基于 Elastic Search 的电影垂直搜索引擎服务端设计

应承峻 3170103456

浙江大学计算机科学与技术学院软件工程专业

摘要：垂直搜索引擎是相对通用搜索引擎的信息量大、查询不准确、深度不够等缺点提出来的新的搜索引擎服务模式。通过针对某一特定领域、某一特定人群或某一特定需求提供的有一定价值的信息和相关服务。基于上述的需求，本文提出了一种以 Elastic Search 搜索引擎为核心的电影垂直搜索引擎服务端系统设计方案，通过合适的系统架构设计，使其能够在海量数据、高并发、快速迭代的生产环境下仍能够有较高的可用性和扩展性。

关键词：Elastic Search，垂直搜索引擎，服务端，大数据

1 引言

随着互联网技术和通信技术的发展，人民生活水平的提高，人们对于娱乐的方式也是越来越多元化。电影作为一个一直以来受众广泛的娱乐方式，在各类搜索引擎中都具有很高的搜索量。为了解决一般搜索引擎对于电影这一特定领域的内容存在的搜索不够精确、不够智能化的问题，本文提出了一种电影垂直搜索引擎的服务端设计方案，其能够实现一个为用户提供支持中文和拼音的实时电影信息搜索，可以对筛选结果按照年份、地区等筛选条件筛选，按照评分、时间等信息排序，同时还能根据用户的搜索历史、电影的类型等对用户可能喜欢的电影进行智能分析推荐。

Elastic Search（ES）是一个基于 Lucene 的分布式的、RESTful 风格的搜索和数据分析引擎。ES 可以用于搜索各种文档，用户只需将数据提交到 ES 的数据库中，再通过分词器和过滤器将对应的语句进行分词和过滤，然后搜索引擎就会将分词结果连同该词的权重一并存入数据库中，当用户搜索数据时，ES 会为每个匹配的文档进行打分排名，最后将返回结果呈现给用户。相较于其他开源搜索引擎，ES 具有如下优点：

（1）高可扩展

ES 是一个分布式的文档数据库，具有很强的水平扩展性，它通过向集群中添加更多的节点来分担负载，增加系统的稳定性。能够在极短的时间内使大量数据具有搜索、分析和探索的能力。随着数据和查询量的持续增长，ES 的分布式特性只需我们水平扩展服务器的数量，而不需要我们对多个节点之间的连接通讯进行手动管理，这让数据在生产环境中具有更

大的价值。

（2）高实时性

ES 使用倒排索引的数据结构，在文本经过分词器后，倒排索引能够列出某一个单词出现在任何文档的位置。在存储文档索引后，搜索引擎能够在 1 秒内近乎实时地为所有类型的数据（无论是结构化文本还是非结构化数据）提供搜索和分析。

（3）方便易用

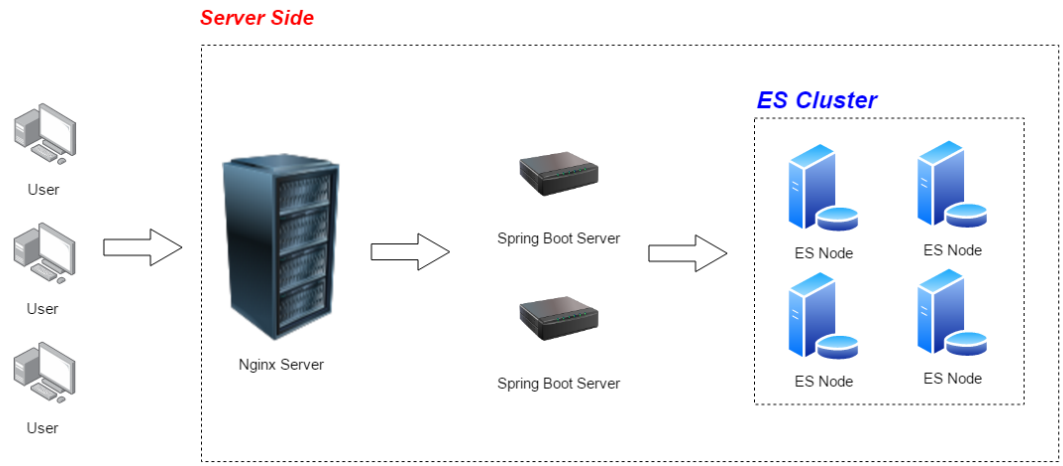
ES 具有动态映射（Dynamic Mapping）的特点，也就是说用户在使用 ES 建立索引时，无需显式地指定文档中每个字段的类型，而是由 ES 代替执行数据类型分析，将输入数据映射成基本数据类型（布尔值、浮点数、文本等）或是复杂数据类型（数组、时间、嵌套对象等），这对于刚开始学习 ES 的用户来说免去了索引配置的烦恼。

