Chapter 2 基本法则

在 Chapter 1 中，你已经编写了你的第一个 F#程序。我把它分解开来让你对你正在做什么有一个了解，然而许多代码还是一个谜团。在这一章中，我提供了必要的基本规则来帮助你完整地了解这些代码，但更重要的是，我介绍了更多的例子，它们能让你在了解更复杂的特性之前了解 F#的基本规则。

本章的第一节包含基元类型，比如 int 和 string，它们是所有 F#程序的基本构件。接下来我将讲解函数来让你能够操纵数据。

第四节详解了基本类型，比如 list、option 以及 unit。精通这些类型能让你形成 F#的面向对象和函数式风格，它们将在后续章节中讨论。

到这章结束时，你将能够编写处理数据的简单 F#程序。在后面的章节中，你将学习如何让代码更加强大和更具表现力，不过现在，让我们熟悉基本规则。

基元类型

类型是一种概念或者一种抽象，主要是为了增加安全性。类型在有转换的情况下表示一类事物。一些类型是直观的(表示一个整数)，而其他的类型这更加抽象(比如函数)。F#是静态类型的，这意味着类型检查会在编译时执行。例如如果一个函数接受一个整数参数，那么当你尝试传入一个字符串时，编译器就会报错。

与 C#和 VB.NET 类似，F#支持完整的.NET 基本类型，它们是许多编程语言的标准。它们被植入 F#语言中来与你自己定义的 user-defined 类型区分开。

要创建一个值，只需简单地通过 let 关键字来实现 let 绑定。例如，下面的代码在一个 FSI 会话中定义了一个新的值 x。你可以使用 let 绑定做更多的事，但我们将保留这些内容到 Chapter3 中：

```
> let x = 1;;
val x : int = 1
```

数字基元

数字基元包含两个类型：整数和浮点数。整数类型随大小变化，以便一些类型可以占用更少的内存来表示更小范围的数字。整数也分为带符号的(signed)和无符号的(unsigned)，这取决于它们能否表示负值。浮点类型也是随大小变化的；作为占用更多内存的回报，它们提供了更高的精度。要定义一个新的数字值，请使用 let 绑

定，后面跟上一个整数或者浮点数的字面值，再加上一个可选的后缀。后缀将决定整数或者浮点数的类型(一个关于可用的基元数字类型的完整的列表提供在 Table 2-1 中)：

```
> let answerToEverything = 42UL;;

val answerToEverything : uint64 = 42UL

> let pi = 3.1415926M;;

val pi : decimal = 3.1415926M

> let avogadro = 6.022e23;;

val avogadro : float = 6.022e23
```

Table 2-1. F#中的基元数字

类型	后缀	.NET 类型	范围
byte	uy	System.Byte	0 ~ 255
sbyte	y	System.SByte	−128 ~ 127
int16	s	System.Int16	−32,768 ~ 32,767
uint16	us	System.UInt16	0 ~ 65,535
int, int32		System.Int32	−231 ~ 231−1
uint32	u	System.UInt32	0 ~ 232−1
int64	L	System.Int64	−263 ~ 263−1
uint64	UL	System.UInt64	0 ~ 264−1
float		System.Double	一种基于 IEEE64 标准的双精度浮点数类型。表示精度约 15 位有效数字的值。
float32	f	System.Single	一种基于 IEEE32 标准的双精度浮点数类型。表示精度约 7 位有效数字的值。
decimal	M	System.Decimal	一种固定精度的浮点数类型。表示精度恰好 28 位有效数字的值。

F#也允许你使用十六进制 hexadecimal (base 16)、八进制 octal (base 8)或者二进制 binary(base 2)来指定值，这只需要使用前缀 0x、0o 或者 0b：

```
> let hex = 0xFCAF;;
```

```
val hex : int = 64687

> let oct = 0o7771L;;

val oct : int64 = 4089L

> let bin = 0b00101010y;;

val bin : sbyte = 42y

> (hex, oct, bin);;

val it : int * int64 * sbyte = (64687, 4089L, 42)
```

如果你熟悉 IEEE 32 和 IEEE 64 标准，你也可以使用 hex, octal 或者 binary 来指定浮点数。F#将会把二进制值转为它所表示的浮点数。当使用不同的进制来表示浮点数时，使用 LF 后缀来表示 float 类型，而使用 lf 后缀来表示 float32 类型：

```
> 0x401E000000000000LF;;

val it : float = 7.5

> 0x00000000lf;;

val it : float32 = 0.0f
```

算术

你可以对数字基元进行标准的算术运算。Table 2-2 列出了所有支持的运算符。与大多数编程语言相似，整数除法是往下舍入，丢弃余数。

Table 2-2. 算术运算符

运算符	描述	示例	结果
+	加法	1 + 2	3
-	除法	1 - 2	-1
*	乘法	2 * 3	6
/	除法	8L / 3L	2L
**	乘方	2.0 ** 8.0	256.0
%	取模	7 % 3	1

Note:乘方运算符(**)只针对 float 和 float32。要对整数值乘方，你要么把它转换为浮点数值，要么使用 pown 函数。

默认情况下，算术运算符并不检查溢出，因此如果你超过一个整型值的允许范围之外，它就会溢出而变成负

值(类似地，如果数字太小以至于无法储存到整数类型中，那么除法就会导致一个正数产生)：

```
> 32767s + 1s;;

val it : int16 =  -32768s

>-32768s + -1s;;

val it : int16 = 32767s
```

如果整数溢出值得关注，你就应该考虑使用更大的类型或者使用待检查的算术，这将在 Chapter7 中讨论。

F#包含你想要的所有标准数学函数，完整的列表包含在 Table 2-3 中。

Table 2-3. 常见数学函数

函数	描述	示例	结果
abs	绝对值	abs -1.0	1.0
ceil	向上取最接近的整数	ceil 9.1	10.0
exp	e 的指定次幂	exp 1.0	2.718
floor	向下取最接近的整数	floor 9.9	9.0
sign	表示数字符号的值	sign -5	-1
log	自然对数	log 2.71828	1.0
log10	常用对数(以 10 为底)	log10 1000.0	3.0
sqrt	算术平方根	sqrt 4.0	2.0
cos	余弦值	cos 0.0	1.0
sin	正弦值	sin 0.0	0.0
tan	正切值	tan 1.0	1.557
pown	计算整数的乘方	pown 2L 10	1024L

转换函数

F#语言的原则之一是没有隐式转换。这意味着编译器不会在后台自动为你进行基元数据类型之间的转换，比如 int16 到 int64 的转换。这通过移除意外的转换来消除细微的错误。作为替代的方式，要转换基元类型，你必须使用 Table 2-4 列出的显式转换函数。所有的标准转换函数都接受其他基元类型的参数——包括 string 和 char。

Table 2-4. 数字基元转换函数

函数	描述	示例	结果
sbyte	转换为 sbyte	sbyte −5	−5y
byte	转换为 byte	byte "42"	42uy
int16	转换为 int16	int16 'a'	97s
uint16	转换为 unit16	uint16 5	5us
int32, int	转换为 int	int 2.5	2
uint32	转换为 uint32	uint32 0xFF	255
int64	转换为 int64	int64 −8	−8L
uint64	转换为 unit64	uint64 "0xFF"	255UL
float	转换为 float	float 3.1415M	3.1415
float32	转换为 float32	float32 8y	8.0f
decimal	转换为 decimal	decimal 1.23	1.23M

