****

分布式与云计算

实验报告

|  |  |
| --- | --- |
| 姓 名： | 谭哲文 |
| 学 号： | 8202191123 |
| 学 院： | 计算机学院 |
| 专业班级： | 计科2105 |
| 指导教师： | 余腊生 |

本科生院制

2024年6月

**分布式爬虫**

**1. 实验目的**

本次实验目标为设计一个分布式网络爬虫实现一下功能：

1.从一个给定的网址中分析其所包含的URL并爬取对应的网页，直到爬取完全部

不重复的网页为止。

2.支持分布式爬取，同时记录输出每一个网页的大小。

3.采用多线程结构设计，实现高性能的网络爬虫。

随着国际互联网的迅速发展，网上的信息越来越多，全球网页数量超过20亿，

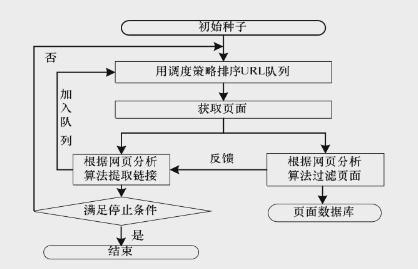
每天新增加730万网页。要在如此浩瀚的信息海洋里寻找信息，就像“大海捞针”一样困难。在实际生活中我们经常会使用像百度、Google这些搜索引擎检索各种信息，搜索引擎正是为了解决这个问题而出现的技术，而网络爬虫正是搜索引擎所需要的关键部分。本次实验主要的内容就是利用IO复用抓取网页，并多线程的分析每个抓取到的网页所包含的URL信息，通过消息队列将抓取网页的部分和分析网页部分进行通信，最终记录下160000网页中所包含的所有URL，实现分布式网络爬虫。

**2. 实验原理**

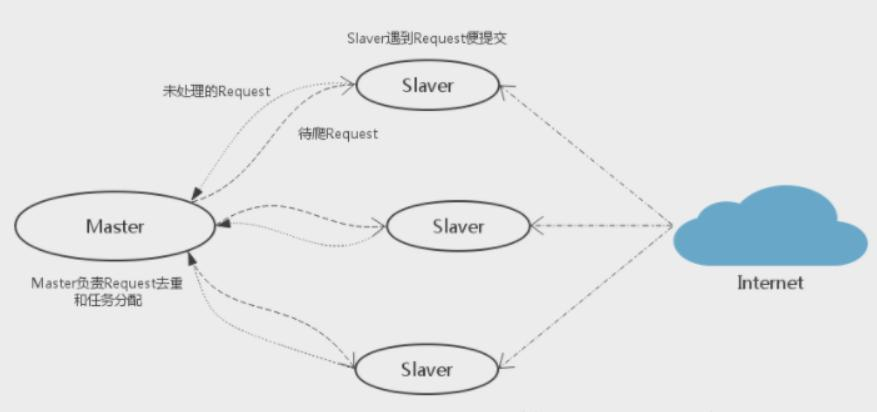
1. 分布式爬虫

网络爬虫是一个自动提取网页的程序，它为搜索引擎从万维网上下载网页，

是搜索引擎的重要组成。传统爬虫从一个或若干初始网页的URL开始作为初始种子，获得初始网页上的页面，通过页面源代码解析新的URL。需要有一个URL队列实现一定的调度策略，把解析得到的新的URL加入队列，如果是相应的目标数据项，则应该存入本地数据库。



由于单机运行爬虫的IO吞吐量和网络传输速率有限，因此其总体效率有限。而

分布式爬虫能够通过多机协同工作解决这个问题。其架构是用多个Slaver端提供性能和硬件资源解决性能的问题，用一个Master端作为核心服务器，对Slaver端进行任务调度、存储Slaver端返回的爬取结果。