2 服务端架构设计

2.1 整体架构设计

服务端运转的整体流程如图 2-1 所示，当用户在浏览器中输入搜索关键词和筛选条件时，请求会以 JSON 数据的格式发送到服务端。在服务端的最外层是一个 Nginx 反向代理和负载均衡服务器，它的主要作用是将来自客户端的请求经过过滤后，转发到计算好的 Tomcat 应用服务器中，服务端处理完数据后再将数据经过 Nginx Server 返回到客户端中。如果涉及到对 ES 数据的操作，对应的 Tomcat 应用服务器将会通过 Restful 接口向 ES 集群中的节点发起请求，ES 节点处理完数据后将会把结果返回到对应的 Tomcat 容器中。

在整一个架构设计中，使用反向代理的好处有：（1）系统背后的服务器不会暴露到任何信息网络上，可以防止攻击者对系统的恶意攻击（2）通过限制对某一 IP 和客户端的访问次数，能够防止 DDos 攻击（3）由于客户端只能看到反向代理服务器的 IP 地址，而无需知道真正提供服务的服务器，服务维护者可以很灵活地改变服务器的配置（4）支持负载均衡，纵向扩展服务器的性能的成本往往非常高，若使用反向代理同时代理多个相同的应用服务器（如部署本项目的 Tomcat），将客户端的请求均匀地分发到各个应用服务器上。这样在高并发量的场景下，各个服务器就能够分摊主服务器的负载，保证系统的稳定性。



（图 2-1 服务端整体架构设计图）

在系统中我们将所有访问/v1/movie/api 路径下的请求全部转发到 Tomcat 容器中，将所有访问根路径下的请求全部转发到前端静态资源目录下，将所有访问/public 路径下的请求全部定位到服务器的静态资源目录。Nginx 的配置如下：

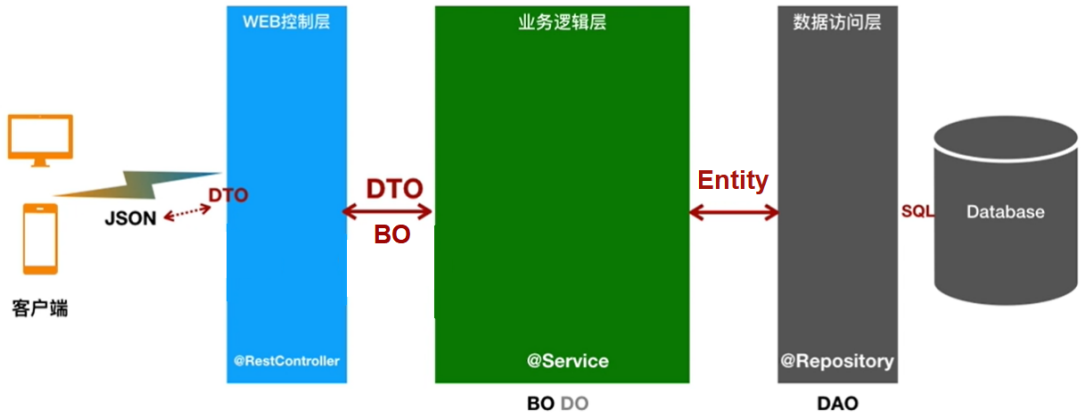
```
upstream images {      # 配置 Tomcat 应用服务器
        server 192.168.128.135:8080;
        server 192.168.128.136:8080;
        ip_hash;    # 根据 IP 取 Hash 值确定转发请求到哪个服务器
}
server {
        listen 443 default_server;
        listen [::]:443 default_server;
        server_name zjufwdx.cn;
        location ~ /movie/public/(.*\.(jpg|jpeg))$ {     # 配置静态资源访问
                alias /var/www/word-cloud/$1;
        }
        location /v1/movie/api {      # 配置转发请求到 Tomcat 应用服务器
                proxy_set_header Cookie $http_cookie;
                proxy_set_header X-Forwarded-Host $host;
                proxy_set_header X-Forwarded-Server $host;
                proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
                proxy_pass http://images;
        }
        location / {        #配置前端界面静态资源目录
                alias /var/www/dist/;
                index index.html index.htm;
                try_files $uri $uri/ =404;
        }
}
```