Note:虽然这些转换函数接受 string 参数，但是它们使用底层的 System.Convert 成员方法来分析字符串，也就意味着对于无效的输入会抛出 System.FormatException 异常。

BigInteger

如果你要处理比 2^{64} 更大的数据，F#还提供了 bigint 类型来表示任意大的类型。(bigint 类型只是 System.Numerics.BigInteger 类型的缩写.)

bigint 被整合到 F#语言中，并且使用后缀 I 来表示字面值。Example 2-1 定义了 bigint 大小的数据存储。

Example 2-1. 表示大整数的 BigInt 类型

```
> open System.Numerics

// 数据存储单元

let megabyte = 1024I * 1024I

let gigabyte = megabyte * 1024I

let terabyte = gigabyte * 1024I

let petabyte = terabyte * 1024I

let exabyte = petabyte * 1024I

let zettabyte = exabyte * 1024I;;
```

```
val megabyte : BigInteger = 1048576

val gigabyte : BigInteger = 1073741824

val terabyte : BigInteger = 1099511627776

val petabyte : BigInteger = 1125899906842624

val exabyte : BigInteger = 1152921504606846976

val zettabyte : BigInteger = 1180591620717411303424
```

Note：虽然 bigint 对性能进行了大量优化，它仍然比基元数字类型要慢很多。

位操作

基元整数类型支持位运算符来实现在二进制水平上操纵值。位运算符通常在从文件中读写二进制数据使用。参见 Table 2-5。

Table 2-5. 位运算符

运算符	描述	示例	结果
&&&	按位与	0b1111 &&& 0b0011	0b0011
|||	按位或	0xFF00 ||| 0x00FF	0xFFFF
^^^	按位异或	b0011 ^^^ 0b0101	0b0110
<<<	按位左移	0b0001 <<< 3	0b1000
>>>	按位右移	0b1000 >>> 3	0b0001

字符

.NET 平台是基于 Unicode 的，因此大多数文字都使用 2-byte 的 UTF-16 字符表示。要定义一个字符值，你可以把任何 Unicode 字符放到单引号中。字符也可以使用 Unicode 十六进制字符代码来指定。

下面的片段定义了定义了一个元音字符的列表并把一个使用十六进制值定义的字符 a 打印出来：

```
> let vowels = ['a'; 'e'; 'i'; 'o'; 'u'];;

val vowels : char list = ['a'; 'e'; 'i'; 'o'; 'u']

> printfn "Hex u0061 = '%c'" '\u0061';;

Hex u0061 = 'a'

val it : unit = ()
```

要表示特殊的控制字符，你需要使用 Table 2-6 中转义序列。转义序列有反斜线跟上一个特定的字符构成。

Table 2-6. 字符转义序列

字符	意义	字符	意义
\'	单引号	\b	Backspace
\"	双引号	\n	换行符
\\	反斜线	\r	回车
\t	Tab		

如果你想获取一个.NET 字符 Unicode 值对应的数字，可以把它传递给 Table 2-4 中列出的任意转换函数。又或者，你可以通过添加后缀 B 来获得表示一个字符的 byte 值。

```
> // 把 'C' 的值转换为 int

int 'C';;

val it : int = 67

> // 把 'C' 的值转换为 byte

'C'B;;

val it : byte = 67uy
```

字符串

字符串字面值通过把一系列的字符用双引号围起来定义，它可以跨越多行。要从一个字符串中获取一个字符，请使用索引器语法(.[ ])，并传入一个从零开始的字符索引：

```
> let password = "abracadabra";;

val password : string = "abracadabra"

> let multiline = "This string

takes up

multiple lines";;

val multiline : string = "This string

takes up

multiple lines"

> multiline.[0];;
```

```
val it : char = 'T'

> multiline.[1];;

val it : char = 'h'

> multiline.[2];;

val it : char = 'i'

> multiline.[3];;

val it : char = 's'
```

作为选择，F#支持三引号，这允许你把引号字符放进字符串中而不需要单独使用转义字符。例如：

```
let xmlFragment = """<Ship Name="Prometheus"></foo>"""
```

如果你想你想指定一个很长的字符串，你可以使用一个反斜线来把它分解成多行。如果一行的最后一个字符是一个反斜线(\)，那么这个字符就会去除所有的空格字符后从下一行继续：

```
> let longString = "abc-\
                    def-\
          ghi";;
val longString : string = "abc-def-ghi"
```

如果想的话，你可以在一个字符串中使用转义序列字符(例如\t 或者\\)。但这会让定义文件路径和注册键出现问题。你可以使用@符号来定义一个逐字字符串(verbatim string)，它将逐字解释引号之间的文字而不会编码任何转义序列字符：

```
> let normalString = "Normal.\n.\n.\t.\t.String";;
val normalString : string = "Normal.

.

.                    .                    .String
> let verbatimString = @"Verbatim.\n.\n.\t.\t.String";;
val verbatimString : string = "Verbatim.\n.\n.\t.\t.String"
```

与在字符后面加上后缀 B 将返回它的 byte 表示类似，在字符串末尾加上 B 将以一个 byte 数组的形式返回字符串中字符的 byte 表示(数组将在 Chapter4 中讨论)：

```
> let hello = "Hello"B;;
```

```
val hello : byte [] = [|72uy; 101uy; 108uy; 108uy; 111uy|]
```

布尔值

对于处理只能为 true 或 false 的值，F#提供了 bool 类型(System.Boolean)以及 Table 2-7 中列出的标准布尔运算符。

Table 2-7. 布尔运算符

运算符	描述	示例	结果
&&	与	true && false	false
||	或	true || false	true
not	非	not false	true

Example 2-2 创建了布尔函数的真值表并把它们打印出来。它定义了一个名叫 printTruthTable 的函数，这个函数接受一个名叫 f 的函数作为它的参数。这个函数被真值表的每个单元调用而它的结果被打印出来。然后，运算符&&和 || 被被传递给 printTruthTable 函数。

Example 2-2. 打印真值表

```
> // 打印给定函数的真值表
let printTruthTable f =
    printfn "        | true  | false |"
    printfn "    +-------+-------+"
    printfn " true  | %5b | %5b |" (f true true) (f true false)
    printfn " false | %5b | %5b |" (f false true) (f false false)
    printfn "    +-------+-------+"
    printfn ""
    ();;
val printTruthTable : f:(bool -> bool -> bool) -> unit
> printTruthTable (&&);;
        |   true | false   |
    +-------+-------+
true |   true |   false |
```

```
false | false    |    false |
      +-------+-------+
val it : unit = ()
> printTruthTable (||);;
      | true     | false |
      +-------+-------+
true | true    | true     |
false | true    | false    |
     +-------+-------+
val it : unit = ()
```

在计算布尔表达式时，F#使用短路求值，这意味着如果在计算两个表达式中的第一个之后结果便可确定，那么第二个值就不会被计算。例如：true || f()将被计算为 true，而不会计算函数 f，同样地，false && g()将被计算为 false，而不会计算函数 g。

