

第 2 章

语法最佳实践——低于类级

回头看看所写的第一个程序，你或许就会赞同此观点：正确的语法应该是很容易阅读的代码，而不好的语法则会令人感到烦恼。

除了实现的算法、贯穿于程序的架构之外，应把重心放在如何解决问题上。许多程序员决定重新写一遍代码的原因就是语法丑陋、API 不清晰或者都没有统一的命名规范。

不过 Python 在最近几年有了很大的发展，你或许会对这些新特性感到惊讶。从最早版本到当前版本（现在是 2.6），已经有了很大改进，Python 语言变得更加清晰、整洁、易于编写。Python 基础没有很大改变，但和它相关的工具有了很多人性化的改变。

本章将介绍现代语法中最重要的元素，以及它们的使用技巧，包括：

- 列表推导（List comprehensions）；
- 迭代器（Iterators）和生成器（generators）；
- 描述符（Descriptors）和属性（properties）；
- 装饰（Decorators）；
- contextlib。



速度提升、内存使用等代码性能技巧将在第 12 章中讲述。

如果在阅读本章时需要查阅 Python 相关语法，可以参考官方文档的 3 个重要组成部分：

- 命令行中的帮助功能；
- 在线教程（<http://docs.python.org/tut/tut.html>）；
- 样式指南（<http://www.python.org/dev/peps/pep-0008>）。

2.1 列表推导

编写如下所示的代码是令人痛苦的。

```
>>> numbers = range(10)
>>> size = len(numbers)
>>> evens = []
>>> i = 0
>>> while i < size:
...     if i % 2 == 0:
...         evens.append(i)
...     i += 1
...
>>> evens
[0, 2, 4, 6, 8]
```

这对于 C 语言而言或许是可行的，但是在 Python 中它确实会使程序的执行速度变慢了，因为：

- 它使解释程序在每次循环中都要确定序系中的哪一个部分被修改；
- 它使得必须通过一个计数器来跟踪必须处理的元素。

List comprehensions 是这种场景下的正确选择，它使用编排好的特性对前述语法中的一部分进行了自动化处理，如下所示。

```
>>> [i for i in range(10) if i % 2 == 0]
[0, 2, 4, 6, 8]
```

这种编写方法除了高效之外，也更加简短，涉及的元素也更少。在更大的程序中，这意味着引入的缺陷更少，代码更容易阅读和理解。

Python 风格的语法的另一个典型例子是使用 `enumerate`。这个内建函数为在循环中使用序列时提供了更加便利的获得索引的方式，例如下面这个代码块。

```
>>> i = 0
>>> seq = ["one", "two", "three"]
>>> for element in seq:
...     seq[i] = '%d: %s' % (i, seq[i])
...     i += 1
...
```

```
>>> seq
['0: one', '1: two', '2: three']
```

它可以用以下简短的代码块代替。

```
>>> seq = ["one", "two", "three"]
>>> for i, element in enumerate(seq):
...     seq[i] = '%d: %s' % (i, seq[i])
...
>>> seq
['0: one', '1: two', '2: three']
```

然后，还可以用一个 List comprehensions 将其重构如下。

```
>>> def _treatment(pos, element):
...     return '%d: %s' % (pos, element)
...
>>> seq = ["one", "two", "three"]
>>> [_treatment(i, el) for i, el in enumerate(seq)]
['0: one', '1: two', '2: three']
```

最后，这个版本的代码更容易矢量化，因为它共享了基于序列中单个项目的小函数。

Python 风格的语法意味着什么？



Python 风格的语法是一种对小代码模式最有效的语法。这个词也适用于诸如程序库这样的高级别事物上。在那种情况下，如果程序库能够很好地使用 Python 的风格，它就被认为是 Python 化（Phthonic）的。在开发社群中，这个术语有时被用来对代码块进行分类。



每当要对序列中的内容进行循环处理时，就应该尝试用 List comprehensions 来代替它。

2.2 迭代器和生成器

迭代器只不过是一个实现迭代器协议的容器对象。它基于两个方法：

- `next` 返回容器的下一个项目；
- `__iter__` 返回迭代器本身。

迭代器可以通过使用一个 `iter` 内建函数和一个序列来创建，示例如下。

```
>>> i = iter('abc')
>>> i.next()
'a'
>>> i.next()
'b'
>>> i.next()
'c'
>>> i.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

当序列遍历完时，将抛出一个 `StopIteration` 异常。这将使迭代器与循环兼容，因为它们将捕获这个异常以停止循环。要创建定制的迭代器，可以编写一个具有 `next` 方法的类，只要该类能够提供返回迭代器实例的 `__iter__` 特殊方法。

```
>>> class MyIterator(object):
...     def __init__(self, step):
...         self.step = step
...     def next(self):
...         """Returns the next element."""
...         if self.step == 0:
...             raise StopIteration
...         self.step -= 1
...         return self.step
...     def __iter__(self):
...         """Returns the iterator itself."""
...         return self
...
>>> for el in MyIterator(4):
...     print el
...
3
2
1
0
```

迭代器本身是一个底层的特性和概念，在程序中可以没有它们。但是它们为生成器这一更有趣的特性提供了基础。

2.2.1 生成器

从 Python 2.2 起，生成器提供了一种出色的方法，使得需要返回一系列元素的函数所需的代码更加简单、高效。基于 `yield` 指令，可以暂停一个函数并返回中间结果。该函数将保存执行环境并且可以在必要时恢复。

例如（这是 PEP 中关于迭代器的实例），Fibonacci 数列可以用一个迭代器来实现，如下所示。