2.2 后端服务架构

后端服务项目使用 SpringBoot 1.5.x 框架，其主要的作用是封装一层对 ES 搜索引擎的接口调用，用户对搜索引擎的数据操作不是通过直接调用搜索引擎接口而是借助后端服务器间接调用（限定条件，过滤不合法的请求），有效地确保了搜索引擎的数据安全，同时也保

证了各个系统之间职责的明确。

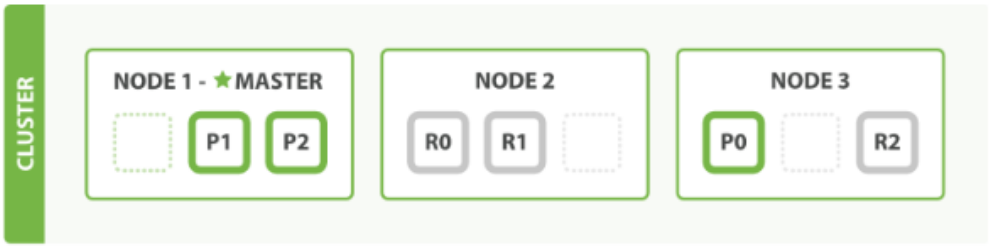
整个代码架构使用严格的 Spring MVC 分层模型，如图 2-2 所示，来自客户端的 JSON 数据以数据传输对象（Data Transfer Object, DTO）被 WEB 控制层 Controller 接收，WEB 控制层根据请求的 URL 将 DTO 传输到对应的业务逻辑层 Service。业务逻辑层真正对请求进行处理，如果涉及到访问数据库，则需要通过实体 Entity（Entity 与数据库中的 Schema 对应）向数据访问层 Repository 发起方法调用，然后获取数据访问层返回的 Entity 数据或查询结果。在业务逻辑层处理完所有的业务逻辑后，将处理结果以业务模型（Business Object, BO）返回给 WEB 控制层，最后由 WEB 控制层给出响应结果。



（图 2-2 后端服务架构）

2.3 搜索引擎集群架构

整个搜索引擎集群由拥有多个相同配置的 ES 节点（实例）组成，如图 2-3 所示，它们共同承载数据和负载的压力，当集群中有节点加入或者移除时，集群将会重新平均分布所有的数据。在整个 ES 集群中会选举出一个主节点（Master Node），它负责管理集群内所有的变更，例如对索引、节点进行增删等操作。由于每个 ES 节点都知道任意文档所处的位置，并且能够将请求直接转发到存储所需文档的节点，因此服务器可以将请求发送到任何节点而无需进行显式地管理。



（图 2-3 搜索引擎集群）

使用多节点的集群架构一方面能够减轻主搜索引擎的压力，另一方面也能够有效地防止单点故障。当在 ES 集群中启动相同的节点时，原节点的副本分片将会分配到这个节点上，这也意味着当集群中的任何节点出现出现问题时，数据都能够完好无损。

3 搜索功能模块设计

3.1 电影索引建立

在 Elastic Search 的搜索引擎中，一个索引的定义由字段名和类型组成。如果该字段的类型是一个对象（Object）或者嵌套对象（Nested），则对象中的各个属性的定义也同样是由字段名和类型组成。以下是定义一个索引基本模板：

```
{
    "mappings": {
        "properties": {
            "字段": {
                "type": "<类型>",
                "analyzer": "<指定的分词器>"
            }
        }
    }
}
```

其中 mapping 指明了这是索引的映射配置，properties 指明里面的内容配置的是索引的字段，type 指明索引的类型，analyzer 指明索引的分词器。ES 中常见的索引类型如下：

