第 4 章

选择好的名称

大部分标准程序库在构建时都不会忽略可用性。例如，内建类型的使用是很自然的，并且设计得十分易于使用。在这方面，Python 可以和建立程序时所考虑的伪代码相比，大部分代码都可以朗读出来。例如，以下片段大家都会感到很容易理解。

```
>>> if 'd' not in my_list:
...     my_list.append('d')
```

这是编写 Python 程序和使用其他语言相比更加简单的原因之一。当编写一段程序时，你的思路很快就能转化成为代码行。

本章将关注于与编写易于理解和使用的代码相关的最佳实践，包括：

- 使用 PEP8 中描述的命名约定，以及一组命名最佳实践；
- 命名空间重构；
- 使用 API，从初始模型到其重构模型。

4.1 PEP 8 和命名最佳实践

PEP 8（http://www.python.org/dev/peps/pep-0008）为 Python 编程提供了一个与编码风格相关的指南。除了空格缩进、每行最大长度以及其他与代码布局有关的细节之类的基本规则以外，PEP8 还提供了一个大部分代码库所遵循的命名约定。

本节只是摘要性地介绍了 PEP，并且给出了针对每种元素的最佳实践指南。

4.2　命 名 风 格

Python 中使用的不同命名风格包括：

- CamelCase（每个单词首字母大写）；
- mixedCase（和 CamelCase 类似，但是第一个单词的首字母仍然为小写）；
- UPPERCASE（大写）和 ER_CASE_WITH_UNDERSCORES（大写并且带下划线）；
- lowercase（小写）和 lower_case_with_underscores（小写并且带下划线）；
- 前缀（_leading）和后缀（trailing_）下划线，或者都加下划线（__doubled__）。

小写和大写元素通常是一个单词，有时候是少数几个词的连接。使用下划线的通常是缩写词。使用单词会更好一些。前缀和后缀下划线用来标记私有和特殊的元素。

这些风格将被应用到：

- 变量；
- 函数和方法；
- 属性；
- 类；
- 模块；
- 包。

4.2.1　变量

Python 中有两种变量：

- 常量；
- 公有和私有变量。

1．常量

对于值不会发生改变的全局变量，使用大写和一个下划线。它告诉开发人员指定的变量代表一个恒定值。

Python 中没有真正的常量——C++中用 const 定义的那种常量。任何变量值都可以修改。这就是 Python 使用命名约定来将一个变量标记为常量的原因。

例如，doctest 模块提供一个选项标志和指令（都是一些短小的句子，清晰地定义了每个选项的用途）的列表（参见http://docs.python.org/lib/doctest-options.html），如下所示。

```
>>> from doctest import IGNORE_EXCEPTION_DETAIL
>>> from doctest import REPORT_ONLY_FIRST_FAILURE
```

这些变量名看上去相当长，但是清晰地描述它们很重要。基本上，都在初始化代码中使用它们，而不会在代码本身的主体中使用，所以冗长的名称并不会令人厌烦。

大部分时候，缩写的名称会使代码变得模糊。在缩写名称不够清晰时，不要害怕使用完整的词语。

一些常量的名称也是基于低层技术派生的。例如，os 模块使用 C 上定义的一些常量，诸如 EX_XXX 系列定义了异常编号，如下所示。

```
>>> import os
>>> try:
...     os._exit(0)
... except os.EX_SOFTWARE:
...     print 'internal softwar error'
...     raise
```

使用常量时，将它们集中放在模块的头部是一个好办法。当它用于如下所示的操作时，应将其组合在新的变量之下。

```
>>> import doctest
>>> TEST_OPTIONS = (doctest.ELLIPSIS |
...                 doctest.NORMALIZE_WHITESPACE |
...                 doctest.REPORT_ONLY_FIRST_FAILURE)
```

2．命名和使用

常量用来定义一组程序所依赖的值，如默认配置文件名称。

将所有常量集中放在包中的独立文件中是一种好方法。例如，Django 采用的就是这种方式。它使用一个名为 config.py 的模块来提供所有常量，如下所示。

```
# config.py
SQL_USER = 'tarek'
SQL_PASSWORD = 'secret'
```

```
SQL_URI = 'postgres://%s:%s@localhost/db' % \
                (SQL_USER, SQL_PASSWORD)
MAX_THREADS = 4
```

另一种方法是使用可以被 ConfigParser 模块或者像 ZConfig 之类的高级工具(用于 Zope 中描述其配置文件的解析器）解析的配置文件。但是有些人认为，在 Python 这种程序文件能够像文本文件一样易于编辑和修改的语言中，采用这种方法是对另一种文件格式的过度使用。

