10 Производительность и оптимизация

Оптимизация — это последнее, о чем задумываются во время разработки, но обязательно настанет тот момент, когда это будет необходимо. Это не значит, что вы должны писать программу с мыслью, что она будет медленной, однако думать об оптимизации без предварительного определения правильных инструментов и механизмов — пустая трата времени. Как писал Дональд Кнут, «Premature optimization is the root of all evil».[1]

В этом разделе мы рассмотрим правильный подход для быстрого создания кода и поймем, что нужно оптимизировать в первую очередь. В этой главе представлена вся необходимая информация для понимания профилирования приложения с целью удаления частей кода, которые могут тормозить выполнение программы.

Структуры данных

Большинство проблем программирования могут быть решены элегантно и просто с помощью правильных структур данных — Python предоставляет большой выбор. Научиться пользоваться этими структурами гораздо полезнее для получения более стабильного и чистого кода, чем постоянное написание пользовательских структур.

Например, все используют `dict`, но как часто встречается код, который обращается к словарю, если обнаружена ошибка `KeyError`?

```
def get_fruits(basket, fruit):
    try:
        return basket[fruit]
```

[1] «Преждевременная оптимизация — это корень всего зла». Donald Knuth, *Structured Programming with go to Statements*, ACM Computing Surveys 6, no. 4 (1974): 261–301.

```
    except KeyError:
        return None
```

Или проверяет сначала наличие ключа:

```
def get_fruits(basket, fruit):
    if fruit in basket:
        return basket[fruit]
```

Если бы вы использовали метод `get()`, уже предоставленный классом `dict`, то смогли бы избежать необходимости искать ошибки и проверять наличие ключа с самого начала:

```
def get_fruits(basket, fruit):
    return basket.get(fruit)
```

Метод `dict.get()` также может возвращать значение по умолчанию вместо `None`; просто вызовите его со вторым аргументом:

```
def get_fruits(basket, fruit):
    # Return the fruit, or Banana if the fruit cannot be found.
    return basket.get(fruit, Banana())
```

Многие разработчики используют базы структур данных Python, не зная обо всех методах, которые они предоставляют. Это также верно для множеств: методы в них могут решить многие проблемы, которые иначе потребовали бы написания вложенных блоков `for/if`. Например, разработчики часто используют циклы `for/if` для определения того, входит ли элемент в список:

```
def has_invalid_fields(fields):
    for field in fields:
        if field not in ['foo', 'bar']:
            return True
    return False
```

Цикл производит итерацию через все элементы в списке и проверяет, что все они `foo` или `bar`. Но этот же цикл можно выразить гораздо эффективнее, убрав итерацию:

```
def has_invalid_fields(fields):
    return bool(set(fields) - set(['foo', 'bar']))
```

Этот код конвертирует `fields` во множество и вычитает из него множество `set(['foo'm'bar'])`. Затем переводит полученное множество в булево значение,

которое принимает значение `True`, если во множестве не осталось элементов. Использование множеств в данном случае не требует итераций и проверки элементов поочередно. Всего лишь одна операция с множествами выполняется средствами Python гораздо быстрее.

У Python есть более продвинутые структуры данных, которые снижают нагрузку на поддержание кода. Пример в листинге 10.1.

Листинг 10.1. Добавление записи в словарь из множеств

```
def add_animal_in_family(species, animal, family):
    if family not in species:
        species[family] = set()
    species[family].add(animal)

species = {}
add_animal_in_family(species, 'cat', 'felidea')
```

Этот код рабочий, но сколько раз понадобится повторять его в программе? Десятки? Сотни?

Python обеспечивает структуру `collections.defaultdict`, которая решает проблему в более элегантном виде:

```
import collections

def add_animal_in_family(species, animal, family):
    species[family].add(animal)

species = collections.defaultdict(set)
add_animal_in_family(species, 'cat', 'felidea')
```

Каждый раз при попытке обратиться к несуществующему элементу в `dict` `defaultdict` будет использовать функцию, которая передается как аргумент в свой конструктор для создания нового значения вместо вызова `KeyError`. В этом случае функция `set()` используется для создания нового `set` каждый раз, когда он нужен.

