ICTCLAS 代码学习笔记

中科院计算技术研究所多语言交互技术评测实验室

黄瑾　huangjin@ict.ac.cn

为了练手，打算写一个新的 ICTCLAS，用 STL 的东东，尽量避免预定义 BUFFER 造成的种种不确定性因素，同时了解一下整个分词的相关算法。

下面是学习笔记，看到哪记到哪。所有自己对代码的注释都用//!来表示，水平有限，写得不一定对@@。

2006-7-27 的学习笔记

关于 ICTCLAS 词典的组织

词典相关的操作都在自定义的类CDictionary里,相关文件为CDictionary.h和CDictionary.cpp；

涉及几个结构体变量，只注明了变量没有写构造函数。

```
struct tagWordResult{
    char sWord[WORD_MAXLENGTH];  //!可以用string代替
    //The word
    int nHandle;
    //the POS of the word
    double  dValue;
    //The -log(frequency/MAX)
};
typedef struct tagWordResult WORD_RESULT,*PWORD_RESULT;

/*data structure for word item*/
struct tagWordItem{
    int nWordLen;    //!可省略
    char *sWord;     //!可以用string代替，有写拷贝功能性能损失不大
    //The word
    int nHandle;
    //the process or information handle of the word
```

```
    int  nFrequency;
    //The count which it appear
};
typedef struct tagWordItem WORD_ITEM,*PWORD_ITEM;
/*data structure for dictionary index table item*/
struct tagIndexTable{    //!索引表，一次性读入应该可以用一个vector<WORD_ITEM>代替
    int nCount;
    //The count number of words which initial letter is sInit
    PWORD_ITEM pWordItemHead;
    //The  head of word items
};
typedef struct tagIndexTable INDEX_TABLE;

/*data structure for word item chain*/
struct tagWordChain{     //!词链，可以用一个list<WORD_ITEM>代替？
       WORD_ITEM data;
       struct tagWordChain *next;
       /*----Added By Huangjin@ict.ac.cn 2006-5-30----*/
       struct tagWordChain()
       {
           next=NULL;
       }
       /*-----------------------------------------------*/
};
typedef struct tagWordChain WORD_CHAIN,*PWORD_CHAIN;
/*data structure for dictionary index table item*/
struct tagModifyTable{
    int nCount;
    //The count number of words which initial letter is sInit
    int nDelete;
    //The number of deleted items in the index table
    PWORD_CHAIN pWordItemHead;
    //The  head of word items
};
typedef struct tagModifyTable MODIFY_TABLE,*PMODIFY_TABLE;
```

词典类的声明如下：

```
class CDictionary
{//!所有的输入参数为char*者都可以使用const string&来代替，至少应该是const char*
//!对返回值所用的char*使用string&来代替
//!对参数为int*的根据情况改为int&或者vector<int>&
//!其他int,double或者char等内置类型一律传值，其他复杂类型如果不涉及修改一律const &
```

```
public:
    bool Optimum();
    bool Merge(CDictionary dict2,int nRatio); //!这里应该用const CDictionary&
    bool OutputChars(char *sFilename);
    bool Output(char *sFilename);
    int GetFrequency(char *sWord,  int nHandle);
    bool GetPOSString(int nPOS,char *sPOSRet);
    int GetPOSValue(char *sPOS);
    bool GetMaxMatch(char *sWord, char *sWordRet, int *npHandleRet);
    bool MergePOS(int nHandle);
    bool GetHandle(char *sWord,int *pnCount,int *pnHandle,int *pnFrequency);
    bool IsExist(char *sWord,int nHandle);
    bool AddItem(char *sWord,int nHandle,int nFrequency=0);
    bool DelItem(char *sWord,int nHandle);
    bool Save(char *sFilename);
    bool Load(char *sFilename,bool bReset=false);
    int  GetWordType(char *sWord);
    bool PreProcessing(char *sWord,int *nId,char *sWordRet,bool bAdd=false);
    CDictionary();
    virtual ~CDictionary();
    INDEX_TABLE   m_IndexTable[CC_NUM];   //!这个换成vector<WORD_ITEM>在构造函数时
大小赋为CC_NUM
    PMODIFY_TABLE m_pModifyTable;
    //The data for modify
protected:
    bool DelModified();
    bool FindInOriginalTable(int nInnerCode,char *sWord,int nHandle,int *nPosRet=0);
    bool FindInModifyTable(int nInnerCode,char *sWord,int nHandle,PWORD_CHAIN
*pFindRet=0);
    /*----Added By huangjin@ict.ac.cn 2006-5-29----*/
    void ClearDictionary(void);
    /*----------------------------------------------*/
};
```

下面是有必要说明的类成员函数

CDictionary::Load 和 CDictionary::Save

主要说明内容为词典本身的存储结果，文件为二进制读写格式，存储 CC_NUM 个索引及相关值。

共有 CC_NUM 个索引项，对于每个索引的内容：

第一个 int 型数据的值为当前索引项的词个数 m_IndexTable[i].nCount，如果大于 0 表示该索引项后面有 m_IndexTable[i].nCount 个词，对于每个词：

前3个 int 型的数据分别表示每个词的出现频率 nFrequency、词条字符串长度 nWordLen 及其 nHandle。如果词条长度大于 0 则后面连续 nWordLen 个 char 型字符为该词的字符

表示。

Load 函数另外一个参数 bReset 主要是控制 nFrequency，如果该值为 true 则 nFrequency 的值为 0

词典的存储过程与读入过程正好相反，需要注意的一点是，存储时要考虑是否有修改词典的部分，即 m_pModifyTable 的相关内容。主要规则如下：

每个索引的词条个数应该为原有个数＋修改表个数－修改表删除个数

写入的顺序为（只针对有修改表的情况）

将修改表中的数据按词条字母序写入，如果词条字符串相等则优先写入 hHandle 值较小的。

2006-7-31 的学习笔记

在家里看代码，心静不下来啊，呵呵。

接着说 CDictionary 类的相关操作，save 函数原来没看太清楚，还得仔细看。

m_pModifyTable 的大小要么为 0 要么与 m_IndexTable 相同，即 CC_NUM 大小。如果 m_pModifyTable 非空则在保存时需要判断。对于第 i 个索引，写入的个数 nCount 等于原始的大小 m_IndexTable[i].nCount 加上新表中的个数并减去新表中删除的个数。后面两个怎么算见后。

如果还没有写够原始的个数 m_IndexTable[i].nCount 且修改表中还没有写完，则重复下面的操作

如果修改表中的当前词条

2006-8-2 的学习笔记

Save 函数不太容易看，先看一下 AddItem 和 DelItem 等会对 m_pModifyTable 有修改的部分再理解 Save 就容易得多了。

AddItem 需要三个参数，分别是要新添的词，其 handle 和频度。首先进行预处理，如果第一个字是汉字，则返回的是这个汉字的 handle 及传入词的剩余部分（即要新添的词）；如果是分割符则返回的是 handle 固定为 3755 且传入词的全部被返回（即要新添的词）（//?3755 的含义未知）。如果不是这两种情况则返回 false。预处理函数中有一个参数 bAdd 在当前版本中没有使用。只有预处理成功的词串才会被添加。首先会在原始表即 m_IndexTable 中查找是否已经有这个词了，如果找到，则对 m_IndexTable 及 m_pModifyTable 进行适当的修改并返回表示成功添加的 true。这里的修改主要根据在原始表中是否已经删除来区别操作，如果原始表中已删除，则重置这项的频度值，并在修改表中相应位置记录删除个数减 1，如果原始表中仍然有词，只要添加相应的频度即可。原始表中没有的情况会相对复杂一些。首先会在修改表中查找，如果找到了则添加相应的位置的频度值并返回 true。如果修改表中也没

有，那就要新建一个元素，并将其赋值到修改表的相应项的词串后面了，同时记录该记录的个数加 1。其实简单的说就是，如果在原始表中有该项，则设置相应的频率，否则如果在修改表中有该项，则设置相应的频度。如果两者都没有，则在修改表中的新添一个元素代表该项并插入到相应的位置中去。

DelItem 的过程可以说是 AddItem 的逆操作，其参数只需要待删除的词及其 handle 值。经过同样的预处理之后首先在原始表中查找，如果找到了将原始表中该项的频率值置为-1，在修改表中记录删除个数加一。需要注意的是，如果传入的 handle 的值为-1 即要求忽略 handle 删除所有相应词的项，需要遍历并修改 m_IndexTable 和 m_pModifyTabale 的值。如果原始中没有找到，那么在修改表中做类似的操作。在修改表中的操作会删除所有词相同且 handle 值相同或者词相同且传入 handle 值为-1 的项。如果在两个表中都没有找到则删除失败返回 false。注意，在修改表中做删除操作时并未对修改表中的 nCount 成员做任何的修改，怀疑为 bug//!。

DelModified 函数其实就是清空修改表，主要是要释放其中每个项的 sWord 所占用的 buffer。如果使用 string 及相应的 stl 类表示则这个函数只需要一条 clear 语句即可。

IsExist 函数用来查找给出的词条及相应的 handle 是否在原始表和修改表中的存在。

GetHandle 函数是一个比较重要的函数，用来查找给定词条相关信息，不仅仅是 handle 值。对于同一个词，可能有多个 handle 及相应的频率值。也是先在原始表中查找，如果没有找到在修改表中查找。

FindInOriginalTable 是在原始表中查找相应词条信息。传入的参数分别为汉字的内码，词以及 handle，返回值为找到的词匹配的位置，就是索引。用的是二分法进行查找，因为 m_IndexTable 中的词条本身是有序的。对于已删除的词条会做进一步的判断，会找到第一个匹配当前词的项的索引。

FindInModifyTable 是类似的一个操作，在修改表中查找相应的词条信息，不同的是返回值的类型为找到词条的前一个位置的指针，而且查找过程中是顺序查找的。