对于表现得像标志的选项，将它们和布尔操作组合是一种好方法，就像 doctest 和 re 模块那样。从 doctest 得到的模式很简单，如下所示。

```
>>> OPTIONS = {}
>>> def register_option(name):
...     return OPTIONS.setdefault(name, 1 << len(OPTIONS))
>>> def has_option(options, name):
...     return bool(options & name)
>>> # 现在定义选项
>>> BLUE = register_option('BLUE')
>>> RED = register_option('RED')
>>> WHITE = register_option('WHITE')
>>>
>>> # 然后尝试它们
>>> SET = BLUE | RED
>>> has_option(SET, BLUE)
True
>>> has_option(SET, WHITE)
False
```

当创建这样一组新的常量时，应避免使用常用的前缀，除非模块中有多个组。模块名本身就是一个常见的前缀。

在 Python 中，使用二进制逐位计算来组合选项是常见的方法。使用或（|）操作符可以将多个选项组合在一个整数中，而使用与（&）操作符将能够检查整数中表示的选项（参见 has_option 函数）。

如果这个整数可以使用<<操作符移位，以确保和组合得到的整数中的另一个不相同，那么它就能起作用。换句话说，它是 2 的幂（参见 register_options）。

3. 公有和私有变量

对于易变和公有的全局变量，当它们需要被保护时应该使用小写和一个下划线。但是这种变量不常使用，因为在需求保护时，模块通常会提供 getter 和 setter 来处理它们。在这种情况下，一个前导下划线可以表示该变量为包的私有元素，如下所示。

```
>>> _observers = []
>>> def add_observer(observer):
...     _observers.append(observer)
>>> def get_observers():
...     """Makes sure _observers cannot be modified."""
...     return tuple(_observers)
```

位于函数和方法中的变量遵循相同的规则，并且永远不会被标志为私有，因为它们对上下文来说是局部的。

对于类或实例变量而言，只在将变量作为公共签名的一部分，并且不能带来任何有用的信息或冗余的情况下才必须使用私有标志（即前导下划线）。

换句话说，如果该变量是在方法内部使用，用来提供一个公共特性，并且只扮演这个角色，那么最好是将其声明为私有变量。

例如，只支持一个属性的特性是好的私有成员，如下所示。

```
>>> class Citizen(object):
...     def __init__(self):
...         self._message = 'Go boys'
...     def _get_message(self):
...         return self._message
...     kane = property(_get_message)
>>> Citizen().kane
'Go boys'
```

另一个例子是用来保持内部状态的变量。这个值对于余下的代码没有用处，但是它参与了类的行为，如下所示。

```
>>> class MeanElephant(object):
...     def __init__(self):
...         self._people_to_kill = []
...     def is_slapped_on_the_butt_by(self, name):
...         self._people_to_kill.append(name)
...         print 'Ouch!'
```

```
...     def revenge(self):
...         print '10 years later...'
...         for person in self._people_to_kill:
...             print 'Me kill %s' % person
>>> joe = MeanElephant()
>>> joe.is_slapped_on_the_butt_by('Tarek')
Ouch!
>>> joe.is_slapped_on_the_butt_by('Bill')
Ouch!
>>> joe.revenge()
10 years later...
Me kill Tarek
Me kill Bill
```

不要轻易断言类进行子类化时可能采用的方式。

4.2.2　函数和方法

函数和方法命名时应该使用小写和下划线。但是，在标准程序库中并不总是遵守这个规则，可以找到一些使用混合大小写（mixedCase）的模式，例如 threading 模块中的 currentThread（这在 Python 3000 中可能会发生改变）。

这种编写方法的方式在小写范式成为标准之前很常见，在诸如 Zope 之类的框架中也使用了混合大小写的方法，使用它的开发人员群体相当大。所以，在混合大小写和小写加下划线之间做什么选择，主要取决于所使用的程序库。

作为一个 Zope 开发人员，保持一致性并不容易，因为构建一个混合纯 Python 模块和引用了 Zope 代码的模块的应用程序很困难。在 Zope 中，有一些类混用了两种约定，因为代码库正在演变成一个基于 egg 的框架，因此每个模块都比之前更接近于纯 Python。

在这种类型的程序库环境中，最正统的方法是只对被输出到框架中的元素使用混合大小写，其余代码保持使用 PEP 风格。

1．关于私有元素的争论

对于私有的方法和函数而言，命名惯例是添加一个前导下划线。考虑到 Python 的名称改编特性，这个规则是相当有争议的。当方法有两个前导下划线时，解释程序会立即对其更名，以避免和子类中的方法产生冲突。

所以有些人倾向于对其私有的特性使用双前导下划线，以避免子类中的命名冲突，如下所示。

```
>>> class Base(object):
...     def __secret(self):
...         print "don't tell"
...     def public(self):
...         self.__secret()
>>> Base.__secret
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'Base' has no attribute '__secret'
>>> dir(Base)
['_Base__secret', ..., 'public']
>>> class Derived(Base):
...     def __secret(self):
...         print "never ever"
>>> Derived().public()
don't tell
```

