WEP Warwick Evans Publishing

International Journal of Computer Science and Information Technology

ISSN: 3005-9682 (Print), ISSN: 3005-7140 (Online) | Volume 4, Number 3, Year 2024
DOI: https://doi.org/10.62051/ijcsit.v4n3.07
Journal homepage: https://wepub.org/index.php/IJCSIT/index

A Personalized Stock Market Acquisition Method Based on Scrapy

Shi Ming, Roznim Mohamad Rasli, Shir Li Wang, Roznim Mohamad Rasli *

Faculty of Computing and Meta-Technology, Universiti Pendidikan Sultan Idris, 35900 Tanjong Malim, Perak, Malaysia

ABSTRACT

Stock data is of great significance for stock analysis and decision-making. In the field of stock data analysis and mining, the quality and diversity of data have a crucial impact on the depth of research and work efficiency. Traditional data acquisition methods, such as official channels or third-party data source API and SDK interfaces, often provide standardized data. Although these data elements meet basic analysis needs to a certain extent, due to their inherent limitations, it is difficult to support personalized and in-depth research. With the development of the stock market and the advancement of data analysis technology, researchers' demand for data is constantly increasing. They not only require real-time and accurate data, but also hope to obtain more diversified and customized data elements, such as investor posting data, social network data, etc. Behind this demand is the desire to build more refined models, understand more complex market behaviors, and provide more accurate decision support. This paper proposes a stock data collection method based on the Scrapy framework, aiming to address the limitations of traditional data acquisition methods. By analyzing specific stock websites and crawling and processing stock data sources, it shows how to use crawler technology to achieve efficient data collection and analysis. The system improves data acquisition efficiency and system stability by optimizing crawler performance and storage algorithms. This research result provides new ideas and technical support for stock data analysis and mining. space before of 18-point and after of 60-point.

KEYWORDS

Stock market data; Web scraping; Scrapy; Data extraction; Crawlers

1. INTRODUCTION

In this context, Web data crawling technology has become an important solution. Web crawling technology can break through the limitations of traditional data sources and directly obtain rich real-time data from the Internet. Especially for individual stock investor comment data in the stock market, Web crawling technology can provide more comprehensive market information and dynamic data, thus providing a more solid foundation for stock data analysis.

As an advanced web crawler framework, Scrapy has become an ideal tool for personalized data acquisition with its efficient asynchronous processing capabilities and flexible architecture. Scrapy can not only handle large-scale data collection tasks, but also support distributed crawling, which greatly improves the efficiency and coverage of data acquisition. In addition, through optimized storage algorithms, Scrapy can effectively manage and store massive data, ensuring data integrity and efficient access.

The purpose of this study is to explore how to use the Scrapy framework to efficiently obtain personalized stock data elements. Through the example of collecting investor comments on specific

stock lines , the practical application and operation process of Scrapy in the data acquisition process are demonstrated. It is hoped that this study can provide new ideas and methods for stock data analysis and provide valuable references for researchers and practitioners in stock data acquisition.

In this study, we will take a financial website as an example to show how to obtain stock- related data from the website. The financial website is a leading stock information service provider in China, providing a wealth of stock data and information, including stock quotes, financial news, financial statements, etc.

the powerful functions and flexible applications of the Scrapy framework in personalized stock data acquisition, as well as methods to improve data management efficiency by optimizing storage algorithms, and provide new technical means and research directions for the field of stock data analysis. We hope to promote the further development of stock data analysis and provide richer and more accurate data support for stock market research and practice.

2. SYSTEM DESIGN

In this chapter, we will introduce in detail the system design of using the Scrapy framework for personalized stock data acquisition. The system design includes aspects such as system architecture, module division, main technology selection, and data flow design. Through reasonable system design, we can ensure the efficiency, stability, and scalability of the crawler system.

2.1. Architecture

The The overall architecture of the crawler system includes the data acquisition layer, data processing layer and data storage layer. The specific architecture is as follows:

(1) Data acquisition layer: The crawler module implemented by the Scrapy framework is responsible for crawling stock data from the target website.

