# 6.4 增量式解析大型XML文件¶

## 问题¶

你想使用尽可能少的内存从一个超大的XML文档中提取数据。

## 解决方案¶

任何时候只要你遇到增量式的数据处理时，第一时间就应该想到迭代器和生成器。 下面是一个很简单的函数，只使用很少的内存就能增量式的处理一个大型XML文件：

```
from xml.etree.ElementTree import iterparse

def parse_and_remove(filename, path):
    path_parts = path.split('/')
    doc = iterparse(filename, ('start', 'end'))
    # Skip the root element
    next(doc)

    tag_stack = []
    elem_stack = []
    for event, elem in doc:
        if event == 'start':
            tag_stack.append(elem.tag)
            elem_stack.append(elem)
        elif event == 'end':
            if tag_stack == path_parts:
                yield elem
                elem_stack[-2].remove(elem)
            try:
                tag_stack.pop()
                elem_stack.pop()
            except IndexError:
                pass
```

为了测试这个函数，你需要先有一个大型的XML文件。 通常你可以在政府网站或公共数据网站上找到这样的文件。 例如，你可以下载XML格式的芝加哥城市道路坑洼数据库。 在写这本书的时候，下载文件已经包含超过100,000行数据，编码格式类似于下面这样：

```
<response>
    <row>
        <row ...>
            <creation_date>2012-11-18T00:00:00</creation_date>
            <status>Completed</status>
            <completion_date>2012-11-18T00:00:00</completion_date>
            <service_request_number>12-01906549</service_request_number>
            <type_of_service_request>Pot Hole in Street</type_of_service_request>
            <current_activity>Final Outcome</current_activity>
            <most_recent_action>CDOT Street Cut ... Outcome</most_recent_action>
            <street_address>4714 S TALMAN AVE</street_address>
            <zip>60632</zip>
            <x_coordinate>1159494.68618856</x_coordinate>
            <y_coordinate>1873313.83503384</y_coordinate>
            <ward>14</ward>
            <police_district>9</police_district>
            <community_area>58</community_area>
            <latitude>41.808090232127896</latitude>
            <longitude>-87.69053684711305</longitude>
            <location latitude="41.808090232127896"
            longitude="-87.69053684711305" />
        </row>
        <row ...>
            <creation_date>2012-11-18T00:00:00</creation_date>
            <status>Completed</status>
            <completion_date>2012-11-18T00:00:00</completion_date>
            <service_request_number>12-01906695</service_request_number>
```

```
            <type_of_service_request>Pot Hole in Street</type_of_service_request>
            <current_activity>Final Outcome</current_activity>
            <most_recent_action>CDOT Street Cut ... Outcome</most_recent_action>
            <street_address>3510 W NORTH AVE</street_address>
            <zip>60647</zip>
            <x_coordinate>1152732.14127696</x_coordinate>
            <y_coordinate>1910409.38979075</y_coordinate>
            <ward>26</ward>
            <police_district>14</police_district>
            <community_area>23</community_area>
            <latitude>41.91002084292946</latitude>
            <longitude>-87.71435952353961</longitude>
            <location latitude="41.91002084292946"
            longitude="-87.71435952353961" />
        </row>
    </row>
</response>
```

假设你想写一个脚本来按照坑洼报告数量排列邮编号码。你可以像这样做：

```
from xml.etree.ElementTree import parse
from collections import Counter

potholes_by_zip = Counter()

doc = parse('potholes.xml')
for pothole in doc.iterfind('row/row'):
    potholes_by_zip[pothole.findtext('zip')] += 1
for zipcode, num in potholes_by_zip.most_common():
    print(zipcode, num)
```

这个脚本唯一的问题是它会先将整个XML文件加载到内存中然后解析。 在我的机器上，为了运行这个程序需要用到450MB左右的内存空间。 如果使用如下代码，程序只需要修改一点点：

```
from collections import Counter

potholes_by_zip = Counter()

data = parse_and_remove('potholes.xml', 'row/row')
for pothole in data:
    potholes_by_zip[pothole.findtext('zip')] += 1
for zipcode, num in potholes_by_zip.most_common():
    print(zipcode, num)
```

结果是：这个版本的代码运行时只需要7MB的内存–大大节约了内存资源。

## 讨论¶

这一节的技术会依赖 `ElementTree` 模块中的两个核心功能。 第一，`iterparse()` 方法允许对XML文档进行增量操作。 使用时，你需要提供文件名和一个包含下面一种或多种类型的事件列表： `start`，`end`，`start-ns` 和 `end-ns` 。 由 `iterparse()` 创建的迭代器会产生形如 `(event, elem)` 的元组， 其中 `event` 是上述事件列表中的某一个，而 `elem` 是相应的XML元素。例如：

```
>>> data = iterparse('potholes.xml',('start','end'))
>>> next(data)
('start', <Element 'response' at 0x100771d60>)
>>> next(data)
('start', <Element 'row' at 0x100771e68>)
>>> next(data)
('start', <Element 'row' at 0x100771fc8>)
>>> next(data)
('start', <Element 'creation_date' at 0x100771f18>)
>>> next(data)
('end', <Element 'creation_date' at 0x100771f18>)
>>> next(data)
('start', <Element 'status' at 0x1006a7f18>)
>>> next(data)
('end', <Element 'status' at 0x1006a7f18>)
```

```
>>>
```

start 事件在某个元素第一次被创建并且还没有被插入其他数据(如子元素)时被创建。 而 end 事件在某个元素已经完成时被创建。 尽管没有在例子中演示， start-ns 和 end-ns 事件被用来处理XML文档命名空间的声明。

这本节例子中， start 和 end 事件被用来管理元素和标签栈。 栈代表了文档被解析时的层次结构， 还被用来判断某个元素是否匹配传给函数 parse_and_remove() 的路径。 如果匹配，就利用 yield 语句向调用者返回这个元素。

在 yield 之后的下面这个语句才是使得程序占用极少内存的ElementTree的核心特性：

```
elem_stack[-2].remove(elem)
```

这个语句使得之前由 yield 产生的元素从它的父节点中删除掉。 假设已经没有其它的地方引用这个元素了，那么这个元素就被销毁并回收内存。

对节点的迭代式解析和删除的最终效果就是一个在文档上高效的增量式清扫过程。 文档树结构从始自终没被完整的创建过。尽管如此，还是能通过上述简单的方式来处理这个XML数据。

这种方案的主要缺陷就是它的运行性能了。 我自己测试的结果是，读取整个文档到内存中的版本的运行速度差不多是增量式处理版本的两倍快。 但是它却使用了超过后者60倍的内存。 因此，如果你更关心内存使用量的话，那么增量式的版本完胜。