Functional Programming in C#: Classic Programming Techniques for Modern Projects

C#函数式程序设计：经典编程技术在现代项目中的应用

[英] Oliver Sturm 著
吴文国 译

清华大学出版社

C#函数式程序设计：

经典编程技术在现代项目中的应用

[英] Oliver Sturm 著

吴文国 译

清华大学出版社

北 京

Oliver Sturm
Functional Programming in C#：Classic Programming Techniques for Modern Projects
EISBN：978-0-470-74458-1

图书在版编目(CIP)数据

C#函数式程序设计：经典编程技术在现代项目中的应用/(英)斯图姆(Sturm, O.) 著；吴文国 译. —北京：清华大学出版社，2013.1

书名原文：Functional Programming in C#：Classic Programming Techniques for Modern Projects

ISBN 978-7-302-30234-6

Ⅰ. ①C… Ⅱ. ①斯… ②吴… Ⅲ. ①C 语言—程序设计 Ⅳ. ①TP312

中国版本图书馆 CIP 数据核字(2012)第 229574 号

责任编辑：王　军
装帧设计：牛艳敏
责任校对：邱晓玉
责任印制：沈　露

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　邮　　编：100084
社 总 机：010-62770175　　邮　　购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：清华大学印刷厂
装 订 者：北京市密云县京文制本装订厂
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　　张**：16.75　　**字　　数**：402 千字
版　　次：2013 年 1 月第 1 版　　**印　　次**：2013 年 1 月第1 次印刷
印　　数：1～4000
定　　价：45.00 元

产品编号：035598-01

作 者 简 介

Oliver Sturm 有 20 多年的专业软件开发经验。他是应用程序体系结构、程序设计语言和 DevExpress 开发的第三方.NET 工具等多个领域的专家。自 2002 年开始，他的主要兴趣在于.NET 平台。Oliver 曾在许多国际会议上发表过演说，编写了 20 多个培训课程，并在杂志上用英语和德语发表了 100 多篇文章。他也曾从事计算机基础编程教学 15 年之久。由于他对.NET 社区所做的贡献，因此多次获得微软英国最佳 C#程序员称号。

以苏格兰为据点，Oliver 主要从事自由咨询师和培训师的工作，同时还是国际咨询公司 thinktecture 的顾问。他的个人博客是 www.sturmnet.org/blog，商业网址是 www.oliversturm.com，电子邮件地址是 oliver@oliversturm.com。

译 者 简 介

吴文国，博士，温州大学物理与电子信息学院副教授。其研究方向是计算机图形学和地球物理及探测技术，主要从事面向对象程序设计、数据结构等基础课程的教学工作。他工作之余还从事软件开发和翻译工作，已翻译出版了《交互式计算机图形学——基于 OpenGL 的自顶向下方法(第 4 版)》、《UNIX 原理与应用(第 4 版)》等 10 多本计算机图书。另外，他还在《计算机辅助设计与图形学学报》、《中国物理快报》、《电子学报》等杂志上发表过多篇文章。

前　言

函数式设计是一种重要的程序设计模式，它可以追溯到很久之前。函数式程序设计总是与教授程序设计的人们有关。函数式程序设计的整洁而富有逻辑的概念是它特别适合于教学的重要原因。广泛使用计算机和自己设计程序的行业也发现函数式程序设计是实现其目标最有效的办法。然而，在许多所谓的“主流”软件公司看来，函数式程序设计一直以来只具有学术研究价值，他们普遍选择传统的指令式设计方法，如面向对象等。

最近几年，在.NET 平台上把越来越多的函数式成分增加到指令式语言中。在 Visual Studio 2010 中增加了 F#语言，它是用微软主流开发平台开发的第一个混合的函数式语言。甚至有更多的函数式功能被引入到 C#和 VB.NET 中，这说明了微软公司对函数式设计的认同。

本书读者对象

C#函数式设计这个主题可以从两个不同的角度来讨论。很多有经验的程序员和开发团队谙熟.NET 平台，他们一直用 C#或 VB.NET 语言(有时甚至用 C++)为.NET 平台开发软件。如果读者具有这样的经验，则有很多理由需要深入研究函数式设计：它是一个整洁的、易于维护的设计模式，正如我们所了解的，它是程序设计的一个重要基础。某些特殊的情形(如并行化)借助于函数式程序设计的思想很容易实现。

