#大標=JDK 5.0 Tiger Enhanced for loop
#副標=徹底剖析 JDK 5.0 Enhanced for loop
#引言=從上期所介紹的 Varargs，相信許多讀者對於 JDK 5.0 Tiger 所增加的新功能，仍然保持著想探望一窺底層的實作細節與關注，本文這次將會介紹另一個好用的功能「Enhanced for loop」，我們就來看看這隻「老虎」是如何的處理機制。
#作者=文/陳縈紳 http://blog.ring.idv.tw
#內文=

在開始進入主題之前，筆者在此先介紹一些迴圈結構，相信許多學過程式語言的朋友們，在學習一門程式語言的過程之中，必然會碰到所謂的迴圈(loop)結構，而在 Java 裡的迴圈結構不外乎就是 for loop、while loop 和 do/while loop，其中 for loop 和 while loop 兩者算是「先判斷後執行的迴圈結構(pretest loop)」，而 do/while loop 則是「先執行然後再判斷是否持續執行的迴圈結構(post-test loop)」，基本上大部份的程式語言都擁有此三種迴圈結構。

談完了迴圈結構我們再回頭來看看本文的主題「Enhanced for loop」，在 Flash ActionScript 的語法中稱爲 for in，而在 PHP 也有稱爲 foreach 的說法，不過在這隻「老虎」的革新之下，我們也可稱它爲「Enhanced for loop」，從它的字眼間我們不難看出它意謂著「增強型」的 for loop，既然是「增強型」的 for loop，想當然定有不同於 for loop 之處，不過筆者在此還是先向各位透露，其實 Enhanced for loop 的底層實作機制就是「for loop」迴圈結構而已，只是透過 Java Compiler 在編譯時期幫我們做一些「料理」，雖然從字面上看起來感覺很微妙，但是相信各位讀者咀嚼過本文之後，便能夠體會這其中的奧秘。

#中標=剖析 for loop 底層

在開始探究「Enhanced for loop」的功能之前，我們先來看看傳統 for loop 的實際演練，通常我們要輸出一整個陣列的值，不外乎需要一個累加計數的整數值，將它用來當做所要存取陣列的索引位置，筆者習慣上都會宣告一個「i」變數，此種傳統 for loop 的優點是我們可以隨心所欲地在迴圈上控制所需要存取的索引值，而缺點就是當我們要依序輪詢所有陣列值時，我們仍然需要「手動」地撰寫這「制式化」的程式碼，現在我們就先來透過 Javap 反組譯器，來剖析下列程式底層的位元碼處理方式!(如程式一)

==========

程式一：

```
public void testForLoop(int[] value)
{
    int valueLength = value.length;
    for( int i = 0 ; i < valueLength ; i++)
    {
        System.out.println(value[i]);
    }
}
```

==========

```
public void testForLoop(int[]);
  Code:
   Stack=3, Locals=4, Args_size=2
   0:   aload_1
   1:   arraylength
   2:   istore_2
   3:   iconst_0
   4:   istore_3
   5:   iload_3
   6:   iload_2
   7:   if_icmpge       25
   10:  getstatic       #2; //Field java/lang/System.out:Ljava/io/PrintStream;
   13:  aload_1
   14:  iload_3
   15:  iaload
   16:  invokevirtual   #3; //Method java/io/PrintStream.println:(I)V
   19:  iinc    3, 1
   22:  goto    5
   25:  return
```

圖一　　testForLoop 方法的位元碼(如附檔：圖一.jpg)

```
LocalVariableTable:
 Start  Length  Slot  Name   Signature
 5      20      3     i      I
 0      26      0     this       Leloop;
 0      26      1     value      [I
 3      23      2     valueLength       I
```

圖二　　testForLoop 方法的 LocalVariableTable(如附檔：圖二.jpg)