Модуль `collections` предлагает еще несколько структур данных, которые можно использовать в работе. Например, необходимо посчитать количество определенных элементов в итерируемом объекте. Для этого можно использовать метод `collection.Counter()`, решающий эту задачу:

```
>>> import collections
>>> c = collections.Counter("Premature optimization is the root of all evil.")
```

```
>>> c
>>> c['P'] # Возвращает количество букв 'P'
1
>>> c['e'] # Возвращает количество букв 'e'
4
>>> c.most_common(2) # Возвращает две самые распространенные буквы
[(' ', 7), ('i', 5)]
```

Объект `collection.Counter` работает с любыми итерируемыми объектами, которые имеют хешируемые элементы, что избавляет от необходимости писать собственные функции подсчета. Он может легко посчитать количество букв в строке и вернуть количество наиболее распространенных элементов. Возможно, вы создавали такую функцию самостоятельно, потому что не знали о существовании готового решения в стандартной библиотеке.

С правильной структурой данных, корректными методами и очевидно адекватным алгоритмом у программы будет хорошая производительность. Однако если программа работает недостаточно хорошо, лучший способ узнать ее узкие места — заняться профилированием кода.

Понимание поведения кода через профилирование

Профилирование — это форма динамического анализа, позволяющего понять, как работает программа. Оно помогает определить, где находится «бутылочное горлышко» программы и где следует применять оптимизацию. Профиль программы принимает вид множества статистических данных, описывающих частоту и продолжительность выполнения разных частей программы.

Python обеспечивает несколько инструментов для профилирования программ. Один из них входит в стандартную библиотеку и не требует установки — это `cProfile`. Также стоит рассмотреть модуль `dis`, который позволяет разбить код на более мелкие части и облегчить понимание внутренних процессов его функционирования.

cProfile

По умолчанию `cProfile` включается с версии Python 2.5. Для использования `cProfile` вызовите его с помощью программы, используя синтаксис `python`

`-m cProfile <program>`. Это загрузит и активирует модуль `cProfile`, а затем запустит программу с включенным инструментарием (листинг 10.2).

Листинг 10.2. Выходные данные cProfile

```
$ python -m cProfile myscript.py
         343 function calls (342 primitive calls) in 0.000 seconds

   Ordered by: standard name
ncalls  tottime  percall  cumtime  percall  filename:lineno(function)
     1    0.000    0.000    0.000    0.000  :0(_getframe)
     1    0.000    0.000    0.000    0.000  :0(len)
   104    0.000    0.000    0.000    0.000  :0(setattr)
     1    0.000    0.000    0.000    0.000  :0(setprofile)
     1    0.000    0.000    0.000    0.000  :0(startswith)
   2/1    0.000    0.000    0.000    0.000  <string>:1(<module>)
     1    0.000    0.000    0.000    0.000  StringIO.py:30(<module>)
     1    0.000    0.000    0.000    0.000  StringIO.py:42(StringIO)
```

Листинг 10.2 отражает выходные данные запуска простейшего скрипта на `cProfile`. Он показывает количество раз запуска каждой функции в программе и продолжительность ее работы. Также можно использовать опцию `-s` для сортировки по любым полям: например, `-s time` отсортирует результаты по внутреннему времени.

Можно визуализировать информацию, сгенерированную `cProfile`, используя отличный инструмент KCacheGrind. Он был создан для обработки программ, написанных на C, но, к счастью, его можно использовать с Python, переводя данные для вызова дерева.

Модуль `cProfile` имеет опцию `-o`, которая позволяет сохранять данные профилирования, а `pyprof2calltree` может конвертировать их из одного формата в другой. Для начала установим `pyprof2calltree`:

```
$ pip install pyprof2calltree
```

Затем запустим его для обеих операций, как показано в листинге 10.3: перевод данных (опция `-i`) и запуск KCacheGrind (опция `-k`).

Листинг 10.3. Запуск cProfile и KCacheGrind

```
$ python -m cProfile -o myscript.cprof myscript.py
$ pyprof2calltree -k -i myscript.cprof
```

При открытии KCacheGrind выведется информация (рис. 10.1). С визуализированными данными можно вызвать график для отслеживания процентов

затраченного на каждую функцию времени, что позволит определить, какая часть программы расходует слишком много ресурсов.

