Erlang并发编程

作者:
- Joe Armstrong
- Robert Virding
- Claes Wikstrőm
- Mike Williams

原文: Concurrent Programming in Erlang (PDF)
译者: 志愿译者列表
CPiE-CN: 《*Erlang*并发编程》中文版翻译计划
离线浏览: 下载（tar.gz）
最后更新: 2009-12-29 20:51

重要: 作品许可协议

对本书中文译稿的勘误或其他意见及建议，敬请联系连城。

注解: 告读者

这份手稿包含了*Concurrent Programming in Erlang*（ISBN 0-13-508301-X）第一部分的完整内容。

Prentice Hall出版社允许我们将之公开。请注意，由于第二部分（应用）文本的缺失，所有对位于第二部分的章节的引用都是无效的并使用问号??代替。

免费版本的Erlang可以从这里获得:

http://www.ericsson.com/technology/opensource/erlang

Erricson
Telecommunications Systems Laboratories
Box 1505
S-125 25 Älvsjő
Sweden

目录:

术语字典

原文	翻译

atom	原子式
built-in function	内置函数
concurrent, concurrency	并发
guard	保护式
guarded clause	保护子句
parallel, parallelism	并行
process	进程
term	项式
unguarded clause	无保护子句
evaluate/evaluation	求值或执行[*]
last call optimization	末尾调用优化

[*] 若上下文强调函数/BIF的返回值，则往往采用“求值”；否则采用“执行”。

《Erlang并发编程》翻译计划

发起人: 连城
译稿: 在线浏览
原文下载: Concurrent Programming in Erlang (PDF)

> 注解: 目前志愿者名额已满，各章节已分配完毕。除非现有志愿者出于某种原因导致无法完成所分配章节的翻译工作，否则不再征集新的志愿译者。不过仍然征集志愿者进行校订工作。

《Erlang并发编程》中文版（CPiE-CN）的翻译计划完全是因个人兴趣而开始，并公开征集志愿者进行翻译。

译稿采用Sphinx作为文档编辑系统，并使用Subversion进行版本控制，Sphinx源码编码为UTF-8。文档发布服务器由连城提供。

志愿译者

以下是CPiE-CN的志愿者列表，按参与时间排序:

姓名 / ID	Email	主页
连城	rhythm.mail@Gmail.com	http://blog.liancheng.info
王飞	fei @innlab.net	N/A
Ken Zhao	zgosken@Gmail.com	N/A
张驰原	pluskid@Gmail.com	http://blog.pluskid.org
丁豪	zzuduoduo@Gmail.com	N/A
赵卫国	weiguo_1987@sina.com	N/A
吴峻	arcpp.zju@Gmail.com	http://lihdd.net

翻译进度

章节	译者	状态
Preface	连城	完成
Acknowledgments	连城	完成
Introduction	连城	完成
1 An ERLANG Tutorial	连城	完成
2 Sequential Programming	连城	完成
3 Programming with Lists	连城	完成
4 Programming with Tuples	Wangfei	完成
5 Concurrent Programming	张驰原	完成
6 Distributed Programming	Ken Zhao	完成
7 Error Handling	丁豪	完成

8 Programming Robust Applications	Wangfei	待校订
9 Miscellaneous Items	连城	待校订
A ERLANG Reference Grammar	吴峻	待校订
B Built-in Functions	吴峻	进行中
C The Standard Libraries	赵卫国	待校订
D Errors in ERLANG	赵卫国	待校订
E Drivers	吴峻	未开始

译者须知

欢迎各位对自己的英语水平有信心且有比较充足的业余时间的同学参与翻译进程。针对本书的翻译计划，完全是出于个人兴趣，因此没有任何来自出版社或别的什么地方的进度压力，乐于参与翻译的同学不必在进度方面有所顾虑，还是以翻译质量为上，对读者负责。有兴趣的同学可以联系我（连城），Email参见这里。

凡报名参与翻译计划者，请提供以下信息：

- 姓名或ID

  总得知道怎么称呼吧 :-) 报名之后，你的大名将会出现在志愿译者列表上。

- Email

  用于平时联络。同时，你的Email也会出现在志愿译者列表上。为了防止你的邮箱被Spam，Email地址将会以图片形式出现在页面中。你可以自己制作图片，或者，如果你的Email服务商是Gmail、Hotmail、Sina、Yahoo等，可以在这里生成不错的邮箱地址图片 :-)

- Gtalk

  可选，用于平时联络。不过工作日时最好还是邮件联系 :-)

- 个人主页地址

  可选，如果提供，也会出现在志愿译者列表，以便于让大家更好地认识你 :-)

- SVN用户名、密码

  用于给各位开通SVN提交权限。

- 目标章节

  在翻译进度中的空缺章节中选择一个或多个。

对于报名的译者，我会在开通SVN权限后发出邮件通知。

必备工具

没有金刚钻，不揽瓷器活。趁手的兵器往往让你事半功倍。为了更好地参与翻译计划，你最好能够熟练使

用Sphinx和Subversion（SVN）。如果二者都不熟悉，那么也可以将无格式的译文文稿通过邮件发送给我。

Subversion

用于进行版本控制。如果你是Linux用户，那么命令行版本就挺好。如果你是Windows用户，那么推荐使用TortoiseSVN。

在使用SVN进行提交时，为了能够让生成的HTML文档直接显示，请注意设置文件的SVN属性，主要是`svn:mime-type`和`svn:eol-style`。对于Linux用户，可以在checkout出的SVN工作目录的根目录中执行以下命令：

```
find . -name "*.html" | xargs svn ps 'svn:mime-type' 'text/html'
find . -name "*.css" | xargssvn ps 'svn:mime-type' 'text/css'
find . -name "*.js" | xargs svn ps 'svn:mime-type' 'application/x-javascript'
find . -name "*.png" | xargs svn ps 'svn:mime-type' 'image/png'
find . -name "*.jpg" | xargs svn ps 'svn:mime-type' 'image/jpeg'
find . -name "*.gif" | xargs svn ps 'svn:mime-type' 'image/gif'

find . -name "*.rst" | xargs svn ps 'svn:eol-style' 'LF'
find . -name "*.txt" | xargs svn ps 'svn:eol-style' 'LF'
```

对于Windows用户，可以使用TortoiseSVN为新加入SVN的文件单独设置SVN属性。

方便起见，我在Sphinx自动生成的Makefile中增加了一个伪目标`svn-ps`，用于自动完成属性设置工作。

Sphinx

Sphinx是一套基于reStructuredText格式的文档编辑系统，相较于LaTeX、DocBook等重量级工具而言非常地轻便好用，对于需要经常撰写技术文档的懒惰程序员而言，是不可多得的利器。

初学者请参考Sphinx 的官方文档和这篇教程。

关于`drafts`目录

如果你不熟悉Sphinx，而且也没有什么精力去学（真可惜），那么也可以将无格式的译文草稿提交到SVN的`drafts`目录，我会将之适配到Sphinx下。译文草稿的命名规范为`章节名称_译者ID`，如`chapter-8_liancheng`或`appendix-a_somebody`。

Sphinx译稿格式约定

所有源文件应符合以下约定：

1. 文件名采用英文，章节译稿命名规则为：

```
chapter-n.rst
```

附录译稿的命名规则为：

```
appendix-x.rst
```

2. 使用UNIX换行符格式。
3. 字符编码统一为UTF-8 。
4. 对中英文斜体的处理：
   - 原文中对应为斜体的中文文本，在译稿中请使用粗体。原因正如你所见，为手写体原本就右倾的英文设计的斜体并不适合于方正的汉字。为了去除不必要的空格，请使用“`\`”对加粗文本前后的空格进行转义：

     ```
     这是一个\ **粗体**\ 词汇
     ```

     其效果为：

     ```
     这是一个粗体词汇
     ```

   - 原文中对应为斜体的英文文本，保留斜体格式。
5. 插图图片约定：
   - 图片格式请保存为PNG格式。
   - 为了保证图片比例，对原文PDF中的插图进行截图时，请将PDF文件的缩放比例调整为100%后进行截图[1]。

脚注

[1] 谢谢Pluskid的提醒。

序言

Erlang[1]是由Ericsson and Ellemtel Computer Science Laboratories的成员为并发分布式系统编程而开发的一种申明式语言。

对Erlang的开发本是起源于一次为了确认现代的申明式编程范式是否能够用于大型工业电信交换系统的编程的调研。不久，人们便发现适合于电信系统编程的语言同样也非常适合用于解决大量工业嵌入式实时控制问题。

Erlang的许多原语为大型并发实时系统开发的问题提供了解决方案。其模块系统允许将大型系统构建为概念上的可管理单元。其错误检测机制可用于构建容错软件。其代码加载原语允许在不停机的情况下替换运行时系统的代码[2]。

Erlang具备一套基于进程的并发模型。并发性是显示的，用户可以精确地控制哪些计算串行哪些计算并行。进程间的消息传递是异步的，即，发送进程一发完消息就立即继续执行。

Erlang进程交换数据的唯一方式就是消息传递。这样一来，应用可以很容易地做到分布式——为单处理器编写的应用可以容易地迁移到多处理器或单处理器网络上。语言中提供的内建机制简化了分布式应用的编写，使得应用既可运行于单机也可运行于计算机网络中。

Erlang的变量具有单次赋值属性[3]——变量一旦被赋值就不可再被更改。该属性对于调试变更中的应用有重要的影响。

程序是完全以函数的方式编写的——函数的选择通过模式匹配来完成，如此一来程序可以非常的简洁紧凑。

Erlang系统具备内建的时间观念——程序员可以指定一个进程在采取某种行动之前需要等待某条消息多久。这就允许了实时系统的开发。Erlang适用于大多数响应时间为毫秒级的软实时系统。

有关Erlang的最新信息可通过http://www.ericsson.se/erlang获取，e-mail咨询可发送至erlang@erix.ericsson.se。

Joe Armstrong
Robert Virding
Claes Wikstrőm
Mike Williams
Computer Science Laboratory
Erricson Telecommunications Systems Laboratories
Box 1505
S-125 25 Älvsjő
Sweden
erlang@erix.ericsson.se

[1] Agner Krarup Erlang (1878-1929)，丹麦数学家，发展了统计均衡中的随机过程理论——他的理论被广泛地应用于电信业。

[2] 这对于电话交换机或空中交通控制系统等嵌入式实时系统非常重要——通常情况下，这些系统是不能

因为软件维护而停机的。

[3] 也叫做write-once variable或non-destructive assignment。

致谢

关于Erlang，已经很难追溯到某个单一起源。许多语言特性都受到了来自计算机科学实验室的朋友及同事的评论的影响并因此而改进，我们在此对他们的帮助和建议表示感谢。我们尤其要感谢Bjarne Däcker——计算机科学实验室的头儿——感谢他的热情支持与鼓励，以及他在该语言的推广过程中的帮助。

许多人对这本书都有所贡献。Richard Ehrenborg编写了第??章中AVL树的代码。Per Hedeland编写了第??章所描述的`pxw`的代码。Roger Skagervall和Sebastian Strollo提供了第??章所描述的面向对象编程方法背后的思想。Carl Wilhelm Welin用Erlang编写了一个LALR(1)分析程序生成器，用于生成Erlang代码，并在附录A中提供了语法参考。

位于Bollama的Ericsson Business System的早期用户，尤其是第一批用户群（ingen nämnd、ingen glőmd），曾坚忍地充当着小白鼠，与Erlang的许多早期的、不完备版本进行着抗争。他们的评论给予了我们极大的帮助。

我们要感谢来自Ellemtel的Torbjőrn和来自Ericsson Telecom的Bernt Ericson，没有他们无尽的帮助，Erlang永远也不会有光明的一天。

本书使用LaTeX排版[1]，并使用了来自Prentice Hall的Richard Fidczuk提供的`ph.sty`宏包。感谢comp.text.tex回答了我们的许多幼稚的问题。

“UNIX”是AT&T贝尔实验室的注册商标，“X Window System”是MIT的商标。

[1] 译者注：中译稿采用Sphinx排版，感谢Sphinx的作者为懒惰的程序员们开发了如此傻瓜的文档撰写系统！

简介

Erlang是一门被设计用于编写并发、实时、分布式系统的新语言。

很多年来，并发实时系统的编程技术一直落后于串行应用的编程。当使用C或Pascal进行串行编程已经成为实践标准时，大多数实时系统的程序员还在倒腾着汇编。如今的实时系统可以使用Ada、Modula2、Occam等为并发编程提供了显式支持的语言来编写，或是仍旧使用C这样缺乏并发结构的语言。

我们对并发的兴趣源自于对一些展现出大规模并发的问题的研究。这是实时控制系统的一个典型属性。Erlang程序员可以显式指定哪些活动需要由并发进程来展现。这种观念与Occam、CSP、Concurrent Pascal等语言类似，但与那些并不为了对现实世界的并发进行建模而引入并发的语言不同，这些语言引入并发的目的只是为了将编译出可在并行处理器上运行的程序以获得更高的性能。

现今Prolog和ML等语言已经被大范围用于工业应用之中，并极大地降低了设计、实现和维护应用的成本。我们对Erlang的设计和实现也是为了将并发实时系统编程提高到一个相似的高度。

申明式语法

> Erlang具备申明式的语法，且大部分都无副作用。

并发

> Erlang具备一个使用消息传递的基于进程的并发模型。Erlang的并发机制是轻量级的，如进程只占用极少的内存，进程的创建、删除以及消息传递也都只涉及极少量的计算。

实时

> Erlang可用于对相应延迟在毫秒数量级的软实时系统进行编程。

持续运作

> Erlang具备替换运行时系统代码的原语，并允许新旧版本的代码同时执行。这在电话交换机、空中交通控制等“不停机”系统中有极大的用处，这些系统在软件变更时都不能停机。

健壮

> 在上述系统中，安全性是一个关键需求。Erlang具备三种结构来检测运行时错误。这些可用于编写健壮的系统。

内存管理

> Erlang是一门具备实时垃圾回收机制的符号计算语言。内存在需要时自动分配，在不需要时自动回收。典型的内存管理相关的编程错误都不再存在。

分布式

> Erlang没有内存共享。所有进程间交互都通过异步消息传递完成。使用Erlang可以轻易地构建

分布式系统。为单处理器编写的应用不花什么力气就可以移植到处理器网络上运行。

集成

Erlang可以简单地调用其他语言编写的程序。通过Erlang的接口系统，这些程序对于程序员来看就好像是用Erlang编写的一样。

我们从申明式语言和并发语言中借鉴了大量思想。早期Erlang的语法多半归功于STRAND，尽管当前的语法让人觉得像是无类型的ML。其并发模型则与SDL类似。

我们的目标是为健壮的大规模并发工业应用编程提供一种精炼、简单和高效的语言。因此，出于效率原因，我们放弃了许多现代函数式语言中的特性。Currying、高阶函数、延迟求值、ZF comprehension、逻辑变量、deep guards等，增强了申明式编程语言的表达能力，但对于工业控制应用而言，没有这些特性也不会有什么显著影响。倒是模式匹配语法的使用和Erlang变量的“单次赋值”属性，使得我们能够编写清晰、短小且可靠的程序。

最初Erlang是边实现边设计的，第一个版本是一个使用Prolog编写的解释器。与此同时，我们非常幸运地拥有了第一批热情的用户群，他们当时正在开发一个新电话交换机的原型。

这便催生了一条极为有效的语言设计途径。不被使用的构件被抛弃，对于令我们的用户不得不编写令人费解的代码的问题，则引入新的语言构件予以解决。尽管我们经常会对语言进行一些不向下兼容的更改，我们的用户还是飞速地编写了成千上万行的代码，并积极地鼓动其他人来使用这门语言。他们的努力催生了一种新的电话交换机编程方法，相关的一些内容被发表在[??]和[??]。

第一版基于Prolog的Erlang解释器很早以前就被编译型的实现所替代了。其中的一个实现可以免费获取，并使用非商业许可证。当前的Erlang实现在兼顾速度和轻量级并发的同时满足了我们的实时性要求。Erlang实现已经被移植到多种操作系统及多种处理器上。

Erlang适用于对很大范围内的并发应用进行编程。不少工具被开发出来用于辅助Erlang编程，如X Window System的接口、ASN.1编译器（使用Erlang编写并生成Erlang代码）、分析程序生成器、调试器……

读者群

本书面向对实时控制系统感兴趣且有一定的编程经验的人。但并不需要对函数式或逻辑语言有所了解。

本书中的材料与近年来在Ericsson及其全世界范围内的子公司以及瑞士大学举办的多次Erlang课程有关。该课程为期四天，不光展示语言本身，也对Erlang的多种编程范式进行介绍。课程的最后一天通常是一个编程练习，学生们将编写一个与第??章所描述的类似的电话交换机控制系统，并在一台实际的交换机上执行！

概要

本书被划分为两大部分。第一部分“编程”，介绍Erlang语言以及在Erlang编程中常见的一些编程范式。第二部分，“应用”，由一系列完备的章节构成，包含典型Erlang应用的案例分析。

编程

第1章是对Erlang的一个介绍性教程。通过一系列示例对语言的一些主要思想予以说明。

第2章介绍串行编程。这里将会介绍模块系统，这会是我们谈及Erlang时的一个基本术语。

第3和第4章包含一系列使用列表和元组进行编程的示例。这里将介绍列表和元组的基本编程技巧。在后续章节中需要用到的一些标准模块在此也会提及。其中包括实现集合、字典、平衡及非平衡二叉树等等的模块。

第5章介绍并发。在串行编程基础之上添加少量的原语便将Erlang变为一门并发编程语言。我们将介绍用于创建并行进程以及在进程间进行消息传递的原语。我们还将介绍为了将进程与一个名称相关联而引入的进程注册机制。

此处将解释服务器—客户端模型背后的基本思想。该模型在后续章节中被大量使用，同时也是用于协调多个并行进程间的活动的一种基本编程技术。我们还将介绍可让我们编写实时程序的超时。

第6章是对分布式编程的一个概述，解释了编写分布式应用的一些动机。我们描述了用于编写分布式Erlang程序的语言原语并解释了如何在Erlang节点网络中排布多组进程。

第7章解释了Erlang中的错误处理机制。我们将Erlang设计用于编写健壮的应用，语言中包含了三种正交的机制用于完成错误检查。我们认为语言应该在运行时检测出尽可能多的错误并让程序员负责纠正这些错误。

第8章展示了如何使用前一章介绍的错误处理原语来构建健壮的容错系统。我们展示了如何将错误的代码拒之门外，提供了一个容错的服务器（通过扩展客户端服务器模型）并展示了如何对计算进行“隔离”以便在出错时对破坏范围进行限制。

第9章包含了一系列在本书其他部分未提及的编程思想和技巧。我们在此讨论尾递归优化。对于希望正确编写出不间断运行程序的程序员而言，对该优化的理解是至关重要的。We then introduce references which provide unique unforgetable symbols. 本章之后的两节描述了如何更改运行时系统的Erlang代码的细节（编写不停机系统需要使用这种技术）以及如何将Erlang与使用其他语言编写的程序对接。之后，我们将介绍用于高效处理大量无类型数据的二进制数据处理、为每个进程提供了可销毁存储能力的进程字典，以及作为分布式Erlang核心的网络内核。最后我们将对执行效率进行讨论，并举例说明如何编写高效的Erlang代码。

应用

第10章展示了如何使用Erlang编写一个数据库。我们从第??章开发的简单字典模块和第??章的客户端—服务器模型开始，由此完成一个简单的并发数据库。然后我们将展示如何以组织成多级树形结构的并行进程来实现该数据库，以此改善其吞吐量。接着我们加入事务性，以使得多个串行数据库操作可以表现出原子性。

在这之后，我们为数据库增加“回滚”机制以允许我们“撤销”先前的数据库操作。回滚示例对非破坏性赋值进行了漂亮的展示。

接着我们讨论如何让我们的数据库能够容错。最后，我们展示如何将一个外部数据库与我们的数据库集成，并为程序员提供统一的接口。

第11章介绍分布式编程技术。我们展示如何使用Erlang实现多种常见的分布式编程技术，如远程过程调

用、广播、promises等等。

第12章研究分布式数据问题。许多情况下，运行在不同物理机器上的多个应用希望共享一些公共数据结构。本章描述用于实现分布式系统中的共享数据的各种技术。

第13章是对Erlang操作系统的讨论。由于所有的进程管理都在Erlang内部完成，我们对传统操作系统提供的服务所需甚少。我们在此展现Erlang操作系统中用于完成语言标准分布式任务的主要组件。该操作系统可以作为更专一化的操作系统的基础以用于某个具体应用之中。

