

分类号 0175.2 密级 公开

UDC 004.72

学 位 论 文

(题名和副题名)

何城贤

(作者姓名)

指导教师姓名、职务、职称、学位、单位名称及地址 曹春 教授

申请学位级别 硕士 专业名称 计算机科学与技术

论文提交日期 2021 年 05 月 30 日 论文答辩日期 2021 年 05 月 25 日

学位授予单位和日期 _____

答辩委员会主席：_____

评阅人：_____

副教授

教授

研究员



南京大學

研究生毕业论文 (申请硕士学位)

论 文 题 目 响应式爬虫框架的研究与实现

作 者 姓 名 何城贤

学 科、专 业 方 向 计算机科学与技术

研 究 方 向 软件方法学

指 导 教 师 曹春 教授

2013 年 05 月 30 日

学 号：**MF1833025**

论文答辩日期：**2021 年 05 月 25 日**

指导教师： (签字)

Research and implementation of reactive crawler framework

by

Chengxian He

Supervised by

Professor Chun Cao

A dissertation submitted to
the graduate school of Nanjing University
in partial fulfilment of the requirements for the degree of
MASTER
in
Computer Software and Theory



Department of Computer Science and Technology
Nanjing University

May 1, 2021

南京大学研究生毕业论文中文摘要首页用纸

毕业论文题目： 响应式爬虫框架的研究与实现

计算机科学与技术 专业 2018 级硕士生姓名：何城贤
指导教师（姓名、职称）：曹春 教授

摘要

随着网络数据的爆炸性增长，网络爬虫技术被广泛应用于现实世界的各个领域，包括搜索引擎、舆情监控、数据挖掘等。然而在多爬虫任务并发爬取场景下，现有的开源爬虫框架，爬虫开发以及爬取效率都较低。现有框架需要对爬取网站的爬取链路、页面的解析规则以及反反爬策略进行繁琐的配置；同时在爬虫任务并发上，采用多线程同步编程模型抑或是单线程事件循环模型来进行爬虫任务的并发，无法充分利用系统 CPU 资源。

因此，本文提出了一种响应式爬虫框架，该框架针对多爬虫任务并发场景构建了一种响应式爬虫编程模型，并且基于该模型提出了一种基于网站结构的对象模型映射方法。通过网站结构映射，框架描述网站链接的爬取模式和网页数据的解析，避免了繁琐的网页解析规则配置，加快爬虫开发的效率。同时，响应式爬虫编程模型通过异步非阻塞的方式来执行爬虫任务，提高爬虫并发爬取过程中资源利用率，弥补了现有框架的不足。具体工作包括：

1. 提出了一种响应式的爬虫编程模型，通过构造异步数据流，将数据爬取过程中的阻塞操作通过异步来进行处理，提高了爬虫的运行效率。同时基于该编程模型，本文还提出了一种基于网站层次树结构的对象模型构造方法，避免了繁琐的网页解析规则配置，加快爬虫开发的效率。
2. 实现了一个响应式爬虫技术框架，针对于爬虫的网页下载、数据解析、代理配置、异常处理等模块进行了实现，支持功能扩展，二次开发。
3. 将本文提出的爬虫框架与其他的开源爬虫框架进行对比实验。实验表明，相比于现有的爬虫框架，本文提出的框架能够有效地提高网络数据爬取的吞吐量，同时提高资源利用率。

关键词：爬虫；响应式编程；流式处理

南京大学研究生毕业论文英文摘要首页用纸

THESIS: Research and implementation of reactive crawler framework

SPECIALIZATION: Computer Software and Theory

POSTGRADUATE: Chengxian He

MENTOR: Professor Chun Cao

Abstract

With the explosive growth of web data, web crawler technology is widely used in various fields of the real world, including search engine, public opinion monitoring, data mining and so on. However, in the scenario of multi crawler task concurrently crawling, the existing open source crawler frameworks have low efficiency of crawler code developing and crawler crawling. The existing framework needs to configure the crawling link, page resolution rules and “anti-anti-crawler” strategy of crawling website; At the same time, the multi-threading synchronous programming model or single threaded event loop model is used to concurrence crawler tasks in the concurrent crawler task, which can not make full use of the CPU source of the system.

Therefore, we propose a reactive crawler framework, which constructs a reactive crawler programming model for scenarios of multi crawler task concurrently crawling, and proposes a mapping method of object model based on the website structure. Through the mapping of website structure, we can describe the crawling mode of website links and the analysis of Web data, avoid the complicated configuration of web page analysis rules and speed up the efficiency of crawler development. Meanwhile, the reactive crawler programming model performs the crawler task by asynchronous non-blocking method, improves the resource utilization rate in the process of crawler concurrent crawling, and makes up for the deficiency of the existing framework. Specifically, the work of this paper mainly includes the following aspects:

1. This paper proposes a reactive crawler programming model. By constructing asynchronous data stream, the blocking operation in the process of data crawling is processed asynchronously, which improves the efficiency of crawler. At the same

time, based on the programming model, this paper also proposes an object model construction method based on hierarchical tree structure to avoid the complicated configuration of web page analysis rules and speed up the efficiency of crawler development.

2. We implement a reactive crawler framework, which implements the modules of web page downloading, data analysis, proxy configuration, exception handling, and supports function expansion and redevelopment.
3. We simulate a special website, and compare the crawler framework with other open source crawler frameworks. Experiments show that compared with the existing crawler framework, the proposed framework can effectively provide the throughput of web data crawling and improve the resource utilization.

keywords: Web Crawler, Reactive Programming, Stream processing

目 次

目 次	v
插图清单	ix
附表清单	xi
1 绪论	1
1.1 研究背景	1
1.2 研究现状	1
1.3 本文工作	3
1.4 本文组织	4
2 相关工作和背景技术	5
2.1 网络爬虫	5
2.1.1 爬虫分类	5
2.1.2 爬虫基本架构	6
2.1.3 爬虫基本流程	7
2.2 响应式编程	9
2.2.1 响应式规范	11
2.2.2 响应式系统	12
2.3 本章小结	12
3 异步流式爬虫框架	15
3.1 传统爬虫编程模型	15
3.1.1 多线程爬虫编程模型	15
3.1.2 单线程事件循环爬虫编程模型	16
3.1.3 分布式爬虫模型	17
3.2 响应式爬虫编程模型	18
3.2.1 组件非阻塞约束	19

3.2.2 网站结构映射	20
3.2.3 异步数据流构建	22
3.2.4 反反爬策略设置	23
3.3 本章小结	26
4 框架实现	27
4.1 Project Reactor 介绍	27
4.2 爬虫框架模块	27
4.2.1 数据模型定义模块	29
4.2.2 组件编排模块	29
4.2.3 流程的编排	37
4.2.4 异常处理模块	37
4.2.5 监控模块	39
4.2.6 Web 模块	40
4.3 Spidereact 爬虫开发	40
4.3.1 数据对象定义	40
4.3.2 组件定义	42
4.3.3 爬虫编排	42
4.3.4 爬虫反反爬策略配置	43
4.4 本章小结	44
5 实验设计与评估	45
5.1 实验内容	45
5.2 实验设置	45
5.2.1 实验环境	45
5.2.2 实验对象	46
5.2.3 模拟延迟	48
5.3 实验结果与分析	49
5.3.1 CPU 利用率验证	49
5.3.2 吞吐量对比	51
5.3.3 资源限制	53
5.4 实验结论	53
5.5 本章小结	55

6 结论	57
6.1 工作总结	57
6.2 研究展望	57
致 谢	59
参考文献	61
简历与科研成果	65
学位论文出版授权书	67

插图清单

2-1 网络爬虫架构	6
2-2 反爬策略与反反爬策略	10
2-3 Reactive Streams 规范	11
3-1 传统爬虫多线程模型	16
3-2 单线程事件循环爬虫编程模型	17
3-3 响应式爬虫编程模型	19
3-4 层次树模型	20
3-5 层次树模型映射	22
3-6 异步数据流	23
3-7 爬虫频率配置	25
3-8 Request 数据流配置	25
3-9 爬取次数限制和背压配置	26
4-1 Spidereact 功能模块图	28
4-2 组件编排模块 UML 类图	30
4-3 下载器下载流程	34
4-4 链接提取流程	35
4-5 对象映射流程	36
4-6 流程编排逻辑视图	38
4-7 异常处理流程图	39
4-8 Web 模块 UML 类图	41
5-1 Scrapy 不同 CONCURRENT_REQUESTS 配置下吞吐量	48
5-2 Webmagic 0 时延不同并发下 CPU 使用情况	49
5-3 Webmagic100 并发度不同时延下的 CPU 利用率	51
5-4 Spidereact 40 并发下不同时延下 CPU 使用情况	52
5-5 Webmagic 不同并发下吞吐量	53

附表清单

4-1 主要使用的注解.....	29
4-2 两种基本数据类型	31
4-3 自定义异常类型.....	38
5-1 硬件配置.....	45
5-2 软件环境	46
5-3 模拟网站压测	47
5-4 不同延迟不同并发下吞吐量对比	54
5-5 不同 CPU 资源不同并发下吞吐量对比	55

第一章 绪论

1.1 研究背景

万维网信息数据的激增和网页数量的爆炸式增长，引发了大数据、物联网、云计算和人工智能等热门概念。根据 IDC《数据时代 2025》的报告显示，预计到 2025 年全球数据总量将攀升至 163ZB，相当于 2016 年所产生的 16.1ZB 数据的十倍^[1]，互联网数据在其中的占比也呈现越来越高的趋势。在此背景下，如何获取这些重要数据成为了利用并开发其价值的前提，在庞大的网络数据中搜寻获取结构化的数据信息成了一个越来越重要的问题。

在此背景下，网络爬虫成为了解决问题的关键。网络爬虫，也被称为网络蜘蛛，是一种用来自动浏览万维网的网络机器人^[2]。当前的爬虫根据其爬取内容和爬取范围来划分，主要可分为通用网络爬虫、聚焦爬虫以及垂直爬虫。通用网络爬虫和聚焦爬虫关注的是网页的收集以及索引库的建立，而垂直爬虫的重点在于特定的结构化数据的抽取。

垂直爬虫，不像通用网络爬虫的全网爬取，往往是针对某些特定的网站，这些网站数据通常具有结构化或者是半结构化的特征。传统的垂直爬虫通常采用多线程模型^[3]，通过线程池对网页数据进行并发爬取，但存在爬虫请求过载、线程安全以及锁机制实现等问题。另外动态实时数据的表示、反爬策略的配置、页面解析规则的编写、异常处理以及登录验证也是垂直爬虫编写的难点^[4]。因此，如何提高垂直爬虫并发的效率以及爬虫开发的效率是垂直爬虫框架亟需解决的问题。

1.2 研究现状

针对于垂直爬虫并发的问题，现有的爬虫框架一般是通过多线程同步处理抑或是单线程事件循环模型来解决，同时爬虫框架通过组件化的构造来提升爬虫开发的效率。从当前垂直爬虫框架的设计目标来看，现有工作旨在提升开发效率、提高资源利用率等。每个爬虫框架都希望能够充分利用系统的计算资

源，通过线程的并发来加快爬虫爬取网页数据的速度，使得垂直爬虫能够在获得较高的吞吐量。

Scrapy^[5] 是一个基于 Python 的网页爬取框架，主要针对网页数据进行爬取，提取出其中的结构化数据。Scrapy 底层网络通信是基于 Twisted 异步通信框架，相比于同步通信框架，Twisted 是基于事件驱动的网络框架，有更高的吞吐量。同时 Scrapy 框架还内置了许多中间件接口，允许用户自定义实现各种中间件，例如 downloader, scheduler, pipeline 等等。其爬取主要通过先通过初始 URL 构造 Request 对象，经 Scrapy Engine 传递给 Scheduler 进行调度，Scheduler 将 Request 传递给 Downloader 进行网页下载，下载返回的 Response 再由 Pipeline 进行处理，生成需要的结构化的数据以及下次爬取的链接，这样构成一个完整的爬取流程。

Apache Nutch^[6] 是一个高度可扩展的网页爬取引擎，主要用于收集网页数据，对数据进行分析，创建索引，然后提供相应的接口来对网页数据进行查询。其底层是基于 Hadoop 来做的分布式计算和存储，索引则使用了 Solr 分布式索引框架来实现。它能够快速地爬取大量的网页，并为这些网页提供索引。但 Nutch 更多的是支持通用爬虫的全网爬取，对于垂直爬虫来说，Nutch 的爬虫定制能力较弱，同时 Nutch 本身还依赖于 Hadoop，需要额外的配置。

Webmagic^[7] 是一个基于 Java 的多线程网页爬虫框架，其架构设计参考了 Scrapy 的架构，同样有 Downloader, Scheduler 以及 Pipeline 等组件。其底层是通过 HttpClient^①线程池来进行网络 IO，以及通过 Jsoup 对网页数据进行解析。该框架提供模块化的结构，还提供了一种通过注解配置爬虫的开发模式。但 Webmagic 不支持 JS 页面抓取，不支持 IP 代理和 User Agent 的切换，同时也没有提供异常处理机制。

现有爬虫框架均能解决垂直爬虫中数据爬取问题，但都有其局限性。现在将其不足主要划分为以下几点：

- 爬虫页面解析繁琐：爬虫代码的编写主要是对网页结构数据的解析，解析过程主要是通过将网页的 DOM^②对象中的元素给提取出来，但爬虫页面的解析代码过于繁琐，难以调试。
- 难以对爬虫进行配置：爬虫需要根据网站的反爬策略进行相应的配置，以便获得正常的网页数据。但目前的爬虫框架没有提供 IP 代理、User Agent

^①<https://hc.apache.org/httpcomponents-client-5.0.x/>

^②https://en.wikipedia.org/wiki/Document_Object_Model

以及 Cookies 的切换机制，无法配置相应的切换策略。

- 无法充分利用系统资源：用传统的同步阻塞的线程池模型进行爬取，往往会遇到这样的问题，对于常见的网络 IO 操作，由于网络延迟或者请求处理时间长的原因，线程会处于阻塞状态，导致线程调度频繁。但线程池中的线程数量是有限的，当同时下载的页面数量超过一定限度时，盲目地增加线程数已经不能解决问题。而单线程异步驱动方案则无法充分利用 CPU 多核的优势。
- 缺少异常处理机制以及登录验证机制：目前爬虫过程中往往遇到各种各样的情况，其中请求失败以及解析异常是最常见的问题，需要爬虫针对不同的异常提供异常处理机制。另外，爬虫的登录验证往往也是爬虫爬取过程中的一个难点，目前的爬虫框架鲜有对登录验证的支持。

1.3 本文工作

针对以上爬虫框架中存在的不足，本文提出了 Spidereact，一种响应式爬虫框架。为了更准确地刻画出网页结构与对象模型的映射关系，Spidereact 首先提出了一种网络数据模型来映射网页元素，使得开发者能够方便地描述网页结构。同时，为了能够提高系统资源利用率，增加吞吐量，Spidereact 对各个爬虫组件进行组装，生成异步处理流，从而能够提高爬取的效率。本文的工作内容总结如下：

- 为了提高吞吐率，提高资源利用率，本文提出了一种异步流式数据爬取模型，通过对爬虫各个组件进行组装，提供流式数据接口，实现数据的流式爬取、组件的异步执行，同时爬虫通过数据流组合进行配置，解决了爬虫配置切换问题。在该模型基础上，本文根据网络数据的结构特性，提出了一种基于网站层次树结构的对象模型构造方法，用来描述网页元素和对象属性之间的关系，避免了对爬虫爬取链路以及页面解析规则的繁琐配置，能够提高爬虫开发的效率。
- 实现了响应式爬虫框架，针对于爬虫的网页下载、数据解析、代理配置以及异常处理等模块进行了实现。开发者可以根据不同的网站来声明式地编排爬虫，获取特定的结构化数据。
- 为了验证框架的有效性，本文对提出的爬虫框架进行了性能对比实验。通过与当前热门爬虫框架的对比分析系统吞吐率，CPU 利用率情况，验证了

`Spidereact` 相比与现有框架，确实在某些应用场景下提高了系统资源的利用率以及吞吐量。

1.4 本文组织

本文各章节组织如下：

第一章绪论，主要介绍论文的研究背景，以及研究现状和研究的主要内容。

第二章本文相关工作和背景技术，首先介绍了当前爬虫的主要分类，然后介绍了爬虫的基本架构以及基本流程，最后再介绍了响应式编程范式以及响应式系统。

第三章异步流式爬虫编程模型，首先介绍了传统爬虫的编程模型，并分析了传统模型的利弊，在此基础上，提出了一种响应式的爬虫编程模型。

第四章系统实现，主要介绍了爬虫框架的主要架构，还依次介绍了数据源初始化组件、网页下载器组件、链接提取器组件、网页数据映射组件以及持久化处理组件等组件情况，以及异常处理模块设计。

第五章实验设计与分析。主要对爬虫框架进行了测试与分析，首先介绍了实验设置和实验的内容，然后将本文提取的爬虫框架主要与当前热门的爬虫框架进行了对比，并验证了本文提出的爬虫框架在爬取速率，资源利用率等方面的性能提升。

第六章总结与展望，对本文工作进行总结，对未来工作进行了展望。

第二章 相关工作和背景技术

2.1 网络爬虫

网络爬虫，也被称为网络蜘蛛，是一种用来自动浏览万维网的网络机器人^[8]。它根据一定的规则，自动的抓取万维网上的网页信息。网络爬虫技术被广泛应用于互联网各个领域。在搜索引擎领域，网络爬虫通过全网的爬取收集网页信息，为搜索引擎建立索引库提供支持。在舆情监控领域，网络爬虫通过对目标网站的信息自动化抓取，实现网络舆情的监控以及新闻专题的追踪等需求^[9]。在数据挖掘领域，网络爬虫为互联网网页数据的采集提供相应技术手段^[10]。

2.1.1 爬虫分类

网络上常见的爬虫大体可分为以下 3 类：

- 通用爬虫：通用爬虫是搜索引擎的重要组成部分^[11]，像百度的通用爬虫 baiduspider，谷歌的通用爬虫 googlebot。通用爬虫通过对网络上的网页进行下载存储，建立相关索引库，提供相应的查询索引接口。通用爬虫从种子 URL 出发，扩展到整个 Web 上的网站，爬取的范围和数据十分巨大，其关注点在于爬取的速度以及数据的存储。
- 聚焦爬虫：又被称为主题爬虫，它主要是面向特定主题的爬取^[12]，例如可以部署一个聚焦爬虫来收集网络上关于太阳能、禽流感或者是其他的抽象主题概念的页面。和通用爬虫相比，聚焦爬虫爬取的范围较小，只会爬取和主题相关的页面，它在爬取过程中会对爬取的页面或是链接根据一定的方法进行筛选，从而节省网络和硬件资源。
- 垂直爬虫：垂直爬虫主要针对于某些特定领域的网站进行爬取，这些网站往往结构比较规范，同时爬取的内容不再是网页，而是需要对网页内容进行解析，得到结构化的数据。例如对于微博、知乎等门户网站进行爬取的垂直爬虫，主要是将网站的结构化信息进行提取，得到结构化的数据之后