另一方面，读者也许不是.NET 程序员，但在一个或多个传统函数式语言上有相当丰富的经验，需要与 C#程序员合作开发软件，或者想自己使用函数式语言。本书将帮助读者理解如何在 C#语言中使用自己熟悉的方法，当需要向没有函数式设计背景的开发团队解释这些思想时，这也许可以提供一个宝贵的起点。

本书假定读者对 C#语言的结构已有一个基本的了解，至少对 3.0 版本之前的语言比较熟悉。然而，第 II 部分要介绍函数式语言的几个特殊功能，这些功能特别重要，也相当复杂或容易引起误解。根据本人的经验，即使读者精通 C#语言，也建议仔细阅读这部分的内容——这一部分介绍的内容包括一些平时很少接触到的、比较复杂的“疑难杂症”，它们往往会引起以后的误解。

本书主要内容

本书绝大部分例子通过微软.NET 平台上的 C# 4.0 语言来实现。少数几个例子采用其他语言，但是它们只是起演示作用。如果读者想测试这些例子，但是当前使用的并不是 C# 4.0 或

Visual Studio 2010 版本，则用 C# 3.0 或 Visual Studio 2008 也能得到同样的效果——在 C# 4.0 中，新增加的功能并不是很多，而且这些功能都没有应用到例子中。但是，有几个例子利用了.NET Framework 的功能(如 Parallel Extensions)，这些功能只出现在.NET 4.0 中。

本书介绍函数式程序设计的基本概念，以及如何把这些概念应用到 C#语言中。作者尽量提供具有实用背景的示例，但是大多数例子只考虑到语言因素。函数式程序设计是一种与代码、算法和程序结构有关的技术——这一点不同于程序的体系结构。当然，它需要与程序的体系结构相兼容。须知，有时很难在太理论化与偏离重点之间做到理想的平衡，但是作者已尽了最大的努力。

在编写本书时，作者专门开发了一个函数式的辅助代码库，即 FCSlib(Functional CSharp Library)。读者在自己的项目中可以随意使用这个库，但是需要指出的是，该库无法提供任何保证。包含这个库代码的下载文件(有关下载文件的更多信息，请参阅“源代码”一节的内容)中还包括一个应用于 FCSlib 代码的 LGPL 许可文件。

本书的结构

本书由 4 个部分组成。第 I 部分分别从历史和当前的角度概括地介绍函数式程序设计思想。第 II 部分介绍 C#背景知识，它们是理解后面比较复杂的例子所必需的。即使读者掌握了 C#语言，作者也建议仔细阅读这部分内容——这一部分确实包含了非常基本的内容，但是这些内容并不是针对新手的语言引论。第III部分是本书最重要的部分。它用 10 章内容从 C#角度介绍函数式程序设计的各个主题，并用大量的示例和代码片段进行说明。本书的配套代码库 FCSlib 就建立在这部分介绍的思想之上。最后，第Ⅳ部分归纳了在 C#中应用函数式程序设计存在的一些实际问题。作者选择了几个具体示例，说明了现有产品和技术中存在的函数式程序设计思想。

使用本书的要求

本书的全部代码都已经在 Visual Studio 2010、C# 4.0 和.NET 4.0 上测试过。其中很多代码是在 C# 3.0 上开发的，因此这些代码在.NET 3.5 上也能正常运行。如果要在更早的版本上运行，则需要修改——在很多情况下，其概念可以转换为 C# 2.0，但这个版本中缺少运行这些函数式代码所需要的语言支持。

作者曾尝试在 Mono 平台上生成这些代码，但遗憾的是，每次遇到的编译错误都让作者垂头丧气。如果读者试图在 Mono 平台使用，情况可能会不一样——因为此平台一直都在变化。

源代码

在练习书中的示例时，可以选择手动输入代码或者使用本书附带的源代码文件。书中用到的所有源代码都可以从 www.wrox.com 或 http://www.tup.com.cn/downpage 下载。进入站点 http://www.wrox.com 后，只需找到本书的书名(使用 Search 搜索框或书名列表)，单击本书详细信息页面上的 Download Code 链接，就可以得到本书所有的源代码。

