第1章 门户网站用户大数据分析系统

实施国家大数据战略是建立数字时代国家竞争优势的基础，世界各国都已充分认识到大数据对于国家的战略意义，并早早开始布局。本章将介绍使用 Java 语言开发一个门户网站用户大数据分析系统的方法，并详细介绍网络爬虫和数据可视化分析的流程。本章项目由网络爬虫+JSP+MySQL+Echarts 实现。

1.1 大数据介绍


扫码看视频

大数据(big data)又称巨量资料，指的是所涉及的资料量规模巨大到无法通过目前主流软件工具，在合理时间内达到获取、管理、处理并整理成为帮助企业完成经营决策目的的数据。在 20 世纪 80 年代，“大数据”这个词就已经出现，但是它仅用来形容数据量大。随着计算机技术的不断发展，数据不再指简单的数字集合，而是指无法在有限时间内用传统的信息技术和软硬件工具对其进行感知、获取、管理、处理的方式。但对于“大数据”的具体定义，目前学术界尚未形成明确的认知。

2012 年，高德纳咨询公司认为：大数据是非常重要的信息资产，但它需要用新的运算方式来处理，以提高这项信息资产的决策力、洞察力，并用这些特征来描述大数据。麦肯锡(McKinsey)认为：想要在特定时间内对大数据的内容进行搜集、存储、分析、运用，依靠过去传统的数据处理方式已不能实现。

1.1.1 大数据的特征

关于“大数据”的特征描述，代表性的观点有：IBM 将“大数据”的特点总结为 3V，即大量化(volume)、多样化(variety)和快速化(velocity)；著名的数据管理大师维克托·迈尔-舍恩伯格(Viktor Mayer-Schönberger)则认为大数据具有 4 个特点，即 4V，在前面的基础上增加了 value(价值密度低)。目前，4V 特点已成为最基本的共识，这些特点使大数据有别于传统的数据概念。

数据量大是大数据的基本属性。想要收集大量数据是十分困难的，过去只有部分机构能够完成抽样调查，而现在，随着互联网的普及，用户通过智能化的媒介有意的分享或无意的点击、浏览都会产生大量数据。数据量大还体现在人们处理数据的方法和理念上。之前人们对事物的认知一直依据抽样调查，以部分数据来描述整体事物，但在某些领域，这种方法不能形成完整的描述，可能会忽略很多重要信息，甚至得到相反的结果。如今大多数领域的大数据依托云计算，不再采取部分样本来反映总体数据，提高了数据的准确性。

1.1.2 大数据技术的应用

数据的丰富意味着信息的丰富。海量信息的合理分析整合，对于企业管理层决策和政府部门决策都有很重要的指导意义。有实力的企业和政府部门都可以建立一套大数据处理系统来指导其作出决策。在数据大爆炸的时代，专门处理大数据的企业将迎来春天，因为还有很多企业不具备建立完善的大数据分析处理系统的能力。

随着大数据时代的到来，新的商业模式正在诞生，能否运用大数据技术完成商业模式的转型将是许多企业能否坚持下去的关键。同样，大数据时代的到来也给了新兴企业一个极佳的发展机遇。小米就是大数据时代新兴企业的一个典型代表，其依托互联网的营销模式和收集用户反馈信息进行分析处理以改善产品体验的方式，就是大数据技术的一个应用。

1.2　系统设计

在开发一个软件项目时，第一步永远是进行系统设计工作，预先规划整个项目的功能模块和运行流程。下面详细讲解本项目的系统设计过程。

扫码看视频

1.2.1　背景介绍

随着科学技术的飞速发展和社会经济水平的不断进步，互联网规模迅速膨胀。据中新社援引中国互联网络信息中心发布的第 51 次《中国互联网络发展状况统计报告》报道，截至 2022 年 12 月，中国网民规模达 10.67 亿，较 2021 年 12 月增长 3549 万，互联网普及率达 75.6%，这充分说明了互联网已经逐渐成为人类生活、学习所依赖的一部分。

网民每天的网络行为带来了网络用户行为数据的爆炸式增长。网络用户行为数据中蕴含着大量有价值、有意义的信息，通过对用户行为日志进行统计、分析，并将结果通过前台直观的报表展示，可以帮助营销商大致掌握用户的喜好，从中发现用户使用产品的规律。将这些规律与网站的营销策略、产品功能、运营策略相结合，对用户进行智能推荐，可以优化用户体验，实现更精细化和精准化的运营与营销，让产品销量获得更好的增长。此外，通过数据分析可以预测用户的行为倾向，为有关部门对网络舆论进行合理的监控和干预提供理论依据，还可以帮助公安部门针对犯罪嫌疑人进行网络行为监控等。

1.2.2　系统目标

根据系统需求分析，本项目的系统目标如下。

- 爬取知乎用户公开的个人资料信息。
- 构建专有爬虫 HTTP 代理池，突破同一客户端访问量的限制。
- 将数据保存到 MySQL 数据库。
- 多线程爬取，提高爬取速度。
- 对爬取到的知乎用户进行数据分析。

1.2.3 系统功能结构

根据系统需求，可以将系统分为两大模块：知乎爬取模块和代理功能模块。两大模块及其具体的功能模块如图 1-1 所示。

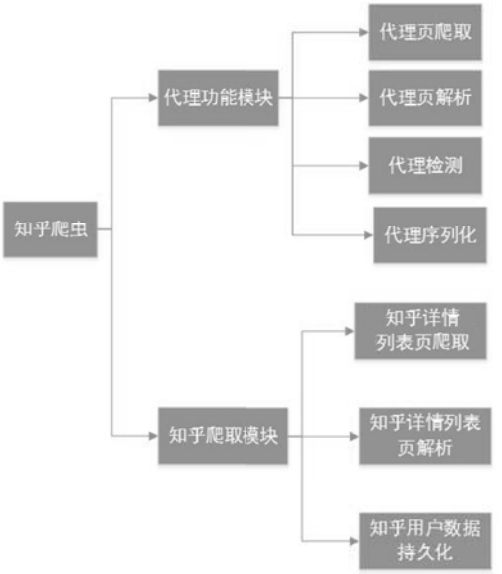

图 1-1 爬取系统功能模块划分

1.3 数据库设计

本系统用 MySQL 数据库存储数据，将爬取到的数据保存到 MySQL 数据库中。作为一个网络爬虫系统，相比其他管理类系统，数据库设计简单许多。本项目只涉及两个表：一个是 user 表，用于存放知乎用户信息；另一个是 url 表，用来存放 URL 数据。其中用户表 user 的设计结构如表 1-1 所示。


扫码看视频

表 1-1 user 表的设计结构

列 名	数据类型	字段说明
id	int(11)	自增 id
user_token	varchar(100)	个性地址令牌，唯一
location	varchar(100)	位置

续表

列　名	数据类型	字段说明
business	varchar(255)	行业
sex	varchar(255)	性别
employment	varchar(255)	企业
education	varchar(255)	教育
position	varchar(255)	职位
username	varchar(255)	用户名
url	varchar(255)	用户首页 url
agrees	int(11)	赞同数
thanks	int(11)	感谢数
asks	int(11)	提问数
answers	int(11)	回答问题数
posts	int(11)	文章数
followees	int(11)	关注数
followers	int(11)	粉丝数
hashId	varchar(255)	用户唯一标识

表 url 的具体设计结构如表 1-2 所示。

表 1-2　url 表的设计结构

列　名	数据类型	字段说明
id	int(11)	自增 id
user_ md5_url	varchar(35)	url 爬取连接的 md5 摘要，唯一键

1.4　爬虫请求分析

就目前的情况，大部分网页上能看到的数据都是直接在网站后台生成好的(有的网页是在网站前端通过 JavaScripts 代码处理后显示的，例如数据混淆、加密等)，并直接在前台显示。虽然也有很多网站采用了 Ajax 异步加载功能，但归根结底它还是一个 HTTP 请求。只要能够分析出对应数据的请求来源，就能很容易地得到想要的数据。在接下来的内容中，将详细讲解 HTTP 请求的方法。

扫码看视频

(1) 以获取笔者知乎账户的所有关注用户资料为例。首先打开笔者关注列表页地址 https://www.zhihu.com/people/wo-yan-chen-mo/following，可以看到，主面板就是笔者关注用户列表。此时有 261 个关注用户，我们的目的就是获取这 261 个用户的个人资料信息。

(2) 打开 Chrome 浏览器，输入网址 https://www.zhihu.com/people/wo-yan-chen-mo/following，然后按 F12 键，依次单击 Network、XHR 按钮，勾选 Preserve log 和 Disable cache 两个复选框，如图 1-2 所示。

图 1-2　勾选 Preserve log 和 Disable cache 复选框

(3) 下拉滚动条，通过单击“下一页”按钮的方式来到第 4 页，获取对应请求(在翻页的过程中会有很多无关的请求，请不要理会)。待页面加载完成后，在请求列表中右击鼠标，在弹出的快捷菜单中选择 Save as HAR with content 命令。这个命令的功能是把当前请求(request)列表保存为 json 格式文本，然后使用 Chrome 浏览器打开这个文件，方法是单击浏览器中的“搜索”按钮(快捷键是 Ctrl+F)，在页面中提示输入搜索关键字，要注意这里中文采用了 Unicode 编码，此时直接搜索 9692(知乎账号“晨光文具”的关注者数)。这一步骤的目的是获取数据(关注用户的个人资料)的请求来源，具体如图 1-3 所示。

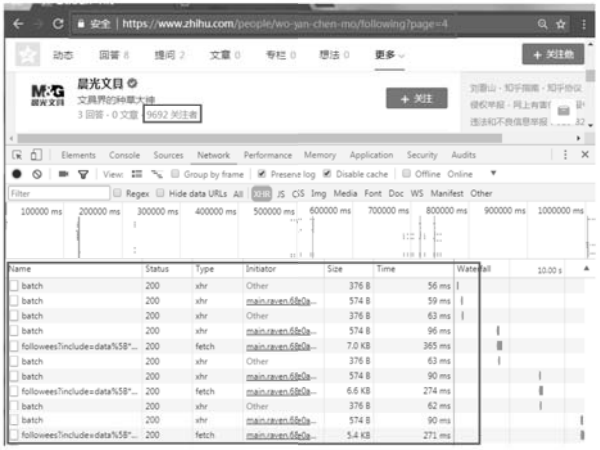

图 1-3　获取请求来源

(4) 由前面的搜索得出，关注用户的资料数据来自以下请求，如图 1-4 所示。URL 解码后为(命名为 url1)：

https://www.zhihu.com/api/v4/members/wo-yan-chen-mo/followees?include=data%5B*%5D.answer_count%2Carticles_count%2Cgender%2Cfollower_count%2Cis_followed%2Cis_following%2Cbadge%5B%3F(type%3Dbest_answerer)%5D.topics&offset=40&limit=20

