第 20 章

项目1：自动添加标签

本章介绍如何使用Python杰出的文本处理功能，包括使用正则表达式将纯文本文件转换为用HTML或XML等语言标记的文件。如果不熟悉这些语言的人编写了一些文本，而你要在系统中使用这些内容并对其进行标记，就必须具备这些技能。

你不能熟练地使用XML？不用为此担心，只要对HTML有大致的了解，你就能完成本章的任务。如果需要阅读HTML简介，网上的相关教程数不胜数。有关XML使用示例，请参阅第22章。

下面先来实现一个只能做基本处理的简单原型，再对这个程序进行扩展，让标记系统更灵活。

20.1 问题描述

你要给纯文本文件添加格式。假设你要将一个文件用作网页，而给你文件的人嫌麻烦，没有以HTML格式编写它。你不想手工添加需要的所有标签，想编写一个程序来自动完成这项工作。

注意 事实上，这种“纯文本标记”在最近几年已非常普遍，主要原因可能是带纯文本界面的维基百科和博客软件呈爆炸式增长。有关这方面的详细信息，请参阅20.6节。

大致而言，你的任务是对各种文本元素（如标题和突出的文本）进行分类，再清晰地标记它们。就这里的问题而言，你将给文本添加HTML标记，得到可作为网页的文档，让Web浏览器能够显示它。然而，创建基本引擎后，完全可以添加其他类型的标记（如各种形式的XML和LATEX编码）。对文本文件进行分析后，你甚至可以执行其他的任务，如提取所有的标题以制作目录。

注意 LATEX是一种用于创建各种技术文档的标记系统，基于TEX排版程序。这里提到它只是想说明所要创建程序的其他用途。要深入了解LATEX，可访问TEX用户组网站（http://www.tug.org）。

你拿到的文本可能包含一些线索（突出的文本形如*like this*），但要让程序能够猜测出文档的结构，可能需要一些技巧。

着手编写原型前，先来定义一些目标。

- 输入无需包含人工编码或标签。

- 程序需要能够处理不同的文本块（如标题、段落和列表项）以及内嵌文本（如突出的文本和URL）。
- 虽然这个实现添加的是HTML标签，但应该很容易对其进行扩展，以支持其他标记语言。

在程序的第一个版本中，可能无法实现所有这些目标，但这正是原型的意义所在。你编写原型旨在找出最初的想法存在的缺陷以及学习如何编写程序来解决面临的问题。

提示　在可能的情况下，最好逐渐修改最初的程序，而不要推倒重来。为清晰起见，我将提供两个完全独立的程序版本。

20.2　有用的工具

想想编写这个程序需要哪些工具。

- 肯定需要读写文件（参见第11章），至少要从标准输入（`sys.stdin`）读取以及使用`print`进行输出。
- 可能需要迭代输入行（参见第11章）
- 需要使用一些字符串方法（参见第3章）。
- 可能用到一两个生成器（参见第9章）。
- 可能需要模块`re`（参见第10章）。
- 如果你不熟悉上述任何概念，请花点时间复习一下。

20.3　准备工作

开始编码前，还需要有评估进度的途径，为此需要一个测试套件。就这个项目而言，一个测试就足够了：一个（纯文本）测试**文档**。代码清单20-1是你要对其进行自动标记的示例文本。

代码清单20-1　一个纯文本文档（test_input.txt）

```
Welcome to World Wide Spam, Inc.

These are the corporate web pages of *World Wide Spam*, Inc. We hope
you find your stay enjoyable, and that you will sample many of our
products.

A short history of the company

World Wide Spam was started in the summer of 2000. The business
concept was to ride the dot-com wave and to make money both through
bulk email and by selling canned meat online.

After receiving several complaints from customers who weren't
satisfied by their bulk email, World Wide Spam altered their profile,
and focused 100% on canned goods. Today, they rank as the world's
```

```
13,892nd online supplier of SPAM.

Destinations

From this page you may visit several of our interesting web pages:

  - What is SPAM? (http://wwspam.fu/whatisspam)

  - How do they make it? (http://wwspam.fu/howtomakeit)

  - Why should I eat it? (http://wwspam.fu/whyeatit)

How to get in touch with us

You can get in touch with us in *many* ways: By phone (555-1234), by
email (wwspam@wwspam.fu) or by visiting our customer feedback page
(http://wwspam.fu/feedback).
```


