**Chapter 4 命令式编程**

—————————————————————————————————————————————

到目前为止我们所写的大多数程序都是纯粹的，意味着它们永远不会改变状态。无论什么时候函数干了一件除返回值之外的事情，它就被称为副作用。虽然纯粹的函数具有一些有趣的特点(例如可复合性)，但事实上除非程序完成这样一些事情：把数据保存到磁盘上、把值打印到屏幕上、发出网络流量等等，它们才有意义。这些副作用才是事情真正完成的地方。

这一章介绍了如何改变程序的状态和如何改变控制流，这被称作命令式编程。与函数式编程相比，这种编程风格被认为更容易导致错误，因为它提供了让事情变坏的机会。给计算机的指令分支越复杂，或者向内存中写入的确定值越多，程序员犯错误的可能性就越大。当你以函数式风格编程时，所有的数据都是不可变的，所以你不可能意外地分配一个错误的值。然而，使用得当，命令式编程能成为F#开发的一个巨大优势。

命令式编程的一些潜在的好处是：

• 增加表现力  
 • 代码清晰，易于维护  
 • 与已有代码的交互  
 命令式编程是一种风格，程序通过改变内存中的数据来完成任务。这通常导致这样一种模式：程序被写成一系列的状态或者命令。Example 4-1展示了一个假想的程序：使用一个杀手机器人来接管地球。这些函数并不返回值，但确实影响了系统的某些部分，比如更新一个内部的数据结构。

Example 4-1.用命令式编程来接管地球

let robot = new **GiantKillerRobot**()

robot.Initialize()

robot.EyeLaserIntensity <- Intensity.Kill

robot.Target <- [| Animals; Humans; Superheroes |]

// 接管地球的序列

let earth = **Planets**. Earth

while robot. Active && earth. ContainsLife do

if robot.CurrentTarget.IsAlive then

robot.FireEyeLaserAt(robot.CurrentTarget)

else

robot.AquireNextTarget()

虽然这个代码片段让接管地球看起来非常容易，但你并没有完全看到后台进行的艰苦工作。Initialize函数可能需要点燃一个核反应堆；如果Initialize被连续调用两次，核反应堆可能会爆炸。如果Initialize是以纯函数的风格编写的，那么它的输出就只取决于函数的输入。相反，在函数调用Initialize过程中会发生什么则取决于当前内存的状态。

虽然这一章不会教你如何编写一个行星统治机器人，但它会详细地介绍如何编写能够改变程序运行环境的F#程序。你将学习如何声明变量，你可以在程序执行过程中改变它的值。你将学习如何使用可变的集合类型，它们提供了F#列表类型的一个方便使用的替代选择。最后，你将了解控制流和异常，这允许你改变代码执行的顺序。

**理解.NET中的内存**

在你可以开始改变内存之前，你首先需要明白在.NET中内存是如何工作的。.NET应用中的值被存储在两种位置之一：在栈(stack)或者在堆(heap)中。(有经验的程序员可能已经对这些概念非常熟悉了) 栈是一块每次操作所需的固定大小的内存，局部变量就存储在这里。局部变量是临时的值，只用作函数的延续，就像一个循环计数器。栈的空间相对有限，而堆(也叫RAM)则可能包含上GB的数据。.NET同时使用栈和堆来，在可能的情况下有效地利用栈中廉价的内存分配，而在必要的情况下把数据存储在堆中。

值在内存中存储的位置将影响你对它的使用方式。

值类型和引用类型

储存在栈中的值被称为值类型(value type)，而储存在堆中的值被称为引用类型(reference type)。

值类型在栈中占据固定数目的字节。int和float都是值类型的例子，因为它们的大小是固定的。而另一方面，引用类型则只在栈中储存一个指针，它是堆中一些内存的地址。因此虽然指针的大小固定——通常是4个或者8个字节——但它所指向的内存可以非常非常大。list和string都是引用类型的例子。

