有趣的.NET 記憶體管理探索-StringBuilder 與 String 效能大車拼

作者：李明儒

大部分的.NET 程式設計師都知道，要做大量字串串接，StringBuilder 會比 String 來得有效率，本文由一個反例談起，探究二者效能差異的根源，順道一窺.NET 記憶體管理的內幕。

StringBuilder 永遠比較快?

事情要從一次 Code Review 談起，看到有段程式在指定 SQL 查詢語法時採取了如圖 1 的分列寫法。在程式碼中陳述 SQL 語句，將 SELECT、FROM、WHERE 分行擺放是很好的習慣，有助後人(或自己)閱讀及修改，是一種"佛心"的表現。但我發現在這段 Code 裡涉及的 SQL 語句內容是固定的，變數部份都是透過":tradedate"這種格式做 ORACLE 參數宣告，稍後再用 OracleParameter 指定變數內容。引起我注意的是，程式中使用了 StringBuilder.Append()串接多列字串，而不是直接用 String 相加。。

```
static string GetSqlString_1()
{
    StringBuilder sqlCmd;
    sqlCmd = new StringBuilder("");
    sqlCmd.Append(
"select customer.customername || ' ' as cstname,vendor.vbename || ' ' as vbename,");
    sqlCmd.Append(" tradepara.cparavalue || ' ' as Bank,");
    sqlCmd.Append(" vendor.brkcap ||' '|| vendor.brkcapvalue as CCASS,");
    sqlCmd.Append(" contract.tradername ||'      '|| contract.traderphone as Person,");
    sqlCmd.Append(" contract.bankno || ' ' as BIC,contract.traderemail || ' ' as email");
    sqlCmd.Append(" from sbtrade");
    sqlCmd.Append(" left join customer on sbtrade.corpid = customer.corpid");
    sqlCmd.Append(" and sbtrade.customerid = customer.customerid");
    sqlCmd.Append(" left join vendor on sbtrade.corpid = vendor.corpid");
    sqlCmd.Append(" and sbtrade.secbrkid = vendor.secbrkid");
    sqlCmd.Append(" left join tradepara on sbtrade.corpid = tradepara.corpid");
    sqlCmd.Append(" and tradepara.tradeid=:tradeid");
    sqlCmd.Append(" and sbtrade.customerid = tradepara.customerid");
    sqlCmd.Append(" and tradepara.cparaid='CUSTBANKNAME'");
    sqlCmd.Append(" left join contract on sbtrade.corpid = contract.corpid");
    sqlCmd.Append(" and sbtrade.mktcodeid = contract.mktcodeid");
    sqlCmd.Append(" and sbtrade.secbrkid = contract.secbrkid");
    sqlCmd.Append(" and contract.sbktype='CSI'");
    sqlCmd.Append(" and sbtrade.secaccount = contract.secaccount");
    sqlCmd.Append(" where sbtrade.corpid=:gCorpId and sbtrade.tradetype='BS'");
    sqlCmd.Append(" and to_char(sbtrade.tradedate,'yyyy/MM/dd') = :tradedate");
    sqlCmd.Append(" and sbtrade.mktcodeid=:mktcodeid");
    sqlCmd.Append(" and sbtrade.secbrkid = :secbrkid");
    sqlCmd.Append(" and sbtrade.secaccount = :secaccount");
    sqlCmd.Append(" and sbtrade.customerid = :customerid");

    return sqlCmd.ToString();
}
```

圖 1 使用 StringBuilder 串接靜態字串

我大膽猜測，程式作者原本想用字串直接相加(如圖 2 GetSqlString_2)，忽然腦海深處響起了眾多程式設計師熟知的鐵律："String 串接極沒效率，請改用 StringBuilder!"，所以才改用

StringBuilder.Append()。然而，針對這個修改，我有不一樣的想法。

```
static string GetSqlString_2()
{
    string sql =
"select customer.customername || ' ' as cstname,vendor.vbename || ' ' as vbename," +
    " tradepara.cparavalue || ' ' as Bank," +
    " vendor.brkcap ||' '|| vendor.brkcapvalue as CCASS," +
    " contract.tradername ||'      '|| contract.traderphone as Person," +
    " contract.bankno || ' ' as BIC,contract.traderemail || ' ' as email" +
    " from sbtrade" +
    //...中間省略...
    " and sbtrade.mktcodeid=:mktcodeid" +
    " and sbtrade.secbrkid = :secbrkid" +
    " and sbtrade.secaccount = :secaccount" +
    " and sbtrade.customerid = :customerid";
    return sql;
}

static string GetSqlString_3()
{
    string sql = @"
select customer.cstname || ' ' as customername,vendor.vbename || ' ' as vbename,
tradepara.cparavalue || ' ' as Bank,
vendor.brkcap ||' '|| vendor.brkcapvalue as CCASS,
contract.tradername ||'      '|| contract.traderphone as Person,
contract.bankno || ' ' as BIC,contract.traderemail || ' ' as email
from sbtrade
    //...中間省略...
and sbtrade.mktcodeid=:mktcodeid
and sbtrade.secbrkid = :secbrkid
and sbtrade.secaccount = :secaccount
and sbtrade.customerid = :customerid ";
    return sql;
}
```