比较和相等

你可以使用 Table 2-8 列出的标准大于、小于、等于运算符来比较数字值。所有的比较和相等运算都计算为一个布尔值；比较函数返回−1, 0, 或者 1，这取决于第一个参数是否比第二个参数小、等或者大。

Table 2-8. 比较运算符

运算符	描述	示例	结果
<	小于	1 < 2	True
<=	小于或等于	4.0 <= 4.0	True
>	大于	1.4e3 > 1.0e2	True
>=	大于或等于	0I >= 2I	False
=	等于	"abc" = "abc"	True
<>	不等于	'a' <> 'b'	True
compare	比较两个值	compare 31 31	0

Note：在.NET 中，相等是一个复杂的话题。包含值相等和引用相等。对于值类型，相等意味着两个值是相同的，比如 1 = 1。然而，对于引用类型，相等是通过重写 System.Object 方法 Equals 来定义的。要了解更多的信息，请查阅 Chapter5 中的”对象相等“章节。

函数

既然我们已经掌握了所有的 F#基元类型，让我们定义函数来操纵它们。

定义函数的方式与定义值相同，除了在函数名后面的东西，它们作为函数的参数。下面的代码定义了一个名叫 square 的函数，它接收一个整数 x，然后返回它的平方：

```
> let square x = x * x;;
val square : x:int -> int
> square 4;;
val it : int = 16
```

F#没有 return 关键字。因此当你定义一个函数时，最后计算的表达式就是函数的返回值。

让我们尝试另一个函数，它对函数的输入值加上 1：

```
> let addOne x = x + 1;;
val addOne : x:int -> int
```

FSI 的输出显示了这个函数的类型是 int -> int，它读作“一个接收一个整数然后返回一个整数的函数”。当你添加多个参数时，这个标识会变得有点复杂：

```
> let add x y = x + y;;
val add : x:int -> y:int -> int
```

严格说来，类型 int -> int -> int 被读作“一个接收一个返回一个接收一个整数的函数的整数然后返回一个整数的函数”。这时还不必担心刚才的”函数返回函数“的绕口令。现在你唯一要了解的是要调用函数，只须简单地提供它的参数并用空格分开：

```
> add 1 2;;
val it : int = 3
```

类型推理

因为 F#是静态类型的，所以对一个浮点数值调用你刚才创建的 add 方法将会导致一个编译错误：

```
> add 1.0 2.0;;
```

```
add 1.0 2.0;;
  add 1.0 2.0;;
  ----^^^
stdin(5,5): error FS0001: 此表达式应具有类型
    int
而此处具有类型
    float
```

你可能会想，为什么编译器认为这个函数只接收整数呢？运算符+也支持浮点数啊！

原因在于类型推理(type inference)。F#编译器并不需要你明确指出一个函数所有参数的类型。编译器会按照习惯推断它们的类型。

Note:注意不要把类型推理与动态类型相混淆。虽然 F#允许你在编写代码是省略类型，但这并不意味着类型检查不会在编译时执行。

虽然+运算符支持许多不同的类型，比如 byte、int 和 decimal，但是编译器在没有附加信息的情况下会默认把它处理为 int 类型。

接下来的 FSI 会话声明了一个函数，它将两个值相乘。就像使用+一样，这个函数被推断为处理整数的，因为没有通过其他使用信息：

```
> // 没有附加的使用信息下的推断
let mult x y = x * y;;
val mult : x:int -> y:int -> int
```

现在，如果我们有一个 FSI 片段，其中不仅定义了 mult 函数，并且在调用它时传入一个浮点数，那么这个函数的标识就会被推断为类型 float ->float -> float：

```
> // 操作中的类型推理
let mult x y = x * y
let result = mult 4.0 5.5;;
val mult : x:float -> y:float -> float
val result : float = 22.0
```

然而，你可以向 F#编译器提供一个类型注释或者暗示来说明这是什么类型。要加上一个类型注释，只须简单

地把函数的参数替换成这样的形式(ident : type)，这里 type 是你想让参数具有的类型。要明确我们的 add 函数的第一个参数是浮点数，只须简单地把函数像这样定义：

```
> let add (x : float) y = x + y;;
val add : x:float -> y:float -> float
```

注意这一点：由于你对值 x 添加了类型注释，函数的类型就转变成了 float -> float -> float。这是因为对+运算符来说，唯一一个接受 float 作为第一个参数的重载是 float -> float -> float，因此 F#编译器现在推断 y 的类型也是 float。

类型推理通过让编译器指出使用何种类型的方式显著地减少了代码的凌乱程度。然而，有时类型注释是必需的并且有时可以提高代码的可读性。

泛型函数

你可以编写一个处理任何类型的参数的函数——例如，一个恒等函数只是简单地返回它的输入：

```
> let ident x = x;;
val ident : x:'a -> 'a
> let ident x = x;;ident "a string";;
val it : string = "a string"
> ident 1234L;;
val it : int64 = 1234L
```

因为在 ident 函数中，类型推理系统并不需要值 x 的完整类型，所以它被视为泛型的。如果一个参数是泛型的，那就意味着这个参数可以是任何类型，比如 integer，string 或者 float。

泛型参数的名称可以是任何前加撇号的有效标识符，但是通常是从 a 开始的字母表中的字符。接下来的代码使用类型注释重新定义了 ident 函数，强制让 x 成为泛型的：

```
> let ident2 (x : 'a) = x;;
val ident2 : x:'a -> 'a
```

编写泛型代码对最大化代码的可重用性是至关重要。随着本书的进展，我们将继续用到类型推理和泛型函数，因此不必纠结于这里的细节。只需要注意无论你在什么时候看到'a，它都可能是 int，float，string 或者一个 user-defined 类型，等等。

作用域

在 F#中声明的每个值都有特定的作用域，它是这个值能被使用的位置的范围(更正式地讲，这被称为声明空间) 。默认情况下，值具有模块级的作用域，这意味着它们可以在声明之后的任何地方使用。然而，在一个函数的内部定义的值的作用域仅限于这个函数。因此一个函数可以使用函数外部任何已定义的值，但外部不能引用函数内部定义的值。

在下面的例子中，一个名为 moduleValue 的值被定义具有模块级作用域，并在一个函数中使用，相反，一个名为 functionValue 的值被定义为在一个函数的内部，当它在作用域之外使用时产生了一个错误：

```
> // 作用域
let moduleValue = 10
let functionA x =
    x + moduleValue;;
val moduleValue : int = 10
val functionA : x:int -> int
> // 错误情况
let functionB x =
    let functionValue = 20
    x + functionValue
// functionValue 不在作用域内
functionValue;;
  functionValue;;
  ^^^^^^^^^^^^^^
stdin(10,1): error FS0039: 未定义值或构造函数“functionValue”
```

值的作用域也许看上去并不比一个细节重要，但你应该小心的一个原因是 F#允许嵌套函数。你可以在函数体内部定义声明新的函数值。嵌套函数有权使用在更高的作用域声明的任何值，比如父函数或者模块，以及任何在嵌套函数内部声明的新值。

接下来的代码展示了嵌套函数的使用。注意函数 g 是如何能够使用它的父函数 f 的参数 fParam 的：

```
> // 嵌套函数
let moduleValue = 1
```

```
let f fParam =
    let g gParam = fParam + gParam + moduleValue
    let a = g 1
    let b = g 2
    a + b;;
val moduleValue : int = 1
val f : fParam:int -> int
```

