第3章

语法最佳实践——类级

本章将聚焦于类的语法最佳实践。在此我们不打算涉及设计模式，因为将在第 14 章中讨论它们。本章将概述 Python 中用于操纵和改进类代码的高级语法。尽管 Python 对象的一些细微之处仍然在发展，但是 2.x 系列版本中的基本方法仍然介绍了一些用于充分理解类的工作方式的语言内部结构。这对于避免一些常见的对象模型缺陷和误用是相当重要的。

本章将讨论以下主题：

- 子类化（Subclassing）内建类型；
- 访问超类中的方法；
- 槽（slots）；
- 元编程。

3.1 子类化内建类型

在 Python 2.2 中，提出了类型（type）和类（class）的统一（相关的草案请参阅 http://www.python.org/download/releases/2.2.3/descrintro）——这使内建类型的子类化成为可能，并且添加了一个新的内建类型 object，用来作为所有内建类型的公共祖先。这对于 Python 中的 OOP 机制有着微妙但不重要的影响，使程序员可以对诸如 list、tuple 或 dict 这样的内建类型进行子类化。所以，每当需要实现一个表现与内建类型十分类似的类时，最佳的方法是对其进行子类化。

接下来将展示一个名为 distinctdict 的类的代码，它就使用了这个技术。这是 Python 中常规的 dict 类型的子类。这个新类的表现几乎和一个平常的 dict 一样，但是它不允许存在多个相同的键值。当有人试图添加一个与原键值相同的新输入项时，它将抛出一个 ValueError 异

常，并带有一个帮助信息，如下所示。

```
>>> class DistinctError(Exception):
...     pass
>>> class distinctdict(dict):
...     def __setitem__(self, key, value):
...         try:
...             value_index = self.values().index(value)
...             # 只要 dict 在两次调用之间没有发生改变
...             # keys()和 values()将返回相应的列表
...             # 否则，dict 类型无法保证序列的顺序
...             existing_key = self.keys()[value_index]
...             if existing_key != key:
...                 raise DistinctError(("This value already
...                                      "exists for '%s'") % \
...                                 str(self[existing_key]))
...     except ValueError:
...         pass
...
...     super(distinctdict, self).__setitem__(key, value)
...
>>> my = distinctdict()
>>> my['key'] = 'value'
>>> my['other_key'] = 'value'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 14, in __setitem__
ValueError: This value already exists for 'value'
>>> my['other_key'] = 'value2'
>>> my
{'other_key': 'value2', 'key': 'value'}
```

如果看看现有的代码，可能会发现许多类部分实现了内建类型，它们就像子类型一样，更快而且更清晰。例如，list 类型就是用一个序列来管理类型内部工作时会使用到的序列，如下所示。

```
>>> class folder(list):
...     def __init__(self, name):
...         self.name = name
```

```
...     def dir(self):
...         print 'I am the %s folder.' % self.name
...         for element in self:
...             print element
...
>>> the = folder('secret')
>>> the
[]
>>> the.append('pics')
>>> the.append('videos')
>>> the.dir()
I am the secret folder:
pics
videos
```

从 Python 2.4 开始，collections 模块已经提供了一些可以用来实现有效的容器类的类型：

- 实现双端队列的 deque 类型；
- defaultdict 类型提供一个类似词典的对象，带有默认的未知键码值，这个类型与 Perl 或 Ruby 中的 hash 工作方式类似。

内建类型覆盖了大部分使用场景

若打算创建一个类似序列或映射的新类，应考虑其特性并观察已有的内建类型。大部分时候最终会使用它们。

3.2 访问超类中的方法

super 是一个内建类型，用来访问属于某个对象的超类中的特性（attribute）。

Python 官方文档将 super 作为内建函数列出。尽管它的使用与函数类似，但它实际上仍然是一个内建类型。

```
>>> super
<type 'super'>
```

如果已习惯于通过直接调用父类并将 self 作为第一个参数来访问类型的特性，它的用法可以会带来一点混乱。参考以下代码。

```
>>> class Mama(object): # 这是老的方法
...     def says(self):
...         print 'do your homework'
...
>>> class Sister(Mama):
...     def says(self):
...         Mama.says(self)
...         print 'and clean your bedroom'
...
>>> anita = Sister()
>>> anita.says()
do your homework
and clean your bedroom
```

特别看一下 Mama.says(self)这一行，使用了刚刚描述的方法调用超类（也就是 MaMa 类）中的 says()方法，将 self 作为第一个参数传入。这意味着，属于 Mama 的 says()方法将被调用。但是调用它的实例将返回 self，在这个例子中，这是一个 Sister 的实例。

而 super 的用法将如下所示。

```
>>> class Sister(Mama): # 这是新方法
...     def says(self):
...         super(Sister, self).says()
...         print 'and clean your bedroom'
...
```