```
>>> def fibonacci():
...     a, b = 0, 1
...     while True:
...         yield b
...         a, b = b, a + b
...
>>> fib = fibonacci()
>>> fib.next()
1
>>> fib.next()
1
>>> fib.next()
2
>>> [fib.next() for i in range(10)]
[3, 5, 8, 13, 21, 34, 55, 89, 144, 233]
```

该函数将返回一个特殊的迭代器，也就是 `generator` 对象，它知道如何保存执行环境。对它的调用是不确定的，每次都将产生序列中的下一个元素。这种语法很简洁，算法的不确定性并没有影响代码的可读性。不必提供使函数可停止的方法。实际上，这看上去像是用伪代码设计的序列一样。



PEP 的含义是 Python 增强建议 (Python Enhancement Proposal)。它是在 Python 上进行修改的文件，也是开发社团讨论的一个出发点。
进一步的信息参见<http://www.python.org/dev/peps/pep-0001>。

在开发社团中，生成器并不那么常用，因为开发人员还不习惯如此思考。开发人员多年来习惯于使用意图明确的函数。当需要一个将返回一个序列或在循环中执行的函数时，就应该考虑生成器。当这些元素将被传递到另一个函数中以进行后续处理时，一次返回一个元素能够提高整体性能。

在这种情况下，用于处理一个元素的资源通常不如用于整个过程的资源重要。因此，它们可以仍然保持位于底层，使程序更加高效。例如，Fibonacci 数列是无穷尽的，但是用来生成它的生成器不需要在提供一个值的时候，就预先占用无穷多的内存。常见的应用场景是使用生成器的流数据缓冲区。使用这些数据的第三方程序代码可以暂停、恢复和停止生成器，所有数据在开始这一过程之前不需要导入。

例如，来自标准程序库的 `tokenize` 模块将在文本之外生成令牌，并且针对每个处理过的行返回一个迭代器，这可以被传递到一些处理中，如下所示。

```
>>> import tokenize
>>> reader = open('amina.py').next
>>> tokens = tokenize.generate_tokens(reader)
>>> tokens.next()
(1, 'from', (1, 0), (1, 4), 'from amina.quality import
                                                                    similarities\n')
>>> tokens.next()
(1, 'amina', (1, 5), (1, 10), 'from amina.quality import
                                                                    similarities\n')
>>> tokens.next()
```

在此我们看到，`open` 函数遍历了文件中的每个行，而 `generate_tokens` 则在一个管道中对其进行遍历，完成一些额外的工作。

生成器对降低程序复杂性也有帮助，并且能够提升基于多个序列的数据转换算法的性能。把每个序列当作一个迭代器，然后将它们合并到一个高级别的函数中，这是一种避免函数变得庞大、丑陋、不可理解的好办法。而且，这可以给整个处理链提供实时的反馈。

在下面的示例中，每个函数用来在序列上定义一个转换。然后它们被链接起来应用。每次调用将处理一个元素并返回其结果，如下所示。

```
>>> def power(values):
...     for value in values:
...         print 'powering %s' % value
...         yield value
...
>>> def adder(values):
```

```

...     for value in values:
...         print 'adding to %s' % value
...         if value % 2 == 0:
...             yield value + 3
...         else:
...             yield value + 2
...
>>> elements = [1, 4, 7, 9, 12, 19]
>>> res = adder(power(elements))
>>> res.next()
powering 1
adding to 1
3
>>> res.next()
powering 4
adding to 4
7
>>> res.next()
powering 7
adding to 7
9

```



保持代码简单，而不是数据

拥有许多简单的处理序列值的可迭代函数，要比一个复杂的、每次计算一个值的函数更好一些。

Python引入的与生成器相关的最后一个特性是提供了与 `next` 方法调用的代码进行交互的功能。`yield` 将变成一个表达式，而一个值可以通过名为 `send` 的新方法来传递，如下所示。

```

>>> def psychologist():
...     print 'Please tell me your problems'
...     while True:
...         answer = (yield)
...         if answer is not None:
...             if answer.endswith('?'):
...                 print ("Don't ask yourself

```

```

...             "too much questions")
...         elif 'good' in answer:
...             print "A that's good, go on"
...         elif 'bad' in answer:
...             print "Don't be so negative"
...
>>> free = psychologist()
>>> free.next()
Please tell me your problems
>>> free.send('I feel bad')
Don't be so negative
>>> free.send("Why I shouldn't ?")
Don't ask yourself too much questions
>>> free.send("ok then i should find what is good for me")
A that's good, go on

```

`send` 的工作机制与 `next` 一样，但是 `yield` 将变成能够返回传入的值。因而，这个函数可以根据客户端代码来改变其行为。同时，还添加了 `throw` 和 `close` 两个函数，以完成该行为。它们将向生成器抛出一个错误：

- `throw` 允许客户端代码传入要抛出的任何类型的异常；
- `close` 的工作方式是相同的，但是将会抛出一个特定的异常——`GeneratorExit`，在这种情况下，生成器函数必须再次抛出 `GeneratorExit` 或 `StopIteration` 异常。

因此，一个典型的生成器模板应该类似于如下所示。

