第 2 章
搜索引擎理解语义

搜索引擎根据用户输入的关键词或者问题查询文本。为了快速返回结果，搜索引擎往往对待查询的文本建立倒排索引。对于用户输入问题和索引库中的文本理解有助于提升查询结果的准确性。本章首先介绍处理文本的整体流程，然后介绍基于文法的语言模型，以及 N 元模型。

2.1 处理文本

对于要处理的文章，首先可以切分成句子，然后再做分词和词性标注等处理。可以使用 java.text.BreakIterator 把文本分成句子。BreakIterator.getSentenceInstance 返回按标点符号的边界切分句子的实例。BreakIterator 支持多种语言的文本处理。简单切分出中文句子的方法是：

```
    String stringToExamine = "这是一种高效富集甲基化 DNA 的方法。在该方法中，可将与 5mC 特异性结合的抗体加入到变性的基因组 DNA 片段中，从而使甲基化的基因组片断免疫沉淀，形成富集。通过与已有 DNA 微芯片技术相结合，从而进行大规模 DNA 甲基化分析。该方法简便，特异性高，适合 DNA 甲基化组学(DNA Methylome)的分析。通过以上论述，不难看到检测甲基化的方法不断涌现，一方面说明其研究难度之大，另一方面也说明种种方法都有其局限性，诸如对酶的依赖，PCR 扩增的问题，芯片数据分析的标准化问题，等等。综上所述，各种表观基因组学技术方法使我们可以绘制出诸如 DNA 甲基化以及组蛋白修饰模式的详细图谱，为我们研究甲基化的生物学功能，以及在肿瘤生成中的作用，以致肿瘤预防、诊断、治疗和预后方面提供更多信息。然而，为了实现这个宏远目标，需要的不仅仅是支助的增加，还有国际同行的密切合作。";
```

```
//根据中文标点符号切分
BreakIterator boundary = BreakIterator.getSentenceInstance(Locale.CHINESE);
//设置要处理的文本
boundary.setText(stringToExamine);
int start = boundary.first(); //开始位置
for (int end = boundary.next(); end != BreakIterator.DONE;
       start = end, end = boundary.next()) {
   //输出子串，也就是一个句子
   System.out.println(stringToExamine.substring(start, end));
}
```

2.2 基于文法的语言模型

可以使用 JSGF(Java Speech Grammar Format)描述的语言模型来匹配标准问答集。JSGF 中的每个文件只定义一个语法。每个语法包含两部分：语法头和语法体。

语法头格式：#JSGF version char-encoding locale；

例如：#JSGF V1.0 ISO8859-5 en；

声明语法头后，需要指定语法名称。语法名称格式如下：

```
grammar grammarName;
```

接下来定义语法体、语法体定义规则。规则定义格式如下：

```
public <ruleName> = ruleExpansion;
```

例如，定义一个名为 greet 的规则：

```
public <greet> = Hello;
```

一个简单的规则扩展可以引用一个或多个符号或规则。

```
public <greet> = Hello;
public <completeGreet> = <greet> World;
```

一个简单的“ Hello World ”语法文件。

```
#JSGF V1.0;

grammar simpleExample;

public <greet> = Hello;

public <completeGreet> = <greet> World;
```

还可以在语法文件中添加注释。

```
//单行注释

/*多行注释*/
```

```
    /**
    *文档注释
    * @author luogang
    */
```

JSGFKit(https://github.com/ExpandingDev/JSGFKit)是一个 JSGF 语言模型的实现。JSGFKit 使用 Grammar 类作为持有语法规则的主要容器。使用 JSGFKit 的代码如下：

```
    Grammar g = new Grammar();
    //创建一个 Rule 对象
    Rule greetRule = new Rule("greet",
         new RequiredGrouping(new RuleReference("greetWord")),
         new RequiredGrouping(new RuleReference("name")));
    //增加一个规则到文法库
    g.addRule(greetRule);
    g.addRule(new Rule("greetWord",
            new AlternativeSet(new Token("hello"), new Token("hi"))));
    g.addRule(new Rule("name",
         new AlternativeSet(new Token("peter"), new Token("john"),
                       new Token("mary"), new Token("anna"))));

    String text = g.compileGrammar();
    System.out.println(text);
```

输出结果如下：

```
    #JSGF V1.0 UTF-8 zh;
    grammar default;
    public <greet> = (<greetWord>) (<name>);
    public <greetWord> = hello | hi;
    public <name> = peter | john | mary | anna;
```

2.3　正则表达式查找文本

正则表达式\w 与任何单词字符匹配，包括下画线，而\w+则匹配一个英文单词。可以在文本编辑器 Nodepad++中测试正则表达式匹配。

java.util.regex 包提供了对正则表达式的支持。其中的 Pattern 类代表一个编译后的正则表达式。通过 Matcher 类根据给定的模式查找输入字符串。通过调用 Pattern 对象的 matcher 方法得到一个 Matcher 对象。使用正则表达式提取字符串的例子如下：

```
    String example = "This is my small example string which I'm going to use for
pattern matching.";
    Pattern pattern = Pattern.compile("\\w+");
    Matcher matcher = pattern.matcher(example);
    //检查所有的出现
    while (matcher.find()) {
       System.out.print("开始位置: " + matcher.start());
```

```
        System.out.print(" 结束位置: " + matcher.end() + " ");
        System.out.println(matcher.group());
    }
```

用正则表达式检查 E-mail 的格式。用这样的正则表达式：

```
\w+@[\w+\.\w+]+
```

可以匹配上 luogang@lietu.com 这样的文本。

还需要“ . ”和“ - ”两个字符。例如邮箱 zhangshna.Mr@163.com，在@符号之前还有个点“ . ”。

检查 E-mail 格式的语句：

```
String mailTo = "abc@sina.com.cn";
System.out.println(mailTo.matches( "[\\w[.-]]+@[\\w[.-]]+\\.[\\w]+" )); //输出 true
```

正则表达式的原理是有限状态自动机。dk.brics.automaton(http://www.brics.dk/automaton/)包含有限状态自动机的实现。BasicAutomata.makeChar 方法生成接收单个字符的自动机。

```
Automaton a = BasicAutomata.makeChar('W');        //创建一个字符 W 组成的自动机
```

repeat 方法重复多次。例如重复字符 A～Z 至少一次：

```
Automaton a = BasicAutomata.makeCharRange('A', 'Z');
Automaton c = BasicOperations.repeat(a,1);        //指定最少重复次数
System.out.println(BasicOperations.run(c, "WW")); //输出 true
System.out.println(BasicOperations.run(c, "WWW")); //输出 true
```

BasicOperations.concatenate 方法连接两个自动机。例如：

```
Automaton a = BasicAutomata.makeCharRange('A', 'Z');
Automaton b = BasicAutomata.makeChar('@');
Automaton c = BasicOperations.concatenate(a, b);
System.out.println(BasicOperations.run(c, "A@")); //输出 true
System.out.println(BasicOperations.run(c, "AW")); //输出 false
```

使用 Automaton 类匹配数字：

```
Automaton num = Automaton.minimize((new RegExp("[0-9]+")).toAutomaton());
System.out.println(BasicOperations.run(num, "12356756700")); //输出 true
```

很多时候对匹配对象有更多的要求。根据上下文过滤匹配结果要用到环视结构。如果条件位于要提取的信息的后面，则叫作向前看表达式，否则叫作向回看。

一个向回看的表达式从模式开始，直到向回看的表达式结束为止。

```
(?<=X)        X，按条件 X 向回寻找
```

例如，查找"http://"后面的文本写做：(?<=http://)。完整的例子如下：

```
//查找"http://"后面的文本
Pattern pat = Pattern.compile( "(?<=http://)\\S+" );
```

```
    String str = "The Java2s website can be found at  http://www.java2s.com. There,
you can find some Java examples.";

    Matcher matcher = pat.matcher(str);
    while (matcher.find())
      System.out.println(":" + matcher.group() + ":");
```

下面的例子提取网页中的链接：

```
    String pageContents = "<a href=\"http://www.lietu.com\">猎兔</a>";
    Pattern p = Pattern.compile("<a\\s+href\\s*=\\s*\"?(.*?)[\"|>]",
                        Pattern.CASE_INSENSITIVE); //忽略大小写
    Matcher m = p.matcher(pageContents);
    while (m.find()) {//打印网页中所有的链接
       String link = m.group(1).trim();
       System.out.println(link);
    }
```

有些链接的形式是：

```
    <a href='http://www.lietu.com'>猎兔</a>
```

为了更好地匹配单引号，可以把模式修改成：

```
    "<a\\s+href\\s*=\\s*[\"|']?(.*?)[\"|\'>]"
```

匹配“ 2009-12-6 ”这样的日期可以使用如下的正则表达式：

```
    Pattern p = Pattern.compile("\\d{2,4}-\\d{1,2}-\\d{1,2}");
    Matcher m = p.matcher(inputStr);
    if(m.find()){
       String strDate = m.group();
    }
```

其他的一些匹配日期的正则表达式有：“ \\d{2,4}/\\d{1,2}/\\d{1,2} ”和“ \\d{2,4}年\\d{1,2}月\\d{1,2}日 ”，以及“ \\d{2,4}\\.\\d{1,2}\\.\\d{2,4} ”。

2.4　中文词语切分与词性标注

英语、法语和德语等西方语言通常采用空格或标点符号将词隔开，具有天然的分隔符，所以词的获取简单，但是为了深入地理解语义，仍然需要标注出每个词的词性。中文、日文和韩文等东方语言，虽然句子之间有分隔符，但词与词之间没有分隔符或者分隔符不够多，所以需要靠程序切分出词。

中文分词是中文自然语言理解的基础，这里首先介绍中文分词的接口与使用方法，然后介绍最长匹配中文分词和 *N* 元中文分词的实现。

2.4.1 使用中文分词

如果不需要切分出词性，则可以用如下简单的接口返回词：

```
String text = "科技进展";
Segmenter seg = new Segmenter(text);          //切分文本
String word;                                  //保存词
while ((word = seg.nextWord()) != null){      //返回一个词
   System.out.println(word);                  //输出切分出的词
};
```

如果需要标注出文本词性，则可以选用输出标注词性的分词接口。为了方便指明词的词性，词性标注程序往往给每个词性编码。例如，根据英文缩写，把“形容词”编码成 a，名词编码成 n，动词编码成 v……表 2-1 是完整的词性编码表。

表 2-1 词性编码表

代 码	名 称	举 例
a	形容词	最/d 大/a 的/u
ad	副形词	一定/d 能够/v 顺利/ad 实现/v 。/w
ag	形语素	喜/v 煞/ag 人/n
an	名形词	人民/n 的/u 根本/a 利益/n 和/c 国家/n 的/u 安稳/an 。/w
b	区别词	副/b 书记/n 王/nr 思齐/nr
c	连词	全军/n 和/c 武警/n 先进/a 典型/n 代表/n
d	副词	两侧/f 台柱/n 上/f 分别/d 雄踞/v 着/u
dg	副语素	用/v 不/d 甚/dg 流利/a 的/u 中文/nz 主持/v 节目/n 。/w
e	叹词	嗬/e ！/w
f	方位词	从/p 一/m 大/a 堆/q 档案/n 中/f 发现/v 了/u
g	语素	例如 dg 或 ag
h	前接成分	目前/t 各种/r 非/h 合作制/n 的/u 农产品/n
i	成语	提高/v 农民/n 讨价还价/i 的/u 能力/n 。/w
j	简称略语	民主/ad 选举/v 村委会/j 的/u 工作/vn
k	后接成分	权责/n 明确/a 的/u 逐级/d 授权/v 制/k
l	习用语	是/v 建立/v 社会主义/n 市场经济/n 体制/n 的/u 重要/a 组成部分/l 。/w
m	数词	科学技术/n 是/v 第一/m 生产力/n
n	名词	希望/v 双方/n 在/p 市政/n 规划/vn
ng	名语素	就此/d 分析/v 时/ng 认为/v
nr	人名	建设部/nt 部长/n 侯/nr 捷/nr
ns	地名	北京/ns 经济/n 运行/vn 态势/n 喜人/a