GetWordType 用于简单判断输入字的类型，如果输入字符串全为汉字则返回汉字类型，如果第一个是分割符则返回分割符类型，否则返回其他。只有汉字类型和分割符类型是合法的词典项类型。

预处理函数 PreProcessing 的功能前面已提过，这里需要注意的是，在预处理这个函数中会删除传入词串的首末空格，所以输入参数 sWord 不能是 const string&型。

MergePOS 函数是用于将词的词性信息压缩保存在 handle 里面。在目前的版本中未使用，会同时处理原始表和修改表。对于每一个索引项中的若干词条，如果某个词条与“前一个词”一样且当前词未被删除（即频率值不为-1）则么会标记该词为删除同时在删除表中的相应索引处的 delete 值加一。如果当前词条为第一个或者比“前一个词”小且未被删除，则置该词条的 handle 值为传入的参数。对修改

表的操作类似，对于每个索引项，依次处理索引项中的每个词条，如果当前词条比“前一个词”大则令当前词条的 handle 值为传入的参数，并重置“前一个词”，考察下一个词条；如果与“前一个词”相同，则删除该项。（不可能存在比前一个词小的情况）。该函数始终返回 true 值。

GetMaxMatch 也是一个重要的函数，多次调用。主要是根据传入的词条获取最大匹配的词条及其相应的 handle 值。首先通过预处理获取传入词条的 ID 值即内码和剩余的词串（如果第一个字是分割符则为全部词串）以及第一个字。在原始表中查找的时候使用的 handle 值是-1 因此会找到该词在原始表中出现的第一个位置。在找到结果之后会继续往后匹配直到确实找到词条完全匹配的结果之后返回。如果原始表中没有，则在修改表中进行类似的查找，由于修改表为链表形式，因此只能依次往后比较匹配。此函数中最后判断是否在修改表中找到相应结果的语句有问题，if(pCur!=NULL&&CC_Find(pCur->data.sWord,sWordGet)!=pCur->data.sWord)这一句中的红色不等号似乎应该为等号,//!bug 吧!。

GetPOSValue 用于将传入的词性字符串转换为相应的词性值。其规则为，如果传入字符串长度小于 3，则词性值等于第一位乘以 256 加上第二位的值，如果大于等于 3，则应该是由若干个加号连接起来的词性字符串表示，递归调用该函数，词性数值等于加号前的部分所得词性值乘以 100 以后加上后面的部分。，这里应该加入对传入参数合法性的判断。

GetPOSString 为 GetPOSValue 的逆过程，将传入的词性数值转换为相应的字符串表示输出。该函数始终返回 true。这个函数返回的词性字符串可能是上位词性也可能是下位词性（即可能为一个字母表示也可能为两个字母表示）

GetFrequency 即获得输入词条及相应的 handle 的频率值，查表即可，如果没有找到返回 0。

Output 和 OutputChars 两个函数分别将词典内容及结构输出到指定的文件中。需要说明的是，在修改模式下不会输出结果（即如果 m_pModifyTable 不为空的话直接返回）这里打开文件和判断是否为修改模式应该交换一下位置。后者只输出频率大于 50 的词串，不知道是为了减小输出的文本情况还是另有深意。

Merge 是一个将新词典合并到已有词典中的函数，除了词典以外还同时提供一个比例因子 nRatio 用于控制两个词典信息的权重。只有在原始词典中没有修改信息而且传入词典中没有修改信息时才会执行合并工作。该函数需要改进的第一点就是传入的参数应该使用 const&类型。函数的基本思想就是，如果词相同 handle 值也相同，则将两个词典中该项的频率值相加。如果已有词典中的项较小（词小或者词相同但 handle 值小）则修改已有词典中该项的频率值，即乘以一个系数 nRatio/(nRatio+1)，否则为已有词典中的词较大，要求新词典中该项的频率值大于(nRation+1)/10 的情况下将这个词通过调用 AddItem 函数加入到已有词典中，注意传入的频率值为新表中该词的频率值除以(nRatio+1)。

Optimum 函数用于优化已有词典 m_IndexTable。具体规则有两条，删除频率值为 0 的词条，删除词条相同但词性一个为另外一个父类的情况（因为子类词类提供更多的信息）。另外就是会删除词性为 x，g 和 qg 的词条（不知道为什么）

ClearDictionary 是我个人在修改版本中加入的清空 m_IndexTable 和 m_pModifyTable 的函数，主要是释放其中动态分配的一些内存。

关于 save 函数在有修改表下的操作。首先需要说明的是三个计数器的作用。原始表中的 m_IndexTable[i].nCount 是用来记录其 pWordItemHead 后一共跟了多少个词条的，并不会进行修改，如果在修改状态下删除了某些词条，则只是简单的将其频率值置为-1 不会删除这个词条，因此需要用修改表的相应位置的 nDelete 记录删除了几个词条。因此，在调用 AddItem，DelItemt 和 MergePOS 这三个会造成修改的函数中都会相应的对 nDelete 进行修改。而修改表中的 nCount 则是记录修改表中相应索引位置跟了多少个词条（新词），因此只有在 AddItem 和 DelItem 中需要修改，前者是新添了一个原始表中没有且现有的修改表中没有的词条时加一，后者是删除了一个修改表中的词条时减一，注意后面这个减一操作在现有的代码中没有，疑为 bug//!。在有修改表中的情况下，每个索引的词条数为原始表的 nCount 加上修改表的 nCount 再减去 nDelete 的值，即原始表中置了频率为-1 的个数。然后同时走原始表和修改表，哪个小就先写哪个，小即词条小或者词条相同但 handle 值小。唯一要注意的就是如果原始表中的词项频率值为-1 表示已删除不要写入。

关于词典类的总结。

词典类的主要数据存储结构为两个表，原始表 m_IndexTable 和修改表 m_pModifyTable，所有的操作都是围绕这两个数据结构进行的。前者为基本数据，从词典文件中读入，为预定义大小的 buffer（因为索引项的个数是固定的），后者为修改数据，只在修改模式下才分配内存并做相关的操作。原始表的索引项是固定的，但是每个索引项的词条数目是不固定的，类似于 vector<list<WORD_ITEM>>的结构，其中 WORD_ITEM 为每个词条项，为词、词的 handle 及频率 frequency。使用 list 结构不能使用二分查找法，也可以用类似 vector<vector<WORD_ITEM>>，因为写入格式中有每个索引项的词条个数这一项，可以预先分配 vector 的大小，不过 vector 的 vector 的情况是否会增加内存分配的开销？m_pModifyTable 是一个明显的 vector<pair<int nDelete, list<WORD_ITEM>>>结构，在现有的程序中也是使用指针遍历方式进行查找或者比较等。一个折中的策略是 vector 中都只放指针，即原始表用 vector<vector<WORD_ITEM>*>来表示，而修改表用 vector<pair<int, list<WORD_ITEM>*>>来表示。这样，对于修改表而言，也会减少一些内存的需求。现有算法中由于删除项目的存在，因此不得不使用一个变量来存储已删除词条的个数辅助为-1 的频率值。词典的另一个要求是每个索引项中的词条必须有序，这样主要是方便二分查找，对于输入的 handle 值为－1 的情况也可以依顺序找到第一个匹配的词条。如果换做 map 一类的结构来存储可以保证有序，查找也很快，就是构造的时候比较慢一些，而且 map 的键值会比较复杂，因为一是词本身，二是其 handle 值，而 map 的映射值为其频率值。需要自定义比较函数，数据结构可能会复杂一些。修改表也要用类似的结构。

方案 1：

原始表：vector<vector<WORD_ITEM>>

修改表: vector<pair<int, list<WORD_ITEM>>>

方案 2：

原始表：vector<vector<WORD_ITEM>*>

修改表：vector<pair<int, list<WORD_ITEM>*>>>

方案 3：

原始表：vector<map<pair<string sword, int nhandle>, int nfrequency>>

修改表：vector<pair<int ndelete, map<pair<string sword, int nhandle>, int nfrequency>>>>

目前考虑用第二种方案。

Utility.h 和 Utility.cpp 是一些常调用的公有函数的集合。声明了一些宏和字符串操作的一些函数，具体说明如下：

定义了句子开始标记 CT_SENTENCE_BEGIN 和结束标记 CT_SENTENCE_END。定义了字符（包括单字节和双字节）的类型属性：单字节字符 CT_SINGLE，一般为 ASCII 码范畴字符；分割符 CT_DELIMITER 主要为双字节情况，对于单字节的分割符在当前版本中也使用 CT_DELIMITER 表示，但是在我的改进版中引入了 CT_SINGLE_DELIMITER 来区别；CT_CHINESE 为汉字；CT_LETTER 为双字节字母；CT_NUM 数字符号，只包括双字节的情况，在我的改进版中引入了 CT_SINGLE_NUM 区别单字节的数字字符。CT_INDEX 为索引字符，其他的双字节字符都使用 CT_OTHER 表示。//?需要再确认一下的是全角的空格是哪一类？

定义了一些常用的后缀，包括单字的 POSTFIX_SINGLE 和多字的 POSTFIX_MUTIPLE；

定义了一些译名常用汉字，包括英语译名常用字 TRANS_ENGLISH、俄语译名常用字 TRANS_RUSSIAN 及日本人名常用汉字 TRANS_JAPANESE。特别注意一下的是前两个常用字中“•”这个特殊的字，而真正使用时这个字不应该在人名的第一个。相应的，定义了三种译名翻译类型 TT_ENGLISH，TT_RUSSIAN 和 TT_JAPANESE。

定义了一些分割符，包括全角的句子分割符 SEPERATOR_C_SENTENCE，全角的子句分割符 SEPERATOR_C_SUB_SENTENCE，半角的句子分割符 SEPERATOR_E_SENTENCE，半角的子句分割符 SEPERATOR_E_SUB_SENTENCE 及连接符 SEPERATOR_LINK。在这些定义中有几点要说明一下：

