記事
Toshihiko Minamoto · 2020年11月10日 8m read

Caché ObjectScript でパフォーマンスの良い loop を書く方法について

最近行われたディスカッションの中で、Caché ObjectScript における for/while loop のパフォーマンンスが話に出ましたので、意見やベストプラクティスをコミュニティの皆さんと共有したいと思います。 これ自体が基本的なトピックではありますが、他の点では合理的と言える方法のパフォーマンスが意味する内容を見逃してしまうことがよくあります。 つまり、$ListNext を使って$ListBuild リストをイテレートするループ、または $Order を使ってローカル配列をイテレートするループが最も高速な選択肢ということです。

興味深い例として、コンマ区切りの文字列をループするコードについて考えます。

そのようなループをできるだけ手短に書くと、次のようになります。

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

とても分かりやすいですね。でも、多くのコーディングスタイルガイドラインは次のようなコードを提案するかもしれません。

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

各イテレーションで終了条件は評価されていないので、この 2 つのコードにパフォーマンス面での違いはありません。
(初めは誤解していましたが、これはパフォーマンスの問題ではなく、単にスタイルの違いだということを Mark が指摘してくれました。)

コンマ区切りの文字列の場合は、このパフォーマンスを高めることが可能です。
文字列が長くなるにつれ、$Piece(string,",",i) は、string を i 個目の piece の終わりまで処理することになるので、どんどん重くなっていきます。 これを改善するには $ListBuild リストを使用します。 例えば、$ListFromString、$ListLength、および $List を使うと、以下のようなコードを書けます。

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

この方が、特に piece が長い場合は、$Length/$Piece を使うよりもパフォーマンスが良くなります。

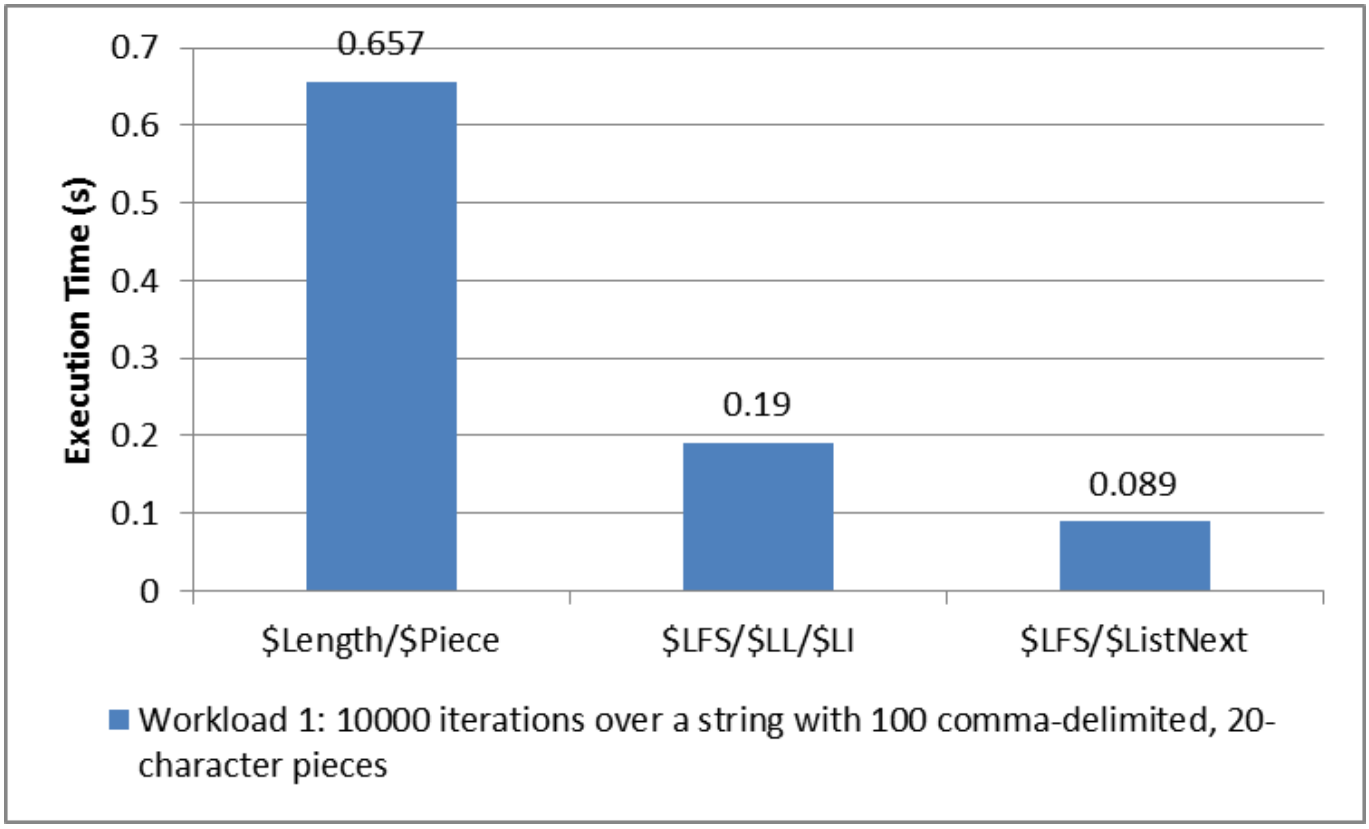
$Length/$Piece を使った方法では、n の各イテレーションで piece の最初の i に渡される文字がスキャンされています。 一方の $ListFromString/$ListLength/$List を使った方法では、n の各イテレーションで $ListBuild 構造の i ポインターを追いかけています。
この方が高いパフォーマンスを得られますが、それでも実行時間は $O(n^2)$ のままです。 loop によりリストの内容が変更されないことを想定した場合、$ListNext を使えば O(n) を改善することができます。

