**网络爬虫**

# 网络爬虫介绍

## 什么是网络爬虫

在大数据时代，信息的采集是一项重要的工作，而互联网中的数据是海量的，如果单纯靠人力进行信息采集，不仅低效繁琐，搜集的成本也会提高。如何自动高效地获取互联网中我们感兴趣的信息并为我们所用是一个重要的问题，而爬虫技术就是为了解决这些问题而生的。

网络爬虫（Web crawler）也叫做网络机器人，可以代替人们自动地在互联网中进行数据信息的采集与整理。它是一种按照一定的规则，自动地抓取万维网信息的程序或者脚本，可以自动采集所有其能够访问到的页面内容，以获取或更新这些网站的内容和检索方式。

从功能上来讲，爬虫一般分为数据采集，处理，储存三个部分。爬虫从一个或若干初始网页的URL开始，获得初始网页上的URL，在抓取网页的过程中，不断从当前页面上抽取新的URL放入队列,直到满足系统的一定停止条件。

我们感兴趣的信息分为不同的类型：如果只是做搜索引擎，那么感兴趣的信息就是互联网中尽可能多的高质量网页；如果要获取某一垂直领域的数据或者有明确的检索需求，那么感兴趣的信息就是根据我们的检索和需求所定位的这些信息，此时，需要过滤掉一些无用信息。前者我们称为通用网络爬虫，后者我们称为聚焦网络爬虫。

## 为什么学网络爬虫

我们初步认识了网络爬虫，但是为什么要学习网络爬虫呢？只有清晰地知道我们的学习目的，才能够更好地学习这一项知识。在此，我们总结了4种常见的学习爬虫的原因：

* + - 1. 可以实现搜索引擎

我们学会了爬虫编写之后，就可以利用爬虫自动地采集互联网中的信息，采集回来后进行相应的存储或处理，在需要检索某些信息的时候，只需在采集回来的信息中进行检索，即实现了私人的搜索引擎。当然，信息怎么爬取、怎么存储、怎么进行分词、怎么进行相关性计算等，都是需要我们进行设计的，爬虫技术主要解决信息爬取的问题。

* + - 1. 大数据时代，可以让我们获取更多的数据源。

在进行大数据分析或者进行数据挖掘的时候，数据源可以从某些提供数据统计的网站获得，也可以从某些文献或内部资料中获得，但是这些获得数据的方式，有时很难满足我们对数据的需求，而手动从互联网中去寻找这些数据，则耗费的精力过大。此时就可以利用爬虫技术，自动地从互联网中获取我们感兴趣的数据内容，并将这些数据内容爬取回来，作为我们的数据源，从而进行更深层次的数据分析，并获得更多有价值的信息。

* + - 1. 可以更好地进行搜索引擎优化（SEO）。

对于很多SEO从业者来说，为了更好的完成工作，那么就必须要对搜索引擎的工作原理非常清楚，同时也需要掌握搜索引擎爬虫的工作原理。而学习爬虫，可以更深层次地理解搜索引擎爬虫的工作原理，这样在进行搜索引擎优化时，才能知己知彼，百战不殆。

* + - 1. 有利于就业。

从就业来说，爬虫工程师方向是不错的选择之一，因为目前爬虫工程师的需求越来越大，而能够胜任这方面岗位的人员较少，所以属于一个比较紧缺的职业方向，并且随着大数据时代和人工智能的来临，爬虫技术的应用将越来越广泛，在未来会拥有很好的发展空间。