第14章与两个实时控制问题相关。第一个是著名的电梯控制问题——这里我们将看到将系统建模为一组并行进程提供了一套简单优雅的解决方案。第二部分讨论“进程控制”，此处我们的“进程”是一个卫星。观测卫星的唯一途径是分析安装在卫星上的传感器所发送的数据。变更卫星行为的唯一途径是向卫星上的仪器发送指令。尽管这里以卫星控制系统为例，相关技术可以应用于更广泛的范围。

第15章是一个小型本地电话交换机的实时控制系统实例。Erlang是Ericsson计算机科学实验室开发出来的，而Ericsson也是世界上主要电话交换机的生产商之一——简化电信编程始终都是我们的主要兴趣所在！

本章中的示例只是一个“玩具”系统。然而麻雀虽小五脏俱全，它展示了用Erlang构建电信软件的许多技术。该示例只是那些用于控制复杂电信应用的庞大Erlang程序的小兄弟。这些程序都由数万行Erlang代码构成，是本章所描述的各种编程技术的扩展应用。

本章的末尾对SDL做了一个简短的介绍（SDL被广泛用于描述电信系统的行为）——我们在此展现了一份SDL规范与实现这份规范的Erlang代码的一一对应关系。SDL与Erlang之间的概念“鸿沟”很小——这将降低实时系统的设计实现成本。

第16章简短地介绍了ASN.1并给出了一个从ASN.1到Erlang的交叉编译器。ASN.1是用于描述通讯协议数据格式的标准。本章展现了ASN.1规范与用于操作ASN.1描述的数据包的Erlang代码间的相似性。为系统中通讯软件部分自动产生大部分代码的能力大大简化了系统的构建过程。

第17章展示了如何为Erlang应用构建用户界面。本章展示了两点：第一，并发进程组如何良好地映射到窗口系统中的一组对象；第二，让Erlang与其他语言编写软件包协同运作。

第18章中我们讨论面向对象程序语言的一些主要属性以及如何在Erlang中予以实现。我们将讨论面向对象设计及其Erlang实现之间的关系。

第 I 部分 编程

第1章 Erlang教程

翻译：连城

串行编程

程序1.1用于计算整数的阶乘：

程序1.1

```
-module(math1).
-export([factorial/1]).

factorial(0) -> 1;
factorial(N) -> N * factorial(N - 1).
```

函数可以通过*shell*程序进行交互式求值。 Shell会提示输入一个表达式，并计算和输出用户输入的任意表达式，例如：

```
> math1:factorial(6).
720
> math1:factorial(25).
15511210043330985984000000
```

以上的“`>`”代表 shell 提示符，该行的其他部分是用户输入的表达式。之后的行是表达式的求值结果。

`factorial` 的代码如何被编译并加载至 Erlang 系统中是一个实现相关的问题。 [1]

在我们的例子中，`factorial`函数定义了两个子句：第一个子句描述了计算`factorial(0)`的规则，第二个是计算`factorial(N)`的规则。当使用某个参数对`factorial`进行求值时，两个子句按照它们在模块中出现的次序被依次扫描，直到其中一个与调用相匹配。当发现一个匹配子句时，符号`->`右边的表达式将被求值，求值之前函数定义式中的变量将被代入右侧的表达式。

所有的 Erlang 函数都从属于某一特定模块。最简单的模块包含一个模块声明、导出声明，以及该模块导出的各个函数的实现代码。导出的函数可以从模块外部被调用。其他函数只能在模块内部使用。

程序1.2是该规则的一个示例。

程序1.2

```
-module(math2).
-export([double/1]).

double(X) ->
    times(X, 2).

times(X, N) ->
```

```
    X * N.
```

函数`double/1`可在模块外被求值[2]，`times/2`则只能在模块内部使用，如：

```
> math2:double(10).
20
> math2:times(5, 2).
** undefined function: math2:times(5,2) **
```

在程序1.2中模块声明`-module(math2)`定义了该模块的名称，导出属性`-export([double/1])`表示本模块向外部导出具备一个参数的函数`double`。

函数调用可以嵌套：

```
> math2:double(math2:double(2)).
8
```

Erlang 中的选择是通过模式匹配完成的。程序 1.3 给出一个示例：

程序**1.3**

```
-module(math3).
-export([area/1]).

area({square, Side}) ->
    Side * Side;
area({rectangle, X, Y}) ->
    X * Y;
area({circle, Radius}) ->
    3.14159 * Radius * Radius;
area({triangle, A, B, C}) ->
    S = (A + B + C)/2,
    math:sqrt(S*(S-A)*(S-B)*(S-C)).
```

如我们所期望的，对`math3:area({triangle, 3, 4, 5})`得到`6.0000`而`math3:area({square, 5})`得到 25 。程序1.3 引入了几个新概念：

元组——用于替代复杂数据结构。我们可以用以下 shell 会话进行演示：

```
> Thing = {triangle, 6, 7, 8}.
{triangle, 6, 7, 8}
> math3:area(Thing).
20.3332
```

此处`Thing`被绑定到`{triangle, 6, 7, 8}`——我们将`Thing`称为尺寸为4的一个元组——它包含 4 个元素。第一个元素是原子式`triangle`，其余三个元素分别是整数6、7和8。

模式识别——用于在一个函数中进行子句选择。`area/1`被定义为包含4个子句。
以`math3:area({circle, 10})`为例， 系统会尝试在`area/1`定义的子句中找出一个与`{circle, 10}`相符的匹配，之后将函数定义头部中出现的自由变量`Radius`绑定到调用中提供的值（在这个例子中是10）。

序列和临时变量——这二者是在`area/1`定义的最后一个子句中出现的。最后一个子句的主体是由两条以逗号分隔的语句组成的序列；序列中的语句将依次求值。函数子句的值被定义为语句序列中的最后一个语句的值。在序列中的第一个语句中，我们引入了一个临时变量`s`。

数据类型

Erlang 提供了以下数据类型：

常量数据类型——无法再被分割为更多原始类型的类型：

- 数值——如：123、-789、3.14159、7.8e12、-1.2e-45。数值可进一步分为整数和浮点数。
- **Atom**——如：`abc`、`'An atom with spaces'`、`monday`、`green`、`hello_word`。它们都只是一些命名常量。

复合数据类型——用于组合其他数据类型。复合数据类型分为两种：

- 元组——如：`{a, 12, b}`、`{}`、`{1, 2, 3}`、`{a, b, c, d, e}`。元组用于存储固定数量的元素，并被写作以花括号包围的元素序列。元组类似于传统编程语言中的记录或结构。
- 列表——如：`[]`、`[a, b, 12]`、`[22]`、`[a, 'hello friend']`。列表用于存储可变数量的元素，并被写作以方括号包围的元素序列。

元组合列表的成员本身可以是任意的 Erlang 数据元素——这使得我们可以创建任意复杂的数据结构。

在 Erlang 中可使用变量存储各种类型的值。变量总是以大写字母开头，例如，以下代码片段：

```
X = {book, preface, acknowledgements, contents,
     {chapters, [
        {chapter, 1, 'An Erlang Tutorial'},
        {chapter, 2, ...}
        ]
     }},
```

创建了一个复杂的数据结构并将其存于变量`X`中。

模式识别

模式识别被用于变量赋值和程序流程控制。Erlang是一种单性赋值语言，即一个变量一旦被赋值，就再也不可改变。

模式识别用于将模式与项式进行匹配。如果一个模式与项式具备相同的结构则匹配成功，并且模式中的所有变量将被绑定到项式中相应位置上出现的数据结构。

函数调用中的模式识别

程序1.4定义了在摄氏、华氏和列式温标间进行温度转换的函数`convert`。`convert`的第一个参数是一个包含了温标和要被转换的温度值，第二个参数是目标温标。

程序**1.4**

```
-module(temp).
-export([convert/2]).

convert({fahrenheit, Temp}, celsius) ->
    {celsius, 5 * (Temp - 32) / 9};
convert({celsius, Temp}, fahrenheit) ->
    {farenheit, 32 + Temp * 9 / 5};
convert({reaumur, Temp}, celsius) ->
    {celsius, 10 * Temp / 8};
convert({celsius, Temp}, reaumur) ->
    {reaumur, 8 * Temp / 10};
convert({X, _}, Y) ->
    {cannot,convert,X,to,Y}.
```

对`convert`进行求值时，函数调用中出现的参数（项式）与函数定义中的模式进行匹配。当找到一个匹配时，“`->`”右侧的代码便被求值，如：

```
> temp:convert({fahrenheit, 98.6}, celsius).
{celsius,37.0000}
> temp:convert({reaumur, 80}, celsius).
{celsius,100.000}
> temp:convert({reaumur, 80}, fahrenheit).
{cannot,convert,reaumur,to,fahrenheit}
```

匹配原语“=”

表达式`Pattern = Expression`致使`Expression`被求值并尝试与`Pattern`进行匹配。匹配过程要么成功要么失败。一旦匹配成功，则`Pattern`中所有的变量都被绑定，例如：

```
> N = {12, banana}.
{12,banana}
> {A, B} = N.
{12,banana}
> A.
12
> B.
banana
```

匹配原语可用于从复杂数据结构中拆分元素。

```
> {A, B} = {[1,2,3], {x,y}}.
{[1,2,3],{x,y}}
>A.
[1,2,3]
>B.
{x,y}
> [a,X,b,Y] = [a,{hello, fred},b,1].
[a,{hello,fred},b,1]
> X.
{hello,fred}
> Y.
1
> {_,L,_} = {fred,{likes, [wine, women, song]},
```

```
  {drinks, [whisky, beer]}}.
{fred,{likes,[wine,women,song]},{drinks,[whisky,beer]}}
> L.
{likes,[wine,women,song]}
```

下划线（写作“_”）代表特殊的匿名变量或无所谓变量。在语法要求需要一个变量但又不关心变量的取值时，它可用作占位符。

如果匹配成功，定义表达式`Lhs = Rhs`的取值为`Rhs`。这使得在单一表达式中使用多重匹配成为可能，例如：

```
{A, B} = {X, Y} = C = g{a, 12}
```

“`=`”是右结合操作符，因此`A = B = C = D`被解析为`A = (B = (C = D))`。

内置函数

有一些操作使用Erlang编程无法完成，或无法高效完成。例如，我们无法获悉一个原子式的内部结构，或者是得到当前时间等等——这些都属于语言范畴之外。因此Erlang提供了若干内置函数（built-in function, BIF）用于完成这些操作。

例如函数`atom_to_list/1`将一个原子式转化为一个代表该原子式的（ASCII）整数列表，而函数`date/0`返回当前日期：

```
> atom_to_list(abc).
[97,98,99]
> date().
{93,1,10}
```

BIF的完整列表参见附录??。

并发

Erlang是一门并发编程语言——这意味着在Erlang中可直接对并行活动（进程）进行编程，并且其并行机制是由Erlang而不是宿主操作系统提供的。

为了对一组并行活动进行控制，Erlang提供了多进程原语：`spawn`用于启动一个并行计算（称为进程）；`send`向一个进程发送一条消息；而`receive`从一个进程中接收一条消息。

`spawn/3`启动一个并发进程并返回一个可用于向该进程发送消息或从该进程接收消息的标识符。

`Pid ! Msg`语法用于消息发送。`Pid`是代表一个进程的身份的表达式或常量。`Msg`是要向`Pid`发送的消息。例如：

```
Pid ! {a, 12}
```

表示将消息`{a, 12}`发送至以`Pid`为标识符的进程（Pid是进程标识符**process identifier**的缩写）。在发送之前，消息中的所有参数都先被求值，因此：

```
foo(12) ! math3:area({square, 5})
```

表示对`foo(12)`求值（必须返回一个有效的进程标识符），并对`math3:area({square, 5})`求值，然后将计算结果（即25）作为一条消息发送给进程。`send`原语两侧表达式的求值顺序是不确定的。

`receive`原语用于接收消息。`receive`语法如下：

```
receive
    Message1 ->
        ... ;
    Message2 ->
        ... ;
        ...
end
```

这表示尝试接收一个由`Message1`、`Message2`等模式之一描述的消息。对该原语进行求值的进程将被挂起，直至接收到一个与`Message1`、`Message2`等模式匹配的消息。一旦找到一个匹配，即对“`->`”右侧的代码求值。

接收到消息后，消息接收模式中的所有未绑定变量都被绑定。

`receive`的返回值是被匹配上的接收选项所对应的语句序列的求值结果。

我们可以简单认为`send`发生一条消息而`receive`接收一条消息，然而更准确的描述则是`send`将一条消息发送至一个进程的邮箱，而`receive`尝试从当前进程的邮箱中取出一条消息。

`receive`是有选择性的，也就是说，它从等候接收进程关注的消息队列中取走第一条与消息模式相匹配的消息。如果找不到与接收模式相匹配的消息，则进程继续挂起直至下一条消息到来——未匹配的消息被保存用于后续处理。

一个echo进程

作为一个并发进程的简单示例，我们创建一个*echo*进程用于原样发回它所接收到的消息。我们假设进程`A`向echo进程发送消息`{A, Msg}`，则echo进程向`A`发送一条包含`Msg`的新消息。如图1.1所示。

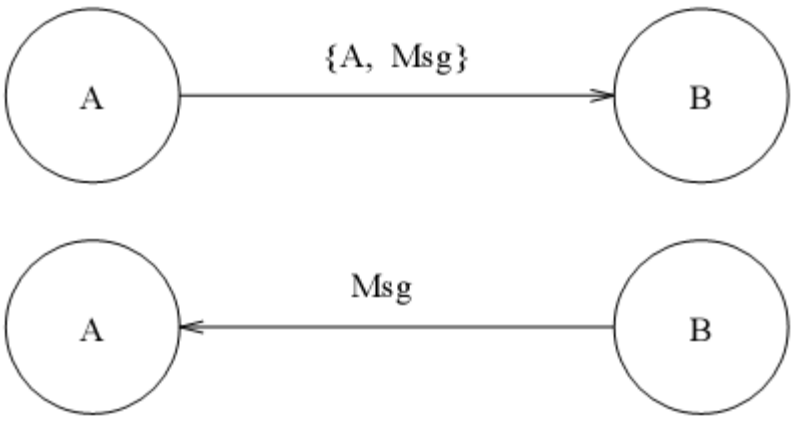

图1.1 一个echo进程

在程序1.5中`echo:start()`创建一个返回任何发送给它的消息的简单进程。

程序 **1.5**

```
-module(echo).
-export([start/0, loop/0]).

start() ->
    spawn(echo, loop, []).
loop() ->
    receive
        {From, Message} ->
            From ! Message,
            loop()
    end.
```

`spawn(echo, loop [])`对`echo:loop()`所表示的函数相对于调用函数并行求值。因此，针对：

```
...
Id = echo:start(),
Id ! {self(), hello}
...
```

进行求值将会启动一个并行进程并向该进程发送消息`{self(), hello}`——`self()`是用于获取当前进程标识符的BIF。

脚注

[1] “实现相关”是指如何完成某个具体操作的细节是系统相关的，也不在本书的讨论范畴之内。
[2] `F/N`标记表示具备`N`个参数的函数`F`。

第2章 串行编程

翻译：连城

本章介绍用于编写串行Erlang程序的概念。我们首先讨论变量赋值的基本机制和如何实现控制流程。为此，我们要先了解一下项式、模式和模式匹配。

项式

Erlang中以下数据类型[1]被称为项式：

- 常量类型
  - 数值
    - 整数，用于存储自然数
    - 浮点数，用于存储实数
  - 原子式
  - Pid（进程标识符process identifier的缩写），用于存储进程标识
  - 引用，用于存储系统范围内唯一的引用
- 复合数据类型
  - 元组，用于存储固定数目的多个项式
  - 列表，用于存储可变数目的多个项式

数值

以下实例都属于数值：

```
123 -34567 12.345 -27.45e-05
```

整数精度与实现相关，但任何Erlang系统都应保证至少24位的整数精度。

`$<Char>`标记表示字符`Char`对应的ASCII值，如`$A`表示整数65。

不以10为基数的整数可写作`<Base>#<Value>`，如16#ffff表示十进制整数65535。`Base`的取值范围为2 .. 16。

浮点数以传统方式书写。

原子式

原子式是有名称的常量。例如在某个用于日历计算的程序中可使用`monday`、`tuesday`等等表示一星期中的各天。原子式用于增强程序的可读性。

一些原子式实例：

```
friday unquoted_atoms_cannot_contain_blanks
'A quoted atom which contains several blanks'
```

```
'hello \n my friend'
```

原子式以小写字母（`a..z`）开头，以非字母数字字符结尾——否则就必须用引号括起来。

通过将原子式以引号括起来，原子式中便可以出现任意字符。原子式总是以可被 Erlang 读取程序读入的格式输出。原子式引号内的字符遵循如下规范：

字符	含义
`\b`	退格符
`\d`	删除符
`\e`	转义符（ESC）
`\f`	换页符
`\n`	换行符
`\r`	回车符
`\t`	制表符
`\v`	垂直制表符
`\\`	反斜线
`\^A .. \^Z`	control A到control Z（即0 .. 26）
`\'`	单引号
`\"`	双引号
`\ooo`	使用八进制格式`ooo`表示的字符

在引号括起来的原子式中如果包含字符序列`\c`，其中`c`的ASCII值小于32，则表示`\c`的这部分源码被忽略（这样我们在编程时就可以使用一个反斜线加换行符来将长原子式分隔为几行）。

元组

以花括号包围的一系列以逗号分隔的项式称为元组。元组用于存储固定数目个项式。它们与传统编程语言中的结构或记录类似。

元组`{E1,E2,...,En}`，其中`n`大于0，称为大小为`n`的元组。元组中的单个项式称为元素。

以下是一些元组实例：

```
{a, 12, 'hello'}
{1, 2, {3, 4}, {a, {b, c}}}
{}
```

列表

以方括号包围的一系列以逗号分隔的项式成为列表。列表用于存储可变数目个项式。

对于列表`[E1,E2,...En]`，其中`n` >= 0 ，称其长度为`n`。

以下是一些元组实例：

```
[1, abc, [12], 'foo bar']
```

```
[]
[a,b,c]
"abcd"
```

被我们称之为字符串的`"..."`标记，实际上是引号中各个字符组成的列表的ASCII简写形式。因此`"abc"`对应于`[97,98,99]`。在原子式中使用的转义规则在字符串中通用。

在对列表进行处理时，往往需要一种方便的手段来引用列表的第一个元素以及除掉第一个元素以外列表的剩余部分。方便起见，我们将列表的第一个元素称为头部，将剩余部分称为 尾部 。

我们使用`[E1,E2,E3,...,En|Variable]`来标记一个前`n`个元素分别为`E1,E2,E3,...,En`而剩余部分记为`Variable`的列表。

注意“`|`”之后的项式不一定要是列表，它可以是任意一个合法的Erlang项式。最后一个尾部为项式`[]`的列表称为真列表或格式良好的列表——大多数（尽管不是全部）Erlang程序都是被编写来处理格式良好的列表的。

模式匹配

模式与项式有着相同的结构，但它们还可以包含变量。变量名都以大写字母开头。

模式示例：

```
{A, a, 12, [12,34|{a}]}
{A, B, 23}
{x, {X_1}, 12, My_cats_age}
[]
```

以上的`A`、`B`、`X_1`和`My_cats_age`都是变量。

模式匹配为变量赋值提供了基本的机制。被赋值后，变量便被绑定——否则便是未绑定变量。给变量赋值的动作称作“绑定”。变量一旦被绑定便不可更改。这种变量属性被称为一次性绑定或单次赋值。这种属性与传统命令式语言的破坏性赋值[2]相反。

如果一个模式与一个项式在结构上同构，且在模式中任一位置出现的原子数据类型也都在项式的相应位置上出现，则称他们它们相互匹配。如果模式中包含未绑定变量，则该变量在匹配过程中将被绑定到项式中相应的元素。如果在模式中相同的变量多次出现，则项式中对应位置的元素也必须相同。