```

>>> def my_generator():
...     try:
...         yield 'something'
...     except ValueError:
...         yield 'dealing with the exception'
...     finally:
...         print "ok let's clean"
...
>>> gen = my_generator()
>>> gen.next()
'something'
>>> gen.throw(ValueError('mean mean mean'))
'dealing with the exception'
>>> gen.close()

```



```
ok let's clean
>>> gen.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

finally 部分在之前的版本中是不允许使用的，它将捕获任何未被捕获的 **close** 和 **throw** 调用，是完成清理工作的推荐方式。**GeneratorExit** 异常在生成器中是无法捕获的，因为它被编译器用来确定调用 **close** 时是否正常退出。如果有代码与这个异常关联，那么解释程序将抛出一个系统错误并退出。

有了这 3 个新的方法，就有可能使用生成器来编写协同程序（**coroutine**）。

2.2.2 协同程序

协同程序是可以挂起、恢复，并且有多个进入点的函数。有些语言本身就提供了这种特性，如 **Io** (<http://iolanguage.com>) 和 **Lua** (<http://www.lua.org>)，它们可以实现协同的多任务和管道机制。例如，每个协同程序将消费或生成数据，然后暂停，直到其他数据被传递。

在 **Python** 中，协同程序的替代者是线程，它可以实现代码块之间的交互。但是因为它们表现出一种抢先式的风格，所以必须注意资源锁，而协同程序不需要。这样的代码可能变得相当复杂，难以创建和调试。但是生成器几乎就是协同程序，添加 **send**、**throw** 和 **close**，其初始的意图就是为该语言提供一种类似协同程序的特性。

PEP 342 (<http://www.python.org/dev/peps/pep-0342>) 实例化了生成器的新行为，也提供了创建协同程序的调度程序的完整实例。这个模式被称为 **Trampoline**，可以被看作生成和消费数据的协同程序之间的媒介。它使用一个队列将协同程序连接在一起。

在 **PyPI** 中的 **multitask** 模块（用 **easy_install multitask** 安装）实现了这一模式，使用也十分简单，如下所示。

```
>>> import multitask
>>> import time
>>> def coroutine_1():
...     for i in range(3):
...         print 'c1'
...         yield i
...
>>> def coroutine_2():
...     for i in range(3):
```

```

...         print 'c2'
...         yield i
...
>>> multitask.add(coroutine_1())
>>> multitask.add(coroutine_2())
>>> multitask.run()
c1
c2
c1
c2
c1
c2

```

在协同程序之间的协作，最经典的例子是接受来自多个客户的查询，并将每个查询委托给对此做出响应的新线程的服务器应用程序。要使用协同程序来实现这一模式，首先要编写一个负责接收查询的协同程序（服务器），以及另一个处理它们的协同程序（句柄）。第一个协同程序在 `trampoline` 中为每个请求放置一个新的句柄。

`multitask` 包为套接字处理（如 `echo` 服务器）提供了很好的 API，通过它实现程序很简单，如下所示。

```

from __future__ import with_statement
from contextlib import closing
import socket
import multitask

def client_handler(sock):
    with closing(sock):
        while True:
            data = (yield multitask.recv(sock, 1024))
            if not data:
                break
            yield multitask.send(sock, data)

def echo_server(hostname, port):
    addrinfo = socket.getaddrinfo(hostname, port,
                                   socket.AF_UNSPEC,
                                   socket.SOCK_STREAM)
    (family, socktype, proto,
     canonname, sockaddr) = addrinfo[0]
    with closing(socket.socket(family,

```

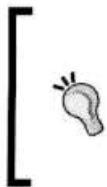
```

        socktype,
        proto)) as sock:
    sock.setsockopt(socket.SOL_SOCKET,
                     socket.SO_REUSEADDR, 1)
    sock.bind(sockaddr)
    sock.listen(5)
    while True:
        multitask.add(client_handler((
            yield multitask.accept(sock))[0]))
if __name__ == '__main__':
    import sys
    hostname = None
    port = 1111
    if len(sys.argv) > 1:
        hostname = sys.argv[1]
    if len(sys.argv) > 2:
        port = int(sys.argv[2])
    multitask.add(echo_server(hostname, port))
    try:
        multitask.run()
    except KeyboardInterrupt:
        pass

```



本章稍后会介绍 contextlib。



另一种协同程序实现

greenlet 是另一种程序库，它的特性之一就是为 Python 协同程序提供了一个良好的实现。

2.2.3 生成器表达式

Python 为编写针对序列的简单生成器提供了一种快捷方式。可以用一种类似列表推导的语法来代替 yield。在此，使用圆括号代替中括号，如下所示。

```
>>> iter = (x**2 for x in range(10) if x % 2 == 0)
>>> for el in iter:
...     print el
...
0
4
16
36
64
```

这种表达式常被称为生成器表达式或 `genexp`。它们使用类似列表推导的方式减少了序列代码的总量。它们和常规的生成器一样，每次输出一个元素。所以整个序列和列表推导一样，都不会事先进行计算。每当在 `yield` 表达式上创建简单的循环时，都应该使用它，或者用它来代替表现类似迭代器的列表推导。