1. 全角的句子分割符中列出了全角的冒号：但是其实还有另外两种表示方法：和：。
2. 全角的子句分割符中列出了全角的双引号“”及单引号‘’，但是其实有其他几种表示方法，分

别是〝〞 ‘ ’ ` ´ 。

3. 连接符中除了/n 和/r 以外，还有全角和半角的空格，也就是说，如果输入的汉语句子中有空格，则被视为天然的分割符。//!应该补充一下制表符的判断，因为也是天然的分割符。

4. 无论是全角还是半角分割符都没有列出点(. . .)原因是点本身的属性具有歧义性，必须要按上下文相关内容进行判断。

定义了句子起始字符串 SENTENCE_BEGIN、SENTENCE_END 及在计算过程中的两个词之间的词分割符 WORD_SEGMENTER

下面是所有函数的说明及其注意事项：

bool GB2312_Generate(char *sFileName);

生成 BG2312 编码的所有字符的文件，写入 sFileName 中。GB2312 编码高位从 161 到 255，低位从 161 到 255，均为左闭右开区间。

bool CC_Generate(char *sFileName);

生成汉字列表及编码到 sFileName 文件中，其中汉字的编码高位从 176 到 255，低位从 161 到 255，均为左闭右开区间。

char *CC_Find(const char *string, const char *strCharSet);

常用函数，用于双字节字符子串的匹配，要求 string 的内容均为双字节。如果找到则返回字符串指针否则返回 NULL，常常用于判断一个汉字是否为指定的若干个汉字中的一个。

int charType(unsigned char *sChar);

判断传入的单字节或者双字节字符的类型。常用函数。该函数中的类型判断对后续的粗分及其他操作都有重要的影响，本人加入了对于 CT_SINGLE_DELIMITER 和 CT_SINGLE_NUM 的处理，前者所包含的半角字符的类型与数字串和时间词串的识别有重要的影响，要十分慎重。例如-的判断目前不敢加入到 CT_SINGLE_DELIMITER 中。

unsigned int GetCCPrefix(unsigned char *sSentence);

求最大的汉字串前缀，返回输入字符串中非汉字的第一个位置，注意汉字的编码范围为 175 到 248，均为开区间。

bool IsAllChinese(unsigned char *sString);

这个函数前的注释部分有误，写的是 IsAllSingleByte 函数的相关说明，在新版本中要注意改掉。这个函数用于判断输入串是否均为汉字组成。

bool IsAllNonChinese(unsigned char *sString);

与上一个函数相反，判断是否全都不是汉字，只要有一个汉字即返回 true。

bool IsAllSingleByte(unsigned char *sString);

判断是否全为单字节。

bool IsAllNum(unsigned char *sString);

这个函数判断输入的字符串是否全为数字，原有的函数被改写了。这个函数主要判断全角和半角的阿拉伯数字组成的相应字符串是否为数词，同时要处理的还有前缀“第”及表示正负的前缀符号，词串中可能出现的连接符例如冒号、点及表示分之或者除的左斜杆，另外就是词末的表示单词的百千万等后缀。//?很重要。

bool IsAllIndex(unsigned char *sString);

判断输入的字符中是否全由索引字符组成。除了类型判断为 CT_INDEX 的字符之外还包括大小写字母的情况。

bool IsAllLetter(unsigned char *sString);

判断输入的字符是否全由类型为 CT_LETTER 的字符组成。

bool IsAllDelimiter(unsigned char *sString);

判断输入的字符是否全由类型为 CT_DELIMTER 的字符组成，注意这里是双字节字符。

int BinarySearch(int nVal, int *nTable,int nTableLen);

二分查找函数，返回索引，找不到返回-1。

bool IsForeign(char *sWord);

判断输入的字符串是否为外来词翻译，计算输入串中含有的外来词的个数，如果字符串长度大于 2 或

者外来字个数不少于长度的一半则认为可能为外来语。

bool IsAllForeign(char *sWord);

判断是否全为外来词，即要求字符串中含有的外来词的个数恰好等于字符串的长度，而且都是双字节的。

bool IsAllChineseNum(char *sWord);

另外一个判断是否为数词表达式的函数，主要判断的是全角的汉字数字例如零一二三等，会加入常用数词前缀，但是原始版本对于前缀与汉字数字的组合情况没有做一定的规则约束。另外就是汉字〇〇是两个不同的字符没有补全。目前版本仍然不够完备。

int GetForeignCharCount(char *sWord);

主要就是根据预定义的译名常用汉字统计出字符串中可能为译语人名的汉字的个数，取最大的那个。

int GetCharCount(char *sCharSet,char *sWord);

这个函数用来判断输入的 sWord 中含有多少个已给定的字符列表中的字符的个数，注意 sCharSet 可以即有双字节字符又有单字节字符，返回的值以字计算，即双字节也计一个。

int GetForeignType(char *sWord);

根据输入的词串中译语字的个数来猜测人名的类型，取匹配字数最多的那一个，如果字数相同，则英文人名优先于俄语人名优先于日语人名。如果即要知道匹配的字符又要知道是哪个类型，在 GetForeignCharCount 和当前这个函数中会有重复操作。

bool PostfixSplit(char *sWord, char *sWordRet, char *sPostfix);

顾名思义是进行后缀的切割工作。后缀主要是地名所用到的内容。优先匹配多字节的部分，其次匹配单字节的情况，如果找到则将输入的字符串切割成两部分。如果没有找到，那么不切割输入的词串，后缀 buffer 置为空。该函数始终返回 true。

对该文件的总结：

主要是常用的操作，有一些操作的代码，例如在判断是否全部为汉字的函数 IsAllChinese 中，仍然对其字节值进行了大于 175 小于 248 的判断，同样的判断在 charType 函数中也出现。直接在 IsAllChinese 函数中调用 charType 来判断返回值是否为 CT_CHINESE 可能会有性能上的损失，可以写一些 inline 函数将这些重复代码整合到一起，毕竟使用裸的数值本身就不太好，更何况在多处使用。

另外，纵观代码的其他部分，常常使用到一个操作就是从字符串中的某个指定位置读入一个字到 buffer 中，可能是双字节也可能是单字节，并相应修改下次读入的位置，这部分重复操作很多，应该写成一个函数供调用。事实上，2.0 以后的版本中确实有该函数，形式为 unsigned int Getchar(char *sBuffer, char *sChar);其中的返加值为读入的字符串的长度，可能为 0、1 或者 2。

CdynamicArray 是一个动态二维数据实现，而且是稀疏矩阵的实现策略，同时使用单链表进行存储。记录着每个结点的行列值、词的字符串表示方式、字符串长度，词性及 value 值。其实可以简单的使用一个 list 来实现。有趣的一点是，这个动态数组可以是行优先也可以是列优先的。动态数组提供以下功能：

SetRowFirst：设置行优先或者列优先：主要是在内部通过一个 bool 量来标记；

获取动态数组的头 GetHead 或者尾 GetTail，如果是尾的话同时返回元素个数。

GetElement：有两种重载形式，一种获取给定位置上的元素的 value 值，如果需要同时返回词性和词的字符表达形式。另外一种是从指定位置开始找，其实就是加快速度一些。

SetElement：设置指定位置上的元素，如果有则修改，如果没有则分配并插入到合适的位置。

SetEmpty：清空动态数组，主要是释放保存字符串所分配的空间。

另外重载了赋值操作符和等于操作符。

动态数组类的总结：

完全可以使用 list 来代替，最多重载 list 加入一个表示行优先还是列优先的成员变量。如果使用 list 可能要注意的就是返回值类型为位置指针的那几个，呵呵。

2006-8-4 的学习笔记

主要看 Cqueue 类，即 Queue.h 和 Queue.cpp 两个文件。Queue 所包含的结点类型主要是用于处理最短路径时的相关信息，即

```
struct tagQueueElem{
    unsigned int nParent;   //!父结点的序号
     unsigned int nIndex;//number of index in the parent node，父结点中的序号
     ELEMENT_TYPE eWeight;//the weight of last path，路径的权重
    struct tagQueueElem *next;
};
```

Cqueue 类是一个有序的队列，主要提供 pop 和 push 的函数接口，需要注意的是，pop 函数有两个附

带参数。另外还提供了判断是否为空及是否只有一个元素的函数接口，具体如下：

Push 函数传入的三个参数分别是父结点的值，父结点中的索引及权重，push 就是将其插入到合适的位置，单链表是按权重值从小到大排序的，所以严格的说并不是先进先出队列。

Pop 的操作也与普通的 pop 函数不同，除了获取队列的第一个元素的相关值以外，bFirstGet 用于判断是否是第一次取元素，这个标记帮助定位 m_pLastAccess 的位置到队列头部。另外一个变量 bModify 则用来标记是否在 pop 的同时删除这个元素，这两个变量不会同时生效，即如果是修改模式，则每次都是直接从头结点删除的，否则，使用 m_pLastAccess 来记录上一次取的位置。采用这样的消极浏览方式是有好处的，例如需要多次遍历队列时。

由于采取了上面这种消极的策略对于是否为空函数同时也需要提供一个标记，在浏览状态下判断 m_pLastAccess 是否到头，而在普通模式下判断头结点是否为空。

判断是否只有一个元素则没有区别浏览状态还是普通状态。

Cquece 类的总结：

与其类名有一些出入，具体实现是一个以权重 weight 大小排序的单链表，对于相同大小权重的结点，则是后进来的排在前面。该类的优点在于提供了一个浏览状态和修改状态，以区别不同的 pop 操作，前者只是按顺序遍历整个链表，后者则同时删除第一个元素。可以考虑使用重载 list 来实现，需要加入的一个成员变量记录在浏览状态下上次访问的那个元素位置即可。