图 1-4　请求 URL

从解码后的 URL 中可以看出，关注列表的数据并不是从 https://www.zhihu.com/people/wo-yan-chen-mo/following?page=4 同步加载而来的，而是直接通过 Ajax 异步请求 url1 来获得关注用户数据，然后通过 JavaScripts 代码填充数据。这里要注意用下方矩形圈住的 request 头 authorization，在实现代码的时候必须加上这个 Header(头)。这个数据并不是动态改变的，通过步骤(3)的方式可以发现它是来自一个 JavaScripts 文件。该步骤要注意的是，随着时间的推移，知乎可能会更新相关 API 接口的 URL，也就是说，通过步骤(3)得出的 URL 有可能并不是上面的 url1，但具体的分析方法是通用的。

(5) 经过多次测试分析后，可以得出以上 url1 的参数含义，如表 1-3 所示。

表 1-3　url1 的参数含义

参数名	类　型	是否必填	值	说　明
include	String	是	data[*]answer_count, articles_count	需要返回的字段(可以根据需要增加一些字段)

续表

参数名	类　型	是否必填	值	说　明
offset	int	是	0	偏移量(可以获取一个用户的所有关注用户资料)
limit	int	是	20	返回用户数(最大为 20，超过 20 无效)

关于如何测试请求，建议采用以下三种方式。

- 原生 Chrome 浏览器：可以做一些简单的 GET 请求测试。这种方式有很大的局限性，不能编辑 HTTP Header(头)。如果直接(未登录知乎)通过浏览器访问 url1，会得到 401 错误的反馈代码。因为它没有带上 request 头 authorization，所以这种方式能测试一些简单且没有特殊 request 头的 GET 请求，如图 1-5 所示。

图 1-5　错误提示

- Chrome 插件 Postman：一个强大的 HTTP 请求测试工具，可以直接编辑 request，包括 cookies。它支持 GET、POST、PUT，几乎可以发送任意类型的 HTTP 请求。通过修改参数的值，来观察服务器响应数据的变化，从而确定参数的含义，如图 1-6 所示。

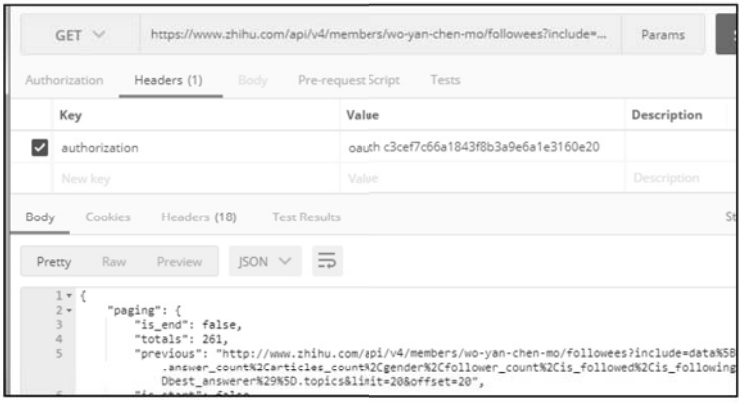

图 1-6　插件 Postman

- intellij idea ultimate 版自带的工具：打开方式是选择 Tools | Test RESTful Web Service 命令。也可以直接编辑 HTTP Header(包括 cookies)，然后发送请求，并且支持 GET、POST、PUT 等请求方式。

1.5　系统组织结构和运行流程图

在系统开发过程中，为了便于整个项目的管理和后期维护，需要规划好整个系统项目文件夹结构，并规划设计系统运行流程图。

扫码看视频

1.5.1　系统组织结构

按照系统功能划分文件夹，本项目的文件夹组织结构如图 1-7 所示。

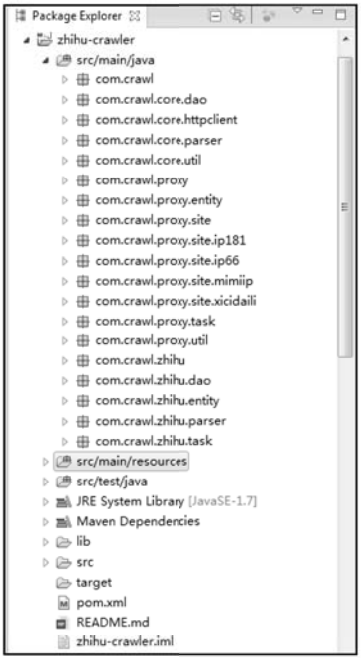

图 1-7　系统文件夹组织结构

1.5.2　系统运行流程图

整个项目程序的运行流程如图 1-8 所示。

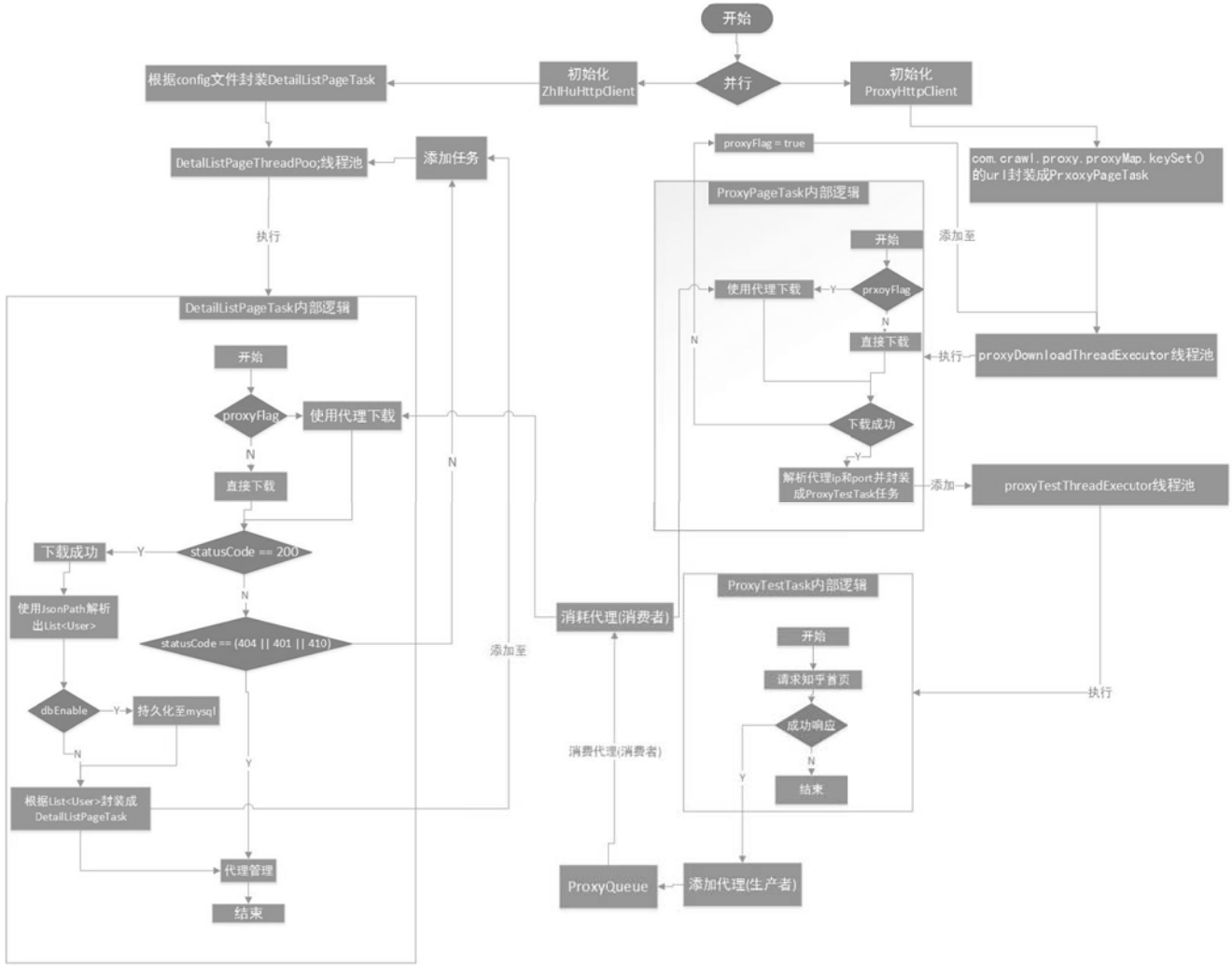
开始
并行
初始化 ZhIHuHttpClient
初始化 ProxyHttpClient
根据config文件封装DetailListPageTask
DetalListPageThreadPoo;线程池
添加任务
执行
DetailListPageTask内部逻辑
开始
proxyFlag
使用代理下载
N
直接下载
下载成功
Y
statusCode == 200
N
使用JsonPath解析出List<User>
statusCode == (404 || 401 || 410)
添加至
dbEnable
Y
持久化至mysql
N
根据List<User>封装成DetailListPageTask
代理管理
结束
proxyFlag = true
com.crawl.proxy.proxyMap.keySet()的url封装成PrxoxyPageTask
ProxyPageTask内部逻辑
开始
添加至
使用代理下载
Y
prxoyFlag
N
直接下载
执行
proxyDownloadThreadExecutor线程池
N
下载成功
Y
解析代理ip和port并封装成ProxyTestTask任务
添加
proxyTestThreadExecutor线程池
消耗代理(消费者)
ProxyTestTask内部逻辑
开始
请求知乎首页
成功响应
N
结束
Y
执行
消费代理(消费者)
ProxyQueue
添加代理(生产者)

图 1-8　系统运行流程图

1.6　实现核心模块

为了提升本项目代码的可读性和重用性，本模块主要实现了整个项目的公共操作类、核心工具类、配置实体类，其中，公共操作包括数据库相关操作，核心工具包括 HTTP 请求工具。