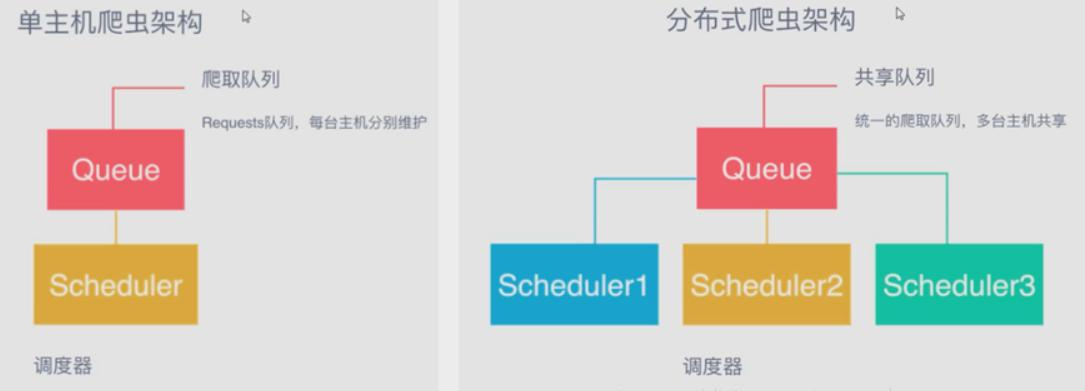
整个流程是：

1. 首先Slaver端从Master端取任务(Request、url)进行数据抓取，Slaver端抓取数

据的同时，产生新任务的Request或item数据项返回给Master处理；

2. Master端数据库，负责任务分配，将处理后的Request加入待爬队列，并且将

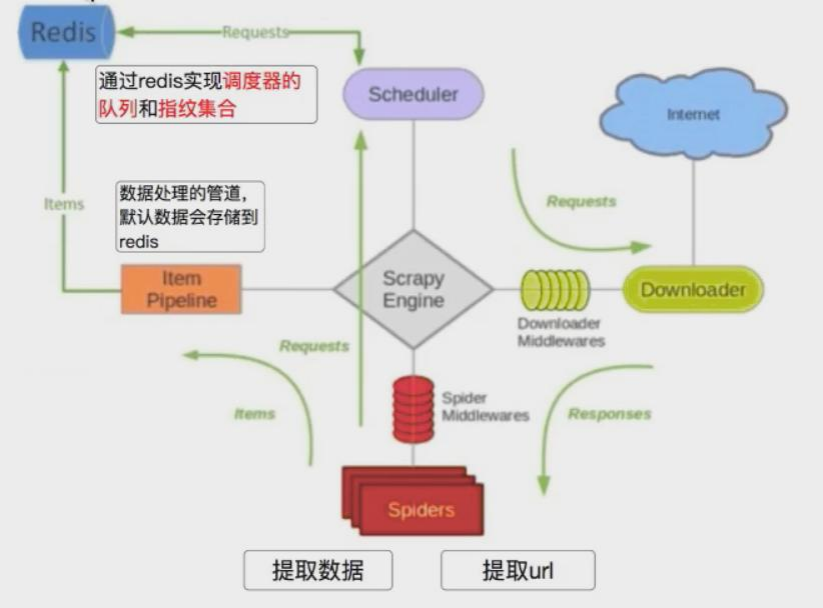
爬取的数据写入数据库。

2. scrapy-redis架构

scrapy是一个基于Redis的scrapy组件，用于快速实现scrapy项目的分布式部署

和数据爬取，在scrapy的基础上实现了更多、更强大的功能，具体体现在：request

去重（不仅仅是爬过的网页不再爬取，保存的数据也会去重），爬虫持久化。



流程如下：

(1) 引擎(Scrapy Engine)向爬虫Q(Spiders)请求第一个要爬取的URL。

(2) 引擎从爬虫中获取到第一个要爬取的URL,封装成请求(Request)并交给调度器

(Scheduler)。

(3) 调度器访问Redis数据库对请求进行判重，如果不重复，就把这个请求添加到

Redis中。

(4) 当调度条件满足时，调度器会从Redis中取出Request，交给引擎，引擎将这

个Request通过下载中间件转发给下载器。

(5) 一旦页面下载完毕，下载器(Downloader)生成一个该页面的响应(Response)，

并将其通过下载中间件发送给引擎。

(6) 引擎从下载器中接收到响应，并通过爬虫中间件(Spider Middlewares)发送给

爬虫处理。

(7) Spider处理Response，并返回爬取到的Item及新的Request给引擎。(8) 引擎将爬取到的Item通过Item Pipeline给Redis数据库，将Request给调度器。