2.2.4 itertools 模块

当 Python 中添加了迭代器后，就为实现常见模式提供了一个新的模块。因为它是以 C 语言编写，所以提供了最高效的迭代器，`itertools` 覆盖了许多模式，但最有趣的是 `islice`、`tee` 和 `groupby`。

1. islice：窗口迭代器

`islice` 将返回一个运行在序列的子分组之上的迭代器。以下实例将按行从标准输入中读取信息，并且输出从第 5 行开始的每行元素，只要该行的元素超过 4 个，如下所示。

```
>>> import itertools
>>> def starting_at_five():
...     value = raw_input().strip()
...     while value != '':
...         for el in itertools.islice(value.split(),
...                                     4, None):
...             yield el
...         value = raw_input().strip()
...
>>> iter = starting_at_five()
>>> iter.next()
one two three four five six
```



```

'five'
>>> iter.next()
'six'
>>> iter.next()
one two
one two three four five six
'five'
>>> iter.next()
'six'
>>> iter.next()
one
one two three four five six seven eight
'five'
>>> iter.next()
'six'
>>> iter.next()
'seven'
>>> iter.next()
'eight'

```

当需要抽取位于流中特定位置的数据时，都可以使用 `islice`。例如，可能是使用记录的特殊格式文件，或者表现元数据（如 SOAP 封套）封装的数据流。在那种情况下，`islice` 可以看作在每行数据之上的一个滑动窗口。

2. tee: 往返式的迭代器

迭代器将消费其处理的序列，但它不会往回处理。`tee` 提供了在一个序列之上运行多个迭代器的模式。如果提供第一次运行的信息，就能帮助我们再次基于这些数据运行。例如，读取文件的表头可以在运行一个处理之前提供特性信息，如下所示。

```

>>> import itertools
>>> def with_head(iterable, headsize=1):
...     a, b = itertools.tee(iterable)
...     return list(itertools.islice(a, headsize)), b
...
>>> with_head(seq)
([1], <itertools.tee object at 0x100c698>)
>>> with_head(seq, 4)
([1, 2, 3, 4], <itertools.tee object at 0x100c670>)

```

在这个函数中，如果用 `tee` 生成两个迭代器，那么第一个迭代器由 `islice` 获取该迭代的第一个 `headsize` 元素，并将它们作为一个普通列表返回；第二个迭代器返回的元素则是一个新的迭代器，可以工作在整个序列之上。

3. `groupby`: `uniq` 迭代器

这个函数有点像 Unix 命令 `uniq`。它可以对来自一个迭代器的重复元素进行分组，只要它们是相邻的，还可以提供另一个函数来执行元素的比较。否则，将采用标识符比较。

`groupby` 的一个应用实例是使用行程长度编码 (RLE) 来压缩数据。字符串中的每组相邻的重复字符将替换成该字符本身和重复次数。如果字符没有重复，则使用 1。

例如：

```
get uuuuuuuuuuuuuuuuuuuup
```

将被替代为：

```
1g1e1t1 8ulp
```

使用 `groupby` 来获取 RLE，只需要几行代码，如下所示。

```
>>> from itertools import groupby
>>> def compress(data):
...     return ((len(list(group)), name)
...             for name, group in groupby(data))
...
>>> def decompress(data):
...     return (car * size for size, car in data)
...
>>> list(compress('get uuuuuuuuuuuuuuuuuuuup'))
[(1, 'g'), (1, 'e'), (1, 't'), (1, ' '),
 (18, 'u'), (1, 'p')]
>>> compressed = compress('get uuuuuuuuuuuuuuuuuuuup')
>>> ''.join(decompress(compressed))
'get uuuuuuuuuuuuuuuuuuuup'
```

压缩算法



如果对压缩感兴趣，可以考虑 LZ77 算法。它是 RLE 的增强版本，将查找相邻的相同模式，而不是相同的字符 (<http://en.wikipedia.org/wiki/LZ77>)。

每当需要在数据上完成一个摘要的时候，都可以使用 `grouper`。这时候，内建的 `sorted` 函数就非常有用，可以使传入的数据中相似的元素相邻。

4. 其他函数

在 <http://docs.python.org/lib/itertools-functions.html> 中，可以看到 `itertools` 函数的完整列表，包括本节中未说明的函数。每个函数都附有以纯粹的 Python 编写的，理解其工作方式的对应代码。

- `chain(*iterables)` 创建一个在第一个可迭代的对象上迭代的迭代器，然后继续下一个，以此类推。
- `count([n])` 返回一个给出连续整数的迭代器，例如一个范围。当未给出 `n` 时，将从 0 开始。
- `cycle(iterable)` 在可迭代对象的每个元素之上迭代，然后重新开始，无限次地重复。
- `dropwhile(predicate, iterable)` 只要断言 (`predicate`) 为真，就从可迭代对象中删除每个元素。当断言为假时则输出剩余的元素。
- `ifilter(predicate, iterable)` 近似于内建函数 `filter`。
- `ifilterfalse(predicate, iterable)` 与 `ifilter` 类似，但是将在断言为假时执行迭代。
- `imap(function, *iterables)` 与内建函数 `map` 类似，不过它将在多个可迭代对象上工作，在最短的可迭代对象耗尽时将停止。
- `izip(*iterables)` 和 `zip` 类似，不过它将返回一个迭代器。
- `repeat(object[, times])` 返回一个迭代器，该迭代器在每次调用时返回 `object`。运行 `times` 次，如果未指定 `times` 则运行无限次。
- `starmap(function, iterable)` 和 `imap` 类似，不过它将把可迭代元素作为星号参数向 `function` 传递。这在返回元素是元组 (`tuples`) 时有帮助，它可以作为参数传递给 `function`。
- `takewhile(predicate, iterable)` 从可迭代对象返回元素，当 `predicate` 返回假时停止。