存储到数据库中。和以上两种爬虫相比，垂直爬虫更注重爬取内容精确性以及数据结构化特征。

通用爬虫的重点是在于对全网的爬取，给搜索引擎提供索引。聚焦爬虫则是对特定主题内容的爬取^[13]，要求对和主题相关性较高的页面进行爬取。垂直爬虫便是针对于特定网站的爬取，其爬取实质是对在同一域名下的网页进行遍历，不会爬取其他域名下的网页。本文将重点放在了垂直爬虫领域，对于特定网页下提取出相应的结构化或者是半结构化的数据。

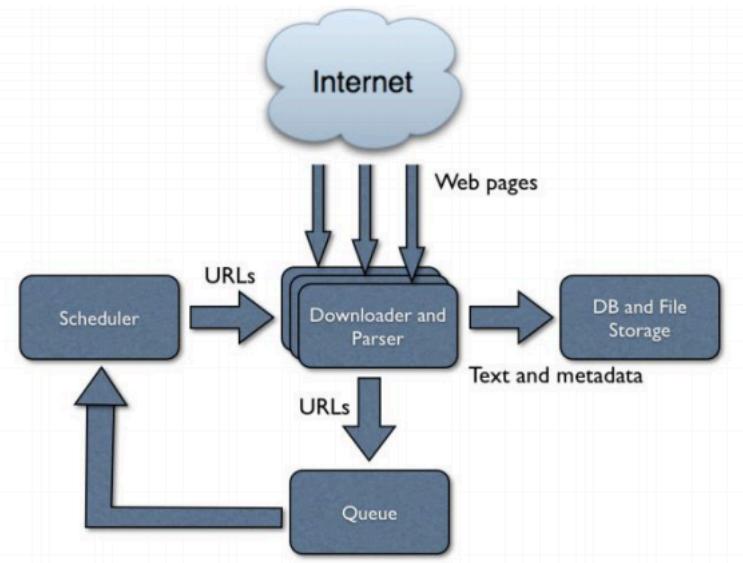


图 2-1: 网络爬虫架构

2.1.2 爬虫基本架构

前文介绍了不同的爬虫框架的各自的特点，但爬虫基本架构基本上是相同的。一个基本的爬虫架构如图 2-1 所示，主要包含了以下几个组件^[14]:

- **Scheduler:** 调度器，通过 Scheduler 决定哪些 URL 将要传入 Downloader 进行下载，根据 URL 的优先级来调度 URL 请求的发布。
- **Queue:** URL 队列，存储着待爬取 URL，在爬虫初始化时初始化该队列。
- **Downloader:** 下载器，从网络环境中下载 HTML 页面。
- **Parser:** 对网络页面进行解析，包括所需要的结构化数据以及页面中的超链接，然后根据相关算法对链接进行筛选，最后倒入到 URL 队列中。
- **Storage:** 数据存储，将已经解析好的结构化或者半结构化数据进行处理，数据持久化，以便后续处理。

2.1.3 爬虫基本流程

一个爬虫基本的爬取流程^[15]，便是从一个初始的 URL 集合出发，下载 URL 对应的页面；并对页面内容进行解析，提取出相应的页面元素，以及页面上的链接；然后对页面链接进行过滤操作，获取所需要追踪的链接，将提取出的新链接放到任务队列中去；最后直到队列中没有新的链接，爬虫便会结束。其中网页内容的解析、链接跟进策略以及链接过滤是爬虫能够爬取目标数据的关键。在实际的爬取场景中，爬虫爬取的网站可能会存在动态页面加载的问题，给爬虫的爬取带来一定的困难。另外，大型网站往往还会设置一定的反爬策略来限制爬虫机器人的爬取，这时便需要为爬虫配置相应的反反爬策略来绕过网站的限制。

2.1.3.1 网页解析：XPath 和 CSS 选择器

网页数据的解析，即对下载器下载的 HTML 文件中的数据进行提取，需要网页元素的定位与匹配。通常爬虫都是采用 XPath 或者是 CSS 选择器来对页面进行解析^[16]。

Xpath 即 XML Path，表示 XML 文档的路径，是一种用来识别 XML 文档元素的查询语言^[17]。爬虫能够通过 Xpath 表达式来寻找文档中的特定节点，Xpath 会根据 XML 树状结构，提供在数据结构树中寻找节点的能力。由于 Xpath 的这一特性，爬虫可以一层一层地遍历 XML 结构树，相较于 CSS 选择器，Xpath 更加灵活，它能够在 XML 树中任一节点的任一方向进行搜索，例如探寻该节点的父辈节点或是查找具有某些属性的子节点。大部分 HTML 页面都通过层叠样式表来设置网页样式，而这个层叠样式表就是 CSS，它是一种样式表语言，用来描述 HTML 或是 XML 文档。CSS 选择器相比于 Xpath，它能更快地找到对应元素，可读性较好。

2.1.3.2 链接跟进策略

爬虫爬取过程中，后续链接的提取是问题的关键，它决定了爬虫爬取的方向^[18]。

广度优先搜索策略是最常见的做法，它会维护一个先进先出队列，每次都把将要爬取的链接入队，并且按照链接提取的顺序进行链接的爬取。该策略属于盲目搜索策略，不会考虑当前搜索位置，会遍历整个网站，通常会爬取到一

些无用的页面，效率较低。

深度优先搜索策略则是从种子页出发，一个链接一个链接的跟进，直到该链路已经没有了后续链接，则结束该链路的爬取，进行回溯，从上一级页面开始爬取。该策略无法保证链接提取的顺序和爬取顺序一致，同时如果没有限定爬取深度，可能会陷入到有着无穷后续链接的链路。

启发式的最佳优先搜索算法，基于某种估计、分数或者是优先级来选取最合适的链接进行跟进。最有名的便是 fish-search 算法^[19]，该算法基于相关文档会包含更多相关文档的链接的假设，维护一个 URL 列表，通过 Depth、Width 和 potential_score 参数来表示网页的深度、每页最多爬取的链接数以及网页相关度，根据 potential_score 来动态改变 URL 在列表中的位置，将优先级更高的 URL 排在列表前列，先进行爬取。该算法能够改变传统策略中按照 URL 出现次序依次爬取的机制，优先处理了对相关网页的爬取。

2.1.3.3 链接过滤

为了防止爬取到重复页面，爬虫需要考虑重复 URL 的识别。爬虫实现重复 URL 过滤，可以通过内存对象存储所有的 URL 或者是嵌入式数据库^[20]（例如：Berkeley DB），每次提取到的新的链接时，需要查询内存对象或嵌入式数据库，查看是否已经重复，但这种方法内存消耗较大，随着爬取的 URL 数量的增加，内存压力会越来越大。链接过滤可以通过将 URL 集合存储到外部数据库（例如 Redis 数据库）中，但这样会存在的一定的网络通信开销。另外还可以通过布隆过滤器^[21]（Bloom Filter）来检索 URL 是否在已爬取 URL 集合中，这种方法的空间效率和时间效率都要超过一般的算法，但布隆过滤器存在一定的误报率，并且删除操作很困难。

2.1.3.4 动态页面加载

动态页面与静态页面相对应^[22]，静态页面是指存在于服务器上的文件系统中的 HTML 文件，当用户通过浏览器访问相应的 URL 时，服务器会将 HTML 文件返回给浏览器，浏览器再加以渲染呈现给用户，早期的网站通常都是些静态页面。

动态页面则与之不同，页面中的许多数据往往需要 Ajax^①技术动态生成，并且许多数据并不会直接出现在 HTML 源代码中，需要经过浏览器渲染才会得

^①[https://en.wikipedia.org/wiki/Ajax_\(programming\)](https://en.wikipedia.org/wiki/Ajax_(programming))

到数据。页面上的元素并不是一成不变的，浏览器会通过 js 调用异步通讯组件并使用格式化的数据来更新 Web 页面上的内容。对于这些需要 js 代码动态加载的数据，爬虫往往很难爬取。开源爬虫框架采用的下载器下载的页面都是网页源代码，并未对 js 动态加载的数据进行渲染，需要相关工具能够对这些动态 js 页面进行渲染，这样才能使爬虫获取相应的数据。

2.1.3.5 反爬策略与反反爬策略

反爬是指网站通过一定的反制措施来干扰爬虫的正常运行，从而间接地起到网站防护的作用。反反爬则是爬虫通过绕过网站设置的反爬措施来获取所需要的数据。如图 2-2 所示，网站反爬策略以及爬虫的反反爬策略主要分为以下几种情况：

1. User Agent 检测：网站通过从请求的 Header 中获取 User Agent 进行检测是一种比较常见的方法，爬虫可以通过模拟 User Agent 的方式绕过。
2. IP 限制：限制 IP 访问次数，通过检测用户行为，如一定时间段内，IP 访问请求次数过多，对 IP 加以限制。爬虫往往通过构造代理 IP 池，以及限制访问频率的方法加以绕过。
3. Ajax 页面动态加载：页面通过 Ajax 进行数据的动态加载，使得爬虫只能爬取页面源代码，无法获取所需要的数据。这种情况，爬虫往往通过一些工具模拟浏览器操作的方式绕过。
4. 登录认证：网站设置某些数据的访问需要用户登录。爬虫通过模拟登录过程，获得 Cookies，每次访问都带上 Cookies 进行网页爬取。
5. 验证码识别：网站设置访问验证，需要用户手动验证来访问网页。爬虫通过 OCR 技术识别验证码，或者通过人工打码平台进行识别，甚至是手动验证。

2.2 响应式编程

响应式编程是基于异步数据流的编程范式，面向数据流和变化传播。响应式编程基于流式处理，可以把包含一堆异步事件的组合用操作符给清楚的表示，将异步回调模式表示成了邻接节点的关系，能够对数据的传播流动的方向加以把握。

实际上 Reactive Programming 并不是一个新鲜的概念，它用来解决异步非

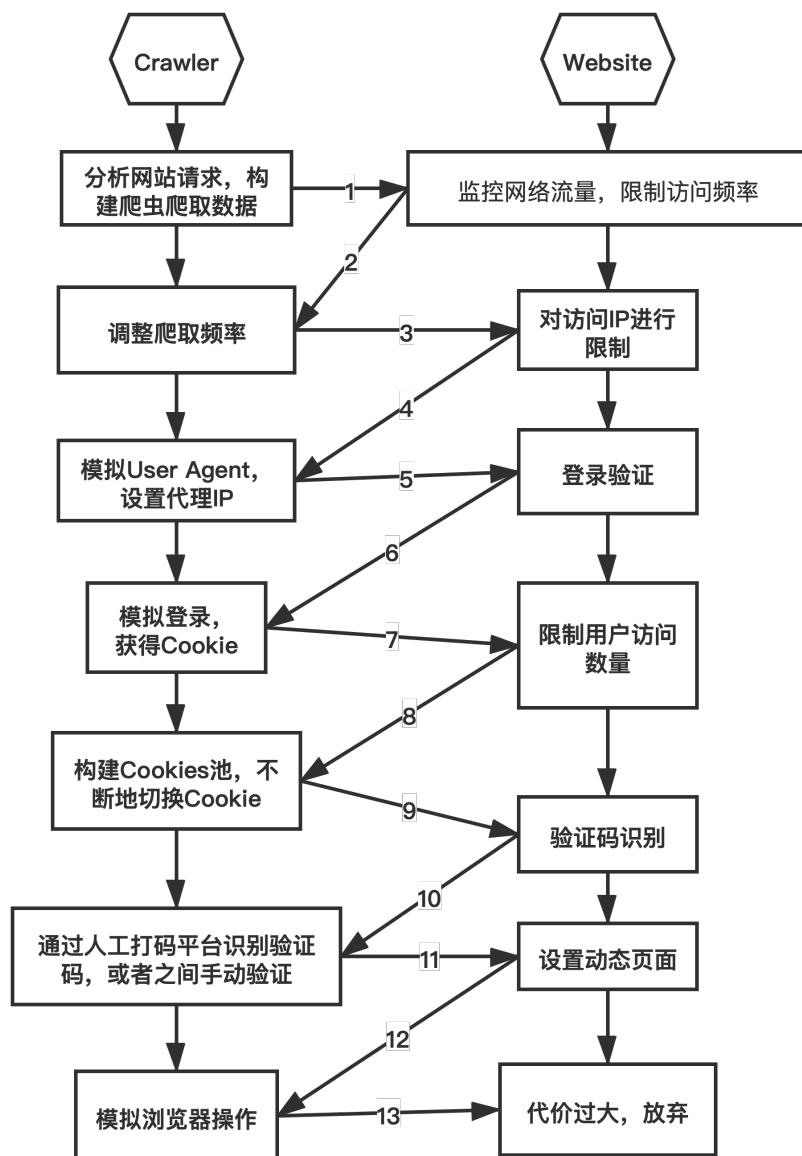


图 2-2: 反爬策略与反反爬策略

阻塞编程中存在的问题。虽然异步编程可以通过回调函数来处理，但是随着业务逻辑越来越复杂，回调函数这种模式会陷入到“回调地狱”的困境，而 Java 8 引入的 `CompletableFuture`，虽然能够处理异步事件的组合，但还是会存在各种各样的问题^[23]:

- 对于 `Future` 对象的处理，往往容易陷入到一种阻塞的情景，依然需要调用阻塞的 `get` 方法。
- 不支持任务的延迟执行。

- 缺乏对多值属性的表示，另外还缺乏友好的异常处理机制。
- 难以优雅地实现异步任务的编排。

2.2.1 响应式规范

响应式编程的出现便是来解决异步编程中存在的问题，响应式编程需要遵守 Reactive Streams^[24] 标准。Reactive Streams 是响应式编程规范，如今已经被多个企业所采用，如 Netflix、Oracle 等。在 Java 9 之后，java.util.concurrent.Flow 也支持了这个标准。

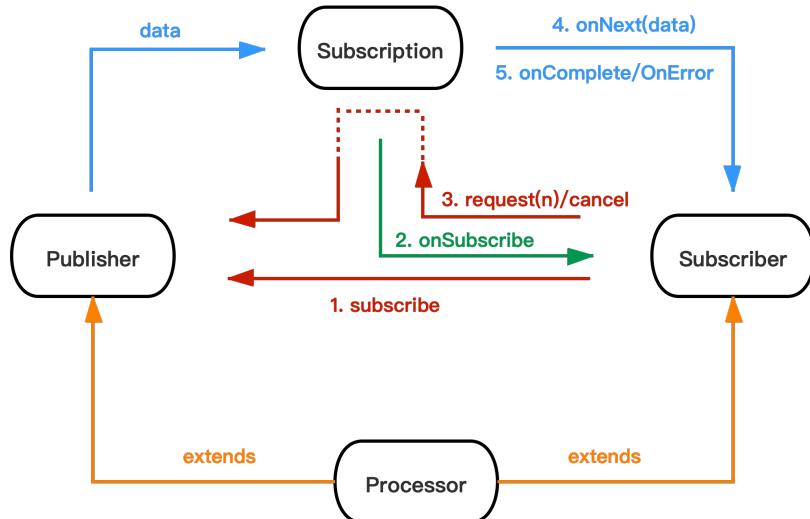


图 2-3: Reactive Streams 规范

Reactive Streams 的目的在于通过流处理机制提供一个异步数据处理序列^[23]。如图 2-3 所示，它需要实现 4 个接口，来管理跨异步边界的流式数据交互，即将数据在线程池中传递，同时还要确保接收方不需要强制缓存任意数量的来自发布方的数据。换而言之，需要实现背压机制，线程之间的缓冲队列是有界限的。

4 个接口如下所示：

- **Publisher:** 数据源，发布 0 到 N 个信号，其中 N 可以是无限的。另外它还可以提高两个终止事件：error 和 completion。
- **Subscriber:** 和数据源对应，是数据消费者，可以消费从 0 到 N 个信号，N 可以是无限的。Subscriber 会在初始化的时候接收一个 Subscription 对象，通

过它来请求要消费的数据。

- **Subscription: Subscriber** 初始化的时候创建，它来控制下一步消费多少数据以及何时停止消费。
- **Processor:** 既可作为数据源，也可以作为消费者。

2.2.2 响应式系统

随着计算机软件技术的发展，应用程序的需求与原来相比已经发生了巨大的改变，之前一个大型应用程序只需要跑在数十台服务器上，要求秒级的响应时间^[25]。然而，现在的用户希望能够达到毫秒级别的响应，以及系统能够处理 PB 级的数据量，之前的软件架构已经无法满足当前的需求。人们需要一套贯通整个系统的架构设计方案，系统需要能更加的灵活，系统的开发和调整也能够变得更加容易；同时系统必须松耦合，系统组件容器被及时替换；另外还必须具备可伸缩的特性，能够水平的扩展应对高负载情况。总结来说，系统必须具有存在着这样的特质^[26]：

- 即时响应性 (Responsive): 只要有可能，系统就会及时地作出反应。即时响应意味着系统专注于提供快速且一致的响应时间，确定可靠的反馈上限，从而提供稳定的服务质量。
- 回弹性 (Resilient): 回弹性意味着即使系统出现了异常或者是失败，仍然能够保持即时响应性，这能确保系统的高可用。而回弹性的实现，是通过复制，隔离和委托来实现。对于一个响应式系统而言，组件与组件之间的隔离，抑制失败在组件间的传播，使得系统部分的失败无法影响到系统整体，并且能够独立恢复。
- 弹性 (Elastic): 系统能够根据工作负载做出反应，调整相应的资源配置来保证系统的即时响应性。换句话说，系统能够根据负载进行自动扩展。
- 消息驱动 (Message-driven): 系统依赖与异步的消息驱动，通过异步的消息来实现系统各个组件之间的松耦合关系，给组件明确的边界。并通过回压等手段，使得系统实现负载管理以及流量控制。

2.3 本章小结

本章首先对爬虫进行了介绍，阐述了三种类型的网络爬虫，分别是通用爬虫、聚焦爬虫以及垂直爬虫，并对爬虫涉及的相关背景技术作了简要介绍。本

章还介绍响应式编程相关内容，包括响应式编程范式基本概念，响应式规范以及响应式系统。

下一章将对本文提出的异步流式爬虫框架进行详细介绍。

第三章 异步流式爬虫框架

本章首先介绍了传统爬虫的编程模型，分析了传统编程模型的利弊，并在此基础上，提出了一种异步流式的响应式爬虫编程模型。

3.1 传统爬虫编程模型

传统爬虫编程模型主要有三类，第一类多线程爬虫编程模型，爬虫内部通过线程池来实现并发爬取；第二类是单线程事件循环爬虫编程模型，爬取主流程是跑在单个线程上的，IO 操作通过事件循环异步执行；第三类则是分布式爬虫模型^[27]，爬虫主要借助于 Hadoop、Storm 等大数据平台，通过平台提供的组件进行爬虫的组装。