(2) Data processing layer: includes functional modules such as data cleaning, data format conversion and data deduplication to ensure the accuracy and consistency of data.

(3) Data storage layer: Uses databases such as MongoDB and MySQL to store the collected stock data, and supports batch writing and incremental updates of data.

2.2. System Modules

The system is mainly divided into the following modules:

(1) Crawler module: responsible for data crawling and scheduling, including Scrapy crawler and Scrapy-Redis distributed crawler.

(2) Middleware module: processes requests and responses, including user agent pool, proxy pool, and middleware.

(3) Data processing module: performs data cleaning, format conversion and deduplication operations.

(4) Data storage module: stores the processed data in the database and implements batch writing and incremental updates.

(5) Log and monitoring module: records crawler operation logs, monitors crawlers, and promptly detects and handles exceptions.

2.3. Main Technology Selection

2.3.1. Scrapy

Scrapy is an open source Python framework designed for web crawling and data extraction. It provides a set of efficient and extensible tools for building web crawlers, crawling web content and extracting structured data from them.

The main features of Scrapy include:

Asynchronous processing: Scrapy is based on Twisted, an asynchronous network library, which enables it to quickly handle a large number of concurrent requests and improve crawling efficiency.

Framework structure: Scrapy provides a complete framework, including spiders, downloaders, middleware, and pipelines, allowing developers to focus on writing crawling logic without having to worry about the underlying details.

Middleware: Scrapy middleware allows you to customize the processing logic of requests and responses, such as handling HTTP caching, redirection, login verification, IP proxy, etc.

Crawler: Scrapy's crawler (Spider) is a user-defined class used to define crawling rules and data parsing logic. They can customize request URLs, parse HTML or other content, follow links, and extract data.

Selectors: Scrapy uses built-in XPath or CSS selectors to parse HTML and XML, conveniently extracting the required data from HTML documents.

Pipeline: Pipelines are used to clean, validate, and store data scraped from the web. They can be used to perform operations such as data cleaning, deduplication, and saving to a database or file.

Command line tool: Scrapy provides a command line tool for creating new projects, starting crawlers, viewing project structure, etc.

Community support: Scrapy has an active community that provides a wealth of plugins and extensions, as well as detailed documentation and tutorials for easy learning and use.

Distributed crawlers: Through extensions such as Scrapy-Redis, Scrapy can be easily extended to a distributed environment to handle large-scale crawling tasks.

Extensibility: Scrapy is designed to be very flexible, allowing users to extend and customize functionality as needed.

Scrapy is often used in data mining, market analysis, search engine indexing, automated testing, and many other scenarios that require collecting and processing large amounts of data from the Internet. Its powerful functions and ease of use make it one of the preferred tools for Python developers to crawl the web.

2.3.2. Scrapy-Redis

Scrapy-Redis is a powerful extension of the Scrapy framework. It integrates Redis to achieve distributed crawling capabilities, enhances Scrapy's performance in sharing tasks among multiple nodes, persisting request queues, and optimizing deduplication mechanisms. This extension not only allows parallel processing and load balancing, but also ensures that data is not lost and crawling is continuous after task interruptions by sharing the request queue in the Redis database among multiple instances. Using Redis's efficient data structure, Scrapy-Redis performs well in deduplication, and supports priority scheduling, breakpoint resumption, and incremental crawling, greatly improving the flexibility and efficiency of the crawler. Therefore, for building complex crawler projects that handle large-scale data crawling and require efficient resource management, Scrapy-Redis is an indispensable tool that fully utilizes Redis's high performance and distributed characteristics.

