# 目录

# §1 ElasticSearch

Elasticsearch 是个开源分布式搜索引擎，它的特点有：分布式，零配置，自动发现，索引自动分片，索引副本机制，Restful 风格接口，多数据源，自动搜索负载等。

## §1.1 What's a mapping?

A mapping not only tells ES what is in a field…it tells ES what terms are indexed and searchable.

A mapping is composed of one or more ‘analyzers’（相当于 Lucene 中的 Analyzer）, which are in turn built with one or more ‘filters’. When ES indexes your document, it passes the field into each specified analyzer, which pass the field to individual filters.

Filters are easy to understand: a filter is a function that transforms data. Given a string, it returns another string (after some modifications). A function that converts strings to lowercase is a good example of a filter.

An analyzer is simply a group of filters that are executed in-order. So an analyzer may first apply a lowercase filter, then apply a stop-word removal filter(中文分词作为其中一个 filter). Once the analyzer is run, the remaining terms are indexed and stored in ES. 因此: a mapping is simply a series of instructions on how to transform your input data into searchable, indexed terms.

使用 **curl -XPUT http://localhost:9200/test/item/1 -d '{"name":"zach", "description": "A Pretty cool guy."}'** 时，ES 生成如下的 mapping:

```
mappings: {
   item: {
      properties: {
         description: { type: string }
         name: { type: string }
      }
   }
}
```

ES 猜测到"description" 这列的类型为"string",When ES implicitly creates a "string" mapping, it applies the default global analyzer. 除非显式地更改 analyzer 设置，否则 ES 将使用默认的 Standard

Analyzer.Standard Analyzer will apply the standard token filter, lowercase filter and stop token filter to your field. 这种情况下，句子末尾的. 号和字符'A' 将被丢弃.Importantly, even though ES continues to store the original document in it’ s original form, the only parts that are searchable are the parts that have been run through an analyzer.So, mappings are not data-types , think of them as instructions on how you will eventually search your data. If you care about stop-words like 'a', you need to change the analyzer so that they aren’ t removed.

## §1.2 Constructing more complicated mapping

Setting up proper analyzers in ES is all about thinking about the search query. You have to provide instructions to ES about the appropriate transformations so you can search intelligently.

The first thing that happens to an input query is tokenization , breaking an input query into smaller chunks called tokens. There are several tokenizers available, which you should explore on your own when you get a chance. The Standard tokenizer is being used in this example, which is a pretty good tokenizer for most English-language search problems. You can query ES to see how it tokenizes a sample sentence:

```
curl -X GET "http://localhost:9200/test/_analyze?tokenizer=standard&pretty=true"
-d 'The quick brown fox is jumping over the lazy dog.'
```

Ok, so our input query has been turned into tokens. Referring back to the mapping, the next step is to apply filters to these tokens. In order, these filters are applied to each token: Standard Token Filter, Lowercase Filter, ASCII Folding Filter.

```
curl -X GET "http://localhost:9200/test/_analyze?filter=standard&pretty=true"
-d 'The quick brown fox is jumping over the lazy dog.'
```

As you can see, we specify both a search and index analyzer. These two separate analyzers instruct ES what to do when it is indexing a field, or searching a field. But why are these needed?

```
"partial":{
```

```
    "search_analyzer":"full_name",
    "index_analyzer":"partial_name",
    "type":"string"
}
```

The index analyzer is easy to understand. We want to break up our input fields into various tokens so we can later search it. So we instruct ES to use the new partial_name analyzer that we built, so that it can create nGrams for us.

The search analyzer is a little trickier to understand, but crucial to getting good relevance. Imagine querying for "Race". We want that query to match "race", "races" and "racecar". When searching, we want to make sure ES eventually searches with the token "race". The full_name analyzer will give us the needed token to search with.

If, however, we used the partial_name nGram analyzer, we would generate a list of nGrams as our search query. The search query "Race" would turn into ["ra", "rac", "race"].Those tokens are then used to search the index. As you might guess, "ra" and "rac" will match a lot of things you don't want, such as "racket" or "ratify" or "rapport".

So specifying different index and search analyzers is critical when working with things like ngrams. Make sure you always double check what you expect to query ES with…and what is actually being passed to ES.

## §1.3 Depth into ElasticSearch

当使用如下命令搜索时:

```
curl -XPOST "http://namenode:9200/_search" -d'
{
    "query": {
      "match_all" : {}
    },
    "filter" : {
      "term" : { "director" : "Francis Ford Coppola"}
    }
}'
```

没有任何结果，而使用

```
curl -XPOST "http://namenode:9200/_search" -d'
{
    "query": {
      "match_all" : {}
    },
    "filter" : {
      "term" : { "year" : "1962"}
    }
}'
```

却能输出结果，What's going on here? We've obviously indexed two movies with "Francis Ford Coppola" as director and that's what we see in search results as well. Well, while ES has a JSON object with that data that it returns to us in search results in the form of the _source property .

When we index a document with ElasticSearch it (simplified) does two things: it stores the original data untouched for later retrieval in the form of _source and it indexes each JSON property into one or more fields in a Lucene index. During the indexing it processes each field according to how the field is mapped. If it isn't mapped , default mappings depending on the fields type (string, number etc) is used.

As we haven't supplied any mappings for our index, Elastic-Search uses the default mappings for the director field. This means that in the index the director fields value isn't **"Francis Ford Coppola"**. Instead it's something like **["francis", "ford", "coppola"]**. We can verify that by modifying our filter to instead match "francis" (or "ford" or "coppola").

