r.encoding='utf-8'
    return r.text
return None

```

## 6. 4HTML 解析器

HTML 解析器使用 BeautifulSoup4 进行 HTML 解析。需要解析的部分主要分为提取相关词条页面的 URL 和提取当前词条的标题和摘要信息。

先使用 Firebug 查看一下标题和摘要所在的结构位置，如图 6.4 所示：

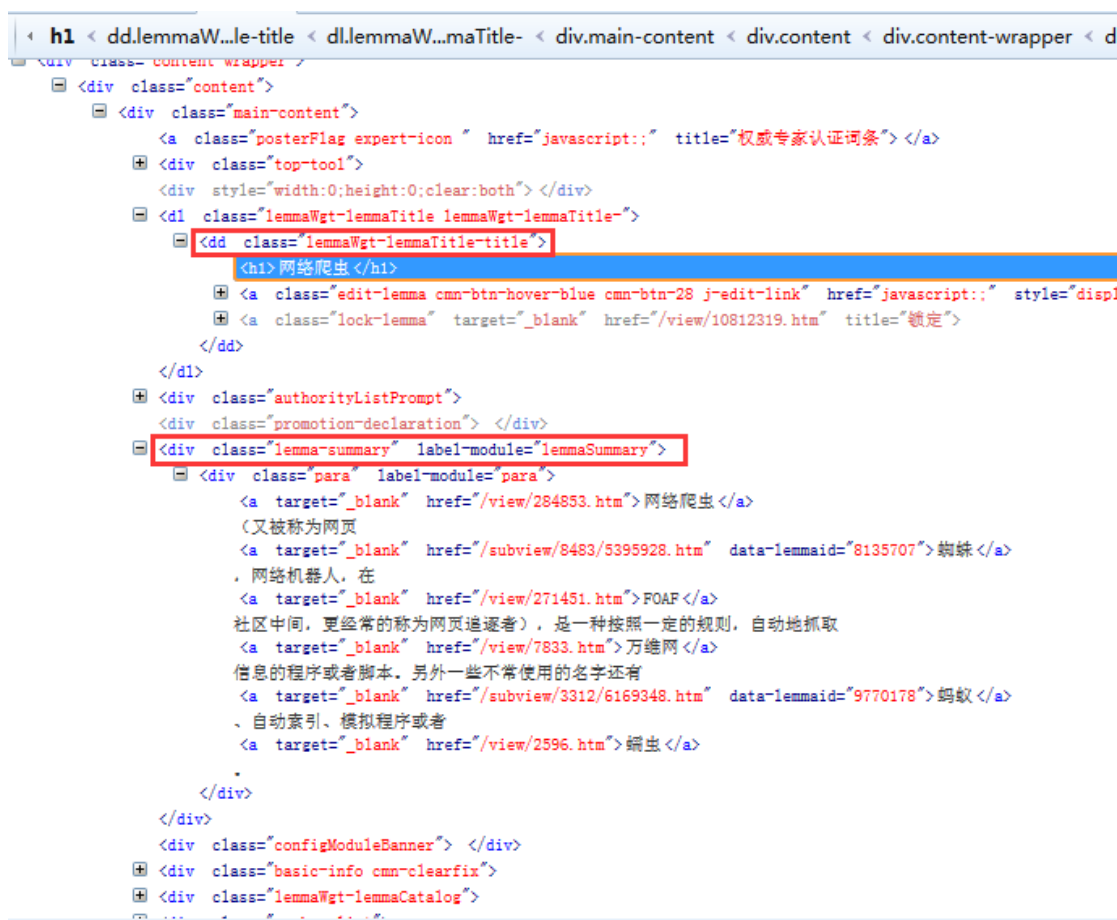


图 6. 4HTML 结构位置

从上图可以看到标题的标签位于<dd class="lemmaWgt-lemmaTitle-title"><h1></h1>，摘要文本位于<div class="lemma-summary" label-module="lemmaSummary">。

最后分析一下需要抽取的 URL 的格式。相关词条的 URL 格式类似于<a target="\_blank" href="/view/7833.htm">万维网</a>这种形式，提取出 a 标签中的 href 属性即可，从格式中可以看到 href 属性值是一个相对网址，可以使用 urlparse.urljoin 函数将当前网址和相对网址进行拼接成完整的 URL 路径。

HTML 解析器主要提供一个 parser 对外接口，输入参数为当前页面的 URL 和 HTML 下载器返回的网页内容。解析器 HtmlParser.py 程序代码如下

---