模式匹配在以下情况下发生：

- 计算形如`Lhs = Rhs`的表达式时
- 调用函数时
- 在`case`和`receive`原语中对指定模式进行匹配时

`Pattern = Expression`

表达式`Pattern = Expression`将致使`Expression`被求值，并将其结果与`Pattern`进行匹配。匹配要么成功要么失败。若匹配成功则`Pattern`中的所有（未绑定）变量都将被绑定。

以下我们将假设模式匹配总是成功。对失败的处理将在第??章详细讨论。

示例：

```
{A, B} = {12, apple}
```

匹配成功后建立绑定关系`A`→`12`[3]和`B`→`apple`。

```
{C, [Head|Tail]} = {{222, man}, [a,b,c]}
```

匹配成功后建立绑定关系`C`→`{222, man}`、`Head`→`a`和`Tail`→`[b, c]`。

```
[{person, Name, Age, _}|T] =
        [{person, fred, 22, male},
         {person, susan, 19, female}, ...]
```

匹配成功后建立绑定关系`T`→`[{person, susan, 19, female}, ...]`、`Name`→`fred`和`Age`→`22`。在最后一个例子中我们利用了写作“`_`”的匿名变量——在语法上需要一个变量出现，但我们又不关心该变量的值的时候便可以使用匿名变量。

当一个变量在一个模式中多次出现时，只有被匹配的对应元素的值都相同时匹配才会成功。因此，举例来说，`{A, foo, A} = {123, foo, 123}`将成功匹配，并将`A`绑定到`123`，然而`{A, foo, A} = {123, foo, abc}`就会失败，因为我们不能将`A`同时绑定到`123`和`abc`。

“`=`”是一个右结合的中缀运算符。因此`A = B = C = D`将被解析为`A = (B = (C = D))`。这种用法可能只有在`{A, B} = X = ...`这样的构造中才有用，这时我们可以同时获悉表达式的值及其组成部分。表达式`Lhs = Rhs`的值被定义为`Rhs`。

函数调用中的模式匹配

Erlang通过模式匹配来提供选择和控制流程。例如，程序2.1定义了一个函数`classify_day/1`，当调用参数为`saturday`或`sunday`时返回`weekEnd`，否则返回`weekDay` 。

程序 **2.1**

```
-module(dates).
-export([classify_day/1]).

classify_day(saturday) -> weekEnd;
classify_day(sunday)   -> weekEnd;
classify_day(_)        -> weekDay.
```

进行函数求值时，会将函数的参数与函数定义中出现的模式一一进行匹配。一旦发现一个成功的匹配，“`->`”之后的符号便被求值，因此：

```
> dates:classify_day(saturday).
weekEnd
> dates:classify_day(friday).
weekDay
```

如果所有的子句都不匹配，则函数调用失败（失败将引发第??章描述的错误捕捉机制）。

当执行流程进入一个函数的某个子句时，描述该子句的模式所包含的变量将被绑定。因此，举例来说，对程序1.3的`math3:area({square, 5})`进行求值将致使变量`Side`被绑定到`5`。

表达式求值

表达式具备与模式相同的语法，同时表达式还可以包含函数调用或传统的中序算术表达式。函数调用的写法很传统，如：`area:triangle(A, B, C)`便代表以参数`A`、`B`和`C`调用函数`area:triangle`。

Erlang 表达式的求值机制如下。

对项式求值得到其本身：

```
> 222.
222
> abc.
abc
> 3.1415926.
3.14159
> {a,12,[b,c|d]}.
{a,12,[b,c|d]}
> {{},[{}],{a,45,'hello world'}}.
{{},[{}],{a,45,'hello world'}}
```

浮点数的输出格式可能与它们的输入格式不完全一致。当表达式与项式同构且表达式中的函数调用都已求值完毕时，表达式将被求值为项式。应用一个函数时其参数首先被求值。

求值过程可以被认为是一个将表达式归结为基础项式的函数：

```
ε(X) when Constant(X)→X
ε({t1,t2,...,tn})→{ε(t1),ε(t2),...,ε(tn)}
ε([t1,t2,...,tn])→[ε(t1),ε(t2),...,ε(tn)]
ε(functionName(t1,t2,...,tn)→
    APPLY(functionName,[ε(t1),ε(t2),...,ε(tn)])
```

其中`APPLY`表示一个将参数应用到函数的函数。

函数求值

函数调用的写法如以下实例所示：

```
> length([a,b,c]).
3
> lists:append([a,b], [1,2,3]).
[a,b,1,2,3]
> math:pi().
3.14159
```

带冒号形式的函数将在和模块相关的章节中解释。调用没有参数的函数必须加上一对空的小括号（以此与原子式相区别）。

求值顺序

函数参数的求值顺序是不确定的。例如，`f({a},b(),g(a,h(b),{f,X}))`表示一个函数调用。对函数`f`的调用有三个参数：`{a}`、`b()`和`g(a,h(b),{f,X})`。第一个参数是一个只包含一个原子项`a`的元组。第二个参数是一个函数调用`b()`。第三个参数是函数调用`g(a,h(b),{f,X})`。在对`f/3`求值时，对`b/0`和`g/3`的求值顺序是不确定的，不过`h(b)`在`g/3`被求值。对`b()`和`h(b)`的求值顺序也是不确定的。

在对形如`[f(a), g(b), h(k)]`的表达式进行求值时，`f(a)`、`g(b)`和`h(k)`的求值顺序是不确定的。

如果`f(a)`、`g(b)`和`h(k)`的求值过程没有副作用（即不发送消息、不创建进程等等），则`[f(a), g(b), h(k)]`的值与求值顺序无关[4]。这种属性叫作引用透明性[5]。

应用

BIF `apply(Mod, Func, ArgList)`和`apply({Mod, Func}, ArgList)`用于将模块`Mod`中的函数`Func`应用到参数列表`ArgList`。

```
> apply(dates, classify_day, [monday]).
weekDay
> apply(math, sqrt, [4]).
2.0
> apply({erlang, atom_to_list}, [abc]).
[97,98,99]
```

使用`apply`对BIF进行求值时，可以使用`erlang`作为模块名。

模块系统

Erlang具备一套模块系统以便我们将大型程序切分为一组模块。每个模块都有自己的名称空间；这样我们就可以在不同的模块中自由地使用相同的函数名而不会有任何冲突。

模块系统以对给定模块中函数的可见性进行限制的方式来工作的。函数的调用方式取决于模块名、函数名以及函数名是否在模块的导入或导出声明中出现。

程序 **2.2**

```
-module(lists1).
-export([reverse/1]).

reverse(L) ->
        reverse(L, []).

reverse([H|T], L) ->
        reverse(T, [H|L]);
reverse([], L) ->
        L.
```

程序2.2定义了一个颠倒列表元素顺序的函数`reverse/1`。`reverse/1`是该模块中唯一可以从该模块之外被调用

的函数。需要从模块外部调用的函数必须出现在模块的导出声明中。

该模块中定义的其他函数，`reverse/2`，仅可供模块内部使用。注意`reverse/1`和`reverse/2`是完全不同的函数。在Erlang中，名字相同但参数数目不同的两个函数是完全不同的函数。

模块间调用

从其他模块中调用函数的方法有两种：

程序 **2.3**

```
-module(sort1).
-export([reverse_sort/1, sort/1]).

reverse_sort(L) ->
        lists1:reverse(sort(L)).
sort(L) ->
        lists:sort(L).
```

`reverse/1`以完全限定名称被调用。

你还可以借助`import`声明使用隐式限定函数名，如程序2.4所示。

程序 **2.4**

```
-module(sort2).
-import(lists1, [reverse/1]).
-export([reverse_sort/1, sort/1]).

reverse_sort(L) ->
        reverse(sort(L)).
sort(L) ->
        lists:sort(L).
```

两种形式都是为了解决二义性。比如，当两个不同的模块导出了重名的函数，则必须显式限定函数名。

函数定义

以下章节更详细地描述了Erlang函数的语法。首先我来给函数的各个语法元素命名。接着将详细描述这些元素。

术语

考虑以下模块：

程序 **2.5**

```
-module(lists2).

-export([flat_length/1]).

%% flat_length(List)
%%  Calculate the length of a list of lists.

flat_length(List) ->
    flat_length(List, 0).

flat_length([H|T], N) when list(H) ->
    flat_length(H, flat_length(T, N));
flat_length([H|T], N) ->
    flat_length(T, N + 1);
flat_length([], N) ->
    N.
```

以“%”打头的是注释。注释可以从一行的任意位置开始，一直持续到行末。

第1行包含模块声明。该行必须出现在任何其他声明或代码之前。

第1行和第3行开头的“-”称为属性前缀。module(list2)便是属性的一个例子。

第2、第4等行是空行——连续的单个或多个空白符、空行、制表符、换行符等，都被当作单个空白符处理。

第3行声明了一个具有一个参数的函数flag_length，该行意味着该函数存在于模块中并会被从模块中导出。

第5、6行是注释。

第8、9行包含了函数flat_length/1的定义。它由单个子句组成。

表达式flat_length(List)称为子句的头部。“->”之后的部分为子句的主体。

第11至16行函数flat_length/2的定义——该函数包含三个子句；子句间以分号“;”分隔，在最后的结尾处以“.”结尾。

第11行中flat_length/2的第一个参数为列表[H|T]。H表示列表的头部，T代表列表的尾部。在关键字when和箭头“->”之间的表达式list(H)称作保护式。只有在参数与函数头部的模式相匹配且保护式断言成立时，函数体才会被求值。

flat_length/2的第一个子句称为保护子句；其他的子句称为无保护子句。

flat_length/2是一个局部函数——即不可从模块外部被调用（因为它没有出现在export属性中）。

模块lists2包含了函数flat_length/1和flat_length/2的定义。它们代表两个完全不同的函数——这与C或Pascal等语言不通，在这些语言中一个函数名只能出现一次，且只能有固定个数的参数。

子句

每个函数都由一组子句组成。子句间以分号“;”分隔。每个子句都包含一个子句头部、一个可选的保护式和子句主体。下面将详细解释。

子句头部

子句的头部包含一个函数名和一组以逗号分隔的参数。每个参数都是一个合法的模式。

当函数调用发生时，将会按顺序对函数定义中的子句头部依次进行匹配。

子句保护式

保护式是子句被选中前必须要满足的条件。

保护式可以是一个简单的断言或是一组由逗号分隔的简单断言。一个简单断言可以是一个算数比较、项式比较，或是一个系统预定义的断言函数。保护式可以看作是模式匹配的一种扩展。用户自定义的函数不能用在保护式内。

对保护式求值时所有的断言都将被求值。若所有断言都为真，则保护式成立，否则就失败。保护式中各个断言的求值顺序是不确定的。

如果保护式成立，则会对子句的主体进行求值。如果保护式失败，则尝试下一个候选子句。

一旦子句的头部和保护式都匹配成功，系统将指定这条子句并对其主体求值。

我们可以写一个保护式版本的`factorial`。

```
factorial(N) when N == 0 -> 1;
factorial(N) when N > 0  -> N * factorial(N - 1).
```

注意对于以上示例，我们可以调换子句的顺序，即：

```
factorial(N) when N > 0  -> N * factorial(N - 1);
factorial(N) when N == 0 -> 1.
```

在这个示例中子句首部模式与保护式的组合可以唯一确定一个正确的子句。

保护式断言

保护式断言的完整集合如下：

保护式	成立条件
`atom(X)`	`X`是一个原子式
`constant(X)`	`X`不是列表或元组
`float(X)`	`X`是一个浮点数
`integer(X)`	`X`是一个整数
`list(X)`	`X`是一个列表或`[]`
`number`	`X`是一个整数或浮点数
`pid(X)`	`X`是一个进程标识符
`port(X)`	`X`是一个端口
`reference(X)`	

	X是一个引用
tuple(X)	X是一个元组
binary(X)	X是一段二进制数据

另外，一些BIF和算术表达式的组合也可以作为保护式。它们是：

```
element/2, float/1, hd/1, length/1, round/1, self/0, size/1
trunc/1,   tl/1, abs/1, node/1, node/0, nodes/0
```

项式比较

可以出现在保护式中的项式比较运算符如下：

运算符	描述	类型
X > Y	X大于Y	coerce
X < Y	X小于Y	coerce
X =< Y	X小于或等于Y	coerce
X >= Y	X大于或等于Y	coerce
X == Y	X等于Y	coerce
X /= Y	X不等于Y	coerce
X =:= Y	X等于Y	exact
X =/= Y	X不等于Y	exact

比较运算符工作机制如下：首先对运算符两边求值（如，在表达式两边存在算术表达式或包含BIF保护式函数时）；然后再进行比较。

为了进行比较，定义如下的偏序关系：

```
number < atom < reference < port < pid < tuple < list
```

元组首先按大小排序，然后再按元素排序。列表的比较顺序是先头部，后尾部。

如果比较运算符的两个参数都是数值类型且运算符为*coerce*型，则如果一个参数是`integer`另一个是`float`，那么`integer`将被转换为`float`再进行比较。

`exact`类型的运算符则不做这样的转换。

因此`5.0 == 1 + 4`为真，而`5.0 =:= 4 + 1`为假。

保护函数子句示例：

```
foo(X, Y, Z) when integer(X), integer(Y), integer(Z), X == Y + Z ->
foo(X, Y, Z) when list(X), hd(X) == {Y, length(Z)}  ->
foo(X, Y, Z) when {X, Y, size(Z)} == {a, 12, X} ->
foo(X) when list(X), hd(X) == c1, hd(tl(X)) == c2 ->
```

注意在保护式中不可引入新的变量。

子句主体

子句的主体有一个或多个有逗号分隔的表达式序列组成。序列中的表达式依次被求值。表达式序列的值被定义为序列中最后一个表达式的值。例如，`factorial`的第二个子句可以写成：

```
factorial(N) when N > 0 ->
    N1 = N - 1,
    F1 = factorial(N1),
    N * F1.
```

在对序列求值的过程中，表达式的求值结果要么与一个模式进行匹配，要么被直接丢弃。将函数主体拆分为序列的原因有这么几条：

- 确保代码的顺序执行——函数主体中的表达式是依次求值的，而在嵌套的函数调用中的函数则可能以任意顺序执行。
- 增强代码可读性——将函数写成表达式序列可以令程序更清晰。
- （通过模式匹配）拆解函数的返回值。
- 重用函数调用的返回值。

对函数返回值的多次重用的示例如下：

```
good(X) ->
        Temp = lic(X),
        {cos(Temp), sin(Temp)}.
```

上面的写法比下面这么写要好：

```
bad(X) ->
        {cos(lic(X)), sin(lic(X)}.
```

二者表达的是同一个含义。`lic`代表长而复杂的计算过程（Long and Involved Calculation），即那些计算代价高的函数。

原语

Erlang提供了元语`case`和`if`，这样在子句中无需借助其他函数便可以直接进行条件求值。

Case

`case`表达式允许在子句主体内部于多个选项中进行选择，语法如下：

```
case Expr of
    Pattern1 [when Guard1] -> Seq1;
    Pattern2 [when Guard2] -> Seq2;
    ...
    PatternN [when GuardN] -> SeqN
end
```

首先，对`Expr`求值，然后，`Expr`的值将依次与模式`Pattern1`、`Pattern2`.....`PatternN`进行匹配，直到匹配成

功。如果找到一个匹配并且（可选的）的保护式成立，则对应的调用序列将被求值。注意`case`保护式与函数保护式形式相同。`case`原语的值就是被选中的序列的值。

至少得有一个模式必须得以匹配——否则就会产生一个运行时错误并引发第??章中的错误处理机制。

举个例子，比方说我们我有个函数`allocate(Resource)`用于分配某种资源`Resource`。假设这个函数只返回`{yes, Address}`或`no`。这样，这个函数便可以放在一个`case`结构里：

```
...
case allocate(Resource) of
    {yes,Address} when Address > 0, Address =< Max ->
        Sequence 1 ... ;
    no ->
        Sequence 2 ...
end
...
```

在`Sequence 1 ...`中，变量`Address`已经被绑定在了`allocate/1`的返回结果上。

为了避免匹配错误的发生，我们常常追加一个必会匹配的模式[6]作为`case`原语的最后一个分支：

```
case Fn of
    ...
    _ ->
        true
end
```

If

`if`表达式的语法如下：

```
if
    Guard1 ->
        Sequence1 ;
    Guard2 ->
        Sequence2 ;
    ...
end
```

在这种情况下，保护式`Guard1,...`将被依次求值。如果一个保护式成立则对与之关联的序列求值。该序列的求值结果便是`if`结构的结果。`if`保护式与函数保护式形式相同。与`case`相同，一个保护式都不成立的话将引发一个错误。如果需要，可以增加保护式断言`true`作为垃圾箱：

```
if
    ...
    true ->
        true
end
```

Case 和 if 使用示例

使用`case`和`if`我们可以以多种方式来编写`factorial`。

最简单的：

```
factorial(0) -> 1;
factorial(N) -> N * factorial(N - 1).
```

使用函数保护式：

```
factorial(0) -> 1;
factorial(N) when N > 0 -> N * factorial(N - 1).
```

使用`if`：

```
factorial(N) ->
    if
        N == 0 -> 1;
        N > 0 -> N * factorial(N - 1)
    end.
```

使用`case`：

```
factorial(N) ->
    case N of
        0 -> 1;
        N when N > 0 ->
            N * factorial(N - 1)
    end.
```

使用变量保持临时结果：

```
factorial(0) ->
    1;
factorial(N) when N > 0 ->
    N1 = N - 1,
    F1 = factorial(N1),
    N * F1.
```

以上所有定义都是正确且等价的[7]——如何进行选择完全是个美学问题[8]。

算术表达式

算术表达式由以下运算符构成：

运算符	描述	类型	操作数类型	优先级
`+ X`	`+ X`	单目	混合	1
`- X`	`- X`	单目	混合	1
`X * Y`	`X * Y`	双目	混合	2
`X / Y`	`X / Y`（浮点除法）	双目	混合	2
`X div Y`	`X`整除`Y`	双目	整数	2
`X rem Y`	`X`除以`Y`的余数	双目	整数	2
`X band Y`				

	X与Y的位与	双目	整数	2
X + Y	X + Y	双目	混合	3
X - Y	X - Y	双目	混合	3
X bor Y	X与Y位或	双目	整数	3
X bxor Y	X与Y的位算数异或	双目	整数	3
X bsl N	X算数左移N位	双目	整数	3
X bsr N	X右移N位	双目	整数	3

单目运算符有一个参数，双目运算符有两个参数。混合意味着参数即可以是`integer` 也可以是`float`。单目运算符的返回值与其参数类型相同。

双目混合运算符（即`*`、`-`、`+`）在参数都是`integer`时返回类型为`integer`的对象，在参数至少包含一个`float`时返回一个`float`。浮点除法运算符`/`总是返回一个`float`。

双目整数运算符（即`band`、`div`、`rem`、`bor`、`bxor`、`bsl`、`bsr`）的参数必须是整数，其返回值也是整数。

求值顺序取决于运算符的优先级：首先计算第1优先级的运算符，然后是第2优先级，以此类推。括号内的表达式优先求值。

优先级相同的运算符从左到右进行求值。比如：

```
A - B - C - D
```

其求值顺序与下面的表达式一致：

```
(((A - B) - C) - D)
```

变量作用域

子句中变量的生存期从它首次被绑定处开始，到子句中对该变量的最后一个引用处结束。变量的绑定只会在模式匹配中发生；可以将之认作是一个变量产生过程。后续对变量的所有引用都是对变量的值的使用。表达式中的变量必须是经过绑定的。变量第一次出现时就被用在表达式中是非法的。比如：

```
1  f(X) ->
2      Y = g(X),
3      h(Y, X),
4      p(Y).
```

第1行中，定义了变量`X`（它在进入函数时被绑定）。第2行中，使用了`X`，定义了`Y`（首次出现）。第3行中，使用了`X`和`Y`，然后在第4行中使用了`Y`。

if、case和receive的作用域规则

在`if`、`case`或`receive`原语中引入的变量会被隐式导出到原语主体之外。比方我们有：

```
f(X) ->
```

```
    case g(X) of
        true -> A = h(X);
        false -> A = k(X)
end,
...
```

