

本章內容：

- * 處理堆疊的指令
- * 區域變數
- * 陣列
- * 物件

第七章 資料操作

本章說明 Java 虛擬機器的「資料操作」指令，這些指令大致上都是用來處理堆疊、區域變數、或物件（含陣列）型式的資料。本章分別將其歸納為四大類，分別是處理堆疊的指令、處理區域變數的指令、處理陣列的指令、以及處理物件的指令。本章以範例來讓讀者更容易瞭解這些指令，相關的細節請查閱第 13 章「指令參考資料」。

處理堆疊的指令

每次行為被呼叫，就會產生自己專用的運算元堆疊，用來傳遞引數給行為，並取得返回值。行為有必要指定此堆疊的大小，在 Jasmin 中，我們可以指定堆疊的大小：

```
.method example()V
.limit stack 20 ; 堆疊的上限是 20 個單 word 的項目
...
.end method
```

我們已經在第二章「快速學習」中看過運算元堆疊了，現在就讓我們來看看這些會動到堆疊的指令。

把常數推入堆疊

有一組指令用來把數字常數或字串常數推入堆疊中（請看表 7-1）。

表 7-1：推入常數的指令

指令名稱	描述
bipush	推入一個位元組的有號數
sipush	推入兩個位元組的有號數
ldc	推入單 word 的常數
ldc_w	推入單 word 的常數（寬的索引）
ldc2_w	推入雙 word 的常數
aconst_null	推入 null
iconst_m1	推入整數常數 -1
iconst_<n>	推入整數常數 0、1、2、3、4、或 5
lconst_<l>	推入長整數 0 或 1
fconst_<f>	推入浮點常數 0.0、1.0、或 2.0
dconst_<d>	推入 double 常數 0.0 或 1.0

比方說，要推入整數 -1，使用敘述如下：

```
iconst_m1 ; 推入整數 -1 到堆疊中
```

推入範圍在 0 到 255 的整數，使用 bipush：

```
bipush 100 ; 推入 100 到堆疊中
```

ldc、ldc_w 與 ldc2_w 就比較特別，他們是用來從常數區中取出值的，比方說：

```
ldc "Hello" ; 推入字串 "Hello"
ldc2_w 10.0 ; 推入 double 10.0
```

表 7-2 列出此三指令會用到的常數型態。

表 7-2：常數區型態與 ldc、ldc_w、ldc2_w

常數型態	指令
CONSTANT_Integer	ldc、ldc_w
CONSTANT_Float	ldc、ldc_w
CONSTANT_String	ldc、ldc_w
CONSTANT_Long	ldc2_w
CONSTANT_Double	ldc2_w

ldc_w 以及 ldc2_w 指令後面接著「_w」，表示它們是「寬」指令。寬指令可以定址到常數區 65535 個項目中的任一個項目，而它們的非寬指令版本只能定址前 255 個項目。（ldc_w 伴隨著 2 個位元組的索引值一起被存放在 bytecode 中，而 ldc 只有伴隨著 1 個位元組的索引值）。Jasmin 的使用者不用擔心該用哪一個指令，因為 Jasmin 組譯器會自動幫你挑選合適的指令。

使用 ldc 來推入字串，這會使得字串被留置起來，你可以用 String.intern() 行為來取得此字串。請看第 4 章的「常數區的解析」一節中更詳細的描述。

堆疊的操作

如果堆疊中已經放了某些項目，你希望改變這些值，JVM 提供了一些指令讓你達到此目的（請看表 7-3）。

表 7-3：堆疊的操作指令

指令名稱	描述
nop	什麼都不做
pop	丟掉堆疊頂端的一個 word
pop2	丟掉堆疊頂端的兩個 word
dup	複製堆疊頂端的一個 word

表 7-3：堆疊的操作指令（續）

指令名稱	描述
dup2	複製堆疊頂端的兩個 word
dup_x1	複製堆疊頂端第 1 個 word，將其插入第 2、3 個 word 之間
dup2_x1	複製堆疊頂端第 1、2 個 word，將其插入第 3、4 個 word 之間
dup_x2	複製堆疊頂端第 1 個 word，將其插入第 3、4 個 word 之間
dup2_x2	複製堆疊頂端第 1、2 個 word，將其插入第 4、5 個 word 之間
swap	交換堆疊頂端的兩個 word