#小標=**testForLoop 方法的位元碼：**

位元碼	說明
0# aload_1 ➔ 將從 local variable[1]載入 object reference 至 Operand Stack。	「在此就是用來取得參考到 int 陣列的物件參考，來將它載入至 Operand Stack」。
1# arraylength ➔ 取得陣列的長度。	「會將參考到 int 陣列的 object reference 從 Operand Stack 取出並推入一個 int 整數至 Operand Stack，而此整數就代表著陣列的長度」
2# istore_2 ➔ 儲存 int 整數至 local variable[2]。	「將陣列長度的值儲存在 local variable[2]」
3# iconst_0 ➔ 推入一個 int 常數「0」至 Operand Stack。	「對應 for loop 的第一個敘述：int i = 0，並將此整數儲存在 local variable[3]」
4# istore_3 ➔ 儲存 int 整數至 local variable[3]。	
5# iload_3 ➔ 取出 local variable[3]的 int 值，並推入 Operand Stack。	「對應 for loop 的第二個敘述： i < valueLength ;」
6# iload_2 ➔ 取出 local variable[2]的 int 值，並推入 Operand Stack。	
7# if_icmpge 25 ➔ 從 Operand Stack 取出前兩個 int 整數值來判斷，若符合「大於或等於」的條件，就跳至第 25 行執行	
10# getstatic ➔ 從類別取得 static 欄位(field)。	「從 10#至 16#這段位元碼，代表著 System.out.println(value[i]); 這段敘述，將值印在 console 上」
13# aload_1 ➔ 從 local variable[1]載入 object reference 至 Operand Stack。	
14# iload_3 ➔ 從 local variable[3]載入 int 值至 Operand Stack。	
15# iaload ➔ 從陣列中取出 int 值。	
16# invokevirtual ➔ 呼叫實體方法。	
19# iinc 3, 1 ➔ 將 local variable[3]的 int 值加 1。	「對應 for loop 的第三個敘述： i++」
22# goto 5 ➔ 跳至 5#繼續執行。	
25# return ➔ 回傳 void。	

上述即爲整個 testForLoop 方法的位元碼，我們可以從上述的位元碼來了解到 JVM 是如何透過「堆疊」的方式來處理 for loop，而讀者也可以根據圖二 LocalVariableTable 來了解 Local Variable 各個元素所對應的區域變數，並同時地對照位元碼來徹底理解底層的處理方式。

#中標=剖析 Enhanced for loop 底層

在看過 for loop 的位元碼之後，緊接著我們來看看這隻「老虎」的 Enhanced for loop 功能，來剖析一下究竟 Java Compiler 幫我們在底層加了那些「調味料」，使我們可以依序地輪詢陣列值，而又不必撰寫「制式化」的程式碼，使我們的程式碼看起來可以更簡化一番!不過或許有些讀者會認爲，既然都能達到同樣的程式效果，那我依照著習慣寫傳統的 for loop 不就好了，其實這完全取決於個人的看法，不過當我們在設計一個系統時，所要面對的是最前端的使用者在操作上順手與否，如果能省下一個「按鈕」按鍵，那一整天操作下來或許能省掉要按滑鼠幾百下或按鍵盤幾百次的動作，同樣的道理，當我們撰寫大量的程式時，如果我們能使用更簡化的程式寫法，相對地也提高我們撰寫程式的效率!

==========

程式二：

```
public void testForInLoop(int[] value)
{
    for( int i : value)
    {
        System.out.println(i);
    }
}
```

==========

```
public void testForInLoop(int[]);
  Code:
   Stack=2, Locals=6, Args_size=2
   0:   aload_1
   1:   astore_2
   2:   aload_2
   3:   arraylength
   4:   istore_3
   5:   iconst_0
   6:   istore  4
   8:   iload   4
   10:  iload_3
   11:  if_icmpge       34
   14:  aload_2
   15:  iload   4
   17:  iaload
   18:  istore  5
   20:  getstatic       #2; //Field java/lang/System.out:Ljava/io/PrintStream;
   23:  iload   5
   25:  invokevirtual   #3; //Method java/io/PrintStream.println:(I)V
   28:  iinc    4, 1
   31:  goto    8
   34:  return
```

