

Optimizamos scripts en Python

# LIGERO DE EQUIPAJE

Yong Hiam Lim, Fotolia

El secreto de la optimización es ahorrar tiempo en el lugar adecuado.

POR STEFAN SCHWARZER

La optimización ahorra tiempo de ejecución, aunque desafortunadamente también incrementa el ciclo de desarrollo. El código fuente optimizado generalmente es más complejo que el código original, lo que incrementa el tiempo de pruebas y depuración. Al aumentar la complejidad, el código también se hace más difícil de mantener. Por tanto, debido a que el proceso de optimización lleva tiempo y aumenta la complejidad, es mejor no optimizar código al escribirlo. Antes de iniciar el proceso deberíamos tener un programa estable. Una vez que nuestro programa sea estable y esté completo, podemos buscar las maneras de mejorar su rendimiento. En este artículo describimos algunas estrategias para optimizar programas escritos en Python.

## Dónde Optimizar

Desde la perspectiva del desarrollador, un programa no es simplemente lento o rápido. Antes de comenzar a acelerar secciones de código es importante descubrir dónde están los cuellos de botella. El primer paso es encontrar si la CPU o el sistema I/O está ralentizando nuestro software cuando se ejecuta una función concreta. No tiene sentido optimizar la ejecución del algoritmo en un factor de 100 sólo para descubrir que el problema está en el disco duro o en la red.

Para averiguar si una CPU lenta, un disco duro lento u otro componente hardware es la causa del problema, podemos hacer uso de una herramienta basada en interfaz gráfica como Xosview [1] o Gkrellm [2]. También, herramientas como Dstat [3] proporcionan estadísticas de las transferencias de formación desde y hacia particiones específicas. Para obtener resultados veraces debemos asegurarnos de que el ordenador no está ejecutando ningún otro proceso que pudiera causar carga adicional. Como alternativa, *top* y *ps* ofrecen información del rendimiento concreto de cada proceso.

El problema de la transferencia de datos es más difícil de identificar que el de un cuello de botella CPU debido a que no hay umbral superior independiente del hardware.

La mejor manera de determinar los valores umbrales es referirse a especificaciones hardware o usar benchmarks. Nótese que la transferencia de información puede referirse a una unidad de CD-ROM, como nuestro ejemplo, pero podría referirse fácilmente a una interfaz de red, a una unidad de cinta para copias de seguridad, etc.

Si una carga del 100% de la CPU está ralentizando nuestro programa, necesitamos identificar las secciones de código que están causando el problema. El módulo *cProfile* de Python puede ayudarnos a eva-

luar los resultados que genera el módulo *Pstats*.

Como ejemplo práctico de un analizador Python podemos considerar el paquete de la herramienta de mantenimiento *Emerge* de Gentoo Linux, que está escrita en Python. Una búsqueda con *emerge --search python* tarda unos 10 segundos en ejecutarse en mi ordenador. No tiene sentido plantearse optimizarlo, a menos que tengamos un ordenador muy lento, pero es un ejemplo de cómo proceder en una fase de análisis.

Generar estadísticas en tiempo de ejecución con el módulo *cProfile* de Python es un poco complicado, debido a que la línea de

## Listado 1: cProfile

```
01 >>> import cProfile
02 >>> import sys
03 >>>
04 >>> sys.argv.append("--search")
05 >>> f =
06 >>> open("/usr/bin/emerge")
07 >>> ef = f.read()
08 >>> f.close()
09 >>> cProfile.run(ef,
10 >>> "emerge.stats")
11 Searching...
12 [ Results for search key :
13 python]
14 [ Applications found : 48 ]
15 ...
```

comandos del analizador no ve la necesidad de pasar parámetros al programa al que estamos llamando, motivo por el cual hemos usado un intérprete interactivo (Listado 1). Tras importar los módulos Python necesarios y preparar los parámetros, tecleamos `cProfile.run()` para comenzar el test de ejecución. El módulo `Pstats` genera una tabla con las estadísticas de tiempo de ejecución (véase el Listado 2).

