14.14 加速程序运行¶

问题¶

你的程序运行太慢，你想在不使用复杂技术比如C扩展或JIT编译器的情况下加快程序运行速度。

解决方案¶

关于程序优化的第一个准则是“不要优化”，第二个准则是“不要优化那些无关紧要的部分”。 如果你的程序运行缓慢，首先你得使用14.13小节的技术先对它进行性能测试找到问题所在。

通常来讲你会发现你得程序在少数几个热点地方花费了大量时间， 比如内存的数据处理循环。一旦你定位到这些点，你就可以使用下面这些实用技术来加速程序运行。

使用函数

很多程序员刚开始会使用Python语言写一些简单脚本。 当编写脚本的时候，通常习惯了写毫无结构的代码，比如：

```
# somescript.py

import sys
import csv

with open(sys.argv[1]) as f:
    for row in csv.reader(f):

        # Some kind of processing
        pass
```

很少有人知道，像这样定义在全局范围的代码运行起来要比定义在函数中运行慢的多。 这种速度差异是由于局部变量和全局变量的实现方式（使用局部变量要更快些）。 因此，如果你想让程序运行更快些，只需要将脚本语句放入函数中即可：

```
# somescript.py
import sys
import csv

def main(filename):
    with open(filename) as f:
        for row in csv.reader(f):
            # Some kind of processing
            pass

main(sys.argv[1])
```

速度的差异取决于实际运行的程序，不过根据经验，使用函数带来15-30%的性能提升是很常见的。

尽可能去掉属性访问

每一次使用点(.)操作符来访问属性的时候会带来额外的开销。 它会触发特定的方法，比如 `__getattribute__()` 和 `__getattr__()` ，这些方法会进行字典操作操作。

通常你可以使用 `from module import name` 这样的导入形式，以及使用绑定的方法。 假设你有如下的代码片段：

```
import math

def compute_roots(nums):
    result = []
    for n in nums:
        result.append(math.sqrt(n))
    return result

# Test
nums = range(1000000)
for n in range(100):
    r = compute_roots(nums)
```

在我们机器上面测试的时候，这个程序花费了大概40秒。现在我们修改 `compute_roots()` 函数如下：

```
from math import sqrt

def compute_roots(nums):

    result = []
    result_append = result.append
    for n in nums:
        result_append(sqrt(n))
    return result
```

修改后的版本运行时间大概是29秒。唯一不同之处就是消除了属性访问。 用 `sqrt()` 代替了 `math.sqrt()` 。 `The result.append()` 方法被赋给一个局部变量 `result_append` ，然后在内部循环中使用它。

不过，这些改变只有在大量重复代码中才有意义，比如循环。 因此，这些优化也只是在某些特定地方才应该被使用。

理解局部变量

之前提过，局部变量会比全局变量运行速度快。 对于频繁访问的名称，通过将这些名称变成局部变量可以加速程序运行。 例如，看下之前对于 `compute_roots()` 函数进行修改后的版本：

```
import math

def compute_roots(nums):
    sqrt = math.sqrt
    result = []
    result_append = result.append
    for n in nums:
        result_append(sqrt(n))
    return result
```

在这个版本中， `sqrt` 从 `match` 模块被拿出并放入了一个局部变量中。 如果你运行这个代码，大概花费25秒（对于之前29秒又是一个改进）。 这个额外的加速原因是因为对于局部变量 `sqrt` 的查找要快于全局变量 `sqrt`

对于类中的属性访问也同样适用于这个原理。 通常来讲，查找某个值比如 `self.name` 会比访问一个局部变量要慢一些。 在内部循环中，可以将某个需要频繁访问的属性放入到一个局部变量中。例如：

```
# Slower
class SomeClass:
    ...
    def method(self):
        for x in s:
            op(self.value)

# Faster
class SomeClass:

    ...
    def method(self):
        value = self.value
        for x in s:
            op(value)
```

