# 第4章 **文本结构解析**

This chapter involves a better understanding of deep structure in text and also how to deep parse text and use it in various NLP applications. Now, we are equipped with various NLP preprocessing steps. Let's move to some deeper aspect of the text. The structure of language is so complex that we can describe it by various layers of structural processing. In this chapter we will touch upon all these structures in text, differentiate between them, and provide you with enough details about the usage of one of these. We will talk about **context-free grammar** (**CFG**) and how it can be implemented with NLTK. We will also look at the various parsers and how we can use some of the existing parsing methods in NLTK. We will write a shallow parser in NLTK and will again talk about NER in the context of chunking. We will also provide details about some options that exist in NLTK to do deep structural analysis. We will also try to give you some real-world use cases of information extraction and how it can be achieved by using some of the topics that you will learn in this chapter.

本章包括更好地理解文本的深层结构中，以及如何深解析文本，并在不同的NLP应用程序中使用它。现在，我们配备了各种NLP预处理步骤。让我们进入到文本的一些深层次的方面。语言的结构是如此复杂，我们可以由结构处理的各种层描述它。在本章中，我们将在后文所有这些结构碰，它们之间的区别，并为您提供关于其中之一的使用足够的细节。我们将谈论上下文无关文法（CFG），以及它如何与NLTK实施。我们也将着眼于不同的解析器和我们如何使用一些在NLTK现有的解析方法。我们将写NLTK浅解析器将再次谈论NER在分块的情况下。我们也将提供关于存在于NLTK做深层结构分析一些选项的详细信息。我们也将尝试给你信息提取的一些真实的使用案例以及如何通过使用一些，你将在本章学习的主题来实现。

We want you to have an understanding of these topics by the end of this chapter.

我们希望您通过本章的最后有这些主题的理解。

In this chapter:

* We will also see what parsing is and what is the relevance of parsing in NLP.
* We will then explore different parsers and see how we can use NLTK for parsing.
* Finally, we will see how parsing can be used for information extraction.
* 我们还将看到什么语法分析，什么是NLP解析的相关性。
* 然后，我们将探讨不同的解析器，看看我们如何可以使用NLTK解析。
* 最后，我们将看到解析如何使用信息提取。

## 浅解析与深解析

In deep or full parsing, typically, grammar concepts such as CFG, and **probabilistic context-free grammar** (**PCFG**), and a search strategy is used to give a complete syntactic structure to a sentence. Shallow parsing is the task of parsing a limited part of the syntactic information from the given text. While deep parsing is required for more complex NLP applications, such as dialogue systems and summarization, shallow parsing is more suited for information extraction and text mining varieties of applications. I will talk about these in the next few sections with more details about their pros and cons and how we can use them for our NLP application.

在深或全解析，通常情况下，语法概念，如CFG，和概率上下文无关文法（PCFG）和搜索策略是用来给一个完整的语法结构一个句子。浅层分析分析是从给定文本的句法信息有限的一部分任务。而需要更复杂的自然语言处理的应用程序，如对话的系统和聚合深解析，浅层分析更适合信息提取和文本挖掘品种的应用程序。我会谈论这些在接下来的几节有关其利弊，我们如何能够利用它们为我们的NLP应用的更多细节。

## 两种解析方法

There are mainly two views/approaches used to deal with parsing, which are as follows:

主要有两种观点/方法用来对付解析，这是如下：

|  |  |
| --- | --- |
| **The rule-based approach** | **The probabilistic approach** |
| This approach is based on rules/grammar | In this approach, you learn rules/grammar by using probabilistic models |
| Manual grammatical rules are coded down in CFG, and so on, in this approach | This uses observed probabilities of linguistic features |
| This has a top-down approach | This has a bottom-up approach |
| This approach includes CFG and Regex- based parser | This approach includes PCFG and the Stanford parser |

## 为什么我们需要进行解析

I again want to take you guys back to school, where we learned grammar. Now tell me why you learnt grammar Do you really need to learn grammar? The answer is definitely yes! When we grow, we learn our native languages. Now, when we typically learn languages, we learn a small set of vocabulary. We learn to combine small chunks of phrases and then small sentences. By learning each example sentence, we learn the structure of the language. Your mom might have corrected you many times when you uttered an incorrect sentence. We apply a similar process when we try to understand the sentence, but the process is so common that we never actually pay attention to it or think about it in detail. Maybe the next time you correct someone's grammar, you will understand.