这个使用场景很容易理解，但是当面对多继承模式时，super 将会变得难以使用。在解释这些问题，包括何时应该避免使用 super 之前，理解方法解析顺序（MRO）在 Python 中的工作方式是很重要的。

3.2.1　理解 Python 的方法解析顺序

Python 2.3 中添加了基于为 Dylan 构建的 MRO（http://www.opendylan.org），即 C3 的一个新的 MRO。Michele Simionato 所编写的参考文档在 http://www.python.org/download/releases/2.3/mro 上可以找到，它描述了 C3 构建一个类的线性化（也称为优先级，即祖先的一个排序列表）的方法。这个列表被用于特性的查找。

C3 算法将在本小节稍后部分介绍。

MRO 的变化用于解决创建公用基本类型（object）所引入的问题。在变成 C3 线性化方法之前，如果一个类有两个祖先（参见图 3.1），MRO 的计算将很简单，如下所示。

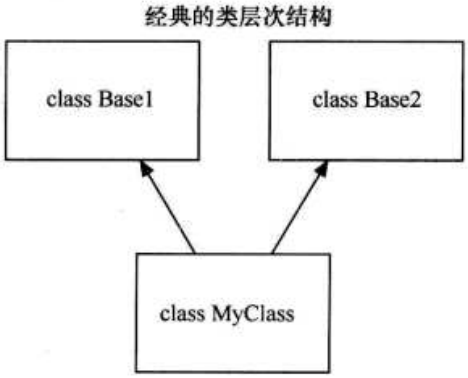

图 3.1

```
>>> class Base1:
...     pass
...
>>> class Base2:
...     def method(self):
...         print 'Base2'
...
>>> class MyClass(Base1, Base2):
...     pass
...
>>> here = MyClass()
>>> here.method()
Base2
```

当 here.method 被调用时，解释程序将查找 MyClass 中的方法，然后在 Base1 中查找，最后在 Base2 中查找。

现在，在两个基类之上引入一个 BaseBase 类（Base1 和 Base2 都从其继承，参见图 3.2）。结果是，根据“从左到右深度优先”规则的旧 MRO，在 Base2 中查找之前将通过 Base1 类回到顶部。

以下的代码将产生一种古怪的表现。

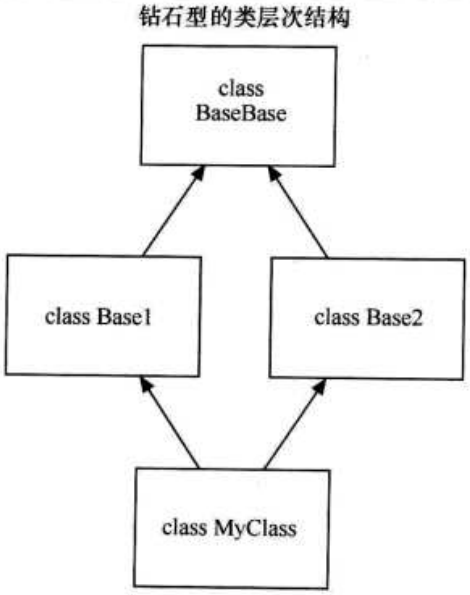

图 3.2

```
>>> class BaseBase:
...     def method(self):
...         print 'BaseBase'
...
>>> class Base1(BaseBase):
...     pass
...
>>> class Base2(BaseBase):
...     def method(self):
...         print 'Base2'
...
>>> class MyClass(Base1, Base2):
...     pass
...
>>> here = MyClass()
>>> here.method()
BaseBase
```

这种继承方案极其罕见，所以这更多的是一个理论问题而不是实践问题。标准程序库不会构建这样的继承层次结构，许多开发人员都认为这是一个坏方法。但是由于在类型层次顶部引入了 object，语言的 C 边（C side）突然出现了多重继承性问题，从而导致了在进行子类型化时的冲突。因为使用已有的 MRO 使其正常工作要花费太多的精力，所以提供一个新的 MRO 是更简单、快捷的解决方案。

所以，相同的实例在当前版本 Python（2.3 以上版本）中应该如下所示。

```
>>> class BaseBase(object):
...     def method(self):
...         print 'BaseBase'
...
>>> class Base1(BaseBase):
...     pass
...
>>> class Base2(BaseBase):
...     def method(self):
...         print 'Base2'
...>>> class MyClass(Base1, Base2):
...     pass
...
>>> here = MyClass()
>>> here.method()
Base2
```

新的 MRO 是基于一个基类之上的递归调用。为了概括本节最开始引用的 Michele Simionato 的文章，将 C3 符号应用到示例中，如下所示。

```
L[MyClass(Base1, Base2)] =
      MyClass + merge(L[Base1], L[Base2], Base1, Base2)
```