变量A在其被定义的case原语之后仍然有效。从if、case或receive原语中导出变量时应注意一些规则：

在if、case或receive原语的不同分支中引入的变量集合必须相同，除非缺少的变量在原语外不再被引用。

例如以下代码：

```
f(X) ->
    case g(X) of
        true -> A = h(X), B = A + 7;
        false -> B = 6
    end,
h(A).
```

这段代码就是非法的。因为在对true分支求值时定义了变量A和B，而在对false分支求值时只定义了B。在case原语之后，又在调用h(A)中引用了A——如果是fase分支被求值，则A尚未被定义。注意如果调用的不是h(A)而是h(B)则这段代码就是合法的，因为B在case原语的两个分支中都有定义。

脚注

[1] 附录A给出了Erlang的形式语法。
[2] 许多人认为破坏性赋值会导致难以理解和易错的不清晰的程序。
[3] 标记Var→Value表示变量Var的值为Value。
[4] 假设所有函数调用都结束。
[5] 即是说函数的值与调用上下文无关。
[6] 有时被称为垃圾箱。
[7] 好吧，几乎是——想想看factorial(-1)？
[8] 如果不知道选哪个，选最漂亮的那个！

第3章 列表编程

翻译：连城

这一章研究对列表的处理。列表是用于存储可变数量的元素的结构。列表的写法以“[”开头以“]”结尾。列表的元素以逗号分隔。例如，[E1,E2,E3,...]指代包含元素E1,E2,E3,...的列表。

标记[E1,E2,E3,...,En|Variable]，其中n ≥ 1，用于表示前n个元素为E1,E2,E3,...,En其余部分由Variable指代的列表。当n = 1时，列表的形式为[H|T]；这个形式的出现频率非常高，通常将H称为列表的头部，而T为列表的尾部。

本章我们将讨论如何处理真列表；即尾部为空列表[]的列表。

应该记住在处理固定数目的元素时总是应该使用元组tuple。元组所占的存储空间仅是列表的一半并且访问也更迅速。在需要处理可变数目个元素时才应该使用列表。

用于列表处理的BIF

一些内置函数可用于列表与其他数据类型间的互相转换。主要的BIF包括：

`atom_to_list(A)`

将原子式A转换为一个ASCII字符列表。

如：`atom_to_list(hello)` `[104,101,108,108,111]`[1]。

`float_to_list(F)`

将浮点数F转换为一个ASCII字符列表。

如：`float_to_list(1.5)` `[49,46,53,48,48,...,48]`。

`integer_to_list(I)`

将整数I转换为一个ASCII字符列表。

如：`integer_to_list(1245)` `[[49,50,52,53]`。

`list_to_atom(L)`

将ASCII字符列表L转换为一个原子式。

如：`list_to_atom([119,111,114,108,100])` `world`。

`list_to_float(L)`

将ASCII字符列表L转换为一个浮点数。

如：`list_to_float([51,46,49,52,49,53,57])` `3.14159`。

`list_to_integer(L)`

将ASCII字符列表`L`转换为一个整数。

如：`list_to_integer([49,50,51,52])` `1234`。

`hd(L)`

返回列表`L`的第一个元素。

如：`hd([a,b,c,d])` `a`。

`tl(L)`

返回列表`L`的尾部。

如：`tl([a,b,c,d])` `[b,c,d]`。

`length(L)`

返回列表`L`的长度。

如：`length([a,b,c,d])` `4`。

有两个BIF `tuple_to_list/1`和`list_to_tuple/1`将放在第??章讨论。还有一些列表处理相关的BIF，如`list_to_pid(AsciiList)`、`pid_to_list(Pid)`。这些将在附录B中描述。

常用列表处理函数

以下各小节给出了一些简单列表处理函数的使用示例。这里所描述的所有函数都包含在标准Erlang发行版的`lists`模块中（细节参见附录C）。

member

`memeber(X, L)`在`X`是列表`L`的元素时返回`true`，否则返回`false`。

```
member(X, [X|_]) -> true;
member(X, [_|T]) -> member(X, T);
member(X, []) -> false.
```

`memeber`的第一个子句匹配的是`X`为列表的第一个元素的情况，这种情况下`member`返回`true`。如果第一个子句不匹配，则第二个子句将匹配第二个参数是非空列表的情况，这种情况下模式`[_|T]`匹配一个非空列表并将`T`绑定到列表的尾部，然后以原来的`X`及列表的尾部`T`递归调用`member`。`member`前两个子句就是在说当`X`是列表的第一个元素（头部），或它被包含在列表的剩余部分（尾部）中时，`X`就是该列表的一个成员。`member`的第三个子句是说`X`不可能是空列表`[]`的成员，并因此返回`false`。

我们将`member`的求值过程列举如下：

```
> lists:member(a,[1,2,a,b,c]).
(0)lists:member(a,[1,2,a,b,c])
(1).lists:member(a, [2,a,b,c])
(2)..lists:member(a,[a,b,c])
(2)..true
(1).true
(0)true
true
> lists:member(a,[1,2,3,4]).
(0)lists:member(a, [1,2,3,4])
(1).lists:member(a, [2,3,4])
(2)..lists:member(a, [3,4])
(3)...lists:member(a, [4])
(4)....lists:member(a, [])
(4)....false
(3)...false
(2)..false
(1).false
(0)false
false
```

append

`append(A,B)`连接两个列表`A`和`B`。

```
append([H|L1], L2) -> [H|append(L1, L2)];
append([], L) -> L.
```

`append`的第二个子句再明白不过了——将任意列表`L`追加至空列表之后仍得到`L`。

第一个子句给出了追加一个非空列表到另一个列表之后的规则。因此，对于：

```
append([a,b,c], [d,e,f])
```

其求值结果为：

```
[a | append([b,c], [d,e,f])]
```

那么`append([b,c], [d,e,f])`的值又是多少呢？它（当然）是`[b,c,d,e,f]`，因此`[a | append([b,c], [d,e,f])]`的值就是`[a|append([b,c], [d,e,f])]`，这也是`[a,b,c,d,e,f]`的另一种写法。

`append`的行为如下：

```
> lists:append([a,b,c],[d,e,f]).
(0)lists:append([a,b,c],[d,e,f])
(1).lists:append([b,c], [d,e,f])
(2)..lists:append([c],[d,e,f])
(3)...lists:append([], [d,e,f])
(3)...[d,e,f]
(2)..[c,d,e,f]
(1).[b,c,d,e,f]
(0)[a,b,c,d,e,f]
[a,b,c,d,e,f]
```

reverse

`reverse(L)`用于颠倒列表`L`中的元素顺序。

```
reverse(L) -> reverse(L, []).

reverse([H|T], Acc) ->
    reverse(T, [H|Acc]);
reverse([], Acc) ->
    Acc.
```

`reverse(L)`利用一个辅助函数`reverse/2`将最终结果累积到第二个参数中。

调用`reverse(L, Acc)`时，若`L`是一个非空列表，则将`L`的第一个元素移除并添加到`Acc`的头部。因此对`reverse([x,y,z], Acc)`的调用将导致`reverse([y,z], [x|Acc])`的调用。最终`reverse/2`的第一个参数将归结为一个空列表，这时`reverse/2`的第二个子句将被匹配并另函数结束。

整个过程如下：

```
> lists:reverse([a,b,c,d]).
(0)lists:reverse([a,b,c,d])
(1).lists:reverse([a,b,c,d], [])
(2)..lists:reverse([b,c,d], [a])
(3)...lists:reverse([c,d], [b,a])
(4)....lists:reverse([d], [c,b,a])
(5).....lists:reverse([], [d,c,b,a])
(5).....[d,c,b,a]
(4)....[d,c,b,a]
(3)...[d,c,b,a]
(2)..[d,c,b,a]
(1).[d,c,b,a]
(0)[d,c,b,a]
[d,c,b,a]
```

delete_all

`delete_all(X, L)`用于删除列表`L`中出现的所有`X`。

```
delete_all(X, [X|T]) ->
    delete_all(X, T);
delete_all(X, [Y|T]) ->
    [Y | delete_all(X, T)];
delete_all(_, []) ->
    [].
```

`delete_all`所使用的递归模式与`member`和`append`类似。

`delete_all`的第一个子句在要删除的元素出现在列表的头部时匹配。

示例

在以下章节中我们将给出一些稍微复杂一些的列表处理函数的使用示例。

sort

程序3.1是著名的快速排序的一个变体。`sort(X)`对列表`X`的元素排序，将结果放入一个新列表并将之返回。

程序**3.1**

```
-module(sort).
-export([sort/1]).

sort([]) -> [];
sort([Pivot|Rest]) ->
    {Smaller, Bigger} = split(Pivot, Rest),
    lists:append(sort(Smaller), [Pivot|sort(Bigger)]).

split(Pivot, L) ->
    split(Pivot, L, [], []).

split(Pivot, [], Smaller, Bigger) ->
    {Smaller,Bigger};
split(Pivot, [H|T], Smaller, Bigger) when H < Pivot ->
    split(Pivot, T, [H|Smaller], Bigger);
split(Pivot, [H|T], Smaller, Bigger) when H >= Pivot ->
    split(Pivot, T, Smaller, [H|Bigger]).
```

此处选取列表的第一个元素为中轴。元列表被分为两个列表`Smaller`和`Bigger`：`Smaller`的所有元素都小于中轴`Pivot`而`Bigger`的所有元素都大于等于`Pivot`。之后，再对列表`Smaller`和`Bigger`分别排序并将结果合并。

函数`split({Pivot, L})`返回元组`{Smaller, Bigger}`，其中所有`Bigger`中的元素都大于等于`Pivot`而所有`Smaller`中的元素都小于`Pivot`。`split(Pivot, L)`通过调用一个辅助函数`split(Pivot, L, Smaller, Bigger)`完成任务。两个累加列表，`Smaller`和`Bigger`分别用于存储`L`中小于和大于等于`Pivot`的元素。`split/4`的代码与`reverse/2`非常相像，只是多用了一个累加列表。例如：

```
> lists:split(7,[2,1,4,23,6,8,43,9,3]).
{[3,6,4,1,2],[9,43,8,23]}
```

如果我们调用`sort([7,2,1,4,23,6,8,43,9,3])`，首先就会以`7`为中轴来调用`split/2`。这将产生两个列表：所有元素都小于中轴`7`的`[3,6,4,1,2]`，以及所有元素都大于等于中轴的`[9,43,8,23]`。

假设`sort`工作正常，则`sort([3,6,4,1,2])` `[1,2,3,4,6]`而`sort([9,43,8,23])` `[8,9,23,43]`。最后，排好序的列表被拼装在一起：

```
> append([1,2,3,4,6], [7 | [8,9,23,43]]).
[1,2,3,4,6,7,8,9,23,43]
```

再动一点脑筋，都`append`的调用也可以省掉，如下所示：

```
qsort(X) ->
    qsort(X, []).

%% qsort(A,B)
%%   Inputs:
%%      A = unsorted List
%%      B = sorted list where all elements in B
```

```
%%          are greater than any element in A
%%   Returns
%%      sort(A) appended to B

qsort([Pivot|Rest], Tail) ->
    {Smaller,Bigger} = split(Pivot, Rest),
    qsort(Smaller, [Pivot|qsort(Bigger,Tail)]);
qsort([], Tail) ->
    Tail.
```

我们可以利用BIF`statistics/1`（用于提供系统性能相关的信息，参见附录??）将之与第一版的`sort`做一个对比。如果我们编译并执行以下代码片段：

```
...
statistics(reductions),
lists:sort([2,1,4,23,6,7,8,43,9,4,7]),
{_, Reductions1} = statistics(reductions),
lists:qsort([2,1,4,23,6,7,8,43,9,4,7]),
{_, Reductions2} = statistics(reductions),
...
```

我们可以得知`sort`和`qsort`的归约（函数调用）次数。在我们的示例中`sort`花费`93`次归约，而`qsort`花费`74`次，提升了百分之二十。

集合

程序3.2是一组简单的集合操作函数。在Erlang中表示集合的最直白的方法就是采用一个不包含重复元素的无序列表。

集合操作函数如下：

`new()`

返回一个空集合。

`add_element(X, S)`

返回将元素`X`并入集合`S`产生的新集合。

`del_element(X, S)`

返回从集合`S`中删去元素`X`的新集合。

`is_element(X, S)`

当元素`X`在集合`S`中时返回`true`，否则返回`false`。

`is_empty(S)`

当集合`S`为空集时返回`true`，否则返回`false`。

`union(S1, S2)`

返回集合`S1`和`S2`的并集，即包含了`S1`或`S2`所有元素的集合。

`intersection(S1, S2)`

返回集合`S1`和`S2`的交集，即仅包含既包含于`S1`又包含于`S2`的元素的集合。

严格地说，我们并不能说`new`返回了一个空集，而应该说`new`返回了一个空集的表示。如果我们将集合表示为列表，则以上的集合操作可以编写如下：

程序**3.2**

```
-module(sets).
-export([new/0, add_element/2, del_element/2,
         is_element/2, is_empty/1, union/2, intersection/2]).

new() -> [].

add_element(X, Set) ->
    case is_element(X, Set) of
        true -> Set;
        false -> [X|Set]
    end.

del_element(X, [X|T]) -> T;
del_element(X, [Y|T]) -> [Y|del_element(X,T)];
del_element(_, [])    -> [].

is_element(H, [H|_])   -> true;
is_element(H, [_|Set]) -> is_element(H, Set);
is_element(_, [])      -> false.

is_empty([]) -> true;
is_empty(_)  -> false.

union([H|T], Set) -> union(T, add_element(H, Set));
union([], Set)    -> Set.

intersection(S1, S2)       -> intersection(S1, S2, []).
intersection([], _, S)     -> S;
intersection([H|T], S1, S) ->
    case is_element(H,S1) of
        true -> intersection(T, S1, [H|S]);
        false -> intersection(T, S1, S)
    end.
```

运行程序3.2的代码：

```
> S1 = sets:new().
[]
> S2 = sets:add_element(a, S1).
[a]
> S3 = sets:add_element(b, S2).
[b,a]
> sets:is_element(a, S3).
true
> sets:is_element(1, S2).
false
> T1 = sets:new().
[]
> T2 = sets:add_element(a, T1).
```

```
[a]
> T3 = sets:add_element(x, T2).
[x,a]
> sets:intersection(S3, T3).
[a]
10> sets:union(S3,T3).
[b,x,a]
```

这个实现并不十分高效，但足够简单以保证其正确性（但愿如此）。今后还可以将之替换为一套更高效的实现。

素数

在我们的最后一个例子（程序3.3）中，我们将来看看如何使用埃拉托色尼筛法来生成一张素数表。

程序 **3.3**

```
-module(siv).
-compile(export_all).

range(N, N) ->
    [N];
range(Min, Max) ->
    [Min | range(Min+1, Max)].

remove_multiples(N, [H|T]) when H rem N == 0 ->
    remove_multiples(N, T);
remove_multiples(N, [H|T]) ->
    [H | remove_multiples(N, T)];
remove_multiples(_, []) ->
    [].

sieve([H|T]) ->
    [H | sieve(remove_multiples(H, T))];
sieve([]) ->
    [].

primes(Max) ->
    sieve(range(2, Max)).
```

注意在程序3.3中我们使用了编译器标注`-compile(export_all)`——这将隐式地导出该模块中的所有函数，于是我们无须显式地给出导出申明便可以调用这些函数。

`range(Min, Max)`返回一个包含从`Min`到`Max`的所有整数的列表。

`remove_multiples(N, L)`从列表`L`删除中`N`的倍数：

```
> siv:range(1,15).
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
> siv:remove_multiples(3,[1,2,3,4,5,6,7,8,9,10]).
[1,2,4,5,7,8,10]
```

`sieve(L)`保留列表`L`的头部，对于尾部的列表，则再递归地删除其头部的所有倍数：

```
```

```
> siv:primes(25).
[2,3,5,7,11,13,17,19,23]
```

列表的常用递归模式

尽管一个典型的程序往往会使用很多不同的函数来操作列表，但大多数列表处理函数都是由少数几种模式演变而来。大部分列表处理函数无非就是在干着这些事情：

- 在一个列表中寻找一个元素，并在找到时做些事情。
- 对输入列表的每个元素做些事情并构造一个与其结构相同的输出列表。
- 在遇到列表中的第*n*个元素时做些事情。
- 对列表进行扫描，并构造一个或一组与原列表相关的新列表。

我们将以此对其进行讨论。

搜索列表元素

给定以下递归模式：

```
search(X, [X|T]) ->
    ... do something ...
    ...;
search(X, [_|T]) ->
    search(X, T);
search(X, []) ->
    ... didn't find it ...
```

第一种情况匹配的是找到了我们所感兴趣的项的情形。第二种情况在列表的头部不是我们所感兴趣的项时匹配，这时将接着处理列表的尾部。最后一种情况匹配的是列表元素耗尽的情形。

将以上代码与`member/2`的代码（第??节）作个比较，我们可以看到我们不过是把`... do something ...`换成了`true`，把`... didn't find it ...`换成了`false`。

构建同构列表

有时我们会想构造一个形如输入列表的列表，但同时又要对输入列表的每个元素做些操作。这时可以这么写：

```
isomorphic([X|T]) ->
    [something(X)|isomorphic(T)];
isomorphic([]) ->
    [].
```

然后，比如我们想写一个将给定列表中的所有元素翻倍的函数，我们就可以这么写：

```
double([H|T]) ->
    [2 * H | double(T)];
double([]) ->
    [].
```

于是便有：

```
> lists1:double([1,7,3,9,12]).
[2,14,6,18,24]
```

事实上这种手法只能作用于列表的最上层，因此如果我们想遍历列表的所有层次，我们就得将函数定义修改如下：

```
double([H|T]) when integer(H)->
    [2 * H | double(T)];
double([H|T]) when list(H) ->
    [double(H) | double(T)];
double([]) ->
    [].
```

后一个版本就可以成功遍历深层的嵌套列表了：

```
> lists1:double([1,2,[3,4],[5,[6,12],3]]).
[2,4,[6,8],[10,[12,24],6]]
```

计数

我们常常要使用到计数器，以便对一个列表的第*n*个元素做些动作：

```
count(Terminal, L) ->
    ... do something ...;
count(N, [_|L]) ->
    count(N-1, L).
```

则返回列表中第*n*个元素（假设其存在）的函数可以写成：

```
nth(1, [H|T]) ->
    H;
nth(N, [_|T]) ->
    nth(N - 1, T).
```

这种递减至一个终止条件的计数方式往往要由于递增计数。为了说明这一点，考虑同样是返回第*n*个元素但是采用递增计数的函数`nth1`：

```
nth1(N, L) ->
    nth1(1, N, L).
nth1(Max, Max, [H|_]) ->
    H;
nth1(N, Max, [_|T]) ->
    nth1(N+1, Max, T).
```

这种做法需要一个额外的参数和一个辅助函数。

收集列表元素

现在我们希望对一个列表中的元素做些动作，生成一个或一组新的列表。对应的模式如下：

```
collect(L) ->
    collect(L, []).

collect([H|T], Accumulator) ->
    case pred(H) of
        true ->
            collect(T, [dosomething(H)|Accumulator]);
        false ->
            collect(T, Accumulator)
    end;
collect([], Accumulator) ->
    Accumulator.
```

在这里我们引入了一个多出一个参数的辅助函数，多出的这个参数用于存储最终要被返回给调用方的列表。

借助这样一种模式，举个例子，我们可以写这样的一个函数：计算输入列表的所有偶元素的平方并删除所有奇元素：

```
funny(L) ->
    funny(L, []).

funny([H|T], Accumulator) ->
    case even(H) of
        true -> funny(T, [H*H|Accumulator]);
        false -> funny(T, Accumulator)
    end;
funny([], Accumulator) ->
    Accumulator.
```