Python 中设计名称改编（name mailing）的原始动机不是提供类似 C++中的私有机关（gimmick），而是用来确保一些基类隐式地避免子类中的冲突，尤其是在多继承环境中。但是如果针对每个特性都使用它，就会使代码变得模糊不清，这根本不是 Python 的风格。

因此，有些人认为应该始终使用显式的名称改编，如下所示。

```
>>> class Base(object):
...     def _Base_secret(self):      # 别这样干!!!
...         print "you told it ?"
```

该类名将在整个代码中不断重复，因此__是首选的。

但是最佳的方法，正如 BDFL（Guido，Benevolent Dictator For Life，参见 http://en.wikipedia.org/wiki/BDFL[1]）所说，是通过在子类中编写方法之前查看__mro__值（方法解析顺序）来避免名称改编。改变基类的私有方法必须小心进行。

关于这个主题的更多信息，几年前在 python-dev 邮件列表上出现过一个有趣的思路，人们争论名称改编的实用性及其在这种语言中的命运。具体信息可以在 http://mail.python.org/

1 译者注：Guido van Rossum是 Python 发明者，人称 Benevolent Dictator For Life（缩写为 BDFL，即仁慈大帝），这个称号常被用来称呼少数的开放源码软件开发领导者。

pipermail/python-dev/2005-December/058555.html 上找到。

2．特殊的方法

特殊方法（http://docs.python.org/ref/specialnames.html）是以两个下划线开始和结束的，常规方法不应该使用这种命名约定。它们被用于操作符重载、容器定义等。为了使程序易读，它们应该被集中放在类定义的最前面，如下所示。

```
>>> class weirdint(int):
...     def __add__(self, other):
...         return int.__add__(self, other) + 1
...     def __repr__(self):
...         return '<weirdo %d>' % self
...     #
...     # 公有 API
...     #
...     def do_this(self):
...         print 'this'
...     def do_that(self):
...         print 'that'
```

对于常规的方法，绝不应该使用这种名称，所以不要创建诸如以下这种方法名称。

```
>>> class BadHabits(object):
...     def __my_method__(self):
...         print 'ok'
```

3．参数

参数使用小写，如果需要的话可以加上下划线。它们遵循与变量相同的命名规则。

4.2.3　属性

属性名称是用小写或者小写加上下划线命名的。大部分时候，它们表示对象的状态，可以是一个名词或一个形容词，在需要的时候也可以是一个小短语，如下所示。

```
>>> class Connection(object):
...     _connected = []
...     def connect(self, user):
...         self._connected.append(user)
```

```
...     def _connected_people(self):
...         return '\n'.join(self._connected)
...     connected_people = property(_connected_people)
>>> my = Connection()
>>> my.connect('Tarek')
>>> my.connect('Shannon')
>>> print my.connected_people
Tarek
Shannon
```

4.2.4 类

类的名称总是使用 CamelCase 格式命名，当定义的是模块的私有类时，还可能有一个前导下划线。

类和实例变量常常是名词短语，其使用逻辑与用动词短语命名方法一致，如下所示。

```
>>> class Database(object):
...     def open(self):
...         pass
>>> class User(object):
...     pass
>>> user = User()
>>> db = Database()
>>> db.open()
```

4.2.5 模块和包

除了特殊模块__init__之外，模块名称都使用不带下划线的小写字母命名。

以下是一些标准程序库中的例子：

- os；
- sys；
- shutil。

当模块对于包而言是私有的时候，将添加一个前导下划线。编译过的 C 或 C++模块名称通常带有一个下划线并且将导入到纯 Python 模块中。

包将遵循相同的规则，因为它们在命名空间中的表现和模块类似。

4.3　命 名 指 南

一组常用的命名规则可以被应用到变量、方法函数和属性上。类和模块的名称在命名空间的构建中也扮演重要的角色，从而也影响了代码易读性。本指南中将提供命名的常见模式和反模式。

4.3.1　使用“has”或“is”前缀命名布尔元素

当一个元素是用来保存布尔值时，“is”和“has”前缀提供一个自然的方式，使其在命名空间中很容易被理解，如下所示。

```
>>> class DB(object):
...     is_connected = False
...     has_cache = False
>>> database = DB()
>>> database.has_cache
False
>>> if database.is_connected:
...     print "That's a powerful class"
... else:
...     print "No wonder..."
No wonder...
```

4.3.2　用复数形式命名序列元素

当一个元素是用来保存一个序列时，使用复数形式命名是个好主意。对一些以类似序列的形式输出的映射而言，也可以从中获益，如下所示。

```
>>> class DB(object):
...     connected_users = ['Tarek']
...     tables = {'Customer': ['id', 'first_name',
...                            'last_name']}
```

