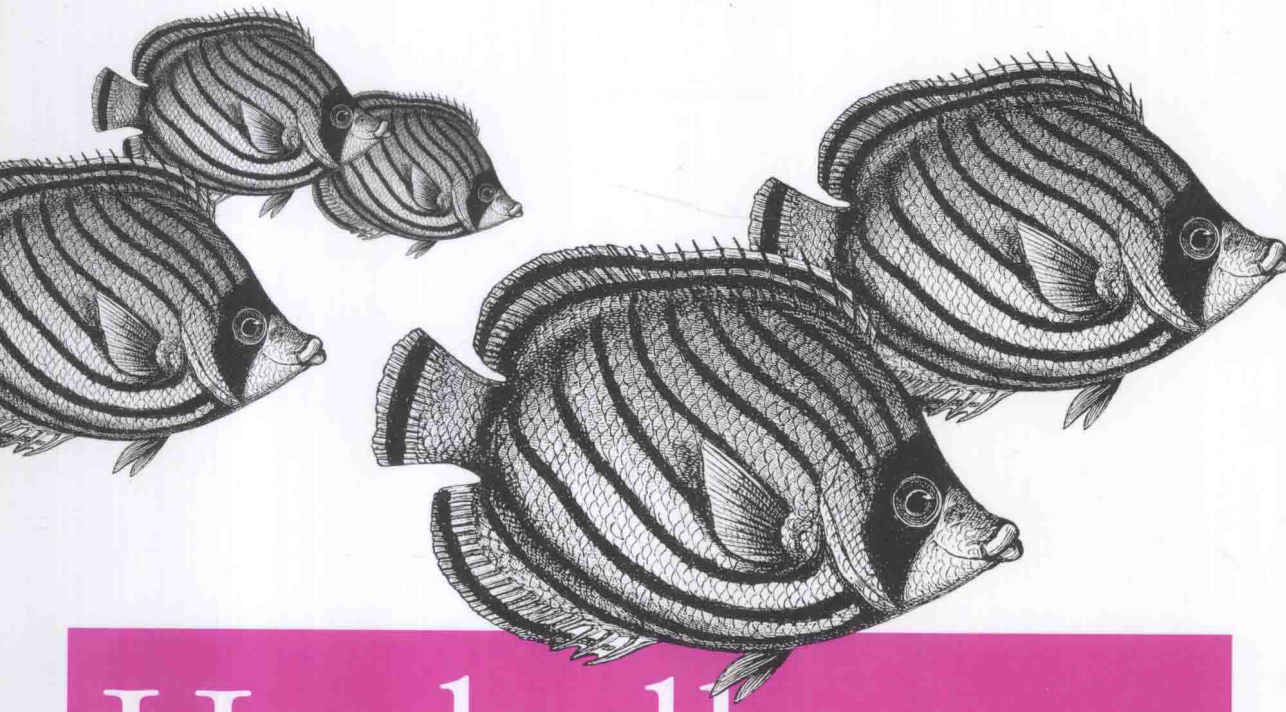
Parallel and Concurrent Programming in Haskell



Haskell 并行与并发编程

[英] *Simon Marlow* 著
喻昌远 译

O'REILLY®
人民邮电出版社
POSTS & TELECOM PRESS

Haskell并行与并发编程

[英] *Simon Marlow* 著

喻昌远 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社

北京

图书在版编目（CIP）数据

Haskell并行与并发编程 / (英) 马洛 (Marlow, S.) 著 ; 喻昌远译. -- 北京 : 人民邮电出版社, 2014.11
书名原文: Parallel and concurrent programming in haskell
ISBN 978-7-115-36718-1

Ⅰ. ①H… Ⅱ. ①马… ②喻… Ⅲ. ①函数－程序设计 Ⅳ. ①TP311.1

中国版本图书馆CIP数据核字(2014)第205862号

版权声明

◆ 著 [英] Simon Marlow
译 喻昌远
责任编辑 杨海玲
责任印制 彭志环 焦志炜

◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路 11 号
邮编 100164 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
三河市海波印务有限公司印刷