```
#coding:utf-8
import re
import urlparse
from bs4 import BeautifulSoup

class HtmlParser(object):

    def parser(self,page_url,html_cont):
        """
        用于解析网页内容抽取 URL 和数据
        :param page_url: 下载页面的 URL
        :param html_cont: 下载的网页内容
        :return: 返回 URL 和数据
        """
        if page_url is None or html_cont is None:
            return
        soup = BeautifulSoup(html_cont,'html.parser',from_encoding='utf-8')
        new_urls = self._get_new_urls(page_url,soup)
        new_data = self._get_new_data(page_url,soup)
        return new_urls,new_data

    def _get_new_urls(self,page_url,soup):
        """
        抽取新的 URL 集合
        :param page_url: 下载页面的 URL
        :param soup:soup
        :return: 返回新的 URL 集合
        """
        new_urls = set()
        #抽取符合要求的a 标签
        links = soup.find_all('a',href=re.compile(r'/view/\d+\\.htm'))
        for link in links:
            #提取 href 属性
            new_url = link['href']
            #拼接成完整网址
            new_full_url = urlparse.urljoin(page_url,new_url)
            new_urls.add(new_full_url)
        return new_urls
    def _get_new_data(self,page_url,soup):
        """
        抽取有效数据
        :param page_url: 下载页面的 URL
        :param soup:
        :return: 返回有效数据
        """
        data={}
        data['url']=page_url
        title = soup.find('dd',class_='lemmaWgt-lemmaTitle-title').find('h1')
```

---

```
data['title']=title.get_text()
summary = soup.find('div',class_='lemma-summary')
#获取 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
data['summary']=summary.get_text()
return data
```

## 6.5 数据存储

数据存储主要包括两个方法：一个是用于将解析出来的数据存储到内存中 `store_data(data)`，另一个用于将储存的数据输出为指定的文件格式 `output_html()`，我们使用的是将数据输出为 Html 格式。`DataOutput.py` 程序如下：

```
#coding:utf-8
import codecs

class DataOutput(object):

    def __init__(self):
        self.datas=[]

    def store_data(self,data):
        if data is None:
            return
        self.datas.append(data)

    def output_html(self):
        fout=codecs.open('baike.html','w',encoding='utf-8')
        fout.write("<html>")
        fout.write("<body>")
        fout.write("<table>")
        for data in self.datas:
            fout.write("<tr>")
            fout.write("<td>%s</td>"%data['url'])
            fout.write("<td>%s</td>"%data['title'])
            fout.write("<td>%s</td>"%data['summary'])
            fout.write("</tr>")
            self.datas.remove(data)
        fout.write("</table>")
        fout.write("</body>")
        fout.write("</html>")
        fout.close()
```

其实上面代码并不是很好的方式，更好地做法应该是将数据分批存储到文件，而不应该将所有数据存储到内存，一次性写入文件，系统出现异常容易造成数据丢失。但是由于我们只需要 100 条数据，速度很快，所以这种方式还是可行的。如果大家数据很多，还是采取分批存储的办法。

---

## 6.6 爬虫调度器

以上几节已经将 URL 管理器、HTML 下载器、HTML 解析器和数据存储器等模块进行了实现，接下来编写爬虫调度器进行协调管理这些模块。爬虫调度器首先要做的是初始化各个模块，然后通过 `crawl(root_url)` 方法传入入口 URL，方法内部实现按照运行流程控制各个模块的工作。爬虫调度器 `SpiderMan.py` 的程序如下：

```
#coding:utf-8

from firstSpider.DataOutput import DataOutput
from firstSpider.HtmlDownloader import HtmlDownloader
from firstSpider.HtmlParser import HtmlParser
from firstSpider.UrlManager import UrlManager
class SpiderMan(object):
    def __init__(self):
        self.manager = UrlManager()
        self.downloader = HtmlDownloader()
        self.parser = HtmlParser()
        self.output = DataOutput()
    def crawl(self,root_url):
        #添加入口 URL
        self.manager.add_new_url(root_url)
        #判断 url 管理器中是否有新的 url，同时判断抓取了多少个 url
        while(self.manager.has_new_url() and self.manager.old_url_size()<100):
            try:
                #从 URL 管理器获取新的 url
                new_url = self.manager.get_new_url()
                #HTML 下载器下载网页
                html = self.downloader.download(new_url)
                #HTML 解析器抽取网页数据
                new_urls,data = self.parser.parser(new_url,html)
                #将抽取到 url 添加到 URL 管理器中
                self.manager.add_new_urls(new_urls)
                #数据存储管理器储存文件
                self.output.store_data(data)
                print "已经抓取%s个链接"%self.manager.old_url_size()
            except Exception,e:
                print "crawl failed"
            #数据存储管理器将文件输出成指定格式
            self.output.output_html()