2.3 装饰器

装饰器是在 Python 2.4 中新加入的，它使得函数和方法封装（接收一个函数并返回增强版本的一个函数）更容易阅读和理解。原始的使用场景是可以将方法在定义的首部将其定义为类方法或静态方法。在添加装饰器之前，相应的语法如下。

```
>>> class WhatFor(object):
...     def it(cls):
...         print 'work with %s' % cls
```

```

...     it = classmethod(it)
...     def uncommon():
...         print 'I could be a global function'
...     uncommon = staticmethod(uncommon)
...

```

如果方法很大，或者在该方法中有多个转换时，这种语法将变得难以阅读。而使用了装饰器之后，其语法更加浅显易懂，如下所示。

```

>>> class WhatFor(object):
...     @classmethod
...     def it(cls):
...         print 'work with %s' % cls
...     @staticmethod
...     def uncommon():
...         print 'I could be a global function'
>>> this_is = WhatFor()
>>> this_is.it()
work with <class '__main__.WhatFor'>
>>> this_is.uncommon()
I could be a global function

```

在装饰器出现之后，社团中的许多开发人员开始使用它们，因为它是实现某些模式的显而易见的方法。关于这个概念最早的邮件主题是由 IronPython 的开发者 Jim Hugunin 提出的，如图 2.1 所示。

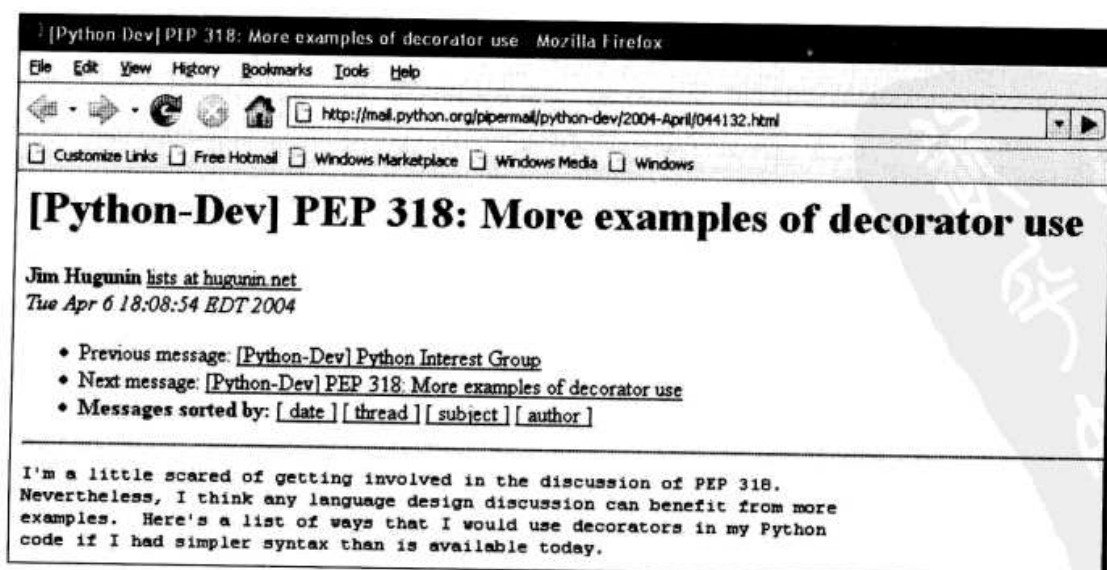


图 2.1

本节剩下的部分将介绍如何编写装饰器，并提供一些实例。

2.3.1 如何编写装饰器

编写自定义装饰器有许多方法，但最简单和最容易理解的方法是编写一个函数，返回封装原始函数调用的一个子函数。

通用的模式如下。

```
>>> def mydecorator(function):
...     def _mydecorator(*args, **kw):
...         # 在调用实际函数之前做些填充工作
...         res = function(*args, **kw)
...         # 做完某些填充工作之后
...         return res
...     # 返回子函数
...     return _mydecorator
...
```

为子函数应用一个诸如 `_mydecorator` 之类的明确的名称，而不是像 `wrapper` 这样的通用名称是一个好习惯，因为明确的名称更方便在错误发生的时候回溯——可以知道正在处理指定的装饰器。

当装饰器需要参数时，必须使用第二级封装。

```
def mydecorator(arg1, arg2):
    def _mydecorator(function):
        def __mydecorator(*args, **kw):
            # 在调用实际函数之前做些填充工作
            res = function(*args, **kw)
            # 做完某些填充工作之后
            return res
        # 返回子函数
        return __mydecorator
    return _mydecorator
```

