网络爬虫剖析，以Pyspider为例

2015/03/04 · [Python](http://python.jobbole.com/category/python/) · 789 阅读 · [Pyspider](http://python.jobbole.com/tag/pyspider/), [爬虫](http://python.jobbole.com/tag/%e7%88%ac%e8%99%ab/)

分享到：**13**

http://python.jobbole.com/81109/

本文由 [伯乐在线](http://python.jobbole.com/) - [xianhu](http://www.jobbole.com/members/xianhu) 翻译，[toolate](http://www.jobbole.com/members/toolate) 校稿。未经许可，禁止转载！  
英文出处：[siscia.github.io](http://siscia.github.io/open-hire/2015/01/17/pyspider-anatomy-of-a-web-crawler.html)。欢迎加入[翻译组](http://group.jobbole.com/category/feedback/trans-team/" \t "_blank)。

在这篇文章中，我们将分析一个网络爬虫。

网络爬虫是一个扫描网络内容并记录其有用信息的工具。它能打开一大堆网页，分析每个页面的内容以便寻找所有感兴趣的数据，并将这些数据存储在一个数据库中，然后对其他网页进行同样的操作。

如果爬虫正在分析的网页中有一些链接，那么爬虫将会根据这些链接分析更多的页面。

搜索引擎就是基于这样的原理实现的。

这篇文章中，我特别选了一个稳定的、”年轻”的开源项目[pyspider](https://github.com/binux/pyspider)，它是由 binux 编码实现的。

注：据认为pyspider持续监控网络，它假定网页在一段时间后会发生变化，因此一段时间后它将会重新访问相同的网页。

**概述**

爬虫pyspider主要由四个组件组成。包括调度程序（scheduler），抓取程序（fetcher），内容处理程序（processor）以及一个监控组件。

调度程序接受任务并决定该做什么。这里有几种可能性，它可以丢弃一个任务（可能这个特定的网页刚刚被抓取过了），或者给任务分配不同的优先级。

当各个任务的优先级确定之后，它们被传入抓取程序。它重新抓取网页。这个过程很复杂，但逻辑上比较简单。

当网络上的资源被抓取下来，内容处理程序就负责抽取有用的信息。它运行一个用户编写的Python脚本，这个脚本并不像沙盒一样被隔离。它的职责还包括捕获异常或日志，并适当地管理它们。

最后，爬虫pyspider中有一个监控组件。

爬虫pyspider提供一个异常强大的网页界面（web ui），它允许你编辑和调试你的脚本，管理整个抓取过程，监控正在进行的任务，并最终输出结果。

**项目和任务**

在pyspider中，我们有项目和任务的概念。

一个任务指的是一个需要从网站检索并进行分析的单独页面。

一个项目指的是一个更大的实体，它包括爬虫涉及到的所有页面，分析网页所需要的python脚本，以及用于存储数据的数据库等等。

在pyspider中我们可以同时运行多个项目。

**代码结构分析**

**根目录**

在根目录中可以找到的文件夹有：

•data，空文件夹，它是存放由爬虫所生成的数据的地方。

•docs，包含该项目文档，里边有一些markdown代码。

•pyspider，包含项目实际的代码。

•test，包含相当多的测试代码。

这里我将重点介绍一些重要的文件：

•.travis.yml，一个很棒的、连续性测试的整合。你如何确定你的项目确实有效？毕竟仅在你自己的带有固定版本的库的机器上进行测试是不够的。

•Dockerfile，同样很棒的工具！如果我想在我的机器上尝试一个项目，我只需要运行Docker，我不需要手动安装任何东西，这是一个使开发者参与到你的项目中的很好的方式。

•LICENSE，对于任何开源项目都是必需的，（如果你自己有开源项目的话）不要忘记自己项目中的该文件。

•requirements.txt，在Python世界中，该文件用于指明为了运行该软件，需要在你的系统中安装什么Python包，在任何的Python项目中该文件都是必须的。

•run.py，该软件的主入口点。

•setup.py，该文件是一个Python脚本，用于在你的系统中安装pyspider项目。

已经分析完项目的根目录了，仅根目录就能说明该项目是以一种非常专业的方式进行开发的。如果你正在开发任何的开源程序，希望你能达到这样的水准。

**文件夹pyspider**

让我们更深入一点儿，一起来分析实际的代码。

在这个文件夹中还能找到其他的文件夹，整个软件背后的逻辑已经被分割，以便更容易的进行管理和扩展。

这些文件夹是：database、fetcher、libs、processor、result、scheduler、webui。

在这个文件夹中我们也能找到整个项目的主入口点，run.py。

**文件run.py**

这个文件首先完成所有必需的杂事，以保证爬虫成功地运行。最终它产生所有必需的计算单元。向下滚动我们可以看到整个项目的入口点，cli()。

**函数cli()**

这个函数好像很复杂，但与我相随，你会发现它并没有你想象中复杂。函数cli()的主要目的是创建数据库和消息系统的所有连接。它主要解析命令行参数，并利用所有我们需要的东西创建一个大字典。最后，我们通过调用函数all()开始真正的工作。

