Java API 文档搜索引擎

认识搜索引擎

先观察, 百度搜索的搜索结果页中, 包含了若干条结果.
每一个结果中, 又包含了图标, 标题, 描述, 展示url, 时间, 子链, 图片等.

搜索引擎的本质

输入一个查询词, 得到若干个搜索结果. 每个搜索结果包含了标题, 描述, 展示 url 和点击 url.

搜索的核心思路

当前我们有很多的网页(假设上亿个), 每个网页我们称为是一个**文档**

如何高效进行检索? 查找出有哪些网页是和查询词具有一定的相关性呢?

我们可以认为, 网页中包含了查询词(或者查询词的一部分), 就认为具有相关性.

那么我们就有了一个直观的解决思路

方案一 -- 暴力搜索

每次处理搜索请求的时候, 拿着查询词去所有的网页中搜索一遍, 检查每个网页是否包含查询词字符串.

这个方法是否可行?

显然, 这个方案的开销非常大. 并且随着文档数量的增多, 这样的开销会线性增长. 而搜索引擎往往对于效率的要求非常高.

方案二 -- 倒排索引

这是一种专门针对搜索引擎场景而设计的数据结构.

文档(doc): 被检索的html页面(经过预处理)
正排索引: "一个文档包含了哪些词". 描述一个文档的基本信息, 包括文档标题, 文档正文, 文档标题和正文的分词
/断句结果
倒排索引: "一个词被哪些文档引用了". 描述了一个词的基本信息, 包括这个词都被哪些文档引用, 这个词在该文档
中的重要程度, 以及这个词的出现位置等.

项目目标

实现一个 Java API 文档的简单的搜索引擎.

Java API 文档线上版本参见 https://docs.oracle.com/javase/8/docs/api/index.html

下载版本参见 https://www.oracle.com/technetwork/java/javase/downloads/index.html

下载之后看到一个 docs 目录, 里面存在一个 api 目录. 这里的 html 就和线上版本的文档是一一对应的.

例如, 我们熟悉的 java.util.Collection 类,

在线文档的链接是 https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html

下载的文档的目录是 `\api\java\util\Collection.html`

这两者存在一定的对应关系.

核心流程

1. 索引模块: 扫描下载到的文档, 分析数据内容构建正排+倒排索引.
2. 搜索模块: 根据输入的查询词, 基于正排+倒排索引进行检索, 得到检索结果.
3. 前端模块: 编写一个简单的页面, 展示搜索结果. 点击其中的搜索结果能跳转到对应的 Java API 文档页面.

关于分词

分词是搜索中的一个核心操作. 尤其是中文分词, 比较复杂(当然, 咱们此处暂不涉及中文分词)

我们可以使用现成的分词库 ansj.

官网网站: https://github.com/NLPchina/ansj_seg

使用的简单示例: https://blog.csdn.net/weixin_44112790/article/details/86756741

```
public class test.TestAnsj {
    public static void main(String[] args) {
        String str = "小明毕业于清华大学计算机专业," +
                     "后来去蓝翔技校和新东方深造," +
                     "擅长使用计算机控制挖掘机炒菜";
        List<Term> terms = ToAnalysis.parse(str).getTerms();
        for (Term term : terms) {
            System.out.print(term.getName() + "/")
        }
    }
}
```

输出结果

```
小明/毕业/于/清华大学/计算机/专业/,/后来/去/蓝/翔/技校/和/新/东方/深造/,/擅长/使用/计算机/控制/挖掘机/炒菜/
```

注意: 当 anjs 对英文分词时, 会自动把单词转为小写.

公共模块: 创建 DocInfo 类

创建 common 包, 创建 DocInfo 类. 这个就作为一个文档对应的结构.

```
public class DocInfo {
    private int docId;
    private String title;
    private String url;
    private String content;
}
```

预处理模块

遍历目录下所有的文件, 并读取每个文件的内容, 把所有文件整理成一个行文本文件.

