14.13 给你的程序做性能测试¶

问题¶

你想测试你的程序运行所花费的时间并做性能测试。

解决方案¶

如果你只是简单的想测试下你的程序整体花费的时间， 通常使用Unix时间函数就行了，比如：

```
bash % time python3 someprogram.py
real 0m13.937s
user 0m12.162s
sys  0m0.098s
bash %
```

如果你还需要一个程序各个细节的详细报告，可以使用 `cProfile` 模块：

```
bash % python3 -m cProfile someprogram.py
        859647 function calls in 16.016 CPU seconds

  Ordered by: standard name

  ncalls  tottime  percall  cumtime  percall filename:lineno(function)
  263169    0.080    0.000    0.080    0.000 someprogram.py:16(frange)
     513    0.001    0.000    0.002    0.000 someprogram.py:30(generate_mandel)
  262656    0.194    0.000   15.295    0.000 someprogram.py:32(<genexpr>)
       1    0.036    0.036   16.077   16.077 someprogram.py:4(<module>)
  262144   15.021    0.000   15.021    0.000 someprogram.py:4(in_mandelbrot)
       1    0.000    0.000    0.000    0.000 os.py:746(urandom)
       1    0.000    0.000    0.000    0.000 png.py:1056(_readable)
       1    0.000    0.000    0.000    0.000 png.py:1073(Reader)
       1    0.227    0.227    0.438    0.438 png.py:163(<module>)
     512    0.010    0.000    0.010    0.000 png.py:200(group)
    ...
bash %
```

不过通常情况是介于这两个极端之间。比如你已经知道代码运行时在少数几个函数中花费了绝大部分时间。 对于这些函数的性能测试，可以使用一个简单的装饰器：

```
# timethis.py

import time
from functools import wraps

def timethis(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.perf_counter()
        r = func(*args, **kwargs)
        end = time.perf_counter()
        print('{}.{} : {}'.format(func.__module__, func.__name__, end - start))
        return r
    return wrapper
```

要使用这个装饰器，只需要将其放置在你要进行性能测试的函数定义前即可，比如：

```
>>> @timethis
... def countdown(n):
...     while n > 0:
...         n -= 1
...
>>> countdown(10000000)
__main__.countdown : 0.803001880645752
>>>
```

要测试某个代码块运行时间，你可以定义一个上下文管理器，例如：

```
from contextlib import contextmanager

@contextmanager
def timeblock(label):
    start = time.perf_counter()
    try:
        yield
    finally:
        end = time.perf_counter()
        print('{} : {}'.format(label, end - start))
```

下面是使用这个上下文管理器的例子：

```
>>> with timeblock('counting'):
...     n = 10000000
...     while n > 0:
...         n -= 1
...
counting : 1.5551159381866455
>>>
```

对于测试很小的代码片段运行性能，使用 `timeit` 模块会很方便，例如：

```
>>> from timeit import timeit
>>> timeit('math.sqrt(2)', 'import math')
0.1432319980012835
>>> timeit('sqrt(2)', 'from math import sqrt')
0.10836604500218527
>>>
```

`timeit` 会执行第一个参数中语句100万次并计算运行时间。 第二个参数是运行测试之前配置环境。如果你想改变循环执行次数， 可以像下面这样设置 `number` 参数的值：

```
>>> timeit('math.sqrt(2)', 'import math', number=10000000)
1.434852126003534
>>> timeit('sqrt(2)', 'from math import sqrt', number=10000000)
1.0270336690009572
>>>
```

讨论¶

当执行性能测试的时候，需要注意的是你获取的结果都是近似值。 `time.perf_counter()` 函数会在给定平台上获取最高精度的计时值。不过，它仍然还是基于时钟时间，很多因素会影响到它的精确度，比如机器负载。如果你对于执行时间更感

的计时值。不过，它仍然还是基于时钟时间，很多因素会影响到它的精确度，比如机器负载。如果你对于执行时间更感兴趣，使用 time.process_time() 来代替它。例如：

```
from functools import wraps
def timethis(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.process_time()
        r = func(*args, **kwargs)
        end = time.process_time()
        print('{}.{} : {}'.format(func.__module__, func.__name__, end - start))
        return r
    return wrapper
```

最后，如果你想进行更深入的性能分析，那么你需要详细阅读 time 、 timeit 和其他相关模块的文档。这样你可以理解和平台相关的差异以及一些其他陷阱。还可以参考13.13小节中相关的一个创建计时器类的例子。