最近在搞ES，结合了IK分词器，偶然间看到IK的主词典中有27万的词，加上其他的拓展词库差不多也有小一百万了，于是比较好奇IK是如何判断用户输入的词是否在词库中的，于是索性下载了IK的源码读一读，接下来是分词流程的解析。

首先先看一下主类，是一个用来测试的类

```
public class IKSegmenterTest {
   static String text = "IK Analyzer是一个结合词典分词和文法分词的中文分词开源工具包。它使用了全新的正向迭代最细粒度切分算法。";
   public static void main(String[] args) throws IOException {
       IKSegmenter segmenter = new IKSegmenter(new StringReader(text), false);
       Lexeme next;
       System.out.print("非智能分词结果：");
       while((next=segmenter.next())!=null){
           System.out.print(next.getLexemeText()+" ");
       }
       System.out.println();
       System.out.println("----------------------------分割线------------------------------");
       IKSegmenter smartSegmenter = new IKSegmenter(new StringReader(text), true);
       System.out.print("智能分词结果：");
       while((next=smartSegmenter.next())!=null) {
           System.out.print(next.getLexemeText() + " ");
       }
   }
}
```

然后从第一行的构造器点击去，这里解释一下这个useSmart。IK分词器有两种分词模式，一种是ik_max_word，这种模式下会对字符串进行最细粒度的拆分，如会将"中华人民共和国人民大会堂"拆分为"中华人民共和国、中华人民、中华、华人、人民共和国、人民、共和国、大会堂、大会、会堂等词语。而另一种ik_smart模式会做粗粒度的拆分比如拆分成"中华人民共和国、人民大会堂"。构造器源码如下

```
    /**
     * IK分词器构造函数
     * @param input
     * @param useSmart 为true，使用智能分词策略
     *
     * 非智能分词：细粒度输出所有可能的切分结果
     * 智能分词： 合并数词和量词，对分词结果进行歧义判断
     */
    public IKSegmenter(Reader input , boolean useSmart){
        this.input = input;
        this.cfg = DefaultConfig.getInstance();
        this.cfg.setUseSmart(useSmart);
        this.init();
    }
```

然后从getInstance点进去看一下如何构造的config

```
    /**
     * 返回单例
     * @return Configuration单例
     */
    public static Configuration getInstance(){
        return new DefaultConfig();
    }
```

在这里我们发现IK通过static搞了一个简单的单例模式，继续追踪new部分源码

```
  private DefaultConfig(){
        props = new Properties();

        InputStream input = this.getClass().getClassLoader().getResourceAsStream(FILE_NAME);
        if(input != null){
            try {
                props.loadFromXML(input);
            } catch (InvalidPropertiesFormatException e) {
                e.printStackTrace();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
```

可以看到这里读取了Resource文件夹下的配置文件放到了props里，也就是IKAnalyzer.cfg.xml，它的配置文件读取过程不像Spring那么复杂，直接就通过类库就完成了。这个文件主要是用来配置用户自己拓展的词库的，顺手贴一下配置文件。

```
  <?xml version="1.0" encoding="UTF-8"?>
  <!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
  <properties>
      <comment>IK Analyzer 扩展配置</comment>
      <!--用户可以在这里配置自己的扩展字典 -->
      <entry key="ext_dict">ext.dic;</entry>
      <!--用户可以在这里配置自己的扩展停止词字典-->
      <entry key="ext_stopwords">stopword.dic;</entry>
  </properties>
```

到这里算是把config初始化完成了，我们回到最上面来看init方法里面是如何初始化的

```
  /**
      * 初始化
      */
    private void init(){
        //初始化词典单例
        Dictionary.initial(this.cfg);
        //初始化分词上下文
        this.context = new AnalyzeContext(this.cfg);
        //加载子分词器
        this.segmenters = this.loadSegmenters();
        //加载歧义裁决器
        this.arbitrator = new IKArbitrator();
    }
```