if __name__=="__main__":
    spider_man = SpiderMan()
    spider_man.crawl("http://baike.baidu.com/view/284853.htm")
```

到这里基础爬虫架构所需的模块都已经完成，启动程序，大约 1 分钟左右，数据都被存储为 `baike.html`。使用浏览器打开，效果如图 6.5 所示：

<a href="http://baike.baidu.com/view/284853.htm">http://baike.baidu.com/view/284853.htm</a>	网络爬虫	网络爬虫（又被称为网页蜘蛛，网络机器人，在FOAF社区中间，更经常的称为网页追逐者），是一种按照一定的规则，
<a href="http://baike.baidu.com/view/286828.htm">http://baike.baidu.com/view/286828.htm</a>	数据源	数据源是指数据库应用程序所使用的数据库或者数据库服务器。数据源（Data Source）顾名思义，数据的来源，是提供
<a href="http://baike.baidu.com/view/6284787.htm">http://baike.baidu.com/view/6284787.htm</a>	HITS算法	HITS 算法是由康奈尔大学( Cornell University ) 的Jon Kleinberg 博士于1997 年首先提出的,为IBM 公司阿尔马登研究
<a href="http://baike.baidu.com/view/2107226.htm">http://baike.baidu.com/view/2107226.htm</a>	处理机	处理机包括中央处理器，主存储器，输入-输出接口，加接外围设备就构成完整的计算机系统。处理机是处理计算机系统
<a href="http://baike.baidu.com/view/108191.htm">http://baike.baidu.com/view/108191.htm</a>	里程碑	里程碑，汉语词语，设于道路旁边，用以指示公路里程的标志。多比喻在历史发展过程中可以作为标志的大事。
<a href="http://baike.baidu.com/view/10812277.htm">http://baike.baidu.com/view/10812277.htm</a>	百度百科：多义词	百度百科里，当同一个词条名可指代含义概念不同的事物时，这个词条称为多义词。如词条“苹果”，既可以代表一种
<a href="http://baike.baidu.com/view/848468.htm">http://baike.baidu.com/view/848468.htm</a>	中华人民共和国企业所得税法	《中华人民共和国企业所得税法》是为了使中国境内企业和其他取得收入的组织缴纳企业所得税制定的法律。由中华人
<a href="http://baike.baidu.com/view/433224.htm">http://baike.baidu.com/view/433224.htm</a>	中华人民共和国外商投	中华人民共和国外商投资企业和外国企业所得税法（已废止）（《中华人民共和国企业所得税法》第六十条 本法自2008
<a href="http://baike.baidu.com/view/4225407.htm">http://baike.baidu.com/view/4225407.htm</a>	接受捐赠收入	1991年4月9日第七届全国人民代表大会第四次会议通过的《中华人民共和国外商投资企业和外国企业所得税法》和199:

图 6.5 baike.html

## 6.7 小结

通过前 6 节，基本上完成了对爬虫架构的各个模块的讲解。无论大型或者小型爬虫都不会脱离这五个模块，希望大家对整个运行流程有个清晰的认识，之后涉及到的实战项目都会见到这五个模块的身影。

购买链接:<https://item.jd.com/12206762.html>

# 第 7 章实战项目:简单分布式爬虫

本章讲的依旧是实战项目，实战内容是打造分布式爬虫，这对初学者来说，是一个不小的挑战，也是一次有意义的尝试。这次打造的分布式爬虫采用比较简单的主从模式，完全手工打造，不使用成熟框架，基本上涵盖了前六章的主要知识点，其中涉及分布式的知识点是分布式进程和进程间通信的内容，算是对 Python 爬虫基础篇的总结。

现在大型的爬虫系统都是采取分布式爬取结构，通过此次实战项目，让大家对分布式爬虫有一个比较清晰地了解，为之后系统的讲解分布式爬虫打下基础，其实它并没有多么困难。实战目标：爬取 2000 个百度百科网络爬虫词条以及相关词条的标题、摘要和链接等信息，采用分布式结构改写第六章的基础爬虫，使功能更加强大。爬取页面请看图 6.1。

## 7.1 简单分布式爬虫结构

本次分布式爬虫采用主从模式。主从模式是指由一台主机作为控制节点负责所有运行网络爬虫的主机进行管理，爬虫只需要从控制节点那里接收任务，并把新生成任务提交给控制节点就可以了，在这个过程中不必与其他爬虫通信，这种方式实现简单利于管理。而控制节点则需要与所有爬虫进行通信，因此可以看到主从模式是有缺陷的，控制节点会成为整个系统的瓶颈，容易导致整个分布式网络爬虫系统性能下降。

此次使用三台主机进行分布式爬取，一台主机作为控制节点，另外两台主机作为爬虫节点。爬虫结构如图 7.1 所示：

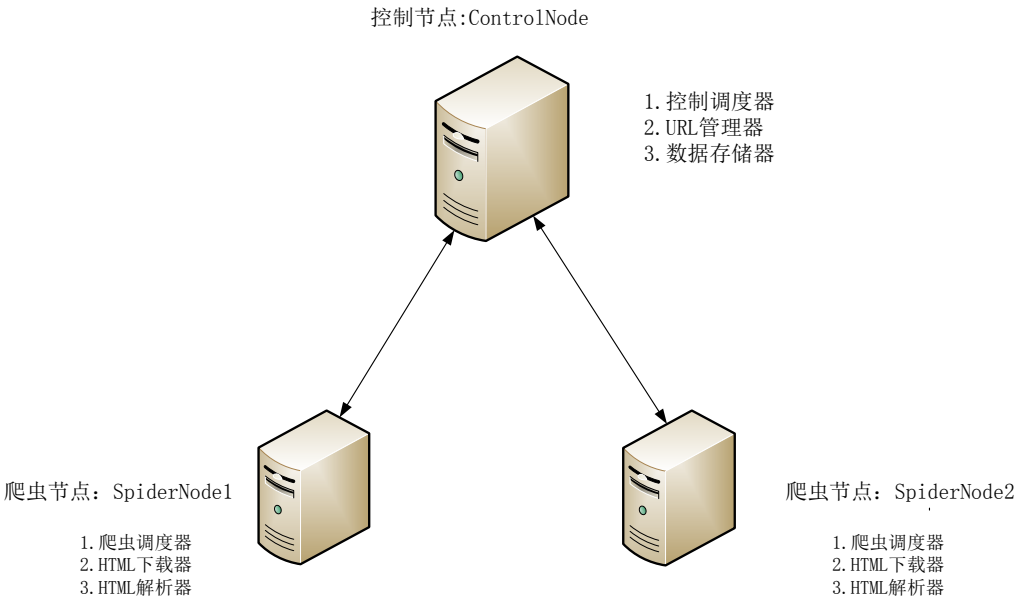


图 7.1 主从爬虫结构

## 7.2 控制节点 ControlNode

控制节点主要分为 URL 管理器、数据存储器和控制调度器。控制调度器通过三个进程来协调 URL 管理器和数据存储器的工作，一个是 URL 管理进程，负责 URL 的管理和将 URL 传递给爬虫节点，一个是数据提取进程，负责读取爬虫节点返回的数据，将返回数据中的 URL 交给 URL 管理进程，将标题和摘要等数据交给数据存储进程，最后一个是数据存储进程，负责将数据提取进程中提交的数据进行本地存储。执行流程如图 7.2 所示：

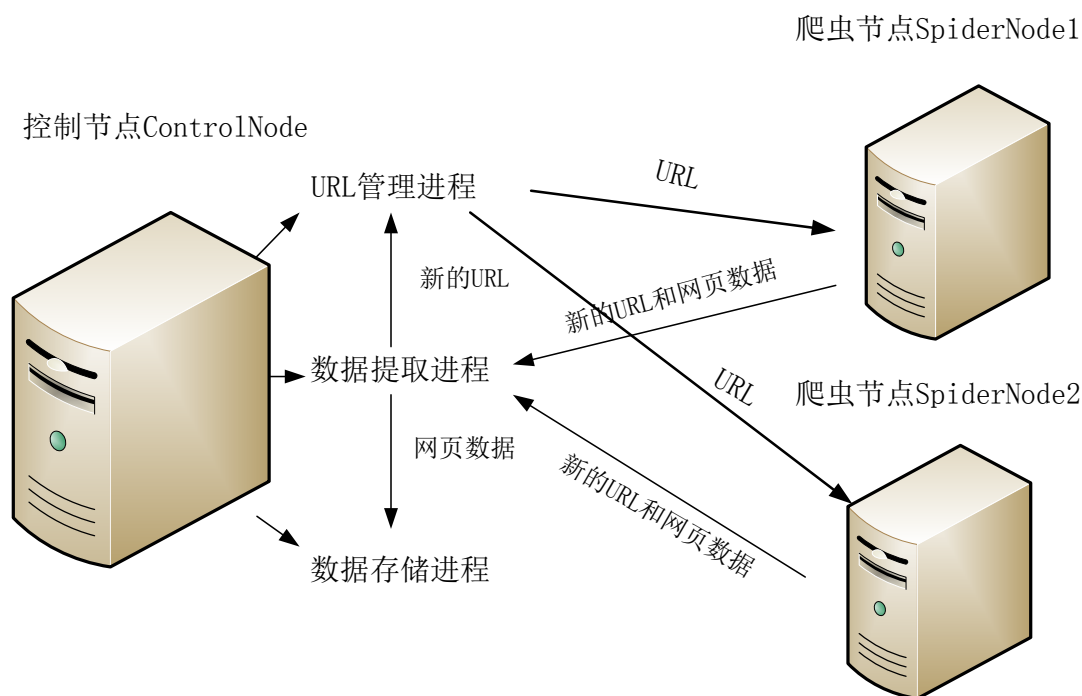


图 7.2 控制节点执行流程

### 7.2.1 URL 管理器

URL 管理器查考第六章的代码，做了一些优化修改。由于我们采用 set 内存去重的方式，如果直接存储大量的 URL 链接，尤其是 URL 链接很长的时候，很容易造成内存溢出，所以我们采用将爬取过的 URL 进行 MD5 处理，由于字符串经过 MD5 处理后的信息摘要长度可以 128bit，将生成的 MD5 摘要存储到 set 后，可以减少好几倍的内存消耗，不过 Python 中的 MD5 算法生成的是 256bit，取中间的 128 位即可。同时添加了 save\_progress 和 load\_progress 方法进行序列化的操作，将未爬取 URL 集合和已爬取的 URL 集合序列化到本地，保存当前的进度，以便下次恢复状态。URL 管理器 URLManager.py 代码如下：