L[MyClass]是 MyClass 类的线性化，而 merge 是合并多个线性化结果的具体算法。

因此综合的描述应该是（正如 Simionato 所说）：

C 的线性化是 C 加上父类的线性化和父类列表的合并的总和。

merge 算法负责删除重复项并保持正确的顺序。其在文章中的描述为（适应于本例）：

取第一个列表的头，也就是 L[Base1][0]。如果这个头不在任何表的尾部，那么将它加到 MyClass 的线性化中，并且从合并中的列表里删除；否则查找下一个列表的头，如果是个好的表头则取用它。

然后重复该操作，直到所有类都被删除或者不能找到好的表头。在这个例子中，构建合并是不可能的，Python 2.3 将拒绝创建 MyClass 类并将抛出一个异常。

head（表头）是列表的第一个元素，而 tail（表尾）则包含其余元素。例如，在（Base1, Base2, ..., BaseN）中，Base1 是表头，（Base2, ..., BaseN）则是表尾。

换句话说，C3 在每个父类上进行递归深度查找以获得列表的顺序。然后当一个类涉及多个列表时，计算一个从左到右的规则使用层次二义性消除来合并所有列表。

其结果如下。

```
>>> def L(klass):
...     return [k.__name__ for k in klass.__mro__]
...
>>> L(MyClass)
['MyClass', 'Base1', 'Base2', 'BaseBase', 'object']
```

类的_mro_特性（只读）用来存储线性化计算的结果，计算将在类定义载入时完成。

还可以调用 MyClass.mro()来计算并获得结果。

注意，这将只对新风格的类起作用，所以在代码库中混杂新旧形式的类并不是好方法。MRO 的表现将会有差异。

3.2.2　super 的缺陷

现在回到 super。当使用了多重继承的层次结构时，再使用它将是相当危险的，这主要是因为类的初始化。在 Python 中，基类不会在__init__中被隐式地调用，所以依靠开发人员来调用它们。以下是几个例子。

1．混用 super 和传统调用

在下面的取自 James Knight 网站（http://fuhm.net/super-harmful）的示例中，类 C 使用__init__方法调用其基类，这样将使类 B 被调用两次。

```
>>> class A(object):
...     def __init__(self):
...         print "A"
...         super(A, self).__init__()
...
>>> class B(object):
...     def __init__(self):
...         print "B"
...          super(B, self).__init__()
...
>>> class C(A,B):
...     def __init__(self):
```

```
...         print "C"
...         A.__init__(self)
...         B.__init__(self)
...
>>> print "MRO:", [x.__name__ for x in C.__mro__]
MRO: ['C', 'A', 'B', 'object']
>>> C()
C A B B
<__main__.C object at 0xc4910>
```

发生这种情况的原因是，C 实例调用了 A.__init__(self)，因而 super(A, self).__init__()将调用 B 的构造程序。换句话说，super 应该被用到整个类层次中。问题是有时候这种层次结构的一部分将位于第三方的代码中。许多由多重继承引入的层次调用相关的缺陷都可以在 James 的页面上找到。为了避免这些问题，应该总是在子类化之前看看__mro__特性，如果它不存在，将要处理的就是一个旧式的类，避免使用 super 可能更安全一些，如下所示。

```
>>> from SimpleHTTPServer import SimpleHTTPRequestHandler
>>> SimpleHTTPRequestHandler.__mro__
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: class SimpleHTTPRequestHandler has no attribute '__
mro__'
```

如果__mro__存在，则快速地看看每个 MRO 所涉及的类的构造程序代码。如果到处都使用了 super，那非常好，也可以使用它。否则，就请试着保持一致性。

在下面的例子中可以看到，collections.deque 能够安全地被子类化，可以使用 super，因为它直接子类化了 object，如下所示。

```
>>> from collections import deque
>>> deque.__mro__
(<type 'collections.deque'>, <type 'object'>)
```

在这个例子中，看上去 random.Random 是一个存在于_random 模块中的另一个类的封装器，如下所示。

```
>>> from random import Random
>>> random.Random.__mro__
(<class 'random.Random'>, <type '_random.Random'>, <type 'object'>)
```

这是一个 C 模块，所以我们应该也是安全的。

最后一个例子是一个 Zope 类，构造程序应该被仔细地检查，如下所示。

```
>>> from zope.app.container.browser.adding import Adding
>>> Adding.__mro__
(<class 'zope.app.container.browser.adding.Adding'>,
<class 'zope.publisher.browser.BrowserView'>,
<class 'zope.location.location.Location'>,
<type 'object'>)
```