3.1.1 多线程爬虫编程模型

多线程爬虫是最常见的爬虫场景，目前大部分的开源框架都是采用的多线程的爬虫模式。例如 Webmagic 框架便是采用的多线程模型进行爬取，通过配置相应的线程池来处理每一个爬虫请求。相比于其他编程模型而言，多线程实现比较简单。

传统多线程模型如图 3-1 所示，使用同步编程模型，每一个 URL 请求都对应一个线程处理^[28]。该线程负责页面的下载、解析、链接过滤等操作，在下载阶段，会进行相应的网络 IO 操作，需要从网站获取 Response，此时请求线程必须同步等待以获得响应。在这种情况下，线程处于等待状态，CPU 空闲，所以该模式会采用一个线程池来处理，目的就是当有线程处于等待状态时，能够通过线程的调度，让已经就绪的线程调度执行，充分利用 CPU 的计算资源。线程池模式对于单位时间内请求量较少的爬虫程序来说是可以接受的，但是对于单位时间内请求量大的爬虫而言，线程池编程模型最终会让爬虫程序变慢或者是请求无法得到响应。这种全链路线程池处理模式，当并发量逐渐增大时，会使得线程间的调度开销增大，同时由于 CPU 资源短缺，可能会使得已经就绪的线程得不到及时的调度。

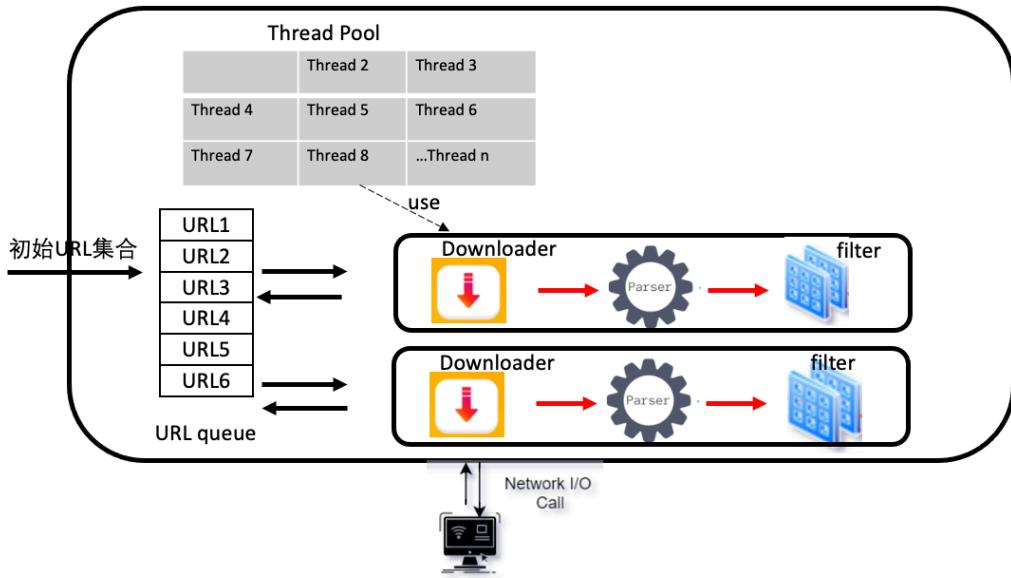


图 3-1: 传统爬虫多线程模型

多线程爬虫编程模型，其优点是编码简单、能够提高响应速度、在多 CPU 的环境下能够体现多核的优势。对于请求量不大的爬虫来说，多线程模型是可以接受的。但在负载大的情况下，该编程模型会消耗大量的内存，因为每个爬虫请求都要维护一个线程堆栈。另外持续的上下文切换也会导致大量的 CPU 时间损失。由此产生的间接影响便是 CPU 高速缓存未命中的概率增加，减少线程池的绝对数量可以提高单个线程的性能，但是限制了爬虫请求的可伸缩性。

3.1.2 单线程事件循环爬虫编程模型

对于单线程事件循环爬虫编程模型^[29]，主线程执行事件循环，通过线程池来执行异步任务，通过事件循环和线程池来实现异步 IO。例如 Scrapy 爬虫，它通过 Twisted 异步通信框架来实现事件循环，Scrapy 框架向开发者屏蔽了底层编程模型的细节，将回调函数暴露给开发者来编写页面解析逻辑。

如图 3-2 所示，该模型下，爬虫通过一个请求队列来缓存到达的每一个请求，通过一个主线程的事件循环不断地处理请求以及回调事件。每一个到达的请求的阻塞网络 IO 操作放在线程池中异步执行，并注册回调函数，当阻塞任务执行完成后，会触发事件，在主线程中调用非阻塞的回调函数。爬虫主线程一直处理非阻塞的页面解析代码，而通过线程池异步执行网页下载请求。

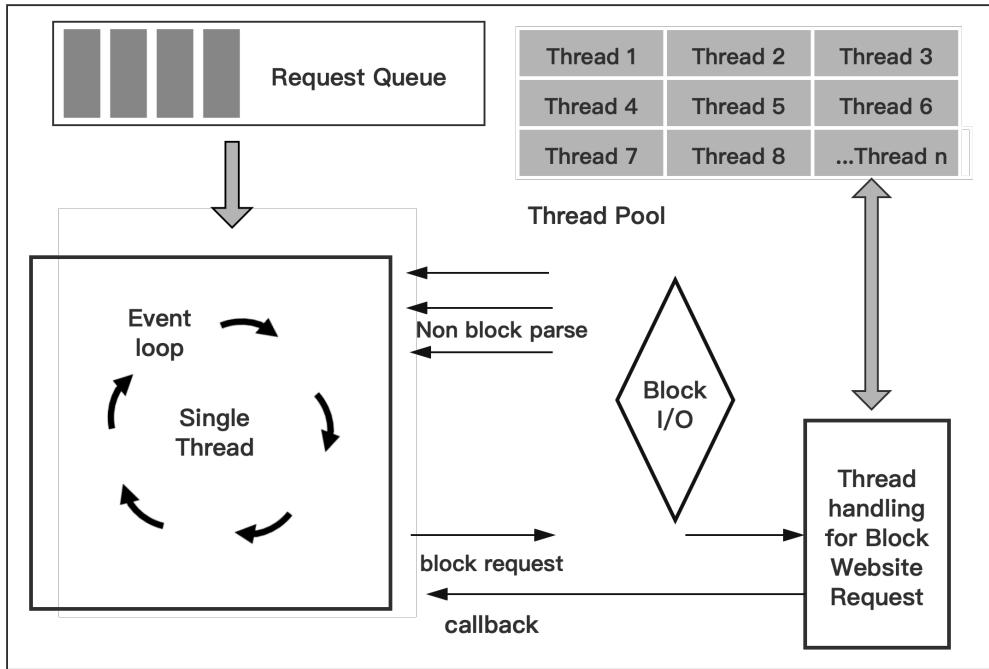


图 3-2: 单线程事件循环爬虫编程模型

该模型的优势能够处理许多并发的爬虫请求，使用了较少的线程，CPU 以及内存资源使用较少，同时避免了多线程死锁以及状态同步的问题。在单 CPU 环境下，该模式会优于多线程模型，请求处理速度比多线程模型快，因为没有线程间的上下文切换以及加锁的开销。但单线程模式无法利用 CPU 多核的优势，同时需要保证回调函数中不存在阻塞操作，不然会使得主线程处于阻塞的状态，无法处理其他请求。另外异常处理存在问题，异步事件处理过程中产生的 Exception 并不能被主线程感知并处理。

3.1.3 分布式爬虫模型

如图 ?? 所示，其中一台机器是 master 节点，并连接到多台工作节点上。master 节点会维护一个中心队列来存储待爬取到 URL 列表，master 会将 URL 根据一定的策略分发到各个 worker 节点；worker 节点进行网页爬取、链接提取以及数据解析等基本流程；然后将提取出来的链接以及解析的数据给传输到中心节点 master 上；master 节点对所爬取到数据进行汇总，然后将数据存储到数据库中^[30]。

分布式爬虫模型能够将任务的生成和数据的抓取两个过程分开，master 节点负责任务的分发和数据的收集工作，worker 节点负责数据的抓取，能够通过水平扩展 worker 节点的数量来提高爬虫的吞吐量，提高系统的可用性。但大部分的分布式爬虫都仅仅是维护一个全局 URL 队列，通过该队列来调度各个 worker 节点的爬取任务，并没有对爬虫各个组件分别管理，没有充分利用分布式系统的特性，只是简单的爬虫实例的堆叠，无法充分利用各节点的资源。

3.2 响应式爬虫编程模型

上述的各种爬虫编程模型都有其各自的应用场景：多线程爬虫适合于爬取网页较少的网站，这样能充分利用多线程的优势，但不至于造成太多额外的 CPU 开销；单线程事件驱动爬虫适合于爬取一些解析逻辑比较简单的网站，这样避免复杂的异常处理操作；分布式爬虫适合于爬取大型的网站，但爬虫的构建比较繁琐。不同编程模型都有其各自的特性，但都有一定的局限性。主要是三个方面：无法充分利用 CPU 资源；爬虫的编写不够简练；缺乏相应的异常处理机制。

对于这样的情况，Reactive Programming 有很好的解决方案。考虑爬虫爬取过程中，网页下载以及页面解析阶段存在大量的耗时操作，参考流式计算的思想，本文基于响应式编程，构造了一个响应式爬虫编程模型。

如图 3-3 所示，爬虫爬取呈现出流式的结构，先从 SeedUrls 开始，数据流经第一个组件，该组件通过 Url 封装成 Request 对象；然后将 Request 对象传递给下一个组件进行下载，转换成了 Page 对象；再将 Page 对象传递给下一级组件进行解析，解析之后得到 Item 数据，将 Item 传递给下一级组件进行解析。响应式爬虫编程模型中有两个相互隔离的工作线程池，Worker Thread Pool 0 和 Worker Thread Pool 1。其中这两个线程池可以动态扩充，但是是有界的。考虑网页下载是阻塞操作，而 Request 生成以及页面解析都是非阻塞操作，响应式爬虫编程模型将网页下载操作放在 Worker Thread Pool 1 进行异步执行，当数据下载完成后，通过消息驱动机制传递给下一级组件进行数据解析操作。该编程模型将阻塞任务调度到有界线程池中执行，该编程模型通过多个工作线程来处理非阻塞操作（页面解析、链接提取等等），与单线程事件循环爬虫编程模型相比，可以充分利用 CPU 多核的优势，提高爬虫爬取的效率。

同时和多线程编程模型相比，响应式爬虫编程模型意味着用更少的线程处

理更高的负载，因为工作线程池大小有界，往往是 CPU 核数的 10 倍，这也意味不会存在过多的线程切换开销。

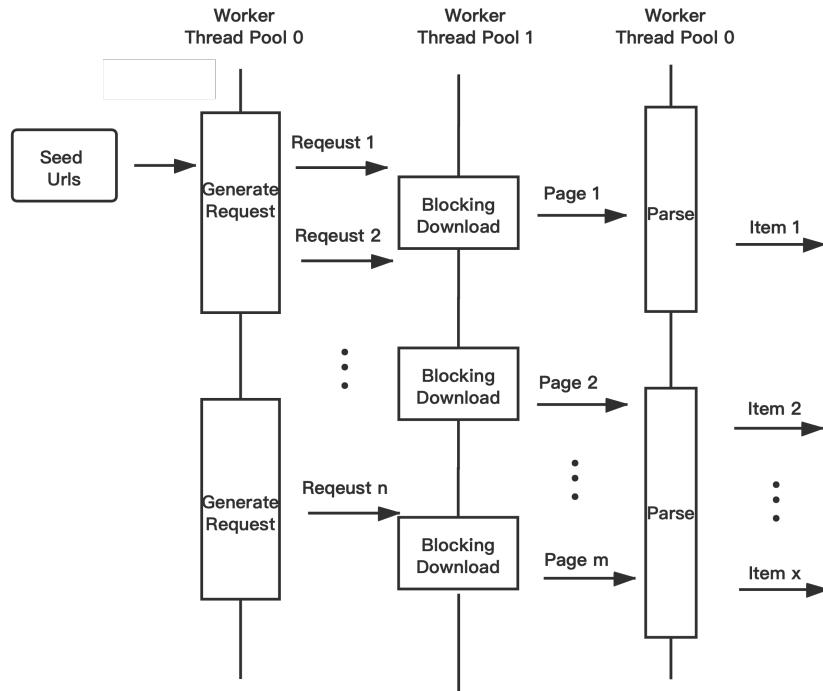


图 3-3: 响应式爬虫编程模型

响应式爬虫编程模型关注于组件算子操作和数据流生成，并对组件算子内部代码的实现提出约束。同时基于该编程模型，本文还提出了一种基于网站层次树结构的对象模型构造方法，并通过对象模型来配置数据流。

3.2.1 组件非阻塞约束

异步流式爬虫模型需要保证各个组件的操作都是非阻塞操作，不能阻塞线程，爬虫组件在处理完后，将处理过后的数据通过信号封装再传播到下游组件中。

爬虫组件中往往都是阻塞代码，阻塞代码会造成线程等待。若通过异步方式运行时，线程则可以执行其他任务，直到阻塞操作完成返回后，再进行相关的处理。爬虫在下载过程中存在着大量的阻塞网络 IO，这个过程如果是采用同步阻塞的方式必然会导致线程的等待，造成资源的浪费。这时，便需要对阻塞代码进行改写或者封装，将阻塞代码改写成异步非阻塞代码，把阻塞代码调度

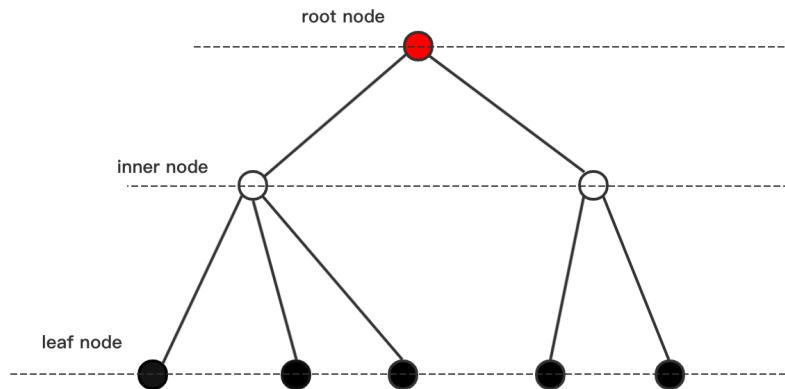


图 3-4: 层次树模型

到隔离的工作线程池中运行，不影响爬取的主流程。爬取流程中的其他阻塞代码也是通过相同的方式转换。

3.2.2 网站结构映射

网站结构是网站对其表现内容的组织形式，即网站的各个子页面是如何相互链接的。真正的网站往往是一个网状结构或者是图结构，网页是图中的一个节点，网页之间通过超链接作为边。传统爬虫遍历网站的图结构，忽略了网站的层次结构，只关注当前页面节点的结构信息以及后续连接的节点页面，往往容易陷入爬取死循环中，同时对于爬取目标页面的数据解析还需要遍历网页的结构，过程繁琐且可读性差。

为此，本文通过层次树模型对网站结构进行描述，并在响应式编程模型基础上提出了一种对象模型构建方法，解决了网页数据解析繁琐以及爬虫开发效率低的问题。

3.2.2.1 层次树模型

为了方便链接的遍历，本文对网站结构进行建模，将网站看作是层次树结构^[31]。如图 3-4 所示，根节点为 Home Page 页面，内部节点则是网站遍历爬取过程中的中间页面，叶子节点则是数据节点，携带着结构化的数据。

为了描述层次树结构，本文引入了两个类型 Node 和 Link，通过两个元组来表示 Node 类型和 Link 类型，即

$$\text{Node} = [\text{id} : \text{Oid}, \dots, a_i : v_i, \dots]$$
$$\text{Link} = [\text{from} : \text{Oid}, \text{to} : \text{Oid}, \dots, b_j : v_j, \dots]$$

在层次树模型中，页面与 Node 节点一一映射，Node 定义中的属性 *id* 表示唯一标识符，用来区分不同的 Node 节点，属性 *a_i* 表示页面中的数据；而页面与页面之间的超链接则表示为 Link 对象，Link 对象描述了 Node 节点间边的关系，Link 对象的 *from* 属性表示连接的上级页面，*to* 属性表示 Link 对象连接的下级页面，属性 *b_j* 表示链接自身的属性标签。其中的 *Oid* 则是 URL，网页通过 Node 节点抽象成了一个数据集合，Node 节点的除了 *id* 之外的各个属性均表示网页数据，Link 用来描述层次树模型中的边关系，其属性除了表示两端的 Node 节点外，还包括了链接自身的属性，如链接自身的 URL 以及链接的描述等等。当然，并不是所有的页面以及超链接都会在层次树中表示，层次模型其实是对网站的子集的描述。

3.2.2.2 对象模型定义

从三层的层次树模型考虑，第一层是网站首页，也就是 root node 节点；第二层则是从首页链接出去的内部节点页面，是 inner node 节点；第三层才是要爬取的目标页面，leaf node 节点。

如图 3-5 所示，为了能够描述层次树模型与对象模型之间的映射，我们通过自定义注解来映射数据对象与网页结构间的关系。如上文所示，层次树模型将网页节点分成了三类，我们需要能够描述这三类节点的层次结构。其中 StartUrls 用来描述网站首页，表示数据的爬取起点，也就是层次树结构的 root 节点；InnerUrl 则是表示中间页面，通过中间页面 URL 的匹配模式给爬虫提供链接跟进的方向；TargetUrl 则是表示目标页面，通过描述目标页面的 URL 的匹配模式来指定爬取的网页，其描述的是层次树中的叶子节点，也就是包含结构化数据的节点。这三个注解都是类注解，通过注解的定义，一个网站的层次树模型也就描述清楚了。然后是属性注解 Selector，该注解是为了描述对象属性与网页元素之间的映射关系，在网页上，数据被包含在 HTML 元素中，需要通过 Selector 选择器进行元素定位。最后通过映射类 Entity 与叶子 Node 对象建立映射关系，而层次树中的 Link 关系则是通过 StartUrls、InnerUrl 以及 TargetUrl 之间的层级依赖来表示。在下一章系统实现部分，本文讲述爬虫是如