这个行文本文件, 每行对应一个 html .

每一行中有三列, 用 \3 分割. 分别是标题, url, 正文.

核心流程:

1. 枚举出文档目录下所有的 html 文件
2. 遍历每个文件, 把文件格式进行转换
3. 把最终结果写入到一个结果文件中.

在 searcher 包中创建 Parser 类. 这个类是一个单独的可执行程序.

```
public class Parser {
    private static final String INPUT_PATH = "D:\\jdk1.8\\docs\\api";
    private static final String OUTPUT_PATH = "D:/raw_data.txt";

    public static void main(String[] args) {
        try {
            File resultFile = new File(OUTPUT_PATH);
            FileWriter resultFileWriter = new FileWriter(resultFile);

            // 1. 枚举出这个目录下的所有文件
            ArrayList<File> fileList = new ArrayList<>();
            enumFile(INPUT_PATH, fileList);
            for (File f : fileList) {
                System.out.println("converting " + f.getAbsolutePath());
                // 2. 针对每个文件, 打开, 并读取内容, 进行转换
                String line = convertLine(f);
                // 3. 把转换结果写入到最终输出文档中
                resultFileWriter.write(line);
            }
            resultFileWriter.close();
            System.out.println("converting done!");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

```

```java
    // 递归完成目录枚举过程
    private static void enumFile(String rootPath, ArrayList<File> fileList)
{
        File rootFile = new File(rootPath);
        File[] files = rootFile.listFiles();
        for (File f : files) {
            if (f.isDirectory()) {
                enumFile(f.getAbsolutePath(), fileList);
            } else if (f.getAbsolutePath().endsWith(".html")) {
                fileList.add(f);
            }
        }
    }

    private static String convertLine(File f) {
        // 1. 转换出标题
        String title = convertTitle(f);
        // 2. 转换出 url
        String url = convertUrl(f);
        // 3. 转换出正文(正文需要去除 html 标签)
        String content = convertContent(f);
        return title + "\3" + url + "\3" + content + "\n";
    }

    private static String convertTitle(File f) {
        // 直接使用文件名作为标题
        String name = f.getName();
        return name.substring(0, name.length() - ".html".length());
    }

    private static String convertUrl(File f) {
        // 这个 url 是指在线文档对应的链接.
        // url 由两个部分构成.
        // 第一部分是 https://docs.oracle.com/javase/8/docs/api
        // 第二部分是 文件路径中 api 之后的部分.
        String part1 = "https://docs.oracle.com/javase/8/docs/api";
        String part2 = f.getAbsolutePath().substring(INPUT_PATH.length());
        return part1 + part2;
    }

    private static String convertContent(File f) {
        // 读取文件内容, 并去除其中的 html 标签和换行
        try {
            FileReader fileReader = new FileReader(f);
            // 是否当前读的字符是正文
            boolean isContent = true;
            StringBuilder output = new StringBuilder();
            while (true) {
                int ret = fileReader.read();
                if (ret == -1) {
                    break;
                }
                char c = (char)ret;
                if (isContent) {
                    if (c == '<') {
                        isContent = false;
                        continue;
                    }
```

```
                    if (c == '\n' || c == '\r') {
                        c = ' ';
                    }
                    output.append(c);
                } else {
                    if (c == '>') {
                        isContent = true;
                    }
                }
            }
            fileReader.close();
            return output.toString();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
        return "";
    }
}
```

索引模块

在 searcher 包中创建 Index 类. 这个类的目的是把刚才的 raw_data 文件做成索引.