爬虫工程师目前来说属于紧缺人才，并且薪资待遇普遍较高。所以，深层次地掌握这门技术，对于就业来说，是非常有帮助的。

# 入门程序

网络爬虫说就是用程序帮助我们访问网络上的资源，我们一直以来都是使用HTTP协议访问互联网的网页，我们也需要编写程序，使用同样的协议访问网页。

这里我们使用Java的HTTP协议库 HttpClient这个技术，来实现抓取网页数据。我们要抓取的是开源中国的HttpClient页面的文章。

## 创建工程

创建工程itheima-crawler

加入以下依赖：

<!-- httpcomponents依赖，包含HttpClient -->

<dependency>

<groupId>org.apache.httpcomponents</groupId>

<artifactId>httpclient</artifactId>

<version>4.5.3</version>

</dependency>

<!-- 日志 -->

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

<version>1.7.25</version>

</dependency>

<!-- 工具类 -->

<dependency>

<groupId>org.apache.commons</groupId>

<artifactId>commons-lang3</artifactId>

<version>3.3.2</version>

</dependency>

<dependency>

<groupId>commons-io</groupId>

<artifactId>commons-io</artifactId>

<version>2.6</version>

</dependency>

<!-- jsoup -->

<dependency>

<groupId>org.jsoup</groupId>

<artifactId>jsoup</artifactId>

<version>1.10.3</version>

</dependency>

加入log4j.properties

log4j.rootLogger=DEBUG,A1

log4j.logger.cn.itcast = DEBUG

log4j.appender.A1=org.apache.log4j.ConsoleAppender

log4j.appender.A1.layout=org.apache.log4j.PatternLayout

log4j.appender.A1.layout.ConversionPattern=%-d{yyyy-MM-dd HH:mm:ss,SSS} [%t] [%c]-[%p] %m%n

## 抓取页面

案例：访问开源中国的HttpComponents项目的介绍页面

<https://www.oschina.net/p/httpclient>

编写以下代码

**public** **static** **void** main(String[] args) **throws** Exception {

// 创建HttpClient

CloseableHttpClient httpClient = HttpClients.*createDefault*();

// 声明HttpGet请求对象

// http://www.oschina.net/p/httpclient

HttpGet httpGet = **new** HttpGet("http://www.oschina.net/p/httpclient");

// 设置请求头包含User-Agent

httpGet.setHeader("User-Agent", "");

// 使用HttpClient发请求,会返回response

CloseableHttpResponse response = httpClient.execute(httpGet);

// 判断响应状态码是否是200

**if** (response.getStatusLine().getStatusCode() == 200) {

// 判断是否有返回数据

**if** (response.getEntity() != **null**) {

// 如果是200，则请求成功，解析返回的数据

String html = EntityUtils.*toString*(response.getEntity(), "UTF-8");

// 把结果输出到文件

// 第一个参数是输出的文件，第二个参数是输出的内容，第二个参数是编码

FileUtils.*writeStringToFile*(**new** File("C:/Users/tree/Desktop/test.html"), html, "UTF-8");

// 解析页面打印文档

System.***out***.println(

StringUtils.*substringBetween*(html, "<div class=\"detail editor-viewer all\" v-pre>", "</div>"));

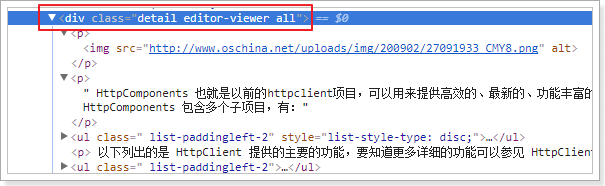
}

}

}

## 解析页面

我们抓取到页面之后，需要对页面进行解析。我们使用浏览器分析发现在一个div标签里面有这个项目文章的正文。这个div标签的class属性为detail editor-viewer all。需要解析这个页面，获取正文内容



我们可以使用以下代码解析页面

// 解析页面，获取需要的内容

String str = StringUtils.*substringBetween*(content, "<section class=\"panel-body\" id=\"v-details\">",

"</section>");

System.***out***.println(str);

这里就已经把内容获取到了，这就是一个最简单的爬虫的实现，无非就是抓取数据，分析数据，存储数据。

我们已经获取了需要的内容，但是还有很多html的标签没有解析，这会影响我们对数据的处理，虽然这些标签也可以手动编写逻辑过滤掉，但是会耗费我们大量时间，一般情况我们会使用专门的html解析工具进行解析。

# Jsoup

jsoup 是一款Java 的HTML解析器，可直接解析某个URL地址、HTML文本内容。它提供了一套非常省力的API，可通过DOM，CSS以及类似于jQuery的操作方法来取出和操作数据。

jsoup的主要功能如下：

从一个URL，文件或字符串中解析HTML；

使用DOM或CSS选择器来查找、取出数据；

可操作HTML元素、属性、文本；

先加入Jsoup依赖：

<!-- jsoup -->

<dependency>

<groupId>org.jsoup</groupId>

<artifactId>jsoup</artifactId>

<version>1.10.3</version>

</dependency>

## jsoup输入

jsou的输入方式有三种：

1. 输入字符串
2. 输入文件
3. 输入url

**public** **static** **void** main(String[] args) **throws** Exception {

// 1.解析字符串

// 读取文件内容，为字符串

// String html = FileUtils.readFileToString(new

// File("C:/Users/tree/Desktop/test.html"), "UTF-8");