N－最短路径算法类，这个是核心算法之一，用于进行中文词语的粗分，算法的相关理论可参见论文《基于 N-最短路方法的中文词语粗分模型》。

类中包含的几个成员变量依次说明如下：

m_apCost 是一个动态数组的指针，用于记录词图中的各结点及其路径的权重值；

m_nValueKind 是无符号整型数据，用于记录共生成几个结果，一般为 1，也可能生成多组结果；

m_nVertex 是无符号整型数据，用于记录词图中结点的个数，取值为词图的行列值中较大的一个再加 1。

m_pParent 和 m_pWeight 都是二维数组指针，前者用于记录每组结果中的每个结点的前驱后者记录 N 个最短路径值。例如传入的词图为 5*4，只要求生成一组结果，则 m_pParent 为[5][1]的 Cqueue 二维数组，而 m_pWeight 为[5][1]的 double 二维数组。前者记录每个结点处的父结点及相关信息，后者记录相应的权重值。

最短路径算法主要调用的函数是 int CNShortPath::ShortPath()。这个函数根据成员变量 m_apCost 中记录的词图中的各个边的相关信息计算出路径并保存在 m_pParent 和 m_pWeight 中。

从第一个结点开始（0 结点为起始结点），对每个结点

读取列值为当前结点的边链表，该边的行值记为前一个结点，该边的 value 值记为权重。然后产

生所需要的 m_nValueKind 个结果，如果前一个结点已经是起始结点（为 0）则将这个结点及其当前权重值压入队列；如果还没有回到起始结点则将这个结点以及当前权重与前一个结点的权重值相加的新权重值压入。需要注意的是，如果前一个结点已经无路可走，即权重为无限值 INFINITE_VALUE 则跳出生成过程。生成结果之后，对当前结点的前一个结点的权重值都置为无限值，然后按权重从小到大依次取出每个结点，如果前一个位置的权重值为无限即没有被走到过则置为当前读出的这个权值，否则如果前一个位置的权重值比读出的这个还要小，那就记录到后面去。记录权值的同时不要忘记还要把前驱结点记录到 m_pParent 中去。这里要说明的另一点是，由于队列中的结果是按权值从小到大排序的，因此不会再现已记录的结果比后面的结果权值大的情况。相等的情况也会在前驱表中有相应的记录。

GetPaths 是另一个重要的函数，虽然其类型为 private 的只供内部使用。函数的声明如下：

void CNShortPath::GetPaths(unsigned int nNode,unsigned int nIndex,int **nResult,bool bBest)

第一个参数是结点 id，第二个参数是结果索引，第三个参数是用来存储路径的二维数组，bBest 为真时只取最好结果忽略其他。算法是回溯从 nNode 处往前找到起点的路径。

首先将传入的 nNode 及其结果索引 nIndex 压入队列，从这个结点开始往前走，即每次读当前结点的父结点中的第一个侯选（即前驱结点中的第一个候选），只要有前驱就更新当前结点并注意将合法的前驱结点及其索引值保存起来，直到到达起始始点，注意这时是浏览状态。如果最后得到的这个结点是起始结点（结点值为 0）则认为找到了一条路径，记录这条路径在 nResult 中。达到最大结果个数或者只要求一个最好结果就返回，否则回溯找下一条路径，直到队列为空或者达到了最大结果个数。

Output 用于将路径结果写入二维数组中，如果结点数不超过 2，那么只有唯一的路径，即只生成了一个结果；否则的话从尾结点回溯调用 getpaths 来获得路径值。最终要几个结果就循环几次，需要注意的一个地方是参数 bBest 用于控制是否只生成一个最优结果即返回。同样，每个结果中的路径候选不超过 MAX_SEGMENT_NUM（10）个。

CshortPath 类总结：

利用已有的词图信息求最短路径，是改进的最短路径算法。核心的两个函数是 GetPaths 和 ShortPath。

CsegGraph 类的相关笔记，这个类为词图类，主要是词的有向图。提供两个核心函数，一个是原子切分，一个是生成词图。为了保存相关的信息，有三个成员变量 m_sAtom，m_nAtomLength 和 m_nAtomPOS 分别存储句中的词（字符串）、每个词的长度（字符串长度）及该词的词性。由于都是预定义大小的 buffer，因此还需要一个无符号整型变量 m_nAtomCount 来记录切分后词的个数。另外，在粗分时可能产生的多个切分结果存储在一个动态数组 CdynamicArray 类的对象 m_segGraph 中。

这里的三个成员变量都是预定义大小的 buffer。句子长度不超过 2000 个字符，而且词长不超过 200。如果改用 vector 来存储则可以避免这种预定义在某些极端情况下造成访问越界的情况。可以考虑用一个 vecotr<struct{}>来存储，其中结构体只需要两个变量，一个是 string 类型的词串，一个是其词性

nPOS，而原来需要的词长这个属性 string 本身就可以提供了。

词图要求为行优先存储。原子切分这个环节，原始的版本判断比较简单，将句子起始符号和结束符号当作一个整体切开，如果读入的类型是汉字（CT_CHINESE）、索引（CT_INDEX）、分割符（CT_DELIMITER）或者其他（CT_OTHER）则切开成一个单独的结点。否则（就是数字、单字节字符或者字母），如果类型相同则连在一起，否则就切开成单独的原子结点。将这些粗分好的结果分别存入三个成员变量中，而词性一行的值就直接为 charType 函数返回的结果，只是一个初步的判断。目前版本的函数中只会重置 m_sAtom 和 m_nAtomLength 而 m_nAtomPOS 在结束位没有相应置空，目前不会引起 bug。

需要说明的是，原子切分这个环节，对于由汉字组成的数词串都切开了而在后面并不一定都能很好的合起来，一方面是概率竞争不过一方面是判断数词串的函数有不全面的部分。另外，由于我修改的版本引入了两个新的类型，因此在原子切分时需要特别处理一下。

GenerateWordNet 函数根据核心词典将原始的句子生成词图并存储在 m_segGraph 中。首先会将句子进行原子切分，然后利用词典中的词及词性词频信息来置相应的权重值。第三个参数 bOriginalFreq 的默认取值为 flase，当这个值为 false 时为非原始的频率值，一般采用默认值最大的或者 0 来设置，具体规则见后。如果为 true 则是利用词典中获取的频率值来做相应的设置。

对于粗切好的每个原子结点，在非原始频率的情况下，如果类型为汉字，则设置词图，其 value 值取 log 最大频率，词性为 0 即未知状态，词本身也不需要存储；如果为其他类型，则需要额外判断一下是不是一些特别的类型，比如说 CT_INDEX、CT_NUM 和 CT_SINGLE_NUM 都被判断为未处理数，设置相应的词性-27904 及 value 值（0），分割符为 30464 而 value 值为最大频率等，不一一列举。value 值为 0 而词性为修改后结果。

第二步词图生成是根据已有的词典，会有一个规则是不要把“月”和“份”切开，这个不太合时宜，呵呵。然后对于粗分后的每个词，如果在词典中找到最大匹配，则需要重新估算其频率值，同样的有一个对于组合数词的判断。如果生成的结果只有一个词性则更新 value 值时同时置这个明确的词性，否则只更新 value。这一步是个最大匹配的过程，而且每个匹配的中间部分都会被记录下来。（粗分结果），在其中还会有一个关于时间词串的扩展规则，也不太合时时宜，主要是针对类似“1 年内、1999 年末”这种词，将末尾的诸如“末内中底前间初”的词处理一下。

词图类 CSegGraph 的总结：

根据论文，这是第五层和第四层、第三层的 HMM 过程，非常重要，是后续工作的基础。针对数词串的处理还不够完美，考虑是换个地方处理还是重新完善一下规则。

2006-8-7 的学习笔记

Csegment 类的相关学习笔记。

啥也没写，汗

2006-9-5 的学习笔记

Csegment 类的相关学习笔记，重拾炉灶，嘿嘿。

CSegment类是一个比较重要的类，词语切分的核心工作是在这一部分完成的。成员变量m_pWordSeg是一个指向WORD_RESULT 的二维指针（[MAX_SEGMENT_NUM][MAX_WORDS]）。注意二维数组的大小是在构造函数中预定义的，MAX_SEGMENT_NUM为10，MAX_WORDS为650，有可能会溢出，程序对于MAX_SEGMENT_NUM做了控制，但是MAX_WORDS却没有做控制，存在潜在危险。这个二维数组用来存切分以后的词而且是个公有的成员变量。另外，类还有两个主要的公有成员变量m_graphOptimum（CDynamicArray类型）和m_graphSeg(CSegGraph类型)，分别存优化的分词词图和词图。注意二者的类型不同，具体使用见后。

另外，还有两个私有成员变量，m_npWordPosMapTable是一个中间结果，用于存储候选词的位置，而m_nWordCount则用来做个计数。

CSegment类的成员变量总结起来如下：

```
class CSegment
{
public:
    PWORD_RESULT *m_pWordSeg;     //!分词的词串结果
    int m_nSegmentCount;          //!N-ShortPath的个数N
    //The segmentation result
    CDynamicArray m_graphOptimum;//The optimumized segmentation graph
    CSegGraph m_graphSeg;//The segmentation graph
private:
    int *m_npWordPosMapTable;//Record the position map of possible words
    int m_nWordCount;//Record the End position of possible words
};
```

使用CSegment类的相应函数涵盖着HHMM多层结构中的第5层（原子切分）、第4层（简单未登陆词识别）和第3层（递归未登陆词识别）。前后数据处理均在此类中实现。其处理过程及各个函数分析如下：

对于传入的一个句子的子句（这里是指被预定义的分隔符切开的部分），先对其进行2元切分（BiSegment），然后对于生成的每个结果（m_nSegmentCount），分别进行未登陆词识别（人名、外国人名及地名）识别结束后对结果进行优化（BiOptimumSegment）。优化的结果通过m_pWordSeg传给词性标注器进行词性标注，最后对结果进行排序并输出。

