ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ

ОБРАЗОВАТЕЛЬНОЕ УЧЕРЕЖДЕНИЕ

“ПОВОЛЖСКИЙ ГОСУДАРСТВЕННЫЙ

ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ”

Факультет информатики

и вычислительной техники

Кафедра информатики

и системного программирования

Проектная работа

на тему:

“Проверка сайта на битые ссылки”

Работу выполнил:

студент группы ПС-31

Арзяев Михаил

Научный руководитель:

Курбатов Евгений

Йошкар-Ола, 2019

Содержание

Введение..................................................................................................................3

1. Постановка задачи.....................................................................................5
2. Детали реализации...................................................................................6
3. Руководство пользователя.....................................................................8
4. Тестирование..............................................................................................9

Приложения

1. Листинг программы................................................................................10

Введение

С древних времён люди постоянно создавали различные средства коммуникации для обмена информацией, раз за разом переходя на всё более высокий технологический уровень. Начав с простых глиняных табличек и световых сигналов, человечество сумело создать компьютеры и искусственный интеллект.

Одним из таких средств обмена информацией является Интернет, позволяющий свободно передавать данные друг другу. Для этого можно использовать как частные каналы передачи, так и публичные. Однако, по факту современный Интернет является собранием большого количества страниц - так называемых сайтов.

Преимущество сайтов в том, что их можно связывать между собой ссылками, ведущими с одной страницы на другую. Современные сайты обладают большим переходов на подстраницы этого же сайта и ссылок на внешние страницы. Более того, многие web-сайты активно используют разного рода подгружаемые ресурсы для дизайна, разного функционала или просто для ускорения общего времени загрузки.

Ежедневно делаются сотни новых сайтов, и далеко не каждый из них выполнен качественно. В большинстве случаев это плохо выполненный дизайн страниц или же какие-либо ошибки функционала, но есть и такие сайты, на которых используются нерабочие, или же «битые», как их называют в обиходе, ссылки.

Битые ссылки - это ссылки, которые либо содержат ошибки, либо ведут на несуществующие страницы. Ссылки, ведущие на временно недоступные страницы, можно также считать битыми, но поскольку такие ссылки на самом деле работают по умолчанию, то полноценно считать их сломанныыми нельзя.

Для проверки сайта на битые ссылки существует множество методов:

1. Ручная проверка всех ссылок. Данный метод крайне долог и неудобен, к тому же, визуально нельзя проверить ссылки на подгружаемые ресурсы.
2. Проверка всех ссылок на одной странице. Распространённый метод, имеющий недостаток в виде невозможности проверить ссылки на других страницах того же сайта, связанных с изначальной.
3. Проверка всех ссылок на всех связанных страницах. Неудачный метод, поскольку есть риск того, что вы можете или зайти в бесконечное выполнение программы (с каждой внешней страницы будут добавляться всё новые и новые ссылки, ведущие всё дальше от исходной) или затратить лишние ресурсы и время, тем самым сделав программу неэффективной.

Выгоднее всего соединить второй и третий методы - проверять все ссылки на страницах одного сайта; если ссылка оказывается внешней - то проверить её и не переходить далее, а если внутренней - то проверить и страницу, которая привязана по ней.

Одним из популярнейших языков программирования для проверки страниц на битые ссылки является Java. Данный язык предоставляет множество систем для реализации приложения, способного работать с web-страницами или же с простыми html-документами. Также в нём поддерживается многопоточность, которая позволяет ускорить работу по проверке ссылок.

Само приложение проверки страницы можно разбить на несколько сегментов:

* Парсер кода страницы для обнаружения всех ссылок. Именно он позволяет найти среди тысяч строк кода необходимые нам вещи.
* Проверяльщик ссылок. Каждая ссылка считается рабочей, если она в ответ на обращение возвращает код 2xx - код успешного запроса.
* Формирователь отчётов. Необходимо собрать все ссылки в документ или вывести на экран, чтобы знать, сколько ссылок сломано. Также здесь можно отмечать, какие именно надо исправить и почему они не работают (например, не стоит проверять ссылки с кодами 5хх - это ошибки на стороне сервера, на котором размещена страница).