◆ 开本：787×1000 1/16
印张：17.25
字数：353 千字 2014 年 11 月第 1 版
印数：1－3 000 册 2014 年 11 月河北第 1 次印刷

著作权合同登记号 图字：01-2013-8794 号

定价：59.00 元

读者服务热线：(010)81055410 印装质量热线：(010)81055316
反盗版热线：(010)81055315

内容提要

本书深入浅出地介绍如何使用 Haskell 语言及相关的库和框架编写并行和并发程序。本书用两个部分分别讲解并行 Haskell 编程和并发 Haskell 编程。根据编程模型的不同，并行部分介绍了 3 种并行编程方式：基于惰性求值的并行（Eval Monad 及求值策略）、基于数据流的并行（Par Monad）以及面向大规模数组算法的并行（Repa 数据并行编程和 Acellerate GPU 编程）。并发部分则按抽象层次由低到高分别涉及线程和 MVar、重叠 I/O、线程的取消和超时、软件事务内存、高级并发抽象、并发网络服务程序、使用线程并行编程和分布式编程等，最后还介绍调试、性能调优以及外部函数接口。书中包含大量可运行的代码示例，并附有详细的注释，读者通过亲身运行、修改和调试代码，可极大地加深对书中内容的理解。

本书适合有一定 Haskell 语言基础的程序员或者对并行或并发编程感兴趣的相关人员阅读。

O'Reilly Media, Inc.介绍

O'Reilly Media通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978年开始，O'Reilly一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了Make杂志，从而成为DIY革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项O'Reilly的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar博客有口皆碑。”

——Wired

“O'Reilly凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O'Reilly Conference是聚集关键思想领袖的绝对典范。”

——CRN

“一本O'Reilly的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照Yogi Berra的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去Tim似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

译者序

十年前，出于好奇，我第一次在真正意义上接触到了 GNU/Linux 操作系统，那时我还刚上大学不久。也正是那一年的暑假，我留在学校，修了 Linux 系统的选修课。整个夏天，随着课程的学习，一个崭新的世界展现在了我的面前。就在这一年，我知道了 Richard Stallman 的自由软件运动和 GNU 项目，被其捍卫软件自由的黑客精神深深震撼，又为 GNU 项目以及其他自由软件所取得的丰硕成果欢欣鼓舞。可以毫不夸张地说，以十年前为起点发生的这一切，对我的大学生活，乃至人生观和价值观，都产生了重大而深刻的影响。

作为影响之一，GNU Emacs 成为了我日常编程开发使用的编辑器。该编辑器最先由 Richard Stallman 开发，历史悠久，功能强大，至今仍然有大量的活跃用户。其与众不同之处在于可以通过名为 Emacs Lisp 的编程语言灵活地实现各种功能。而 Emacs Lisp 是函数式编程语言 Lisp 的一种方言。因此在深入学习 Emacs 的过程中，我了解了 Lisp 这一简洁而优雅的语言，进而对函数式编程产生了浓厚的兴趣。

我第一次听说 Haskell 还是因为 Perl 6，由于在校的时候选修了 Perl 语言的课程，所以那时对 Perl 6 比较感兴趣，而当时最完整的 Perl 6 实现是用 Haskell 编写的。真正开始 Haskell 的学习已经是毕业之后的事情了，那时我希望找到一种表达能力强，性能也不错的语言，作为以后日常学习和生活使用的编程语言。Haskell 作为函数式语言，有着优异的表达能力，作为静态、编译型语言，可以达到相当不错的性能，尤为重要的是，Haskell 最主要的实现 GHC（Glasgow Haskell Compiler）是自由软件。因此，我开始了 Haskell 语言的学习和使用。值得提一下的是，我本人是数字集成电路设计工程师，而数字电路设计和函数式编程是非常相似的，实际上，在 Hackage 上有不少用来设计数字电路的 Haskell 库或工具。