2.3.3. MongoDB

MongoDB is a popular open source NoSQL database, known for its JSON-like document storage and flexible schema, suitable for processing large-scale semi-structured or unstructured data. It provides high performance, scalability, rich query functions, and support for a wide range of programming languages. Through sharding and replica sets, MongoDB supports horizontal expansion and high availability, while also having powerful security management capabilities. With the iteration of versions, MongoDB continues to introduce new features to meet the needs of modern data processing, making it widely used in many fields such as big data, real-time analysis, the Internet of Things, and mobile applications.

2.4. Data Flow

Data flow design describes the entire process of data capture, processing, and storage. The specific steps are as follows:

Start the crawler: Through Scrapy-Redis scheduling, the crawler node obtains the URL from the Redis queue and starts the crawling task.

Crawl data: The crawler module sends HTTP requests to the target website and parses the returned HTML page to extract the required stock data.

Processing data: The data processing module cleans, converts the format and removes duplicates of the captured data to ensure the accuracy and consistency of the data.

Storing data: The processed data is stored in MongoDB or MySQL database through the data storage module to achieve batch writing and incremental updates.

Logging and monitoring: During system operation, the log and monitoring modules record the crawler's operating status and crawling status, and detect and handle exceptions in a timely manner.

2.5. System Implementation

Stock data collection system designed in this paper includes three main modules: data acquisition, processing and storage. The data acquisition module implements the crawler module through the Scrapy framework, which is responsible for crawling stock data from the target website (such as Dongfang Finance Network). The data processing module cleans, converts and removes duplicates from the crawled data. The data storage module uses the MongoDB database to store the collected data.

2.5.1. Target site analysis

Stock details pages usually contain multiple parts of information, such as fundamental data, market conditions, financial statements, news, and investor comments. Each part of the information has different values for different analysis needs. This study focuses on investor comment data because these comment data can reflect market sentiment, investors' psychological state, and their views on stocks, thus providing a reference for investment decisions. Generally, the following steps are required to complete the acquisition of data.

Overall page crawling: Use the Scrapy framework to crawl the stock details page of the target website as a whole.

Page structure analysis: Analyze the HTML source code to determine the location of investor comments.

Data Extraction: Extract review data using XPath or CSS selectors.

Data cleaning and processing:

Remove duplicate comments

Remove HTML tags and special characters

Conduct sentiment analysis

Data storage: Store the processed comment data in the database.

2.5.2. Project Structure

The overall project structure is illustrated in Figure 1.

Scrapy. cfg: The configuration file of the Scrapy project, which defines the configuration file path, deployment information, etc.

Items. py: defines the Item data structure, all Item definitions can be placed here

Pipelines.py: defines the implementation of Item Pipeline. All Item Pipeline implementations can be placed here.

Settings. py: defines the global configuration of the project

Middlewares.py: defines the implementation of Downloader Middlewares and Spider Middlewares

Spiders: contains the implementation of a Spider, each Spider corresponds to a Python file