```
#coding:utf-8
import cPickle
import hashlib
```

---

```
class UrlManager(object):
    def __init__(self):
        self.new_urls = self.load_progress('new_urls.txt')#未爬取 URL 集合
        self.old_urls = self.load_progress('old_urls.txt')#已爬取 URL 集合
    def has_new_url(self):
        """
        判断是否有未爬取的 URL
        :return:
        """
        return self.new_url_size() != 0

    def get_new_url(self):
        """
        获取一个未爬取的 URL
        :return:
        """
        new_url = self.new_urls.pop()
        m = hashlib.md5()
        m.update(new_url)
        self.old_urls.add(m.hexdigest()[8:-8])
        return new_url

    def add_new_url(self, url):
        """
        将新的 URL 添加到未爬取的 URL 集合中
        :param url: 单个 URL
        :return:
        """
        if url is None:
            return
        m = hashlib.md5()
        m.update(url)
        url_md5 = m.hexdigest()[8:-8]
        if url not in self.new_urls and url_md5 not in self.old_urls:
            self.new_urls.add(url)

    def add_new_urls(self, urls):
        """
        将新的 URLS 添加到未爬取的 URL 集合中
        :param urls: url 集合
        :return:
        """
        if urls is None or len(urls) == 0:
            return
        for url in urls:
            self.add_new_url(url)

    def new_url_size(self):
        """
        获取未爬取 URL 集合的 s 大小
```



---

```

        :return:
        """
        return len(self.new_urls)

def old_url_size(self):
    """
    获取已经爬取 URL 集合的大小
    :return:
    """
    return len(self.old_urls)

def save_progress(self,path,data):
    """
    保存进度
    :param path: 文件路径
    :param data: 数据
    :return:
    """
    with open(path, 'wb') as f:
        cPickle.dump(data, f)

def load_progress(self,path):
    """
    从本地文件加载进度
    :param path: 文件路径
    :return: 返回 set 集合
    """
    print '[+] 从文件加载进度: %s' % path
    try:
        with open(path, 'rb') as f:
            tmp = cPickle.load(f)
            return tmp
    except:
        print '[!] 无进度文件, 创建: %s' % path
    return set()

```

## 7.2.2 数据存储器

数据存储器的内容基本上和第六章的一样, 不过生成文件按照当前时间进行命名避免重复, 同时对文件进行缓存写入。代码如下:

```

#coding:utf-8
import codecs
import time
class DataOutput(object):
    def __init__(self):
        self.filepath='baike_%.html'%(time.strftime("%Y_%m_%d_%H_%M_%S", time.localtime()))
        self.output_head(self.filepath)
        self.datas=[]

```

---