要对实现进行测试，只需将这个文档作为输入，并在Web浏览器中查看结果（或直接检查添加的标签）即可。

注意 相比于人工检查结果，使用自动测试套件通常是更佳的选择。（你能想出让测试自动化的方法吗？）

20.4 初次实现

首先要做的事情之一是将文本分成段落。从代码清单20-1可知，段落之间有一个或多个空行。比**段落**更准确的说法是**块**（block），因为块也可以指标题和列表项。

20.4.1 找出文本块

要找出这些文本块，一种简单的方法是，收集空行前的所有行并将它们返回，然后重复这样的操作。不需要收集空行，因此不需要返回空文本块（即多个空行）。另外，必须确保文件的最后一行为空行，否则无法确定最后一个文本块到哪里结束。（当然，有其他确定这一点的方法。）

代码清单20-2演示了这种方法的一种实现。

代码清单20-2 一个文本块生成器（util.py）

```
def lines(file):
    for line in file: yield line
    yield '\n'

def blocks(file):
    block = []
    for line in lines(file):
        if line.strip():
```

```
            block.append(line)
        elif block:
            yield ''.join(block).strip()
            block = []
```

生成器lines是个简单的工具，在文件末尾添加一个空行。生成器blocks实现了刚才描述的方法。生成文本块时，将其包含的所有行合并，并将两端多余的空白（如列表项缩进和换行符）删除，得到一个表示文本块的字符串。（如果不喜欢这种找出段落的方法，你肯定能够设计出其他方法。请看看你最终能设计出多少种方法，这可能很有趣。）我将这些代码存储在文件util.py中，这意味着你稍后可在程序中导入这些生成器。

20.4.2　添加一些标记

使用代码清单20-2提供的基本功能，可创建简单的标记脚本。为此，可按如下基本步骤进行。

(1) 打印一些起始标记。

(2) 对于每个文本块，在段落标签内打印它。

(3) 打印一些结束标记。

这不太难，但用处也不大。这里假设要将第一个文本块放在一级标题标签（h1）内，而不是段落标签内。另外，还需将用星号括起的文本改成突出文本（使用标签em）。这样程序将更有用一些。由于已经编写好了函数blocks，使用re.sub实现这些需求的代码非常简单，如代码清单20-3所示。

代码清单20-3　一个简单的标记程序（simple_markup.py）

```
import sys, re
from util import *

print('<html><head><title>...</title><body>')

title = True
for block in blocks(sys.stdin):
    block = re.sub(r'\*(.+?)\*', r'<em>\1</em>', block)
    if title:
        print('<h1>')
        print(block)
        print('</h1>')
        title = False
    else:
        print('<p>')
        print(block)
        print('</p>')

print('</body></html>')
```

要执行这个程序，并将前面的示例文件作为输入，可像下面这样做：

```
$ python simple_markup.py < test_input.txt > test_output.html
```

这样，文件test_output.html将包含生成的HTML代码。图20-1是在Web浏览器中显示这些HTML代码的结果。

图20-1 初次尝试生成的网页

这个原型虽然不是很出色，但确实执行了一些重要任务。它将文本分成可独立处理的文本块，再依次对每个文本块应用一个过滤器（这个过滤器是通过调用re.sub实现的）。这种方法看起来不错，可在最终的程序中使用。

如果要扩展这个原型，该如何办呢？可在for循环中添加检查，以确定文本块是否是标题、列表项等。为此，需要添加其他的正则表达式，代码可能很快变得很乱。更重要的是，要让程序输出其他格式的代码（而不是HTML）很难，但是这个项目的目标之一就是能够轻松地添加其他输出格式。这里假设你要重构这个程序，以采用稍微不同的结构。

20.5 再次实现

你从初次实现中学到了什么呢？为了提高可扩展性，需提高程序的**模块化**程度（将功能放在独立的组件中）。要提高模块化程度，方法之一是采用面向对象设计（参见第7章）。你需要找出一些抽象，让程序在变得复杂时也易于管理。下面先来列出一些潜在的组件。