构造函数和析构函数就是分配和释放相应的空间，因为空间是根据预定义的宏的大小来分配的，可能会出现溢出的情况。另外就是原始代码中对空间的释放不完全，对于m_pWordSeg[i]的释放

应该使用delete[]，而且也没有释放m_npWordPosMapTable的相应空间，在修改的代码中均已做了处理。

Segment函数在实际的代码中并没有用到，和BiSegment函数的功能类似，但是后者多利用了一些信息。Segment函数首先分配存储路径的空间，然后根据传入的子句和核心词典生成词图，并根据词图的结果进行NSP算法计算相应的路径，最后输出结果。而BiSegment函数还引入了平滑参数dSmoothingPara及二元词典dictBinary来进行切词。因此在生成一元词图之后还要根据二元词图进一步生成二元词图以传入NSP算法类中进行路径计算。生成的结果也要先经过一步由二元路径到一元路径的转换之后才输出结果。

GenerateWord函数是一个重要的函数，它根据传入的二维数组nSegRoute（指针）中相应的路径将分好的词的相关信息依次拷贝到m_pWordSeg[nIndex]的各个元素，注意这里没有对nSegRoute[nIndex][i]和[m_pWordSeg[nIndex][k]中的i及k做任何越界的测试，而实际上这两个buffer的大小都是有限制的，这也就为日后较大较长的句子切分造成越界埋下了隐患。这里的nIndex是指生成多个分词结果时的结果索引，一般我们只得到一个结果也就多为0，而且nIndex不会超过最多结果MAX_SEGMENT_NUM（在生成时做了判断）因此不会越界。

在生成词的过程中，主要是考虑到了数词串及时间词串的差别及合并问题。即根据当前切词的结果，考虑其后面的连续若干个词合并的结果，如果合并结果是一个数词的话，那么就保留这个合并结果。这里有一个小的问题就在于诸如“二分之一”这样的例子，会在前面的粗分时被切为“二分”“之一”，前面一个候选词送入此循环时不被判断为数词因此合并后面结果之后的“二分之一”不会进行循环被判断为数词，造成切分错误。另外就是通过一些规则来判断类型是否为时间串“未##时”等。在时间词部分的判断上，对于“月份”、“点钟”、“刻钟”这一类较明确的时间词后缀没有做处理，另外对于时间词串的判断也过于粗略，例如“3．15分”、“2000分”等也被判断为时间词。对于“点”这个容易引起歧义的时间词后缀(数词表示小数点)的判断也不够完善，诸如“三点五十分“这种例子前面的“三点“词性标注并不正确也是由于此处判断不足造成的。对于年代词专门有一个成员函数来进行判断，而其他时间词的判断不够完善。

函数的最终结果是根据路径信息nSegRoute[nIndex]，将切好的每个词都送入m_pWordSeg[nIndex]的相应位置。个人觉得这部分代码显得比较凌乱，一是没有越界的判断，二是规则多，代码本身也较长，容易潜藏错误。

2006-9-6的学习笔记

继续读CSegment，感觉时间上压力好大，唉。

OptimumSegmet函数和后面的BiOptimumSegment函数也是对应的关系，前者处理一元切词，二者则是二元的，分别对应Segment和BiSegment。(所以OptimumSegmet在目前的代码中其实也没有用到)。其实过程都很简单，就是新开一块空间，使用优化词图m_graphOptimum中的内容重新进行一次NSP算法，抛弃原来的词图，使用优化词图并根据新开辟空间中的路径结果生成切分结果(调用GenerateWord函数)。BiOptimumSegment函数的基本过程如上，就是类似BiSegment函数中加入了二元词典、平滑参数，并且NSP算法用的是二元词图，后成的路径结果还要改为一元路径。最后，不要忘记释放分配的空间。

GetResultCount函数比较简单，就是根据传入的“链”计算“有效”元素的个数。这里的“有效”就是指词串非空，那么之前的清理工作就一定要注意了。程序本身没有对传入的参数的有效性做判断，不是很好。

GetLastWord函数也如其名，根据传入的“链”求最后一个“有效”元素的词串。要么通过调用GetResultCount来求得最后一个元素的索引值来直接定位，要么一边搜索一边保存现场。程序本身采用的是第二种方法，但是中间过程的拷贝其实都是没有必要的，浪费了时间。另外，程序本身也没有对传入参数的有效性做判断(指针是否为空)，有潜在的危险。

IsYearTime函数用于判断传入的字符串是否为年代词，接受大概以下几种格式的输入为年份：

1. 诸如XXXX，其中X为单字节，这里判断可能有误，例如输入为AAAA此类；
2. 诸如XX，其中X为单字节且第一个字符的值大于'4',形如50年，92年之类；这里的4不知道是怎么确定的；同样存在上面的误判断的问题；
3. 全是数字且长度不小于6，例如“５８８年”、“１９８９年”
4. 全是数字且长度为4且头一个字符是“５６７８９”中的某个，例如５６年；
5. 由“零〇〇一二三四五六七八九壹贰叁肆伍陆柒捌玖”组成的两字以上串，程序中判断组成字符数时稍微有些问题，不应该nLen/2而是把2乘到等号左边，因为除2存在取整问题；
6. 四字串且包含两个“千仟零〇〇“，例如二仟零二年
7. 一字串为“千”或“仟”，后面跟年也构成时间串
8. 两字串为古时年代表示方法，诸如“甲子年”。

BiGraphGenerate 函数用于生成二元词图，传入的参数 aWord 中已经包含了一元词图的信息，而生成的结果则写入 aBinaryWordNet 中。函数中通过一个动态大小的一维数组 m_npWordPosMapTable 记录下一元词图中的路径信息，然后遍历一元词图中的所有结点，对于每个结点，获取词的频率值，求得当前结点词的后一个词，如果允许其组合，则查二元词典 DictBinary 获得其组合概率并更新 aBinaryWordNet 的相应位置的值。 程序中的变量 pTail 似乎没有啥用处。

BiPath2UniPath 函数将二元路径改写为接受输出的一元路径。根据原始的一元词图位置信息及后来生成的二元词图切词位置信息更新得到。注意 BiSegment 函数以相应的 BiOptimumSegment 函数中，二元词图生成之后，根据 NSP 算法计算相应的路径信息并输出到了预先开好的空间 nSegRoute 中，对于每个 nSegRoute[i]即为一种切分方法，BiPath2UniPath 对于输入的这个一维数组 nSegRoute[i]做处理。m_npWordPosMapTable 中的信息包含词图的行值及列值，而 nSegRoute[i]则只是列值,所以根据 m_npWordPosMapTable 中当前结点的行值和列值取余后更新结点的列值，即下一个切分的位置。这里只需要注意收尾的问题。

原始的程序没有做清理数据的操作，因此在新的版本中加入了一个 ClearSegmentWord 函数，主要是用于将 m_pWordSeg 中的所有字符串 sWord 重置以便诸如 GetResultCount、GetLastWord 一类使用 sWord 作为结束判断依据的函数不会出错。

CSegment 类的总结：

这个类的内容稍微有些多，除了一些辅助功能性的成员函数，个人认为最主要的是四个，即 BiSegment、BiOptimumSegment、BiGraphGenerate 和 GenerateWord。前两个函数用于做二元切分，过程基本上是一致的，都是先分配存储路径的一块空间，根据句子、核心词典及二元词典生成二元词图，然后将生成的词图送去做 NSP 算法求得粗分结果存储在预分配的空间中，然后对粗分的二元结果进行二元路径到一元路径的转换并切成词输出，最后释放分配的空间，只不过 BiSegment 用的是 m_graphSeg.m_segGraph 做词图，BiOptimumSegment 用的是 m_graphOptimum 并且最后抛弃了原来的 m_graphSeg.m_segGraph。这两个函数完全可以合并大部分。BiGraphGenerate 的主要技巧在于用一个一维数组来存储切分结果的二维信息(这是由于二维数组的行数是固定的)，另外就是根据二元词典估算二元结果的概率了。GenerateWord 函数根据切分结果再做一些时间词数词的组合，代码写得有些凌乱，但主要含义还是出来了，希望可以对其中的一些规则部分提出来做一定的整合以方便阅读。

至此，Segment 目录下的所有文件全部分析完了，这也是比较核心的一部分代码，长出一口气！在粗分结束之后，要进行的就是未登录词(包括命名实体)部分的工作了，这一部分的代码主要是在 Unknown 目录下的 UnknowWord.h 和 UnknowWord.cpp 文件中。

未登录词识别(CUnknowWord 类)

ICTCLAS 的未登陆词识别是基于角色标注的。具体的算法大家可以参照张华平等人 03 年的文章<Chinese Named Entity Recognition Using Role Model>。算法的基本思想就是，首先定义一些与命名实体类型相关的角色，针对中国人名可能就有“姓“、“两字名的第一个字”、“两字名的第二个字”、“名字的前缀”、“名字的后缀”等等，然后通过一个已标注了角色的训练语料库进行学习，得到一些显著的角色串模板，例如“姓“+“两字名的第一个字”+“两字名的第二个字”是一个人名等，同时得到每个字为某种角色的概率。通过上述两组信息来进行未登陆词的识别。

CUnknowWord类的成员变量有四个成员变量,其中前三个是私有成员变量。m_roleTag是一个CSpan类的标注器，用于标注角色；m_nPos为此类的一个词性标签，即该类是一个通用的未登陆词的识别类，如果是用于识别人名m_nPos就是相应的人名对应的词性标签，如果用于识别地名，就是地名对应的词性标签等。另外m_sUnknowFlags是一个标记数组，记录着m_nPos对应的类相应的类标签，例如如果m_nPos为人名，则m_sUnknowFlags的值就是“未##人”，诸如此类。公有成员变量m_Dict是一个词典类对象，存储的为该类的词典。

Configure函数就是一个读配置文件的函数，要注意的就是传入的第二个参数type确定了该对象要识别的类别，即用于确认m_nPOS和m_sUnknowFlags的值,在这个版本中未登陆词只处理了人名(中国人名和外国人名)以及地名。

IsGivenName是专门为人名类写的，用来判断传入的字符串是否为人名。传入的第一个被当作姓，

后面的一个字被当作名。通过内置的未登陆词词典和角色标注结果的概率得到GivenName的概率和SingleName的概率(均取log)，如果前者较小(即概率较大)则返回TRUE。

Recognition函数是这个类的重头戏，用于识别是否为给定的未登陆词类型。首先使用自带的角色标注器将切分好的词串进行角色标注，然后根据角色标注的结果，把原始切分中的一些零碎的字串连起来并更新优化词图graphOptimum。

CUnknown类总结：
这个类是用来做未登陆词识别的，但是之前的工作已经都做好了，即从语料库中训练得到角色概率，得到了最显著类型未登陆词的模板。类型差别的工作也是由自带的角色标注器相应函数完成的。

还剩下三个文件，CSpan、CContextStat 和 CResult，加油!