扫码看视频

1.6.1　HTTP 请求的执行

本例是一个实战爬虫项目，最基本的功能便是实现 HTTP 请求。本功能的实现文件是 HttpClientUtil.java，本功能的难点在于 HttpClientContext 这个对象不是线程安全的，在多线程的情况下有一定的出错概率，并且问题不容易排查。官方文档对这点也有说明，建议读者在写代码的过程中要养成看官方文档的习惯，而不是遇到问题后才去搜索引擎寻找解决方法。网上相关资料很少，一般还是通过阅读源码才能解决这个问题。实例文件 HttpClientUtil.java 的主要代码如下。

```
/**
 * 根据 url，返回网页响应内容
 * @param url 网页地址
 * @return 网页内容
 * @throws IOException
 */
public static String getWebPage(String url) throws IOException {
    //根据网页地址创建一个 httpget 请求
   HttpGet request = new HttpGet(url);
   //调用本类中的方法
   return getWebPage(request, "utf-8");
}

/**
 * 根据 request 请求对象，返回响应内容
 * @param request http 请求
 * @return 网页内容
 * @throws IOException
 */
public static String getWebPage(HttpRequestBase request) throws IOException {
    //调用本类中的方法
   return getWebPage(request, "utf-8");
}

/**
```

```
 * @param encoding 字符编码
    * @return 网页内容
    */
public static String getWebPage(HttpRequestBase request,
        String encoding) throws IOException {
   //调用本类中的方法，获取 CloseableHttpResponse 对象
   CloseableHttpResponse response = null;
   response = getResponse(request);
   //从 response 对象中获取响应状态码(200 表示响应成功)，并输出到日志
   logger.info("status---" + response.getStatusLine().getStatusCode());
      //通过 HttpClient 工具类 EntityUtils 从 response 对象解析出网页响应内容
   String content = EntityUtils.toString(response.getEntity(),encoding);
   //释放 http 连接
   request.releaseConnection();
   //返回网页内容
   return content;
}

/**
 * 根据 request 对象，返回 CloseableHttpResponse 对象
 * @param request http 请求
 * @return 网页内容
 * @throws IOException
 */
public static CloseableHttpResponse getResponse(HttpRequestBase request) throws
        IOException {
    //判断该请求是否有额外配置(如是否有代理情况、超时时间等配置)，如果没有则使用项目默认配置
   if (request.getConfig() == null){
      request.setConfig(requestConfig);
   }
   /**
    * 从 Constants 类中随机获取一个 User-Agent，并设置到 http request header，伪装成浏览器。
    * 防止服务器限制访问，此处随机的目的是防止服务器通过 User-Agent 来判断是否为同一用户
    *
    **/
   request.setHeader("User-Agent", Constants.userAgentArray[new
Random().nextInt(Constants.userAgentArray.length)]);
   /**
    * 创建 HttpClientContext(HttpClient 上下文，维护 cookie 相关)
    * 由于 HttpClientContext 不是线程安全，当有大量 302 状态码的 http 请求出现时，有很大概率会
      抛出异常
    * 所以此处将 HttpClientContext 设置为线程独享，共同维护同一个 CookieStore 对象
    **/
   HttpClientContext httpClientContext = HttpClientContext.create();
   // 设置 Cookie
   httpClientContext.setCookieStore(cookieStore);
   // 携带 http 上下文执行 http 请求，并获得 CloseableHttpResponse 响应对象
```

```
    CloseableHttpResponse response = httpClient.execute(request, httpClientContext);
    //返回 response 对象
    return response;
}
/**
 * 执行 http post 请求
 * @param postUrl 请求地址
 * @param params 请求参数，键值对
 * @return
 * @throws IOException
 */
public static String postRequest(String postUrl, Map<String, String> params) throws
        IOException {
    /**
     * 创建一个 httppost 请求对象
     */
    HttpPost post = new HttpPost(postUrl);
    //设置 post 请求参数
    setHttpPostParams(post, params);
    //根据 request 对象获取响应内容，响应编码为 utf-8
    return getWebPage(post, "utf-8");
}
/**
 * 设置 request 请求参数
 * @param request http post 对象
 * @param params post 请求参数
 */
public static void setHttpPostParams(HttpPost request,Map<String,String> params){
    //创建一个 NameValuePair http 表单键值对参数集合
    List<NameValuePair> formParams = new ArrayList<NameValuePair>();
    //遍历 map，根据其参数创建 NameValuePair 对象，并添加至 formParms list 集合中
    for (String key : params.keySet()) {
        formParams.add(new BasicNameValuePair(key,params.get(key)));
    }
    UrlEncodedFormEntity entity = null;
    try {
    //对参数进行 url 编码
        entity = new UrlEncodedFormEntity(formParams, "utf-8");
    } catch (UnsupportedEncodingException e) {
        e.printStackTrace();
    }
    //把表单参数添加至 request 请求对象
    request.setEntity(entity);
}
```

1.6.2 数据库连接

本系统提供了持久化知乎用户数据的功能，此功能涉及 JDBC 相关技术。本功能的实现文件是 ConnectionManager.java，其功能是创建数据库新连接、获取数据库连接和关闭连接。文件 ConnectionManager.java 的主要实现代码如下。

```
/**
 * 创建一个新数据库 connection 并返回
 * @return 数据库 connection
 */
public static Connection createConnection(){
   //获取配置文件中的数据库 host 属性
   String host = Config.dbHost;
   //获取配置文件中的数据库登录用户名
   String user = Config.dbUsername;
   //获取配置文件中的数据库登录密码
   String password = Config.dbPassword;
   //获取配置文件中的数据库名
   String dbName = Config.dbName;
   //生成连接 url
   String url="jdbc:mysql://" + host + ":3306/" + dbName + "?characterEncoding=utf8";
   Connection con=null;
   try{
      //创建连接
      con = DriverManager.getConnection(url,user,password);
      logger.debug("success!");
   } catch(MySQLSyntaxErrorException e){
      logger.error("数据库不存在..请先手动创建数据库:" + dbName);
      e.printStackTrace();
   } catch(SQLException e2){
      logger.error("SQLException",e2);
   }
   return con;
}

/**
 * 返回当前 ConnectionManager 中的数据库连接，没有则创建新连接并返回
 * @return 数据库连接
 */
public static Connection getConnection(){
   try {
      if(conn == null || conn.isClosed()){
         conn = createConnection();
      } else{
         return conn;
```

```
        }
    } catch (SQLException e) {
        logger.error("SQLException",e);
    }
    return conn;
}

/**
 * 关闭数据库连接
 */
public static void close(){
    if(conn != null){
        try {
            conn.close();
        } catch (SQLException e) {
            logger.error("SQLException",e);
        }
    }
}
```

1.6.3 数据库 dao 操作

上一小节讲到了 connection 数据库连接功能，connection 连接主要是为 dao(data access object)层服务的，下面开始介绍该系统所有数据库相关操作(增、删、改、查)功能的实现。本功能的实现文件是 ZhiHuDaoImp.java，其功能是初始化数据库表。文件 ZhiHuDaoImp.java 的具体实现过程如下。

(1) 初始化表功能，根据配置的数据库连接参数连接数据库，然后查询当前库是否已经创建表，如果不存在表则初始化 table。对应代码如下。

```
/**
 * 初始化 table
 */
public static void DBTablesInit() {
    ResultSet rs = null;
    //创建 properties 文件对象
    Properties p = new Properties();
    //获取一个数据库连接
    Connection cn = ConnectionManager.getConnection();
    try {
        //加载 config.properties 文件
        p.load(ZhiHuDaoImp.class.getResourceAsStream("/config.properties"));
        //查询 url table
        rs = cn.getMetaData().getTables(null, null, "url", null);
        //创建数据库 Statement 对象
```

```
        Statement st = cn.createStatement();
        if(!rs.next()){
            //不存在 url 表，创建 url 表
            st.execute(p.getProperty("createUrlTable"));
            logger.info("url 表创建成功");
        }
        else{
            logger.info("url 表已存在");
        }
        //查询 user 表
        rs = cn.getMetaData().getTables(null, null, "user", null);
        if(!rs.next()){
            //不存在 user 表，创建 user 表
            st.execute(p.getProperty("createUserTable"));
            logger.info("user 表创建成功");
        }
        else{
            logger.info("user 表已存在");
        }
        //关闭数据库结果集
        rs.close();
        //关闭数据库操作对象
        st.close();
        //关闭连接
        cn.close();
    } catch (SQLException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

(2) 实现插入用户功能。此处需要先判断数据库中是否存在该用户，如果存在，就不执行 insert 操作并返回 false。如果不存在，则执行 insert 操作，然后返回 true。对应代码如下。

```
/**
 * insert user 资料至数据库
 * @param cn 数据库连接
 * @param u 用户对象
 * @return insert 操作结果
 */
@Override
public boolean insertUser(Connection cn, User u) {
    try {
        //判断数据库是否存在该用户，存在则直接返回 false
        if (isExistUser(cn, u.getUserToken())){
            return false;
```

```
        }
        //创建 insert sql 语句
        String column = "location,business,sex,employment,username,url,agrees,
                thanks,asks," +
                "answers,posts,followees,followers,hashId,education,user_token";
        String values = "?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?";
        String sql = "insert into user (" + column + ") values(" +values+")";
        PreparedStatement pstmt;
        //根据数据库连接创建 PreparedStatement 对象
        pstmt = cn.preparedStatement(sql);
        //设置用户所在位置
        pstmt.setString(1,u.getLocation());
        //设置用户所在行业
        pstmt.setString(2,u.getBusiness());
        //设置用户性别
        pstmt.setString(3,u.getSex());
        //设置用户所在企业
        pstmt.setString(4,u.getEmployment());
        //设置用户名
        pstmt.setString(5,u.getUsername());
        //知乎主页 url
        pstmt.setString(6,u.getUrl());
        //获得的赞同数
        pstmt.setInt(7,u.getAgrees());
        //获得的感谢数
        pstmt.setInt(8,u.getThanks());
        //获得的提问数
        pstmt.setInt(9,u.getAsks());
        //获得的回答问题数
        pstmt.setInt(10,u.getAnswers());
        //获得的文章数
        pstmt.setInt(11,u.getPosts());
        //获得的粉丝数
        pstmt.setInt(12,u.getFollowees());
        //获得的关注数
        pstmt.setInt(13,u.getFollowers());
        //hashId，用户唯一标识
        pstmt.setString(14,u.getHashId());
        //教育
        pstmt.setString(15,u.getEducation());
        //用户令牌，用户唯一标识
        pstmt.setString(16,u.getUserToken());
        pstmt.executeUpdate();
        pstmt.close();
        logger.info("插入数据库成功---" + u.getUsername());
    } catch (SQLException e) {
        e.printStackTrace();
```

```
    } finally {
    }
    return true;
}
```

(3) 实现查询功能。根据 userToken 查询数据库中是否已爬取过该用户，如果数据库中存在该用户就返回 true，否则返回 false。对应代码如下。

