Windows X64-2021-SpecCCPro使用教程

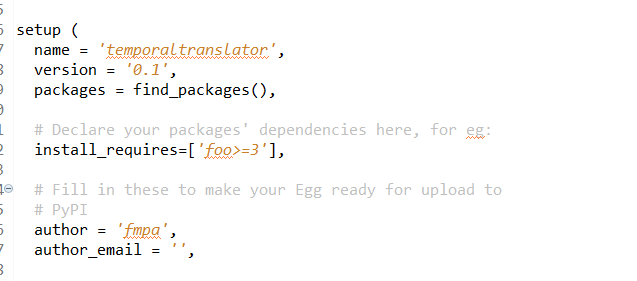
作者：李春奕

（1）安装 Everything-1.4.1.1009.x64-Setup用来配环境

（2）安装Anaconda2-2019.10-Windows-x86\_64,里面python版本下载

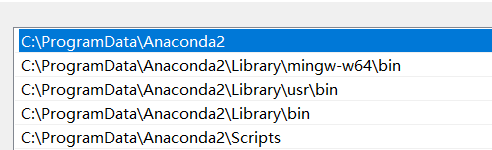


如果不使用Anaconda，直接安装python2.7，需要自己安装pip,setuptools,easyinstall,numpy等包，安装基本需要自己去官网下载whl包

1. 安装JPype1-0.7.1-cp27-cp27m-win\_amd64.whl，使用pip install命令
2. 在setup.py中

里面foo包是已经不存在了,但是不影响程序运行。

1. 安装jdk-8u301-windows-x64，并配好电脑环境

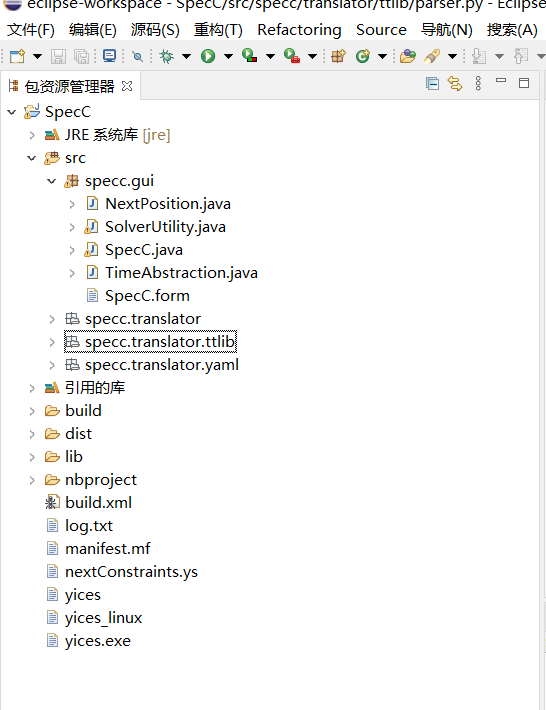




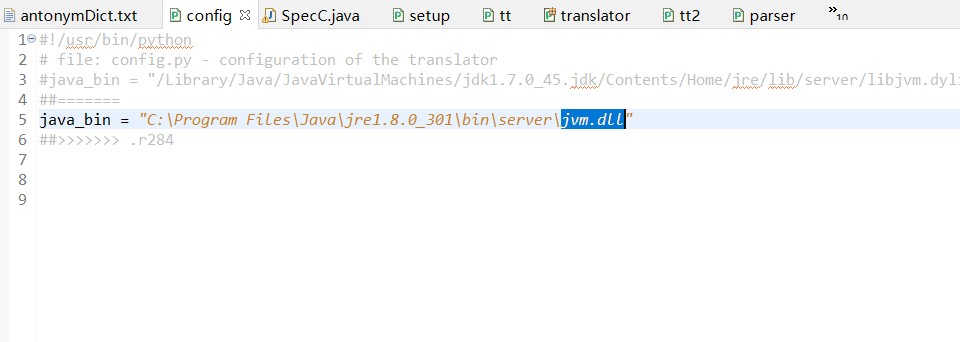


1. 将项目加载进eclipse

用引入文件的操作就可，注意安装python-dev等可调试python的功能，以便修改bug。运行主类SpecC.java可自动构建出项目，主类在manifest.mf配置。



1. 更改配置文件



1. 更改tt.py文件

|  |
| --- |
| # load the standford parser libraries through jpype  ous = OutSuppresser( sys.stdout ) #helper  ous.suppress( )  jvmPath = jpype.getDefaultJVMPath()  if jvmPath.find(*"libjvm.so"*)>=0 or jvmPath.find(*"libjvm.dylib"*)>= 0 or config.java\_bin.find(*"jvm.dll"*)>=0:  jpype.startJVM( jvmPath, *"-Djava.class.path=src/specc/translator/stanford-parser.jar"*, *"-Xmx500m"* )  else:  if config.java\_bin.find(*"libjvm.so"*)>=0 or config.java\_bin.find(*"libjvm.dylib"*)>=0 or *config.java\_bin.find("jvm.dll")>=0:*  jpype.startJVM( config.java\_bin, *"-Djava.class.path=src/specc/translator/stanford-parser.jar"*, *"-Xmx500m"* )  else:  sys.stderr.write(*"Unable to obtain javabin path automatically and please pecify path to your Java JVM library (libjvm.so or jvm.dll or libjvm.dylib) in config.py file."*)  ous.revert()    # create instance of the translator  tt = TemporalTranslator() |

1. 更改stanford包的方法

shortName 改变为 getShortName()

specific改变为getSpecific()

parent改变为getParent()