（1）text：普通文本，分词器会将这一类文本进行分词，进行倒排索引

（2）keyword：关键词，分词器不会将这类文本进行分词

（3）date：日期，格式为”yyyy-MM-dd”或毫秒数

（4）integer：普通整数

（5）long：长整数

（6）double：双精度浮点数

（7）object：普通对象，在 Elastic Search 的底层存储中，object 对象会被扁平化成数组存储。

（8）nested：嵌套对象，在 Elastic Search 的底层存储中，nested 对象不会被扁平化，而是将嵌套子对象子存储在单独的文档（doc）中

在本文中，我们使用如表 3-1 所示的索引设计，其中 ik_pinyin_analyzer 分词器能够对中文进行分词，将分词后的文本连同其拼音一起作为索引。

（表 3-1 电影索引结构表）

Properties	Type	Analyzer	Description
movieId	keyword		电影 ID
title	text	ik_smart	电影标题
title.pinyin	text	ik_pinyin_analyzer	用于拼音匹配
title.suggestion	keyword		用于搜索提示
alias	text	ik_pinyin_analyzer	别名
poster	keyword		海报图片
staff	object		
staff.staffId	keyword		演员 ID
staff.name	text	ik_pinyin_analyzer	演员姓名
staff.role	text	ik_pinyin_analyzer	演员角色
staff.photo	keyword		演员照片
directors	text	ik_pinyin_analyzer	导演
scriptwriters	text	ik_pinyin_analyzer	脚本
releaseTime	nested		
releaseTime.time	date		发行时间
releaseTime.region	keyword		发行地区
year	integer		发行年份
type	keyword		电影类型
country	keyword		拍摄国家
duration	long		时长
onlineVideo	nested		
onlineVideo.source	keyword		在线视频源
onlineVideo.url	keyword		在线视频 URL
onlineVideo.tag	keyword		在线视频标签
tags	keyword		标签
introduction	text		简介
rate	nested		
rate.source	keyword		评分来源
rate.rating	double		评分值
rate.ratingNum	long		评分数量

3.2 后端应用从搜索引擎获取数据

后端 SpringBoot 应用从搜索引擎中查询数据主要借助官方的 Elasticsearch Rest High Level Client 这一封装库间接地进行 RESTful 接口调用，开发者只需要根据该库公开的 API 方法，编写特定的请求和响应处理对象即可。

使用该库首先需要对连接的 ES 数据源进行配置，配置的代码如下。配置时，需要在整个 SpringBoot 工程的配置文件 application.properties 中分别注入 spring.elasticsearch.host 和 spring.elasticsearch.port 的值，代表连接 ES 集群中任意一个节点的主机 IP 和端口号。在启动

SpringBoot 工程时，该值会被自动注入到 Spring 容器中。

```
@Configuration
public class ElasticConfig {

    @Value("${spring.elasticsearch.port}")
    private String port;

    @Value("${spring.elasticsearch.host}")
    private String host;

    @Bean
    public RestHighLevelClient restHighLevelClient() {
        return new RestHighLevelClient(RestClient.builder(new HttpHost(host, Integer.parseInt(port))));
    }

}
```

整个获取数据的过程可以被拆分为以下五个阶段：

（1）查询参数校验：校验用户输入的参数是否合法，比如涉及到分页的配置中页码是否是正整数，每页的数量是否超过限制。

（2）获取请求对象：通过确定需要搜索到的索引，来生成 SearchRequest 对象，例如，在下面的程序中，构建了一个在电影索引下请求的对象。此外 generateSearchRequest 方法是通用查询 DTO 中的一个覆盖方法，当日后需要扩展搜索的方式时，只需重写该方法即可。例如当加入一个电影资讯类时，并想构建一个能在电影和电影资讯类中进行搜索的请求对象时（SearchRequest 类的构造函数参数是支持多索引的可变参数），只需重新创建一个通用查询 DTO 的子类并覆盖该方法即可。这也充分体现出了设计模式中的开闭原则——软件中的对象（类，模块，函数等等）应该对于扩展是开放的，但是对于修改是封闭的。这样的设计能够充分提高程序的可复用性和可维护性。

```
public SearchRequest generateSearchRequest() {
    return new SearchRequest(Constant.ES_INDEX_MOVIE);
}
```