我再次想带你们回学校，在那里我们学到的语法。现在告诉我，为什么你学到的语法你真的需要学习语法？答案当然是肯定的！当我们长大，我们学习母语。现在，当我们通常学习语言，我们学习小组词汇。我们学习短语的小块，然后小句子结合起来。通过学习每个例句中，我们学习语言的结构。当你说出一个不正确的句子你妈妈可能已经纠正了你很多次。我们采用类似的过程，当我们试图理解句子，但这个过程是很常见的，我们从来没有真正重视起来，或认为它在细节。也许下次你纠正别人的语法的时候，你就明白了。

When it comes to writing a parser, we try to replicate the same process here. If we come up with a set of rules that can be used as a template to write the sentences in a proper order. We also need the words that can fit into these categories. We already talked about this process. Remember POS tagging, where we knew the category of the given word?

当谈到编写解析器，我们试图在这里重复同样的过程。如果我们拿出一组规则可以被用作模板来写的句子在一个适当的顺序。我们还需要能够适应这些类别的话。我们已经谈到了这个过程。记住词性标注，我们知道定单词的类别？

Now, if you've understood this, you have learned the rules of the game and what moves are valid and can be taken for a specific step. We essentially follow a very natural phenomenon of the human brain and try to emulate it. One of the simplest grammar concepts to start with is CFG, where we just need a set of rules and a set of terminal tokens.

Let's write our first grammar with very limited vocabulary and very generic rules:

让我们写我们的第一个语法非常有限的词汇和非常通用的规则：

# 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()

Now, this grammar concept can generate a finite amount of sentences. Think of a situation where you just know how to combine a noun with a verb and the only verbs and nouns you knew were the ones we used in the preceding code. Some of the example sentences we can form from these are:

现在，这一概念的语法可以产生的句子数量是有限的。想想一个情况下，你只是知道如何将一个名词与动词，你就知道是我们在前面的代码中使用的那些唯一的动词和名词相结合。一些例子的句子就能形成从它们是：

* President eats apple
* Obama drinks coke
* 总统吃苹果
* 奥巴马饮料可乐

Now, understand what's happening here. Our mind has created a grammar concept to parse based on the preceding rules and substitutes whatever vocabulary we have.

现在，了解这里发生了什么。我们的头脑创造了一个语法概念，基于前面的规则和替代任何词汇，我们有进行解析。

If we are able to parse correctly, we understand the meaning.

如果我们能够正确解析，我们明白其中的含义。

So, effectively, the grammar we learnt at school constituted the useful rules of English. We still use those and also keep enhancing them and these are the same rules we use to understand all English sentences. However, today's rules do not apply to William Shakespeare's body of work.

因此，有效的，我们在学校学到的语法构成的英语的使用规则。我们仍然使用这些也不断增强他们，这些都是我们用来理解所有英语句子相同的规则。然而，今天的规则并不适用于工作莎士比亚的尸体。

On the other hand, the same grammar can construct meaningless sentences such as:

另一方面，同样的语法可以构造无意义的句子，例如：

* Apple eats coke
* President drinks Obama
* 苹果吃焦
* 总统奥巴马饮料

When it comes to a **syntactic parser**, there is a chance that a syntactically formed sentence could be meaningless. To get to the semantics, we need a deeper understanding of semantics structure of the sentence. I encourage you to look for a semantic parser in case you are interested in these aspects of language.

当涉及到句法分析，有一个语法上形成有句话是毫无意义的机会。要到语义，我们需要的句子语义结构的更深层次的理解。我建议你找如果你有兴趣的语言这些方面语义解析器。

## 不同的解析器类型

A parser processes an input string by using a set of grammatical rules and builds one or more rules that construct a grammar concept. Grammar is a declarative specification of a well-formed sentence. A parser is a procedural interpretation of grammar. It searches through the space of a variety of trees and finds an optimal tree for the given sentence. We will go through some of the parsers available and briefly touch upon their workings in detail for awareness, as well as for practical purposes.

解析器通过使用一组语法规则处理输入字符串，并生成该构建语法的概念的一个或多个规则。语法是一个结构良好的一句话的声明规范。解析器是语法的过程解读。它搜索通过多种树木空间和找到一个最优树给定的句子。我们将通过一些现有的解析器，并简要详细谈及自己运作的认识，以及对实际用途。

### A recursive descent parser

One of the most straightforward forms of parsing is recursive descent parsing. This is a top-down process in which the parser attempts to verify that the syntax of the input stream is correct, as it is read from left to right. A basic operation necessary for this involves reading characters from the input stream and matching them with the terminals from the grammar that describes the syntax of the input. Our recursive descent parser will look ahead one character and advance the input stream reading pointer when it gets a proper match.