Aunque aquí no ocurre un auténtico cuello de botella, podríamos optimizar un par de puntos. Por ejemplo, emerge tarda 1,2 segundos de 9 en actualizar el indicador de progreso (método `update_twirl`, podemos ver esto en la columna `cumtime`, o tiempo acumulado, del Listado 2). Sin embargo, una opción de emerge puede deshabilitar esta visualización. Se usan unos 1,2 segundos para copia profunda intensiva. Si las copias profundas no son necesarias, hay margen para algún ahorro.

Para acelerar el código con un cuello de botella de CPU, podemos hacer las cosas más rápido o bien hacerlas con menor frecuencia. Si reemplazamos nuestro sencillo sistema de administración de datos con una base de datos como SQLite [4], a menudo alcanzaremos ambos objetivos.

Por supuesto, el objetivo es alcanzar el máximo, o al menos suficientes mejoras en la velocidad con el mínimo esfuerzo en desarrollo. Y tenemos que tener en cuenta también la mantenibilidad del código.

Antes de lanzarnos a medir la velocidad de nuestro código tenemos que asegurarnos de que esté libre de errores, tanto como humanamente sea posible. Si nuestro código tiene errores, el peligro es que podríamos estar “optimizando” código que se ejecuta de manera lenta simplemente porque tiene fallos. Los tests automatizados, escritos con los módulos `doctest` y `unittest` de Python pueden ayudarnos a reducir los errores al modificar el código.

El siguiente paso es comenzar a perfilar cuáles son las secciones de código más importantes de cara a la optimización. Los mejores candidatos para la optimización son generalmente las secciones con el mayor tiempo de ejecución total, es decir, secciones en las que el producto del tiempo de ejecución y la frecuencia de ejecución es particularmente alto. Normalmente tiene más sentido optimizar una función que se ejecuta 10.000 veces y tarda un segundo en ejecutarse, que optimizar

una función que sólo se ejecuta cinco veces y tarda 10 segundos. Pero también podríamos considerar hasta cierto punto las zonas en las que el tiempo de ejecución afecta a la experiencia del usuario. Podríamos descubrir que el programa simplemente sufre un pequeño retraso en el primer caso, mientras que el segundo caso suponga una espera de 10 segundos para el usuario.

## Técnicas de Optimización

Una manera de acelerar un programa es reemplazando un algoritmo por otro algoritmo más eficiente. Mientras que la mayoría de las técnicas de optimización acelerarán el código en un factor de un 10% como mucho, reemplazar un algoritmo puede suponer mejoras en la velocidad de varios cientos por cien.

Se usa la notación Gran-O (Big-O) para describir la complejidad de un algoritmo. La “O” significa el “orden” del algoritmo. La expresión entre paréntesis describe los cambios del rendimiento en función del cambio en la información de entrada (el número de valores en una lista, o la longitud de una cadena, por ejemplo). Un algoritmo  $O(n)$  tardará en ejecutarse el doble si

### Listado 2: Pstats

```
01 >>> import pstats
02 >>> s = pstats.Stats("emerge.stats")
03 >>> s.sort_stats('time')
04 <pstats.Stats instance at 0xb7d80eac>
05 >>> s.print_stats(10)
06 Sun Oct 1 23:12:36 2006 emerge.stats
07
08      602508 function calls (586701 primitive calls) in 9.052 CPU seconds
09
10 Ordered by: internal time
11 List reduced from 609 to 10 due to restriction <10>
12
13 ncalls tottime percall cumtime percall filename:lineno(function)
14 1240 1.022 0.001 1.022 0.001 {method 'readlines' of 'file' objects}
15 11387 0.849 0.000 0.849 0.000 {method 'flush' of 'file' objects}
16 1096 0.579 0.001 1.513 0.001 /usr/lib/portage/pym/portage.py:200(cacheddir)
17 14550/160 0.421 0.000 1.173 0.007 /home/schwa/python2.5/lib/python2.5/copy.py:144(deepcopy)
18 76352 0.359 0.000 0.359 0.000 {method 'append' of 'list' objects}
19 1 0.335 0.335 2.513 2.513 <string>:468(output)
20 66288/66173 0.316 0.000 0.317 0.000 {len}
21 11383 0.256 0.000 1.225 0.000 <string>:94(update_twirl)
22 36953 0.224 0.000 0.224 0.000 {method 'split' of 'str' objects}
```

se dobla el volumen de los datos, mientras que un algoritmo  $O(n^2)$  tardará cuatro veces más cuando se duplica el volumen de la información de entrada.

