Artículo
Daniel Aguilar · 24 feb, 2020 Lectura de 6 min

Escribir bucles más eficientes en Caché ObjectScript

El tema del rendimiento de los bucles for/while en Caché ObjectScript surgió recientemente en una discusión, y me gustaría compartir algunas ideas/prácticas recomendadas con el resto de la comunidad. Aunque este es un tema básico por sí mismo, es útil conocer con cuales se obtiene un mayor rendimiento.

En resumen, las opciones más rápidas son los bucles que se iteran por $ListBuild, las listas que se forman con $ListNext o sobre un conjunto local mediante $Order.

Por ejemplo, consideraremos iterar un bucle por cada elemento de una cadena delimitada por comas.

La manera más sencilla para escribir un bucle de este tipo, con la mínima cantidad de código, es:

```
For i=1:1:$Length(string,",") {
 Set piece = $Piece(string,",",i)
 //Hacer algo con el piece...
}
```

Resulta bastante sencillo, aunque podrían sugerirse muchas guías de estilo para el código:

```
Set n = $Length(string,",")
For i=1:1:n {
 Set piece = $Piece(string,",",i)
 //Hacer algo con el piece...
}
```

En realidad, no existe una diferencia de rendimiento entre los dos, ya que la última condicional no se evalúa en cada iteración. (En un principio me había equivocado, gracias a Mark por señalar que solo es una cuestión de estilo y no de rendimiento).

En el caso de una cadena definida, podemos lograr un mejor rendimiento. Conforme la cadena se hace más larga, el comando $Piece(cadena,",",i) se vuelve más costoso: debe recorrer la cadena hasta el final del i-ésimo elemento. Esto puede mejorarse mediante el uso de listas con $ListBuild. Por ejemplo, utilizando $ListFromString, $ListLength, y $List:

```
Set list = $ListFromString(string,",")
Set n = $ListLength(list)
For i=1:1:n {
 Set piece = $List(list,i)
 //Hacer algo con el piece...
}
```

Normalmente esto funcionará mejor que $Length/$Piece, en especial, cuando las variables piece aumentan su extensión. En el método $Length/$Piece, cada una de las n iteraciones recorre los caracteres de los primeros i elementos. En el método $ListFromString/$ListLength/$List, los punteros i se encuentran en la estructura

