#大標=JDK 5.0 Tiger Autoboxing/Unboxing
#副標=徹底剖析 JDK 5.0 Autoboxing/Unboxing
#引言=經過上一期所品嚐的「Enhanced for loop」功能之後，本文這次將會針對 JDK 5.0 所新增的「Autoboxing/Unboxing」功能來加以剖析，看看這隻「老虎」又在底層幫我們加了那些「調味料」，相信經由本文的介紹能帶給各位朋友獲益良多的感受。
#作者=文/陳鑅紳 http://blog.ring.idv.tw
#內文=

經過前二期所介紹的「Varargs」和「Enhanced for loop」之後，我們可以看到從 JDK 5.0 所新增加的功能裡，擁有不少功能仰賴著 Java Compiler 在編譯時期幫我們加上「調味料」來處理，雖然更簡單易學易用，不過終究仍需了解底層的實作細節，才能徹底掌握程式本身的特性，尤其是本文所即將剖析的「Autoboxing/Unboxing」功能，常會使許多初學 Java 的朋友們產生「知其然而不知其所以然」的現象，正如俗話所說的「水能載舟亦能覆舟」，「Autoboxing/Unboxing」的確可以幫我們減少許多在敲打鍵盤上所需要的寶貴光陰，但若是沒能徹底掌握它，彷彿也爲自己埋下了一顆不定時的炸彈，現在我們就來了解「Autoboxing/Unboxing」究竟是何方神聖。

在 Java 程式語言裡有分成兩種型別，一種是基本資料型別(Primitive Data Types)，另一種則是物件參考型別(Object Reference Types)，而本文的「Autoboxing/Unboxing」功能便是將針對基本資料型別(表 1)來做轉換處理，所以當我們想將這八種基本資料型別轉換成物件來操作時，這時候我們就必須將它「裝箱」(Boxing)，何謂「裝箱」呢？簡單的說法就是將 Java 所提供對應的 Wrapper Class 來加以包裝，當然既然有「裝箱」必然也會有「拆箱」(unboxing)的動作，那何謂「拆箱」呢？拆箱所指的就是呼叫其對應的 Wrapper Class 所各自擁有的「xxxxValue()」方法，諸如：intValue()、floatValue()、charValue()…，所以「Autoboxing/Unboxing」簡單地來說，也就是 Java Compiler 在編譯時期幫我們做這層的轉換處理。

表 1 基本資料型別

整數型別(Integer type)	byte、short、int、long
浮點數型別(Floating point type)	float、double
字元型別(Textual type)	char
邏輯型別(Logical type)	boolean

#中標=**剖析「Autoboxing/Unboxing」**

在開始剖析「Autoboxing/Unboxing」的功能之前，筆者先來介紹在這隻「老虎」出閘前的「裝箱」和「拆箱」方式，如下列程式 1：

==========

程式 1：

```
public void BoxingUnboxing()
{
    Integer i1 = new Integer(5); //Boxing
    int i2 = i1.intValue(); //Unboxing
}
```

==========

從上述的程式中可以看得出，當我們要將基本資料型別的整數轉換成物件，通常會建立一個新的 Integer 物件來將此整數加以包裝，而當我們要將它反轉成基本資料型別的時候，我們也必須呼叫「intValue()」方法來取得基本資料型別的整數值，總之在 JDK 5.0 尚未問世前，上述程式的轉換動作是不可或缺的，緊接著我們就來看看「老虎」出閘後有什麼樣的改觀，如程式 2：

==========

程式 2：

```
public void AutoboxingUnboxing()
{
    Integer i1 = 5; //Autoboxing
    int i2 = i1; //Unboxing
}
```

==========

```
public void AutoboxingUnboxing();
  Code:
   Stack=1, Locals=3, Args_size=1
   0:   iconst_5
   1:   invokestatic    #2; //Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
   4:   astore_1
   5:   aload_1
   6:   invokevirtual   #3; //Method java/lang/Integer.intValue:()I
   9:   istore_2
   10:  return
```

圖 1 AutoboxingUnboxing 方法的位元碼

```
LocalVariableTable:
 Start  Length  Slot  Name   Signature
 0      11      0     this      LBoxing;
 5      6       1     i1        Ljava/lang/Integer;
 10      1      2     i2         I
```