2. 不同种类的参数

super 用法的另一个问题是初始化中的参数传递。类在没有相同签名的情况下怎么调用其基类的__init__代码？这将引发以下问题。

```
>>> class BaseBase(object):
...     def __init__(self):
...         print 'basebase'
...         super(BaseBase, self).__init__()
...
>>> class Base1(BaseBase):
...     def __init__(self):
...         print 'base1'
...         super(Base1, self).__init__()
...
>>> class Base2(BaseBase):
...     def __init__(self, arg):
...         print 'base2'
...         super(Base2, self).__init__()
...
>>> class MyClass(Base1 , Base2):
...     def __init__(self, arg):
...         print 'my base'
...         super(MyClass, self).__init__(arg)
...
>>> m = MyClass(10)
my base
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<stdin>", line 4, in __init__
TypeError: __init__() takes exactly 1 argument (2 given)
```

一种解决方案是使用*args 和**kw 魔法。这样，所有构造程序将传递所有的参数，即使不使用它们，如下所示。

```
>>> class BaseBase(object):
...     def __init__(self, *args, **kw):
...         print 'basebase'
...         super(BaseBase, self).__init__(*args, **kw)
...
>>> class Base1(BaseBase):
...     def __init__(self, *args, **kw):
...         print 'base1'
...         super(Base1, self).__init__(*args, **kw)
...
>>> class Base2(BaseBase):
...     def __init__(self, arg, *args, **kw):
...         print 'base2'
...         super(Base2, self).__init__(*args, **kw)
...
>>> class MyClass(Base1 , Base2):
...     def __init__(self, arg):
...         print 'my base'
...         super(MyClass, self).__init__(arg)
...
>>> m = MyClass(10)
my base
base1
base2
basebase
```

但是这是一个糟糕的修复方法，因为它使所有构造程序将接受任何类型的参数。这会导致代码变得很脆弱，因为任何参数都被传递并且通过。另一个解决方法是在 MyClass 中使用经典的__init__调用，但是这将会导致产生第一种缺陷。

3.3 最佳实践

为了避免出现前面提到的所有问题，在 Python 在这个领域上取得进展之前，必须考虑以

下几点。

- 应该避免多重继承　可以采用第 14 章中提出的一些设计模式来代替它。
- super 的使用必须一致　在类层次结构中，应该在所有地方都使用 super 或者彻底不使用它。混用 super 和传统调用是一种混乱的方法，人们倾向于避免使用 super，这样代码会更明晰。
- 不要混用老式和新式的类　两者都具备的代码库将导致不同的 MRO 表现。
- 调用父类时必须检查类层次　为了避免出现任何问题，每次调用父类时，必须快速地查看一下所涉及的 MRO（使用__mro__）。

3.4　描述符和属性

当许多 C++和 Java 程序员第一次学习 Python 时，他们会对 Python 中没有 private 关键字感到惊讶。最接近的概念是“name mangling”（名称改编）。每当一个特性前面加上了“__”前缀，它就将被解释程序立刻重新命名，如下所示。

```
>>> class MyClass(object):
...     __secret_value = 1
...
>>> instance_of = MyClass()
>>> instance_of.__secret_value
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'MyClass' object has no attribute '__secret_value'
>>> dir(MyClass)
['_MyClass__secret_value', '__class__', '__delattr__', '__dict__',
'__doc__', '__getattribute__', '__hash__', '__init__', '__module__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__str__', '__weakref__']
>>> instance_of._MyClass__secret_value
1
```

Python 提供它主要是用来避免继承带来的命名冲突，特性将被重命名为带有类名前缀的名称。这实际上并不是强制性的，因为可以通过其组合名称来访问特性。这个功能可被用来保护一些特性的访问，但是在实践中，从不使用“__”。当一个特性不是公开的时，惯例是使用一个“_”前缀。这不会调用任何改编算法，而只是证明这个特性是该类的私有元素，也是流行的样式。

Python 中还有其他可用的机制来构建类的公共部分和私有代码。描述符和属性这些 OOP 设计的关键特征应该被用于设计清晰的 API。

3.4.1 描述符

描述符用来自定义在引用一个对象上的特性时所应该完成的事情。

描述符是 Python 中复杂特性访问的基础。它们在内部使用，以实现属性、类、静态方法和 super 类等。它们是定义另一个类特性可能的访问方式的类。换句话说，一个类可以委托另一个类来管理其特性。

描述符类基于三个必须实现的特殊方法：

- __set__ 在任何特性被设置的时候调用，在后面的实例中，将称其为 setter；
- __get__ 在任何特性被读取时调用（被称为 getter）；
- __delete__ 在特性上请求 del 时调用。

这些方法将在__dict__特性之前被调用。例如，指定一个 MyClass 的实例 instance，在读取 instance.attribute 时使用的算法如下。

```
# 1.查找定义
if hasattr(MyClass, 'attribute'):
    attribute = MyClass.attribute
    AttributeClass = attribute.__class__
    # 2.属性定义是否有 setter
    if hasattr(AttributeClass, '__set__'):
        # let's use it
        AttributeClass.__set__(attribute, instance,
                               value)
        return
# 3.常规方法
instance.__dict__['attribute'] = value
    #    or 'attribute' is not found in __dict__
    writable = (hasattr(AttributeClass, '__set__') or
                'attribute' not in instance.__dict__)
    if readable and writable:
    # 4.调用描述符
    return AttributeClass.__get__(attribute,
                                  instance, MyClass)
# 5.用__dict__正常访问
```

```
    return instance.__dict__['attribute']
```

