The Pragmatic Programmers

图灵程序设计丛书 TURING

“Erlang是目前唯一成熟可靠的能够开发高扩展性并发软件系统的语言，它将成为下一个Java。”

——Ralph Johnson，软件开发大师，《设计模式》作者之一

Erlang程序设计

Programming Erlang Software for a Concurrent World

- Erlang之父权威著作
- 领先一步，精通下一代主流编程语言
- 从这里开始，拥抱未来

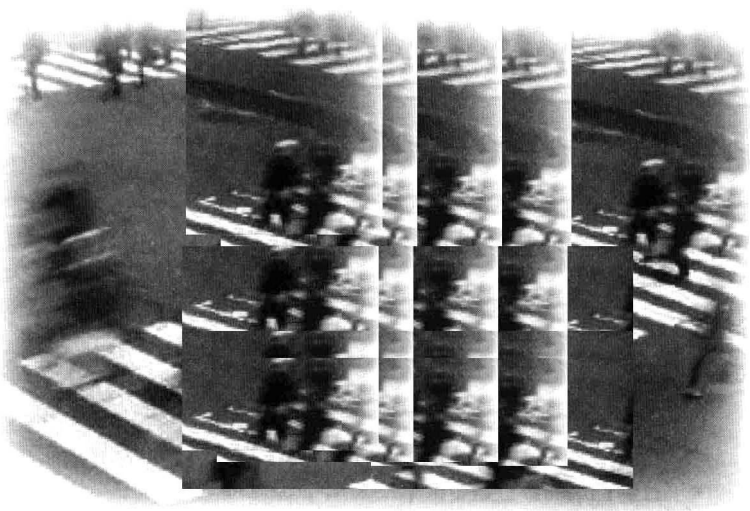
[瑞典] Joe Armstrong 著
赵东炜 金尹 译

人民邮电出版社
POSTS & TELECOM PRESS

TURING 图灵程序设计丛书

Erlang程序设计

Programming Erlang Software for a Concurrent World



[瑞典] Joe Armstrong 著
赵东炜 金尹 译

人民邮电出版社
北 京

图书在版编目（CIP）数据

Erlang 程序设计 /（瑞典）阿姆斯特朗（Armstrong, J.）著；赵东炜，金尹译. —北京：人民邮电出版社，2008.12（2009.1 重印）
（图灵程序设计丛书）
书名原文：Programming Erlang: Software for a Concurrent World
ISBN 978-7-115-18869-4

Ⅰ. E… Ⅱ. ①阿…②赵…③金… Ⅲ. 程序语言 — 程序设计 Ⅳ. TP312

中国版本图书馆CIP数据核字（2008）第142666号

内 容 提 要

本书是讲述下一代编程语言 Erlang 的权威著作，主要涵盖顺序型编程、异常处理、编译和运行代码、并发编程、并发编程中的错误处理、分布式编程、多核编程等内容。本书将帮助读者在消息传递的基础上构建分布式的并发系统，免去锁与互斥技术的羁绊，使程序在多核 CPU 上高效运行。本书讲述的各种设计方法和行为将成为设计容错与分布式系统中的利器。

图灵程序设计丛书

Erlang程序设计

◆ 著 [瑞典] Joe Armstrong
译 赵东炜 金 尹
责任编辑 傅志红
执行编辑 杨 爽

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子函件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京顺义振华印刷厂印刷

◆ 开本：800×1000 1/16
印张：27.75
字数：691 千字 2008 年 12 月第 1 版
印数：3 001 – 5 000 册 2009 年 1 月北京第 2 次印刷