Index 类中要包含正排索引和倒排索引, 结构如下

```
class Weight {
    public String word;
    public int docId;
    public int weight;
}

public class Index {
    // 正排索引, 下标对应 docId
    private ArrayList<DocInfo> forwardIndex = new ArrayList<>();
    // 倒排索引, key 是分词结果, value 是这个分词 term 对应的倒排拉链(包含一堆
docid)
    private HashMap<String, ArrayList<Weight>> invertedIndex = new HashMap<>
();
}
```

Index 类中要支持的接口如下:

```
// 根据 docId 查正排
public DocInfo getDocInfo(int docId) {
    return forwardIndex.get(docId);
}

// 根据 分词结果 查倒排
public ArrayList<Weight> getInverted(String term) {
    return invertedIndex.get(term);
```

```
}

// 构建索引, 根据 raw_data 文件在内存中构造索引结构
public void build(String inputPath) {
    // TODO
}
```

接下来重点实现 Index.build 方法, 这也是实现索引模块最复杂的方法.

```
// 构建索引, 根据 raw_data 文件在内存中构造索引结构
public void build(String inputPath) {
    System.out.println("build start! ");
    long startTime = new Date().getTime();
    try {
        // 1. 按行读取文件内容(每行是一个 html)
        FileReader fileReader = new FileReader(inputPath);
        BufferedReader bufferedReader = new BufferedReader(fileReader);
        while (true) {
            String line = bufferedReader.readLine();
            if (line == null) {
                break;
            }
            // 2. 构造 DocInfo 并更新正排索引
            DocInfo docInfo = buildForward(line);
            if (docInfo == null) {
                continue;
            }
            // 3. 构造 Weight 并更新倒排索引
            buildInverted(docInfo);
            System.out.println("Build " + docInfo.getTitle() + " done!");
        }
        System.out.println("Build Done! " + (new Date().getTime() -
startTime));
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

实现 buildForward (比较简单)

```
private DocInfo buildForward(String line) {
    String[] tokens = line.split("\3");
    if (tokens.length != 3) {
        System.err.println("tokens 长度不为 3: " + line);
        return null;
    }
    DocInfo docInfo = new DocInfo();
    docInfo.setDocId(forwardIndex.size());
    docInfo.setTitle(tokens[0]);
    docInfo.setUrl(tokens[1]);
    docInfo.setContent(tokens[2]);
    forwardIndex.add(docInfo);
    return docInfo;
```

```
}
```

实现 buildInverted (比较复杂)

```
private void buildInverted(DocInfo docInfo) {
    // 构造 Weight 对象, 并更新倒排索引.
    // 此处 "权重" 简单粗暴的认为词出现的次数.
    // weight = 10 * 这个词标题中出现的次数 + 1 * 这个词正文中出现的次数
    // 核心流程:
    // 1. 对标题进行分词
    // 2. 遍历分词结果, 统计标题中每个词出现的次数
    // 3. 对正文进行分词
    // 4. 遍历分词结果, 统计正文中每个词出现的次数
    // 5. 把以上内容都整理到一个 HashMap 中
    // 6. 遍历 HashMap, 可以得到 词 -> 权重 这样的映射关系, 更新到倒排索引中
    // 这个类用于辅助统计词出现的次数
    class WordCnt {
        public int titleCount;
        public int contentCount;

        public WordCnt(int titleCount, int contentCount) {
            this.titleCount = titleCount;
            this.contentCount = contentCount;
        }
    }
    HashMap<String, WordCnt> wordCntMap = new HashMap<>();

    // 1. 对标题进行分词
    List<Term> terms = ToAnalysis.parse(docInfo.getTitle()).getTerms();
    // 2. 遍历分词结果, 统计标题中每个词出现的次数
    for (Term term : terms) {
        // 注意, 此时的 word 已经被 anjs 转成小写了.
        String word = term.getName();
        WordCnt wordCnt = wordCntMap.get(word);
        if (wordCnt == null) {
            wordCntMap.put(word, new WordCnt(1, 0));
        } else {
            wordCnt.titleCount++;
        }
    }
    // 3. 对正文进行分词
    terms = ToAnalysis.parse(docInfo.getContent()).getTerms();
    // 4. 统计正文中出现的词的个数
    for (Term term : terms) {
        // 注意, 此时的 word 已经被 anjs 转成小写了.
        String word = term.getName();
        WordCnt wordCnt = wordCntMap.get(word);
        if (wordCnt == null) {
            wordCntMap.put(word, new WordCnt(0, 1));
        } else {
            wordCnt.contentCount++;
        }
    }
    // 5. 把以上内容都整理到一个 HashMap 中
    for (HashMap.Entry<String, WordCnt> entry : wordCntMap.entrySet()) {
```