К программе можно выдвинуть ряд дополнительных требований:

1. Возможность настраивать число потоков, используемых для проверки.
2. Возможность задать файл вывода
3. Возможность проверять несколько файлов, даже если придётся делать очередь.

В целом, разработка подобной программы не является чем-то сложным и осуществляется достаточно просто, несмотря на то, что количество ссылок на странице может достигать нескольких сотен или даже тысяч.

К тому же, подобные приложения активно переходят в web-среду, тем самым не требуя установки каких-либо программ: достаточно лишь интернет-подключения с хорошей скоростью. Язык Java здесь крайне эффективен ввиду своей простоты и удобства, хотя он и не является завязанным на web-разработку напрямую.

1. Постановка задачи

В качестве задачи предполагается разработать поисковик битых ссылок в переданных в качестве аргументов командной строки HTML-документах.

Нефункциональные требования:

* Java 11
* Следование принципам SOLID
* Хранение программы на GitHub
* Maven для сборки проекта
* JUnit 4 для модульных тестов
* Коэффициент покрытия тестами не ниже 70%

Особенности программы:

* Вывод отчётов в CSV-файл;
* Интерфейс командной строки для всех операций;
* Проверка всех ссылок из атрибутов.
* Ссылка считается нерабочей, если она не вернула код 2xx.
* HTTP-запросы используют многопоточность
* Кол-во допустимых потоков конфигурируется отдельным properties-файлом

Так как не уточнено, как страницы используют многопоточность - страница на поток или все потоки на одну страницу, то здесь можно свободно выбрать. Для удобства воспользуемся использованием всех потоков для одной страницы, но при этом ссылки будут расставляться по потокам.

2. Детали реализации задачи

2.1. Алгоритмы задачи.

Основным алгоритмом в решении задачи будет обход всех файлов в последовательном порядке с записью всех ссылок в массив. Любые ошибки при выполнении программы будут успешно перекрываться выбрасываемыми исключенями.

Для конфигурации числа потоков в файле создаётся отдельный файл с расширением .properties, где будет указан единственный параметр, задающий кол-во потоков. Эти потоки будет использовать исполнитель, который берёт поток из пула и ставит задачу по проверке ссылки и добавлении её в отчёт в случае нерабочего состояния. После этого исполнитель ожидает завершения выполнения задач некоторое время или же сразу идёт далее к формированию краткой выписки по результатам программы.

Поскольку точкой входа в приложение Java обычно является функция static void main(String[] args), то большую часть деталей программы приходится объявлять статическими, чтобы к ним можно было обращаться (из нестатического нельзя обратиться к статическому).

2.2. Детали реализации

В первую очередь нам понадобятся несколько полей, которые будут использоваться в ряде функций программы:

* Списки файлов и ссылок
* Название файла отчёта
* Кол-во потоков в пуле
* Поток вывода
* Счётчик найденных ссылок

Для начала, обработаем командную строку посредством библиотеки commons-cli, которая позволит распарсить командную строку на опции и их значения. Если у нас найдутся лишние аргументы, то передадим, что у нас перебор аргументов и выдадим ошибку. При недостаточном числе аргументов или неверных аргументах функция просто выкинет ParseException. Одновременно с парсингом командной строки должен появиться и список документов.

Создаём поток вывода для отчёта и начинаем обход документов. Здесь проводится парсинг и перевод всех данных об HTML-документе в Document из библиотеки jsoup. Затем переводим из jsoup-Document в w3c.dom.Document, который уже парсится через XPath, формируя список, в котором сначала идут ссылки из параметров href, а потом из параметров src. Сформированный список передаётся в цикл, который обращается к функции добавления в отчёт, которая вызывает проверку ссылки и получает Map.Entry<Integer, String>, где число отвечает за статус, а строка - за сообщение о статусе. Если у нас ссылка окажется нерабочей, то вернётся HTTP-статус ошибки и она запишется в отчёт, но на некоторых системах нерабочие ссылки (обычно вызывают ошибку 404) могут вызывать UnknownHostException, который наследуется от IOException - его мы вынуждены обработать и записать ссылку с кодом ответа -1 и сообщением о неизвестном хосте в отчёт.