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

$List のように、毎回、次のポインタはリストの先頭から i個目の piece までを移動するのではなく、変数「pointer」がリスト内の現在位置を把握しています。 したがって、合計 n(n+1)/2 回 ($Listではn 回のイテレーションでそれぞれ i 回) の「次のポインタ」操作は行わずに、単純に操作を n 回 ($ListNextではイテレーションごとに 1 回) 実行するということになります。

最後に、文字列を整数の添え字が付いた配列に変換すると良いかもしれません。一般的に、$Order を使ってローカル配列をイテレートすると、$ListNext を使った場合よりも処理速度が少し、または大幅に改善します (リスト要素の長さによる)。
もちろん、カンマ区切りの文字列の場合は、配列に変換するのに少し手間がかかります。 繰り返しイテレートする場合や、リストを部分的に変更する必要がある場合、または逆方向にイテレートする必要がある場合は、手間をかけてでも行う価値があるでしょう。

以下は、異なる入力サイズごとに示した実行時間のサンプルです (必要な変換をすべて含む)。



これらの数字は以下から得ています。

```
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
```

コード (DC.LoopPerformance) を Gist で表示する

追加

ディスカッションの際に、他の方法でも良いパフォーマンスが得られることが判明しましたので、お互いを比較しておく価値があるでしょう。 RunLinearOnly メソッドとテストを実施した様々な実装を最新版の Gist でご覧ください。

```
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
```

以下のチャートは、これらのメソッドの While/Do...While をそれぞれ比較したものです。
特に、$ListFromString/$ListNext と $Extract/$Find、および文字列から $ListBuild
リストへの変換をせずに使用する場合の $ListNext と整数の添え字を付けたローカル配列で使用する $Order
を比較した相対的なパフォーマンスに注目してください。

まとめ

- コンマ区切りの文字列から始める場合は、$ListFromString/$ListNext
  を使った方がわずかにより直感的なコードを書けますが、パフォーマンスの面では $Find/$Extract
  が最良な選択肢となります。
- データ構造のチョイスを考慮した場合、 $ListBuild リストのトラバーサルは、入力の小さい同等のローカル配列よりもわずかにパフォーマンスが優れているように思える一方で、入力が大きい場合はローカル配列の方がかなり高いパフォーマンスを提供します。
  パフォーマンスにそれほど大きな違いがある理由は分かっていません。
  (これに関連して、整数の添え字を付けたローカル配列と$ListBuild
  リストにおけるランダムな挿入と削除のコストを比較する価値はあるでしょう。このような配列では、set $list を使った方が、わざわざ値を移動させるよりも処理が速くなると思います。)
- 引数なしの同等の for loop と比較すると、While loop または Do...While loop の方が *わずかに*
  高いパフォーマンスを発揮します。

#Code Snippet #ObjectScript #コーディングのガイドライン #ヒントとコツ #Caché

ソースURL:
https://jp.community.intersystems.com/post/cach%C3%A9-objectscript-%E3%81%A7%E3%83%91%E3%83%95%E3%82%A9%E3%83%BC%E3%83%9E%E3%83%B3%E3%82%B9%E3%81%AE%E8%89%AF%E3%81%84-loop-%E3%82%92%E6%9B%B8%E3%81%8F%E6%96%B9%E6%B3%95%E3%81%AB%E3%81%A4%E3%81%84%E3%81%A6