LTP 使用文档 v1.3

李正华
lzh@ir.hit.edu.cn
2007 年 10 月 6 日整理

目录

LTP总体介绍

LTP底层以XML表示文本，以DOM处理文本

LTP中包含 9 个模块（不考虑分段）

模块之间的依赖关系

LTP自动解决模块之间的依赖关系，用户无需考虑

分句模块必须单独调用（和以前不同）

LTP共享包文件结构

LTP运行平台

LTP接口说明

1. **DOM操作**
2. **以句子为单位的处理模块**
3. **以篇章为单位的处理模块**
4. **Count段落数、句子数、词语数**
5. **抽取句子级的处理结果**
   1) **每次抽取一个句子中所有词语的信息**
   2) **每次抽取一个词语的信息**
   3) **抽取Semantic Role**
6. **抽取篇章级处理模块的结果**
7. **main2 函数、抽取句子内容等**

LTP调用示例

1. **处理一个文本，抽取分词、词性信息**
2. **处理包含 200 万个句子的文本，抽取分词、词性标注、句法信息**
3. **从之前保存的xml中抽取分词词性信息**

FAQ

什么时候LTP会出错？

如何获得LTP的错误信息？

LTP分段方法？

LTP分句方法？

将LTP的处理结果存储到xml文件有什么用处？

如果不想使用LTP中的分句模块怎么办?

如果希望使用自己的分句模块怎么办？

如何扩展分词模块的词表？

如何配置LTP？

LTP 总体介绍

LTP 底层以 XML 表示文本，以 DOM 处理文本

LTP is a language platform based on XML presentation. All operations are done in DOM. For now, LTP is oriented to single document.

A document is presented as a DOM in computer memory, which can be saved as an XML file. The XML format defined in LTP is as following:

```
<?xml version="1.0" encoding="gb2312" ?>
<?xml-stylesheet type="text/xsl" href="nlp_style.xsl" ?>
<xml4nlp>
    <note sent="y" word="y" pos="y" ne="y" parser="y" wsd="y" srl="y" class="y" sum="y" cr="y" />
    <doc>
        <para id="0">
            <sent id="0" cont="伊拉克军方官员 20 日宣布，...">
                <word id="0" cont="伊拉克" pos="ns" ne="O" parent="1" relate="ATT" wsd="Di02" wsdexp="国家_行政区划" />
                ...
                <word id="4" cont="宣布" pos="v" ne="O" parent="-1" relate="HED" wsd="Hc11" wsdexp="召集_宣布_下令">
                    <arg id="0" type="Arg0" beg="0" end="2" />
                    <arg id="1" type="ArgM-TMP" beg="3" end="3" />
                    <arg id="2" type="Arg1" beg="6" end="28" />
                </word>
                ...
            <sent id="1" cont="20 日上午，搜救人员在一座变电站附近找到这两名军人的尸体。">
            ...
        </para>
        ...
    </doc>
    <class>军事</class>
    <coref>
        <cr id="0">
            <mention id="0" beg="496" end="496" />
            <mention id="1" beg="516" end="516" />
        </cr>
        ...
    </coref>
    <sum>伊拉克军方官员 20 日宣布，上周五在巴格...</sum>
</xml4nlp>
```

As we can see from above:A document after fully processed (Split Paragraph, Split Sentence, Word Segment, POS, NE Recognition, Word Sense Disambiguation, Parser, Semantic Role Labeling, Text Classify, Text Summary, Coreference Resolution), is organized as following:

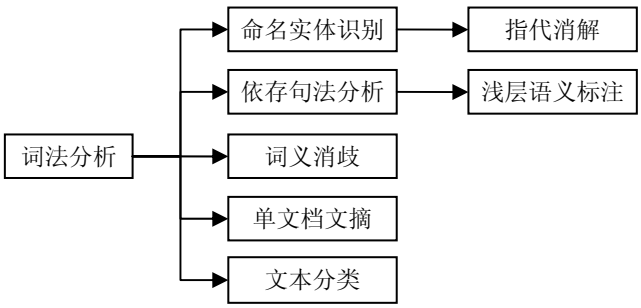
Each <doc> is composed of several <para>
Each <para> is composed of several <sent>
Each <sent> is composed of several <word>
Each <word> has several attributes to represent the POS, NE, Parser, WSD info of this word.
Each <word> has zero or several <arg>, which reprsents the SRL info of this word.
<class> presents the Text Class.
<sum> presents the Text Summarization.
<coref> presents the Coreference Resolution result, which is composed of several <cr>
Each <cr> represents an entity, which is composed of several <mention>
Each <mention> represents an entity mention.

Note:
The "beg" and "end" attributes in <mention> is the global word index in the whole document.
While, the "beg" and "end" attributes in <arg> is the local word index in current sentence.

LTP 中包含 9 个模块（不考虑分段）

- 分句，SplitSentence
- 分词及词性标注 (Word segment and POS)，IRLAS
- 命名实体识别 (Named entity recognition)，NER
- 词义消歧 (Word sense disambiguation)，WSD
- 依存句法分析 (Dependency parser)，Parser
- 语义角色标注 (Semantic role labeling)，SRL
- 单文档文摘 (Single document summarization)，SDS
- 文本分类 (Text classification)，TextClassify
- 共指消解 (Coreference resolution)，CR

模块之间的依赖关系



LTP 自动解决模块之间的依赖关系，用户无需考虑

如果您只需要句法信息，那么您只需要调用 Parser()处理即可，无需先调用 IRLAS()然后再调用 Parser()，无论您对模块间的依赖关系是否了解。当要运行某一模块的时候，LTP 内部会检测这个模块所依赖的模块是否完成。如果没有完成，则自动调用。

另外，模块调用的顺序没有严格的规定。唯一的要求是分句模块必须在其他模块的前面调用。

分句模块必须单独调用（和以前不同）

考虑到很多时候 LTP 被用于句子级处理，每一次用户将一个句子作为一篇文档输入，或者输入文本中每一句构成一行。有的时候用户不希望 LTP 进行分句处理。因此我们改变了 LTP 内部的依赖关系。词法分析不再依赖于分句模块。这意味着用户必须自己决定是否调用 LTP 的分句模块。如果用户不调用分句模块，那么 LTP 将会把一行文字作为一句话（同时也是一个段落）。相反，如果用户需要 LTP 中的分句模块进行分句的话，那么必须显式的调用 SplitSentence()。

分句模块必须在其他模块的前面调用。

LTP 共享包文件结构

+ LTP DLL Package
  + ltp_data (包含各个模块使用的数据)
  + cplusplus
    + dll
      - *.dll
      - __ltp_dll.lib (静态使用 LTP dll 时需要)
      - __ltp_dll.h (LTP dll 接口)
      - __ltp_dll_x.cpp (实现 dll 无法实现的接口，如涉及 STL 容器的接口，方便用户的使用)
      - ltp_modules_to_do.conf (当用户使用 LTP 提供的 main2 接口时，指定需要在 main2 中调用哪些处理模块)
      - ltp_all_modules.conf (LTP 配置文件，如数据资源的位置，某些模块的处理方式等)
      - nlp_style.xsl (可视化文件)
    + test
      - *.dll | __ltp_dll.lib | __ltp_dll.h | __ltp_dll_x.cpp | *.conf | nlp_style.xsl
      - test.cpp (测试程序，包含各种使用 LTP 的方法)
      - test.txt (测试时使用的文本)
      - test_log.txt (测试时存储从 DOM 中抽取的信息)
      - *.xml (测试时生成的 xml)
      - VC6.0 工程文件 (.dsw .dsp .nsb .plg)
      - VC7.1 工程文件 (.sln .vcproj .suo)
  + python
    - *.dll | *.conf | nlp_style.xsl
    - ltp_interface.py (python 接口，与 C++接口完全对应，使用方法也相同)
    - test.py (测试程序，包含一些 LTP 的使用方法，详细的使用方法还需要看 C++的例子)
    - test.txt
    - *.xml

LTP 运行平台

VC6, VC7.1 (VC .NET)

LTP 接口说明

如上所述，LTP 处理文本时，以 XML 组织文本。XML 在内存中以 DOM 树表示。

When error occurs (wrong operation such as access an inexistent word index), LTP will output error info through stderr, and return some value to transfer the error state to the caller.

