Chapter 3 函数式编程

有了一些基础，你可以开始使用特定的风格来测试 F#。这一章致力于研究 F#的主要范式：函数式编程(functional programming)。简而言之，函数式编程更关注于代码中的声明。在命令式编程(我们将在 Chapter4 在讨论)中，你把时间花在列出完成任务的特定步骤上。在函数式编程中，你需要的是指定做什么，而不是怎么做。即使函数式编程并不是最好的，但它的结果是程序会变得更清晰，一些问题——比如并发和并行编程——会变得十分容易。

函数式编程本身并不打算重复命令式或者面向对象编程；相反；它只是提供了不同的使用方式，以便在某些应用中你可以更方便地编程。

对于一门被认为是函数式的语言，它通常需要支持一些关键特性：

- 不可变数据
- 支持复合函数
- 函数可以像数据那样对待
- 惰性求值
- 模式匹配

我们将在本章中逐一介绍这些新的函数式的概念以及它们提供了什么。到本章末尾，你将能够编写纯粹的函数式代码，并体会到声明式编程的简洁和优雅。关于对函数式编程更深层次的探讨，比如尾递归和闭包，将在 Chapter7 中讨论。

理解函数

函数式编程的核心是把代码考虑成数学函数的形式。考虑两个函数 f 和 g：

```
f(x) = x^2 + x g(x) = x + 1
```

由此可以得到：

```
f(2) = (2)^2 + (2) g(2) = (2) + 1
```

如果你想复合这两个函数，或者把它们放在一起，就会得到：

```
f g (2) = f(g(2))
```

= (g(2))^2 + (g(2))

= (2+1)^2 + (2+1)

= 12

在 F#中你不必成为一个数学家，但许多函数式编程的基础都根植于数学。例如，在前面的片段中，并没有明确指定返回值的类型。f(x)接收一个整数还是浮点数呢？这个数学符号并不关心数据类型或者返回值。等效的 F#代码是这样的：

```
let f x = x ** 2.0 + x

let g x = x + 1.0
```

事实上 F#与数学符号类似并非巧合。函数式编程在基础上便是以一种抽象的方式来思考计算过程的——再一次，计算什么而不是怎么计算。

你甚至可以把整个程序都考虑成一个函数，它们的输入就是鼠标和键盘的状态，而输出便是程序的退出代码。当你开始以这种方式看待程序时，一些在通常的编程中十分复杂的事情就会变得很简单。

第一，如果你把程序考虑成一系列的函数，那么你就不需要花费所有的时间来详细地介绍要完成任务的每一个步骤。函数只是简单地接受它们的输入，然后生产一个输出就行。第二，算法是通过函数的形式来表达的，而不是类或者对象，因此使用函数式编程可以很方便地把这些概念翻译成代码。

你将在本章中看到一些函数式编程如何简化复杂的代码的例子，不过首先你需要以函数的方式来思考问题。要做到这一点，你需要抛弃一些从已有的命令式语言中形成的观念。在下一节中，我们将介绍不可变性，作为值的函数以及函数复合，来展示如何以函数式风格来开始编程。

不可变性

你也许已经注意到以前我从来没有使用变量(variable)这个词，而是把所有东西都称作值(value)。原因是在函数式编程中，你声明的事物的名称默认情况下是不可变的(immutable)，这意味着它们不能够被改变。

如果一个函数以某种方式改变了程序的状态——比如写入一个文件或者改变内存中的一个全局变量——这就被认为是一个副作用(side effect)。例如，调用 printfn 函数会返回 unit，但会有副作用，即向屏幕上打印文字。类似地，如果一个函数更新了内存中的值，这也是一个副作用——函数除返回值之外的一些多余效应。

副作用并不总是坏的，但意料之外的副作用是许多 bug 的根源。如果不小心一个函数的副作用，即使是最优秀的程序员也可能犯错误。不可变的值能够帮助你编写更安全的代码，因为你不可能弄错你不能够改变的东西。

如果你习惯于命令式编程语言，那么拥有变量却不能够改变也许看上去像一个负担。但不可变性提供了一些显著

的好处。考虑 Example 3-1 中的两个函数。它们都只是计算一个列表中数字的平方和，其中一个使用命令式风格的可变数据而另一个使用函数式风格。命令式风格利用了一个可变的变量，也就意味着在 imperativeSum 的执行过程中值被完全地改变了。

Example 3-1. 使用命令式和函数式风格来计算列表数字的平方和。

```
let square x = x * x
let imperativeSum numbers =
    let mutable total = 0
    for i in numbers do
        let x = square i
        total <- total + x
    total
let functionalSum numbers =
    numbers
    |> Seq. map square
    |> Seq. sum
```

你可能首先注意到的是第二个函数式的例子更简短。它从列表 numbers 开始，对每个数平方，然后把它们加在一起。虽然你对 F#的语法并不熟悉，但这段代码更具声明式风格并且直接写出了你想要发生的事情。

命令式的版本，虽然能更容易地在你的头脑中通过，但却需要你通读这段代码，才能知道会发生什么。

你也许认为即使有一点冗长，但命令式的例子仍比函数式的更好，因为它更熟悉。至少在现在，请抵制住这冲动，因为在函数风格中还有一些解决问题的精妙优点等待你学习。

例如，如果你想要以并行的方式运行命令式版本，那么你就必须完全重写代码。因为你太专注于如何指定程序的运行，所以你需要重新声明这个程序如何在并行状态下工作。

然而，函数式的版本并没有指定如何做这件事情，所以你可以轻松地用在并行状态下的实现来代替 map 和 sum 函数。你将在 Chapter11 中看到用 F#使得并行编程多么容易。

如果函数式编程语言不允许副作用，那么它们就被认为是纯粹的。在这种观点看来，F#是不纯粹的，因为它允许你在命令式风格的编程中改变变量的值。

函数值

在许多别的编程语言中，函数和数据被认为是两种完全不同的东西。然而，在函数式编程语言中，函数就像其他形式的数据一样被对待，例如，函数可以作为参数传递给其他的函数。此外，函数还可以创建并返回一个新的函数！接收或者返回其他函数作为输入或者输出的函数被称为高阶函数(higher-order functions)，这是常见函数式编程中的关键内容。

这个功能可以让你抽象化和重复利用代码中的算法。在 Chapter2 的 List.iter, List.map 和 List.fold 部分，你已经看到了这样的一些例子。

Example 3-2 定义了一个函数 negate，它对整数取负。当这个函数被作为一个参数传递给 List.map 时，这个函数就被应用到整个列表，对每个元素取负。

Example 3-2. 高阶函数示例

```
> let negate x = -x;;
val negate : x:int -> int
> List.map negate [1 .. 10];;
val it : int list = [-1; -2; -3; -4; -5; -6; -7; -8; -9; -10]
```

使用函数值是非常方便的，但结果是你最终会编写许多简单的函数，它们本身并不具有许多的值。例如，在 Example 3-2 中我们的 negate 函数除了在对一个列表取负之外可能永远不会在程序的其他地方使用。比起命名所有的作为参数的小型函数，你可以使用匿名函数(anonymous function)，也被称为 lambda 表达式，来创建一个内联函数。要创建一个 lambda 表达式，你只须简单地使用 fun 关键字，后面跟上函数的参数以及一个箭头符号->。

下面的片段创建了一个 lambda 表达式并且把值 5 作为参数传入。当函数执行时，参数 x 会增加 3，而结果当然就是 8：

```
> (fun x -> x + 3) 5;;
val it : int = 8
```

我们可以使用 lambda 表达式来重写 negate 函数，它接收参数 i：

```
> List.map (fun i -> -i) [1 .. 10];;
val it : int list = [-1; -2; -3; -4; -5; -6; -7; -8; -9; -10]
```

Note:注意让 lambda 表达式保存简洁 。随着它们变大，对它们进行调试将变得更加困难。这一点在你发现自己在代码中到处复制和传递 lambda 时尤为明显。

