本章內容：

* 運算元的轉換
* 指定運算子
* 算術運算子
* 位元運算子
* 位移運算子
* 邏輯運算子
* 遞加與遞減運算子
* 比較運算子
* 型態運算子
* 條件運算子
* 逗號運算子
* 優先等級與結合性

第十一章

WMLScript 之運算子與運算式

現在，你對 WMLScript 裡的各種資料型態都已有些認識，接下來要做的工作，便是如何使用這些運算子（operator），組成需要的運算式（expression）。

如果你撰寫過像 C 語言或 Java 這一類的程式，那麼本章所要介紹的內容，對你而言，可能會有似曾相識之感。然而，其中仍有些微的差別，大部分是因為 WMLScript 的動態型態的關係。當遇到這種差異時，我們會特別加以標明。

附錄 C「WMLScript 運算子總覽」中，列出了 Script 裡所有的運算子，並概略說明每一個的作用。如前面所述，假使你曾碰過 C 或 Java 程式，你可以直接跳往下一章，當看見不熟悉的運算子時，再回來查詢。不過，下一節卻是蠻值得閱讀的，我們將介紹在不同的運算子下，如何做資料型態的轉換，而這些都延續先前第十章的討論。

運算元的轉換

有些運算子可處理一種以上的資料型態。舉例來說，像 * 這個乘法運算子，可用來做兩個整數的相乘，亦可用在浮點數的計算。WMLScript 中，定義了在哪些情形下，必須做適當的資料轉換：

- 若運算子的運算元（operand）具有相同的型態，那麼可將資料直接轉換成它們的型態。萬一轉換失敗的話，整個運算結果會是 `invalid`（無效的）。
- 如果該運算子的運算元可以是任何型態，則直接使用各自的數值。因為不需要，也不必做任何轉換。

如同你看到的這些規則，有時候可能會有一些預料之外的事情發生，包含你認為該做的處理卻沒有出現。

假使某一時刻，你不確定該數值的型態，也許可以使用標準函式庫中的函數，強迫做一個型態轉換。這些函數在第十五章裡會有詳細說明。

單元的整數或浮點數

只帶有一個整數或浮點數作為參數的單元運算子（unary operator，如單一的 + 或 -），首先會試著將運算元轉換為整數。如果成功的話，便以整數方式執行這項運算，其結果將會是一個整數。

但若這個轉換失敗了，則會繼續試著將運算元轉換成浮點數。假使一切順利，此運算會以數值方式加以處理。

萬一這兩種轉換（整數或是浮點數）都不成功，結果會是 `invalid`。

舉例來說：

+2 其結果為整數 2。

+2.0 其結果為浮點數 2.0。

-true 其結果為整數 -1。

-false 其結果為整數 0。

+"123.456" 其結果為浮點數 123.456。

+"a" 其結果為 invalid（因為無法轉換成整數或浮點數）。

-"1e99" 其結果為 invalid（因為超過浮點數所能表示的合理範圍）。

二元的整數或浮點數

帶有兩個整數或浮點數為參數的二元運算子（binary operator，像是 * 或是 -），首先會試著將兩個運算元都轉換為整數。如果成功的話，便以整數方式執行這項運算，其結果同樣也會是一個整數。

假使有任何一個運算元無法轉換成整數，則運算子會試著將它們都轉換為浮點數。相同地，若處理成功，則執行這項動作。

萬一兩項轉換都失敗了，運算結果便是 invalid。

同樣地，我們舉例加以說明：

3*4 其結果為整數 12。

3.0*4 其結果為浮點數 12.0。

"3"*4 其結果為整數 12。

"2"-"9" 其結果為整數 -7。

"2"-"9e0" 其結果為浮點數 -7。

6.3*"x" 其結果為 invalid（因為無法轉換成整數或浮點數）。

0.1*"1e999" 其結果為 invalid（因為超過浮點數所能表示的合理範圍）。

整數、浮點數與字串

唯一可做數值運算，還能處理字串連結的，就是二元運算子 +。舉例來說，"x"+"y" 的結果會變成 "xy"。

如果運算元中，有任何一個是字串（string）的話，那麼運算子會試著將兩個都轉換成字串。若是成功，兩個字串便會連結在一起，並傳回結果。反之，若另一個轉換失敗的話，則結果會是 invalid。