4.3.3　用显式的名称命名字典

当一个变量是用来保存一个映射时，应该尽可能使用显式的名称。例如，有一个用来保

存个人地址的 dict，那么可以将其命名为 person_address，如下所示。

```
>>> person_address = {'Bill': '6565 Monty Road',
...                   'Pamela': '45 Python street'}
>>> person_address['Pamela']
'45 Python street'
```

4.3.4 避免通用名称

如果在代码中不构建一个新的抽象数据类型，使用诸如 list、dict、sequence 或 elements 这样的名称是有害的，即使对于局部变量也一样。它将使代码难以阅读理解和使用。还应该避免使用内建的名称，以避免在当前命名空间中遮蔽它。还要避免通用的动词，除非它们在该命名空间中有意义。

作为替代，应该使用与问题域相关的词语，如下所示。

```
>>> def compute(data): # too generic
...     for element in data:
...         yield element * 12
>>> def display_numbers(numbers): # better
...     for number in numbers:
...         yield number * 12
```

4.3.5 避免现有名称

使用已经存在于上下文中的名称也是一个坏习惯，因为它会使得在阅读程序时，特别是调试时产生很多混乱，如下所示。

```
>>> def bad_citizen():
...     os = 1
...     import pdb; pdb.set_trace()
...     return os
>>> bad_citizen()
> <stdin>(4)bad_citizen()
(Pdb) os
1
(Pdb) import os
(Pdb) c
<module 'os' from '/Library/Frameworks/Python.framework/Versions/2.5/
```

```
lib/python2.5/os.
pyc'>
```

在这个例子中，os 名称将被代码遮蔽。内建的来自标准程序库的模块名称都应该避免。

尝试创建独特的名称，即使它们是上下文中的局部名称。对于关键字而言，添加一个后缀下划线是避免冲突的一种方法，如下所示。

```
>>> def xapian_query(terms, or_=True):
...     """if or_ is true, terms are combined
...     with the OR clause"""
...     pass
```

注意，class 常常被改成 klass 或 cls，如下所示。

```
>>> def factory(klass, *args, **kw):
...     return klass(*args, **kw)
```

4.4　参数最佳实践

函数和方法的签名是代码完整性的保证，它们驱动函数和方法的使用并构建其 API。除了我们已经了解的命名规则，对参数也要特别小心。下面是 3 个简单的规则：

- 根据迭代设计构建参数；
- 信任参数和测试；
- 小心使用魔法参数*args 和 **kw。

4.4.1　根据迭代设计构建参数

如果每个函数都有一个固定的、精心设计的参数列表，会使代码更加健壮。但是这在第一个版本中不能完成，所以参数必须根据迭代设计来构建。它们应该反映创建该元素所针对的使用场景，并且相应地进行演化。

例如，当附加一些参数时，它们应该尽可能有默认值以避免任何退化，如下所示。

```
>>> class BD(object): # version 1
...     def _query(self, query, type):
...         print 'done'
...     def execute(self, query):
...         self._query(query, 'EXECUTE')
```

```
>>> BD().execute('my query')
done
>>> import logging
>>> class BD(object): # version 2
...     def _query(self, query, type, logger):
...         logger('done')
...     def execute(self, query, logger=logging.info):
...         self._query(query, 'EXECUTE', logger)
>>> BD().execute('my query') # old-style call
>>> BD().execute('my query', logging.warning)
WARNING:root:done
```

当一个公共元素的参数必须变化时，将使用一个 deprecation 进程，本节稍后部分将对此进行说明。

4.4.2 信任参数和测试

基于 Python 的动态类型特性的考虑，有些开发人员在函数和方法的头部使用断言来确保参数有正确的内容，如下所示。

```
>>> def division(dividend, divisor):
...     assert type(dividend) in (long, int, float)
...     assert type(divisor) in (long, int, float)
...     return dividend / divisor
>>> division(2, 4)
0
>>> division(2, 'okok')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in division
AssertionError
```

这通常是那些习惯于动态类型并且感觉 Python 中缺少点什么的开发人员的杰作。

这种检查参数的方法是契约式设计编程风格（Design by Contract，参见 http://en.wikipedia.org/wiki/Design_By_Contract）的一部分。在这种设计中，代码在实际运行之前会先检查预言。