换句话说，在类特性被定义并且有一个 getter 和一个 setter 方法时，平常的包含一个对象实例的所有元素的__dict__映射都将被劫持。

实现了_get_和_set_的描述符被称为数据描述符。
只实现了_get_的描述符被称为非数据描述符。

下面将创建一个数据描述符，并通过一个实例来使用它。

```
>>> class UpperString(object):
...     def __init__(self):
...         self._value = ''
...     def __get__(self, instance, klass):
...         return self._value
...     def __set__(self, instance, value):
...         self._value = value.upper()
...
>>> class MyClass(object):
...     attribute = UpperString()
...
>>> instance_of = MyClass()
>>> instance_of.attribute
''
>>> instance_of.attribute = 'my value'
>>> instance_of.attribute
'MY VALUE'
>>> instance.__dict__ = {}
```

现在，如果在实例中添加一个新的特性，它将被保存在__dict__映射中，如下所示。

```
>>> instance_of.new_att = 1
>>> instance_of.__dict__
{'new_att': 1}
```

但是，如果将一个新的数据描述符添加到类中，它将优先于实例的__dict__，如下所示。

```
>>> MyClass.new_att = MyDescriptor()
>>> instance_of.__dict__
```

```
{'new_att': 1}
>>> instance_of.new_att
''
>>> instance_of.new_att = 'other value'
>>> instance_of.new_att
'OTHER VALUE'
>>> instance_of.__dict__
{'new_att': 1}
```

这对于非数据描述符无效。在那种情况下，该实例将优先于描述符，如下所示。

```
>>> class Whatever(object):
...     def __get__(self, instance, klass):
...         return 'whatever'
...
>>> MyClass.whatever = Whatever()
>>> instance_of.__dict__
{'new_att': 1}
>>> instance_of.whatever
'whatever'
>>> instance_of.whatever = 1
>>> instance_of.__dict__
{'new_att': 1, 'whatever': 1}
```

建立这个额外的规则以避免递归特性查找。

用于将一个特性设置为一个值的算法（与用于删除特性的相似）如下。

```
# 1.查找定义
if hasattr(MyClass, 'attribute'):
    attribute = MyClass.attribute
    AttributeClass = attribute.__class__
    # 2.属性定义是否有 setter
    if hasattr(AttributeClass, '__set__'):
        # let's use it
        AttributeClass.__set__(attribute, instance,
                               value)
        return
# 3.常规方法
instance.__dict__['attribute'] = value
```

Raymond Hettinger 写了一个名为 How-To Guide for Descriptors 的有趣文档，在 http://users.rcn.com/python/download/Descriptor.htm 中可以找到。该文章是对本小节内容的补充。

除了隐藏类的内容这个主要角色之外，描述符还可以实现一些有趣的代码模式，例如：

- 内省描述符（Introspection descriptor）　这种描述符将检查宿主类签名，以计算一些信息；
- 元描述符（Meta descriptor）　这种描述符使用类方法本身来完成值计算。

1．内省描述符

使用类时有一个公共的需求，那就是对其特性进行一次自我测量（内省）。例如，在 Zope（http://zope.org）的可发布类上计算一个安全性映射时，就实现了这种自我测量。Epydoc（http://epydoc.sourceforge.net）也做了相似的工作来计算文档。

计算这种文档的属性类，可以通过检查公共方法来呈现一个易于理解的文档。以下是基于内建函数 dir 的这样一个非数据描述符的实例，它可以工作于任何对象类型之上。

```
>>> class API(object):
...     def _print_values(self, obj):
...         def _print_value(key):
...             if key.startswith('_'):
...                 return ''
...             value = getattr(obj, key)
...             if not hasattr(value, 'im_func'):
...                 doc = type(value).__name__
...             else:
...                 if value.__doc__ is None:
...                     doc = 'no docstring'
...                 else:
...                     doc = value.__doc__
...             return ' %s : %s' % (key, doc)
...         res = [_print_value(el) for el in dir(obj)]
...         return '\n'.join([el for el in res
...                           if el != ''])
...     def __get__(self, instance, klass):
```

```
...                 if instance is not None:
...                     return self._print_values(instance)
...                 else:
...                     return self._print_values(klass)
...
>>> class MyClass(object):
...     __doc__ = API()
...     def __init__(self):
...         self.a = 2
...     def meth(self):
...         """my method"""
...         return 1
...
>>> MyClass.__doc__
    ' meth : my method'
>>> instance = MyClass()
>>> print instance.__doc__
    a : int
    meth : my method
```

