# XML解析

XML是可扩展标记语言（Extensible Markup Language）的缩写，其中的 标记（markup）是关键部分。XML是用户通过自定义**标签对**来描述数据的，XML是W3C的推荐标准。

## xml文件

<?xml version="1.0"?>

<doc>

  <branch name="codingpy.com" hash="1cdf045c">

    text,source

  </branch>

  <branch name="release01" hash="f200013e">

    <sub-branch name="subrelease01">

      xml,sgml

    </sub-branch>

  </branch>

  <branch name="invalid">

  </branch>

</doc>

## 解析xml

### dom解析

DOM解析器在任何处理开始之前，必须把基于XML文件生成的树状数据放在内存，所以DOM解析器的内存使用量完全根据输入资料的大小。

xml.dom.minidom是DOM API的极简化实现，比完整版的DOM要简单的多，而且这个包也小的多。

### sax解析

xml.sax模块实现的是SAX API，这个模块牺牲了便捷性来换取速度和内存占用。SAX是Simple API for XML的缩写，它并不是由W3C官方所提出的标准。它是事件驱动的，并不需要一次性读入整个文档，而文档的读入过程也就是SAX的解析过程。所谓事件驱动，是指一种基于回调（callback）机制的程序运行方法。

### ElementTree解析

xml.etree.ElementTree模块提供了一个轻量级、Pythonic的API，同时还有一个高效的C语言实现，即xml.etree.cElementTree。自Python 3.3之后，ElemenTree模块会**自动优先使用C加速器**，如果不存在C实现，则会使用Python实现。

与DOM相比，ET的速度更快，API使用更直接、方便。与SAX相比，ET.iterparse函数同样提供了按需解析的功能，不会一次性在内存中读入整个文档。ET的性能与SAX模块大致相仿，但是它的API更加高层次，用户使用起来更加便捷。

建议：**在使用Python进行XML解析时，首选使用ET模块**

（1）、将XML文档解析为树（tree）

**import** xml.etree.ElementTree **as** ET  
  
tree = ET.ElementTree(file=**'doc.xml'**)  
root = tree.getroot()  
print(root.tag)  
print(root.attrib)

（2）、通过索引值来访问特定的子元素：

# 读取节点的内容nodeFirst = root[0]  
print(nodeFirst.text)  
  
*#读取节点的属性*print(nodeFirst.attrib)  
  
*#读取节点的指定属性的值*print(nodeFirst.attrib[**'name'**])

（3）、遍历子节点：

*#遍历子节点*print(**'0-------------------'**)  
**for** child\_of\_root **in** root:  
 print(child\_of\_root.tag, child\_of\_root.attrib, child\_of\_root.text)

Element对象有一个iter方法，可以对某个元素对象之下所有的子元素进行深度优先遍历（DFS）。ElementTree对象同样也有这个方法。

*#遍历节点和所有子节点（包括子节点的子节点），也可以for child\_of\_root in tree.iter():*print(**'1-------------------'**)  
**for** child\_of\_root **in** root.iter():  
 print(child\_of\_root.tag, child\_of\_root.attrib, child\_of\_root.text)

（4）、查找需要的元素

iter方法还可以接受tag名称，然后遍历所有具备所提供tag的元素：

*#过滤不符合条件的节点*print(**'2-------------------'**)  
**for** elem **in** tree.iter(tag=**'branch'**):  
 print(elem.tag, elem.attrib, elem.text)  
  
  
  
*# 按某个属性来查找*print(**'4-------------------'**)  
**for** elem **in** tree.iterfind(**'branch[@name="release01"]'**):  
 print(elem.tag, elem.attrib, elem.text)

（5）、支持通过XPath查找元素

# 按xpath路径形式访问  
print('3-------------------')  
for elem in tree.iterfind('branch/sub-branch'):  
 print(elem.tag, elem.attrib, elem.text)  
myNode = tree.iterfind('branch/sub-branch')  
print(type(myNode))  
print(next(myNode).text)  
  
myNode2 = tree.find('branch/sub-branch')  
print(myNode2.text)

使用XPath查找感兴趣的元素，更加方便。Element对象中有一些find方法可以接受Xpath路径作为参数，find方法会返回第一个匹配的子元素，findall以列表的形式返回所有匹配的子元素, iterfind则返回一个所有匹配元素的迭代器（iterator）。

查找所有具备某个name属性的branch元素：

*# 按某个属性来查找*print(**'4-------------------'**)  
**for** elem **in** tree.iterfind(**'branch[@name="release01"]'**):  
 print(elem.tag, elem.attrib, elem.text)

（6）、构建XML文档

ElementTree对象的write方法就可以实现这个需求。

修改文档：modifyXml.py

**import** xml.etree.ElementTree **as** ET  
  
tree = ET.ElementTree(file=**'doc.xml'**)  
  
root = tree.getroot()  
**del** root[2]  
root[0].set(**'foo'**, **'bar'**)  
tree.write(**"doc2.xml"**, encoding=**"utf-8"**)

在上面的代码中，我们删除了root元素的第三个子元素，为第一个子元素增加了新属性。这个树可以重新写入至文件中。

注意，文档中元素的属性顺序与原文档不同。这是因为ET是以字典的形式保存属性的，而字典是一个无序的数据结构。XML不关注属性的顺序，所以这个不是问题。

创建新的XML文档：createXml.py

ET模块提供了一个SubElement工厂函数：

**import** xml.etree.ElementTree **as** ET  
**import** sys  
  
a = ET.Element(**'elem'**)  
c = ET.SubElement(a, **'child1'**)  
c.text = **"some text"**d = ET.SubElement(a, **'child2'**)  
b = ET.Element(**'elem\_b'**)  
root = ET.Element(**'root'**)  
root.extend((a, b))  
tree = ET.ElementTree(root)  
tree.write(**"new.xml"**, encoding=**"UTF-8"**, xml\_declaration=**True**)

5、利用iterparse解析XML流

ET可以将XML文档加载为保存在内存里的树（in-memory tree），然后再进行处理。但是在解析大文件时，这应该也会出现和DOM一样的内存消耗大的问题。为了解决这个问题，ET提供了一个类似SAX的特殊工具——iterparse，可以循序地解析XML。

统计一下文档中出现了**branch节点的个数**。

用ET.parse的标准方法：parse.py

**import** xml.etree.ElementTree **as** ET

tree = ET.parse(**'doc.xml'**)

count = 0

for elem in tree.iter(tag='**branch**'):

    count += 1

print(count)

上面的代码会将全部元素载入内存，逐一解析。当解析一个约100MB的XML文档时，运行上面脚本的Python进程的内存使用峰值为约560MB，总运行时间问2.9秒。

用iterparse方法：iterparse.py

**import** xml.etree.ElementTree **as** ET  
  
count = 0  
**for** event, elem **in** ET.iterparse(**'doc.xml'**):  
 **if** event == **'end'**:  
 **if** elem.tag == **'branch'**:  
 count += 1  
 elem.clear() *# 将元素废弃*print(count)

上面的for循环会遍历iterparse事件，首先检查事件是否为end，然后判断元素的tag是否为**'branch'**。调用elem.clear()非常关键：因为iterparse仍然会生成一个树，只是循序生成的而已。废弃掉不需要的元素，就相当于废弃了整个树，释放出系统分配的内存。