# 第 1 部分 Xpath

1. XPath 是一门在 XML 文档中查找信息的语言，用来在 XML 文档中对元素和属性进行遍历。
2. XPath 使用路径表达式在 XML 文档中选取节点。节点是通过沿着路径或者 step 来选取的。下面列出了最有用的路径表达式：

| 表达式 | 描述 |
| --- | --- |
| nodename | 选取此节点的所有子节点。 |
| / | 从根节点选取。 |
| // | 从匹配选择的当前节点选择文档中的节点，而不考虑它们的位置。 |
| . | 选取当前节点。 |
| .. | 选取当前节点的父节点。 |
| @ | 选取属性。 |

在下面的表格中，我们已列出了一些路径表达式以及表达式的结果：

| 路径表达式 | 结果 |
| --- | --- |
| bookstore | 选取 bookstore 元素的所有子节点。 |
| /bookstore | 选取根元素 bookstore。<br>注释：假如路径起始于正斜杠( / )，则此路径始终代表到某元素的绝对路径！ |
| bookstore/book | 选取属于 bookstore 的子元素的所有 book 元素。 |
| //book | 选取所有 book 子元素，而不管它们在文档中的位置。 |
| bookstore//book | 选择属于 bookstore 元素的后代的所有 book 元素，而不管它们位于 bookstore 之下的什么位置。 |
| //@lang | 选取名为 lang 的所有属性。 |

3. 谓语用来查找某个特定的节点或者包含某个指定的值的节点。谓语被嵌在方括号中。在下面的表格中，我们列出了带有谓语的一些路径表达式，以及表达式的结果：

| 路径表达式 | 结果 |
| --- | --- |
| /bookstore/book[1] | 选取属于 bookstore 子元素的第一个 book 元素。 |
| /bookstore/book[last()] | 选取属于 bookstore 子元素的最后一个 book 元素。 |
| /bookstore/book[last()-1] | 选取属于 bookstore 子元素的倒数第二个 book 元素。 |
| /bookstore/book[position()<3] | 选取最前面的两个属于 bookstore 元素的子元素的 book 元素。 |
| //title[@lang] | 选取所有拥有名为 lang 的属性的 title 元素。 |
| //title[@lang='eng'] | 选取所有 title 元素，且这些元素拥有值为 eng 的 lang 属性。 |
| /bookstore/book[price>35.00] | 选取 bookstore 元素的所有 book 元素，且其中的 price 元素的值须大于 35.00。 |
| /bookstore/ | 选取 bookstore 元素中的 book 元素的所有 title 元素，且 |

| | |
|---|---|
| book[price>35.00]/title | 其中的 price 元素的值须大于 35.00。 |

| 路径表达式 | 结果 |
|---|---|
| /bookstore/book[1] | 选取属于 bookstore 子元素的第一个 book 元素。 |
| /bookstore/book[last()] | 选取属于 bookstore 子元素的最后一个 book 元素。 |
| /bookstore/book[last()-1] | 选取属于 bookstore 子元素的倒数第二个 book 元素。 |
| /bookstore/book[position()<3] | 选取最前面的两个属于 bookstore 元素的子元素的 book 元素。 |
| //title[@lang] | 选取所有拥有名为 lang 的属性的 title 元素。 |
| //title[@lang='eng'] | 选取所有 title 元素，且这些元素拥有值为 eng 的 lang 属性。 |
| /bookstore/book[price>35.00] | 选取 bookstore 元素的所有 book 元素，且其中的 price 元素的值须大于 35.00。 |
| /bookstore/book[price>35.00]/title | 选取 bookstore 元素中的 book 元素的所有 title 元素，且其中的 price 元素的值须大于 35.00。 |

4. XPath 可以使用通配符：

| 路径表达式 | 结果 |
|---|---|
| /bookstore/* | 选取 bookstore 元素的所有子元素。 |
| //* | 选取文档中的所有元素。 |
| //title[@*] | 选取所有带有属性的 title 元素。 |

5. 通过在路径表达式中使用运算符 “|”，可以选取若干个路径。

| 路径表达式 | 结果 |
|---|---|
| //book/title \| //book/price | 选取 book 元素的所有 title 和 price 元素。 |
| //title \| //price | 选取文档中的所有 title 和 price 元素。 |
| /bookstore/book/title \| //price | 选取属于 bookstore 元素的 book 元素的所有 title 元素，以及文档中所有的 price 元素。 |

6. 有些网页并没有很多元素并没有 id、name 等属性，这时可以使用轴来查找特定的元素，轴的使用方式为“轴::标签名”。xpath 有以下轴：

| 轴名称 | 结果 |
| --- | --- |
| ancestor | 选取当前节点的所有先辈（父、祖父等）。 |
| ancestor-or-self | 选取当前节点的所有先辈（父、祖父等）以及当前节点本身。 |
| attribute | 选取当前节点的所有属性。 |
| child | 选取当前节点的所有子元素。 |
| descendant | 选取当前节点的所有后代元素（子、孙等）。 |
| descendant-or-self | 选取当前节点的所有后代元素（子、孙等）以及当前节点本身。 |
| following | 选取文档中当前节点的结束标签之后的所有节点。 |
| namespace | 选取当前节点的所有命名空间节点。 |
| parent | 选取当前节点的父节点。 |
| preceding | 选取文档中当前节点的开始标签之前的所有节点。 |
| preceding-sibling | 选取当前节点之前的所有同级节点。 |
| self | 选取当前节点。 |

例如：

```
//div[contains(text(),'产证地址')]/parent::td
```

选择指定 div 节点的父亲节点，该父节点必须为 td 标签。

```
//div[contains(text(),'产证地址')]//ancestor::table
```

选择指定 div 节点的所有祖先<table>标签，注意，它只选择祖先节点，但并不匹配祖先节点的兄弟节点。

7．轴的使用举例：

| 例子 | 结果 |
| --- | --- |
| child::book | 选取所有属于当前节点的子元素的 book 节点。 |
| attribute::lang | 选取当前节点的 lang 属性。 |
| child::* | 选取当前节点的所有子元素。 |
| attribute::* | 选取当前节点的所有属性。 |
| child::text() | 选取当前节点的所有文本子节点。 |
| child::node() | 选取当前节点的所有子节点。 |
| descendant::book | 选取当前节点的所有 book 后代。 |
| ancestor::book | 选择当前节点的所有 book 先辈。 |
| ancestor-or-self::book | 选取当前节点的所有 book 先辈以及当前节点（如果此节点是 book 节点） |
| child::*/child::price | 选取当前节点的所有 price 孙节点。 |

8．xpath 的位置路径可以是绝对的，也可以是相对的。

# 第 2 部分 Scrapy

1. Scrapy 爬取的循环类似下文:

(1) 以初始的 URL 初始化 Request，并设置回调函数。 当该 request 下载完毕并返回时，将生成 response，并作为参数传给该回调函数。

(2) 在回调函数内分析返回的网页内容，返回 Item 对象或者 Request 或者一个包括二者的可迭代容器。 返回的 Request 对象之后会经过 Scrapy 处理，下载相应的内容，并调用设置的 callback 函数。

(3) 在回调函数内，您可以使用选择器来分析网页内容，并根据分析的数据生成 item。

(4) 最后，由 spider 返回的 item 将被存到数据库或使用 Feed exports 存入到文件中。

2. 使用 scrapy startproject 命令来创建 scrapy 项目:

```
scrapy startproject tutorial
```

该命令创建的文件和目录如下:

```
tutorial/
   scrapy.cfg
   tutorial/
      __init__.py
      items.py
      pipelines.py
      settings.py
      spiders/
         __init__.py
         ...
```