```
/**
 * 根据 userToken 查询数据库是否存在该用户
 * @param userToken 用户令牌，唯一标识
 * @return 查询结果
 */
@Override
public boolean isExistUser(String userToken) {
    //调用本类中的方法
    return isExistUser(ConnectionManager.getConnection(), userToken);
}

/**
 * 根据 userToken 查询数据库是否存在该用户
 * @param cn 数据库连接
 * @param userToken 用户令牌，唯一标识
 * @return
 */
@Override
public boolean isExistUser(Connection cn, String userToken) {
    //查询 sql
    String isContainSql = "select count(*) from user WHERE user_token='" + userToken + "'";
    try {
        if(isExistRecord(isContainSql)){
            return true;
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return false;
}

/**
 * 根据查询语句，判断是否存在查询结果
 * @param sql 查询语句
 * @return
 * @throws SQLException
 */
@Override
public boolean isExistRecord(String sql) throws SQLException{
```

```
    //调用本类中的方法
    return isExistRecord(ConnectionManager.getConnection(), sql);
}

/**
 * 根据查询语句，判断是否存在查询结果
 * @param cn 数据库连接
 * @param sql 查询语句
 * @return
 * @throws SQLException
 */
@Override
public boolean isExistRecord(Connection cn, String sql) throws SQLException {
    int num = 0;
    PreparedStatement pstmt;
    //创建 PreparedStatement 对象
    pstmt = cn.prepareStatement(sql);
    //执行查询，并返回结果
    ResultSet rs = pstmt.executeQuery();
    while(rs.next()){
        num = rs.getInt("count(*)");
    }
    //关闭 ResultSet 对象
    rs.close();
    //关闭 PreparedStatement 对象
    pstmt.close();
    if(num == 0){
        return false;
    }else{
        return true;
    }
}
```

1.6.4 实现相关实体类

一般来说，实体类对应数据库表中的一条记录，也就是说，数据库表的一条记录是一个实体对象，而某行的一列就表示一个实体对象的一个属性。本项目实体类的实现文件是User.java，具体代码如下。

```
/**
 * 知乎用户资料
 */
public class User {
    //用户名
    private String username;
```

```
    //user token
    private String userToken;
    //位置
    private String location;
    //行业
    private String business;
    //性别
    private String sex;
    //企业
    private String employment;
    //企业职位
    private String position;
    //教育
    private String education;
    //用户首页 url
    private String url;
    //赞同数
    private int agrees;
    //感谢数
    private int thanks;
    //提问数
    private int asks;
    //回答问题数
    private int answers;
    //文章数
    private int posts;
    //关注数
    private int followees;
    //粉丝数
    private int followers;
    // hashId，用户唯一标识
    private String hashId;

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }
     //其他普通字段的 get、set 方法同上，此处省略部分 get、set 方法
    @Override
    public String toString() {
        return "User{" +
                "username='" + username + '\'' +
                ", userToken='" + userToken + '\'' +
                ", location='" + location + '\'' +
```

```
                ", business='" + business + '\'' +
                ", sex='" + sex + '\'' +
                ", employment='" + employment + '\'' +
                ", position='" + position + '\'' +
                ", education='" + education + '\'' +
                ", url='" + url + '\'' +
                ", agrees=" + agrees +
                ", thanks=" + thanks +
                ", asks=" + asks +
                ", answers=" + answers +
                ", posts=" + posts +
                ", followees=" + followees +
                ", followers=" + followers +
                ", hashId='" + hashId + '\'' +
                '}';
    }
}
```

1.7 数据爬取模块

数据爬取模块是整个项目的核心模块之一，主要包含的功能有整个爬虫的初始化、知乎页面的下载、知乎页面数据的解析、整个爬虫的运行流程控制、爬取异常处理、重试机制等。

扫码看视频

1.7.1 爬虫爬取初始化

1) authorization 字段的初始化

通过浏览器对知乎网站进行抓包后可知，抓取页面的请求不是普通的 GET 请求，而是要携带额外的 HTTP 验证头，即当通过爬虫方式来实现爬取的时候，需要每次携带该验证头才能请求成功。通过详细分析抓包可知，authorization 字段在 JavaScripts 脚本文件中，而 JavaScripts 的地址又需要通过知乎用户关注页面才能拿到，所以在初始化 authorization 字段时需要如下两个步骤。

(1) 请求并下载关注页面，解析出 authorization 所在 JavaScripts 文件的 URL。

(2) 请求下载该 URL 的数据，最终解析出 authorization 字段。

上述功能的实现文件是 ZhiHuHttpClient.java，对应代码如下。

```
/**
 * 初始化authorization
 * @return
 */
```

```
private String initAuthorization(){
    logger.info("初始化 authorization 中...");
    String content = null;
    //创建一个页面下载任务
    GeneralPageTask generalPageTask = new GeneralPageTask(Config.startURL, true);
    //执行下载任务，直接调用 run 方法
    generalPageTask.run();
    //获取下载成功的网页内容
    content = generalPageTask.getPage().getHtml();
    //创建一个正则表达式，获取 authorization 所在 js 文件地址的 url
    Pattern pattern = Pattern.compile("https://static\\.zhihu\\.com/heifetz/
                      main\\.app\\.([0-9]|[a-z])*\\.js");
    Matcher matcher = pattern.matcher(content);
    String jsSrc = null;
    if (matcher.find()){
        //解析出 js 文件的 url
        jsSrc = matcher.group(0);
    } else {
        throw new RuntimeException("not find javascript url");
    }
    String jsContent = null;
    //创建一个页面下载任务，地址为刚刚解析出的 js 文件 url
    GeneralPageTask jsPageTask = new GeneralPageTask(jsSrc, true);
    jsPageTask.run();
    //获取下载成功的 js 文件内容
    jsContent = jsPageTask.getPage().getHtml();
    //创建一个正则表达式，解析出 authorization 字段值
    pattern = Pattern.compile("oauth\\\"\\),h=\\\"(([0-9]|[a-z])*)\"");
    matcher = pattern.matcher(jsContent);
    if (matcher.find()){
        //获取 authorization 字段成功
        String authorization = matcher.group(1);
        logger.info("初始化 authorization 完成");
        return authorization;
    }
    throw new RuntimeException("not get authorization");
}
```

2) 列表详情页线程池初始化

本项目采用多线程的方式来提高爬取速度，也就是将每一个 URL 抽象成一个具体线程任务，而这个任务又在短时间内结束其生命周期。如果每执行一个任务就创建一个线程，当任务数量达到百万级别的时候，在线程创建上的开销是很大的，所以这里采用线程池模型来执行任务。线程池适用于执行那些任务多而耗时短的操作。它的一个基本原理就是：创建指定数量的线程后，当有新的任务到来时，直接从线程池中获取线程来执行任务，这样就减少了频繁创建线程的开销。线程池的主要初始化代码见 ZhiHuHttpClient 文件，对应

代码如下。

```
/**
 * 初始化线程池
 */
private void intiThreadPool(){
    /**
     * 创建一个 corePoolSize 为 100，maxmumPoolSize 为 100，任务队列长度为 2000 的线程池，
     * 用于执行知乎详情列表页下载解析任务，其中 poolSize 可以通过配置文件修改。
     * 在线程池中，当线程数量达到设定的最大值(maxmumPoolSize 为 100)且任务队列长度达到设定的
     * 最大值(2000)时，如果此时有新的任务要添加到线程池中，这些新任务将被直接丢弃，而不会被执行。
     * 这里调用的是 SimpleThreadPoolExecutor，继承 ThreadPoolExecutor，可以通过构造方法直
接为线程池命名。
     * 这里继承的目的是在输出日志中观察各个线程池的运行状态。
     */
    detailListPageThreadPool = new SimpleThreadPoolExecutor(Config.downloadThreadSize,
            Config.downloadThreadSize,
            0L, TimeUnit.MILLISECONDS,
            new LinkedBlockingQueue<Runnable>(2000),
            new ThreadPoolExecutor.DiscardPolicy(),
            "detailListPageThreadPool");
    //开启一个新线程，用于监视列表详情页测试线程池执行情况
    new Thread(new ThreadPoolMonitor(detailListPageThreadPool,
"detailListPageThreadPool")).start();
}
```

3) 管理 ZhiHuHttpClient

整个项目启动后，不可能无休止地运行下去。当爬取指定数量的网页后，需要平滑地关闭整个爬虫功能，方法是轮询检测 detailListPageThreadPool 线程池的执行情况，当完成指定任务数后关闭该线程池，不再接受新的任务。此外，还需要关闭线程池监视工具类。当 detailListPageThreadPool 关闭后，关闭线程池连接和 ProxyHttpClient 类的 proxyTestThreadExecutor 线程池及 proxyDownloadThreadExecutor 线程池。本功能的实现文件是 ZhiHuHttpClient.java，对应代码如下。

```
/**
     * 管理知乎 HttpClient
     * 关闭整个爬虫
     */
    public void manageHttpClient(){
        //每秒执行一次轮询，检测整个爬虫执行情况
        while (true) {
            /**
             * 下载网页数
             */
```

```
            long downloadPageCount = detailListPageThreadPool.getTaskCount();
            //下载网页数达到配置下载数，关闭 detailListPageThreadPool 线程池
            if (downloadPageCount >= Config.downloadPageCount &&
                    !detailListPageThreadPool.isShutdown()) {
                isStop = true;
                //设置 ThreadPoolMonitor，isStopMonitor 字段为 true。关闭线程池监视类
                ThreadPoolMonitor.isStopMonitor = true;
                //关闭 detailListPageThreadPool 线程池
                detailListPageThreadPool.shutdown();
            }
            //判断 detailListPageThreadPool 线程池是否关闭，完成关闭后，再关闭数据库连接
            if(detailListPageThreadPool.isTerminated()){
                //关闭数据库连接
                Map<Thread, Connection> map = DetailListPageTask.getConnectionMap();
                for(Connection cn : map.values()){
                    try {
                        if (cn != null && !cn.isClosed()){
                            cn.close();
                        }
                    } catch (SQLException e) {
                        e.printStackTrace();
                    }
                }
                //关闭代理检测线程池
                ProxyHttpClient.getInstance().getProxyTestThreadExecutor().shutdownNow();
                //关闭代理下载页线程池