So, what to do if we want to filter by the exact name of the director? We modify how it's mapped. There are a number of ways to add mappings to ElasticSearch, through a HTTP request that creates by calling the _mapping endpoint. Using this approach, we could fix the above issue by adding a mapping for the "director" field , to instruct ElasticSearch not to analyze (tokenize etc.) the field at all.

```
curl -X PUT namenode:9200/movies/movie/_mapping -d'
{
   "movie": {
      "properties": {
         "director": {
```

```
            "type": "string",
            "index": "not_analyzed"
        }
      }
   }
}'
```

In many cases it's not possible to modify existing mappings. Often the easiest work around for that is to create a new index with the desired mappings and re-index all of the data into the new index. The second problem is that, even if we could add it, we would have limited our ability to search in the director field. That is, while a search for the exact value in the field would match we wouldn't be able to search for single words in the field.

幸运地是存在更简单的办法。We add a mapping that upgrades the field to a multi field. What that means is that we'll map the field multiple times for indexing. Given that one of the ways we map it match the existing mapping both by name and settings that will work fine and we won't have to create a new index.

```
curl -XPUT "namenode:9200/movies/movie/_mapping" -d'
{
   "movie": {
      "properties": {
         "director": {
            "type": "multi_field",
            "fields": {
               "director": {"type": "string's"},
               "original": {"type" : "string", "index" : "not_analyzed"}
            }
         }
      }
   }
}'
```

## §1.4 ElasticSearch 集群设置

This will print the number of open files the process can open on startup. Alternatively, you can retrieve the max_file_descriptors for each node using the Nodes Info API, with:

**curl -XGET 'namenode:9200/_nodes/process?pretty=true'**，下面是删除 ES 上名为 jdbc 的 index 的 Restfule 命令:

**curl -XDELETE 'http://namenode:9200/jdbc/'**

配置文件在 **{$ES_HOME}/config** 文件夹下，**elasticsearch.yml** 和 **logging.yml**，修改 **elasticsearch.yml** 文件中的 cluster.name，当集群名称相同时，每个 ES 节点将会搜索它的伙伴节点，因此必须保证集群内每个节点的 cluster.name 相同，下面是关闭 ES 集群的 Restful 命令：

```
# 关闭集群内的某个ES节点'_local'
$ curl -XPOST 'http://namenode:9200/_cluster/nodes/_local/_shutdown'
# 关闭集群内的全部ES节点
$ curl -XPOST 'http://namenode:9200/_shutdown'
```

注意，如果一台机器上不止一个 ES 在运行，那么通过**./bin/elasticsearch** 开启的 ES 的 http_address 将会使用 9200 以上的接口（形如 9201,9202,···），而相应的 transport_address 也递增（形如: 9301,9302,···），因此，为使用 9200 端口，可使用上述命令关闭其它 ES 进程，可通过 conf 目录下的 log 文件来查看某些端口是否被占用。**elasticsearch.yml** 文件存在如下配置信息：

(1) node.master: true,node.data: true，允许节点存储数据，同时作为主节点；

(2) node.master: true,node.data: false，节点不存储数据，但作为集群的协调者；

(3) node.master: false,node.data: true，允许节点存储数据，但不作为主节点；

(4) node.master: false,node.data: false，节点不存储数据，也不作为协调者，但作为搜索任务的一个承担者；

(5) cluster.name: HadoopSearch，node.name: "ES-Slave-02",HadoopSearch 必须相同，但 node.name 每个节点可以自由设置；

如想将 ES 作为一个服务，需要从 github 上下载 elasticsearch-servicewrapper，然后调用 chkconfig，将其添加到/etc/rc[0~6].d/中。

```
curl -L https://github.com/elasticsearch/elasticsearch-servicewrapper/
archive/master.zip > master.zip
unzip master.zip
cd elasticsearch-servicewrapper-master/
mv service /opt/elasticsearch/bin
/opt/elasticsearch/bin/service/elasticsearch install
## 如果想卸载该服务调用:
/opt/elasticsearch/bin/service/elasticsearch remove
## 如果想让ES开机启动
chkconfig elasticsearch on
## 如果想现在开启ES服务
service elasticsearch start
```

配置完后，可通过 *curl -X GET 'http://192.168.50.75:9200/_cluster/nodes?pretty'* 命令，查询集群下的节点信息。

为连接 hive 与 ES，运行 hive 后，在 hive 命令行内执行 **add jar /opt/elasticsearch-hadoop-1.3.0/dist/elasticsearch-hadoop-1.3.0.jar;** 或者 hdfs 上的 jar 包:**add jar hdfs://namenode:9000/elasticsearch-hadoop-1.3.0.jar** 可加载 elasticsearch-hadoop 插件，使用该插件的具体操作如下:

```
DROP TABLE IF EXISTS artist_1;
CREATE EXTERNAL TABLE artists_1 (
  cardid STRING, date STRING, time STRING)
STORED BY 'org.elasticsearch.hadoop.hive.ESStorageHandler'
TBLPROPERTIES('es.resource' = 'liubo/artists/',
              'es.host' = '192.168.50.75',
              'es.mapping.names' = 'text:time'
);
-- 集群下应使用'192.168.50.75'，而非'localhost'(es-hadoop的默认值)
-- insert data to Elasticsearch from another hive table
INSERT OVERWRITE TABLE artists_1
SELECT * FROM cable.temptable;
```

下面的代码是将 Mysql 中的表导入到 ES 中，建立名为 jdbc 的 index，表名称为 jiangsu。

```
curl -XPUT 'localhost:9200/_river/jiangsu/_meta' -d '{
    "type" : "jdbc",
    "jdbc" : {
        "driver" : "com.mysql.jdbc.Driver",
        "url" : "jdbc:mysql://192.168.50.75:3306/jsyx",
        "user" : "root",
        "password" : "123456",
        "sql" : "select * from jiangsu"
    },
    "index" : {
        "index" : "jdbc",
        "type" : "jiangsu"
    }
}'
```