圖 2 AutoboxingUnboxing 方法的 LocalVariableTable

表 2 AutoboxingUnboxing 方法的位元碼

位元碼	用途	說明
0# iconst_5	推入一個 int 常數「5」至 Operand Stack。	
1# invokestatic	呼叫類別(static)方法。	「呼叫 Integer.valueOf() 類別方法，並將儲存在 Operand Stack 的整數 5 當參數傳遞進去」
4# astore_1	儲存 object reference 至 local variable[1]。	「將參考到 Integer 物件的 object reference 儲存至區域變數 **i1**」
5# aload_1	從 local variable[1] 載入 object reference 至 Operand Stack。	
6# invokevirtual	呼叫實體(instance)方法。	「呼叫 intValue() 實體方法」
9# istore_2	儲存 int 整數至 local variable[2]。	「將此整數儲存至區域變數 **i2**」
10# return	回傳 void。	

上述就是整個 AutoboxingUnboxing 方法的位元碼，我們從上述的位元碼便不難發現，JDK 5.0 所提供的「Autoboxing」裝箱處理和我們的程式 1 有所出入，因爲這隻「老虎」所採用的「裝箱」方式是呼叫「valueOf()」方法來達成的，而不是使用「new」來產生物件並加以「裝箱」，至於「拆箱」的方法就如同程式 1 的方式相同，下列程式 3 就是經過 Java Compiler 處理過後的原始程式：

==========

程式 3：

```
public void SourceCode()
{
    Integer i1 = Integer.valueOf(5); //Boxing
    int i2 = i1.intValue(); //Unboxing
}
```

==========

#中標=**剖析「裝箱」後的遞增與遞減**

在剖析過「Autoboxing/Unboxing」的處理機制之後，接著我們就來看看「裝箱」後是如何處理遞增與遞減，一般我們會使用「++」或「--」的方式來處理，但是透過「裝箱」之後，那底層又是如何處理的呢？現在我們就來剖析一下：

==========

程式 4：

```
public void add()
{
    Integer i = 5;
    ++i;
    System.out.println(i);
}
```

==========

```
public void add();
  Code:
   Stack=2, Locals=2, Args_size=1
   0:   iconst_5
   1:   invokestatic    #2; //Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
   4:   astore_1
   5:   aload_1
   6:   invokevirtual   #3; //Method java/lang/Integer.intValue:()I
   9:   iconst_1
   10:  iadd
   11:  invokestatic    #2; //Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
   14:  astore_1
   15:  getstatic       #4; //Field java/lang/System.out:Ljava/io/PrintStream;
   18:  aload_1
   19:  invokevirtual   #5; //Method java/io/PrintStream.println:(Ljava/lang/Object;)V
   22:  return
```

圖 3 add 方法的位元碼

表 3 add 方法的位元碼

位元碼	用途	說明
0# iconst_5	推入一個 int 常數「5」至 Operand Stack。	
1# invokestatic	呼叫類別(static)方法。	「呼叫 Integer.valueOf() 類別方法，並將儲存在 Operand Stack 的整數 5 當參數傳遞進去」
4# astore_1	儲存 object reference 至 local variable[1]。	「將參考到 Integer 物件的 object reference 儲存至區域變數 **i**」
5# aload_1	從 local variable[1]載入 object reference 至 Operand Stack。	
6# invokevirtual	呼叫實體(instance)方法。	「呼叫 intValue() 實體方法，並將回傳值推入至 Operand Stack」

9# iconst_1	推入一個 int 常數「1」至 Operand Stack。	「目前 Operand Stack 將有兩個常數，一個爲 intValue()的回傳值：**5**，另一個爲常數：**1**」
10# iadd	將兩個整數相加。	「將位於 Operand Stack 前兩位的數值取出相加，並將結果再儲存於 Operand Stack」
11# invokestatic	呼叫類別(static)方法。	「呼叫 Integer.valueOf() 類別方法，並將儲存在 Operand Stack 的整數 **6** 當參數傳遞進去」
14# astore_1	儲存 object reference 至 local variable[1]。	
15# getstatic	取得類別(靜態)欄位。	「對應程式：System.out.println(i);，將 **i** 值印出」
18# aload_1	從 local variable[1]載入 object reference 至 Operand Stack。	
19# invokevirtual	呼叫實體(instance)方法。	
22# return	回傳 void。	

從上述所剖析的位元碼中，我們可以發現 Java Compiler 會將我們程式 4 的原始程式，編譯成如程式 5 的樣式，也就是說，上述程式在處理「Autoboxing/Unboxing」的遞增時，都必須先將已「裝箱」的物件將以「拆箱」成基本資料型別，然後經由加減運算處理後，再加以「裝箱」成物件參考型別，幸好這些動作在「老虎」出閘後終究獲得解決，一方面提高程式的可閱讀性，另一方面也減少了許多 Programmer Coding 的時間。