比方說，下面的程式用來交換堆疊頂端的兩個項目：

```
bipush 10
bipush 20
; 堆疊現在包含 10,20

swap
; 堆疊現在包含 20,10

pop
; 堆疊現在包含 20

dup
; 堆疊現在包含 20,20

pop2
; 堆疊現在是空的
```

請注意 double 或 long 型態的資料在堆疊內需要佔用兩個項目（譯者註：亦即兩個 word），如果你要取出 double 或 long 型態的資料，你必須用 pop2，不可以用兩個 pop，因為此二項目必須被同時取出。

區域變數

每次行為引用都有自己的區域變數，區域變數用來存放部份的運算結果和狀態資訊。當一個行為引用結束之後，其區域變數會被毀棄。

行為必須宣告它們所使用的區域變數之個數，在 Jasmin 中，這要透過「`.limit`」假指令來達到，用法如下：

```
.method example()V
    .limit locals 100 ; 最多可以使用 100 個區域變數
.end method
```

每個區域變數都有自己的編號，由 0 開始遞增。對非靜態行為來說，第 0 號區域變數一定是 `this`（採用此行為的物件）。每個行為的區域變數最多可有 65535 個。

你可以在區域變數上做兩件事，你可以把區域變數的值放進堆疊，或把堆疊的值放進區域變數。

把區域變數的值放進堆疊

表 7-4 所列出的指令用來把區域變數的值放進堆疊。

表 7-4：把區域變數的值放進堆疊

指令名稱	描述
<code>iload</code>	從區域變數中取出整數
<code>iload_<n></code>	從第 <n> 號區域變數中取出整數
<code>lload</code>	從區域變數中取出長整數
<code>lload_<n></code>	從第 <n> 號和第 <n+1> 號區域變數中取出長整數
<code>fload</code>	從區域變數中取出浮點數
<code>fload_<n></code>	從第 <n> 號區域變數中取出浮點數
<code>dload</code>	從區域變數中取出 double 數
<code>dload_<n></code>	從第 <n> 號和第 <n+1> 號區域變數中取出 double 數
<code>aload</code>	從區域變數中取出物件指引
<code>aload_<n></code>	從第 <n> 號區域變數中取出物件指引

把堆疊的值放進區域變數

表 7-5 所列出的指令用來把堆疊的值放進區域變數。

表 7-5：把堆疊的值放進區域變數

指令名稱	描述
<code>istore</code>	把整數放進區域變數中
<code>istore_<n></code>	把整數放進第 <n> 號區域變數中
<code>lstore</code>	把長整數放進區域變數中
<code>lstore_<n></code>	把長整數放進第 <n> 號和第 <n+1> 號區域變數中
<code>fstore</code>	把浮點數放進區域變數中
<code>fstore_<n></code>	把浮點數放進第 <n> 號區域變數中
<code>dstore</code>	把 double 數放進區域變數中
<code>dstore_<n></code>	把 double 數放進第 <n> 號和第 <n+1> 號區域變數中
<code>astore</code>	把物件指引放進區域變數中
<code>astore_<n></code>	把物件指引放進第 <n> 號區域變數中

比方說：

```

; 在第 1 號區域變數內存放空的物件指標
aconst_null
astore_1

; 在第 5 號區域變數內存放整數 20
bipush 20
istore 5

iload 5
; 現在堆疊內包含 5

```

`double` 和 `long` 型態的資料需要兩個連續的區域變數來容納，所以如果你寫出下面的程式碼：

```
dconst_0
dstore 5
; 現在堆疊內包含 5
```

