# 第4章 **文本结构解析**

本章的内容将会让我们对文本的深层结构有一个更好的理解，并掌握解析文本的具体方法，以及如何在不同的NLP应用中使用它这些方法。如您所知，我们目前已经完成了NLP中的各种预处理步骤，接下来就该进入到一些更深层次的文本处理了。语言的结构是非常复杂的，我们需要按照其结构处理的各层次来对它进行描述。在本章中，我们将带您了解所有的文本结构，介绍这些结构之间的区别，并为您详细介绍其中部分结构的具体用法。另外，我们还将讨论**上下文无关语法（context-free grammar，简称CFG）**，以及它在NLTK库中的具体实现。我们也会带您浏览各种不同的文本解析器，并介绍如何使用NLTK库中现有的一些解析方法。具体而言就是：我们会用NLTK库来写一个浅解析器，其中将会再次讨论到语块化词处理语境中的NER问题。我们也会详细地为您介绍NLTK库中现有的一些可用于深层文本结构分析的选项。我们会试着为您提供一些关于信息提取的真实用例，以便介绍本章提及的这些话题所发挥的具体作用。总而言之，我们希望读者在阅读完本章之后能对这些话题有一定程度的理解。

在阅读完本章之后：

* 首先，我们会介绍文本解析是什么，以及与NLP相关的文本解析究竟是怎样的。
* 然后，我们会讨论各种不同的文本解析器，以及如何用NLTK库来执行解析。
* 最后，我们还将讨论文本解析在信息提取操作中的作用。

## 浅解析与深解析

通常情况下，在深入解析或者全面解析的过程中，像CFG、PCFG（即**probabilistic context-free grammar，概率性上下文无关语法**）以及搜索策略这样的语法概念的作用都是要将一套完整的语法结构应用到某个句子上。其中，浅解析（Shallow parsing）是一种面向给定文本的，对其语法信息部分所进行的有限解析任务。而深解析（deep parsing）则是一种更为复杂的应用。一般来说，深解析比较适合于对话系统和文本综述这样的应用，而浅解析则更适合于信息提取和文本挖掘这一类的应用。我们接下来将会用几节的篇幅来讨论它们各自的有缺利弊，以及在NLP应用中的具体用法。

## 两种解析方法

业界对于文本解析这个话题，主要存在着两种观点/方法，其具体情况如下表所示：

|  |  |
| --- | --- |
| **基于规则的方法** | **基于概率的方法** |
| 该方法基于规则和语法。 | 在该方法中，我们会通过运用概率模型来学习规则和语法。 |
| 在该方法中，我们将会基于GFG等语法概念来编撰语法规则手册。 | 该方法使用的是我们所观测到的相关语言特征的出现概率。 |
| 这是一个自上而下的方法。 | 这是一个自下而上的方法。 |
| 该方法中包含了CFG和基于表达式的解析器。 | 该方法中包含了PCFG和Stanford解析器。 |

## 为什么我们需要进行解析？

想要回答这个问题，我们就得再次请您先回忆一下当年在学校学习语法时的情况。现在请您告诉我，我们为什么要学习语法？我们真的需要学习语法吗？答案当然是肯定的！人长大了自然就要学习自己的母语。如今学习语言的过程通常是这样的：我们会先学习少量的词汇，接着学习少量的短语语块，再来就是少量的句子。在学习各个例句的过程中，我们会学到语言的结构。我们的母亲就是这样一次次地纠正我们说错的句子的。所以，当我们试图理解一个句子时都会采用这样一个类似的过程。但这个过程太常见了，以至于我们从来没有真正重视过，或者说仔细地思考过这个过程。也许等下次我们自己去纠正别人的语法的时就明白了。

于是当我们要编写解析器时，自然会想着要重复上述过程。如果我们能提出一组可被当作某种模版的规则，这些规则就能按照某种适当的顺序写出句子。另外，我们也需要将相关的单词进行分门别类。当让这个过程我们已经讨论过了，记得吗？词性标注的目的就是让我们知道给定单词所属的类别。

如果您理解了上述内容，那么就等于已经掌握了游戏规则，知道应该采取什么有效动作和特定步骤了。我们基本上追随的是一个人脑中非常自然的变化过程，并试图将其模拟出来。其中最简单的语法概念要从CFG 开始切入，这里我们所需要的只是一组规则和一个术语分词集。