Obviamente, sería deseable que la expresión entre paréntesis creciera muy poco, a pesar de un incremento del valor de  $n$ . La tabla 1 muestra el rendimiento de varios algoritmos en Python. El rendimiento decrece desde la parte superior a la inferior.

Para bucles múltiplemente anidados, o en algunos problemas combinatoriales, el ratio de rendimiento puede ser superior al cuadrático. En este caso, incluso pequeños valores de  $n$  provocarán un pobre rendimiento. Si el código causa un cuello de botella, podríamos tratar de evitar algoritmos de orden superior al cuadrático para valores grandes de  $n$ .

Si estamos ejecutando un algoritmo con un volumen de información menor, un algoritmo más complejo puede ser más rápido. Con algunos algoritmos, el tiempo de ejecución se incrementará lentamente al crecer  $n$ . Sin embargo, podría ser necesario un paso preparatorio mayor. En este caso, un algoritmo “más lento” que evite la necesidad del paso preparatorio podría ser en última instancia más rápido.

La comparación de algoritmos en base a su orden es útil en principio, pero podría no ser aplicable en la práctica, o al menos no para cualquier volumen de información. Por ejemplo, sin importar la eficiencia teórica de un algoritmo, el rendimiento podría reducirse si la lista que necesitamos ordenar no cabe en memoria y el sistema operativo tiene que comenzar a hacer uso de la memoria de swap en disco. Efectos como éste deben tenerse en cuenta si la administración de memoria de la librería C estándar subyacente tiene gran influencia en el tiempo de ejecución.

## Conjunto Intersección Optimizado

Veamos otro ejemplo. Cada una de las funciones Python que vamos a investigar

encuentra el conjunto intersección de dos listas (elementos que están en ambas) y devuelve una nueva lista con los resultados.

El primer algoritmo que usamos tiene complejidad cuadrática dadas dos listas con  $n$  elementos (véase el Listado 3). El bucle exterior itera sobre todos los elementos de la primera lista, con complejidad lineal. Este bucle contiene un segundo bucle implícito, que está escondido en el test de condición *value in list2*.

La búsqueda de *list2* es lineal, por lo que tenemos que considerar dos sucesos de complejidad lineal. La anidación del bucle explícito externo y el bucle implícito interno hace que el algoritmo sea de tipo  $O(n^2)$ . A pesar de que la determinación de las claves de la instrucción *return* es lineal, esto es insignificante en comparación con la complejidad cuadrática del algoritmo previo con respecto a  $n$ . En general, es preferible evitar bucles anidados. En el mejor de los casos, esto nos supondrá tener un rendimiento cuadrático. Dicho esto, la optimización probablemente no merezca la pena para volúmenes pequeños de entrada.

El algoritmo del Listado 4 es una versión modificada del algoritmo anterior, pero con complejidad lineal. El código se parece al del listado anterior, pero crea un diccionario a partir de la segunda lista antes de entrar en el bucle externo y usa entonces el diccionario dentro del bucle.

*value in dict2* muestra un rendimiento constante, por lo que el balance final da como resultado un bucle externo lineal. El último algoritmo también tiene rendimiento lineal (véase el Listado 5). La operación que convierte la primera lista en un conjunto es lineal, al igual que la generación del conjunto resultante gracias, al método *intersection* y la conversión del conjunto resultante a una lista. Aunque la sintaxis de estas operaciones está anidada, en realidad se ejecutan de manera secuencial. Tres pasos lineales secuenciales dan como resultado un algoritmo  $O(n)$ .

## Algoritmos Mejores

Los ejemplos anteriores sugieren unas cuantas reglas de optimización que puede que conozca. Por ejemplo, las operaciones cuyos resultados no cambian tras múltiples iteraciones de un bucle podrían desplazarse al principio del bucle, evitando de esta manera la necesidad de ejecutarlas en cada iteración.