- **解析器**：添加一个读取文本并管理其他类的对象。
- **规则**：对于每种文本块，都制定一条相应的规则。这些规则能够检测不同类型的文本块并相应地设置其格式。
- **过滤器**：使用正则表达式来处理内嵌元素。
- **处理程序**：供解析器用来生成输出。每个处理程序都生成不同的标记。

这里的设计虽然不太详尽，但至少让你知道应如何将代码分成不同的部分，并让每部分都易于管理。

20.5.1　处理程序

先来看处理程序。处理程序负责生成带标记的文本，并从解析器那里接受详细指令。假设对于每种文本块，它都提供两个处理方法：一个用于添加起始标签，另一个用于添加结束标签。例如，它可能包含用于处理段落的方法start_paragraph和end_paragraph。生成HTML代码时，可像下面这样实现这些方法：

```
class HTMLRenderer:
    def start_paragraph(self):
        print('<p>')
    def end_paragraph(self):
        print('</p>')
```

当然，对于其他类型的文本块，需要提供类似的处理方法。（HTMLRenderer类的完整代码见稍后的代码清单20-4。）这好像足够灵活了：要添加其他类型的标记，只需再创建相应的处理程序（或渲染程序），并在其中包含添加相应起始标签和结束标签的方法。

注意　这里之所以使用术语**处理程序**（而不是**渲染程序**等），旨在指出它负责处理解析器生成的方法调用（参见20.5.2节），而不必像HTMLRenderer那样使用标记语言来渲染文本。XML解析方案SAX也使用了类似的处理程序机制，这将在第22章介绍。

如何处理正则表达式呢？你可能还记得，函数re.sub可通过第二个参数接受一个函数（替换函数）。这样将对匹配的对象调用这个函数，并将其返回值插入文本中。这与前面讨论的处理程序理念很匹配——你只需让处理程序实现替换函数即可。例如，可像下面这样处理要突出的内容：

```
def sub_emphasis(self, match):
    return '<em>{}</em>'.format(match.group(1))
```

如果你不知道方法group是做什么的，应复习一下第10章介绍的模块re。

除start、end和sub方法外，还有一个名为feed的方法，用于向处理程序提供实际文本。在简单的HTML渲染程序中，只需像下面这样实现这个方法：

```
def feed(self, data):
    print(data)
```

20.5.2　处理程序的超类

为提高灵活性，我们来添加一个Handler类，它将是所有处理程序的超类，负责处理一些管理性细节。在有些情况下，不通过全名调用方法（如start_paragraph），而是使用字符串表示文本块的类型（如'paragraph'）并将这样的字符串提供给处理程序将很有用。为此，可添加一些通用方法，如start(type)、end(type)和sub(type)。另外，还可让通用方法start、end和sub检查是否实现了相应的方法（例如，start('paragraph')检查是否实现了start_paragraph）。如果没有实现，就什么都不做。这个Handler类的实现如下（摘自代码清单20-4所示的模块handlers）：

```
class Handler:
    def callback(self, prefix, name, *args):
        method = getattr(self, prefix + name, None)
        if callable(method): return method(*args)
    def start(self, name):
        self.callback('start_', name)
    def end(self, name):
        self.callback('end_', name)
    def sub(self, name):
        def substitution(match):
            result = self.callback('sub_', name, match)
            if result is None: match.group(0)
            return result
        return substitution
```

对于这些代码，有几点需要说明。

- 方法callback负责根据指定的前缀（如'start_'）和名称（如'paragraph'）查找相应的方法。这是通过使用getattr并将默认值设置为None实现的。如果getattr返回的对象是可调用的，就使用额外提供的参数调用它。例如，调用handler.callback('start_', 'paragraph')时，将调用方法handler.start_paragraph且不提供任何参数——如果start_paragraph存在的话。
- 方法start和end都是辅助方法，它们分别使用前缀start_和end_调用callback。
- 方法sub稍有不同。它不直接调用callback，而是返回一个函数，这个函数将作为替换函数传递给re.sub（这就是它只接受一个匹配对象作为参数的原因所在）。

下面来看一个示例。假设HTMLRenderer是Handler的子类，并像前一节介绍的那样实现了方法sub_emphasis（有关handlers.py的实际代码，请参阅代码清单20-4）。现在假设变量handler存储着一个HTMLRenderer实例。

```
>>> from handlers import HTMLRenderer
>>> handler = HTMLRenderer()
```