假使兩個運算元都不是字串的話，那會先試著將兩者都轉換為整數（integer）。若一切順利，則執行兩整數相加的動作，其結果也是個整數。

同樣地，若兩運算元無法都轉換為整數，則此運算元會盡可能將其轉換成浮點數（floating-point）。如果轉換成功，便執行浮點數相加的處理，所以結果會是一個浮點數。

萬一，這三種情況都失敗的話，那麼運算的結果會是 invalid。

我們也來舉些例子，說明其處理方式：

"x"+"y" 其結果為字串 "xy"。

"x"+1 其結果為字串 "x1"。

1+"2" 其結果為字串 "12"。

1+2其結果為整數 3。

1+2e0 其結果為浮點數 3.0。

+ 運算子

+ 運算子對於特別的字串處理，可能會造成一些錯誤發生。例如，像 1+"2" 這個運算，其結果會是字串 "12"，而不是整數 3。其它的運算如 1-"2" 與 1*"2"，結果分別會是我們所預期的 -1 與 2。所以，我們必須特別注意這種狀況的處理。

另外，還有一種情況也要小心，就是當你使用標準函式庫裡的函數，讀取瀏覽器變數的數值時，其結果總是會以字串的方式傳回。因此，+ 運算子可能不會做你所預期的事。例如：

```
var count = WMLBrowser.getVar ("count");
count = count + 1;
```

它會將字元 1 串接在字串 count 的結尾，而不是做加 1 的動作。為了確實執行加 1 的運算，你必須以這種方式轉寫：

```
var count = Lang.parseInt (WMLBrowser.getVar ("count"));
count = count + 1;
```

或是另一種方法：

```
var count = WMLBrowser.getVar ("count") - 0;
count = count + 1;
```

（其中第二個方法是執行數值減 0 的動作，以強迫其轉換為整數或浮點數。）

我們衷心希望，在未來的 WMLScript 規格說明中，這一類模糊不清的狀況能加以修訂，才可免去撰寫程式時所需的額外處理。

比較運算子

比較運算子（comparison operator，例如等於、小於）的運算元也可以是字串、整數或浮點數。唯一和 + 運算子不同的是，它的比較結果會是一個布林值（除非所有的轉換處理都失敗，才會造成結果變成 invalid）。

如果運算元中，有任何一個是字串的話，那麼運算子會先試著將另一個也轉成字串。若是成功，則以字母順序逐一比較兩字串裡的字元。

假使兩個運算元都不是字串，它會試著將兩者都轉為整數。若一切順利，則執行兩整數比較的動作。

同樣地，若兩運算元都無法轉成整數，則此運算元會試著將它們轉換為浮點數。如果轉換成功，便以浮點數形式來比較大小。

萬一這三種情況都失敗，那麼運算的結果也會變成 invalid。

舉例說明：

"FOO"<"foo" 其結果為 true（以字串方式作比較）。

2<"10" 其結果為 false（以字串方式作比較："2"<"10"）。

2<10 其結果為 true（以整數方式作比較）。

1<"1e-6" 其結果為 true（以字串方式作比較）。

1<1e-6 其結果為 false（以浮點數方式作比較：1.0<0.000001）。

如同 + 運算子，你也必須特別注意字串的比較處理。尤其像 2>"10" 這種情況，其結果會是 true，因為字元 2 確實大於字元 1。你要謹記在心，以免日後犯錯。

指定運算子

基本上，運算式並不常使用到，除非你能指定其值到某個變數裡，而這種設定值的動作，也就是指定運算子（assignment operator）的主要作用。任何一個指定運算子的左邊部分，必須是一個需要指定值的變數。

最普遍常見的指定運算子，可說是最簡單的運算，只是將某個運算式指定至該變數中。通常用 = 來表示。

在 WMLScript 中，指定運算子可和其它運算子結合操作，成為一種新的簡略形式。例如：

```
x += y
```

就是這個運算的簡潔寫法：

```
x = (x + y)
```

像這種可與指定運算子組合運用的二元運算子，總共有十二個：+=、-=、*=、/=、div=、%=、&=、|=、^=、<<=、>>= 與 >>>=。每種運算子都遵循一般的通用規則，例如，+= 能處理字串的連結，正如 + 一樣，而 /= 的運算結果也總會是一個浮點數。唯一要注意的是，布林值（Boolean）與比較運算子，無法與指定運算子結合。如果你真的需要這種指定的動作，那只有以完整的形式表達才行（不過，大部分情形是不太需要）。