注意：因为很多书的书名都相似，所以用 ISBN 搜索更为容易。本书英文版的 ISBN 是 978-0-470-74458-1。

下载完代码后，用您喜欢的压缩工具把它解压缩。此外，也可以去 Wrox 的主下载页面 www.wrox.com/dynamic/books/download.aspx 找到本书或 Wrox 出版的其他书籍的代码。

勘误表

尽管我们竭尽所能来确保在正文和代码中没有错误，但人无完人，错误难免会发生。如果您在 Wrox 出版的书中发现了错误(例如拼写错误或代码错误)，我们将非常感谢您的反馈。发送勘误表将节省其他读者的时间，同时也会帮助我们提供更高质量的信息。

要找到本书的勘误表页面，可以进入 www.wrox.com，使用 Search 搜索框或书名列表定位本书，然后在本书的详细信息页面上单击 Book Errata 链接。在这个页面上可以查看为本书提交的、Wrox 编辑粘贴上去的所有错误。完整的书名列表(包括每本书的勘误表)也可以从 www.wrox.com/misc-pages/booklist.shtml 上获得。

如果您在本书的勘误页面上没有看到您发现的错误，可以到 www.wrox.com/contact/techsupport.shtml 上填写表单，把您发现的错误发给我们。我们会检查这些信息，如果属实，就把它添加到本书的勘误页面上，并在本书随后的版本中更正错误。

p2p.wrox.com

如果想和作者或同行进行讨论，请加入 http://p2p.wrox.com 上的 P2P 论坛。该论坛是一个基于 Web 的系统，您可以发布有关 Wrox 图书及相关技术的消息，与其他读者或技术人员交流。该论坛提供了订阅功能，当您感兴趣的主题有新帖子发布时，系统会邮件通知。Wrox 的作者、编辑、其他业界专家和像您一样的读者都会出现在这些论坛中。

在 http://p2p.wrox.com 网站上，您会找到很多不同的论坛，它们不但有助于您阅读本书，还有助于开发自己的应用程序。加入论坛的步骤如下：

(1) 进入 http://p2p.wrox.com，单击 Register 链接。

(2) 阅读使用条款，然后单击 Agree 按钮。

(3) 填写加入该论坛必需的信息和其他您愿意提供的信息，单击 Submit 按钮。

(4) 您将收到一封电子邮件，描述如何验证您的账户和完成加入过程。

注意：不加入 P2P 也可以阅读论坛里的消息。但是如果要发布自己的消息，就必须加入。

加入之后，就可以发布新的消息和回复其他用户发布的消息。可以随时在 Web 上阅读论坛里的消息。如果想让某个论坛的新消息以电子邮件的方式发给您，可以单点击论坛列表中论坛名称旁边的 Subscribe to this Forum 图标。

要了解如何使用 Wrox P2P 的更多信息，请阅读 P2P FAQ，其中回答了论坛软件如何使用的问题，以及许多与 P2P 和 Wrox 图书相关的问题。要阅读 FAQ，单击任何 P2P 页面上的 FAQ 链接即可。

目　录

第 I 部分

函数式程序设计引言

- 第 1 章　函数式程序设计简史
- 第 2 章　函数式程序设计思想在现代项目中的应用

第1章 函数式程序设计简史

本章主要内容

- 函数式程序设计简介
- 几种函数式程序设计语言
- 与面向对象程序设计的关系

函数式程序设计很久以前就已经出现了。许多人把 1958 年 LISP 语言的出现作为函数式程序设计的起点。LISP 建立在现有的概念之上，其中最重要的概念是由 Alonzo Church 在 20 世纪 30 年代与 40 年代期间在他的 Lambda 演算中提出的概念。这听起来像数学那样抽象，确实如此，数学思想很容易用 LISP 语言建立模型，这使它成为学术领域中首选的语言。LISP 引入了许多其他概念，这些概念至今仍然对程序设计语言的发展起重要作用。