## §1.5 ES 性能优化

一个 Elasticsearch 节点会有多个线程池，但重要的是下面四个:

- 索引（index）: 主要是索引数据和删除数据操作（默认是 cached 类型）;
- 搜索（search）: 主要是获取，统计和搜索操作（默认是 cached 类型）;

- 批量操作（bulk）：对索引的批量操作，尚且不清楚它是不是原子性的，如果是原子的，则放在 MapReduce 里是没有问题的；
- 更新（refresh）：主要是更新操作，如当一个文档被索引后，何时能够通过搜索看到该文档；

在建立索引（index 相当于数据库，type 相当于数据库中的表）的过程中，需要修改如下配置参数：

- index.store.type： mmapfs。因为内存映射文件机制能更好地利用 OS 缓存；
- indices.memory.index_buffer_size： 30% 默认值为 10%，表示 10% 的内存作为 indexing buffer；
- index.translog.flush_threshold_ops： 50000，当写日志数达到 50000 时，做一次同步；
- index.refresh_interval:30s，默认值为 1s，新建的索引记录将在 1 秒钟后查询到；

```
curl -XPUT 'http://namenode:9200/hivetest/?pretty' -d '{
    "settings" : {
       "index" : {
       "refresh_interval" : "30s",
       "index.store.type": "mmapfs",
       "indices.memory.index_buffer_size": "30%",
       "index.translog.flush_threshold_ops": "50000"
        }
    }
}'
```

但上述设置性能低下，也不知 why？ES 索引的过程相对 Lucene 的索引过程多了分布式数据的扩展，而 ES 主要是用 tranlog 进行各节点间的数据平衡，因此设置"index.translog.flush_threshold_ops" 为"100000"，意思是当 tranlog 数据达到多少条进行一次平衡操作，默认为 5000，而这个过程相对而言是比较浪费资源的，必要时可以将这个值设为 -1 关闭，进而手动进行 tranlog 平衡。"refresh_interval" 是刷新频率，设置为 30s 是指索引在生命周期内定时刷新，一但有数据进来就在 Lucene 里面 commit，因此其值设置大些可以提高索引效率。另外，如果有副本存在，数据也会马上同步到副本中去，因此在索引过程中，将副本设为 0，待索引完成后将副本个数改回来。

```
curl -XPUT 'namenode:9200/hivetest/' #新建一个名为hivetest的索引
curl -XPOST 'namenode:9200/hivetest/_close' #关闭索引，为修改参数做准备
curl -XPUT 'namenode:9200/hivetest/_settings?pretty' -d '{
   "index" : {
      "refresh_interval" : "30s",
      "index.translog.flush_threshold_ops": "100000",
      "number_of_replicas" : 0
   }
}'
curl -XPOST 'namenode:9200/hivetest/_open'
```

Linux 主机上硬盘空间有限，经常发现 root 目录下已没有可利用磁盘空间，为此，将 ES 的日志和数据输出目录设置在/home 目录下，修改 config 目录下的 elasticsearch.yml 文件中的选项，其中 path.data 为索引文件存放目录，path.work 为临时文件存放目录，path.logs 为日志文件存放目录。

```
path.data: /home/elasticSearch/data
path.work: /home/elasticSearch/work
path.logs: /home/elasticSearch/logs
```

## §1.6 Lucene 全文检索引擎

### §1.6.1 Lucene 文件结构

Files and detailed format:

- .tim: Term Dictionary
- .tip: Term Index
- .doc: Frequencies and Skip Data
- .pos: Positions
- .pay: Payloads and Offsets

The .tim file contains the list of terms in each field along with per-term statistics (such as docfreq) and pointers to the frequencies, positions, payload and skip data in the .doc, .pos, and .pay files.

The .doc file contains the lists of documents which contain each term, along with the frequency of the term in that document.

The .pos file contains the lists of positions that each term occurs at within documents. It also sometimes stores part of

payloads and offsets for speedup.TermPositions are order by term (terms are implicit, from the term dictionary), and position values for each term document pair are incremental, and ordered by document number.

### §1.6.2 Lucene 的基本分词过程

Analyzer 类是所有分词器的基类，它是个抽象类，所有的子类必须实现它，它提供了如下方法：

```
@Override
protected TokenStreamComponents createComponents(String fieldName,
        final Reader reader) {
    Tokenizer tokenizer = new AnsjTokenizer(new IndexAnalysis(reader),
        reader, filter, pstemming);
    return new TokenStreamComponents(tokenizer);
}
```

TokenStream 是能够产生词序列的类，有两个类型的子类 TokenFilter 和 Tokenizer，Tokenizer 通过读取汉字字符串创建词序列 List<word>，其中每一 word 将作为 Lucene 索引的 key，Tokenizer 还记录每个 word 的偏移值，以及在文档中属于第几个 word（$0, \cdots, n$）等，而 TokenFilter 则对产生的词序列进行转换（比如过滤等）.