All interfaces have returns. The meanings of them are:

1）For the Counting functions (function name starts with Count*), if error occurs, they return 0. But, it does not mean an error when they return 0.

2）For functions whose return type is const char *, if error occurs, they return NULL. But, it does not mean an error when they return NULL.

3）For other functions, they return 0 if successfully executed.

1. DOM 操作

- int CreateDOMFromTxt(const char *cszTxtFileName);

从纯文本文件创建 DOM。LTP 会按照回车换行，进行分段。每一行为一段。如果为空行或者只包含空白符，则扔掉，不作为一段。

此时 DOM 树的结构如下：

```
<?xml version="1.0" encoding="gb2312" ?>
<?xml-stylesheet type="text/xsl" href="nlp_style.xsl" ?>
<xml4nlp>
    <note sent="n" word="n" pos="n" ne="n" parser="n" wsd="n" srl="n" class="n" sum="n" cr="n" />
    <doc>
        <para id="0">伊拉克军方官员 20 日宣布，上周五在巴格达南部地区“失踪”的两名美军士兵被当地的反美武装俘虏并且惨遭杀害。20 日上午，搜救人员在一座变电站附近找到这两名军人的尸体。调查人员表示，有迹象表明，这两名美军在死前曾遭到“非常残酷地虐待”。据悉，这两名只有 23 岁和 25 岁的美军被俘前曾在巴格达南部的公路检查站执勤。武装分子上周五偷袭了该检查站时除将上述两人俘虏外，还将另一名美军打死。美军和伊拉克安全部队随后派出了 8000 多人开展了大规模的搜救工作，最终找到了这两名士兵的遗体。</para>
        <para id="1">据介绍，上述三名美军在事发前坐…</para>
        <para id="2">…</para>
        <para id="3">… </para>
        <para id="4">… </para>
        <para id="5">… </para>
    </doc>
</xml4nlp>
```

- int CreateDOMFromXml(const char *cszXmlFileName);

从以前使用 LTP 处理文本，然后保存下来的 xml 文件创建 DOM。通过这个接口创建 DOM 树之后，您既可以从中抽取您需要的处理结果，也可以对其做更深层次的处理。比如之前只完成了分词、词性标注，那么这次您可以进一步进行句法分析等。

- int CreateDOMFromString(const char *str);

和 CreateDOMFromTxt 完成的工作相同，区别在于文本内容变成 str 中的内容。

- int CreateDOMFromString(const string &str)

  重载上一个接口。

- int ClearDOM();

  释放 DOM 树占用的内存。这个接口您可以不理睬，因为在程序退出或者 CreateDOM 之前总会调用 ClearDOM。如果您感觉内存很紧张，您可以调用它。DOM 树占用的内存和处理的文本大小约成正比例。

- int SaveDOM(const char *cszSaveFileName);

将 DOM 保存为 xml 文件。如果您觉得保存处理结果很有帮助，那您将需要这个接口。以后便可以使用 CreateDOMFromXml，重新建立 DOM。如果不需要处理结果，那您便没有必要使用这个接口。xml 文件比较占用磁盘，一般是处理文本的几十倍甚至上百倍。

2. 以句子为单位的处理模块

- int SplitSentence(); // 严格来讲是以段落为单位的处理模块
- int IRLAS(); // Word segment and POS
- int NER(); // Named entity recognition
- int WSD(); // Word sense disambiguation
- int Parser(); // Dependency parser
- int SRL(); // Semantic role labeling