Самый простой способ прочитать отчет KCacheGrind — это начать с таблицы слева, в которой перечислены все функции и методы, выполненные программой. Можно отсортировать их по времени выполнения, а затем определить функцию, которая затратила больше всего вычислительных мощностей.

Правая панель KCacheGrind выводит информацию о функциях, которые вызвали рассматриваемую функцию, и о количестве этих запросов, а также о функциях, которые были вызваны рассматриваемой функцией. График запросов программы, включая время выполнения каждой части, прост в навигации.

Это поможет лучше понять, какие части кода требуют оптимизации. Как оптимизировать код — зависит уже от целей, которые программа должна выполнять.

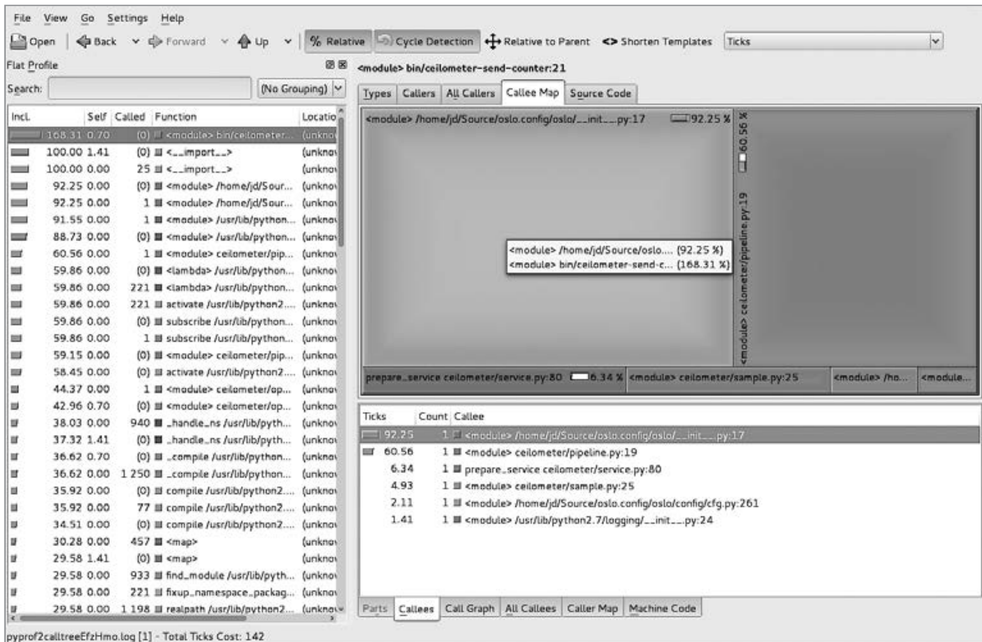

Рис 10.1. Пример выходных данных KCacheGrind

Обзор полученной информации о работе программы предоставляет макроскопический портрет ее выполнения, но иногда нужно вникнуть в более мелкие детали

работы и проинспектировать код на микроскопическом уровне. В таком случае лучше воспользоваться модулем `dis` для оценки процессов внутри программы.

Дизассемблинг модулем dis

Модуль `dis` — это дизассемблер байт-кода Python. Разобрать код на составляющие может быть полезно для понимания работы каждой строки с целью оптимизации выполнения программы. Например, листинг 10.4 показывает работу функции `dis.dis()`, которая проводит дизассемблинг любой функции, переданной в `dis()` в качестве параметра, и выводит список инструкций на байт-коде.

Листинг 10.4. Дизассемблинг функции

```
>>> def x():
...     return 42
...
>>> import dis
>>> dis.dis(x)
  2       0 LOAD_CONST               1 (42)
          3 RETURN_VALUE
```

В листинге 10.4 функция `x` проходит дизассемблинг и ее содержимое выводится в виде инструкций байт-кода. В инструкции всего две операции: загрузка константы (`LOAD_CONST`), то есть `42`, и возврат этого значения (`RETURN_VALUE`).