因为装饰器在模块第一次被读取时由解释程序装入，所以它们的使用必须受限于总体上可以应用的封装器。如果装饰器与方法的类或所增强的函数签名绑定，它应该被重构为常规的可调用对象，从而避免复杂性。在任何情况下，当装饰器处理 API 时，一个好的方法是将它们聚集在一个易于维护的模块中。



装饰器应该关注于所封装的函数或方法接收和返回的参数。如果需要，应该尽可能限制其内省（introspection）工作。

常见的装饰器模式包括：

- 参数检查；
- 缓存；
- 代理；
- 上下文提供者。

2.3.2 参数检查

检查函数接收或返回的参数，在特定上下文执行时可能有用。例如，如果一个函数通过 XML-RPC 调用，Python 将不能和静态类型语言中一样直接提供它的完整签名。当 XML-RPC 客户要求函数签名时，就需要这个功能来提供内省能力。

XML-RPC 协议

XML-RPC 协议是一种轻量级的远程过程调用协议，它通过 HTTP 上的 XML 来对调用进行编码。它通常用于在简单客户-服务器交换中代替 SOAP。



和 SOAP 不同（SOAP 能够列出所有可调用函数的页面，即 WSDL），XML-RPC 没有可用函数的目录。

现在该协议已提出了一个扩展，可以用来发现服务器 API，Python 的 xmlrpclib 模块实现这个扩展（参见 <http://docs.python.org/lib/serverproxy-objects.html>）。

装饰器能提供这种签名类型，并确保输入输出与此有关，如下所示。

```
>>> from itertools import izip
>>> rpc_info = {}
>>> def xmlrpc(in_=(), out=(type(None),)):
...     def _xmlrpc(function):
...         # 注册签名
...         func_name = function.func_name
```

```

...     rpc_info[func_name] = (in_, out)
...
...     def _check_types(elements, types):
...         """Subfunction that checks the types."""
...         if len(elements) != len(types):
...             raise TypeError('argument count is wrong')
...         typed = enumerate(izip(elements, types))
...         for index, couple in typed:
...             arg, of_the_right_type = couple
...             if isinstance(arg, of_the_right_type):
...                 continue
...             raise TypeError('arg #%d should be %s' % \
...                             (index, of_the_right_type))
...
...     # 封装函数
...     def __xmlrpc(*args):      # 没有允许的关键字
...         # 检查输入的内容
...         checkable_args = args[1:] # removing self
...         _check_types(checkable_args, in_)
...         # 执行该函数
...         res = function(*args)
...
...         # 检查输出的内容
...         if not type(res) in (tuple, list):
...             checkable_res = (res,)
...         else:
...             checkable_res = res
...         _check_types(checkable_res, out)
...
...         # 函数及其类型检查成功
...         return res
...     return __xmlrpc
... return _xmlrpc
...

```

装饰器将该函数注册到全局字典中，并为其参数和返回值保存一个类型列表。注意，该函数做了很大的简化，以示范参数检查装饰器。

以下就是一个使用实例。

```
>>> class RPCView(object):
...
...     @xmlrpc((int, int)) # two int -> None
...     def meth1(self, int1, int2):
...         print 'received %d and %d' % (int1, int2)
...
...     @xmlrpc((str,), (int,)) # string -> int
...     def meth2(self, phrase):
...         print 'received %s' % phrase
...         return 12
...
...
```

当读取类定义时，将填写 `rpc_infos` 词典，并且可以将其用于一个特定的环境。此时，将检查参数类型，如下所示。

```
>>> rpc_infos
{'meth2': ((<type 'str'>,), (<type 'int'>,)),
'meth1': ((<type 'int'>, <type 'int'>),
          (<type 'NoneType'>,))}
>>> my = RPCView()
>>> my.meth1(1, 2)
received 1 and 2
>>> my.meth2(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 16, in _wrapper
  File "<stdin>", line 11, in _check_types
TypeError: arg #0 should be <type 'str'>
```

参数检查装饰器有许多其他使用场景，如类型强制（参见 <http://wiki.python.org/moin/PythonDecoratorLibrary#head-308f2b3507ca91800def19d813348f78db34303e>），可以根据给定的全局配置值来定义多种级别的类型检查：

- 什么也不检查；
- 检查器仅弹出警告；
- 检查器将抛出 `TypeError` 异常。

2.3.3 缓存

缓存装饰器与参数检查很相似，不过它关注于内部状态而不影响输出的函数。每组参数

可以链接到一个唯一结果上。这种编程风格是函数型编程的特性（参见 http://en.wikipedia.org/wiki/Functional_programming），当输入值有限时可以使用。

因此，缓存装饰器可以将输出与计算它所需的参数放在一起，并且直接在后续的调用中返回它。这种行为被称为 Memoizing（参见 <http://en.wikipedia.org/wiki/Memoizing>，常译为自动缓存），很容易被实现为一个装饰器，如下所示。

```
>>> import time
>>> import hashlib
>>> import pickle
>>> from itertools import chain
>>> cache = {}
>>> def is_obsolete(entry, duration):
...     return time.time() - entry['time'] > duration
...
>>> def compute_key(function, args, kw):
...     key = pickle.dumps((function.func_name, args, kw))
...     return hashlib.sha1(key).hexdigest()
...
>>> def memoize(duration=10):
...     def _memoize(function):
...         def __memoize(*args, **kw):
...             key = compute_key(function, args, kw)
...
...             # 是否已经拥有它了？
...             if (key in cache and
...                 not is_obsolete(cache[key], duration)):
...                 print 'we got a winner'
...                 return cache[key]['value']
...
...             # 计算
...             result = function(*args, **kw)
...
...             # 保存结果
...             cache[key] = {'value': result,
...                             'time': time.time()}
...             return result
...         return __memoize
...     return _memoize
```

```
... return _memoize
...
```