那麼，第 5 號和第 6 號區域變數會被用來合力容納 0.0。

有關區域變數的其它指令，

好吧！我承認我言過其實了，事實上，你除了可以把區域變數的值放進堆疊，或把堆疊的值放進區域變數之外，還可以對區域變數作一些其它的操作，表 7-6 列出這些指令。

表 7-6：wide 和 iinc 指令

指令名稱	描述
iinc	遞增整數區域變數的值
wide	下面的一道指令使用 16 位元的索引

iinc 用來遞增區域變數內的整數值，比方說：

```
iinc 1 2
```

這用來把第 1 號區域變數內的值加 2。下面的敘述也做得到相同的功能，但程式長多了：

```
iload_1 ; 把第 1 號區域變數的值推入堆疊中
iconst_2 ; 把整數 2 推入堆疊中
iadd ; 兩者相加
istore_1 ; 把結果存回第 1 號區域變數
```

wide 修飾字用來延伸你所能觸及的區域變數。不用 wide，你只能定位 256 個區域變數。使用 wide，你能夠定位 65536 個區域變數。比方說：

```
wide
iload 300 ; 取出第 300 號區域變數的值
```

在 Jasmin 中，如果你所指定的區域變數之編號超過 255，Jasmin 會自動為你插入 wide 的操作碼。

陣列

在 JVM 中，陣列其實就是物件，陣列實體屬於某特定類別。你可以呼叫陣列的行為——比方說，你可以用 `clone()` 行為來複製陣列，也可以用 `getClass()` 來取得其類別。陣列和物件之間的主要差異有：

- 陣列的內容是透過索引值來存取，類別則是透過欄位名稱。
- 陣列是同質的，一個陣列內的每個元素都是相同型態。
- 陣列實體的長度可以不同。（譯者註：這句話是說，一樣屬於整數陣列，A 陣列的長度是 10，而 B 陣列的長度可以是 5。）
- JVM 會自動為陣列產生類別，不需要到類別檔中讀取。

每個陣列都有自己的類別，需要用到此類別時，JVM 會自動為陣列產生類別。比方說，當你要產生整數的二維陣列時（型態描述子為「`[[I]`」，那麼 JVM 會自動產生一個「`[[I]`」的類別（如果目前還沒有此類別的話）。陣列類別一定是繼承自 `java.lang.Object`，所以適用於 `Object` 的行為也適用於陣列。陣列也實作了物件該有的介面，比方說，`Cloneable`。陣列的類別階層請看圖 7-1。

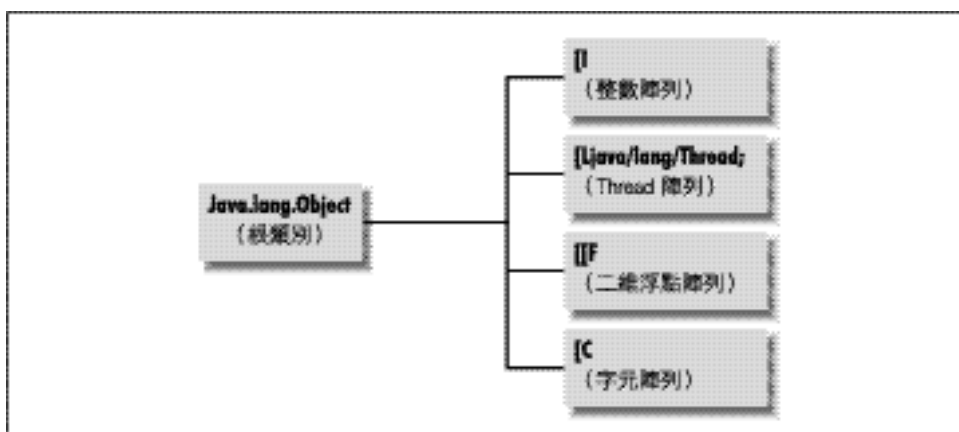


圖 7-1：某些陣列類別

JVM 指令中適用物件者一般來說也適用陣列。比方說，你可以用 `instanceof` 來測試陣列型態：

```

aload_1          ; 把第 1 號區域變數內的物件指引放進陣列中
instanceof [I    ; 測試是否為整數陣列
; 如果是的話，堆疊內容為 1，否則為 0