看上去在函数内部定义函数只会导致迷惑，然而限制函数的作用域的功能是非常有用的。通过允许你把小的、特定的函数限定的它们被需要的位置可以减少对模块周围的污染。当我们开始 Chapter3 中的函数式编程风格后，这一点将变得更加明显。

一旦你开始使用嵌套函数，把所有的值直接放在作用域内也许会变得枯燥无味。如果你想要什么一个名为 x 的值，但这个值已经在更高的作用域中使用会发生什么呢？在 F#中，有两个相同名称的值并不会导致编译错误；相反，它只是导致屏蔽。当这一点发生时，两个值都存在于内存之中，只是没有访问前面声明的值的途径。相反，最后定义的那个值”胜利了”。

下面的代码定义了一个函数，它接收一个参数 x，然后又定义了几个新的值，名称也都为 x：

```
> open System.Numerics
// 把 byte 转换为 gigabyte
let bytesToGB x =
    let x = x / 1024I // B to KB
    let x = x / 1024I // KB to MB
    let x = x / 1024I // MB to GB
    x;;
val bytesToGB : x:BigInteger -> BigInteger
> let hardDriveSize = bytesToGB 268435456000I;;
val hardDriveSize : BigInteger = 250
```

前面的例子中，在每一个 let 绑定的后面，名为 x 的值被屏蔽并被一个新的值代替。看上去 x 的值好像在改变，但实际上它只是创建了一个新的值 x 并给它相同的名称罢了。下面展示了说明一个代码如何被编译的例子：

```
let bytesToGB x =
    let x_2 = x / 1024I // B to KB
    let x_3 = x_2 / 1024I // KB to MB
    let x_4 = x_3 / 1024I // MB to GB
    x_4
```

有意屏蔽值的技术对于在不依赖可变性的情况下给出更新值的假象来说是十分有用的。如果你真的要更新值 x，你需要借助可变性，它将在 Chapter4 中讨论。

控制流

在函数内部，你可以使用使用关键字 if 来对控制流进行分支。条件表达式必须是 bool 类型，如果它计算为 true，那么给出的代码将被执行，在接下来的片段中，这写代码是向控制台打印一条信息：

```
> // If 语句
let printGreeting shouldGreet greeting =
    if shouldGreet then
        printfn "%s" greeting;;
val printGreeting : shouldGreet:bool -> greeting:string -> unit
> printGreeting true "Hello!";;
Hello!
val it : unit = ()
> printGreeting false "Hello again!";;
val it : unit = ()
```

使用 if 表达式还可以实现更复杂的代码分支。

if 表达式就像你预想的那样工作：如果条件表达式计算为 true，就执行第一个部分的代码；否则就执行第二个部分的代码。然而，一些使得 F#与其他语言很不同的事情是 if 表达式会返回一个值。

例如，在下面的代码中，值 result 与 if 表达式的结果绑定到一起。因此如果 x % 2 = 0，那么 result 的值就会是"Yes it is" ;否则就是"No it is not"：

```
> // If 表达式
```

```
let isEven x =
    let result =
        if x % 2 = 0 then
            "Yes it is"
        else
            "No it is not"
    result;;
val isEven : x:int -> string
> isEven 5;;
val it : string = "No it is not"
```

你可以嵌套 if 表达式来构成更复杂的分支，但这很快就会变得难以维护：

```
let isWeekend day =
    if day = "Sunday" then
        true
    else
        if day = "Saturday" then
            true
        else
            false
```

F#也有一些语法糖来帮助你减少 if 表达式的深层嵌套，即使用 elif 关键字。有了它，你就可以把多个 if 表达式连接到一起，而不必使用嵌套：

```
let isWeekday day =
    if day = "Monday" then true
    elif day = "Tuesday" then true
    elif day = "Wednesday" then true
    elif day = "Thursday" then true
```

```
    elif day = "Friday" then true
    else false
```

因为 if 表达式的结果是一个值，所以每个 if 表达式分支都必须返回相同的类型：

```
> // error：if 表达式分支具有不同的类型
let x =
    if 1 > 2 then
        42
    else
        "a string";;
              "a string";;
    --------^^^^^^^^^^
stdin(5,9): error FS0001: 此表达式应具有类型
    int
而此处具有类型
    string
```

但如果你只要一个单一的 if 而没有对应的 else 会怎样呢？在这种情况下，分支必须返回 unit，这是 F#中的一种特殊类型，意味着实际上没有值。(又或者，你可以把 unit 想成其他编程语言中 void 的一种表示。)我们很快就会详细讨论 unit。

核心类型

前面我们已经讨论了.NET 平台上可用的基元类型，但仅凭它们还不足以创建意义丰富的 F#程序。F#库包含了几种核心类型，它们能允许你组织、操纵和处理数据。Table 2-9 列出了一些基本类型，你可以在你的 F#应用中使用它们。事实上，这些基本类型能够让编程以函数式风格进行，我们将在 Chapter3 中看到这一点。

Table 2-9. F#中的核心类型

标识	名称	描述	示例
unit	Unit	unit 值	()
int, float	具体类型	具体类型	42, 3.14
'a, 'b	泛型类型	泛型类型	

'a -> 'b	函数类型	返回一个值的函数	fun x -> x + 1
'a * 'b	元组类型	值的有序组合	("eggs", "ham")
'a list	列表类型	值的列表	[1; 2; 3], [1 .. 3]
'a option	选项类型	一个可选的值	Some(3), None

Unit

unit 类型是一个表示没有意义的值，在代码中用()表示：

```
> let x = ();;

val x : unit

> ();;

val it : unit = ()
```

没有相匹配的 else 的 if 表达式必须返回 unit，因为如果它们真的返回一个值，那如果是 else 分句被计算的话又会发生什么呢？同样地，在 F#中每个函数都必须返回一个值，因此如果函数在概念上并不返回任何东西——就像 printf，那么它就应该返回一个 unit 值。

如果你想返回 unit 的话，ignore 函数可以吞下一个函数的返回值。这通常在调用一个只需要它的副作用而想要忽略它的返回值的函数时使用：

```
> let square x = x * x;;

val square : x:int -> int

> ignore (square 4);;

val it : unit = ()
```

元组

元组(tuple,也读作 two-pull 或者 tuh-pull)是一个数据的有序集合，并且是一种将常见的数据块组合到一起的简单方式。例如，元组可以被用来存放计算的中间结果。F#的元组使用底层的 System.Tuple<_>类型，虽然在实践中你永远不会直接使用 Tuple<_>类。

要创建一个元组实例，请使用逗号把一组值分开，然后还可以用括号把它们括起来。元组的类型被描述为元组各元素类型的列表。在下面的例子中，dinner 是一个元组实例，而 string * string 是这个元组的类型：

```
> let dinner = ("green eggs", "ham");;

val dinner : string * string = ("green eggs", "ham")
```

元组可以包含任意数目、任意类型的值。事实上，你甚至可以把一个元组包含到其他元组之中！

下面的代码片段定义了两个元组。第一个，名为 zeros，定义了一个 0 的各种表示的元组。第二个，名为 nested，定义了一个嵌套元组。这个元组有三个元素，其中第二个和第三个元素是元组本身：

```
> let zeros = (0, 0L, 0I, 0.0);;
val zeros : int * int64 * BigInteger * float = (0, 0L, 0, 0.0)
> let nested = (1, (2.0, 3M), (4L, "5", '6'));;
val nested : int * (float * decimal) * (int64 * string * char) = (1, (2.0, 3M), (4L, "5", '6'))
```