这种方法的两个主要问题是：

（1）DBC 的代码解释它应该如何使用，导致程序易读性降低；

（2）这可能使代码执行速度变得更慢，因为每次调用都要进行断言。

后者可以通过在执行解释程序时加上“-O”选项来避免。在这种情况下，所有断言都将在字节码被创建之前从代码中删除，这样检查也就会丢失。

在任何情况下，断言都必须小心进行，并且不应该导致 Python 变成一种静态类型的语言。唯一的使用场景就是保护代码不被无意义地调用。

一个健康的测试驱动开发风格在大部分情况下将提供健壮的基础代码。在这里，功能和单元测试验证了所有创建代码所针对的使用场景。

当程序库中的代码被外部元素使用时，建立断言可能是有用的，因为输入数据可能会导致程序结束甚至造成破坏。这在处理数据库或文件系统的代码中是很可能发生的。

另一种方法是“模糊测试（fuzz testing）”（http://en.wikipedia.org/wiki/Fuzz_testing），它通过向程序发送随机的数据块以检测其弱点。当发现一个新的缺陷时，代码可以相应地被修复，并加上一次新的测试。

让我们来关注一个遵循 TDD 方法，向正确的方向演变并且每当新的缺陷出现时进行调整，从而使健壮性越来越好的代码库。当它以正确的方式完成时，测试中的断言列表在某种程度上变得和预言列表相似。

不管怎么说，Python 中已经有许多为爱好者所做的 Dbc 程序库，可以在 *Contracts for Python* 中看到（http://www.wayforward.net/pycontract/）。

4.4.3　小心使用*args 和**kw 魔法参数

*args 和**kw 参数可能会破坏函数或方法的健壮性。它们会使签名变得模糊，而且代码常常开始在不应该出现的地方构建小的参数解析器，如下所示。

```
>>> def fuzzy_thing(**kw):
...     if 'do_this' in kw:
...         print 'ok i did'
...     if 'do_that' in kw:
...          print 'that is done'
...     print 'errr... ok'
>>> fuzzy_thing()
errr... ok
>>> fuzzy_thing(do_this=1)
ok i did
errr... ok
>>> fuzzy_thing(do_that=1)
that is done
errr... ok
```

```
>>> fuzzy_thing(hahahahaha=1)
errr... ok
```

如果参数列表很长而且很复杂，那么添加魔法参数是很有诱惑力的。但这通常意味着，它是一个脆弱的函数或方法，它们应该被分解或者重构。

当*args 被用来处理一系列在函数中以同样方式处理的元素时，要求传入诸如 iterator 之类的唯一容器参数会更好些，如下所示。

```
>>> def sum(*args): #可行
...     total = 0
...     for arg in args:
...         total += arg
...     return total
>>> def sum(sequence): # 更好!
...     total = 0
...     for arg in args:
...         total += arg
...     return total
```

对于**kw 而言，适用同样的规则。修复命名的参数，会使方法签名更有意义，如下所示。

```
>>> def make_sentence(**kw):
...     noun = kw.get('noun', 'Bill')
...     verb = kw.get('verb', 'is')
...      adj = kw.get('adjective', 'happy')
...      return '%s %s %s' % (noun, verb, adj)
>>> def make_sentence(noun='Bill', verb='is', adjective='happy'):
...     return '%s %s %s' % (noun, verb, adjective)
```

另一个有趣的方法是创建一个聚集多个相关参数以提供执行环境的容器类。这种结构与*args 或**kw 不同，因为它可以提供工作于数值之上并且能够独立演化的内部构件。将它当作参数来使用的代码将不必处理其内部构件。

例如，传递到一个函数的 Web 请求通常由一个类实例表示。这个类负责保存 Web 服务器传递的数据，如下所示。

```
>>> def log_request(request): # 版本 1
...     print request.get('HTTP_REFERER', 'No referer')
>>> def log_request(request): # 版本 2
...     print request.get('HTTP_REFERER', 'No referer')
...     print request.get('HTTP_HOST', 'No host')
```

魔法参数有时候是无法避免的，特别是在元编程的时候，例如，针对带有任何种类签名的函数的装饰器（decorator）。总地来说，当涉及仅仅位于函数中的未知数据时，魔法参数很擅长，如下所示。

```
>>> import logging
>>> def log(**context):
...     logging.info('Context is:\n%s\n' % str(context))
```

4.5 类　　名

类的名称必须简明、精确，并足以从中理解类所完成的工作。常见的一个方法是使用表示其类型或特性的后缀，例如：

- SQL**Engine**
- Mime**Types**
- String**Widget**
- Test**Case**

对于基类而言，可以使用一个 Base 或 Abstract 前缀，如：

- **Base**Cookie
- **Abstract**Formatter

最重要的是要和类特性保持一致。例如，应尝试避免类及其特性名称之间的冗余，如下所示。

```
>>> SMTP.smtp_send()      # 命名空间中存在冗余信息
>>> SMTP.send()           # 可读性更强，也更易于记忆
```

4.6 模块和包名称

在模块和包的名称中，应该体现出其内容的用途。这些名称应简短、使用小写字母，不使用下划线，例如：

- sqlite
- postgres
- sha1

如果它们实现一个协议，那么通常会使用 lib 前缀，如下所示。

```
>>> import smtplib
```

```
>>> import urllib
>>> import telnetlib
```