根据这种映射关系进行链接跟进以及页面解析的。

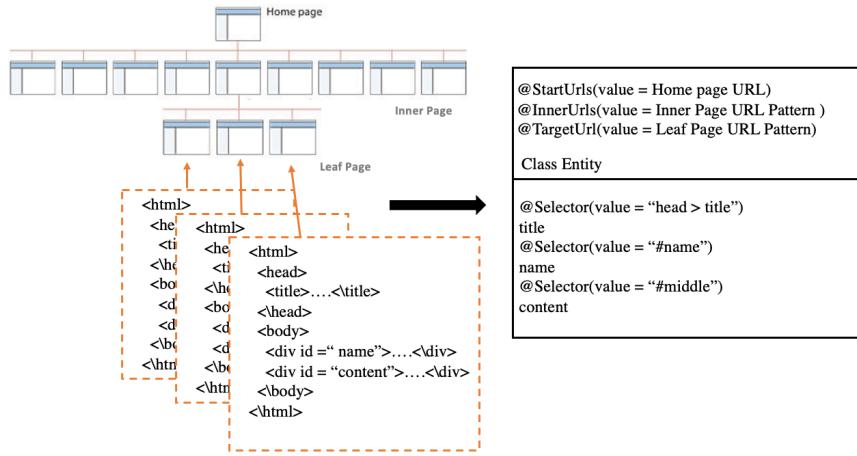


图 3-5: 层次树模型映射

相比于现有爬虫框架，层次树模型能够简化爬虫解析路径的编写，并且通过层次结构避免了爬虫死循环陷阱。开发者无需定义每个层次页面的处理逻辑，同时将爬虫的页面解析和爬虫的页面跟进逻辑分离开来。爬虫只需要定义对象模型，根据注解建立映射关系，描述网站层次结构，之后传统爬虫框架所需要的页面解析以及后续链接跟进都无需开发者描述，我们会对对象模型进行解析，自动化生成页面的解析逻辑以及待跟进链接的提取，以此来避免了爬虫繁琐的页面解析规则以及爬取链路的配置，提高爬虫开发的效率。

3.2.3 异步数据流构建

相比于前文提到的三种编程模型，Reactive 流式爬虫不需要再维护一个待爬取 URL 队列，而是通过数据流的形式表示待爬取的 URL 请求。为了获取待爬取的 Request 数据流，爬虫必须明确链接的遍历方式。

如图 3-6 所示，爬虫首先会根据上文定义的对象模型来构造一个 Request 数据流；之后设置对 Request 数据流的操作，将 Request 转化成 page，得到中间的 Page 数据流；然后对 Page 设置了解析操作，将解析出来的链接封装成 Request，再重新导入到 Request 数据流中，同时对页面解析数据解析，将页面映射成数据对象。最后生成一个对象流，以供后续的处理。

整个爬虫过程都被描述成了对数据流的处理，通过数据流之间的算子来描述数据处理过程，这个过程是异步非阻塞的，因为每个算子计算时是通过信号

来触发的。下游的算子接收到上游算子处理后的数据后，进行相应的处理，再将数据封装成信号传播到下游节点。

通过将各个算子进行组合，构建了异步数据流。数据在整个异步流程中向下传播，不断地处理转换，最终得到结果数据。

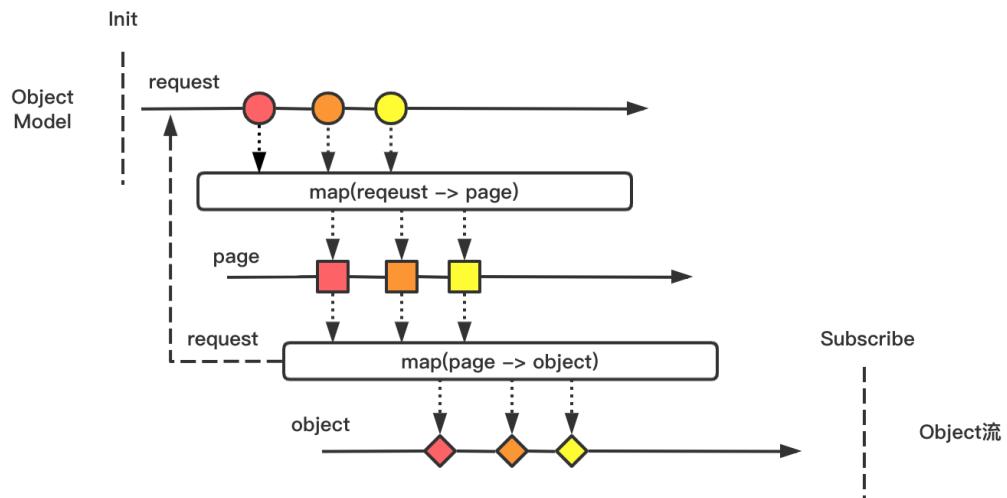


图 3-6: 异步数据流

3.2.3.1 异步数据流构建算法

爬虫本身就是对网站的遍历，爬虫通过遍历整个层次树结构来构建异步数据流。本文通过基于剪枝的广度优先搜索的方法对树结构进行遍历，获取需要爬取的链接，将链接导入到 Request 数据流中。数据爬取遍历过程如算法 1 所示，用户从对象模型出发，会对对象模型解析，生成 Request 数据流。当数据流中有数据时，会通知下游进行处理。将通过数据流中的 URL 去请求网页，对网页进行异步下载，之后将网页进行解析，得到当前网页的所有链接。这里采用的是广度优先搜索方法对链接进行跟进，同时为了防止搜索空间过大，本文通过链接正则匹配的模式来跟进链接，缩小搜索空间。

3.2.4 反反爬策略设置

大多数网站都设置相关的反爬策略，本文仅从研究的角度来实现响应式爬虫模型的反反爬策略的配置。由于响应式爬虫模型是对数据流的异步操作，爬虫在设置反反爬策略时，是通过对数据流的处理来实现。响应式爬虫模型是基

Algorithm 1 异步数据流构建算法

Require: m 代表定义的对象模型, $stream$ 表示 Request 数据流, $targetUrl$ 表示目标页面匹配模式, $pattern$ 代表目标链接匹配模式,

```

1:  $visited \leftarrow \{\}$ 
2:  $s \leftarrow \text{parse}(m)$  解析对象模型获得初始爬取起点
3:  $\text{EmitNext}(stream, s)$  初始化数据流
4:  $result \leftarrow \{\}$ 
5: while  $stream \neq \emptyset$  do
6:    $url = \text{OnNext}(stream)$  数据流中的数据到达
7:    $visited.\text{Append}(url)$ 
8:    $page = \text{AsyncDownload}(url)$  页面下载
9:    $links = \text{AsyncLinkExtract}(page)$  页面解析
10:  if  $url \text{ Match } targetUrl$  then
11:     $r = \text{Parse}(page)$ 
12:     $result.\text{Append}(r)$ 
13:  end if
14:  for  $link \in links$  do
15:    if  $link \notin visited$  and  $link \text{ Match } pattern$  then
16:       $\text{EmitNext}(stream, link)$ 
17:       $visited.\text{Append}(link)$ 
18:    end if
19:  end for
20: end while
```

于响应式编程来实现的流式爬虫，提供了许多算子来对数据流进行处理，我们可以通过这些算子来对爬虫的 Request 数据流、Page 数据流以及最终被消费的对象数据流进行处理。

对于网站 IP 限制的问题，爬虫通过调整 IP 爬取的频率和次数来应对。响应式爬虫通过对数据流中数据加上延迟的限定来处理。如图 3-7 所示，通过 `delayElements` 算子，将每个 Request 请求都设定相应的时间间隔，这样减少请求的频次，减缓了被爬取网站的负载。

针对于网站的 UA 限定、IP 限制，Cookies 限制以及验证码等问题，响应式爬虫模型通过构建 User Agent 数据流、代理 IP 数据流和 Cookies 数据流来对 Request 数据流进行操作。如图 3-8 所示：通过与各个配置流的组合，每个 Request 配置相应的 User Agent，都有各自的代理 IP 以及 Cookies 设置。另外在 User Agent 流、代理 IP 流以及 Cookies 流生成时，同样可以配置相应的策略，例如设置 User Agent、IP 以及 Cookies 的更换频次等。

对于反爬策略中次数的限定可以通过 `take` 算子来解决，同时 `take` 算子

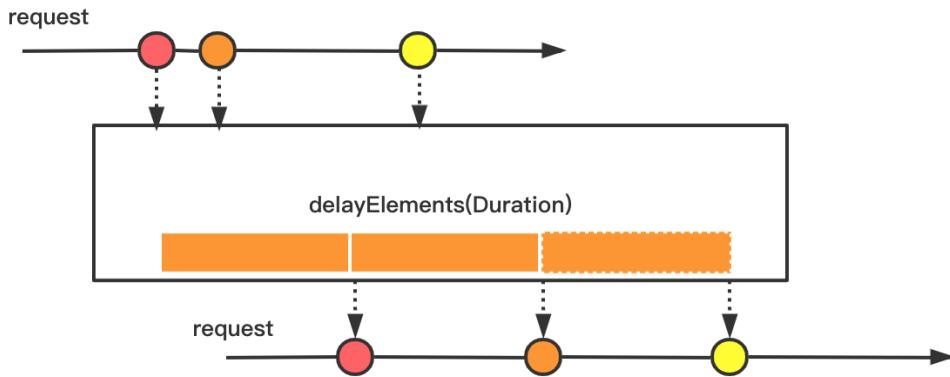


图 3-7: 爬虫频率配置

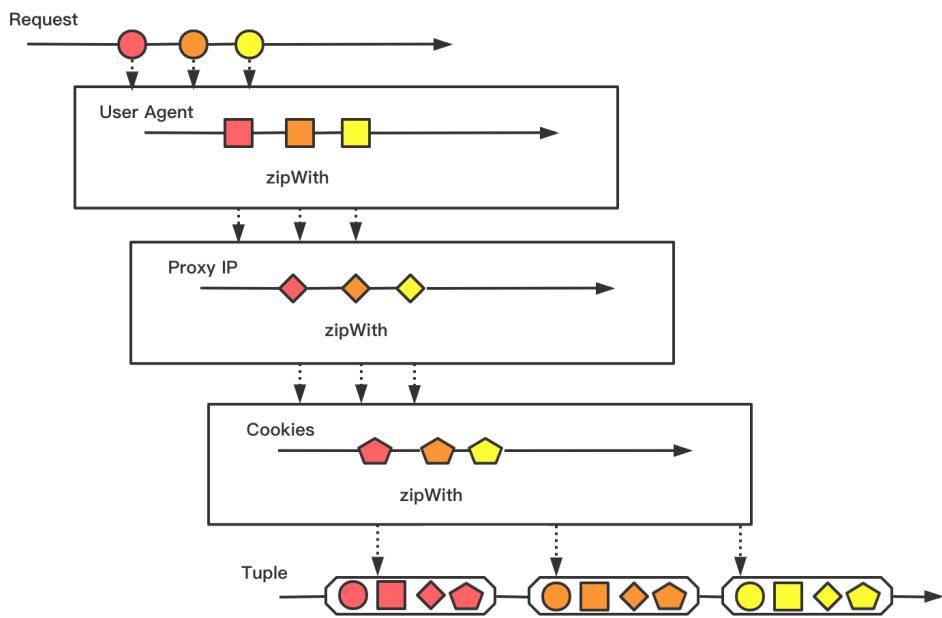


图 3-8: Request 数据流配置

提供了相应的背压机制，通过 `limitrate` 来对上游的组件进行流量控制。如图 3-9所示，能够根据爬取数量来设置爬虫的停止的条件，当爬取到一定数量的数据后，下游组件会将 `cancel` 信号传播到上游组件，最后停止爬虫的爬取，而 `limitrate` 则向上游反馈下游组件的消费能力，防止生产者生产速度远大于消费者消费速度，引起任务的大量堆积。

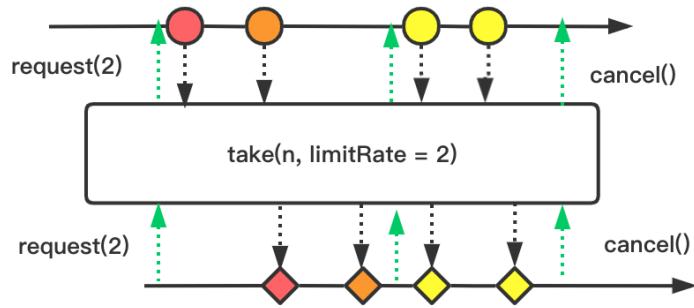


图 3-9: 爬取次数限制和背压配置

3.3 本章小结

本章从网站结构出发，概述了三种爬虫编程模型，并分析各个模型的优劣，提出了一种响应式爬虫编程模型，从对数据流处理的角度，介绍了爬虫爬取模型的运用场景。

下一章介绍响应式爬虫模型的具体实现。

第四章 框架实现

本章介绍响应式爬虫的具体实现，首先介绍了各个组件的基本实现接口以及各个爬虫组件的实现。然后介绍了系统各个模块的实现以及响应式爬虫开发过程。

4.1 Project Reactor 介绍

本文以目前主流的响应式编程库为基础，提出了 Spidereact 响应式爬虫技术框架。Project Reactor 是第四代响应式编程库^[32]，建立在 Reactive Streams 规范基础上，实现了 Reactive Programming 编程范式。它能让你快速地构建一个遵循响应式编程范式的响应式系统，提供完全无阻塞的异步数据流操作。

Project Reactor 提供了两个基本的数据类型：Mono 和 Flux。Mono 表示 0 个或 1 个数据项（数据项有可能是 error 信号），而 Flux 则表示 0 个或多个数据项，或是一个 error。Mono 和 Flux 就是数据源，它们都实现了 Publisher 接口，可以通过订阅这些数据源并声明操作流程，对这些将要到达的数据进行处理^[33]。Spring 也对 Reactor 进行了集成，同时 Project Reactor 还有很多的扩展，提供了 R2DBC，一种响应式关系型数据库连接，将关系性数据库 JDBC 中的阻塞调用转变成了异步非阻塞模式，通过整合 Spring, Reactor, R2DBC 等等 Reactive 组件，我们就能够构建一个响应式系统。

本文将通过 Project Reactor 与爬虫框架进行整合，提出一种响应式的爬虫编程模型，来提高在有大量的网络 IO 场景下爬虫系统的资源利用率。本文将原有的爬虫框架中的各个组件均通过异步非阻塞的形式组合，构建了一种新的响应式爬虫流程。

4.2 爬虫框架模块

Spidereact 是基于 Project Reactor 的 Java 响应式爬虫框架，通过异步数据流，解决传统爬虫框架的一个请求对应一个线程处理的模式存在的问题。通

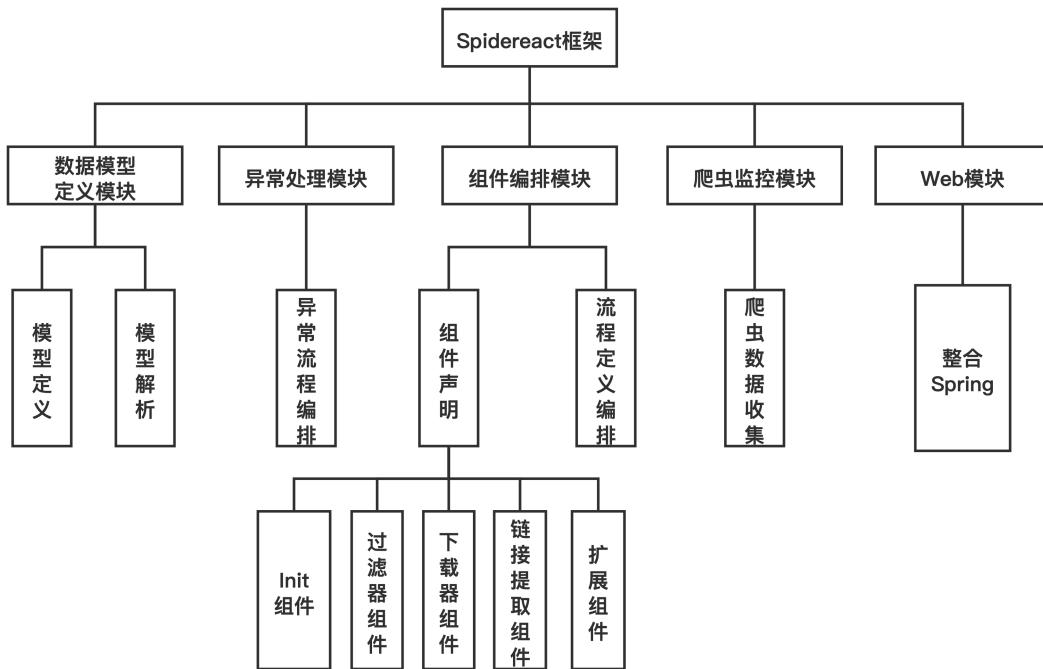


图 4-1: Spidereact 功能模块图

过 Spidereact 开发一个爬虫，首先要分析网页数据结构，构建数据对象模型与网页数据之间的映射关系，同时通过 Annotation 来表明待爬取的 URL；对爬虫的数据初始化组件、页面下载组件以及网页数据解析组件等模块进行组件化声明，通过对组件的组合构造爬虫处理流程图，执行爬虫获得响应式数据流；后续可以对得到的数据流进行过滤、筛选和数据持久化操作。如图 4-1 所示，Spidereact 的各个模块功能介绍如下：

- **数据模型定义模块**：通过自定义注解来定义数据对象与待爬取网页对象直接的映射关系，通过这种映射关系模型来对网页结构中的元素进行解析，同时生成相应的数据对象。
- **组件编排模块**：对爬虫的各个组件进行编排组装，最后构建一个爬虫组件流。
- **异常处理模块**：定义爬虫阶段的各个异常情况，提供相应的异常处理接口，方便二次扩展。
- **监控模块**：监控爬虫的各个阶段情况，统计爬取成功次数、失败次数等。
- **Web 模块**：整合 Spring，使得 Spidereact 能够集成到 Spring 框架中。

表 4-1: 主要使用的注解

注解	描述
StartUrls	初始 URL 列表
InnerUrl	中间页面 URL，用来表明链接跟进的方向
TargetUrl	目标 URL，表明最终要解析的页面
Selector	CSS 选择器，匹配网页中的元素

4.2.1 数据模型定义模块

数据模型定义模块会根据需要爬取的网页的网页结构来定义对应的数据对象，会通过几个注解来定义爬虫的爬取方向以及定义待解析的数据。如表 4-1 所示，Spidereact 通过 StartUrls、InnerUrl、TargetUrl 以及 Selector 来定义数据对象，首先根据 StartUrls 来构建初始 Request 数据流，Request 数据流用来描述对 URL 的请求信息。TargetUrl 则描述在页面解析过程中需要提取的网页链接，作为后续爬取的 URL，重新导入到 Request 数据流中。前文也提到了，Selector 注解是用来解析网页数据，网页上的数据都是以文本形式读取，而数据对象中往往有各种各样的类型，这是需要类型间的映射。网页上除了文本以外的图像、音频以及视频数据，需要开发者自定义下载器中的下载逻辑进行处理。目前支持以下几种类型：