要从一个二重元组中提取元素的值，你可以使用 fst 和 snd 函数。fst 返回元组的第一个元素，而 snd 返回第二个元素：

```
> let nameTuple = ("John", "Smith");;
val nameTuple : string * string = ("John", "Smith")
> fst nameTuple;;
val it : string = "John"
> snd nameTuple;;
val it : string = "Smith"
```

又或者，你可以简单地使用 let 绑定来从元组中提取值。如果你让 let 后面跟上多个由逗号分开的标识符，那么这些名称就会获得元组的值。

下面的例子创建了一个名为 snacks 的元组值。然后这个元组的值被提取到新的标识符 x，y 和 z 上：

```
> let snacks = ("Soda", "Cookies", "Candy");;
val snacks : string * string * string = ("Soda", "Cookies", "Candy")
> let x, y, z = snacks;;
val z : string = "Candy"
val y : string = "Cookies"
val x : string = "Soda"
> y, z;;
val it : string * string = ("Cookies", "Candy")
```

如果你想从元组中提取过多或者过少的值，你就会收到一个编译错误：

```
> let x, y = snacks;;

  let x, y = snacks;;

  -----------^^^^^^

stdin(7,12): error FS0001: 类型不匹配。应为

    string * string

而给定的是

    string * string * string

元组具有不同长度的 2 和 3
```

也可能把元组作为参数传递给你函数，就像任意值一样。同样地，一个函数可以返回一个元组。在下面的例子中，函数 tupledAdd 接受以元组的形式构成的两个参数 x 和 y。注意 add 与 tupledAdd 函数在类型标识上的不同：

```
> let add x y = x + y;;

val add : x:int -> y:int -> int

> let tupledAdd(x, y) = x + y;;

val tupledAdd : x:int * y:int -> int

> add 3 7;;

val it : int = 10

> tupledAdd(3, 7);;

val it : int = 10
```

当进入函数式编程风格之后，接收一个单一的元组作为参数的函数会有许多不同的意义；参见 Chapter3 中的”部分函数应用“一节。

列表

虽然元组把值组合到单一的实体中，但是列表(list)允许你把数据链接到一起来构成一个链。这样做允许你使用即将讨论的聚合函数来批量地处理列表中的元素。

定义一个列表的最简单的方式是用中括号将一系列由分号分开的值的集合括起来，虽然等会你会学到使用更强大的列表推导语法来声明列表。空列表(其中不包含任何元素)由 [] 来表示：

```
> // 声明列表
```

```
let vowels = ['a'; 'e'; 'i'; 'o'; 'u']

let emptyList = [];;

val vowels : char list = ['a'; 'e'; 'i'; 'o'; 'u']

val emptyList : 'a list = []
```

在我们的示例中，空列表具有类型 'a list，因为空列表可以时任意的类型。如果有更多基于用法的信息，类型推理系统就能够把它确定为更特定的类型。

不同于其他语言中的列表，F#的列表对访问和操纵它们是十分限制的。事实上，对于一个列表，你只能进行两个操作。(要了解这个限制如何能用作你的优势，请查阅 Chapter7。)

第一个基本的列表操作是 cons，由 cons 运算符(::)表示。它在一个列表的前面或者开头加入一个元素。下面的例子把值'y'放到列表 vowels 的开头

```
> // 使用 cons 运算符

let sometimes = 'y' :: vowels;;

val sometimes : char list = ['y'; 'a'; 'e'; 'i'; 'o'; 'u']
```

第二个基本的列表操作，称作 append，使用@运算符。append 把两个列表合并到一起。下面的例子把列表 odds 和列表 evens 合并到一起，产生一个新的列表：

```
> // 使用 append 运算符

let odds = [1; 3; 5; 7; 9]

let evens = [2; 4; 6; 8; 10];;

val odds : int list = [1; 3; 5; 7; 9]

val evens : int list = [2; 4; 6; 8; 10]

> odds @ evens;;

val it : int list = [1; 3; 5; 7; 9; 2; 4; 6; 8; 10]
```

列表范围

把列表的元素用分号隔开来声明列表很快就会变得冗长乏味，尤其是对大的列表。要声明一个有序数字值构成的列表，你可以使用列表范围(list range)语法。

第一个表达式指定了范围的下界，而第二个表达式指定其上界。随后的结果便是一个从下界到上界的的值构成的列表，每次增加 1：

```
> let x = [1 .. 10];;

val x : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
```

如果还提供了可选的步长，那么结果就是从下界到上界且每两个值相差一个步长的值构成的列表。注意步长值可以为负数：

```
> // 列表范围

let tens = [0 .. 10 .. 50]

let countDown = [5L .. -1L .. 0L];;

val tens : int list = [0; 10; 20; 30; 40; 50]

val countDown : int64 list = [5L; 4L; 3L; 2L; 1L; 0L]
```

列表推导

创建一个列表的最有效的方法是使用列表推导(list comprehensions),这是一个有用的语法，它允许你使用内联的 F#代码来生成列表。在最简单的水平上，列表推导是一些由方括号([ ])围起来的代码。列表推导的部分将被持续计算直到它结束，而列表将由通过 yield 关键字返回的元素组成(注意列表的创建完全在内存中进行；如果你发现你要创建的列表有几千个元素，请考虑使用 seq<_>代替,这将在 Chapter3 中讨论。)：

```
> // 简单的列表推导

let numbersNear x =

    [

        yield x - 1

        yield x

        yield x + 1

    ];;

val numbersNear : int -> int list

> numbersNear 3;;

val it : int list = [2; 3; 4]
```

大多数的 F#代码都可以存在于列表推导的内部，包括函数声明和 for 循环。下面的代码片段展示了一个列表推导，其中定义了一个函数 negate，返回从 1 到 10 的数字，并且把偶数设为负值：

```
> // 更复杂的列表推导
```

```
let x=
    [
        let negate x = -x
        for i in 1 .. 10 do
            if i % 2 = 0 then
                yield negate i
            else
             yield i
    ];;
val x : int list = [1;  −2; 3;  −4; 5;  −6; 7;  −8; 9;  −10]
```

当在列表推导中使用 for 循环时，你可以使用->代替 do yield 来简化代码。下面的两个代码片段是相同的：

```
// 生成前十个数的组合
let multiplesOf x = [ for i in 1 .. 10 do yield x * i ]
// 简化的列表推导
let multiplesOf2 x = [ for i in 1 .. 10 -> x * i ]
```

使用列表推导语法可以让你快速简明地生成数据的列表，它们接下来便可在你的代码中处理。Example 2-3 展示了如何使用列表推导来生成给定数字以内的所有质数。

这个例子通过循环遍历从 1 到给定最大值之间的所有数字。然后使用列表推导来生成每个数字的所有因数。再检查是否生成的因数列表只有两个元素，若是，则给出这个数字，因为它是一个质数。当然也有更高效的计算质数的方法，不过这个例子只是展示列表推导能表达什么。