3. 以篇章为单位的处理模块

- int SDS(); // Single document summarization
- int TextClassify(); // Text classification
- int CR(); // Coreference resolution

4. Count 段落数、句子数、词语数

- int CountParagraphInDocument();

  察看整篇文章的段落数

- int CountSentenceInParagraph(int paragraphIdx);

  察看某一个段落的句子数，paragraphIdx 为段落号，从 0 开始编号。

- int CountSentenceInDocument();

  察看整篇文章的句子数

- int CountWordInSentence(int paragraphIdx, int sentenceIdx);

  察看第 paragraphIdx 个段落中的第 sentenceIdx 个句子的词语数，sentenceIdx 为句子在段落中的编号，从 0 开始编号。

- int CountWordInSentence(int globalSentIdx);

  察看整篇文章中的某一个句子的词语数，globalSentIdx 为句子在整篇文章中的编号，从 0 开始编号。

- int CountWordInDocument();

  察看整篇文章的词语数

5. 抽取句子级的处理结果

包括 word, POS, NE, Word Sense Disambiguation(WSD), Dependency Relation, Semantic Role。

有两种方式可以抽取句子级的结果：可以每次抽取一个词语的信息，也可以每次抽取一个句子中所有的词的信息，以 vector 的形式返回。推荐使用后者，因为后者更快一些。抽取 Semantic Role 信息时比较特殊，在 3)中单独说明。

1) 每次抽取一个句子中所有词语的信息

// Get words

- int GetWordsFromSentence(vector<const char *> &vecWord, int paragraphIdx, int sentenceIdx);
- int GetWordsFromSentence(vector<const char *> &vecWord, int globalSentIdx);

第一个函数抽取第 paragraphIdx 个段落中的第 sentenceIdx 个句子的词语。

第二个函数抽取整个篇章第 globalSentIdx 个句子的词语。

// Get POSs

- int GetPOSsFromSentence(vector<const char *> &vecPOS, int paragraphIdx, int sentenceIdx);
- int GetPOSsFromSentence(vector<const char *> &vecPOS, int globalSentIdx);

// Get NEs

- int GetNEsFromSentence(vector<const char *> &vecNE, int paragraphIdx, int sentenceIdx);
- int GetNEsFromSentence(vector<const char *> &vecNE, int globalSentIdx);

// Get WSDs

// 抽取同义词词林的语义代码

- int GetWSDsFromSentence(vector<const char *> &vecWSD, int paragraphIdx, int sentenceIdx);
- int GetWSDsFromSentence(vector<const char *> &vecWSD, int globalSentIdx);

// 抽取同义词词林的语义解释

- int GetWSDExplainsFromSentence(vector<const char *> &vecWSDExplain, int paragraphIdx, int sentenceIdx);
- int GetWSDExplainsFromSentence(vector<const char *> &vecWSDExplain, int globalSentIdx);

// Get Parses

- int GetParsesFromSentence(vector< pair<int, const char *> > &vecParse, int paragraphIdx, int sentenceIdx);
- int GetParsesFromSentence(vector< pair<int, const char *> > &vecParse, int globalSentIdx);

pair<int, const char *>中 int 表示父亲节点在本句子中的编号，从 0 开始编号；const char* 为关系类型。

父亲节点编号为-1 表示为句子的核心节点。

父亲节点编号为-2 表示为标点符号，没有父亲节点。

2) 每次抽取一个词语的信息

// Get Word

- const char *GetWord(int paragraphIdx, int sentenceIdx, int wordIdx);
- const char *GetWord(int globalSentIdx, int wordIdx);
- const char *GetWord(int globalWordIdx);

第一个函数返回第 paragraphIdx 个段落第 sentenceIdx 个句子第 wordIdx 个词语
第二个函数返回整篇文章第 globalSentIdx 个句子第 wordIdx 个词语
第二个函数返回整篇文章第 globalWordIdx 个词语，从 0 开始编号。