一个解析的最简单的形式是递归下降解析。这是一个顶向下的方法，其中所述分析器试图验证该输入流的语法是正确的，因为它是从左至右读取。为此需要一个基本的操作涉及从输入流中读取字符，并将它们与从描述输入的语法的语法端子匹配。我们的递归下降解析器会向前看一个字符和超前的输入流中读出指针时，它得到正确的匹配。

### A shift-reduce parser

The shift-reduce parser is a simple kind of bottom-up parser. As is common with all bottom-up parsers, a shift-reduce parser tries to find a sequence of words and phrases that correspond to the right-hand side of a grammar production and replaces them with the left-hand side of the production, until the whole sentence is reduced.

该移位归约解析器是一个简单的一种自下而上的解析器。如同所有自下而上解析器常见，移位 - 归约解析器试图找到对应于一个语法生产右手侧，并与生产的左侧它们替换的单词和短语的序列，直到整个句子被降低。

### A chart parser

We will apply the algorithm design technique of dynamic programming to the parsing problem. Dynamic programming stores intermediate results and reuses them when appropriate, achieving significant efficiency gains. This technique can be applied to syntactic parsing. This allows us to store partial solutions to the parsing task and then allows us to look them up when necessary in order to efficiently arrive at a complete solution. This approach to parsing is known as chart parsing.

我们将应用动态规划的算法设计技术来解析问题。动态规划保存中间结果，并重新使用他们在适当的时候，取得了显著效率。这一技术可以应用到句法分析。这使我们能够部分解决存储解析任务，然后让我们来看看他们在必要时，以一个完整的解决方案，有效地到达。这种方法解析被称为图表解析。

|  |
| --- |
| For a better understanding of the parsers, you can go through an example at http://www.nltk.org/howto/parse.html. |

### A regex parser

A regex parser uses a regular expression defined in the form of grammar on top of a POS-tagged string. The parser will use these regular expressions to parse the given sentences and generate a parse tree out of this. A working example of the regex parser is given here:

一个正则表达式解析器使用在一个POS标签的字符串之上语法形式定义的正则表达式。解析器将使用这些正则表达式来分析给定的句子并生成分析树出于此。正则表达式解析器的工作的例子在这里给出：

# 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')])])

The following is a graphical representation of the tree for the preceding code:

Root

Mr

.

Obama

played

DT

VBD

NNP

NP

NP

NP

DT

NN

NN

NN

IN

PP

in

insurance

role

big

JJ

VP

s

a

NNP

NNP

Health

the

bill

In the current example, we define the kind of patterns (a regular expression of the POS) we think will make a phrase, for example, anything that {<DT>? <JJ>\* <NN>\*} has a starting determiner followed by an adjective and then a noun is mostly a noun phrase. Now, this is more of a linguistic rule that we have defined to get the rule-based parse tree.

在当前的例子，我们定义的那种模式（在POS的正则表达式），我们认为将一个短语，例如，任何{<DT>？ <JJ> \* <NN>\*}有一个由形容词出发确定后面，然后一个名词大多是名词短语。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。

## 依存句法分析

**Dependency parsing** (**DP**) is a modern parsing mechanism. The main concept of DP is that each linguistic unit (*words*) is connected with each other by a directed link. These links are called **dependencies** in linguistics. There is a lot of work going on in the current parsing community. While **phrase structure parsing** is still widely used for free word order languages (Czech and Turkish), dependency parsing has turned out to be more efficient.

依存分析（DP）是一座现代化的解析机制。 DP的主要概念是每个语言单位（字）由定向链路彼此连接。这些链接被称为语言学的依赖。有很多工作在当前解析社区开展的活动。虽然短语结构解析仍然被广泛用于免费词序语言（捷克和土耳其），依存分析已被证明是更有效的。

A very clear distinction can be made by looking at the parse tree generated by phrase structure grammar and dependency grammar for a given example, as the sentence "The big dog chased the cat". The parse tree for the preceding sentence is:

一个很明显的区别可以通过查看短语结构语法和依存语法对于给定的例子中产生的解析树进行，因为句中的“大狗追猫”。对于前一句的分析树：

**Phrase**

**Structure**

**tree**

**Dependency**

**Tree**

The

big

dog

chased

the

cat

S

NP

VP

Art

Adj

N

V

N

P

Art

N

the

big

dog

chased

the

cat

If we look at both parse trees, the phrase structures try to capture the relationship between words and phrases and then eventually between phrases. While a dependency tree just looks for a dependency between words, for example, *big* is totally dependent on *dog*.