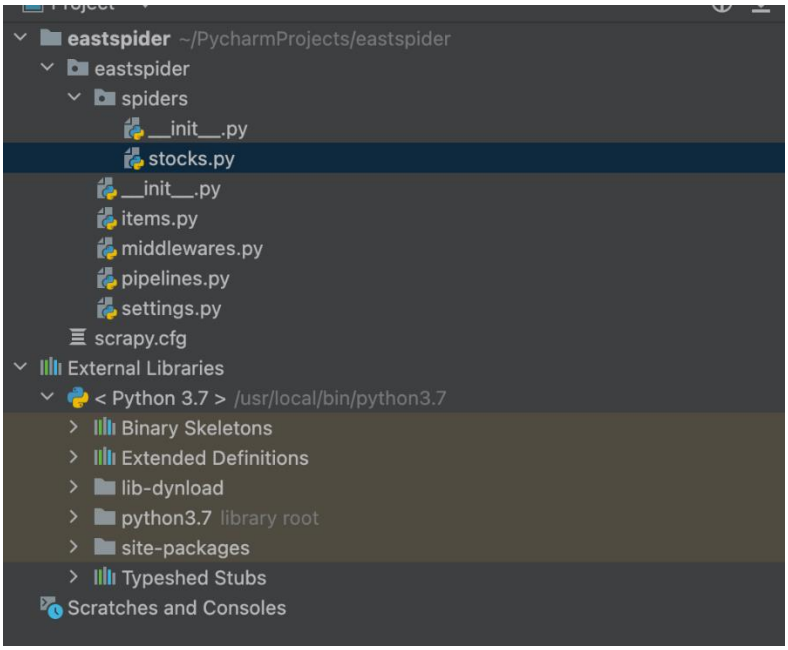

Figure 1. the structure of the project

2.5.3. Crawler module

The specific implementation of Scrapy crawler includes steps such as crawler configuration, request sending, page parsing and data extraction. The following is a simple example:

```
import scrapy
from stockdata.items import StockDataItem


class StocksSpider(scrapy.Spider):
name = 'stocks'
allowed_domains = ['xxxxx.com']
start_urls = ['http://xxxxxx.com/stocks']
```

```
def parse(self, response):
for stock in response.css('div.stock'):
item = StockDataItem()
item['name'] = stock.css('div.name::text').get()
item['price'] = stock.css('div.price::text').get()
item['volume'] = stock.css('div.volume::text').get()
item['change'] = stock.css('div.change::text').get()
yield item
```

2.5.4. Middleware Module

Scrapy middleware is a customizable component inserted into the Scrapy crawler request sending, response receiving and data processing process. Through a series of preset hook functions (such as process_request, process_response, etc.), developers can add custom logic before and after the core operation of the crawler to implement functions such as request and response modification, error handling, request priority adjustment and security authentication, which greatly enhances the flexibility and scalability of the framework. By adjusting the DOWNLOADER_MIDDLEWARES setting, you can control the activation and execution order of the middleware to meet diverse crawling needs.

User-Agent blocking is a security measure adopted by websites to protect content, maintain performance, copyright and security by identifying and blocking access requests from specific crawlers or automated tools. Faced with such restrictions, crawler developers often adopt strategies such as disguising or randomizing User-Agent, setting request delays, refusing to follow robots.txt rules and using proxy servers to reasonably circumvent blocking and ensure the effectiveness of crawling activities.

```
class RandomUserAgentMiddleware(object):
def __init__(self, user_agents):
self.user_agents = user_agents

@classmethod
def from_crawler(cls, crawler):
return cls(crawler.settings.get('USER_AGENTS'))

def process_request(self, request, spider):
request.headers.setdefault('User-Agent', random.choice(self.user_agents))
```

Website IP blocking is a security measure that aims to defend against malicious attacks, control crawler activities, protect copyrighted content, manage access rights, and maintain service stability by identifying and denying access requests from specific IP addresses. When a user's IP is identified as a potential threat or a violation of the usage policy, the website content will not be accessible, and an access error message will usually be displayed. Solving the blockade usually involves requesting unblocking, changing the IP address, or using proxy/VPN technology. Scrapy crawlers can use a variety of methods to break through IP blocking. For example, using a proxy is as follows

```
class ProxyMiddleware(object):
```

```
def __init__(self, proxies):
self.proxies = proxies

@classmethod
def from_crawler(cls, crawler):
return cls(crawler.settings.get('PROXIES'))

def process_request(self, request, spider):
request.meta['proxy'] = random.choice(self.proxies)
```

2.5.5. Data processing module

In the process of data collection, data cleaning and deduplication are important links to ensure data quality. The purpose of data cleaning is to remove noise and irrelevant information in the data to make the data neater and more structured. The captured data may contain some special characters, such as line breaks, tabs, etc. These characters are usually not needed during the analysis process and can be deleted or replaced. Some data fields may be empty or missing. For these missing data, they can be processed according to the specific situation, such as filling default values, deleting missing values, etc. Data standardization refers to converting data into a unified format for subsequent analysis. For example, the date format is unified into the "YYYY-MM-DD" format.