Example 2-3. 使用列表推导来计算质数

```
>  // 用列表推导来生成质数
let primesUnder max =
    [
        for n in 1 .. max do
        let factorsOfN =
```

```
        [
            for i in 1 .. n do
                if n % i = 0 then
                    yield i
        ]
    // n 是质数当且仅当它的因数只有 1 和 n
    if List.length factorsOfN = 2 then
        yield n
];;
val primesUnder : max:int -> int list
> primesUnder 50;;
val it : int list = [2; 3; 5; 7; 11; 13; 17; 19; 23; 29; 31; 37; 41; 43; 47]
```

List 模块函数

F#库中的 List 模块包含了许多的方法来帮助你处理列表。Table 2-10 中列出的方法是你在 F#中与列表交互的基本方式。

Table 2-10. 常见的 List 模块函数

函数	描述
List.length	'a list -> int 返回列表的长度
List.head	'a list -> 'a 返回列表的第一个元素
List.tail	'a list -> 'a list 返回给定的列表，并除去第一个元素
List.exists	('a -> bool) -> 'a list -> bool 返回列表中是否有元素满足查找函数的条件
List.rev	'a list -> 'a list 倒序排列列表中的元素

List.tryfind	('a -> bool) -> 'a list -> 'a option 返回 Some(x)，这里 x 是对给定的函数返回 true 的第一个元素，没有就返回 None。 (Some 和 None 很快就会讨论。)
List.zip	'a list -> 'b list -> ('a * 'b) list 给定两个具有相同长度的列表，返回一个组合列表元素的元组构成的列表
List.filter	('a -> bool) -> 'a list -> 'a list 返回一个列表，里面只包含给定函数返回 true 的元素
List.partition	('a -> bool) -> 'a list -> ('a list * 'a list) 给定一个 predicate 函数和一个列表，返回两个新的列表；第一个列表是函数返回 true 的地方，而第二个是函数返回 false 的地方。

最初你也许并不清楚如何使用某些 List 模块函数，但很快你就能够仅仅通过查看它的类型注释来确定一个函数能做什么。

下面的例子展示了 List.partition 函数，把一个由 1 到 15 的数字构成的列表分解为两个新的列表：一个由 5 的倍数构成，而另一个包含剩下的元素。需要注意的是 List.partition 会返回一个元组，而在本例中值 multOf5 和 nonMultOf5 是同时绑定在一起的元组中的元素：

```
> // 使用 List.partition
let isMultipleOf5 x = (x % 5 = 0)
let multOf5, nonMultOf5 =     List.partition isMultipleOf5 [1 .. 15];;
val isMultipleOf5 : x:int -> bool
val nonMultOf5 : int list = [1; 2; 3; 4; 6; 7; 8; 9; 11; 12; 13; 14]
val multOf5 : int list = [5; 10; 15]
```

Note:那么什么是 List 呢？所有这些函数都是在 Microsoft.FSharp.Collections 命名空间的 List 模块中定义的。因为 Microsoft.FSharp.Collections 模块默认被导入，所以要使用这些方法，你只须指定 List 模块和函数名就行。

聚合函数

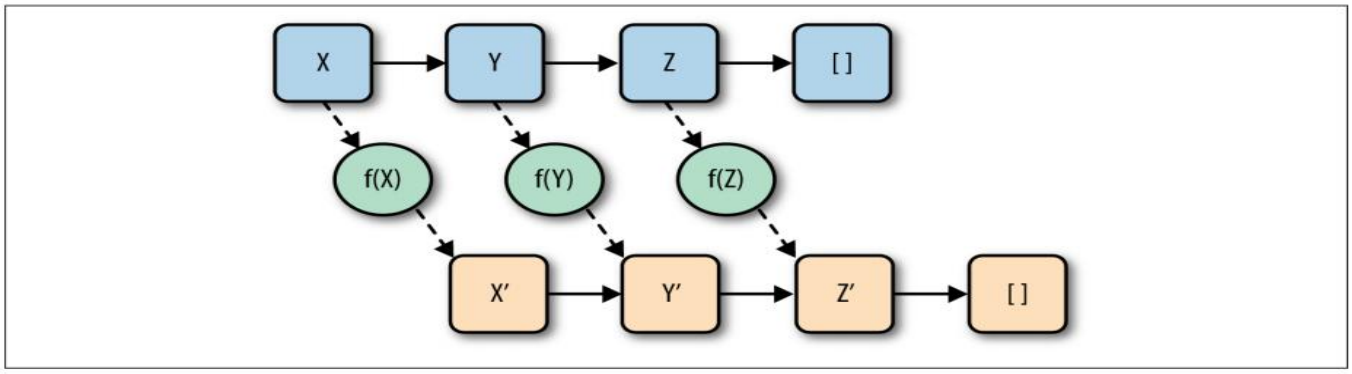
虽然列表提供了一个把数据链接到一起的方式，但事实上它们并没有特殊之处。列表的真正强大之处在于聚合函数，它们是一些强大的函数的集合，并可使用于任何值的集合。你将在讨论序列(Chapter3)和数组(Chapter4)时再次见到这些方法。

List.map

List.map 是一个映射操作，它创建一个基于给定函数的新列表。新列表中的每一个元素都是对原始列表中的元素执行该函数的结果。List.map 的类型是('a -> 'b) -> 'a list -> 'b list

你可以形象地表示一个函数 f 对列表[x; y; z]的映射，就像 Figure 2-1 中展示的那样。

Figure 2-1. 可视化的 List.map



Example 2-4 展示了平方函数对一个整数列表的映射结果。

Example 2-4. 使用 List.map 来对列表中的数字平方

```
> let square x = x * x;;

val square : x:int -> int

> List.map square [1 .. 10];;

val it : int list = [1; 4; 9; 16; 25; 36; 49; 64; 81; 100]
```

也许现在看上去并不有用，但 List.map 是 F#语言中最有用的函数之一。它提供了一个优雅的方式来让你进行转换操作，虽然被反复使用，却能大大地简化你编写的代码结构。

List.fold

这个函数非常强大，但是也比较复杂。它用于枚举列表中的所有成员，并对它们进行某种操作。

List 中最重要的有两个 fold(折叠)操作，第一个是 List.reduce。它的类型为('a -> 'a -> 'a) -> 'a list -> 'a。

List.reduce 函数枚举列表中的每个成员，并且产生一个累加值(accumulator value)，用于保存对列表进行处理的中间结果。传递的函数('a -> 'a -> 'a)用于将当前的累加值和枚举到的列表成员进行计算，产生新的累加值。当所有成员枚举完成时，给定函数返回的累加值作为整个 List.reduce 处理的返回值。

Example 2-5 展示了如何对使用 List.reduce 来把一个字符串列表用逗号分解开。函数 insertCommas 接收累加值和一个值，然后简单地返回一个新的字符串，其中包含了由一个逗号分开的累加值和该值。当被传递给 List.reduce 时，累加值的初始值就是列表的第一个元素，因此在对列表中的每个元素都处理之后的结果就是一个

单一的字符串，它包含了列表中的所有值，并用逗号将它们分开。

Example 2-5. 使用 List.reduce 来把一个字符串列表有逗号分解开

```
> let insertCommas (acc : string) item = acc + ", " + item;;

val insertCommas : acc:string -> item:string -> string

> List.reduce insertCommas ["Jack"; "Jill"; "Jim"; "Joe"; "Jane"];;

val it : string = "Jack, Jill, Jim, Joe, Jane"
```

