8.23 循环引用数据结构的内存管理¶

问题¶

你的程序创建了很多循环引用数据结构(比如树、图、观察者模式等)，你碰到了内存管理难题。

解决方案¶

一个简单的循环引用数据结构例子就是一个树形结构，双亲节点有指针指向孩子节点，孩子节点又返回来指向双亲节点。 这种情况下，可以考虑使用 `weakref` 库中的弱引用。例如：

```
import weakref

class Node:
    def __init__(self, value):
        self.value = value
        self._parent = None
        self.children = []

    def __repr__(self):
        return 'Node({!r:})'.format(self.value)

    # property that manages the parent as a weak-reference
    @property
    def parent(self):
        return None if self._parent is None else self._parent()

    @parent.setter
    def parent(self, node):
        self._parent = weakref.ref(node)

    def add_child(self, child):
        self.children.append(child)
        child.parent = self
```

这种是想方式允许parent静默终止。例如：

```
>>> root = Node('parent')
>>> c1 = Node('child')
>>> root.add_child(c1)
>>> print(c1.parent)
Node('parent')
>>> del root
>>> print(c1.parent)
None
>>>
```

讨论¶

循环引用的数据结构在Python中是一个很棘手的问题，因为正常的垃圾回收机制不能适用于这种情形。 例如考虑如下代码：

```
# Class just to illustrate when deletion occurs
class Data:
```

```
    def __del__(self):
        print('Data.__del__')

# Node class involving a cycle
class Node:
    def __init__(self):
        self.data = Data()
        self.parent = None
        self.children = []

    def add_child(self, child):
        self.children.append(child)
        child.parent = self
```

下面我们使用这个代码来做一些垃圾回收试验：

```
>>> a = Data()
>>> del a # Immediately deleted
Data.__del__
>>> a = Node()
>>> del a # Immediately deleted
Data.__del__
>>> a = Node()
>>> a.add_child(Node())
>>> del a # Not deleted (no message)
>>>
```

可以看到，最后一个的删除时打印语句没有出现。原因是Python的垃圾回收机制是基于简单的引用计数。 当一个对象的引用数变成0的时候才会立即删除掉。而对于循环引用这个条件永远不会成立。 因此，在上面例子中最后部分，父节点和孩子节点互相拥有对方的引用，导致每个对象的引用计数都不可能变成0。

Python有另外的垃圾回收器来专门针对循环引用的，但是你永远不知道它什么时候会触发。 另外你还可以手动的触发它，但是代码看上去很挫：

```
>>> import gc
>>> gc.collect() # Force collection
Data.__del__
Data.__del__
>>>
```

如果循环引用的对象自己还定义了自己的 `__del__()` 方法，那么会让情况变得更糟糕。 假设你像下面这样给Node定义自己的 `__del__()` 方法：

```
# Node class involving a cycle
class Node:
    def __init__(self):
        self.data = Data()
        self.parent = None
        self.children = []

    def add_child(self, child):
        self.children.append(child)
        child.parent = self

    # NEVER DEFINE LIKE THIS.
    # Only here to illustrate pathological behavior
```

```
    # Only here to illustrate pathological behavior
    def __del__(self):
        del self.data
        del.parent
        del.children
```

这种情况下，垃圾回收永远都不会去回收这个对象的，还会导致内存泄露。 如果你试着去运行它会发现，`Data.__del__` 消息永远不会出现了,甚至在你强制内存回收时：

```
>>> a = Node()
>>> a.add_child(Node()
>>> del a # No message (not collected)
>>> import gc
>>> gc.collect() # No message (not collected)
>>>
```

弱引用消除了引用循环的这个问题，本质来讲，弱引用就是一个对象指针，它不会增加它的引用计数。 你可以通过 `weakref` 来创建弱引用。例如：

```
>>> import weakref
>>> a = Node()
>>> a_ref = weakref.ref(a)
>>> a_ref
<weakref at 0x100581f70; to 'Node' at 0x1005c5410>
>>>
```

为了访问弱引用所引用的对象，你可以像函数一样去调用它即可。如果那个对象还存在就会返回它，否则就返回一个None。 由于原始对象的引用计数没有增加，那么就可以去删除它了。例如;

```
>>> print(a_ref())
<__main__.Node object at 0x1005c5410>
>>> del a
Data.__del__
>>> print(a_ref())
None
>>>
```

通过这里演示的弱引用技术，你会发现不再有循环引用问题了，一旦某个节点不被使用了，垃圾回收器立即回收它。 你还能参考8.25小节关于弱引用的另外一个例子。