续表

代　码	名　称	举　例
nt	机构团体	[冶金/n　工业部/n　洛阳/ns　耐火材料/l　研究院/n]nt
nx	字母专名	ATM/nx　交换机/n
nz	其他专名	德士古/nz　公司/n
o	拟声词	汩汩/o　地/u　流/v　出来/v
p	介词	往/p　基层/n　跑/v　。/w
q	量词	不止/v　一/m　次/q　地/u　听到/v　，/w
r	代词	有些/r　部门/n
s	处所词	移居/v　海外/s　。/w
t	时间词	当前/t　经济/n　社会/n　情况/n
tg	时语素	秋/tg　冬/tg　连/d　旱/a
u	助词	工作/vn　的/u　政策/n
ud	结构助词	有/v　心/n　栽/v　得/ud　梧桐树/n
ug	时态助词	你/r　想/v　过/ug　没有/v
uj	结构助词的	迈向/v　充满/v　希望/n　的/uj　新/a　世纪/n
ul	时态助词了	完成/v　了/ ul
uv	结构助词地	满怀信心/l　地/uv　开创/v　新/a　的/u　业绩/n
uz	时态助词着	眼看/v　着/uz
v	动词	举行/v　老/a　干部/n　迎春/vn　团拜会/n
vd	副动词	强调/vd　指出/v
vg	动语素	做好/v　尊/vg　干/j　爱/v　兵/n　工作/vn
vn	名动词	股份制/n　这种/r　企业/n　组织/vn　形式/n　，/w
w	标点符号	生产/v　的/u　5G/nx　、/w　8G/nx　型/k　燃气/n　热水器/n
x	非语素字	生产/v　的/u　5G/nx　、/w　8G/nx　型/k　燃气/n　热水器/n
y	语气词	已经/d　30/m　多/m　年/q　了/y　。/w
z	状态词	势头/n　依然/z　强劲/a　；/w

使用词性编码输出给句子标注词性的结果，例如：“ 不/d 忘/v 群众/n 疾苦/n 温暖/v 送/v 进/v 万/m 家/q ”。

调用输出词性标注的接口：

```
String text = "地球是一颗美丽的蓝色星球";

WordToken[] tokens = SentProcessor.tag(text);

System.out.println("输出标注结果：");
for (WordToken w : tokens) {
```

```
        System.out.println(w.term()+"|"+w.type());  //输出切分出来的词和词性
    }
```

2.4.2 正向最大长度匹配法

假如要切分“印度尼西亚地震”这个词组，希望切分出“印度尼西亚”，而不希望切分出“印度”这个词。正向找最长词是正向最大长度匹配的思想。倾向于写更短的词，除非必要，才用长词表述，所以倾向切分出长词。

正向最大长度匹配的分词方法实现起来很简单。每次从词典找和待匹配串前缀最长匹配的词，如果找到匹配词，则把这个词作为切分词，待匹配串减去该词，如果词典中没有词匹配上，则按单字切分。例如，检索树结构的词典中包括如下 8 个词语：

大　大学　大学生　活动　生活　中　中心　心

输入：“大学生活动中心”，首先匹配出开头的最长词“大学生”，然后匹配出“活动”，最后匹配出“中心”。切分过程如图 2-1 所示。

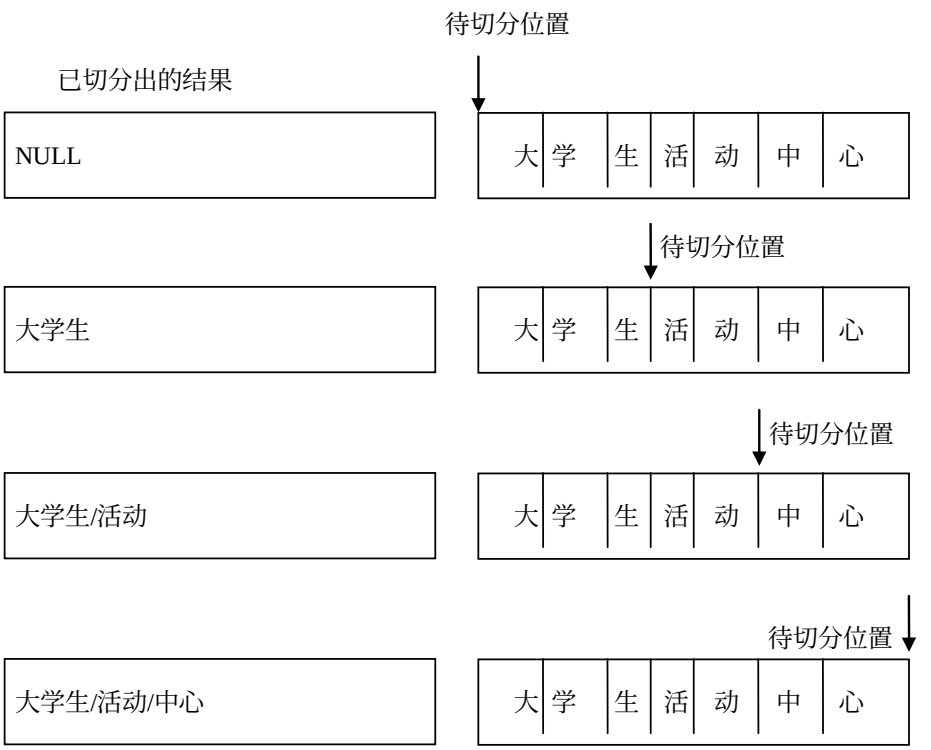

图 2-1　正向最大长度匹配切分过程

最后分词结果为：“ 大学生/活动/中心 ”。

在分词类 Segmenter 的构造方法中输入要处理的文本。然后通过 nextWord 方法遍历单词，其中，text 变量记录切分文本；offset 变量记录已经切分到哪里。分词类基本实现如下：

```
    public class Segmenter {
       String text = null; //切分文本
       int offset; //已经处理到的位置

       public Segmenter(String text) {
          this.text = text; //更新待切分的文本
          offset = 0; //重置已经处理到的位置
       }

       public String nextWord() { //得到下一个词，如果没有，则返回 null
          //返回最长匹配词，如果没有匹配上，则按单字切分
       }
    }
```

使用 Apache Commons Configuration 读入词表相关的配置信息，然后根据配置信息加载词表。build.gradle 文件增加依赖项：

```
    dependencies {
       compile group: 'org.apache.commons', name: 'commons-configuration2',
version: '2.4'
       compile group: 'commons-beanutils', name: 'commons-beanutils', version:
'1.9.3'
    }
```

将配置信息写入配置文件：

```
    Configurations configs = new Configurations();
    Configuration config = configs.properties(new File("config.properties"));
    String dicDir = config.getString("dicDir");  //从配置文件读取词典路径
    System.out.println(dicDir);
```

2.4.3　未登录串识别

切分结果中，英文和数字要连在一起，不管这些英文串或者数字串是否在词典中。例如“ Twitter 正式发布音乐服务 Twitter#Music ”这句话，即使词典中没有“ Twitter ”这个词，切分出来的结果也应该把 Twitter 合并在一起。另外，对于像[ATM 机]这样的英文和汉字混合的词也要合并在一起。

吃苹果时，比发现苹果中有一条虫更糟糕的是，发现里面只有半条虫。如果“ 007 ”在词表中，则会把“ 0078999 ”这样的数字串切分成多段。为了把一些连续的数字和英文切分到一起，需要区分全数字组成的词和全英文组成的词。如果匹配上了全数字组成

的词，则继续往后看还有没有更多的数字。如果匹配上了全英文组成的词，则继续往后看还有没有更多的字母。

匹配数字的有限状态自动机：

```
Automaton num = BasicAutomata.makeCharRange('0', '9').repeat(1);
num.determinize(); //转换成确定自动机
num.minimize();  //最小化
```

匹配英文单词的有限状态自动机：

```
Automaton lowerCase = BasicAutomata.makeCharRange('a', 'z');
Automaton upperCase = BasicAutomata.makeCharRange('A', 'Z');
Automaton c = BasicOperations.union(lowerCase, upperCase);
Automaton english = c.repeat(1);
english.determinize();
english.minimize();
```

匹配日期的有限状态自动机：

```
Automaton a = BasicAutomata.makeCharRange('0', '9');
Automaton b = a.repeat(2,4);
Automaton yearUnit = BasicAutomata.makeChar('年');
Automaton yearNum = BasicOperations.concatenate(b, yearUnit);
Automaton monUnit = BasicAutomata.makeChar('月');

Automaton twoNum = a.repeat(1,2);
Automaton monNum = BasicOperations.concatenate(twoNum, monUnit);
Automaton yearWithMon = BasicOperations.concatenate(yearNum,monNum);

Automaton dayUnit = BasicAutomata.makeChar('日');
Automaton dayNum = BasicOperations.concatenate(twoNum, dayUnit);
Automaton yearMonDay = BasicOperations.concatenate(yearWithMon,
      dayNum.optional());

Automaton finalDate = BasicOperations.union(yearNum,yearMonDay);
finalDate.determinize();
```

可以根据 Automaton 实例得到有限状态转换，例如得到匹配时间的有限状态转换：

```
public static FST createDate() throws Exception {
   Automaton dateAutomaton = AutomatonFactory.getCnDate();
   FST fstDate = new FST(dateAutomaton, "t"); //时间类型
   return fstDate;
}
```

如下方法返回同时匹配日期和数字的有限状态转换：

```
public static FST createAll()throws Exception  {
   FST dateFST = createDate(); //日期
   FST simpleFST = createSimple(); //数值
   FSTUnion union = new FSTUnion(dateFST, simpleFST);
   return union.union();
}
```

用于原子切分的 SplitPoints 类：

```
    public class SplitPoints {
       public BitSet endPoints;              //可结束点
       public BitSet startPoints;            //可开始点
       public HashSet<POSType>[] atomPOS;

       public SplitPoints(int senLen) {
          endPoints = new BitSet(senLen);    //存储所有可能的切分点
          startPoints = new BitSet(senLen);  //存储所有可能的切分点
          atomPOS = new HashSet[senLen];     //存储可能的词性
       }

       @Override
       public String toString() {
          return "SplitPoints [endPoints=" + endPoints + "\r\n startPoints="
                + startPoints + "]";
       }
    }
```

根据句子返回切分词图：

```
    public AdjList getLattice(String sentence) {
       int atomCount = sentence.length();

       //原子切分
       SplitPoints splitPoints = fstSeg.splitPoints(sentence);

       AdjList g = new AdjList(atomCount + 1); //初始化在 Dictionary 中词组成的图
       sucNode = new CnToken[g.verticesNum];
       prob = new double[g.verticesNum];       //节点概率

       int start = 0;
       int currentEnd = splitPoints.endPoints.nextSetBit(0);

       while (start >= 0) {
          TernarySearchTrie.PrefixRet prefix = new TernarySearchTrie.PrefixRet();
          boolean matchRet = dic.matchAll(sentence, start, prefix, splitPoints.
endPoints);

          if (matchRet) {//匹配上
            for (WordEntry word : prefix.values) {
                int end = start + word.word.length();//词的结束位置
                double logProb = Math.log(word.freq) - Math.log(dic.n);
                CnToken tokenInf =
                      new CnToken(start, end, logProb, word.word, word.types);
                g.addEdge(tokenInf);
            }

            start = splitPoints.startPoints.nextSetBit(start + 1);
            currentEnd = splitPoints.endPoints.nextSetBit(currentEnd + 1);
```

```
          } else {
             double logProb = Math.log(1) - Math.log(dic.n);

             HashSet<POSType> types = null;

             if(splitPoints.atomPOS[start]==null ) {
                 types = new HashSet<POSType>();
                 types.add(new POSType("n", 1)); //默认为名词
             }
             else {
                 types = splitPoints.atomPOS[start];
             }

             g.addEdge(
      new CnToken(start, currentEnd, logProb, sentence.substring(start, currentEnd),
types));
             start = splitPoints.startPoints.nextSetBit(start + 1);
             currentEnd = splitPoints.endPoints.nextSetBit(currentEnd + 1);
          }
       }
       return g;
    }
```