SHA hash 键值使用已排序的参数值建立，该结果将保存在一个全局字典中。hash 使用一个 pickle 来建立，这是冻结所有作为参数传递的对象状态，以确保所有参数均为良好候选者的一个快捷方式。例如，如果一个线程或套接字被用作一个参数，将抛出一个 `PicklingError`。

`duration` 参数用于在上次函数调用之后，使存在太久的缓存值失效。

以下是一个使用实例。

```
>>> @memoize()
... def very_very_very_complex_stuff(a, b):
...     # 如果执行该计算会让计算机过热，请考虑停止它
...     return a + b
...
>>> very_very_very_complex_stuff(2, 2)
4
>>> very_very_very_complex_stuff(2, 2)
we got a winner
4
>>> @memoize(1) # 1 秒之后令缓存失效
... def very_very_very_complex_stuff(a, b):
...     return a + b
...
>>> very_very_very_complex_stuff(2, 2)
4
>>> very_very_very_complex_stuff(2, 2)
we got a winner
4
>>> cache
{'c2727f43c6e39b3694649ee0883234cf': {'value': 4, 'time':
1199734132.7102251}}
>>> time.sleep(2)
>>> very_very_very_complex_stuff(2, 2)
4
```

注意，由于第一个调用是两级封装，所以使用了空的括号。

应用了缓存的函数可以显著地提高程序的总体性能，但是必须小心使用。缓存值可能绑定到函数本身上，以管理其范围和生命周期，代替集中化的字典。但是在任何情况下，更有


```
...     @protect('admin')
...     def waffle_recipe(self):
...         print 'use tons of butter!'
...
>>> these_are = MySecrets()
>>> user = tarek
>>> these_are.waffle_recipe()
use tons of butter!
>>> user = bill
>>> these_are.waffle_recipe()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 7, in wrap
__main__.Unauthorized: I won't tell you
```

2.3.5 上下文提供者

上下文装饰器用来确保函数可以运行在正确的上下文中, 或者在函数前后执行一些代码。换句话说, 它用来设置或复位特定的执行环境。例如, 当一个数据项必须与其他线程共享时, 就需要使用一个锁来确保它在多重访问时得到保护。这个锁可以在一个装饰器中编写, 示例如下。

```
>>> from threading import RLock
>>> lock = RLock()
>>> def synchronized(function):
...     def _synchronized(*args, **kw):
...         lock.acquire()
...         try:
...             return function(*args, **kw)
...         finally:
...             lock.release()
...     return _synchronized
>>> @locker
... def thread_safe(): # make sure it locks the resource
...     pass
...
```


上下文装饰器可以使用 Python 2.5 中新添加的 `with` 语句来代替。创造这条语句的目的是使 `try..finally` 模式更加流畅，在某些情况下，它覆盖了上下文装饰器的使用场景。

要想了解更多装饰器的使用场景，可浏览 <http://wiki.python.org/moin/PythonDecoratorLibrary>。


2.4 with 和 contextlib

对于要确保即使发生一个错误时也能运行一些清理代码而言，`try...finally` 语句是很有用的。对此有许多使用场景，例如：

- 关闭一个文件；
- 释放一个锁；
- 创建一个临时的代码补丁；
- 在特殊环境中运行受保护的代码。

`with` 语句覆盖了这些使用场景，为在一个代码块前后调用一些代码提供了一种简单的方法。例如，使用一个文件通常可以如下实现。

```
>>> hosts = file('/etc/hosts')
>>> try:
...     for line in hosts:
...         if line.startswith('#'):
...             continue
...         print line
... finally:
...     hosts.close()
...
127.0.0.1      localhost
255.255.255.255 broadcasthost
::1           localhost
```

 本示例仅针对 Linux，因为它将在 `etc` 文件夹中读取主机文件，不过任何文本文件都可以以相同方式被使用。

通过使用 `with` 语句，以上代码可以重写如下。

```
>>> from __future__ import with_statement
>>> with file('/etc/hosts') as hosts:
...     for line in hosts:
...         if line.startswith('#'):
...             continue
...         print host
...
127.0.0.1          localhost
255.255.255.255    broadcasthost
::1               localhost
```

注意，在 2.5 系列版本中 `with` 语句仍然位于 `__future__` 模块里，而在 2.6 系列版本中则可以直接可用。它的相关描述在 <http://www.python.org/dev/peps/pep-0343> 中可以找到。

与这条语句兼容的其他项目是来自 `thread` 和 `threading` 模块的类：

- `thread.LockType`
- `threading.Lock`
- `threading.RLock`
- `threading.Condition`
- `threading.Semaphore`
- `threading.BoundedSemaphore`