Чтобы продемонстрировать пользу `dis`, объявим две функции, которые выполняют одну и ту же работу — конкатенацию трех букв, — и разберем их, чтобы показать разницу в их решениях:

```
abc = ('a', 'b', 'c')

def concat_a_1():
    for letter in abc:
            abc[0] + letter

def concat_a_2():
    a = abc[0]
    for letter in abc:
            a + letter
```

Обе функции делают одно и то же, но если подвергнуть их дизассемблингу с помощью `dis.dis`, как показано в листинге 10.5, становится видно, что их байт-код разный:

Листинг 10.5. Оценка функций

```
>>> dis.dis(concat_a_1)
  2           0 SETUP_LOOP          26 (to 29)
              3 LOAD_GLOBAL          0 (abc)
              6 GET_ITER
        >>    7 FOR_ITER            18 (to 28)
             10 STORE_FAST           0 (letter)

  3          13 LOAD_GLOBAL          0 (abc)
             16 LOAD_CONST           1 (0)
             19 BINARY_SUBSCR
             20 LOAD_FAST            0 (letter)
             23 BINARY_ADD
             24 POP_TOP
             25 JUMP_ABSOLUTE        7
        >>   28 POP_BLOCK
        >>   29 LOAD_CONST           0 (None)
             32 RETURN_VALUE
>>> dis.dis(concat_a_2)
  2           0 LOAD_GLOBAL          0 (abc)
              3 LOAD_CONST           1 (0)
              6 BINARY_SUBSCR
              7 STORE_FAST           0 (a)

  3          10 SETUP_LOOP          22 (to 35)
             13 LOAD_GLOBAL          0 (abc)
             16 GET_ITER
        >>   17 FOR_ITER            14 (to 34)
             20 STORE_FAST           1 (letter)

  4          23 LOAD_FAST            0 (a)
             26 LOAD_FAST            1 (letter)
             29 BINARY_ADD
             30 POP_TOP
             31 JUMP_ABSOLUTE       17
        >>   34 POP_BLOCK
        >>   35 LOAD_CONST           0 (None)
             38 RETURN_VALUE
```

Вторая функция в листинге 10.5 хранит `abc[0]` в виде временной переменной перед запуском цикла. Это делает байт-код, выполняемый внутри цикла, короче байт-кода первой функции, так как `abc[0]` не осуществляет поиск при каждой итерации. Измерения с `timeit` показывают, что вторая версия быстрее на 10 %; она выполняется на целую микросекунду быстрее! Естественно, заниматься оптимизацией ради такой выгоды не имеет смысла, если, конечно, операция

не повторяется миллиарды раз, — для определения этого и нужно заглянуть внутрь процесса с помощью модуля `dis`.

Не имеет значения, храните ли вы переменные вне цикла, от которого она зависит, — эта работа по оптимизации должна выполняться компилятором. Но с другой стороны, компилятору сложно определить, отразится ли это положительно на работе программы из-за динамической природы Python. В листинге 10.5 использование `abc[0]` вызовет `abc._getitem_`, что может привести к нежелательным последствиям, если оно было переопределено наследованием. В зависимости от версии применяемой функции метод `abc._getitem_` будет вызван единожды или несколько раз, что может привести к значительной разнице. Поэтому следует осторожно подходить к вопросам оптимизации кода!

Эффективное объявление функций

Одна из распространенных ошибок, которая мне неоднократно встречалась в чужом коде, — объявление функции внутри функции. Это неэффективно, потому что приводит к бесполезному многократному объявлению одной и той же функции внутри других функций. В листинге 10.6 показано, как функция `y()` была объявлена несколько раз.

Листинг 10.6. Повторное объявление функции

```
>>> import dis
>>> def x():
...     return 42
...
>>> dis.dis(x)
  2           0 LOAD_CONST               1 (42)
              3 RETURN_VALUE
>>> def x():
...     def y():
...         return 42
...     return y()
...
>>> dis.dis(x)
  2           0 LOAD_CONST               1 (<code object y at
x100ce7e30, file "<stdin>", line 2>)
              3 MAKE_FUNCTION            0
              6 STORE_FAST               0 (y)

  4           9 LOAD_FAST                0 (y)
             12 CALL_FUNCTION            0
             15 RETURN_VALUE
```

В этом примере можно увидеть вызовы MAKE_FUNCTION, STORE_FAST, LOAD_FAST и CALL_FUNCTION, которые требуют гораздо больше кодов операций, чем в примере для возврата числа 42 из листинга 10.4.