2.4.4 基本的 *N* 元模型

两个词可以组合成一个词的情况叫作组合歧义。例如：“ 上海/银行 ”和“ 上海银行 ”。最大长度匹配算法无法正确切分组合歧义。例如，会把 “ 请在一米线外等候 ” 错误地切分成 “ 一/米线 ” 而不是 “ 一/米/线 ”。

对于输入字符串 *C* “ 有意见分歧 ”，有下面两种切分可能：

S_1: 有/ 意见/ 分歧/

S_2: 有意/ 见/ 分歧/

这两种切分方法分别叫作 S_1 和 S_2。如何评价这两个切分方案？哪个切分方案更有可能在语料库中出现就选择哪个切分方案。

计算条件概率 $P(S_1|C)$和 $P(S_2|C)$，然后根据 $P(S_1|C)$和 $P(S_2|C)$的值来决定选择 S_1 还是 S_2。

因为联合概率 $P(C,S)= P(S|C)P(C)= P(C|S)P(S)$，所以有

$$P(S \mid C) = \frac{P(C \mid S) \times P(S)}{P(C)}。$$

这也叫作贝叶斯公式。$P(C)$是字串在语料库中出现的概率。比如说语料库中有 1 万个句子，其中有一句是 “ 有意见分歧 ” 那么 $P(C)=P($“ 有意见分歧 ”$)=$万分之一。

在贝叶斯公式中，$P(C)$只是一个用来归一化的固定值，所以实际分词时并不需要计算。

$P(C|S)$是由从词串 S 恢复到汉字串 C 的概率，该值为 1，即 $P(C|S_1)=P(C|S_2)=1$。因此，比较 $P(S_1|C)$和 $P(S_2|C)$的大小变成比较 $P(S_1)$和 $P(S_2)$的大小，即

$$\frac{P(S_1 \mid C)}{P(S_2 \mid C)} = \frac{P(S_1)}{P(S_2)}$$

因为 $P(S_1)=P$(有,意见,分歧)> $P(S_2)=P$(有意,见,分歧)，所以选择切分方案 S_1 而不是 S_2。

在具体的分词过程中，输入是一个字符串 $C=C_1,C_2,\cdots,C_n$，输出是一个词串 $S=w_1,w_2,\cdots,w_m$，其中 $m \le n$。对于一个特定的字符串 C，会有多个切分方案 S 与之对应，分词的任务就是在这些 S 中找出一个切分方案 S，使得 $P(S|C)$的值最大。$P(S|C)$就是由字符串 C 产生切分 S 的概率。最可能的切分方案为

$$\begin{aligned}\mathrm{BestSeg}(C) &= \arg\max_{S \in G} P(S \mid C) = \arg\max_{S \in G} \frac{P(C \mid S)P(S)}{P(C)} \\ &= \arg\max_{S \in G} P(S) = \arg \max_{w_1,w_2,\cdots,w_m \in G} P(w_1, w_2, \cdots, w_m)\end{aligned}$$

也就是对输入字符串切分出最有可能的词序列。

这里的 G 表示切分词图。待切分字符串 C 中的某个子串构成一个词 w，把这个词看成是从开始位置 i 到结束位置 j 的一条有向边。把 C 中的每个位置看成点，词看成边，可以得到一个有向图，这个图就是切分词图 G。

概率语言模型分词的任务是：在全切分所得的所有结果中求某个切分方案 S，使得 $P(S)$为最大。那么，如何来表示 $P(S)$呢？为了简化计算，假设每个词之间的概率是上下文无关的，则

$$P(S) = P(w_1, w_2, \cdots, w_m) \approx P(w_1)P(w_2)\cdots P(w_m)$$

式中，$P(w)$就是词 w 出现在语料库中的概率。例如：

$$P(S_1) = P\text{(有,意见,分歧)} \approx P\text{(有)}\ P\text{(意见)}\ P\text{(分歧)}$$

对于不同的 S，m 的值是不一样的。一般来说，m 越大，$P(S)$会越小。也就是说，分出的词越多，概率越小。这符合实际的观察，如最大长度匹配切分往往会使得 m 较小。

词表中的词往往很多，分摊到一个词的概率可能很小，所以 $P(S)$一般是通过很多小数值的连乘积算出来的。如果一个数太小，可能会向下溢出变成零。例如 0.00000000000000000000000000001，double 类型表示不出如此小的数。因为函数 $y=\log x$，当 x 增大时，y 也会增大，所以是单调递增函数。取对数后，表示一个小于 1 的正数的精确度加大了。

$P(S) \approx P(w_1)P(w_2)\cdots P(w_m) \propto \log P(w_1) + \log P(w_2) + \cdots + \log P(w_m)$，这里的$\propto$是正比符号。因为词的概率小于 1，所以取对数后是负数。最后算 $\log P(w)$。

计算任意一个词出现的概率如下：

$$P(w_i)=\frac{w_i\text{在语料库中的出现次数}n}{\text{语料库中的总词数}N}$$

因此 $\log P(w_i)=\log \text{freq}_w - \log N$

如果词概率的对数值事前已经算出来了，则结果直接用加法就可以得到 log $P(S)$，而加法比乘法的运算速度更快。

这个计算 $P(S)$的公式也叫作基于一元概率语言模型的计算公式。这种分词方法简称一元分词。它综合考虑了切分出的词数和词频。一般来说，词数少、词频高的切分方案概率更高。考虑一种特殊的情况：若所有词的出现概率相同，则一元分词退化成最少词切分方法。

句子切分的准确性在很大程度上取决于词语的上下文。比如，“上海银行间的拆借利率上升”，因“上海银行”后接词为“间”，这决定了“上海银行”应该切分为两个词“上海”和“银行”，而不是一个专有名词“上海银行”。

在一元分词中假设前后两个词的出现概率是相互独立的，但在实际中这不太可能。语言学家认为，一个词语的含义取决于它周围的词语。也就是说，某些词语会以很大概率经常出现在一起。比如，甜品店附近经常有咖啡馆，所以这两个词是正相关，但是很少会有人把“甜品店”和“沙县小吃”相提并论。[羡慕][嫉妒][恨]这三个词有时候会连续出现。切分出来的词序列越通顺，越有可能是正确的切分方案。N 元模型使用 n 个单词组成的序列来衡量切分方案的合理性。比如，估计单词 w_1 后出现 w_2 的概率，根据条件概率的定义

$$P(w_2 \mid w_1)=\frac{P(w_1,w_2)}{P(w_1)}$$

可以得到 $P(w_1,w_2)= P(w_1)P(w_2|w_1)$

同理 $P(w1,w_2,w_3)= P(w_1,w_2)P(w_3|w_1,w_2)$

所以 $P(w_1,w_2,w_3)= P(w_1)P(w_2|w_1)P(w_3|w_1,w_2)$

更加一般的形式为

$$P(S)=P(w_1,w_2,\cdots,w_n)= P(w_1)P(w_2|w_1)P(w_3|w_1,w_2)\cdots P(w_n|w_1w_2\cdots w_{n-1})$$

这叫作概率的链规则。其中，$P(w_2|w_1)$表示 w_1 之后出现 w_2 的概率。如果词 w_1 和 w_2 独立出现，则 $P(w_2|w_1)$等价于 $P(w_2)$。

这样需要考虑在 n-1 个单词序列后出现的单词 w 的概率。直接使用这个公式计算 $P(S)$存在两个致命的缺陷：一是参数空间过大，不可能实用化；另外，是数据稀疏严重。例如，词汇量(V)=20000 时，可能的二元语法（bigram）组合数量有 400000000 个，可能的三元语法（trigram）组合数量有 8000000000000 个，可能的四元语法（4-gram）组合数量有 1.6×10^{17} 个。

为了解决这个问题，我们引入了马尔可夫假设：一个词的出现仅仅依赖于它前面出现的有限的一个或者几个词。

如果简化成一个词的出现仅依赖于它前面出现的一个词，那么就称为二元语法模型，即

$$P(S) = P(w_1,w_2,\cdots,w_n)= P(w_1)\ P(w_2|w_1)\ P(w_3|w_1,w_2)\cdots P(w_n|w_1w_2\ldots w_{n-1})$$

$$\approx P(w_1)\ P(w_2|w_1)P(w_3|w_2)\cdots P(w_n|w_{n-1})$$

例如，$P(S_1)= P(\text{有})P(\text{意见}|\text{有})P(\text{分歧}|\text{意见})$，如果简化成一个词的出现仅依赖于它前面出现的两个词，就称之为三元语法模型。如果一个词的出现不依赖于它前面出现的词，称为一元语法（Unigram）模型，也就是已经介绍过的概率语言模型的分词方法。

如果切分方案 S 是由 n 个词组成的序列，那么 $P(w_1)P(w_2|w_1)P(w_3|w_2)\cdots P(w_n|w_{n-1})$也是 n 项连乘积。语言模型无论采用一元语法、二元语法还是三元语法都是 n 项连乘积。只不过二元语法以上模型是条件概率的连乘积。例如：对于切分“产品和服务”来说，二元语法模型计算为 $P(\text{产品})P(\text{和}|\text{产品})P(\text{服务}|\text{和})$，三元语法模型计算为 $P(\text{产品})P(\text{和}|\text{产品})P(\text{服务}|\text{产品,和})$。

因为 $P(w_i|w_{i-1})= \text{freq}(w_{i-1},w_i)/\text{freq}(w_{i-1})$，所以二元分词不仅用到二元词典，还需要用到一元词典。

概率语言模型中文分词切分过程说明如下。

（1）把输入字符串切分成句子：对一段文本进行切分，依次从这段文本中切分出一个个句子，然后对句子逐个进行分词。

（2）原子切分：对于一个句子的切分，首先是通过原子切分，将整个句子切分成一个个的原子单元（即不可再切分的形式，例如 Java 这样的英文单词可以被看成不可再切分的）。

（3）生成 n 元切分词图：根据基本词库对句子进行全切分，并且生成一个以邻接链表表示的基本词图。再根据基本词图得到 n 元词图。

（4）计算最佳切分路径：在这个词图的基础上，运用动态规划算法找出切分最佳路径。

（5）按 Lucene 和 Elasticsearch 定义的 API 输出结果。

二元切分词图简称二元词图，n 元切分词图简称 n 元词图。考虑如何得到二元词图：一个词的开始位置和结束位置组成的节点组合是二元词图中的点；前后两个词的转移概率作为边的权重。

例如，“有意见分歧”这句话中节点的组合有：{0,1}、{0,2}、{1,2}、{1,3}、{2,3}、{3,4}、{3,5}。得到的二元切分词图如图 2-2 所示。