著作权合同登记号 图字：01-2008-3858号

ISBN 978-7-115-18869-4/TP

定价：79.00元

读者服务热线：(010)88593802 印装质量热线：(010)67129223
反盗版热线：(010)67171154

版 权 声 明

这个世界是并行的。

如果希望将程序的行为设计得与真实世界物体的行为相一致，那么程序就应具有并发结构。

使用专门为并发应用设计的语言，开发将变得极为简便。

Erlang 程序模拟了我们如何思考，如何交互。

——Joe Armstrong

推 荐 序

Erlang算不上是一种“大众流行”的程序设计语言，而且即使是Erlang的支持者，大多数也对于Erlang成为“主流语言”并不持乐观态度。然而，自从2006年以来，Erlang语言确实在国内外一批精英程序员中暗流涌动，光我所认识和听说的，就有不少于一打技术高手像着了魔一样迷上了这种已经有二十多年历史的老牌语言。这是一件相当奇怪的事情。因为就年龄而言，Erlang大约与Perl同年，比C++年轻四岁，长Java差不多十岁，但Java早已经是工业主流语言，C++和Perl甚至已经进入其生命周期的下降阶段。照理说，一个被扔在角落里二十多载无人理睬的老家伙合理的命运就是坐以待毙，没想到Erlang却像是突然吃了返老还童丹似的在二十多岁的“高龄”又火了一把，不但对它感兴趣的人数量激增，而且还成立了一些组织，开发实施了一些非常有影响力的软件项目。这是怎么回事呢？

根本原因在于Erlang天赋异禀恰好适应了计算环境变革的大趋势：CPU的多核化与云计算。

自2005年C++标准委员会主席Herb Sutter在*Dr. Dobb's Journal*上发表《免费午餐已经结束》一文以来，人们已经确凿无疑地认识到，如果未来不能有效地以并行化的软件充分利用并行化的硬件资源，我们的计算效率就会永远停滞在仅仅略高于当前的水平上，而不得动弹。因此，未来的计算必然是并行的。Herb Sutter本人曾表示，如果一个语言不能够以优雅可靠的方式处理并行计算的问题，那它就失去了在21世纪的生存权。“主流语言”当然不想真的丧失掉这个生存权，于是纷纷以不同的方式解决并行计算的问题。就C/C++而言，除了标准委员会致力于以标准库的方式来提供并行计算库之外，标准化的OpenMP和MPI，以及Intel的Threading Building Blocks库也都是可信赖的解决方案；Java在5.0版中引入了意义重大的concurrency库，得到Java社区的一致推崇；而微软更是采用了多种手段来应对这一问题：先是在.NET中引入APM，随后又在Robotics Studio中提供了CCR库，最近又发布了Parrallel FX和MPI.NET，可谓不遗余力。然而，这些手法都可以视为亡羊补牢，因为这些语言和基础设施在创造时都没有把并行化的问题放到优先的位置来考虑。与它们相反，Erlang从其构思的时候起，就把“并行”放到了中心位置，其语言机制和细节的设计无不从并行角度出发和考虑，并且在长达二十年的发展完善中不断成熟。今天，Erlang可以说是为数不多的天然适应多核的可靠计算环境，这不能不说是一种历史的机缘。

另一个可能更加迫切的变革，就是云计算。Google的实践表明，用廉价服务器组成的服务器集群，在计算能力、可靠性等方面能够达到价格昂贵的大型计算机的水准，毫无疑问，这是大型、超大型网站和网络应用梦寐以求的境界。然而，要到达这个境界并不容易。目前一般的网站为了达成较好的可延展性和运行效率，需要聘请有经验的架构师和系统管理人员，手工配置网络服务