（3）使用建造者模式包装请求参数：考虑到对于整个系统的需求是关键词搜索、筛选和过滤等功能，我们可以考虑设计一种通用的查询结构——即使用 ES 的布尔查询加上排序选项以及高亮选项，通用的查询结构如下。布尔查询主要由与（Must）、或（Should）、非（Must Not）以及一个过滤器组成。根据通用的查询语法，“+<Key>”表示这个关键字必须在查询结果中出现，“-<Key>”表示这个字段不应该在查询结果中出现，“<Key1> <Key2>”表示表示两个关键字匹配任意一个均可，“:intitle <Key>”表示匹配的查询结果的标题中必须要出

现这个关键字。因此 Must 的作用主要是匹配查询字符串中的“+”符号， Should 字符串主要匹配查询字符串中的空格，Must Not 主要匹配查询字符串中的“-”符号。Filter 的主要作用是过滤器，它不会影响查询结果的评分，但是会将不符合条件的结果进行过滤。比如筛选同时匹配年份在 2020 年国家地区是中国大陆条件的电影。Sort 的主要作用则是指定字段的排序，默认按照匹配程度进行降序排序。HighLight 能够对匹配的字段进行高亮。From 和 Size 则用于对查询结果的起始位置和数量进行限制，即分页。

```
{
    "query": {
        "bool": {
            "must": [],
            "should": [],
            "filter": [],
            "must_not": []
        }
    },
    "sort": [],
    "highlight": {
        "pre_tags": "<em>",
        "post_tags": "</em>",
        "fields": {}
    },
    "from": 0,
    "size": 10
}
```

Elasticsearch Rest High Level Client 为我们提供了所有查询结构的建造器，这使得我们只需按照上面总结出来的模式进行查询条件封装即可。

在解析的过程中，我们又一次用到了 OO（Object Oriented）编程的多态特性，我们对于任何的查询设定一个通用查询 DTO，作为所有查询的基类，它的定义如下。即它包含了最基本的分页信息和查询字符串，以及五个被用于覆盖的重载方法（但不是抽象方法，子类可以选择保留默认配置）。现在对于电影解析类 QueryMovieDto，我们在设计时可以直接继承它的结构，然后加上我们所需要的额外的参数，比如筛选的条件、排序的方式。

在整个处理过程中，generateConfig 方法最先被调用，它生成了我们查询的一些默认配置，比如前面提到的分页的起始位置 from 和返回的数量 size；每个待匹配字段的权重 boost（用于查询参数调优，比如给标题设置更大的权重，因为用户很多时候更关注电影的标题而不是其他信息）；高亮的字段以及排序的方式等等。

```
public class QueryDto {

    protected String query;
    protected Integer page = 1;

    /**
     * 生成默认的索引字段权重配置, 页号, 高亮等配置
     * @return SearchConfigBo
     */
    public SearchConfigBo generateConfig();

    /**
     * 生成默认的索引 SearchRequest 对象，默认搜索全部索引
     * @return SearchRequest
     */
    public SearchRequest generateSearchRequest();

    /**
     * 校验搜索页号和搜索参数
     * @throws Exception       ApiBusinessException
     */
    public void validateSearchRequest();

    /**
     * 生成用于排序的 SortBuilder
     * @return List<SortBuilder<?>>
     * @throws Exception ApiBusinessException
     */
    public List<SortBuilder<?>> generateSortBuilders();

    /**
     * 生成用于筛选的过滤器 Filter
     * @return List<QueryBuilder>
     * @throws Exception ApiBusinessException
     */
    public List<QueryBuilder> generateFilter();
}
```

在生成基本的配置后，通过设计模式中的工厂模式，我们对通用查询 DTO 的查询字符串进行解析，整个查询字符串被解析成为许多的 Token，我们只需将 Token 提供给静态工厂，工厂就会返回我们需要的 AbstractToken 对象。AbstractToken 类如下图所示，它主要包含了一个 parse 方法，通过提供一个基本配置，能够将从查询字符串中解析出的 Must、Should、Filter 等条件注入到配置中。对于生成的每一个 AbstractToken 的子类，我们只需调用其实现的抽象方法将配置注入即可。这种方式在需要添加解析规则时非常容易扩展。

```
public abstract class AbstractToken {
    protected String token;
    public abstract void parse(SearchConfigBo searchConfigBo);
}
```