任何指定運算的結果，就是被指定的數值。例如：

```
foo (a = b)
```

其將 a 的值設定為 b 的內容，再以新的 a 值呼叫函數 foo()。同樣地，下面的程式：

```
foo (a *= b)</C>
```

是將 a 乘以 b、放入 a 後，再以新的 a 值執行函數 foo()。

算術運算子

WMLScript 提供了所有一般的算術運算子（arithmetic operator），實際上，其文法與 C 或 Java 完全相同。

其中最簡單的兩個算術運算子，分別為單元的正號（以 + 表示），與單元的負號（以 - 表示）。單元的正號是將該值轉換為某個數字【註】，而單元的負號則是在完成轉換後，再取該值的負數（即用零減去該數的結果）。

舉例來說：

+2 其結果為整數 2。

-2 其結果為整數 -2。

-"-2" 其結果為整數 2。

-"2e0" 其結果為整數 -2.0。

除了這兩個單元運算子外，另外還有六個二元的算術運算子。其中包含了以 + 表示的加法，以 - 表示的減法，以 * 表示的乘法，以及兩個除法運算子：一個運用在浮點數的除法 /，其結果會是浮點數，與運用在整數中的 div，其結果亦會是一個整數。最後一個運算子為 modulo，即所謂的餘數運算子（remainder operator），用來表示計算整數除法中的餘數（以 % 表示）。

如果你使用過 C 語言或是 Java 的話，相信除了 div 之外，其它所有的運算子你應該會覺得蠻熟悉的。（在 C 與 Java 中，主要利用運算元與計算結果的資料型態，來決定要做整數或浮點數的除法，因此不需要有 div 這個運算子。）

在這些運算子當中，會發生一個稍微複雜的情況：本章前幾節曾提及「運算元轉換」的問題，特別是 + 運算子；如果有任一運算元是字串的話，該運算的作用會是字串的連結，而非數值相加。如果使用者一時失察，這可能會導致錯誤的發生。

註　如果該值原本即為一個數字，便不需轉換。這裡所謂的「轉換成某個數字」，意思是「若該值不是一個數字，便執行轉換處理」。

讓我們舉例說明算術運算子的運作方式：

3 * 2 其結果為整數 6。

"3" * 2其結果為整數 6。

3 / 2 其結果為浮點數 1.5。

3 div 2 其結果為整數 1。

3.0 div 2 其結果為 invalid（因為 3.0 不能被轉換成整數）。

1 + 2 其結果為整數 3。

1 + 2.0 其結果為浮點數 3。

1 + "2" 其結果為字串 "12"。

8 % 3 其結果為整數 2。

-8 % 3 其結果為整數 -1。

8 % -3 其結果為整數 1。

-8 % -3 其結果為整數 -2。

位元運算子

除了一般的算術運算子之外，WMLScript 也提供了四種針對整數的各個位元，從事布林的算術操作。這些運算都只能處理整數部分，假使其中任一運算元無法轉換成整數，則其運算結果會是 invalid。

位元補數（bitwise complement，又稱作 not 運算）運算子的作用為翻轉參數中的每一個位元（將每個為 0 的位元變成 1，而每個為 1 的位元變成 0）。它是以 ~ 符號來表示。

位元運算子中的 *and*、*or* 與 *exclusive or* 等運算，都需作用在兩個整數上。對每一個位元（總共 32 個位元）而言，其運算結果都是經由這兩個運算元裡相對應的位置，做位元處理後所得的值。如果兩運算元的對應位元皆為 1 的話，則位元運算子的 *and*（以 & 表示），會將該位元設定為 1。如果兩運算元的對應位元中，若任一位元為 1 的話，那麼，位元運算子的 *or*（以 | 表示），會將該位元設定為 1。若是兩運算元的對應位元只有一個為 1 的話（即另一個為 0），則位元運算子的 *exclusive or*（以 ^ 表示），會將該位元設定為 1。這乍聽之下有點複雜，不過，稍後看過例子，就會比較清楚了。