### §1.6.3 开发 ES 插件

插件一般情况下是一个 zip 文件，它包含了若干 Jar 包，为安装插件，依赖于 Maven 的安装。Maven 工程的 main 目录里须包含三个目录(java,resources,assembly)。新建 src/main/resources/es-plugin.properties 文件，文件内容为：

```
plugin=${project.groupId}.${project.artifactId}
```

告诉 Maven 插件为当前工程，然后在配置文件 pom.xml 的 <build>/<plugins>/<plugin>/<artifactId>maven-assembly-plugin</artifactId> 标签中添加如下内容，因为 maven-assembly-plugin 插件负责打包 Maven 工程：

```
<descriptors>
   <descriptor>
    ${basedir}/src/main/assembly/plugin.xml
  </descriptor>
</descriptors>
```

然后新建 src/main/assembly/plugin.xml 文件，写入如下语句：

```
<?xml version="1.0"?>
<assembly>
  <id>plugin</id>
  <formats>
     <format>zip</format>
  </formats>
  <includeBaseDirectory>false</includeBaseDirectory>
  <dependencySets>
     <dependencySet>
         <outputDirectory>/</outputDirectory>
         <useProjectArtifact>true</useProjectArtifact>
         <useTransitiveFiltering>true</useTransitiveFiltering>
         <excludes>
             <exclude>org.elasticsearch:elasticsearch</exclude>
          </excludes>
      </dependencySet>
  </dependencySets>
</assembly>
```

```
|
|——— pom.xml
|——— src
|    |——— main
|        |——— assembly
|        |    |——— plugin.xml
|        |——— java
|        |    |——— org
|        |        |——— elasticsearch
|        |            |——— plugin
|        |                |——— example
|        |                    |——— ExamplePlugin.java
|        |——— resources
|            |——— es-plugin.properties
|
```

## §1.7 ES 与 hive

Hive 从 0.8.0 版本后支持两个虚拟列:**INPUT__FILE__NAME**（即 mapper 任务的输出文件名）和 **BLOCK__OFFSET__INSIDE__FILE**（即读取记录在当前文件的全局偏移量）。对于块压缩文件，就是当前块的文件偏移量，即当前块的第一个字节在文件中的偏移量。Simple Example:

```
use cable;
select INPUT__FILE__NAME, cardid, BLOCK__OFFSET__INSIDE__FILE from test;
select min(BLOCK__OFFSET__INSIDE__FILE),count(INPUT__FILE__NAME) from test
  where BLOCK__OFFSET__INSIDE__FILE < 12000 group by cardid;
select * from test where BLOCK__OFFSET__INSIDE__FILE < 12000;
```

开始时将'es.mapping.id' = 'id' 写成了'es.mapping.id ' = 'id'，由于多了一个空格，程序一直没有获得期望的结果。'es.mapping.id' 属性告诉

Hive，使用 hive1 表中的 id 列的值作为 ElasticSearch 的 _id 域，使用剩余几列的值作为 ElasticSearch 的 _source。表 hive1 不是存储在 hive 内部，所以使用 **EXTERNAL** 关键字，对于使用 **EXTERNAL** 存储的表，必须提供 **STORED BY** 关键字，否则 hive 无法确定用哪个类来存储外部表。

在向 ElasticSearch 添加索引过程中，必须提供 id 域，否则当某 MapTask 失败若干次时，相同的记录可能会插入很多遍。因为如果不指定 id，ES 会给当前记录随机分配一个 id，这点类似于往数据库中 insert 记录时，必须保证对相同 record，其主键也相同，所以在使用 hive 向 ElasticSearch 中插入记录时，其 id 域为相应行在 hdfs 文件中的位置。

```
set mapred.max.split.size=32000000;
add jar hdfs://namenode:9000/user/liubo/1.jar;
DROP TABLE IF EXISTS hive1;
CREATE EXTERNAL TABLE hive1(
   id STRING,
   cardId STRING,
   playDate STRING,
   playTime STRING,
   channel STRING,
   program STRING
)
STORED BY 'org.elasticsearch.hadoop.hive.ESStorageHandler'
TBLPROPERTIES('es.resource' = 'eshive/hive1/',
              'es.host' = '192.168.50.75',
              'es.mapping.id' = 'id'
);
INSERT OVERWRITE TABLE hive1 SELECT BLOCK__OFFSET__INSIDE__FILE,* FROM testData2;
```

# §2 中文分词

词是最小的能够独立活动的有意义的语言成分，英文单词之间是以空格作为自然分界符的，而汉语是以字为基本的书写单位，词语之间没有明显的区分标记，因此，中文词语分析是中文信息处理的基础与关键。Lucene 对中文的处理是基于自动切分的单字切分，或者二元切分。除此之外，还有最大切分（包括向前、向后、以及前后相结合）、最少切分、全切分等等。

中文分词算法可分为三大类：基于字典、词库匹配的分词方法；基于词频度统计的分词方法和基于知识理解的分词方法。第一类方法应用词典匹配、汉语词法或其它汉语知识进行分词，如：最大匹配法、最小分词方法等。这类方法简单、分词效率较高，但汉语语言现象复杂丰富，词典的完备性、规则的一致性等问题使其难以适应开放的大规模文本的分词处理。第二类基于统计的分词方法则基于字和词的统计信息，如把相邻字间的信息、词频及相应的共现信息等应用于分词，由于这些信息是通过调查真实语料取得的，因而基于统计的分词方法具有较好的实用性。基于知识理解的分词方法主要基于句法、语法分析，并结合语义分析，通过对上下文内容所提供信息的分析对词进行定界，它通常包括三个部分：分词子系统、句法语义子系统、总控部分。在总控部分的协调下，分词子系统可以获得有关词、句子等的句法和语义信息来对分词歧义进行判断。这类方法试图让机器具有人类的理解能力，需要使用大量的语言知识和信息。由于汉语语言知识的笼统、复杂性，难以将各种语言信息组织成机器可直接读取的形式。因此目前基于知识的分词系统还处在试验阶段。

Bag of words，也叫做“词袋”，词袋模型假定对一个文本，忽略其词序和语法，句法，将其仅仅看做是一个词集合，或者说是词的一个组合，文本中每个词都是独立的随机变量，不依赖于其他词是否出现，这种假设虽然对自然语言进行了简化，便于模型化，但是其假定在有些情况下是不合理的，例如在新闻个性化推荐中，采用词袋模型就会出现问题。例如用户甲对“南京醉酒驾车事故”这个短语很感兴趣，采用词袋模型忽略了顺序和句法，则认为用户甲对“南京”、“醉酒”、“驾车”和“事故”感兴趣，因此可能推荐出和“南京”，“公交车”，“事故”相关的新闻，这显然不合理。

