# 搜索引擎技术基础课程实验报告

计 75 班 赵成钢 2017011362
计 71 班 郑逢时 2016012177

2020 年 6 月

## 目录

# 1 问题描述和项目简介

本小组选择了司法搜索引擎作为选题，具体的问题是：给定特定的司法案例数据集，实现一个搜索引擎，可以通过长案件输入或者关键词查找相关的案例并根据相关性排序，实现标签、法官、法律过滤功能，在用户未完成的输入上实现联想的功能。

我们基于上述问题的描述，实现了 ElasticJury 搜索引擎，名字模仿 ElasticSearch，意在实现高性能、高扩展、高质量结果的弹性司法搜索引擎。

# 2 实现模块和关键功能

## 2.1 架构设计

在架构上，我们共分为 4 个模块：数据库、数据清洗、前端和后端，如下图所示：

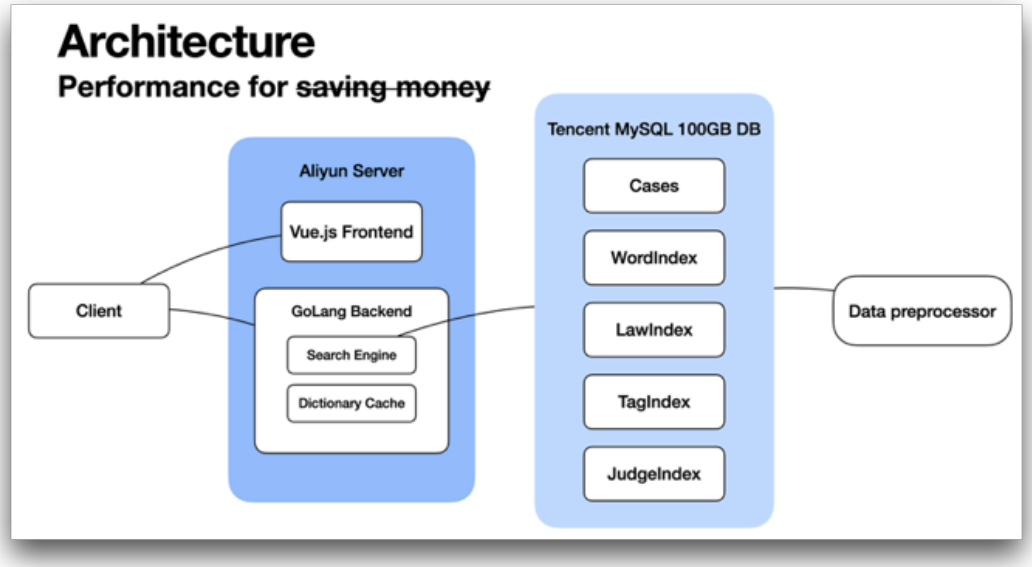


图 1: 架构设计

下面将会分为对应的四个部分进行介绍。

## 2.2 数据库设计

### 2.2.1 数据表

为了实现基本的查找和对应得分排序的功能，我们采用了倒排索引的数据库组织结构。其中数据表分为 5 个，分别为：

- 案件
- 词倒排

- 法官倒排
- 法律倒排
- 标签倒排

具体实现和组织对应着 database/init-tables.sql 文件。

#### 2.2.2 案件数据表

其中的案件数据表用来存取每个案件的信息和结构，每个案件对应一条数据，其中包括：

- 标号（id 段）
- 法官（judges 段）
- 法律（laws 段）
- 标签（tags 段）
- 全文（detail 段）
- 树结构（tree 段）

同时，我们把标号字段作为唯一索引（Primary Key）。而后面每个字段是通过数据清洗得出的，将在后面会解释。

#### 2.2.3 词、法官、法律和标签倒排索引数据表

由于这四个数据表结构完全一致，下面我们一起介绍。这四个表存储的是倒排索引的数据，每条数据对应有：

- 信息（word/judge/law/tag 段）
- 案件标号（caseId 段）
- 权重（weight 段）

同时，我们把信息和案件标号组成的元组组织为唯一索引，并也建立权重段的索引，方便 MySQL 查询时直接按照权重排序。

对于四个表的权重段，我们采取了不同的方案：对于词语，我们采用了 TF-IDF 作为权重，为了同时突出词语的频率和区分性；对于法官和标签，出于同等地位的考虑，我们把权值固定为常数 1；而对于法律，我们首先按照法条出现的频率进行加权，再加权到法条中的法典上，这样同时保证了通过法条和法典都可以进行查询，而且也可以通过权值区分其重要程度。