Data deduplication is to delete duplicate records and ensure that each piece of data is unique in the database. In this study, the post_id of the crawled content can be used as the unique field to remove duplicate data.

2.6. System Optimization

When using Scrapy for large-scale data collection, it is crucial to optimize crawler performance and efficiency. Here are some Scrapy optimization points that can significantly improve the speed and quality of data collection:

2.6.1. Concurrent request optimization

Scrapy uses an asynchronous request mechanism by default, which can handle multiple requests at the same time. You can increase the number of concurrent requests to speed up crawling by configuring:

```
# settings.py

CONCURRENT_REQUESTS = 32 # Default value is 16
CONCURRENT_REQUESTS_PER_DOMAIN = 16 # Default value is 8
CONCURRENT_REQUESTS_PER_IP = 16 # The default value is 0, indicating that it is not set
```

2.6.2. Request delay configuration

Properly setting the request delay can avoid excessive pressure on the target website and reduce the risk of being blocked:

```
# settings.py

DOWNLOAD_DELAY = 0.5 # 0.5 seconds between each request
```

RANDOMIZE_DOWNLOAD_DELAY = True # Random delay request

2.6.3. Using asynchronous download middleware

Enabling and configuring asynchronous download middleware, such as aiohttp or twisted, can improve download speed and concurrency performance:

settings.py

DOWNLOADER_MIDDLEWARES = {

'scrapy.downloadermiddlewares.httpcompression.HttpCompressionMiddleware': 810,

}

2.6.4. Enable caching

Enabling HTTP caching can reduce the number of requests to the target website during development and debugging, improving efficiency:

settings.py

HTTPCACHE_ENABLED = True

HTTPCACHE_EXPIRATION_SECS = 3600 # cache for one hour

HTTPCACHE_DIR = 'httpcache'

HTTPCACHE_IGNORE_HTTP_CODES = [500, 502, 503, 504, 400, 403, 404]

2.6.5. Distributed crawling using Scrapy-Redis

Combine Scrapy-Redis to achieve distributed crawling and improve the scalability and performance of the crawler:

settings.py

Enable scheduling storing requests queue in redis.

SCHEDULER = "scrapy_redis.scheduler.Scheduler"

Ensure all spiders share same duplicates filter through redis.

DUPEFILTER_CLASS = "scrapy_redis.dupefilter.RFPDupeFilter"

Use Redis pipeline for item processing.

ITEM_PIPELINES = {

'scrapy_redis.pipelines.RedisPipeline': 300

}

Specify the host and port to use when connecting to Redis (optional).

REDIS_HOST = 'localhost'

REDIS_PORT = 6379

2.6.6. Using the AutoThrottle extension

Enable the AutoThrottle extension to automatically adjust the crawling speed according to the load to avoid excessive pressure on the target website:

```
# settings.py

AUTOTHROTTLE_ENABLED = True
AUTOTHROTTLE_START_DELAY = 1 # Initial download delay
AUTOTHROTTLE_MAX_DELAY = 60 # Maximum download delay
AUTOTHROTTLE_TARGET_CONCURRENCY = 1.0 # Average number of concurrent requests per second
AUTOTHROTTLE_DEBUG = False # Enable debugging
```

2.6.7. Disable unnecessary components

Disable unnecessary middleware and extensions to reduce overhead and improve crawler performance:

```
# settings.py

SPIDER_MIDDLEWARES = {
'scrapy.spidermiddlewares.httperror.HttpErrorMiddleware': None,
'scrapy.spidermiddlewares.offsite.OffsiteMiddleware': None,
'scrapy.spidermiddlewares.referer.RefererMiddleware': None,
'scrapy.spidermiddlewares.urllength.UrlLengthMiddleware': None,
'scrapy.spidermiddlewares.depth.DepthMiddleware': None,
}
```

Through the above optimization measures, the performance and efficiency of the Scrapy crawler can be significantly improved, ensuring that the required data can be obtained efficiently and stably during large-scale data collection, providing reliable technical support for stock data analysis and mining.