Parse.py代码如下

|  |
| --- |
| #!/usr/bin/python  # project: Temporal Logic For Men  # program: Temporal Translator  # description: Translates a sentence from English to LTL (temporal logic).  # author: Lukas Zilka  # date: March 2010  # file: parser.py - methods and classes for parsing natural language  import numpy  print *"hello"*  import jpype  import sys  import warnings  from ttlib import \*  class **Parser**:  *""" do nothing."""*  pass  class **Parse**:  *"""Holds results of a particular parse."""*  def **get\_dependencies**():  pass  class **StanfordParse**( Parse ):  *"""Holds parser output of The Stanford Parser."""*  def **\_\_init\_\_**( *self*, tdl ):  *self*.tdl = tdl  *self*.deps = *self*.build\_deps( tdl )    def **get\_dependencies**( *self* ):  return *self*.deps  def **build\_deps**( *self*, tdl ):  *"""Populate self with dependencies from stanford @tdl."""*  res = TypedDependencies()  for td in tdl:  if not td.reln().getShortName()==*"root"*:  d = TypedDependency()  d.relns = []  top\_act = act = td.reln()    d.reln\_specific=act.getSpecific()  #print d.reln\_specific  while act is not None:  d.relns.append( act.getShortName() )  act = act.getParent()  d.relns.append( *"%s\_%s"* % ( top\_act.getShortName(), d.reln\_specific, ) )  d.gov = Word( td.gov().value(), td.gov().index(), td.gov().parent().value() )#td.gov().value()?  d.dep = Word( td.dep().value(), td.dep().index(), td.dep().parent().value () )  #print d #delete  res.deps.append( d )    #print res.deps #delete  return res  class **StanfordParser**( Parser ):  *"""Implementation of parser interface to The Stanford Parser."""*  def **\_\_init\_\_**( *self* ):    # do the imports  lexparser = jpype.JPackage(*"edu.stanford.nlp.parser.lexparser"*)  *self*.\_trees = jpype.JPackage(*"edu.stanford.nlp.trees"*)    # suppress messages that the parser would print out  ous = OutSuppresser( sys.stderr )  ous.suppress()    # load and construct parser, and parse our sentence  LexicalizedParser = lexparser.LexicalizedParser  *self*.lp = LexicalizedParser.loadModelFromZip( *"src/specc/translator/englishPCFG.zip"*,*"englishPCFG.ser"* )  ous.revert()  def **parse**( *self*, sentence ):  *"""Parse the given sentence and return the corresponding Parse object."""*  parse = *self*.lp.apply( sentence )  tlp = *self*.\_trees.PennTreebankLanguagePack();  gsf = tlp.grammaticalStructureFactory();  gs = gsf.newGrammaticalStructure(parse);  tdl = gs.typedDependenciesCollapsed();  #print tdl #delete  res = StanfordParse( tdl )  return res  class **WordStemmer**:  def **\_\_init\_\_**( *self* ):  *self*.\_process = jpype.JPackage(*"edu.stanford.nlp.process"*)  def **stem**( *self*, word ):  *"""Get stem of @word."""*  ws = *self*.\_process.Morphology()  stem = ws.stem(word.value)  nw = Word( stem, word.index, word.type )  return nw  def **stemstr**( *self*, word ):  return *self*.stem( word ).value  class **Word**:  *"""Holds information of a word."""*  # the text  value = None  # the index in the sentence  index = None  def **\_\_init\_\_**( *self*, value, index = -1, type = None ):  *self*.value = value.lower()  *self*.index = index  *self*.type = type  def **\_\_repr\_\_**( *self* ):  return *"<Word: %s-%d>"* % ( *self*.value, *self*.index, )  def **\_\_eq\_\_**( *self*, other ):  # when comparing two words its enough to compare indexes as 2 words in one sentence  # will never have the same index.  if isinstance( other, Word ):  if *self*.index != -1 and other.index != -1:  return *self*.index == other.index  else:  return *self*.value == other.value  else:  return False  def **\_\_cmp\_\_**( *self*, other ):  return cmp( *self*.index, other.index )  def **\_\_hash\_\_**( *self* ):  return *self*.index  *@classmethod*  def match( *self*, w1, w2 ):  return w1.value.lower() == w2.value.lower()  class **Dependency**:  *"""Generic class representing gramatical dependency. E.g.: subj(assert, signal)"""*  # relationship name  def **reln**():  def **fget**( *self* ):  return *self*.relns[ 0 ]  return locals()  reln = property( \*\*reln() )    # last relationship name in the tree of relationship types  def **reln\_last**():  def **fget**( *self* ):  return *self*.relns[ -1 ]  return locals()  reln\_last = property( \*\*reln\_last() )  def **in\_relns**( *self*, what ):  if what in *self*.relns:  return True  else:  return False  relns = None # list of relation names (chronologically from current to ancestors)  gov = None # dependency head (governor)  dep = None # dependency tail (dependant)  #  # Typed dependencies  #  class **TypedDependency**( Dependency ):  def **\_\_repr\_\_**( *self* ):  return *"<TD: %s(%s, %s)>"* % ( *self*.reln, *self*.gov, *self*.dep, )  class **TypedDependencies**:  *"""Holds a set of typed dependencies (result of a parse)."""*  deps = None  def **\_\_init\_\_**( *self* ):  *self*.deps = []  def **\_\_iter\_\_**( *self* ):  for i in *self*.deps:  yield i    def **\_\_getitem\_\_**( *self*, item ):  return *self*.deps[ item ]    def **\_\_len\_\_**( *self* ):  return len( *self*.deps )  def **remove**( *self*, x ):  *self*.deps.remove( x ) |