8.25 创建缓存实例¶

问题¶

在创建一个类的对象时，如果之前使用同样参数创建过这个对象， 你想返回它的缓存引用。

解决方案¶

这种通常是因为你希望相同参数创建的对象时单例的。 在很多库中都有实际的例子，比如 `logging` 模块，使用相同的名称创建的 `logger` 实例永远只有一个。例如：

```
>>> import logging
>>> a = logging.getLogger('foo')
>>> b = logging.getLogger('bar')
>>> a is b
False
>>> c = logging.getLogger('foo')
>>> a is c
True
>>>
```

为了达到这样的效果，你需要使用一个和类本身分开的工厂函数，例如：

```
# The class in question
class Spam:
    def __init__(self, name):
        self.name = name

# Caching support
import weakref
_spam_cache = weakref.WeakValueDictionary()
def get_spam(name):
    if name not in _spam_cache:
        s = Spam(name)
        _spam_cache[name] = s
    else:
        s = _spam_cache[name]
    return s
```

然后做一个测试，你会发现跟之前那个日志对象的创建行为是一致的：

```
>>> a = get_spam('foo')
>>> b = get_spam('bar')
>>> a is b
False
>>> c = get_spam('foo')
>>> a is c
True
>>>
```

讨论¶

编写一个工厂函数来修改普通的实例创建行为通常是一个比较简单的方法。 但是我们还能否找到更优雅的解决方案呢？

例如，你可能会考虑重新定义类的 `__new__()` 方法，就像下面这样：

```
# Note: This code doesn't quite work
import weakref

class Spam:
    _spam_cache = weakref.WeakValueDictionary()
    def __new__(cls, name):
        if name in cls._spam_cache:
```

```
            return cls._spam_cache[name]
        else:
            self = super().__new__(cls)
            cls._spam_cache[name] = self
            return self
    def __init__(self, name):
        print('Initializing Spam')
        self.name = name
```

初看起来好像可以达到预期效果，但是问题是 `__init__()` 每次都会被调用，不管这个实例是否被缓存了。例如：

```
>>> s = Spam('Dave')
Initializing Spam
>>> t = Spam('Dave')
Initializing Spam
>>> s is t
True
>>>
```

这个或许不是你想要的效果，因此这种方法并不可取。

上面我们使用到了弱引用计数，对于垃圾回收来讲是很有帮助的，关于这个我们在8.23小节已经讲过了。 当我们保持实例缓存时，你可能只想在程序中使用到它们时才保存。 一个 `WeakValueDictionary` 实例只会保存那些在其它地方还在被使用的实例。 否则的话，只要实例不再被使用了，它就从字典中被移除了。观察下下面的测试结果：

```
>>> a = get_spam('foo')
>>> b = get_spam('bar')
>>> c = get_spam('foo')
>>> list(_spam_cache)
['foo', 'bar']
>>> del a
>>> del c
>>> list(_spam_cache)
['bar']
>>> del b
>>> list(_spam_cache)
[]
>>>
```

对于大部分程序而已，这里代码已经够用了。不过还是有一些更高级的实现值得了解下。

首先是这里使用到了一个全局变量，并且工厂函数跟类放在一块。我们可以通过将缓存代码放到一个单独的缓存管理器中：

```
import weakref

class CachedSpamManager:
    def __init__(self):
        self._cache = weakref.WeakValueDictionary()

    def get_spam(self, name):
        if name not in self._cache:
            s = Spam(name)
            self._cache[name] = s
        else:
            s = self._cache[name]
        return s

    def clear(self):
            self._cache.clear()

class Spam:
    manager = CachedSpamManager()
    def __init__(self, name):
        self.name = name

def get_spam(name):
    return Spam.manager.get_spam(name)
```

这样的话代码更清晰，并且也更灵活，我们可以增加更多的缓存管理机制，只需要替代manager即可。

还有一点就是，我们暴露了类的实例化给用户，用户很容易去直接实例化这个类，而不是使用工厂方法，如：

```
>>> a = Spam('foo')
>>> b = Spam('foo')
>>> a is b
False
>>>
```

有几种方式可以防止用户这样做，第一个是将类的名字修改为以下划线(_)开头，提示用户别直接调用它。 第二种就是让这个类的 `__init__()` 方法抛出一个异常，让它不能被初始化：

```
class Spam:
    def __init__(self, *args, **kwargs):
        raise RuntimeError("Can't instantiate directly")

    # Alternate constructor
    @classmethod
    def _new(cls, name):
        self = cls.__new__(cls)
        self.name = name
```

然后修改缓存管理器代码，使用 `Spam._new()` 来创建实例，而不是直接调用 `Spam()` 构造函数：

```
# ------------------------最后的修正方案------------------------
class CachedSpamManager2:
    def __init__(self):
        self._cache = weakref.WeakValueDictionary()

    def get_spam(self, name):
        if name not in self._cache:
            temp = Spam3._new(name)  # Modified creation
            self._cache[name] = temp
        else:
            temp = self._cache[name]
        return temp

    def clear(self):
            self._cache.clear()

class Spam3:
    def __init__(self, *args, **kwargs):
        raise RuntimeError("Can't instantiate directly")

    # Alternate constructor
    @classmethod
    def _new(cls, name):
        self = cls.__new__(cls)
        self.name = name
        return self
```

最后这样的方案就已经足够好了。 缓存和其他构造模式还可以使用9.13小节中的元类实现的更优雅一点(使用了更高级的技术)。