于是有：

```
> lists:funny([1,2,3,4,5,6])
[36,16,4]
```

注意在这种情况下结果列表中的元素的顺序与原列表中对应元素的顺序是相反的。

在递归过程中使用累加列表来构造结果经常是一种推荐的做法。这样可以编写出运行时只适用常数空间的扁平的代码（细节参见第??节）。

函数式参数

将函数名作为参数传递给另一个函数是一种很有用的抽象特定函数行为的方法。本节将给出两个使用这种编程技术的示例。

map

函数`map(Func, List)`返回一个列表`L`，其中的元素由函数`Func`依次作用于列表`List`的各个元素得到。

```
map(Func, [H|T]) ->
    [apply(F, [H])|map(Func, T)];
map(Func, []) ->
    [].
```

```
> lists:map({math,factorial}, [1,2,3,4,5,6,7,8]).
[1,2,6,24,120,720,5040,40320]
```

filter

函数`filter(Pred, List)`对列表`List`中的元素进行过滤，仅保留令`Pred`的值为`true`的元素。这里的`Pred`是一个返回`true`或`false`的函数。

```
filter(Pred, [H|T]) ->
    case apply(Pred,[H]) of
        true ->
        [H|filter(Pred, T)];
    false ->
        filter(Pred, T)
    end;
filter(Pred, []) ->
    [].
```

假设函数`math:even/1`在参数为偶数时返回`true`，否则返回`fale`，则：

```
> lists:filter({math,even}, [1,2,3,4,5,6,7,8,9,10]).
[2,4,6,8,10]
```

脚注

[1] 标记`Lhs  Rhs`代表对`Lhs`求值的结果为`Rhs`。

第4章 使用元组

翻译：王飞
校订：连城

元组用以将多个对象组合成一个新的复杂对象。对象`{E1,E2,E3,...En}`表示一个大小为 **n** 的元组。元组用于描述包含固定数目的元素的数据结构；对于可变数目的元素，应该使用列表来存储。

处理元组的BIF

以下是一些可以用来操纵元组的BIF：

`tuple_to_list(T)`

将元组`T`转化成一个列表。

如：`tuple_to_list({1,2,3,4})` `[1,2,3,4]`。

`list_to_tuple(L)`

将列表`L`转化成一个元组。

如：`list_to_tuple([a,b,c])` `{a,b,c}`。

`element(N, T)`

返回元组`T`的第`N`个元素。

如：`element(3,{a,b,c,d})` `c`。

`setelement(N, T, Val)`

返回一个新的元组，这个元组是将元组`T`的第`N`个元素用`Val`替换之后的一个拷贝。

如：`setelement(3, {a,b,c,d}, xx)` `{a,b,xx,d}`。

`size(T)`

返回元组`T`包含的元素个数。

如：`size({a,b,c})` `3` 。

返回多个值

我们经常想让一个函数返回多个值，使用元组来实现这一目的是十分方便的。

例如，函数`parse_int(List)`从一个由ASCII字符构成的列表`List`中提取最开始的数字，如果存在，就返回一

个由被提取出来的数字和列表剩下的部分组成的元组，如果列表中没有数字的话，就返回原子式`eoString`。

```
parse_int(List) ->
    parse_int(skip_to_int(List), 0).

parse_int([H|T], N) when H >= $0, H =< $9 ->
    parse_int(T, 10 * N + H - $0);
parse_int([], 0) ->
    eoString;
parse_int(L, N) ->
    {N,L}.
```

`skip_to_int(L)`返回`L`中第一个以ASCII字符`0`到`9`中的任意一个开始的子列表。

```
skip_to_int([]) ->
    [];
skip_to_int([H|T]) when H >= $0, H =< $9 ->
    [H|T];
skip_to_int([H|T]) ->
    skip_to_int(T).
```

如果我们使用字符串`"abcd123def"`（`"abcd123def"`的列表形式是`[97,98,99,49,50,51,100,101,102]`）来测试`parse_int`：

```
> tuples:parse_int("abc123def").
{123,[100,101,102]}}
```

在`parse_int`的基础上，可以实现一个提取所有嵌入在字符串里面的数字的解释器。

```
parse_ints([]) ->
    [];
parse_ints(L) ->
    case parse_int(L) of
        eoString ->
            [];
        {H,Rest} ->
            [H|parse_ints(Rest)]
    end.
```

因此：

```
> tuples:parse_ints("abc,123,def,456,xx").
[123,456]
```

密码加密

几乎每天笔者们都不得不记住许多不同的密码——信用卡的密码，门禁密码等等。这些密码可以用一种方法记录下来，并且不会被犯罪分子利用吗？

假设我们有一张密码为`3451`的LISA信用卡，它的密码可以像这样被编码：

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
1 0 5 3 4 3 2 7 2 5 4 1 9 4 9 6 3 4 1 4 1 2 7 8 5 0   lisa
```

这样密码就可以写在一张纸上，即使这张纸落在他人手上，密码也是安全的。

我们如何解码信息呢？用来加密密码的密钥是公开的——因此我们可以很容易地读出密码（3451）–试试看！

我们很容易的就可以构造一个用来执行加密的函数`encode(Pin, Password)`[1]：

```
encode(Pin, Password) ->
    Code = {nil,nil,nil,nil,nil,nil,nil,nil,nil,
            nil,nil,nil,nil,nil,nil,nil,nil,nil,
            nil,nil,nil,nil,nil,nil,nil,nil},
    encode(Pin, Password, Code).

encode([], _, Code) ->
    Code;
encode(Pin, [], Code) ->
    io:format("Out of Letters~n",[]);

encode([H|T], [Letter|T1], Code) ->
    Arg = index(Letter) + 1,
    case element(Arg, Code) of
        nil ->
            encode(T, T1, setelement(Arg, Code, index(H)));
        _ ->
            encode([H|T], T1, Code)
    end.

index(X) when X >= $0, X =< $9 ->
    X - $0;

index(X) when X >= $A, X =< $Z ->
    X - $A.
```

我们看一下以下的例子：

```
> pin:encode("3451","DECLARATIVE").
{nil,nil,5,3,4,nil,nil,nil,nil,nil,nil,1,nil,nil,nil,
 nil,nil,nil,nil,nil,nil,nil,nil,nil,nil,nil,nil}
```

我们现在使用随机数来替换没有被填充的`nil`元素：

```
print_code([], Seed) ->
    Seed;

print_code([nil|T], Seed) ->
    NewSeed = ran(Seed),
    Digit = NewSeed rem 10,
    io:format("~w ",[Digit]),
    print_code(T, NewSeed);

print_code([H|T],Seed) ->
    io:format("~w ",[H]),
print_code(T, Seed).

ran(Seed) ->
    (125 * Seed + 1) rem 4096.
```

然后我们需要一些小函数将所有东西连接在一起：

```
test() ->
    title(),
    Password = "DECLARATIVE",
    entries([{"3451",Password,lisa},
             {"1234",Password,carwash},
             {"4321",Password,bigbank},
             {"7568",Password,doorcode1},
             {"8832",Password,doorcode2},
             {"4278",Password,cashcard},
             {"4278",Password,chequecard}]).

title() ->
    io:format("a b c d e f g h i j k l m \
               n o p q r s t u v w x y z~n",[]).

entries(List) ->
    {_,_,Seed} = time(),
    entries(List, Seed).

entries([], _) -> true;

entries([{Pin,Password,Title}|T], Seed) ->
    Code = encode(Pin, Password),
    NewSeed = print_code(tuple_to_list(Code), Seed),
    io:format(" ~w~n",[Title]),
    entries(T, NewSeed).
```

最后我们可以运行这个程序了：

```
1> pin:test().
a b c d e f g h i j k l m n o p q r s t u v w x y z
1 0 5 3 4 3 2 7 2 5 4 1 9 4 9 6 3 4 1 4 1 2 7 8 5 0 lisa
9 0 3 1 2 5 8 3 6 7 0 4 5 2 3 4 7 6 9 4 9 2 7 4 9 2 carwash
7 2 2 4 3 1 2 1 8 3 0 1 5 4 1 0 5 6 5 4 3 0 3 8 5 8 bigbank
1 0 6 7 5 7 6 9 4 5 4 8 3 2 1 0 7 6 1 4 9 6 5 8 3 4 doorcode1
1 4 3 8 8 3 2 5 6 1 4 2 7 2 9 4 5 2 3 6 9 4 3 2 5 8 doorcode2
7 4 7 4 2 5 6 5 8 5 8 8 9 4 7 6 5 0 1 2 9 0 9 6 3 8 cashcard
7 4 7 4 2 7 8 7 4 3 8 8 9 6 3 8 5 2 1 4 1 2 1 4 3 4 chequecard
true
```

之后这些信息可以用很小的字体打印出来，粘在一张邮票的背后，藏在你的领带里面[2]。

字典

我们将一组键惟一的`键–值`（`Key-Value`）对定义为字典[3]。存在字典里的值可能会重复。对`Key`和`Value`的数据类型都没有限制，但是只能通过`Key`来查询字典。

我们定义一下字典操作：

`new()`

创建并返回一个空字典。

`lookup(Key, Dict)`

在字典`Dict`中查找一个`Key-Value`对，如果找到则返回`{value, Value}`，否则返回`undefined`。

`add(Key, Value, Dict)`

添加一个新的`Key-Value`对到字典`Dict`中，并返回一个新的字典，以反映`add`函数对字典造成的改变。

`delete(Key, Dict)`

从字典`Dict`里删除`Key`所对应的`Key-Value`对，并返回一个新的字典。

程序4.1展示了一个字典是怎样将`Key-Value`对以元组的形式存放到列表里面的。它并不是实现一个字典最好的方法，在这里它只是一个例子。

程序**4.1**

```
-module(dictionary).
-export([new/0,lookup/2,add/3,delete/2]).

new() ->
    [].

lookup(Key, [{Key,Value}|Rest]) ->
    {value,Value};

lookup(Key, [Pair|Rest]) ->
    lookup(Key, Rest);

lookup(Key, []) ->
    undefined.

add(Key, Value, Dict) ->
    NewDict = delete(Key, Dict),
    [{Key,Value}|NewDict].

delete(Key, [{Key,Value}|Rest]) ->
    Rest;

delete(Key, [Pair|Rest]) ->
    [Pair|delete(Key, Rest)];
delete(Key, []) ->
    [].
```

我们用`字典`来构建和管理一个包含了各位作者鞋码的小型数据库：

```
D0 = dictionary:new().
[]
> D1 = dictionary:add(joe, 42, D0).
[{joe,42}]
> D2 = dictionary:add(mike, 41, D1).
[{mike,41},{joe,42}]
> D3 = dictionary:add(robert, 43, D2).
[{robert,43},{mike,41},{joe,42}]
> dictionary:lookup(joe, D3).
{value,42}
> dictionary:lookup(helen, D3).
undefined
...
```

非平衡二叉树

字典适合保存少量的数据项，但是当项的数量不断增加，更好的方法是用通过使用键的序列关系来访问数据的树形结构来组织数据。这种结构的访问时间与它所包含的项的数量成对数关系—列表是线性的访问时间。

我们认为最简单的树组织形式是非平衡二叉树。树内部的结点用`{Key, Vlue, Smaller, Bigger}`来表示。`Value`是被存储在树的一些结点中对象的值，它的键为`Key`。`Smaller`是一棵子树，它的所有结点的键值都小于`Key`，`Bigger`也是一棵子树，它的所有结点的键值都大于或等于`Key`。树的叶子用原子式`nil`表示。

我们从`lookup(Key, Tree)`函数开始，这个函数搜索`Tree`以确定树中是否有与`Key`相关的项。

```
lookup(Key, nil) ->
    not_found;

lookup(Key, {Key,Value,_,_}) ->
    {found,Value};

lookup(Key, {Key1,_,Smaller,_}) when Key < Key1 ->
    lookup(Key, Smaller);

lookup(Key, {Key1,_,_,Bigger}) when Key > Key1 ->
    lookup(Key, Bigger).
```

函数`insert(Key, Value, OldTree)`将数据`Key-Value`添加到树`OldTree`中，并返回一棵新树。

```
insert(Key, Value, nil) ->
    {Key,Value,nil,nil};

insert(Key, Value, {Key,_,Smaller,Bigger}) ->
    {Key,Value,Smaller,Bigger};

insert(Key, Value, {Key1,V,Smaller,Bigger}) when Key < Key1 ->
    {Key1,V,insert(Key, Value, Smaller),Bigger};

insert(Key, Value, {Key1,V,Smaller,Bigger}) when Key > Key1 ->
    {Key1,V,Smaller,insert(Key, Value, Bigger)}.
```

第一个子句得到数据，并插入到一棵新树当中，第二个子句将复写已经存在的结点，第三个和第四个子句确定当`Key`的值小于、大于或等于树中当前结点的`Key`时，应该采取什么样的行为。

当构建了一棵树之后，我们会想用一种方法将这棵树的结构打印出来。

```
write_tree(T) ->
    write_tree(0, T).

write_tree(D, nil) ->
    io:tab(D),
    io:format('nil', []);
write_tree(D, {Key,Value,Smaller,Bigger}) ->
    D1 = D + 4,
    write_tree(D1, Bigger),
    io:format('~n', []),
    io:tab(D),
    io:format('~w ===> ~w~n', [Key,Value]),
    write_tree(D1, Smaller).
```

我们可以用一个测试函数将数据插入到树中，并把它打印出来：

```
test1() ->
    S1 = nil,
    S2 = insert(4,joe,S1),
    S3 = insert(12,fred,S2),
    S4 = insert(3,jane,S3),
    S5 = insert(7,kalle,S4),
    S6 = insert(6,thomas,S5),
    S7 = insert(5,rickard,S6),
    S8 = insert(9,susan,S7),
    S9 = insert(2,tobbe,S8),
    S10 = insert(8,dan,S9),
    write_tree(S10).
```

图**4.1** 一棵非平衡二叉树

```
        nil
    12 ===> fred
                nil
            9 ===> susan
                    nil
                8 ===> dan
                    nil
        7 ===> kalle
                nil
            6 ===> thomas
                    nil
                5 ===> rickard
                    nil
4 ===> joe
        nil
    3 ===> jane
            nil
        2 ===> tobbe
            nil
```

注意这棵树并不是十分“平衡”。按照严格的顺序插入键的队列，比如像这样：

```
T1 = nil,
T2 = insert(1,a,T1),
T3 = insert(2,a,T2),
T4 = insert(3,a,T3),
T5 = insert(4,a,T4),
...
T9 = insert(8,a,T8).
```

使这棵树看起来变成了一个列表（见图4.2）。

当键的顺序随机的时候，我们使用的方法是很好的。如果在一个插入序列里，键是有序排列的，这棵树就变成了一个列表。我们将在第??章讲述怎样构建平衡二叉树。

图**4.2** 变化后的非平衡二叉树

```
                                    nil
```

```
                            8 ===> a
                                nil
                        7 ===> a
                            nil
                    6 ===> a
                        nil
                5 ===> a
                    nil
            4 ===> a
                nil
        3 ===> a
            nil
    2 ===> a
        nil
1 ===> a
    nil
```

我们也需要能够删除二叉树内的元素：

```
delete(Key, nil) ->
    nil;

delete(Key, {Key,_,nil,nil}) ->
    nil;

delete(Key, {Key,_,Smaller,nil}) ->
    Smaller;

delete(Key, {Key,_,nil,Bigger}) ->
    Bigger;

delete(Key, {Key1,_,Smaller,Bigger}) when Key == Key1 ->
    {K2,V2,Smaller2} = deletesp(Smaller),
    {K2,V2,Smaller2,Bigger};

delete(Key, {Key1,V,Smaller,Bigger}) when Key < Key1 ->
    {Key1,V,delete(Key, Smaller),Bigger};

delete(Key, {Key1,V,Smaller,Bigger}) when Key > Key1 ->
    {Key1,V,Smaller,delete(Key, Bigger)}.
```

当要删除的结点是树中的叶子，或者在这个结点下面只有一颗子树时，删除操作是很容易的（子句1到4）。子句6和7中，要删除的结点并没有被确定位置，而是继续在合适的子树中向前搜索。

在子句5当中，要删除的结点被找到，但是它是树中的一个内部结点（例如结点同时有`Smaller`和`Bigger`子树）。这种情况下，`Smaller`子树中具有最大键的结点将被删除，并且整棵树在这个点重建。

```
deletesp({Key,Value,nil,nil}) ->
    {Key,Value,nil};

deletesp({Key,Value,Smaller,nil}) ->
    {Key,Value,Smaller};

deletesp({Key,Value,Smaller,Bigger}) ->
    {K2,V2,Bigger2} = deletesp(Bigger),
    {K2,V2,{Key,Value,Smaller,Bigger2}}.
```

平衡二叉树

在前面几节里，我们学会了怎样构建一棵非平衡二叉树。但不幸的是非平衡二叉树可能会变成一个列表，这样对树的插入和删除操作就是非随机的了。

一个更好的方法是保持树在任何情况下都是平衡的。

Adelsom-Velskii和Landis [?]（在[?]中描述）使用一个简单的标准来衡量平衡这个概念：如果一棵树的每个结点的两个子树高度之差不超过1，我们就说这棵树是平衡的。具有这种特性的树常常被称作*AVL*树。平衡二叉树能够在`O(logN)`的时间规模里完成查找、插入和删除操作，`N`是树中结点的个数。

假设我们用元组`{Key, Value, Height, Smaller, Bigger}`表示一棵 AVL树，用`{_, _, 0, _, _}`表示一棵空树。然后在树中的查找操作就很容易实现了：

```
lookup(Key, {nil,nil,0,nil,nil}) ->
    not_found;

lookup(Key, {Key,Value,_,_,_}) ->
    {found,Value};

lookup(Key, {Key1,_,_,Smaller,Bigger}) when Key < Key1 ->
    lookup(Key,Smaller);

lookup(Key, {Key1,_,_,Smaller,Bigger}) when Key > Key1 ->
    lookup(Key,Bigger).
```

`lookup`的代码和非平衡二叉树中的基本一样。插入操作这样实现：

```
insert(Key, Value, {nil,nil,0,nil,nil}) ->
    E = empty_tree(),
    {Key,Value,1,E,E};

insert(Key, Value, {K2,V2,H2,S2,B2}) when Key == K2 ->
    {Key,Value,H2,S2,B2};

insert(Key, Value, {K2,V2,_,S2,B2}) when Key < K2 ->
    {K4,V4,_,S4,B4} = insert(Key, Value, S2),
combine(S4, K4, V4, B4, K2, V2, B2);

insert(Key, Value, {K2,V2,_,S2,B2}) when Key > K2 ->
    {K4,V4,_,S4,B4} = insert(Key, Value, B2),
    combine(S2, K2, V2, S4, K4, V4, B4).

empty_tree() ->
    {nil,nil,0,nil,nil}.
```

思路是找到要插入的项将被插入到什么地方，如果插入使得树变得不平衡了，那么就平衡它。平衡一棵树的操作通过`combine`函数实现[4]。

```
combine({K1,V1,H1,S1,B1},AK,AV,
        {K2,V2,H2,S2,B2},BK,BV,
        {K3,V3,H3,S3,B3} ) when H2 > H1, H2 > H3 ->
            {K2,V2,H1 + 2,
             {AK,AV,H1 + 1,{K1,V1,H1,S1,B1},S2},
             {BK,BV,H3 + 1,B2,{K3,V3,H3,S3,B3}}
            };
combine({K1,V1,H1,S1,B1},AK,AV,
        {K2,V2,H2,S2,B2},BK,BV,
```

```
        {K3,V3,H3,S3,B3} ) when H1 >= H2, H1 >= H3 ->
            HB = max_add_1(H2,H3),
    HA = max_add_1(H1,HB),
    {AK,AV,HA,
            {K1,V1,H1,S1,B1},
            {BK,BV,HB,{K2,V2,H2,S2,B2},{K3,V3,H3,S3,B3}}
           };
combine({K1,V1,H1,S1,B1},AK,AV,
        {K2,V2,H2,S2,B2},BK,BV,
        {K3,V3,H3,S3,B3} ) when H3 >= H1, H3 >= H2 ->
            HA = max_add_1(H1,H2),
    HB = max_add_1(HA,H3),
    {BK,BV,HB ,
            {AK,AV,HA,{K1,V1,H1,S1,B1},{K2,V2,H2,S2,B2}},
            {K3,V3,H3,S3,B3}
           }.