// Get POS

- const char *GetPOS(int paragraphIdx, int sentenceIdx, int wordIdx);
- const char *GetPOS(int globalSentIdx, int wordIdx);
- const char *GetPOS(int globalWordIdx);

// Get NE

- const char *GetNE(int paragraphIdx, int sentenceIdx, int wordIdx);
- const char *GetNE(int globalSentIdx, int wordIdx);
- const char *GetNE(int globalWordIdx);

// Get WSD

- int GetWSD(pair<const char *, const char *> &WSD_explain, int paragraphIdx, int sentenceIdx, int wordIdx);
- int GetWSD(pair<const char *, const char *> &WSD_explain, int globalSentIdx, int wordIdx);
- int GetWSD(pair<const char *, const char *> &WSD_explain, int globalWordIdx);

// Get Parser

- int GetParse(pair<int, const char *> &parent_relate, int paragraphIdx, int sentenceIdx, int wordIdx);
- int GetParse(pair<int, const char *> &parent_relate, int globalSentIdx, int wordIdx);
- int GetParse(pair<int, const char *> &parent_relate, int globalWordIdx);

3) 抽取 Semantic Role

Semantic Role 比较特殊，某些词语并没有语义信息。而如果一个词语有语义信息，其参数一般为多个。如下例所示。

```
<word id="3" cont="20 日" pos="nt" parent="4" relate="ADV" />
<word id="4" cont="宣布" pos="v" parent="-1" relate="HED">
    <arg id="0" type="Arg0" beg="0" end="2" />
    <arg id="1" type="ArgM-TMP" beg="3" end="3" />
    <arg id="2" type="Arg1" beg="6" end="28" />
</word>
```

抽取语义信息的时候，需要先察看这个词语是否存在语义信息，如果有语义信息，然后调用 GetPredArgToWord(…)抽取语义信息。

// 判断词语是否有语义信息的接口，如果返回为 0 则表示不存在语义信息，否则则有语义信息。

- int CountPredArgToWord(int paragraphIdx, int sentenceIdx, int wordIdx);
- int CountPredArgToWord(int globalSentIdx, int wordIdx);
- int CountPredArgToWord(int globalWordIdx);

// 抽取语义信息的接口

- int GetPredArgToWord(vector<const char *> &vecType, vector< pair<int, int> > &vecBegEnd,
  int paragraphIdx, int sentenceIdx, int wordIdx);
- int GetPredArgToWord(vector<const char *> &vecType, vector< pair<int, int> > &vecBegEnd,
  int globalSentIdx, int wordIdx);
- int GetPredArgToWord(vector<const char *> &vecType, vector< pair<int, int> > &vecBegEnd,
  int globalWordIdx);

vecType 为语义关系。vecBegEnd 为句子中和此词语存在语义关系的词语的起止编号。编号以句子为单位，从 0 开始。如上例中第 4 个词语“宣布”的第 0 个 arg，这个句子中第 0 个词到第 2 个词与这个词构成了"Arg0"关系。

6. 抽取篇章级处理模块的结果

包括 text summary, coreference resolution, text categorization

// Get summarization

- const char *GetTextSummary();

// Get text classification

- const char *GetTextClass();

// Get CR

// typedef pair<int, int> LTP_MENTION;

// typedef vector< LTP_MENTION > LTP_ENTITY;

- int GetEntities(vector<LTP_ENTITY> &vecEntity);

pair<int, int>表示一个 entity mention 包含的词语的起止编号。注意：这个编号是词语在整篇文章中的编号，从 0 开始。

7. main2 函数、抽取句子内容等

// main2 函数

- int main2(const char *inFile, const char *outFile, const char* confFile =
  "ltp_modules_to_do.conf");

main2 完成的功能是，CreateDOM(inFile)，然后调用 LTP 的某些模块处理，最后 SaveDOM(outFile)。调用哪些模块取决于 confFile 中的配置。inFile 可以是文本文件，也可以是 xml 文件，也由 confFile 来告诉 LTP。

