#大標=程式領域中的「++」
#副標=原來這就是「i++」和「++j」
#引言=在程式設計的領域中，遞增運算子是一種節省程式並兼顧美觀的寫法，本文將透過不同的角度來探討「i++」和「++j」，並詳加地說明處理方式，更確實掌握觀念的建立。
#作者=文/陳縈紳 http://blog.ring.idv.tw
#內文=

小明是一位騎著腳踏車去上班的工讀生，他每天在一間「XX 服飾店」擔任摺襯衫的重複性工作，而那位嚴厲的老闆規定他每天的工作量必須摺好 300 件的襯衫，無疑地他必須去累積計算他所摺好的襯衫件數，這對小明來說唯有將事情做好才能交代給老闆，頓時仍在就讀進修部資管系的會計小姐，腦海裡浮現了一個程式演算法，如程式 1：

==========

程式 1：

```
public class XX 服飾店
{
    public static void main(String arg[])
    {
        int i = 0;
        int count;
        while( i < 300)
        {
            count = i++;
            System.out.println("小明已摺好了"+ count +"件襯衫");
        }
    }
}
```

==========

會計小姐對於上述程式的「i++」深感疑惑，心裡仍然矛盾著爲什麼會印出「小明已摺好了 0 件襯衫」呢？老師在課堂上不是說「i++」等於「i = i + 1」嗎？會計小姐不禁感嘆的說：「好的老師帶你上天堂，不好的老師帶你住…」。

在程式設計的領域中，運算子(operator)不乏出現在我們所撰寫的程式之中，這當中包含算術運算子、邏輯運算子、位元運算子等…，而其中便包括本文所將探討的「遞增運算子」，從上述會計小姐所遭遇到的問題點，我們可以很清楚地了解到她的瓶頸出現在「count = i++」這一行程式，擁有程式經驗的朋友大概都知曉如何修正此行程式，修改的方法便是將其改成「count = ++i」即可，可見這位會計小姐在上課時有所遺漏的地方，我們可以在本文將它做成補足。

本文中並非僅以簡單的敘述來解釋「i++」和「++i」的觀念，而是透過組合語言(Assembly Language)、Java 位元碼(Bytecode)和 Flash 行爲模式(ActionModel)來探討，並一一地以抽絲剝繭的方式來分析作法上的不同之處，當然在開始剖析「i++」和「++i」之前，筆者將再次地介紹「遞增運算子」的觀念，來使本文的內容更臻於完整。

#中標=**介紹「i++」和「++j」的觀念**

==========

程式 2：

```
public class Inc
```

```
{
    public static void main(String arg[])
    {
        int i = 1;
        int j = 1;
        int count;

        count = i++;
        System.out.println(count);

        count = ++j;
        System.out.println(count);
    }
}
```

==========

我們從程式 2 的執行結果可以發現，先在命令列印出變數「count」值的會是「1」接著印出「2」，這是因爲「遞增運算子」和「遞減運算子」撰寫在變數前後的不同之處，以「count = i++」來解釋之，我們可以將它看作「count = i;」和「i = i + 1;」的組合，所以變數「i」的值在遞增之前就已經指派給變數「count」，然後才會執行「i = i + 1;」的敘述來執行遞增的動作，所以命令列印出的變數「count」值必然會是「1」，而另一個「count = ++j」想必讀者能猜中處理的方式了，我們可以將它看作是「j = j + 1;」和「count = j;」的組合，剛好和「count = i++」的處理方式相反，也就是先執行遞增的動作，然後再指派給變數「count」，這就是「i++」和「++j」最主要的相異之處，接下來我們就開始一一地透過實作來印證。

#中標=**從組合語言來欣賞「i++」和「++j」**

在了解了「i++」和「++j」的觀念後，我們接著就先從組合語言來開始探討處理方式，首先我們必須先撰寫一個 C 語言程式，然後將此程式編譯並產生組合語言碼，筆者將以 GCC(GNU Compiler Collection)做爲編譯器來使用，如程式 3 所示：