1.1 函数式程序设计简介

尽管早期的函数式程序设计与 LISP 之间存在紧密的关系，但是它经常被认为是一种可以应用于许多计算机语言的程序设计模式——甚至包括那些原本没有打算与此模式一起使用的语言。顾名思义，函数式程序设计把重点放在函数的应用上。函数式程序设计人员以函数为基本模块来建立新函数，这并不是说没有其他语言成分，而是说函数是程序体系创建的主要构造。

引用透明(referential transparency)是函数式程序设计领域中的一个重要思想。一个引用透明的函数的返回值只取决于传递给它的参数的值。这正好与指令式程序设计的基本思想相反。在指令式程序设计中，程序的状态通常会影响函数的返回值。虽然函数式程序设计和指令式程序设计都用函数(function)这个术语，但是引用透明的函数的数学意义仅存在于函数式程序设计中。这样的函数称为纯函数，没有副作用。

一方面，通常无法判断某个程序设计语言是否是函数式语言；另一方面，很容易看出某一个程序设计语言在多大程度上支持在函数式程序设计模式中经常使用的方法，例如递归法。大多数程序设计语言是在这个意义上实现递归的：在某个特定函数、过程或方法中调用自身。但是，如果编译器或语言的运行时环境也如很多指令式语言那样，利用堆栈跟踪技术跟踪跳转的返回地址，并且无法利用优化技术防止堆栈溢出，则递归的应用就会受到严重的限制。在指令式语言中，通常有专用的语法结构实现循环，而对递归的更进一步的支持并没有受到语言设计者或编译器设计者的重视。

高阶函数在函数式程序设计中也十分重要。高阶函数是指以其他函数为参数的函数，或者返回结果为其他函数的函数。许多程序设计语言在一定程度上支持此功能。甚至 C 语言也有一个语法定义函数类型，用 C 术语来说就是通过函数指针引用函数。显然，这使得 C 语言程序员可以把这些函数指针传递给函数，或者让函数返回函数指针。许多 C 语言库含有搜索和排序函数，它们需要通过高阶函数来实现，需要接受特定数据的比较函数作为参数。其次，C 语言不支持匿名函数(所谓匿名函数是指实时创建的内联函数，如 Lambda 表达式)或者闭包等相关概念。

在下面几章中，将介绍几个与语言功能有关的其他例子，它们有助于了解函数式程序设计的定义。

对于有些程序员而言，函数式程序设计是命令计算机如何操作的一种自然描述方法，它用简洁的语言描述某个问题的特性。读者可能听过这样的说法：函数式程序设计更重视告诉计算机需要解决的问题是什么，但是并不十分强调问题解决的详尽步骤。之所以有这样的说法，是因为函数式程序设计提供了高度抽象。引用透明意味着，程序员的任务就是定义一组函数，并用它们描述某个问题集的求解步骤。在此基础上，计算机确定最佳的求值顺序、可能的并行化时机或者某个函数是否需要求值。

对于另外一些程序员，函数式程序设计并不是他们的起点。他们有过程式、指令式或者面向对象程序设计等背景。有关他们如何处理日常遇到的问题有很多奇闻轶事，这些问题既包括打算要通过程序设计解决的问题，也包括在设计这些程序的过程中遇到的问题。函数式程序设计的思想经常提供非常自然的解决方法，另外从不同角度可以得到相同的结果这个事实更进一步支持了这个观点。

1.2　函数式程序设计语言

函数式程序设计并不是只针对某个特定的程序设计语言。但是，在这个领域中有些程序设计语言已出现很久，它们对函数式程序设计方法的影响并不亚于它们受到函数式程序设计方法的影响。本书绝大部分内容只提供用 C#编写的示例，但是对过去用于函数式程序设计的语言或者最初以函数式程序设计为主要目标的语言有所了解也是有帮助的。

下面是两个简单的 LISP 函数：

```
(defun calcLine (ch col line maxp)
```

```
  (let
   ((tch (if (= col (- maxp line)) (cons ch nil) (cons 46 nil))))
   (if (= col maxp) tch (append (append tch (calcLine ch (+ col 1) line maxp)) tch))
   )
  )

(defun calcLines (line maxp)
  (let*
   ((ch (+ line (char-int #\A)))
      (l (append (calcLine ch 0 line maxp) (cons 10 nil)))
     )
   (if (= line maxp) l (append (append l (calcLines (+ line 1) maxp)) l))
   )
  )
```