max_add_1(X,Y) when X =< Y ->
    Y + 1;
max_add_1(X,Y) when X > Y ->
    X + 1.
```

打印一棵树也很简单:

```
write_tree(T) ->
    write_tree(0, T).

write_tree(D, {nil,nil,0,nil,nil}) ->
    io:tab(D),
    io:format('nil', []);

write_tree(D, {Key,Value,_,Smaller,Bigger}) ->
    D1 = D + 4,
    write_tree(D1, Bigger),
    io:format('~n', []),
    io:tab(D),
    io:format('~w ===> ~w~n', [Key,Value]),
    write_tree(D1, Smaller).
```

现在让我们来看看我们的劳动成果。假设我们在一棵AVL树中插入键为`1,2,3,...,16`的16个数据。结果如图4.3，它是一棵平衡的树了（跟上一节那棵变形的树比较一下）。

最后是AVL树中的删除操作:

```
delete(Key, {nil,nil,0,nil,nil}) ->
    {nil,nil,0,nil,nil};

delete(Key, {Key,_,1,{nil,nil,0,nil,nil},{nil,nil,0,nil,nil}}) ->
    {nil,nil,0,nil,nil};

delete(Key, {Key,_,_,Smaller,{nil,nil,0,nil,nil}}) ->
    Smaller;

delete(Key, {Key,_,_,{nil,nil,0,nil,nil},Bigger}) ->
    Bigger;

delete(Key, {Key1,_,_,Smaller,{K3,V3,_,S3,B3}}) when Key == Key1 ->
    {K2,V2,Smaller2} = deletesp(Smaller),
    combine(Smaller2, K2, V2, S3, K3, V3, B3);

delete(Key, {K1,V1,_,Smaller,{K3,V3,_,S3,B3}}) when Key < K1 ->
```

```
    Smaller2 = delete(Key, Smaller),
    combine(Smaller2, K1, V1, S3, K3, V3, B3);

delete(Key, {K1,V1,_,{K3,V3,_,S3,B3},Bigger}) when Key > K1 ->
    Bigger2 = delete(Key, Bigger),
    combine( S3, K3, V3, B3, K1, V1, Bigger2).
```

图**4.3** 一棵平衡二叉树

```
                    nil
                16 ===> a
                    nil
            15 ===> a
                nil
        14 ===> a
                nil
            13 ===> a
                nil
    12 ===> a
                nil
            11 ===> a
                nil
        10 ===> a
                nil
            9 ===> a
                nil
8 ===> a
                nil
            7 ===> a
                nil
        6 ===> a
                nil
            5 ===> a
                nil
    4 ===> a
                nil
            3 ===> a
                nil
        2 ===> a
                nil
            1 ===> a
                nil
```

`deletisp`函数删除并返回树中最大的元素。

```
deletesp({Key,Value,1,{nil,nil,0,nil,nil},{nil,nil,0,nil,nil}}) ->
    {Key,Value,{nil,nil,0,nil,nil}};
deletesp({Key,Value,_,Smaller,{nil,nil,0,nil,nil}}) ->
    {Key,Value,Smaller};
deletesp({K1,V1,2,{nil,nil,0,nil,nil},
         {K2,V2,1,{nil,nil,0,nil,nil},{nil,nil,0,nil,nil}}}) ->
           {K2,V2,
            {K1,V1,1,{nil,nil,0,nil,nil},{nil,nil,0,nil,nil}}
           };

deletesp({Key,Value,_,{K3,V3,_,S3,B3},Bigger}) ->
        {K2,V2,Bigger2} = deletesp(Bigger),
        {K2,V2,combine(S3, K3, V3, B3, Key, Value, Bigger2)}.
```

脚注

[1] `encode/2`和本章其它一些例子的代码调用了`io`模块中的函数。这个模块是一个提供给用户进行格式化输入输出的标准模块。它的详细特性将在第??章和附录??中描述。

[2] 只有一个作者是系领带的。

[3] 这在数据库管理系统的数据字典里面是不用怀疑的。

[4] 有关合并规则的详细描述可以在第[??]章找到。

第5章 并行编程

翻译：张驰原
校订：连城

进程和进程见通信都是Erlang以及所有并行编程中最基本的概念，进程的创建和进程间的通信都是显式进行的。

进程的创建

一个进程是一个独立自治的计算单元，它与系统中其他的进程并行地存在。进程之间没有继承的层次关系，不过应用程序的设计者也可以显式地创建这样一个层次关系。

BIF `spawn/3`创建并开始执行一个新的进程，它的参数和`apply/3`是一样的：

```
Pid = spawn(Module, FunctionName, ArgumentList)
```

和`apply`不同的是，`spawn`并不是直接对函数进行求值并返回其结果，而是启动一个新的并行进程用于对函数进行求值，返回值是这个新创建的进程的`Pid`（进程标识符）。和一个进程的所有形式的交互都是通过`Pid`来进行的。`spawn/3`会在启动新进程之后立即返回，而不会等待它对函数完成求值过程。

在图5.1(a)中，我们有一个标识符为`Pid1`的进程调用了如下函数：

```
Pid2 = spawn(Mod, Func, Args)
```

在`spawn`返回之后，会有两个进程`Pid1`和`Pid2`并行地存在，状态如图5.1(b)所示。现在只有进程`Pid1`知道新进程的标识符，亦即`Pid2`。由于`Pid`是一切进程间通讯的必要元素，一个Erlang系统中的安全性也是建立在限制进程`Pid`分发的基础上的。

图5.1

当传递给`spawn`的函数执行完毕之后，进程会自动退出。这个顶层函数的返回值将被丢弃[1]。

进程标识符是一个普通的数据对象，可以像操作其他对象那样对其进行处理。例如，它可以被放到一个列表或元组中，可以与其他标识符进行比较，也可以当做消息发送给其他进程。

进程间通信

在Erlang中进行进程间通信的唯一方法就是消息传递。一个消息通过原语`!`（`send`）发送给另一个进程：

```
Pid ! Message
```

`Pid`是要向其发送消息的进程的标识符。任何合法的Erlang表达式都可以作为一个消息发送。`send`是一个会对其参数进行求值的原语。它的返回值是发送的消息。因此：

```
foo(12) ! bar(baz)
```

会分别对`foo(12)`和`bar(baz)`进行求值得到进程标识符和要发送的消息。如同其他的Erlang函数一样，`send`对其参数的求值顺序是不确定的。它会将消息参数求值的结果作为返回值返回。发送消息是一个异步操作，因此`send`既不会等待消息送达目的地也不会等待对方收到消息。就算发送消息的目标进程已经退出了，系统也不会通知发送者。这是为了保持消息传递的异步性——应用程序必须自己来实现各种形式的检查（见下文）。消息一定会被传递到接受者那里，并且保证是按照其发送的顺序进行传递的。

原语`receive`被用于接收消息。它的语法如下：

```
receive
    Message1 [when Guard1] ->
        Actions1 ;
    Message2 [when Guard2] ->
        Actions2 ;
    ...
end
```

每个进程都有一个邮箱，所有发送到该进程的消息都被按照它们到达的顺序依次存储在邮箱里。在上面的例子中，`Message1`和`Message2`是用于匹配进程邮箱中的消息的模式。当找到一个匹配的消息并且对应的保护式（Guard）满足的时候，这个消息就被选中，并从邮箱中删除，同时对应的`ActionsN`会被执行。`receive`会返回`ActionosN`中最后一个表达式求值的结果。就如同Erlang里其他形式的模式匹配一样，消息模式中未绑定（unbound）量会被绑定（bound）被`receive`选中的消息会按照原来的顺序继续留在邮箱中，用于下一次`recieve`的匹配。调用`receive`的进程会一直阻塞，直到有匹配的消息为止。

Erlang有一种选择性接收消息的机制，因此意外发送到一个进程的消息不会阻塞其它正常的消息。不过，由于所有未匹配的消息会被留在邮箱中，保证系统不要完全被这样的无关消息填满就变成了程序员的责任。

消息接收的顺序

当`receive`尝试寻找一个匹配的消息的时候，它会依次对邮箱中的每一个消息尝试用给定的每个模式去进行匹配。我们用下面的例子来解释其工作原理。

图5.2(a)给出了一个进程的邮箱，邮箱里面有四个消息，依次是`msg_1`、`msg_2`、`msg_3`和`msg_4`。运行

```
receive
    msg_3 ->
```

```
        ...
end
```

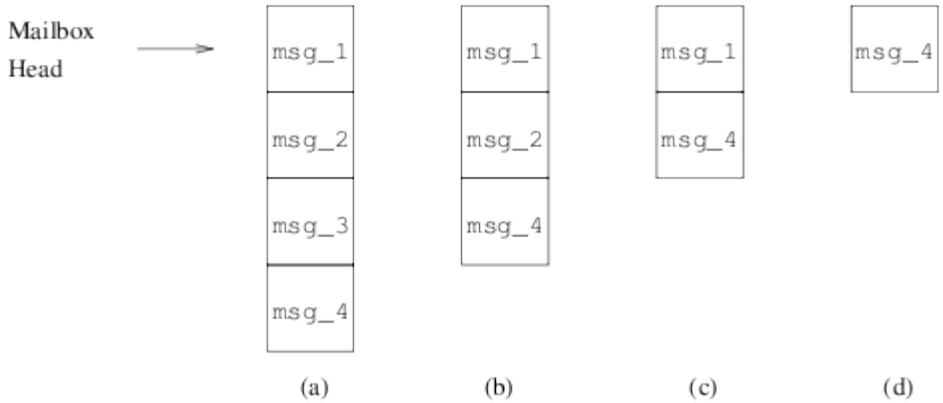

图5.2

会匹配到邮箱中的`msg_3`并导致它被从邮箱中删除。然后邮箱的状态会变成如图5.2(b)所示。当我们再运行

```
receive
    msg_4 ->
        ...
    msg_2 ->
        ...
end
```

的时候，`receive`会依次对邮箱中的每一个消息，首先尝试与`msg_4`匹配，然后尝试与`msg_2`匹配。结果是`msg_2`匹配成功并被从邮箱中删除，邮箱的状态变成图5.2(c)那样。最后，运行

```
receive
    AnyMessage ->
        ...
end
```

其中`AnyMessage`是一个未绑定（unbound）的变量，`receive`会匹配到邮箱里的`msg_1`并将其删除，邮箱中最终只剩下`msg_4`，如图5.2(d)所示。

这说明`receive`里的模式的顺序并不能直接用来实现消息的优先级，不过这可以通过超时的机制来实现，详见第??小节。

只接收来自某个特定进程的消息

有时候我们会只希望接收来自某一个特定进程的消息。要实现这个机制，消息发送者必须显式地在消息中包含自己的进程标识符：

```
Pid | {self(),abc}
```

BIF `self()`返回当前进程的标识符。这样的消息可以通过如下方式来接收：

```
receive
    {Pid,Msg} ->
        ...
```

```
end
```

如果`Pid`已经预先绑定（bound）到发送者的进程标识符上了，那么如上所示的`receive`就能实现只接收来自该进程[2]的消息了。

一些例子

程序5.1中的模块实现了一个简单的计数器，可以用来创建一个包含计数器的进程并对计数器进行递增操作。

程序 **5.1**

```
-module(counter).
-export([start/0,loop/1]).

start() ->
    spawn(counter, loop, [0]).

loop(Val) ->
    receive
        increment ->
            loop(Val + 1)
    end.
```

这个例子展示了一些基本概念：

- 每个新的计数器进程都通过调用`counter:start/0`来创建。每个进程都会以调用`counter:loop(0)`启动。
- 用于实现一个永久的进程的递归函数调用在等待输入的时候会被挂起。`loop`是一个尾递归函数，这让计数器进程所占用的空间保持为一个常数。
- 选择性的消息接收，在这个例子中，仅接收`increment`消息。

不过，在这过例子中也有不少缺陷，比如：

- 由于计数器的值是一个进程的局部变量，只能被自己访问到，却其他进程没法获取这个值。
- 消息协议是显式的，其他进程需要显式地发送`increment`消息给计数器进程。

程序**5.2**

```
-module(counter).
-export([start/0,loop/1,increment/1,value/1,stop/1]).

%% First the interface functions.
start() ->
    spawn(counter, loop, [0]).

increment(Counter) ->
    Counter ! increment.

value(Counter) ->
    Counter ! {self(),value}
    receive
        {Counter,Value} ->
```

```
            Value
    end.

stop(Counter) ->
    Counter ! stop.

%% The counter loop.
loop(Val) ->
    receive
        increment ->
            loop(Val + 1);
        {From,value} ->
            From ! {self(),Val},
            loop(Val);
        stop ->                  % No recursive call here
            true;
        Other ->                 % All other messages
            loop(Val)
    end.
```

下一个例子展示了如何修正这些缺陷。程序5.2是`counter`模块的改进版，允许对计数器进行递增、访问计数器的值以及停止计数器。

同前一个例子中一样，在这里一个新的计数器进程通过调用`counter::start()`启动起来，返回值是这个计数器的进程标识符。为了隐藏消息传递的协议，我们提供了接口函数`increment`、`value`和`stop`来操纵计数器。

计数器进程使用选择性接收的机制来处理发送过来的请求。它同时展示了一种处理未知消息的方法。通过在`receive`的最后一个子句中使用未绑定（unbound）的变量`Other`作为模式，任何未被之前的模式匹配到的消息都会被匹配到，此时我们直接忽略这样的未知消息并继续等待下一条消息。这是处理未知消息的标准方法：通过`receive`把它们从邮箱中删除掉。

为了访问计数器的值，我们必须将自己的`Pid`作为消息的一部分发送给计数器进程，这样它才能将回复发送回来。回复的消息中也包含了发送方的进程标识符（在这里也就是计数器进程的`Pid`），这使得接收进程可以只接收包含回复的这个消息。简单地等待一个包含未知值（在这个例子中是一个数字）的消息是不安全的做法，任何不相关的碰巧发送到该进程的消息都会被匹配到。因此，在进程之间发送的消息通常都会包含某种标识自己的机制，一种方法是通过内容进行标识，就像发送给计数器进程的请求消息一样，另一种方法是通过在消息中包含某种“唯一”并且可以很容易识别的标识符，就如同计数器进程发回的包含计数器值的回复消息一样。

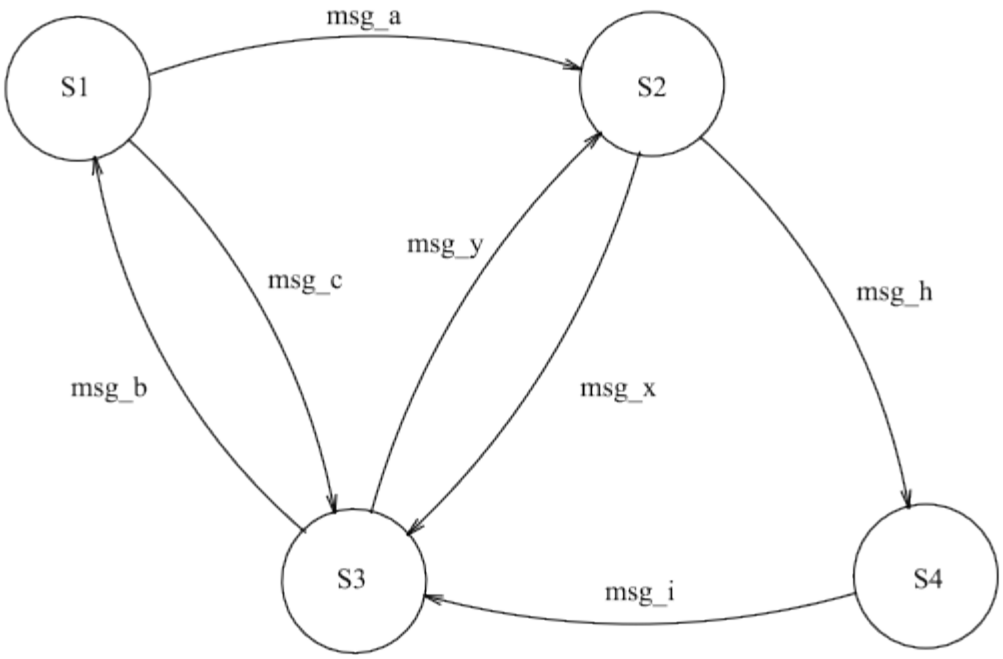

图5.3

现在我们再来考虑对一个有穷自动机（FSM）进行建模。图5.3展示了一个4状态的简单FSM以及可能的状态转移和相应的触发事件。一种编写这样的“状态-事件”机器的方法如程序5.3所示。在这段代码中，我们只专注于如何表示状态以及管理状态之间的转移。每个状态由一个单独的函数表示，而事件则表示为消息。

程序 **5.2**

```
s1() ->
    receive
        msg_a ->
            s2();
        msg_c ->
            s3()
    end.

s2() ->
    receive
        msg_x ->
            s3();
        msg_h ->
            s4()
    end.

s3() ->
    receive
        msg_b ->
            s1();
        msg_y ->
            s2()
    end.

s4() ->
    receive
        msg_i ->
            s3()
    end.
```

转台函数通过`receive`来等待事件所对应的消息。当收到消息时，FSM通过调用相应的状态函数转移到指定的状态。通过保证每次对于新状态的函数的调用都是最后一个语句（参见第??小节），FSM进程可以在一个常数大小的空间中进行求值。

状态数据可以通过为状态函数添加参数的方式来处理。需要在进入状态的时候执行的动作在调用`receive`之前完成，而需要在离开状态时执行的动作可以放在对应的`receive`子句中调用新的状态函数之前。

超时

Erlang中用于接收消息的基本原语`receive`可以通过添加一个可选的超时子句来进行增强，完整的语法变成这样：

```
receive
    Message1 [when Guard1] ->
        Actions1 ;
    Message2 [when Guard2] ->
        Actions2 ;
    ...
after
    TimeOutExpr ->
        ActionsT
end
```

`TimeOutExpr`是一个整数值表达式，表示毫秒数。时间的精确程度受到具体Erlang实现的底层操作系统以及硬件的限制——这是一个局部性问题（local issue）。如果在指定的时间内没有任何消息被匹配到，超时将会发生，`ActionsT`会被执行，而具体什么时候执行则是依赖与很多因素的，比如，和系统当前的负载有关系。

例如，对于一个窗口系统，类似于下面的代码可能会出现在处理事件的进程中：

```
get_event() ->
    receive
        {mouse, click} ->
            receive
                {mouse, click} ->
                    double_click
            after double_click_interval() ->
                single_click
            end
        ...
    end.
```

在这个模型中，事件由消息来表示。`get_event`函数会等待一个消息，然后返回一个表示对应事件的原子式。我们希望能检测鼠标双击，亦即在某一个较短时间段内的连续两次鼠标点击。当接收到一个鼠标点击事件时我们再通过`receive`试图接收下一个鼠标点击事件。不过，我们为这个`receive`添加了一个超时，如果在指定的时间内（由`double_click_interval`指定）没有发生下一次鼠标点击事件，`receive`就会超时，此时`get_event`会返回`single_click`。如果第二个鼠标点击事件在给定的超时时限之内被接收到了，那么`get_event`将会返回`double_click`。

在超时表达式的参数中有两个值有特殊意义：

`infinity`

原子式`infinity`表示超时永远也不会发生。如果超时时间需要在运行时计算的话，这个功能就很有用。我们可能会希望通过对一个表达式进行求值来得到超时长度：如果返回值是`infinity`的话，则永久等待。

`0`

数值0表示超时会立即发生，不过在那之前系统仍然会首先尝试对邮箱中已有的消息进行匹配。

在`receive`中使用超时比一下子想象到的要有用得多。函数`sleep(Time)`将当前进程挂起`Time`毫秒：

```
sleep(Time) ->
    receive
        after Time ->
            true
    end.
```

`flush_buffer()`清空当前进程的邮箱：

```
flush_buffer() ->
    receive
        AnyMessage ->
            flush_buffer()
        after 0 ->
            true
    end.
```

只要邮箱中还有消息，第一个消息会被匹配到（未绑定变量`AnyMessage`会匹配到任何消息，在这里就是第一个消息），然后`flush_buffer`会再次被调用，但是如果邮箱已经为空了，那么函数会从超时子句中返回。

消息的优先级也可以通过使用0作为超时长度来实现：