## 2.3 数据清洗

我们使用了 Python 作为框架实现了提供的数据集的清洗以及对远程数据库的对接，其中清洗工作分为下面几个流程和模块：

- 全部词语 IDF 的计算
- 英文标签到中文名的映射
- 构造数据库条目

上述功能代码对应 preprocessor 目录，运行入口均为 preprocessor/main.py 文件，具体参数和方法请参考该文件的帮助内容。

### 2.3.1 词语 IDF 计算

对应 preprocessor/idf_dict_collector.py 文件。

具体执行过程则是遍历整个数据集，通过 xml 解析得到每个数据的 QW 字段（全文）。后调用 jieba 分词，再并放入一个 dict 中统计每个词出现的次数。

当数据集遍历完成后，会把 dict 中的信息进一步通过：

$$\mathrm{IDF}(\mathrm{word}) = \log(\frac{\mathrm{total}}{\mathrm{count}_{\mathrm{word}}}) \tag{1}$$

公式计算出每个词的 IDF 得分，也就是每个词在整个数据中的区分能力。后保存到 idf_dict.json 文件中，方便后面的数据清洗阶段或者后端模块使用。

### 2.3.2 中英文标签爬取

在观察数据集的过程中我们发现，存在着形如 FYCJM 这样的 xml 标签（对应着 “法院层级码”），同时也存在着一个英文标签对应着多个中文意思的情况，也存在有的数据没有中文只有英文标签的情况，为了进一步修正这些情况，我们进行了整个数据集的英文标签到中文意思的爬取。

该功能对着 preprocessor/mapping_collector.py 文件。其具体过程为遍历数据集，分析每个 xml 节点的中英文标签，最后放到一个 dict 中，最后保存到 mapping.json 文件中，方便后面的过程使用。

同时，利用分析出来的结果，我们决定了数据库案件条目和原 xml 的对应关系：

- 词语：QW 段分词以及停用词过滤后的结果
- 法律：FT 段
- 法官：FGRYXM 段
- 标签：AJLB, AJLX, AJSJ, WSMC, SPCX, AY, SJLY_AY, WZAY 段（分别为案件类别和案由等）
- 结构：整个 xml 树字符化的结果

最后，一些无法修复或者有映射冲突的条目，我们选择了舍弃。

### 2.3.3 构造数据条目

有了上面两个模块提供给我们的基本信息后，我们再利用 preprocessor/processor.py 文件中提供的 xml 树分析模块遍历每个数据进行分析并构造最后的数据库条目。

遍历的同时，也将对每个字段的权重进行计算，如词语字段会把之前爬取的 IDF 字典再乘上每个词的频率得到每个词在案例中的 TF-IDF 值。

最后，遍历模块将根据结果利用 preprocessor/entry.py 文件构造数据库接口的条目，并通过 preprocessor/database.py 提供的连接提交到远程的数据库中。

### 2.3.4 清洗结果

提供的数据集分为了 xml_1 到 xml_4 一共四个文件夹，根据上述清洗方法，我们从 1 到 3 文件夹中爬取出了 162351 个案例，余下有 45 个 xml 解析错误。而对于 xml_4 文件夹，其中共有 1282 个案例（仅占 0.7%），xml 格式也比较混乱，存在标签缺失等情况，我们选择舍弃。

在倒排数据库中，共有 65672144 个词索引，1301986 个法律索引，378575 个法官索引和 775041 个标签索引。

## 2.4 前端设计

### 2.4.1 总体设计

前端即 Web 形式的用户界面，我们采用的技术栈是 Vue.js + Vuetify 框架。响应式的开发框架结合简洁的风格，给用户以清爽流畅的体验，对应代码在 ui 文件夹。