端架构，并且常备一个高水准的系统运维部门，随时准备处理各种意外情况。可以说，虽然大多数Web企业只不过是想在这些基础设施上运行应用而已，但仅仅为了让基础设施正常运转，企业就必须投入巨大的资源和精力。现在甚至可以说，这方面的能力成了大型和超大型网站的核心竞争力。这与操作系统成熟之前人们自己动手设置硬件并且编写驱动程序的情形类似——做应用的人要精通底层细节。这种格局的不合理性一望便知，而解决的思路也是一目了然——建立网络服务端计算的操作系统，也就是类似Google已经建立起来的“云计算”那样的平台。所谓“云计算”，指的是结果，而当前的关键不是这个结果，而是作为手段的“计算云”。计算云实际上就是控制大型网络服务器集群计算资源的操作系统，它不但可以自动将计算任务并行化，充分调动大型服务器集群的计算能力，而且还可以自动应对大多数系统故障，实现高水平的自主管理。计算云技术是网络计算时代的操作系统，是绝对的核心技术，也正因此，很多赫赫有名的中外大型IT企业都在不惜投入巨资研发计算云。包括我在内的很多人都相信，云计算将不仅从根本上改变我们的计算环境，而且将从根本上改变IT产业的盈利模式，是真正几十年一遇的重大变革，对于一些企业和技术人员来说是重大的历史机遇。恰恰在这个主题上，Erlang又具有先天的优势，这当然也是归结于其与生俱来的并行计算能力，使得开发计算云系统对于Erlang来说格外轻松容易。现在Erlang社区已经开发了一些在实践中被证明非常有效的云计算系统，学习Erlang和这些系统是迅速进入这个领域并且提高水平的捷径。

由此可见，Erlang虽然目前还不是主流语言，但是有可能会在未来一段时间发挥重要的作用，因此，对于那些愿意领略技术前沿风景的“先锋派”程序员来说，了解和学习Erlang可能是非常有价值的投资。即使你未来不打算使用Erlang，也非常有可能从Erlang的设计和Erlang社区的智慧中得到启发，从而能够在其他语言的项目中更好地完成并行计算和云计算相关的设计和实现任务。再退一步说，就算只是从开启思路、全面认识计算本质和并行计算特性的角度出发，Erlang也值得了解。所以，我很希望这本书在中国程序员社区中不要遭到冷遇。

本书是由Erlang创造者Joe Armstrong亲自执笔撰写的Erlang语言权威参考书，原作以轻松引导的方式帮助读者在实践中理解Erlang的深刻设计思路，并掌握以Erlang开发并行程序的技术，在技术图书中属于难得的佳作。两位译者我都认识，他们都是技术精湛而思想深刻的“先锋派”，对Erlang有着极高的热情，因此翻译质量相当高，阅读起来流畅通顺，为此书中译本添色不少。有兴趣的读者集中一段时间按图索骥，完全有可能就此踏上理解Erlang、应用Erlang的大路。

孟岩

CSDN首席分析师兼《程序员》杂志技术主编

2008年10月

译 者 序

另辟蹊径——从容面对容错、分布、并发、多核的挑战。

作为一名程序员，随着工作经验的增长，如果足够幸运的话，终有一日，我们都将会直面大型系统的挑战。最初的手忙脚乱总是难免的，经历过最初的迷茫之后，你会惊讶地发现这是一个完全不同的“生态系统”。要在这样的环境中生存，我们的代码需要具备一些之前我们相当陌生或者闻所未闻的“生存技能”。容错、分布、负载均衡，这些词会频繁出现在搜索列表之中。经历过几轮各种方案的轮番上阵之后，我们会开始反思这一系列问题的来龙去脉，重新审视整个系统架构，寻找瓶颈所在。你可能会和我一样，最终将目光停留在那些之前被认为是无懈可击的优美代码上。开始琢磨：究竟是什么让它们在新的环境中“水土不服”，妨碍其更加有效地利用越来越膨胀的计算资源？

其实，早在多年以前硬件领域上的革命就已经开始，现在这个浪潮终于从高端应用波及常规的计算领域——多核芯片已经量产，单核芯片正在下线——这场革命正在我们的桌面上演。时代已经改变，程序员们再也不能继续稳坐家中装作什么事情也没发生，问题已经自己找上门来了。由单核CPU频率提升带来软件自动加速的时代已经终止，性能的免费大餐已经结束，“生态环境的变化”迫使代码也必须“同步进化”。锁、同步、线程、信号量，这些之前只是在教科书中顺带提及的概念，越来越多地出现在我们日常的编程中，接踵而至的各类问题也开始折磨我们的神经。死锁、竞态，越来越多的锁带来了越来越复杂的问题。