// 使用jsoup解析

// 使用jsoup把字符串解析为Document对象

// Document doc = Jsoup.parse(html);

// 2. 解析文件

// Document doc = Jsoup.parse(new

// File("C:/Users/tree/Desktop/test.html"), "UTF-8");

// 3. 解析url

Document doc = Jsoup.*parse*(**new** URL("http://www.oschina.net/p/httpclient"), 10000);

// 使用dom方式获取数据

Element element = doc.getElementById("v-details").child(0);

System.***out***.println(element.text());

}

## Jsoup解析

### 使用dom方式遍历文档

查找元素

getElementById(String id)

getElementsByTag(String tag)

getElementsByClass(String className)

getElementsByAttribute(String key) (and related methods)

Element siblings: siblingElements(), firstElementSibling(), lastElementSibling(); nextElementSibling(), previousElementSibling()

Graph: parent(), children(), child(int index)

元素数据

attr(String key)获取属性attr(String key, String value)设置属性

attributes()获取所有属性

id(), className() and classNames()

text()获取文本内容text(String value) 设置文本内容

html()获取元素内HTMLhtml(String value)设置元素内的HTML内容

outerHtml()获取元素外HTML内容

data()获取数据内容（例如：script和style标签)

tag() and tagName()

### 使用选择器语法查找元素

jsoup elements对象支持类似于CSS (或jquery)的选择器语法，来实现非常强大和灵活的查找功能。这个select 方法在Document, Element,或Elements对象中都可以使用。且是上下文相关的，因此可实现指定元素的过滤，或者链式选择访问。