## §2.1 机械分词法

基于字典、词库匹配的分词方法（机械分词法），这种方法按照一定策略将待分析的汉字串与一个“充分大的”机器词典中的词条进行匹配，若在词典中找到某个字符串，则匹配成功。识别出一个词，根据扫描方向的不同分为正向匹配和逆向匹配。根据不同长度优先匹配的情况，分为最大（最长）匹配和最小（最短）

匹配。根据与词性标注过程是否相结合，又可以分为单纯分词方法和分词与词性标注相结合的一体化方法。包含如下两种方法:

最大正向匹配法 (Maximum Matching Method) 通常简称为 MM 法。其基本思想为: 假定分词词典中的最长词有 i 个汉字字符，则用被处理文档的当前字串中的前 i 个字作为匹配字段，查找字典。若字典中存在这样的一个 i 字词，则匹配成功，匹配字段被作为一个词切分出来。如果词典中找不到这样的一个 i 字词，则匹配失败，将匹配字段中的最后一个字去掉，对剩下的字串重新进行匹配处理……如此进行下去，直到匹配成功，即切分出一个词或剩余字串的长度为零为止。这样就完成了一轮匹配，然后取下一个 i 字字串进行匹配处理，直到文档被扫描完为止，其算法描述如下:

**Algorithm 1:** Maximum Matching Method

```
Input: 文档 doc，字典 vocabulary
Output: 分词序列 list<word>
当前位置计数器 pos=0, n= 文档长度,i=vocabulary 中最长词长度;
for pos = 0; pos < n; ; do
    str= 从 pos 开始的连续 i 个字符，str 作为匹配字段;
    while str.length > 1 or str ∉ vocabulary do
        去掉 str 最后一个字符;
    end
    if str ∈ vacabulary then
        匹配字段 str 作为一个词被切分，放入分词序列 List<word>;
        pos += str 的长度;
    else
        str 作为一个单字切分出来，放入分词序列 List<word>;
        pos += 1;
    end
end
```

逆向最大匹配法 (Reverse Maximum Matching Method) 通常简称为 RMM 法。RMM 法的基本原理与 MM 法相同，不同的是分词切分的方向与 MM 法相反，而且使用的分词辞典也是逆序词典，其中每个词条都将按逆序方式存放。实际处理时，先将文档倒排处理，生成逆序文档，然后根据逆序词典，对逆序文档用正向最大匹配法处理。汉语中偏正结构较多，若从后向前匹配，可以适当提高精确度。逆向最大匹配法比正向最大匹配法的误差要小。统计结果表明，单纯使用正向最大匹配的错误率为 $\frac{1}{169}$，单纯使用逆向最大匹配的错误率为 $\frac{1}{245}$。例如切分字段“硕士研究生产”(假定最大长度为 5)，正向最大匹配法的结果会是“硕士研究生/产”，而逆向最大匹配法利用逆向扫描，可得到正确的分词结果“硕士/研究/生产”，其算法描述如下:

最大匹配算法是一种基于分词词典的机械分词法，不能根据文档上下文的语

**Algorithm 2:** Reverse Maximum Matching Method

```
Input: 文档 doc，字典 vocabulary
Output: 分词序列 List<word>
foreach word ∈ vocabulary do
    vocabulary− = word;
    倒排 word，例如“硕士”被倒排为' 士硕';
    vocabulary+ = word;
end
倒排 doc，例如“硕士研究生产”被倒排为' 产生究研士硕';
List<word>= 调用 Maximum Matching Method;
foreach word ∈ List < word > do
    倒排 word;
end
```

义特征来切分词语，对词典的依赖性较大，所以在实际使用时，难免会造成一些分词错误，为了提高系统分词的准确度，可以采用正向最大匹配法和逆向最大匹配法相结合的分词方案（即双向匹配法）。

## §2.2 MMSEG 算法

MMSEG 是中文分词中一个常见的、基于词典的分词算法，简单、效果相对较好。由于它的简易直观性，实现起来不是很复杂，运行速度也比较快。总的来说现在的中文分词算法，大概可以笼统的分为两大类：一种基于词典的，一种是非基于词典的。基于词典的分词算法比较常见，比如正向／逆向最大匹配，最小切分（使一句话中的词的数量最少）等。具体使用的时候，通常是多种算法合用，或者一种为主、多种为辅，同时还会加入词性、词频等属性来辅助处理（运用某些简单的数学模型）。

非基于词典的算法，一般主要是运用概率统计、机器学习方面的方法，目前常见的是 CRF（Conditional random field）。此类方法可以让计算机根据现成的资料，“学习”如何分词。具体的实现可参考（http://nlp.stanford.edu/software/segmenter.shtml）。基于词典的方法，实现、部署比较容易，但是分词精度有限，且对于未登录词（词典里没有的词语）识别较差；非基于词典的方法，对未登录词识别效果较好，能够根据使用领域达到较高的分词精度，但是实现比较复杂，需要大量的前期工作。

MMSEG 是一种基于词典的分词算法，以正向最大匹配为主，多种消除歧义的规则为辅。根据作者在原文中的阐述，对 MMSEG 的解释分为“匹配算法（Matching algorithm）”和“消除歧义的规则（Ambiguity resolution rules）”这两部分。“匹配算法”是说如何根据词典里保存的词语，对要切分的语句进行匹配（正向？逆向？粒度？），“消除歧义的规则”是说当一句话可以这样分，也可以那样分