1. String 和几种基本数据类型：float, double, int, long, boolean。
2. java.util.Date 类型以及 org.joda.time.DateTime。
3. 属性都由 Selector 注解的 POJO 类。
4. List 类型，其中 List 存储的类型必须是 Selector 支持的类型。

定义的数据模型会在爬虫构建的时候进行解析，其中 StartUrls 中的 value 属性会被传入 Init 组件生成初始数据流；而 InnerUrl 中定义的中间页面 URL 模版则会被导入到链接提取组件中，来对网页上的链接进行过滤，然后提取出爬虫想要爬取的链接，再倒入到数据流中；TargetUrl 中定义的目标 URL 模版用来识别目标页面，然后通过映射组件将网页映射成数据对象。

4.2.2 组件编排模块

组件编排模块负责对各个爬虫组件的流式构建，开发者只需要关注各个组件具体的实现逻辑，组件处理链路编排则由组件编排模块来实现，开发者能很

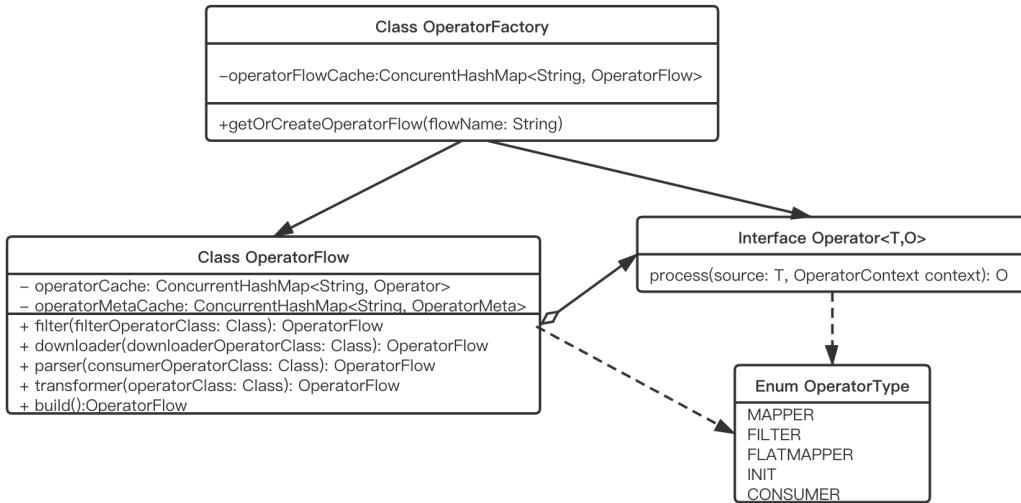


图 4-2: 组件编排模块 UML 类图

方便地构造一个从爬取到解析到数据处理分析的逻辑链路。

组件编排模块的设计参考了工厂设计模式，本文结合图 4-2 来进一步介绍。OperatorFactory 负责管理和创建 OperatorFlow，它会根据传入的 Flow 名字来加载爬虫流程实例，OperatorFlow 用来表示爬取流程，管理爬取过程中的各个组件，各流程中组件均可相互复用。组件编排模块主要分为了两个部分，一部分是基本组件声明，还有一部分则是流程编排。

4.2.2.1 爬虫基本组件

在 Spidereact 的设计中，组件是位于流程之中，具有一定的先后依赖顺序，其中基本组件如初始化组件、下载器组件以及解析器组件的顺序是固定的。各个组件无法独立执行，必须构建完整的爬取流程才能执行，组件可被复用于不同流程之中。

Spidereact 爬虫框架是基于 Project Reactor 构建的，每个爬虫组件都会被 Spidereact 给隐式加载成异步非阻塞模式，开发者只需要注重各个组件的内部逻辑，无需感知流程编排的内部细节。开发者通过声明组件对应的类，而不需要实例化组件，Spidereact 会根据声明的组件类在编排阶段对组件进行实例化，然后编排流程图。

Spidereact 会维护一个内部的数据流，各个组件分别对流数据上的数据进行处理，数据在不同组件上传递、处理、转换。在 Spidereact 中，数据一定会经

表 4-2: 两种基本数据类型

类型	描述
Request	请求的封装
Page	对下载的页面的封装

历这样一个转变过程，从 Request 到 Page 再到 Object。如表 4-2 所示，Request 是对一个爬取请求的封装，包括了请求的 URL，请求的请求头，Cookies 等等相关信息，Page 对象则是对于下载页面的封装。

爬虫首先通过 Init 组件来初始化构造的 Request 数据流；初始的 Request 流会经过过滤组件，通过过滤规则来去除某些不需要再爬取的网页；然后经过下载器组件的处理，将 Request 给转换成了 Page 页面，然后传入下一级组件；链接提取组件在 Page 数据流上进行操作，提取出 Page 页面上所需要的链接，链接再重新倒入到 Request 数据流中，该组件操作不会对 Page 数据流中的数据产生任何影响，然后将 Page 数据流传到对象映射组件中；对象映射组件会对传过来的 Page 数据流进行映射，将 Page 中的 HTML 与已经定义好的对象模型进行映射，构建出对象流；同时，Spidereact 支持自定义扩展组件，实现相关的处理逻辑，对对象流进行处理，如数据的持久化存储，数据的展示等等。

Spidereact 对爬虫组件进行了抽象，如代码 4.1 所示，Spidereact 的所有组件均实现这个接口，这个接口中就只有一个 process 方法，组件需要实现该方法来对上游组件传入的数据进行处理。泛型 T 表示上游传送的数据类型，泛型 O 表示当前组件输出的数据类型，对于开发者需要扩展爬虫组件时，只需要实现该接口，描述组件的处理逻辑。

```

1  public interface Operator<T, O> {
2      // 从上游组件中获取元素进行处理，得到的结果传入下游组件
3      O process(T source, OperatorContext context);
4
5  }

```

Listing 4.1: Operator 接口

4.2.2.2 Init 组件

Init 组件是初始化组件，它会根据定义好的数据模型 Class 类进行解析，生成相应的种子 Request 数据流。该数据流有一个线程安全 FluxSink 接口，

供其他线程异步导入 Request 数据。同时 Init 组件会将 FluxSink 给传入到 OperatorContext 对象中，该对象能够被各个组件获取，这样每个组件都能异步地传入 Request 请求进行处理。

Init 组件生成无界 Request 数据流作为爬取的起点，Spidereact 提供了两种不同的 Init 组件来生成 Request 数据流。第一种 Init 组件不会对下游组件的状态进行响应，实际上是一种 Push 模式，Init 组件会将数据之间 Push 到下游的组件去进行处理，这种情况下将会失去背压机制^①，如果下游组件无法全部处理来自于 Init 组件的数据（信号）时，已经发出的数据将会按照某种策略删除或者时缓冲，默认情况下时缓冲，直到下游组件需要消费更多的数据。第二种 Init 组件则是采取一种 Pull 模式，Init 组件生产 Request 数据会按照下游组件的要求来生成，这种情况下存在背压机制，下游组件根据自身处理能力的情况来要求 Init 组件生成数据。

4.2.2.3 过滤组件

过滤组件主要是对 Request 数据流本身数据的过滤，在爬虫爬取的过程中，需要对请求数据进行相应的筛选。最基本的是去重的处理，去重的方法需要综合查重效率、内存使用量等多方面考虑。

在 Spidereact 中，考虑到查重的空间效率和时间效率，内置了布隆过滤器对重复 URL 进行过滤，同时由于不需要有删除的操作，避免了布隆过滤器删除困难的问题，另外针对于布隆过滤器自身存在的误报率，可能会使得一些本没有访问过的 URL 被误报为已经被访问过了，导致某些 Request 请求被直接跳过，但是对于爬虫数据爬取来说，一定程度的误报率是可以接受的。

对数据的筛选除了去重，还可以在过滤器组件中进行一些特定链接的删除，例如不在该网站域名下的链接，同时还可以设定相关规则匹配一些不相关的链接，通过过滤这些无关链接来提高整体爬取的效率。对于不相关链接的匹配，可以采取正则表达式的形式来配置过滤规则。这些需要用户自定义规则实现过滤组件。

如代码 4.2 所示，Spidereact 的自定义过滤组件需要在 process 方法中实现，通过对 source 对象的处理，返回值 false 则表示过滤此元素，返回值 true 则保留。

^①<https://jstobigdata.com/java/backpressure-in-project-reactor/>

```
1  public interface FilterOperator<T> extends Operator<T, Boolean> {
2      // 对上游数据source进行过滤
3      @Override
4      Boolean process(T source, OperatorContext context);
5  }
```

Listing 4.2: 过滤组件接口

4.2.2.4 下载器组件

下载器是将 Request 转换成页面的核心组件，也是整个爬虫流程中开销最大的组件，由于服务器本身处理请求的时间以及网络环境中的延迟，Request 请求响应是一个阻塞的过程，系统需要有大量的网络 IO。如上文爬虫编程模型中所描述的，通常的爬虫框架会对整个爬取链路都放到同一个线程中进行处理，用同一个线程来处理下载、解析、过滤等同步操作。Spidereact 将下载器组件通过一个与其他组件隔离的工作线程池进行运行调度，使得下载器组件能够以异步非阻塞的方式运行在一个有边界的弹性线程池中。下载器组件有两种默认实现，一种是通过 HttpClient 库^①，模拟网络请求的发送与响应的接收，另一种方式是通过模拟浏览器来处理网页的下载以及渲染问题。

通过 HttpClient 库进行下载相比于浏览器模拟的方式，速度较快，因为不需要对页面进行渲染，但缺点是有很多数据无法获取，只能获取到网页的源代码。为了处理网页 js 代码动态加载，下载器组件还整合了 Selenium 来对网页进行渲染，得到渲染后的数据。Selenium 是 Web 浏览器自动化工具，被设计用于自动化 Web 应用测试，当然它也被用在许多其他的应用程序中，它通过 Selenium Webdriver 组件能够模拟用户操作浏览器的场景，同时还支持市场上大部分浏览器，如 Google Chrome、IE、Firefox、Safari 等等。通过 Selenium，下载器能够实现对 js 渲染页面内容的快速获取，简化数据解析过程。传统 js 页面数据获取需要通过分析网页 js 代码来获得数据接口，现在通过 Selenium 直接获取渲染过后的页面。同时，Selenium 可以实现与网站相关的交互操作，某些情况下爬虫需要实现登录验证的过程，以普通的方式访问网站比较困难，Selenium 提供相关接口来实现与交互操作，简化了模拟登录的流程。与传统的方法相比，Selenium 因为能够模拟浏览器操作，基本上能够获得网站上的所有

^①<https://hc.apache.org/httpcomponents-client-5.0.x/>

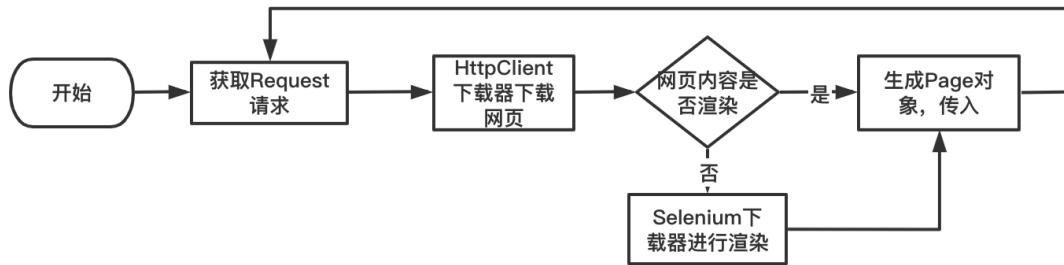


图 4-3: 下载器下载流程

数据，但是需要下载安装相关浏览器的驱动，另外网站网页的渲染开销比较耗时，会一定程度上拖慢爬虫的爬取效率。

针对不同的场景，根据不同下载器的特性，爬虫会采用不同的下载器进行网页下载。当然也可以通过结合两者的优势，通过条件判断来选择不同的下载器。如流程图 4-3 所示，首先从上游组件节点中获取 Request 请求，通过传统的 HttpClient 库来请求网站网页数据，对于网站响应网页进行判断，查看爬取内容是否渲染成功，不然将通过 Selenium 下载器重新渲染页面，然后获得最终的页面数据，如此循环往复执行这样一个下载的流程。

另外下载器组件可以通过 URL 匹配的模式来甄别是否是动态网页，对于一个网站来说，其内部网站结构和网页布局一段时间内是稳定的。动态网页的 URL 具有一定的规律性，Spidereact 可以通过正则表达式的方式来识别动态网页，将它通过 Selenium 下载器去解析，得到渲染页面，而被识别为静态页面的网页则交给 HttpClient 下载器进行下载。

同样地，下载器组件也提供相应的扩展接口，能够自定义下载行为。例如：可以实现通过网络请求的形式将页面下载渲染的任务交给远程浏览器进行渲染，将浏览器和爬虫程序给分离开来，渲染行为操作直接交给远程浏览器进行，减轻了爬虫过程浏览器渲染的开销，同时还能动态地水平扩展远程浏览器的负载能力。

4.2.2.5 链接提取组件

链接提取是爬虫流程的一个重要过程，其目的是从网页中抽取那些要被跟进的链接。如流程图 4-4 所示，获取 TargetUrl 注解，其中的属性定义了匹配链接的正则表达式，链接提取组件会将网页上的链接全部提取处理，并进行正则

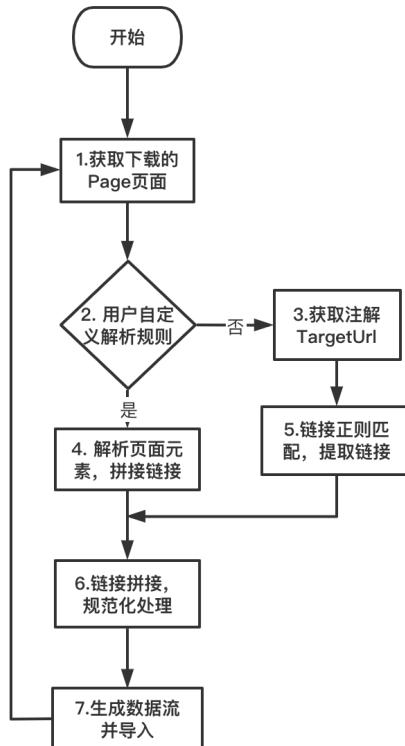


图 4-4: 链接提取流程

匹配，然后将提取处理的链接封装成 Request 数据流，异步倒入到 Init 组件生成的数据流中。上述流程是链接提取组件的默认实现，当正则匹配无法完全地解析出目标链接时，例如这样一个场景：后续链接的生成并不是靠网页上的链接进行相应的拼接得到的，而是根据网页上某些文本以及网站链接的规律来生成，框架允许扩展链接提取组件，通过重写链接提取方法，实现用户自定义链接提取逻辑。

4.2.2.6 对象映射组件

对象映射组件是将获取到的网络页面给映射到具体的对象中，将网页上对应的数据装填到对象实例中。通过对对象映射组件，将网页上的数据转换成了结构化或者半结构化的数据，方便后续组件的处理。

对象映射的过程实质上是页面内容解析的过程，通过上文提到的 Selector 注解来提供映射关系。如图 4-5 所示，具体的流程可以描述为：

1. 从上游组件中获取推送的 Page 对象，Page 中包括了整个网页的 HTML，对

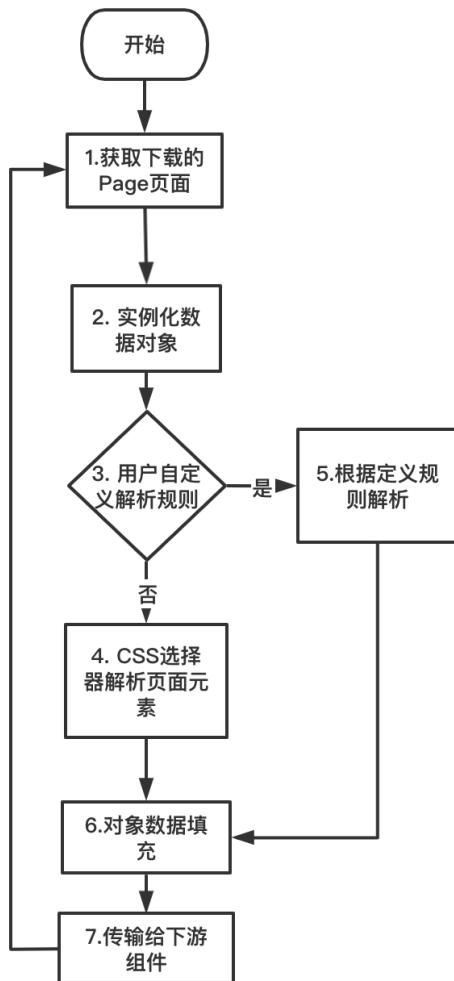


图 4-5: 对象映射流程

象映射组件主要是对 HTML 进行相应的解析，获取其中的数据。

2. 实例化数据对象，首先调用数据对象的默认构造函数，对对象进行实例化。
3. 如果没有自定义解析规则，那么根据 Selector 注解属性，提取其中定义的 CSS 选择器来定位页面元素，提取出元素内容。
4. 如果有自定义解析规则，则根据自定义规则对 HTML 页面进行解析匹配。
5. 将提取的内容进行类型转换，匹配数据对象中定义的属性类型，然后装填对象实例。
6. 将处理好的对象传输到下游组件进行处理。然后转 1 开始处理下一个 Page 对象。

4.2.2.7 扩展组件

扩展组件是基于 Operator 接口进行扩展，实现组件逻辑。扩展组件可以存在于整个爬虫流程 Init 组件之后的每一个阶段，可以存在于下载器组件和链接提取组件之间，也可以位于过滤组件之后。可以根据编排需要，将扩展组件放置在任意一个位置。

Spidereact 通过扩展组件可以实现内容去重，前文提到的过滤组件只是在链接的基础上的去重，如果更进一步处理，可以在映射组件之后加入通过扩展组件实现的内容去重组件。同样地，Spidereact 可以实现持久化存储组件，将结构化的数据存储到数据库中。

4.2.3 流程的编排

流程的编排主要分成三个部分，第一部分是初始化数据流部分 (Source)，第二部分是对数据的转换操作 (Transformation)，最后是将转换的结果输出到一个目的地 (Sink)。初始化数据流部分是通过 Init 组件来实现，数据的转换操作则是由前文提到爬虫的一系列基本组件实现，最后的结果输出会通过 Subscriber 订阅者来处理，前两部分构成了一个数据流的发布者，最后通过发布者订阅者模式来对数据的最终处理。