这里的代码也是比较简单的，能看出来这里是通过刚才加载的配置文件来加载词库，加载分词器。但是上下文和裁决器还是不知道是什么，我们先看一下词典是如何初始化的

```
    /**
     * 词典初始化
     * 由于IK Analyzer的词典采用Dictionary类的静态方法进行词典初始化
     * 只有当Dictionary类被实际调用时，才会开始载入词典，
     * 这将延长首次分词操作的时间
     * 该方法提供了一个在应用加载阶段就初始化字典的手段
     * @return Dictionary
     */
    public static Dictionary initial(Configuration cfg){
        if(singleton == null){
            synchronized(Dictionary.class){
                if(singleton == null){
                    singleton = new Dictionary(cfg);
                    return singleton;
                }
            }
        }
        return singleton;
    }
```

我们可以看到这里用了一个面试中经常被问到但是我们从来没用过（手动狗头）的双重检查的方式实现了一个单例模式，然后点进去看一下new方法部分

```
  private Dictionary(Configuration cfg){
        this.cfg = cfg;
        this.loadMainDict();
        this.loadStopWordDict();
        this.loadQuantifierDict();
    }
```

我们可以看到这里load了三个词典，分别是主词典，停止词词典以及数量词词典，我们挑主词典的加载看一下

```
    private void loadMainDict(){
        //建立一个主词典实例
        _MainDict = new DictSegment((char)0);
        //读取主词典文件
        InputStream is = this.getClass().getClassLoader().getResourceAsStream(cfg.getMainDictionary());
        if(is == null){
            throw new RuntimeException("Main Dictionary not found!!!");
        }

        try {
            BufferedReader br = new BufferedReader(new InputStreamReader(is , "UTF-8"), 512);
            String theWord = null;
            do {
                theWord = br.readLine();
                if (theWord != null && !"".equals(theWord.trim())) {
                    _MainDict.fillSegment(theWord.trim().toLowerCase().toCharArray());
                }
            } while (theWord != null);

        } catch (IOException ioe) {
            System.err.println("Main Dictionary loading exception.");
            ioe.printStackTrace();

        }finally{
            try {
                if(is != null){
                    is.close();
                    is = null;
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        //加载扩展词典
        this.loadExtDict();
    }
```

首先它new了一个DictSegment对象，并传了一个0进去，我们点击去看一下。

```
  /**
   * 词典树分段，表示词典树的一个分枝
   */
  class DictSegment implements Comparable<DictSegment>{

      //公用字典表，存储汉字
      private static final Map<Character , Character> charMap = new HashMap<Character , Character>(16 , 0.95f);
      //数组大小上限
      private static final int ARRAY_LENGTH_LIMIT = 3;


      //Map存储结构
      private Map<Character , DictSegment> childrenMap;
      //数组方式存储结构
      private DictSegment[] childrenArray;


      //当前节点上存储的字符
      private Character nodeChar;
      //当前节点存储的Segment数目
      //storeSize <=ARRAY_LENGTH_LIMIT ，使用数组存储， storeSize >ARRAY_LENGTH_LIMIT ,则使用Map存储
      private int storeSize = 0;
      //当前DictSegment状态 ,默认 0 , 1表示从根节点到当前节点的路径表示一个词
      private int nodeState = 0;
```

通过查看DictSegment这个类，我们可以确认IK是使用的字典树对用户的输入进行的匹配，可以说是用空间换取了时间，这棵树的根结点是0，然后每个节点存储一个字符。

然后根据config中的默认配置把主词典的文件加载到内存中，这里是个循环，一次加载一行。这里给大家看一下主词典的文件，一行一个词，一共27万行

image.png

然后我们往下看是如何使用加载到内存中的词进行fillSegment操作的

```
    /**
     * 加载填充词典片段
     * @param charArray
     */
    void fillSegment(char[] charArray){
        this.fillSegment(charArray, 0 , charArray.length , 1);
    }
```

这里貌似没什么好说的，继续往下看

```
    private synchronized void fillSegment(char[] charArray , int begin , int length , int enabled){
        //获取字典表中的汉字对象
        Character beginChar = new Character(charArray[begin]);
        Character keyChar = charMap.get(beginChar);
        //字典中没有该字，则将其添加入字典
        if(keyChar == null){
            charMap.put(beginChar, beginChar);
            keyChar = beginChar;
        }

        //搜索当前节点的存储，查询对应keyChar的keyChar，如果没有则创建
        DictSegment ds = lookforSegment(keyChar , enabled);
        if(ds != null){
            //处理keyChar对应的segment
            if(length > 1){
                //词元还没有完全加入词典树
                ds.fillSegment(charArray, begin + 1, length - 1 , enabled);
            }else if (length == 1){
                //已经是词元的最后一个char,设置当前节点状态为enabled，
                //enabled=1表明一个完整的词，enabled=0表示从词典中屏蔽当前词
                ds.nodeState = enabled;
            }
        }

    }
```