```
priority_receive() ->
    receive
        interrupt ->
            interrupt
    after 0 ->
        receive
            AnyMessage ->
                AnyMessage
        end
    end
```

函数`priority_receive`会返回邮箱中第一个消息，除非有消息`interrupt`发送到了邮箱中，此时将返回`interrupt`。通过首先使用超时时长0来调用`receive`去匹配`interrupt`，我们可以检查邮箱中是否已经有了这个消息。如果是，我们就返回它，否则，我们再通过模式`AnyMessage`去调用`receive`，这将选中邮箱中的第一条消息。

程序 **5.4**

```
-module(timer).
-export([timeout/2,cancel/1,timer/3]).
```

```
timeout(Time, Alarm) ->
    spawn(timer, timer, [self(),Time,Alarm]).

cancel(Timer) ->
    Timer ! {self(),cancel}.

timer(Pid, Time, Alarm) ->
    receive
        {Pid,cancel} ->
            true
    after Time ->
        Pid ! Alarm
    end.
```

在`receive`中的超时纯粹是在`receive`语句内部的，不过，要创建一个全局的超时机制也很容易。在程序5.4中的`timer`模块中的`timer::timeout(Time,Alarm)`函数就实现了这个功能。

调用`timer:timeout(Time, Alarm)`会导致消息`Alarm`在时间`Time`之后被发送到调用进程。该函数返回计时器进程的标识符。当进程完成自己的任务之后，可以使用该计时器进程标识符来等待这个消息。通过调用`timer::cancel(Timer)`，进程也可以使用这个标识符来撤销计时器。需要注意的是，调用`timer:cancel`并不能保证调用进程不会收到`Alarm`消息，这是由于`cancel`消息有可能在`Alarm`消息被发送出去之后才被收到的。

注册进程

为了向一个进程发送消息，我们需要事先知道它的进程标识符（`Pid`）。在某些情况下，这有些不切实际甚至不太合理。比如，在一个大型系统中通常存在许多全局服务器，或者某个进程由于安全方面的考虑希望隐藏它自己的标识符。为了让一个进程在并不事先知道对方的进程标识符的情况下向其发送消息，我们提供了注册进程的机制，换句话说，给进程一个名字。注册进程的名字必须是一个原子式。

基本原语

Erlang提供了四个BIF来操纵注册进程的名字：

`register(Name, Pid)`

> 将原子式`Name`关联到进程`Pid`。

`unregister(Name)`

> 删除原子式`Name`与对应进程的关联。

`whereis(Name)`

> 返回关联到注册名`Name`的进程标识符，如果没有任何进程关联到这个名字，则返回原子式`undefined`。

`registered()`

> 返回一个包含所有当前已注册过的名字。

消息发送的原语“`!`”允许直接使用一个注册进程的名字作为目标，例如：

```
number_analyzer ! {self(), {analyse,[1,2,3,4]}}
```

表示将消息`{Pid,{analyse,[1,2,3,4]}}`发送到注册为`numeber_analyser`的进程那里。`Pid`是调用`send`的进程的标识符。

“客户端-服务端”模型

注册进程的一个主要用途就是用于支持“客户端-服务端”模型编程。在这个模型中有一个服务端管理着一些资源，一些客户端通过向服务端发送请求来访问这些资源，如图5.4所示。要实现这个模型，我们需要三个基本组件——一个服务端，一个协议和一个访问库。我们将通过几个例子来阐明基本原则。

在先前的程序5.2中展示的`counter`模块里，每一根计数器都是一个服务端。客户端通过调用模块所定义的函数来访问服务端。

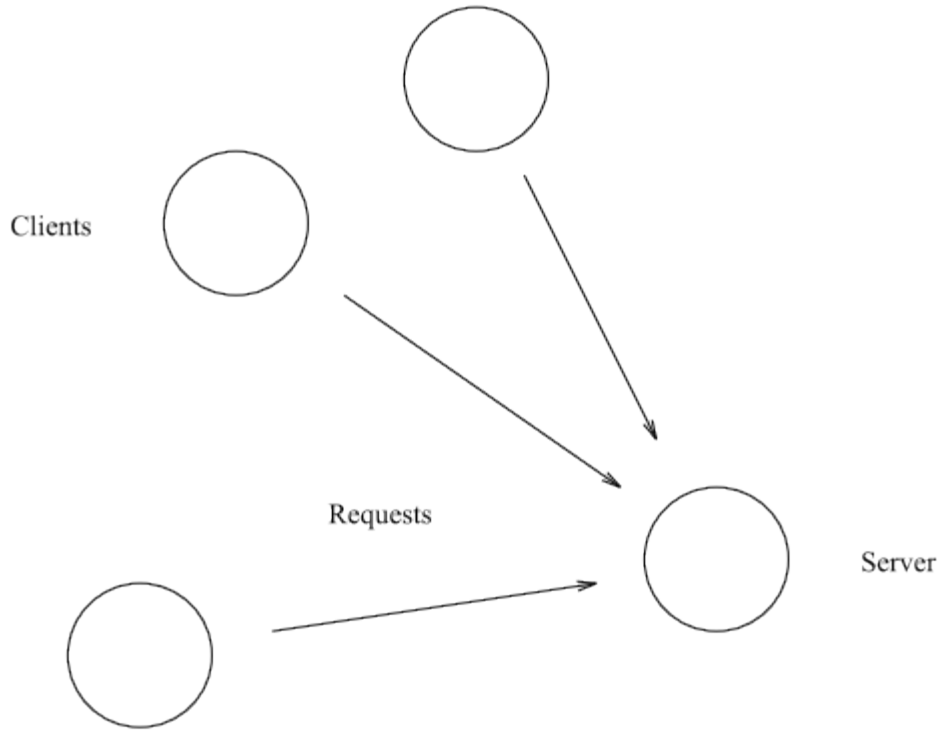

图5.4

程序5.5中展示的例子是一个可以用于电话交换机系统里分析用户所拨打的号码的服务端。`start()`会调用`spawn`并将新建的进程注册为`number_analyser`，这就完成了号码分析服务端的创建。之后服务端进程会在`server`函数中不断循环并等待服务请求。如果收到了一个形如`{add_number,Seq,Dest}`的请求，该号码序列（`Seq`）以及对应的目标进程（`Dest`），以及分析出结果之后将会发送的目的地，会被添加到查找表中。这是由函数`insert`完成的。之后消息`ack`将会被发送到请求的进程。如果服务端收到了形如`{analyse,Seq}`的消息，那么它将通过调用`lookup`完成号码序列`Seq`的分析，并将包含分析结果的消息发回发送请求的进程。我们在这里没有给出函数`insert`和`lookup`的具体定义，因为那对于我们目前讨论的问题而言并不重要。

客户端发送到服务端的请求消息包含了自己的进程标识符。这让服务端可以向客户端发送回复。发回的回复消息中也包含了一个“发送者”的标识，在这里就是服务端的注册名字，这使得客户端可以选择性地接收回复消息。这比简单地等待第一个消息到达要更加安全一些——因为客户端的邮箱中也许已经有了一些消

息，或者其他进程也许会在服务端回复之前给客户端发送一些消息。

程序 **5.5**

```
-module(number_analyser).
-export([start/0,server/1]).
-export([add_number/2,analyse/1]).

start() ->
    register(number_analyser,
             spawn(number_analyser, server, [nil])).

%% The interface functions.
add_number(Seq, Dest) ->
    request({add_number,Seq,Dest}).

analyse(Seq) ->
    request({analyse,Seq}).

request(Req) ->
    number_analyser ! {self(), Req},
    receive
        {number_analyser, Reply} ->
            Reply
    end.

%% The server.
server(AnalTable) ->
    receive
        {From, {analyse,Seq}} ->
            Result = lookup(Seq, AnalTable),
            From ! {number_analyser, Result},
            server(AnalTable);
        {From, {add_number, Seq, Dest}} ->
            From ! {number_analyser, ack},
            server(insert(Seq, Dest, AnalTable))
    end.
```

现在我们已经实现了服务端并定义了协议。我们在这里使用了一个异步协议，每个发送到服务端的请求都会有一个回复。在服务端的回复中我们使用`number_analyser`（亦即服务端的注册名字）作为发送者标识，这样做是因为我们不希望暴露服务端的`Pid`。

接下来我们定义一些接口函数用于以一种标准的方式访问服务端。函数`add_number`和`analyse`按照上面描述的方式实现了客户端的协议。它们都使用了局部函数`request`来发送请求并接收回复。

程序**5.6**

```
-module(allocator).
-export([start/1,server/2,allocate/0,free/1]).

start(Resources) ->
    Pid = spawn(allocator, server, [Resources,[]]),
    register(resource_alloc, Pid).

% The interface functions.
allocate() ->
    request(alloc).
```

```
free(Resource) ->
    request({free,Resource}).

request(Request) ->
    resource_alloc ! {self(),Request},
    receive
        {resource_alloc,Reply} ->
            Reply
    end.

% The server.
server(Free, Allocated) ->
    receive
        {From,alloc} ->
            allocate(Free, Allocated, From);
        {From,{Free,R}} ->
            free(Free, Allocated, From, R)
    end.

allocate([R|Free], Allocated, From) ->
    From ! {resource_alloc,{yes,R}},
    server(Free, [{R,From}|Allocated]);
allocate([], Allocated, From) ->
    From ! {resource_alloc,no},
    server([], Allocated).

free(Free, Allocated, From, R) ->
    case lists:member({R,From}, Allocated) of
        true ->
            From ! {resource_alloc,ok},
            server([R|Free], lists:delete({R,From}, Allocated));
        false ->
            From ! {resource_alloc,error},
            server(Free, Allocated)
    end.
```

下一个例子是如程序5.6中所示的一个简单的资源分配器。服务端通过一个需要管理的初始的资源列表来启动。其他进程可以向服务端请求分配一个资源或者将不再使用的资源释放掉。

服务端进程维护两个列表，一个是未分配的资源列表，另一个是已分配的资源列表。通过将资源在两个列表之间移动，服务端可以追踪每个资源的分配情况。

当服务端收到一个请求分配资源的消息时，函数`allocate/3`会被调用，它会检查是否有未分配的资源存在，如果是则将资源放在回复给客户端的`yes`消息中发送回去，否则直接发回`no`消息。未分配资源列表是一个包含所有未分配资源的列表，而已分配资源列表是一个二元组`{Resource,AllocPid}`的列表。在一个资源被释放之前，亦即从已分配列表中删除并添加到未分配列表中去之前，我们首先会检查它是不是一个已知的资源，如果不是的话，就返回`error`。

讨论

接口函数的目的是创建一个抽象层并隐藏客户端和服务端之间使用的协议的细节。一个服务的用户在使用服务的时候并不需要知道协议的细节或者服务端所使用的内部数据结构以及算法。一个服务的具体实现可以在保证外部用户接口一致性的情况下自由地更改这些内部细节。

此外，回复服务请求的进程还有可能并不是实际的服务器进程，而是一个不同的进程——所有的请求都被

委转发到它那里。实际上，“一个”服务器可能会是一个巨大的进程网络，这些互通的进程一起实现了给定的服务，但是却被接口函数隐藏起来。应当发布的是接口函数的集合，它们应当被暴露给用户，因为这些函数提供了唯一合法的访问服务端提供的服务的方式。

在Erlang中实现的“客户端-服务端”模型是非常灵活的。*monitor*或*remote procedure call*之类的机制可以很容易地实现出来。在某些特殊的情况下，具体实现也可以绕过接口函数直接与服务端进行交互。由于Erlang并没有强制创建或使用这样的接口函数，因此需要由系统设计师来保证在需要的时候创建它们。Erlang并没有提供用于远程过程调用之类的现成解决方案，而是提供了一些基本原语用于构造这样的解决方案。

进程调度，实时性以及优先级

到目前为止我们还没有提到过一个Erlang系统中的进程是如何调度的。虽然这是一个实现相关的问题，但是也有一些所有实现都需要满足的准则：

- 调度算法必须是公平的，换句话说，任何可以运行的进程都会被执行，并且（如果可能的话）按照它们变得可以运行的顺序来执行。
- 不允许任意一个进程长期阻塞整个系统。一个进程被分配一小段运行时间（称为时间片），再那之后它将被挂起并等待下一次运行调度，以使得其他可运行的进程也有机会运行。

典型情况下，时间片被设为可以让当前进程完成500次规约（reduction）[3]的时间。

Erlang语言实现的一个要求是要保证让它能够适用于编写软实时的应用程序，也就是说，系统的反应时间必须至少是毫秒级别的。一个满足以上准则的调度算法通常对于一个这样的Erlang实现来说已经足够了。

要让Erlang系统能应用于实时应用程序的另一个重要的特性是内存管理。Erlang对用户隐藏了所有的内存管理工作。内存在需要的时候被自动分配并在不需要之后一段时间内会被自动回收。内存的分配和回收的实现必须要保证不会长时间地阻塞系统的运行，最好是比一个时间片更短，以保证不会影响其实时性。

进程优先级

所有新创建的进程都在运行在同一个优先级上。不过有时候也许会希望一些进程以一个比其他进程更高或更低的优先级运行：例如，一个用于跟踪系统状态的进程也许只需要偶尔运行一下。BIF `process_flag`可以用来改变进程的优先级：

```
process_flag(priority, Pri)
```

`Pri`是进程的新的优先级，可以是`normal`或者`low`，这将改变调用该BIF的进程的运行优先级。优先级为`normal`的进程会比优先级为`low`的进程运行得更加频繁一些。所有进程默认的优先级都是`normal`。

进程组

所有Erlang进程都有一个与其相关联的`Pid`，称作进程的组长。当一个新进程被创建时，它会被自动归属到调用`spawn`语句的那个进程所属的进程组中。一开始，系统中的第一关进程是它自身的组长，因此也是所有后来创建的进程的组长。这表示所有的Erlang进程被组织为一个树形结构，第一个进程是树根。

以下的BIF可以被用于操控进程组：

`group_leader()`

返回调用该BIF的进程的组长`Pid`。

`group_leader(Leader, Pid)`

将进程`Pid`的组长设置为`Leader`。

Erlang的输入输出系统中用到了进程组的概念，详见第??章的描述。

脚注

[1] 因为并没有专门用于存放这些计算结果的地方。
[2] 或者其他知道该进程标识符的进程。
[3] 一次规约（reduction）等价于一次函数调用。

第6章 分布式编程

翻译：Ken Zhao
校订：连城

本章描述如何编写运行于Erlang节点网络上的分布式Erlang程序。我们描述了用于实现分布式系统的语言原语。Erlang进程可以自然地映射到分布式系统之中；同时，之前章节所介绍的Erlang并发原语和错误检测原语在分布式系统和单节点系统中仍保持原有属性。

动机

我们有很多理由去编写分布式应用，比如：

速度

> 我们可以把我们的程序切分成能够分别运行于多个不同节点的几个部分。比如，某个编译器可以将一个模块里的各个函数分发到不同节点分别编译，编译器本身则负责协调各节点的活动。
>
> 在例如一个具备一个节点池的实时系统，作业以round-robin的方式指派给不同的节点，以此降低系统的响应延迟。

可靠性和容错

> 为了增加系统的可靠性，我们可以部署多个互相协作的节点，以求一个或多个节点的失败不致影响整个系统的运作。

访问其他节点上的资源

> 某些软硬件资源可能只可被特定的计算机访问。

秉承应用固有的分布式特质

> 会议系统、订票系统以及许多多计算机实时系统都属于这类应用。

可扩展性

> 系统可以被设计成能够通过添加额外节点来增加系统的容量的形式。如果系统太慢，购买更多的处理器便可提高性能。

分布式机制

以下的BIF可用于分布式编程：

`spawn(Node, Mod, Func, Args)`

> 在远程节点产生一个新的进程。

`spawn_link(Node, Mod, Func, Args)`

在远程节点产生一个新的进程并创建一个指向这个进程的链接。

`monitor_node(Node, Flag)`

若`Flag`为`true`，该BIF令当前进程监视节点`Node`。如果`Node`出错或消失，一个`{nodedown, Node}`消息将被发送给当前进程，若`Flag`为`false`，则关闭监视。

`node()`

返回当前节点名称。

`nodes()`

返回已知的所有其他节点的名称列表。

`node(Item)`

返回`Item`所处节点的名称。`Item`可以是Pid、引用或端口。

`disconnect_node(Nodename)`

断开与节点`Nodename`的连接。

节点是分布式Erlang的一个核心概念。在分布式Erlang系统中，术语节点指一个可参与分布式Erlang事务的运行着的Erlang系统。独立的Erlang可通过启动一个称为网络内核的特殊进程来加入一个分布式Erlang系统。这个进程将计算BIF`alive/2`。网络内核将在??详述。一旦启动了网络内核，系统就处于活动状态。

处于活动状态的系统会被分配一个节点名称，该名称可以通过BIF `node(Item)`获得。该名称是一个全局唯一的原子式。不同的Erlang实现中节点名称的格式可能不同，但总是一个被`@`分为两部分的原子式。

BIF `node(Item)`返回创建`Item`的节点的名称，其中`Item`是一个Pid、端口或引用。

BIF `nodes/0`返回网络中与当前节点连接的所有其他节点的名称列表。

BIF `monitor_node(Node, Flag)`可用于监视节点。当节点`Node`失败或到`Node`的网络连接失败时，执行了`monitor_node(Node, true)`的进程将收到消息`{nodedown, Node}`。不幸的是，我们无法区分节点失败和网络失败。例如，以下代码会一直挂起到节点`Node`失败为止：

```
.....
monitor_node(Node, true),
receive
    {nodedown, Node} ->
        .....
end,
        .....
```

如果连接不存在，且`monitor_node/2`被调用，系统将尝试建立连接；若连接建立失败则投递一个`nodedown`消息。若针对同一节点连续两次调用`monitor_node/2`则在节点失败时将投递两条`nodedown`消息。

对`monitor_node(Node, false)`的调用只是递减一个计数器，该计数器用于记录`Node`失败时需要向调用进程发

送的`nodedown`消息的数量。之所以这么做，是因为我们往往会用一对匹配的`monitor_node(Node, true)`和`monitor_node(Node, false)`来封装远程过程调用。

BIF `spawn/3`和`spawn_link/3`用于在本地节点创建新进程。要在任意的节点创建进程，需要使用BIF `spawn/4`，所以：

```
Pid = spawn(Node, Mod, Func, Args),
```

将在`Node`产生一个进程，而`spawn_link/4`会在远程节点产生一个进程并建立一个与当前进程的链接。

这两个BIF各自会返回一个Pid。若节点不存在，也会返回一个Pid，当然由于没有实际的进程被执行，这个Pid没什么用处。对于`spawn_link/4`，在节点不存在的情况下当前进程会收到一个“`EXIT`”消息。

几乎所有针对本地Pid的操作同样都对远程Pid有效。消息可以被发送至远程进程，也可以在本地进程和远程进程间建立链接，就好像远程进程执行于本地节点一样。这意味着，比方说，发送给远程进程的消息总是按发送顺序传送、不会受损也不会丢失。这些都是由运行时系统来保障的。消息接收的唯一可能的错误控制，就是由程序员掌控的`link`机制，以及消息发送方和接收方的显式同步。

注册进程

BIF `register/2`用于在本地节点上为进程注册一个名称。我们可以这样向远程节点的注册进程发送消息：

```
{Name, Node} ! Mess.
```

若在节点`Node`上存在一个注册为名称`Name`的进程，则`Mess`将被发送到该进程。若节点或注册进程不存在，则消息被丢弃。

连接

Erlang节点间存在一个语言层面的连接概念。系统初被启动时，系统无法“觉察”任何其他节点，对`nodes()`求值将返回`[]`。与其他节点间的连接不是由程序员显式建立的。到远程节点`N`的连接是在`N`首次被引用时建立的。如下所示：