El principio de *divide y vencerás* podría funcionar bien con los datos. Un conocido ejemplo de esto es una búsqueda binaria que requiera ordenación previa de información pero que devuelve los resultados con una complejidad de  $O(\ln n)$  en lugar de  $O(n)$ . Sin embargo, si manejamos pequeñas cantidades de información de entrada, puede bastar con una búsqueda lineal trivial.

En lugar de recargar constantemente o recalcular, podríamos almacenar valores en caché. Aunque debemos considerar las consecuencias y la posible inconsistencia de los datos, especialmente en sistemas que usan múltiples hilos o transacciones. Asimismo, debemos tener en consideración la restricción del tamaño de la caché para mantener el sistema libre del swapping a disco y echar por tierra cualquier incremento de velocidad. En escenarios en los cuales tiene sentido la memoria caché, el uso de un servidor de base de datos a menudo nos proporciona una importante mejora del rendimiento.

Si guardamos un objeto en disco o transferimos un objeto por la red, podemos acelerar la operación simplemente guardando los cambios en lugar de todo el objeto. Como desventaja, este tipo de optimización pueda afectar a las abstracciones de clase o a otro código. Intente mantener la abstracción de la interfaz incluso si estamos optimizando internamente.

**Tabla 1: Rendimiento de Algoritmos Python**

Orden	Descripción	Ejemplos
$O(1)$	Tiempo constante	<i>key in dict, dict[key] = value, list.append(value)</i>
$O(\ln n)$	Tiempo logarítmico	Búsqueda binaria
$O(n)$	Tiempo lineal	<i>value in list, str.join(list)</i>
$O(n \ln n)$		<i>list.sort()</i>
$O(n^2)$	Tiempo cuadrático	Bucles anidados [para bucles $O(1)$ ]

### Listado 3: intersection1

```
01 def intersection1(list1,
02     list2):
03     """Determine resulting set
04     with O(n^2) algorithm."""
05     result = {}
06     for value in list1:
07         if value in
08             list2:
09             result[value] = True
10     return result.keys()
```

Las siguientes reglas se aplican a la manipulación de texto línea a línea: si los archivos son pequeños, generalmente es más sencillo y rápido leer el texto completamente antes de procesar la información.

Para grandes archivos (los archivos de log son el ejemplo típico), es mejor leer y procesar cada línea independientemente. Si no lo hacemos así corremos el riesgo de quedarnos sin memoria, y caer en un swapping continuo que podría colgar nuestro equipo.

La elección de las estructuras de datos correcta está íntimamente relacionada con la elección del algoritmo, y de hecho, la elección de la estructura de datos tendrá influencia implícitamente en la elección de los algoritmos de acceso a la información. Como se demostró anteriormente, la búsqueda de una clave en un diccionario es mucho más rápida que una búsqueda lineal del mismo valor en una lista.

La arquitectura del software también influye en los rendimientos. Podemos considerar la arquitectura como el algoritmo que sigue el sistema completo, y considerarlo antes de comenzar el desarrollo.

## Trucos Python

Las optimizaciones específicas de Python tienen distintos efectos en función de la versión de Python que se esté utilizando. Una versión reciente puede incluso hacer más lento un fragmento de código, aunque esto suele ser la excepción. La manera más fácil de optimizar un script Python es usar la opción -O del intérprete para optimizar automáticamente el bytecode Python generado por el intérprete. No debemos usar `from module import *` en nuestros scripts, pues esto impide que el intérprete de Python lleve a cabo algunas optimizaciones internas, y además hace que el manteni-

miento sea más complicado. Debemos evitar las operaciones de búsqueda a través de múltiples espacios de nombres, uniendo un objeto directamente al espacio de nombres local. Por ejemplo, tras una línea `opj = os.path.join` podemos acceder a la función `join` más rápidamente que a `opj`. Este tipo de optimización afecta a la legibilidad de nuestro código.