Select方法将返回一个Elements集合，并提供一组方法来抽取和处理结果。

#### Selector选择器概述

**tagname**: 通过标签查找元素，比如：a

**ns|tag**: 通过标签在命名空间查找元素，比如：可以用 fb|name 语法来查找 <fb:name> 元素

**#id**: 通过ID查找元素，比如：#logo

**.class**: 通过class名称查找元素，比如：.masthead

**[attribute]**: 利用属性查找元素，比如：[href]

**[^attr]**: 利用属性名前缀来查找元素，比如：可以用[^data-] 来查找带有HTML5 Dataset属性的元素

**[attr=value]**: 利用属性值来查找元素，比如：[width=500]

**[attr^=value], [attr$=value], [attr\*=value]**: 利用匹配属性值开头、结尾或包含属性值来查找元素，比如：[href\*=/path/]

**[attr~=regex]**: 利用属性值匹配正则表达式来查找元素，比如： img[src~=(?i)\.(png|jpe?g)]

**\***: 这个符号将匹配所有元素

#### Selector选择器组合使用

**el#id**: 元素+ID，比如： div#logo

**el.class**: 元素+class，比如： div.masthead

**el[attr]**: 元素+class，比如： a[href]

任意组合，比如：a[href].highlight

**ancestor child**: 查找某个元素下子元素，比如：可以用.body p 查找在"body"元素下的所有 p元素

**parent > child**: 查找某个父元素下的直接子元素，比如：可以用div.content > p 查找 p 元素，也可以用body > \* 查找body标签下所有直接子元素

**siblingA + siblingB**: 查找在A元素之前第一个同级元素B，比如：div.head + div

**siblingA ~ siblingX**: 查找A元素之前的同级X元素，比如：h1 ~ p

**el, el, el**:多个选择器组合，查找匹配任一选择器的唯一元素，例如：div.masthead, div.logo

### 伪选择器selectors

**:lt(n)**: 查找哪些元素的同级索引值（它的位置在DOM树中是相对于它的父节点）小于n，比如：td:lt(3) 表示小于三列的元素

**:gt(n)**:查找哪些元素的同级索引值大于n，比如： div p:gt(2)表示哪些div中有包含2个以上的p元素

**:eq(n)**: 查找哪些元素的同级索引值与n相等，比如：form input:eq(1)表示包含一个input标签的Form元素

**:has(seletor)**: 查找匹配选择器包含元素的元素，比如：div:has(p)表示哪些div包含了p元素

**:not(selector)**: 查找与选择器不匹配的元素，比如： div:not(.logo) 表示不包含 class=logo 元素的所有 div 列表

**:contains(text)**: 查找包含给定文本的元素，搜索不区分大不写，比如： p:contains(jsoup)

**:containsOwn(text)**: 查找直接包含给定文本的元素

**:matches(regex)**: 查找哪些元素的文本匹配指定的正则表达式，比如：div:matches((?i)login)

**:matchesOwn(regex)**: 查找自身包含文本匹配指定正则表达式的元素

# 爬虫分类

网络爬虫按照系统结构和实现技术，大致可以分为以下几种类型：通用网络爬虫、聚焦网络爬虫、增量式网络爬虫、深层网络爬虫。 实际的网络爬虫系统通常是几种爬虫技术相结合实现的

## 通用网络爬虫

简单的说就是互联网上抓取所有数据。

通用网络爬虫又称全网爬虫（Scalable Web Crawler），爬行对象从一些种子 URL 扩充到整个 Web，主要为门户站点搜索引擎和大型 Web 服务提供商采集数据。

这类网络爬虫的爬行范围和数量巨大，对于爬行速度和存储空间要求较高，对于爬行页面的顺序要求相对较低，同时由于待刷新的页面太多，通常采用并行工作方式，但需要较长时间才能刷新一次页面。

## 聚焦网络爬虫

简单的说就是互联网上只抓取某一种数据。

聚焦网络爬虫（Focused Crawler），又称主题网络爬虫（Topical Crawler），是指选择性地爬行那些与预先定义好的主题相关页面的网络爬虫。

和通用网络爬虫相比，聚焦爬虫只需要爬行与主题相关的页面，极大地节省了硬件和网络资源，保存的页面也由于数量少而更新快，还可以很好地满足一些特定人群对特定领域信息的需求 。

## 增量式网络爬虫

简单的说就是互联网上只抓取刚刚更新的数据。

增量式网络爬虫（Incremental Web Crawler）是 指 对 已 下 载 网 页 采 取 增量式更新和只爬行新产生的或者已经发生变化网页的爬虫，它能够在一定程度上保证所爬行的页面是尽可能新的页面。

和周期性爬行和刷新页面的网络爬虫相比，增量式爬虫只会在需要的时候爬行新产生或发生更新的页面 ，并不重新下载没有发生变化的页面，可有效减少数据下载量，及时更新已爬行的网页，减小时间和空间上的耗费，但是增加了爬行算法的复杂度和实现难度。

## Deep Web 爬虫

Web 页面按存在方式可以分为表层网页（Surface Web）和深层网页（Deep Web，也称 Invisible Web Pages 或 Hidden Web）。