在这种情况下，调用handler.sub('emphasis')的结果将如何呢？

```
>>> handler.sub('emphasis')
<function substitution at 0x168cf8>
```

将返回一个函数（substitution）。如果你调用这个函数，它将调用方法handler.sub_emphasis。这意味着可在re.sub语句中使用这个函数：

```
>>> import re
>>> re.sub(r'\*(.+?)\*', handler.sub('emphasis'), 'This *is* a test')
'This <em>is</em> a test'
```

太神奇了！（这里的正则表达式与用星号括起的文本匹配，将在稍后讨论。）但为何要这么绕呢？为何不像初次实现中那样使用r'\1'呢？因为如果这样做，就只能添加em标签，但你希望处理程序能够根据情况添加不同的标签。例如，如果处理程序为（虚构的）LaTeXRenderer，应生成完全不同的结果。

```
>> re.sub(r'\*(.+?)\*', handler.sub('emphasis'), 'This *is* a test')
'This \\emph{is} a test'
```

代码还是原来的代码，但添加的标签不同了。

我们还提供了备用方案，以应对没有实现替换函数的情形。方法callback查找方法sub_something，但如果没有找到，就返回None。由于要返回一个用于re.sub中的替换函数，因此你不想返回None。相反，如果没有找到替换函数，就原样返回匹配对象。换而言之，如果callback返回None，在sub中定义的substitution将返回匹配的文本，即match.group(0)。

20.5.3　规则

至此，处理程序的可扩展性和灵活性都非常高了，该将注意力转向解析（对文本进行解读）了。为此，我们将规则定义为独立的对象，而不像初次实现中那样使用一条包含各种条件和操作的大型if语句。

规则是供主程序（解析器）使用的。主程序必须根据给定的文本块选择合适的规则来对其进行必要的转换。换而言之，规则必须具备如下功能。

- 知道自己适用于那种文本块（**条件**）。
- 对文本块进行转换（**操作**）。

因此每个规则对象都必须包含两个方法：condition和action。

方法condition只需要一个参数：待处理的文本块。它返回一个布尔值，指出当前规则是否适用于处理指定的文本块。

提示　要实现复杂的解析规则，可能需要让规则对象能够访问一些状态变量，从而让它知道之前发生的情况或已应用了哪些规则。

方法action也将当前文本块作为参数，但为了影响输出，它还必须能够访问处理器对象。

在很多情况下，适用的规则可能只有一个。换而言之，发现使用了标题规则（这表明当前文本块为标题）后，就不应再试图使用段落规则。为实现这一点，一种简单的方法是让解析器依次尝试每个规则，并在触发一个规则后不再接着尝试。这样做通常很好，但在有些情况下，应用一个规则后还可应用其他规则。有鉴于此，需要给方法action再添加一项功能：让它返回一个布尔值，指出是否就此结束对当前文本块的处理。（也可使用异常来实现这项功能，这种异常类似于迭代器的StopIteration机制。）

标题规则的伪代码可能类似于：

```
class HeadlineRule:
    def condition(self, block):
        如果文本块符合标题的定义，就返回True；
        否则返回False。
    def action(self, block, handler):
        调用诸如handler.start('headline')、handler.feed(block)
        和handler.end('headline')等方法。
        我们不想尝试其他规则，因此返回True，以结束对当前文本块的处理。
```

20.5.4 规则的超类

虽然并非一定要提供规则超类，但多个规则可能执行相同的操作：调用处理程序的方法start、feed和end，并将相应的类型字符串作为参数，再返回True（以结束对当前文本块的处理）。假设所有的规则子类都有一个type属性，其中包含类型字符串，则可像下面这样实现规则超类。（Rule类包含在模块rules中，这个模块的完整代码见代码清单20-5。）

```
class Rule:
    def action(self, block, handler):
        handler.start(self.type)
        handler.feed(block)
        handler.end(self.type)
        return True
```

方法condition由各个子类负责实现。Rule类及其子类都放在模块rules中。

20.5.5 过滤器

你无需实现独立的过滤器类。由于Handler类包含方法sub，每个过滤器都可用一个正则表达式和一个名称（如emphasis或url）来表示。下一节介绍如何处理解析器时，你将看到这是如何实现的。