下面，我们用非常有限的词汇量和非常通用的规则来写我们的第一个语法：

# toy CFG

>>> from nltk import CFG

>>> toy\_grammar = nltk.CFG.fromstring( """

S -> NP VP # S indicate the entire sentence

VP -> V NP # VP is verb phrase the

V -> "eats" | "drinks" # V is verb

NP -> Det N # NP is noun phrase (chunk that has noun in it)

Det -> "a" | "an" | "the" # Det is determiner used in the sentences

N -> "president" |"Obama" |"apple"| "coke" # N some example nouns

""")

>>> toy\_grammar.productions()

目前，这一语法概念所能产生的句子数量是很有限的。下面我们来思考一种情况：如果我们现在只知道如何将一个名词和一个动词搭配使用，并且这些动词和名词只能来自于我们在上述代码中所列出的单词，那么我们大概可以搭配出这样例句：[[1]](#footnote-1)

* President eats apple
* Obama drinks coke

现在我们来了解一下这个过程中究竟发生了些什么。我们在自己的脑海中创建了一个语法概念，它会基于上述规则和我们所提供的这些词汇来进行文本解析。如果我们能够正确地完成这个解析，就能够理解例句的含义。

由此可见，我们在学校里所学到的英文语法规则是有效的。显然，我们仍在使用这些语法，同时也在不断增强它们，我们用来理解所有英文句子的都是这一套相同的规则。但是，今天的规则显然不适用于莎士比亚时期所用的文体。

另一方面，同一套语法也可能会构造出一些毫无意义的句子，例如：

* Apple eats coke
* President drinks Obama

当我们具体涉及到某个**文本解析器（syntactic parser）**时，事实上本身就一定的机率在语法上会形成一些毫无意义的句子。因此想要获得其中的语义的话，我们就需要对句子的语义结构有一个更为深入的理解。我们建议读者应该具体查看一下与自己所感兴趣的语言相关的文本解析器。

## 不同的解析器类型

解析器通常要先对一个用于表达一组语法规则的输入字符串进行处理，然后构建出一个或多个可用于构成某种语法概念的规则。语法是我们用于衡量一个句子是否结构良好的的一份规范说明，而解析器则是一个用于解读语法的程序。该程序会通过搜索各种不同的树结构空间，找出给定句子的最佳树结构。下面我们来看一些现有的解析器，简要地认识一下这些解析器的运作细节及其实际用途。

### 递归下降解析器

先来看一个最简单的解析形式，我们称之为递归下降解析（recursive descent parsing）。这是一个自上而下的处理过程，由于该解析器会从左向右读取输入流中的信息，所以它会试图去验证其语法的正确性。该解析器的基本操作是从输入流中读取字符，然后将它们与终端所输入的语法规则说明进行匹配。我们的递归下降解析器会在其得到一次正确匹配时超前查看一个字符，并领先于其输入流的读取指针。

### 移位-归约解析器

移位-归约解析器（shift-reduce parser）是一种简单的、自下而上的解析器。和所有常见的、自下而上的解析器一样，移位-归约解析器也会试着去找到一个单词和短语序列，它们一方面对应着语法生成器的右侧，另一方面则会用生成器左侧的内容对其进行替换，直到其归简出完整的句子为止。

### 图表解析器

我们还会将算法设计中的动态规划技术应用到解析问题上。动态规划技术可以将中间结果保存下来，然后在适当的时候重新启用它们，以便显著地提高效率。这一技术也可以被应用到文本解析中。这样我们就可以将解析任务分成几部分来解决，并将各部分的结果先储存起来，然后在必要时再来考虑如何将其有效地组合成一个完整的解决方案。我们称这种方法为图表解析（chart parsing）。

|  |
| --- |
| 如果想更好地了解解析器的相关信息，读者也可以看看下面链接中的例子：<http://www.nltk.org/howto/parse.html> |

### 正则表达式解析器

正则表达式解析器使用的是一个正则表达式，定义该表达式的语法形式是在完成了词性标注的字符串之上构建而成的。该解析器将使用这些正则表达式来解析给定的句子，并为它们生成相应的解析树。下面我们就具体来看看正则表达式解析器的操作实例：

# Regex parser

>>> chunk\_rules=ChunkRule("<.\*>+","chunk everything")

>>> import nltk

>>> from nltk.chunk.regexp import \*

>>> reg\_parser = RegexpParser('''