并进行重复。

**3.** 爬虫设计

爬虫主要爬取贝壳网的二手房信息，通过xpath分析网页源代码。以长沙二手房

信息为例。

由于页面默认只提供最多100页的数据，因此需要将查询范围具体到区，做法是

获取如下区域的链接。

1. **def** parse(self, response):

2. # 获取该城市各区的二手房链接

3. urls = response.xpath('//div/a[@class="CLICKDATA"]/@href').extract()

4. **for** url **in** urls:

5. **yield** Request(response.url+url.split('/')[-2], callback=self.parse\_district\_totalpage)

得到各区二手房的主页面后，由于各区二手房信息数量不同，页数也不同，因

此需要获取各区的页面数。具体信息在页码框的源代码中，因此需要用xpath解析后 运用正则表达式，解析到页码后，再将页码数个url放入共享队列中。

1. **def** parse\_district\_totalpage(self, response):

2. # 获取各区二手房页面的totalpage(总页数)

3. page\_data = response.xpath(

4. '//div[@class="page-box fr"]/div/@page-data').extract\_first()

5. totalpage = int(re.findall('\d+', page\_data)[0])

6. **for** i **in** range(1, totalpage+1):

7. **yield** Request(response.url+'pg{}/'.format(i), callback=self.parse\_district\_page)

之后即可获取各二手房详情页面的url。

1. # 解析各区二手房页面信息

2. **def** parse\_district\_page(self, response):