如果我们看一下这两个分析树，短语结构，试图捕捉的单词和短语，然后短语之间最终的关系。虽然依赖关系树只是看起来单词之间的相关性，例如，大的是完全依赖于狗。

NLTK provides a couple of ways to do dependency parsing. One of them is to use a **probabilistic**, **projective dependency parser**, but it has the restriction of training with a limited set of training data. One of the state of the art dependency parsers is a Stanford parser. Fortunately, NLTK has a wrapper around it and in the following example, I will talk about how to use a Stanford parser with NLTK:

NLTK提供了一些方法可以做到依存分析。其中之一是使用概率，投射依赖分析器，但它具有的训练与有限的一组训练数据的限制。本领域的依赖解析器的状态是斯坦福大学的解析器。幸运的是，NLTK周围有包装，并在下面的例子中，我将讨论如何使用斯坦福解析器与NLTK：

# 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)

The output looks quite complex but, in reality, it's not. The output is a list of three major outcomes, where the first is just the POS tags and the parsed tree of the given sentences. The same is plotted in a more elegant way in the following figure. The second is the dependency and positions of the given words. The third is the enhanced version of dependency:

输出看起来非常复杂，但在现实中，它不是。输出是三个主要结果，其中第一仅仅在POS标签和给定的句子的语法分析树列表。同样是在如下图更优雅的方式绘制。二是依赖和给定的话立场。第三是依赖性的增强版本：

Root

is

NP

s

VBZ

VP

this

DT

NP

DT

NN

NN

parser

JJ

english

the

test

|  |
| --- |
| For a better understanding of how to use a Stanford parser, refer to  http://nlpviz.bpodgursky.com/home and http://nlp.stanford.edu:8080/parser/index.jsp. |

## 分词处理

Chunking is shallow parsing where instead of reaching out to the deep structure of the sentence, we try to club some chunks of the sentences that constitute some meaning.

分块是浅层分析的地方，而不是深入到句子的深层结构，我们尽量会构成一定的意义句子的一些块。

A chunk can be defined as the minimal unit that can be processed. So, for example, the sentence "the President speaks about the health care reforms" can be broken into two chunks, one is "the President", which is noun dominated, and hence is called a **noun phrase** (**NP**). The remaining part of the sentence is dominated by a verb, hence it is called a **verb phrase** (**VP**). If you see, there is one more sub-chunk in the part "speaks about the health care reforms". Here, one more NP exists that can be broken down again in "speaks about" and "health care reforms", as shown in the following figure:

一大块可以被定义为可处理的最小单元。因此，例如，句子“总统谈到的医疗改革”可以被分解为两个组块，一个是“总统”，这是名词支配，并且因此被称为一个名词短语（NP）。这句话的剩余部分是由动词为主，因此，它被称为一个动词短语（VP）。如果你看到，还有一个分块的部分“谈到了医疗改革”。在这里，多了一个NP存在，可以再次细分“谈”和“医疗改革”，如图所示如下图：

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| VP   |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | The | President | Speaks | about | The | Health | Care | Reforms |   NP NP |

This is how we broke the sentence into parts and that's what we call chunking. Formally, chunking can also be described as a processing interface to identify non-overlapping groups in unrestricted text.

这是我们打破了句子分成部分，这就是我们所说的分块。形式上，分块也可以被描述为一个处理接口，以确定在不受限制的文本不重叠的组。

Now, we understand the difference between shallow and deep parsing. When we reach the syntactic structure of the sentences with the help of CFG and understand the syntactic structure of the sentence. Some cases we need to go for semantic parsing to understand the meaning of the sentence. On the other hand, there are cases where, we don't need analysis this deep. Let's say, from a large portion of unstructured text, we just want to extract the key phrases, named entities, or specific patterns of the entities. For this, we will go for shallow parsing instead of deep parsing because deep parsing involves processing the sentence against all the grammar rules and also the generation of a variety of syntactic tree till the parser generates the best tree by using the process of backtracking and reiterating. This entire process is time consuming and cumbersome and, even after all the processing, you might not get the right parse tree. Shallow parsing guarantees the shallow parse structure in terms of chunks which is relatively faster. So, let's write some code snippets to do some basic chunking:

现在，我们了解浅层和深层解析之间的区别。当我们到达与CFG的帮助下，句子的句法结构，理解句子的句法结构。有些情况下，我们需要去的语义分析来理解句子的含义。另一方面，在有些情况下，我们不需要分析此深的情况。比方说，从非结构化文本中的很大一部分，我们只是想提取关键短语，命名实体或实体的具体模式。对于这一点，我们会去浅层分析，而不是深层分析，因为深分析涉及到处理对所有的语法规则的句子，也产生了各种语法树，直到解析器使用回溯并重申的过程中会产生最好的树。这整个过程是耗时和麻烦的，并且即使之后所有的处理，则可能不会得到正确的分析树。浅层分析保证了块而言浅解析结构，这是相对较快的。所以，让我们写一些代码片段做一些基本的分块：

# 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.

### Named-entity recognition (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。

### Relation extraction

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。