NP: {<DT>? <JJ>\* <NN>\*} # NP

P: {<IN>} # Preposition

V: {<V.\*>} # Verb

PP: {<P> <NP>} # PP -> P NP

VP: {<V> <NP|PP>\*} # VP -> V (NP|PP)\*

''')

>>> test\_sent="Mr. Obama played a big role in the Health insurance bill"

>>> test\_sent\_pos=nltk.pos\_tag(nltk.word\_tokenize(test\_sent))

>>> paresed\_out=reg\_parser.parse(test\_sent\_pos)

>>> print paresed\_out

Tree('S', [('Mr.', 'NNP'), ('Obama', 'NNP'), Tree('VP', [Tree('V',

[('played', 'VBD')]), Tree('NP', [('a', 'DT'), ('big', 'JJ'), ('role',

'NN')])]), Tree('P', [('in', 'IN')]), ('Health', 'NNP'), Tree('NP',

[('insurance', 'NN'), ('bill', 'NN')])])

现在，我们再来看看上述代码所对应的树结构，它的图形化表示如下：

（图： 图中文字不用翻译）

在上述例子中，我们（基于POS的正则表达式）定义了模式的种类，希望用模式来产生短语，例如，如果某个短语匹配{<DT>？ <JJ> \* <NN>\*}这个模式，也就是说它是一个限定词后跟着一个形容词，再加一个名词，那么它大多数情况下会是一个名词性的短语。现在，这更是我们已经定义得到基于规则的解析树的语言规则。现在，我们已经越过了之前定义的语法规则，得到的是基于规则的解析树。

## 依存性文本解析

**依存性文本解析（dependency parsing，简称DP）**是一种现代化的文本解析机制。DP的主要概念是将各个语法单元（*单词*）用定向链路串联起来。这些链路就我们称之为语法上的依存关系（**dependencies**）。在目前的文本解析社区中，存在着大量这样的工作在进行。尽管短语结构式文本解析（phrase structure parsing）在一些词序自由的语言（例如捷克语和土耳其语）中仍被广泛使用，但依存性文本解析已经被证明是一种更为有效的方法。

在短语结构式文本解析与依存性文本解析之间存在着一个明显的区别，这点可以从它们所产生的解析树上看出来。例如，下面是“The big dog chased the cat”这句话的解析树：

（图：

Phrase structure tree ：短语结构树

Dependency tree ：依存关系树

其余部分不用翻译

）

如果我们仔细看一下这两棵解析树，短语结构树试图捕捉的首先是单词与短语之间的关系，然后再是短语与短语之间的关系。而依存关系树则只关心单词与单词之间的依存关系，例如*big*是完全依赖于*dog*的。

NLTK库中也提供了一些可用于执行依存性文本解析的方法。其中一个方法就是使用**概率性、投射依存性解析器（probabilistic**, **projective dependency parser）**，但该解析器得经由某个有限调校数据集来进行调校。依存性解析器的另一种形态就是Stanford解析器。幸运的是，NLTK库中也封装了该解析器。在下面的例子中，我们就来看看如何使用NLTK库中的Stanford解析器：

# Stanford Parser [Very useful]

>>> from nltk.parse.stanford import StanfordParser

>>> english\_parser = StanfordParser('stanford-parser.jar', 'stanfordparser-3.4-models.jar')

>>> english\_parser.raw\_parse\_sents(("this is the english parser test")

Parse

(ROOT

(S

(NP (DT this))

(VP (VBZ is)

(NP (DT the) (JJ english) (NN parser) (NN test)))))

Universal dependencies

nsubj(test-6, this-1)

cop(test-6, is-2)

det(test-6, the-3)

amod(test-6, english-4)

compound(test-6, parser-5)

root(ROOT-0, test-6)

Universal dependencies, enhanced

nsubj(test-6, this-1)

cop(test-6, is-2)

det(test-6, the-3)

amod( test-6, english-4)

compound(test-6, parser-5)

root(ROOT-0, test-6)

这段输出咋看之下似乎很复杂，其实一点都不复杂。这段输出是一个三段式的结果列表，其中第一段输出的只是给定句子中的POS标签及其解析树。我们会在下图中用更为优雅的绘图方式为您呈现了相同的结构。接下来的第二段输出的是给定单词之间的依存关系以及它们各自的位置。最后的第三段输出的是这些依存关系的扩大版：