事實上，這幾個運算子的作用和 C 或 Java 裡的完全相同。因此，若你已知道這一類程式語言，那麼對這些運算應該都蠻能理解的。

讓我們舉些例子來說明；其中有些是以十六進位來表示，這樣會比較方便知道該數值裡各位元的狀況：

0x0110 & 0x0011 其結果為整數 0x0010。

0x0110 | 0x0011 其結果為整數 0x0111。

0x0110 ^ 0x0011 其結果為整數 0x0101。

6 & 3 其結果為整數 2。

6 | 3 其結果為整數 7。

6 ^ 3 其結果為整數 5。

~1 其結果為整數 0xFFFFFFFE。

~0 其結果為整數 0xFFFFFFFF。

實際上，位元運算子（bitwise operator）並不常被使用，所以，若你還不是很瞭解，也不必太過操心。

位移運算子

除了前面介紹過處理各個位元的運算子外，WMLScript 還提供了位移運算子（shift operator），允許使用者將所有位元向左或向右移動數個位置。也因為整數在電腦裡的表示方式，這種動作就相當於乘以或除以 2 的某個次方。

向右移位有兩種運算子：>> 與 >>>。這兩者之間的差異，在於 >> 可以正確地處理負數，所以，你能用它來執行除以 2 的某次方的運算。另一個運算子 >>>，會將所有參數視為無正負號的數字來處理。但是，因為在 WMLScript 中，所有整數都帶有正負號，所以，它無法如你所預期般地處理負數運算。不過，它卻也是最常使用的運算子，如同 &、|、^ 一樣，當你只是使用一個整數來儲存 32 個各具意義的位元，而非用以儲存一個標準整數時，你就必須用這個運算子來處理。

向左移位是以 << 表示。它對所有數值的運作方式都相同，故只需一種運算子即可。

所有的位移運算子都會要求它們的兩個運算元必須皆為整數才行。此外，右邊的數值（位移多少個位置）不能為負數。（第二項條件在 WMLScript 規格說明中，並未明確地加以闡述，因此不同的解譯器，處理位移為負的方式可能會不太一致。）

舉例說明位移運算子的運作方式：

10 << 3 其結果為 80（相當於 10 * 2 * 2 * 2）。

10 >> 3 其結果為 1（相當於 10 div 2 div 2 div 2）。

10 >>> 3 其結果為 1（對於正整數而言，>> 與 >>> 的作用相同）。

-10 << 3 其結果為 -80（相當於 -10 * 2 * 2 * 2）。

-10 >> 3 其結果為 -1（相當於 -10 div 2 div 2 div 2）。

-10 >>> 3 其結果為 536870911（>>> 無法處理負數的運算）。

雖然 >> 的作用幾乎等於該整數除以 2 的某個次方，但其捨棄進位的結果卻可能有所不同。運算子 div 總是會捨棄進位而變成 0，所以，像這個例子：

```
-1 div 2
```

結果會是 0。但是運算子 << 會將負數進位成 -1，而不會是 0（但正整數捨棄進位的結果都是一樣）。這意味著：

```
-1 >> 1
```

數值會是 -1，而非 div 的執行結果 0。

事情可能比原本看起來的還要複雜。你也許會想，只使用 * 與 div 就好了，何必對位移運算子到底該如何操作而自尋煩惱。事實上，這也沒錯，並沒有什麼非用它們不可的理由，假如你不想用也無所謂。在實際的 WMLScript 程式中，這些運算子並不多見。它們會被包含在內，只是為了讓運算處理更為完整而已。

邏輯運算子

位元運算子在分別的情況下相當有用，但更多時候，你只會想要一個布林的 *and* 或 *or* 運算子，來將運算元轉換成一個布林值，而不是整數。

這一類運算子（稱作邏輯運算子（logical operator），為了與前面介紹的位元運算子有所區別）也可用來做短路估算（short-circuit evaluation）；也就是說，先計算左邊的運算元，看看在尚未計算右邊運算元之前，是否已能算出最後的結果。

邏輯運算子的 *and* 是以 && 來表示，它會先計算左邊的運算元，假如結果是布林值 false，那麼，整個運算的結果就是 false。若左邊的運算元不能轉成布林值，那運算結果會是 invalid。在這兩種情形下，右邊的運算元則完全不須計算。然而，若左邊的運算元可轉為布林值 true，那麼，運算結果則端視右邊運算元而定；即為它轉換為布林值後的值。

邏輯運算子 *or* 是以 || 來表示。它一樣會先計算左邊的運算元。假如結果是布林值 true，那麼整個運算結果就是 true。如同邏輯運算子 *and*，若左邊的運算元不能轉換成布林值，那麼運算結果會是 invalid。同樣地，在這兩種情形下，右邊的運算元完全不須計算。然而，若左邊的運算元可轉為布林值 false，則結果端視右邊運算元，也就是它轉換成布林值後的值。