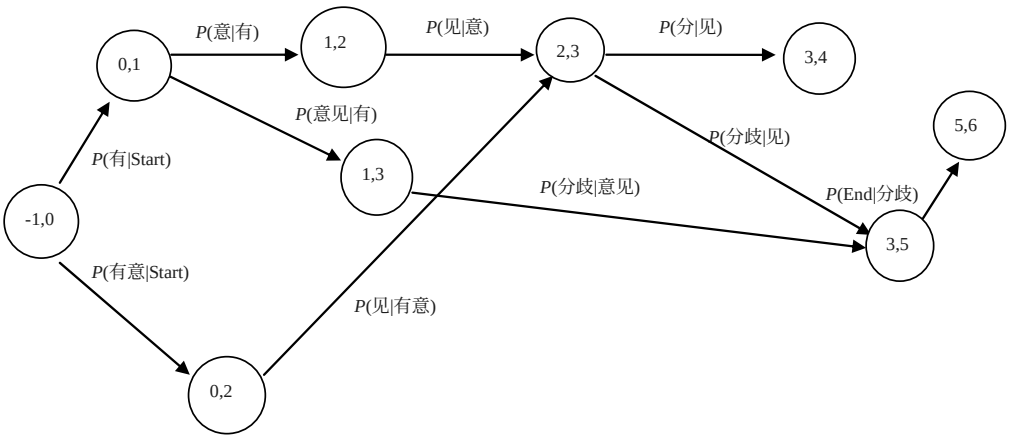

图 2-2　二元分词词图

Sqlite 数据库中存储的二元连接表的创建语句如下：

```
CREATE TABLE "BIGRAM" ("PREV" string NOT NULL DEFAULT (null) , --前一个词
"NEXT" string DEFAULT (null)                                  --后一个词
 ,"FRQ" int                                                   --二元频次
)
```

用于测试词条件概率的类：

```
public class DBBigramProb {
   Connection con;

   static double lambda1 = 0.3;      //平滑参数
   static double lambda2 = 0.7;

   public DBBigramProb() throws Exception {
      con = getSqliteConnection();
   }

   public double wordConditionalProbability(String prev,String next) {
   //条件概率
      QueryRunner runner = new QueryRunner();
      String sql = "select sum(frq) from UNIGRAM_WORD";
      Integer totalFreq = runner
            .query(con, sql, new ScalarHandler<Integer>());

      int dic_totalFreq = totalFreq;  //总频次

      int bigramFreq = getBigramFreq(prev, next);//从二元词典找二元频次

      int t1_freq = getFreq(next);
      int t2_freq = getFreq(prev);
```

```
        double wordProb = lambda1 * t1_freq / dic_totalFreq + lambda2
              * (bigramFreq / t2_freq);
        return wordProb;
    }

    private int getFreq(String t1) throws SQLException {
        QueryRunner runner = new QueryRunner();

        String sql = "select frq from UNIGRAM_WORD where WORD =?";

        Integer freq = runner.query(con, sql, new ScalarHandler<Integer>(), t1);

        if (freq == null)
          return 0;

        return freq;
    }

    private int getBigramFreq(String t2, String t1)
          throws SQLException {  //二元连接频次
        QueryRunner runner = new QueryRunner();

        String sql = "select frq from BIGRAM where PREV =? and NEXT = ?";

        Integer freq = runner.query(con, sql, new ScalarHandler<Integer>(), t2,
              t1);

        if (freq == null)
          return 0;

        return freq;
    }

    public static Connection getSqliteConnection() throws Exception {
                              //得到数据库连接
        try {
          Class.forName("org.sqlite.JDBC");
          String absolute_path_to_sqlite_db = "./dic/bigram.sqlite";
                              //二元词库
          return DriverManager.getConnection("jdbc:sqlite:"
                  + absolute_path_to_sqlite_db);
        } catch (Exception e) {
          e.printStackTrace();
          return null;
        }
    }
}
```

使用这个测试类：

```
String prev = "有";
```

```
String next = "意见";

DBBigramProb bigramProb = new DBBigramProb();
double wordCondProb = bigramProb.wordConditionalProbability(prev, next);

double logProb = Math.log(wordCondProb);
System.out.println(logProb);
```

组合节点类定义如下：

```
public class Node {
   public int start;                                    //开始节点
   public int end;                                      //结束节点

   public double logProb;                               //节点本身的概率
   public int frq;                                      //组合频次
   public double nodeProb;                              //节点累积概率
   public Node bestPrev;                                //最佳前驱节点

   public Node(int s, int e, int fq, double p) {        //构造方法
      start = s;
      end = e;
      frq = fq;
      logProb = p;
   }

   @Override
   public int hashCode() {
      return start ^ end;
   }

   @Override
   public boolean equals(Object o) {
      if (!(o instanceof Node)) //判断传入对象的类型
        return false;
      Node that = (Node) o;

      return (this.start == that.start && this.end == that.end);
   }

   public String toString() {
      return start + ":" + end+" frq "+frq+" logProb "+logProb;
   }
}
```

LatticeFactory 类生成词图：

```
public class LatticeFactory {
   public static TernarySearchTrie dic = null;    //词典树
   static FSTSGraph fstSeg;                       //用于原子切分的有限状态转换
```

```
        static {
           try {
             fstSeg = new FSTSGraph();
           } catch (Exception e) {
             e.printStackTrace();
           }

           dic = (new DicFileFactory()).create();      //创建词典树
           //如果要根据数据库中的表创建词典树，则使用如下语句
           //(new DicDBFactory()).create();
        }

        public static AdjList getLattice(String text) throws Exception {
           int sLen = text.length();                   //字符串长度
           AdjList g = new AdjList(sLen + 2);          //存储所有被切分的可能的词

           //原子切分
           SegScheme schema = fstSeg.seg(text);

           //用来存放前驱词的集合
           ArrayList<WordEntry> prevWords = new ArrayList<WordEntry>();

           //从前往后求出每个节点的最佳前驱节点和它的节点概率
           int fromPoint = 0;
           while (fromPoint >= 0) {
             int start = schema.startPoints.nextSetBit(fromPoint);     //开始点
             int end = schema.endPoints.nextSetBit(fromPoint + 1);     //结束点

             fromPoint = schema.startPoints.nextSetBit(start + 1);

             //从词典中查找前驱词的集合
             boolean match = dic.matchAll(text, start, prevWords,
                     schema.endPoints);

             if (!match) {
                 //词典中找不到对应的词,则返回开始点和结束点之间的字符串
                 String word = text.substring(start, end);
                 prevWords.add(new WordEntry(word, 1));
                 Node newEdge = new Node(start, start + word.length(),1,
                         Math.log((double) 1 / dic.n));
                 g.addEdge(newEdge);      //词图增加边
             } else {
                 for (WordEntry w : prevWords) {  //遍历找到的每个词
                     Node newEdge = new Node(start, start + w.word.length(),w.
freq,
                             Math.log((double) w.freq / dic.n));
                     g.addEdge(newEdge);  //词图增加边
                 }
             }
```

```
        }

        return g;
    }
}
```

Segmenter.bestPrev()方法从后往前计算词图中每个节点的最佳前驱节点：

```
public static AdjList bestPrev(AdjList wordGraph) throws Exception {
    for (Node currentNode : wordGraph) { //从前往后遍历切分词图中的每个节点

        //得到当前节点的前驱节点集合
        NodeLinkedList prevNodes = wordGraph.prevNodes(currentNode);

        double nodeProb = minValue;         //候选词概率
        Node minNode = null;
        if (prevNodes == null) {
            currentNode.nodeProb = 0;
            continue;
        }
        for (Node prevNode : prevNodes) {
            double currentProb = transProb(prevNode, currentNode) //转移概率
                    + prevNode.nodeProb;     //前一个节点的节点概率

            if (currentProb > nodeProb) {
                nodeProb = currentProb;
                minNode = prevNode;
            }
        }
        currentNode.bestPrev = minNode;     //设置当前节点的最佳前驱节点
        currentNode.nodeProb = nodeProb;    //设置当前节点的节点概率
    }

    return wordGraph;
}
```

二元分词方法切分文本的代码如下：

```
public static List<String> split(String text) throws Exception {
    AdjList wordGraph = LatticeFactory.getLattice(text);   //得到词图
    bestPrev(wordGraph);                    //从后往前计算最佳前驱节点
    ArrayDeque<Node> seq = new ArrayDeque<Node>();     //切分出来的节点序列
    //从后向前找最佳前驱节点
    for (Node t = wordGraph.endNode.bestPrev; t.start > -1; t = t.bestPrev) {
        seq.addFirst(t);
    }
    //根据最佳前驱节点数组回溯求解词序列
    return bestPath(text, seq);
}
```

测试分词的代码如下：

```
String sentence = "中国成立了";
```

```
List<String> ret = Segmenter.split(sentence);
System.out.println("切分结果 ");
for (String word : ret) {
   System.out.print(word+" / ");
}
```

2.5　隐马尔可夫模型

可以使用隐马尔可夫模型（hidden markov model，HMM）实现词性标注。

2.5.1　数据基础

词典要能够识别每个词可能的词性。例如，可以根据词性编码在文本文件 n.txt 中存放名词，在文本文件 v.txt 中存放动词，在文本文件 a.txt 中存放形容词，等等。例如，v.txt 的内容如下：

```
欢迎
迎接
```

可以把这些按词性分放到不同文件的词表合并成一个大的词表文件，每行一个词和对应的一个词性。例如：

```
把:p
把:q
```

如果把词表放到数据库中，则设置词和词性两列。为了避免重复插入词，词和词性联合作为主键。

```
CREATE TABLE "AI_BASEWORDS" ("WORD" string NOT NULL , --词
 "PARTSPEECH" string,  --词性
 "FRQ" int, --词频
 "PINYIN" string) --拼音
```

从 MySQL 数据库读出词的代码如下：

```
TernarySearchTrie dic = new TernarySearchTrie();

Connection conn = getConnect();      //得到数据库连接

String sql = "SELECT WORD,PARTSPEECH,FRQ from AI_BASEWORD";
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(sql);

while (rs.next()) {
   String word = rs.getString(1);
```

```
        String pos = rs.getString(2);
        int frq = rs.getInt(3);

        if(frq<=0) {
           System.out.println("词频错误 "+word +" frq "+frq);
           frq = 1;
        }

        dic.addWord(word, pos,frq);   //增加词表到词典树
        dic.n +=frq;                   //总频次
    }

    conn.close();
```

2.5.2 维特比算法

解决标注歧义问题最简单的一种方法是从单词所有可能的词性中选出这个词最常用的词性作为这个词的词性，也就是一个概率最大的词性，比如“改革”大部分时候作为一个名词出现，那么可以机械地把这个词总是标注成名词，但是这样标注的准确率会比较低，因为只考虑了频率特征。

考虑词所在的上下文可以提高标注的准确率。一般，在动词后接名词的概率很大。例如，“推进/改革”中的“推进”是动词，所以后面的“改革”很有可能是名词。这样的特征叫作上下文特征。

隐马尔可夫模型和基于转换的学习方法是两种常用的词性标注方法。这两种方法都整合了频率和上下文两方面的特征来取得好的标注结果。具体来说，隐马尔可夫模型同时考虑到了词的生成概率和词性之间的转移概率。

很多生物也懂得同时利用两种特征信息。例如，箭鼻水蛇是一种生活在水中以吃鱼或虾为生的蛇。它是唯一一种长着触须的蛇类。箭鼻水蛇最前端的触须能够感触非常轻微的变动，这表明它可以感触鱼类移动时产生的细微水流变化。在光线明亮的环境中，箭鼻水蛇能够通过视觉捕食小鱼。因此，它能够同时利用触觉和视觉，即通过光线的变化和水流的变化信息来捕鱼。