==========

程式 5：

```
public void add()
{
    Integer i = Integer.valueOf(5);
    i = Integer.valueOf(i.intValue()+1);
    System.out.println(i);
}
```

==========

#中標=**流程控制與斷言(Assertions)的「拆箱」處理**

了解基本的「Autoboxing/Unboxing」處理之後，接著我們就來測試一些流程控制與斷言的範例，看看「Autoboxing/Unboxing」是否也能如預期地，在這些述敘句中正確地執行成功。(如程式 6)

==========

程式 6：

```
public void testFlowControls()
{
    Boolean B = true;
    Integer I = 100;
    int i = 99;

    while(I > i)
    {
        if(B)
```

```
        {
            switch(I){
                case 100:
                    System.out.println("switch");
            }
            System.out.println("if-else");
            i++;
        }
        System.out.println("while-loop");
    }

    assert !B : "assert";
}
```

```
 13:  istore_3
 14:  aload_2
 15:  invokevirtual   #3; //Method java/lang/Integer.intValue:()I
 18:  iload_3
 19:  if_icmple       82
 22:  aload_1
 23:  invokevirtual   #16; //Method java/lang/Boolean.booleanValue:()Z
 26:  ifeq    71
 29:  aload_2
 30:  invokevirtual   #3; //Method java/lang/Integer.intValue:()I
 33:  lookupswitch{ //1
              100: 52;
              default: 60 }
 52:  getstatic       #4; //Field java/lang/System.out:Ljava/io/PrintStream;
 55:  ldc     #17; //String switch
 57:  invokevirtual   #7; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
 60:  getstatic       #4; //Field java/lang/System.out:Ljava/io/PrintStream;
 63:  ldc     #18; //String if-else
 65:  invokevirtual   #7; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
 68:  iinc    3, 1
 71:  getstatic       #4; //Field java/lang/System.out:Ljava/io/PrintStream;
 74:  ldc     #19; //String while-loop
 76:  invokevirtual   #7; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
 79:  goto    14
 82:  getstatic       #20; //Field $assertionsDisabled:Z
 85:  ifne    105
 88:  aload_1
 89:  invokevirtual   #16; //Method java/lang/Boolean.booleanValue:()Z
 92:  ifeq    105
```

圖 4 testFlowControls 方法的部份位元碼

從圖 4 的位元碼所示，我們可以很清楚地知道這些已經被「裝箱」後的物件，在遇到 while 迴圈、if/else 判斷和 switch 判斷，甚至是斷言，都會經由 Java Compiler 自動地加入「拆箱」的動作，所以此程式將可正確無誤地執行。

閱讀到目前為止，我們大都圍繞在「Autoboxing/Unboxing」底層位元碼的轉換處理方式，接著開始我們將從另一個角度去深入探討，在運用「Autoboxing/Unboxing」時必須注意那些「重要」的細節呢？

#中標=**知其然更要知其所以然**

通常我們要比對兩個數值是否相等時，最簡單的方式就是直接使用「==」比較運算子，但是經過「Autoboxing/Unboxing」的處理之後，我們是否能直接使用「==」比較運算子來比較兩個數值呢？我們現在就來給它剖析一翻!

#小標=使用「==」來比較「Autoboxing」的整數(一)

==========

程式 7：

```
public void isEqual()
{
    Integer i1 = 128;
    Integer i2 = 128;
    if(i1 == i2)
        System.out.println(“equal”);
    else
        System.out.println(“not equal”);
}
```

==========

```
public void isEqual();
  Code:
   Stack=2, Locals=3, Args_size=1
   0:   sipush  128
   3:   invokestatic    #2; //Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
   6:   astore_1
   7:   sipush  128
   10:  invokestatic    #2; //Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
   13:  astore_2
   14:  aload_1
   15:  aload_2
   16:  if_acmpne       30
   19:  getstatic       #4; //Field java/lang/System.out:Ljava/io/PrintStream;
   22:  ldc     #6; //String equal
   24:  invokevirtual   #7; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
   27:  goto    38
   30:  getstatic       #4; //Field java/lang/System.out:Ljava/io/PrintStream;
   33:  ldc     #8; //String not equal
   35:  invokevirtual   #7; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
   38:  return
```