ProxyHttpClient.getInstance().getProxyDownloadThreadExecutor().shutdownNow();
                break;
            }
            double costTime = (System.currentTimeMillis() - startTime) / 1000.0;//单位为s
            logger.debug("爬取速率: " + parseUserCount.get() / costTime + "个/s");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
```

4) 爬取入口

通过配置 startUserToken，可以指定从哪一个知乎用户开始爬取。需要注意的是，爬取路线是顺着用户的关注用户一直往下爬取，所以配置的 startUserToken 必须有关注用户。方法是创建一个 DetailListPageTask，并添加至 detailListPageThreadPool 线程池中执行。该功能的实现文件是 ZhiHuHttpClient.java，对应代码如下。

```
/**
 * 开始爬取
 */
@Override
public void startCrawl() {
    //调用本类中的方法，初始化 authorization 字段
    authorization = initAuthorization();
    //获取爬取入口用户 token
    String startToken = Config.startUserToken;
    //根据 token 构建请求 url
    String startUrl = String.format(Constants.USER_FOLLOWEES_URL, startToken, 0);
    //根据 url 创建一个 GET 请求
    HttpGet request = new HttpGet(startUrl);
    //设置 authorization header
    request.setHeader("authorization", "oauth " + ZhiHuHttpClient.getAuthorization());
    //创建一个 DetailListPageTask，并添加至 detailListPageThreadPool 中执行
    detailListPageThreadPool.execute(new DetailListPageTask(request, Config.isProxy));
    manageHttpClient();
}
```

1.7.2 知乎网页下载

1) 知乎 HTTP 请求抽象页任务实现

HTTP 请求是不能保证百分之百成功的，在爬取过程中需要处理各种可能的请求失败，以及是否使用代理、使用代理失败的后续处理逻辑、代理的耗时统计等。该功能的实现文件是 AbstractPageTask.java，对应代码如下。

```
/**
 * 线程任务
 */
public void run(){
    long requestStartTime = 0l;
    HttpGet tempRequest = null;
    try {
        Page page = null;
        if(url != null){
            if (proxyFlag){
                //使用代理
                tempRequest = new HttpGet(url);
                //从代理池延时队列中获取一个代理
                currentProxy = ProxyPool.proxyQueue.take();
                //判断代理是否为 Direct(直连)
                if(!(currentProxy instanceof Direct)){
                    //不是本机直接连接，创建一个 HttpHost 代理对象
```

```
                    HttpHost proxy = new HttpHost(currentProxy.getIp(),
                                        currentProxy.getPort());
                    //设置代理
                    tempRequest.setConfig(HttpClientUtil.
                        getRequestConfigBuilder().setProxy(proxy).build());
                }
                requestStartTime = System.currentTimeMillis();
                //执行 HttpGet 请求，获取响应内容
                page = zhiHuHttpClient.getWebPage(tempRequest);
            }else {
                //不使用代理
                requestStartTime = System.currentTimeMillis();
                page = zhiHuHttpClient.getWebPage(url);
            }
        } else if(request != null){
            if (proxyFlag){
                //使用代理，从代理池延时队列中获取一个代理
                currentProxy = ProxyPool.proxyQueue.take();
                //判断代理是否为 Direct(直连)
                if(!(currentProxy instanceof Direct)) {
                    //不是本机直接连接，创建一个 HttpHost 代理对象
                    HttpHost proxy = new HttpHost(currentProxy.getIp(),
                                        currentProxy.getPort());
                    //设置代理
                    request.setConfig(HttpClientUtil.
                        getRequestConfigBuilder().setProxy(proxy).build());
                }
                requestStartTime = System.currentTimeMillis();
                //执行请求，获取响应内容
                page = zhiHuHttpClient.getWebPage(request);
            }else {
                //直接下载
                requestStartTime = System.currentTimeMillis();
                page = zhiHuHttpClient.getWebPage(request);
            }
        }
        long requestEndTime = System.currentTimeMillis();
        page.setProxy(currentProxy);
        //获取响应状态码
        int status = page.getStatusCode();
        //拼接日志
        String logStr = Thread.currentThread().getName() + " " + currentProxy +
                " executing request " + page.getUrl() + " response statusCode:"
                + status + " request cost time:" +
                  (requestEndTime - requestStartTime) + "ms";
        if(status == HttpStatus.SC_OK){
            /**
```

```
                * 返回 SC_OK 状态不一定表示响应成功，由于部分异常代理，不会返回目标请求 url
                  的内容。所以此处需要二次判断
                */
               if (page.getHtml().contains("zhihu")
                   && !page.getHtml().contains("安全验证")){
                   logger.debug(logStr);
                   //代理请求次数+1
                   currentProxy.setSuccessfulTimes
                                (currentProxy.getSuccessfulTimes() + 1);
                   //记录代理总共请求耗时
                   currentProxy.setSuccessfulTotalTime
                        (currentProxy.getSuccessfulTotalTime() +
                        (requestEndTime - requestStartTime));
                   //计算成功请求平均耗时
                   double aTime = (currentProxy.getSuccessfulTotalTime() + 0.0) /
                                   currentProxy.getSuccessfulTimes();
                   currentProxy.setSuccessfulAverageTime(aTime);
                   currentProxy.setLastSuccessfulTime(System.currentTimeMillis());
                   //处理响应成功网页，具体处理由子类实现
                   handle(page);
               }else {
                   /**
                    * 代理异常，没有正确返回目标 url
                    */
                   logger.warn("proxy exception:" + currentProxy.toString());
               }
           }
           /**
            * 401--不能通过验证
            */
           else if(status == 404 || status == 401 ||
                   status == 410){
               logger.warn(logStr);
           }
           else {
               logger.error(logStr);
               Thread.sleep(100);
               retry();
           }
       } catch (InterruptedException e) {
           logger.error("InterruptedException", e);
       } catch (IOException e) {
           //请求异常
        if(currentProxy != null){
           /**
            * 该代理可用，将该代理继续添加到 proxyQueue
            */
```

```
            currentProxy.setFailureTimes(currentProxy.getFailureTimes() + 1);
        }
        if(!zhiHuHttpClient.getDetailListPageThreadPool().isShutdown()){
            //重试，具体重试方法由子类实现
                retry();
            }
        } finally {
            if (request != null){
                //释放连接
                request.releaseConnection();
            }
            if (tempRequest != null){
                //释放连接
                tempRequest.releaseConnection();
            }
            if (currentProxy != null && !ProxyUtil.isDiscardProxy(currentProxy)){
                //代理过滤，失败次数达到一定条件，丢弃代理
                currentProxy.setTimeInterval(Constants.TIME_INTERVAL);
                ProxyPool.proxyQueue.add(currentProxy);
            }
        }
    }
```

2) 知乎详情列表页任务功能

该功能是对下载成功的用户详情列表页进行后续处理，解析出用户资料并入库，包括对 URL 的去重处理、构造待爬取的 DetailListPageTask 任务。本功能的难点是 DetailListPageTask 执行任务时，又构造新的 DetailListPageTask 添加至线程池中。注意，当爬取一定数量的用户后，某一次任务爬取的用户全是已经爬取过的用户，这时就没有新的任务添加至线程池了，从而导致线程池一直处于空任务的状态。该功能的实现文件是 DetailListPageTask.java，对应代码如下。

```
/**
 * 对下载成功的知乎用户列表详情页进行后续处理
 * @param page 网页
 */
@Override
void handle(Page page) {
    if(!page.getHtml().startsWith("{\"paging\"")){
        //代理异常，未能正确返回目标请求数据，丢弃
        currentProxy = null;
        return;
    }
    //从下载成功的详情列表页中解析出用户
    List<User> list = proxyUserListPageParser.parseListPage(page);
    for(User u : list){
```

```
        logger.info("解析用户成功:" + u.toString());
        if(Config.dbEnable){
            //数据库可用，获取数据库 connection
            Connection cn = getConnection();
            if (zhiHuDao.insertUser(cn, u)){
                //insert user
                parseUserCount.incrementAndGet();
            }
            //根据解析出的 user 信息，获取当前用户关注的用户数
            for (int j = 0; j < u.getFollowees() / 20; j++){
                if (zhiHuHttpClient.getDetailListPageThreadPool().getQueue().size()
                                        > 1000){
                    continue;
                }
                //构造获取当前用户所关注用户的 url
                String nextUrl = String.format(USER_FOLLOWEES_URL, u.getUserToken(),
                                j * 20);
                /**
                * 在这里，我们对生成的 URL 进行 MD5 摘要计算，并将其插入数据库，以便进行去重操作。
                * 如果当前 URL 尚未被访问过，或者 detailListPageThreadPool 的 activeCount
                  为 1 时，我们执行以下操作。
                * 设置这个条件的目的是避免在爬取数量达到一定量后，在某个用户所关注的用户已经全
                  部爬取完毕的情况下，防止线程池任务一直处于等待状态而导致爬取停止。
                */
                if (zhiHuDao.insertUrl(cn, Md5Util.Convert2Md5(nextUrl)) ||
                        zhiHuHttpClient.getDetailListPageThreadPool().
                                getActiveCount() == 1){
                    //根据生成的 url 构造 HttpGet 对象
                    HttpGet request = new HttpGet(nextUrl);
                    //设置 authorization 验证 header
                    request.setHeader("authorization", "oauth " +
                                        ZhiHuHttpClient.getAuthorization());
                    //创建 DetailListPageTask，并添加至 detailListPageThreadPool 中
                    zhiHuHttpClient.getDetailListPageThreadPool().execute
                                    (new DetailListPageTask(request, true));
                }
            }
        }
        else if(!Config.dbEnable || zhiHuHttpClient.
                getDetailListPageThreadPool().getActiveCount() == 1){
            //不使用数据库，则不做去重处理
            parseUserCount.incrementAndGet();
            for (int j = 0; j < u.getFollowees() / 20; j++){
                //构造 nextUrl
                String nextUrl = String.format(USER_FOLLOWEES_URL,
                                u.getUserToken(), j * 20);
                //根据 url 创建 HttpGet
```

```
                HttpGet request = new HttpGet(nextUrl);
                //设置 authorization 验证 header
                request.setHeader("authorization", "oauth " +
ZhiHuHttpClient.getAuthorization());
                //创建 DetailListPageTask，并添加至 detailListPageThreadPool 中
                zhiHuHttpClient.getDetailListPageThreadPool().execute(new
DetailListPageTask(request, true));
            }
        }
    }
}
```

1.7.3 解析知乎详情列表页

本项目中的爬虫模块抓取的页面并不是普通的 HTML 标签文档，而是 json 格式的数据文档，所以使用 jsonPath 库来解析数据。在实现本功能时需要注意，为了兼容以前的代码，知乎服务器所返回的数据字段与本地 user 对象字段名大多不一样，所以这里采用反射的方式直接将值注入对象中。解析知乎详情列表页功能的实现文件是 ZhiHuUserListPageParser.java，对应代码如下。