3. CONCLUSIONS

This study systematically explores the method of using the Scrapy framework to obtain personalized stock data, and uses the collection of individual stock comment data as an example to demonstrate how to use crawler technology to achieve efficient data collection and processing. By optimizing crawler performance and storage algorithms, the limitations of traditional data acquisition methods in terms of dimension and efficiency are solved, providing a richer and more personalized data source for stock data analysis and mining.

3.1. Test Results

Taking the stock 601988 as a sample, 7009 data items were obtained. The following figure 2 is a partial screenshot

ts_code	post_id	stockbar_code
601988	1433219316	601988
601988	1433217452	601988
601988	1433208245	601988
601988	1433206393	601988
601988	1433106757	601988
601988	1432322589	601398
601988	1433059910	601988
601988	1432760411	600016
601988	1433196531	601988
601988	1432833128	601988

Figure 2. Sample of the result

3.2. Main Result

System design and implementation: Designed and implemented a stock data collection system based on the Scrapy framework. The system includes data acquisition layer, data processing layer and data storage layer, and combines Scrapy-Redis to achieve distributed crawling, which significantly improves data acquisition efficiency and system scalability.

Optimization strategies: The performance of the crawler system and the data storage efficiency are improved through multiple optimization strategies such as concurrent request optimization, request delay configuration, enabling asynchronous download middleware, HTTP caching, batch writing and incremental updates.

Technology selection: We rationally selected and applied technologies such as Scrapy, Scrapy-Redis, and MongoDB to ensure the efficiency, stability, and scalability of the system.

Data processing: Through data cleaning, format conversion, deduplication and other data processing methods, the accuracy and consistency of the data are ensured, the data quality is improved, and reliable data support is provided for subsequent stock data analysis and mining.

3.3. Significance

This study has made important explorations in stock data acquisition and provided new ideas and technical means for stock data analysis and mining. Through the application and optimization of the Scrapy framework, efficient acquisition of personalized data is achieved, overcoming the limitations of traditional data sources in terms of dimension and efficiency, and meeting the needs of stock analysts for high-quality, multi-dimensional data.

3.4. Future Outlook

Although this study has achieved certain results, there is still room for further optimization and improvement in practical applications. Future research can be deepened in the following aspects:

(1) Big data processing technology: Combined with big data processing platforms such as Hadoop and Spark, the collected data can be analyzed and processed on a larger scale to explore the deeper value of stock data.

(2) Machine Learning and Data Mining: Machine learning and data mining technologies are used to conduct intelligent analysis and prediction of collected data to provide more accurate stock decision support.

(3) Dynamic data collection: Develop a more intelligent crawler system that can dynamically adjust crawling strategies according to market changes and obtain the latest stock data in real time.

stock data from different data sources, multi-dimensional and multi-level data analysis, providing more comprehensive stock market insights.

In short, through continuous technological innovation and optimization, stock data collection and analysis will become more efficient and intelligent, providing stronger support for stock market research and decision-making.

REFERENCES

[1] Bootorabi, F., Haapasalo, J., Smith, E., Haapasalo, H. and Parkkila, S. (2011) Carbonic Anhydrase VII—A Potential Prognostic Marker in Gliomas. Health, 3, 6-12.

[2] Chen Hui. (2020). Research on cracking the web crawler blocking technology based on the scrapy framework. Science and Technology Horizon (6), 2.

[3] Maganioti, A.E., Chrissanthi, H.D., Charalabos, P.C., Andreas, R.D., George, P.N. and Christos, C.N. (2010) Cointegration of Event-Related Potential (ERP) Signals in Experiments with Different Electromagnetic Field (EMF) Conditions. Health, 2, 400-406.

[4] Zhao Benben [1], Yin Xudong [1], & Wang Wei [2]. (2016). GitHub data crawler based on scrapy. Electronic Technology and Software Engineering (6), 4.