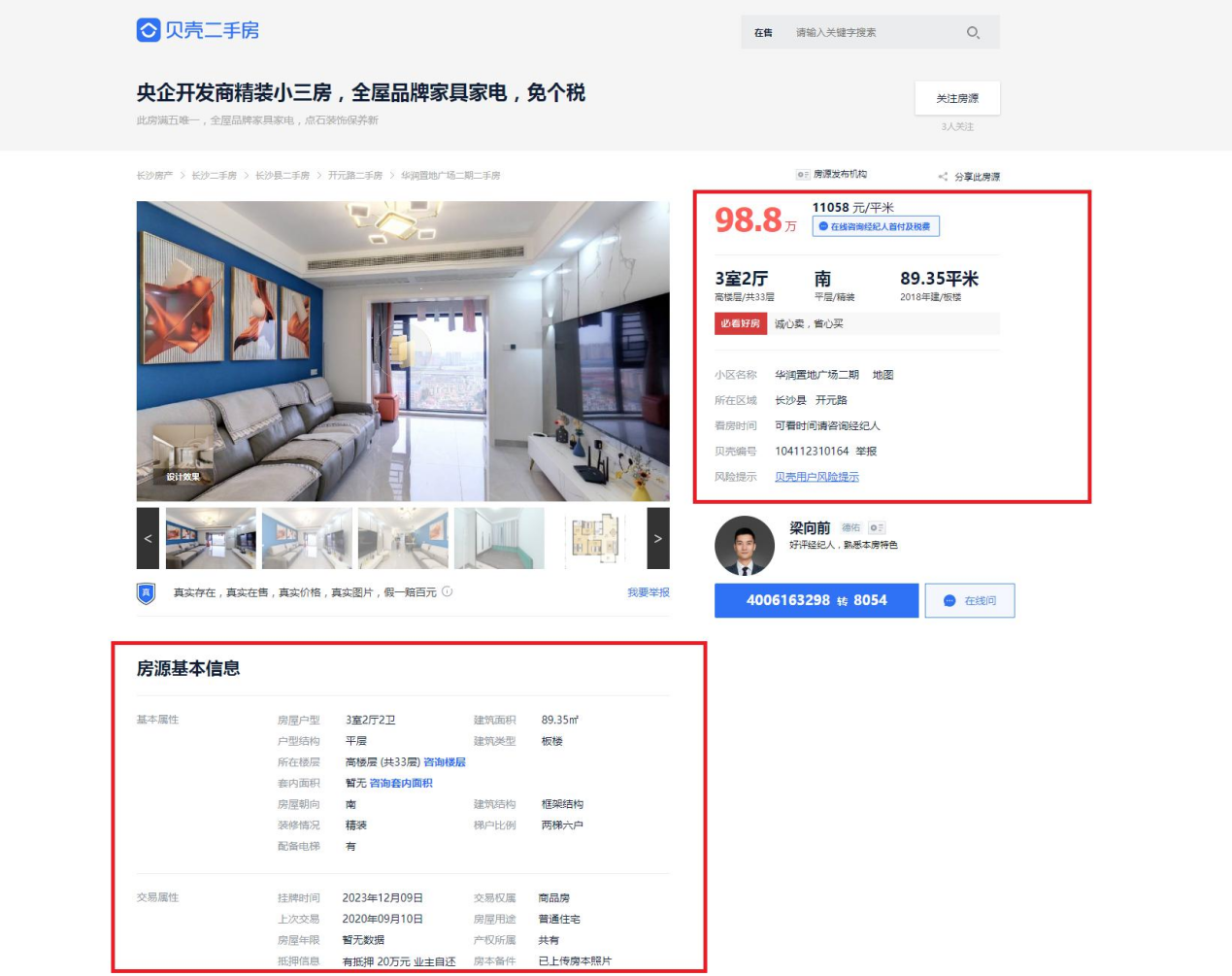
3. # 获取二手房信息

4. urls =response.xpath('//div/div[@class="title"]/a/@href').extract()

5. **for** url **in** urls:

6. **yield** Request(url, callback=self.parse\_house\_page)

最后可以解析详情页，封装成item存储入Redis数据库。

# 解析二手房页面信息

2. **def** parse\_house\_page(self, response):

3. address = response.xpath('//div[@class="fl l-txt"]/a').extract()

4. # 城市

5. city = address[1][:-3]

6. # 区

7. district = address[2][:-3]

8. # 街道

9. street = address[3][:-3]

10. # 社区

11. community = address[4][-3]

12. # 总价

13. total\_price = response.xpath(

14.'//div[@class="price"]/span[@class="total"]/text()').extract\_first()

15. # 均价

16. unit\_price = response.xpath(

17.'//div[@class="unitPrice"]/span[@class="unitPriceValue"]/text()').extract\_first()

18. # 户型

19.house\_type = response.xpath(

20.'//div[@class="room"]/div[@class="mainInfo"]/text()').extract\_first()

21. # 楼层

22. floor = response.xpath(

23.'//div[@class="room"]/div[@class="subInfo"]/text()').extract\_first()

24. # 朝向

25. orientation = response.xpath(

26.'//div[@class="type"]/div[@class="mainInfo"]/text()').extract\_first()

27. # 面积

28. area = response.xpath(

29.'//div[@class="area"]/div[@class="mainInfo"]/text()').extract\_first()

30. # 建成年份/楼房类型

31. complete\_year = response.xpath(

32.'//div[@class="area"]/div[@class="subInfonoHidden"]/text()').extract\_first().strip()

33. # 网页大小

34. page\_size=len(response.text.encode('utf-8'))35. **yield** {'url':response.url,"page\_size":page\_size,'city': city, 'district': di

strict, 'street': street, 'community': community, 'total\_price': total\_price, 'unit\_price': unit\_price, 'house\_type': house\_type, 'floor': floor, 'orientation':orientation, 'area': area, 'complete\_year': complete\_year}