圖 5 isEqual 方法的位元碼

表 4 isEqual 方法的部份位元碼

位元碼	用途	說明
0# sipush 128	推入一個 short (-32768~32767)整數值至 Operand Stack。	「將 128 整數值推入至 Operand Stack」
16# if_acmpne 30	如果兩個 object reference 所參考的並非同一個物件則跳躍	「對應 if(i1 == i2) 程式，如果 i1 和 i2 參考的並非同一個物件則跳至 30#執行」

從圖 5 的位元碼中，我們可以知道從 0#至 13#的位元碼所對應的就是「裝箱」的動作，但是這裡並沒有任何的「拆箱」動作，因爲「16# if_acmpne」的位元碼是用來比較物件參考型別，而不是用來比較基本資料型別，顯然地，當我們執行此程式時將會印出「not equal」，所以由此我們可以得知，如果我們要使用「==」比較運算子來比較兩個已「裝箱」的整數值，我們就必須想辦法使 Java Compiler 幫我們自動轉換成基本資料型別，或是採用字串的比較方法「equal()」，這樣才能正確地取得我們所要的結果。

#小標=使用「==」來比較「Autoboxing」的整數(二)

==========

程式 8：

```
public void isEqualInt()
```

```
{
    Integer i1 = 128;
    Integer i2 = 128;
    if((int)i1 == i2)
        System.out.println(“equal”);
    else
        System.out.println(“not equal”);
}
```

==========

```
public void isEqualInt();
  Code:
   Stack=2, Locals=3, Args_size=1
   0:   sipush  128
   3:   invokestatic    #2; //Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
   6:   astore_1
   7:   sipush  128
   10:  invokestatic    #2; //Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
   13:  astore_2
   14:  aload_1
   15:  invokevirtual   #3; //Method java/lang/Integer.intValue:()I
   18:  aload_2
   19:  invokevirtual   #3; //Method java/lang/Integer.intValue:()I
   22:  if_icmpne       36
   25:  getstatic       #4; //Field java/lang/System.out:Ljava/io/PrintStream;
   28:  ldc     #7; //String equal
   30:  invokevirtual   #8; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
   33:  goto    44
   36:  getstatic       #4; //Field java/lang/System.out:Ljava/io/PrintStream;
   39:  ldc     #9; //String not equal
   41:  invokevirtual   #8; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
   44:  return
```

圖 6 isEqualInt 方法的位元碼

表 5 isEqualInt 方法的部份位元碼

位元碼	用途	說明
22# if_icmpne 36	如果兩個 int 整數值不相等則跳躍。	「對應 if((int)i1 == i2) 程式，如果 i1 和 i2 所拆箱後的整數值不相等則跳至 36#執行」

從圖 6 的位元碼裡，我們可以發覺到從 0#至 19#的位元碼所對應的就是「裝箱」和「拆箱」的動作，然後透過「22# if_icmpne」的位元碼來比較基本資料型別的整數值，當然執行此程式時將會印出「equal」，所以從程式 7 和程式 8 來看，我們只將「if(i1 == i2)」改成「if((int)i1 == i2)」就可以使 Java Compiler 幫我們自動轉換成基本資料型別，當然也就順利地取得我們所要的結果囉!

#小標=**使用「==」來比較「Autoboxing」的整數(三)**

看過前兩個「==」比較運算子的範例之後，我們再來看看程式 9 這個特殊的例子，詐看之下程式 9 和程式 7 相比似乎沒有太大的變化，我們只是將整數值的部份從「128」改成「127」而已，但是執行之後的結果，卻出乎意料之外地印出「equal」？究竟爲什麼改個數值就造成這天攘之別的結果呢？難道是 Java 的 Bug 嗎？底下我們就來探討這個例子。

==========

程式 9：

```
public void isEqual2()
{
    Integer i1 = 127;
    Integer i2 = 127;
```

```
    if(i1 == i2)
        System.out.println(“equal”);
    else
        System.out.println(“not equal”);
}
```

==========

```
public void isEqual2();
  Code:
   Stack=2, Locals=3, Args_size=1
   0:   bipush  127
   2:   invokestatic    #2; //Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
   5:   astore_1
   6:   bipush  127
   8:   invokestatic    #2; //Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
   11:  astore_2
   12:  aload_1
   13:  aload_2
   14:  if_acmpne       28
   17:  getstatic       #4; //Field java/lang/System.out:Ljava/io/PrintStream;
   20:  ldc     #6; //String equal
   22:  invokevirtual   #7; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
   25:  goto    36
   28:  getstatic       #4; //Field java/lang/System.out:Ljava/io/PrintStream;
   31:  ldc     #8; //String not equal
   33:  invokevirtual   #7; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
   36:  return
```