```
        // 6. 遍历 HashMap, 可以得到 词 -> 权重 这样的映射关系, 更新到倒排索引中
        Weight weight = new Weight();
        weight.setDocId(docInfo.getDocId());
        weight.setWeight(entry.getValue().titleCount * 10
                         + entry.getValue().contentCount);
        weight.setWord(entry.getKey());
        // 这个逻辑也可以使用 Map.putIfAbsent. 此处为了直观, 还是直接用 get 吧
        ArrayList<Weight> invertedList = invertedIndex.get(entry.getKey());
        if (invertedList == null) {
            invertedList = new ArrayList<>();
            invertedIndex.put(entry.getKey(), invertedList);
        }
        invertedList.add(weight);
    }
}
```

测试下索引的构建, 在 Index 中写一个 main 方法, 运行一下.

```
public static void main(String[] args) {
    Index index = new Index();
    index.build("d:/raw_data.txt");
    ArrayList<Weight> weights = index.getInverted("arraylist");
    for (Weight weight : weights) {
        System.out.println("docId: " + weight.getDocId());
        System.out.println("weight: " + weight.getWeight());
        System.out.println("title: " +
index.getDocInfo(weight.getDocId()).getTitle());
    }
}
```

挑两个 case 来看看 weight 计算的是否正确即可.

搜索模块

在 searcher 包中, 创建 DocSearcher 类. 实现搜索的核心流程; 创建 Result 类, 表示结果

```
// 不同于之前的 DocInfo, 这个 Result 是用来表示最终展示结果的.
public class Result {
    private String title;
    private String url;
    // desc 是正文的一段摘要
    private String desc;

    // getter setter 略
}
```

DocSearcher 需要实现的方法:

1. 构造方法, 要构建索引
2. search 方法, 搜索的核心流程

```java
public class DocSearcher {
    private Index index = new Index();

    public DocSearcher() {
        index.build("d:/raw_data.txt");
    }

    // 根据查询词，进行搜索，得到搜索结果集合.
    // 结果集合中包含若干条记录，每个记录中包含搜索结果的标题，描述，url
    public List<Result> search(String query) {

    }
}
```

实现 search 方法

```java
// 根据查询词，进行搜索，得到搜索结果集合.
// 结果集合中包含若干条记录，每个记录中包含搜索结果的标题，描述，url
// 1. [分词] 对查询词进行分词
// 2. [触发] 对每个分词结果查找倒排索引，得到一个倒排拉链
// 3. [排序] 针对结果集合进行排序，按权重降序排序即可
// 4. [返回] 构造返回结果
public List<Result> search(String query) {
    // 1. [分词] 对查询词进行分词
    List<Term> terms = ToAnalysis.parse(query).getTerms();
    // 2. [触发] 对每个分词结果查找倒排索引，得到一个倒排拉链
    ArrayList<Weight> allTokenResult = new ArrayList<>();
    for (Term term : terms) {
        String word = term.getName();
        ArrayList<Weight> invertedList = index.getInverted(word);
        // 倒排拉链可能找不到，比如这个词根本就在索引中不存在.
        if (invertedList == null) {
            continue;
        }
        allTokenResult.addAll(invertedList);
    }
    // 3. [排序] 针对结果集合进行排序，按权重降序排序即可
    allTokenResult.sort(new Comparator<Weight>() {
        @Override
        public int compare(Weight o1, Weight o2) {
            // 如果升序就是 o1 - o2，降序是 o2 - o1
            return o2.getWeight() - o1.getWeight();
        }
    });
    // 4. [返回] 构造返回结果
    ArrayList<Result> results = new ArrayList<>();
    for (Weight weight : allTokenResult) {
        DocInfo docInfo = index.getDocInfo(weight.getDocId());
        Result result = new Result();
        result.setTitle(docInfo.getTitle());
        result.setUrl(docInfo.getUrl());
        result.setDesc(GenDesc(docInfo.getContent(), weight.getWord()));
        results.add(result);
    }
    return results;
}
```

实现下 GenDesc 方法, 生成一下描述信息