这一点可以在Figure 4-1中直观的看出。整数5存在于栈中，在堆中没有副本。而一个字符串则作为一个内存地址存在于栈中，指向堆中的一些字符序列。

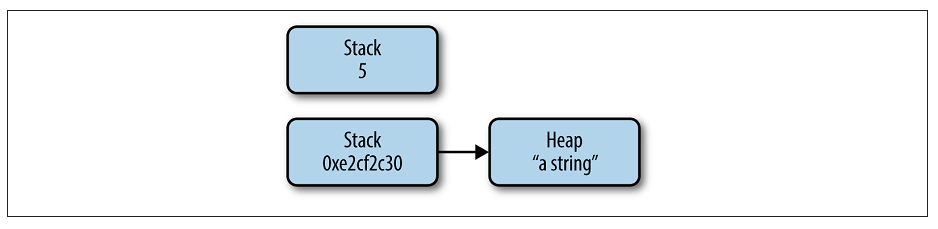


Figure 4-1.值类型与引用类型

默认值

到目前为止，你在F#中声明的每个值都会在它被创建后马上被初始化，因为以函数式风格编程时值一旦被声明便不能被改变。然而在命令式编程中，没有必要完整地初始化值，因为你可以在后面更新它们。这意味着对于值类型和引用类型都有一个默认值(default value)的概念。它是值未初始化之前具有的状态。

要获得某一类型的默认值，你可以使用类型函数Unchecked.default of<'a>。它将返回指定类型的默认值。

Note: 类型函数是特殊的函数类型，它们只接受泛型类型的参数。有几个有用的函数，你将在后面的章节中探索它们：

• Unchecked.defaultof<'a> 获取'a的默认值。  
• typeof<'a> 返回描述'a的System.Type对象。

• sizeof<'a> 返回'a的底层栈的大小。

对于值类型，它们的默认值只是一个zero-bit模式。因为一旦值类型被创建，它的大小就是已知的，所以它以byte的形式分配在栈中，每个byte都给定值0b00000000。引用类型的默认值稍微复杂一些。

在引用类型被初始化之前，它们首先指向一个称作null的特殊地址。这被用来表示一个未初始化的引用类型。在F#中，你可以使用null关键字来检查一个引用类型是否等于null。下面的代码定义了一个函数来检查它的输入是否为null，然后分别对一个初始化的和一个未初始化的string值调用它：

> let isNull = function null -> true | \_ -> false;;

val isNull : \_arg1:'a -> bool when 'a : null

> isNull "a string";;

val it : bool = false

> isNull (null : **string**);;

val it : bool = true

然而，在F#中定义的引用类型并不以null为固有值，这意味着它们不可能被分配为null：

> type **Thing** = Plant | Animal | Mineral;;

type Thing =

| Plant

| Animal

| Mineral

> // ERROR: 不能为null

let testThing thing =

match thing with

| Plant -> "Plant"

| Animal -> "Animal"

| Mineral -> "Mineral"

| null -> "(null)";;

| null -> "(null)";;

------^^^^

stdin(8,7): error FS0043: 类型“Thing”未将“null”用作适当的值

这看上去像是一个奇怪的限制，但它排除了过多的null检查的需求。(如果对一个未初始化的引用类型调用方法，你的程序将会抛出一个NullReferenceException，所以在其他的.NET语言中保护性地检查所有的函数参数是否为null是很常见的。) 如果你实在需要在F#中表示未初始化的状态，考虑使用选项类型来代替值为null的引用类型，None表示未初始化的状态而Some('a)表示已初始化的状态。

Note: 你可以设置某些F#类型的属性来接受null作为固有值以缓和与其他.NET语言的交互(查阅Appendix B以了解更多信息)。这个附录还介绍了System.Nullable<T>类型，它被用作其他.NET语言在的F#选项类型的原型。

引用类型别名

有可能两个引用类型指向堆中的相同内存地址。这被称为别名(aliasing)。当这件事情发生时，修改一个值将会静默地修改另一个值，因为它们都指向同一内存地址。如果你不注意，这种情况可能导致bug。