这个描述符将过滤以下划线开始的元素，并且显示方法的 docstrings。

2. 元描述符

元描述符使用宿主类的一个或多个方法来执行一个任务。这可能对降低使用提供步骤的类所需的代码量很有用，比如，一个链式的描述符可以调用类的一系列方法以返回一组结果。它可以在失败的时候被停止，并且配备一个回调机制以获得对过程的更多控制，如下所示。

```
>>> class Chainer(object):
...     def __init__(self, methods, callback=None):
...         self._methods = methods
...         self._callback = callback
...     def __get__(self, instance, klass):
...         if instance is None:
...             # 只针对实例
...             return self
...         results = []
...         for method in self._methods:
...             results.append(method(instance))
```

```
...             if self._callback is not None:
...                 if not self._callback(instance,
...                                       method,
...                                       results):
...                     break
...         return results
```

这一实现使各种在类方法之上运行的计算能与记录器这样的外部元素结合起来，如下所示。

```
>>> class TextProcessor(object):
...     def __init__(self, text):
...         self.text = text
...     def normalize(self):
...         if isinstance(self.text, list):
...             self.text = [t.lower()
...                          for t in self.text]
...         else:
...             self.text = self.text.lower()
...     def split(self):
...         if not isinstance(self.text, list):
...             self.text = self.text.split()
...     def treshold(self):
...         if not isinstance(self.text, list):
...             if len(self.text) < 2:
...                 self.text = ''
...         self.text = [w for w in self.text
...                            if len(w) > 2]
...
>>> def logger(instance, method, results):
...     print 'calling %s' % method.__name__
...     return True
...
>>> def add_sequence(name, sequence):
...     setattr(TextProcessor, name,
...             Chainer([getattr(TextProcessor, n)
...                      for n in sequence], logger))
```

add_sequence 用来动态地定义一个新的链式调用方法的描述符。这一组合的结果可以被保存在类定义中，如下所示。

```
>>> add_sequence('simple_clean', ('split', 'treshold'))
>>> my = TextProcessor(' My Taylor is Rich ')
>>> my.simple_clean
calling split
calling treshold
[None, None]
>>> my.text
['Taylor', 'Rich']
>>> # 执行另一个序列
>>> add_sequence('full_work', ('normalize',
...                            'split', 'treshold'))
>>> my.full_work
calling normalize
calling split
calling treshold
[None, None, None]
>>> my.text
['taylor', 'rich']
```

由于 Python 的动态特性，可以在运行时再添加这种描述符以执行元编程。

定义

元编程（Meta-programming）是在运行时通过添加新的计算功能，或者改变已有功能来改变程序行为的一种技巧。它与普通的编程（具备 C++背景的人可能对此比较熟悉）不同。在 C++中，将需要创建新的代码块，而不是提供可以应对最多情况的简单代码块。

它也和“生成性编程”（generative programming）不同，这种编程方式是通过模板来生成一段静态的源代码。具体信息可参见 http://en.wikipedia.org/wiki/Generative_programming。

3.4.2 属性

属性（Property）提供了一个内建的描述符类型，它知道如何将一个特性链接到一组方法上。属性采用 fget 参数和 3 个可选的参数——fset、fdel 和 doc。最后一个参数可以提供用来

定义一个链接到特性的 docstring，就像是个方法一样，如下所示。

```
>>> class MyClass(object):
...     def __init__(self):
...         self._my_secret_thing = 1
...
...     def _i_get(self):
...         return self._my_secret_thing
...
...     def _i_set(self, value):
...         self._my_secret_thing = value
...
...     def _i_delete(self):
...         print 'neh!'
...
...     my_thing = property(_i_get, _i_set, _i_delete,
...                         'the thing')
...
>>> instance_of = MyClass()
>>> instance_of.my_thing
1
>>> instance_of.my_thing = 3
>>> instance_of.my_thing
3
>>> del instance_of.my_thing
neh !
>>> help(instance_of)
Help on MyClass in module __main__ object:
class MyClass(__built-in__.object)
 |  Methods defined here:
 |
 |  __init__(self)
 |
 |  ----------------------------------------------------
 |  Data descriptors defined here:
 |  ...
 |  my_thing
 |  the thing
```

属性简化了描述符的编写，但是在使用类的继承时必须小心处理。所创建的特性使用当前类的方法创建，而不应使用在派生类中重载的方法。这有些混乱，因为后者是大部分实现属性的语言中相当合乎逻辑的行为。

例如，以下的例子将不能像预想中那样工作。

```
>>> class FirstClass(object):
...     def _get_price(self):
...         return '$ 500'
...     price = property(_get_price)
...
>>> class SecondClass(FirstClass):
...     def _get_price(self):
...         return '$ 20'
...
...
>>> plane_ticket = SecondClass()
>>> plane_ticket.price
'$ 500'
```