20.5.6 解析器

现在来讨论应用程序的核心部分：Parser类。它使用一个处理程序以及一系列规则和过滤器将纯文本文件转换为带标记的文件（这里是HTML文件）。这个类需要包含哪些方法呢？完成准备工作的构造函数、添加规则的方法、添加过滤器的方法以及对文件进行解析的方法。

下面是Parser类的代码（摘自代码清单20-6，这个代码清单详细列出了markup.py的代码）：

```
class Parser:
    """
    读取文本文件、应用规则并控制处理程序的解析器
    """
    def __init__ (self, handler):
        self.handler = handler
        self.rules = []
        self.filters = []
    def addRule(self, rule):
        self.rules.append(rule)
    def addFilter(self, pattern, name):
        def filter(block, handler):
            return re.sub(pattern, handler.sub(name), block)
        self.filters.append(filter)
    def parse(self, file):
        self.handler.start('document')
        for block in blocks(file):
            for filter in self.filters:
                block = filter(block, self.handler)
```

```
            for rule in self.rules:
                if rule.condition(block):
                    last = rule.action(block, self.handler)
                    if last: break
        self.handler.end('document')
```

虽然这个类中需要理解的内容有很多，但大都不太复杂。构造函数将提供的处理程序赋给一个实例变量（属性），再初始化两个列表：一个规则列表和一个过滤器列表。方法addRule在规则列表中添加一个规则。然而，方法addFilter所做的工作更多：与方法addRule类似，它在过滤器列表中添加一个过滤器，但在此之前还要先创建过滤器。过滤器就是一个函数，它调用re.sub并将参数指定为合适的正则表达式（模式）和处理程序中的替换函数（handler.sub(name)）。

方法parse虽然看起来有点复杂，但可能是最容易实现的，因为它只是完成一直计划要完成的任务。它以调用处理程序的方法start('document')开头，并以调用处理程序的方法end('document')结束。在这两个调用之间，它迭代文本文件中的所有文本块。对于每个文本块，它都应用过滤器和规则。应用过滤器就是调用函数filter，并以文本块和处理程序作为参数，再将结果赋给变量block，如下所示：

```
block = filter(block, self.handler)
```

这能让每个过滤器都完成其任务，即将部分文本替换为带标记的文本（如将*this*替换为this）。

遍历规则时涉及的逻辑要多些。对于每个规则，都使用一条if语句来检查它是否适用——这是通过调用rule.condition(block)实现的。如果规则适用，就调用rule.action，并将文本块和处理程序作为参数。前面说过，方法action返回一个布尔值，指出是否就此结束对当前文本块的处理。为结束对文本块的处理，将方法action的返回值赋给变量last，再在这个变量为True时退出for循环。

```
if last: break
```

注意　可将这两条语句压缩成一条，以避免使用变量last。

```
if rule.action(block, self.handler): break
```

是否这样做在很大程度上取决于你的偏好。避免使用临时变量可让代码更简单，但使用临时变量可清晰地标识返回值。

20.5.7　创建规则和过滤器

至此，万事俱备，只欠东风——还没有创建具体的规则和过滤器。到目前为止你编写的大部分代码都旨在让规则和过滤器与处理程序一样灵活。你可编写多个独立的规则和过滤器，再使用方法addRule和addFilter将它们添加到解析器中，同时确保在处理程序中实现了相应的方法。

通过使用一组复杂的规则，可处理复杂的文档，但我们将保持尽可能简单。只创建分别用于

处理题目、其他标题和列表项的规则。应将相连的列表项视为一个列表，因此还将创建一个处理整个列表的列表规则。最后，可创建一个默认规则，用于处理段落，即其他规则未处理的所有文本块。

下面以不太正式的方式定义了这些规则。

- 标题是只包含一行的文本块，长度最多为70个字符。以冒号结束的文本块不属于标题。
- 题目是文档中的第一个文本块，前提条件是它属于标题。
- 列表项是以连字符（-）打头的文本块。
- 列表以紧跟在非列表项文本块后面的列表项开头，以后面紧跟着非列表项文本块的列表项结束。