它们在命名空间中也必须保持一致，这样使用起来会更简单，如下所示。

```
>>> from widgets.stringwidgets import TextWidget     # 不好的
>>> from widgets.strings import TextWidget           # 更好的
```

同样，应该始终避免使用与来自标准程序库的模块相同的名称。

当一个模块变得比较复杂、包含许多类时，创建一个包并将模块的元素分到其他模块中是一个好习惯。

__init__模块也可以用来将一些 API 放回最高级别中，因为它不会影响这些 API 的使用，而且帮助将代码组织为更小的部件。例如，foo 包中的一个模块，如下所示。

```
from module1 import feature1, feature2
from module2 import feature3
```

将允许用户直接地导入功能，如下所示。

```
>>> from foo import feature1
>>> from foo import feature2, feature3
```

但是要意识到，这可能会增加循环依赖的可能性，在__init__模块中添加的代码将被实例化。所以，要小心使用。

4.7 使用 API

在前一节中，已经了解了包和模块都是一等公民，可以简化程序库或应用程序的使用。我们应该小心地组织它们，因为它们将一起创建一个 API。

本节将介绍一些关于解决以下问题的见解：

- 跟踪冗长；
- 构建命名空间树；
- 分解代码；
- 使用 deprecation 过程；
- 使用 egg。

4.7.1 跟踪冗长

创建程序库时，最常见的错误是“API 冗长（API verbosity）”。当一个功能对包的调用是

一组而不是一个时，将出现这一错误。

下面举一个允许执行一些代码的 script_engine 包的例子。

```
>>> from script_engine import make_context
>>> from script_engine import compile
>>> from script_engine import execute
>>> context = make_context({'a': 1, 'b':3})
>>> byte_code = compile('a + b')
>>> print execute(byte_code)
4
```

这个使用场景应该在一个包中的新函数之后提供，如下所示。

```
>>> from script_engine import run
>>> print run('a + b', context={'a': 1, 'b':3})
4
```

之后，低等级和高等级的函数都将可用于高等级调用和其他低等级函数的组合。

这个原则将在第 14 章中通过 Façade 设计模式来描述。

4.7.2　构建命名空间树

要组织一个应用程序的 API，一种简单的技术是通过使用场景构建一个命名空间树，并了解代码的组织方式。

我们来举个例子。一个名为 acme 的应用程序要提供一个知道如何创建 PDF 文件的引擎。它将基于一系列模板文件和一个 MySQL 数据库上的查询。

acme 应用的 3 个部分是：

- 一个 PDF 生成器；
- 一个 SQL 引擎；
- 一个模板集合。

由此，命名空间树的初稿可能是：

- acme
  - pdfgen.py
    - class PDFGen
  - sqlengine.py

    - class SQLEngine
  - templates.py
    - class Template

现在，在一个代码示例中尝试这个命名空间，并了解 PDF 如何从这个应用程序中创建。我们将猜测类和函数如何命名，并在类似于 acme 的功能的一个粘合程序中被调用，如下所示。

```
>>> from acme.templates import Template
>>> from acme.sqlengine import SQLEngine
>>> from acme.pdfgen import PDFGen
>>> SQL_URI = 'sqlite:///:memory:'
>>> def generate_pdf(query, template_name):
...     data = SQLEngine(SQL_URI).execute(query)
...     template = Template(template_name)
...     return PDFGen().create(data, template)
```

第一个版本给了一个关于命名空间可用性的反馈，而且可以被重构，以通过 API 冗长跟踪和常识来简化。

例如，PDFGen 不需要在调用者中创建，因为任何该类的实例都可以生成任何 PDF 实例。因此，它可以保持为私有的。templates 的使用也可以通过以下的风格来简化。

```
>>> from acme import templates
>>> from acme.sqlengine import SQLEngine
>>> from acme.pdf import generate
>>> SQL_URI = 'sqlite:///:memory:'
>>> def generate_pdf(query, template_name):
...     data = SQLEngine(SQL_URI).execute(query)
...     template = templates.generate(template_name)
...     return generate(data, template)
```

第二稿的命名空间将变成：

- acme
  - config.py
    - SQL_URI
  - utils.py
    - **function** generate_pdf
  - pdf.py
    - **function** generate
    - **class** _Generator

- sqlengine.py
  - **class** SQLEngine
- templates.py
  - **function** generate
  - **class** _Template

所做的修改如下：

- config.py 中包含配置元素；
- utils.py 提供高等级的 API；
- pdf.py 提供了唯一的一个函数；
- templates.py 提供一个工厂。