通过编排会形成如图 4-6 所示的逻辑视图。该图展示了一个普通的爬虫程序中，数据在不同的组件间流动的情况。每个圆圈表示组件，圆圈间的箭头表示数据流，数据流经过不同的组件的处理，最后生成结果到 Sink。其中 Spidereact 通过 Init()、Filter()、Downloader()、Parser()、Transformer() 等方法将组件编排到爬取流程 OperatorFlow 中。各个组件中的处理操作有可能是阻塞的或是非阻塞的，为了能够构建统一的异步非阻塞处理流，Spidereact 编排阶段会对阻塞的组件给分配隔离的异步工作线程池进行处理，将阻塞代码转化成了非阻塞模式运行，使得爬虫流程中每一部分都是非阻塞处理。

4.2.4 异常处理模块

异常处理模块用以管理 Spidereact 爬取期间上整个爬虫链路上的异常情况。异常处理模块会捕获所有在各个组件执行期间引发的异常，并对异常加以处理，使得异常信号不再向下传播。传统多线程编程模型无法捕获另一个线程中抛出的异常，只能在每个线程内部通过 try-catch 语句分别处理，或是使用异

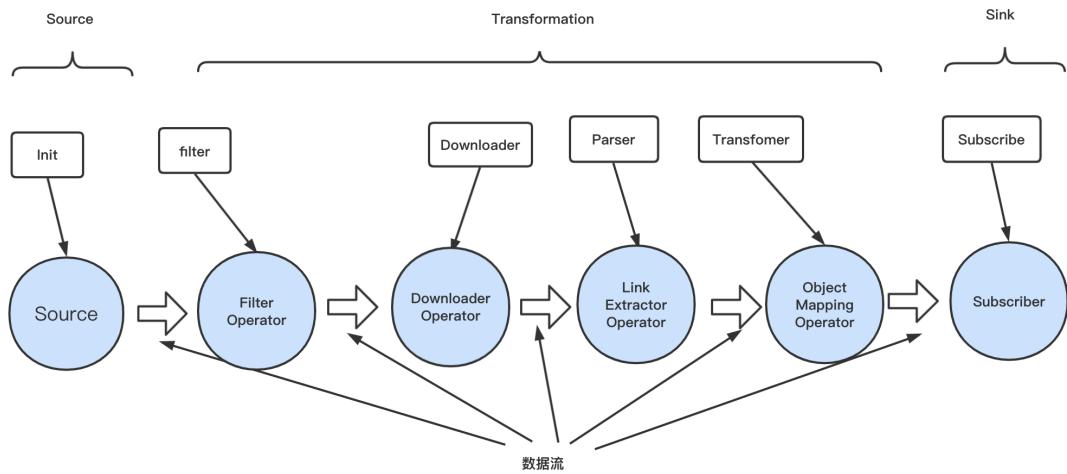


图 4-6: 流程编排逻辑视图

表 4-3: 自定义异常类型

异常类型	描述
PageDownloadException	页面下载时出现的 page 页面无法下载异常
LinkExtractException	页面链接无法提取异常
InitException	初始化 Request 数据流异常
HtmlToPojoException	数据映射时出现的数据不匹配异常

常处理器 `UncaughtExceptionHandler` 来进行异常处理。但这两种方法都不够优雅，Spidereact 通过 `Project Reactor` 库对受检异常进行封装并重新传播，使得异常处理程序能够在下游及时捕捉，并加以处理。

为了管理组件异常，Spidereact 自定义了一组异常类型来描述不同组件的异常情况，同时扩展了异常处理接口，能够让用户自定义异常处理逻辑。如表 4-3 所示，Spidereact 通过这些 `Exception` 类型封装了各个组件处理数据流时遇到的异常，不必在组件的处理逻辑中对异常进行捕获处理，而是将异常重新抛出，让下游的异常处理程序对异常进行处理。

如异常处理流程图 4-7 所示，异常处理通过 `hook` 模式来监听各个组件的运行状况，当某个组件有异常事件抛出时，`hook` 会截获该异常并加以处理。异常处理过程类似于事件监听机制，整个爬虫是基于异步消息驱动，会将异常作为消息信号向下游传播，若下游未定义异常处理，那么整个爬取流程将会终止。

当爬取流程正常运行时，数据流从 `source` 出发，途径 `filter`、`downloader`、

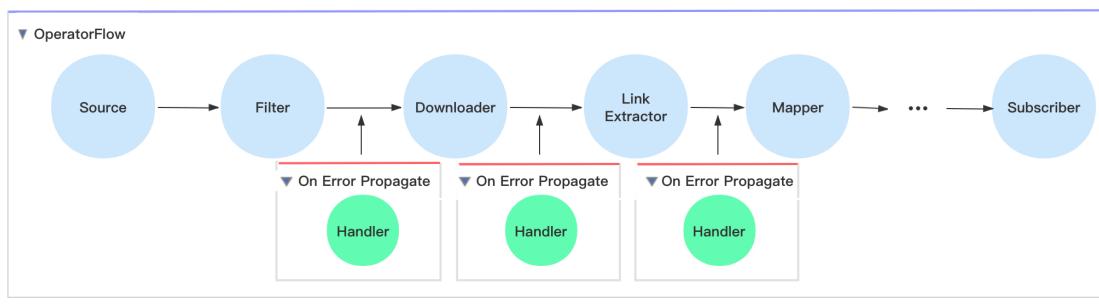


图 4-7: 异常处理流程图

linkExtractor 等等，最后被 subscriber 消费。但当某个组件处理数据时出现异常情况，该组件会将异常抛出，异常也被当作一个信号，和正常的数据信号一样会向下游传播。该异常会被自定义的 handler 给截获，触发异常处理程序进行处理，整个异常处理流程并不会影响到爬取的正常处理流程。

4.2.5 监控模块

监控模块用来通知和收集爬虫数据，通常是通过键值对的形式来统计数据，其中的值往往也是计数器。Spidereact 实现了 StatsCollector 来对数据进行收集，并且提供了相关 API 方便访问。

StatsCollector 收集器会收集以下信息：

1. 总共爬取的网页数量：所有的请求总数。
2. 当前正在爬取的网页数量：目前下载器中正在处理的网页数量。
3. 成功爬取的网页数量：成功解析到数据的页面数量。
4. 爬取失败次数：在爬取的任一节点失败的次数总量。
5. 爬取速率：爬取成功页面数量除以时间。
6. 请求等待数量：目前正在等待下载器处理的请求数量。
7. 各组件处理时间：数据流经每个组件需要的处理时间。
8. 各组件处理数量：各个组件各自处理的数据总量。
9. 各组件吞吐量：各个组件单位时间内处理的数据数量。
10. 端到端平均处理时间：从 source 组件出发到数据最后被消费所平均花费的总时间。

通过统计这些信息来实时监控一个爬取流程，在爬取时可以实时获取，有助于检测爬虫状态。

开发者可以通过继承 StatsCollector 来自定义爬虫数据收集逻辑，通过定义相关指标，来实时观察爬虫过程中指标的变化。

4.2.6 Web 模块

Web 模块主要是对 Spring 对整合，将 Spidereact 中的实例加载为 Spring Bean，从而使得开发者能够结合 Spring 框架中使用 Spidereact 获取网络数据。

为了能够被 Spring 加载成 Bean，Spidereact 定义了 2 个基础注解：

1. Operator: Operator 注解用来标识会被 Spring 加载对组件，同时注解属性中通过 index 还定义了组件在爬取流程图中的相对位置，方便 OperatorFlow 的构建。
2. OperatorScan: OperatorScan 注解用来描述组件所在的包，Spring 容器在加载 Bean 的过程中，会通过 OperatorScan 对包目录下的所有 Operator 注解标识的组件进行加载。

如图 4-8 所示，InitializingBean 接口是由 Spring 框架提供的，可自定义初始化 Bean。当 Spring 容器启动后，会对 SpiderreactAutoConfiguration 进行加载，同时将 OperatorFlowAwarePostProcessor 加载，之后 OperatorScanRegistrar 会扫描所有的 Operator 组件，构造 OperatorFlow，并将 OperatorFlow 传入到 FlowController 中。应用通过 FlowController 来获取爬虫爬取的结果数据流加以处理。

4.3 Spidereact 爬虫开发

下面通过介绍 Spidereact 爬虫开发过程，包括爬虫数据对象定义和爬取流程编排，后面还有关于爬虫反爬策略的设置。

4.3.1 数据对象定义

Spidereact 爬虫开发需要根据爬虫的经验知识，以及对网页数据结构和网站链接结构的了解，来对爬取的数据模型进行建模。爬虫数据模型既需要描述对象属性，还需要描述数据模型与网络结构的映射关系，将网页结构数据与数据模型进行关联。

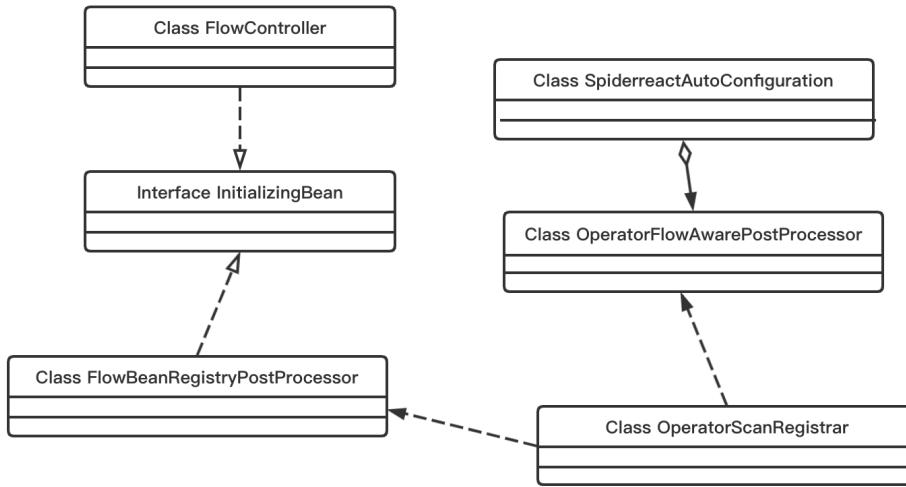


图 4-8: Web 模块 UML 类图

如代码 4.3 所示，通过 StartUrls 描述爬取起点，InnerUrl 以及 TargetUrl 描述爬取链接的跟进策略，通过两者的描述来构建网页爬取遍历的层次树状结构。然后通过 Selector 中的 CSS 选择器进行页面元素和对象数据的映射。

```

1  @StartUrls(values = "http://www.example.com")
2  @InnerUrl(value = "")
3  @TargetUrl(value = "")
4  public class Doc {
5      @Selector(value = "")
6      String title;
7
8      @Selector(value = "")
9      String text;
10
11     @Selector(value = "")
12     String comments;
13
14     public String toString(){
15         return "{ " + title + ", " + text + ", " + comments + " }";
16     }
17
18 }
  
```

Listing 4.3: 数据模型定义

4.3.2 组件定义

首先是 Init 组件，如代码 4.4 所示，通过继承 SourceOperator 来定义了初始化组件。由于初始化组件已经有了默认实现，开发者只需要指定泛型类型即可。

```
1 public class Init extends SourceOperator<Doc> {  
2 }
```

Listing 4.4: Init 组件定义

过滤组件默认实现是 DuplicateFilter，同时爬虫开发者还可以通过重写组件来自定义组件逻辑，如代码 4.5 所示，通过 Filter 组件重写了 Request 重复过滤组件。

```
1 public class Filter extends DuplicateFilter<Request> {  
2  
3     @Override  
4     public Boolean process(Request source, OperatorContext context) {  
5         // some code  
6         XXXXX;  
7     }  
8 }
```

Listing 4.5: Filter 组件定义

爬虫的下载器组件也有默认实现类 DefaultDownloader，链接提取组件的默认实现则是 DefaultPageProcessor 类。

接着是对象映射组件组件，如代码 4.6 所示，通过继承 HtmlToPojoAdapter 来定义了映射组件。

```
1 public class HtmlToPojo extends HtmlToPojoAdapter<Doc> {  
2 }
```

Listing 4.6: 映射组件定义

4.3.3 爬虫编排

爬虫通过 OperatotFlow 来编排管理各个组件，通过 fluent 编程风格来创建了一个流式爬虫。如代码 4.7 所示，爬虫通过编排 Init、Filter、Downloader 等组件类，便构建了 Spidereact 爬虫。同时，开发者根据 subscribe 方法对爬虫生成的结构化的流式数据进行处理。

```

1 flow.init(Init.class)
2   .filter(Filter.class)
3   .downloader(DefaultDownloader.class)
4   .parser(DefaultPageProcessor.class)
5   .transformer(HtmlToPojo.class)
6   .build()
7   .subscribe();

```

Listing 4.7: 爬虫编排

4.3.4 爬虫反反爬策略配置

下面本文通过一个例子来介绍 Spidereact 的反反爬策略配置。

```

1 \\\ 代理IP、User Agent、Cookies 切换间隔分别为2000ms、3000ms、10000ms
2 Flux<Proxy> proxyFlux = ProxyProvider.setSwitchInterval(2000)
3   .generateFlux();
4 Flux<String> userAgentFlux = UserAgentProvider.setSwitchInterval(3000).
5   .generateFlux();
6 Flux<Cookies> cookiesFlux = CookiesProvider.setSwitchInterval(10000).
7   .generateFlux();
8 flow.init(Init.class)
9   .delayElements(Duration.ofMillis(100))
10  .zipWith(proxyFlux)
11  .zipWith(userAgentFlux)
12  .zipWith(cookiesFlux)
13  .filter(Filter.class)
14  .downloader(DefaultDownloader.class)
15  .onErrorContinue(error -> Mono.empty())
16  .parser(DefaultPageProcessor.class)
17  .transformer(HtmlToPojo.class)
18  .build()
19  .subscribe();

```

Listing 4.8: 爬虫反反爬策略配置

正如上一章 3.2.4 所述，Spidereact 可以通过不同的配置流对 Request 流进行配置。如代码 4.8 所示，proxyFlux、userAgentFlux 以及 cookiesFlux 分别配置了不同的切换时间间隔。Request 数据流经过一段固定时间的爬取后，其相应的代理 IP、User Agent 以及请求的 Cookies 会进行切换，以此来绕过网站设置的

反爬措施。同时 Spidereact 通过 `delayElements` 来设置爬虫请求的延迟，以及通过 `onErrorContinue` 算子来处理各个爬虫组件数据处理过程中的异常。

4.4 本章小结

本章对于结合 Reactive Programming 的流式爬虫方法的设计与实现进行了描述。首先对框架的整体编程模型进行了介绍，并和传统爬虫框架的多线程编程模型进行了对比。然后介绍了爬虫的几个基本组件以及扩展组件的设计，阐述了各个组件的功能。接着对框架的各个模块进行了介绍。

下一章则是实验设计与评估，验证框架的有效性。

第五章 实验设计与评估

前面章节介绍了 Spidereact 的设计与实现，本章则对 Spidereact 的性能进行评估，通过和当前开源爬虫框架的对比来验证 Spidereact 爬虫在资源利用率以及爬取效率方面的提升。

5.1 实验内容

正如前文所诉，Spidereact 是为了解决多线程模型爬虫中存在的阻塞 IO 问题实现的响应式爬虫框架，其目的是提高系统资源的使用率，并提高爬取的效率。因此实验主要对比 Spidereact 爬虫在相同的条件下与其他爬虫框架实现的爬虫在 CPU 利用率以及吞吐量上的差异。同时，还对其他有可能影响实验结果的因素进行了分析，排除其他因素的干扰。具体的实验主要对比两个方面：

- 当资源不受限时，爬虫的爬取效率以及 CPU 使用率之间的对比。
- 在相同的资源限制下，爬虫的爬取效率对比。

5.2 实验设置

5.2.1 实验环境

实验主要的硬件和软件环境如表 5-1 和表 5-2 所示，实验运行在 3 台服务器上，每台服务器上都有一块四核 CPU，并且都采用了超线程技术，每台服务

表 5-1: 硬件配置

硬件类型	型号	数量(大小)
CPU	Intel(R) Xeon(R) CPU E31265L @ 2.40GHz	4 核
内存	TEAMGROUP-SD3-1600 DDR3	16G
网卡	Intel Corporation 82579LM	1Gps
磁盘	C400-MTFDDAT064M	60G

表 5-2: 软件环境

名称	软件版本
操作系统	Linux ubuntu 4.4.0-186-generic
容器引擎	Docker 18.09.7
Java	OpenJDK 1.8

器有 8 个逻辑处理器。

5.2.2 实验对象

对于爬虫性能的测试，实验参考了 Scrapy 框架 Benchmark 测试的方法^[34]，它通过生成本地的 HTTP 服务器，通过 Scrapy 爬虫配置最快的速度进行网页爬取，从而来测试 Scrapy 框架在本地硬件的具体表现，来获得一个通用的比较标准。实验需要模拟搭建一个虚拟网站，其中网站能够模拟现实环境中的网站，即发送请求后，经过一段时间延迟后，返回响应结果。搭建的模拟网站返回的是 Apache Tomcat 的文档，这些文档页面都是静态页面，网站总共 1230 个页面。为了尽量地爬取多个页面，比较性能，实验并未设置爬虫的重复 URL 过滤，意味着爬虫仍然会爬取已经爬取过的页面。

5.2.2.1 网站模拟与压测实验

模拟网站跑在两台服务器上，通过 Nginx 进行负载均衡，使得网站能够尽可能地承受更大的负载，而不至于随着爬虫程序请求量的增加而崩溃。同时我们对网站进行了压测实验，确保了网站自身因素不会影响爬虫测试的效果。

我们通过 gatling^①——一款基于 AKKA^② 和 Netty^③开发的高性能压测工具，对网站进行了压测实验，设定每秒恒定的并发数发送请求，持续 100 秒时间，查看网站吞吐量，平均响应时间等指标。

如表 5-3 所示，我们发现，随着压测实验访问并发度的提高，网站吞吐量也随着增加，请求的平均响应时间也在增加，而请求的成功率随之下降，但变化不大，绝大部分请求都能得到响应。另外在 1000 并发度的情况下压测，我们仍然可以认定请求依旧可以被成功处理，而且在垂直爬虫的实践中，往往不会

^①<https://gatling.io/>

^②<https://akka.io/>

^③<https://netty.io/>

表 5-3: 模拟网站压测

场景	总请求数	平均 TPS	成功率	平均响应时间 (ms)	TP95(ms)
200 并发	20000	200	100%	27	102
500 并发	50000	413	99.2%	1371	2750
1000 并发	100000	757	89.6%	2102	3621