的时候，用什么规则来判定使用哪种分法，比如“设施和服务”这个短语，可以分成“设施/和服/务”，也可以分成“设施/和/服务”，选择哪个分词结果，就是“消除歧义的规则”的功能。

MMSEG 的“匹配方法”有两种：Simple 方法，即简单的正向匹配，根据开头的字，列出所有可能的结果。比如“一个劲儿的说话”，可以得到“一个”，“一个劲”，“一个劲儿”，“一个劲儿的”这四个匹配结果（假设这四个词都包含在词典里）。Complex 方法，匹配出所有的“三个词的词组”（原文中使用了 chunk，这里感觉用“词组”比较合适），即从某一既定的字为起始位置，得到所有可能的“以三个词为一组”的所有组合。比如“研究生命起源”，可以得到“研/究/生”，“研/究/生命”，“研究生/命/起源”，“研究/生命/起源”这些“词组”（根据词典，可能远不止这些，仅此举例）。“消除歧义的规则”有四个，依次使用这四个规则过滤，直到只有一种结果或者四个规则使用完毕。这个四个规则分别是：

(1) Maximum matching (最大匹配)，有两种情况，分别对应于使用“simple”和“complex”的匹配方法。对“simple”匹配方法，选择长度最大的词，用在上文的例子中，选择“一个劲儿的”；对“complex”匹配方法，选择长度最大的那个词组，上文的例子中即“研究生/命/起源”或者“研究/生命/起源”。

(2) Largest average word length（最大平均词语长度）。经过规则 (1) 过滤后，如果剩余的词组超过 1 个，那就选择平均词语长度最大的那个（平均词长 = 词组总字数 / 词语数量）。比如“生/活水/平”(4/3=1.33)，“生活/水/平”(4/3=1.33)，“生活/水平”(4/2=2)，根据此规则，就可以确定选择“生活/水平”这个词组。

(3) Smallest variance of word lengths（词语长度的最小变化率），由于词语长度的变化率可以由标准差反映，所以此处直接套用标准差公式即可。比如“研究/生命/起源”(标准差 $=\sqrt{\frac{(2-2)^2+(2-2)^2+(2-2^2)}{3}}$)，“研究生/命/起源”（标准差 $=\sqrt{\frac{(2-3)^2+(2-1)^2+(2-2)^2}{3}}$），于是选择“研究/生命/起源”这个词组。

(4) Largest sum of degree of morphemic freedom of one/character words，其中 degree of morphemic freedom 可以用一个数学公式表达：$\log_e^{frequency}$，即词频的自然对数。这个规则的意思是“计算词组中的所有单字词词频的自然对数，然后将得到的值相加，取总和最大的词组”。比如：“设施/和服/务”，“设施/和/服务”，这两个词组中分别有“务”和“和”这两个单字词，假设“务”作为单字词时的频率是 5，“和”作为单字词时

候的频率是 10，对 5 和 10 取自然对数，然后取最大值者，所以取“和”字所在的词组，即“设施/和/服务”。

取自然对数的原因在于，词组中单字词词频总和可能一样，但是实际的效果并不同，比如：A/BBB/C（单字词词频，A:3，C:7）,DD/E/F（单字词词频，E:5，F:5）表示两个词组，A、C、E、F 表示不同的单字词，如果不取自然对数，单纯就词频来计算，那么这两个词组是一样的（3+7=5+5），所以这里取自然对数，以表区分（ln(3)+ln(7) < ln(5)+ln(5)）。

这四个过滤规则中，如果使用 simple 的匹配方法，只能使用第一个规则过滤，如果使用 complex 的匹配方法，则四个规则都可以使用。实际使用中，一般都是使用 complex 的匹配方法 + 四个规则过滤。（simple 的匹配方法实质上就是正向最大匹配，实际中很少只用一个方法）。MMSEG 分词是一个“直观”的分词方法，它把一个句子“尽可能长（指所切分的词尽可能的长）”“尽可能均匀”地切分，稍微想象一下，便感觉与中文的对称语法习惯比较相符。如果对分词精度要求不是特别高，MMSEG 是一个简单、可行、快速的方法。

基于词典的分词算法中，词典的结构对速度的影响是比较大的（词典的结构一般决定了匹配的方法和速度）。一般的构造词典的方法有很多，比如“首字索引 + 整词二分”，将所有词语的首字用哈希做索引，然后将词体部分排序，使用二分查找。这样的方法可行，但不是最快的。对于词典匹配，trie 结构一般是首选。trie 也有一些变种和实现方法，对于大量静态数据的匹配（比如词典，一旦创建，便很少去修改里面的内容，故称之为“静态”），一般采用“双数组 trie 树（double array trie tree）”，比如 Darts 类库。

MMSEG 的分词效果与词典关系较大（词典里有哪些词语，以及词频的精确度），尤其是词典中单字词的频率。可以根据使用领域，专门定制词典（比如计算机类词库，生活信息类词库，旅游类词库等），尽可能的细分词典，这样得到的分词效果会好很多。同时也可以通过词典达到一些特殊目的（地名分词等）。关于词库，可以参考“搜狗”的细胞词库（http://pinyin.sogou.com/dict/）以及其提供的语料库（可以根据其划分好的语料库，统计某一方面的词频，http://www.sogou.com/labs/resources.html）。

## §2.3 基于词的频度统计的分词方法

基于词的频度统计的分词方法不依靠词典，而是将文章中任意两个字同时出现的频率进行统计，次数越高的就可能是一个词。它首先切分出与词表匹配的所有可能的词，运用统计语言模型和决策算法决定最优的切分结果。