3. Item 是保存爬取到的数据的类，其使用方法和 python 字典类似。

```
import scrapy

class DmozItem(scrapy.Item):
   title = scrapy.Field()
   link = scrapy.Field()
   desc = scrapy.Field()
```

创建一个保存标题、标签的描述的类，后面的 Field 表示该保存的数据的类型

4. CustomSpider 是用户实现的爬虫类。其包含了一个用于下载的初始 URL，如何跟进网页中的链接以及如何分析页面中的内容，提取生成 item 的方法。用户必须继承

自 scrapy.Spider 类，且定义一些属性:

- name：爬虫名。 该名字必须是唯一的，您不可以为不同的 CustomSpider 设定相同的爬虫名。
- start_urls:：爬虫启动时的初始 url 列表。
- parse()： 该函数用于解析爬取到的页面数据，提取数据以及生成需要进一步处理的 URL 的 Request 对象。

```
class DmozSpider(Spider):
        name = "dmoz"
        allowed_domains = ["dmoz.org"]
        start_urls = [
                "http://www.dmoz.org/Computers/Programming/Languages/Python/Books/",

"http://www.dmoz.org/Computers/Programming/Languages/Python/Resources/"
        ]

        def parse(self, response):
                filename = response.url.split("/")[-2]
                open(filename, 'wb').write(response.body)
```

5. 进入项目的根目录，执行下列命令启动 CustomSpider:

```
scrapy crawl CustomSpider
```

该命令启动了我们刚刚添加的 CustomSpider，向目标地址发送一些请求。

6. Scrapy 终端是一个交互终端，在未启动 spider 的情况下尝试及调试爬取代码。 其本意是用来测试提取数据的代码，不过可以将其作为正常的 Python 终端，在上面测试任何的 Python 代码。

```
scrapy shell "http://www.dmoz.org/Computers/Programming/Languages/Python/Books/"
```

scrapy shell 后接爬虫的初始链接。终端运行 Scrapy 时，一定要为 url 地址加上引号，否则包含参数的 url 会导致 Scrapy 运行失败。

7. 选择器用于从 html 页面中提取数据，Scrapy 通过特定的 XPath 或者 CSS 表达式来提取 HTML 文件中的数据。
8. xpath()用于匹配特定的 html 节点，例如，匹配 html 文件的所有的<li>元素:

```
response.xpath('//ul/li')
```

9. extract()用于提取数据。

```
<html>
<head>
        <base href='http://example.com/' />
        <title>Example website</title>
</head>
<body>
        <div id='images'>
                <a href='image1.html'>Name: My image 1 <br /><img
src='image1_thumb.jpg' /></a>
                <a href='image2.html'>Name: My image 2 <br /><img
src='image2_thumb.jpg' /></a>
                <a href='image3.html'>Name: My image 3 <br /><img
src='image3_thumb.jpg' /></a>
                <a href='image4.html'>Name: My image 4 <br /><img
src='image4_thumb.jpg' /></a>
                <a href='image5.html'>Name: My image 5 <br /><img
src='image5_thumb.jpg' /></a>
        </div>
</body>
</html>
```

提取网站的标题：

```
esponse.xpath('//title/text()').extract()
```

输出为 Example website。

10. 爬虫获取数据后得到一个 Response 对象，其中有 header、body 等元素。

11. 使用-o 参数将爬取到的数据采用 JSON 格式进行序列化，并保存为指定的 json 文件：

```
scrapy crawl dmoz -o items.json
```

将爬取到的数据保存到 items.json 文件。

12. 默认的 Scrapy 项目结构：

```
scrapy.cfg            项目的配置文件
myproject/            该项目的 python 模块。之后您将在此加入代码。
  __init__.py
  items.py          项目中的 item 文件
  pipelines.py      项目中的 pipelines 文件
  settings.py
  spiders/          放置 spider 代码的目录
    __init__.py
    spider1.py
    spider2.py
    ...
```

scrapy.cfg 文件所在的目录为项目的根目录 。该文件中包含了项目设置信息。例如:

setting.py

```
BOT_NAME = 'lusongsong'

SPIDER_MODULES = ['lusongsong.spiders']
NEWSPIDER_MODULE = 'lusongsong.spiders'
FEED_EXPORT_ENCODING = 'utf-8'
```

13. 在项目中使用 scrapy 工具来对其进行控制和管理， scrapy 工具包含各种 scrapy 命令，例如 scrapy genspider 命令等。
14. 一个项目可以包含多个爬虫，scrapy genspider 命令用于创建一个新的 spider:

```
scrapy genspider mydomain mydomain.com
```

其中 mydomain 是爬虫名，mydomain.com 是爬虫要爬的地址。

15. Scrapy 提供了两种类型的命令。一种必须在 Scrapy 项目中运行，另外一种是全局命令。使用-h 查看命令的详细内容:

```
scrapy startproject  -h
```

16. 相关命令:

- startproject：创建 scrapy 项目。

```
$ scrapy startproject myproject
```

- Genspider：在当前项目中创建 spider

```
scrapy genspider mydomain mydomain.com
```

。这仅仅是创建 spider 的一种快捷方法。该方法可以使用提前定义好的模板来生成 spider。您也可以自己创建 spider 的源码文件。

- Crawl：使用 spider 进行爬取。

```
$ scrapy crawl myspider
[ ... myspider starts crawling ... ]
```

- check：运行 contract 检查。

```
scrapy check myspider
```

- List：列出当前项目中所有可用的 spider，每行输出一个 spider。

```
$ scrapy list
spider1
```

```
spider2
```

- fetch：使用 Scrapy 下载器下载给定的 URL，并将获取到的内容送到标准输出。

```
$ scrapy fetch --nolog http://www.example.com/some/page.html
```

如果 spider 修改 user_agent 属性，该命令将会使用该属性。如果不是在项目中运行，则该命令会使用默认 Scrapy downloader。

```
$ scrapy fetch --nolog http://www.example.com/some/page.html
```

--nolog 参数用于关闭 log 输出。

- 注意，该命令必须加上 http 头，否则会出错。

- View：在浏览器中打开给定的 URL，并以 Scrapy spider 获取到的形式展现。

```
$ scrapy view http://www.example.com/some/page.html
```

有时 spider 获取到的页面和普通用户看到的并不相同,因此该命令可以用来检查 spider 所获取到的页面，确认是我们所希望获取的页面。

- shell：Scrapy shell，后面可以接一个 url， 查看 Scrapy 终端获取更多信息。

```
$ scrapy shell http://www.example.com/some/page.html
[ ... scrapy shell starts ... ]
```

- Runspider：在未创建项目的情况下，运行一个编写在 Python 文件中的 spider。

```
scrapy runspider spiderfile.py
```

16. Scrapy 的 Item 类用于从非结构性的数据源提取结构性数据，例如网页。Item 使用简单的 class 定义语法以及 Field 对象来声明。例如:

```
import scrapy

class Product(scrapy.Item):
   name = scrapy.Field()
   price = scrapy.Field()
   stock = scrapy.Field()
   last_updated = scrapy.Field(serializer=str)
```

Item 字段：Field 对象指明了每个字段的元数据。Field 仅仅是内置的 dict 类的一个别名，并没有提供额外的方法或者属性。换句话说， Field 对象完完全全就是 Python 字典。被用来基于类属性的方法来支持 item 声明语法 。

17. Spider 类定义爬取的动作和分析网页内容。

18. 在运行 crawl 时使用-a 可以向项目传递 Spider 参数:

```
scrapy crawl myspider -a category=electronics
```