不推荐使用这个函数。这个函数完全可以由几行其他 LTP 接口调用替代。

// 抽取句子的内容

- const char *GetSentence(int paragraphIdx, int sentenceIdx);

- const char *GetSentence(int globalSentIdx);

为了照顾某些模块的需要，LTP 的 xml 数据表示中存储了句子的内容。如下所示。

```
<para id="0">
    <sent id="0" cont="伊拉克军方官员 20 日宣布，上周五在巴格达南部地区“失踪”的两名美军士兵被
                        当地的反美武装俘虏并且惨遭杀害。">
        <word id="0" cont="伊拉克" pos="ns" parent="1" relate="ATT" />
        <word id="1" cont="军方" pos="n" parent="2" relate="ATT" />
        <word id="2" cont="官员" pos="n" parent="4" relate="SBV" />
```

// 抽取段落的内容

- const char *GetParagraph(int paragraphIdx);

注意，只有在没有分句之前才可以抽取段落的内容，否则返回 NULL。分句之后，段落内容不再存储。这主要是从存储空间的角度考虑。

LTP 调用示例

1. 处理一个文本，抽取分词、词性信息

假设文本名为 test.txt。

```
#include "__ltp_dll.h"
#pragma comment(lib, "__ltp_dll.lib")
using namespace HIT_IR_LTP;     // Important!
int main() {
        CreateDOMFromTxt(“test.txt”);
        SplitSentence();  // LTP splits sentence
        IRLAS();            // LTP does word segmentation and POS
        // Extract word and POS sentence by sentence
        int sentNum = CountSentenceInDocument()
        int i = 0;
        for (; i<sentNum; ++i) {
                vector<const char *> vecWord;
                vector<const char *> vecPOS;
                GetWordsFromSentence(vecWord,    i);
                GetPOSsFromSentence(vecPOS,    i);
                // Use the word and POS of this sentence
                …
        }
        return 0;
}
```

2. 处理包含 200 万个句子的文本，抽取分词、词性标注、句法信息

假设文本名为 test_200_in.txt，每一行为一个句子。不需要 LTP 进行分句。将抽取结果保存到 test_200_out.txt。

Input：“其次，在近年间，本港一直受通胀所困扰。”

Output：

```
其次，   在   近   年间，   本港 一直 受   通胀 所   困扰 。        // 分词
c    wp  p    a    n    wp   r    d    v    j    u    v    wp      // 词性
8    -2  8    4    2    -2   8    8    -1   11   11   8    -2      // 父亲节点编号
CNJ  PUN ADV  ATT  POB  PUN  SBV  ADV  HED  SBV  LAD  VOB  PUN     // 依存关系
                                                                  // 空行
```

```
int main() {
    ifstream inf("test_200_in.txt");
    ofstream outf("test_200_out.txt");
    string strText;
    while (getline(inf, strText))  // 逐行处理
    {
        CreateDOMFromString(strText);
        // SplitSentence(); 不调用 LTP 的分句模块！
        IRLAS();
        Parser();
        vector<const char *> vecWord, vecPOS;
        GetWordsFromSentence(vecWord, 0);   // 全篇只有一句话
        GetPOSsFromSentence(vecPOS, 0);
        copy(vecWord.begin(), vecWord.end(), ostream_iterator<const char *>(outf, "\t")); outf << endl;
        copy(vecPOS.begin(), vecPOS.end(), ostream_iterator<const char *>(outf, "\t")); outf << endl;
        vector< pair<int, const char *> > vecParse;
        vector<int> vecDep;
        vector<string> vecRel;
        GetParsesFromSentence(vecParse, 0);   // 抽取句法信息
        split_pair_vector(vecParse, vecDep, vecRel);
        copy(vecDep.begin(), vecDep.end(), ostream_iterator<int>(outf, "\t")); outf << endl;
        copy(vecRel.begin(), vecRel.end(), ostream_iterator<string>(outf, "\t")); outf << endl << endl;
    }
    inf.close();
    outf.close();
    return 0;
}
```