部分函数应用

在实践中的另一个高阶函数的例子是部分函数应用(partial function application)。部分函数应用是指定一个函数的部分参数然后生产出一个那些参数被填充的新的函数的功能。例如 f(x, y, z)可以部分地应用 x 和 y，也就变成了新的函数 f'(z)。

让我们看一个实际的例子，它使用.NET 类库来向文件中写入文本：

```
> // 向文件中写入文本
open System.IO
let appendFile (fileName : string) (text : string) =
    use file = new StreamWriter(fileName, true)
    file.WriteLine(text)
    file.Close();;
val appendFile : string -> string -> unit
> appendFile @"D:\Log.txt" "Processing Event X...";;
val it : unit = ()
```

函数 appendFile 看上去十分简单，但如果你想要重复地写入相同的 log 文件呢？你将不得不保留你的 log 文件的路径并每次都把它作为第一个参数传递进去。然而，创建一个新版本的 appendFile 函数，其第一个参数被我们的逐字字符串@"D:\Log.txt"填充将会更好。

你可以通过部分地对 appendFile 应用第一个参数，这将生产出一个新的函数，它只接收一个参数，即需要记录的信息。

```
> // 部分调用 appendFile 来创建一个新的函数
let appendLogFile = appendFile @"D:\Log.txt";;
val appendLogFile : (string -> unit)
> // 向'D:\Log.txt'中写入文本
appendFile @"D:\Log.txt" "Processing Event X...";;
val it : unit = ()
```

部分函数应用是在函数的类型中参数之间有箭头的原因。函数 appendFile 的具有的类型是：

string -> string -> unit

因此在第一个参数被传入之后，结果便是一个接收 string 并返回 unit 的函数，或者说是 string -> unit。要明白在后台会发生什么，让我来先介绍一下柯里化(currying)。把一个接收 n 个参数的函数转化为 n 个只接收一个参数的功能就被称为柯里化。F#编译器会对函数进行柯里化来使得部分函数应用得以实现。

string * string -> int 和 string -> string -> int 之间有很大的不同。这两个函数类型都接收两个 string 参数并返回一个 int。然而，其中一个只接收元组形式(string * string)的参数。这意味着所有的参数必须同时指定。而另一个函数的参数是柯里化的(string -> string -> unit)，因此只应用一个参数就会产生新的函数值。

柯里化和部分函数应用也许看上去并不特别强大，但它可以明显提高你的代码的优雅性。考虑 printf 函数，它接收一个格式化的字符串作为参数，后面跟上要填充到格式化字符串中的值。如果你只用"%d"给出了第一个参数，那么结果便是一个部分应用的函数，它接收一个整数并把它打印到屏幕上。

下面的例子展示了如何传递一个部分应用的 printf 版本来避免 lambda 表达式的使用：

```
> // 不使用部分应用
List.iter (fun i -> printfn "%d" i) [1 .. 3];;
1
2
3
val it : unit = ()
> // 对 printfn 部分应用"%d"参数，这将返回一个新的函数
//其类型是 int -> unit 而不是 string -> int -> unit。
List.iter (printfn "%d") [1 .. 3];;
1
2
3
val it : unit = ()
```

在本章的后续部分你将看到如何有效地利用部分函数应用，那时我们将会讨论函数复合以及 pipe-forward 运算符。

Note:虽然部分函数应用能够使代码更简洁，但它们也可能会使得代码难以调试。因此不要滥用柯里化，否则你就会冒着使程序变得比它们本需要的更加复杂的风险。

返回函数的函数

既然函数式编程把函数当成数据一样看待，那么函数返回其他函数值也就不无可能。当你考虑局部值的生命周期时，这一点会产生一些有趣的情形。

Example 3-3 定义了函数 generatePowerOfFunc，它返回对给定数字进行乘方的一个函数。于是两个函数 powerOfTwo 和 powerOfThree 被创建，它们分别进行平方和立方。

Example 3-3. 返回函数的函数

```
> // 返回函数的函数
let generatePowerOfFunc baseValue =
    (fun exponent -> baseValue ** exponent);;
val generatePowerOfFunc : float -> float -> float
> let powerOfTwo = generatePowerOfFunc 2.0;;
val powerOfTwo : (float -> float)
> powerOfTwo 8.0;;
val it : float = 256.0
> let powerOfThree = generatePowerOfFunc 3.0;;
val powerOfThree : (float -> float)
> powerOfThree 2.0;;
val it : float = 9.0
```

如果你仔细观察我们的 generatePowerOfFunc 函数，你就会注意到它的参数 baseValue 在它返回的两个 lambdas 中均被使用。但是当你在后面调用值 powerOfTwo 和 powerOfThree later 时，如果 baseValue 不是函数的参数，那它又是从哪里来的呢？在 generatePowerOfFunc 使用值 2.0 作为参数被初始调用时，你可能会想 2.0 被存储到哪里呢。这里有一个小小的魔术，它被称作闭包(closure)。现在不要把自己纠结于这个概念或者它如何工作。 只需要知道如果一个值包含在作用域内，它就可以被使用并可能被函数返回。在 Chapter 7 中，我们将深入讨论闭包，并看到使用这个 F#编译器表演的魔术你可以做什么事情。

递归函数

如果一个函数需要调用本身，那么它就被称作递归函数，它们在以函数式风格编程时非常有用，就像你很快就会看到那样。

要定义一个递归函数，你只需添加关键字 rec 就行。下面的片段定义了一个函数来计算阶乘(一个整数的阶乘是不大于它的所有正整数的乘积，比如 4 的阶乘就是 4 * 3 * 2 * 1)：

```
> // 定义一个递归函数
let rec factorial x =
   if x <= 1 then
      1
   else
      x * factorial (x - 1);;
val factorial : int -> int
> factorial 5;;
val it : int = 120
```

关键字 rec 的使用可能十分突出，因为其他的一些语言不需要你显式地指明递归函数。 rec 关键字的真正目的是通知类型推理系统允许函数被用作类型推断过程的一部分。 rec 允许你在类型推断系统判定函数的类型之前使用调用它。

将递归函数和高阶函数组合起来使用，你就可以轻松地仿造命令式语言中的循环结构，而不需要改变值。下面的例子创建了一个常见的 for 和 while 循环的函数式版本。注意在这个例子的 for 循环中，被更新的计数器只是作为一个参数传递给递归调用：

```
> // 函数式的 for 循环
let rec forLoop body times =
    if times <= 0 then
       ()
    else
       body()
       forLoop body (times - 1)
// 函数式的 while 循环
```

```
let rec whileLoop predicate body =
    if predicate() then
        body()
        whileLoop predicate body
    else
        ();;
val forLoop : body:(unit -> unit) -> times:int -> unit
val whileLoop : predicate:(unit -> bool) -> body:(unit -> unit) -> unit
> forLoop (fun () -> printfn "Looping...") 3;;
Looping...
Looping...
Looping...
val it : unit = ()
> // 一个标准的工作周...
open System
whileLoop
    (fun () -> DateTime.Now.DayOfWeek <> DayOfWeek.Saturday)
    (fun () -> printfn "I wish it were the weekend...");;
I wish it were the weekend...
I wish it were the weekend...
I wish it were the weekend...
    * * * 这将持续好几天 * * *
val it : unit = ()
```

事实上，在大多数函数式编程语言中，递归是实现循环的唯一方式。(记住，如果你不能改变变量，你就不能够增加循环计数器)。遵循严格的函数式风格的 F#程序更喜欢使用递归函数来代替循环。不过你应该始终保持代码尽可能的清晰和简洁。

相互递归

两个互相调用的函数被称为相互递归(mutually recursive)，这给 F#的类型推理系统带来了独特的挑战。要确定第一个函数的类型，你需要知道第二个函数的类型，反之亦然。