### §2.3.1 隐马尔科夫模型与维特比

机械分词算法的分词效果有限，比如下面这样一句话：产量三年中将增长两倍。按照机械分词的算法，它可能会被分成这样一种形式：产量 | 三年 | 中将 | 增长 | 两倍。机械分词将‘中将’分成了一个词，的确‘中将’在词典中是有这么一个词，但在这句话中将它们划分成一个词显然是不合理的，于是产生了一种新的方法，基于隐马尔科夫模型的维特比算法。

中文分词可以看成这样一种问题，即给定一个字串（句子），找到对应的词串（分词）。给定的字串用公式表示就是 $Y = y_1, y_2, \cdots, y_m$，其中 $y_i$ 表示一个字。需要的词串表示为：$X = x_1, x_2, \cdots, x_n$，其中 $x_i$ 表示一个词。期望目标很简单，就是希望找到最可能的 $X$，使得后验概率 $P(X|Y)$ 最大，也就是说需要找到一种最可能的分词方法，使得在句子 $Y$ 存在的前提条件下，该分词结果出现的概率最大。根据贝叶斯公式有：

$$P(X|Y) = \frac{P(X)P(Y|X)}{P(Y)}$$

那么为了最大化 $P(X|Y)$，经过变化就变成了下面的形式，因为在给定 $Y$ 的情况下，$P(Y)$ 可以看作 1：

$$\begin{aligned}
\hat{X} &= max(\frac{P(X)P(Y|X)}{P(Y)}) && (1)\\
&= max(P(X)P(Y|X)) && (2)\\
&= max\{P(x_1 \cdots x_n)P(y_1 \cdots y_m|x_1 \cdots x_n)\} && (3)\\
&= max\{P(x_1 \cdots x_n)\} && (4)
\end{aligned}$$

$P(y_1 \cdots y_m|x_1 \cdots x_n) = P(y_1 \cdots y_{m_1}|x_1)P(y_{m_1+1} \cdots y_{m_2}|x_2) \cdots P(y_{m_k+1} \cdots y_m|x_n)$，事实上，$P(y_1 \cdots y_m|x_1 \cdots x_n)$ 的值总为 1，因为当给定分词序列 $x_1 \cdots x_n$ 时，$y_1 \cdots y_m$ 就已经确定了。描述的分词模型是一种马尔科夫模型，句子中某个词不是凭空出现的，而和它前面的 $n$ 个单词有关联，这变成了 $n$ 阶马尔科夫模型。考虑最简单的一阶马尔科夫模型，也就是说某个分词的出现只和它紧邻的前一个分词有关，而与之前的分词无关，同时分词相对于字串的条件概率又是独立分布的（独立输出假设）。那么有如下公式：

$$\begin{aligned}
P(x_1 x_2 \cdots x_n) &= P(x_1 x_2 \cdots x_{n-1})P(x_n|x_1 x_2 \cdots x_{n-1}) && (5)\\
&= P(x_1 x_2 \cdots x_{n-1})P(x_n|x_{n-1}) && (6)\\
&= P(x_1 x_2 \cdots x_{n-2})P(x_{n-1}|x_{n-2})P(x_n|x_{n-1}) && (7)\\
&= P(x_1)P(x_2|x_1) \cdots P(x_n|x_{n-1}) && (8)
\end{aligned}$$

其中，$X$（分词）是隐变量，$Y$（字串）是观测变量，最终的目标是找到一个词串 $x_1 \cdots x_n$，使得能最大化 $P(x_1 x_2 \cdots x_n)$。由于当给定 $P(x_1 x_2 \cdots x_{n-1})$ 后，再知道 $P(x_n|x_{n-1})$ 后，就可以计算出 $P(x_1 x_2 \cdots x_n)$，因此计算最大概率的方法实际上是一种动态规划算法，先计算 $P(x_0 x_1)$，再计算 $P(x_0 x_1 x_2)$，以此类推。对 $P(x_n|x_{n-1})$ 的计算可通过对语料（如人民日报）的分析获得，例如求'P(总理 | 国务院)'，可先统计' 国务院' 的出现次数 $count$，然后统计' 总理' 紧挨着' 国务院' 的出现次数 $count_1$，则 P(总理 | 国务院)=$\frac{count_1}{count}$。

### §2.3.2 维特比算法

想象一个乡村诊所，村民有着非常理想化的特性，要么健康要么发烧，他们只有问诊所的医生才能知道是否发烧。聪明的医生通过询问病人的感觉诊断他们是否发烧。村民只回答他们感觉正常、头晕或冷。假设一个病人每天来到诊所并告诉医生他的感觉。医生相信病人的健康状况如同一个离散马尔可夫链。病人的状态有两种“健康”和“发烧”，但医生不能直接观察到，这意味着状态对他是“隐含”的。每天病人会告诉医生自己有以下几种由他的健康状态决定的感觉的一种：正常、冷或头晕。这些是观察结果。整个系统为一个隐马尔可夫模型 (HMM)。医生知道村民的总体健康状况，还知道发烧和没发烧的病人通常会抱怨什么症状。

```
states = ('Healthy', 'Fever')
observations = ('normal', 'cold', 'dizzy')
start_probability = {'Healthy': 0.6, 'Fever': 0.4}
transition_probability = {
   'Healthy' : {'Healthy': 0.7, 'Fever': 0.3},
   'Fever' : {'Healthy': 0.4, 'Fever': 0.6}}
emission_probability = {
   'Healthy' : {'normal': 0.5, 'cold': 0.4, 'dizzy': 0.1},
   'Fever' : {'normal': 0.1, 'cold': 0.3, 'dizzy': 0.6}}
```