Spider 在构造函数中获取参数:

```
import scrapy

class MySpider(Spider):
   name = 'myspider'

   def __init__(self, category=None, *args, **kwargs):
      super(MySpider, self).__init__(*args, **kwargs)
      self.start_urls = ['http://www.example.com/categories/%s' % category]
      # ...
```

19. Spider 类是最简单的爬虫类。每个其他的爬虫类必须继承自该类。 Spider 并没有提供什么特殊的功能。

- allowed_domains：包含了 spider 允许爬取的域名列表。 当 OffsiteMiddleware 启用时， 域名不在列表中的 URL 不会被跟进。
- name：定义 spider 名字。
- start_urls：URL 列表。当没有制定特定的 URL 时，spider 将从该列表中开始进行爬取。 因此，第一个被获取到的页面的 URL 将是该列表之一。 后续的 URL 将会从获取到的数据中提取。
- start_requests()：该方法用于从 start_urls 中读取 url，传入 make_requests_from_url 中生成一个 request。如果没有定义 start_urls，可以重写该方法。
- make_requests_from_url(url)：使用 url 生成 request 对象。 该方法在初始化 request 时被 start_requests() 调用，用于使用 url 生成 request。
- parse(response)：Scrapy 处理 response 的默认方法。
- log(message[, level, component])：使用 scrapy.log.msg() 方法记录 message。
- closed(reason)：当 spider 关闭时，该函数被调用。

```
class DmozSpider(Spider):
        name = "dmoz"
        allowed_domains = ["dmoz.org"]
        start_urls = [
                "http://www.dmoz.org/Computers/Programming/Languages/Python/
Books/",

"http://www.dmoz.org/Computers/Programming/Languages/Python/Resources/"
        ]

        def parse(self, response):
                filename = response.url.split("/")[-2]
```

```
                open(filename, 'wb').write(response.body)
```

20. CrawlSpider：爬取一般网站常用的 spider，该类提供了一些规则来提取链接。除了从 Spider 继承过来的属性外，其提供了一个新的属性:

- rules：Rule 对象的集合，每个 Rule 对象定义了如何提取页面中的链接。如果多个 Rule 匹配了相同的链接，则根据他们在 rule 中的顺序来调用，第一个匹配的 Rule 会被调用。
- parse_start_url(response)：用于处理服务器返回的 html 页面。 该方法的返回值并必须是一个 Item 对象、一个 Request 对象或者一个包含二者的可迭代对象。

```
import scrapy
from scrapy.contrib.spiders import CrawlSpider, Rule
from scrapy.contrib.linkextractors import LinkExtractor

class MySpider(CrawlSpider):
   name = 'example.com'
   allowed_domains = ['example.com']
   start_urls = ['http://www.example.com']

   rules = (
      # 提取匹配 'category.php' (但不匹配 'subsection.php') 的链接并跟进链接(没有
callback 意味着 follow 默认为 True)
      Rule(LinkExtractor(allow=('category\.php', ), deny=('subsection\.php', ))),

      # 提取匹配 'item.php' 的链接并使用 spider 的 parse_item 方法进行分析
      Rule(LinkExtractor(allow=('item\.php', )), callback='parse_item'),
   )

   def parse_item(self, response):
      self.log('Hi, this is an item page! %s' % response.url)

      item = scrapy.Item()
      item['id'] = response.xpath('//td[@id="item_id"]/text()').re(r'ID: (\d+)')
      item['name'] = response.xpath('//td[@id="item_name"]/text()').extract()
      item['description'] =
response.xpath('//td[@id="item_description"]/text()').extract()
      return item
```

21. Rule 对象用于定义 url 的匹配规则其构造函数如下:

```
Rule(link_extractor, callback=None, cb_kwargs=None, follow=None,
```

```
process_links=None, process_request=None)
```

- link_extractor：使用正则表达式的形式匹配特定的 url，表示提取哪些 url。
- callback：回调函数， 从 link_extractor 中每次获取到链接时都会调用该函数。
- cb_kwargs：传递到回调函数的参数的字典。
- follow：是一个布尔值，从 response 提取的链接是否需要跟进。
- process_links：是一个回调函数。，每次从 link_extractor 中获取到链接时都会调用该函数，该方法主要用来过滤。
- process_request：是一个回调函数，用来过滤 request，每次提取到 request 时都会调用该函数。

22. 当 request 的返回为 xml 页面时，使用 XMLFeedSpider 类处理。
23. SitemapSpider 是一个使用 Sitemaps 来获取爬取的 URL 的爬虫类。其支持嵌套的 sitemap，并且可以从 robots.txt 中获取 sitemap 的 url。

- sitemap_urls：包含您要爬取的 url 的 sitemap 的 url 列表。 您也可以指定为一个 robots.txt ，spider 会从中分析并提取 url。
- sitemap_rules：用于匹配特定的 url，并设置回调函数。

```
sitemap_rules = [('/product/', 'parse_product')]
```

- sitemap_follow：一个用于匹配要跟进的 sitemap 的正则表达式的列表。其仅仅被应用在使用 Sitemap index files 来指向其他 sitemap 文件的站点。
- sitemap_alternate_links：指定当一个 url 有可选的链接时，是否跟进。 有些非英文网站会在一个 url 块内提供其他语言的网站链接。例如:

```
<url>
   <loc>http://example.com/</loc>
   <xhtml:link rel="alternate" hreflang="de" href="http://example.com/de"/>
</url>
```

当 sitemap_alternate_links 设置时，两个 URL 都会被获取。 当 sitemap_alternate_links 关闭时，只有 http://example.com/ 会被获取。

24. Scrapy 选择器，用于通过特定的 XPath 或者 CSS 表达式来提取 HTML 中的数据。
25. response.xpath()用于匹配特定的 HTML 节点：

```
>>> response.xpath('//title/text()').extract()
[u'Example website']
```

26. response.css()类似于 css 中的选择器：


```
for item in response.css('div a'):
```

27. Selector 有四个基本的方法:

(1) xpath()：传入 xpath 表达式，返回该表达式所对应的所有节点列表 。

(2) css()：传入 CSS 表达式，返回该表达式所对应的所有节点列表.

(3) extract()：根据表达式提取相对应的文本数据。

```
response.xpath('//ul/li/text()').extract()
```

(4) re()：根据传入的正则表达式对数据进行提取数据。

28. Selector 也有一个 re() 方法，用来通过正则表达式来提取数据。然而，不同于使用 xpath() 或者 css() 方法，re() 方法返回 unicode 字符串的列表。所以无法构造嵌套式的 re() 调用。

```
>>> response.xpath('//a[contains(@href, "image")]/text()').re(r'Name:\s*(.*)')
[u'My image 1',
 u'My image 2',
 u'My image 3',
 u'My image 4',
 u'My image 5']
```

29. 在 Spider 中被收集到 Item 里之后，数据被传递到 Pipeline，通过自定义 Pipeline，可以对数据进行操作，例如：

- 清理 HTML 数据
- 验证爬取的数据

- 查重
- 将爬取结果保存到数据库中

30. 自定义 item pipeline 很简单，每个 item pipeline 组件是一个独立的 Python 类，同时必须实现以下方法：

- process_item(item, spider)：每个 item pipeline 组件都需要调用该方法，这个方法必须返回一个 Item 对象， 或是抛出 DropItem 异常，被丢弃的 item 将不会被之后的 pipeline 组件所处理。

  此外，他们也可以实现以下方法：

- open_spider(spider)：当 spider 被开启时，这个方法被调用。

  参数：Spider 对象

- close_spider(spider)：当 spider 被关闭时，这个方法被调用

  参数：Spider 对象

31. 在 setting 配置中添加类似如下配置启用自定义的 Item Pipeline 组件：

```
ITEM_PIPELINES = {
   'myproject.pipelines.PricePipeline': 300,
   'myproject.pipelines.JsonWriterPipeline': 800,
}
```