==========

程式 3：

```
#include <stdio.h>
main()
{
        int i = 1;
        int j = 1;
        int count;

        count = i++;
        printf("%d\n",count);

        count = ++j;
        printf("%d\n",count);
}
```

==========

```
 1         .file   "plus.c"
 2         .section        .rodata
 3 .LC0:
 4         .string "%d\n"
 5         .text
 6 .globl main
 7         .type   main, @function
 8 main:
 9         pushl   %ebp
10         movl    %esp, %ebp
11         subl    $24, %esp
12         andl    $-16, %esp
13         movl    $0, %eax
14         addl    $15, %eax
15         addl    $15, %eax
16         shrl    $4, %eax
17         sall    $4, %eax
18         subl    %eax, %esp
19         movl    $1, -4(%ebp)
20         movl    $1, -8(%ebp)
21         movl    -4(%ebp), %edx
22         leal    -4(%ebp), %eax
23         incl    (%eax)
24         movl    %edx, -12(%ebp)
25         subl    $8, %esp
26         pushl   -12(%ebp)
27         pushl   $.LC0
28         call    printf
29         addl    $16, %esp
30         leal    -8(%ebp), %eax
31         incl    (%eax)
32         movl    -8(%ebp), %eax
33         movl    %eax, -12(%ebp)
34         subl    $8, %esp
35         pushl   -12(%ebp)
36         pushl   $.LC0
37         call    printf
38         addl    $16, %esp
39         leave
40         ret
41         .size   main, .-main
42         .section        .note.GNU-stack,"",@progbits
43         .ident  "GCC: (GNU) 3.4.2 20041017 (Red Hat 3.4.2-6.fc3)"
```

圖 1 程式 3 組合語言碼

表 1 程式 3 的部份組合語言碼說明

組合語言碼	說明
19# movl $1, -4(%ebp)	相當於上述程式的「int i = 1;」，也就是將整數「1」搬移到「-4(%ebp)」的記憶體位址，由此可知「-4(%ebp)」的記憶體位址對應到變數「i」。
20# movl $1, -8(%ebp)	相當於上述程式的「int j = 1;」，也就是將整數「1」搬移到「-8(%ebp)」的記憶體位址，由此可知「-8(%ebp)」的記憶體位址對應到變數「j」。
21# movl -4(%ebp), %edx	將變數「i」的值搬移到「edx 暫存器」。
22# leal -4(%ebp), %eax	把變數「i」的記憶體位址搬移到「eax 暫存器」。
23# incl (%eax)	利用間接定址的方式將變數「i」的值加「1」。
24# movl %edx, -12(%ebp)	將「edx 暫存器」的值搬移到「-12(%ebp)」的記憶體位址，所以「-12(%ebp)」的記憶體位址是對應到變數「count」。
26# pushl -12(%ebp)	將變數「count」的值推入堆疊。
27# pushl $.LC0	將「.LC0」標記的記憶體位址推入堆疊。
28# call printf	呼叫 C 語言的 printf 函式。
30# leal -8(%ebp), %eax	把變數「j」的記憶體位址搬移到「eax 暫存器」。
31# incl (%eax)	利用間接定址的方式將變數「j」的值加「1」。
32# movl -8(%ebp), %eax	把變數「j」的值搬移到「eax 暫存器」。
33# movl %eax, -12(%ebp)	將「eax 暫存器」的值搬移到變數「count」。

35# pushl -12(%ebp)	將變數「count」的值推入堆疊。
36# pushl $.LC0	將「.LC0」標記的記憶體位址推入堆疊。
37# call printf	呼叫 C 語言的 printf 函式