圖 2 String 字串相接與@-Quoted String Literal 表示法

大部分的.NET 開發者都知道，要做大量的字串相加，StringBuilder 會比 String 相加快上 N 倍[參 1]。但若進一步探究，這個效能差異源於 String 物件的不可變動(Immutable)特性-- String 物件一旦建立，就無法改變長度，每次"動態相加"後都必須捨棄原字串佔用的記憶體空間，重新配置記憶體建立新字串物件儲存相加後的內容，重新配置的過程十分消耗資源，因而扼殺了效能。只是這背後的原理頗為曲折，我們往往忘了所以然，腦海只留下"串接字串千萬要用 StringBuilder，用 String 相加會被人恥笑"這個過度簡化的結論。

回到前述的例子，整段 SQL 指令內容在撰寫程式時就固定了，並無涉及 Runtime 才決定的變數；這段靜態的文字內容，理應在 Compiler 編譯時就被整合成一長串，不需要在執行期間進行任何字串串接動作。既然沒有動態的字串串接，前面所提到的重新配置記憶體問題就不會發生，也就沒有任何 Runtime 時期的效能損耗；改寫成 StringBuilder.Append()，涉及 Runtime 時期的物件建立與 Method 呼叫，效能反而較差。

針對這個推論，我打算從兩個方向驗證：

疑問 1 是否程式碼中的固定內容字串串接，在編譯(Compile)時會自動接成一長串，不必在執行期間執行字串

接合?

疑問 2 沒有實際的字串串接動作，直接指定 String 是否會比 StringBuilder 來得快?

靜態字串相接的效能比較

針對疑問 1，我的驗證方法是寫一小段包含靜態字串相加的程式碼，編譯後再用 Reflector 檢視。如圖 3 所示，反編譯後的程式碼中呈現的是已接合過的字串內容，表示靜態字串相接已在編譯過程做掉，不會產生執行期間的額外動作。

```
public class sample
{
    static void Main()
    {
        string s = "Hello" + " " + "World!";
        Console.WriteLine(s);
    }
}
```