对这种行为的解决方法是，使用另一种方法手工将属性实例重定向到正确的方法上，如下所示。

```
>>> class FirstClass(object):
...     def _get_price(self):
...         return '$ 500'
...     def _get_the_price(self):
...         return self._get_price()
...     price = property(_get_the_price)
...
>>> class SecondClass(FirstClass):
...     def _get_price(self):
...         return '$ 20'
...
>>> plane_ticket = SecondClass()
>>> plane_ticket.price
'$ 20'
```

尽管如此，大部分时候属性都将被添加到类中，以隐藏其复杂性。链接到它们的方法是私有的，所以重载它们是不好的做法。在这种情况下，重载属性本身更好一些，如下所示。

```
>>> class FirstClass(object):
...     def _get_price(self):
...         return '$ 500'
...     price = property(_get_price)
...
>>> class SecondClass(FirstClass):
...     def _cheap_price(self):
...         return '$ 20'
...     price = property(_cheap_price)
...
>>> plane_ticket = SecondClass()
>>> plane_ticket.price
'$ 20'
```

3.5　槽

几乎从未被开发人员使用过的一种有趣的特性是槽。它允许使用__slots__特性为指定的类设置一个静态特性列表，并且跳过每个类实例中__dict__列表的创建工作。它们用来为特性很少的类节省存储空间，因为将不在每个实例中创建__dict__。

除此之外，它们有助于设计签名必须被冻结的类。例如，如果必须限制类之上的语言动态特性，定义槽也是有帮助的，如下所示。

```
>>> class Frozen(object):
...     __slots__ = ['ice', 'cream']
...
>>> '__dict__' in dir(Frozen)
False
>>> 'ice' in dir(Frozen)
True
>>> glagla = Frozen()
>>> glagla.ice = 1
>>> glagla.cream = 1
>>> glagla.icy = 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Frozen' object has no attribute 'icy'
```

因为任何新的特性都将在__dict__中被添加，所以这无法在派生类上工作。

3.6 元 编 程

new-style 类带来了一种能力，可以通过两个特殊方法——__new__和 __metaclass__在运行时修改类和对象的定义。

3.6.1 __new__方法

特殊方法__new__是一个元构造程序，每当一个对象必须被 factory 类实例化时就将调用它，如下所示。

```
>>> class MyClass(object):
...     def __new__(cls):
...         print '__new__ called'
...         return object.__new__(cls) # default factory
...     def __init__(self):
...         print '__init__ called'
...         self.a = 1
...
>>> instance = MyClass()
__new__ called
__init__ called
```

__new__方法必须返回一个类的实例。因此，它可以在对象创建之前或之后修改类。这对于确保对象构造程序不会被设置成一个不希望的状态，或者添加一个不能被构造程序删除的初始化是有帮助的。

例如，因为__init__在子类中不会被隐式调用，所以__new__可以用来确定已经在整个类层次中完成了初始化工作，如下所示。

```
>>> class MyOtherClassWithoutAConstructor(MyClass):
...     pass
>>> instance = MyOtherClassWithoutAConstructor()
__new__ called
__init__ called
```

```
>>> class MyOtherClass(MyClass):
... def __init__(self):
...     print 'MyOther class __init__ called'
...     super(MyOtherClass, self).__init__()
...     self.b = 2
...
>>> instance = MyOtherClass()
__new__ called
MyOther class __init__ called
__init__ called
```

例如，网络套接字或数据库初始化应该在__new__中而不是__init__中控制。它在类的工作必须完成这个初始化以及必须被继承的时候通知我们。

例如，threading 模块中的 Thread 类就使用了这种机制，以避免存在未初始化的实例，如下所示。

```
>>> from threading import Thread
>>> class MyThread(Thread):
...     def __init__(self):
...         pass
...
>>> MyThread()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Library/Frameworks/Python.framework/Versions/2.5/lib/
python2.5/threading.py", line 416, in __repr__
    assert self.__initialized, "Thread.__init__() was not called"
AssertionError: Thread.__init__() was not called
```

这实际上是通过方法之上的断言来完成的（assert self.__initialized），并且可以简化为__new__中的单一调用，因为这个实例除此之外就没有什么作用。

避免令人头痛的链式初始化

__new__是对于对象状态隐式初始化需求的回应。它使得可以在比_init_更低的层次上定义一个初始化，这个初始化总是会被调用。

3.6.2 __metaclass__方法

元类（Metaclass）提供了在类对象通过其工厂方法（factory）在内存中创建时进行交互的能力。它们的效果与__new__类似，只不过是在类级别上运行。内建类型 type 是内建的基本工厂，它用来生成指定名称、基类以及包含其特性的映射的任何类，如下所示。

```
>>> def method(self):
...     return 1
...
>>> klass = type('MyClass', (object,), {'method': method})
>>> instance = klass()
>>> instance.method()
1
```