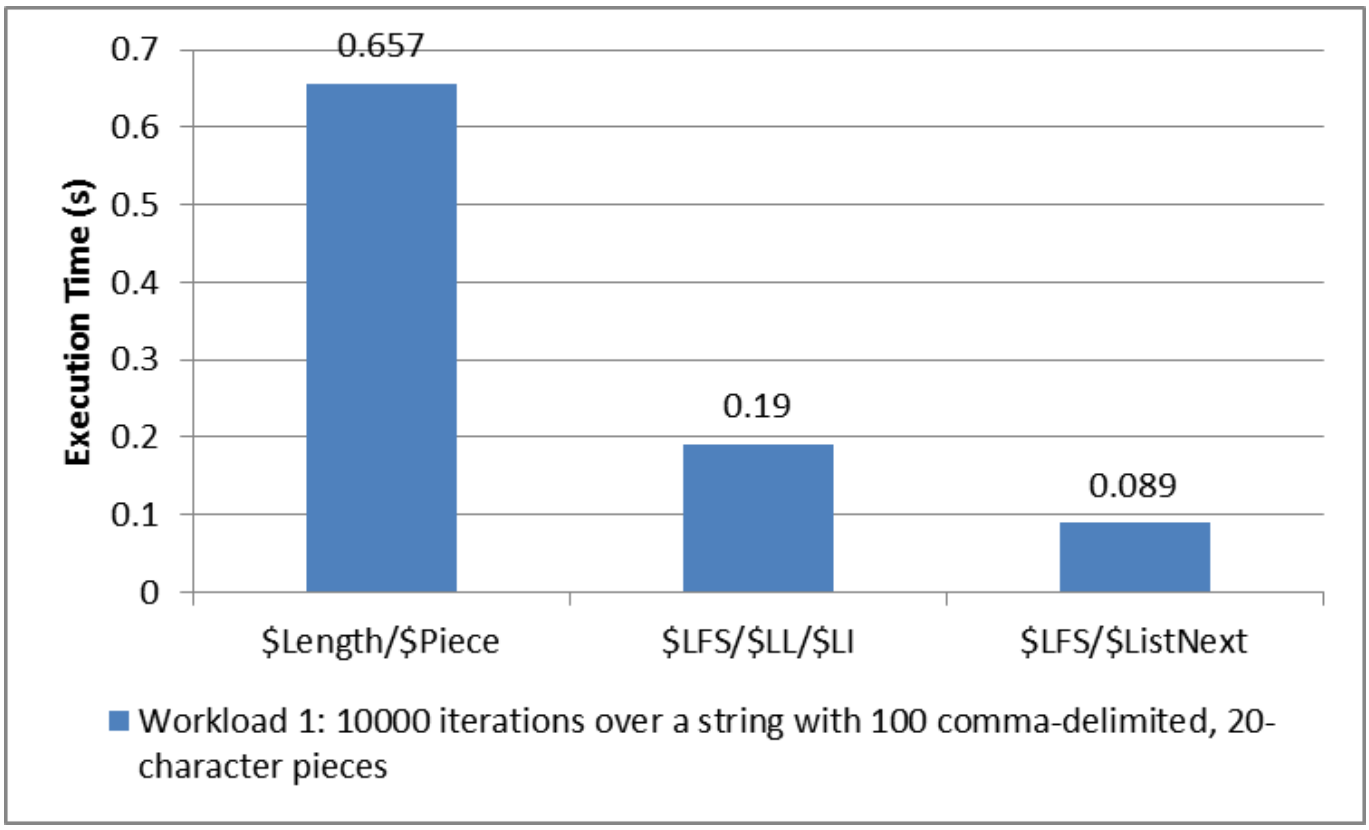
$ListBuild para cada una de las n iteraciones. Esperaríamos que tuviera mejor rendimiento, y lo tiene, pero el tiempo de ejecución sigue siendo O(n2). Si suponemos que el bucle no modificará la lista, podemos mejorar el O(n), con $ListNext:

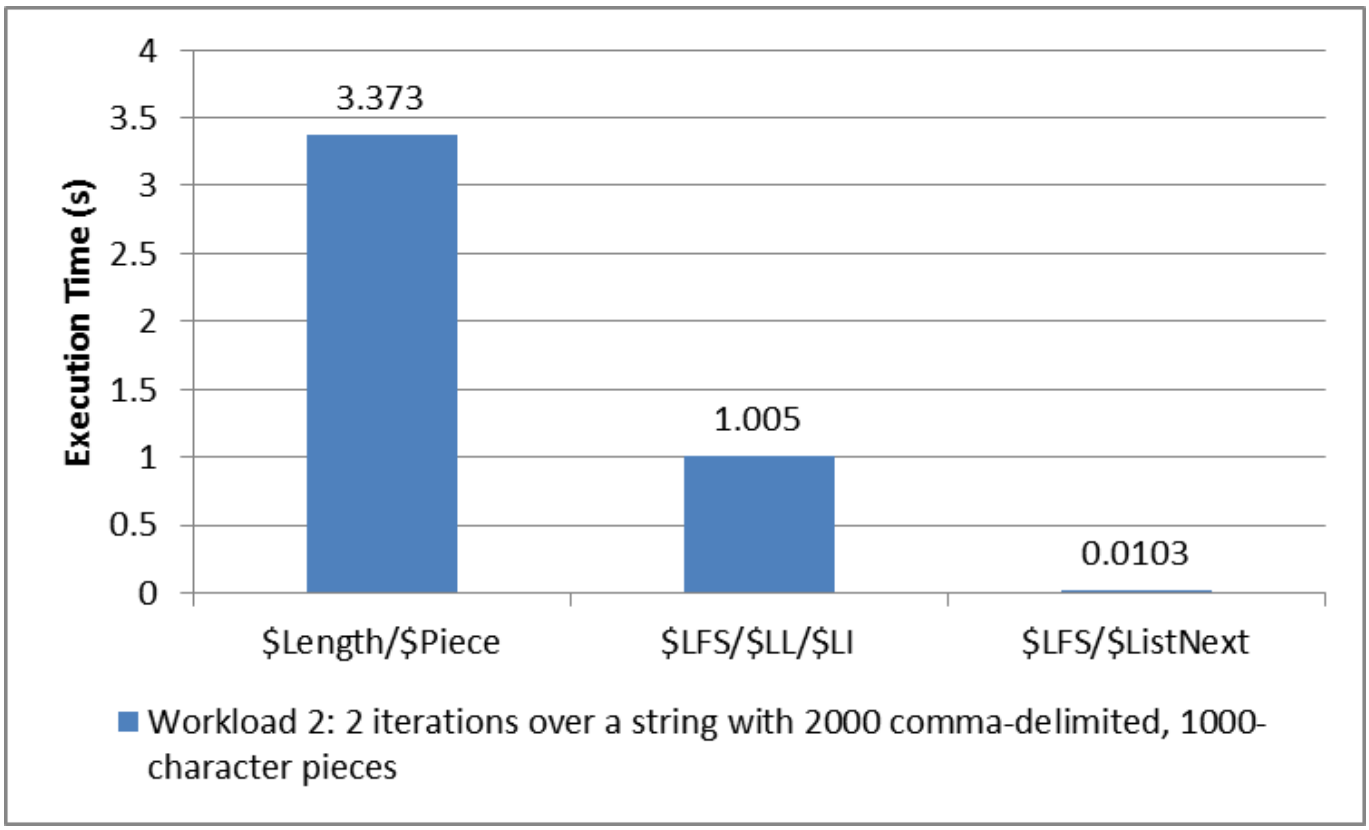
```
Set list = $ListFromString(string,",")
Set pointer = 0
While $ListNext(list,pointer,piece) {
 //Do something with piece...
}
```

En vez de seguir a los punteros a través de la lista desde el principio hasta el i-ésimo elemento, como lo hace $List siempre que se utiliza, la variable " pointer " mantiene un registro de la posición actual en la lista. Por lo tanto, en vez de utilizar n(n+1)/2 como total de operaciones para " seguir al puntero " (i en cada una de las n iteraciones para $List), ahora solo hay n de ellas (una en cada iteración para $ListNext).

Por último, puede ser una buena opción convertir los elementos de las cadenas en un conjunto con subíndices enteros, en general, iterar en un conjunto local con $Order puede ser desde un poco hasta significativamente más rápido que hacerlo con $ListNext (dependiendo de la longitud de elementos en la lista). Por supuesto, si comienza con una cadena definida, será un poco difícil convertirla en un conjunto. Si itera varias veces, necesitará modificar las partes de la lista o iterar al revés, probablemente valdría la pena realizar esta cantidad de trabajo adicional.

A continuación, se muestran algunos ejemplos de los tiempos de ejecución (incluidas todas las conversiones necesarias) para algunos tamaños de entrada diferentes:





Estos números provienen de:

```
USER>d ##class(DC.LoopPerformance).Run(10000,20,100)
Iterating 10000 times over all the pieces of a string with 100 ,-delimited pieces of
length 20:
Using $Length/$Piece (hardcoded delimiter): .657383 seconds
Using $Length/$Piece: 1.083932 seconds
Using $ListFromString/$ListLength/$List (hardcoded delimiter): .189867 seconds
Using $ListFromString/$ListLength/$List: .189938 seconds
Using $ListFromString/$ListNext (hardcoded delimiter): .089618 seconds
Using $ListFromString/$ListNext: .089242 seconds
Using $Order over an equivalent local array with integer subscripts: .072485 seconds
**************************************************
Using $ListFromString/$ListNext (not including conversion to $ListBuild list): .05832
9 seconds
Using one-argument $Order over an equivalent local array with integer subscripts: .06
0327 seconds
Using three-argument $Order over an equivalent local array with integer subscripts: .
069508 seconds
USER>d ##class(DC.LoopPerformance).Run(2,1000,2000)
Iterating 2 times over all the pieces of a string with 2000 ,-delimited pieces of len
gth 1000:
Using $Length/$Piece (hardcoded delimiter): 3.372927 seconds
Using $Length/$Piece: 11.739316 seconds
Using $ListFromString/$ListLength/$List (hardcoded delimiter): 1.004757 seconds
Using $ListFromString/$ListLength/$List: .997821 seconds
Using $ListFromString/$ListNext (hardcoded delimiter): .010489 seconds
Using $ListFromString/$ListNext: .010268 seconds
Using $Order over an equivalent local array with integer subscripts: .000839 seconds
**************************************************
Using $ListFromString/$ListNext (not including conversion to $ListBuild list): .00305
3 seconds
```

```
Using one-argument $Order over an equivalent local array with integer subscripts: .00
0938 seconds
Using three-argument $Order over an equivalent local array with integer subscripts: .
000677 seconds
```

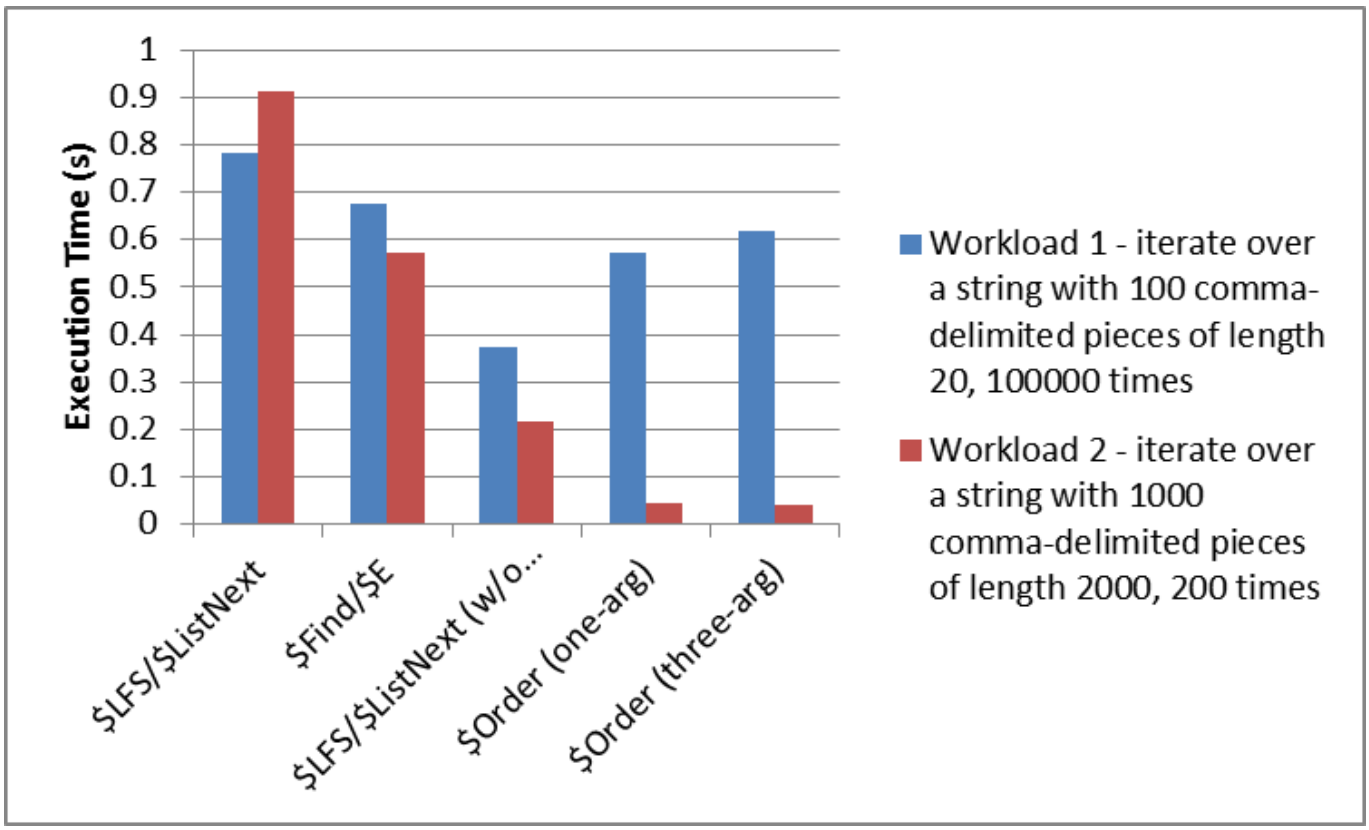
Consultar el código (DC.LoopPerformance) en Git