```
/**
 * 根据网页对象，解析出用户资料列表
 * @param page
 * @return 用户资料列表
 */
@Override
public List<User> parseListPage(Page page) {
    List<User> userList = new ArrayList<>();
    String baseJsonPath = "$.data.length()";
    DocumentContext dc = JsonPath.parse(page.getHtml());
    Integer userCount = dc.read(baseJsonPath);
    for (int i = 0; i < userCount; i++){
        User user = new User();
        String userBaseJsonPath = "$.data[" + i + "]";
        //userToken
        setUserInfoByJsonPath(user, "userToken", dc, userBaseJsonPath +
                              ".url_token");
        //username
        setUserInfoByJsonPath(user, "username", dc, userBaseJsonPath + ".name");
        //hashId
        setUserInfoByJsonPath(user, "hashId", dc, userBaseJsonPath + ".id");
        //关注人数
        setUserInfoByJsonPath(user, "followees", dc, userBaseJsonPath +
                              ".following_count");
```

```
            //位置
            setUserInfoByJsonPath(user, "location", dc, userBaseJsonPath +
                                  ".locations[0].name");
            //行业
            setUserInfoByJsonPath(user, "business", dc, userBaseJsonPath +
                                  ".business.name");
            //公司
            setUserInfoByJsonPath(user, "employment", dc, userBaseJsonPath +
                                  ".employments[0].company.name");
            //职位
            setUserInfoByJsonPath(user, "position", dc, userBaseJsonPath +
                                  ".employments[0].job.name");
            //教育
            setUserInfoByJsonPath(user, "education", dc, userBaseJsonPath +
                                  ".educations[0].school.name");
            //回答数
            setUserInfoByJsonPath(user, "answers", dc, userBaseJsonPath +
                                  ".answer_count");
            //提问数
            setUserInfoByJsonPath(user, "asks", dc, userBaseJsonPath + ".question_count");
            //文章数
            setUserInfoByJsonPath(user, "posts", dc, userBaseJsonPath +
                                  ".articles_count");
            //粉丝数
            setUserInfoByJsonPath(user, "followers", dc, userBaseJsonPath +
                                  ".follower_count");
            //赞同数
            setUserInfoByJsonPath(user, "agrees", dc, userBaseJsonPath +
                                  ".voteup_count");
            //感谢数
            setUserInfoByJsonPath(user, "thanks", dc, userBaseJsonPath +
                                  ".thanked_count");
            try {
                //性别
                Integer gender = dc.read(userBaseJsonPath + ".gender");
                if (gender != null && gender == 1){
                    user.setSex("male");
                }
                else if(gender != null && gender == 0){
                    user.setSex("female");
                }
            } catch (PathNotFoundException e){
                //没有该属性
            }
            userList.add(user);
        }
        return userList;
```

```
    }
    /**
     * jsonPath 获取值，并通过反射直接注入 user 中
     * @param user user 对象
     * @param fieldName user 对象中的字段名
     * @param dc 文档上下文
     * @param jsonPath jsonPath 表达式
     */
    private void setUserInfoByJsonPath(User user, String fieldName,
                                       DocumentContext dc , String jsonPath){
        try {
            //根据 jsonPath 表达式获取对应的值
            Object o = dc.read(jsonPath);
            //根据 field 字段名获取对象的 Field 对象
            Field field = user.getClass().getDeclaredField(fieldName);
            //设置为可被访问
            field.setAccessible(true);
            //设置 user 对象的 field 字段的值
            field.set(user, o);
        } catch (PathNotFoundException e1) {
            //no results
        } catch (Exception e){
            e.printStackTrace();
        }
    }
```

1.8 代理功能模块

代理功能模块是为知乎爬取模块服务的，突破同一客户端访问知乎服务器的并发连接限制，以此提高整个项目的爬取速度。代理功能模块主要包括的功能有代理页面的下载、代理页面的解析、代理的测试。代理打分丢弃是指在代理池中对每个代理进行评分，并根据得分来决定保留或丢弃该代理。

扫码看视频

1.8.1 代理功能模块初始化

为了提高代理的可重用性，同一个 Proxy 的请求速率会被限制，而该类正是通过 JDK 中的 DelayQueue(延时队列)来实现这一功能的。DelayQueue 的元素必须实现 Delayed 接口。在此需要注意的是，这个类中增加了一些额外的属性，用于统计代理的一些请求信息，便于对代理进行打分。定义 Proxy 代理类的实现文件是 Proxy.java，对应代码如下。

```
/**
 * http 代理实体
 * 实现 Delayed 接口，作为 DelayQueue 的元素
 */
public class Proxy implements Delayed, Serializable{
    private static final long serialVersionUID = -7583883432417635332L;
    //使用该代理的最小间隔时间，单位为 ms
    private long timeInterval;
    //代理 ip 地址
    private String ip;
    //代理端口
    private int port;
    //该代理是否可用
    private boolean availableFlag;
    //是否匿名
    private boolean anonymousFlag;
    //最近一次请求成功时间
    private long lastSuccessfulTime;
    //请求成功总耗时
    private long successfulTotalTime;
    //请求失败次数
    private int failureTimes;
    //请求成功次数
    private int successfulTimes;
    //请求成功平均耗时
    private double successfulAverageTime;
    public Proxy(String ip, int port, long timeInterval) {
        this.ip = ip;
        this.port = port;
        this.timeInterval = timeInterval;
        this.timeInterval = TimeUnit.NANOSECONDS.convert(timeInterval,
                            TimeUnit.MILLISECONDS) + System.nanoTime();
    }
    public String getIp() {
        return ip;
    }

    public void setIp(String ip) {
        this.ip = ip;
    }
    //其他普通字段的 get、set 方法同上，此处省略部分 get、set 方法

    @Override
    public int compareTo(Delayed o) {
        Proxy element = (Proxy)o;
        if (successfulAverageTime == 0.0d ||element.successfulAverageTime == 0.0d){
            return 0;
```

```
        }
        return successfulAverageTime > element.successfulAverageTime ?
                1:(successfulAverageTime < element.successfulAverageTime ? -1 : 0);
    }

    @Override
    public String toString() {
        return "Proxy{" +
                "timeInterval=" + timeInterval +
                ", ip='" + ip + '\'' +
                ", port=" + port +
                ", availableFlag=" + availableFlag +
                ", anonymousFlag=" + anonymousFlag +
                ", lastSuccessfulTime=" + lastSuccessfulTime +
                ", successfulTotalTime=" + successfulTotalTime +
                ", failureTimes=" + failureTimes +
                ", successfulTimes=" + successfulTimes +
                ", successfulAverageTime=" + successfulAverageTime +
                '}';
    }
    //此处重写 equals，如果 ip 地址和 port 相同，则表示是同一个代理
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Proxy proxy = (Proxy) o;
        if (port != proxy.port) return false;
        return ip.equals(proxy.ip);

    }
    @Override
    public int hashCode() {
        int result = ip.hashCode();
        result = 31 * result + port;
        return result;
    }
    public String getProxyStr(){
        return ip + ":" + port;
    }
}
```

1.8.2 代理初始化

爬虫初始化获取有用代理是一个比较耗时的过程，经常会导致前期爬取速度非常慢。为了解决这个问题，需要让爬虫快速启动，并且能很快地爬取。在爬取数据过程中，每隔

一段时间把代理序列化至文件，然后每次启动文件再将代理反序列化至内存中。如果代理在最近一小时内使用过，则直接使用。在此需要注意的是，为什么选择一小时作为超时时间？因为目前网上公开的免费代理大多都有一个特点：使用的时效特别短，在公开一小时后还能使用的代理非常少。代理初始化功能的实现文件是 ProxyHttpClient.java，对应代码如下。

```
/**
 * 初始化 proxy
 */
private void initProxy(){
    Proxy[] proxyArray = null;
    try {
        //反序列化代理文件
        proxyArray = (Proxy[]) HttpClientUtil.deserializeObject(Config.proxyPath);
        int usableProxyCount = 0;
        for (Proxy p : proxyArray){
            if (p == null){
                continue;
            }
            //设置
            p.setTimeInterval(Constants.TIME_INTERVAL);
            p.setFailureTimes(0);
            p.setSuccessfulTimes(0);
            long nowTime = System.currentTimeMillis();
            if (nowTime - p.getLastSuccessfulTime() < 1000 * 60 *60){
                //上次成功离现在少于一小时
                ProxyPool.proxyQueue.add(p);
                ProxyPool.proxySet.add(p);
                usableProxyCount++;
            }
        }
        logger.info("反序列化 proxy 成功，" + proxyArray.length + "个代理,可用代理"
                    + usableProxyCount + "个");
    } catch (Exception e) {
        logger.warn("反序列化 proxy 失败");
    }
}
```

1.8.3 代理页下载线程池和代理测试线程池初始化

在本项目中，代理功能模块主要负责两种类型的任务。

第一种：根据 ProxyPool.java 文件中配置的 URL 去下载并解析代理网页任务。

第二种：检测解析出的 Proxy 的可用性。

本项目是通过创建 proxyTestThreadPool 和 proxyDownloadThreadPoll 两个线程池来分别执行这两种类型的任务。线程池的主要初始化代码见 ZhiHuHttpClient 文件，对应代码如下。

```
/**
 * 初始化线程池
 */
private void initThreadPool(){
    //创建一个 corePoolSize 为 100，maxnumPoolSize 为 100，任务队列长度为 10000 的线程池，
    //用于执行代理测试任务
    proxyTestThreadExecutor = new SimpleThreadPoolExecutor(100, 100,
            0L, TimeUnit.MILLISECONDS,
            new LinkedBlockingQueue<Runnable>(10000),
            new ThreadPoolExecutor.DiscardPolicy(),
            "proxyTestThreadExecutor");
    //创建一个 corePoolSize 为 10，maximumPoolSize 为 10，任务队列长度为
    //nteger.MAX_VALUE 的线程池，用于执行代理页面下载任务
    proxyDownloadThreadExecutor = new SimpleThreadPoolExecutor(10, 10,
            0L, TimeUnit.MILLISECONDS,
            new LinkedBlockingQueue<Runnable>(), "" +
            "proxyDownloadThreadExecutor");
    //开启一个新线程用于监视代理测试线程池执行情况
    new Thread(new ThreadPoolMonitor(proxyTestThreadExecutor,
                "ProxyTestThreadPool")).start();
    //开启一个新线程用于监视代理页下载线程池执行情况
    new Thread(new ThreadPoolMonitor(proxyDownloadThreadExecutor,
                "ProxyDownloadThreadExecutor")).start();
}
/**
 * 爬取代理
 */
public void startCrawl(){
    new Thread(new Runnable() {
        @Override
        public void run() {
            while (true){
                for (String url : ProxyPool.proxyMap.keySet()){
                    /**
                     * 本机首次直接下载代理页面
                     */
                    proxyDownloadThreadExecutor.execute(new ProxyPageTask(url, false));
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                try {
```