这与类的显式定义类似，如下所示。

```
>>> class MyClass(object):
...     def method(self):
...         return 1
...
>>> instance = MyClass()
>>> instance.method()
1
```

有了这个功能，开发人员可以在调用 type 之前或之后与类创建交互。一个特殊的特性已经被创建以链接到一个定制的工厂上。

__metaclass__（在 Python 3000 中，其将被一个显式的构造程序参数替代）可以被添加到一个类定义中，以与创建过程进行交互。__metaclass__特性必须被设置为：

- 接受和 type 相同的参数（即，一个类名、一组基类和一个特性映射）；
- 返回一个类对象。

完成以上事情的是类似下面使用的那种无束缚的函数（equip 函数），还有另一个类对象上的一个方法，这些并不重要，只要它能够满足规则 1 和 2 就行。在这个实例中，如果类有一个空的 docstring，那么在 3.4.1 小节中提出的 API 描述符就将被自动地添加到类中，如下所示。

```
>>> def equip(classname, base_types, dict):
...     if '__doc__' not in dict:
```

```
...         dict['__doc__'] = API()
...     return type(classname, base_types, dict)
...
>>> class MyClass(object):
...     __metaclass__ = equip
...     def alright(self):
...         """the ok method"""
...         return 'okay'
...
>>> ma = MyClass()
>>> ma.__class__
<class '__main__.MyClass'>
>>> ma.__class__.__dict__['__doc__'] # __doc__ is replaced !
<__main__.API object at 0x621d0>
>>> ma.y = 6
>>> print ma.__doc__
    alright : the ok method
    y : int
```

这个变化在其他地方可能不可行，因为__doc__是内建的基本元类 type 的只读特性。

但是元类使代码变得更加复杂，而且在将其用于工作于所有类型的类时，会使其健壮性变得更差。例如，可能在类中使用槽时遇到不好的交互，或者在一些基类已经实现了一个元类时，这个类与所做的冲突。可能它们只是没有构造好。

对于修改可读写的特性或添加新特性而言，可以避免使用元类，而采用更简单的基于动态修改类实例的解决方案。这些修改更容易管理，因为它们不需要被组合到一个类中（一个类只能有一个元类）。例如，如果两个具体的行为必须被应用到一个类，可以使用一个“增强函数”来添加它们，如下所示。

```
>>> def enhancer_1(klass):
...     c = [l for l in klass.__name__ if l.isupper()]
...     klass.contracted_name = ''.join(c)
...
>>> def enhancer_2(klass):
...     def logger(function):
...         def wrap(*args, **kw):
...             print 'I log everything !'
...             return function(*args, **kw)
```

```
...         return wrap
...     for el in dir(klass):
...         if el.startswith('_'):
...             continue
...         value = getattr(klass, el)
...         if not hasattr(value, 'im_func'):
...             continue
...         setattr(klass, el, logger(value))
...
>>> def enhance(klass, *enhancers):
...     for enhancer in enhancers:
...         enhancer(klass)
...
>>> class MySimpleClass(object):
...     def ok(self):
...         """I return ok"""
...         return 'I lied'
...
>>> enhance(MySimpleClass, enhancer_1, enhancer_2)
>>> thats = MySimpleClass()
>>> thats.ok()
I log everything !
'I lied'
>>> thats.score
>>> thats.contracted_name
'MSC'
```

这是很强大的表现——可以动态地在已经被实例化的类定义上创建许多不同的变化。

在任何情况下，请记住元类或动态增强只是一种“补丁”，它可能会很快地使精心定义的、清晰的类层次结构变得一团糟。它们应该只在以下情况下使用：

- 在框架级别，一个行为在许多类中是强制的时候；
- 当一个特殊的行为被添加的目的不是与诸如记录日志这样的类提供的功能交互时。

更多的例子，可以参看 David Mertz 对元类编程的一个很棒的介绍，其链接为 http://www.onlamp.com/pub/a/python/2003/04/17/metaclasses.html? page=1。

3.7　小　　结

本章重点介绍了以下内容。

- 子类型化内建类型是个很好的特性，但是在这么做之前，应确定不对现有的类型进行子类化是不合适的。
- 因为 super 的使用十分棘手，所以：在代码中避免使用多重继承；用法要保持一致，不要混用新旧样式的类；在子类中调用方法之前应先检查类层次。
- 通过描述符可以自定义在一个对象上引用特性时所应该做的任务。
- 属性对于构建一个公共 API 是很棒的。
- 元编程非常强大，但是记住，它会影响类设计的易读性。

下一章中将重点介绍如何为编码元素选择好的名称，以及 API 设计的最佳实践。