從表 1 我們可以清楚地得知，圖 1 組合語言碼在處理「count = i++;」的關鍵點在於「21# movl -4(%ebp), %edx」和「24# movl %edx, -12(%ebp)」這兩行敘述，這兩行敘述表徵著對應「count = i」，而「22# leal -4(%ebp), %eax」和「23# incl (%eax)」這兩行敘述也對應著「i = i + 1;」，最後再透過 C 語言的 prinf 函式來印出變數「i」的值，讀者可以對照著表 1 來細細地推敲。接著我們繼續來欣賞「count = ++j;」的處理方式，從「30# leal -8(%ebp), %eax」和「31# incl (%eax)」這兩行敘述便可得知對應著「j = j + 1;」，而「32# movl -8(%ebp), %eax」和「33# movl %eax, -12(%ebp)」當然也就是對應「count = j」囉!會計小姐在欣賞組合語言的處理方式之後，總算有點更深入的體會了，半工半讀的她當然更不會放過 Java 位元碼的處理方式囉!會計小姐開始認真了起來…。

#中標=**從 Java 位元碼來欣賞「i++」和「++j」**

此時，會計小姐趁著中午的休息時間，開始著手這一段落的閱讀。在欣賞完組合語言的「i++」和「++j」後，我們將從 Java 位元碼的角度來接續著剖析「i++」和「++j」，順便可以瞧瞧 Java 位元碼和組合語言兩者之間的處理方式，其實兩者處理的方式大同小異，但重要的是那觀念的建立，這才是取其精要的不可或缺之法則，如程式 4：

==========

程式 4：

```
public class Test
{
    public static void main(String arg[])
    {
        int i = 1;
        int j = 1;
        int count;

        count = i++;
        System.out.println(count);

        count = ++j;
        System.out.println(count);
    }
}
```

==========

```
public static void main(java.lang.String[]);
  Code:
   Stack=2, Locals=4, Args_size=1
   0:   iconst_1
   1:   istore_1
   2:   iconst_1
   3:   istore_2
   4:   iload_1
   5:   iinc    1, 1
   8:   istore_3
   9:   getstatic       #2; //Field java/lang/System.out:Ljava/io/PrintStream;
   12:  iload_3
   13:  invokevirtual   #3; //Method java/io/PrintStream.println:(I)V
   16:  iinc    2, 1
   19:  iload_2
   20:  istore_3
   21:  getstatic       #2; //Field java/lang/System.out:Ljava/io/PrintStream;
   24:  iload_3
   25:  invokevirtual   #3; //Method java/io/PrintStream.println:(I)V
   28:  return
```

圖 2 Test 類別的結構

表 2 Java 位元碼

Opcode	用途	說明
0# iconst_1	推入一個 int 常數值「1」至 Operand Stack。	如同上述程式「int i = 1;」敘述，變數「i」即對應至 local variable[1]。
1# istore_1	儲存 int 整數至 local variable[1]。	
2# iconst_1	推入一個 int 常數值「1」至 Operand Stack。	如同上述程式「int j = 1;」敘述，變數「j」即對應至 local variable[2]。
3# istore_2	儲存 int 整數至 local variable[2]。	
4# iload_1	取出 local variable[1]的 int 值，並推入 Operand Stack。	將目前 local variable[1]的整數值「1」推入至 Operand Stack。
5# iinc 1,1	將 local variable[1]的 int 值加 1。	
8# istore_3	儲存 int 整數至 local variable[3]。	將「4# iload_1」所推入至 Operand Stack 的整數值，儲存至 local variable[3]，也就是儲存至變數「count」。
9# getstatic	取得類別(靜態)欄位。	對應上述程式「System.out.println(count);」敘述，所以在 Console 印出「1」。
12# iload_3	取出 local variable[3]的 int 值，並推入 Operand Stack。	
13# invokevirtual	呼叫實體(instance)方法。	
16# iinc 2,1	將 local variable[2]的 int 值加 1。	將目前 local variable[2]的整數值加「1」。
19# iload_2	取出 local variable[2]的 int 值，並推入 Operand Stack。	將 local variable[2]的整數值「2」推入至 Operand Stack。
20# istore_3	儲存 int 整數至 local variable[3]。	將「19# iload_2」所推入至 Operand Stack 的整數值，儲存至 local variable[3]。
21# getstatic	取得類別(靜態)欄位。	對應上述程式「System.out.println(count);」敘述，所以在 Console 印出「2」。
24# iload_3	取出 local variable[3]的 int 值，並推入 Operand Stack。	
25# invokevirtual	呼叫實體(instance)方法。	