```
                    Thread.sleep(1000 * 60 * 60);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }).start();
    new Thread(new ProxySerializeTask()).start();
}
public ThreadPoolExecutor getProxyTestThreadExecutor() {
    return proxyTestThreadExecutor;
}

public ThreadPoolExecutor getProxyDownloadThreadExecutor() {
    return proxyDownloadThreadExecutor;
}
}
```

1.8.4 代理爬取入口

根据文件 ProxyPool.java 中配置的代理页 URL，逐一构造 ProxyPageTask，并将其添加至 proxyDownloadThreadPool 中，然后创建一个代理序列化任务。代理爬取入口功能的实现文件是 ProxyHttpClient.java，对应代码如下。

```
/**
 * 爬取代理
 */
public void startCrawl(){
    //开启一个新线程
    new Thread(new Runnable() {
        @Override
        public void run() {
            while (true){
                for (String url : ProxyPool.proxyMap.keySet()){
                    /**
                     * 首次本机直接下载代理页面
                     */
                    proxyDownloadThreadPool.execute(new ProxyPageTask(url, false));
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                try {
```

```
                //每隔 1 小时重新获取代理
                Thread.sleep(1000 * 60 * 60);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}).start();
//创建代理序列化任务
new Thread(new ProxySerializeTask()).start();
}
```

1.8.5 代理页面下载

根据文件 ProxyPool.java 中的代理页 URL 下载代理页面，下载成功后解析出代理。根据代理构造 ProxyTestTask，并添加至 proxyTestThreadPooll 中。下载失败的代理页面构造新的 ProxyPageTask，通过代理重新下载，直至下载成功。本功能的实现文件是 ProxyPageTask.java，对应代码如下。

```
public void run(){
    //获取当前时间戳，单位为 ms
    long requestStartTime = System.currentTimeMillis();
    HttpGet tempRequest = null;
    try {
        Page page = null;
        if (proxyFlag){
            //使用代理下载，创建 HttpGet 请求对象
            tempRequest = new HttpGet(url);
            //从延时队列获取一个代理
            currentProxy = proxyQueue.take();
            //判断是否为本机直接连接
            if(!(currentProxy instanceof Direct)){
                //不是直接连接，创建一个 HttpHost 代理对象
                HttpHost proxy = new HttpHost(currentProxy.getIp(),
                                    currentProxy.getPort());
                //设置代理至创建的请求
                tempRequest.setConfig(HttpClientUtil.
                        getRequestConfigBuilder().setProxy(proxy).build());
            }
            //执行请求，获取网页内容
            page = proxyHttpClient.getWebPage(tempRequest);
        }else {
            //不使用代理，直接下载
            page = proxyHttpClient.getWebPage(url);
        }
```

```
            page.setProxy(currentProxy);
            //获取响应状态码
            int status = page.getStatusCode();
            //获取当前时间戳，单位为ms，用于统计请求耗时
            long requestEndTime = System.currentTimeMillis();
            String logStr = Thread.currentThread().getName() + " "
                        + getProxyStr(currentProxy) + "  executing request " +
                        page.getUrl()  + " response statusCode:" + status +
                    " request cost time:" + (requestEndTime - requestStartTime) + "ms";
            if(status == HttpStatus.SC_OK){
                //获取代理页成功
                logger.debug(logStr);
                handle(page);
            } else {
                //获取代理页失败
                logger.error(logStr);
                Thread.sleep(100);
                //重试
                retry();
            }
        } catch (InterruptedException e) {
            logger.error("InterruptedException", e);
        } catch (IOException e) {
            retry();
        } finally {
            if(currentProxy != null){
                currentProxy.setTimeInterval(Constants.TIME_INTERVAL);
                proxyQueue.add(currentProxy);
            }
            if (tempRequest != null){
                //释放连接
                tempRequest.releaseConnection();
            }
        }
    }
    public void retry(){
        //创建ProxyPageTask任务，通过代理下载
        proxyHttpClient.getProxyDownloadThreadPool().execute
                        (new ProxyPageTask(url, true));
    }
    /**
     * 处理下载成功的代理页
     * @param page
     */
    public void handle(Page page){
        if (page.getHtml() == null || page.getHtml().equals("")){
            return;
```

```
        }
        //根据 url 获取代理页面解析器
        ProxyListPageParser parser = ProxyListPageParserFactory.
                getProxyListPageParser(ProxyPool.proxyMap.get(url));
        //解析出 proxy list
        List<Proxy> proxyList = parser.parse(page.getHtml());
        for(Proxy p : proxyList){
            if(!ZhiHuHttpClient.getIrstance().getDetailListPageThreadPool().
               isTerminated()){
                //获取 ProxyPool 读锁
                ProxyPool.lock.readLock().lock();
                //判断当前代理是否已被添加过
                boolean containFlag = ProxyPool.proxySet.contains(p);
                //释放 ProxyPool 读锁
                ProxyPool.lock.readLock().unlock();
                if (!containFlag){
                    //未被添加过，获取响应写锁，添加代理至 proxySet
                    ProxyPool.lock.writeLock().lock();
                    ProxyPool.proxySet.add(p);
                    //释放写锁
                    ProxyPool.lock.writeLock().unlock();
                    //创建一个 ProxyTestTask(代理测试任务)，并添加至 proxyTest 线程池
                    proxyHttpClient.getProxyTestThreadPool().execute
                        (new ProxyTestTask(p));
                }
            }
        }
    }
```

1.8.6 代理页面解析

代理页面和知乎详情列表的网页内容不太一致，目前所爬取的 4 个代理网页的内容是 HTML 标签文档，所以并没有采用 jsonPath 解析功能，而是采用 Jsoup 库进行解析，Jsoup 在解析 HTML 文档时非常灵活方便。代理页面解析功能的具体代码详见 com.crawl.proxy.site 包，实现过程如下所示。

(1) 登录代理网站 http://www.66ip.cn/，代理解析类的实现文件是 Ip66ProxyListPageParser.java，对应代码如下。

```
/**
 * http://www.66ip.cn/
 */
public class Ip66ProxyListPageParser implements ProxyListPageParser {
    /**
```

```
     * 根据 http://www.66ip.cn/网页内容解析出 proxy list
     * @param content 网页内容
     * @return proxy list
     */
    @Override
    public List<Proxy> parse(String content) {
        List<Proxy> proxyList = new ArrayList<>();
        if (content == null || content.equals("")){
            return proxyList;
        }
        //根据网页内容创建 Document 对象
        Document document = Jsoup.parse(content);
        //查找标签 table 的子标签 tr，并且 tr 索引大于 1
        Elements elements = document.select("table tr:gt(1)");
        for (Element element : elements){
            //第一个 td 标签，text 为 ip
            String ip = element.select("td:eq(0)").first().text();
            //第二个 td 标签，text 为 port
            String port  = element.select("td:eq(1)").first().text();
            //第三个 td 标签，text 为匿名标志
            String isAnonymous = element.select("td:eq(3)").first().text();
            if(!anonymousFlag || isAnonymous.contains("匿")){
                //只添加匿名代理至 proxyList 中
                proxyList.add(new Proxy(ip, Integer.valueOf(port), TIME_INTERVAL));
            }
        }
        return proxyList;
    }
}
```

(2) 登录网站 http://www.ip181.com/，该代理解析类的实现文件是 Ip181ProxyListPageParser.java，对应代码如下。

```
/**
 * http://www.ip181.com/
 */
public class Ip181ProxyListPageParser implements ProxyListPageParser {
    /**
     * 根据 http://www.ip181.com/网页内容解析出 proxy list
     * @param content 网页内容
     * @return proxy list
     */
    @Override
    public List<Proxy> parse(String content) {
        //根据网页内容创建 Document 对象
        Document document = Jsoup.parse(content);
        //获取 table 标签下索引大于 0 的 tr 标签
```

```
        Elements elements = document.select("table tr:gt(0)");
        List<Proxy> proxyList = new ArrayList<>(elements.size());
        for (Element element : elements){
            //获取第一个 td 标签，text 为 ip
            String ip = element.select("td:eq(0)").first().text();
            //获取第二个 td 标签，text 为 port
            String port = element.select("td:eq(1)").first().text();
            //获取第三个 td 标签，text 为匿名标志
            String isAnonymous = element.select("td:eq(2)").first().text();
            if(!anonymousFlag || isAnonymous.contains("匿")){
                //添加匿名代理至 proxyList
                proxyList.add(new Proxy(ip, Integer.valueOf(port), TIME_INTERVAL));
            }
        }
        return proxyList;
    }
}
```

其他代理网站的代理原理同以上两个网站，本文不再提供具体实现方法。

1.8.7 代理可用性检测

从代理网站爬取代理，大部分代理是不可用的，需要对代理进行检测，可用的代理再拿来作为爬虫代理。检测的方式是通过访问知乎首页，看是否能返回正确的响应。该功能的实现文件是 ProxyTestTask.java，对应代码如下。

```
/**
 * 代理检测 task
 * 访问知乎首页，判断能否正确响应
 * 将可用代理添加到 DelayQueue(延时队列)中
 */
public class ProxyTestTask implements Runnable{
    private final static Logger logger =
LoggerFactory.getLogger(ProxyTestTask.class);
    private Proxy proxy;
    public ProxyTestTask(Proxy proxy){
        this.proxy = proxy;
    }

    /**
     * 多线程任务
     */
    @Override
    public void run() {
        //获取当前时间戳，单位为 ms
```