**函数all()**

一个网络爬虫会进行大量的IO操作，因此一个好的想法是产生不同的线程或子进程来管理所有的这些工作。通过这种方式，你可以在等待网络获取你当前html页面的同时，提取前一个页面的有用信息。

函数all()决定是否运行子进程或者线程，然后调用不同的线程或子进程里的所有的必要函数。这时pyspider将产生包括webui在内的，爬虫的所有逻辑模块所需要的，足够数量的线程。当我们完成项目并关闭webui时，我们将干净漂亮地关闭每一个进程。

现在我们的爬虫就开始运行了，让我们进行更深入一点儿的探索。

**调度程序**

调度程序从两个不同的队列中获取任务（newtask\_queue和status\_queue），并把任务加入到另外一个队列（out\_queue），这个队列稍后会被抓取程序读取。

调度程序做的第一件事情是从数据库中加载所需要完成的所有的任务。之后，它开始一个无限循环。在这个循环中会调用几个方法：

1.\_update\_projects()：尝试更新的各种设置，例如，我们想在爬虫工作的时候调整爬取速度。

2.\_check\_task\_done()：分析已完成的任务并将其保存到数据库，它从status\_queue中获取任务。

3.\_check\_request()：如果内容处理程序要求分析更多的页面，把这些页面放在队列newtask\_queue中，该函数会从该队列中获得新的任务。

4.\_check\_select()：把新的网页加入到抓取程序的队列中。

5.\_check\_delete()：删除已被用户标记的任务和项目。

6.\_try\_dump\_cnt()：记录一个文件中已完成任务的数量。对于防止程序异常所导致的数据丢失，这是有必要的。

Python



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | def run(self):        while not self.\_quit:            try:                time.sleep(self.LOOP\_INTERVAL)                self.\_update\_projects()                self.\_check\_task\_done()                self.\_check\_request()                while self.\_check\_cronjob():                    pass                self.\_check\_select()                self.\_check\_delete()                self.\_try\_dump\_cnt()                self.\_exceptions = 0            except KeyboardInterrupt:                break            except Exception as e:                logger.exception(e)                self.\_exceptions += 1                if self.\_exceptions > self.EXCEPTION\_LIMIT:                    break                continue |

循环也会检查运行过程中的异常，或者我们是否要求python停止处理。

Python



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | finally:        # exit components run in subprocess        for each in threads:            if not each.is\_alive():                continue            if hasattr(each, 'terminate'):                each.terminate()            each.join() |

**抓取程序**

抓取程序的目的是检索网络资源。

pyspider能够处理普通HTML文本页面和基于AJAX的页面。只有抓取程序能意识到这种差异，了解这一点非常重要。我们将仅专注于普通的html文本抓取，然而大部分的想法可以很容易地移植到Ajax抓取器。

这里的想法在某种形式上类似于调度程序，我们有分别用于输入和输出的两个队列，以及一个大的循环。对于输入队列中的所有元素，抓取程序生成一个请求，并将结果放入输出队列中。

它听起来简单但有一个大问题。网络通常是极其缓慢的，如果因为等待一个网页而阻止了所有的计算，那么整个过程将会运行的极其缓慢。解决方法非常的简单，即不要在等待网络的时候阻塞所有的计算。这个想法即在网络上发送大量消息，并且相当一部分消息是同时发送的，然后异步等待响应的返回。一旦我们收回一个响应，我们将会调用另外的回调函数，回调函数将会以最适合的方式管理这样的响应。

爬虫pyspider中的所有的复杂的异步调度都是由另一个优秀的开源项目[tornado](http://www.tornadoweb.org/en/stable/)完成。

现在我们的脑海里已经有了极好的想法了，让我们更深入地探索这是如何实现的。

Python



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | def run(self):      def queue\_loop():          if not self.outqueue or not self.inqueue:              return          while not self.\_quit:              try:                  if self.outqueue.full():                      break                  task = self.inqueue.get\_nowait()                  task = utils.decode\_unicode\_obj(task)                  self.fetch(task)              except queue.Empty:                  break      tornado.ioloop.PeriodicCallback(queue\_loop, 100, io\_loop=self.ioloop).start()      self.\_running = True      self.ioloop.start()  <strong>  函数run()</strong> |