词性标注的任务是：给定词序列 $W=w_1,w_2,\cdots,w_n$，寻找词性标注序列 $T=t_1,t_2,\cdots,t_n$，使得 $P(t_1,t_2,\cdots,t_n|w_1,w_2,\cdots,w_n)$这个条件概率最大。

例如，词序列是：[他] [会] [来]这句话。为了简化计算，假设只有词性：代词（r）、动词（v）、名词（n）和方位词（f）。这里：[他]只可能是代词，[会]可能是动词或者名词，而[来]可能是方位词或者动词。所以有 4 种可能的标注序列。需要比较：P(r,v,v|他,会,来)、P(r,n,v|他,会,来)、P(r,v,f|他,会,来)和 P(r,n,f|他,会,来)，发现 P(r,v,v|他,会,来)是这 4 个概率中最大的，所以选择词性标注序列[r,v,v]。

使用贝叶斯公式重新描述这个条件概率：

$$P(t_1,t_2,\cdots,t_n)P(w_1,w_2,\cdots,w_n|t_1,t_2,\cdots,t_n)/P(w_1,w_2,\cdots,w_n)$$

忽略掉分母 $P(w_1,w_2,\cdots,w_n)$。

$$P(t_1,t_2,\cdots,t_n)= P(t_1)P(t_2|t_1)P(t_3|t_1,t_2)\cdots P(t_n|t_1t_2\cdots t_{n-1})$$

做独立性假设，使用 n 元模型近似计算 $P(t_1,t_2,\cdots,t_n)$。例如使用二元语法模型，则有

$$P(t_1,t_2,\cdots,t_n)\approx\prod_{i=1}^{n}P(t_i\mid t_{i-1})$$

近似计算 $P(w_1,w_2,\cdots,w_n|t_1,t_2,\ \cdots,t_n)$：假设一个类别中的词独立于它的邻居，则有

$$P\left(w_1,w_2,\cdots,w_n\mid t_1,t_2,\cdots,t_n\right)\approx\prod_{i=1}^{n}P(w_i\mid t_i)$$

寻找最有可能的词性标注序列实际的计算公式为

$$P(t_1,t_2,\cdots,t_n)P(w_1,w_2,\cdots,w_n|t_1,t_2,\cdots,t_n)\approx\prod_{i=1}^{n}P(t_i\mid t_{i-1})P(w_i\mid t_i)$$

因为词是已知的，所以这里把词 w 称为显状态。因为词性是未知的，所以把词性 t 称为隐状态。条件概率 $P(t_i|t_{i-1})$称为隐状态之间的转移概率。条件概率 $P(w_i|t_i)$称为隐状态到显状态的发射概率，也称为隐状态生成显状态的概率。注意，不要把 $P(w_i|t_i)$算成 $P(t_i|w_i)$。

因为出现某个词性的词可能很多，所以对很多词来说，发射概率 $P(w_i|t_i)$往往很小。而词性往往只有几十种，所以转移概率 $P(t_i|t_{i-1})$往往比较大。就好像这世界有各种各样的动物，在所有的动物中，正好碰到啄木鸟的可能性比较小。

使用 byte 类型表示一个词性，定义词性的 POS 类实现如下：

```
public class POS {
   public final static byte start = 0;       //开始
   public final static byte end = 1;         //结束
   public final static byte a = 2;           //形容词
   public final static byte ad = 3;          //副形词
   public final static byte ag = 4;          //形语素
   public final static byte an = 5;          //名形词
   public final static byte b = 6;           //区别词
   public final static byte c = 7;           //连词
   public final static byte d = 8;           //副词
   public final static byte dg = 9;          //副语素
   public final static byte e = 10;          //叹词
   public final static byte f = 11;      //方位词
   public final static byte g = 12;      //语素
```

```
        public final static byte h = 13;       //前接成分
        public final static byte i = 14;       //成语
        public final static byte j = 15;       //简称略语
        public final static byte k = 16;       //后接成分
        public final static byte l = 17;       //习用语
        public final static byte m = 18;       //数词
        public final static byte n = 19;       //名词
        public final static byte ng = 20;      //名语素
        public final static byte nr = 21;      //人名
        public final static byte ns = 22;      //地名
        public final static byte nt = 23;      //机构团体
        public final static byte nx = 24;      //字母专名
        public final static byte nz = 25;      //其他专名
        public final static byte o = 26;       //拟声词
        public final static byte p = 27;       //介词
        public final static byte q = 28;       //量词
        public final static byte r = 29;       //代词
        public final static byte s = 30;       //处所词
        public final static byte t = 31;       //时间词
        public final static byte tg = 32;      //时语素
        public final static byte u = 33;       //助词
        public final static byte ud = 34;      //结构助词
        public final static byte ug = 35;      //时态助词
        public final static byte uj = 36;      //结构助词的
        public final static byte ul = 37;      //时态助词了
        public final static byte uv = 38;      //结构助词地
        public final static byte uz = 39;      //时态助词着
        public final static byte v = 40;       //动词
        public final static byte vd = 41;      //副动词
        public final static byte vg = 42;      //动语素
        public final static byte vn = 43;      //名动词
        public final static byte w = 44;       //标点符号
        public final static byte x = 45;       //非语素字
        public final static byte y = 46;       //语气词
        public final static byte z = 47;       //状态词
        public final static byte unknow = 48; //未知
    }
```

存储词的转移频次的 POSTransFreq.txt 文件内容如下：

```
start:uj:1
start:v:5230
start:vd:4
start:vg:54
```

```
start:vn:440
start:w:5047
start:y:2
start:z:58
a:end:173
a:a:833
a:ad:8
a:ag:6
a:an:127
a:b:62
a:c:451
a:d:296
a:dg:2
a:f:84
a:i:13
a:j:80
a:k:4
a:l:125
a:m:896
```

Tagger 类中的属性如下：

```
//转移频次
private int[][] transFreq = new int[POS.names.length][POS.names.length];
//每个词性的频次
private int[] typeFreq = new int[POS.names.length];
private int totalFreq;                              //所有词的总频次
```

读取 POSTransFreq.txt 文件，得到类型频次和转移频次的代码如下：

```
String transFrq = "./dic/POSTransFreq.txt";
InputStream file = new FileInputStream(new File(transFrq));

BufferedReader read = new BufferedReader(new InputStreamReader(
      file, "GBK"));

String line = null;

while ((line = read.readLine()) != null) {
   StringTokenizer st = new StringTokenizer(line, ":");
   int pre = POS.values.get(st.nextToken());      //前面的词类
   int next = POS.values.get(st.nextToken());     //后面的词类
   int frq = Integer.parseInt(st.nextToken());    //词频
   transFreq[pre][next] = frq;                    //转移频次
   typeFreq[next] += frq;                         //类型频次
   totalFreq += frq;
   //如果有从开始类型到其他类型的转移频次，就把这样的转移频次计入开始频次的类型频次
   if (pre == 0) {
      typeFreq[0] += frq;
   }
}
read.close();
```

为了避免乘积项为零，平滑转移概率的公式为

$$P(t_i \mid t_j) = \lambda_1 \frac{\text{Freq}(t_j, t_i)}{\text{Freq}(t_j)} + \lambda_2 \frac{\text{Freq}(t_j)}{\text{Freq}(\text{total})}$$

这里，取 λ_1=0.9， λ_2=0.1。

根据平滑公式计算转移概率的 getTransProb()方法实现如下：

```
    /**
     * 计算上一个到下一个词的转移概率
     * @param curState
     *          前一个词性
     * @param toTranState
     *          后一个词性
     * @return
     */
    public double getTransProb(byte curState, byte toTranState) {
       return Math.log((0.9 * transFreq[curState][toTranState]
            / typeFreq[curState] + 0.1 * typeFreq[curState] / totalFreq));
    }
```

2.6 英文文本切分与标注

这里首先介绍英文句子切分的方法，然后介绍英文词性标注。

2.6.1 句子切分

英文句子切分并不是一个简单的问题。标点符号“？”和“！”的含义比较单一。但是“.”有很多种不同的用法，并不一定是句子的结尾。例如：“Mr. Vinken is chairman of Elsevier N.V.,the Dutch publishing group.”需要排除掉一部分情况。如果“.”是某个短语中间的一部分，则它不是句子的结尾。这里的“Mr. Vinken”是一个人名短语。如果这个人名正好不在词典中，则可以根据上下文识别规则识别出这个短语。

```
    String text= "Mr. Vinken is chairman of Elsevier N.V., the Dutch publishing
group.";
    EnText enText = new EnText(text);
    for(Sentence sent:enText){
       System.out.println(sent); //因为输入的是一个句子,所以这里只会打印出一个句子
    }
```

Java 中的 BreakIterator 类已经包含了切分句子的功能。用它实现一个英文句子迭代器：

```
private final static class SentBreakIterator implements Iterator<Sentence> {
   String text;
   int start;
   int end;
   //根据英文标点符号切分
   static final BreakIterator boundary = BreakIterator
         .getSentenceInstance(Locale.ENGLISH);

   public SentBreakIterator(String t) {
      text = t;
      //设置要处理的文本
      boundary.setText(text);
      start = boundary.first(); //开始位置
      end = boundary.next();
   } //用于迭代的类

   @Override
   public boolean hasNext() {
      return (end != BreakIterator.DONE);
   }

   @Override
   public Sentence next() {
      String sent = text.substring(start, end);

      Sentence sentence = new Sentence(sent, start, end);
      start = end;
      end = boundary.next();
      return sentence;
   }
}
```

BreakIterator 分得不太准确。所以我们自己写一个句子切分器。输入当前切分点，找下一个切分点的代码如下：

```
public static int nextPoint(String text, int lastEOS) {
   int i = lastEOS;
   while (i < text.length()) {
      //跳过短语
      i = skipPhrase(text, i);

      //然后再找标点符号
      String toFind = eosDic.matchLong(text, i); //匹配标点符号词典
      if (toFind != null) {
        //判断是否有效的可切分点。例如，在括号中的标点符号不是有效的可切分点
        boolean isEndPoint = isSplitPoint(text, lastEOS, i);
        if (isEndPoint) {
            return i + toFind.length();
        }
```

```
            i = i + toFind.length();
         } else { //没找到
            i++;
         }
      }
      return text.length(); //返回最大长度
   }
```

SentIterator 是一个用于迭代英文文本返回句子的内部类，实现代码如下：

```
   private final static class SentIterator implements Iterator<Sentence> {
      String text;
      int lastEOS = 0;

      public SentIterator(String t) {
         text = t;
      }

      @Override
      public boolean hasNext() {
         return (lastEOS < text.length());
      }

      @Override
      public Sentence next() {
         int nextEOS = EnSentenceSpliter.nextPoint(text, lastEOS);
         String sent = text.substring(lastEOS, nextEOS);
         Sentence sentence = new Sentence(sent, lastEOS, nextEOS);
         lastEOS = nextEOS;
         return sentence;
      }
   }
```

2.6.2 标注词性

一段英文：Cats never fail to fascinate human beings. They can be friendly and affectionate towards humans,but they lead mysterious lives of their own as well.