```
def store_data(self,data):
    if data is None:
        return
    self.datas.append(data)
    if len(self.datas)>10:
        self.output_html(self.filepath)

def output_head(self,path):
    """
    将 HTML 头写进去
    :return:
    """
    fout=codecs.open(path,'w',encoding='utf-8')
    fout.write("<html>")
    fout.write("<body>")
    fout.write("<table>")
    fout.close()

def output_html(self,path):
    """
    将数据写入 HTML 文件中
    :param path: 文件路径
    :return:
    """
    fout=codecs.open(path,'a',encoding='utf-8')
    for data in self.datas:
        fout.write("<tr>")
        fout.write("<td>%s</td>"%data['url'])
        fout.write("<td>%s</td>"%data['title'])
        fout.write("<td>%s</td>"%data['summary'])
        fout.write("</tr>")
        self.datas.remove(data)
    fout.close()

def output_end(self,path):
    """
    输出 HTML 结束
    :param path: 文件存储路径
    :return:
    """
    fout=codecs.open(path,'a',encoding='utf-8')
    fout.write("</table>")
    fout.write("</body>")
    fout.write("</html>")
    fout.close()
```

---

## 7.2.3 控制调度器

控制调度器主要是产生并启动 URL 管理进程、数据提取进程和数据存储进程，同时维护 4 个队列保持进程间的通信，分别为 url\_queue, result\_queue, conn\_q, store\_q。4 个队列说明如下：

- url\_q 队列是 URL 管理进程将 URL 传递给爬虫节点的通道
- result\_q 队列是爬虫节点将数据返回给数据提取进程的通道
- conn\_q 队列是数据提取进程将新的 URL 数据提交给 URL 管理进程的通道
- store\_q 队列是数据提取进程将获取到的数据交给数据存储进程的通道

因为要和工作节点进行通信，所以分布式进程必不可少。参考 1.4.4 小节分布式进程中服务进程中的代码(linux 版)，创建一个分布式管理器，定义为 start\_manager 方法。方法代码如下：

```
def start_Manager(self,url_q,result_q):
'''
    创建一个分布式管理器
:param url_q: url 队列
:param result_q: 结果队列
:return:
'''
#把创建的两个队列注册在网络上，利用 register 方法，callable 参数关联了 Queue 对象，
# 将 Queue 对象在网络中暴露
BaseManager.register('get_task_queue',callable=lambda:url_q)
BaseManager.register('get_result_queue',callable=lambda:result_q)
#绑定端口 8001，设置验证口令 'baike'。这个相当于对象的初始化
manager=BaseManager(address=('',8001),authkey='baike')
#返回 manager 对象
return manager
```

URL 管理进程将从 conn\_q 队列获取到的新 URL 提交给 URL 管理器，经过去重之后，取出 URL 放入 url\_queue 队列中传递给爬虫节点，代码如下：

```
def url_manager_proc(self,url_q,conn_q,root_url):
url_manager = UrlManager()
url_manager.add_new_url(root_url)
while True:
    while(url_manager.has_new_url()):

        #从 URL 管理器获取新的 url
        new_url = url_manager.get_new_url()
        #将新的 URL 发给工作节点
        url_q.put(new_url)
        print 'old_url=',url_manager.old_url_size()
        #加一个判断条件，当爬去 2000 个链接后就关闭,并保存进度
        if(url_manager.old_url_size()>2000):
            #通知爬行节点工作结束
            url_q.put('end')
            print '控制节点发起结束通知!'
```

---

```
        #关闭管理节点，同时存储set 状态
        url_manager.save_progress('new_urls.txt',url_manager.new_urls)
        url_manager.save_progress('old_urls.txt',url_manager.old_urls)
        return

#将从 result_solve_proc 获取到的 urls 添加到 URL 管理器之间
try:
    if not conn_q.empty():
        urls = conn_q.get()
        url_manager.add_new_urls(urls)
except BaseException,e:
    time.sleep(0.1)#延时休息
```

数据提取进程从 result\_queue 队列读取返回的数据，并将数据中的 URL 添加到 conn\_q 队列交给 URL 管理进程，将数据中的文章标题和摘要添加到 store\_q 队列交给数据存储进程。代码如下：

```
def result_solve_proc(self,result_q,conn_q,store_q):
while(True):
    try:
        if not result_q.empty():
            content = result_q.get(True)
            if content['new_urls']=='end':
                #结果分析进程接受通知然后结束
                print '结果分析进程接受通知然后结束!'
                store_q.put('end')
                return
            conn_q.put(content['new_urls'])#url 为set 类型
            store_q.put(content['data'])#解析出来的数据为 dict 类型
        else:
            time.sleep(0.1)#延时休息
    except BaseException,e:
        time.sleep(0.1)#延时休息
```

数据存储进程从 store\_q 队列中读取数据，并调用数据存储器进行数据存储。代码如下：