在 Example 3-4 中，相互递归函数编译失败，因为在处理 isOdd 函数(判断是否为奇数)时，isEven 函数(判断是否为偶数)还没有被定义。

Example 3-4. 相互递归函数

```
> // Error: 没有 isEven 不能定义 isOdd，反之亦然。
let isOdd x =
    if x = 0 then false
    elif x = 1 then true
    else isEven(x - 1)
let isEven x =
    if x = 0 then true
    elif x = 1 then false
    else isOdd(x - 1);;
         else isEven(x - 1)
   ---------^^^^^^
stdin(5,10): error FS0039: 未定义值或构造函数 “isEven”
```

为了定义相互递归函数，你必须使用 and 关键字来把它们合到一起，这将通知 F#编译器同时对这两个函数就行类型推断：

```
> // 相互递归函数，使用 rec 和 and。
let rec isOdd x =
    if x = 0 then false
    elif x = 1 then true
    else isEven(x - 1)
and isEven x =
    if x = 0 then true
```

```
    elif x = 1 then false
    else isOdd(x - 1);;
val isOdd : x:int -> bool
val isEven : x:int -> bool
> isOdd 314;;
val it : bool = false
> isEven 314;;
val it : bool = true
```

符号化运算符

试想如果你不能够写 1 + 2 而每次都要写 add 1 2，那么编程将会变得多么困难。幸运的是，F#不仅拥有内建的处理加法和乘法等事情的符号化运算符，而且还允许你定义自己的符号化运算符。这允许你以更清晰和优雅的方式来编写代码。

Note：不要把符号化函数想成一种运算符重载的形式，相反，它们就是函数，只是名称由符号构成罢了。

一个符号化运算符可以由!%&*+-./<=>@^|?这些符号的任意序列构成。下面的代码定义了一个新的函数 !，它用来计算数字的阶乘：

```
> // 阶乘
let rec (!) x =
    if x <= 1 then 1
    else x * !(x - 1);;
val ( ! ) : x:int -> int
> !5;;
val it : int = 120
```

默认情况下，如果符号化运算符不止一个参数，那么它就使用中缀表示法。这意味着第一个参数处于符号之前，这是你应用符号化函数的最常见的形式。下面的例子定义===，它把一个字符串与一个常规表达式相比较：

```
> // 定义===来把字符串与常规表达式相比较
open System.Text.RegularExpressions;;
let (===) str (regex : string) =
```

```
    Regex.Match(str, regex).Success;;

val ( === ) : str:string -> regex:string -> bool

> "The quick brown fox" === "The (.*) fox";;

val it : bool = true
```

除了允许你以更接近数学的方式命名函数之外，符号化运算符也可以传递给高阶函数，只要你在符号外加上括号。例如，如果你想计算列表中元素的总和或者乘积，你可以写：

```
> // 使用(+)符号函数来求和

List.fold (+) 0 [1 .. 10];;

val it : int = 55

> // 使用(*)符号函数来计算列表在所有元素的乘积

List.fold (*) 1 [1 .. 10];;

val it : int = 3628800

> let minus = (-);;

val minus : (int -> int -> int)

> List.fold minus 10 [3; 3; 3];;

val it : int = 1
```

函数复合

一旦你能够有效地控制函数，你就可以尝试把它们组合成更大，更有用的函数。这被称为函数复合(function composition)，它是函数式编程的另一大原则。

在我们开始学习如何组合函数之前，让我们先看看它能解决的问题。这里是一个不要做什么的例子：把所有的东西都放到一个巨大的函数中。考虑获取磁盘上给定文件夹大小的代码。我已经添加了类型注释来帮助弄清楚返回值：

```
open System

open System. IO

let sizeOfFolder folder =

    // 获取该路径下的所有文件

    let filesInFolder : string [] =
```

```
    Directory. GetFiles(
        folder, "*.*" ,
        SearchOption. AllDirectories)
// 把这些文件与它们对应的 FileInfo 对象相对应
let fileInfos : FileInfo [] =
    Array. map
        (fun (file : string) -> new FileInfo(file))
        filesInFolder
// 把这些 fileInfo 对象与文件的大小相对应
let fileSizes : int64 [] =
    Array. map
        (fun (info : FileInfo) -> info. Length)
        fileInfos
// 总的文件大小
let totalSize = Array. sum fileSizes
// 返回文件的总大小
totalSize
```

这段代码主要有三个问题：

- 类型推理系统无法自动判断正确的类型，因此我们必须为每个 lambda 表达式中的参数通过类型注释。这是因为类型推断以从上到下，由左到右的顺序处理代码，因此它会在看到数组参数的类型之前看到传递给 Array.map 的 lambda。(因此 lambda 的参数类型就不知道了)。
- 每个计算的结果都只是作为下一步计算的一个参数，因此函数里到处都是不必要的 let 语句。
- 它太难看了。它花了太多时间来解释接下来要干什么，这本不应该做的。

函数复合就是关于处理这样的代码的，把它分解成更小的函数，然后把它们复合起来获得最终的结果。

前面的例子一直把一个函数的计算结果传给下一个。在数学上，如果你想要把 f(x)传给 g(x)的结果，我们会写：g(f(x))。我们本可以嵌套所有的中间结果来避免这些 let 绑定的使用，但那将非常难以阅读，就像你在这里看到的那样：

```
let uglySizeOfFolder folder =
   Array. sum
      (Array. map
         (fun (info : FileInfo) -> info. Length)
         (Array. map
            (fun file -> new FileInfo(file))
            (Directory. GetFiles(
               folder, "*.*" ,
               SearchOption. AllDirectories))))
```

Pipe-forward 运算符

幸运的是，F#使用 pipe-forward 运算符，|> ，来简洁地解决了把中间结果传给下一个函数的问题。它的定义如下：

```
let (|>) x f = f x
```

pipe-forward 运算符允许你重新排列函数的参数以便把最后一个参数放到最前面。然而鉴于 List.iter 的最后一个参数是将要迭代的列表，使用 pipe-forward 运算符你可以把列表用管道(pipe)传到 List.iter 中，所以你可以指定哪一个列表先被迭代：

```
> [1 .. 3] |> List.iter (printfn "%d");;
val it : unit = ()
```

Note:严格来说，pipe-forward 运算符和它的姐妹函数 pipe-backward 运算符，实际上并不是复合函数。相反，它们只是参与函数应用罢了。

pipe-forward 运算符的好处是你可以不断地应用连锁函数。因此一个函数的结果就被用管道传给了下一个。我们可以像下面这样重写 sizeOfFolder 函数(注意 Directory.GetFiles 的参数是一个元组，因此不能被部分应用；所有的参数必须同时指定)：

```
let sizeOfFolderPiped folder =
   let getFiles path =
      Directory. GetFiles(path, "*.*" , SearchOption. AllDirectories)
   let totalSize =
```

```
    folder
    |> getFiles
    |> Array.map (fun file -> new FileInfo(file))
    |> Array.map (fun info -> info. Length)
    |> Array.sum
totalSize
```

使用 pipe-forward 运算符所带来的简洁是函数柯里化多么有用的很好示例。pipe-forward 运算符接收一个值和一个只接收一个参数的函数；然而，我们使用的函数，比如 Array.map，明显是接收两个参数(一个用来映射的函数和数组本身)。它依然工作的原因是我们部分应用了第一个参数，产生了一个只接收一个参数的函数，因此可以轻松地与 pipe-forward 运算符配合使用。

Note:虽然 pipe-forward 运算符可以通过移除不必要的中间结果的声明来极大地简化代码，但这也使得调试经过 pipe 的序列更加困难，因为你不能够检查任何中间结果。

pipe-forward 运算符的另一个好处是它能帮助类型推断的进行。如果编译器不知道值的类型，你就不能访问它的任何属性或方法。因此，你必须使用类型注释来

```
> // ERROR: 编译器不知道 s 有 Length 属性
List.iter
    (fun s -> printfn "s has length %d" s.Length)
    ["Pipe"; "Forward"];;
        (fun s -> printfn "s has length %d" s.Length)
    ----------------------------------------^^^^^^^^
stdin(2,41): error FS0072: 查找基于此程序点之前的信息的不确定类型的对象。在此程序点之前可能需要类型批注来约束对象的类型。这可能允许解析查找。
```