```
        long startTime = System.currentTimeMillis();
        //创建 url 为 https://www.zhihu.com 的 HttpGet 请求
        HttpGet request = new HttpGet(Constants.INDEX_URL);
        try {
            //配置 request，设置超时时间和代理
            RequestConfig requestConfig =
                    RequestConfig.custom().setSocketTimeout(Constants.TIMEOUT).
                    setConnectTimeout(Constants.TIMEOUT).
                    setConnectionRequestTimeout(Constants.TIMEOUT).
                    setProxy(new HttpHost(proxy.getIp(), proxy.getPort())).
                    setCookieSpec(CookieSpecs.STANDARD).
                    build();
            request.setConfig(requestConfig);
            //执行请求，获取响应
            Page page = ZhiHuHttpClient.getInstance().getWebPage(request);
            //获取当前时间戳
            long endTime = System.currentTimeMillis();
            String logStr = Thread.currentThread().getName() + " " +
                            proxy.getProxyStr() +
                    "  executing request " + page.getUrl()  + " response statusCode:"
                      + page.getStatusCode() +
                    "  request cost time:" + (endTime - startTime) + "ms";
            if (page == null || page.getStatusCode() != 200){
                //未能正确响应，直接返回
                logger.warn(logStr);
                return;
            }
            //释放连接
            request.releaseConnection();
            //记录日志
            logger.debug(proxy.toString() + "---------" + page.toString());
            logger.debug(proxy.toString() + "----------代理可用--------请求耗时:" +
                    (endTime - startTime) + "ms");

            //正确响应，该代理可用，添加代理至延时队列
            ProxyPool.proxyQueue.add(proxy);
        } catch (IOException e) {
            logger.debug("IOException:", e);
        } finally {
            //延时队列
            if (request != null){
                request.releaseConnection();
            }
        }
    }
    private String getProxyStr(){
```

```
        return proxy.getIp() + ":" + proxy.getPort();
    }
}
```

1.8.8 代理序列化

代理序列化的时间间隔为 1 分钟。把 ProxyPool 代理池延时队列中的代理序列化至磁盘文件中，代理序列化功能的实现文件是 ProxySerializeTask.java，对应代码如下。

```
/**
 * 代理序列化 task
 */
public class ProxySerializeTask implements Runnable{
    private static Logger logger = LoggerFactory.getLogger(ProxySerializeTask.class);

    /**
     * 实现 Runnable 接口方法
     * 每隔 1 分钟序列化当前可用代理至文件
     */
    @Override
    public void run() {
        while (!ZhiHuHttpClient.isStop){
            try {
                Thread.sleep(1000 * 10 * 1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            Proxy[] proxyArray = null;
            //将 ProxyPool 中可用的代理添加至 proxyArray
            proxyArray = ProxyPool.proxyQueue.toArray(new Proxy[0]);
            //序列化代理至硬盘
            HttpClientUtil.serializeObject(proxyArray, Config.proxyPath);
            logger.info("成功序列化" + proxyArray.length + "个代理");
        }
    }
}
```

1.9 数据可视化分析

将爬取的知乎用户数据保存到数据库后，接下来开始分析用户的信息，并绘制出对应的可视化结果图。

扫码看视频

1.9.1　数据展示模块

在前文讲解了对数据的爬取过程，接下来可以对爬取的数据进行简单的分析和处理。具体的方式是用 SQL 查询出想要的数据，然后通过前端库 echarts 展示在浏览器中。该项目并不是一个 Web 项目，但是需要在浏览器中对数据进行可视化展示，所以需要通过 JDK 自带的轻量级 httpserver 作为 WebServer，后端采用 velocity 模板引擎对数据进行渲染。传递数据到浏览器端后，再通过 echarts 对数据做可视化渲染。

1. HTTP 请求处理

该功能负责对 httpserver 接收的请求进行处理，实现文件是 HttpRequestHandler.java，对应代码如下。

```
/**
 * 具体请求处理方法
 * @param httpExchange
 * @throws IOException
 */
private void process(HttpExchange httpExchange) throws IOException
{httpExchange.sendResponseHeaders(200, 0);
    //获取 response 流
    OutputStream responseBody = httpExchange.getResponseBody();
    List<ChartVO> chartVOList = new ArrayList<>();
    //创建可视化 vo 对象
    chartVOList.add(new ChartVO("followers", "粉丝数", "知乎粉丝数 top10"));
    chartVOList.add(new ChartVO("agrees", "赞同数", "知乎赞同数 top10"));
    chartVOList.add(new ChartVO("thanks", "感谢数", "知乎感谢数 top10"));
    chartVOList.add(new ChartVO("asks", "提问数", "知乎提问数 top10"));
    chartVOList.add(new ChartVO("answers", "回答数", "知乎回答数 top10"));
    chartVOList.add(new ChartVO("posts", "文章数", "知乎文章数 top10"));
    chartVOList.add(new ChartVO("followers", "关注数", "知乎关注数 top10"));
    for(ChartVO vo : chartVOList){
        //调用本类中的方法
        gettop10(vo);
    }
    //调用本类中的方法，渲染数据
    String response = render(chartVOList);
    //write 数据至 response 流中
    responseBody.write(response.getBytes());
    responseBody.close();
}
```

2. 展示数据查询

该功能是通过调用 dao 模块实现的，需要从数据库中查询所需字段的 top10 数据，再根据返回的 vo 对象构造前端页面 echarts 数据字符串对象。该功能的实现文件是 HttpRequestHandler.java，对应的代码如下。

```
/**
 * 根据 vo 对象，从数据库中查询对应数据 top10
 * 通过反射获取对应字段的值，然后构造前端所需要的格式数据
 * @param vo
 * @return
 */
private ChartVO getTop10(ChartVO vo){
    //创建数据库操作对象
    ZhiHuDao zhiHuDao = new ZhiHuDaoImp();
    //创建 sql 查询语句
    String followersTop10Sql = "select * from user order by " + vo.getColumnName()
                               + " desc limit 0, 10";
    //根据 sql 查询数据
    List<User> userList = zhiHuDao.queryUserList(followersTop10Sql);

    //创建前端页面 echarts 横坐标数组字符串对象
    StringBuilder xAxis = new StringBuilder();
    //创建前端页面 echarts 纵坐标数组字符串对象
    StringBuilder series = new StringBuilder();
    xAxis.append("[");
    series.append("[");

    //根据字段名构造对应字段 get 方法名
    StringBuilder methodName = new StringBuilder(vo.getColumnName());
    int c = methodName.charAt(0);
    c =  c - 32;
    methodName.deleteCharAt(0);
    methodName.insert(0, (char) c);
    Method method = null;
    try {
        //获取字段对应的 method 对象
        method = User.class.getMethod("get" + methodName);
    } catch (NoSuchMethodException e) {
        e.printStackTrace();
    }

    //设置数据
    for (User user : userList){
        xAxis.append("'" + user.getUsername() + "',");
        try {
```

```
            int value = (int) method.invoke(user);
            series.append("'" + value + "',");
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        }
    }
    xAxis.deleteCharAt(xAxis.length() - 1);
    series.deleteCharAt(series.length() - 1);
    xAxis.append("]");
    series.append("]");
    vo.setxAxis(xAxis.toString());
    vo.setSeries(series.toString());
    return vo;
}
```

1.9.2　运行展示

启动项目程序文件 Main.java，打开浏览器，输入网址 http://localhost:8080/zhihu-data-analysis 后可以看到爬取数据的实时统计情况。其中，top10 学校用户数统计如图 1-9 所示，知乎粉丝数 top10 用户统计如图 1-10 所示。

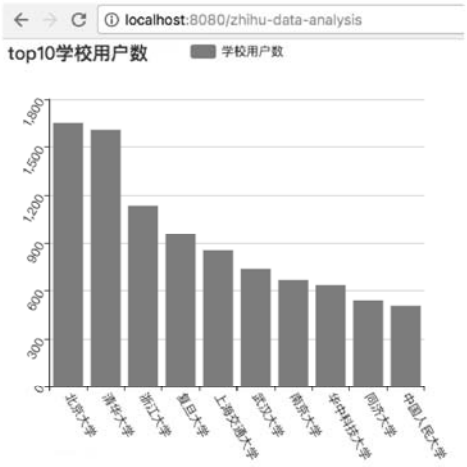

图 1-9　top10 学校用户数统计

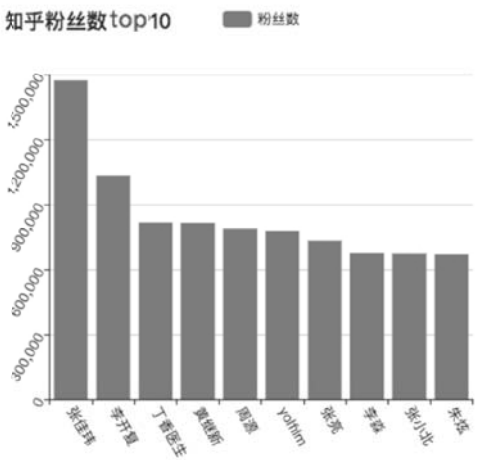

图 1-10　知乎粉丝数 top10 用户统计

知乎赞同数 top10 的用户统计如图 1-11 所示。

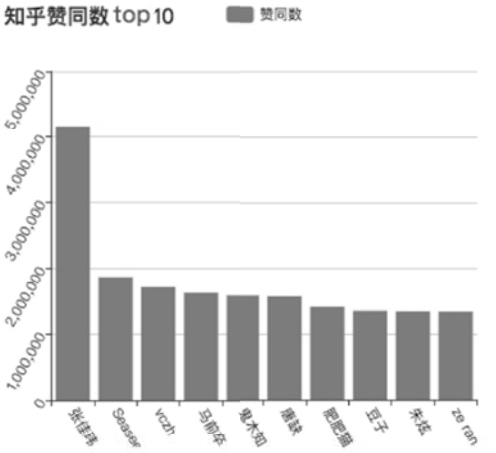

图 1-11　知乎赞同数 top10 的用户统计

1.10　项目开发难点分析


扫码看视频

现在来回顾整个系统的开发过程。对于简单的小型爬虫项目来说，当爬取数据量比较小的时候，项目没什么问题。但是当爬取的数据量比较大的时候，就会带来很多额外的问题，比如网站的统一客户端访问量的限制、数据的存储、URL 去重等。当前的知乎网站对同一客户端就有访问限制，而且一段时间内同一客户端请求量达到一定阈值时，会被知乎服务器禁止访问。此时若要爬取数据，就需要构建自己的免费爬虫代理池。具体的 HTTP 代理去哪儿找呢？可以用爬虫的方式去抓取网上免费公开的代理，然后把这些代理拿来爬取知乎上我们想要的用户数据。在此需要注意的是，网上免费公开代理的可用率非常低，几十个代理中可能只有一两个代理可用。如果想从这些代理中找出可用的代理，就需要编写专门的代理检测模块。拿到可用代理后，具体怎么使用这些代理呢？为了让这些代理达到最大利用率，在多线程抓取时，可通过延时队列来控制同一时刻同一代理最多只有一个请求在请求知乎服务器，并且使同一代理使用间隔为 1 秒，这样就能大大增加代理的复用性。