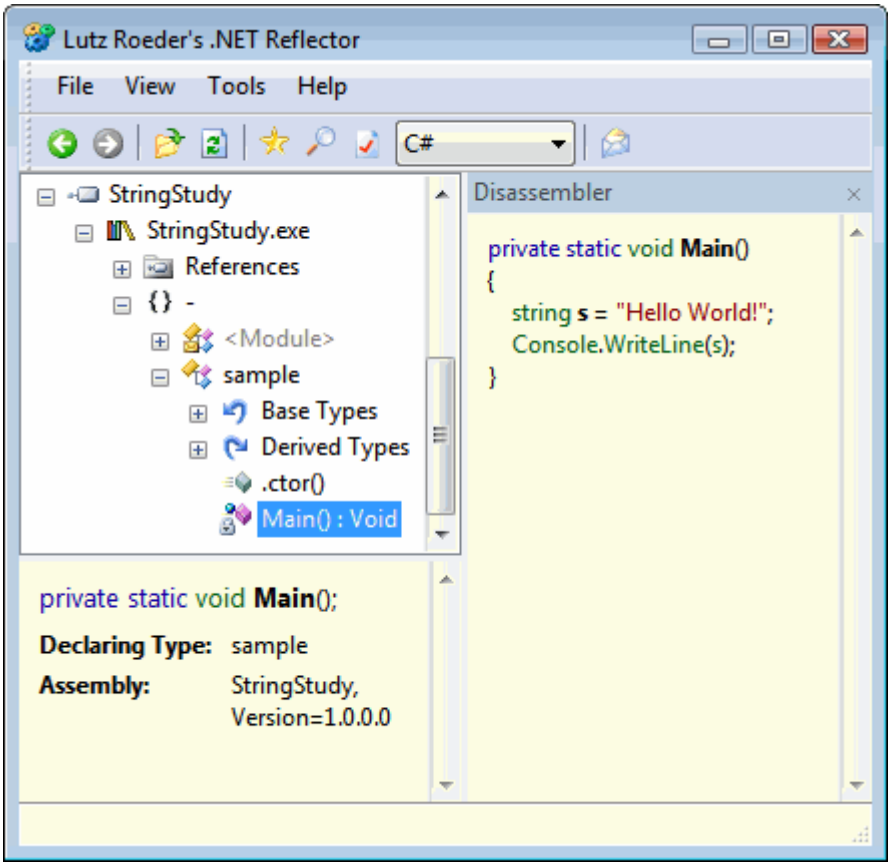

圖 3 用 Reflector 驗證靜態字串相接在編譯時會自動接合

至於疑問 2，要得到答案最簡單的方法也是寫一段程式實地 Benchmark 一番，答案立見分曉。我在測試程式中另外補上@-Quoted String Literal[參 2]的寫法(如圖 2 GetSqlString_3)，對於這種包含大量換

行，或需要多行表示的文字內容(例如: SQL 語句、Javascript 等)，我偏愛使用 C#提供的@” “表示法，輸入文字時可以直接換行而不必改寫成”\n”，寫來順手，又十分直覺易讀。

測試程式碼如程式 1，分別執行 GetSqlString_1(),GetSqlString_2()及 GetSqlString_3()各 100 萬次，事前預測是 1 最慢，2, 3 一樣快，沒想到，StringBuilder 慢的程度比想像嚴重許多:

```
Test1=4152ms
Test2=6ms
Test3=7ms
```

程式 1 靜態字串串接比較

```
Stopwatch sw = new Stopwatch();
int times = 1000000;
sw.Start();
for (int i = 0; i < times; i++)
    GetSqlString_1();
sw.Stop();
Console.WriteLine(string.Format("Test1={0}ms", sw.ElapsedMilliseconds));
sw.Reset();
sw.Start();
for (int i = 0; i < times; i++)
    GetSqlString_2();
sw.Stop();
Console.WriteLine(string.Format("Test2={0}ms", sw.ElapsedMilliseconds));
sw.Reset();
sw.Start();
for (int i = 0; i < times; i++)
    GetSqlString_3();
sw.Stop();
Console.WriteLine(string.Format("Test3={0}ms", sw.ElapsedMilliseconds));
```

SOS

寫到這裡，可能讓許多人迷糊了，老師有教、書上有講，StringBuilder 根本就是為了改善字串串接效率才誕生的，怎麼會在接字串這檔事上輸得慘不忍睹? 關鍵在於靜態字串相接的工作已由編譯器搞定，StringBuilder 做事情的速度再快，也不可能比”什麼都不必做”來得快。

不過，這個議題引起我對字串相接效能低落理論的興趣，如前面所提，字串相接的無效率，源於每次改變字串內容，.NET Runtime 就必須放棄原來存放字串物件的記憶體位置，重新要一塊記憶體擺放變更後的新字串，這是字串物件的特性。重新配置記憶體涉及到複雜的記憶體管理，若短時間內頻繁地反覆執行笨重的記憶體配置工作，便成了拖累效能的主因。話雖如此，我們能不能親眼見證這一點呢?

記憶體管理屬於.NET Runtime 核心層的工作，我們不容易撰寫.NET 程式直接獲知它的細節，所以要藉

助較低階的偵錯工具。提到.NET 進階偵錯，就不能不提到 SOS Debugging Extension(SOS.DLL)。

SOS Debugging Extension 被設計用來協助 Windows Debugger 及 Visual Studio 獲取.NET Runtime(CLR)環境的資訊，要揭開 CLR 記憶體管理的神祕面紗，非它莫屬。要使用 SOS，可以透過 Windows Debugger 或 Visual Studio，Windows Debugger 的操作頗為複雜繁瑣，從 Visual Studio 下手較為簡便。

啟用 SOS 前，Visual Studio 要做些設定，首先要設定 Debugger 時參照的 Symbol 檔，微軟有一台伺服器 http://msdl.microsoft.com/download/symbols 儲放了各版本 Windows、.NET Runtime 裡大小 dll 所對應的 pdb 檔，Debugger 可以視需要自動下載。所以在 Visual Studio 要設定下載來源 URL 以及 Cache 目錄位置(如圖 4)。第一次得花較長時間下載檔案，但之後可沿用 Cache 中的檔案，速度就會快很多。