```

JVM 也提供陣列專用指令，包括了建立陣列，把值放進陣列，取出陣列內容（這些稍後再討論）。

請注意，雖然你可以建立 `boolean`、`byte`、`char`、`short` 的陣列（使用 `newarray`），但是當你從這種陣列中取出值時，JVM 會自動將其轉成 `int` 型態。

同樣地，如果你要傳值到 `boolean`、`byte`、`char`、`short` 陣列中，你給 JVM 的是整數，但 JVM 會將其截短之後才放進陣列中。

建立陣列

用來建立陣列的指令有三個，列於表 7-7 中。

表 7-7：建立陣列的指令

指令名稱	描述
<code>newarray</code>	配置原始型態的陣列
<code>anewarray</code>	配置物件的陣列
<code>multianewarray</code>	配置多維陣列

`newarray` 用來建立整數、浮點數、字元等八種原始型態的陣列。陣列內的元素會被初始化為 0。

比方說，建立一個五個元素的整數陣列，程式如下：

```

bipush 5        ; 把 5 推到堆疊內
newarray int    ; 建立長度為 5 的整數陣列
astore_1       ; 把陣列存到第 1 號區域變數中

```

這就好像下面的 Java 敘述：

```
int x[] = new int[5];
```

`anewarray` 用來建立指引陣列（指引指向物件）。比方說，下面的程式用來建立五個元素的執行緒陣列：

```
bipush 2
anewarray java/lang/Thread
astore_1
```

對等的 Java 敘述是：

```
Thread x[] = new Thread[5];
```

在分配完陣列空間之後，請務必記得初始化，否則此陣列內的每個元素都只是 `null`。

`multianewarray` 比起前二者複雜多了，用來建立多維陣列。比方說：

```
float matrix[][] = new float[4][4];
```

要建立類似上面 Java 敘述所建立的陣列，你應該使用下面的 JVM 敘述：

```
bipush 4
bipush 4
multianewarray [[F 2 ; 建立浮點陣列的陣列
astore_1
```

請看參考資料中的相關細節，藉以瞭解 Java 對多維陣列的管理方式。

讀出陣列內的元素值

表 7-8 內所列出來的指令用來讀出陣列內的元素值。

表 7-8：讀出陣列內的元素值

指令名稱	描述
<code>iaload</code>	讀出陣列內的整數值
<code>laload</code>	讀出陣列內的長整數值
<code>faload</code>	讀出陣列內的浮點數值
<code>daload</code>	讀出陣列內的 <code>double</code> 值
<code>aaload</code>	讀出陣列內的物件指引值
<code>baload</code>	讀出陣列內的 <code>byte/boolean</code> 值
<code>caload</code>	讀出陣列內的字元
<code>saload</code>	讀出陣列內的短整數值

比方說，當你寫出下面的 Java 程式：

```
arr[5]
```

假設 `arr` 是整數陣列，編譯成 JVM 碼之後如下：

```
aload 1      ; 把第 1 號區域變數內的陣列指引（也就是 arr）放到堆疊中
bipush 5     ; 把整數 5 放到堆疊
iaload       ; 把 arr 和 5 取出堆疊，把 arr[5] 放進堆疊
```

對於其它的陣列型態，把 `iaload` 用其它指令取代即可（比方說，`aaload` 用來處理物件陣列；`laload` 用來處理長整數陣列 等）。

寫入陣列內的元素值

表 7-9 內所列出來的指令用來寫入陣列內的元素值。

表 7-9：寫入陣列內的元素值

指令名稱	描述
iastore	把整數值寫入陣列內
lastore	把長整數值寫入陣列內
fastore	把浮點數值寫入陣列內
dastore	把 double 值寫入陣列內
aastore	把物件指引值寫入陣列內
bastore	把 byte/boolean 值寫入陣列內
castore	把字元寫入陣列內
sastore	把短整數值寫入陣列內

