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。

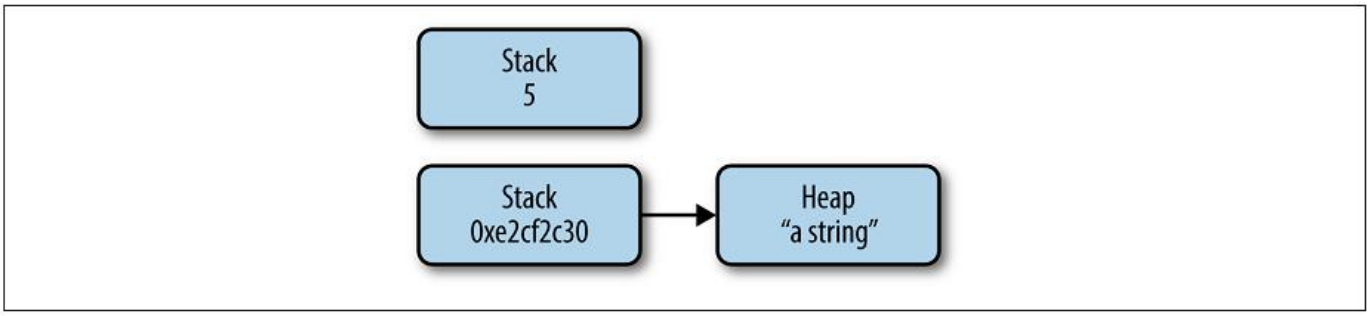

Figure 4-1.值类型与引用类型

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。可变的 variable 是你能改变的东西，可以通过 mutable 关键字来定义。要改变一个可变的值的内容，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 = ()
```

对可变的值有几个限制，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的值 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，因为可变的值只能够在它们被定义的函数中使用。

Example 4-3. 在闭包中使用可变的值造成的错误

```
> // ERROR: 不能在可变的值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。
```

与可变的值有关的两个限制如下：

- 可变的值不能从函数中返回(创建一个副本作为代替)
- 可变的值不能在内部函数(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。