Actualización:

Con la discusión aparecieron otras variantes interesantes con buen rendimiento, las cuales merecen compararse entre sí. Consulta el método RunLinearOnly y las diferentes implementaciones que se probaron en las actualizaciones de Git.

```
USER>d ##class(DC.LoopPerformance).RunLinearOnly(100000,20,100)
Iterating 100000 times over all the pieces of a string with 100 ,-delimited pieces of
 length 20:
Using $ListFromString/$ListNext (While): .781055 seconds
Using $ListFromString/$ListNext (For/Quit): .8438 seconds
Using $ListFromString/$ListNext (While, not including conversion to $ListBuild list):
 .37448 seconds
Using $Find/$Extract (Do...While): .675877 seconds
Using $Find/$Extract (For/Quit): .746064 seconds
Using one-argument $Order (For): .589697 seconds
Using one-argument $Order (While): .570996 seconds
Using three-argument $Order (For): .688088 seconds
Using three-argument $Order (While): .617205 seconds
USER>d ##class(DC.LoopPerformance).RunLinearOnly(200,2000,1000)
Iterating 200 times over all the pieces of a string with 1000 ,-delimited pieces of l
ength 2000:
Using $ListFromString/$ListNext (While): .913844 seconds
Using $ListFromString/$ListNext (For/Quit): .925076 seconds
Using $ListFromString/$ListNext (While, not including conversion to $ListBuild list):
 .21842 seconds
Using $Find/$Extract (Do...While): .572115 seconds
Using $Find/$Extract (For/Quit): .610531 seconds
Using one-argument $Order (For): .044251 seconds
Using one-argument $Order (While): .04467 seconds
Using three-argument $Order (For): .043631 seconds
Using three-argument $Order (While): .042568 seconds
```

En la siguiente gráfica se comparan las versiones con los bucles While/Do...While de estos métodos. Principalmente, observa el rendimiento relativo de $ListFromString/$ListNext en comparación con el de $Extract/$Find, y el rendimiento de $ListNext (sin realizar la conversión desde la cadena en una lista con $ListBuild) en comparación con el de $Order en un conjunto local con subíndices enteros.



En resumen:

- Si operas con una cadena definida, $Find/$Extract es la opción que proporciona mayor rendimiento, aunque el código es un poco menos intuitivo que el utilizado por $ListFromString/$ListNext
- Dada una opción para la estructura de los datos, las opciones de las listas $ListBuild parece tener una ligera ventaja de rendimiento en un conjunto local equivalente para entradas pequeñas, mientras que el conjunto local ofrece un rendimiento significativamente mejor cuando se trata de entradas más grandes. No puedo explicar por qué existe una diferencia tan grande en el rendimiento. (En este sentido, valdría la pena comparar el costo de las inserciones y eliminaciones aleatorias en una lista donde se utilice $ListBuild vs. un conjunto local con subíndices enteros, intuyo que set $list sería mucho más rápido que desplazar los valores en dicho conjunto)
- El bucle While o Do...While funciona *ligeramente* mejor que un bucle equivalente For sin argumentos.

#Consejos y trucos #Code Snippet #ObjectScript #Reglas de codificación #Caché

URL de fuente:https://es.community.intersystems.com/post/escribir-bucles-m%C3%A1s-eficientes-en-cach%C3%A9-objectscript