标注词性后的结果是：

```
   Cats(n.) never fail(v.) to(prep) fascinate(v.) human(n.) beings(n.).
They(pron.) can(aux.) be(v.) friendly(adj.) and(conj.) affectionate(adj.)
towards(prep) humans(n.) but(conj.) they(n.) lead(v.) mysterious(adj.) lives(n.)
of(prep.) their(n.) own(n.) as well(adv.).
```

这里用编码来表示词性。括号中的输出是词性编码。汉语中的量词是英语中没有的，例如件、个、艘。英语中也有一些独有的词性，例如冠词 a、an、the。英文词性编码表如表 2-2 所示。

表 2-2　英文词性编码表

代　码	名　称
n	名词
adj	形容词
adv	副词
art	冠词
pos	所有格
pron	代词
aux	情态助动词
conj	连接词
v	动词
num	数词
prep	介词
punct	标点符号
int	感叹词

词性标注的流程图如图 2-3 所示。

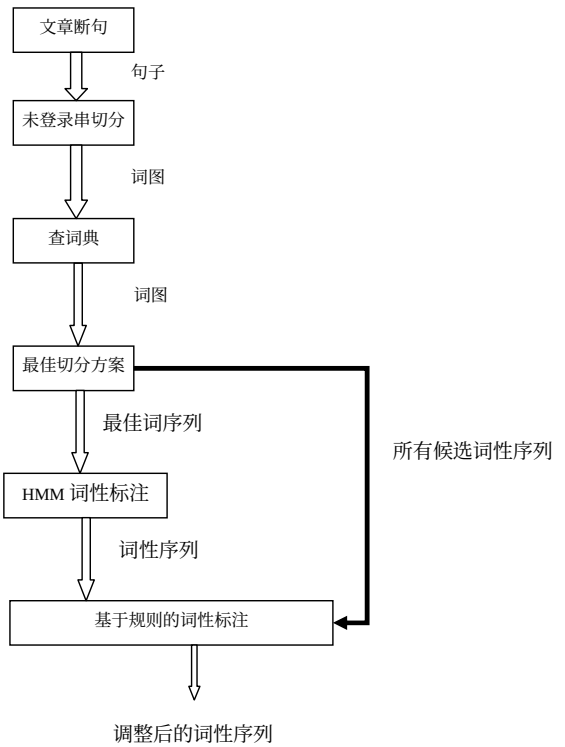

图 2-3　词性标注流程图

关于利用隐马尔可夫模型做词性标注在中文词性标注实现中已经介绍过了。英文词性标注语料库和中文词性标注语料库不同。

标注规则例如 I like it 对应的词性序列[r v r]。

```
key = new ArrayList<PartOfSpeech>();
key.add(PartOfSpeech.pron);  //I
key.add(PartOfSpeech.v);     //like
key.add(PartOfSpeech.pron);  //it
posTrie.addProduct(key);
```

实现代码：

```
public static ArrayList<WordToken> getWords(Sentence sent){
   ArrayList<WordTokenInf> words = Segmenter.seg(sent); //先分词
   WordType[] tags = g.tag(words); //然后标注词性

   //再把词性和词本身结合起来，返回完整的词性标注结果
   int i=0;
   ArrayList<WordToken> tokens = new ArrayList<WordToken>();

   for(WordTokenInf w:words){
     WordToken t = new WordToken(w.baseForm,w.termText,w.start,w.end,tags[i]);

      ++i;
      tokens.add(t);
   }

   return tokens;
}
```

2.7 命名实体识别

命名实体识别包括人名识别、地名识别和机构名识别等。因人名、地名和机构名的识别方法基本相同，故本书主要介绍人名识别。

2.7.1 人名识别

中文文本中的未登录人名包括中国人名和外国译名以及日本人名等。例如，“彭帅、郑洁 1：2 不敌阿根廷选手杜尔科和意大利选手佩内塔”，其中包括中国人名{彭帅、郑洁}，还有外国人名“杜尔科”和“佩内塔”。

英文人名的全名是由 first name, middle name 和 last name 三部分组成的, 其中 middle name 的主要目的只是为了防止重名，一般生活中会省略中间名。

对于没有能够根据词典与相邻字组成 2 个字以上词的字符，切分出来的结果称为切分碎片。例如，“ 素与杨宝森先生交好 ”，如果 “ 杨宝森 ” 这个词不在词典中，则切分出来的结果是：“ 素/与/杨/宝/森/先生/交好 ”。

人名往往在切分碎片中，但是也有特例：

- 人名内部相互成词。指姓与名、名与名之间本身就是一个已经被收录的词。例如，[王国]维、[高峰]、[汪洋]、张[朝阳]、冯[胜利]。
- 人名与其上下文组合成词。例如，“ 这里[有关]天培的壮烈 ”。

对识别人名有用的信息：

- 人名所在的上下文。例如：“ **教授 ”，这里 “ 教授 ” 是人名的下文；“ 邀请** ”，这里 “ 邀请 ” 是人名的上文。
- 人名本身的概率。例如：不依赖上下文，直观地来看，“ 刘宇 ” 可能是个人名，而 “ 史光 ” 不太可能是个人名。若采用未登录词的概率作为这种可能性的衡量依据，则 “ 刘宇 ” 作为人名的概率是：“ 刘宇 ” 作为人名出现的次数/人名出现的总次数。怎么算当前这个人名的出现概率？用姓的概率×名字的概率。

2.7.2　人名识别规则

人名识别有一些规则，例如 “ 让<nr>和<nr>一起 ”。

分析中国人名所在的上下文，表明身份的词有：

- 出现在人名之前的词：工人、教师、影星、犯人。
- 出现在人名之后的词：先生、同志。
- 既可能出现在前面，也可能出现在后面的词：校长、经理、主任、医生。

地名或机构名往往出现在人名之前，例如：静海县大丘庄禹作敏。

“ 的 ” 字结构往往出现在人名之前，例如：年过七旬的王贵芝。

有的动作词出现在人名之前，例如：批评，逮捕，选举。有的动作词出现在人名之后，例如：说，表示，吃，结婚。

未登录人名识别过程是：首先从输入串找所有可能的人名，然后再按照 N 元模型做分词，过滤候选人名。例如输入串原文是：程正泰的父亲是一位京剧票友，素与杨宝森先生交好。从中提取出候选人名{程正泰,杨宝,杨宝森}，把候选人名 “ 杨宝 ” 过滤掉。

识别候选人名的两层信息：底层是组成未登录姓名的单字特征和上下文词特征，特征串作为一个整体的搭配。

未登录中国人名相关的特征见表 2-3。

最容易想到的方法是：先找姓，然后找名。但有的未登录姓名只有名字，没有姓。

表 2-3　未登录中国人名相关的特征

编　　号	特　　征	举　　例
1	姓	张/nr　卫涛/nr
2	双名首字	张/nr　卫涛/nr
3	双名尾字	张/nr　卫涛/nr
4	单名	赵/nr　红/nr
11	上文	主席/n　李/nr　贤哲/nr
12	下文	李/nr　贤哲/nr　首先/d
13	同时做上文和下文	李/nr　贤哲/nr　和/d　陈/nr　涛/nr

把人名特征存放在 nr.txt 文本中。根据特征词表 nr.txt 对输入串全切分，形成人名特征词图。采用邻接链表（AdjList）存储切分结果。也就是说，用邻接链表表示人名特征词图。例如输入串“我爸是李刚”组成的特征词图如图 2-4 所示。

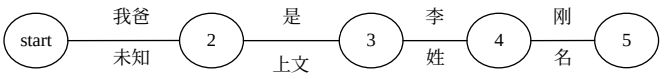

图 2-4　人名特征词图

根据特征序列识别出人名。例如：我爸叫李刚。这里[动词,姓,名,标点符号]组成了一个包含中国人名的特征序列。把[动词,姓,名,标点符号]称为一个识别规则。可以根据这个识别规则识别出“李刚”这个人名。这个规则的完整形式是：

动词　中国人名　标点符号 ⟹ 动词　姓　名　标点符号

例如有一条识别规则[姓氏,单名,下文]，用特征编号序列表示是[1,4,12]。

一个词可以同时是两种类型。例如“彭帅、郑洁 1：2 不敌阿根廷选手杜尔科和意大利选手佩内塔”这句话，这里的[、]既是[彭帅]这个人名的下文，[、]也是[郑洁]这个人名的上文。

因为未登录人名往往在分词结果中出来的是切分碎片，所以最简单的方法是把分词结果再标注一次人名特征，如图 2-5 所示。其中的人名特征用编号表示。

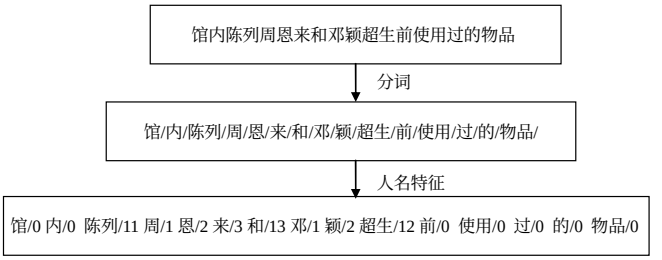

图 2-5　从分词序列中找人名特征

因为“超生”形成了一个词，所以用词序列只能识别出“邓颖”这样的人名，无法正确的识别出“邓颖超”，所以用人名特征专用的词图，如图 2-6 所示。也就是说，用存放在 nr.txt 中的人名特征词切分出人名特征词图。

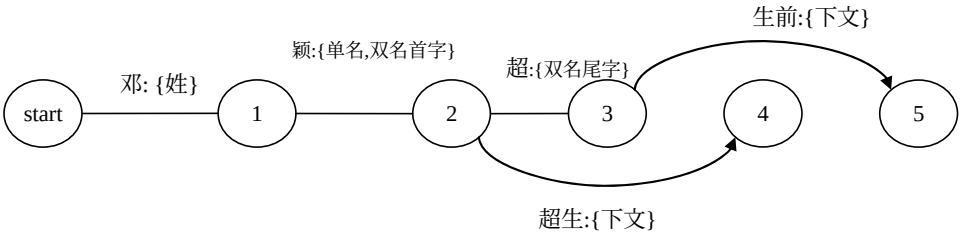

图 2-6　“邓颖超生前”特征词图

在图 2-6 所示的切分特征图里找到人名的识别规则序列[1,2,3,12]，找到后就能把“邓颖超”识别成一个人名。也就是从特征词图找到[邓,颖,超,生前]对应的特征序列[姓氏,双名首字,双名尾字,下文]。

首先有一些和识别未登录词相关的特征词表，然后输入串根据特征词表形成特征词图。最后根据未登录词识别规则从特征词图中找候选未登录词。

人名规则，例如[姓+名]，类似的规则还有很多，所以可用有限状态自动机求交集的方法同时找出所有可能的人名。存在一个句子中的人名特征词图，还有一个是由规则树组成的检索树（trie tree）。人名特征词图（也就是一个 AdjList 的实例）相当于一个 DFA，规则树组成的检索树相当于另外一个 DFA。找候选人名相当于对这两个 DFA 求交集。

在特征词序列上识别人名，不是在原始的字符上识别人名。特征词类型定义成为枚举类型：

```
public enum PersonType {
   preContext,       //上下文中的上文，例如，邀请**
   postContext,      //上下文中的下文，例如，**说
   surName,          //姓
   singleName,       //单名
   doubleName1,      //双名第一个字
   doubleName2       //双名第二个字
}
```