（4）发起请求并得响应：在包装查询参数完成后，直接调用之前被注入 Spring 容器中的 ES Client 的 request 方法，传入 SearchRequest 对象即可向 ES 发起请求。

（5）从响应中解析数据：ES 返回数据时，我们只需将其返回的 Map 对象进行属性拷贝，映射成我们需要的 BO，然后做一些后续处理即可。这里的 BO 也是一个抽象类，它的 cutOff、highLight 等方法都是可被子类覆盖的方法。

```
public List<QueryResBo> getSearchResponseData(SearchResponse response) throws Exception {
    SearchHit[] hits = response.getHits().getHits();
    List<QueryResBo> queryResBoList = new ArrayList<>();
    for (SearchHit hit : hits) {
        Map<String, Object> attrMap = hit.getSourceAsMap();        // 获取源对象
        QueryResBo bo = QueryResBo.getQueryInstance(hit.getIndex());      // 从工厂中生成目标对象
        if (bo != null) {
            BeanUtils.populate(bo, attrMap);      //拷贝 Map 中数据到对象中
            bo.cutOff();      //截断长内容
            bo.highLight(hit);   //设置高亮
            bo.setScore(hit.getScore());      //设置得分
            queryResBoList.add(bo); //添加到结果集
        } else {
            throw ApiExceptionInstance.SERVER_SIDE_ERROR;
        }
    }
    return queryResBoList;
}
```

4 查询优化与参数调优

4.1 索引结构优化

4.1.1 拼音分词器优化

在整个搜索引擎中，为了支持对中文的搜索和拼音的搜索，我们在 ES 中引入了中文分词和拼音分词插件，但是相应地引入拼音分词的代价就是，它会造成查询准确率的急剧降低和存储空间的急剧上升，因此我们需要对拼音搜索的准确率和召回率进行权衡。

根据 ES 官方文档的描述，输入的内容会分别经过 Character filters （字符过滤器），Tokenizer （分词器），Token filters （token 过滤器）之后才会正式地去生成索引结构。为了支持拼音分词，我们在配置索引中的分词器的时，首先会让它经过 ik_max_word 中文分词器做一个最细粒度的拆分，然后将分好的词再经过 pinyin 的 filter 进行拼音分词，最后存储到索引中。

```
# ES 拼音分词器的配置
"settings": {
    "analysis": {
        "analyzer": {
            "ik_pinyin_analyzer": {
                "type": "custom",
                "tokenizer": "ik_max_word",
                "filter": [
                    "pinyin_filter"
                ]
            }
        },
        "filter": {
            "pinyin_filter": {
                "type": "pinyin",
                "keep_first_letter": true,
                "keep_joined_full_pinyin": true,
                "keep_separate_first_letter": false,
                "none_chinese_pinyin_tokenize": true,
                "keep_full_pinyin": true
            }
        }
    }
}

# 标题的索引结构
"title": {
    "type": "text",
    "analyzer": "ik_max_word",
    "fields": {
        "suggest": {
            "type": "completion"
        },
        "pinyin": {
            "type": "text",
            "analyzer": "ik_pinyin_analyzer"
        }
    }
}
```

基于上述原理，为了减少拼音分词的误差，可以做出的处理是：

（1）在分词的阶段将中文标题和中文标题的拼音视为两个不同的域，同时降低拼音域匹配的权重，即另外为 title 字段设置一个 text 类型的 pinyin 域，将其的权重设置为 title 的 25%。这样假如我们需要通过中文来搜索，毫无疑问匹配到的中文会拥有更高权重，分数高于拼音匹配的结果，能够过滤掉拼音匹配带来不需要的数据；而假如我们需要通过拼音来搜索中文，那么权重较大的 title 字段中所有的中文其实都无法匹配到，而只能通过 title.fields.pinyin 中的字段来匹配到，这样所有的中文匹配到的拼音权重都一样，也就能够正常的进行拼音匹配了（当然这是需要建立在标题中绝大部分内容不会出现拼音的情况。英文