比方說，如果第 1 號區域變數是整數陣列的指引，你可以藉由下面的程式把陣列中的第五個元素設定成整數 3：

```

aload_1      ; 把第 1 號區域變數內的整數陣列放到堆疊中
bipush 5
bipush 3

iastore      ; array[5] := 3

```

有關陣列的其它指令

還有一個指令我們尚未提及，同時這也是最有用的指令之一，這個指令是 `arraylength`，用來取得陣列的長度。

下面是此指令的使用範例：

```

.method public static main([Ljava/lang/String;)V
  aload_0
  arraylength
  ; 堆疊內頂端資料是 main 的字串陣列參數的長度值
  ...
.end method

```

物件

JVM 提供一些物件相關的指令，包括建立新物件、處理物件欄位、辨識物件、以及引用物件的行為。行為的引用將在第 9 章「流程控制」中介紹。本節要介紹建立物件的方法、欄位的使用、以及檢查物件種類。

建立物件

我們使用 `new` 指令來建立新物件。請注意，`new` 本身並不足以建立有用的新物件，還必須呼叫其任一個 `<init>` 行為才行，如下面的範例所示：

```
new java/lang/StringBuffer    ; 配置一個 StringBuffer 的記憶體
; 現在初始化此記憶體，我們先多準備一份指引 (透過 dup 指令),
; 然後再呼叫 <init> 行為來初始化。
dup
invokespecial java/lang/StringBuffer/<init>()V
; 物件現在已經備妥在堆疊中了，
; 我們將其存到第 1 號區域變數中。
astore_1
```

Java 的驗證者要求未經初始化的物件不可以存到區域變數或欄位中（譯者註：但可以在堆疊中）。當運算元堆疊內有未初始化的物件時，不可以往回跳。

處理欄位

每個 Java 物件都是由一些欄位組成的，欄位若不是類別自己定義的，就是繼承自父類別而來的。你不可以覆蓋（`override`）繼承來的欄位，但可以遮蔽（`shadow`）它們。比方說，如果你有下面兩個類別：（譯者註：「覆蓋」是指「取代」，只有新的欄位留下來；而「遮蔽」是指兩者同時存在，但後者的出現讓前者黯然失色。）

```
class Foo {
    int myField;
}
class Foo2 extends Foo{
    float myField;
}
```

Foo 的 myField 與 Foo2 的 myField 名稱衝突。事實上，在 JVM 中此二者是不同的欄位。如果你去看看 Foo2 實體的資料區塊，你會發現其中包含：

- (繼承自 java.lang.Object 的數個欄位)
- int Foo.myField
- float Foo2.myField

這種情況我們就說 Foo2 的 myField 把 Foo 的 myField 遮蔽住了。在 JVM 中，不會有這種曖昧不明的狀況，因為一律要在前面要冠上定義的類別。

除了經由物件所取得的欄位之外，JVM 也支援經過類別來取得的欄位，稱之為靜態欄位。靜態欄位被存在類別的記錄中，而不是在實體中，所以你不需要為了使用靜態欄位而建立物件實體。

在 Jasmin 中，你可以透過「.field」假指令來增加新的欄位。比方說，第 2 章的 HelloWeb 程式中，有下面這列敘述：

```
.field private HelloWeb/font Ljava/awt/Font
```

這就在 HelloWeb 類別中宣告了一個名為 font 的私有實體欄位，這個欄位容納一個指向一個 Font 物件的指引。

表 7-10 列出 10 個存取欄位的指令。

表 7-10：欄位指令

指令名稱	描述
putfield	設定物件欄位的值
getfield	取得物件欄位的值
putstatic	設定靜態欄位的值
getstatic	取得物靜態位的值