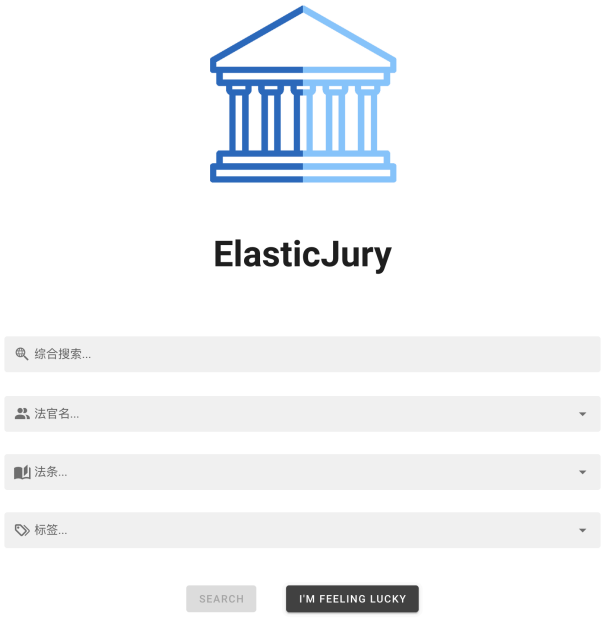


图 2: 主页/搜索页


图 3: 详情页

前端主要包含两个页面，包含的功能如下：

- 搜索框与搜索结果页面：
  - 关键词或类案检索；
  - 法官过滤；
  - 法律过滤；
  - 标签过滤；
  - 智能联想；
  - 搜索结果摘要显示；
  - 搜索结果点击交互；
  - “手气不错”。
- 案件详情页面：
  - 法官名、法条名、标签列表、全文内容；
  - 树形结构可视化；
  - 安检内关键词搜索。

### 2.4.2 简要细节

由于前端不属于项目的重点，所以这里只介绍一些比较难的细节。

**智能联想** 智能联想功能需要捕捉用户正在输入的行为，并在适当时候向后端发送请求。难点包括：

- 即时，准确发现用户的输入行为，减少用户等待时间；
- 开销，不能太高频地向后端发送请求，减小后端压力。

解决方案是基于 Vuetify 的多选框组件做修改，在监听到用户有非空格的输入行为时，先 sleep 半秒，再判断这段时间内用户是否有更多输入，若没有，表明用户希望使用联想功能，向后端发送请求。


图 4: 联想功能示意图

**案件树形结构可视化** 案件数据集的数据大部分是 xml 格式，案件本身的树形结构比单纯的文本更加生动直观。我们最后对 xml 信息进行了一些系列转换，挂载到了 Vuetify 的 TreeViewer 组件上来实现可视化。


图 5: 案件树形结构可视化效果

## 2.5 后端设计

后端采用高并发 Golang 语言，用来处理获取内容、搜索和联想三类请求，下面将结合搜索策略分别介绍三个请求的处理。主要代码在 app 文件夹中。

### 2.5.1 获取内容请求

获取内容请求分为 2 种，一种是获得一组案件标号对应的简要内容，另一类请求则是某个案例的全部内容，这两类操作直接构造 MySQL 查询即可。对应的代码在 app/search.go 文件中。

### 2.5.2 搜索请求

对于搜索请求，为了同时实现关键词搜索和案例搜索，我们选择把用户的输入直接转换为一个语义词向量，然后以这个词向量来评估和每个案例的相关程度，最后以相关程度的顺序返回。

**词向量** 在词向量转换方面，我们调用了 jieba 分词的 Top K 功能，该接口对语义进行建图，用类似 PageRank 的方法实现了 TextRank，分析出来语义的前 K 大的关键词，并带有权值信息（权值是信息上的重要程度，由 jieba 返回）。上述向量的提取对应代码在 app/natural/natural.go 中。

**词向量维度 K 选取** 我们把关键词和权值组作为词向量，而存在一个问题是 K 的具体选择，这里我们做了简单的实验，保证不选择太多词导致检索变慢，也不会因为太少损失信息。

实验的方式就是让每个案例自己搜自己，从每个案例提取出来 K 维的向量，然后查询，比较这个案例在搜索结果中的位置。具体搜索算法在后面会详细描述。

| Case | 5-RR | 5-RT | 10-RR | 10-RT | 20-RR | 20-RT | 100-RR | 100-RT |
|---|---|---|---|---|---|---|---|---|
| 24617 | 1.0 | 1.70s | **1.0** | 1.96s | 1.0 | 2.93s | 1.0 | 10.36s |
| 120924 | 0.5 | 2.22s | **1.0** | 2.50s | 1.0 | 3.38s | 1.0 | 10.77s |
| 137837 | 0.33 | 1.72s | **1.0** | 1.80s | 1.0 | 2.28s | 1.0 | 7.03s |
| 88510 | 0.17 | 2.41s | **1.0** | 2.61s | 1.0 | 2.94s | 1.0 | 9.88s |

表 1: 词向量维数选取的实验结果（RT 为响应时间）

经过权衡，我们选择了 10 个词作为维度，这也意味着 10 个词以内的关键词搜索也可以无损信息进行搜索。

**相似度评估** 最后一个关键技术在于词向量和案例相似程度的评估，即设计 $\mathrm{Similarity}(\mathrm{vector}, \mathrm{case})$，首先这里的 vector 是查询词和其语义和信息上的重要程度构成的向量，我们需要评估两件事情：第一是查询词在对应的案例是否显著（TF-IDF），第二是在对应案例中是否尽可能出现了这些词。