因为 pipe-forward 运算符允许编译器提前看到函数的最后一个参数的类型，所以类型推理系统就能更快地确定函数正确的类型，减少了类型注释的需求。

在下面的片段中，由于使用了 pipe-forward 运算符，传给 List.iter 的 lambda 的参数就知道是 string 类型，因此不需要类型注释：

```
> //  使用 pipe-forward 运算符来协助类型推断
```

```
["Pipe"; "Forward"] |> List.iter (fun s -> printfn "s has length %d" s.Length);;

s has length 4

s has length 7

val it : unit = ()
```

forward composition 运算符

forward composition 运算符，>>，把两个函数组合到一起，左边的函数被首先调用：

```
let (>>) f g x = g(f x)
```

当在函数中使用 pipe-forward 运算符时，你需要一个可用的占位符来安放管道 。在我们最后的例子中，函数接收一个 folder 参数，我们直接把它传给第一个经过 pipe 的函数：

```
let sizeOfFolderPiped2 folder =
   let getFiles folder =
      Directory. GetFiles(folder, "*.*" , SearchOption. AllDirectories)
   folder
   |> getFiles
   |> Array. map (fun file -> new FileInfo(file))
   |> Array. map (fun info -> info. Length)
   |> Array. sum
```

然而，当使用函数复合运算符时，并不需要任何参数。我们只是把所有的函数复合到一起，产生一个新的函数，它接收一个参数并计算结果：

```
> // 函数复合
open System.IO
open System
let sizeOfFolderComposed (*没有参数!*) =
    let getFiles folder =
        Directory.GetFiles(folder, "*.*", SearchOption.AllDirectories)
    // 这个表达式的结果是一个函数，它只接收一个参数
    // 这个函数将被传给 getFiles 并传输到后面的函数
```

```
    getFiles
    >> Array.map (fun file -> new FileInfo(file))
    >> Array.map (fun info -> info.Length)
    >> Array.sum;;
val sizeOfFolderComposed : (string -> int64)
>sizeOfFolderComposed
        (Environment.GetFolderPath(Environment.SpecialFolder.MyPictures));;
val it : int64 = 3797851L
```

这里是函数复合的另一个例子。下面的片段计算一个整数的平方的位数：

```
> // 基本的函数复合
let square x = x * x
let toString (x : int) = x.ToString()
let strLen (x : string) = x.Length
let lenOfSquare = square >> toString >> strLen;;
val square : x:int -> int
val toString : x:int -> string
val strLen : x:string -> int
val lenOfSquare : (int -> int)
> square 128;;
val it : int = 16384
> lenOfSquare 128;;
val it : int = 5
```

pipe-backward 运算符

乍一看，pipe-backward 运算符，<|，接收左边的一个函数然后把它应用到右边的值上。这看上去好像没有必要，因为它做的只是把一个函数和它的最后一个参数分开罢了，这实在不需要一个特别的运算符：

```
let (<|) f x = f x
```

这里是一个使用 pipe-backward 运算符的例子：

```
> List.iter (printfn "%d") [1 .. 3];;

1

2

3

val it : unit = ()

> List.iter (printfn "%d") <| [1 .. 3];;

1

2

3

val it : unit = ()
```

你也许会很奇怪，但 pipe-backward 运算符确实有一些重要的用处：它允许你改变优先级(precedence)。到目前为止我一直避免提到运算符的优先级，但它是函数应用的顺序。函数的参数被从左到右地计算，这意味着如果你想要调用一个函数并把结果传给另一个函数，你就有两种选择：在表达式周围添加括号，或者使用 pipe-backward 运算符：

```
> printfn "The result of sprintf is %s" (sprintf "(%d, %d)" 1 2);;

The result of sprintf is (1, 2)

val it : unit = ()

> printfn "The result of sprintf is %s" <| sprintf "(%d, %d)" 1 2;;

The result of sprintf is (1, 2)

val it : unit = ()
```

pipe-backward 运算符并不像 pipe-forward 或者函数复合运算符那样常见，但它在整理 F#代码时担当着重要的角色。

backward composition 运算符

就像 pipe-backward 运算符一样，等效的组合是 backward composition 运算符，`<<` 。backward composition 运算符接收两个函数，首先调用右边的函数，然后是左边的两个。它在你想要以颠倒的顺序来表达想法时非常有用。它的定义如下：

```
let (<<) f g x = f(g x)
```

下面的代码展示了如何获取一个数的相反数的平方。使用 backward composition 运算符允许你向函数实际工作的顺序阅读程序代码：

```
> // Backward Composition

let square x = x * x

let negate x = -x;;

val square : x:int -> int

val negate : x:int -> int

> // 使用>>来对平方取负

(square >> negate) 10;;

val it : int = -100

> (* 但我们实际想要的是相反数的平方，因此我们需要使用<< *)

(square << negate) 10;;

val it : int = 100
```

另一个使用 backward composition 运算符的例子是用它来过滤掉由列表元素组成的列表中的空列表。再一次，backward composition 运算符用来改变程序员阅读代码的方式：

```
> // 过滤列表

[ [1]; []; [4;5;6]; [3;4]; []; []; []; [9] ]

|> List.filter(not << List.isEmpty);;

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

Note: |>，<|，>> 和 << 运算符是整理 F#代码的一种方式。通常 F#在适当的位置使用这些运算符来是代码更加容易阅读和更加简洁。如果添加它们只会造成混乱或者疑惑，那你就应该避免使用它们。

模式匹配

所有的程序都需要对数据进行排序和筛选。要用函数式编程来完成这件事，你需要使用模式匹配(pattern matching)。模式匹配类似于其他编程语言中的 switch 结构，不过它强大得多。一个模式匹配是一系列的规则来计算模式是否与输入匹配。然后模式匹配表达式就会返回匹配成功的规则对应的结果，因此，模式匹配中的所有规则都必须返回相同的类型。

要实现模式匹配，你可以使用 match 和 with 关键字，加上一系列的匹配规则，每个规则后面都跟上一个箭头，->。下面的片段展示了对表达式 isOdd x 使用模式匹配来模仿 if 表达式的行为。第一个规则匹配值 true，如果这个规则匹配成功，就向控制台打印 "x is odd"：

```
> // 简单的模式匹配
let isOdd x = (x % 2 = 1)
let describeNumber x =
    match isOdd x with
    | true -> printfn "x is odd"
    | false -> printfn "x is even";;
val isOdd : x:int -> bool
val describeNumber : x:int -> unit
> describeNumber 4;;
x is even
val it : unit = ()
```

最简单的模式匹配是对常量值进行的。Example 3-5 通过同时对元组中的两个值进行匹配构建了一个布尔函数 And 的真值表。

Example 3-5. 使用模式匹配构建一个真值表

```
> // 通过模式匹配创建 And 的真值表
let testAnd x y =
    match x, y with
    | true, true -> true
    | true, false -> false
    | false, true -> false
    | false, false -> false;;
val testAnd : x:bool -> y:bool -> bool
> testAnd true true;;
val it : bool = true
```

注意对模式匹配的类型推断。在 Example 3-5 中，因为第一个规则把 x 与布尔值 true 相匹配，把 y 和另一个布尔值相匹配，所以 x 和 y 都被推断为布尔值。

下划线，_，是一个通配符，能与任何东西匹配。因此你可以使用通配符来捕获任何不是 true, true 的输入来简化前面的例子：

```
let testAnd x y =
    match x, y with
    | true, true -> true
    | _ , _ -> false
```

模式匹配规则按照它们被声明的顺序依次检查。因此如果你把通配符放到第一位，后面的规则永远都不会被检查。(幸运的是，在这种情况想 F#编译器会给出警告)

匹配失败

你也许会想，如果我们漏掉了原始版本的 testAnd 中的一种可能的真值表匹配会发生什么。比如：