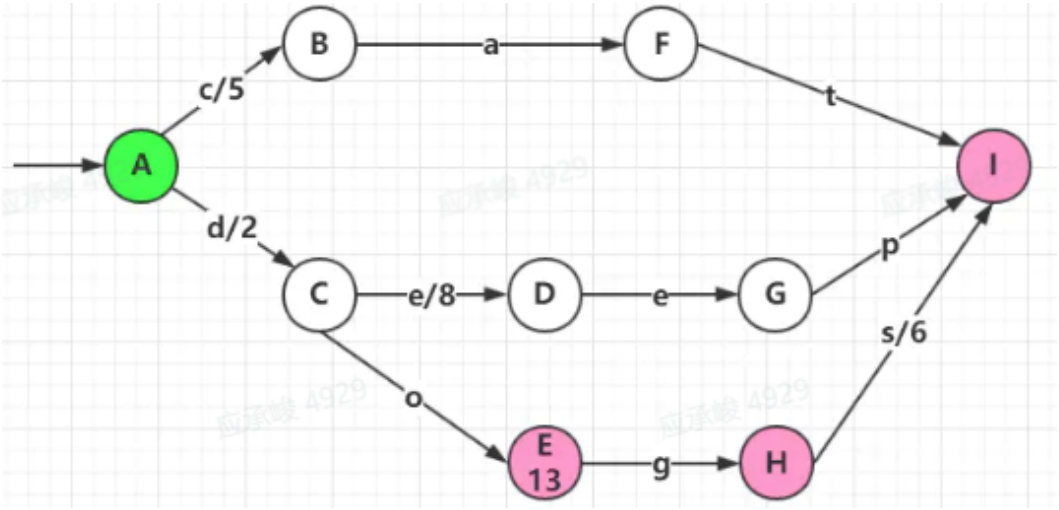
单词和拼音的相似的概率非常小，带来的影响可以忽略不计）

（2）修改拼音分词的匹配选项，只保留拼音的全拼索引，去掉会导致很大误差的首字母缩写索引。在修改索引后，只能通过 zidan 搜索子弹而不能通过 zd 搜索到，这也意味着我们需要以降低召回率的代价来提高准确率。

4.1.2 搜索提示优化

为了实现在搜索框中输入前缀自动补全的功能，有很多可选的实现方案。其中最常见的就是使用正则匹配或是 PrefixQuery，即使用 key*的形式来匹配关键词，但是这样往往要对索引进行全文搜索，其搜索效率在数据量大的时候非常低。

在这样的背景下，为了提高搜索效率，可以使用 ES 中的 Completion Suggester 进行前缀匹配，它通过了改造索引的结构来提升性能。在索引结构的 title 字段中，与前面的拼音一样，我们新加一个 suggest 域，并设置类型为 completion（经过测试研究，我们发现不能对 title 直接进行 completion 类型的设置，它的分词会导致该类型消耗约 20 倍的存储空间）。Completion Suggester 与 ES 中提供的其他 Suggester 相比，采用了不同的数据结构，索引并非通过倒排来完成，而是将分析过的数据编码成 FST 和索引一起存放，FST 有着非常高的数据压缩率和查询效率。对于一个开放状态的索引，FST 会被 ES 整个装载到内存里的，能够进行极快速度的前缀查找。



（图 4-1 FST 编码示意图）

4.2 模糊查询

有时候为了避免用户搜索时打错文字，我们会对查询的关键词进行模糊处理，即允许 2 个字符存在误差，但是同时考虑到在短查询时这样容易导致较大的误差，我们将进行模糊处理的最小长度限制为 6。

4.3 多权重查询

为了支持多索引（电影、资讯、种子）和多字段（标题、内容、演职员、作者等）查询，但同时又为了需要确保我们的查询结果是用户需要的，考虑到用户对不同的索引和不同的字段的查询的频率、权重，因此可以为需要查询的字段设置不同的权重，经过多次参数调优，我们选择了如下的权重：

（表 4-1 多字段查询权重表）

Properties	Type
title	4.0f
title.pinyin	1.0f
alias	3.0f
staff.role	1.6f
staff.name	1.6f
introduction	2.0f
content	2.0f
directors	1.0f
scriptwriters	1.0f
author	1.0f