长久以来，我一直受惠于自由软件，而因此一直希望能为其发展贡献出自己的一份力量。恰逢去年上海 Linux 用户组（SHLUG）的宋方睿同学向人民邮电出版社推荐我参与本书的翻译工作，经过慎重的考虑，我向编辑表达希望翻译本书的意愿。虽然缺乏翻译的经验，但觉得自己经过努力应该能够胜任翻译工作，而且作为一位长期的 Haskell 用户，我也很希望能够为 Haskell 语言的推广作出自己的贡献，回报社区。非常幸运的是，我的努力和能力得到了本书的责任编辑的认可，我得以进行本书的翻译工作。

在翻译的过程中，我首先要感谢的是我的亲人和朋友，感谢他们的理解和支持。

其次，我要感谢宋方睿同学，他本人也是《Haskell 趣学指南》一书的译者之一。除了

感谢他的推荐外，在本书的前期翻译过程中，他也给予了我很大的帮助。

我还要感谢人民邮电出版社的编辑，特别是本书的责任编辑杨海玲，非常感谢她对我信任，还有在贯穿全书的翻译过程中对我的支持和帮助。

最后，限于本人水平有限，译文中错误在所难免，望读者谅解。

喻昌远

2014 年 7 月

前言

作为一名 Glasgow Haskell 编译器（GHC）的开发者近 15 年，我目睹了 Haskell 从一门合适的研究性语言成长为一个繁荣丰富的生态系统。在 GHC 的并行和并发的支持上，我花费了大量的时间。在 1997 年，我重写了其运行时系统（Runtime System），这是该年我对 GHC 做的头几件事情之一。而且当时我们还做了一个关键性的决定，即直接在系统的核心支持并发，而非在额外的可选库或附加库中支持。对于这个决定，我希望是基于敏锐的先见之明，但事实上却很大程度和我们找到一种方法有关，这种方法使得支持并发性的额外开销能减少到接近零（我们一直困扰于性能问题，之前的额外开销在 2%左右）。尽管如此，成为实现的必要部分，意味着并发性在 GHC 中的地位是一等的，而我很确信，正是这个决定给 GHC 带来了稳定且高性能的并发支持。

Haskell 有着与并行性（parallelism）相关联的悠久传统。仅举几个项目为例，有专为并行计算设计的衍生自 Id 语言的并行 Haskell 变体，有在多台机器上运行并行 Haskell 程序的 GUM 系统，以及 GRiP 系统——一个为运行并行函数式程序设计的完整计算机架构。所有的这些都发生在现在的多核革命之前，而问题是当时摩尔定律让我们仍能制造出更快的电脑。并行性的实现非常困难，而当普通的电脑可以以指数级的速度变快时，为此而努力似乎得不偿失。

在 2004 年前后，为了使 GHC 运行时系统能在共享内存的多处理器上运行，我们决定为其开发一个并行实现（parallel implementation），这在之前还未曾做过。这恰好是在多核革命的前夕，当时多处理器的机器还很常见，但多核的仍未出现。这里我再次希望，决定此时解决并行性问题是出于远见卓识，但这个决定却和下面的事实关系更大，即开发内存共享的并行实现是一个有趣的研究性问题，而且听起来很有意思。Haskell 的纯粹是根本性的——这意味着可以避免在运行时系统和垃圾回收器中加锁而造成的部分额外开销，也就是说，可以将使用并行造成的额外开销减小到很少的几个百分点。然而，直到经过了更多的探索研究，调度器的重写，以及一个全新的并行垃圾回收器的使用，这个新实现才真正达到可用的程度，能够加速大量的程序。我在 2009 年国际函数式程序设计大会（International Conference on Functional Programming，ICFP）上提交的论文是这一并行实现从一个有趣的原型变为实用工具的转折点。