下面的表显示了在处理每个列表元素后累加值是怎么变化的：

累加值	列表元素
“Jack” (第一个列表元素)	“Jill” (第二个列表元素)
“Jack, Jill”	“Jim”
“Jack, Jill, Jim”	“Joe”
“Jack, Jill, Jim, Joe”	“Jane”

虽然 reduce fold 是很有用的，但是它要求累加值的类型必须和列表成员的类型一致。但如果你想要一些更强大东西(比如把一个购物卡的列表换算成现金)又该怎么办呢？

如果你想使用自定义的累加类型，你可以使用 List.fold。fold 函数接收三个参数。第一，一个在提供了累加值和列表元素时返回一个新的累加值的函数。第二，一个初始的累加值。最后一个参数是将要 fold 的列表。Fold 的返回值是累加值的最终状态。正式的类型是：('acc -> 'b -> 'acc) -> 'acc -> 'b list -> 'acc

若要举出一个简单的例子，可以考虑 fold 一个整数列表到它们的和：

```
> let addAccToListItem acc i = acc + i;;

val addAccToListItem : acc:int -> i:int -> int

> List.fold addAccToListItem 0 [1; 2; 3];;

val it : int = 6
```

但再一次，fold 的累加值不必与列表元素的类型相同。Example 2-6 fold 一个字符串中的字符到一个元组，它计算每个元音字母出现的次数(As、Es、Is 的数值，等等)。fold 函数应用到列表中的每个字母。如果是元音字母，就返回更新后的累加值；否则就返回原来的累加值。

Example 2-6. 使用 List.fold 来数元音字母

```
> // 计算一个字符串中元音字母的数目
```

```
let countVowels (str : string) =
    let charList = List.ofSeq str
    let accFunc (As, Es, Is, Os, Us) letter =
        if letter = 'a' then (As + 1, Es, Is, Os, Us)
        elif letter = 'e' then (As, Es + 1, Is, Os, Us)
        elif letter = 'i' then (As, Es, Is + 1, Os, Us)
        elif letter = 'o' then (As, Es, Is, Os + 1, Us)
        elif letter = 'u' then (As, Es, Is, Os, Us + 1)
        else (As, Es, Is, Os, Us)
    List.fold accFunc (0, 0, 0, 0, 0) charList;;
val countVowels : str:string -> int * int * int * int * int
> countVowels "The quick brown fox jumps over the lazy dog";;
val it : int * int * int * int * int = (1, 3, 1, 4, 2)
```

从右到左 fold

List.reduce 和 List.fold 处理列表都是按照从左到右的顺序。也有可选的函数 List.reduceBack 和 List.foldBack 来按照从右到左的顺序处理列表。依赖于你想要做什么，以相反的顺序处理列表可能会对性能产生重大影响。要更深层次地了解列表处理时的性能影响，请参阅 Chapter7。

List.iter

最后一个聚合函数，List.iter，遍历列表中的每个元素并将一个函数应用于其上，其中函数是作为一个参数传进的。它的类型是：('a -> unit) -> 'a list -> unit

因为 List.iter 返回 unit，所以它主要用来计算给定方法的副作用(side effect)。副作用简单地意味着计算的函数有一些除返回值之外的影响；例如，printf 的副作用便是向控制台里打印文字，此外，它还返回 unit。

Example 2-7. 使用 List.iter 来遍历一个列表中的每个数字，并把它打印到控制台中。

Example 2-7. 使用 List.iter 来打印列表中的数字

```
> // 使用 List.iter
let printNumber x = printfn "Printing %d" x
List.iter printNumber [1 .. 5];;
```

```
Printing 1

Printing 2

Printing 3

Printing 4

Printing 5

val printNumber : x:int -> unit
```

选项

如果你想要表示一个不确定是否存在的值，最好的方式便是使用选项(option)类型。选项类型只有两个可能的值：Some('a)和 None。考虑一个把字符串当作 int 来进行分析的问题。如果字符串是正确地格式化的，那么函数就应该返回整数值，但如果字符串并不是正确格式化的又该如何呢？这就是你应该使用选项类型的最佳位置。

Note：在其他语言中，常见的习惯是使用 null 来表示值的缺少。然而，null 也被用来表示未初始化的值。这种二元性质会导致疑惑和 bug，如果你使用选项类型，就不会存在值到底代表什么的问题。

选项可被认为与 System.Nullable 类型类似，我们将在 Chapter4 中讨论它。

Example 2-8 定义了一个函数 isInteger，它尝试使用 Int32.TryParse 函数来解析一个整数。如果解析成功，函数将返回 Some(result)；否则就会返回 None。这确保了函数的使用者知道对哪些输入来说结果可能是不被定义的，因此返回 None。

Example 2-8. 选项类型，存储是否字符串成功解析到整数

```
> // 使用选项来返回一个值(或者 None)

open System

let isInteger str =

    let successful, result = Int32.TryParse(str)

    if successful then Some(result)

   else None;;

val isInteger : str:string -> int option

> isInteger "This is not an int";;
```

```
val it : int option = None

> isInteger "400";;

val it : int option = Some 400
```

要获取选项的值，你可以使用 Option.get。(如果对 None 使用 Option.get，一个异常将被抛出。)下面的代码片段定义了一个函数 containsNegativeNumbers，它对列表中所有的负数返回 Some(_)。然后，列表中的负数通过 Option.get 提取出来：

```
> // 使用 Option.get
let isLessThanZero x = (x < 0)
let containsNegativeNumbers intList =
    let filteredList = List.filter isLessThanZero intList
    if List.length filteredList > 0 then Some(filteredList)
    else None;;
val isLessThanZero : x:int -> bool
val containsNegativeNumbers : intlist:int list -> int list option
> let negativeNumbers = containsNegativeNumbers [6; 20; -8; 45; -5];;
val negativeNumbers : int list option = Some [-8; -5]
> Option.get negativeNumbers;;
val it : int list = [-8; -5]
```

Option 模块包含了其他一些有用的函数，它们列在 Table 2-11 中。

Table 2-11. 常见的 Option 模块方法

函数	类型	描述
Option.isSome	' a option -> bool	如果选项是 Some 则返回 true，否则返回 false
Option.isNone	'a option -> bool	如果选项是 Some 则返回 false，否则返回 true

Printf

向控制台中写数据是最简单的 I/O 操作，这是通过使用 printf 家族函数完成的。printf 有三个主要的版本：printf, printfn 以及 sprintf。

printf 接收输入并把它写到屏幕上，而 printf 还在后面加上一个换行标识：

```
> // printf 和 printfn

printf "Hello, "

printfn "World";;

Hello, World
```

已有的.NET System.Console 类也可以用来在屏幕上写文字，但是 printf 更适合函数式风格，因为它的参数是强类型的，这有助于类型推理。然而 System.Console.Read 仍被用来读取输入。向控制台中打印文字并不特别刺激，不过 printf 的包含的格式化和检查功能增添了许多强大之处。通过使用 Table 2-12 中列出的格式说明符，你也可以顺便访问数据：

```
> // 格式说明符

let mountain = "K2"

let height = 8611

let units = 'm';;

val mountain : string = "K2"

val height : int = 8611

val units : char = 'm'

> printfn "%s is %d%c high" mountain height units;;

K2 is 28251m high

val it : unit = ()
```