（图：图中内容无需翻译）

|  |
| --- |
| 如果想对Stanford解析器的使用有一个更好的理解，读者可以参考下面链接中的内容http://nlpviz.bpodgursky.com/home and http://nlp.stanford.edu:8080/parser/index.jsp. |

## 语块化处理

语块化处理属于浅解析，它并不会深入到句子的深层结构。在该操作中，我们的目的是尽量将句子分割成一些有意义的构成语块。

我们可以将语块定义成文本解析中最小的可处理单元。例如，我们可以将“the President speaks about the health care reforms”这个句子分解成两个语块。第一个语块是“the President”，该语块由名词主导，所以我们称之为**名词短语（NP）**。而该句子的其余部分则是一个以动词为主导的语块，因而我们称之为**动词短语（VP）**。如果我们观察得再仔细一些，就会发现“speaks about the health care reforms”这部分还可以再分出子语块。具体来说就是其中还存在更多的NP，因而还可以再继续被分解成“speaks about”和“health care reforms”，整个过程如下图所示：

（图：图中内容无需翻译）

这种将句子划分成各个部分的过程，就是我们之前所说的语块化处理。从形式上来看，语块化处理也可以被看作是一种处理接口，作用是识别出任意文本中互不重叠的分组。

现在，我们已经了解了浅解析与深解析之间的区别。当我们能在CFG的帮助下进入到句子的语法结构，并理解了这些句子的语法结构时。有些时候我们就需要用文本解析来理解句子的具体含义。当然在另一方面，也有一些情况是不需要我们做如此深度的分析的。比方说，对于大部分非结构化文本，我们一般只会想要提取其中的关键短语、命名实体或者相关项目的特定模式。在这种情况下，我们要做的是浅解析而非深解析，因为深解析会去处理所有违反语法规则的句子，也会产生各种不同的语法树，直到解析器在反复回溯的过程中找到最佳的解析树。整个过程非常耗时和繁琐，并且即使完成了所有的这些处理过程，我们也未必会得到正确的解析树。而浅解析则可以用语块来保证其浅解析结构，这种处理相对而言要较快一些。下面，我们就写一些代码来做一些基本的语块化处理：

# Chunking

>>> from nltk.chunk.regexp import \*

>>> test\_sent="The prime minister announced he had asked the chief government whip, Philip Ruddock, to call a special party room meeting for 9am on Monday to consider the spill motion."

>>> test\_sent\_pos=nltk.pos\_tag(nltk.word\_tokenize(test\_sent))

>>> rule\_vp = ChunkRule(r'(<VB.\*>)?(<VB.\*>)+(<PRP>)?', 'Chunk VPs')

>>> parser\_vp = RegexpChunkParser([rule\_vp],chunk\_label='VP')

>>> print parser\_vp.parse(test\_sent\_pos)

>>> rule\_np = ChunkRule(r'(<DT>?<RB>?)?<JJ|CD>\*(<JJ|CD><,>)\*(<NN.\*>)+', 'Chunk NPs')

>>> parser\_np = RegexpChunkParser([rule\_np],chunk\_label="NP")

>>> print parser\_np.parse(test\_sent\_pos)

(S

The/DT

prime/JJ

minister/NN

(VP announced/VBD he/PRP)

(VP had/VBD asked/VBN)

the/DT

chief/NN

government/NN

whip/NN

….

….

….

(VP consider/VB)

the/DT

spill/NN

motion/NN

./.)

(S

(NP The/DT prime/JJ minister/NN) # 1st noun phrase

announced/VBD

he/PRP

had/VBD

asked/VBN

(NP the/DT chief/NN government/NN whip/NN) # 2nd noun phrase

,/,

(NP Philip/NNP Ruddock/NNP)

,/,

to/TO

call/VB

(NP a/DT special/JJ party/NN room/NN meeting/NN) # 3rd noun phrase

for/IN

9am/CD

on/IN

(NP Monday/NNP) # 4th noun phrase

to/TO

consider/VB

(NP the/DT spill/NN motion/NN) # 5th noun phrase

./.)