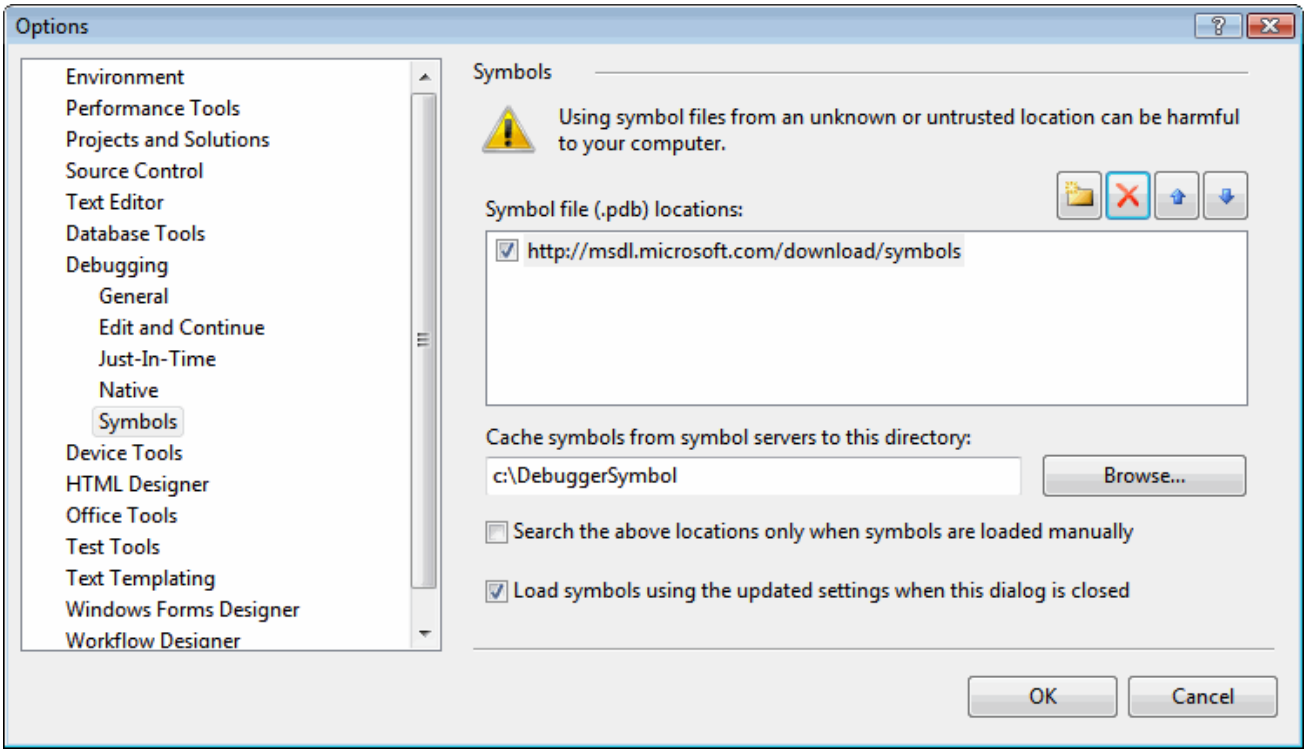

圖 4 在 Visual Studio 設定 Symbol 檔下載及 Cache 目錄

除了設定 Symbol 檔，專案還要開啟非.NET Code(Unmanaged code)的偵錯功能(如圖 5)才能啟用 SOS。把這兩個選項設定好，就可以準備一窺.NET CLR 記憶體管理的奧祕了。