```
private String GenDesc(String content, String word) {
    // 先找一下 word 在 content 中第一次出现的位置
    int firstPos = content.indexOf(word);
    if (firstPos == -1) {
        return "";
    }
    // 直接截取 firstPos 周围的文本
    // 从 firstPos 往前找 60 个字符作为描述开始,
    // 然后从描述开始位置往后找 160 个字符作为整个描述
    // 注意: 此处的 60, 160 都是拍脑门出来的
    int descBeg = firstPos < 60 ? 0 : firstPos - 60;
    if (descBeg + 160 > content.length()) {
        return content.substring(descBeg);
    }
    // 正文长度充足, 在最后加上 ...
    return content.substring(descBeg, descBeg + 160) + "...";
}
```

写个 main 方法测试一下 DocSearcher 类

```
public static void main(String[] args) {
    DocSearcher docSearcher = new DocSearcher();
    List<Result> results = docSearcher.search("ArrayList");
    for (Result result : results) {
        System.out.println(result);
    }
}
```

实现 Servlet

接口设计

```
请求:
/search?query=ArrayList

响应:
{
    ok: true,
    data: [
        {
            title: "title1",
            url: "url1",
            desc: "desc1",
        },
        {
            title: "title2",
            url: "url2",
            desc: "desc2",
        },
    ]
}
```

关于 Json

Json 是一种常见是数据格式组织方式. 源于 JavaScript, 是一种键值对风格的数据格式.

Java 中可以使用 Gson 库来完成 Json 的解析和构造.

在 Maven 中新增 Gson 的依赖

```
<dependency>
    <groupId>com.google.code.gson</groupId>
    <artifactId>gson</artifactId>
    <version>2.8.2</version>
</dependency>
```

简单示例 (创建一个 TestGson 类)

```
public class TestGson {
    public static void main(String[] args) {
        HashMap<String, Object> data = new HashMap<>();
        data.put("name", "曹操");
        data.put("skill1", "剑气");
        data.put("skill2", "三段跳");
        data.put("skill3", "吸血加攻击");
        data.put("skill4", "释放技能加攻速");
        Gson gson = new GsonBuilder().create();
        String jsonData = gson.toJson(data);
        System.out.println(jsonData);
    }
}
```

实现 DocSearcherServlet

创建 servlet 包, 创建 DocSearcherServlet 类, 继承自 HttpServlet . 此处需要使用 Gson 作为 Json 解析库.

```
public class DocSearcherServlet extends HttpServlet {
    private DocSearcher docSearcher = new DocSearcher();

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
throws                          ServletException, IOException {
        resp.setContentType("application/json; charset=utf-8");
        HashMap<String, Object> data = new HashMap<>();
        Gson gson = new GsonBuilder().create();
        String query = req.getParameter("query");
        if (query == null || query.equals("")) {
            data.put("ok", false);
            data.put("reason", "query is empty");
            String respData = gson.toJson(data);
            resp.getWriter().write(respData);
            return;
        }
```

```
            List<Result> results = docSearcher.search(query);
            data.put("ok", true);
            data.put("data", results);
            String respData = gson.toJson(data);
            resp.getWriter().write(respData);
        }
    }
```

修改 web.xml