后面的数字表示运行顺序，item 按数字从低到高的顺序，通过 pipeline，通常将这些数字定义在 0-1000 范围内。

32. 下载中间件是用于全局修改 Scrapy request 和 response 的一个轻量、底层的系统。举个例子，当需要挂代理访问某个网站、需要修改爬虫的 User-Agent、或者需要带 cookie 访问某个网站的时候，下载器中间件的使用就成为了必要！

➢ 注意，下载中间件中的“下载”并非指下载文件，而是指下载页面。

33. 通过在 setting 中加入类似如下的配置来启用 spider 中间件，数字值为中间件的顺序。样例:

```
SPIDER_MIDDLEWARES = {
   'myproject.middlewares.CustomSpiderMiddleware': 543,
}
```

34. 编写 spider 中间件十分简单。每个中间件组件是一个定义了以下一个或多个方法的 SpiderMiddleware 类：

- process_spider_input(response, spider)：当 response 通过 spider 中间件时，该方法被调用，处理该 response。
- process_spider_output(response, result, spider)：当 Spider 处理 response 返回 result 时，该方法被调用。

- process_spider_exception(response, exception, spider)：当 spider 或(其他 spider 中间件的) process_spider_input() 跑出异常时， 该方法被调用。

34. spider 类回调函数可以通过生成器返回下一个要爬取的 URL，并返回解释的数据。

35. 内置的 downloader middleware 有 13 个：

(1) CookiesMiddleware：是否向 web server 发送 cookie

(2) DefaultHeadersMiddleware：将所有 request 的头设置为默认模式

(3) DownloadTimeoutMiddleware：设置 request 的 timeout

(4) HttpAuthMiddleware：对来自特定 spider 的 request 授权

(5) HttpCacheMiddleware：给 request&response 设置缓存策略

(6) HttpCompressionMiddleware：

(7) ChunkedTransferMiddleware：

(8) HttpProxyMiddleware：给所有 request 设置 http 代理

(9) RedirectMiddleware：处理 request 的重定向

(10) MetaRefreshMiddleware：根据 meta-refresh html tag 处理重定向

(11) RetryMiddleware：失败重试策略

(12) RobotsTxtMiddleware：robots 封禁处理

(13) UserAgentMiddleware：支持 user agent 重写

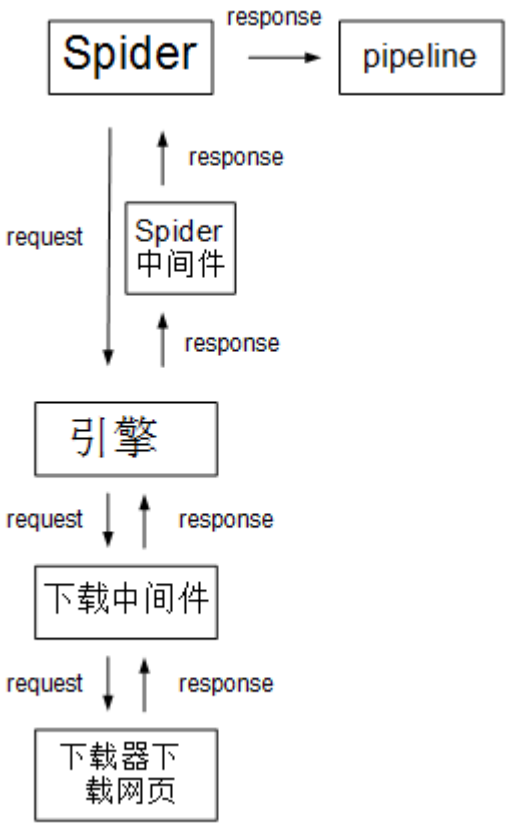
36. 默认情况下，Scrapy 使用深度优先顺序 。深度优先对大多数情况下是更方便的。如果您想以 广度优先顺序 进行爬取，你可以设置以下的设定：

```
DEPTH_PRIORITY = 1
SCHEDULER_DISK_QUEUE = 'scrapy.squeue.PickleFifoDiskQueue'
SCHEDULER_MEMORY_QUEUE = 'scrapy.squeue.FifoMemoryQueue'
```

37. 爬虫爬取网站时碰到 403 错误表示网站启用了防爬虫配置，这时需要设置爬虫的 setting 文件 USER_AGENT 属性：

```
USER_AGENT = 'Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/60.0.3112.113 Safari/537.36'
```

38. Scrapy 的整个数据处理流程由 Scrapy 引擎进行控制，其主要的运行方式为：




39.总结：

(1) spider 类用于从 response 中提取数据并返回下一个要获取的链接，这里可以指定回调函数。

(2) 连接数据库等存储数据由 pipeline 来完成。

(3) item 定义要爬取的数据类。

(4) setting 用于设置相关信息。

40.

# 第 3 部分 Selenium

1. 一个简单的 selenium 脚本

```
# coding = utf-8
from selenium import webdriver          #导入 webdriver
browser = webdriver.Firefox()           #指定浏览器
browser.get("http://www.baidu.com")
browser.find_element_by_id("kw").send_keys("selenium")      #查找指定 ID 并给 ID
发送内容
browser.find_element_by_id("su").click()          #查找指定 ID 并点击
browser.quit()                          #退出
```

2．webdriver 提供了一系列的对象定位方法，常用的有以下几种

(1) id

(2) name

(3) class name

(4) link text

(5) partial link text

(6) tag name

(7) xpath

(8) css selector

一个百度的输入框，可以用这么用种方式去定位：

```
<input id="kw" class="s_ipt" type="text" maxlength="100" name="wd"
autocomplete="off">
```

```
#coding=utf-8
from selenium import webdriver
browser = webdriver.Firefox()
browser.get("http://www.baidu.com")
##########百度输入框的定位方式###########
#通过 id 方式定位
browser.find_element_by_id("kw").send_keys("selenium")
#通过 name 方式定位
browser.find_element_by_name("wd").send_keys("selenium")
#通过 tag name 方式定位
browser.find_element_by_tag_name("input").send_keys("selenium")
#通过 class name 方式定位
browser.find_element_by_class_name("s_ipt").send_keys("selenium")
#通过 CSS 方式定位
browser.find_element_by_css_selector("#kw").send_keys("selenium")        #注意这
```

```
个有#号
#通过 xpath 方式定位
browser.find_element_by_xpath("//input[@id='kw']").send_keys("selenium")  #注意这个
##############################################
browser.find_element_by_id("su").click()
time.sleep(3)
browser.quit()
```

tag name 定位应该是所有定位方式中最不靠谱的一种了，因为在一个页面中具有相同 tag name 的元素极其容易出现。

3. id 和 name 是我们最最常用的定位方式，因为大多数控件都有这两个属性，而且在对控件的 id 和 name 命名时一般使其有意义也会取不同的名字。通过这两个属性使我们找一个页面上的属性变得相当容易。
4. CSS 比较灵活，上例选择取百度输入框的 id 属性来定义，同时也可以选择控件的其他任意属性：

- 可以取 name 属性：

```
<a href="http://news.baidu.com" name="tj_news">新闻</a>

driver.find_element_by_css_selector("a[name=\"tj_news\"]").click()
```

- 可以取 title 属性：

```
<a onclick="queryTab(this);" mon="col=502&pn=0" title="web" href="http://www.baidu.com/">网页</a>

driver.find_element_by_css_selector("a[title=\"web\"]").click()
```

- 也可以是取类：

```
<a class="RecycleBin xz" href="javascript:void(0);">

driver.find_element_by_css_selector("a.RecycleBin").click()
```