表层网页是指传统搜索引擎可以索引的页面，以超链接可以到达的静态网页为主构成的 Web 页面。

Deep Web 是那些大部分内容不能通过静态链接获取的、隐藏在搜索表单后的，只有用户提交一些关键词才能获得的 Web 页面。

# 案例介绍

前面介绍了几种爬虫的分类，我们可以使用聚焦网络爬虫，抓取京东上面的商品数据。

同时也需要不断的爬去商品数据，以获取最新的商品数据信息，所以也要用增量式网络爬虫。可以使用到定时任务Quartz定时爬去数据

## 抓取流程分析

获取基础数据：

1. 获取京东商城所有商品的分类url
2. 保存到数据库

抓取商品数据：

1. 分析商品分类，获取所有的商品详情url
2. URL管理器处理URL，作用：

1). 添加新的待爬url

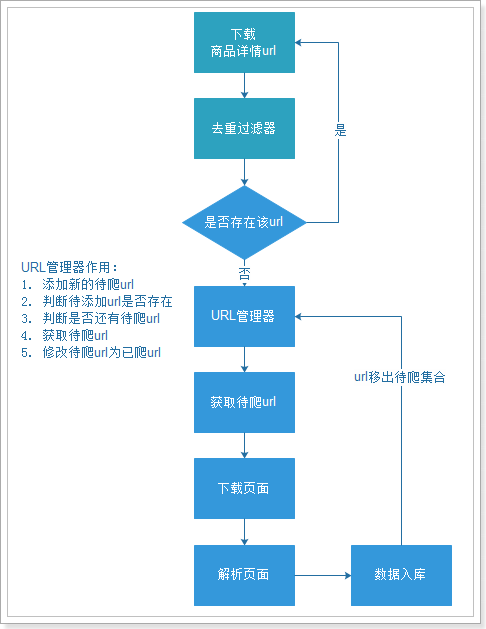
2). 判断待添加url是否存在

3). 判断是否还有待爬url

4). 获取待爬url

5). 修改待爬url为已爬url

1. 网页下载器获取待爬url
2. 网页解析器解析页面获取商品数据
3. 保存商品数据到数据库中



## 去重过滤器

在使用网络爬虫过程中，去重是一个不可避免的问题，这里需要对将要爬取得商品详情url进行去重操作

传统的去重，可以使用Map或者Set集合，或者哈希表的方式来实现。在数据量较小的情况下，使用这种方式没有问题，可是当我们需要大量爬去数据的时候，这种方式就存在很大问题。因为会极大的占用内存和系统资源，导致爬虫系统崩溃。

这里将会给大家介绍两种过滤方式：布隆过滤器和redis

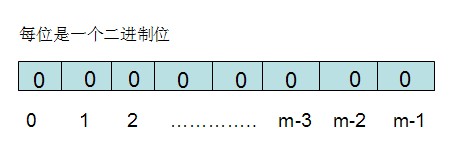
### 布隆过滤器

布隆过滤器 (Bloom Filter)是由Burton Howard Bloom于1970年提出，它是一种space efficient的概率型数据结构，用于判断一个元素是否在集合中。在垃圾邮件过滤的黑白名单方法、爬虫(Crawler)的网址判重模块中等等经常被用到。

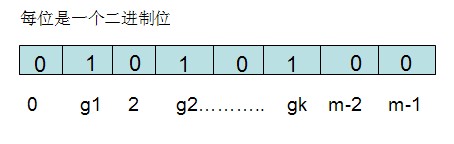
哈希表也能用于判断元素是否在集合中，但是布隆过滤器只需要哈希表的1/8或1/4的空间复杂度就能完成同样的问题。布隆过滤器可以插入元素，但不可以删除已有元素。其中的元素越多，误报率越大，但是漏报是不可能的。

**布隆过滤器原理**

布隆过滤器需要的是一个位数组(和位图类似)和K个映射函数(和Hash表类似)，在初始状态时，对于长度为m的位数组array，它的所有位被置0。



对于有n个元素的集合S={S1,S2...Sn},通过k个映射函数{f1,f2,......fk}，将集合S中的每个元素Sj(1<=j<=n)映射为K个值{g1,g2...gk}，然后再将位数组array中相对应的array[g1],array[g2]......array[gk]置为1：



如果要查找某个元素item是否在S中，则通过映射函数{f1,f2,...fk}得到k个值{g1,g2...gk}，然后再判断array[g1],array[g2]...array[gk]是否都为1，若全为1，则item在S中，否则item不在S中。

布隆过滤器会造成一定的误判，因为集合中的若干个元素通过映射之后得到的数值恰巧包括g1,g2,...gk，在这种情况下可能会造成误判，但是概率很小。

布隆过滤器实现：

//ip去重过滤器，布隆过滤器

**public** **class** BloomFilter {

/\* BitSet初始分配2^24个bit \*/

**private** **static** **final** **int** ***DEFAULT\_SIZE*** = 1 << 24;

/\* 不同哈希函数的种子，一般应取质数 \*/

**private** **static** **final** **int**[] ***seeds*** = **new** **int**[] { 5, 7, 11, 13, 31, 37 };

**private** BitSet bits = **new** BitSet(***DEFAULT\_SIZE***);

/\* 哈希函数对象 \*/

**private** SimpleHash[] func = **new** SimpleHash[***seeds***.length];

**public** BloomFilter() {

**for** (**int** i = 0; i < ***seeds***.length; i++) {

func[i] = **new** SimpleHash(***DEFAULT\_SIZE***, ***seeds***[i]);