```
let testAnd x y =
    match x, y with
    | true, true -> true
    | true, false -> false
    // | false, true -> false - 哎呀，false, true 的情况被忽略了!
    | false, false -> false
```

如果模式匹配中有一种对应的匹配没有找到，就会产生一个 Microsoft.FSharp.Core.MatchFailure Exception 类型的异常。你可以通过确保所有可能的情况都包含在内来避免这一点。幸运的是，如果 F#编译器能够判断出模式匹配规则不完整，那么它就会给出一个警告：

```
> // 不完整的模式匹配。哎呀，不是所有的字母都被匹配！
let letterIndex l =
    match l with
    | 'a' -> 1
    | 'b' -> 2;;
        match l with
```

```
  ----------^

stdin(3,11): warning FS0025: 此表达式中的模式匹配不完整。 例如，值 "' '" 可以指示模式未涵盖的用例。

val letterIndex : l:char -> int

> letterIndex 'k';;

letterIndex 'k';;

Microsoft.FSharp.Core.MatchFailureException: 大小写匹配不完整

   在 <StartupCode$FSI_0003>.$FSI_0003.main@()

已因出错而停止
```

命名模式

到目前为止我们只对常量值进行了匹配，但你也可以使用命名模式(Named Patterns)来提取数据并把它绑定到一个新的值上。考虑下面的例子。它对一个特定的字符串进行匹配，但对其他的任何东西它都会捕获一个新的值 x：

```
> // 命名模式
let greet name =
    match name with
    | "Robert" -> printfn "Hello, Bob"
    | "William" -> printfn "Hello, Bill"
    | x -> printfn "Hello, %s" x;;

val greet : name:string -> unit

> greet "Earl";;

Hello, Earl

val it : unit = ()
```

最后一个匹配规则并不匹配一个常量；相反，它把被匹配的值绑定到一个新的值上，然后你就可以在这个匹配规则内部使用这个值。值的捕获，就像通配符一样，可以与任何东西匹配，因此你应确保把特定的规则放到最前面。

匹配字面值

命名模式是强大的，但也阻止你使用已经存在的值来作为模式匹配的一部分。在前面的例子中，把模式匹配

的一部分的名称写死会导致代码难以维护，甚至可能导致 bug。

如果你想要匹配一些有名的值，但又不想每次都复制和粘贴把这些字面值，你就可能犯使用命名模式的错误。然而，这并不会随着你的意愿工作，只会影射已经存在的值。

在下面的例子中，第一个模式匹配规则并不将值 name 与 bill 相比较，相反，它只会引入一个叫 bill 的新值。这就是为什么第二个模式匹配规则会给出警告，因为前面的规则已经捕获了所有的模式匹配输入：

```
> // Error: 意料之外的值捕获
let bill = "Bill Gates"
let greet name =
    match name with
    | bill -> "Hello Bill!"
    | x -> sprintf "Hello, %s" x;;
      | x -> sprintf "Hello, %s" x;;
  ------^
stdin(6,7): warning FS0026: 从不与此规则匹配
val bill : string = "Bill Gates"
val greet : name:string -> string
```

为了匹配已经存在的值，你必须添加[<Literal>]特性。使用这个特性标记并以大写字母开头的任何字面值(一个常量)都可以在模式匹配的内部使用：

```
> // 定义一个字面值 [<Literal>]
let Bill = "Bill Gates";;
val Bill : string = "Bill Gates"
> // 对字面值进行匹配
let greet name =
   match name with
   | Bill -> "Hello Bill!"
   | x -> sprintf "Hello, %s" x;;
val greet : name:string -> string
```

```
> greet "Bill G.";;

val it : string = "Hello, Bill G."

> greet "Bill Gates";;

val it : string = "Hello Bill!"
```

Note:特性(Attributes)是一种使用元数据来注释.NET 代码的方式。要了解更多有关特性和.NET 元数据的信息，请查阅Chapter12。

只有整数、字符、布尔值、字符串以及浮点数可以被标记为字面值，如果你想匹配更复杂的类型，比如字典或者地图，你应该使用 when 条件。

when 条件

虽然模式匹配是一个强大的概念，但有时你需要自定义的逻辑来决定是否规则匹配。这就是 when 条件所有解决的问题。如果一个模式被匹配，可选的 when 条件就会进行计算，这个规则将起作用当且仅当 when 表达式的计算结果为 true。

下面的例子实现了一个简单的猜数游戏，when 条件用来检查猜测的数与随机器生成的数字相比是大了还是小了还是恰好相等：

```
> // High / Low 游戏

open System

let highLowGame () =

    let rng = new Random()

    let secretNumber = rng.Next() % 100

    let rec highLowGameStep () =

        printfn "请猜测随机器生成的数"

        let guessStr = Console.ReadLine()

        let guess = Int32.Parse(guessStr)

        match guess with

        | _ when guess > secretNumber

            -> printfn "猜大了！"

               highLowGameStep()
```

```
        | _ when guess = secretNumber
            -> printfn "猜对了！"
               ()
        | _ when guess < secretNumber
            -> printfn "猜小了！"
               highLowGameStep()
    // 开始游戏
    highLowGameStep();;
val highLowGame : unit -> unit
> highLowGame();;
请猜测随机器生成的数
0
猜小了！
请猜测随机器生成的数
100
猜大了！
请猜测随机器生成的数
50
猜对了！
val it : unit = ()
```

合并模式

随着你的模式匹配包含越来越多的规则，你可能想把模式组合到一起。有两种组合的方式。第一种是使用 Or 模式，用一个竖线(|)表示，它把模式组合到一起，如果组合中的任何一个模式匹配成功，这个规则就执行。第二种方式是使用 And 模式，用一个&表示，它也把模式组合到一起，只有组合中的所有模式都匹配成功，这个规则才会执行：

```
let vowelTest c =
    match c with
```

```
    | 'a' | 'e' | 'i' | 'o' | 'u'
        -> true
    | _ -> false
let describeNumbers x y =
    match x, y with
    | 1, _     | _ , 1
        -> "其中一个数字是 1。"
    | (2, _ ) & ( _ , 2)
        -> "两个数字都是 2。"
    | _ -> "其他情况。"
```

在通常的模式匹配中 And 模式很少被使用；然而，当使用活动模式(Chapter7)时它是十分重要的。

匹配数据结构

模式匹配也可用于匹配数据结构。

元组

你已经看到如何匹配元组。如果在模式匹配中元组的元素由逗号分开，那么每个元素都会被同时匹配。然而，如果元组输入被用作命名模式，这两个值都会具有元组类型

在下面的例子中，第一个规则绑定值 tuple，它同时捕获值 x 和 y。另一个规则同时匹配元组的元素：

```
let testXor x y =
    match x, y with
    | tuple when fst tuple <> snd tuple
        -> true
    | true, true -> false
    | false, false -> false
```

列表

Example 3-6 示范了如何对列表结构进行模式匹配。函数 listLength 匹配固定大小的列表；否则，它就伴随列表的尾部递归调用本身。

Example 3-6. 判断列表的长度

```
let rec listLength l =
    match l with
    | [] -> 0
    | [ _ ] -> 1
    | [ _ ; _ ] -> 2
    | [ _ ; _ ; _ ] -> 3
    | hd :: tail -> 1 + listLength tail
```

前四个模式匹配规则对特定长度的列表进行匹配，使用通配符来表明列表的元素并不重要。然而，最后一行的模式匹配，使用了 cons 运算符(::)来匹配列表的第一个元素 hd，以及列表中剩余的元素 tail。tail 可以是任意列表，从空列表到包含上百万个元素的列表。(不过，因为模式匹配中有前面四个规则，所以我们可以推断 tail 至少有 3 个元素。)

选项

模式匹配也为选项类型的使用提供了一种更函数式的方式：

```
let describeOption o =
    match o with
    | Some(42) -> "答案是 42，但问题是什么？"
    | Some(x) -> sprintf "答案是 %d" x
    | None -> "没找到答案。"
```

匹配表达式之外

模式匹配是一个非常强大的概念，但它的使用绝不仅限于 match with 表达式。模式匹配贯穿整个 F#语言。

let 绑定

let 绑定实际上就是模式匹配规则。因此如果你写：

```
let x = f()
...
```

你可以认为你写的是：

```
match f() with | x -> ...
```

这就是为什么我们可以使用 let 绑定来从元组中提取值：

```
// 这个...

let x, y = (100, 200)

// ...和这个一样...

match (100, 200) with

| x, y -> ...
```

Note:这就是为什么 F#会怀疑地接收这段代码：let 1 = 2。在 FSI 窗口中尝试它，看看会发生什么。

函数参数

函数的参数也是伪装的模式匹配！

```
// 给定一个选项值的元组，返回它们的和

let addOptionValues = fun (Some(x), Some(y)) -> x + y
```

通配符模式

设想你想要编写一个函数但并不关心其中的某个参数，或者是想要忽略元组中的值。在这种情况下，你应该使用一个通配符模式：

```
> List.iter (fun _ -> printfn "Step...") [1 .. 3];;

Step...