圖三　　testForInLoop 方法的位元碼(如附檔：圖三.jpg)

```
LocalVariableTable:
  Start  Length  Slot  Name   Signature
  20      8       5    i      I
  2       32      2    arr$      [I
  5       29      3    len$      I
  8       26      4    i$     I
  0       35      0    this      Leloop;
  0       35      1    value      [I
```

圖四　　testForInLoop 方法的 LocalVariableTable(如附檔：圖四.jpg)

#小標=**testForInLoop 方法的位元碼：**

位元碼	說明
0# aload_1 ➔ 將從 local variable[1]載入 object reference 至 Operand Stack。	「用來取得參考到 int 陣列的 object reference，來將它載入至 Operand Stack」。
1# astore_2 ➔ 儲存 object reference 至 local variable[2]。	「將上述的 object reference，儲存至一個新的區域變數 **arr$**」
2# aload_2 ➔ 將從 local variable[2]載入 object reference 至 Operand Stack。	如 0# 的說明。
3# arraylength ➔ 取得陣列的長度。	請參考 testForLoop 方法的位元碼 1# 說明。
4# istore_3 ➔ 儲存 int 整數至 local variable[3]。	「將此陣列的長度存至一個新的區域變數 **len$**」
5# iconst_0 ➔ 推入一個 int 常數「0」至 Operand Stack。	「這裡如同 for loop 的第一個敘述：int **i$** = 0，並將此整數儲存在 local variable[4]」
6# istore 4 ➔ 儲存 int 整數至 local variable[4]。	
8# iload 4 ➔ 取出 local variable[4]的 int 值，並推入 Operand Stack。	「對應 for loop 的第二個敘述： **i$** < **len$** ;」
10# iload_3 ➔ 取出 local variable[3]的 int 值，並推入 Operand Stack。	
11# if_icmpge 34 ➔ 從 Operand Stack 取出前兩個 int 整數值來判斷，若符合「大於或等於」的條件，就跳至第 34 行執行	
14# aload_2 ➔ 從 local variable[2]載入 object reference 至 Operand Stack。	「可以將這幾個位元碼看成在 For 迴圈裡的一段敘述：**int i = arr$[i$];**」 **P.S.** 剩下的位元碼讀者可以自行參考上述 testForLoop 方法。
15# iload 4 ➔ 取出 local variable[4]的 int 值，並推入 Operand Stack。	
17# iaload ➔ 從陣列中取出 int 值。	
18# istore 5 ➔ 儲存 int 整數至 local variable[5]。	

在剖析 Enhanced for loop 的底層之後，我們可以清楚地了解這隻「老虎」幫我們加了「**arr$**」、「**len$**」、「**i$**」 這三個「調味料」，也就是如此地來完成簡化程式碼，使得可閱讀性大爲提高，所以當我們要依序地取出陣列的值或是輪詢集合(Collection)類別時，請別忘了有「Enhanced for loop」這個好幫手! 若讀者對上述位元碼說明仍有不甚了解之處，可以將下列已經經由 Java Compiler 摻雜「調味料」之後的程式來一併參考：(如程式三)

==========

程式三：

```
public void testForInLoop(int[] value)
{
        int[] arr$ = value;
        int len$ = arr$.length;
        for(int i$ = 0; i$ < len$; i$++)
        {
            int i = arr$[i$];
            System.out.println(i);
        }
```

```
}
```
==========

#中標=「For loop」Collection 類別

在品嚐了「Enhanced for loop」輪詢陣列的調味料之後，緊接著我們換換口味來品嚐輪詢 Collection 類別的滋味，現在回想一下在這隻「老虎」出閘之前，當我們要輪詢 Collection 類別時，通常的作法大都類似底下程式四：