避免不必要的抽象

任何时候当你使用额外的处理层（比如装饰器、属性访问、描述器）去包装你的代码时，都会让程序运行变慢。 比如看下如下的这个类：

```
class A:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    @property
    def y(self):
        return self._y
    @y.setter
    def y(self, value):
        self._y = value
```

现在进行一个简单测试：

```
>>> from timeit import timeit
>>> a = A(1,2)
>>> timeit('a.x', 'from __main__ import a')
0.07817923510447145
>>> timeit('a.y', 'from __main__ import a')
0.35766440676525235
>>>
```

可以看到，访问属性y相比属性x而言慢的不止一点点，大概慢了4.5倍。 如果你在意性能的话，那么就需要重新审视下对于y的属性访问器的定义是否真的有必要了。 如果没有必要，就使用简单属性吧。 如果仅仅是因为其他编程语言需要使用getter/setter函数就去修改代码风格，这个真的没有必要。

使用内置的容器

内置的数据类型比如字符串、元组、列表、集合和字典都是使用C来实现的，运行起来非常快。 如果你想自己实现新的数据结构（比如链接列表、平衡树等）， 那么要想在性能上达到内置的速度几乎不可能，因此，还是乖乖的使用内置的吧。

避免创建不必要的数据结构或复制

有时候程序员想显摆下，构造一些并没有必要的数据结构。例如，有人可能会像下面这样写：

```
values = [x for x in sequence]
squares = [x*x for x in values]
```

也许这里的想法是首先将一些值收集到一个列表中，然后使用列表推导来执行操作。 不过，第一个列表完全没有必要，可以简单的像下面这样写：

```
squares = [x*x for x in sequence]
```

与此相关，还要注意下那些对Python的共享数据机制过于偏执的程序所写的代码。 有些人并没有很好的理解或信任Python的内存模型，滥用 `copy.deepcopy()` 之类的函数。 通常在这些代码中是可以去掉复制操作的。

讨论¶

在优化之前，有必要先研究下使用的算法。 选择一个复杂度为 O(n log n) 的算法要比你去调整一个复杂度为 O(n**2) 的算法所带来的性能提升要大得多。

如果你觉得你还是得进行优化，那么请从整体考虑。 作为一般准则，不要对程序的每一个部分都去优化,因为这些修改会导致代码难以阅读和理解。 你应该专注于优化产生性能瓶颈的地方，比如内部循环。

你还要注意微小优化的结果。例如考虑下面创建一个字典的两种方式：

```
a = {
   'name' : 'AAPL',
   'shares' : 100,
   'price' : 534.22
}

b = dict(name='AAPL', shares=100, price=534.22)
```

后面一种写法更简洁一些（你不需要在关键字上输入引号）。 不过，如果你将这两个代码片段进行性能测试对比时，会发现使用 `dict()` 的方式会慢了3倍。 看到这个，你是不是有冲动把所有使用 `dict()` 的代码都替换成第一种。 不够，聪明的程序员只会关注他应该关注的地方，比如内部循环。在其他地方，这点性能损失没有什么影响。

如果你的优化要求比较高，本节的这些简单技术满足不了，那么你可以研究下基于即时编译（JIT）技术的一些工具。 例如，PyPy工程是Python解释器的另外一种实现，它会分析你的程序运行并对那些频繁执行的部分生成本机机器码。 它有时候能极大的提升性能，通常可以接近C代码的速度。 不过可惜的是，到写这本书位置，PyPy还不能完全支持Python3. 因此，这个是你将来需要去研究的。你还可以考虑下Numba工程， Numba是一个在你使用装饰器来选择Python函数进行优化时的动态编译器。 这些函数会使用LLVM被编译成本地机器码。它同样可以极大的提升性能。 但是，跟PyPy一样，它对于Python 3的支持现在还停留在实验阶段。

最后我引用John Ousterhout说过的话作为结尾：“最好的性能优化是从不工作到工作状态的迁移”。 直到你真的需要优化的时候再去考虑它。确保你程序正确的运行通常比让它运行更快要更重要一些（至少开始是这样的）.