所有这些研究和实现的工作都非常有趣，但教导程序员如何使用 Haskell 中的并行与并发特性的高质量资源依然匮乏。在过去的几年，我有机会两次在暑期班讲授 Haskell 并行与并发编程，一次是在 Budapest 举办的 2011 中欧函数式程序设计（CEFP）暑期班，另一次是在法国南部 Cadarache 举办的 2012 CEA/EDF/INRIA 暑期班。在为这些课程准

备材料时，我首次有理由编写一些深入的指南，并且开始收集一些优秀的实例。在 2012 年的暑期班之后，我已经有了 100 页的指南，在一些人（见致谢部分）的鼓励下，我决定将这些内容编写成书。当时我觉得已经完成了一半左右，但实际上可能才接近四分之一。实在有太多内容要写了！希望你能享受我最终的成果。

本书面向的读者

阅读本书需要一些基本的 Haskell 使用知识，而本书中并未涉及这些内容。为此，阅读一本入门书籍，例如《Real World Haskell》（O'Reilly）、《HaskellProgramming in Haskell》（剑桥大学出版社）、《Learn You a Haskell for Greate Good》[①]（No Starch Press）或者《Haskell：The Craft of Functional Programming》（Addison-Wesley），应该是一个很好的开始。

如何阅读本书

本书的主要目标是让读者能够进行并行与并发 Haskell 编程。但是，正如你大概已经知道的，学习编程并非是单独看书就可以的。这正是本书的编写刻意切合实际，包含大量的可以运行、修改和扩展的例子的原因所在。书中有几章包含了练习，建议读者尝试，通过这些练习可以熟悉该章介绍的主题，本人强烈推荐做一些练习，或按自己的想法写些代码。

在探讨本书的主题时，我不会回避陷阱和系统中不完善的部分。Haskell 已经发展了 20 多年，但其在今天的发展比以往任何时刻都更为迅速。因此自然会遇到一些不一致情况和系统中不那么好的部分。本书涉及的一些主题是新近发展起来的：第 4、5、6 和 14 章涉及了一些最近几年开发的框架。

本书包含两个基本独立的部分，第一部分和第二部分。读者可以随意从一部分开始阅读，或者在两部分间切换着阅读（也就是并发地阅读两部分）。两部分中仅有一处存在依赖关系：如果先阅读了第一部分，那么将更容易理解第 13 章，尤其是在 13.2.5 节前，应该先读第 4 章。

尽管两部分是基本独立的，但同一部分内的各章还是应按顺序阅读。本书并非是参考书，书中包含的一些可运行的例子和主题是贯穿数章开发和发展出来的。

本书的排版约定

本书使用如下排版约定。

① 中文版书名《Haskell 趣学指南》已由人民邮电出版社出版。——编者注

- 楷体：用于强调、新术语。
- 等宽字体：表示变量、函数、类型、参数、对象以及其他的程序构成部分。

此图标表示一个小技巧、建议或者一般性注释。

此图标表示需要警惕的陷阱，一般是一些不那么明显的情况。

代码示例如下：

```
timetable1.hs
search :: ( partial -> Maybe solution )   -- ❶
       -> ( partial -> [ partial ] )
       -> partial
       -> [solution]
```

标题是代码片段所在的源代码文件的文件名，该文件可以在示例代码中找到；关于如何获取示例代码，见 1.3 节。当引用同一个文件中的多个片段时，仅第一次会有文件名标题。

❶ 代码片段中常会出现这样的关于某行的注释。

输入 shell 中的命令示例如下：

```
$ ./logger
hello
bye
logger: stop
```

其中$字符是提示符，紧接着是命令，剩下的几行是命令产生的输出。

GHCi 会话示例如下：

```
> extent arr
(Z :. 3) :. 5
> rank (extent arr)
2
> size (extent arr)
15
```

由于 GHCi 的默认提示符会在导入几个模块后显得过长，本人常将 GHCi 的提示符设定为>后面接着一个空格。通过在 GHCi 中使用下列命令，可以完成该设置：

```
Prelude> :set prompt "> "
>
```