Debemos evitar también `exec` y `eval`. Python es flexible, por lo que deberíamos encontrar una variante que no necesite estas funciones. En muchos casos, esta práctica incluso mejora la legibilidad. Aplicar técnicas de código en línea (“In-lining”) al cuerpo de la función puede ayudar a acelerar código que ejecuta funciones con un corto tiempo de ejecución dentro de bucles, pero éstos frecuentemente generan más redundancia y hace que el software sea más difícil de mantener.

Si tenemos que concatenar múltiples cadenas, podemos agruparlas en una lista y unir los elementos de la lista con `"".join(list)`. Este método es más rápido que usar el operador `+`. El argumento `key` de `list.sort` genera un código más rápido que el argumento `cmp`.

## C al Rescate

Para algunas operaciones puede ser mejor usar código C altamente optimizado, pero sin sacrificar ventajas de Python. Para ello podemos reescribir nuestro código o partes de él, usar las funciones internas de Python (por ejemplo, `range` en lugar de un bucle) o tipos de datos (listas, tuplas, diccionarios, conjuntos...). Usamos librerías C para código crítico respecto del tiempo de ejecución. Podemos usar la librería `libxml2` [5] para analizar XML, y `SWIG` [6] y `Ctypes` [7] son útiles para encapsular las librerías C existentes. Este último es parte de la distribución estándar desde Python 2.5.

`PyInline` [8] y `Weave` [9] ofrecen al desarrollador la capacidad de integrar fragmentos de C dentro de código Python. `Pyrex`

[10], un lenguaje muy similar a Python, permite al desarrollador encapsular librerías C existentes y definir sus propias extensiones (que se convierten a C). El método más flexible, pero al mismo tiempo el más complicado, es Python/ C-API. Como alternativa a programar simplemente en C, podríamos robar `Psyco` [11], un compilador just-in-time para Python que, desafortunadamente, sólo está disponible para sistemas x86 de 32 bits.

## Conclusiones

Los lenguajes interpretados como Python [12] no son impedimento para el desarrollo rápido de programas, pero debemos acordarnos de probar la velocidad del programa y averiguar si éste es suficientemente rápido para la aplicación requerida sin necesidad de optimización. En caso de que no sea así, deberíamos encontrar los cuellos de botella con las herramientas adecuadas, y poder concentrar nuestros esfuerzos de optimización. La modificación de los algoritmos y de las estructuras de datos, o sencillamente reemplazar el hardware, son nuestras mejores apuestas para la mejora en los tiempos de ejecución.

Las optimizaciones específicas de Python también pueden ayudar. Si fuese posible, podríamos usar código C implícito en forma de código Python en el caso de, por ejemplo, estructuras de datos Python o librerías C externas. Y siempre debemos tener presente la mantenibilidad del código cada vez que lo optimicemos. ■

### Listado 4: intersection2

```
01 def
   intersection2(list1,list2):
02     """Determine
   resulting set with O(n)
   algorithm."""
03     result = {}
04     dict2 = dict((value,
   True) for value in list2)
05     for value in list1:
06         if value in
   dict2:
07             result[value] = True
08     return result.keys()
```

### Listado 5: intersection3

```
01 def intersection3(list1,
   list2):
02     """Determine
   resulting set with O(n)
   algorithm."""
03     return
   list(set(list1).intersection(
   list2))
04 @KE
```

## RECURSOS

- [1] Xosview: <http://sourceforge.net/projects/xosview/>
- [2] Gkrellm: <http://www.gkrellm.net/>
- [3] Dstat: <http://dag.wieers.com/home-made/dstat/>
- [4] SQLite: <http://www.sqlite.org/>
- [5] Libxml2: <http://xmlsoft.org/>
- [6] SWIG: <http://www.swig.org/>
- [7] Ctypes: <http://starship.python.net/crew/theller/ctypes/>
- [8] PyInline: <http://pyinline.sourceforge.net/>
- [9] Weave: <http://www.scipy.org/Weave>
- [10] Pyrex: <http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/>
- [11] Psyco: <http://psyco.sourceforge.net/>
- [12] Python: <http://www.python.org/>