这些规则是根据我对文本文档结构的直觉制定的，你对文本文档结构的看法可能不同。另外，这些规则存在一些缺陷。例如，如果文档以列表项结尾怎么办？你完全可以改进这些规则。定义这些规则的完整源代码见后面的代码清单20-5（rules.py，这个文件还包含Rule类）。首先来定义标题规则：


```
class HeadingRule(Rule):
    """
    标题只包含一行，不超过70个字符且不以冒号结尾
    """
    type = 'heading'
    def condition(self, block):
        return not '\n' in block and len(block) <= 70 and not block[-1] == ':'
```

这里将属性type设置成了字符串'heading'，这个属性是供从Rule类继承而来的方法action使用的。方法condition核实文本块不包含换行符（\n）、长度不超过70且最后一个字符不是冒号。

题目规则与此类似，但只使用一次——用于处理第一个文本块。从此以后，它将忽略所有的文本块，因为其first属性已设置为False。

```
class TitleRule(HeadingRule):
    """
    题目是文档中的第一个文本块，前提条件是它属于标题
    """
    type = 'title'
    first = True

    def condition(self, block):
        if not self.first: return False
        self.first = False
        return HeadingRule.condition(self, block)
```

列表项规则的方法condition是根据前面的定义直接实现的。

```
class ListItemRule(Rule):
    """
    列表项是以连字符打头的段落。在设置格式的过程中，将把连字符删除
    """
    type = 'listitem'
    def condition(self, block):
        return block[0] == '-'
```

```
    def action(self, block, handler):
        handler.start(self.type)
        handler.feed(block[1:].strip())
        handler.end(self.type)
        return True
```

它重新实现了方法action。相比于Rule的方法action，这个方法唯一的不同之处在于，它删除了文本块中的第一个字符（连字符），并删除了余下文本中多余的空白。标记会生成列表项目符号，因此不再需要连字符。

到目前为止，所有规则的action方法都返回True。列表规则的action方法不能这样，因为它在遇到非列表项后面的列表项或列表项后面的非列表项时触发。由于它不实际标记这些文本块，而只是标记列表（一组列表项）的开始和结束位置，因此你不希望对文本块的处理到此结束，从而要让它返回False。

```
class ListRule(ListItemRule):
    """
    列表以紧跟在非列表项文本块后面的
    列表项开头，以相连的最后一个列表
    项结束
    """
    type = 'list'
    inside = False
    def condition(self, block):
        return True
    def action(self, block, handler):
        if not self.inside and ListItemRule.condition(self, block):
            handler.start(self.type)
            self.inside = True
        elif self.inside and not ListItemRule.condition(self, block):
            handler.end(self.type)
            self.inside = False
        return False
```

对于这个列表规则，可能需要做进一步的解释。它的方法condition总是返回True，因为你要检查所有的文本块。在方法action中，需要处理两种不同的情况。

如果属性inside（指出当前是否位于列表内）为False（初始值），且列表项规则的方法condition返回True，就说明刚进入列表中。因此调用处理程序的start方法，并将属性inside设置为True。

相反，如果属性inside为True，且列表项规则的方法condition返回False，就说明刚离开列表。因此调用处理程序的end方法，并将属性inside设置为False。

完成这些处理后，这个方法返回False，以继续根据其他规则对文本块进行处理。（当然，这意味着规则的排列顺序至关重要。）

最后一个规则是ParagraphRule，其方法condition总是返回True，因为这是默认使用的规则。这个规则是加入规则列表中的最后一个元素，对其他规则未处理的所有文本块进行处理。

```
class ParagraphRule(Rule):
    """
```

```
        段落是不符合其他规则的文本块
        """
        type = 'paragraph'
        def condition(self, block):
            return True
```

过滤器就是正则表达式。我们来添加三个过滤器，分别用来找出要突出的内容、URL和Email地址。为此，我们使用下面三个正则表达式：

```
r'\*(.+?)\*'
r'(http://[\.a-zA-Z/]+)'
r'([\.a-zA-Z]+@[\.a-zA-Z]+[a-zA-Z]+)'
```