5. XPath 是一种在 XML 文档中定位元素的语言。因为 HTML 可以看做 XML 的一种实现，所以 selenium 用户可以使用这种强大语言在 web 应用中定位元素。
6. link 定位：有时候不是一个输入框也不是一个按钮，而是一个文字链接，我们可以通过 link 定位；

```
#coding=utf-8
from selenium import webdriver
browser = webdriver.Firefox()
browser.get("http://www.baidu.com")
browser.find_element_by_link_text("贴吧").click()
browser.quit()
```

7. Partial link text 定位：通过部分链接定位，这个有时候也会用到。拿上面的例子，我可以只用链接的一部分文字进行匹配：

```
browser.find_element_by_partial_link_text("贴").click()
```

8. 通过添加 implicitly_wait() 方法就可以方便的实现智能等待；implicitly_wait(30) 的用法应该比 time.sleep() 更智能，后者只能选择一个固定的时间的等待，前者可以在一个时间范围内智能的等待。用法:

```
browser.implicitly_wait(30)
```

示例：

```
# coding = utf-8
from selenium import webdriver
import time                                    #调入 time 函数
browser = webdriver.Firefox()
browser.get("http://www.baidu.com")
browser.implicitly_wait(30)                    #智能等待 30 秒
browser.find_element_by_id("kw").send_keys("selenium")
browser.find_element_by_id("su").click()
browser.quit()
```

9. 把刚才访问页面的 title 打印出来。

```
coding = utf-8
from selenium import webdriver
driver = webdriver.Chrome()
driver.get('http://www.baidu.com')
print driver.title                    # 把页面 title 打印出来
driver.quit()
```

10.打印 URL：

```
#coding=utf-8
from selenium import webdriver
import time
browser = webdriver.Firefox()
url= 'http://www.baidu.com'
#通过 get 方法获取当前 URL 打印
print "now access %s" %(url)
browser.get(url)
```

```
time.sleep(2)
browser.find_element_by_id("kw").send_keys("selenium")
browser.find_element_by_id("su").click()
time.sleep(3)
browser.quit()
```

11.浏览器最大化：

```
browser.get("http://www.baidu.com")
print "浏览器最大化"
browser.maximize_window()                    #将浏览器最大化显示
```

12.设置浏览器宽、高：

```
print "设置浏览器宽 480、高 800 显示"
browser.set_window_size(480, 800)
```

13.一般来说，webdriver 中比较常用的操作对象的方法有下面几个

- click 点击对象
- clear 清除对象的内容，如果可以的话
- send_keys 在对象上模拟按键输入
- submit 提交表单
- text 用于获取元素的文本信息

14.鼠标点击与键盘输入：

- send_keys("xx") 用于在一个输入框里输入 xx 内容。
- click() 用于点击一个按钮。
- clear() 用于清除输入框的内容，比如百度输入框里默认有个“请输入关键字”的信息，再比如我们的登陆框一般默认会有“账号”“密码”这样的默认信息。clear 可以帮助我们清除这些信息。

```
coding=utf-8
from selenium import webdriver
import time

driver = webdriver.Firefox()
driver.get("http://www.baidu.com")
driver.find_element_by_id("kw").clear()
driver.find_element_by_id("kw").send_keys("selenium")
```

```
time.sleep(2)
#通过 submit() 来操作
driver.find_element_by_id("su").submit()
time.sleep(3)
driver.quit()
```

15.submit 提交表单：

```
#coding=utf-8
from selenium import webdriver
import time
driver = webdriver.Firefox()
driver.get("http://www.baidu.com")
driver.find_element_by_id("kw").send_keys("selenium")
time.sleep(2)
#通过 submit() 来操作
driver.find_element_by_id("su").submit()
time.sleep(3)
driver.quit()
```

16.text 用于获取元素的文本信息：

```
data=driver.find_element_by_id("cp").text
```

17.get_attribute 用于获得属性值。

```
select = driver.find_element_by_tag_name("select")
allOptions = select.find_elements_by_tag_name("option")
for option in allOptions:
   print "Value is: " + option.get_attribute("value")
   option.click()
.....
```

18.通过 send_keys()调用按键：

- send_keys(Keys.TAB) # TAB
- send_keys(Keys.ENTER) # 回车

```
#coding=utf-8
from selenium import webdriver
from selenium.webdriver.common.keys import Keys #需要引入 keys 包
import os,time
```

```
driver = webdriver.Firefox()
driver.get("http://passport.kuaibo.com/login/?referrer=http%3A%2F
%2Fwebcloud
.kuaibo.com%2F")
time.sleep(3)
driver.maximize_window() # 浏览器全屏显示
driver.find_element_by_id("user_name").clear()
driver.find_element_by_id("user_name").send_keys("fnngj")
#tab 的定位相相于清除了密码框的默认提示信息，等同上面的 clear()
driver.find_element_by_id("user_name").send_keys(Keys.TAB)
time.sleep(3)
driver.find_element_by_id("user_pwd").send_keys("123456")
#通过定位密码框，enter 来代替登陆按钮
driver.find_element_by_id("user_pwd").send_keys(Keys.ENTER)
'''
#也可定位登陆按钮，通过 enter 代替 click()
driver.find_element_by_id("login").send_keys(Keys.ENTER)
'''
time.sleep(3)
driver.quit()
```

注意：这个操作和页面元素的遍历顺序有关，假如当前定位在账号输入框，按键盘的 tab 键后遍历的不是密码框，那就不法输入密码。假如输入密码后，还有需要填写验证码，那么回车也起不到登陆的效果。

19. 键盘组合键用法：

```
#ctrl+a 全选输入框内容
driver.find_element_by_id("kw").send_keys(Keys.CONTROL,'a')
time.sleep(3)

#ctrl+x 剪切输入框内容
driver.find_element_by_id("kw").send_keys(Keys.CONTROL,'x')
time.sleep(3)

#输入框重新输入内容，搜索
driver.find_element_by_id("kw").send_keys(u"虫师 cnblogs")
driver.find_element_by_id("su").click()
```

selenium2 python 在 send_keys()中输入中文一直报错，其实前面加个小 u 就解决了。

20. 鼠标事件 ActionChains 类

- context_click() 右击

- double_click() 双击
- drag_and_drop() 拖动

21. 鼠标右键

```
from selenium.webdriver.common.action_chains import ActionChains
……
#定位到要右击的元素
qqq=driver.find_element_by_xpath("/html/body/div/div[2]/div[2]/div/div[3]/table/tbody/tr/td[2]")
#对定位到的元素执行鼠标右键操作
ActionChains(driver).context_click(qqq).perform()
'''
#你也可以使用三行的写法，但我觉得上面两行写法更容易理解
chain = ActionChains(driver)
implement = driver.find_element_by_xpath("/html/body/div/div[2]/div[2]/div/div[3]/table/tbody/tr/td[2]")
chain.context_click(implement).perform()
'''
```

22. 鼠标双击：

```
#定位到要双击的元素
qqq =driver.find_element_by_xpath("xxx")
#对定位到的元素执行鼠标双击操作
ActionChains(driver).double_click(qqq).perform()
```

23. 鼠标拖放：

```
#定位元素的原位置
element = driver.find_element_by_name("source")
#定位元素要移动到的目标位置
target = driver.find_element_by_name("target")
#执行元素的移动操作
ActionChains(driver).drag_and_drop(element, target).perform()
```

24. webdriver 可以很方便的使用 findElement 方法来定位某个特定的对象，不过有时候我们却需要定位一组对象，这时候就需要使用 findElements 方法。定位一组对象一般用于以下场景：