首先这里把传入的这一行字符串中的第一个词拿了出来，放到一个用来存储汉字的公共字典里，至于为什么不像主词库一样放到文件里，可能是觉得维护两个文件比较麻烦，也可能后面能看到原因，但是这里最让我奇怪的是这个用来做公共字典的HashMap，它的负载因子没有用默认的0.75，而是用了0.95，这里是个需要之后思考的地方

```
    //公用字典表，存储汉字
    private static final Map<Character , Character> charMap = new HashMap<Character , Character>(16 , 0.95f);
```

然后走到lookforSegment方法，这里是将词库中的词加入到字典树的方法，也是一个比较重要的方法是，我们点进去看一下

```
  /**
       * 查找本节点下对应的keyChar的segment   *
       * @param keyChar
       * @param create  =1如果没有找到，则创建新的segment ; =0如果没有找到，不创建，返回null
       * @return
       */
    private DictSegment lookforSegment(Character keyChar ,  int create){

        DictSegment ds = null;

        if(this.storeSize <= ARRAY_LENGTH_LIMIT){
            //获取数组容器，如果数组未创建则创建数组
            DictSegment[] segmentArray = getChildrenArray();
            //搜寻数组
            DictSegment keySegment = new DictSegment(keyChar);
            int position = Arrays.binarySearch(segmentArray, 0 , this.storeSize, keySegment);
            if(position >= 0){
                ds = segmentArray[position];
            }

            //遍历数组后没有找到对应的segment
            if(ds == null && create == 1){
                ds = keySegment;
                if(this.storeSize < ARRAY_LENGTH_LIMIT){
                    //数组容量未满，使用数组存储
                    segmentArray[this.storeSize] = ds;
                    //segment数目+1
                    this.storeSize++;
                    Arrays.sort(segmentArray , 0 , this.storeSize);

                }else{
                    //数组容量已满，切换Map存储
                    //获取Map容器，如果Map未创建,则创建Map
                    Map<Character , DictSegment> segmentMap = getChildrenMap();
                    //将数组中的segment迁移到Map中
                    migrate(segmentArray ,  segmentMap);
                    //存储新的segment
                    segmentMap.put(keyChar, ds);
                    //segment数目+1 ，  必须在释放数组前执行storeSize++ ， 确保极端情况下，不会取到空的数组
                    this.storeSize++;
                    //释放当前的数组引用
                    this.childrenArray = null;
                }

            }

        }else{
            //获取Map容器，如果Map未创建,则创建Map
            Map<Character , DictSegment> segmentMap = getChildrenMap();
            //搜索Map
            ds = (DictSegment)segmentMap.get(keyChar);
            if(ds == null && create == 1){
                //构造新的segment
                ds = new DictSegment(keyChar);
                segmentMap.put(keyChar , ds);
                //当前节点存储segment数目+1
                this.storeSize ++;
            }
        }

        return ds;
    }
```

我们可以看到，IK的字典树存储规则是按是否超过ARRAY_LENGTH_LIMIT也就是3来决定的

1. 如果没超过3，则子节点用数组存储。

- 判断如果当前节点的存储个数在3及以下，先在数组中进行一个二分查找，如果能够找到就把值拿出来。
- 如果没有找到且允许新建的话，再判断是否容量超过3，没超过则直接使用数组存储，加入到数组中并进行sort，由于DictSegment重写了compareTo方法，所以是按存储字符的字典序进行排序的。如果超过3，则改为用map存储，将之前数组中的内容放到新建的map中并将数组的内存释放掉。

2. 超过3则用map存储。

- 搜索整个map，若有则把值拿出来，没有则put

之后我们递归执行这个过程，直到这一个字符串中的所有字符都加入到字典树中，当最后一个字符放入字典树时，会设置一个标记位，也就是之前的enabled，来表示当前是一个完整词的结尾。当主词典所有词都加入字典树之后，主词典加载完毕，然后加载扩展词典等其他词典，过程完全一致