函数run()是抓取程序fetcher中的一个大的循环程序。

函数run()中定义了另外一个函数queue\_loop()，该函数接收输入队列中的所有任务，并抓取它们。同时该函数也监听中断信号。函数queue\_loop()作为参数传递给tornado的类PeriodicCallback，如你所猜，PeriodicCallback会每隔一段具体的时间调用一次queue\_loop()函数。函数queue\_loop()也会调用另一个能使我们更接近于实际检索Web资源操作的函数：fetch()。

**函数fetch(self, task, callback=None)**

网络上的资源必须使用函数phantomjs\_fetch()或简单的http\_fetch()函数检索，函数fetch()只决定检索该资源的正确方法是什么。接下来我们看一下函数http\_fetch()。

**函数http\_fetch(self, url, task, callback)**

Python



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | def http\_fetch(self, url, task, callback):      '''HTTP fetcher'''      fetch = copy.deepcopy(self.default\_options)      fetch['url'] = url      fetch['headers']['User-Agent'] = self.user\_agent        def handle\_response(response):          ...          return task, result        try:          request = tornado.httpclient.HTTPRequest(header\_callback=header\_callback, \*\*fetch)          if self.async:              self.http\_client.fetch(request, handle\_response)          else:              return handle\_response(self.http\_client.fetch(request)) |

终于，这里才是完成真正工作的地方。这个函数的代码有点长，但有清晰的结构，容易阅读。

在函数的开始部分，它设置了抓取请求的header，比如User-Agent、超时timeout等等。然后定义一个处理响应response的函数：handle\_response()，后边我们会分析这个函数。最后我们得到一个tornado的请求对象request，并发送这个请求对象。请注意在异步和非异步的情况下，是如何使用相同的函数来处理响应response的。

让我们往回看一下，分析一下函数handle\_response()做了什么。

**函数handle\_response(response)**

Python



|  |  |
| --- | --- |
| 1  2  3  4  5  6 | def handle\_response(response):      result = {}      result['orig\_url'] = url      result['content'] = response.body or ''      callback('http', task, result)      return task, result |

这个函数以字典的形式保存一个response的所有相关信息，例如url，状态码和实际响应等，然后调用回调函数。这里的回调函数是一个小方法：send\_result()。

**函数send\_result(self, type, task, result)**

Python



|  |  |
| --- | --- |
| 1  2  3 | def send\_result(self, type, task, result):      if self.outqueue:          self.outqueue.put((task, result)) |

这个最后的函数将结果放入到输出队列中，等待内容处理程序processor的读取。

**内容处理程序processor**

内容处理程序的目的是分析已经抓取回来的页面。它的过程同样也是一个大循环，但输出中有三个队列（status\_queue, newtask\_queue 以及result\_queue）而输入中只有一个队列（inqueue）。

让我们稍微深入地分析一下函数run()中的循环过程。

**函数run(self)**

Python



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | def run(self):      try:          task, response = self.inqueue.get(timeout=1)          self.on\_task(task, response)          self.\_exceptions = 0      except KeyboardInterrupt:          break      except Exception as e:          self.\_exceptions += 1          if self.\_exceptions > self.EXCEPTION\_LIMIT:              break          continue |

这个函数的代码比较少，易于理解，它简单地从队列中得到需要被分析的下一个任务，并利用on\_task(task, response)函数对其进行分析。这个循环监听中断信号，只要我们给Python发送这样的信号，这个循环就会终止。最后这个循环统计它引发的异常的数量，异常数量过多会终止这个循环。

**函数on\_task(self, task, response)**

Python



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | def on\_task(self, task, response):      response = rebuild\_response(response)      project = task['project']      project\_data = self.project\_manager.get(project, updatetime)      ret = project\_data['instance'].run(        status\_pack = {          'taskid': task['taskid'],          'project': task['project'],          'url': task.get('url'),          ...          }      self.status\_queue.put(utils.unicode\_obj(status\_pack))      if ret.follows:          self.newtask\_queue.put(              [utils.unicode\_obj(newtask) for newtask in ret.follows])        for project, msg, url in ret.messages:          self.inqueue.put(({...},{...}))        return True |

函数on\_task()是真正干活的方法。

它尝试利用输入的任务找到任务所属的项目。然后它运行项目中的定制脚本。最后它分析定制脚本返回的响应response。如果一切顺利，将会创建一个包含所有我们从网页上得到的信息的字典。最后将字典放到队列status\_queue中，稍后它会被调度程序重新使用。

如果在分析的页面中有一些新的链接需要处理，新链接会被放入到队列newtask\_queue中，并在稍后被调度程序使用。

现在，如果有需要的话，pyspider会将结果发送给其他项目。

最后如果发生了一些错误，像页面返回错误，错误信息会被添加到日志中。

结束！