最好的一点，通过使用 F#的类型推理系统，当数据与给定的格式说明符不吻合时，编译器会给你错误提示：

```
printfn "An integer = %d" 1.23;;

  printfn "An integer = %d" 1.23;;

  -------------------------^^^^

stdin(1,27): error FS0001: 类型“float”与类型

byte,int16,int32,int64,sbyte,uint16,uint32,uint64,nativeint,unativeint 中的任何类型都不兼容，因为使用了

printf 样式的格式字符串
```

此外，由于 F#编译器知道对给定列表的格式说明需要什么类型，所以类型推理系统可以确定这些值的类型。例如，在下面的片段中，函数参数的类型根据用途推断出来：

```
> // 有关 printf 的类型推断

let inferParams x y z = printfn "x = %f, y = %s, z = %b" x y z;;

val inferParams : x:float -> y:string -> z:bool -> unit
```

Table 2-12. Printf 格式说明符

说明符	描述	示例	结果
%d, %i	打印整数	printf "%d" 5	5
%x, %o	以 16 进制(Hex)或者八进制(Octal)格式打印整数	printfn "%x" 255	ff
%s	打印字符串	"%s" "ABC"	ABC
%f	打印浮点数	printf "%f" 1.1M	1.100000
%c	打印字符	printf "%c" '\097'	a
%b	打印布尔值	printf "%b" false	false
%O	打印任意对象	printfn "%O" (1,2)	(1, 2)
%A	打印任意事物	printf "%A" (1, [])	(1, [])

Note:%O 格式说明符装入对象并调用虚方法 Object.ToString。%A 格式说明符也同样工作，只是它在调用 Object.ToString 前会检查在[<StructuredFormatDisplay>] 中是否有特殊的打印指示。作为底线，你应尽可能地使用%A 代替%O。

sprintf 用于获取打印成字符串的结果：

```
> let location = "World";;

 val location : string = "World"

> sprintf "Hello, %s" location;;

val it : string = "Hello, World"
```

组织 F#代码

到现在你可能想要把我们在 FSI 中编写的代码转换成实际的程序了。但事实上，到目前为止你看到的每一个片段都是一个完整的程序(虽然不得不承认，这不是你想象的程序类型)。

不过别担心。在 Chapter11 中你将对 F#的真实世界有更多了解。现在，我们需要专注于所有 F#应用的组织构建块：模块(module)和命名空间(namespace)。

模块

到目前为止我们编写的代码都属于一个模块。默认情况下，F#会把你所有的代码都放到一个与代码文件同名的匿名模块中，只是首字母大写。因此，如果你有一个名为 value1 的值，并且你的代码包含在 file1.fs 中，那么你就可以使用完整路径名称：File1.value1 来引用它。

创建模块

你可以在资源文件顶部使用 module 关键字来明确地命名你的代码模块。在该点之后，所有的值、函数或者类型定义都会属于这个模块：

```
module Alpha

// 要在模块外引用这个值

// 使用： Alpha.x

let x = 1
```

嵌套模块

文件中也可包含嵌套模块。要声明一个嵌套模块，请使用 module 关键字，跟上你的模块名称以及一个等号标记(=)。嵌套模块必须缩进以便与顶级模块相区别：

```
module Utilities

   module ConversionUtils =

   // Utilities.ConversionUtils.intToString

      let intToString (x : int) = x. ToString()

   module ConvertBase =

   // Utilities.ConversionUtils.ConvertBase.convertToHex

      let convertToHex x = sprintf "%x" x

   // Utilities.ConversionUtils.ConvertBase.convertToOct

      let convertToOct x = sprintf "%o" x

   module DataTypes =

   // Utilities.DataTypes.Point

      type Point = Point of float * float * float
```

命名空间

模块的替代选择是命名空间。命名空间是类似于模块的组织代码的单位，只是命名空间不可以包含值，只能有类型声明。而且，命名空间不可以像模块那样嵌套。相反，你可以简单地在同一文件中添加多个命名空间。

Example 2-9 在两个命名空间中定义了几个类型

Example 2-9. 命名空间

```
namespace PlayingCards

// PlayingCards.Suit

type Suit =

    | Spade

    | Club

    | Diamond

    | Heart

// PlayingCards.PlayingCard

type PlayingCard =

    | Ace of Suit

    | King of Suit

    | Queen of Suit

    | Jack of Suit

    | ValueCard of int * Suit

namespace PlayingCards.Poker

// PlayingCards.Poker.PokerPlayer

type PokerPlayer = { Name : string; Money : int; Position : int }
```

看上去在 F#中既有命名空间又有模块似乎有点奇怪。就像你目前看到的那样，模块是快速项目和快速地探索解决方案的最佳选择。而另一方面，命名空间更适合使用面向对象方式的大型项目。随着你见到更多的 F#代码，何时使用模块来代替命名空间的细微差别将逐渐明朗。当你开始时，只须把所有的东西都放在一个模块中，而不必担心命名空间。

程序起点

命名空间和模块是用来组织 F#资源文件中创建的代码的方式。但程序到底会从哪里开始执行呢？

在 F#中,程序将从最后的那个代码文件的顶部开始执行，它必须是一个模块。考虑这个简单的只有一个代码文件的 F#程序：

```
// Program.fs
let numbers = [1 .. 10]
let square x = x * x
let squaredNumbers = List. map square numbers
printfn "SquaredNumbers = %A" squaredNumbers
open System
printfn "(press any key to continue)"
Console. ReadKey(true)
```

现在在 Visual Studio 中打开这个项目，然后添加一个新的、空白的 F#代码文件。当你按下 F5 来运行你的程序时，什么事情都不会发生。这是因为向项目中添加的最新文件——它是空白的——被最后加入，自然在程序启动时运行(也可以说，这是因为在解决方案资源管理器中 Program.fs 是在 File1.fs 之前加入的)。这个特性很适合快速成型以及节省你几次按键，不过在更大一点的项目值最好还是明确地定义程序起点(entry point)。

对于更一般的程序启动语义，你可以使用 [<EntryPoint>] 特性来定义一个 main 方法。要合乎标准，你的 main 方法必须满足下面的要求：

- 是定义在项目中最后编译的那个文件中的最后一个函数。这确保对 F#程序在什么地方启动没有歧义。
- 接收单一的 string 数组参数，它是你的程序的特殊对象(argument)。(数组将在 Chapter4 中讨论)
- 返回一个整数，这是你的程序的退出代码。

要让 main 方法明确，你可以这样编写前面的应用：

```
// Program.fs
open System [<EntryPoint>]
let main (args : string[] ) =
    let numbers = [1 .. 10]
    let square x = x * x
    let squaredNumbers = List. map square numbers
    printfn "SquaredNumbers = %A" squaredNumbers
```

```
printfn "(press any key to continue)"

Console. ReadKey(true) |> ignore

// Return 0

0
```

现在你已经具备了编写简单的 F#程序所需的所有工具。在下一章在，你将学习如何使用函数式风格编来让你编写更强大的 F#应用，预计你的旅程目标将是成为一个 level 9 的 F#大师。