Example 4-2创建了一个数组(很快就会讨论)的实例。修改值x也会修改y，反之亦然。

Example 4-2. 引用类型别名

> // 值x指向一个数组，而y指向与x相同的内存地址

let x = [| 0 |]

let y = x;;  
val x : int [] = [|0|]  
val y : int [] = [|0|]  
> // 如果你修改x的值...

x.[0] <- 3;;

val it : unit = ()  
> // ... x将会改变...

x;;

val it : int [] = [|3|]

> // ... 但y也一样...

y;;

val it : int [] = [|3|]

**改变值**

既然你已经理解了在.NET数据是如何存储的，你就可以了解如何改变这些值。可变变量是你能改变的东西，可以通过mutable关键字来定义。要改变一个可变变量的内容，请使用左箭头运算符，<- :

> let mutable message = "World";;

val mutable message : string = "World"

> printfn "Hello, %s" message;;

Hello, World

val it : unit = ()

> message <- "Universe";;

val it : unit = ()

> printfn "Hello, %s" message;;

Hello, Universe

val it : unit = ()

对可变变量有几个限制，它们都来源于CLR中与安全有关的限制。这阻止你编写一些使用可变值的代码。

Example 4-3尝试定义一个内部函数incrementX，它在闭包中捕获一个可变变量x(这意味着它能够访问x，即使x并不作为参数传入）。这导致了一个来自F#编译器的错误，因为可变变量只能够在它们被定义的函数中使用。

Example 4-3. 在闭包中使用可变变量造成的错误

> // ERROR: 不能在可变变量被定义的函数之外使用它们

let invalidUseOfMutable() =

let mutable x = 0

let incrementX() = x <- x + 1

incrementX()

x;;

let incrementX() = x <- x + 1

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

stdin(4,24): error FS0407: 可变变量“x”的使用方式无效。无法由闭包来捕获可变变量。请考虑取消此变量使用方式，或通过“ref”和“!”使用堆分配的可变引用单元格。

与可变变量有关的两个限制如下：

• 可变变量不能从函数中返回(创建一个副本作为代替)

• 可变变量不能在内部函数(闭包)中被捕获

如果你遇到这两个问题之一，简单的处理方式是使用引用单元来把可变数据存储在堆中。

引用单元

ref类型，有时也称作引用单元(ref cell)，允许你把可变的数据存储在堆中，从而避开存储在栈中的可变变量的限制。要检索引用单元的值，请使用 ! 符号运算符，要设置它的值，请使用 := 运算符。

ref不仅是一种类型的名称，而且也是输出ref值的函数的名称，这样的函数的标识为：val ref: 'a -> 'a ref

ref函数接受一个值，然后返回它的一个包装成引用单元的副本。Example 4-4展示了把一个行星的列表传递给ref函数，然后修改返回的引用单元的内容。

Example 4-4. 使用引用单元来修改数据

let planets =

ref [

"Mercury" ; "Venus" ; "Earth" ;

"Mars" ; "Jupiter" ; "Saturn" ;

"Uranus" ; "Neptune" ; "Pluto"

]

// 哎呀！对不起Pluto...

// 筛选出所有不等于"Pluto"的行星

// 使用获取 ! 来获取planets引用单元的值

// 然后使用 := 来分配新的值

planets := !planets |> **List**.filter (fun p -> p <> "Pluto")

Note: C#程序员在使用ref类型和Boolean值时应该注意。！x只是获取x的值，而不是对x应用布尔运算的not函数：  
> let x = ref true;;  
val x : bool ref  
> !x;;  
val it : bool = true

F#库中有两个函数，decr和incr，来简化对int ref类型的增减操作：

> let x = ref 0;;

val x : int ref = {contents = 0;}  
> incr x;;  
val it : unit = ()  
> x;;

val it : int ref = {contents = 1;}  
> decr x;;

val it : unit = ()  
> x;;  
val it : int ref = {contents = 0;}

可变记录  
 可变性也可在简单值之外应用；记录的字段也可以被标记为可变的。这允许你以命令式的风格使用记录。要使一个记录字段可变，只需在字段名称前面加上mutable关键字。