**4. 存储程序设计**

主要是使用mongodb数据库，将Redis中的数据取出存储到mongodb数据库中，

由于数据库都可以远程连接，因此数据项的存储同样可以分布式进行。Slaver端除运

行爬虫程序外，也可运行存储程序，提高利用率。核心代码如下：

1. redis\_client=redis.Redis(host="127.0.0.1",port=6379,db=0)

2. mongo\_client=pymongo.MongoClient()

3. collection=mongo\_client.distributed\_spider.ke

4.

5. **while** True:

6.

key,data=redis\_client.blpop(['kespider:item'])

7.

collection.insert(json.loads(data))

**5. 去重模块**

去重模块主要使用的是scrapy-redis中的DFPDupeFilter。

1. **class** RFPDupeFilter(BaseDupeFilter):

2. """Request Fingerprint duplicates filter"""

3.

4. def \_\_init\_\_(self, path=None, debug=False):

5. self.file = None

6. self.fingerprints = set()

7. self.logdupes = True

8. self.debug = debug

9. self.logger = logging.getLogger(\_\_name\_\_)

10. **if** path:

11. # 此处可以看到去重其实打开了一个名叫 requests.seen的文件

12. # 如果是使用的磁盘的话

13. self.file = open(os.path.join(path, 'requests.seen'), 'a+')

14. self.file.seek(0)

15. self.fingerprints.update(x.rstrip() **for** x in self.file)

16.

17. @classmethod

18. def from\_settings(cls, settings):

19. debug = settings.getbool('DUPEFILTER\_DEBUG')

20. **return** cls(job\_dir(settings), debug)

21.

22. def request\_seen(self, request):

23. fp = self.request\_fingerprint(request)

24. **if** fp in self.fingerprints:

25. # 判断我们的请求是否在这个在集合中

26. **return** True

27. # 没有在集合就添加进去

28. self.fingerprints.add(fp)

29. # 如果用的磁盘队列就写进去记录一下

30. **if** self.file:

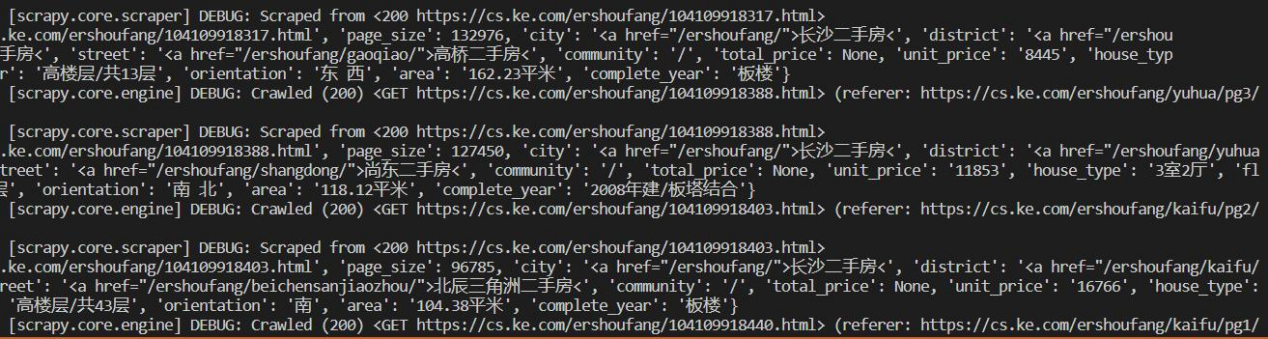
31. self.file.write(fp + os.linesep)

其主要是通过集合实现的，通过request生成指纹(fingerprint)将fingerprint放入集

合中，如果没有在集合中，则进行添加，在集合中则返回True不再进行后续的处理。

**6. 运行结果**

Slaver端运行爬虫程序

Master端mongodb数据库