这段代码中，起始概率 start_probability 表示病人第一次到访时医生认为其所处的 HMM 状态，他唯一知道的是病人倾向于是健康的。这里用到的特定概率分布不是均衡的，如转移概率大约是'Healthy': 0.57, 'Fever': 0.43。转移概率 transition_probability 表示潜在的马尔可夫链中健康状态的变化。当天健康的病人仅有 30% 的机会第二天会发烧。放射概率 emission_probability 表示每天病人感觉的可能性，即 $P(normal|Healthy) = 0.5$,$P(cold|Healthy) = 0.4$,$P(dizzy|Healthy) = 0.1$。假如他是健康的，50% 会感觉正常。病人连续三天看医生，医生发现第一天他感觉正常，第二天感觉冷，第三天感觉头晕。于是医生产生了一个问题：怎样的健康状态序列最能够解释这些观察结果，维特比算法可以回答这个问题。

**Algorithm 3:** 维特比算法

**Input:** $obs, States, startP, transP, emitP$

**Output:** 最大概率 $prob$, 健康状态序列 $P[]$

$V[][] = 0; Path[][] = \emptyset$;
**for** $t = 0; t < len; t++$ **do**
  **if** $t == 0$ **then**
    **for** $y_i \in States$ **do**
      $V[0][y_i] = startP[y_i] * P(obs[0]|y_i)$ ;
      $Path[0][y_i] = \emptyset$;
    **end**
  **else**
    **for** $y_i \in States$ **do**
      $Path[t][y_i] = y_i$;
      $V[t][y_i] = 0$;
      **for** $y_j \in States$ **do**
        **if** $V[t][y_i] \leq V[t-1][y_j] \times P(y_i|y_j) \times P(obs[t]|y_i)$ **then**
          $V[t][y_i] = V[t-1][y_j] \times P(y_i|y_j) \times P(obs[t]|y_i)$;
          $Path[t][y_i] = y_j$;
        **end**
      **end**
    **end**
  **end**
**end**
$prob = max(V[len-1][0], V[len-1][1], \cdots)$;
$y = y_i$, 其中 $V[len-1][y_i]$ 为 $V[len-1][*]$ 中最大值;
定义堆栈 $stack = \emptyset, stack.push(y)$;
**for** $t = len-1; t > 0; t--$ **do**
  $y = Path[t][y]$;
  $stack.push(y)$;
**end**
**repeat**
  $P.add(stack.pop())$;
**until** $stack.empty()$;
**return** $prob, P[]$

viterbi 算法中，obs 为观察结果序列，如 ['normal','cold','dizzy']，states 为一组隐含状态，start_p 为起始状态概率，trans_p 为转移概率，而 emit_p 为放射概率。维特比算法揭示了观察结果 ['normal','cold','dizzy'] 最有可能由状态序列 ['Healthy','Healthy','Fever'] 产生。换句话说，对于观察到的活动，病人第一天感到正常，第二天感到冷时都是健康的，而第三天发烧了。在实现维特比算法时需注意许多编程语言使用浮点数计算，当 p 很小时可能会导致结果下溢。避免这一问题的常用技巧是在整个计算过程中使用对数概率，在对数系统中也使用了同样的技巧。当算法结束时，可以通过适当的幂运算获得精确结果。

### §2.3.3 实施方案

计划解决的问题

1．繁简转换，（已实现，使用繁简转换表）

2．拼音转汉字，

3．同音词拼写错误，先把汉字转成拼音，然后进行可能的比对，比如输入中华人名共和国 --> 中华人民共和国。

4．英文拼写错误，（暂时不作考虑）

5．形近词错误，（这个需要吗，当用户使用五笔或记错某个字的样子时）

拼音转汉字想法是较为直接的，建立一个以拼音为 term 的查询词索引，posting list 中只保存查询频率最高的 K 个查询词，如

| | |
|---|---|
| jiujingkaoyan | “久经考验”，“酒精考验” |
| zhijiucaotang | “子九草堂', “子久草堂” |
| xufuniuza | “徐福牛杂”，“许府牛杂”，“徐府牛杂” |
| shoujichongzhi | “手机冲值”，“手机充值” |

这一步可以在自动提示中使用，但自动提示与它的区别是，自动提示在拼音输入了一部分的情况下也要提示，比如输入“xufu”就要提示“许府牛杂”。

同音词拼写错误也基于同样的想法，但是需要一个可能出错的查询词列表，这个列表可以为借鉴于下列几种情况:

(1) 以 carot 为例，返回有 carot 的文档，也返回一些包含纠错后的 term carrot 和 torot 的文档。

(2) 与 (1) 相似，但仅当 carot 不在词典中时，返回纠错后的结果。

(3) 与 (1) 相似，但仅当包含 carot 的文档数小于一个预定义的阈值时，即当原始查询返回文档数小于预定义的阈值时，搜索引擎给出纠错后的词列表。

情况 (1) 相当于是对所有查询都进行纠错处理，发现那些搜索比较少的，就给出一个纠错提示，比如“天浴”搜索次数比较少，而“天娱”搜索次数比较多，那么在用户搜索“天浴”时就提示“天娱”，即使“天浴”也是一个正常的查询词。情况 (2) 就是当查询没有获得文档，才对它进行纠错处理，然后查询相应结果。情况 (3) 是一个查询它返回的文档数少于一个预定义阈值时，才进行纠错处理。

英文拼写错误，在 lucene 中已有贡献者实现了 spellchecker 模块，主要算法有：Jaro Winkler distance，Levenstein Distance(Edit Distance)，NGram Distance。但 Lucene 中的实现过于简单，使用两两比较，时间复杂性是 $O(n^2)$。

形近字错误，形近字一般是用户记错了形声字，或是使用五笔的用户输入错误。在网上可以下载 SunWb_mb 文件，它里面包含五笔的编码和笔画的编码，但字根比如“马”比“口”笔画更多，也更有代表性，但在这种方法中却是相同的。

方言纠错，可以用 soudex 进行纠错