例如：[老生] [虞] [子] [期] [小生]。这里的[老生]是 preContext，而[小生]则是 postContext。

人名识别规则有很多，所以用标准检索树存储规则。例如，有规则[1,4,12]，[11,1,2,3]，[11,1,4,12]，如图 2-7 所示。

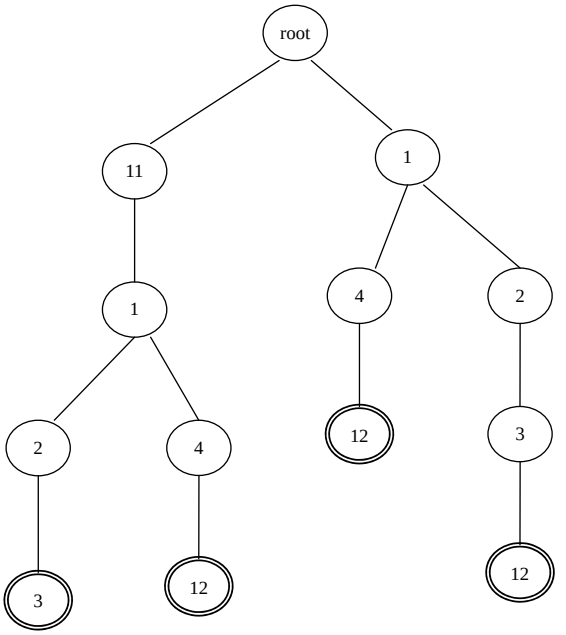

图 2-7　人名识别规则组成的标准检索树

一条规则中可能有多个人名，例如“李/nr 鹏/nr 和/d 江/nr 泽民/nr”对应一个复杂的规则：[1,4,13,1,2,3]。

定义标准检索树的节点：

```
public class TrieNode{
   private PersonType nodeKey;                //键
   private ArrayList<NameSpan> nodeValue;     //值
   private boolean terminal;                  //标志这个节点是否可以结束的节点
   private Map<PersonType, TrieNode> children =
             new HashMap<PersonType, TrieNode>();  //引用到所有的孩子节点
}
```

所以可以通过匹配规则来识别未登录词。为了实现同时查找多个规则，可以把右边的模式组织成检索树，左边的模式作为节点属性。

NameSpan 用来指定一个区间，就是合并多长的未登录词语素成为一个未登录词。识别规则的左边部分就是一个 NameSpan 序列，而识别规则的右边部分就是一个 PersonType 序列。规则检索树的实现如下：

```
public class Trie {
   public TrieNode rootNode = new TrieNode(); //根节点
```

```
       //放入键/值对
      public void addProduct(ArrayList<PersonType> key, ArrayList<NameSpan> lhs)
{
          TrieNode currNode = rootNode;           //当前节点
          for (int i = 0; i < key.size(); ++i) {  //从前往后找键中的类型
            PersonType c = key.get(i);
            Map<PersonType, TrieNode> map = currNode.getChildren();
            currNode = map.get(c);                //向下移动当前节点
            if (currNode==null) {
                currNode = new TrieNode();
                map.put(c, currNode);             //孩子放入散列表
            }
          }
          currNode.setTerminal(true);             //设置成可以结束的节点
          currNode.setNodeValue(lhs);             //设置值
       }

       //根据键查找对应的值，也就是根据右边的 PersonType 序列看有没有对应的识别规则
       public ArrayList<NameSpan> find(ArrayList<PersonType> key) {
          TrieNode currNode = rootNode;           //当前节点
          for (int i = 0; i < key.size(); ++i) {  //从前往后找键中的类型
            PersonType c = key.get(i);
            currNode = currNode.getChildren().get(c);//向下移动当前节点
            if (currNode==null) {
                return null;
            }
          }
          if (currNode.isTerminal()) {            //是结束节点
            return currNode.getNodeValue();
          }
          return null;                            //没找到
       }
    }
```

把规则加入到规则检索树的代码如下：

```
    //构造规则的右部分： 人名上文 姓氏 + 单人名
    rhs = new ArrayList<PersonType>();
    rhs.add(PersonType.preContext);      //人名上文
    rhs.add(PersonType.surName);         //姓氏
    rhs.add(PersonType.singleName);      //单名
    //构造规则的左部分： 人名上文之后是姓名
    lhs = new ArrayList<NameSpan>();
    lhs.add(new NameSpan(1, 2, PersonType.name)); //姓氏 和 单人名 组成完整的人名
    rules.addProduct(rhs, lhs);          //把人名识别规则加入规则库
```

从特征词图中找规则检索树上可以匹配上的规则。也就是说，在特征词图上有一条路径正好也是可以在规则检索树上从开始走到结束节点。例如，图 2-8 中左边的特征词图状

态 0 接收输入“ 姓 ”以后转换到状态 1，状态 0 接收输入“ 上文 ”以后转换到状态 2。状态 1 和状态 2 被映射到右边的规则检索树，因为右边的规则检索树也存在从开始状态接收输入“ 姓 ”以后转换到一个新状态，从开始状态接收输入“ 上文 ”以后转换到另外一个新状态。

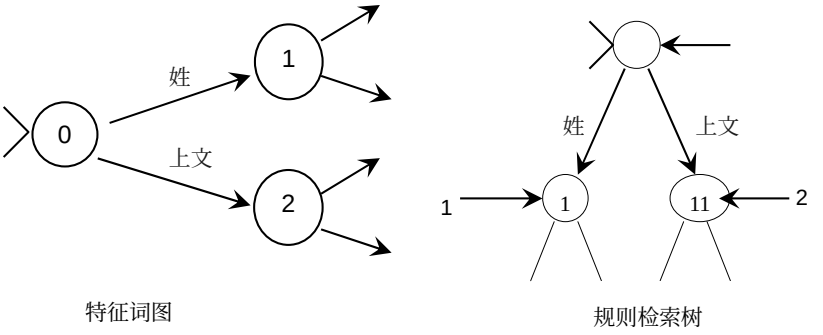

图 2-8　从人名特征词图上找匹配规则

把特征词图中的当前状态叫作 s_1，规则检索树的当前状态称为 s_2。状态 s_1 和 s_2 组成一个当前状态对（s_1, s_2）。例如，图 2-8 存在状态对（1,1）和（2,11）。

当前状态对 StatePair 类的部分代码如下：

```
public static class StatePair {
   int s1;  //特征词图中的当前状态编号
   TrieNode s2; //规则检索树的当前状态节点
}
```

在每个当前状态对中，都对状态 s_1 和 s_2 的所有可能接收的输入求交集。从特征词图找规则序列的每一步都要找输入交集，也就是求词图和规则树中的 PersonType 的交集。其实现代码如下：

```
public static class NextInput {  //有限状态自动机中的下一个输入
      int end;                    //词的结束位置，词图中下一个状态对的依据
      PersonType type;            //经过的类型，规则树中下一个状态对的依据
      String term;                //经过的词
}
```

UnknowGrammar 类的 intersection()方法的实现代码如下：

```
/**
 * 取得词图和规则树都可以向前进的步骤
 * @param edges  词图上的边
 * @param s 规则树上的类型
 * @return 共同的有效输入
 */
public ArrayList<NextInput> intersection(EntityLinkedList edges,
```

```
                                Set<PersonType> s) {
      ArrayList<NextInput> tmp = new ArrayList<NextInput>();
      for (EntityTokenInf x : edges) {
         if (x.data == null)
           continue;
         for (EntityTypes.EntityTypeInf typeInf : x.data) {
           if (s.contains(typeInf.pos)) {  //规则树上的类型包含词所属的类型
               tmp.add(new NextInput(x.end, typeInf.pos, x.termText));
           }
         }
      }
      return tmp;
   }
```

找出人名相关的序列，也就是把词图映射到检索树上。

```
   public static class MatchValue {
      ArrayList<NameSpan> left;     //规则的左边部分
      ArrayList<PersonType> right;  //规则的右边部分
      ArrayList<String> term;       //对应的词序列
   }
```

词图中的每个节点都可能有几条路径通过，但只保留那些能走到底的路。查找过程的输入是特征词图开始找的位置，返回多个可能的识别规则的 UnknowGrammar.intersect()方法实现如下：

```
   /**
    * 词图映射到检索树上，也就是从词图指定位置开始找识别规则
    * @param g 人名特征词图
    * @param offset 开始位置
    * @return  匹配结果
    */
   public ArrayList<MatchValue> intersect(AdjList g, int offset) {
      ArrayList<MatchValue> match = new ArrayList<MatchValue>(); //映射结果
      Stack<StatePair> stack = new Stack<StatePair>(); //存储遍历状态的堆栈
      ArrayList<PersonType> path = new ArrayList<PersonType>();  //类型序列
      ArrayList<String> term = new ArrayList<String>();  //人名特征词序列

      stack.add(new StatePair(path, offset, rules.rootNode, term));
      while (!stack.isEmpty()) { //堆栈内容不是空
         StatePair stackValue = stack.pop(); //弹出堆栈

         //取出图中当前节点对应的边
         EntityLinkedList edges = g.edges(stackValue.s1);

         //取出树中当前节点对应的类型
         Set<PersonType> types = stackValue.s2.getChildren().keySet();
         ArrayList<NextInput> ret = intersection(edges, types);
         if (ret == null)
```

```
            continue;
          for (NextInput edge : ret) { //遍历每个有效的输入
            //向下遍历树
            TrieNode state2 = stackValue.s2.getChildren().get(edge.type);
            //向前遍历图上的边
            int end = edge.end;
            if (state2 != null) {
                ArrayList<PersonType> p = new ArrayList<PersonType>(
                        stackValue.path);
                p.add(edge.type);

                ArrayList<String> t = new ArrayList<String>(stackValue.term);
                t.add(edge.term);

                stack.add(new StatePair(p, end, state2, t)); //压入堆栈
                if (state2.isTerminal()) { //是可以结束的节点
                    match.add(new MatchValue(state2.getNodeValue(), p, t));
                }
            }
          }
       }

       return match;
    }
```

特征词图的每个节点开始向后找规则。

```
    UnknowGrammar unknowGrammar = UnknowGrammar.getInstance();

    for (int i = 0; i < atomCount; ++i) {
       //从特征词图指定位置开始求交集
       ArrayList<MatchValue> match = unknowGrammar.intersect(g, i);
       //处理找到的未登录词
    }
```

为“程正泰的父亲是一位京剧票友，素与杨宝森先生交好。”准备识别模板：

```
    UnknowGrammar g = new UnknowGrammar();
    String right =
    "<Begin><surName>{surName}<doubleName1>{doubleName1}<doubleName2>{doubleN
ame2}的父亲是一位";                              //匹配人名的规则
    String handlerName = "PersonNamesd1d2";  //处理器名
    g.add(handlerName, right);               //人名处理器
```

表示文本中的人名的 PersonToken 类定义如下：

```
    public class PersonToken {
       public String person;                 //人名
       public int start;                     //开始位置

       public PersonToken(String person, int start) {
          this.person = person;
```

```
            this.start = start;
        }
    }
```

存储提取参数的 PairListPersonToken 类：

```
public class PairListPersonToken {
   List<Map.Entry<String,PersonToken>> args;  //存储类型和对应的 PersonToken

   public PairListPersonToken() {
      args = new ArrayList<Map.Entry<String,PersonToken>>();
   }

   public void add(String k,PersonToken v){   //增加键和对应的值
      Map.Entry<String,PersonToken> entry =
              new AbstractMap.SimpleEntry<String, PersonToken>(k, v);
      args.add(entry);
   }

   public PersonToken getFirst(String key){   //取得第一个键对应的值
      for(Map.Entry<String,PersonToken> e:args) {
        if(e.getKey().equals(key)) {
            return e.getValue();
        }
      }
      return null;
   }
}
```

提取人名的代码如下：

```
String text ="程正泰的父亲是一位京剧票友，素与杨宝森先生交好。";       //文法提取器
GrammarExtractor unknowFind = new GrammarExtractor();
List<PersonToken> result = unknowFind.extract(text);          //提取结果
System.out.println(result);
```

2.8　文本归一化

为了把日期和事件联系起来，可以归一化日期。

例如：

```
King George VI of England died on Feb 6, 1952.
```

可以转换成：

```
King George VI of England died on February 6, 1952.
```

归一化是识别数字、缩写、首字母缩略词和惯用语的过程，并根据需要将它们转换为全文。

想要识别的绝对日期将采用以下模式的规范化形式：

- 18 Feb 1997 → 1997/02/18
- 20th of July → XXXX/07/20
- 1992 → 1992

2.9 依存树模型

可以使用依存语言模型来建模单词之间的句法依赖关系。

使用依存文法构建依存语言模型。依存文法认为词之间的关系是有方向的，通常是一个词支配另一个词，这种支配与被支配的关系就称作依存关系，而且包括汉语和英语的大多数语言满足投射性。所谓投射性是指：如果词 p 依存于词 q，那么 p 和 q 之间的任意词 r 就不能依存到 p 和 q 所构成的跨度之外。