//!这部分目前还没有概念，先略过，后补！

ContextStat.h 和 ContextStat.cpp 是上下文无关类 CContextStat 的相关声明和实现。该类的相关操作只在 Cspan 类中调用。

该类中使用一个结构体 tagContext，具体如下：

```
struct tagContext{
    int nKey;//The key word
    int **aContextArray;//The context array
    int *aTagFreq;//The total number a tag appears
    int nTotalFreq;//The total number of all the tags
    struct tagContext *next;//The chain pointer to next Context
};
```

这个结构体的具体含义待补。

CcontextStat 类的成员

2006-9-7 的学习笔记

CSpan 是标注器，词性标注和角色标注都是使用这个类来完成的，使用的是相同结构的词典，另标注算法是隐马模型(HMM)完成的。

类中使用的一些宏变量如下：

```
#define MAX_WORDS_PER_SENTENCE 120          //!用于记录每个句子中最多词的个数
#define MAX_UNKNOWN_PER_SENTENCE 200        //!用于定义每个句子中的最多未登陆词
#define MAX_POS_PER_WORD 20                 //!用于定义每个词的最多词性
#define LITTLE_FREQUENCY 6                  //!频率？
enum TAG_TYPE{
        TT_NORMAL,
        TT_PERSON,
        TT_PLACE,
        TT_TRANS_PERSON
};
//!标注的类型，如果是词性标注用的是第一个，否则是未登陆词，目前只处理这些类型
```

CSpan类的公有成员变量里面，用一个未登陆词的计数m_nUnknownIndex和二维数组

m_nUnknownWords来配合记录未登陆词的个数及每个未登陆词起始位置，数组m_dWordsPossibility则配合记录每个未登陆词的概率值(即dValue)。为什么感觉m_nUnknownIndex的名字有些名不符实呢？似乎应该是m_nUnknownCount才对，呵呵。

另外一个公有成员m_context是CContextStat类的，用于处理上下文文法吧，呵呵。待定//!

私有成员变量比较多，依次说明。m_tagType表示标记的类型，如果是TT_NORMAL就是词性标记，否则则是各个不同命名实体的角色标注；m_nStartPos记录起始位置；而数组m_nBestTag记录每个词的最佳标注结果；一起配合使用的几个数组m_sWords记录切出来的词，m_nTags记录每个词的所有可能词性(或者标注角色类型)，m_nWordPosition记录词的位置，而m_nBestPrev是用于Viterbi解码算法中记录最佳路径的数组；m_dFrequency记录每个词的每个词性(标注类型)的概率。最后，m_nCurLenght记录当前长度，其实也可以看作是一个上述几个数组的有效元素的个数。

下面是重头戏，即各个成员函数的分析。

构造函数就是将上述数组、变量初始化，先看看这段代码再说要注意的事项：

```
if(m_tagType!=TT_NORMAL)
        m_nTags[0][0]=100;//Begin tag
   else
        m_nTags[0][0]=0;//Begin tag
   m_nTags[0][1]=-1;
   m_dFrequency[0][0]=0;
   m_nCurLength=1;
   m_nUnknownIndex=0;
   m_nStartPos=0;
   m_nWordPosition[1]=0;
   m_sWords[0][0]=0;
   m_tagType=TT_NORMAL;//Default tagging type
```

第一就是构造函数的第一句对m_tgType的判断，由于是在构造函数中m_tagType其实未被赋值，那么m_tagType就是编译器的默认值(这里就是枚举类型的第一个值TT_NORMAL)，而且最后还对m_tagType又赋了个初值TT_NORMAL,从最终结果上看似乎不错，但感觉赋值很无厘头。

另外就是m_nCurLength的初值不是0而是1，m_nWordPosition初值不是m_nWordPosition[0]而是m_nWordPosition [1]。具体的含义看看代码再解释//!

POSTagging函数应该是进行标注的主函数部分，在这之前要做的首先是初始化，即读入相应的未登陆词词典和上下文规则，这部分应该是在系统的初始化过程中完成的，对应的函数为LoadContext。传入的参数用做m_context初始化时使用。

下面正式说POSTagging函数。传入的第一个参数是NSP算法生成的粗分结果，后面的两个词典分别是核心词典和未登陆词词典。读分词结果，从当前读到的词开始，猜测该词的最好词性(或角色标注结果)，然后根据当前标注器的类型(标词性或者相应的未登陆词)进行相应的处理。如果是词性，则根据猜测的结果对词性进行相应的更新；如果是命名实体，则调用相应的函数进一步处理，这里有一点要注意的是TT_PLACE(地名)和TT_TRANS_PERSON(音译人名)用的都是PlaceRecognize函数，不知道是写错了还是怎么着。每猜完一次都要对变量进行重置Reset()。大概过程就是这样，这里面比较核心的部分是猜测词性以及命名实体识别对应的几个函数。

GetFrom函数用于从传入的词链中根据起始位置nIndex读入一个词，读入的信息存在相应的成员变量中。主循环中有两个索引变量,I从1开始取值，控制的是m_sWords、m_nWordPosition、m_nTags、m_nBestPrev、m_dFrequency等成员变量的索引值，最后的值记录在m_nCurLength中用于表示标注后词的个数。而nWordsIndex则从传入的起始位置nIndex开始取值，控制的是传入的原始词串数据结构pWordItems的索引。I的取值不会超过MAX_WORDS_PER_SENTENCE(120)而nWordsIndex的取值不会超过MAX_WORDS(650)。主循环对I的最大取值进行了判断但是没有判断nWordsIndex的最大取值，只是根据对应位置的字符串是否为空来决定的，如果前面的计算中没有清空工作现场或者传入参数非法则有可能非法访问造成系统崩溃。
在标注的主循环中，如果进行词性标记或者未登陆词典中没有出现当前处理的这个词pWordItems[nWordsIndex].sWord，那么简单的将词拷入m_sWords[i]中，并记录下位置信息在m_nWordPosition[I+1]中，这也是为什么构造函数初始化时m_nWordPosition是从m_nWordPosition [1]开始赋值的，因为I的取值从1开始。这里同样存在一个隐患，因为只判断i<MAX_WORDS_PER_SENTENCE，当I的取值为MAX_WORDS_PER_SENTENCE-1时，对m_nWordPosition[I+1]的访问会造成越界。对于角色标注且在词典中找到词的情况，会做一个切分，把第一个字和该串的其余部分分开考虑。

记录下当前词的起始位置在m_nStartPos中，然后根据标注的不同类型进行处理。对于角色标注，如果是外国人名标注，会对半角符号“．”和“-”分别替换为相应的全角符号“．”“－”进行处理。从未登陆词词典中读入该词的所有词性及相应概率值，并记录在相应的m_nTags和m_dFrequency中。如果当前词为开始标记或者结束标记且从未登陆词词典中只出现了一次要更新m_nTags和m_dFrequency的相应值，注意对于开始标记更新的是[i][j-1]而结束标记更新的是[i][j]。其他的需要标记角色的词，从核心词典中读入该词的词性及相应频率，如果词典中已经有相应的词及词性，则将角色标记结果置为0，概率结果也重置。注意这里更新了j的值，以便后面判断是否已有标注结果。相对而言词性标注的工作则比较简单，如果当前词条的nHandle值大于0，即只有一个唯一确定的词性那么记录下来就OK了；否则，属于有多个词性的情况，记录词性的时候取个负值即可。然后从核心词典中读入该词的多个词性，依次记录入m_nTags和m_dFrequency的后面的位置。

上面两种类型的标记都做完以后，如果j的取值仍然为0，即什么标记也没有得到，那么调用GuessPOS猜一个词性。至此会有一个标记了，不要忘记给m_nTags[i][j]添上结束标记－1。如果此时j的取值为1即只有一个标注结果且又不是开始标注（结束标记就可以退出了）那么表示无歧

义了可以跳出循环，否则继续。最后要做一些清理工作，包括判断是否已到词串的结尾，下面这段代码的含义没有看太明白，先记录下后补：

```
if(m_nTags[i-1][1]!=-1)//||m_sWords[i][0]==0

  {//Set end for words like "张/华/平"

      if(m_tagType!=TT_NORMAL)

            m_nTags[i][0]=101;

      else

            m_nTags[i][0]=1;



      m_dFrequency[i][0]=0;

     m_sWords[i][0]=0;//Set virtual ending

      m_nTags[i++][1]=-1;

  }
```

GuessPos函数只对未登陆词进行角色的猜测，如果是词性标记则直接忽略掉了。对于中国人名、音译人名和地名，分别有更新m_nTags和m_dFrequency的计算方法，但是全部使用的是数值不清楚确切含义。