Единственный случай, когда надо будет объявить функцию внутри функции, — это создание функции замыкания, и это явно определенный для Python способ применения кодов операций, так как для него есть код LOAD_CLOSURE (листинг 10.7).

Листинг 10.7. Объявление замыкания

```
>>> def x():
...     a = 42
...     def y():
...             return a
...     return y()
...
>>> dis.dis(x)
  2           0 LOAD_CONST               1 (42)
              3 STORE_DEREF              0 (a)

  3           6 LOAD_CLOSURE             0 (a)
              9 BUILD_TUPLE              1
             12 LOAD_CONST               2 (<code object y at
x100d139b0, file "<stdin>", line 3>)
             15 MAKE_CLOSURE             0
             18 STORE_FAST               0 (y)

  5          21 LOAD_FAST                0 (y)
             24 CALL_FUNCTION            0
             27 RETURN_VALUE
```

Использование дизассемблинга — не ежедневная практика, но полезно знать о применении этого инструмента, если возникнет необходимость заглянуть внутрь процессов.

Упорядоченные списки и bisect

Теперь посмотрим, как оптимизировать списки. Если список не отсортирован, то наихудший сценарий для поиска элемента описывается сложностью $O(n)$. То есть необходимый элемент вы, скорее всего, найдете после обхода всего списка.

Распространенное решение для оптимизации этой проблемы — использование отсортированного списка. Он применяет алгоритм двоичного поиска, что по-

зволяет достичь времени выполнения *O(log n)*. Идея в том, чтобы разделить список пополам и посмотреть, в какую часть мог попасть нужный элемент, а затем искать уже в ней.

Для этого в Python есть модуль `bisect`, содержащий алгоритм двоичного поиска, пример которого представлен в листинге 10.8.

Листинг 10.8. Использование bisect для поиска иголки в стоге сена[1]

```
>>> farm = sorted(['haystack', 'needle', 'cow', 'pig'])
>>> bisect.bisect(farm, 'needle')
3
>>> bisect.bisect_left(farm, 'needle')
2
>>> bisect.bisect(farm, 'chicken')
0
>>> bisect.bisect_left(farm, 'chicken')
0
>>> bisect.bisect(farm, 'eggs')
1
>>> bisect.bisect_left(farm, 'eggs')
1
```

В этом примере функция `bisect.bisect()` возвращает позицию, в которой должен находиться искомый элемент для обеспечения отсортированного состояния списка. Очевидно, что это работает, если список с самого начала правильно отсортирован. Осуществив первоначальную выборку, можно получить теоретический индекс искомого элемента: `dissect()` возвращает не информацию о наличии элемента в списке, а только информацию о том, где он должен быть. Возврат элемента по этому индексу ответит на вопрос, находится ли он в списке.

Если требуется вставить элемент сразу в правильную отсортированную позицию, модуль `bisect` поможет в этом функциями `insort_left()` и `insort_right` (листинг 10.9):

Листинг 10.9. Вставка элемента в отсортированный список

```
>>> farm
['cow', 'haystack', 'needle', 'pig']
>>> bisect.insort(farm, 'eggs')
>>> farm
['cow', 'eggs', 'haystack', 'needle', 'pig']
```

[1] В листинге 10.8 список `farm` состоит из слов «Сено, иголка, корова, свинья».

```
>>> bisect.insort(farm, 'turkey')
>>> farm
['cow', 'eggs', 'haystack', 'needle', 'pig', 'turkey']
```

Использование модуля `bisect` позволит также создать особый класс `SortedList`, наследованный из `list` для создания списка, который всегда будет отсортирован как в листинге 10.10:

Листинг 10.10. Реализация объекта SortedList

```
import bisect
import unittest

class SortedList(list):
    def __init__(self, iterable):
        super(SortedList, self).__init__(sorted(iterable))

    def insort(self, item):
        bisect.insort(self, item)

    def extend(self, other):
        for item in other:
            self.insort(item)

    @staticmethod
    def append(o):
        raise RuntimeError("Cannot append to a sorted list")

     def index(self, value, start=None, stop=None):
         place = bisect.bisect_left(self[start:stop], value)
         if start:
             place += start
         end = stop or len(self)
         if place < end and self[place] == value:
             return place
         raise ValueError("%s is not in list" % value)

class TestSortedList(unittest.TestCase):
    def setUp(self):
        self.mylist = SortedList(
            ['a', 'c', 'd', 'x', 'f', 'g', 'w']
        )

    def test_sorted_init(self):
        self.assertEqual(sorted(['a', 'c', 'd', 'x', 'f', 'g', 'w']),
                         self.mylist)

    def test_sorted_insort(self):
```

```
        self.mylist.insort('z')
        self.assertEqual(['a', 'c', 'd', 'f', 'g', 'w', 'x', 'z'],
                         self.mylist)
        self.mylist.insort('b')
        self.assertEqual(['a', 'b', 'c', 'd', 'f', 'g', 'w', 'x', 'z'],
                         self.mylist)

    def test_index(self):
        self.assertEqual(0, self.mylist.index('a'))
        self.assertEqual(1, self.mylist.index('c'))
        self.assertEqual(5, self.mylist.index('w'))
        self.assertEqual(0, self.mylist.index('a', stop=0))
        self.assertEqual(0, self.mylist.index('a', stop=2))
        self.assertEqual(0, self.mylist.index('a', stop=20))
        self.assertRaises(ValueError, self.mylist.index, 'w', stop=3)
        self.assertRaises(ValueError, self.mylist.index, 'a', start=3)
        self.assertRaises(ValueError, self.mylist.index, 'a', start=333)

    def test_extend(self):
        self.mylist.extend(['b', 'h', 'j', 'c'])
        self.assertEqual(
            ['a', 'b', 'c', 'c', 'd', 'f', 'g', 'h', 'j', 'w', 'x']
            self.mylist)
```

Класс `list`, конечно, более медленный для добавления элемента в список, так как программа должна найти правильное место для его вставки. Однако этот класс быстрее использует метод `index()`, чем его родитель. Очевидно, что нельзя применять метод `list.append()` к этому классу: невозможно добавить элемент в конец списка, потому что это приведет к неотсортированному состоянию.

Многие библиотеки Python реализуют разные версии кода из листинга 10.10 для разных структур данных, таких как бинарные или красно-черные деревья. Пакеты `blist` и `bintree` содержат для этих целей код и их применение удобнее, чем реализация и отладка своих аналогов.

В следующем разделе мы рассмотрим, как можно ускорить работу кортежей и, как следствие, выполнение всей программы.

Именованные кортежи и Slots

Часто в программировании необходимо создать простые объекты, обладающие всего парой фиксированных атрибутов. Простейшая реализация может выглядеть примерно так:

```
class Point(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

Это работоспособный вариант. Но при таком подходе есть один минус — создается класс, который наследуется из классового объекта, и с помощью класса `Point` инстанцируются все объекты, что приводит к высоким затратам памяти.

В Python обычные объекты содержат все атрибуты внутри словаря, который хранится в атрибуте `_dict_`, как показано в листинге 10.11.

Листинг 10.11. Хранение атрибутов внутри объекта

```
>>> p = Point(1, 2)
>>> p.__dict__
{'y': 2, 'x': 1}
>>> p.z = 42
>>> p.z
42
>>> p.__dict__
{'y': 2, 'x': 1, 'z': 42}
```

Преимущество использования `dict` состоит в том, что он позволяет добавлять в объект сколько угодно атрибутов. Недостатком являются большие затраты памяти — надо хранить объект, ключи, значения и многое другое. Поддержка этого атрибута становится сложной, медленной и затратной по количеству используемой памяти.

Взгляните на пример неоправданного использования памяти:

```
class Foobar(object):
    def __init__(self, x):
        self.x = x
```

Создается простой объект `Point` с единственным атрибутом `x`. Проверим количество памяти, используемой этим классом, с помощью пакета `memory_profiler`, который позволяет посмотреть строка за строкой затраты памяти. Для этого применим скрипт, создающий 100 000 объектов, как показано в листинге 10.12.

Листинг 10.12. Использование memory_profiler на полученных объектах

```
$ python -m memory_profiler object.py
Filename: object.py
```