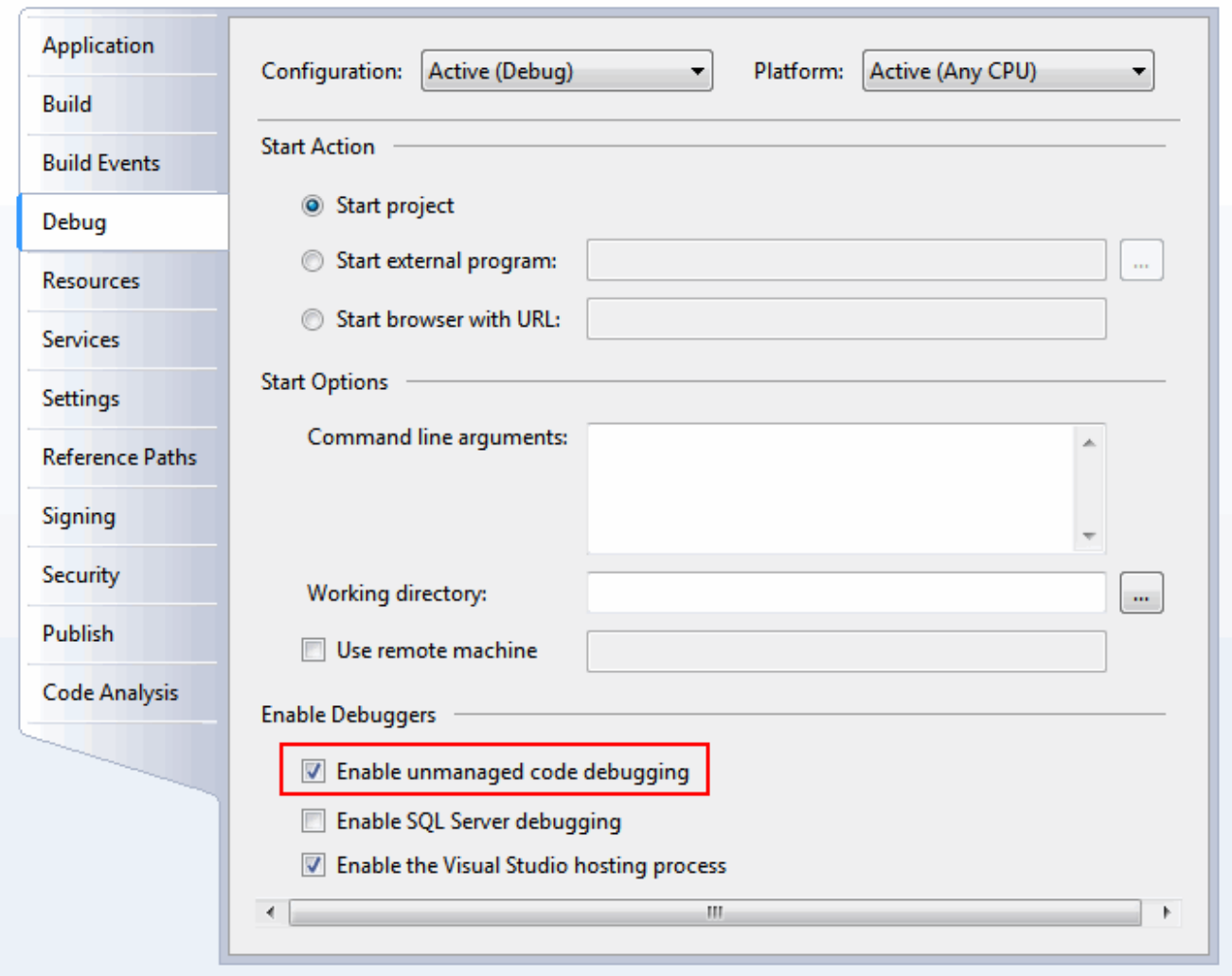

圖 5 啟用專案的非.NET 程式碼偵錯

基本 SOS 指令

要在 Visual Studio 中使用 SOS，可在程式中設定中斷點(Break Point)，進入偵錯模式後在即時運算視窗(Immediate Window)輸入 load sos 載入 SOS.DLL，接著就可使用 SOS 的指令查詢.NET CLR 相關資訊。

部分 SOS 指令需具備完整的 CLR 底層知識才易理解，我們的目標在觀察字串物件的記憶體配置原理，因此只聚焦三個簡單的指令，完整的 SOS 指令列表可參考[參 3]。另外，實地觀察之前，需要先搞懂 Stack、Heap、Value Type、Reference Type 幾項.NET CLR 基本觀念，[參 4]有頗為詳實易懂的說明可供參考，在此不多贅述。

clrstack -l	列出儲放於堆疊(Stack)中的區域變數
dumpobj <address>	顯示特定記憶體位址上的物件資訊
dumpheap -stack	顯示堆積(Heap)使用狀況統計

光看指令有點讓人迷惘，不如就透過實例來了解吧！我們想用 SOS 指令做的第一件事是，證明 String 在相加後，會另外配置記憶體存放新的 String 物件。以下是一段簡單的字串相接邏輯：

```
public class sample
```

```
{
    static void Main()
    {
        string s = "";
        s += "ABC";
        s += "123";
        return;
    }
}
```

為了要觀察記憶體使用的變化，我們打算採行的方式是在 string s="";上設定偵錯中斷點，然後再一列一列向下執行，每執行一列就觀察一次記憶體使用狀況。進入偵錯中斷後，開啟即時運算視窗(Immediate Window)，利用.load sos 載入 SOS，接著!clrstack -l，便可看到區域變數在 Stack 中的儲放情形。由於 String 是 Reference Type，真正的資料放在 Heap，Stack 中儲存的是指向記憶體位址的指標。不過在第一個中斷點時，s 還沒有被指定成空字串，相當於 null 的狀態，因此會看到 0x00000000。

```
.load sos
extension C:\Windows\Microsoft.NET\Framework\v2.0.50727\sos.dll loaded
!clrstack -l
PDB symbol for mscorwks.dll not loaded
OS Thread Id: 0x28dc (10460)
ESP      EIP
003feee4 00520094 sample.Main()
    LOCALS:
        <CLR reg> = 0x00000000
003ff13c 79e7c74b [GCFrame: 003ff13c]
```