В цикле используется java.concurrent, предоставляющая класс Executors для работы с многопоточностью. Каждая ссылка ставится в свой поток (необязательно уникальный), перед циклом происходит инициализация пула потоков. После обхода происходит вывод системного сообщения.

3. Руководство пользователя

После сборки программы её можно запускать из командой строки в следующем формате:

*~~brokenLinkFinder --files <file1> [<file2>...] --out <report>~~*

<file1>, <file2> и т.д - файлы на проверку

<report> - имя файла вывода.

Ошибки, которые могут возникнуть:

* Command line reading error - лишние аргументы командной строки. Проверьте их количество.
* Parsing failed - ошибка в ходе считывания командной строки (недостаточно аргументов или они неверны). Проверьте их написание и количество.
* Failed report creation - ошибка создания файла отчёта. Удостоверьтесь, что у программы или пользователя есть право на создание файла.
* Read error - ошибка считывания из файлов. Проверьте файлы на корректность или существование.
* Config error - ошибка конфигурации потоков. Возможно, вы ошиблись при создании файла config.properties - в нём должен быть лишь один параметр, и это параметр thread\_number.

4. Тестирование.

3 функции программы возвращают аргумент void, и не могут быть проверены напрямую. Приведём листинг тестового класса:

import org.apache.commons.cli.ParseException;  
import org.junit.Test;  
import java.io.FileInputStream;  
import java.io.FileNotFoundException;  
import java.io.IOException;  
import java.util.AbstractMap;  
import java.util.Map;  
import java.util.Properties;  
import static org.junit.Assert.\*;

public class HtmlLinkCheckerTest {  
 @Test  
 public void CLCheck(String[] args)  
 {  
 try {  
 *assertTrue*(HtmlLinkChecker*.CLCheck*(args));  
 }  
 catch (ParseException e)  
 {  
 System.*out*.println("Command line check test failed");  
 }  
 }  
 @Test  
 public void checkLinkTest() {  
 Map.Entry<Integer, String> entry = new AbstractMap.SimpleEntry<>(200, "OK");  
 try {  
 *assertEquals*(entry, HtmlLinkChecker*.checkLink*("http://google.com"));  
 }  
 catch (IOException e)  
 {  
 System.*out*.println("Working link check test failed");  
 }  
 }  
 @Test  
 public void configProperties() {  
 Properties props = new Properties(1);  
 try  
 {  
 props.load(new FileInputStream("config.properties"));  
 }  
 catch (IOException e)  
 {  
 System.*out*.println("Error: properties read fail");  
 }  
 int pool = Integer*.parseInt*(props.getProperty("thread\_number", "8"));  
 try  
 {  
 *assertEquals*(pool, HtmlLinkChecker*.configProperties*());  
 }  
 catch (IOException e)  
 {  
 System.*out*.println("Properties config test failed");  
 }  
 }  
}

Приложение А: Листинг программы

import org.apache.commons.cli.\*;  
import org.jsoup.Jsoup;  
import org.jsoup.helper.W3CDom;  
import org.jsoup.nodes.Document;  
import org.w3c.dom.NodeList;  
import javax.xml.xpath.XPath;  
import javax.xml.xpath.XPathConstants;  
import javax.xml.xpath.XPathExpressionException;  
import javax.xml.xpath.XPathFactory;  
import java.io.File;  
import java.io.\*;  
import java.io.IOException;  
import java.net.\*;  
import java.util.\*;  
import java.util.concurrent.ExecutorService;  
import java.util.concurrent.Executors;  
import java.io.PrintStream;  
import java.util.concurrent.TimeUnit;  
  
public class HtmlLinkChecker {  
 static ArrayList<String> *fileList* = new ArrayList<String>();  
 static String *reportFile*;  
 static ArrayList<String> *linkList* = new ArrayList<String>();  
 static int *pool*;  
 static PrintStream *reportStream*;  
 static int *counter* = 0;  
 static boolean CLCheck(String[] args) throws ParseException {  
 Options options = new Options();  
 Option filesOption = new Option("f", "files", true, "Имена проверочных файлов");  
 filesOption.setRequired(true);  
 options.addOption(filesOption);  
 Option outOption = new Option("o", "out", true, "Имя выходного файла");  
 outOption.setRequired(true);  
 options.addOption(outOption);  
 DefaultParser parser = new DefaultParser();  
 CommandLine commandLine = parser.parse(options, args);  
 *fileList*.addAll(Arrays*.asList*(commandLine.getOptionValue("f").split(",")));  
 *reportFile* = commandLine.getOptionValue("o");  
 if (commandLine.getArgs().length > 0)  
 {  
 return false;  
 }  
 return true;  
 }  
  