- 批量操作对象，比如将页面上所有的 checkbox 都勾上

- 先获取一组对象，再在这组对象中过滤出需要具体定位的一些对象。比如定位出页面上所有的 checkbox，然后选择最后一个

```
# 选择所有的 checkbox 并全部勾上
checkboxes = dr.find_elements_by_css_selector('input[type=checkbox]')
for checkbox in checkboxes:
   checkbox.click()
   time.sleep(2)
# 打印当前页面上有多少个 checkbox
print len(dr.find_elements_by_css_selector('input[type=checkbox]'))
time.sleep(2)
```

25. 还有一个问题，有时候我们并不想勾选页面的所有的复选框，可以通过下面办法把最后一个被勾选的框去掉。如下：

```
# 选择所有的 checkbox 并全部勾上
checkboxes =
dr.find_elements_by_css_selector('input[type=checkbox]')
for checkbox in checkboxes:
   checkbox.click()
   time.sleep(2)
# 把页面上最后 1 个 checkbox 的勾给去掉
dr.find_elements_by_css_selector('input[type=checkbox]').pop().cl
ick()
```

26. 有时候我们定位一个元素，定位器没有问题，但一直定位不了，这时候就要检查这个元素是否在一个 frame 中，seelnium webdriver 提供了一个 switch_to_frame 方法，可以很轻松的来解决这个问题。

```
browser.get(……)
browser.implicitly_wait(30)
#先找到到 id=f1 的 frame 标签
browser.switch_to_frame("f1")
#再找到其下面的 id=f2 的 frame 标签
browser.switch_to_frame("f2")
#下面就可以正常的操作元素了
browser.find_element_by_id("kw").send_keys("selenium")
browser.find_element_by_id("su").click()
```

27. 有可能嵌套的不是框架，而是窗口，还有针对窗口的方法：switch_to_window 用法与 switch_to_frame 相同：

```
driver.switch_to_window("windowName")
```

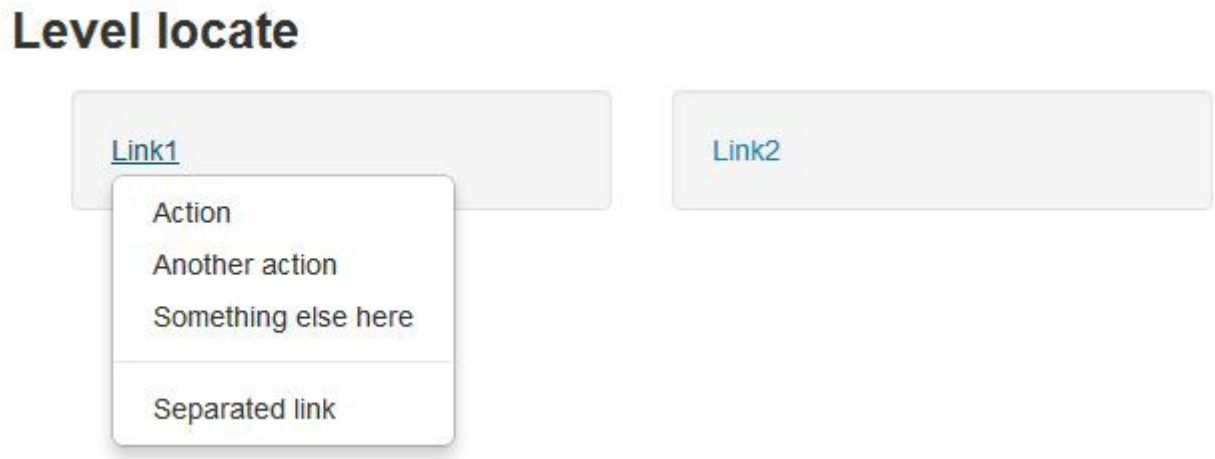
28. unit()和 until_not()和 c++中的 until 语句类似，都是直到指定条件成立或不成立才返回：

```
from selenium.webdriver.support.ui import WebDriverWait
....
element = WebDriverWait(driver, 10).until(lambda x:
x.find_element_by_id(“someId”))
is_disappeared = WebDriverWait(driver, 30, 1, (ElementNotVisibleException)).
                    until_not(lambda x:
x.find_element_by_id(“someId”).is_displayed())
```

WebDriverWai()一般由 unit()或 until_not()方法配合使用，下面是 unit()和 until_not()方法的说明。

- until(method, message=' ')：直到指定条件成立才返回。
- until_not(method, message=' ')：直到指定条件不成立才返回。

29. 在实际的测试中也经常会遇到这种问题：页面上有很多个属性基本相同的元素，现在需要具体定位到其中的一个。由于属性基本相当，所以在定位的时候会有些麻烦，这时候就需要用到层级定位。先定位父元素，然后再通过父元素定位子孙元素。




具体思路是：先点击显示出 1 个下拉菜单，然后再定位到该下拉菜单所在的 ul，再定位这个 ul 下的某个具体的 link。在这里，我们定位第 1 个下拉菜单中的 Action 这个选项。

```
dr = webdriver.Firefox()
file_path = 'file:///' + os.path.abspath('level_locate.html')
dr.get(file_path)

#点击 Link1 链接，弹出下拉列表
dr.find_element_by_link_text('Link1').click()

#找到 id 为 dropdown1 的父元素
```

```
WebDriverWait(dr, 10).until(lambda
the_driver:the_driver.find_element_by_id('dropdown1').is_displayed())

#在父亲元件下找到 link 为 Action 的子元素
menu = dr.find_element_by_id('dropdown1').find_element_by_link_text('Action')

#鼠标定位到子元素上
webdriver.ActionChains(dr).move_to_element(menu).perform()
```

- driver.find_element_by_id('xx').find_element_by_link_text('xx').click()

  这里用到了二次定位，通过对 Link1 的单击之后，出现下拉菜单，先定位到下拉菜单，再定位下拉菜单中的选项。当然，如果菜单选项需要单击，可通过二次定位后也直接跟 click()操作。

- WebDriverWait(dr, 10)：10 秒内每隔 500 毫秒扫描 1 次页面变化，当出现指定的元素后结束。dr 就不解释了，前面操作 webdriver.firefox()的句柄。
- is_displayed()：该元素是否用户可以见。
- move_to_element(menu)：移动鼠标到一个元素中，menu 上面已经定义了他所指向的哪一个元素。
- perform()：执行所有存储的行为。

30. webdriver 操作 cookie 的方法有：

- get_cookies()：获得 cookie 信息。
- get_cookie(name)：返回特定 name 有 cookie 信息。
- add_cookie(cookie_dict)：向 cookie 添加会话信息。
- delete_cookie(name)：删除特定的 cookie。
- delete_all_cookies()：删除所有 cookie。

31. 获得 cookie 信息：

```
driver = webdriver.Chrome()
driver.get("http://www.youdao.com")

# 获得 cookie 信息
cookie= driver.get_cookies()
```

32. 打印自己想要的 cookie 信息：

```
#coding=utf-8
from selenium import webdriver
import time
driver = webdriver.Firefox()
```

```
driver.get("http://www.youdao.com")

#向 cookie 的 name 和 value 添加会话信息。
driver.add_cookie({'name':'key-aaaaaaa', 'value':'value-bbbb'})

#遍历 cookies 中的 name 和 value 信息打印，当然还有上面添加的信息
for cookie in driver.get_cookies():
print "%s -> %s" % (cookie['name'], cookie['value'])

##### 下面可以通过两种方式删除 cookie #####
# 删除一个特定的 cookie
driver.delete_cookie("CookieName")

# 删除所有 cookie
driver.delete_all_cookies()
time.sleep(2)
driver.close()
```