对同一个网站有这么大的并发量，本次爬虫性能测试实验的并发量也不会超过 1000。压测实验表明，网站自身的状况不会影响到爬虫程序之间的性能对比。

5.2.2.2 实验对比框架

Scrapy 是基于事件循环机制的单线程爬虫，通过 Twisted 框架实现的异步网络通信。Scrapy 爬虫运行过程中会启动多个线程，但是爬虫爬取主流程都是跑在单个线程上的，其余的线程用来处理爬取主流程无关的事情。Scrapy 将爬取的主流程是交给 Twisted 线程池，任务结束后，会发起回调执行回调函数。Scrapy 通过 CONCURRENT_REQUESTS 参数来控制下载器 Downloader 的并发，但该参数只是为了控制 Downloader 最大的并发请求数，并不是爬虫本身的线程并发，CONCURRENT_REQUESTS 只能控制传给 Twisted 的异步任务数量。换句话说，无论是在 CPU 单核还是多核环境下，Scrapy 爬虫的 CPU 利用率不会超过 100%，Scrapy 爬虫无法利用多核 CPU 优势来提高爬虫吞吐量。如图 5-1 所示，实验测试了 Scrapy 爬虫在不同 CONCURRENT_REQUESTS 配置下的吞吐量情况。可以看出，随着参数的增大，爬虫的吞吐量并没有显著的增加，一直维持在 60/sec 上下波动。实验过程中，我们确保 Scrapy 的 Request 队列中有足够的待爬取的 Request，排除了 Request 队列待爬取请求数量不足的影响。实验发现不论如何配置 Scrapy 爬虫的相关参数，Scrapy 爬取的吞吐量基本不会发生太大的变化。实验结果表明，Scrapy 爬虫的爬取效率取决于 Twisted 主线程的处理能力，无法利用 CPU 多核的优势。

由于 Scrapy 爬虫的单线程限制，实验无法比较 Scrapy 爬虫框架与 Spidereact 在不同并发下的 CPU 性能以及吞吐量情况。基于这种考虑以及 Python 运行环境的因素，本次实验将重点比较 Webmagic 和 Spidereact 两个爬虫框架之间不同并发下的性能差异。

实验对比的对象是 Spidereact 与 Webmagic 框架，Webmagic 框架是基于 Java 实现的典型的多线程模型网页爬取的爬虫框架，而 Spidereact 则是基于异步非

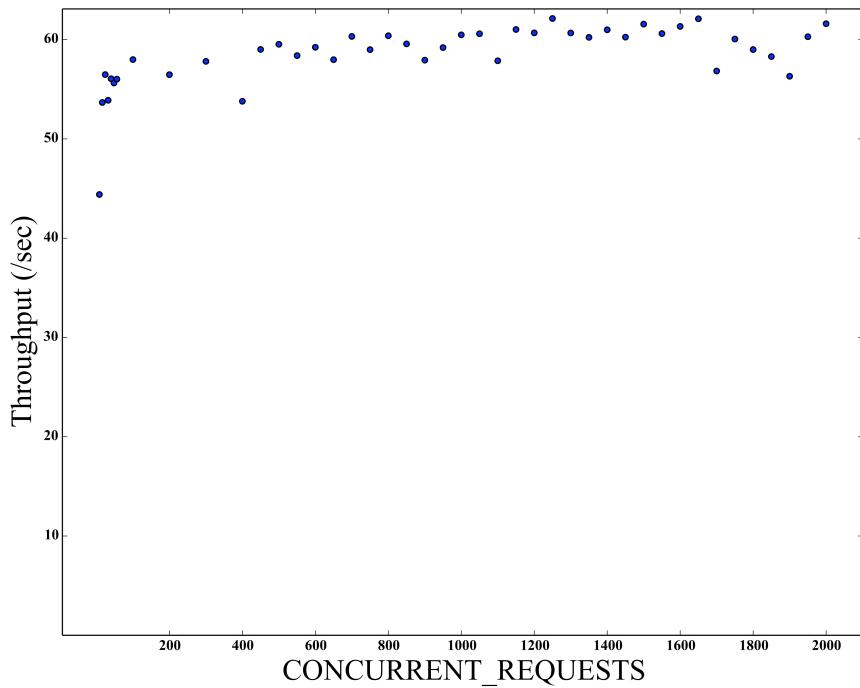


图 5-1: Scrapy 不同 CONCURRENT_REQUESTS 配置下吞吐量

阻塞的响应式爬虫编程模型来进行网页爬取的。为了能够对比爬虫框架的优劣，本次实验通过 Webmagic 和 Spidereact 开发了针对测试网站的爬虫，并对爬虫的并发度进行不同的配置，分别比较不同并发度下 Spidereact 与 Webmagic 的具体性能情况。

5.2.3 模拟延迟

上面提到的实验是针对一个模拟的静态网站的爬取。与真实网站不同的是，爬虫的请求发送到网站服务器端后，网站能够做出很快的响应。因为模拟网站为静态网站，没有额外的处理流程，只是将存储的静态网页返回给了爬虫。为了更进一步模拟真实网站爬取，实验通过 Linux 下的一个流量控制工具 tc (traffic control) 来控制网站端的发包操作，tc 控制直接对物理网卡生效，将物理网卡的发包时延设置 40ms、100ms、200ms 来延迟发送，并设置延迟时间有上下 10ms 的波动，来模拟真实网站的响应处理延迟。

5.3 实验结果与分析

5.3.1 CPU 利用率验证

Spidereact 旨在提高爬虫对 CPU 的利用率，能够有效地利用 CPU 的计算资源来加快爬取的速度。通过对比 Spidereact 爬虫与 Webmagic 爬虫之间的 CPU 使用率，来验证 Spidereact 是否能够提高 CPU 的利用率。

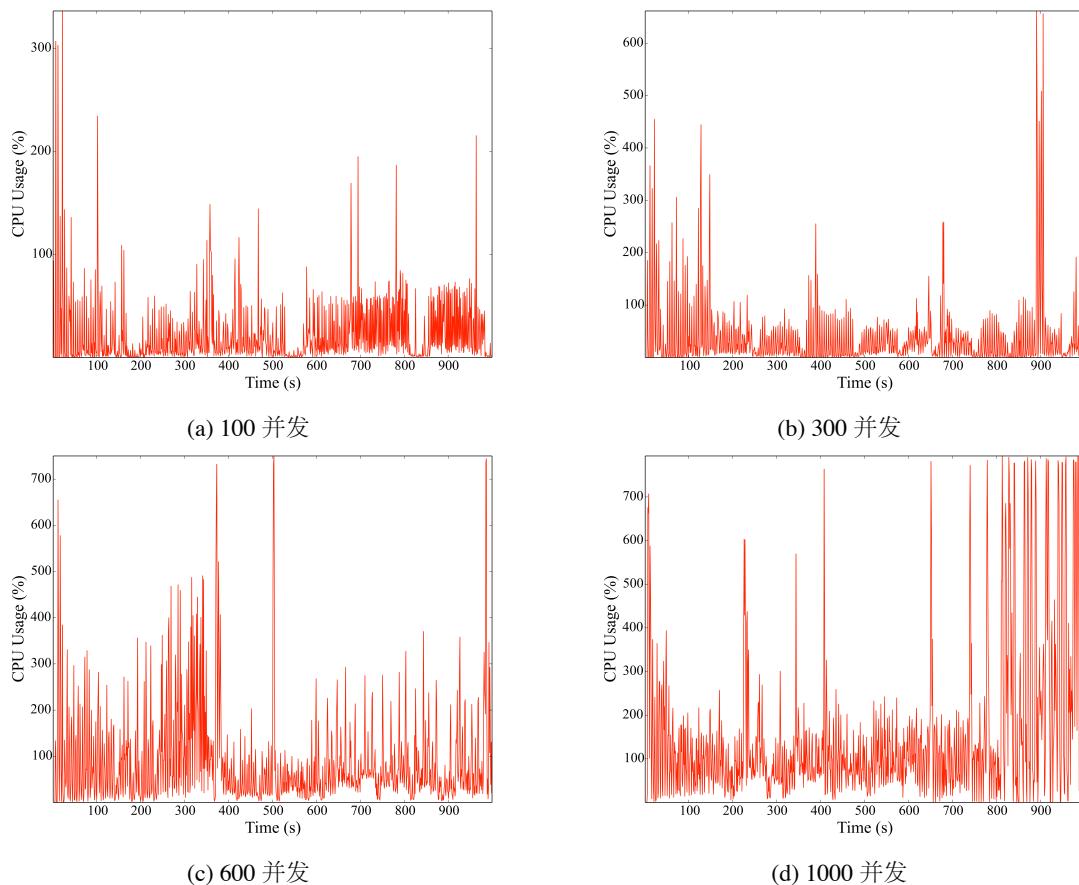


图 5-2: Webmagic 0 时延不同并发下 CPU 使用情况

爬虫 CPU 利用率是爬虫进程的 CPU 时间除以总的 CPU 时间，进程的 CPU 时间包括了执行用户态代码占用的时间和执行内核态代码占用的时间。又由于爬虫是运行在有 8 个逻辑处理器的服务器上，需要将各个逻辑处理器上的 CPU 利用率相加。CPU 利用率计算公式如下所示，其中 Δt 表示进程的 CPU 占用时间， T 表示总 CPU 时间。

$$Usage\% = \sum_{i=1}^n \frac{\Delta t}{T} \quad (5-1)$$

实验基于 Docker 环境来对爬虫进行资源的限制，首先在无资源限制的条件下对 Webmagic 和 Spidereact 进行实验对比。从图 5-2 中我们可以发现，Webmagic 在 0ms 时延^①情况下不同并发度的 CPU 利用率情况。可以发现，在不同并发度下，Webmagic 的 CPU 利用率随时间变化波动都很大。随着爬虫的启动，CPU 利用率陡增，之后回落，接着 CPU 利用率上下波动，呈现一种周期性的特点，并且波动的范围也随着并发数量的增加而变大，这是由于爬虫的内在行为导致的。Webmagic 爬虫在爬取过程中，会对每个到来的 URL 都单独开启一个线程进行处理，这样会导致 CPU 使用率急剧升高，随着爬虫线程被网络 IO 阻塞，CPU 使用率下降，之后随着网页的下载解析，提取出来的链接进队列中，重新开始下一轮的爬取，CPU 使用率又再次升高。另外传统爬虫框架像 Webmagic，并没有设置对上游组件的流量控制，在下载器阶段爬虫会对每个到达的 URL 创建一个线程进行处理，导致了 CPU 利用率的上升。然而，这种模式并不会无限地提高处理速度，随着线程数量的不断增加，由于线程存在的网络阻塞 IO 操作，存在着上下文切换开销，使得系统爬取效率下降。

为了模拟真实网站响应时延，实验分别设置 40ms、100ms、200ms 发包时延。如图 5-3 所示，这是 Webmagic 爬虫在 100 并发度下 40ms，100ms，200ms 延迟设定时的 CPU 使用率情况。我们发现不同延迟场景下的 CPU 使用情况并无太大的明显差异。

Spidereact 爬虫 CPU 利用率如图 5-4 所示，在不同时延设置下，Spidereact 爬虫的 CPU 资源使用率始终能够维持在 100% 以上，CPU 利用率存在一定的波动，但波动范围并没有 Webmagic 爬虫表现得那么剧烈。这是由于 Spidereact 为了能够减少线程切换的开销，异步线程池的并发度默认设置成了 CPU 核数的 10 倍，同时由于 Spidereact 的异步非阻塞编程模型，线程池的增加，并发度的加大并不会对爬虫的性能造成有利的影响，真正影响爬虫效率的关键在于 CPU 运行非阻塞的页面解析代码的效率。所以实验按照了 Spidereact 的默认并发度来进行对比。和 Webmagic 爬虫相比，可以发现 Spidereact 爬虫的 CPU 利用率明显高于 Webmagic 爬虫的 CPU 利用率。

^①0ms 时延是指并未设置模拟延迟，并不是说不存在网络时延

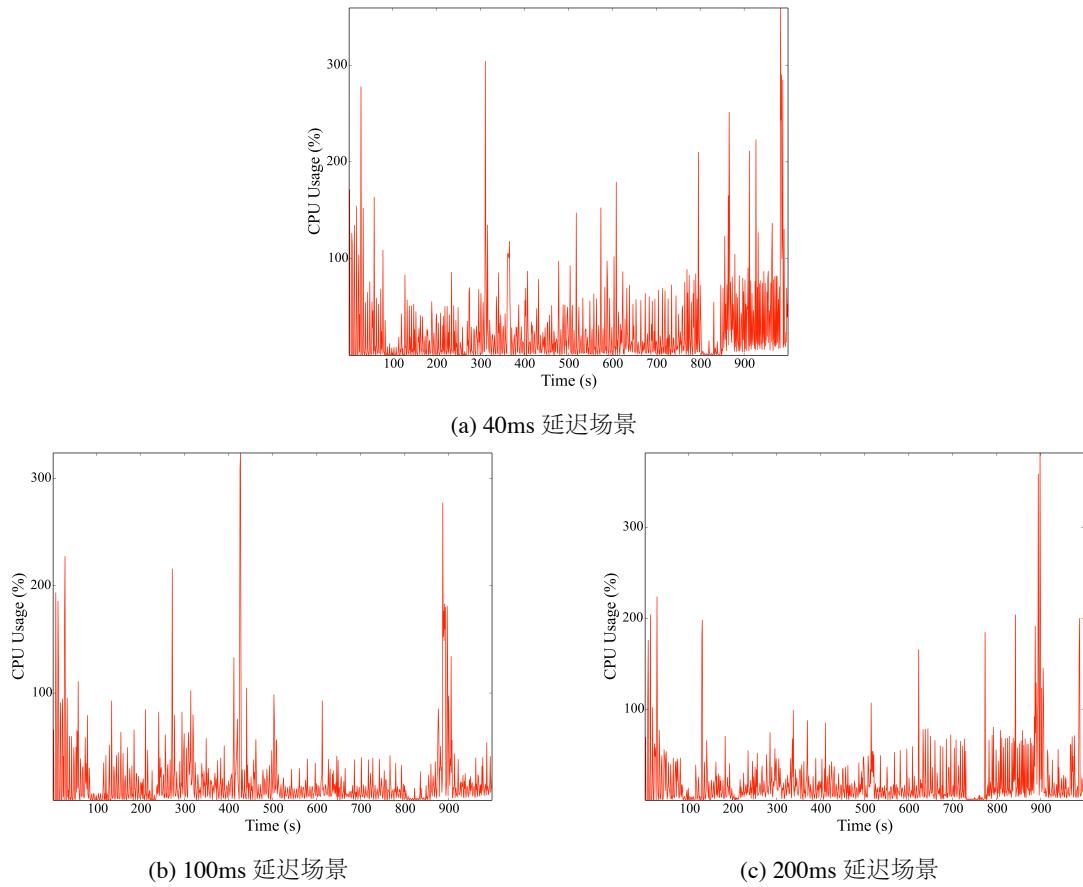


图 5-3: Webmagic100 并发度不同时延下的 CPU 利用率

5.3.2 吞吐量对比

为了测试了 Spidereact 爬虫的吞吐量，实验将 Spidereact 实现的爬虫和 Webmagic 实现的爬虫进行了吞吐量之间的比较。首先，在并没有设置任何时延的情况下，实验先比较了 Webmagic 在不同并发下的吞吐量情况。如图 5-5 所示，可以发现，当并发度较小的时候，吞吐量与并发度之间存在着线性关系，并发度提高，吞吐量也随之线性增长。但是当并发度超过 400 之后，爬虫的吞吐量就没有随之并发度的提高而增长，而是在 80 上下起伏。这种情况也符合我们的预期，因为对于多线程模型 Webmagic 爬虫来说，由于它从下载、解析、处理整个同步流程都是在同一个线程中进行的，整个过程都是线程并发执行的，其加速比是固定的，所以并发度较小的时候，随着线程数的增多，吞吐量呈现线性增长。但当并发度上升到一定程度之后，导致 CPU 并发线程过多，线程上下文切换频繁，内存消耗增加，从而使得吞吐量可能还会下降。从图 5-5 中也可以看出并发度在 500 至 1000 之间变化时，吞吐量有时反而还下降了。

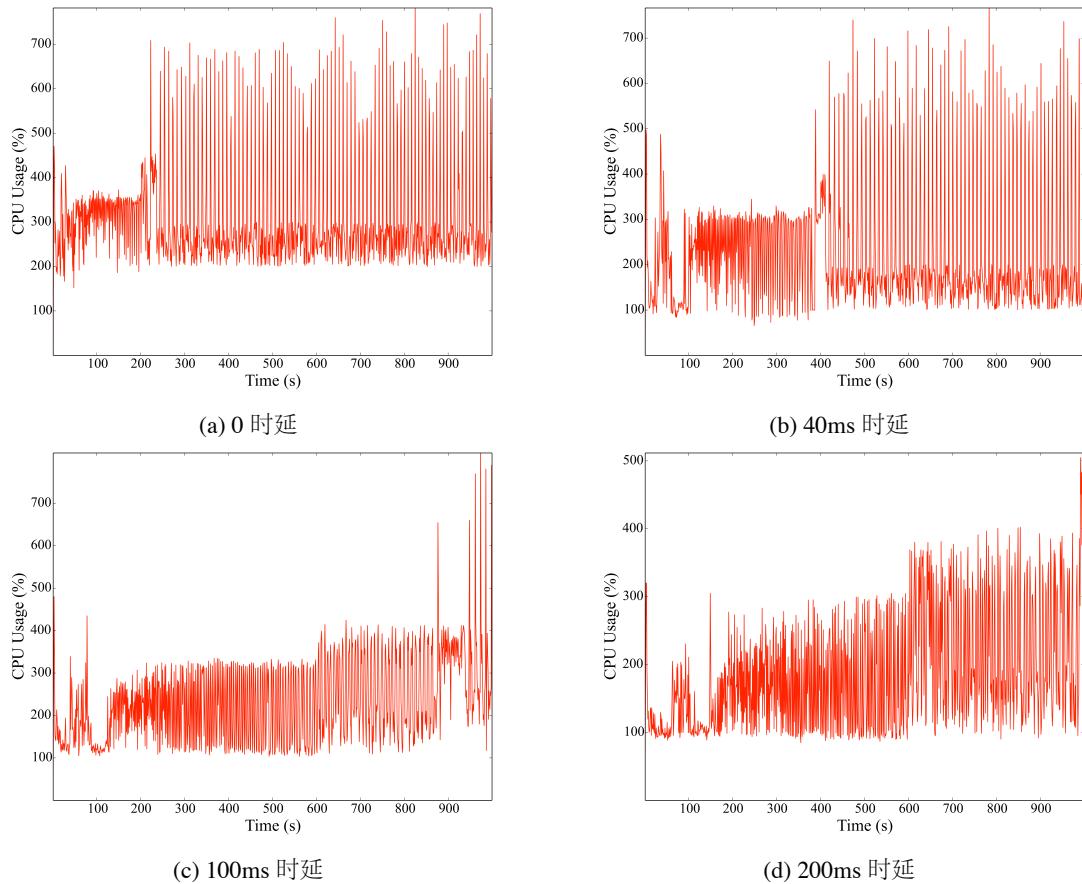


图 5-4: Spidereact 40 并发下不同时延下 CPU 使用情况

抛去偶然因素的影响，吞吐量稳定在一个数值左右。后续与 Spidereact 的对比实验中，实验分别采样了 Webmagic 在 100、300、600、1000 并发下的吞吐量数值。