這些指令所需要的參數都一樣，分別是：類別全名 + 欄位名稱、欄位的型態描述子，如下面的範例所示：

```

; 取得 java.lang.System 中名為 out 的欄位, 其型態為 PrintStream
getstatic java/lang/System/out Ljava/io/PrintStream;

; 堆疊內現在有一個 PrintStream 物件的指引

; 取得 this.font, 假設 -this- 是 HelloWeb 的實體
aload_0      ; 把 -this- 推入堆疊
getfield HelloWeb/font Ljava/awt/Font;      ; 取得 this.font
; 堆疊現在有一個 Font 物件的指引

```

第十三章「指令參考資料」對此四指令有範例以及更詳細的說明。

有關物件的其它操作

表 7-11 列出用來測試物件型態的指令。

表 7-10：欄位指令

指令名稱	描述
checkcast	確定物件或陣列屬於某型態
instanceof	測試物件或陣列是否屬於某型態

比方說：

```

; 檢查某物件的型態
aload_0
instanceof java/lang/Thread
; 如果第 0 號區域變數指向一個 Thread (或 Thread 的子類別) 的物件,
; 那麼堆疊會被放進整數 1, 否則放入 0。

; 檢查陣列型態
aload_0
instanceof [I
; 如果第 0 號區域變數指向一個 int[] 的陣列,
; 那麼堆疊會被放進整數 1, 否則放入 0。

; 檢查一個介面

```

```
aload_0
checkcast java/lang/Throwable
; 如果第 0 號區域變數指向一個物件，此物件實踐了 Throwable 介面，
; 那麼此物件仍被保留在堆疊中，且繼續執行後面的指令。
; 否則，丟出 ClassCastException。
```

指定的相容性

`checkcast` 和 `instanceof` 指令用到了「相容性」的觀念。如果類別 A 定義了類別 B 全部的操作方式，那麼我們就說 A 和 B 相容。

對物件來說，下面的假碼用來示範 Java 如何偵測某物件（其型態為 `myType`）和一特定類別（`requiredType`）相容：（譯者註：這只是假碼，所以有的行為其實不存在，撰寫價碼的目的只是為了讓你容易理解罷了。）

```
; 檢查某物件的型態
boolean compatible(Class myType, Class requiredType) {
    if (requiredType.isInterface()) { // 是介面嗎？
        // 如果的確實作了此介面，傳回 true
        return (myType.implements(requiredType));
    } else if (requiredType.isClass()) { // 是類別嗎？
        // 測試型態是否和 requiredType 一樣，
        if (myType == requiredType) return true;
        // 或者 requiredType 是父類別
        while (myType != java.lang.Object) {
            if (myType == requiredType)
                return true;
            myType = myType.getSuperClass();
        }
    }
    return false; // 不相容
}
```

對陣列來說，需要更複雜一點，因為你還必須檢查陣列內元素的型態。下面是簡化後的假碼，其中 `myType` 是陣列物件的型態，`requiredType` 是你期望的型態：

```
boolean compatible(Class myType, Class requiredType) {
    if (requiredType.isInterface()) {
        // 只要是陣列，一定都實作了 java.lang.Object
        return (java.lang.Object.implements(requiredType));
    } else if (requiredType.isClass()) {
        // 陣列型態的父類別是 java/lang/Object，所以：
        return (requiredType == myType || requiredType == java.lang.Object);
    } else {
        // myType 是陣列型態，且 requiredType 也是陣列型態。
        // 比較它們的陣列元素之型態
        Class type1 = myType.getComponentType();
        Class type2 = requiredType.getComponentType();
        // 遞迴地測試
        return (type1 == type2 || compatible(type1,type2));
    }
}
```

有了上面兩個行為，我們可以把 instanceof 和 checkcast 的功能表達如下：

```
boolean instanceof(Type t) {
    return compatible(this.getClass(), t);
}

void checkcast(Type t) {
    if (!compatible(this.getClass(), t)) {
        throw new ClassCastException();
    }
}
```

指定操作（譯者註：指的是「=」）常常需要用到相容的概念。當你指定某物件到欄位或陣列（用 astore）時，JVM 會檢查物件和目的區的型態相容否，如果不相容，就丟出一個例外。