示例代码的使用

随书附带的示例代码可在线获取，具体如何获取和构建请参见 1.3 节。关于示例代码的使用、修改和再发布的权利的相关信息，请参见示例代码发布包中的 LICENSE 文件。

联系我们

如果你想就本书发表评论或有任何疑问，敬请联系出版社。

美国：

O'Reilly Media Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

我们为本书提供了专门的网页，上面有勘误表、示例，以及其他额外的信息，可以通过 http://oreil.ly/parallel-concurrent-prog-haskell 访问该网页。

若要留言或询问关于本书的技术问题，请发送电子邮件到 bookquestions@oreilly.com。

想了解更多关于我们的书籍、课程、会议，以及新闻等信息，请登录我们的网站：http://www.oreilly.com。

我们的其他联系方式如下。

Facebook: http://facebook.com/oreilly
Twitter: http://twitter.com/oreillymedia
YouTube:http://www.youtube.com/oreillymedia

致谢

有好几个月的时间我满脑子都是并行和并发 Haskell 而无法容下其他东西，所以我最先也是最重要的是要感谢我的妻子，感谢她对我的鼓励、她的耐心，以及在写书时提供的蛋糕。

其次，本书的编写很大程度上要归功于 Simon Peyton Jones，他自 GHC 诞生之初就开始领导该项目，他坚持不懈的热情和对技术的深刻了解是 GHC 发展的持续推动力。他一直是我最丰富的灵感之源。

我还要感谢 Mary Sheeran 和 Andres Löh，他们说服我将授课笔记编写成书。感谢 CEPH 和 CEA/EDF/INRIA 暑期班的组织者邀请我授课，让我有了开始的动力。感谢作为我的“试验品”，上我课的学生们。

非常感谢本书的编辑 Andy Oram 及 O'Reilly 的其他人，在他们的帮助下，本书才得以出版。

下面这些人都以某种方式对本书的编写提供了帮助，他们或者是审查了早期的书稿、给我提出了建议、给在线章节留言，或者是我借用了他们写的代码（希望已经进行了署名），或者用了他们撰写的论文或博客中的想法，或者是其他的帮助（若遗漏了一些人的话，实在抱歉）：Joey Adams、Lennart Augustsson、Tuncer Ayaz、Jost Berthold、Manuel Chakravarty、Duncan Coutts、Andrew Cowie、Iavor Diatchki、Chris Dornan、Sigbjorn Finne、Kevin Hammonad、Tim Harris、John Hughes、Mikolaj Konarski、Erik Kow、Chris Kuklewicz、John Launchbury、Roman Leshchinskiy、Ben Lippmeier、Andres Löh、Hans-Wolfgang Loidl、Ian Lynagh、Trevor L. McDonell、Takayuki Muranushi、Ryan Newton、Mary Sheeran、Wrenng Thornton、Bryan O'Sullivan、Ross Paterson、Thomas Schilling、Michael Snoyman、Simon Thomson、Johan Tibell、Phil Trinder、Bas Van Dijk、Phil Wadler、Daniel Winograd-Cort、Nicolas Wu 和 Edward Yang.

最后，我要感谢 Haskell 社区，这是我遇到的最友好、包容、有帮助以及鼓舞人心的在线开源社区之一。朋友们，我们有很多值得骄傲的地方，坚持下去。

目录

第二部分　并发 Haskell

第 1 章

绪论

长久以来，编程社区已熟知线程和锁难以使用。即使很简单的问题也常常需要过高的专业技能，而且会导致难以诊断的故障。然而，线程和锁依然十分通用，从并行图像处理到并发 Web 服务器，足以表达出所需要编写的任何内容，并且有一个专一的通用 API 也有不可否认的好处。但是，若想让并行和并发编程变得更容易，就要接受“不同的问题要用不同的工具解决”的思想，一种工具无法解决所有的问题。图像处理可以自然地表达成并行矩阵运算，而线程则很好地适用于并发 Web 服务器的情况。