從表 2 Java 位元碼的角度來剖析後，我們可以了解到 Java 處理「count = i++;」會將「4# iload_1」、「8# istore_3」對應到「count = i」，而「5# iinc 1,1」便對應著「i = i + 1;」，會計小姐此時感覺和組合語言比較起來更容易理解多了，當然她更進一步地追問說，那「16# iinc 2,1」對應著「j = j + 1;」而「19# iload_2」和「20# istore_3」也對應著「count = j;」囉!嘻…果然沒錯，而且也清楚地了解到重點在於，「count = i++;」是先處理「iload」再進行「iinc」，而「count = ++j;」則剛好相反，閱讀到這裡終於將會計小姐的任督二脈給打通了，最後還有更精彩的 Flash 行爲模式來供品嚐，當然更不可以錯過囉!

#中標=**從 Flash 行爲模式來欣賞「i++」和「++j」**

在 Flash 2 的時代，那時還尚未有行爲指令(Action)的互動運用，直到 Flash 3 加入了 11 個簡單的指令，如「play()」、「stop()」、「nextFrame()」等…，正式宣告行爲指令的誕生，但這並不足以造成 Flash 大放異彩的主要原因，Flash 真正光芒四射的時代在於 Flash 4 的革新降臨，Flash 4 大大地增加了許多行爲指令，如算術運算子、邏輯運算子、字串操作和堆疊操作等…，因爲它併入了堆疊機器(Stack Machine)和程式計數器(Program Counter)來直譯與執行 Flash 4 的行爲指令，這才使得愈來愈多人投入 Flash 的世界來遨遊，直到現在 Flash 已經推出到第八版了，市面上的書籍也不勝枚舉，而且更是許多學校教授多媒體課程的工具首選，所以本文也將此納入來剖析一翻，如程式 5。

==========

程式 5：

```
var i:Number = 1;
var j:Number = 1;
var count:Number;

count = i++;
trace(count);

count = ++j;
trace(count);
```

==========

```
00000000:  46 57 53 08 8d 00 00 00 78 00 05 5f 00 00 0f a0
00000010:  00 00 0c 01 00 44 11 00 00 00 00 43 02 ff ff ff
00000020:  3f 03 63 00 00 00 88 0c 00 03 00 69 00 6a 00 63
00000030:  6f 75 6e 74 00 96 07 00 08 00 07 01 00 00 00 3c
00000040:  96 07 00 08 01 07 01 00 00 00 3c 96 02 00 08 02
00000050:  41 96 04 00 08 02 08 00 1c 96 04 00 08 00 08 00
00000060:  1c 50 1d 1d 96 02 00 08 02 1c 26 96 06 00 08 02
00000070:  08 01 08 01 1c 50 87 01 00 00 1d 96 02 00 04 00
00000080:  1d 96 02 00 08 02 1c 26 00 40 00 00 00
```