The preceding code is good enough to do some basic chunking of verb and noun phrases. A conventional pipeline in chunking is to tokenize the POS tag and the input string before they are ed to any chunker. Here, we use a regular chunker, as rule NP / VP defines different POS patterns that can be called a verb/noun phrase. For example, the NP rule defines anything that starts with the determiner and then there is a combination of an adverb, adjective, or cardinals that can be chunked in to a noun phrase. Regular expression-based chunkers rely on chunk rules defined manually to chunk the string. So, if we are able to write a universal rule that can incorporate most of the noun phrase patterns, we can use regex chunkers. Unfortunately, it's hard to come up with those kind of generic rules; the other approach is to use a machine learning way of doing chunking. We briefly touched upon ne\_chunk() and the Stanford NER tagger that both use a pre-trained model to tag noun phrases.

前面的代码是不够好，做动词和名词短语一些基本的分块。在分块传统的管道来标记POS标签和他们编到任何细节化之前输入的字符串。在这里，我们使用普通的概括化的，因为规则NP/ VP定义了不同的POS模式，可以称得上是动词/名词短语。例如，对NP规则定义任何与确定器启动，然后有一个副词，形容词，或红雀可以分块到一个名词短语的组合。基于正则表达式，chunkers依靠手动定义为块中的字符串块规则。所以，如果我们能够编写可以将大部分的名词短语模式的普遍规律，我们可以使用正则表达式chunkers。不幸的是，这是很难拿出这些类型的一般规则;另一种方法是使用干分块的机器学习方法。在ne\_chunk（）和斯坦福大学NER恶搞因为它们都使用一个预先训练模型标记名词短语我们短暂触及。

## 信息提取

We learnt about taggers and parsers that we can use to build a basic information extraction engine. Let's jump directly to a very basic IE engine and how a typical IE engine can be developed using NLTK.

我们了解了标注器和分析器，我们可以用它来建立一个基本的信息提取引擎。让我们直接跳转到一个非常基本的IE引擎，以及如何一个典型的IE引擎可以使用NLTK开发。

Any sort of meaningful information can be drawn only if the given input stream goes to each of the following NLP steps. We already have enough understanding of sentence tokenization, word tokenization, and POS tagging. Let's discuss NER and relation extraction as well.

任何类型的有意义的信息可以仅在给定的输入数据流去以下每个NLP步骤绘制。我们已经有句话符号化，符号化这个词，和词性标注足够的认识。让我们来讨论NER和关系抽取为好。

A typical information extraction pipeline looks very similar to that shown in the following figure:

一个典型的信息抽取管道看起来非常相似，在如下图所示：

Sentence

T

okenization

W

ord

T

okenization

P

art-of

Speech

-

T

agging

Entity

Detection

Relation

Extraction

raw

text

relations

list

(

of

strings)

list

(

of

list

of

strings)

list

(

of

list

of

tuples)

list

(

of

trees)

String

Some of the other preprocessing steps, such as stop word removal and

stemming, are generally ignored and do not add any value to an IE

engine.

### 命名实体识别（NER）

We already briefly discussed NER generally in the last chapter. Essentially, NER is a way of extracting some of the most common entities, such as names, organizations, and locations. However, some of the modified NER can be used to extract entities such as product names, biomedical entities, author names, brand names, and so on. Let's start with a very generic example where we are given a text file of the content and we need to extract some of the most insightful named entities from it:

我们已经简要地在最后一章讨论NER一般。从本质上讲，NER是提取一些最常见的实体，如姓名，组织，和位置的方法。然而，一些改性NER可用于提取的实体，如产品名，生物医学的实体，作者名称，品牌名称，等等。让我们开始，我们被赋予的内容的文本文件，一个很普通的例子，我们需要提取一些从它最有见地的命名实体：

# NP chunking (NER)

>>> f=open(# absolute path for the file of text for which we want NER)

>>> text=f.read()

>>> sentences = nltk.sent\_tokenize(text)

>>> tokenized\_sentences = [nltk.word\_tokenize(sentence) for sentence in sentences]

>>> tagged\_sentences = [nltk.pos\_tag(sentence) for sentence in tokenized\_ sentences]

>>> for sent in tagged\_sentences:

>>> print nltk.ne\_chunk(sent)

In the preceding code, we just followed the same pipeline provided in the preceding figure. We took all the preprocessing steps, such as sentence tokenization, tokenization, POS tagging, and NLTK. NER (pre-trained models) can be used to extract all NERs.