因此在 Haskell 中，我们的目标是尽可能为多种工作提供正确的工具。若发现一项工作在 Haskell 中没有正确的对应工具，则会尝试找出一种方法将其制造出来。工具的多样化不可避免会带来负面影响，即需要大量的学习，而这正是本书所要进行讲解的。在本书中，将会讨论如何用 Haskell 编写并行和并发程序，从简单地利用并行来加速计算密集型程序，到使用轻量级线程编写高速并发的网络服务程序。在这个过程中，还将会看到如何使用 Haskell 编写程序，让其运行在现代图形显卡中强大的处理器（GPU）上，也会看到如何编写程序，使之能够在网络中的多台机器上运行（分布式编程）。

这并非说本人打算对每一种最新出现的试验性编程模型都进行介绍。若仔细查看 Hackage 上的软件包，可以发现各式各样用于并行和并发编程的库，其中许多是为了特定目的而开发的，更不用说所有的未到实用阶段的研究性项目。在本书中，我将重点关注现在已能用来完成工作，足够稳定，并在生产中可信赖的 API。此外，本人的目标是让读者牢固地掌握最底层的工作原理，以便在需要的时候能够在此之上搭建出自己的抽象结构。

1.1 术语：并行性和并发性

在许多领域*并行*和*并发*是同义词，但在编程中则不然，它们被用来描述在根本上完全

不同的两个概念。

*并行*程序是指使用多个硬件参与计算（如多个处理器核心）使之更快的程序。目标是通过将计算的不同部分分配给不同的处理器，使之能够同时执行，从而更早得到问题的答案。

与之相对的，*并发*则是一种包含多个*控制线程*的程序构成技术。从概念上说，这些控制线程是“同时”被执行的，也就是说，用户所观察到的最终影响，是由这些线程交替作用产生的。到底是否真的是同时执行则属于实现上的细节。并发程序可以通过交替的方式在单个处理器上执行，或在多个物理处理器上执行。

并行编程仅关注效率，而并发编程则关注程序的构成，使之能够和多个独立的外部媒介交互（如用户、数据库或一些外部客户）。并发性使得这类程序能够模块化，和用户交互的线程可以明确地区分于和数据库通信的线程。在没有并发的情况下，这类程序必须通过事件循环和回调的方式编写，与线程所提供的相比，通常更加低效且模块化不足。

在纯函数式的程序中，“控制线程”这个概念是没有意义的，因为没有需要观测的效果，而且程序的结果也和求值的顺序无关。因此，并发性是用于构成产生效果的代码的技术；在 Haskell 中，即 IO monad 中的代码。

一个与之相关的内容是*确定性*和*非确定性*编程模型的区分。确定性编程模型中每个程序只会给出一个结果，而非确定性编程模型则容许——取决于执行时某些情况——程序可以有不同的结果。由于必须和外部媒介交互，而这些交互会导致事件在不可预测的时刻发生，所以并发编程模型必然是非确定性的。而非确定性有一些显著的缺点：程序将会变得非常难以测试和推断。

对于并行编程来说，应尽可能地使用确定性编程模型。既然目标仅仅是更快地获得答案，那么还是不要让程序在这个过程中变得更难调试。确定性并行编程是这两方面的最佳结合：测试、调试和推断可以以顺序的方式执行，但加入更多处理器后程序则能运行得更快。事实上，大多数计算机处理器自身通过流水线和多个执行单元的形式实现了确定性并行。

虽然可以使用并发的方式实现并行编程，但这通常不是一个明智的选择，因为并发的使用牺牲了确定性。在 Haskell 中，大多数的并行编程模型都是确定性的。然而，注意到确定性编程模型并不足以表达所有的并行算法是非常重要的，有些算法是依赖于内部不确定性的，特别是要对解空间进行搜索的问题。此外，有时希望并行化确实有副作用的程序，那就别无选择，只能使用非确定性并行编程或并发编程了。

最后，希望在同一个程序中混合使用并行和并发编程是完全合理的。大多数的交互式程序需要使用并发来维持一个快速响应的用户界面，与此同时在后台执行计算密集型的任务。