==========

程式四：

```
public void testForLoopCollection(Collection c)
{
    for(Iterator i = c.iterator() ; i.hasNext() ; )
    {
        Integer value = (Integer) i.next();
        System.out.println(value);
    }
}
```
==========

```
public void testForLoopCollection(java.util.Collection);
  Code:
   Stack=2, Locals=4, Args_size=2
   0:   aload_1
   1:   invokeinterface #2,  1; //InterfaceMethod java/util/Collection.iterator:
()Ljava/util/Iterator;
   6:   astore_2
   7:   aload_2
   8:   invokeinterface #3,  1; //InterfaceMethod java/util/Iterator.hasNext:()Z

   13:  ifeq    36
   16:  aload_2
   17:  invokeinterface #4,  1; //InterfaceMethod java/util/Iterator.next:()Ljav
a/lang/Object;
   22:  checkcast       #5; //class Integer
   25:  astore_3
   26:  getstatic       #6; //Field java/lang/System.out:Ljava/io/PrintStream;
   29:  aload_3
   30:  invokevirtual   #7; //Method java/io/PrintStream.println:(Ljava/lang/Obj
ect;)V
   33:  goto    7
   36:  return
```

圖五　testForLoopCollection 方法的位元碼(如附檔：圖五.jpg)

```
LocalVariableTable:
  Start  Length  Slot  Name   Signature
  26      7      3    value      Ljava/lang/Integer;
  7       29     2    i       Ljava/util/Iterator;
  0       37     0    this       Leloop;
  0       37     1    c       Ljava/util/Collection;
```

圖六　testForLoopCollection 方法的 LocalVariableTable(如附檔：圖六.jpg)

#小標=**testForInLoopCollection 方法的位元碼：**

由於此方法的位元碼比較單純，所以筆者便不再一一地去深入剖析，而只針對 13# 來加以解釋，至於其它部份就留著給各位讀者自行細嚼探究了!

位元碼	說明
13# ifeq 36 ➔ 判斷位於 Operand Stack 最頂端的值是否相等於「0」，若相等則跳至 #36。	因爲 Boolean 值在 Java 位元碼是以「1」代表 true，而「0」代表 false，所以也就代表著上述 for loop 的第二個判斷敘述「i.hasNext()」。

#中標=「Enhanced for loop」Collection 類別

在開始剖析之前，我們先來觀察一下 JDK 5.0 Collection 繼承結構上的變化，從圖七可以發現 Collection 增加了繼承 java.lang.Iterable 介面(Interface)，簡單來說，就是將 Collection 原本的 iterator() method 獨立出來成爲一個新的 java.lang.Iterable 介面，也之所以如此 Enhanced for loop 除了可輪詢陣列型態之外，還可以輪詢所有只要繼承或實作(implement)java.lang.Iterable 介面，這樣的做法也爲「Enhanced for loop」帶來了更多的彈性，讀者若有興趣可以去閱讀 Design Pattrns：Iterator pattern 的相關資訊。

```
public interface Collection<E>
extends Iterable<E>
```

圖七　Collection 的繼承結構(如附檔：圖七.jpg)

現在我們就來看看「Enhanced for loop」輪詢 Collection 的作法，看看 Java Compiler 又幫我們摻雜了那些「調味料」：(如程式五)

==========

程式五：

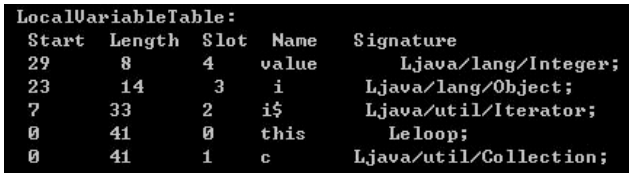
```
public void testForInCollection(Collection c)
{
    for(Object i : c)
    {
        Integer value = (Integer) i;
        System.out.println(value);
    }
}
```