3. 从之前保存的 xml 中抽取分词词性信息

假设 xml 文件名为”test.xml”。

```
int main() {
	CreateDOMFromXml(“test.xml”);
	SplitSentence();  // LTP splits sentence
	IRLAS();		// LTP does word segmentation and POS
	int sentNum = CountSentenceInDocument()
	for (int i = 0; i<sentNum; ++i) { // Extract word and POS sentence by sentence
		vector<const char *> vecWord, vecPOS;
		GetWordsFromSentence(vecWord,    i);
		GetPOSsFromSentence(vecPOS,    i);
		// Use the word and POS of this sentence
	}
	return 0;
}
```

FAQ

什么时候 LTP 会出错？

1. 配置文件出错：如数据文件位置
2. LTP 中的底层模块出现错误。这种错误一般是因为处理的文本不规范，比如网页文本等。由于底层模块的健壮性不够，所以 LTP 无法从这种错误中恢复。需要您对要处理的文本进行一些预处理，使文本变得规范。

如何获得 LTP 的错误信息？

LTP 错误信息通过 stderr 输出，对于一些 GUI 程序，可以通过将 stderr 重定向到文件的方法获取这些错误信息。

LTP 分段方法？

按照回车换行{‘\r’, ‘\n’}分段。每一行为一个段落。如果段落为空或者只包含空白符，则扔掉，不作为一段。

LTP 分句方法？

根据标点

{

“。”, “！”, “？”, “；”, “：”, “　”[全角空格], “\r”, “\n”, “?”, “!”, “:”

}

将 LTP 的处理结果存储到 xml 文件有什么用处？

1. 通过 IE 打开，方便的查看文档的处理结果。注意当前目录下应该有一个 nlp_style.xsl 文件，否则会报错。

由于 nlp_style.xsl 需要执行脚本，因此 IE 可能会在地址栏下面蹦出一个警告："to help you protect your security, Internet Explorer has restricted this file from showing active content that could access your computer. Click here for options…" 请放心，我们的 nlp_style.xsl 绝对没有伤害您计算机安全的企图和功能。感兴趣的同僚可以看一下其内容。您可以使用左键点击一下出现的提示，然后选择 "Allow blocked content"，然后就可以查看了。

2. 作为处理结果保存，无须使用 LTP 再次处理文本，节省时间。如果您需要从已经生成的 xml 文件中抽取信息，只需要利用 LTP 接口将 xml 文件加载入内存，然后利用 LTP 接口抽取所需要的处理结果即可。您没有必要自己写脚本去解析 xml，因为我们的 xml 格式很有可能发生变化。

3. 保存 xml 文件唯一缺点是：xml 文件比较耗磁盘空间。

如果不想使用 LTP 中的分句模块怎么办?

不调用 SplitSentence()即可。

如果希望使用自己的分句模块怎么办？

您可以先使用自己的分句模块将文本分成句子，将每一句存成一行，然后交给 LTP 处理。在 LTP 处理过程中，不要调用 SplitSentence()。这样处理结果中，每一段都只包含一个句子。这个句子就是您分句模块分出的句子。这样做缺陷是，丢失了段落信息。对一些篇章级处理模块有一些影响。

如何扩展分词模块的词表？

在 ltp_data/irlas_data/extend_dict.dat 中的相应词性下添加新词即可。

如何配置 LTP?

ltp_all_modules.conf 可以配置 LTP 的数据资源位置，及某些模块的调用方式。目前为止，其中包含三项内容：

LTP_dataFolder 表示 LTP 各个底层模块所需的数据资源相对于工作目录的路经。请不要修改 ltp_data 下面各个子文件夹的结构。

IRLAS_confFile 表示分词词性标注 IRLAS 模块的配置文件名称，其路径无需设定，默认为$LTP_dataFolder/irlas_data/。IRLAS 的一些高级配置都在这个文件进行设置。

最后一个配置单文档文摘 SDS 的调用方式。