在多核CPU的系统上，程序的性能不增反降，或者暴露出隐藏的错误？

在更强的硬件平台上，程序的并发处理能力却没有得到提升，瓶颈在哪里？

在分布式计算网络中，不得不对程序结构进行重大调整，才能适应新的运行环境？

在系统的关键应用上，不得不为软件故障或者代码升级，进行代价高昂的停机维护？

……

这一系列的问题，归根结底，都是因为主流技术体系在基本模型上存在着与并发计算相冲突的设计。换句话说，问题广泛地存在于我们所写的每一行代码中。在大厦初具规模时，却猛然发现每一块砖石都不牢固，这听来很有一些耸人听闻，但这种事并不罕见。

比如：x = x + n，即使是这个再常见不过的语句，也暗藏着烦恼的根源（你也可以把它称做共享内存陷阱）。从机器指令的角度来思考，这个语句可能做了这么几件事（仅仅只是在概念上）。

(1) mov ax，[bp+x]；将寄存器ax赋值为变量x所指示的内存中的数据。

(2) mov bx，n；将寄存器bx赋值为n。

(3) add ax, bx; 将寄存器ax加n。

(4) mov [bp+x],ax; 将变量x所指示的内存赋值为寄存器ax中的数据。

理论上，这是一个原子操作，在单核CPU下情况也确实如此，但在多核CPU下，就全然不同了。如果在每个核心上都有一个正在执行上述代码的进程（也就是试图对这个代码执行并行计算），问题就出现了。这里的x是在两个进程之间共享的内存。很明显，在第(1)步到第(4)步之间，需要某种机制来保证“某一时刻”只有一个进程正在执行，否则就会破坏数据的完整性。这也就意味着，这样的代码无法充分利用多核CPU的运算能力。也罢，咱们不加速就是了，但更糟糕的是，为了保证不出错，还需要引入锁的机制来避免数据被破坏。现有的主流技术体系全都建立在共享内存的模型之上，像x = x + n这样的代码几乎无处不在，但在多核环境下，每一处这样的代码（逻辑上的或者事实上的）都需要小心地处理锁。更糟糕的是，大量的锁彼此互相影响又会导致更为复杂的问题。这迫使程序员们在实现复杂的功能之余，还要拿出极度的耐心和娴熟的技巧来处理好这些沉闷和易错的锁，且不说是否可能，但这至少也是一个极为繁重的额外负担。

Erlang为了并发而生。20多年前，它的创建者们就已经意识到了这一问题，转而选择了一条与主流语言完全不同的路（还有为数不多的另外几种语言也是如此）。它采用的是消息模型，进程之间并不共享任何数据，因而也就完全地避免了引入锁的必要。对于多核系统而言，完全无锁，也就意味着相同的代码在更多核心的CPU上会很容易具有更高的性能，而对于分布式系统，则意味着尽可能地避免了顺序瓶颈，可以把更多的机器无缝地加入到计算网络中来。甩掉了锁的桎梏，无疑是对程序员们的解放。

不仅如此，Erlang在编程模型上走得更远。它在语言级别提供了一系列的并发原语，通过这些原语，我们可以用进程＋消息的模型来建模现实世界中多人协作的场景。一个进程表示一个人，人与人之间并不存在任何共享的内存，彼此之间的协作完全通过消息（说话、打手势、做表情，等等）交互来完成。这正是我们每一个人生而知之的并发模式！软件模拟现实世界协作和交互的场景——这也就是所谓的COP（面向并发编程）思想。