这些类都实现了两个方法——`__enter__` 和 `__exit__`，这都来自于 `with` 协议。换句话说，任何类都可以实现为如下所示。

```
>>> class Context(object):
...     def __enter__(self):
...         print 'entering the zone'
...     def __exit__(self, exception_type, exception_value,
...                   exception_traceback):
...         print 'leaving the zone'
...         if exception_type is None:
...             print 'with no error'
...         else:
...             print 'with an error (%s)' % exception_value
...
>>> with Context():
...     print 'i am the zone'
...
```

```

entering the zone
i am the zone
leaving the zone
with no error
>>> with Context():
...     print 'i am the buggy zone'
...     raise TypeError('i am the bug')
...
entering the zone
i am the buggy zone
leaving the zone
with an error (i am the bug)
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
TypeError: i am the bug

```

`__exit__` 将获取代码块中发生错误时填入的 3 个参数。如果没有发生错误，那么这 3 个参数都将被设置为 `None`。当发生一个错误时，`__exit__` 不应该重新抛出这个错误，因为这是调用者的责任。但是，它可以通过返回 `True` 来避免抛出该异常。这用来实现一些特殊的使用场景，例如下一小节中将会看到的 `contextmanager` 装饰器。但是对于大部分使用场景而言，这个方法的行为是执行与 `finally` 类似的清理工作；不管代码块中发生什么，它都不返回任何东西。

2.4.1 contextlib 模块

为了给 `with` 语句提供一些辅助类，标准程序库中添加了一个模块。最有用的辅助类是 `contextmanager`，这是一个装饰器，它增强了包含以 `yield` 语句分开的 `__enter__` 和 `__exit__` 两部分的生成器。使用这个装饰器来编写前面的实例，可以写成如下所示。

```

>>> from contextlib import contextmanager
>>> from __future__ import with_statement
>>> @contextmanager
... def context():
...     print 'entering the zone'
...     try:
...         yield
...     except Exception, e:

```

```

...         print 'with an error (%s)' % e
...         # 在此需要重新抛出错误
...         raise e
...     else:
...         print 'with no error'
...

```

如果发生任何异常，该函数需要重新抛出这个异常，以便传递它。注意，`context` 在需要时可以有的一些参数，只要它们在调用中提供这些参数即可。这个小的辅助类简化了常规的基于类的上下文 API，正如生成器使用基于类的迭代器 API 所做的一样。

这个模块提供两个其他辅助类：

- `closing(element)` 这是一个由 `contextmanager` 装饰的函数，它将输出一个元素，然后在退出时调用该元素的 `close` 方法。例如，这对于处理流的类而言就很有帮助。
- `nested(context1, context2, ...)` 这是一个合并上下文并使用它们创建嵌套的 `with` 调用的函数。

2.4.2 上下文实例

`with` 语句有一种有趣的用法，就是在进入上下文时记录可以被装饰的代码，然后在它结束时将其改回原来的样子。这避免对代码本身进行修改，例如，允许一个单元测试得到关于代码使用的一些反馈。

下面的例子中创建了一个上下文，以装备指定类的所有公共 API。

```

>>> import logging
>>> from __future__ import with_statement
>>> from contextlib import contextmanager
>>> @contextmanager
... def logged(klass, logger):
...     # 记录器
...     def _log(f):
...         def __log(*args, **kw):
...             logger(f, args, kw)
...             return f(*args, **kw)
...         return __log
...
...     # 装备该类
...     for attribute in dir(klass):

```



```

...     if attribute.startswith('_'):
...         continue
...     element = getattr(klass, attribute)
...     setattr(klass, '__logged_%s' % attribute, element)
...     setattr(klass, attribute, _log(element))
...
...     # 正常工作
...     yield klass
...
...     # 移除日志
...     for attribute in dir(klass):
...         if not attribute.startswith('__logged_'):
...             continue
...             element = getattr(klass, attribute)
...             setattr(klass, attribute[len('__logged_'):],
...                     element)
...             delattr(klass, attribute)
...

```

记录器函数之后可以被用于记录指定上下文中调用的 API。在下面的例子中，调用被添加到一个列表中以跟踪 API 的使用，然后用于执行一些断言。例如，如果以下 API 被调用超过一次，它可能意味着类的公共签名可以重构，以避免重复调用。

```

>>> class One(object):
...     def _private(self):
...         pass
...     def one(self, other):
...         self.two()
...         other.thing(self)
...         self._private()
...     def two(self):
...         pass
...
>>> class Two(object):
...     def thing(self, other):
...         other.two()
...
>>> calls = []

```

```
>>> def called(meth, args, kw):
...     calls.append(meth.im_func.func_name)
...
>>> with logged(One, called):
...     one = One()
...     two = Two()
...     one.one(two)
...
>>> calls
['one', 'two', 'two']
```

2.5 小 结

本章介绍了以下内容。

- 如果要对现有可迭代的对象做一些处理，然后生成新的列表，那么列表推导将是最便利的方法。
- 迭代器和生成器提供了一组高效的生成和处理序列的工具。
- 对于包装具有附加行为的现有函数和方法而言，装饰器提供了一种易于理解的方法。它带来了实现和使用都很简单的新的编码模式。
- with 语句改善了 try..finally 模式。

下一章仍将介绍语法最佳实践，不过将专注于类。