33.%r 的含义是“不管什么都打印出来”。

```
print "my name is %r ,age is %r" %(name,age)
```

# 第 4 部分 selenium2 python 自动化测试

1. XPath 定位示例：

```
<html class="w3c">
<body>
  <div class="page-wrap">
  <div id="hd" name="q">
    <form target="_self" action="http://www.so.com/s">
    <span id="input-container">
      <input id="input" type="text" x-webkit-speech="" autocomplete="off"
suggestwidth="501px" >
```

我们看到的是一个有层级关系页面，下面我看看如果用 xpath 来定位最后一个元素。

- 用绝对路径定位：

```
find_element_by_xpath("/html/body/div[2]/form/span/input")
```

- 相对路径定位：

```
find_element_by_xpath("//input[@id=’input’]") #通过自身的 id 属性定位
find_element_by_xpath("//span[@id=’input-container’]/input") #通过上一级目录的 id 属性定位
find_element_by_xpath("//div[@id=’hd’]/form/span/input") #通过上三级目录的 id 属性定位
find_element_by_xpath("//div[@name=’q’]/form/span/input")#通过上三级目录的 name 属性定位
```

XPath 可以做布尔逻辑运算，例如：//div[@id=’hd’ or @name=’q’]。

2. 以快播私有云登录实例来展示常见元素操作的使用快播私有云登录实例来展示常见元素操作的使用

```
#coding=utf-8
from selenium import webdriver
driver = webdriver.Firefox()
driver.get("http://passport.kuaibo.com/login/?referrer=http%3A%2F%2Fwebcloud.kuaibo.com%2F")
driver.find_element_by_id("user_name").clear()
driver.find_element_by_id("user_name").send_keys("username")
driver.find_element_by_id("user_pwd").clear()
driver.find_element_by_id("user_pwd").send_keys("password")
driver.find_element_by_id("dl_an_submit").click()
#通过 submit() 来提交操作
#driver.find_element_by_id("dl_an_submit").submit()
driver.quit()
```

submit() 提交表单。可以使用 submit()方法来代替 click()对输入的信息进行提交，在有些情况下两个方法可以相互使用；submit()要求提交对象是一个表单，更强调对信息的提交。click()更强调事件的独立性。

3. python 是个容易出现编码问题的语言，有时候当我们在 send_keys()方法中输入中文时，然后脚本在运行时就报编码错误，这个时候我们可以在脚本开头声明编码为 utf-8，然后在中文字符的前面加个小 u 就解决了：

```
#coding=utf-8
send_keys(u"中文内容")
```

4. WebElement 接口除了我们前面介绍的方法外，它还包含了别一些有用的方法。下面，我们例举例几个比较有用的方法。

(1) size：返回元素的尺寸。例：

```
#返回百度输入框的宽高
size=driver.find_element_by_id("kw").size
print size
```

(2) text：获取元素的文本，例：

```
#返回百度页面底部备案信息
text=driver.find_element_by_id("cp").text
print text
```

(3) get_attribute(name)：获得属性值。例：

```
#返回元素的属性值，可以是 id、name、type 或元素拥有的其它任意属性
attribute=driver.find_element_by_id("kw").get_attribute('type')
print attribute
```

需要说明的是这个方法在定位一组时将变得非常有用。

(4) is_displayed()：设置该元素是否用户可见。例：

```
#返回元素的结果是否可见，返回结果为 True 或 False
result=driver.find_element_by_id("kw").is_displayed()
print result
```

(5) WebElement 接口的其它更多方法请参考 webdriver API。

5. ActionChains 类鼠标操作的常用方法：

- context_click() 右击
- double_click() 双击
- drag_and_drop() 拖动
- move_to_element() 鼠标悬停在一个元素上
- click_and_hold() 按下鼠标左键在一个元素上

6. 假如一个 web 应用的列表文件提供了右击弹出快捷菜单的的操作。可以通过 context_click()方法模拟鼠标右键，参考代码如下：

```
#引入 ActionChains 类
from selenium.webdriver.common.action_chains import ActionChains
…
#定位到要右击的元素
right =driver.find_element_by_xpath("xx")
#对定位到的元素执行鼠标右键操作
ActionChains(driver).context_click(right).perform()
….
```

from selenium.webdriver.common.action_chains import ActionChains 这里需要注意的是，在使用 ActionChains 类下面的方法之前，要先将包引入。

ActionChains 用于生成用户的行为；所有的行为都存储在 actionchains 对象。通过 perform()执行存储的行为。

7．按下鼠标左键：click_and_hold()

```
#引入 ActionChains 类
from selenium.webdriver.common.action_chains import ActionChains
…
#定位到鼠标按下左键的元素
left=driver.find_element_by_xpath("xxx")
#对定位到的元素执行鼠标左键按下的操作
ActionChains(driver).click_and_hold(left).perform()
```

8．键盘事件：

```
from selenium.webdriver.common.keys import Keys
……
driver.find_element_by_id("su").send_keys(Keys.ENTER)
……
```

在使用键盘按键方法前需要先导入 keys 类包。下面经常使用到的键盘操作：

- send_keys(Keys.BACK_SPACE) 删除键
- send_keys(Keys.SPACE) 空格键(Space)。
- send_keys(Keys.TAB)：tab 键。
- send_keys(Keys.ESCAPE)：回退键。
- send_keys(Keys.ENTER)：回车键。
- send_keys(Keys.CONTROL,'a')：Ctrl+A。
- send_keys(Keys.CONTROL,'c')：Ctrl+C。
- send_keys(Keys.CONTROL,'x')：Ctrl+X。
- send_keys(Keys.CONTROL,'v')：Ctrl+V。

9．获取当前 URL 也是非常重要的一个操作，在某些情况下，你访问一个 URL，这时系统会自动对这个 URL 进行跳转，这就是所谓的“重定向”。一般测试重定向的方法是访问这个 URL，然后等待页面重定向完毕之后，获取当前页面的 URL，判断该 URL 是否符合预期。如果页面的 ULR 返回不正确，而表示当前操作没有进行正常的跳转。下面通过快播私有云登录实例进行讲解：

```
#coding=utf-8
from selenium import webdriver
driver = webdriver.Firefox()
driver.get("http://passport.kuaibo.com/login/?referrer=http%3A%2F
%2Fwebcloud.kuaibo.com%2F")

#登录
driver.find_element_by_id("user_name").clear()
```

```
driver.find_element_by_id("user_name").send_keys("username")
driver.find_element_by_id("user_pwd").clear()
driver.find_element_by_id("user_pwd").send_keys("password")
driver.find_element_by_id("dl_an_submit").click()

#获得前面 title，打印
title = driver.title
print title

#拿当前 URL 与预期 URL 做比较
if title == u"快播私有云":
   print "title ok!"
else:
   print "title on!"

#获得前面 URL，打印
now_url = driver.current_url
print now_url

#拿当前 URL 与预期 URL 做比较
if now_url == "http://webcloud.kuaibo.com/":
   print "url ok!"
else:
   print "url on!"

#获得登录成功的用户，打印
now_user=driver.find_element_by_xpath("//div[@id='Nav']/ul/li[4]/a[1]/span").text
print now_user
driver.quit()
```

本例中涉及到新的方法如下：

- title：返回当前页面的标题
- current_url：获取当前加载页面的 URL

10. 有时候为了保证脚本运行的稳定性，需要脚本中添加等待时间。

- sleep()：设置固定休眠时间。python 的 time 包提供了休眠方法 sleep()，导入 time 包后就可以使用 sleep()进行脚本的执行过程进行休眠。
- implicitly_wait()：是 webdirver 提供的一个超时等待。隐的等待一个元素被发现，或一个命令完成。如果超出了设置时间的则抛出异常。
- WebDriverWait()：同样也是 webdirver 提供的方法。在设置时间内，默认每隔一段时间检测一次当前页面元素是否存在，如果超过设置时间检测不到则抛出异常。