在错误处理上，Erlang也有与众不同的设计决策，这使得实现“容错系统”不再遥不可及。COP假设进程难免会出错——不像其他某些语言一样假设程序不会出错。它假设程序随时可能会出错，如果发生出错的情况，则不要尝试自行处理，而是直接退出，交给更高级的进程来对这种情况进行处理。通过引入“速错”和“进程监控”的概念，我们将错误分层，并由更高层的进程来妥善处理（比如，重启进程，或重启一系列进程）。有了这样的概念作为支撑，构造“容错系统”就会变得易如反掌。在这样的系统之下，软件错误不会导致整个系统的瘫痪，发现错误也无须停机就可直接更新代码，在配置了备份硬件之后，硬件的错误也不会影响服务的正常运行。这么做的结果相当惊人，使用Erlang的电信关键产品，达到了传说中的99.9999999%可用性（即9个9的最高可用性标准）。

Erlang采用虚拟机技术实现，用它编写的程序可以不经修改直接运行在从手机到大型机几乎所有的计算平台之上。这是一项有着20多年历史的成熟技术，有着相当多的成熟库（OTP）和开源软件，这些资产使得它有极高的实用价值。Erlang本身也是开源软件，这扫清了对于语言本身生命力的疑惑。Erlang还是一个充满活力的语言，在它的社区，常常能够见到Joe Armstrong等语

言的创建者在回答问题，这一点尤其宝贵。在熟悉了Erlang的思维方法和社区之后，很多人都发出了相见恨晚的感慨。

虽说对于并发而言，Erlang确实是非常好的选择，但这么多年以来，业界对于并发预料之中的增长却一直没有真正发生。此前，这类应用更多地局限在一些相对高端的领域，而Erlang身上浓厚的电信背景，又使得第一眼看来它似乎只适用于电信行业（实际情况远非如此）。长期以来Erlang的使用群体仅局限在一个狭小的技术圈子之内，它处于“非主流语言”的边缘位置已经很久了。这种情况直到最近才有所改观，最近两年，Google的成功使得其引为核心的大规模分布式应用模式广为人知，而多核CPU进入桌面也迫使“工业主流”开始认真对待并发计算。直到此时，解决这类问题最为成熟的Erlang技术，才因为其难以忽视的优势而引起人们的广泛关注。

从历史的眼光来看，在计算机语言的荣誉堂内，上演着一代又一代程序设计语言的繁荣和更替，潮来潮往让人难以捉摸。这与其说是技术，还不如说是时尚。对于Erlang这种有些怪异的小众语言来说，是否真的会成为“下一个Java”？实难预测，而且也不重要。但是有一点已经毫无疑问，那就是“下一代语言”至少也要像Erlang一样，处理好与并发相关的一系列问题（或者做得更好）。也许将来的X++（或X#）语言在吸收了它的精髓之后，又会成为新的工业主流语言。但即便如此，先跟随本书作者开辟的小径信步浏览这些饶有趣味的问题肯定也会大有帮助。

对于想要学习Erlang的读者，虽说语言本身相当简单，但想要运用自如也有一些难度。比如，在适应COP之后会觉得非常自然，但对于有OOP背景的程序员而言，从固有的思维习惯转换到COP和FP上（主要是和自己的思维惯性较劲）需要有一个过程。此外OTP和其他Erlang社区多年积累的财富（这些好比JDK、EJB之于Java）也需要一些时间才能被充分地理解和吸收。但这些有价值的资料大多零星地散落于邮件列表和独立的文档之中，给学习造成了很多不必要的麻烦。现在好了，有了Erlang创建者Joe Armstrong为我们撰写的“官方教程”，这些问题都已迎刃而解。

一般而言，由语言的创建者亲自撰写的教程，常常都是杰作。在翻译的过程中，译者也常常会发出这种赞叹。在本书中，Joe Armstrong不仅全面地讲述了Erlang语言本身，还详细交代了这些语言特性的来龙去脉。为什么要这么设计？除了掌握语言本身之外，能有幸窥见大师精微思辨的轨迹，也是难得的机缘。书中的例子，还会将你为之惊异的那些Erlang特性一一解密。通常是从一个不起眼的小问题开始，从宏观分析到微观实现，层层深入细细道来。问题是什么？要如何建模？该怎么重构？各个版本之间的精微演化全然呈现，但这些微小的改进，最终演化出了那些让人惊喜的特性，整个过程可谓相当精彩。