```
def store_proc(self,store_q):
output = DataOutput()
while True:
    if not store_q.empty():
        data = store_q.get()
        if data=='end':
            print '存储进程接受通知然后结束!'
            output.output_end(output.filepath)

        return
    output.store_data(data)
    else:
        time.sleep(0.1)
```

最后将分布式管理器、URL 管理进程、数据提取进程和数据存储进程进行启动，并

---

初始化 4 个队列。代码如下：

```
if __name__=='__main__':
    #初始化 4 个队列
    url_q = Queue()
    result_q = Queue()
    store_q = Queue()
    conn_q = Queue()
    #创建分布式管理器
    node = NodeManager()
    manager = node.start_Manager(url_q,result_q)
    #创建 URL 管理进程、 数据提取进程和数据存储进程
    url_manager_proc = Process(target=node.url_manager_proc,
    args=(url_q,conn_q,'http://baike.baidu.com/view/284853.htm',))
    result_solve_proc = Process(target=node.result_solve_proc, args=(result_q,conn_q,store_q,))
    store_proc = Process(target=node.store_proc, args=(store_q,))
    #启动 3 个进程和分布式管理器
    url_manager_proc.start()
    result_solve_proc.start()
    store_proc.start()
    manager.get_server().serve_forever()
```

## 7.3 爬虫节点 SpiderNode

爬虫节点相对简单，主要包含 HTML 下载器、HTML 解析器和爬虫调度器。执行流程如下：

- 爬虫调度器从控制节点中的 url\_q 队列读取 URL
- 爬虫调度器调用 HTML 下载器、HTML 解析器获取网页中新的 URL 和标题摘要
- 最后爬虫调度器将新的 URL 和标题摘要传入 result\_q 队列交给控制节点

### 7.3.1 HTML 下载器

HTML 下载器的代码和第六章的一致，只需要注意网页编码即可。代码如下：

```
#coding:utf-8
import requests
class HtmlDownloader(object):

    def download(self,url):
        if url is None:
            return None
        user_agent = 'Mozilla/4.0 (compatible; MSIE 5.5; Windows NT)'
        headers={'User-Agent':user_agent}
        r = requests.get(url,headers=headers)
        if r.status_code==200:
            r.encoding='utf-8'
            return r.text
        return None
```

---

## 7. 3. 2HTML 解析器

HTML 解析器的代码和第六章的一致，详细的网页分析过程可以回顾第六章。代码如下：

```
#coding:utf-8
import re
import urlparse
from bs4 import BeautifulSoup

class HtmlParser(object):

    def parser(self,page_url,html_cont):
        """
        用于解析网页内容抽取 URL 和数据
        :param page_url: 下载页面的 URL
        :param html_cont: 下载的网页内容
        :return: 返回 URL 和数据
        """
        if page_url is None or html_cont is None:
            return
        soup = BeautifulSoup(html_cont,'html.parser',from_encoding='utf-8')
        new_urls = self._get_new_urls(page_url,soup)
        new_data = self._get_new_data(page_url,soup)
        return new_urls,new_data

    def _get_new_urls(self,page_url,soup):
        """
        抽取新的 URL 集合
        :param page_url: 下载页面的 URL
        :param soup:soup
        :return: 返回新的 URL 集合
        """
        new_urls = set()
        #抽取符合要求的a 标签
        links = soup.find_all('a',href=re.compile(r'/view/\d+\..htm'))
        for link in links:
            #提取 href 属性
            new_url = link['href']
            #拼接成完整网址
            new_full_url = urlparse.urljoin(page_url,new_url)
            new_urls.add(new_full_url)
        return new_urls

    def _get_new_data(self,self,page_url,soup):
        """
        抽取有效数据
        :param page_url: 下载页面的 URL
        :param soup:
```

---

```
:return: 返回有效数据
'''
data={}
data['url']=page_url
title = soup.find('dd',class_='lemmaWgt-lemmaTitle-title').find('h1')
data['title']=title.get_text()
summary = soup.find('div',class_='lemma-summary')
# 获取 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
data['summary']=summary.get_text()
return data
```

### 7.3.3 爬虫调度器

爬虫调度器需要用到分布式进程中工作进程的代码，具体内容可以参考第一章的分布式进程章节。爬虫调度器需要先连接上控制节点，然后依次完成从 url\_q 队列中获取 URL，下载并解析网页，将获取的数据交给 result\_q 队列，返回给控制节点等各项任务，代码如下：