对于第二件事，我们首先对词向量进行平滑处理，给每个词加一个单位 1 （这里选择了平均值）作为出现这个词的奖励：

$$(\mathrm{word}, \mathrm{weight}') = (\mathrm{word}, \mathrm{weight} + \overline{\mathrm{weight}}) \tag{2}$$

然后一个案例对应的得分取为：

$$\Sigma_{\mathrm{word}} \mathrm{weight}'_{\mathrm{word}} \times (\mathrm{TFIDF}_{\mathrm{word}} \times 0.4 + 0.6) \tag{3}$$

其中的 TF-IDF 即为倒排数据库中的权值，体现了这个词是否显著，而之所以要和一个单位 1 进行加权（0.4 和 0.6 是调参得到的结果），也是为了解决第二个问题，我们发现当用户输入多个

关键词时，可能某个词只在一个案例中出现，TF-IDF 得分会非常高，这导致这篇文章即使没有出现其他关键词可以排到很靠前，所以我们必须对 TF-IDF 打一个折扣，平衡这种情况。

**MySQL 构造** 最后，上面的式子我们会构造 MySQL 语句进行查询，具体做法是先构建词向量的临时表，再进行联表查询，而标签等精细化的搜索则也通过 MySQL 来查询，具体做法是查询是否有 (tag, caseId) 元组在对应的精细化倒排数据表中的命中，而不是把全部文章查出来之后再过滤。

下图为一个具体的输入和对应构造出来的全部 MySQL 查询：

```
[Search] Got request:
[Search]  > Misc: 北京 财产 纠纷
[Search]  > Tag: 执行裁定书
[Search]  > Law:
[Search]  > Judge: 邓少宇
CREATE TABLE Weights1591276895785420232
(
        item   VARCHAR(512) NOT NULL,  # 检索词
        weight FLOAT        NOT NULL,  # 检索词语义信息权重
        PRIMARY KEY (item)
) CHAR SET utf8;
INSERT INTO Weights1591276895785420232 (item, weight) VALUES ('纠纷',14.165350),('财产',13.232950),('北京',11.080544)
SELECT b.caseId AS caseId, sum((b.weight * 0.4 + 0.6) * a.weight) AS weight
        FROM Weights1591276895785420232 a, WordIndex b, TagIndex c, JudgeIndex d
        WHERE a.item = b.word AND b.caseId=c.caseId AND (c.tag = '执行裁定书') AND b.caseId=d.caseId AND (d.judge = '邓少宇')
        GROUP BY caseId ORDER BY weight DESC
DROP TABLE Weights1591276895785420232
[Search] Reply with 1 cases
[GIN] 2020/06/04 - 21:21:36 | 200 |  287.084095ms |       127.0.0.1 | POST     "/search"
```

图 6: 案例和对应的 MySQL 查询

查询返回直接就是排序好的结果，为了过滤相关性太差的结果，我们也设置了一个阈值过滤。对应的构造和处理在 app/search.go 文件中。

### 2.5.3 智能联想

首先，我们假设常见词用户本身输入法就可以完成，只需要补全有关法律/法官/标签的字段。我们选取了倒排数据库中全部的标签、法律和法官字段，把每个词的权值规定为全部出现词的案例的原权值和，然后导出为 csv 文件。

后端初始化时会把全部词条和对应的权值加载到内存中，枚举每个词条的每个词（比如在“二审案件”中枚举“二审”和“案件”），然后用对应的全部前缀（如前缀“二”和“二审”）当做键值把整个词插入到一个 map 中，每个前缀对应的整词列表仅保留权值最大的前 10 个，然后按照字符串的长度排序，这样不仅仅是可以完成整个词的前缀联想，输入词中间的部分也可以进行联想。对应的预处理代码在 app/natural/dict.go 中。

利用上述做法，我们可以构造出标签、法律和法官三个字段的精细化搜索中的智能联想，而综合搜索的联想则用三个合并后的 map。当有用户请求时，直接以 $O(\log n)$ 的复杂度查询对应 map 返回对应的 10 个词即可。