这段代码使用的程序设计语言是 Common Lisp，它是 LISP 的一个重要“方言”。这里不要求读者完全理解这段代码所执行的操作。LISP 语言家族的一个比较令人感兴趣的方面是它的结构和语法简洁性。据说，LISP 的 Scheme“方言”比起 Common Lisp 更强调这一点。Scheme 是一个非常简单的 LISP 语言，并且具有极强的扩展功能。但是其基本思想很容易明白：使用最少的语法、少量的关键字和操作符、一眼了然的模块结构。许多我们认为是关键字或其他内置结构的元素，如 defun 或 append，实际上都是宏、函数或过程。对于 LISP 系统而言，它们可能确实可以开箱即用，但是它们并不是编译器魔法。用户可以设计自己的程序或者修改现有的程序。许多程序员并不认为只用标准的圆括号会大大改善代码的可读性，但他们不得不承认这样一个基本系统的优雅性。

下面这段代码用比较新的 Haskell 语言重写了上述两个函数以及算法：

```
calcLine :: Int -> Int -> Int -> Int -> String
calcLine ch col line maxp =
  let tch = if maxp - line == col then [chr ch] else "." in
  if col == maxp
    then tch
    else tch ++ (calcLine ch (col+1) line maxp) ++ tch

calcLines :: Int -> Int -> String
calcLines line maxp =
  let ch = (ord 'A') + line in
  let l = (calcLine ch 0 line maxp) ++ "\n" in
  if line == maxp
    then l
    else l ++ (calcLines (line+1) maxp) ++ l
```

与之前的语言相比，Haskell 代码具有迥异的结构。它用不同类型的括号创建列表复合体。if...then...else 结构是一个内置的结构，而++运算符的作用是在列表的末尾添加内容。函数的类型签名是 Haskell 语言的一个惯常用法，但是它并不是一个严格的要求。一个不容易看出来但是非常重要的区别是：Haskell 属于一种强类型语言，而 LISP 是一种类型可动态创建的语言。由于 Haskell 使用了非常强的类型推断，因此通常没有必要显式告诉编

译器变量的类型，它们的类型会在编译时知道。在 Haskell 与 LISP 之间还有很多不甚明显的区别，但是它们不是本书的重点。

最后，下面是一段用 Erlang 语言编写的代码，代码中使用了几个 Erlang 特有的元素。

```
add(A, B) ->
    Calc = whereis(calcservice),
    Calc ! {self(), add, A, B},
    receive
        {Calc, Result} -> Result
    end.

mult(A, B) ->
    Calc = whereis(calcservice),
    Calc ! {self(), mult, A, B},
    receive
        {Calc, Result} -> Result
    end.

loop() ->
    receive
        {Sender, add, A, B} ->
            Result = A + B,
            io:format("adding: ~p~n", [Result]),
            Sender ! {self(), Result},
            loop();
        {Sender, mult, A, B} ->
            Result = A * B,
            io:format("multiplying: ~p~n", [Result]),
            Sender ! {self(), Result},
            loop();
        Other ->
            io:format("I don't know how to do ~p~n", [Other]),
            loop()
  end.
```

这是学习 Erlang 语言的一个非常简单的例子。然而，这个例子使用了指向 Actor 模型的结构体，Actor 模型基于 Erlang 语言和它的运行时系统提供的并行支持。Erlang 不是严格意义上的函数式语言——io:format 提供的类型有副作用，这在 Haskell 中是不可能的。但是在许多行业应用中，Erlang 由于其稳定性和提供的特殊功能集在当今已成为一个非常重要的角色。

可以看到，如同指令式语言一样，函数式语言有很多不同的形式。从 LISP 的最简单方式到 Haskell 的复杂语法或 Erlang 的特殊功能集，以及介于两者之间的众多语言形式，所有这些为想要选择以函数式语言为根源的语言的程序员提供了很大的选择范围。当前，由于强大的运行时系统的支持，这三类语言系列都可用，即使像 LISP 的“方言”Clojure 也可以在.NET 上使用。这些语言表现出来的思想将在本书后面几章中进一步讨论。