  
  
 static void parseDocument(String name) throws IOException  
 {  
 File input = new File(name);  
 Document doc = Jsoup*.parse*(input, "UTF-8");  
 W3CDom w3CDom = new W3CDom();  
 org.w3c.dom.Document w3cDoc = w3CDom.fromJsoup(doc);  
 XPath xPath = XPathFactory*.newInstance*().newXPath();  
 try {  
 NodeList nodeList = (NodeList) xPath.compile("//@href").evaluate(w3cDoc, XPathConstants.*NODESET*);  
 for (int i=0;i<nodeList.getLength();i++)  
 {  
 *linkList*.add(nodeList.item(i).getNodeValue());  
 }  
 }  
 catch (XPathExpressionException e) { }  
 try {  
 NodeList nodeList = (NodeList) xPath.compile("//@src").evaluate(w3cDoc, XPathConstants.*NODESET*);  
 for (int i=0;i<nodeList.getLength();i++)  
 {  
 *linkList*.add(nodeList.item(i).getNodeValue());  
 }  
 }  
 catch (XPathExpressionException e) { }  
 }  
  
 static Map.Entry<Integer, String> checkLink(String link) throws IOException  
 {  
 URL url = new URL(link);  
 HttpURLConnection con = (HttpURLConnection) url.openConnection();  
 con.setRequestMethod("GET");  
 int status = con.getResponseCode();  
 String mess = con.getResponseMessage();  
 con.disconnect();  
 return new AbstractMap.SimpleEntry<>(status, mess);  
 }  
  
 static void addToReport(String link)  
 {  
 try  
 {  
 Map.Entry<Integer, String> map = *checkLink*(link);  
 if ((map.getKey() < 200)||(map.getKey() >= 300))  
 {  
 *reportStream*.println(link+","+map.getKey()+","+map.getValue());  
 *counter*++;  
 }  
 }  
 catch (IOException e)  
 {  
 *reportStream*.println(link+",-1,No Such Host");  
 *counter*++;  
 }  
 }  
  
 static int configProperties() throws IOException  
 {  
 Properties props = new Properties(1);  
 props.load(new FileInputStream("config.properties"));  
 int pool = Integer*.parseInt*(props.getProperty("thread\_number", "8"));  
 return pool;  
 }

static public void main(String[] args)  
 {  
 try  
 {  
 if (!*CLCheck*(args))  
 {  
 System.*out*.println("Command line reading error");  
 return;  
 }  
 }  
 catch (ParseException e)  
 {  
 System.*out*.println("Parsing failed");  
 return;  
 }  
 try  
 {  
 *reportStream* = new PrintStream(*reportFile*);  
 }  
 catch (IOException e)  
 {  
 System.*out*.println("Failed report creation");  
 return;  
 }  
 try  
 {  
 for (String string: *fileList*)  
 {  
 *parseDocument*(string);  
 }  
 }  
 catch (IOException e)  
 {  
 System.*out*.println("Read error");  
 return;  
 }  
 try  
 {  
 *pool* = *configProperties*();  
 }  
 catch (IOException e)  
 {  
 System.*out*.println("Config error");  
 return;  
 }  
 ExecutorService executor = Executors*.newFixedThreadPool*(*pool*);  
 for (String string: *linkList*)  
 {  
 executor.execute(()->{  
 *addToReport*(string);  
 });  
 }  
 executor.shutdown();  
 try  
 {  
 executor.awaitTermination(3, TimeUnit.*MINUTES*);  
 }  
 catch (InterruptedException e) { }  
 System.*out*.println("Found "+*counter*+" broken links, for details check file "+*reportFile*);  
 }  
}