舉例說明：

(1+1 == 2) || foo() 其結果為 true，則不須執行函數 foo()。

(1+1 == 3) || foo() 其結果為執行 foo() 後經轉換的布林值。

(1/0) || foo() 其結果為 invalid，故不必執行函數 foo()。

(1+1 == 2) && foo() 其結果為執行 foo() 後經轉換的布林值。

(1+1 == 3) && foo() 其結果為 false，則不必執行函數 foo()。

(1/0) && foo() 其結果為 invalid，故不必執行函數 foo()。

遞加與遞減運算子

WMLScript 裡有四種運算子，可提供方便又簡潔的寫法，用來處理常見的運算－某變數值加一或減一的運算。

前遞加（preincrement）與前遞減（predecrement）運算子分別寫成：

```
++var
```

與

```
--var
```

如果想將該變數的內容轉換成整數或浮點數時，會先做加 1（++）或減 1（--）的動作，再把值存回該變數中。所以，運算的結果會是變數更改後的新值。

另外，後遞加（postincrement）與後遞減（postdecrement）運算子可分別表示為：

```
var++
```

與

```
var--
```

其動作會與先前有些不同。它們仍然會改變該變數的值，但運算子所回傳的結果，卻是變數未更改前的舊值（而非新值）。

例如，假設變數 a 的值為整數 42，變數 b 的值為整數 0x7FFFFFFF（此為最大且有效的正整數），變數 c 為浮點數 2.3，變數 d 則為字串 "1e2"，以及變數 e 是字串 "foo"。以下列出幾種運算方式與結果：

a++ 其結果為整數 42，並設定 a 為整數 43。

++a 其結果為整數 43，並設定 a 為整數 43。

a-- 其結果為整數 42，並設定 a 為整數 41。

--a 其結果為整數 41，並設定 a 為整數 41。

b++ 其結果為整數 0x7FFFFFFF，並設定 b 為 invalid（溢位，因為數值過大）。

++b 其結果為 invalid，並設定 b 為 invalid（同樣是溢位）。

c++ 其結果為浮點數 2.3，並設定 c 為浮點數 3.3。

--c 其結果為浮點數 1.3，並設定 c 為浮點數 1.3。

d-- 其結果為字串 "1e2"，並設定 d 為浮點數 99.0。

++d 其結果為浮點數 101.0，並設定 d 為浮點數 101.0。

e++ 其結果為字串 "foo"，並設定 e 為 invalid（無法轉換）。

--e 其結果為 invalid，並設定 e 為 invalid（無法轉換）。

從上面的例子中，我們須注意後遞加與後遞減兩種形式的運算結果，是該變數更改前的數值。

另外，還有一點要瞭解的是，雖然在運算式中，使用這些運算子是合理且有效的，但在大部分情形下，它們都會單獨出現，就好像只為了做遞加或遞減的處理而已。舉例來說，我們常在 for 迴圈裡，使用一個遞加的運算：

```
for (var i=0; i<end; i++) {
    ...
}
```

我們也可以使用前遞加的方式來撰寫：

```
for (var i=0; i<end; ++i) {
    ...
}
```

比較運算子

WMLScript 提供了六種標準的比較運算子（comparison operator）：== 表示「等於」，!= 表示「不等於」，< 表示「小於」，<= 表示「小於或等於」，> 表示「大於」，以及 >= 表示「大於或等於」。

在 C 這一類的語言中，使用「等於」來做比較運算，是相當典型的處理方式。如果你要做兩數值是否相等的比較運算，必須小心地輸入，它的正確形式是 ==，而不是單一的 =。假使不注意寫成：

```
if (a = b)
```

而非正確的：

```
if (a==b)
```

其結果仍然會是一個有效的 WMLScript 敘述。不過，實際執行的動作並非比較 a 與 b 是否相等，而是將 b 的值複製到變數 a 中，再測試 a 能否轉換成布林值 true。這可能會導致一些錯誤的發生，所以要特別小心處理。

這些比較運算子的處理過程，首先會檢查是否有任何運算元為一個字串。如果是的話，便把另一個運算元也轉換成字串形式，再以字母優先順序比較兩字串。大寫字母小於小寫字母，所以，"one" 小於 "two"，"THREE" 小於 "four"。若一字串是另一個字串前面的一部分，則較短者小於較長字串，如 "six" 小於 "sixteen"。