第一个模式找出要突出的内容，它与用两个星号括起的内容匹配（它要匹配尽可能少的内容，因此使用了问号）。第二个模式找出URL，它与这样的内容匹配：字符串'http://'（你可在这里添加其他协议）后跟一个或多个句点、字母或斜杠。（这个模式并不能与所有合法的URL匹配，你可对其进行改进。）最后，Email模式与这样的内容匹配：中间为@，@前面为字母和句点组成的序列，@后面也是字母和句点组成的序列，最后为字母组成的序列，从而不与以句点结束的内容匹配。（同样，你可对这个模式进行改进。）

20.5.8 整合起来

现在，只需创建一个Parser对象，并添加相关的规则和过滤器。下面就来这样做：创建一个在构造函数中完成初始化的Parser子类，再使用它来解析sys.stdin。

最终的程序如代码清单20-4~代码清单20-6所示（这些代码清单依赖于代码清单20-2所示的工具代码）。可以像运行原型那样运行最终的程序。

```
$ python markup.py < test_input.txt > test_output.html
```

代码清单20-4 处理程序（handlers.py）

```
class Handler:
    """
    对Parser发起的方法调用进行处理的对象

    Parser将对每个文本块调用方法start()和end()，并将合适
    的文本块名称作为参数。方法sub()将用于正则表达式替换，
    使用诸如'emphasis'等名称调用时，这个方法将返回相应的
    替换函数
    """
    def callback(self, prefix, name, *args):
        method = getattr(self, prefix + name, None)
        if callable(method): return method(*args)
    def start(self, name):
        self.callback('start_', name)
    def end(self, name):
        self.callback('end_', name)
    def sub(self, name):
        def substitution(match):
```

```
            result = self.callback('sub_', name, match)
            if result is None: match.group(0)
            return result
        return substitution

class HTMLRenderer(Handler):
    """
    用于渲染HTML的具体处理程序

    HTMLRenderer的方法可通过超类Handler的方法
    start()、end()和sub()来访问。这些方法实现了
    HTML文档使用的基本标记
    """
    def start_document(self):
        print('<html><head><title>...</title></head><body>')
    def end_document(self):
        print('</body></html>')
    def start_paragraph(self):
        print('<p>')
    def end_paragraph(self):
        print('</p>')
    def start_heading(self):
        print('<h2>')
    def end_heading(self):
        print('</h2>')
    def start_list(self):
        print('<ul>')
    def end_list(self):
        print('</ul>')
    def start_listitem(self):
        print('<li>')
    def end_listitem(self):
        print('</li>')
    def start_title(self):
        print('<h1>')
    def end_title(self):
        print('</h1>')
    def sub_emphasis(self, match):
        return '<em>{}</em>'.format(match.group(1))
    def sub_url(self, match):
        return '<a href="{}">{}</a>'.format(match.group(1), match.group(1))
    def sub_mail(self, match):
        return '<a href="mailto:{}">{}</a>'.format(match.group(1), match.group(1))
    def feed(self, data):
        print(data)
```

代码清单20-5　规则（rules.py）

```
class Rule:
    """
    所有规则的基类
    """
    def action(self, block, handler):
        handler.start(self.type)
```

```
        handler.feed(block)
        handler.end(self.type)
        return True

class HeadingRule(Rule):
    """
    标题只包含一行，不超过70个字符且不以冒号结尾
    """
    type = 'heading'
    def condition(self, block):
        return not '\n' in block and len(block) <= 70 and not block[-1] == ':'

class TitleRule(HeadingRule):
    """
    题目是文档中的第一个文本块，前提条件是它属于标题
    """
    type = 'title'
    first = True

    def condition(self, block):
        if not self.first: return False
        self.first = False
        return HeadingRule.condition(self, block)

class ListItemRule(Rule):
    """
    列表项是以连字符打头的段落。在设置格式的过程中，将把连字符删除
    """
    type = 'listitem'
    def condition(self, block):
        return block[0] == '-'
    def action(self, block, handler):
        handler.start(self.type)
        handler.feed(block[1:].strip())
        handler.end(self.type)
        return True

class ListRule(ListItemRule):
    """
    列表以紧跟在非列表项文本块后面的列表项打头，以相连的最后一个列表项结束
    """
    type = 'list'
    inside = False
    def condition(self, block):
        return True
    def action(self, block, handler):
        if not self.inside and ListItemRule.condition(self, block):
            handler.start(self.type)
            self.inside = True
        elif self.inside and not ListItemRule.condition(self, block):
            handler.end(self.type)
            self.inside = False
        return False
```

```
class ParagraphRule(Rule):
    """
    段落是不符合其他规则的文本块
    """
    type = 'paragraph'
    def condition(self, block):
        return True
```