按下F10執行string s="";，再做一次!clrstack -l，變數指標有了數值0x015b1574，這是一個記憶體位址，也就是實際儲存字串物件的地方。接著用另一個指令!dumpobj 0x015b1574檢視物件內容。由dumpobj的結果，可以得到幾項情報：1)這是一個System.String物件，雖然為空字串，但有幾個基本屬性：2個Int32(4 bytes), 1個char(2 bytes)及String.Empty、String.WhitespaceChars兩個指標(4 bytes)，共佔用了4*2 + 2 + 4*2 = 18 bytes，即Size: 18(0x12) bytes的由來。m_stringLength為0代表現為空字串，m_arrayLength相當於Capacity+1，空字串時為1。

```
!clrstack -l
OS Thread Id: 0x28dc (10460)
ESP      EIP
003feee4 0052009c sample.Main()
    LOCALS:
```

```
       <CLR reg> = 0x015b1574

003ff13c 79e7c74b [GCFrame: 003ff13c]

!dumpobj 0x015b1574
Name: System.String
MethodTable: 790fd8c4
EEClass: 790fd824
Size: 18(0x12) bytes
 (C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll)
String:
Fields:
     MT    Field   Offset                 Type VT     Attr    Value Name
79102290  4000096       4        System.Int32  1 instance        1 m_arrayLength
79102290  4000097       8        System.Int32  1 instance        0 m_stringLength
790ff328  4000098       c         System.Char  1 instance        0 m_firstChar
790fd8c4  4000099      10       System.String  0   shared   static Empty
   >> Domain:Value  005602e8:790d884c <<
7912dd40  400009a      14       System.Char[]  0   shared   static WhitespaceChars
   >> Domain:Value  005602e8:015b1440 <<
```

按下F10執行s += "ABC";，再次!clrstack -l，我們會發現字串物件的記憶體位址由剛才的0x015b1574變成了0x015c6590，使用!dumpobj追查，這還是一個字串物件，只是m_stringLength=3，字串內容為ABC，Size也因為ABC三個字元的Unicode編碼佔用6個bytes而變成18 + 6 = 24 bytes。

```
!clrstack -l
OS Thread Id: 0x28dc (10460)
ESP      EIP
003feee4 005200ad sample.Main()
   LOCALS:
       <CLR reg> = 0x015c6590
(…略…)

!dumpobj 0x015c6590 (省略部分顯示內容)
Name: System.String
(...略…)
Size: 24(0x18) bytes
String: ABC
```

```
Fields:
      MT    Field   Offset                 Type VT     Attr    Value Name
79102290  4000096       4        System.Int32  1 instance        4 m_arrayLength
79102290  4000097       8        System.Int32  1 instance        3 m_stringLength
790ff328  4000098       c         System.Char  1 instance       41 m_firstChar
(…略…)
```

跑完s += "123"; 再做一次!clrstack -l，會發現字串物件的記憶體位址再次變動成0x015c65c0，新位址物件的字串長度為6，佔用30 bytes。

```
!clrstack -l
(…略…)
   LOCALS:
      <CLR reg> = 0x015c65c0

!dumpobj 0x015c65c0
Name: System.String
(…略…)
Size: 30(0x1e) bytes
String: ABC123
Fields:
      MT    Field   Offset                 Type VT     Attr    Value Name
79102290  4000096       4        System.Int32  1 instance        7 m_arrayLength
79102290  4000097       8        System.Int32  1 instance        6 m_stringLength
790ff328  4000098       c         System.Char  1 instance       41 m_firstChar
```

由以上的觀察，借助SOS檢視CLR內部資訊的能力，驗證了"字串相加後會變成放在新記憶體位址的新物件"。那StringBuilder呢?