如果運算元都不是字串形式，則會試著將兩個都轉換成整數。若轉換成功，則以整數形式比較兩運算元。

但若兩者皆無法轉換成整數，則繼續試著轉換成浮點數運算。如果成功了，便以浮點數形式比較兩運算元。

萬一這三種處理都失敗（在這種情況下，其中一個是 invalid），則比較的結果會是 invalid。

舉例說明：

2 < 10 其結果為 true（以整數形式作比較）。

"2" < 10 其結果為 false（以字串形式作比較："2" < "10"）。

2.0 == 2 其結果為 true（以浮點數形式作比較）。

"six" > "seven" 其結果為 true（以字串形式作比較）。

true > false 其結果為 true（以整數形式作比較：1 > 0）。

"true" > false 其結果為 true（以字串形式作比較："true" > "false"）。

型態運算子

之前，我們已經介紹過 WMLScript 所提供眾多不同的資料型態，以及在需要時，如何處理型態轉換的問題。

在某些情形下，你可能想知道某個數值確實的型態為何，或者純粹檢查是否為 invalid（invalid 通常意味著某個錯誤的發生，所以，這種檢查可用來確認該運算是否正確地執行）。

很幸運地，WMLScript 提供了一些運算子，可用來檢驗其資料型態，以及該值是否有效。

其中，typeof 運算子可用來檢查該參數的資料型態，並傳回一個整數來表示結果。這個運算子並不會做任何的轉換動作，所回傳的整數只是標明參數的型態而已。

由於 typeof 完全不做轉換的處理，而且即使參數是 invalid，仍然會傳回一個整數值，所以，它的執行結果永遠都不會是 invalid。它的回傳值永遠會是個整數。

表格 11-1 列出五個整數值，分別代表 WMLScript 裡的五種資料型態。

表格 11-1 typeof 運算子的回傳結果

資料型態	回傳值
整數	0
浮點數	1
字串	2
布林值	3
Invalid	4

通常 typeof 所提供的資訊會比你想的還要多。例如，假設有一個複雜的算術運算式。你知道結果可能有兩種－不是一個整數，就是一個浮點數，除非運算過程中，發生錯誤，導致結果是 invalid。你可能得做個檢查，以確定程式是否正確地執行。於是，你需要類似下面的檢查過程：

```
if (typeof result != 4) {
     /* Type not invalid: result was OK. */
     ...
} else {
     /* Type invalid: evaluation failed. */
     ...
}
```

WMLScript 也提供一個簡單的運算子 isvalid，用來測試某數值是否為 invalid。如同 typeof 一樣，它亦不會試著去轉換該數值。

因此，之前那一小段程式，我們可用 isvalid 來改寫，使其更有效率，但作用仍然相同：

```
if (isvalid result) {
     /* Type not invalid: result was OK. */
     ...
} else {
     /* Type invalid: evaluation failed. */
     ...
}
```

讓我們舉例說明 typeof 的運作：

typeof 0 其結果為 0。

typeof 0.0 其結果為 1。

typeof "" 其結果為 2。

typeof true 其結果為 3。

typeof invalid 其結果為 4。

typeof "0" 其結果為 2。

typeof "true" 其結果為 2。

typeof (17+1/0) 其結果為 4。

typeof (10/5) 其結果為 1（/ 運算結果總為浮點數）。

條件運算子

在 WMLScript 中，所謂的條件運算子（conditional operator）是依照條件運算式的結果，從兩個子運算式中挑選出一個，其語法如下：

condition-expression ? *true-expression* : *false-expression*
（條件運算式？為真之運算式：為假之運算式）

其中的 *condition-expression* 會經轉換而成布林值，假使其值為 true 的話，結果便為 *true-expression*，而不會是 *false-expression*。反之，若其值為 false 或無法轉換，則結果便是 *false-expression*，同樣地 *true-expression* 亦不被使用。不管選擇的是 *true-expression* 還是 *false-expression*，都不會做轉換的處理。條件運算子的結果，其型態會與被選到的子運算式相同。