下面的例子创建了一个包含可变字段Miles的记录，它能够被修改，就好像它是一个可变变量。现在你可以更新记录字段而不需要克隆整个记录：

> // Mutable record types  
open System  
type MutableCar = { Make : string; Model : string; mutable Miles : int }  
let driveForASeason car =  
let rng = new Random()  
car.Miles <- car.Miles + rng.Next() % 10000;;  
type MutableCar =  
{Make: string;  
Model: string;  
mutable Miles: int;}  
val driveForASeason : MutableCar -> unit

> // Mutate record fields  
let kitt = { Make = "Pontiac"; Model = "Trans Am"; Miles = 0 }  
driveForASeason kitt  
driveForASeason kitt  
driveForASeason kitt  
driveForASeason kitt;;  
val kitt : MutableCar = {Make = "Pontiac";  
Model = "Trans Am";  
Miles = 4660;}  
 正如Uncle曾经说过的，“能力越大，责任越大。”改变值的能力也不例外。幸运的是，在F#中，Fortunately, in F#, it is difficult to get into too much trouble with incorrect mutations because of the type system’s ability to enforce  
correctness.  
Units of Measure  
There are several universal truths in this world: the acceleration of gravity is 9.8 meters  
per second squared, water will boil at over 100 degrees Celsius (at one atmosphere of  
pressure), and any programmer, no matter how talented or careful, will have bugs related  
to units of measure.  
If you are ever writing code that deals with real-world units, you will invariably get it  
wrong. For example, you might pass in seconds when the function takes minutes, or  
mistake acceleration for velocity. The result of these sorts of bugs in software has ranged  
from minor annoyances to loss of life.  
The problem is that if you represent a value with just a floating-point number, you have  
no additional information about what that number means. If I give you a float with  
value 9.8, you have no idea if it is in miles, meters per second, hours, or even megabytes.  
A powerful language feature for combating these dimensional analysis issues is units of  
measure. Units of measure allow you to pass along unit information with a floatingpoint value—float, float32, decimal—or signed integer types in order to prevent an  
entire class of software defects. Consider Example 4-5, which describes a temperature.  
Notice how the parameter temp is encoded to only take fahrenheit values. We will cover  
exactly what float<\_> means later in this section.  
Example 4-5. Converting Fahrenheit to Celsius with units of measure  
[<Measure>]  
type fahrenheit  
let printTemperature (temp : float<fahrenheit>) =  
if temp < 32.0<\_> then

printfn "Below Freezing!"  
elif temp < 65.0<\_> then  
printfn "Cold"  
elif temp < 75.0<\_> then  
printfn "Just right!"  
elif temp < 100.0<\_> then  
printfn "Hot!"  
else  
printfn "Scorching!"  
Because the function only accepts fahrenheit values, it will fail to work with any  
floating-point values encoded with a different unit of measure. Calling the function with  
an invalid unit of measure will result in a compile-time error (and prevent potentially  
disastrous results at runtime):  
> let seattle = 59.0<fahrenheit>;;  
val seattle : float<fahrenheit> = 59.0  
> printTemperature seattle;;  
Cold  
val it : unit = ()  
> // ERROR: Different units  
[<Measure>]  
type celsius  
let cambridge = 18.0<celsius>;;  
[<Measure>]  
type celsius  
val cambridge : float<celsius> = 18.0  
> printTemperature cambridge;;  
printTemperature cambridge;;  
-----------------^^^^^^^^^  
stdin(18,18): error FS0001: Type mismatch. Expecting a  
float<fahrenheit>  
but given a  
float<celsius>.  
The unit of measure 'fahrenheit' does not match the unit of measure 'celsius'  
Units of measure also can be compounded by multiplication or division. So if you divide  
a meter unit of measure by another such as second, the result will be encoded as  
float<meter/second>:  
> // Define a measure for meters  
[<Measure>]  
type m;;  
[<Measure>]