Disamb()函数用于消歧，即使用Viterbi算法选取全局最优的标注结果。对每个词的每个标注结果m_nTags[i][j]，考察当前词的前面一个词的每个词性标注结果m_nTags[i-1][k]，得到最小代价的路径之后将路径和频率值分别记录在m_nBestPrev[i][j]和m_dFrequency[i][j]之中。

Reset函数顾名思义用来重置内部变量，参数bContinue是用来控制重置的程度，如果bContinue为真则只是将上次的结果拿来做为初始状态，否则清空重新开始。

GetBestPOS()函数就很简单的，先进行消歧（调用Disamb函数），根据计算得到的结果从后向前回溯，将最好结果记录在m_nBestTag中。

PersonRecognize函数是专门用来做中国人名识别的，几种不同模式的定义在代码中都有说明，例如BBCD是表示姓+姓+名1+名2，这种模式的参数值是0.003606，模式的长度为4。先将计算得到的最佳角色标注结果的数值转换为字符。然后对于切好的词串进行于全排列的方式进行组合来匹配所有的模式，如果匹配到则更新相应的m_nUnknownWords和m_dWordsPossibility的值，并增加m_nUnknownIndex的计数。

PlaceRecognize的过程类似，但代码写得没有人名部分那么清楚。只有在m_nBestTag[i]的值为1或者2的时候才会引发识别过程，这两个数字代表的含义目前未知。程序同时引入了惩罚因子来对长度进行惩罚。最终更新m_nUnknownWords和m_dWordsPossibility的值。这个函数中使用多处while循环，并没有索引的值进行判断，可能会出现溢出的情况吧。

ComputePossibility函数就是计算从传入的起始位置nStartPos长度为nLength的词串的相应概论，用的是词的最佳标注结果来进行计算，没有更多可说的内容。

CSpan类的总结：

至此，CSpan的内容基本上写完了，因为最初写的时候也没有完整的概念体系所以比较凌乱，现在再总结一下。这个类是一个“ 通用的 ”进行标注类，使用的算法是隐马模型，解码用的是Viterbi算法。使用此类可进行词性的标注，也可以进行角色的标注，在部分代码中需要一定的判断。类的对外接口有两种功能类型，一种是初始化的时候读入相应的词典（未登陆词词典）及相应的上下文无关信息，另外一种就是对传入的词串进行标注，可能是词性也可能是命名实体，这部分的框架在POSTagging函数即可看到。读入一个词串和相应的词性概率等信息，用Viterbi算法计算得到最好的标注结果后，如果是词性标记则修改相应的词性标注结果，否则调用相应的命名实体单元来进行进一步的识别工作。直到处理完所有的结果。

2006-9-8的学习笔记

现在回到不能回避的CContextStat类上。这个类的具体作用暂时没看清楚，从操作的步骤来一点一点写吧，呵呵。

构造函数和析构函数就是初使化一些成员变量及释放相应的空间，并不真正的读入内容。Load函数用于读入上下文相关内容。相应文件的格式如下，第一个int型大小的数据为表的长度，读入到m_nTableLen中，然后为m_pSymbolTable分配大小为m_nTableLen的int型空间，并读入相应m_nTableLen个int型数据到m_pSymbolTable中。然后读上下文无关m_pContext链表中的内容。第一个int型数据为链表结点中的关键字pCur->nKey，第二个int型数据为结点的频率值pCur->nTotalFreq，接下来的m_nTableLen个int型数据为相应的每个标记类型的频率值，分别读入到pCur->aTagFreq中。接下来的m_nTableLen×m_nTableLen个int型数据是aContextArray的值，其中aContextArray是一个大小为m_nTableLen的int型指针数组，每个m_nTableLen[i]指向一个大小为m_nTableLen的int型数组。上下文无关内容m_pContext每个结点都通过next指针串起来直到文件结束。Save函数就是load的逆过程，依上文所述之格式写入文件即可，这里不再详述，唯一注意的就是新添了一个以文本格式记录了文件的内容以供查看和参考。

对于该类的修改性函数还有Add，即添加一个新的结点，传入的参数分别是该结点的关键字nKey，前一个符号的id，当前要插入的符号的id以及相应的频率值。先查找是否已存在相同nKey的结点，如果没有找到则新建一个插入到相应的位置，注意新建这个结点的所有频率值都被置为0，只有关键字为nKey。无论是否找到nKey结点，根据传入的符号nPrevSymbol和nCurSymbol分别在符号表中查找相应的索引值。如果没有找到则修改其符号（为nPrevSymbol-nPrevSymbol%256）重新查找一下。如果都找到了那么更新相应的频率值及总频率值。

SetTableLen函数顾名思义就是设置符号表的大小，即修改m_nTableLen的值，同时会发生变化的还有m_pSymbolTable所指空间等，但是这个函数中没有做原有空间的释放工作直接就赋空了不太合理可能造成内存泄漏，不过这个函数在目前的代码中没有用到，暂时未构造潜在的危害。

SetSymbol也有类似SetTableLen的问题，直接进行了内存拷贝memcpy，而没有考空m_pSymbolTable是否有足够的空间，传入的参数是否合法。

GetItem用于查找给定关键字nKey的上下文无关结点，返回其指针。如果找到了则返回true并将其指针记录在pItemRet中，如果没有找到则返回其前一个结点的指针并返回false。注意如果链表为空则前一个结点的指针为NULL。

GetFrequency就是查找给定关键字给定符号的频率值，通过查找m_pContext所带的链表找到相应的关键字所在的结点pFound，通过查找m_pSymbolTable找到相应的符号的索引nIndex，然后返回pFound->aTagFreq[nIndex]。如果没有找到则返回0。

GetContextPossibility函数则是获得上下文无关文法的相应概率值。基本的策略是从符号表中读取当前符号和前一个符号的索引值，从上下文无关文法链表中读入相应关键词所在的结点位置，如果没有找到相应的结点，或者没有找到上述两个符号，又或者结点对应的前一个符号的词性标注概率值为0，又或者结点对应的这两个符号的上下文无关概率值为0，则返回一个很小的概率值避免数据稀疏。否则，返回按经验公式计算得到的相应概率值，计算公式如下：

0.9*nPrevCurConFreq/nPrevFreq+0.1*nPrevFreq/pCur->nTotalFreq;

其中

nPrevCurConFreq=pCur->aContextArray[nPrevIndex][nCurIndex];

nPrevFreq=pCur->aTagFreq[nPrevIndex];

注意上述公式中计算时要将整型数据强制转换为double型数据，否则除法的结果会有问题。另外0.9和0.1都是经验值。

CContextStat类的总结。

至此，CContextStat类的函数就写完了。总的来说，这个类提供关于上下文无关文法的概率计算操作。

2006-9-11的学习笔记

最后一个核心文件CResult类。这个类是最外层的包装，私有成员变量包括一个切分器（m_Seg）一个词性标记器m_POSTagger、两个词典（核心词典m_dictCore和二元词典m_dictBigram）、三个未登陆词识别器（m_uPerson,m_uTransPerson,m_uPlace）。公有成员变量中的m_pResult记录切词及标注的结果，m_dResultPossibility用于记录生成多个结果是相应的概率值。可供用户进行设置的几个参数分别是m_dSmoothingPara用来做概率计算时的平滑参数，m_nResultCount记录生成多少个切分标注结果，这个数值被MAX_SEGMENT_NUM限制，即不超过10个，m_nOperateType记录生成结果的标注类型，如果为0则只切分不标注，如果为1表示上位标注，如果为2则为下位标注（更详细的标注类型），另外m_nOutputFormat控制输出结果的表示方法（以及词性标注集的转换），包括北大标准，973标准及XML表示方式。

构造函数用于初始化资源，这个过程会比较慢，包括给m_pResult分配存储分词及标注结果的空间，最多生成MAX_SEGMENT_NUM（10）个结果，每个结果最多包括MAX_WORDS（650）个词。然后是词典和标注器的初始化，m_dictCore和m_dictBigram分别读入词典，m_POSTagger读入词性标注的内容，并设置其标注类型为TT_NORMAL，即词性标记。三个未登陆词标注器分别读入相关数据并设置标注类型，默认的标注类型为下位词性，默认的输出标准为北大标准，默认的平滑参数

为0.1。感觉还落下了m_nResultCount和m_dResultPossibility的初始化，前者应该为1后者应该全为0。

析构函数释放所有的空间，这里主要是m_pResult的空间，其他词典及相应的标注器都会在析构函数中自行释放空间。该类的初始化和析构过程都非常慢，应该考虑是否有优化的办法。

提供了三个级别的切分及标注接口。FileProcessing读文件并将结果输出到指定的文件名中。ParagraphProcessing读段落并将结果输出到给定的buffer中。Processing则是将句子的切分标准结果输出到内置的m_pResult中，可以生成多个结果，但是还需要进一步使用Output函数输出到相应的buffer中。前两个函数最终都要调用Processing来进行真正的处理过程。

FileProcessing中使用fgets读入最多4*1024-1个字符，有可能一段文字非常长有可能发生截断成半个字符或者不完整的情况都没有处理。将读入的段落送入ParagraphProcessing函数中进行处理生成的结果写入指定的文件，如果是XML文件格式输出则还会控制前后的XML标签<>的生成。

ParagraphProcessing函数处理一段文字。它需要将一段文字根据标点符号切成若干子句。子句首尾会根据需要添加上句子开始结束标记后再送入Processing进行处理。使用了一些技巧来减少重复操作，加快速度。依次读入字符（可能是全角可能是半角，这个部分处理得不好，应该写一个函数来操作，否则代码变得很长）。如果不是全角的句子（子句）结束符、不是半角的句子（子句）结束符也不是是段落结束符，则追加到待处理的句子串中再取下一个字符，否则进一步处理。对于已经生成的待处理句子串，如果非空也不是只有开始标记送入Processing进行处理，只生成一个结果，并将生成的切分结果输出并追加到结果串中。注意如果待处理句子串的最后一个字符不是子句结束符则要追加一个句子结束符。处理结束之后，要重置待处理的句子，如果最后一个字符是换行符或者是句子结束符则重置新句子为开始标记，否则简单的把原来的最后一个字符拷入新句子。待用来读字符的索引超出段落长度之后，不要忘记把未处理完的句子送入Processing进行切分。