```
class SpiderWork(object):
def __init__(self):
    # 初始化分布式进程中的工作节点的连接工作
    # 实现第一步：使用 BaseManager 注册获取 Queue 的方法名称
    BaseManager.register('get_task_queue')
    BaseManager.register('get_result_queue')
    # 实现第二步：连接到服务器:
    server_addr = '127.0.0.1'
    print('Connect to server %s...' % server_addr)
    # 端口和验证口令注意保持与服务进程设置的完全一致:
    self.m = BaseManager(address=(server_addr, 8001), authkey='baike')
    # 从网络连接:
    self.m.connect()
    # 实现第三步：获取 Queue 的对象:
    self.task = self.m.get_task_queue()
    self.result = self.m.get_result_queue()
    # 初始化网页下载器和解析器
    self.downloader = HtmlDownloader()
    self.parser = HtmlParser()
    print 'init finish'

def crawl(self):
    while(True):
        try:
            if not self.task.empty():
                url = self.task.get()

                if url == 'end':
                    print '控制节点通知爬虫节点停止工作...'
                    # 接着通知其它节点停止工作
```

---

```
        self.result.put({'new_urls':'end','data':'end'})
        return
    print '爬虫节点正在解析:%s'%url.encode('utf-8')
    content = self.downloader.download(url)
    new_urls,data = self.parser.parser(url,content)
    self.result.put({"new_urls":new_urls,"data":data})
except EOFError,e:
    print "连接工作节点失败"
    return
except Exception,e:
    print e
    print 'Crawl  fail '

if __name__=="__main__":
    spider = SpiderWork()
    spider.crawl()
```

在爬虫调度器设置了一个本地 IP: 127.0.0.1, 大家可以将在一台机器上测试代码的正确性。当然也可以使用三台 VPS 服务器, 两台运行爬虫节点程序, 将 IP 改为控制节点主机的公网 IP, 一台运行控制节点程序, 进行分布式爬取, 这样更贴近真实的爬取环境。下面图 7.3 为最终爬取的数据, 图 7.4 为 new\_urls.txt 内容, 图 7.5 为 old\_urls.txt 内容, 大家可以进行对比测试, 这个简单的分布式爬虫还有很大发挥的空间, 希望大家发挥自己的聪明才智进一步完善。





```
c__builtin__
set
p1
((lp2
Vhttp://baike.baidu.com/view/404402.htm
p3
aVhttp://baike.baidu.com/view/1765562.htm
p4
aVhttp://baike.baidu.com/view/381469.htm
p5
aVhttp://baike.baidu.com/view/1117503.htm
p6
aVhttp://baike.baidu.com/view/166248.htm
p7
aVhttp://baike.baidu.com/view/593053.htm
p8
aVhttp://baike.baidu.com/view/167593.htm
p9
aVhttp://baike.baidu.com/view/82343.htm
p10
aVhttp://baike.baidu.com/view/1106925.htm
p11
aVhttp://baike.baidu.com/view/2205439.htm
p12
aVhttp://baike.baidu.com/view/87781.htm
p13
aVhttp://baike.baidu.com/view/649188.htm
p14
aVhttp://baike.baidu.com/view/2147419.htm
p15
aVhttp://baike.baidu.com/view/576327.htm
```

图 7.4 new\_urls.txt

```
c__builtin__
set
p1
((lp2
S'0d3fc7ed58aed814'
p3
aS'0dec0af6d6c26bc8'
p4
aS'b642a68d43457326'
p5
aS'1192ed1347b2d15e'
p6
aS'711109c4b4df5017'
p7
aS'b1ce07d693a11a8d'
p8
aS'cb918f37fce954bb'
p9
aS'432ed15851e1e31a'
p10
aS'fe2177cf37985908'
p11
aS'540b6606c1339741'
p12
aS'fa745ba7eca67e28'
p13
aS'215ebf8bd4c1cf6e'
p14
aS'0e35077f014394fd'
p15
```

图 7.5 old\_urls.txt

## 7.4 小结

本章讲解了一个简单的分布式爬虫结构，主要目的是帮助大家将 Python 爬虫基础篇的知识进行总结和强化，开拓大家的思维，同时也让大家知道分布式爬虫并不是这么高不可攀。不过当你亲手打造一个分布式爬虫后，就会知道分布式爬虫的难点在于节点的调度，什么样的结构能让各个节点稳定高效的运作才是分布式爬虫要考虑的核心内容。到本章为止，Python 爬虫基础篇已经结束，这个时候大家基本上可以编写简单的爬虫，爬取一些静态网站的内容，但是 Python 爬虫开发不仅如此，大家接着往下学习吧。

购买链接:<https://item.jd.com/12206762.html>