在上面的代码中，我们只是跟着前面的图中所提供的相同的管道。我们把所有的预处理步骤，如句子符号化，符号化，词性标注和NLTK。 NER（预训练机型）可以用来提取所有净入学率。

### 关系提取

Relation extraction is another commonly used information extraction operation. Relation extraction as it sound is the process of extracting the different relationships between different entities. There are variety of the relationship that exist between the entities. We have seen relationship like inheritance/synonymous/analogous. The definition of the relation can be dependent on the Information need. For example in the case where we want to look from unstructured text data who is the writer of which book then authorship could be a relation between the author name and book name. With NLTK the idea is to use the same IE pipeline that we used till NER and extend it with a relation pattern based on the NER tags.

关系抽取是另一种常用的信息提取操作。关系提取，因为它的声音中提取不同实体之间的不同关系的过程。有各种各样的实体之间存在的关系。我们已经看到，如继承/同义词/类似的关系。关系的定义可以是依赖于信息的需要。例如，在这里我们想从谁是书的作者，然后可能是作者的名字和书籍的名称之间的关系的作家非结构化的文本数据看的情况。随着NLTK的想法是使用相同的IE浏览器的管道，我们沿用到NER并与基于NER标记的关系模式扩展它。

So, in the following code, we used an inbuilt corpus of ieer, where the sentences are tagged till NER and the only thing we need to specify is the relation pattern we want and the kind of NER we want the relation to define. In the following code, a relationship between an organization and a location has been defined and we want to extract all the combinations of these patterns. This can be applied in various ways, for example, in a large corpus of unstructured text, we will be able to identify some of the organizations of our interest with their corresponding location:

因此，在下面的代码中，我们使用的能源与环境研究所，这里的句子的标签，直到NER，我们需要指定的唯一的事情就是我们想要的，我们要的关系来定义的关系模式和一种NER的一个内置的语料库。在下面的代码，组织和位置之间的关系已经确定，我们要提取这些模式的所有组合。这可以以各种方式被应用，例如，在一个大的语料库非结构化文本的，我们将能够确定一些我们的与它们对应的位置感兴趣的组织：

>>> import re

>>> IN = re.compile(r'.\*\bin\b(?!\b.+ing)')

>>> for doc in nltk.corpus.ieer.parsed\_docs('NYT\_19980315'):

>>> for rel in nltk.sem.extract\_rels('ORG', 'LOC', doc, corpus='ieer', pattern = IN):

>>> print(nltk.sem.rtuple(rel))

[ORG: u'WHYY'] u'in' [LOC: u'Philadelphia']

[ORG: u'McGlashan &AMP; Sarrail'] u'firm in' [LOC: u'San Mateo']

[ORG: u'Freedom Forum'] u'in' [LOC: u'Arlington']

[ORG: u'Brookings Institution'] u', the research group in' [LOC: u'Washington']

[ORG: u'Idealab'] u', a self-described business incubator based in' [LOC: u'Los Angeles'] ..

## 本章小结

We moved beyond the basic preprocessing steps in this chapter. We looked deeper at NLP techniques, such as parsing and information extraction. We discussed parsing in detail, which parsers are available, and how to use NLTK to do any NLP parsing. You understood the concept of CFG and PCFG and how to learn from a tree bank and build a parser. We talked about shallow and deep parsing and what the difference is between them.

我们超越了本章的基本预处理步骤。我们在NLP技术，如分析和信息提取看起来更深。我们讨论分析详细，其中解析器是可用的，以及如何使用NLTK做任何NLP解析。你了解CFG和PCFG以及如何从树上银行学习和建立一个解析器的概念。我们谈到了浅层和深层分析和什么区别它们之间。

We also talked about some of the information extraction essentials, such as entity extraction and relation extraction. We talked about a typical information extraction engine pipeline. We saw a very small and simple IE engine that can be built in less than 100 lines of code. Think about this kind of system running on an entire Wikipedia dump or an entire web content related to an organization. Cool, isn't it?

我们也谈到了一些信息提取要领，如实体提取和关系抽取的。我们谈到一个典型的信息提取引擎的管道。我们看到，可建在不到100行代码一个非常小而简单的IE引擎。想想这种对整个维基百科转储运行的系统或与之相关的一个组织一个完整的网页内容。很酷，不是吗？

We will use some of the topics we've learnt in this chapter in further chapters to build some useful NLP applications.

我们将使用一些我们在这一章中所学到的进一步章节的主题建立一些有用的NLP应用。

1. 译者注：由于原文针对的是英文语法分析，所以这里的例句就不翻译了。 [↑](#footnote-ref-1)