Processing的过程就相对简单了。应该首先清空m_Seg和m_pResult中相应的sWord的值，原始函数中没有可能造成一些错误在新版本中已改正。将传入的句子使用切分器m_Seg进行二元切分BiSegment，生成的结果会存储在m_Seg.m_pWordSeg中，结果个数存在m_Seg.m_nSegmentCount中。然后对于每个生成的结果（一般为1，最多10个），分别使用三个未登陆词识别器进行识别，结果存在m_Seg.m_graphOptimum中。然后调用m_Seg的二元优化切词(m_Seg.BiOptimumSegment)，即根据m_Seg.m_graphOptimum的结果重新进行切词。对于生成的每个结果（存在m_Seg.m_pWordSeg中），调用词性标注器m_POSTagger进行词性标注。最后将生成的结果进行排序，最终的结果会按概率高低存储在m_pResult中。

Sort函数很简单，就是用冒泡法将每个结果的概率从大到小排序然后相应的按概率将结果存到m_pResult中。这里会调用两个函数，ComputePossibility用于计算传入的结果的概率值，Adjust用于调用输出的结果。

ComputePossibility的计算公式是，对于传入的一种切分结果，其概率值等于每一项的概率值dValue，再加上这个词和后面那个词的上下文无关概率值加1以后取log的值，再减去这个词本身词性标记的概率值加1以后取log的值。

Adjust函数则是在最后引入了一些规则来处理统计方法无法涵盖的情况。
规则一：对于中国人名，会将其切成姓加名的形式（还有可能是双姓）；
规则二：对于ABB这种切散了的格式（一段段、一片片）会合并；
规则三：对于AA这种切散了的格式会合并，并且根据原始的词性重新设置新的词性，默认为a，如果原来都是名词则修改为名词，如果原来都是动词则修改为动词。另外，会进一步判断是否为AAB这种格式，如果符合则进一步合并。例如洗/洗/脸、蒙蒙亮。
规则四：对于没有切散的AAB格式，一般会切成A/AB的类型，例如洗/洗澡。这种情况下合并并重新设置词性。默认为'a'，如果原来为动词则仍然为动词。
规则五：如果原来词性为uj,ud,uv,uz,ul,ug等等，则统一输出为u。
规则六：对于AABB型的切散成A、AB、B的格式，则合并。词性取AB的词性；
规则七：对于后缀类型的词（28275），如果后面一个词在地名词典中出现则合并，或者后面一个词的长度为1（即一个汉字）且是“队语文字杯裔”中的任何一个则合并并相应的设置新的词性。
规则八：如果词性为30208或者28160且后面一个词是单字“员”则合并，例如运动员；
规则九：如果词性为28280（未登陆词）且后面的一个词是未登陆词、或者是全角或半角的符号点或者后面的词是数字串则合并起来，例如/www/nx ./w sina/nx
规则十：默认规则，即如果上述规则都不满足则简单的将原来的切词结果拷入。

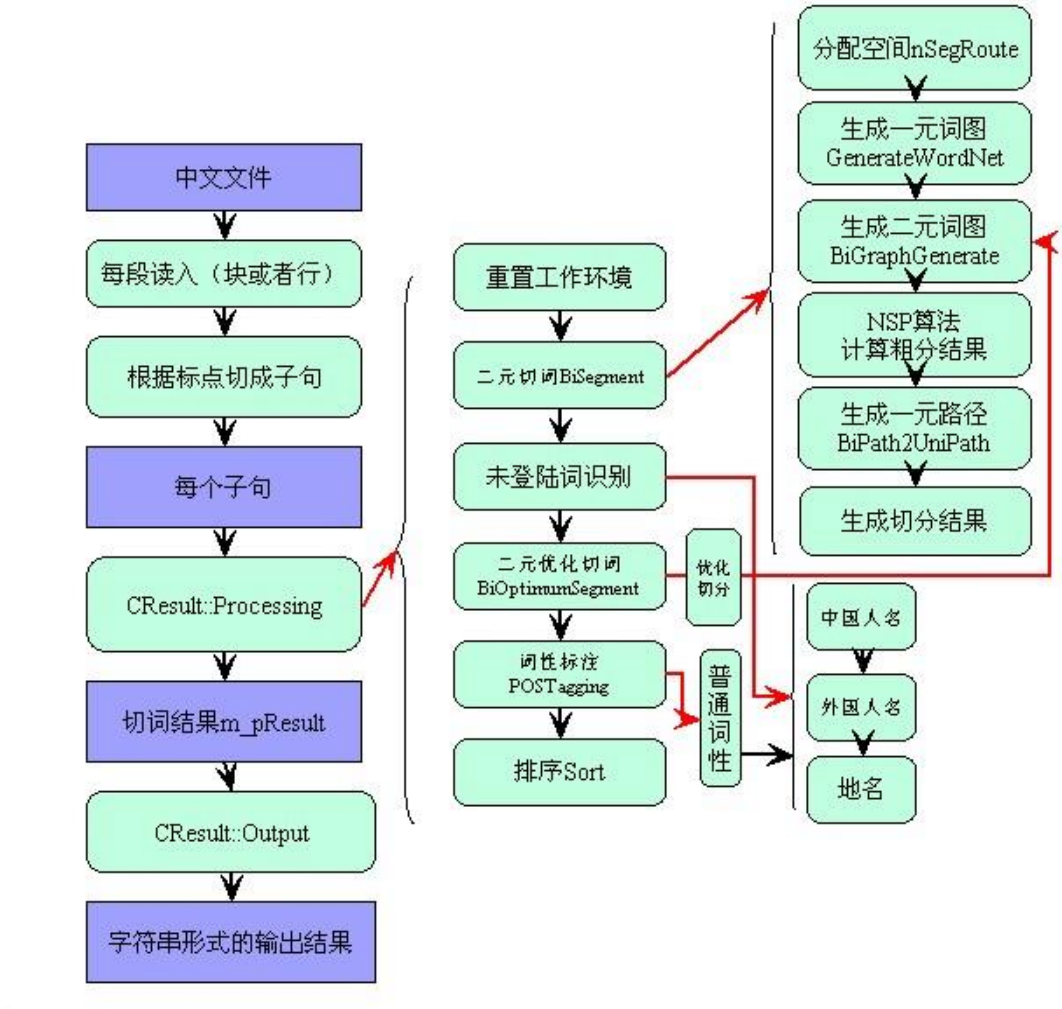
使用上述规则进行处理后得到最终的切词结果。

Output函数将PWORD_RESULT类型数据中的切词结构转成字符串表示形式。依次读切词数据链，对于每个词条，将其词性标记转换为字符串表示方式。只在这个函数中涉及输出的格式问题和词性标注的程度问题。PKU2973POS函数就用来将北大标准的词性标记集转换为973标准，即做一个一一对应。

ChineseNameSplit是一个额外的函数用于将中国人名的姓和名两部分切分开。就是查人名典来确定姓与名的分割点。

另外，在新添的版本中增加了一个重置函数ResetWord，用于清理每次计算完成之后的m_pResult的相应sWord值。

ICTCLAS1.0 版的源码已经差不多看完了，除了上下文无关文法那块有些具体的含义没有明确以外其他的代码基本理清。下面应该是给出一些总体性的框架来进一步帮助整理思路的时候了，呵呵。



上面是一个简单的数据流程图，淡蓝色的部分为数据，淡绿色的部分为函数或相应函数的功能概括。很多细节无法在这张图中展示，但也可以基本看出数据的走向和整个系统的框架。

程序的实用化，要考虑如下几个方面的问题：

1. 如何更好的使用用户词典；

   一般的策略有两种，预处理和后处理，无论是哪种，用户词典都很难给出我们所需要的相应概率。前处理的方法把用户词典看作普通词典并给出一个默认的概率值，统一进行切分；后处理的方法一般就只能是最大匹配了。两种方法的速度都不会很快，而且不可避免引入错误。尤其是组合型歧义。

2. 如何避免系统崩溃级的错误；

   目前的版本因为采用的是预定义空间大小的方法，对于一些特别的例子非常容易出现数组越界的情况。能够减少数组越界的方法要么是增加预定义空间的大小，但是很可能申请不到那么多的内存；另外一种方法是增强越界的检测，但是模块之间的耦合变大，而且越界部分的

处理结果会丢失；再就是使用动态分配内存的方法，但是涉及到多次的构造析构操作速度会变慢。

3. 如何加快分词的速度。

   仅从代码级而言就有很多地方可以改进系统的性能，例如使用一些更好的数据结构，对于一些常量的数组不要每次调用函数都重新分配而是使用 static 变量等。但是这些改进不足以使得系统的性能大幅度提高；速度减慢的主要原因一个是查词典速度很慢，虽然已经用了二分查找达到 Logn 的级别，但是还有提高的余地，例如使用双 Trie 树的数据结构；另外一个速度慢的原因是大量的内存申请释放操作。提高速度也可以考虑放弃一部分的正确性使得部分模块简化或者干脆不用。

对于程序的代码风格，不得不说存在两大问题：

1. 很多空间都是采用预定义大小的方式，但对于空间大小的检测并不完全，使得数组越界的潜在危险始终存在；

2. 函数传递时不用 const 类型，使得代码的可读性下降，尤其是词典类的使用和一些字符串参数的情况，建议还是应该使用 const 类型来帮助阅读。