# 3 测试结果和样例分析

## 3.1 搜索性能评估

结合课程中所学的各种指标，我们对系统在一些测例下进行了性能评估。

测例包括单个或若干关键词、句子、长段的文本以及干扰文本。评估指标包括响应时间（RT），RR 值和 P@10：

| 搜索输入 | 类型 | 结果条数 | RT | RR | P@10 |
|---|---|---|---|---|---|
| 离婚 | 单个词 | 8905 | 631ms | 1.0 | 100% |
| 危险驾驶 | 单个词 | 18914 | 896 ms | 1.0 | 100% |
| 唐山杀人 | 多个词 | 1904 | 446 ms | 1.0 | 70% |
| 北京财产纠纷 | 多个词 | 103203 | 2.61s | 1.0 | 100% |
| 浙江省嘉兴市发生了刑事案件 | 句子 | 101008 | 2.21s | 1.0 | 80% |
| 四川发生的强奸案件 | 句子 | 134312 | 3.70s | 0.5 | 70% |
| 一长段醉酒驾驶新闻 | 长段文本 | 53077 | 1.58 s | 1.0 | 100% |
| 一长段离婚新闻 | 长段文本 | 122487 | 2.08 s | 0.5 | 80% |
| 鸡你太美 | 干扰文本 | 0 | 284 ms | N/A | N/A |

表 2: 评估结果

从 RR 和 P@10 来看，我们的搜索引擎对多种输入情况都能给出较为相关的结果，同时在结果条数较多、数据库与后端之间网络传输瓶颈较大的情况下是还能保持不超过 4 秒的响应时间，证明了后端高性能的 Golang 选型的优势，也证明了我们对 SQL 语句优化的有效性。

## 3.2 部分测例分析

为了进一步说明性能和特点，我们对两个测例的搜索结果进行详细分析。

### 3.2.1 多关键词和多标签搜索

输入以下搜索关键词，并用上多种标签输入：


图 7: 多关键词和多标签搜索

检索用时 277.628956ms，由于标签较多，匹配到了 7 条结果。


图 8: 搜索结果前两条

第一条结果匹配完全一致。而第二条结果的法官中只有曹新新出现了，郑晔没有出现，其他正确。余下 5 条情况类似，两位法官都只出现了一位，实际上数据库中完全满足情况的也只有第一条。

而搜索引擎之所以匹配出了 7 个而非 1 个结果，是因为考虑到用户可能输入不够准确，而用户的预期更可能是既希望搜索到两位法官都包含的案件，也不要筛掉只含其中一位法官的结果，导致结果太少。所以我们的过滤规则中，是让同级标签（例如多位法官）索引到的案件取并，而非交集，不同级标签（例如法官与法条）索引到的案件取交。最后由于取合并前的权重之和，能保证两位法官都出现的案件的权重比只出现了一位法官的案件权重大，更容易出现在头几条搜索结果中。

本测例的结果恰好说明了我们这种标签过滤和结果排序机制的合理性。

需要指出，这种同级取并、不同级取交的过滤方法也有一定局限性，例如在输入的同级标签过多的时候，响应时间和返回结果数可能会过大。

### 3.2.2 相关案件搜索

以案件 137837 为例，在详情页点击 “相关案件” 后能跳转到其相关案件的结果页面：


图 9: 137837 的相关案件搜索结果前三条

前三条结果的相似度都很高：第一条是该案件本身，第二条结果是该案件对应的起诉书，第三条是该案件对应的一审案件。说明我们的相关案件搜索算法具有合理性。

需要指出，我们通过 TopK/TF-IDF 向量化匹配来检索相关案件的算法存在局限性，算法会过于强调文本层面的相似性，而对于案件审判结构相似而字面不尽相同的案件可能无能为力。此外，我们为了响应时间上的权衡，将词向量维数限制在 10，也会导致在一些情况下，某些 TopK/TF-IDF 权重较低而同样重要的词被算法忽略（例如“谋杀”、“婚姻”相对于“唐山市”、“曹新新”权重就更低，但对决定案件性质十分重要）。

后期可以考虑让后端从输入文本中预先识别出这些关键词，然后把它们放到标签/法条/法官的过滤步骤中，来强调这些词的重要性。

# 4 分工合作

在本项目中，郑逢时同学负责前端、后端的开发，赵成钢同学负责了数据清洗、后端的开发。在开发过程中也通过 Issue 驱动的敏捷开发形式进行了大量有关对搜索引擎技术的讨论，体现了对于课程传授内容的思考。


图 10: Issue 记录和讨论

# 5 参考


- 刘奕群老师的课件
- Vue.js 文档
- Vuetify 文档
- Golang Jieba 文档