Step...

Step...

val it : unit = ()

> let _ , second, _ = (1, 2, 3);;

val second : int = 2
```

替代的 Lambda 语法

模式匹配的最后一个用途是简化 lambda 语法。在编写 F#代码时，你会发现经常把参数直接传给一个模式匹配表达式，比如：

```
let rec listLength theList =

    match theList with

    | [] -> 0

    | [ _ ] -> 1
```

```
    | [ _ ; _ ] -> 2
    | [ _ ; _ ; _ ] -> 3
    | hd :: tail -> 1 + listLength tail
```

一种更简单的写法是使用关键字 function，它与 fun 关键字创建 lambda 很像，只是 function lambda 只接收一个参数，且这个参数必须放在一个模式匹配内部。下面的例子使用 function 关键字重写了 listLength 函数。

```
> // function 关键字
let rec funListLength =
    function
    | [] -> 0
    | [ _ ] -> 1
    | [ _ ; _ ] -> 2
    | [ _ ; _ ; _ ] -> 3
    | hd :: tail -> 1 + funListLength tail;;
val funListLength : _arg1:'a list -> int
> funListLength [1 .. 5];;
val it : int = 5
```

可区分联合

函数式编程的一个基本类型是可区分联合(discriminated union)，一种只能为可能值的集合中的一种的类型。可区分联合的每一个可能值都被称为一种联合情况。由于可区分联合只能是值的集合中的一种，所以编译器会添加额外的检查来确保代码的正确性。特别地，F#编译器会确保模式匹配涵盖所有的可区分联合情况。

Note:你已经使用过可区分联合类型。选项类型就是一种可区分联合，它有两种联合情形：Some('a)和 None。

要定义一个可区分联合，请使用关键字 type，后面跟上类型名称以及由一个管道(|)分开的每种联合情况。在一副标准的纸牌中，一组纸牌(Suit)可以由下面的可区分联合表示：

```
> // 一组纸牌的可区分联合
type Suit =
    | Heart // 红心
```

```
    | Diamond // 方块
    | Spade // 黑桃
    | Club // 梅花;;
type Suit =
    | Heart
    | Diamond
    | Spade
    | Club
> let suits = [ Heart; Diamond; Spade; Club ];;
val suits : Suit list = [Heart; Diamond; Spade; Club]
```

你也可以选择性地将数据与每种联合情况相关联。要继续我们的纸牌示例，每个纸牌都可以与一个 Suit 相关联，纸牌的值是一个整数和 Suit 构成的组合(int * Suit 语法也许看上去更熟悉——它是一个元组的类型标识)：

```
// 纸牌角色的可区分联合
type PlayingCard =
    | Ace of Suit
    | King of Suit
    | Queen of Suit
    | Jack of Suit
    | ValueCard of int * Suit
// 使用列表推导来生成一副纸牌
let deckOfCards =
    [
        for suit in [ Spade; Club; Heart; Diamond ] do
            yield Ace(suit)
            yield King(suit)
            yield Queen(suit)
            yield Jack(suit)
```

```
            for value in 2 .. 10 do
                yield ValueCard(value, suit)
    ]
```

也可以在同一行声明可区分联合，这时第一个管道(|)可以省略：

```
type Number = Odd | Even
```

可区分联合也可以是递归的。如果你需要定义一系列相互递归的可区分联合，那么它们需要用 and 关键字联系到一起。

下面定义了一个简单的格式来描述一门编程语言：

```
// 程序声明
type Statement =
    | Print of string
    | Sequence of Statement * Statement
    | IfStmt of Expression * Statement * Statement
// 程序表达式
and Expression =
    | Integer of int
    | LessThan of Expression * Expression
    | GreaterThan of Expression * Expression
(*
   if (3 > 1)
   print "3 is greater than 1"
   else
   print "3 is not"
   print "greater than 1"
*)
let program =
    IfStmt(
```

```
        GreaterThan(
            Integer(3),
            Integer(1)),
        Print("3 is greater than 1" ),
        Sequence(
            Print("3 is not" ),
            Print("greater than 1" )
        )
    )
```

使用可区分联合来描述树结构

可区分联合很适合表示树状的数据结构，就像在前面的代码片段里那样。

Example 3-7 仅用 12 行代码便定义了一个二叉树和一个用来遍历它的函数。

Example 3-7. 使用可区分联合的二叉树

```
type BinaryTree =
    | Node of int * BinaryTree * BinaryTree
    | Empty


let rec printInOrder tree =
    match tree with
    | Node (data, left, right) ->
        printInOrder left
        printfn "Node %d" data
        printInOrder right
    | Empty -> ()
(*
      2
    /   \
```

```
   1       4

         /     \

       3         5

*)

let binTree =

    Node(2,

        Node(1, Empty, Empty),

        Node(4,

            Node(3, Empty, Empty),

                Node(5, Empty, Empty)

                )

    )
```

当在一个 FSI 会话中执行时，前面的例子会打印下面的内容：

```
> printInOrder binTree;;

Node 1

Node 2

Node 3

Node 4

Node 5

val it : unit = ()
```

模式匹配

你可以对可区分联合进行模式匹配，只需把情况的标签作为模式。如果联合的标签有与之相关的数据，那么你就可以像通常的模式匹配那样把它与一个常量、一个通配符或者是捕获值进行匹配。

下面的例子通过描述在一次扑克牌游戏中的两张底牌来示范了模式匹配和可区分联合的强大威力：

```
// 描述一次扑克牌游戏中的一对牌

let describeHoleCards cards =

    match cards with
```

```
    | []
    | [ _ ] -> failwith "牌太少。 "
    | cards when List.length cards > 2 -> failwith "牌太多。 "
    | [ Ace( _ ); Ace( _ ) ] -> "火箭"
    | [ King( _ ); King( _ ) ] -> "牛仔"
    | [ ValueCard(2, _ ); ValueCard(2, _ ) ] -> "鸭子"
    | [ Queen( _ ); Queen( _ ) ] | [ Jack( _ ); Jack( _ ) ] -> "一组副牌"
    | [ ValueCard(x, _ ); ValueCard(y, _ ) ] when x = y -> "一对"
    | [ first; second ] -> sprintf "两张牌: %A 和 %A" first second
```

你也可以在模式匹配中使用递归定义的可区分联合。注意在下面的例子中，第三个规则使用了一个嵌套模式，它只匹配有两个 Worker 员工的 Manager 值：

```
type Employee =
    | Manager of string * Employee list
    | Worker of string
let rec printOrganization worker =
    match worker with
    | Worker(name) -> printfn "Employee %s" name
    // Manager 和一个只含一个 Worker 的列表
    | Manager(managerName, [ Worker(employeeName) ] )
        -> printfn "Manager %s with Worker %s" managerName employeeName
    // Manager 和一个含两个 Worker 的列表
    | Manager(managerName, [ Worker(employee1); Worker(employee2) ] ) ->
        printfn
            "Manager %s with two workers %s and %s"
            managerName employee1 employee2
    // Manager 和一个含多个 Worker 的列表
    | Manager(managerName, workers) ->
```