对于每个新的使用场景，这样的结构化改变对设计一个可用的 API 而言是有帮助的。这必须在包被发布和使用之前完成。对于已经发布的包，必须设置一个启用过程，这将在本章稍后做说明。

命名空间树必须通过实际的使用场景小心设计。第 11 章中将介绍如何通过测试来创建它。

4.7.3　分解代码

小就是美，这也适用于所有级别的代码。当一个函数、类或一个代码太大时，就应该对其进行分解。

一个函数或一个方法的内容不应该超过一个屏幕，也就是大约 25～30 行，否则它将很难跟踪和理解。

关于代码复杂性的更多信息，可以参见 Eric Raymond 所著的 *Art of Unix Programming*（中译版《Unix 编程艺术》）中的相关章节。

类的方法的数量应该有一定的限制。当方法超过 10 个时，即使创建者对其也很难做出完整的描绘。一个常见的方法是分离功能并且在该类之外创建多个类。

一个模块的大小也应该有一定的限制。当它超过 500 行时，就应该被分解为多个模块。

这个工作将会影响 API，并且意味着在包的级别上需要付出额外的工作来确保代码分解

和组织的方式不会使 API 难以使用。

换句话说，API 应该总是从用户的角度来测试，以确保它可用、易于记忆和简明。

4.7.4 使用 Egg

当应用程序不断成长时，主文件夹下的包数量也可能会变得相当大。例如，像 Zope 这样的框架在根包 zope 命名空间中就有超过 50 个包。

为了避免使整个代码库都在同一个文件夹里，并能够单独发布每个包，可以使用“Python eggs”（http://peak.telecommunity.com/DevCenter/PythonEggs）来解决。它们提供了一种构建“命名空间的包”的简单方法，就像 Java 中提供的 JAR 一样。

例如，如果希望将 acme.templates 作为单独的包分发，可以使用 setuptools（用于建立 Python Egg 的程序库）来构建一个基于 egg 的包，在 acme 文件夹中的特殊文件__init__.py 里添加以下内容（http://peak.telecommunity.com/DevCenter/setuptools#namespace-packages）。

```
try:
    __import__('pkg_resources').declare_namespace(__name__)
except ImportError:
    from pkgutil import extend_path
    __path__ = extend_path(__path__, __name__)
```

然后，acme 文件夹中可以保存 templates 文件夹，并使其能够位于 acme.templates 命名空间之下。acme.pdf 甚至可以从独立的 acme 文件夹中被分离出来。

遵循相同的规则，来自相同组织的包可以通过 egg 收集在相同的命名空间中，即使它们互不相关。例如，所有来自 Ingeniweb 的包都是使用 iw 命名空间的，并且可以使用前缀 http://pypi.python.org/pypi?%3Aaction=search&term=iw.&submit=search，在 Cheeseshop 上找到。

除了命名空间之外，以 egg 形式分发应用程序也对模块化有帮助，因为可以视每个 egg 为一个独立的组件。

第 6 章将介绍如何创建、发布和部署一个基于 egg 的应用程序。

4.7.5 使用 deprecation 过程

当包已经被发布并且被第三方代码使用时，对 API 的修改就必须小心进行了。处理这种

修改最简单的方法是遵循一个 deprecation 过程，在此是包含两个版本的中间发行版本。

例如，如果一个类有个 run_script 方法被替换成简化的 run 命令，内建的 DeprecationWarning 异常可以和 warnings 模块一起被用在中间结果中，如下所示。

```
>>> class SomeClass(object): # 版本 1
...     def run_script(self, script, context):
...         print 'doing the work'
>>> import warnings
>>> class SomeClass(object): # 版本 1.5
...     def run_script(self, script, context):
...         warnings.warn(("'run_script' will be replaced "
... "                       "by 'run' in version 2"),
...                       DeprecationWarning)
...         return self.run(script, context)
...     def run(self, script, context=None):
...         print 'doing the work'
>>> SomeClass().run_script('a script', {})
__main__:4: DeprecationWarning: 'run_script' will be replaced by 'run'
in version 2
doing the work
>>> SomeClass().run_script('a script', {})
doing the work
>>> class SomeClass(object): # version 2
...     def run(self, script, context=None):
...         print 'doing the work'
```

warnings 模块将在第一次调用时警告用户，并且忽略下一个调用。这个模块的另一个很好的功能是可以创建过滤器，用来管理影响应用程序的警告信息。例如，警告可以自动忽略或者变为异常，以进行强制修改（参见 http://docs.python.org/lib/warning-filter.html）。

4.8　有用的工具

前面的约定和方法的一部分可以使用以下的工具来控制和解决：

- Pylint　一个非常灵活的元代码分析器；
- CloneDigger　一个重复代码侦测工具。

4.8.1 Pylint

除了一些质量保证方面的度量之外，Pylint 能够检查指定的源代码是否遵循某种命名约定。它的默认设置对应于 PEP 8，一个 Pylint 脚本提供了一个 shell 报告输出。