正如前文所提到的，Spidereact 是基于 Project Reactor 开发的响应式爬虫框架，其并发度默认为系统的 CPU 核数的 10 倍，在本次实验中，Spidereact 的并发度看作是 40。通过表 5-4 将 Webmagic 各个并发度的吞吐量和 Spidereact 的对比，我们发现，在 Spidereact 在吞吐量方面要优于 Webmagic，Spidereact 可以用更少的资源，达到更高的吞吐量。

各个延迟场景下吞吐量对比如表5-4所示，可以看出随着延迟的慢慢增大，两种爬虫的吞吐量都是随之变小的，但设定延迟在40ms、100ms和200ms，吞吐量变化不大。实验再对比WebMagic爬虫相同延迟，不同并发度的场景，可以发现随之并发量的增大，系统吞吐量也随之增大，但增长也是有上限的。可以看到并发度600时，吞吐量最大。从前面的实验也证实了Webmagic

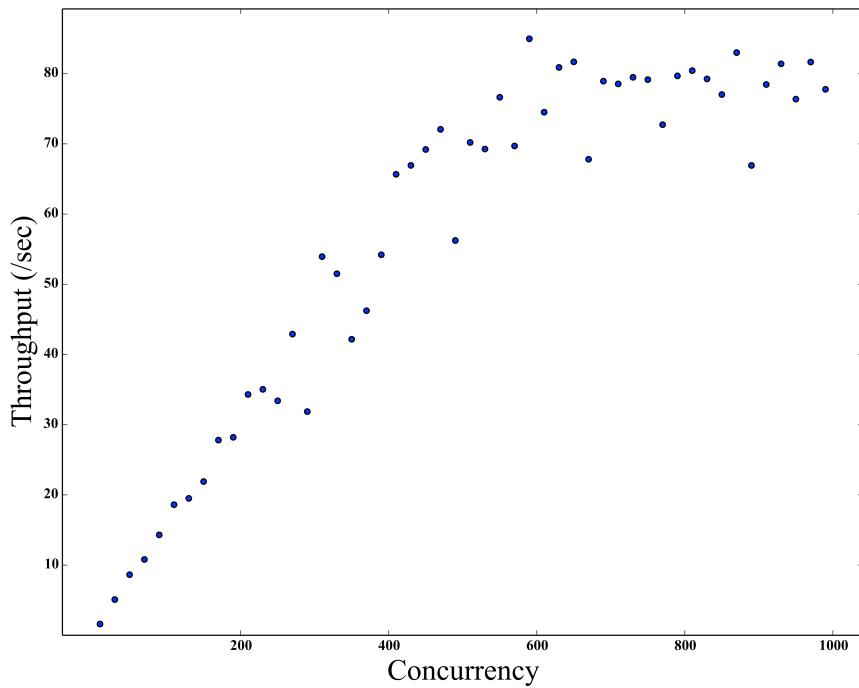


图 5-5: Webmagic 不同并发下吞吐量

爬虫的吞吐量增长限制。针对 Spidereact 爬虫，我们发现随着时延的增加，爬虫的吞吐量变化不大。而且和同时延下的 Webmagic 爬虫比较，可以发现 Spidereact 爬虫吞吐量是优于 WebMagic 的。

5.3.3 资源限制

如表 5-5 所示，实验通过 Docker 在容器启动时对容器资源进行限制，分别设置 CPU 核数为 0.3、1、2，观察在 CPU 资源限制情况下爬虫的吞吐率。通过 Spidereact 爬虫与 Webmagic 爬虫的对比，我们发现在 CPU 资源受限情况下，Spidereact 的吞吐量也优于 Webmagic。

5.4 实验结论

从上两节的实验中，可以看出本文提出的 Spidereact 响应式编程框架，不论是从 CPU 资源利用率方面，还是吞吐量方面，都具有较好的性能表现。

- CPU 资源利用率

在爬虫 CPU 资源利用率方面，Spidereact 爬虫框架明显具有优势。Spidereact

表 5-4: 不同延迟不同并发下吞吐量对比

实验并发度	延迟设置	爬取总量	时间 (ms)	吞吐量 (/sec)
Webmagic-100 并发	0ms	13705	1000000	13.7
Webmagic-100 并发	40ms	13060	1000000	13.06
Webmagic-100 并发	100ms	12932	1000000	12.9
Webmagic-100 并发	200ms	12668	1000000	12.7
Webmagic-300 并发	0ms	36413	1000000	36.4
Webmagic-300 并发	40ms	37336	1000000	37.3
Webmagic-300 并发	100ms	36854	1000000	36.8
Webmagic-300 并发	200ms	35993	1000000	36.0
Webmagic-600 并发	0ms	89482	1000000	89.5
Webmagic-600 并发	40ms	75322	1000000	75.3
Webmagic-600 并发	100ms	72038	1000000	72.0
Webmagic-600 并发	200ms	70142	1000000	70.1
Webmagic-1000 并发	0ms	85839	1000000	85.8
Webmagic-1000 并发	40ms	70026	1000000	70.0
Webmagic-1000 并发	100ms	67928	1000000	67.9
Webmagic-1000 并发	200ms	67402	1000000	67.4
Spidereact	0ms	202530	1000000	202.5
Spidereact	40ms	176721	1000000	176.7
Spidereact	100ms	164236	1000000	164.2
Spidereact	200ms	158825	1000000	158.8

框架能保证 CPU 始终处于忙碌的状态，将爬虫频繁的网络 IO 操作，调度到异步线程池中执行，不阻塞 CPU。

● 吞吐量

在吞吐量方面，Spidereact 爬虫框架，不论是否对 CPU 资源进行限制，相比于其他框架，都具有较好表现。Spidereact 通过爬虫的异步化以及减少线程上下文切换来提高爬虫的吞吐量。

表 5-5: 不同 CPU 资源不同并发下吞吐量对比

CPU 核数	并发数	爬取总量	时间 (ms)	吞吐量 (/sec)
0.3	Spidereact	11381	1000000	11.4
0.3	Webmagic-100 并发	7336	1000000	7.3
0.3	Webmagic-300 并发	7192	1000000	7.2
0.3	Webmagic-600 并发	7015	1000000	7.0
0.3	Webmagic-1000 并发	6945	1000000	6.9
1	Spidereact	111152	1000000	111.2
1	Webmagic-100 并发	12294	1000000	12.3
1	Webmagic-300 并发	34938	1000000	35.0
1	Webmagic-600 并发	36862	1000000	36.9
1	Webmagic-1000 并发	33155	1000000	33.2
2	Spidereact	170066	1000000	170.7
2	Webmagic-100 并发	13247	1000000	13.2
2	Webmagic-300 并发	34793	1000000	34.8
2	Webmagic-600 并发	68566	1000000	68.6
2	Webmagic-1000 并发	64072	1000000	64.1

5.5 本章小结

本章设计的实验主要与 Webmagic 爬虫框架进行了对比，结果验证了 Spidereact 框架的爬虫方法的有效性。本章主要是从系统吞吐量以及 CPU 利用率等方面进行了比较，分析了 Spidereact 相比于多线程模型爬虫框架以及单线程事件循环爬虫框架的优势。

下一章将对本文工作进行总结并对未来工作进行展望。

第六章 结论

6.1 工作总结

垂直爬虫专注于对网页的深层次爬取来获取结构化的数据，目前的爬虫框架都能实现垂直爬虫的功能，并且提供并发爬取的手段，却忽略了爬取过程中大量的阻塞 IO 操作，以及繁琐的解析代码编写过程。本文总结并归纳了目前爬虫框架所存在的问题，结合了 Reactive Programming 思想，提出了 Spidereact 响应式爬虫技术框架，提高了网页数据结构的表示能力，同时在爬取过程中，提高了整个爬虫系统的吞吐量以及系统资源的利用率。

具体总结如下：

- 本文提出了一种异步流式数据爬取模型，通过构造异步数据流，将爬虫爬取过程转变为数据转换过程，通过异步处理爬虫过程中的阻塞操作。同时基于该编程模型，本文还提出了基于层次树模型的对象模型构建方法，建立网站数据与对象模型的映射关系。
- 实现了响应式爬虫技术框架 Spidereact，通过对爬虫各个组件进行组装，提供流式数据接口，实现数据的流式爬取，组件异步执行。同时 Spidereact 支持功能扩展和二次开发。
- 本文对提出的爬虫技术框架 Spidereact 进行了性能对比实验来验证其有效性，通过与当前开源爬虫框架的对比分析系统吞吐率以及 CPU 利用率情况，验证了 Spidereact 的有效性，相比与现有的开源爬虫框架，确实在某些应用场景下提高了系统资源的利用率以及吞吐量。

6.2 研究展望

本文提出的爬虫框架 Spidereact 与其他开源爬虫框架相比，取得了不错的效果。但在未来工作上，本文还有许多的需要改进的地方。

- 本文对于网页结构数据映射模型，还是需要一定的人工参与，需要对网页结构进行分析。未来考虑将模型的创建提供一种半自动的方式，通过预先

对爬取网站结构的扫描，来自动化生成映射模型，使得爬虫的创建编写更加简单且自动化。

- 另外还考虑与 Kubernetes 容器编排框架结合，将 Spidereact 以 Kubernetes Operator 的形式运行在 Kuberneate 平台上，提供相应的 Web UI 接口，使得爬虫开发者能够管理整个爬虫从编写、到部署、最后到结束的整个生命周期。

致 谢

三年的研究生生涯转瞬即逝，在这篇论文即将完稿之际，我想对所有帮助过我的老师、同学、朋友和家人表示感谢。

首先感谢我的导师曹春老师，这篇论文的完成离不开曹老师的悉心指导。在读研期间，曹老师无论是在学习上还是生活上都给予我无微不至的关怀与照顾。曹老师他严谨的治学态度、精益求精的工作作风让我受益匪浅，对我未来的发展有着极其重要的意义。谨在此表达我对曹老师的深深敬意和感激之情。

感谢南京大学软件所的所有老师。感谢吕建老师、马晓星老师、陶先平老师、徐锋老师、徐畅老师、黄宇老师、胡昊老师、余萍老师、李樾老师、马骏老师、汪亮老师、姚远老师、魏恒峰老师、徐经纬老师、蒋炎岩老师等所有关心和帮助过我的老师。

感谢实验室的同学以及我的室友，感谢你们的陪伴，陪我度过了充实而又难忘的三年时光。

最后感谢我的家人，你们的支持与鼓励永远是支撑我前进的最大动力。

参考文献

- [1] RYDNING D R-J G-J. The digitization of the world from edge to core[J]. Framingham: International Data Corporation, 2018.
- [2] ABITEBOUL S, VIANU V. Queries and computation on the Web[J]. Theoretical Computer Science, 2000, 239(2) : 231 – 255.
- [3] SHRIVASTAVA V. A methodical study of web crawler[J]. Vandana Shrivastava Journal of Engineering Research and Application, 2018, 8(11) : 01 – 08.
- [4] DESAI K, DEVULAPALLI V, AGRAWAL S, et al. Web Crawler: Review of Different Types of Web Crawler, Its Issues, Applications and Research Opportunities.[J]. International Journal of Advanced Research in Computer Science, 2017, 8(3).
- [5] ZYTE, CONTRIBUTORS S. A Fast and Powerful Scraping and Web Crawling framework[EB/OL]. 2021 (2021/4/7) [2021/4/15].
<https://scrapy.org/>.
- [6] NAGEL S. Web crawling with Apache Nutch[J]. ApacheCon EU, 2014.
- [7] LU M, WEN S, XIAO Y, et al. The design and implementation of configurable news collection system based on web crawler[C] // 2017 3rd IEEE International Conference on Computer and Communications (ICCC). 2017 : 2812 – 2816.
- [8] SIDDESH G, SURESH K, MADHURI K, et al. Optimizing Crawler4j using MapReduce Programming Model[J]. Journal of The Institution of Engineers (India): Series B, 2017, 98(3) : 329 – 336.
- [9] Yasser Ganjisaffar. crawler4j[EB/OL]. 2018.
<https://github.com/yasserg/crawler4j>.

-
- [10] Nioche. StormCrawler[EB/OL]. 2021 (2020/7/20) [2021/4/15].
[http://stormcrawler.net/.](http://stormcrawler.net/)
 - [11] SUDEEPTHI G, ANURADHA G, BABU M S P. A survey on semantic web search engine[J]. International Journal of Computer Science Issues (IJCSI), 2012, 9(2): 241.
 - [12] AGRE G H, MAHAJAN N V. Keyword focused web crawler[C] // 2015 2nd International Conference on Electronics and Communication Systems (ICECS). 2015 : 1089 – 1092.
 - [13] KUMAR M, BINDAL A, GAUTAM R, et al. Keyword query based focused Web crawler[J]. Procedia Computer Science, 2018, 125 : 584 – 590.
 - [14] KAUSAR M A, DHAKA V, SINGH S K. Web crawler: a review[J]. International Journal of Computer Applications, 2013, 63(2).
 - [15] Wikipedia contributors. Web Crawler[EB/OL]. Wikipedia, The Free Encyclopedia, 2021 (2021/4/16) [2021/4/16].
[https://en.wikipedia.org/wiki/Web_crawler.](https://en.wikipedia.org/wiki/Web_crawler)
 - [16] PANUM T K, HANSEN R R, PEDERSEN J M. Kraaler: A User-Perspective Web Crawler[C] // 2019 Network Traffic Measurement and Analysis Conference (TMA). 2019 : 153 – 160.
 - [17] GRIGALIS T, ČENYS A. Using XPaths of inbound links to cluster template-generated web pages[J]. Computer Science and Information Systems, 2014, 11(1): 111 – 131.
 - [18] MENDELZON A O, MIHAILA G A, MILO T. Querying the world wide web[J]. International Journal on Digital Libraries, 1997, 1(1): 54 – 67.
 - [19] DE BRA P, POST R. Searching for arbitrary information in the www: The fish-search for mosaic[C] // WWW Conference. 1994.
 - [20] SINGH A V, VIKAS A M. A review of web crawler algorithms[J]. International Journal of Computer Science & Information Technologies, 2014, 5(5): 6689 – 6691.

- [21] CHRISTENSEN K, ROGINSKY A, JIMENO M. A new analysis of the false positive rate of a bloom filter[J]. *Information Processing Letters*, 2010, 110(21): 944–949.
- [22] HAND D J, ADAMS N M. Data mining[J]. Wiley StatsRef: Statistics Reference Online, 2014: 1–7.
- [23] SALVANESCHI G, MARGARA A, TAMBURRELLI G. Reactive Programming: A Walkthrough[C/OL] // 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering : Vol 2. 2015 : 953–954.
<http://dx.doi.org/10.1109/ICSE.2015.303>.
- [24] SATYANARAYAN A, RUSSELL R, HOFFSWELL J, et al. Reactive vega: A streaming dataflow architecture for declarative interactive visualization[J]. *IEEE transactions on visualization and computer graphics*, 2015, 22(1): 659–668.
- [25] Kaur K, Rani R. Modeling and querying data in NoSQL databases[C/OL] // 2013 IEEE International Conference on Big Data. 2013 : 1–7.
<http://dx.doi.org/10.1109/BigData.2013.6691765>.
- [26] FIONDA V, PIRRÒ G, GUTIERREZ C. NautiLOD: A formal language for the web of data graph[J]. *ACM Transactions on the Web (TWEB)*, 2015, 9(1): 1–43.
- [27] BOLDI P, CODENOTTI B, SANTINI M, et al. Ubicrawler: A scalable fully distributed web crawler[J]. *Software: Practice and Experience*, 2004, 34(8): 711–726.
- [28] RHEINLÄNDER A, LEHMANN M, KUNKEL A, et al. Potential and pitfalls of domain-specific information extraction at web scale[C] // Proceedings of the 2016 international conference on management of data. 2016 : 759–771.
- [29] SINGH M, VARNICA B. Web Crawler: Extracting the Web Data[J]. *International Journal of Computer Trends and Technology*, 2014, 13(3): 132–137.
- [30] BRIN S, PAGE L. The anatomy of a large-scale hypertextual web search engine[J]. *Computer networks and ISDN systems*, 1998, 30(1-7): 107–117.

-
- [31] KLEINBERG J M, KUMAR R, RAGHAVAN P, et al. The web as a graph: Measurements, models, and methods[C] // International Computing and Combinatorics Conference. 1999 : 1 – 17.
 - [32] Project Reactor contributors. Project Reactor[EB/OL]. 2012 (2021/3/10 [2021/4/15]).
<https://projectreactor.io/>.
 - [33] CHEN M. Improving website structure through reducing information overload[J]. Decision Support Systems, 2018, 110 : 84 – 94.
 - [34] THALHEIM B, SCHEWE K-D, ROMALIS I, et al. Website modeling and website generation[C] // International Conference on Web Engineering. 2004 : 577 – 578.

简历与科研成果

基本信息

何城贤，男，汉族，1997年12月出生，湖南省郴州人。

教育背景

2018 年 9 月 – 2021 年 6 月	南京大学计算机科学与技术系	硕士
2014 年 9 月 – 2018 年 6 月	南京大学计算机科学与技术系	本科

攻读硕士学位期间完成的学术成果

1. 曹春, 马晓星, **何城贤**, 徐经纬, “一种流式爬虫实现方法及系统”, 专利号 202110558553.9

攻读硕士学位期间参与的科研课题

1. 国家重点研发计划“软件定义的人机物融合云计算支撑技术与平台”课题年限（2018年5月-2021年4月）

学位论文出版授权书

本人完全同意《中国优秀博硕士学位论文全文数据库出版章程》（以下简称“章程”），愿意将本人的学位论文提交“中国学术期刊（光盘版）电子杂志社”在《中国博士学位论文全文数据库》、《中国优秀硕士学位论文全文数据库》中全文发表。《中国博士学位论文全文数据库》、《中国优秀硕士学位论文全文数据库》可以以电子、网络及其他数字媒体形式公开出版，并同意编入《中国知识资源总库》，在《中国博硕士学位论文评价数据库》中使用和在互联网上传播，同意按“章程”规定享受相关权益。

作者签名：_____
____年____月____日

论文题名	响应式爬虫框架的研究与实现				
研究生学号	MF1833025	所在院系	计算机科学与技术系	学位年度	2018
论文级别	<input checked="" type="checkbox"/> 硕士 <input type="checkbox"/> 硕士专业学位 <input type="checkbox"/> 博士 <input type="checkbox"/> 博士专业学位 (请在方框内画勾)				
作者电话	18169209977		作者 Email	2364684794@qq.com	
第一导师姓名	曹春 教授		导师电话	18951679203	

论文涉密情况：

不保密

保密，保密期：____年____月____日至____年____月____日

注：请将该授权书填写后装订在学位论文最后一页（南大封面）。