4.4 重打分与结果过滤

在使用排序的功能时，为了达到严格的排序效果，往往会严重降低查询的准确性，而这往往很难满足用户需要，因此我们需要对排序效果做出一定的牺牲，来换取查询结果的准确性。即我们在对时间进行排序时，不是严格地按照时间进行排序，而是根据其匹配的分值得分乘上时间衰减率，进行重打分计算。

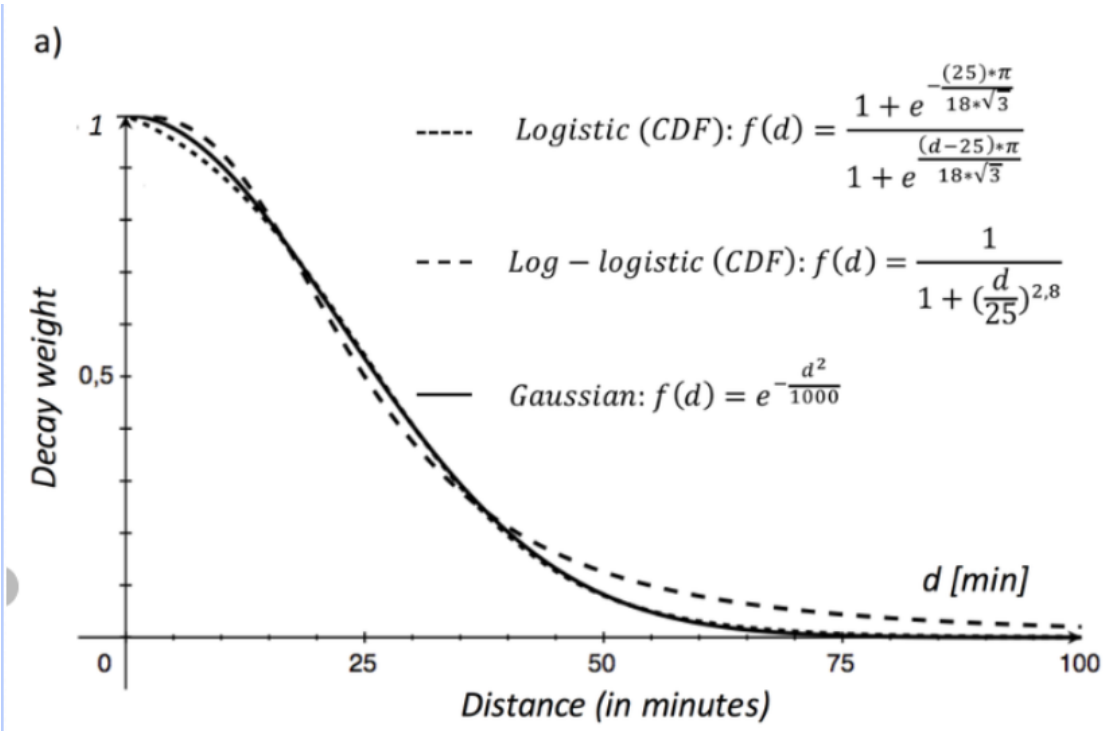
为此我们可以建立一个时间打分模型，通过在 ScriptSort 引入指数衰减曲线和 Gauss 衰减函数曲线分别对评分和时间进行衰减处理。在 Gauss 衰减函数曲线中，可以根据年份、评分的分布情况对下图中衰减函数参数进行微调。

经过参数调优检验，最终我们为年份使用如下的公式进行衰减：

$$f(year) = e^{-\frac{(year-2020)^2}{64}}$$

最终的得分公式如下：

$$f(score, year) = f(year) * sqrt(score)$$



（图 4-2 Gauss 衰减曲线示意图）

5 小结

在设计一个基于 ES 的电影垂直搜索引擎的服务端时，选取合适的技术进行系统架构来设计、合适的设计模式来进行代码编写能极大地提高整个系统的可用性和可维护性。此外，还需要更加关注搜索引擎返回给用户的数据能否满足用户的预期，这就需要后端应用的接口设计者对 ES 的接口调用进行参数调优，通过选用恰当的分词方式、调整每个字段的权重、对筛选的结果根据需求进行加权重打分、过滤分数较低的结果这些途径来提高搜索结果的准确度。

参考文献：

[1] Elasticsearch Reference Version 7.8

[2] Nginx Ducumentation