圖 3 Flash 位元碼

表 3 Flash 的部份位元碼說明

bytecode	Actions	說明
88 0c 00 03 00 69 00 6a 00 63 6f 75 6e 74 00	ActionConstantPool “i” , “j” , “count”	建立一個常數池，此常數池包含三個字串常數，它們分別爲「i」、「j」、「count」。
96 07 00 08 00 07 01 00 00 00	ActionPush “i”, 1	將常數「i」、「1」推入堆疊。
3c	ActionDefineLocal	從堆疊取出「1」、「i」，並將「i」定義爲區域變數，且將它的值設爲「1」。
96 07 00 08 01 07 01 00 00 00	ActionPush “j”, 1	將常數「j」、「1」推入堆疊。
3c	ActionDefineLocal	從堆疊取出「1」、「j」，並將「j」定義爲區域變數，且將它的值設爲「1」。
96 02 00 08 02	ActionPush “count”	將常數「count」推入堆疊。
41	ActionDefineLocal2	從堆疊取出「count」，並定義爲區域變數但未給予初始值。
96 04 00 08 02 08 00	ActionPush “count” , ”i”	將常數「count」、「i」推入堆疊。
1c	ActionGetVariable	從堆疊取出「i」，並將「i」的變數值推入堆疊。
96 04 00 08 00 08 00	ActionPush “i” , “i”	將常數「i」、「i」推入堆疊。 *目前堆疊➔「count」、「1」、「i」、「i」。
1c	ActionGetVariable	從堆疊取出「i」，並將「i」的變數值推入堆疊。 *目前堆疊➔「count」、「1」、「i」、「1」。

50	ActionIncrement	從堆疊取出一個值，並將它加「1」。 *目前堆疊➔「count」、「1」、「i」、「2」。
1d	ActionSetVariable	從堆疊取出「2」、「i」，並將「2」設定至「i」變數。 *目前堆疊➔「count」、「1」。
1d	ActionSetVariable	從堆疊取出「1」、「count」，並將「1」設定至「count」變數。
96 02 00 08 02	ActionPush “count”	將常數「count」推入堆疊。
1c	ActionGetVariable	從堆疊取出「count」，並將「count」的變數值推入堆疊。 *目前堆疊➔「1」。
26	ActionTrace	從堆疊取出一個值並顯示在 Output Window，所以將從 Output Window 印出「1」。
96 06 00 08 02 08 01 08 01	ActionPush “count” , ”j” , ”j”	將常數「count」、「j」、「j」推入堆疊。
1c	ActionGetVariable	從堆疊取出「j」，並將「j」的變數值推入堆疊。 *目前堆疊➔「count」、「j」、「1」。
50	ActionIncrement	從堆疊取出一個值，並將它加「1」。 *目前堆疊➔「count」、「j」、「2」。
87 01 00 00	ActionStoreRegister 0	從堆疊讀取一個物件(非取出)，並儲存在 Register 0。 *目前堆疊➔「count」、「j」、「2」。 *目前 Register 0 ➔ 「2」。
1d	ActionSetVariable	從堆疊取出「2」、「j」，並將「2」設定至「j」變數。 *目前堆疊➔「count」。
96 02 00 04 00	ActionPush Register 0	將 Register 0 的值推入堆疊。 *目前堆疊➔「count」、「2」。
1d	ActionSetVariable	從堆疊取出「2」、「count」，並將「2」設定至「count」變數。
96 02 00 08 02	ActionPush “j”	將常數「j」推入堆疊。
1c	ActionGetVariable	從堆疊取出「j」，並將「j」的變數值推入堆疊。 *目前堆疊➔「2」。
26	ActionTrace	從堆疊取出一個值並顯示在 Output Window，所以將從 Output Window 印出「2」。
00	ActionEndFlag	Action 結束。

程式 5 Flash ActionScript 相較於 Java 或 C 語言程式簡單了許多，雖然表 3 Flash 部份位元碼顯得較爲冗長，但卻是這三者之中最容易理解與推敲的，讀者可以對照著 Flash 位元碼一步一步地來推演出結果，這不但有助於更進一步地來了解 Flash ActionModel，對於觀念的建立自然是不在話下。

#中標=**結語**

本文「原來這就是 i++和++j」到此就告一段落，從組合語言、Java 位元碼到 Flash 行爲模式這三種

角度的切入點，相信讀者們對於「i++」和「++j」的來龍去脈有了扎根的基礎，那位會計小姐也終於慶幸地不用住「套房」了，最後當然更希望讀者能有獲益良多的感受，倘若本文有任何謬誤，希望請不吝地賜教，若能指正不勝感激。

#參考資料=

1.The JavaTM Virtual Machine Specification Second Edition
2.Macromedia Flash (SWF) File Format Specification Version 7

#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。