請注意，下面的運算：

```
a = b ? c : d;
```

作用完全等於：

```
if (b) {
a = c;
} else {
a = d;
}
```

（雖然作用相同，但使用條件運算子，通常會比使用 if 與 else 來的有效率。）

舉例說明條件運算子的操作：

`true ? "yes" : "no"` 其結果為字串 `"yes"`。

`false ? "yes" : "no"` 其結果為字串 `"no"`。

`invalid ? "yes" : "no"` 其結果為字串 `"no"`。

`true ? 1/2 : 1/0` 其結果為浮點數 `0.5`。

`false ? 1/2 : 1/0` 其結果為 `invalid`。

`true ? "" : foo()` 其結果為字串 `""`，且不呼叫 `foo ()`。

多重運算作用

一個具有多重運算作用（Multiple side effects）的運算式，對同一個變數的作用，並未明確地定義在 WMLScript 中。這種情況包含了一般典型的例子：

```
x = x++
```

（這裡的變數 x，因為 ++ 的運算結果而加 1，但同樣地，因 = 運算子而被指定新值）

由於各家瀏覽器與編譯器的不同，其運算方式也會有所差異。而且，沒有任何理由，要你以這種方式來做運算。所以，請不要撰寫類似上述的程式。

逗號運算子

在 WMLScript 裡，逗號運算子（,）並不常被使用在程式中。其作用主要為，計算左邊運算元，將結果捨棄，再計算右邊運算元。逗號運算的結果相當簡單，即為右邊運算元。

如果之前沒碰過這種運算，你可能會對它的處理行為感到不可思議。基本上，只有當左邊運算元具有某種形式的邊際效應時，它才會出現（如指定為某個變數，或呼叫某函數的回傳值）。在一般情形下，最常使用 , 運算子的時機，就是在 for 迴圈裡的初始與增值兩區域內（initializer 與 increment sections，請參考第十二章「WMLScript 敘述」）。例如：

```
for (i=0, order=1, total=0; i<count; ++i, order *= 10) {
total += foo (i, order);
}
```

在迴圈中，初始區使用了 , 運算子，來設定三個變數 i、order 與 total 的起始值。此外，在 increment 處，每重複一次，也以此運算子對 i 值加 1，對 order 乘以 10。

假使，, 運算子使用在函數呼叫的參數裡，或在迴圈中的初始區域，都可能會造成混淆不清的現象－究竟 , 代表的是運算本身，或者只是用來區隔下一個參數與變數宣告中的各個變數？為了避免這種問題的發生，我們必須將包含 , 的整個運算式括入括弧內。不過，最好的方式還是盡量少用這個運算子，取而代之的方法，就是多寫幾行程式囉。例如，像下面這段程式碼：

```
var x = foo(), bar();
```

就是錯的，因為在右邊運算式的前後，沒有使用括弧括起來。正確的寫法應該是：

```
var x = (foo (), bar ());
```

然而，最好的方法還是將它們分開，變成：

```
foo ();
var x = bar ();
```

優先等級與結合性

就像大部分的程式語言，WMLScript 對每一個運算子，都設定了不同層次的優先等級（precedence）。此外，每一個二元運算子還有一個結合性（associativity）之分。這兩種性質可讓運算式更加明確，意即不必使用很多的括號，就可表達運算的處理順序。

當運算式中出現了不同的運算子時，優先等級可以決定哪一部份必須先執行。例如，乘法運算優先於加法運算，所以這個運算：

```
3 + 4 * 5
```

事實上會等於：

```
3 + (4 * 5)
```

而不是：

```
(3 + 4) * 5
```

故其結果為 23，不是 35。

當運算式中出現了相同優先等級的運算子時，結合性決定了運算式中哪一部份必須先做計算（所有優先等級相同的運算子，其結合性也是一樣）。結合性的分類只有兩種－左邊或是右邊。具左結合性的運算子，會先執行左邊的運算；同樣地，具右結合性的運算子，則先計算右邊的結果。

舉例來說，+ 與 - 都是左結合性，而它們的優先等級也一樣，所以：

```
3 - 4 + 5
```

即相等於

```
(3 - 4) + 5
```

而不是

```
3 - (4 + 5)
```

故答案是 4 而不是 -6。

相同地，所有的指定運算子都具右結合性，其優先等級也一樣，因此：

```
a = b = c
```

實際上會等於：

```
a = (b = c)
```

而不是：

```
(a = b) = c
```

第三個指定方式是不合法的，因為：

```
a = b
```

結果不是一個變數，所以不能出現在 = 運算子的左邊。

在附錄 C 裡，當我們簡述所有的 WMLScript 運算子時，會連同它們的優先等級與結合性一起介紹。