==========

```
public void testForInCollection(java.util.Collection);
  Code:
   Stack=2, Locals=5, Args_size=2
   0:   aload_1
   1:   invokeinterface #9,  1; //InterfaceMethod java/util/Collection.iterator:
()Ljava/util/Iterator;
   6:   astore_2
   7:   aload_2
   8:   invokeinterface #5,  1; //InterfaceMethod java/util/Iterator.hasNext:()Z

   13:  ifeq    40
   16:  aload_2
   17:  invokeinterface #6,  1; //InterfaceMethod java/util/Iterator.next:()Ljav
a/lang/Object;
   22:  astore_3
   23:  aload_3
   24:  checkcast       #7; //class java/lang/Integer
   27:  astore  4
   29:  getstatic       #2; //Field java/lang/System.out:Ljava/io/PrintStream;
   32:  aload   4
   34:  invokevirtual   #8; //Method java/io/PrintStream.println:(Ljava/lang/Obj
ect;)V
   37:  goto    7
   40:  return
```

圖八　testForInCollection 方法的位元碼(如附檔：圖八.jpg)



```
LocalVariableTable:
 Start  Length  Slot  Name   Signature
 29      8       4    value       Ljava/lang/Integer;
 23      14      3     i        Ljava/lang/Object;
 7       33      2    i$       Ljava/util/Iterator;
 0       41      0    this       Leloop;
 0       41      1    c        Ljava/util/Collection;
```

圖九　　testForInCollection 方法的 LocalVariableTable(如附檔：圖九.jpg)

從上圖的位元碼相較於輪詢陣列的位元碼之下，看起來這次的「調味料」似乎沒有添加多少!我們可以從 LocalVariableTable 發現只有「**i$**」變數在編譯時期由 Java Compiler 添加進來，而且從圖八和圖五的位元碼就能發現其實兩者的程式結構上幾乎相差無幾，當然筆者也一併附上摻雜「調味料」之後的程式碼，以供讀者參考：(如程式六)

==========

程式六：

```
public void testForInCollection(Collection c)
{
    for(Iterator i$ = c.iterator ; i$.hasNext() ; )
    {
        Object i = i$.next();
        Integer value = (Integer)i;
        System.out.println(value);
    }
}
```

==========

#中標=「Enhanced for loop」所帶來的好處

簡單而言，「Enhanced for loop」所帶來的好處就是要保護我們廣大的 Java 程式開發人員，那雙在鍵盤上敲敲打打的寶貴雙手，若是再和泛型(Generic)結合一起的話，那更大大地省下許多敲打的功夫，從下列程式七便不難發現程式已經更爲精簡，而且若運用在 2 維陣列上也能省下許多「制式化」的程式寫法，程式看起來也乾淨了許多!

==========

程式七：

```
public void testForInGeneric(Collection<Integer> c)
{
    for(Integer i : c)
        System.out.println(i);
}
```

==========

==========

程式八：

```
public void test2DArrayForIn()
{
    int[][] value = new int[][]{{1,2},{3,4}};

    for(int[] v : value)
        for(int i : v)
```

```
            System.out.println(i);
}
================================================
```

#中標=結語

從本文的剖析之中，相信許多讀者對於「Enhanced for loop」都已經歷了底層的洗禮，當然 JDK5.0 仍然擁有許多的新特色與功能，倘若還有機會筆者將會再針對「老虎」Autoboxing/Unboxing 的底層來剖析，看看還有那些「調味料」等待著我們一一去品嚐，最後也希望本文能帶給各位讀者獲益良多的感受。

#參考資料=

1.The JavaTM Virtual Machine Specification Second Edition
2. Java 5.0 Tiger: A Developer's Notebook

#作者介紹=陳縉紳<chingshenchen@gmail.com>

目前就讀國立台中技術學院多媒體設計研究所。