要安装 Pylint，可以通过 easy_install 来使用 logilab.installer egg，如下所示。

```
$ easy_install logilab.pylintinstaller
```

这步之后，该命令就可用了，并且可以对一个模块或使用通配符表示的多个模块运行，如下所示。

```
$ pylint bootstrap.py
No config file found, using default configuration
************* Module bootstrap
C: 25: Invalid name "tmpeggs" (should match (([A-Z_][A-Z1-9_]*)|(__.*__
))$)
C: 27: Invalid name "ez" (should match (([A-Z_][A-Z1-9_]*)|(__.*__))$)
W: 28: Use of the exec statement
C: 34: Invalid name "cmd" (should match (([A-Z_][A-Z1-9_]*)|(__.*__))$)
C: 36: Invalid name "cmd" (should match (([A-Z_][A-Z1-9_]*)|(__.*__))$)
C: 38: Invalid name "ws" (should match (([A-Z_][A-Z1-9_]*)|(__.*__))$)
...
Global evaluation
-----------------
Your code has been rated at 6.25/10
```

注意，Pylint 总会在某些时候给出不好的评级或抱怨。例如，一个不被模块本身的代码使用的 import 语句在某些情况下也是很好的（使其在命名空间中可用）。

如果调用采用了混合大小写的方法命名的程序库，那么它也可能降低评级。在任何情况下，全局评估不像 C 中的“lint”那么重要。Pylint 只是一个指出潜在改进点的工具。

调优 Pylint 第一件要做的事情就是，使用 - generate-rcfile 选项在原始目录下创建一个.pylinrc 配置文件，如下所示。

```
$ pylint --generate-rcfile > ~/.pylintrc
```

在 Windows 下，“～”文件夹必须替换成用户文件夹，一般位于 Documents and Settings 文件夹中（参见 HOME 环境变量）。

配置文件中首先需要修改的是，在 REPORTS 小节中将 reports 变量设置为 no，以避免生成冗长的报告。在我们的例子中，只需要用这个工具侦测名称。当完成这个修改之后，这个

工具将只会显示警告，如下所示。

```
$ pylint boostrap.py
************* Module bootstrap
C: 25: Invalid name "tmpeggs" (should match (([A-Z_][A-Z1-9_]*)|(__.*__
))$)
C: 27: Invalid name "ez" (should match (([A-Z_][A-Z1-9_]*)|(__.*__))$)
W: 28: Use of the exec statement
C: 34: Invalid name "cmd" (should match (([A-Z_][A-Z1-9_]*)|(__.*__))$)
C: 36: Invalid name "cmd" (should match (([A-Z_][A-Z1-9_]*)|(__.*__))$)
C: 38: Invalid name "ws" (should match (([A-Z_][A-Z1-9_]*)|(__.*__))$)
```

4.8.2　CloneDigger

CloneDigger（http://clonedigger.sourceforge.net）是一个很好的工具，它尝试访问代码树以侦测代码中的相似之处。它基于网站上所说明的相当复杂的算法，有效地补充了 Pylint。

要安装它，可以使用 easy_install，如下所示。

```
$ easy_install CloneDigger
```

将得到一个可被用于侦测重复代码的 clonedigger 命令，命令选项可以在 http://clonedigger.sourceforge.net/documentation.html 上找到。

```
$ clonedigger html_report.py ast_suppliers.py
Parsing html_report.py ... done
Parsing ast_suppliers.py ... done
40 sequences
average sequence length: 3.250000
maximum sequence length: 14
Number of statements: 130
Calculating size for each statement... done
Building statement hash... done
Number of different hash values: 52
Building patterns... 66 patterns were discovered
Choosing pattern for each statement... done
Finding similar sequences of statements... 0 sequences were found
Refining candidates... 0 clones were found
Removing dominated clones... 0 clones were removed
```

output.html 是它生成的一个 HTML 输出，其中包含了 CloneDigger 的一个工作报告。

4.9 小　结

本章介绍了以下内容。

- PEP 8 是命名约定的绝对参考。
- 选择名称的时候应该遵循以下几条规则：为布尔元素命名时使用“has”或“is”前缀；为序列元素命名时使用复数；避免使用通用名称；避免遮蔽现有名称，尤其是内建的名称。
- 针对参数的一组好习惯是：根据设计来构建参数；不要试图使用断言实现静态类型检查；不要误用*args 和**kw。
- 使用 API 时的一些公共实践是：跟踪冗长；根据设计构建命名空间树；将代码分解为小块；为程序库在一个公共名空间下使用 egg；使用 deprecation 过程。
- 使用 Pylint 和 CloneDigger 控制代码。

下一章将说明如何编写一个包。