```
<servlet>
    <servlet-name>DocSearcherServlet</servlet-name>
    <servlet-class>servlet.DocSearcherServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>DocSearcherServlet</servlet-name>
    <url-pattern>/search</url-pattern>
</servlet-mapping>
```

此时发现一个问题. 索引制作的过程是在第一次收到请求的时候才开始的. 这显然不科学.

如果想让索引能在服务启动的时候就进行制作, 也就需要让 Servlet 在 Tomcat 启动时就被创建和加载.

此时可以在 web.xml 中新增一个 load-on-startup 选项即可.

```
<servlet>
    <servlet-name>DocSearcherServlet</servlet-name>
    <servlet-class>servlet.DocSearcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
```

其中这个 值 如果填成 <= 0 的, 就表示第一次收到请求再加载. 如果 > 0 的, 就表示服务器启动就加载. 其中数字越小, 就越先加载.

参考 https://blog.csdn.net/lvxiangan/article/details/80582487

实现前端页面

前端页面结构比较简单, 只要包含一个输入框和一个按钮即可. 另外要对返回的结果进行一个比较合理的组织.

创建一个 index.html. 这个就不讲了吧, 直接发给大家.

```
<html>
<head>
    <!-- Bootstrap 文档: https://v3.bootcss.com/css/ -->
    <!-- Vue 文档: https://cn.vuejs.org/v2/guide/ -->
    <!-- Required meta tags -->
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1,
shrink-to-fit=no">

    <!-- Bootstrap CSS -->
```

```
    <link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/bootstrap@3.3.7/dist/css/bootstrap.min.cs
s" integrity="sha384-
BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u"
crossorigin="anonymous">

    <title>Java API 搜索</title>
    <style>
    #app {
        margin-left:50px;
        margin-right:50px;
    }
    div button {
        width:100%;
    }
    .row {
        padding-top: 10px;
    }
    .col-md-5,.col-md-1 {
        padding-left:2;
        padding-right:2;
    }
    .title {
        font-size: 22px;
    }
    .desc {
        font-size: 18px;
    }
    .url {
        font-size: 18px;
        color: green;
    }
    </style>
</head>
<body>
<div id="app">
    <div class="row">我是 logo</div>
    <div class="row">
        <div class="col-md-5">
            <input type="text" class="form-control" placeholder="请输入关键字"
v-model="query">
        </div>
        <div class="col-md-1">
            <button class="btn btn-success" v-on:click="search()">搜索
</button>
        </div>
    </div>
    <div class="row" v-for="result in results">
        <!--用来存放结果-->
        <div class="title"><a v-bind:href="result.url">{{result.title}}</a>
</div>
        <div class="desc">{{result.desc}}</div>
        <div class="url">{{result.url}}</div>
    </div>
</div>
</body>
<script src="https://apps.bdimg.com/libs/jquery/2.1.4/jquery.min.js">
</script>
```

```
<script
src="https://cdn.jsdelivr.net/npm/bootstrap@3.3.7/dist/js/bootstrap.min.js"
integrity="sha384-
Tc5IQib027qvyjSMfHjOMaLkfuWVxZxUPnCJA7l2mCWNIpG9mGCD8wGNIcPD7Txa"
crossorigin="anonymous"></script>
<script src="https://cdn.jsdelivr.net/npm/vue"></script>
<script>
    var vm = new Vue({
        el: "#app",
        data: {
            query: "",
            results: [ ]
        },
        methods: {
            search() {
                $.ajax({
                    url:"/java_doc_searcher/search?query=" + this.query,
                    type: "get",
                    context: this,
                    success: function(respData, status) {
                        this.results = respData.data;
                    }
                })
            },
        }
    })
  </script>
</html>
```

后续改进

1. 数据量更大
2. 请求量更大
3. 业务更复杂(相关性策略)