```
        printfn "Manager %s with workers..." managerName
        workers |> List.iter printOrganization
```

在 FSI 会话中，前面的例子将产生这样的结果：

```
> let company = Manager("Tom", [ Worker("Pam"); Worker("Stuart") ] );;
val company : Employee = Manager ("Tom",[Worker "Pam"; Worker "Stuart"])
> printOrganization company;;
Manager Tom with two workers Pam and Stuart
val it : unit = ()
```

因为在编译时编译器知道与可区分联合有关的所有可能的数据标签，所以任何不完整的模式匹配都会给出警告。如果联合情形 Ace 被遗忘，那么 F#编译器就会知道：

```
> // 哎呀！搞忘了联合情形 Ace...
let getCardValue card =
    match card with
    | King( _ ) | Queen( _ ) | Jack( _ ) -> 10
    | ValueCard(x, _ ) -> x;;
match card with
  ----------^^^^
stdin(13,11): warning FS0025: 此表达式中的模式匹配不完整。 例如，值“Ace (_)”可以指示模式未涵盖的用例。
val getCardValue : card:PlayingCard -> int
```

Note:对可区分联合使用通配符时应格外小心。如果在后面联合被扩展，那么你就会在模式匹配不完整的地方收到一个警告。但是，如果你使用了通配符，那么就不会有任何警告，因为它占据了其他所有的情况。在代码可能匹配失败的地方给出警告对阻止运行时的缺陷有很大的用处。

方法和属性

你可以通过添加方法(method)和属性(propertie)来给可区分联合增加威力。在下面的片段中添加了属性 Value，因此给定任何一个 PlayingCard 对象，你可以获得它的 value(要了解更多添加方法和属性的信息，请查阅 Chapter5)：

```
type PlayingCard =
    | Ace of Suit
    | King of Suit
    | Queen of Suit
    | Jack of Suit
    | ValueCard of int * Suit
    member this.Value =
        match this with
        | Ace( _ ) -> 11
        | King( _ ) | Queen ( _ )| Jack( _ ) -> 10
        | ValueCard(x, _ ) when x <= 10 && x >= 2 -> x
        | ValueCard( _ ) -> failwith "Card has an invalid value!"
let highCard = Ace(Spade)
let highCardValue = highCard.Value
```

记录

可区分联合对定义数据的层次结构非常有用，但在尝试获取一个可区分联合之外的值时，它们与元组有着相同的问题。换句话说，这里的问题是对每个值没有与之相关的实际意义——相反，数据是以一些固定的顺序组合到一起的。例如，考虑一个描述人的单一情形的可区分联合。它的两个 string 字段指代的是先姓后名呢，还是先名后姓呢？

```
type Person =
| Person of string * string * int
let steve = Person("Steve" , "Holt" , 17)
let gob = Person("Bluth" , "George Oscar" , 36)
```

当你想要把你的数据组合到一种结构化的格式，而又不需要强健的语法时，你可以使用 F#的记录类型(Record)。记录给你提供了一种将值组合到一种类型的方式，同时提供字段来命名这些值。

要定义一个记录，你只需简单地在大括号里定义一系列的 name/type 组合。要创建一个记录的实例，只需提供每个字段(field)的值，类型推理将完成剩余的事。请看 Example 3-8。

Example 3-8. 创建和使用记录

```
> // 定义一个 Record 类型

type PersonRec = { First : string; Last : string; Age : int};;

type PersonRec =

  {First: string;

   Last: string;

   Age: int;}

> // 创建一个 Record

let steve = { First = "Steve"; Last = "Holt"; Age = 17 };;

val steve : PersonRec = {First = "Steve";

                         Last = "Holt";

                         Age = 17;}

> // 使用'.field'来获取记录字段

printfn "%s is %d years old" steve.First steve.Age;;

Steve is 17 years old

val it : unit = ()
```

Note:对于有经验的.NET 开发人员，看上去 Record 就好像标准的.NET class 的简化版本。在 F#中你可以轻松地创建属性和方法，因此为何还使用记录呢？记录比传统的面向对象的数据结构提供了一些独特的优势。

- 类型推理可以推断记录的类型。而不需要多余的类型注释。
- 记录字段默认是不可变的，而 class 类型则没有提供安全的保障。
- 记录不可以被继承，这给出了一个安全保障。
- 记录可以作为标准的模式匹配的一部分使用；class 不借助活动模式或者 when 条件则不能实现这一点。
- 记录(以及可区分联合)自动获得 comparison 和 equality 语义。.NET 中的 equality 和 comparison 将在 Chapter6 中讨论。

克隆记录

记录可以轻松地使用 with 关键字进行克隆：

```
type Car =
```

```
    {
        Make : string
        Model : string
        Year : int
    }
let thisYear's = { Make = "FSharp" ; Model = "Luxury Sedan" ; Year = 2012 }
let nextYear's = { thisYear's with Year = 2013 }
```

这样写是一样的：

```
let nextYear's =
    {
        Make = thisYear's. Make
        Model = thisYear's. Model
        Year = 2013
    }
```

模式匹配

你可以对记录进行模式匹配，只要值被捕获并且字面值匹配。注意并非所有的记录字段都要成为模式匹配的一部分。在下面的例子中，一个 Car 的列表被过滤，只留下 Model 字段等于"Coup"的元素。Fields Make 和 Year 则未被考虑：

```
let allCoups =
    allNewCars
    |> List. filter
        (function
        | { Model = "Coup" } -> true
        | _ -> false)
```

类型推理

记录的一个独特的优点在于如何与 F#类型推理系统一起工作。与.NET class 类型必须被注释才能使用相比，记录的类型可以通过你使用的字段被推断出来。

在 Example 3-9 中,并没有提供值 pt1 和 pt2 的类型注释。因为使用了字段 X 和 Y 而编译器知道存在一个含 X 和 Y 字段的记录类型，所以 pt1 和 pt2 被推断为具有类型 Point。

Example 3-9. 对记录的类型推理

```
> type Point = { X : float; Y : float };;

type Point =

  {X: float;

   Y: float;}

> // 两点间的距离(不需要类型注释！)

let distance pt1 pt2 =

    let square x = x * x

    sqrt <| square (pt1.X - pt2.X) + square (pt1.Y - pt2.Y);;

val distance : pt1:Point -> pt2:Point -> float

> distance {X = 0.0; Y = 0.0} {X = 10.0; Y = 10.0};;

val it : float = 14.14213562
```

当使用两个具有相同字段名称的记录时，类型推理系统要么会给出一个错误，要么推断成不同于你想要的类型。要解决这个问题，你既可以提供类型注释，也可以完全限制记录字段：

这为类型推理系统提供了指出你的意思的所有必要的信息。

```
type Point = { X: float; Y: float;}

type Vector3 = { X : float; Y : float; Z : float }

// 提供一个类型注释来防止把 pt1 和 pt2 推断为 Vector3

// (因为 Vector3 也有字段 X 和 Y，且它是最近定义的)

let distance (pt1 : Point) (pt2 : Point) =

    let square x = x * x

    sqrt <| square (pt1.X - pt2.X) + square (pt1.Y - pt2.Y)

// 通过完全限制记录字段来消除 Point 和 Vector 类型之间的歧义

let origin = { Point.X = 0.0; Point.Y = 0.0 }
```

方法和属性

与可区分联合相似，你也可以给记录添加方法和属性：