圖 7 isEqual2 方法的位元碼

表 6 isEqual2 方法的位元碼

位元碼	用途	說明
0# bipush 127	推入一個 byte (-128~127)整數值至 Operand Stack。	「將 127 整數值推入至 Operand Stack」

從圖 7 的位元碼和圖 5 的位元碼之中，我們可以發現最大的相異處在於，128 整數使用「sipush」而 127 整數使用「bipush」，這兩者的格式差別於「sipush」採用兩個「byte」來表示「-32768~32767」的整數範圍，而「bipush」則是採用一個「byte」來表示「-128~127」的整數範圍，但究竟爲何這兩者的差異就造成「i1 == i2」的結果呢？根據 Java Language Specification 的說法，這是因爲 JVM 會將「裝箱」在「-128~127」範圍的整數、char (\u0000~\u007f)和 boolean 來作爲快取，以減少記憶體的使用來增加效率，所以在程式 10 的範例中都將印出「equal」，也就代表著它們都參考到同一個物件，這在撰寫程式時或是想考取 Java 認證的朋友們都必須注意到的細節。

==========

程式 10：

```
public void isCache()
{
    Byte b1 = -128;
    Byte b2 = -128;
    System.out.println((b1 == b2)?"equal":"not equal");

    Short s1 = 127;
    Short s2 = 127;
    System.out.println((s1 == s2)?"equal":"not equal");
```

```
        Integer i1 = 127;
        Integer i2 = 127;
        System.out.println((i1 == i2)?"equal":"not equal");

        Long l1 = -128l;
        Long l2 = -128l;
        System.out.println((l1 == l2)?"equal":"not equal");

        Boolean B1 = false;
        Boolean B2 = false;
        System.out.println((B1 == B2)?"equal":"not equal");

        Character c1 = '\u007f';
        Character c2 = 127;
        System.out.println((c1 == c2)?"equal":"not equal");
}
```

==========

#中標=**Method Overloading 與「Autoboxing/Unboxing」**

最後我們來談談「Autoboxing/Unboxing」與 Method Overloading 之間的影響，Method Overloading 的規則必須是相同的方法名稱和不同的參數型別或個數，主要的目的就是方便我們在撰寫程式時，能重複使用方法的名稱，不過由於「Autoboxing/Unboxing」的出現，試想 Java Compiler 會如何決定呼叫的方法呢？它會先「Autoboxing/Unboxing」再呼叫嗎？我們就來測試一下程式。(如程式 11)

==========

程式 11：

```
public static void doSomething(Integer i)
{
    System.out.println("Integer");
}
public static void doSomething(double i)
{
    System.out.println("double");
}
public static void doSomething(Integer... i)
{
    System.out.println("Integer…");
}
public static void main(String arg[])
{
    int i = 5;
    doSomething(i);
}
```

==========

執行上述的程式之後，我們可以發現 Java Compiler 會先根據 Java 1.4 的方式來決定所要呼叫的方法，當然一方面也是爲了維持著與舊有版本的相容性，接著我們先將「doSomething(double i)」方法註解起來，然後再重新執行一次程式，這次 Java Compiler 便會以「Autoboxing/Unboxing」的方式來決定呼叫「doSomething(Integer i)」方法，所以最後才會考慮 Varargs 的方法，但是有一點必須注意，請避免「程式 12」的寫法，如果同時存在著「int」(基本資料型別)及對應的 Wrapper Class「Integer」(物件參考型別)的這兩個 Varargs 方法，這將會使得 Java Compiler 無法判別該決定呼叫那個一方法，進而產生編繹時期的錯誤!

==========

程式 12：

```
public static void doSomething(int... i)
{
    System.out.println("int…");
}
public static void doSomething(Integer... i)
{
    System.out.println("Integer…");
}
public static void main(String arg[])
{
    int i = 5;
    doSomething(i);
}
```

==========

#中標=結語

本文「Autoboxing/Unboxing」到此就告一段落，相信讀者們對於「Autoboxing/Unboxing」的來龍去脈有了扎根的基礎，當然 JDK5.0 還是擁有許多的新特色與功能，例如：Enumerated、Generic、Annotation…筆者也將會再針對「老虎」的「Enumerated」底層來剖析，窺視一探這隻「老虎」到底還有什麼新奇的東西，筆者也必當深入剖析與讀者們來一同分享，最後本文若有任何謬誤，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. Java 5.0 Tiger: A Developer's Notebook
3. The Java™ Language Specification, Third Edition

#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。