StringBuilder內部也是透過String方式保存資料，只是在記憶體配置上採行較積極的管理策略。建構時若未指定Capacity，一開始它會宣告一個最大長度為16的字串，Append後若字串長度超過16個字元，才會重新宣告一個兩倍大(32個字元)的字串，還不夠，再逐步放大成64、128、256個字元長度，以此類推。由以上的說明，可以得知StringBuilder也會發生重新配置記憶體給字串物件的情況，只是它每次並不只是配置剛好儲放新字串長度的大小，而是採取16, 32, 64, 128, 256, 512…加倍放寬的策略，在大量字串相接的情境中，可減少可觀的重建新字串次數。

了解StringBuilder的原理，就用SOS來驗證一下吧! 測試程式的寫法很簡單，建立StringBuilder，先Append("ABCD")，再Append("1234")，此時長度為8，內部字串物件還不需更新，但再Append("0123456789")，便會因為長度變成18>16而需另外建立新的內部字串物件。

```
public class sample
{
    static void Main()
    {
        StringBuilder sb = new StringBuilder();
        sb.Append("ABCD");
        sb.Append("1234");
        sb.Append("0123456789");
        return;
    }
}
```

在sb.Append("ABCD")設好偵錯中斷點，透過!clrstack -l及!do(dumpobj可縮寫為do)，可以看到StringBuilder物件共有三個Property，其中m_StringValue即是儲存文字內容的字串物件，用!do再檢視m_StringValue，可以發現它的m_stringLength是0，m_arrayLength卻是17，表示這是一個可以容納16個字元的字串物件，但此時沒有資料，為空字串。

```
!clrstack -l
(…略…)
    LOCALS:
        <CLR reg> = 0x013a65f0

!do 0x013a65f0
Name: System.Text.StringBuilder
(..略…)
Size: 20(0x14) bytes
Fields:
     MT    Field   Offset                 Type VT     Attr    Value Name
791016bc  40000b1       8       System.IntPtr  1 instance 000CB688 m_currentThread
79102290  40000b2       c        System.Int32  1 instance 2147483647 m_MaxCapacity
790fd8c4  40000b3       4       System.String  0 instance 013a6604 m_StringValue

!do 013a6604
Name: System.String
(…略…)
Size: 50(0x32) bytes
 (C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll)
String:
```

```
Fields:
      MT    Field   Offset                 Type VT     Attr    Value Name
79102290  4000096       4        System.Int32  1 instance       17 m_arrayLength
79102290  4000097       8        System.Int32  1 instance        0 m_stringLength
790ff328  4000098       c         System.Char  1 instance        0 m_firstChar
```

接著陸續執行sb.Append("ABCD")、sb.Append("1234")，可以發現m_StringValue仍指向同一個物件，但字串內容由空字串變成"ABCD"，再變成"ABCD1234"， m_stringLength也由0增加到4，再變到8。

```
!do 013a6604
(…略…)
String: ABCD
Fields:
      MT    Field   Offset                 Type VT     Attr    Value Name
79102290  4000096       4        System.Int32  1 instance       17 m_arrayLength
79102290  4000097       8        System.Int32  1 instance        4 m_stringLength
(…略…)

!do 013a6604
(…略…)
String: ABCD1234
Fields:
      MT    Field   Offset                 Type VT     Attr    Value Name
79102290  4000096       4        System.Int32  1 instance       17 m_arrayLength
79102290  4000097       8        System.Int32  1 instance        8 m_stringLength
(…略…)
```

sb.Append("0123456789")故意讓字串總長度變成18，此時我們預期StringBuilder必須要另行建立最大長度為32的字串。果不其然，m_ StringValue由013a6604變成了013a6778，再檢視新字串的m_arrayLength=33，相當於最大長度為32，符合我們的推測。

```
!do 0x013a65f0
Name: System.Text.StringBuilder
(…略…)
Size: 20(0x14) bytes
Fields:
      MT    Field   Offset                 Type VT     Attr    Value Name
```

```
791016bc  40000b1       8       System.IntPtr  1 instance 00000000 m_currentThread
79102290  40000b2       c        System.Int32  1 instance 2147483647 m_MaxCapacity
790fd8c4  40000b3       4       System.String  0 instance 013a6778 m_StringValue
(…略…)

!do 013a6778
(…略…)
Size: 82(0x52) bytes
String: ABCD12340123456789
Fields:
     MT    Field   Offset                 Type VT     Attr    Value Name
79102290  4000096       4        System.Int32  1 instance       33 m_arrayLength
79102290  4000097       8        System.Int32  1 instance       18 m_stringLength
(…略…)
```

由以上的觀察，得知 StringBuilder 之所以比直接字串相接來得有效率，是透過預先配置較大的字串可用長度，並在長度不夠用時採取放大兩倍的策略，有效減少重新建立字串物件的次數；換言之，如果可以預知 StringBuilder 要儲放內容的最大長度，在宣告時一次給足，甚至可省略 16, 32, 64, 128 的倍增過程，達到最佳效能。