代码清单20-6　主程序（markup.py）

```
import sys, re
from handlers import *
from util import *
from rules import *

class Parser:
    """
    Parser读取文本文件，应用规则并控制处理程序
    """
    def __init__(self, handler):
        self.handler = handler
        self.rules = []
        self.filters = []
    def addRule(self, rule):
        self.rules.append(rule)
    def addFilter(self, pattern, name):
        def filter(block, handler):
            return re.sub(pattern, handler.sub(name), block)
        self.filters.append(filter)

def parse(self, file):
    self.handler.start('document')
    for block in blocks(file):
        for filter in self.filters:
            block = filter(block, self.handler)
            for rule in self.rules:
                if rule.condition(block):
                    last = rule.action(block,
                           self.handler)
                    if last: break
    self.handler.end('document')

class BasicTextParser(Parser):
    """
    在构造函数中添加规则和过滤器的Parser子类
    """
    def __init__(self, handler):
        Parser.__init__(self, handler)
        self.addRule(ListRule())
        self.addRule(ListItemRule())
        self.addRule(TitleRule())
        self.addRule(HeadingRule())
        self.addRule(ParagraphRule())

        self.addFilter(r'\*(.+?)\*', 'emphasis')
        self.addFilter(r'(http://[\.a-zA-Z/]+)', 'url')
```

```
        self.addFilter(r'([\.a-zA-Z]+@[\.a-zA-Z]+[a-zA-Z]+)', 'mail')
handler = HTMLRenderer()
parser = BasicTextParser(handler)

parser.parse(sys.stdin)
```

将前面的示例文本作为输入时，这个程序的运行结果如图20-2所示。

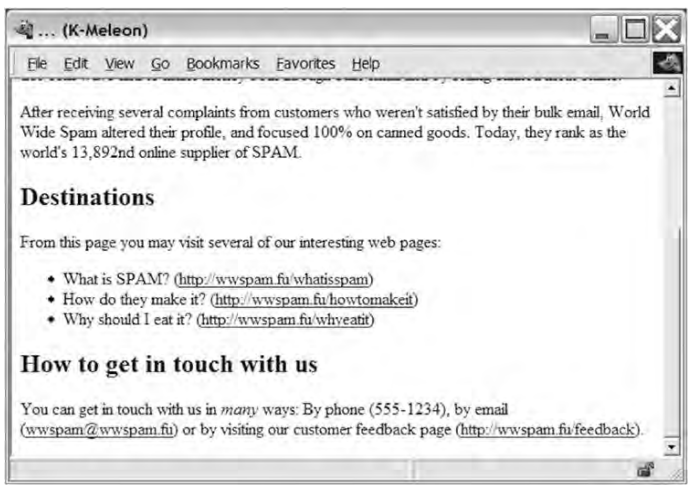

图20-2 再次尝试生成的网页

相比初次实现，再次实现显然更复杂，涉及范围更广。值得花精力去实现这样的复杂性，因为创建出的程序更灵活、可扩展性更强。要对其进行修改，以支持其他的输入和输出格式，只需派生出子类并初始化既有的类，而不像原型那样需要推倒重来。

20.6 进一步探索

这个程序存在如下潜在的扩展空间。

- 增加对表格的支持。为此，只需找出左对齐内容的边界，并将文本块分成多列。
- 突出全部大写的单词。为此，需要考虑缩略语、标点、姓名和其他首字母大写的单词。
- 支持LATEX格式的输出。
- 编写一个执行其他处理（而不是添加标记）的处理程序，如以某种方式对文档进行分析。
- 创建一个脚本，将特定目录中的所有文本文件都自动转换为HTML文件。
- 了解其他纯文本格式，如Markdown、reStructuredText或维基百科使用的格式。

预告

为完成这个可能很有用的项目，我们费了九牛二虎之力，该介绍点轻松的内容了。下一章将根据从网上自动下载的数据创建一些图表，这易如反掌。