```
> // 为记录类型 Vector 添加一个属性
type Vector =
    { X : float; Y : float; Z : float }
    member this.Length =
        sqrt <| this.X ** 2.0 + this.Y ** 2.0 + this.Z ** 2.0;;
type Vector =
  {X: float;
   Y: float;
   Z: float;}
  with
    member Length : float
  end
> let v = { X = 10.0; Y = 20.0; Z = 30.0 };;
val v : Vector = {X = 10.0;
                  Y = 20.0;
                  Z = 30.0;}
> v.Length;;
val it : float = 37.41657387
```

惰性求值

到目前为止我们所写的代码都是被热情求值的，也就意味着它在我们把它写入 FSI 之后就立刻被计算。如果你编译并运行你的 F#程序效果也是一样。然而，在某些情形下你只想要执行需要的代码——例如，你只想在必要的时候才执行计算代价昂贵的操作。这就被称为惰性求值(lazy evaluation)。

使用惰性求值能够明显地减少内存占用，因为你只在需要值的时候才在内存中创建它们。

F#有两种方式来支持惰性求值——通过 Lazy 类型和序列(seq<'a>类型)。

Lazy 类型

Lazy 类型是某些计算的形实转换程序或者占位符。一旦被创建，你就可以传递 lazy 值，就好像它们已经被计算了一样。但事实上它们只会被计算一次，即在被强制要求的时候。

Example 3-10 创建了两个值 x 和 y，两者的副作用都是向控制台打印文字。因为它们被延迟初始化，所以它们在声明时并不会被执行。只有当 y 被需要的时候 y 的计算才会被强制执行，这反过来又要求 x 计算的执行。

要创建一个 lazy 值，你可以使用 lazy 关键字或者 Lazy<_>.Create。

Example 3-10. 使用惰性求值

```
> // 定义两个 lazy 值
let x = Lazy<int>.Create(fun () -> printfn "Evaluating x..."; 10)
let y = lazy (printfn "Evaluating y..."; x.Value + x.Value);;
val x : System.Lazy<int> = 未创建值。
val y : Lazy<int> = 未创建值。
> // 直接获取 y 的值将强迫它执行
y.Value;;
Evaluating y...
Evaluating x...
val it : int = 20
> // 再次获取 y 的值将使用缓存的值(没有副作用)
y.Value;;
val it : int = 20
```

序列

在 F#中，惰性求值最常见的使用是通过序列(sequence)或者 seq 类型完成的，它表示一个有序的元素序列，与 List 很相似。下面的片段定义了一个有 5 个元素的序列，并使用 Seq.iter 函数来迭代它们(就像 List 模块一样，也有一大堆可用的函数来操作序列)：

```
> let seqOfNumbers = seq { 1 .. 5 };;
val seqOfNumbers : seq<int>
> seqOfNumbers |> Seq.iter (printfn "%d");;
1
```

```
2

3

4

5

val it : unit = ()
```

Note: seq 只是.NET 接口 System.Collections.Generic.IEnumerable<'a>的一个缩写

那么既然你能够使用列表，又为什么还会有这两种类型呢？答案与 seq<_>和 list 类型对内存的影响有关。列表的内容完整地储存在内存中，而序列的元素则以所谓的“pull”的方式动态地产生。你可以十分轻松地定义有关无穷序列 ，但是一个无穷列表则会耗尽所有的内存。同样，对于列表，你必须提前知道每个元素的值，然而序列的元素可能会与程序事件相关。

Example 3-11 定义了一个以字符串的方式表示的所有的 32 位正整数的序列。然后尝试创建一个等价的列表，但由于内存的限制而失败了。

Example 3-11. 所有整数的序列

```
> // 所有正整数的序列
let allPositiveIntsSeq =
    seq { for i in 1 .. System.Int32.MaxValue do
            yield i };;
val allPositivesIntsSeq : seq<int>
> allPositiveIntsSeq;;
val it : seq<int> = seq [1; 2; 3; 4; ...]
> // 所有正整数的列表——ERROR: 内存装不下！
let allPositiveIntsList = [ for i in 1 .. System.Int32.MaxValue -> i ];;
System.OutOfMemoryException: 引发类型为“System.OutOfMemoryException”的异常。
已因出错而停止
```

序列表达式

我们可以使用与列表表达式相同的语法来定义序列(严格来说应称为序列表达式(sequence expressions)。序列由 seq 跟上花括号开头：

```
> let alphabet = seq { for c in 'A' .. 'Z' -> c };;

val alphabet : seq<char>

> Seq.take 4 alphabet;;

val it : seq<char> = seq ['A'; 'B'; 'C'; 'D']
```

序列被延迟计算，因此每次产生一个元素时，序列内部的代码计算仍在运行。让我们再次尝试前面的例子，不过这次在每个元素被返回时都添加一个副作用。除了返回 alphabet 中的字母外，它还会把这些字母打印到控制台：

```
> // 加上一个副作用的序列
let noisyAlphabet =
    seq {
        for c in 'A' .. 'Z' do
        printfn "Yielding %c..." c
        yield c
    };;

val noisyAlphabet : seq<char>

> let fifthLetter = Seq.nth 4 noisyAlphabet;;

Yielding A...

Yielding B...

Yielding C...

Yielding D...

Yielding E...

val fifthLetter : char = 'E'
```

序列表达式的一个有趣的特点是它们可以是递归的。通过使用关键字 yield! (读作 yield bang)，你可以返回一个子序列，其中的结果将按照顺序合并到主序列中。

Example 3-12 展示了如何在序列表达式中使用 yield!。函数 allFilesUnder 返回一个 seq<string> ，它递归地获取给定文件夹下的所有文件。Directory.GetFiles 返回一个包含 basePath 文件夹下所有文件的 sting 数组。因为数据与序列兼容，所以 yield!会返回所有这些文件。

Example 3-12. 用序列列出一个文件夹下的所有文件

```
open System. IO

let rec allFilesUnder basePath =

    seq {

        // 输出基础文件夹下的所有文件

        yield! Directory.GetFiles(basePath)

        // 输出它的子文件夹中的所有文件

        for subdir in Directory.GetDirectories(basePath) do

            yield! allFilesUnder subdir

    }
```

Seq Module Functions

The Seq module contains many helpful functions for using sequence types.

Seq.take

Returns the first n items from a sequence:

```
> // Sequence of random numbers

open System

let randomSequence =

 seq {

 let rng = new Random()

 while true do

 yield rng.Next()

 };;
```

```
val randomSequence : seq<int>

> randomSequence |> Seq.take 3;;

val it : seq<int> = seq [2101281294; 1297716638; 1114462900]
```

Seq.unfold

Seq.unfold generates a sequence using the provided function. It has type ('a -> ('b * 'a) option) -> 'a -> seq<'b>.

The supplied function takes an input value, and its result is an option type with a tuple of the next value in the sequence combined with the input to the next iteration of Seq.unfold. The function returns None to indicate the end of the sequence.

Example 3-13 generates all Fibonacci numbers under 100 (the nth Fibonacci number is the sum of the two previous items in the sequence; or 1, 1, 2, 3, 5, 8, etc.). The example also uses the Seq.toList function, which converts a sequence to a list.

Example 3-13. Using Seq.unfold

```
> // Generate the next element of the Fibonacci sequence given the previous
// two elements. To be used with Seq.unfold.
let nextFibUnder100 (a, b) =
if a + b > 100 then
None
else
let nextValue = a + b
Some(nextValue, (nextValue, a));;
val nextFibUnder100 : int * int -> (int * (int * int)) option
```

```
> let fibsUnder100 = Seq.unfold nextFibUnder100 (0, 1);;

val fibsUnder100 : seq<int>

> Seq.toList fibsUnder100;;

val it : int list = [1; 1; 2; 3; 5; 8; 13; 21; 34; 55; 89]
```

Other useful Seq module functions can be found in Table 3-1.

Table 3-1. Common Seq module functions

Function and Type	Description
Seq.length seq<'a> -> int	Returns the length of the sequence.
Seq.exists ('a -> bool) -> seq<'a> -> bool	Returns whether or not an element in the sequence satisfies the search function.
Seq.tryFind ('a -> bool) -> seq<'a> -> 'a option	Returns Some(x) for the first element x in which the given function returns true. Otherwise returns None.
Seq.filter ('a -> bool) -> seq<'a> -> seq<'a>	Filters out all sequence elements for which the provided function does not evaluate to true.
Seq.concat (seq< #seq<'a> > -> seq<'a>	Flattens a series of sequences so that all of their elements are returned in a single seq.

Aggregate Operators

The Seq module also contains the same aggregate operators available in the List module.