字串相接大車拼

最後，我們就用一個實地效能測試做結尾。目標是以每次 16 個字元為單位串接出一個長字串，執行 65536 次，最後串成 1M 大小的字串。我們分別用直接字串相加、new StringBuilder()、new StringBuilder(1024*1024)三種方法測試，使用 StopWatch 計時，並用!dumpheap -stat 來檢視記憶體使用狀況。

```
public class sample
{
    static void test1()
    {
        string s = "";
        for (int i = 1; i < 1024 * 64; i++)
            s += "0123456789ABCDEF";
        Console.WriteLine(s.Length);
    }
    static void test2()
    {
        StringBuilder sb =
```

```
            new StringBuilder();
        for (int i = 1; i < 1024 * 64; i++)
            sb.Append("0123456789ABCDEF");
        Console.WriteLine(sb.Length);
    }
    static void test3()
    {
        StringBuilder sb =
            new StringBuilder(1024 * 1024);
        for (int i = 1; i < 1024 * 64; i++)
            sb.Append("0123456789ABCDEF");
        Console.WriteLine(sb.Length);
    }
    static void Main()
    {
        System.Diagnostics.Stopwatch sw =
            new System.Diagnostics.Stopwatch();
        sw.Start();
        test1(); //or test2(), test3()
        sw.Stop();
        Console.WriteLine(sw.ElapsedMilliseconds);
        return; //Break here to !dumpheap -stat
    }

}
```

測試結果如下，直接字串相加耗時 125,267ms 且一路 CPU 滿載破表，new StringBuilder()耗時 9ms，而 new StringBuilder(1024*1024)最快，只耗時 6ms。

使用!dumpheap -stat 可以看出在 Heap 中存放字串的次數及大小統計，字串直接相加耗用了約 6M，new StringBuilder()耗用了約 4.5M，而 StringBuilder(1024*1024)只耗用了約 2.4M。表示準確掌握 StringBuilder 的 Capacity 可以達到最佳效能與記憶體使用效率。

```
test1()
!dumpheap -stat
total 6689 objects
Statistics:
      MT    Count    TotalSize Class Name
791334a8        1           12
```

```
System.Collections.Generic.GenericEqualityComparer`1[[System.String, mscorlib]]
79119954       1          12 System.Security.Permissions.ReflectionPermission
…略…
004a05d0      14     2097044     Free
790fd8c4    5233     6585276 System.String
Total 6689 objects

test2()
!dumpheap -stat
total 6948 objects
Statistics:
     MT    Count    TotalSize Class Name
791334a8       1          12
System.Collections.Generic.GenericEqualityComparer`1[[System.String, mscorlib]]
79128334       1          12 System.Security.SecurityZone
…略…
7912d8f8     394       40076 System.Object[]
790fd8c4    5256     4491280 System.String
Total 6948 objects

test3()
!dumpheap -stat
total 6928 objects
Statistics:
     MT    Count    TotalSize Class Name
791334a8       1          12
System.Collections.Generic.GenericEqualityComparer`1[[System.String, mscorlib]]
79128334       1          12 System.Security.SecurityZone
…略…
7912d8f8     394       40076 System.Object[]
790fd8c4    5240     2393824 System.String
Total 6928 objects
```

測試結果整理如後，證明了 StringBuilder 在大量動態字串相加的效能評比上獲得了壓倒性的勝利。但若字串相加次數不多，差距就不會這麼明顯，換句話說，如果字串不長，且只相加個兩三次，執行頻率也不高，使用字串直接相加倒也不至於罪無可赦；但如果力求保險，除非如一開始所舉的純靜態字串接合特例，其餘情境用 StringBuilder 執行字串相加肯定效能不會較差。

方法	耗時	耗用記憶體
直接字串相加	125,267 ms	6,585,276 bytes
new StringBuider()	9 ms	4,491,280 bytes
new StringBuilder(1024*1024)	6 ms	2,393,824 bytes

結論

由一場 StringBuilder()意外輸給 String 直接相加的效能測試開始，我們對 String 的 Immutable 特性，StringBuilder 字串相接的效能優勢做了剖析，並使用 SOS 偵錯延伸模組驗證理論，也對.NET CLR 的記憶體管理做了粗淺的觀察，希望藉著這些實驗，讀者對於字串相加的效能議題從此能有更深刻的體認。

參考資料

參 1 動態字串相加的效能比較 http://www.codeproject.com/KB/cs/stringperf.aspx
參 2 @-Quoted String Literals
http://msdn2.microsoft.com/en-us/library/362314fe(VS.71).aspx
參 3 SOS 指令說明 http://msdn2.microsoft.com/en-us/library/bb190764(VS.80).aspx
參 4 .NET 記憶體配置概論 **http://tinyurl.com/pnyo4**