本书由Erlang中文社区（erlang-china.org）组织翻译。其中，第1章到第14章由金尹翻译，第15章到附录F由赵东炜翻译，全书由赵东炜统稿润色和审校。

由于时间仓促，加之译者水平有限，译文难免会有不足之处，欢迎读者批评指正。

赵东炜

2008年3月于北京

译者介绍

赵东炜（Jackyz）

独立软件顾问，一直专注于Web应用开发，曾负责设计和维护某大型门户网站的多个核心应用，对高并发大容量的分布式应用领域有独到见解。曾担任过软件开发工程师、系统架构师、技术经理、产品经理、创业者等多种不同的角色。闲暇时以思考技术问题为乐，从事软件行业10余年来，从最初的ASP/PHP到之后的Java/.NET以及现在的Ajax和Erlang，一直都活跃在技术的最前沿。2006年作为主要译者参与了*Ajax in Action*（中译本《Ajax实战》，由人民邮电出版社出版）的翻译工作。之后为Erlang强大的并发能力所吸引，是国内学习和传播Erlang技术的第一批人，迄今已有2年多的实际开发经验。在2007年3月创建了Erlang中文社区（erlang-china.org），现在是国内Erlang爱好者聚集和分享资料的主要网站。

金尹（TrustNo1）

金尹，长期从事电信行业的大规模语音通信程序的研发，有丰富的并发/分布式网络系统的开发经验。业余从事于数学与编程语言理论，以及并行计算方面的研究。致力于在国内推广函数式语言的发展，分别在2001年和2006年在《程序员》杂志上介绍Python、Erlang等前卫的编程理念。

目　录

第1章 引 言

哦，不！别再来一种编程语言了！我一定要再学另外一种吗？现在的这些难道还不够吗？我能理解你的反应。程序语言浩若烟海，再学一种理由何在？

学习Erlang的理由，可列出如下5条。

- 希望编写能在多核计算机上运行更快的程序。
- 希望编写不停机即可修改的可容错性程序。
- 希望尝试传闻中的“函数式语言”是否切实可行。
- 希望使用一种语言，它既在大规模工业产品中经过实战检验，又不乏优秀的类库与活跃的社区。
- 不希望在冗长烦琐的代码中耗费时间。

我们能如愿以偿吗？在20.3节中，我们会看到一些能在32核计算机上以线性增速运行的程序。在第18章中，我们将会关注如何构造可以历经数年全天候运行的高可靠性系统。在16.1节中，我们还将讨论编写服务器程序的技术，这些服务器可以在不停机的情况下更新软件。

很多时候，我们会不断地夸耀函数式语言的各种长处。函数式语言禁止代码具有“副作用”，副作用与并发水火不容。你要么编写有副作用的顺序代码，要么编写无副作用的并发代码。在这两者之间你必须选择，没有中间情况。

Erlang的并发特性源自语言本身而非操作系统。它把现实世界模拟成一系列的进程，其间仅靠交换消息进行互动，由此Erlang简化了并行编程。在Erlang世界中，存在并行进程但是没有锁，没有同步方法，也不存在共享内存污染的可能，因为Erlang根本没有共享内存。

Erlang程序可以由几百万个超轻量级的进程组成。这些进程可以运行在单处理器、多核处理器或者处理器网络上。

1.1 路线图

- 第2章让你能对Erlang快速起步。
- 第3章是有关顺序型编程的第一章。它介绍了模式匹配和非破坏性赋值的概念。
- 第4章是异常处理。程序总免不了出错。该章讲述了Erlang顺序型编程中的错误侦测和处理。
- 第5章是有关顺序型编程的第二章。它从一些高级的主题开始，涵盖了顺序型编程的其他