}

}

// 将url标记到bits中

**public** **void** add(String url) {

**for** (SimpleHash f : func) {

bits.set(f.hash(url), **true**);

}

}

// 判断是否已经被bits标记

**public** **boolean** contains(String url) {

**if** (StringUtils.*isBlank*(url)) {

**return** **false**;

}

**boolean** ret = **true**;

**for** (SimpleHash f : func) {

ret = ret && bits.get(f.hash(url));

}

**return** ret;

}

/\* 哈希函数类 \*/

**public** **static** **class** SimpleHash {

**private** **int** cap;

**private** **int** seed;

**public** SimpleHash(**int** cap, **int** seed) {

**this**.cap = cap;

**this**.seed = seed;

}

// hash函数，采用简单的加权和hash

**public** **int** hash(String value) {

**int** result = 0;

**int** len = value.length();

**for** (**int** i = 0; i < len; i++) {

result = seed \* result + value.charAt(i);

}

**return** (cap - 1) & result;

}

}

}

### redis过滤

无论我们使用Map、Set、Hash表还是布隆过滤器的方式去重，都是需要占用网络爬虫所在的服务器资源，所以如果碰到超大型的数据处理，还是有所不足

redis也有去重功能，就是set数据类型，我们也可以把需要去重的数据放到redis的set集合中，根据返回结果来判断这条数据是否重复

# 动态网页

上面抓取商品详情的例子中，我们发现可以分析静态页面html，但是对js的解析部分还是很薄弱。虽然我们可以读取js的运作机制并且找到相关数据，但是这样会耗费大量时间。

除了人力解析js以外，我们还可以使用工具来模拟浏览器的运行，直接获取解析结果。这里我们使用

## phantomJs+selenium

（1）一个基于webkit内核的无头浏览器，即没有UI界面，即它就是一个浏览器，只是其内的点击、翻页等人为相关操作需要程序设计实现。

（2）提供javascript API接口，即通过编写js程序可以直接与webkit内核交互，在此之上可以结合java语言等，通过java调用js等相关操作，从而解决了以前c/c++才能比较好的基于webkit开发优质采集器的限制。

（3）提供windows、linux、mac等不同os的安装使用包，也就是说可以在不同平台上二次开发采集项目或是自动项目测试等工作。

下载phantomJs

在官网上下载

http://phantomjs.org/download.html

selenium官网

<http://www.seleniumhq.org/projects/webdriver/>

工程加入依赖

<!-- selenium -->

<dependency>

<groupId>org.seleniumhq.selenium</groupId>

<artifactId>selenium-java</artifactId>

<version>3.4.0</version>

</dependency>

<!-- phantomjsdriver -->

<dependency>

<groupId>com.codeborne</groupId>

<artifactId>phantomjsdriver</artifactId>

<version>1.4.3</version>

</dependency>

测试案例：

**public** **static** **void** main(String[] args) {

// 设置必要参数

DesiredCapabilities dcaps = **new** DesiredCapabilities();

// ssl证书支持

dcaps.setCapability("acceptSslCerts", **true**);

// 截屏支持

dcaps.setCapability("takesScreenshot", **true**);

// css搜索支持

dcaps.setCapability("cssSelectorsEnabled", **true**);

// js支持

dcaps.setJavascriptEnabled(**true**);

// 驱动支持

dcaps.setCapability(PhantomJSDriverService.***PHANTOMJS\_EXECUTABLE\_PATH\_PROPERTY***,

"E:\\phantomjs-2.1.1-windows\\bin\\phantomjs.exe");

// // 创建无界面浏览器对象

WebDriver driver = **new** PhantomJSDriver(dcaps);

**try** {

// 让浏览器访问空间主页

driver.manage().timeouts().implicitlyWait(5, TimeUnit.***SECONDS***);

driver.get("https://item.jd.com/4391570.html");

Thread.*sleep*(5000l);

String html = driver.getPageSource();

Writer out = **new** FileWriter(**new** File("D:/httpclient.html"));

out.write(html);

out.close();

} **catch** (Exception e) {

e.printStackTrace();

} **finally** {

// 关闭并退出浏览器

driver.close();

driver.quit();

}

}