下面通过实例来展示方法的具体使用：

```
#coding=utf-8
from selenium import webdriver

#导入 WebDriverWait 包
from selenium.webdriver.support.ui import WebDriverWait

#导入 time 包
import time
driver = webdriver.Firefox()
driver.get("http://www.baidu.com")

#WebDriverWait()方法使用
element=WebDriverWait(driver, 10).until(lambda
driver :driver.find_element_by_id("kw"))
element.send_keys("selenium")

#添加智能等待
driver.implicitly_wait(30)
driver.find_element_by_id("su").click()

#添加固定休眠时间
time.sleep(5)
driver.quit()
```

implicitly_wait()方法比 sleep() 更加智能，后者只能选择一个固定的时间的等待，前者可以在一个时间范围内智能的等待。

WebDriverWait()详细格式如下：

```
WebDriverWait(driver, timeout, poll_frequency=0.5, ignored_exceptions=None)
```

- driver：WebDriver 的驱动程序。
- timeout：最长超时时间，默认以秒为单位。
- poll_frequency：休眠时间的间隔时间，默认为 0.5 秒。
- ignored_exceptions：超时后的异常信息，默认情况下抛。NoSuchElementException 异常。

11. 对话框处理：


```
#coding=utf-8
from selenium import webdriver
driver = webdriver.Firefox()
driver.get("http://www.baidu.com/")

#点击登录链接
driver.find_element_by_name("tj_login").click()

#通过二次定位找到用户名输入框
div=driver.find_element_by_class_name("tang-
content").find_element_by_name("userName")
div.send_keys("username")

#输入登录密码
driver.find_element_by_name("password").send_keys("password")

#点击登录
driver.find_element_by_id("TANGRAM__PSP_10__submit").click()
driver.quit()
```

本例中并没有用到新方法，唯一的技巧是用到了二次定位，这个技巧在层级定位中已经有过使用。第一次定位找到弹出的登录框，在登录框上再次进行定位找到了用户名输入框。

12. webdriver 提供了相关相方法可以很轻松的在多个窗口之间切换并操作不同窗口上的元素。


要想在多个窗口之间切换，首先要获得每一个窗口的唯一标识符号（句柄）。通过获得的句柄来区别分不同的窗口，从而对不同窗口上的元素进行操作。

```
#coding=utf-8
from selenium import webdriver
import time
driver = webdriver.Firefox()
driver.get("http://www.baidu.com/")

#获得当前窗口
nowhandle=driver.current_window_handle

#打开注册新窗口
driver.find_element_by_name("tj_reg").click()

#获得所有窗口
allhandles=driver.window_handles

#循环判断窗口是否为当前窗口
for handle in allhandles:
    if handle != nowhandle:
        driver.switch_to_window(handle)
        print 'now register window!'
```

```
    #切换到邮箱注册标签
    driver.find_element_by_id("mailRegTab").click()
    time.sleep(5)
    driver.close()

#回到原先的窗口
driver.switch_to_window(nowhandle)
driver.find_element_by_id("kw").send_keys(u"注册成功！")
time.sleep(3)
driver.quit()
```

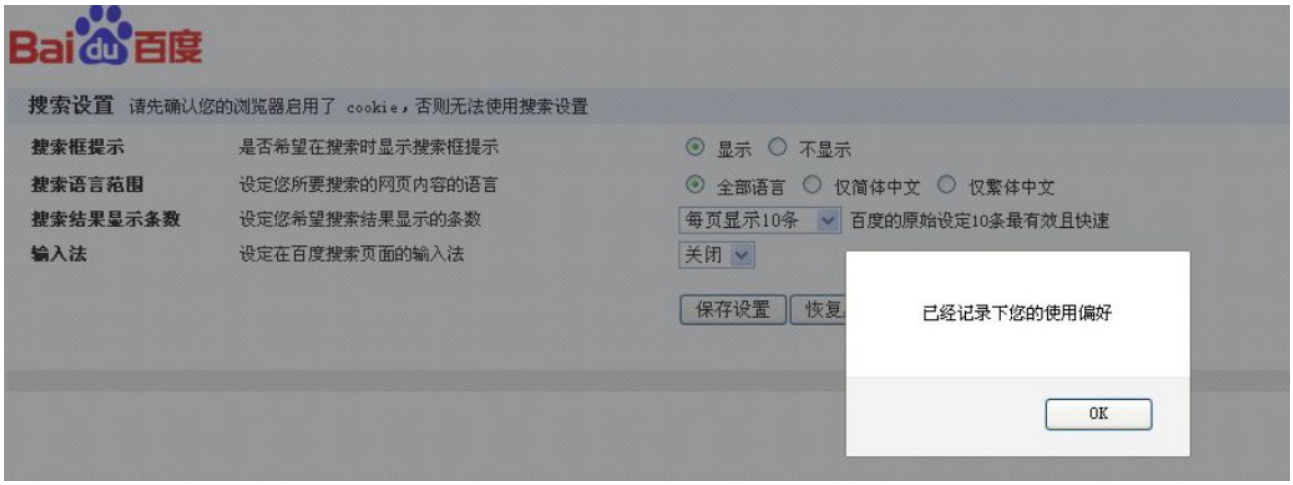
在本例中所有用到的新方法：

- current_window_handle：获得当前窗口句柄
- window_handles：返回的所有窗口的句柄到当前会话
- switch_to_window()：用于处理多窗口操作的方法，与我们前面学过的 switch_to_frame() 是类似，switch_to_window()用于处理多窗口之前切换，switch_to_frame() 用于处理多框架的切换。
- close()如果你足够细心会发现我们在关闭“注册页”时用的是 close()方法，而非 quit()；close()用于关闭当前窗口，quit()用于退出驱动程序并关闭所有相关窗口。

13. webdriver 中处理 JavaScript 所生成的 alert、confirm 以及 prompt 是很简单的。具体思路是使用 switch_to.alert()方法定位到 alert/confirm/prompt。然后使用 text/accept/dismiss/send_keys 按需进行操做。

- text()：返回 alert/confirm/prompt 中的文字信息。
- accept()：点击确认按钮。
- dismiss()： 点击取消按钮，如果有的话。
- send_keys()： 输入值，这个 alert\confirm 没有对话框就不能用了，不然会报错。

示例：



```
#coding=utf-8
from selenium import webdriver
import time
driver = webdriver.Firefox()
driver.get("http://www.baidu.com/")

#点击打开搜索设置
driver.find_element_by_name("tj_setting").click()
driver.find_element_by_id("SL_1").click()

#点击保存设置
driver.find_element_by_xpath("//div[@id='gxszButton']/input").click()
#获取网页上的警告信息
alert=driver.switch_to_alert()

#接收警告信息
alert.accept()
dirver.quit()
```

switch_to_alert()：用于获取网页上的警告信息。我们可以对警告信息做以下操作：

```
#接受警告信息
alert = driver.switch_to_alert()
alert.accept()

#得到文本信息并打印
alert = driver.switch_to_alert()
print alert.text().quit()

#取消对话框（如果有的话）
alert = driver.switch_to_alert()
alert.dismiss()

#输入值（如果有的话）
alert = driver.switch_to_alert()
alert.send_keys(“xxx”)
```

14. 在命令行下输入以下信息可以启动 selenium 的本地文档：

```
Python -m pydoc -p 4567
```

然后在浏览器输入 http://localhost:4567/可以访问相关内容。

15.