例如，汉语句子“这是一本书。”的依存文法结构如图 2-9 所示。

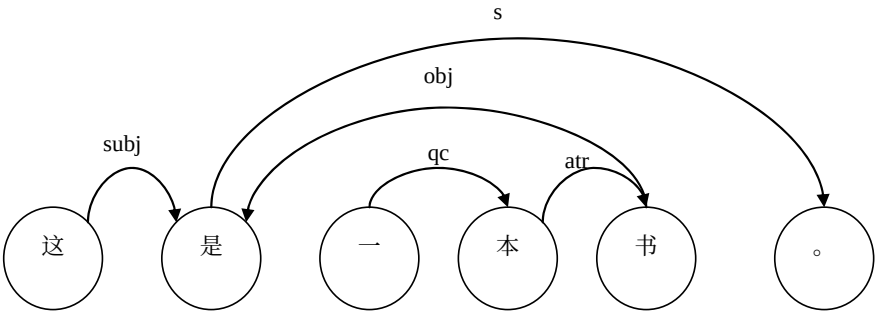

图 2-9 “这是一本书。”的依存文法结构图

图 2-9 中带箭头的弧的起点为从属词，箭头指向的是支配词，弧上标记为依存关系标记。例如，句号“。”支配“是”；动词“是”是句子的谓语，它支配主语“这”和宾语“书”，故“是”是支配词，“这”和“书”是从属词；“s”“subj”“obj”是依存关系标记。支配词也叫核心词，从属词也叫修饰词。

数词“一”作量词“本”的量词补足语，“本”是支配词，“一”是从属词，“qc”是依存关系标记。数量短语“一本”作名词“书”的定语，名词“书”支配量词“本”，“atr”是依存关系标记。

依存文法也可以表示成图 2-10 这样的树结构。因为总是连接线下面的词依赖上面的词，所以图 2-10 中的箭头可以省略。

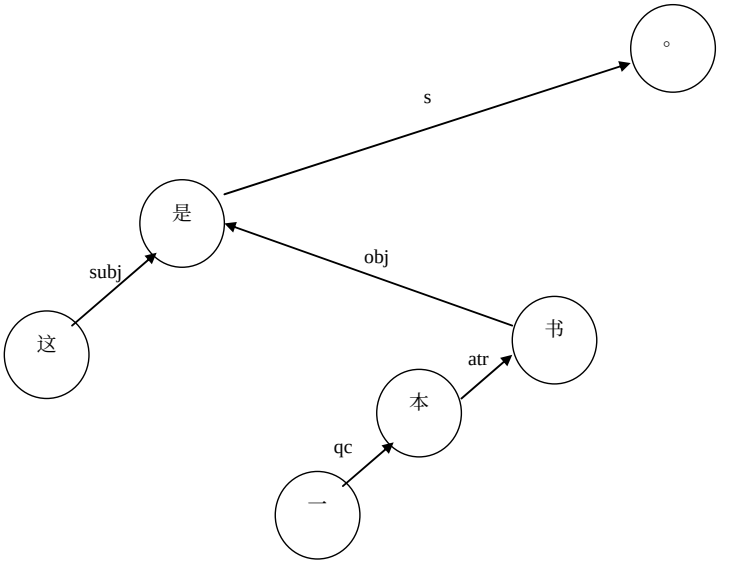

图 2-10 “ 这是一本书。” 的依存文法树

在依存语言模型中利用依存树有很多可能的方法。

构造依存语言模型的最简单方法是使用依存树的拓扑结构 T。每个单词都由其父亲调节。例如图 2-11 所示句子的概率计算公式如下：

```
    P (s | T )= P ( the | boy) P ( boy | find) P(will| find ) P (find | <NONE> )
P ( it | find )P (interesting |find)
```

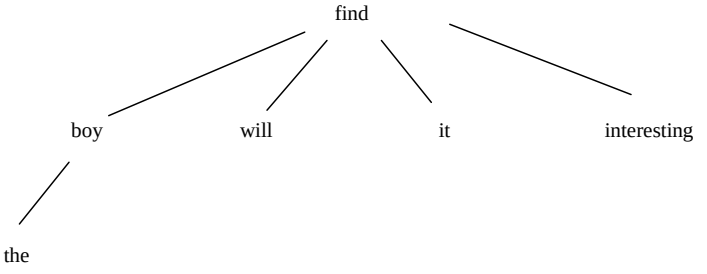

图 2-11　句子 “ the boy will find it interesting ” 的依存树

2.10　情感分析

人们常常会对某个事物（如产品）发表自己的看法或评论，计算机可以判断该看法

或评论是属于对该事物的积极或消极意见。这就是文本倾向性分析。可以结合核磁共振，通过对大脑中的兴奋区域成像来分析“喜欢”或“厌恶”等情感倾向。这里讨论的文本倾向性分析，英文称为 sentiment analysis 或 opinion mining。基本的目标就是实现区分出正面、负面或者中性，这称为极性分类（polarity classification）。可以按好恶程度分出更多的级别，例如 1~5 星级，这称为星级评分（multi-way scale）。

有文档级别的情感识别，例如对某个电影或酒店的评论自动分类出极性或者星级，这样区分出好评和差评。也许想进一步对好在哪里，差在何处做更细致的分析，所以出现了更细粒度的基于特征的情感识别。例如，区分出对手机的屏幕或者照相机的画质的评价。为了准确地识别级性，可以考虑对文本的主客观语句分类，提取出 *n* 个最主观的句子来概括整个评论的褒贬倾向。从技术上来说，就是从主客观混合文本语料中抽取表示主观性的文本。

为了实现基于特征的情感识别，需要从上下文提取出评价的对象，需要提取描述对象的特征，然后判断倾向性描述在每个特征上的极性。“特征”一词在这里既表示描述对象的组成，也表示属性。

特征抽取是获得关于主题某一方面的具体描述，如汽车的油耗与操控性，数码相机的电池寿命。和信息抽取相比，情感分析中的特征抽取更加自由，因为获得的结果不要求是结构化的。在某些应用中，特征抽取比情感取向判断更加重要，因为需要关注用户的具体意见。例如对某款手机的评价统计：

```
手机：
褒义：125 <独立的评价句子>
贬义：7 <独立的评价句子>
特征：续航能力
褒义：123 <独立的评价句子>
贬义：6 <独立的评价句子>
特征：大小
褒义：82 <独立的评价句子>
贬义：10 <独立的评价句子>
```

对事物的观点有直接观点和对比观点两种：

- 直接观点（direct opinion）：例如，这款手机的画质的确有点烂。
- 对比观点（comparative opinion）：例如，这款手机的画质比 iPhone-x 好。进行这类情感分析时，首先要确定观点的目标对象是谁。在这个例子中需要用到指代消解确定这款手机指哪款手机。

有时候，作者把情感和事实一起来表达，例如“即便是面对强逆光，iPhone-x 的表现也堪称专业”。也就是说，情感和具体的特征是分不开的。

除了这些经典的问题外，在针对社会媒体的情感分析中，我们面临更多的挑战。例

如，并非所有以与主题相关的用户为中心的内容都是重要的，而其中只有少部分能引起关注和讨论，甚至进而影响其他用户的观念和行为。因此，评估它们的影响力和预测它们是否得到关注具有重要的应用价值。

除此以外，不合理地利用社会媒体的影响力也值得关注。制造事端打击竞争对手，恶作剧心理造谣生事，收受商家好处为特定产品夸大宣传，是典型的误导公众行为。

首先从文本中抽取描述对象的特征。例如，针对汽车的用户体验信息，关于操控性、舒适性、油耗、内饰、配置等方面的评价等被分别抽取列出，因此可以收集到不同用户关于同一特征的描述并在不同品牌、不同时间段、不同用户群的范围内统计加以比较评估，这样的数据能直接地、准确地反映用户的消费情况和市场反应。其次，需要评估一个用户言论的内在价值和预测将来的关注度。从实务操作上来说，有些重要的言论和事件在几小时内就会引起广泛的关注。相关的厂家可以及时发现和跟进这种对其产品销售和品牌形象具有重要影响的言论。

为了获取标注好的文本倾向，可以从评论网站，比如 Booking.com 等网站抓取所有的酒店评论，这些评论用星级评价来代表褒贬度。

常见具有语义倾向词语的词性及示例如表 2-4 所示。

表 2-4　有语义倾向词语的词性表

词 性 编 码	词　　性	示　　例
a	形容词	美丽、丑陋
n	名词	英雄、熊市、粉丝、流氓
v	动词	发扬、贬低
d	副词	昂然、暗地
i	成语	宾至如归、叶公好龙
p	习惯语	双喜临门、顺竿爬

事实上，对一篇文章而言，它所表达的情感的正面或负面是通过主观语句体现出来的，如“产品质量好!”但是像“它的售价刚好是￥50 元!”这样的客观语句，虽然有“好”这一特征词，但并不表达任何情感。但是如果能区分一篇文章中的主观语句和客观语句，只对主观语句进行特征选择，会对分类的准确率有很大提高。

观点搜索系统使得用户能够查找关于一个对象的评价观点。典型的观点搜索查询包括以下两种类型：

- 搜索关于一个特定对象或对象特征的观点。搜索用户只要简单给出对象和/或对象的特征即可。
- 搜索一个人或组织关于一个特定对象或者对象特征的观点。用户需要给出观点拥有者的名字和对象的名字。

判断用户的情感取向（polarity）是喜欢、不喜欢还是中性的，可以通过对大量用户的感情取向进行统计，进而了解用户对特定产品的好恶，甚至对具体的某个特征（如数码相机的镜头、电池寿命等）作出直接的判断和比较。

开源机器学习框架 ML.NET(https://github.com/dotnet/machinelearning)包含了情感识别的实现。

要开始使用 ML.NET，请从包管理器安装 ML.NET 的 NuGet 包：

```
Install-Package Microsoft.ML
```

用于训练模型以预测文本样本中的情绪的代码如下：

```
var dataPath = "sentiment.csv";
var mlContext = new MLContext();
var loader = mlContext.Data.CreateTextLoader(new[]
   {
      new TextLoader.Column("SentimentText", DataKind.String, 1),
      new TextLoader.Column("Label", DataKind.Boolean, 0),
   },
   hasHeader: true,
   separatorChar: ',');
var data = loader.Load(dataPath);
var learningPipeline = mlContext.Transforms.Text.FeaturizeText("Features",
                                    "SentimentText")
     .Append(mlContext.BinaryClassification.Trainers.FastTree());
var model = learningPipeline.Fit(data);
现在从训练出的模型我们可以做出预测：
var predictionEngine =
    model.CreatePredictionEngine<SentimentData,
SentimentPrediction>(mlContext);
var prediction = predictionEngine.Predict(new SentimentData
{
   SentimentText = "Today is a great day!"
});
Console.WriteLine("prediction: " + prediction.Prediction);
```

如果需要采用 Java 实现，则 Stanford Core NLP(https://github.com/stanfordnlp/CoreNLP)中包含现成的实现。

2.11 本章小结

词性标注是简单而有用的语言学分析过程。例如，可以提取文本中的名词用作文本分类的依据。有很多常用词有多个可能的词性。早期的词性标注往往使用 HMM 这样的纯概率的方法来标注词性，后续增加了语法和语义知识来帮助提高词性标注的准确性。