```
1> nodes().
   []
2> P = spawn('klacke@super.eua.ericsson.se', M, F, A).
   <24.16.1>
3> nodes().
   ['klacke@super.eua.ericsson.se']
4> node(P).
   'klacke@super.eua.ericsson.se'
```

要想建立到远程节点的连接，我们只需要在任意涉及远程节点的表达式中引用到节点的名称即可。检测网络错误的唯一手段就是使用链接BIF或`monitor_node/2`。要断开与某节点的连接可使用BIF `disconnect_node(Node)`。

节点之间是松散耦合的。节点可以像进程一样动态地被创建或消失。耦合不那么松散的系统可以通过配置文件和配置数据来实现。在生产环境下，通常只会部署固定数目个具备固定名称的节点。

银行业务示例

这一节我们将展示如何结合BIF `monitor_node/2`和向远程节点的注册进程发送消息的能力。我们将实现一个非常简单的银行服务，用以处理远程站点的请求，比如ATM机上存款、取款业务。

程序**6.1**

```
-module(bank_server).
-export([start/0, server/1]).

start() ->
    register(bank_server, spawn(bank_server, server, [[]])).

server(Data) ->
    receive
        {From, {deposit, Who, Amount}} ->
            From ! {bank_server, ok},
            server(deposit(Who, Amount, Data));
        {From, {ask, Who}} ->
            From ! {bank_server, lookup(Who, Data)},
            server(Data);
        {From, {withdraw, Who, Amount}} ->
            case lookup(Who, Data) of
                undefined ->
                    From ! {bank_server, no},
                    server(Data);
                Balance when Balance > Amount ->
                    From ! {bank_server, ok},
                    server(deposit(Who, -Amount, Data));
                _ ->
                    From ! {bank_server, no},
                    server(Data)
            end
    end.

lookup(Who, [{Who, Value}|_]) -> Value;
lookup(Who, [_|T]) -> lookup(Who, T);
lookup(_, _) -> undefined.

deposit(Who, X, [{Who, Balance}|T]) ->
    [{Who, Balance+X}|T];
deposit(Who, X, [H|T]) ->
    [H|deposit(Who, X, T)];
deposit(Who, X, []) ->
    [{Who, X}].
```

程序6.1的代码运行于银行总部。而在出纳机（或分行）中执行的是程序6.2，该程序完成与总行服务器的交互。

程序**6.2**

```
-module(bank_client).
-export([ask/1, deposit/2, withdraw/2]).

head_office() -> 'bank@super.eua.ericsson.se'.
```

```
ask(Who) ->              call_bank({ask, Who}).
deposit(Who, Amount)  -> call_bank({deposit, Who, Amount}).
withdraw(Who, Amount) -> call_bank({withdraw, Who, Amount}).
call_bank(Msg) ->
    Headoffice = head_office(),
    monitor_node(Headoffice, true),
    {bank_server, Headoffice} ! {self(), Msg},
    receive
        {bank_server, Reply} ->
            monitor_node(Headoffice, false),
            Reply;
        {nodedown, Headoffice} ->
            no
    end.
```

客户端程序定义了三个访问总行服务器的接口函数:

`ask(Who)`

返回客户`Who`的余额

`deposit(Who, Amount)`

给客户`Who`的帐户里面存入资金数`Amount`

`withdraw(Who, Amount)`

尝试从客户`Who`的帐户里面取出资金数`Amount`

函数`call_bank/1`实现了远程过程调用。一旦总行节点停止运作，`call_bank/1`将会及时发现，并返回`no`。

总行节点的名称是硬编码在源码中的。在后续章节中我们将展示集中隐藏该信息的手段。

第7章 错误处理

翻译：丁豪
校对：连城

即便是Erlang程序员也难免会写出有问题的程序。代码中的语法错误（和一些语义错误）可以借助编译器检测出来，但程序仍可能含有逻辑错误。对需求理解的偏差或对需求实现的不完备所造成的逻辑错误只能通过大量的一致性测试来检测。其他的错误则以运行时错误的形式出现。

函数是在Erlang进程中执行的。函数可能出于多种原因而失败，比如：

- 一次匹配操作失败
- 使用错误的参数调用BIF
- 我们可能打算对一个算术表达式求值，然而其中的一个项式并不是数值

Erlang本身当然无法修正这些情况，但它为程序员提供了一些检测和处理失败情况的机制。借助这些机制，程序员可以设计出健壮和容错的系统。Erlang具备如下机制：

- 监视表达式的求值
- 监视其他进程的行为
- 捕获对未定义函数的求值

Catch和Throw

`catch`和`throw`提供了一种表达式求值的监视机制，可以用于

- 处理顺序代码中的错误（`catch`）
- 函数的非本地返回（`catch`结合`throw`）

表达式求值失败（如一次匹配失败）的一般后果是导致求值进程的异常退出。通过以下方式可以借助`catch`来更改这个默认行为：

```
catch Expression
```

若表达式的求值过程没有发生错误，则`catch Expression`返回`Expression`的值。于是`catch atom_to_list(abc)`会返回`[97,98,99]`、`catch 22`会返回`22`。

若求值过程失败，`catch Expression`将返回元组`{'EXIT', Reason}`，其中`Reason`是用于指明错误原因的原子式（参见第??节）。于是`catch an_atom - 2`会返回`{'EXIT', badarith}`、`catch atom_to_list(123)`会返回`{'EXIT', badarg}`。

函数执行结束后，控制流程便返还者。`throw/1`可以令控制流程跳过调用者。如果我们像上述的那样计算`catch Expression`，并在`Expression`的求值过程中调用`throw/1`，则控制流程将直接返回至`catch`。注意`catch`可以嵌套；在嵌套的情况下，一次失败或`throw`将返回至最近的`catch`处。在`catch`之外调用`throw/1`将导致运行时错误。

下面的例子描述了`catch`和`throw`的行为。定义函数`foo/1`：

```
foo(1) ->
    hello;
foo(2) ->
    throw({myerror, abc});
foo(3) ->
    tuple_to_list(a);
foo(4) ->
    exit({myExit, 222}).
```

假设在不使用`catch`的情况下，一个进程标识为`Pid`的进程执行了这个函数，则：

`foo(1)`

返回`hello`。

`foo(2)`

执行`throw({myerror,abc})`。由于不在`catch`的作用域内，执行`foo(2)`的进程将出错退出。

`foo(3)`

执行`foo(3)`的进程执行BIF `tuple_to_list(a)`。这个BIF用于将元组转换为列表。在这个例子中，参数不是元组，因此该进程将出错退出。

`foo(4)`

执行BIF `exit/1`。由于不在`catch`的范围内，执行`foo(4)`的函数将退出。很快我们就会看到参数`{myExit,222}`的用途。

`foo(5)`

执行`foo(5)`的进程将出错退出，因为函数`foo/1`的首部无法匹配`foo(5)`。

现在让我们来看看在`catch`的作用域内对`foo/1`以相同的参数进行求值会发生什么：

```
demo(X) ->
    case catch foo(X) of
        {myerror, Args} ->
            {user_error, Args};
        {'EXIT', What} ->
            {caught_error, What};
        Other ->
            Other
    end.
```

`demo(1)`

像原来一样执行`hello`。因为没有任何失败发生，而我们也没有执行`throw`，所以`catch`直接返回`foo(1)`的求值结果。

`demo(2)`

求值结果为`{user_error,abc}`。对`throw({myerror,abc})`的求值导致外围的`catch`返回`{myerror,abc}`同时`case`语句返回`{user_error,abc}`。

`demo(3)`

求值结果为`{caught_error,badarg}`。`foo(3)`执行失败导致`catch`返回`{'EXIT',badarg}`。

`demo(4)`

求值结果为`{caught_error,{myexit,222}}`。

`demo(5)`

求值结果为`{caught_error,function_clause}`。

注意，在`catch`的作用域内，借助`{'EXIT', Message}`，你能够很容易地“伪造”一次失败——这是一个设计决策[1]。

使用catch和throw抵御不良代码

下面来看一个简单的Erlang shell脚本：

```
-module(s_shell).
-export([go/0]).

go() ->
    eval(io:parse_exprs('=> ')),    % '=>' is the prompt
    go().

eval({form,Exprs}) ->
    case catch eval:exprs(Exprs, []) of % Note the catch
        {'EXIT', What} ->
            io:format("Error: ~w!~n", [What]);
        {value, What, _} ->
            io:format("Result: ~w~n", [What])
    end;
eval(_) ->
    io:format("Syntax Error!~n", []).
```

标准库函数`io:parse_exprs/1`读取并解析一个Erlang表达式，若表达式合法，则返回`{form,Exprs}`。

正确情况下，应该匹配到第一个子句`eval({form,Expr})`并调用库函数`eval:exprs/2`对表达式进行求值。由于无法得知表达式的求值过程是否为失败，我们在此使用`catch`进行保护。例如，对`1 - a`进行求值将导致错误，但在`catch`内对`1 - a`求值就可以捕捉这个错误[2]。借助`catch`，在求值失败时，`case`子句与模式`{'EXIT',what}`匹配，在求值成功时则会与`{value, What, _}`匹配。

使用catch和throw实现函数的非本地返回

假设我们要编写一个用于识别简单整数列表的解析器，可以编写如下的代码：

```
parse_list(['[',']' | T])
    {nil, T};
parse_list(['[', X | T]) when integer(X) ->
```

```
    {Tail, T1} = parse_list_tail(T),
    {{cons, X, Tail}, T1}.

parse_list_tail([',', X | T]) when integer(X) ->
    {Tail, T1} = parse_list_tail(T),
    {{cons, X, Tail}, T1};
parse_list_tail([']' | T]) ->
    {nil, T}.
```

例如：

```
> parse_list(['[',12,',',20,']']).
{{cons,12,{cons,20,nil}},[]}
```

要是我们试图解析一个非法的列表，就会导致如下的错误：

```
> try:parse_list(['[',12,',',a]).
!!! Error in process <0.16.1> in function
!!!     try:parse_list_tail([',',a])
!!! reason function_clause
** exited: function_clause **
```

如果我们想在跳出递归调用的同时仍然掌握是哪里发生了错误，可以这样做：

```
parse_list1(['[',']' | T]) ->
    {nil, T};
parse_list1(['[', X | T]) when integer(X) ->
    {Tail, T1} = parse_list_tail1(T),
    {{cons, X, Tail}, T1};
parse_list1(X) ->
    throw({illegal_token, X}).

parse_list_tail1([',', X | T]) when integer(X) ->
    {Tail, T1} = parse_list_tail1(T),
    {{cons, X, Tail}, T1};
parse_list_tail1([']' | T]) ->
    {nil, T};
parse_list_tail1(X) ->
    throw({illegal_list_tail, X}).
```

现在，如果我们在`catch`里对`parse_list/1`求值，将获得以下结果：

```
> catch parse_list1(['[',12,',',a]).
{illegal_list_tail,[',',a]}
```

通过这种方式，我们得以从递归中直接退出，而不必沿着通常的递归调用路径逐步折回。

进程终止

当一个进程的进程执行函数（通过`spawn/4`创建进程时第3个参数所指定的函数）执行完毕，或是（在`catch`之外）执行`exit(normal)`，便会正常退出。参见程序7.1：

`test:start()`

创建一个注册名为`my_name`的进程来执行`test:process()`。

程序**7.1**

```
-module(test).
-export([process/0, start/0]).
start() ->
    register(my_name, spawn(test, process, [])).
process() ->
    receive
        {stop, Method} ->
            case Method of
                return ->
                    true;
                Other ->
                    exit(normal)
            end;
        Other ->
            process()
    end.
```

`my_name ! {stop, return}`

令`test:process()`返回`true`，接着进程正常终止。

`my_name ! {stop, hello}`

也会令进程正常终止，因为它执行了BIF `exit(normal)`。

任何其它的消息，比如`my_name ! any_other_message`都将令进程递归执行`test:process()`（采用尾递归优化的方式，参见第??章）从而避免进程终止。

若进程执行BIF `exit(Reason)`，则进程将异常终止。其中`Reason`是除了原子式`normal`以外的任意的Erlang项式。如我们所见，在`catch`上下文中执行`exit(Reason)`不会导致进程退出。

进程在执行到会导致运行时失败的代码（如除零错误）时，也会异常终止。后续还会讨论各种类型的运行时失败。

链接进程

进程可以互相监视。这里要引入两个概念，进程链接和`EXIT`信号。在执行期间，进程可以与其他进程（和端口，参见??章节）建立链接。当一个进程终止（无论正常或非正常终止）时，一个特殊的`EXIT`信号将被发送到所有与即将终止的进程相链接的进程（及端口）。该信号的格式如下：

```
{'EXIT', Exiting_Process_Id, Reason}
```

`Exiting_Process_Id`是即将终止的进程的进程标识，`Reason`可以是任意的Erlang项式。

收到`Reason`不是原子式`normal`的`EXIT`信号时，信号接收进程的默认动作是立即终止并，同时向当前与之链接的进程发送`EXIT`信号。默认情况下，`Reason`为原子式`normal`的`EXIT`信号将被忽略。

`EXIT`信号的默认处理方式行为可以被覆写，以允许进程在接收到`EXIT`信号时采取任意必要的动作。

创建和删除链接

进程可以链接到其它进程和端口。进程间的链接都是双向的，也就是说，如果进程A链接到进程B，那么进程B也会自动链接到进程A。

通过执行BIF `link(Pid)`便可创建链接。调用`link(Pid)`时，若调用进程和`Pid`之间已经存在链接，则不会产生任何影响。

进程终止时，它所持有的链接都将被删除。也可以通过执行BIF `unlink(Pid)`显式删除链接。由于所有链接都是双向的，删除这一端到另一端的链接的同时，另一端的到这一端的链接也会被删除。若调用进程和`Pid`之间原本就没有链接，`unlink(Pid)`不会产生任何影响。

BIF `spawn_link/3`在创建新进程的同时还会在调用进程和新进程间建立链接。其行为可以定义为：

```
spawn_link(Module, Function, ArgumentList) ->
    link(Id = spawn(Module, Function, ArgumentList)),
    Id.
```

只不过`spawn`和`link`是原子方式执行的。这是为了避免调用进程在执行`link`之前就被`EXIT`信号杀死。尝试向一个不存在的进程发起链接将导致信号`{'EXIT', Pid, noproc}`被发送至`link(Pid)`的调用进程。

程序7.2中，函数`start/1`建立了若干以链式互联的进程，其中第一个进程的注册名为`start`（参见图7.1）。函数`test/1`向该注册进程发送消息。每个进程不断打印自己在链中的位置及收到的消息。消息`stop`令链中最后一个进程执行BIF `exit(finished)`，该BIF将导致该进程异常终止。

程序**7.2**

```
-module(normal).
-export([start/1, p1/1, test/1]).

start(N) ->
    register(start, spawn_link(normal, p1, [N - 1])).

p1(0) ->
    top1();
p1(N) ->
    top(spawn_link(normal, p1, [N - 1]),N).

top(Next, N) ->
    receive
         X ->
             Next ! X,
             io:format("Process ~w received ~w~n", [N,X]),
             top(Next,N)
    end.

top1() ->
    receive
         stop ->
             io:format("Last process now exiting ~n", []),
             exit(finished);
         X ->
             io:format("Last process received ~w~n", [X]),
             top1()
```

```
    end.

test(Mess) ->
    start ! Mess.
```

我们启动三个进程（参见图7.1(a)）

```
> normal:start(3).
true
```

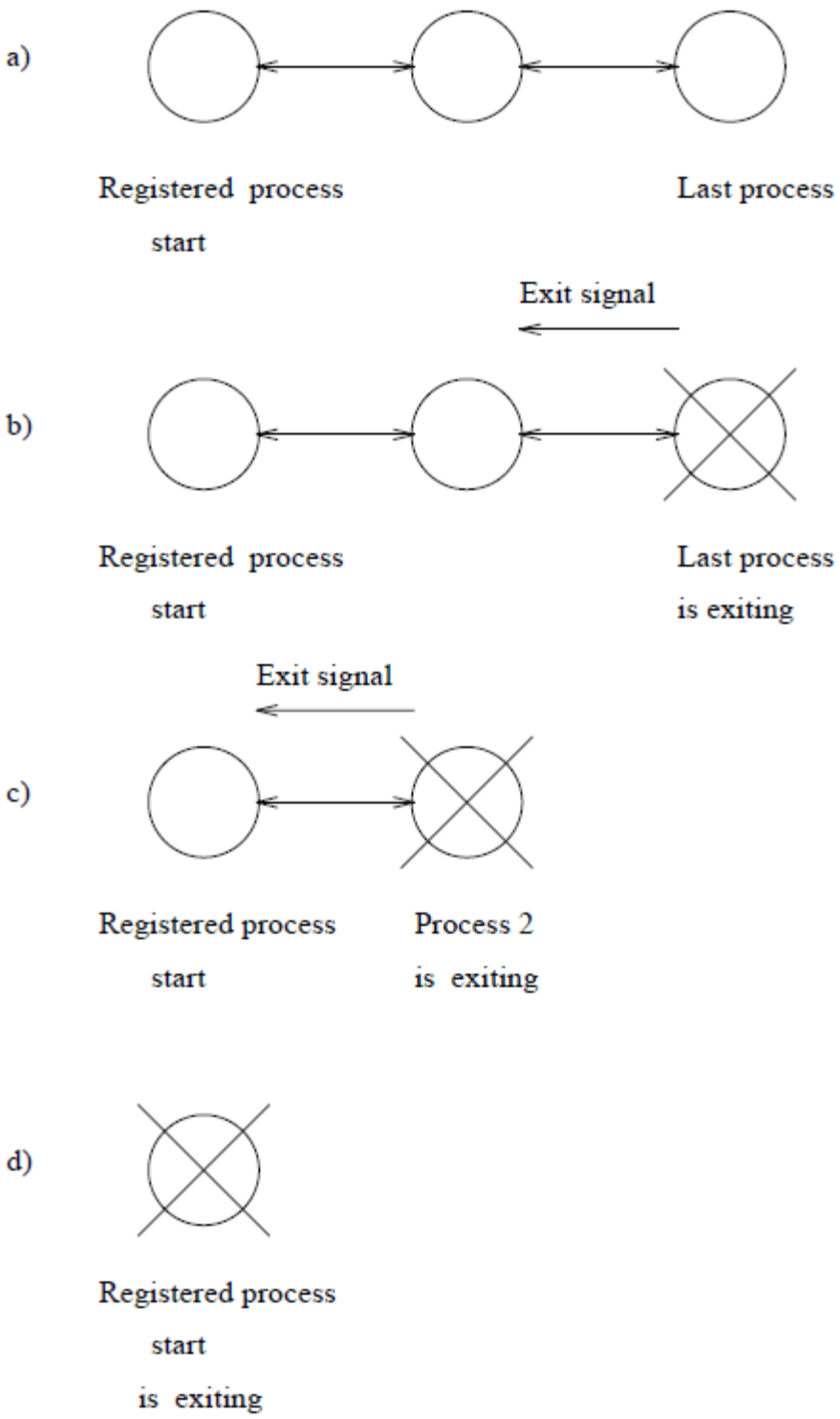

图7.1 进程退出信号的传递

然后向第一个进程发送消息`123`：

```
> normal:test(123).
Process 2 received 123
Process 1 received 123
Last process received 123
123
```

再向第一个进程发送消息`stop`：

```
> normal:test(stop).
Process 2 received stop
Process 1 received stop
Last process now exiting
stop
```

这条消息顺着进程链传递下去，我们将看到它最终导致链中最后一个进程的终止。这会引发一个发送给倒数第二个进程的`EXIT`信号，致其异常终止（图7.1(b)），接着又向第一个进程发送`EXIT`信号（图7.1(c)），于是注册进程`start`也异常终止（图 7.1(d)）。

若这时再向注册进程`start`发送一条新消息，将由于目标进程不存在而失败：

```
> normal:test(456).
!!! Error in process <0.42.1> in function
!!!     normal:test(456)
!!! reason badarg
** exited: badarg **
```

运行时失败

如前所述，`catch`作用域以外的运行时失败将导致进程的异常终止。进程终止时，将向与其链接的所有进程发送`EXIT`信号。这些信号包括一个指明失败原因的原子式。常见的失败原因如下：

`badmatch`

> 匹配失败。例如，尝试匹配`1 = 3`的进程将终止并向链接进程发送`EXIT`信号`{'EXIT', From, badmatch}`。

`badarg`

> BIF调用参数错误。例如，执行`atom_to_list(123)`将导致调用进程终止，并向链接进程发送`EXIT`信号`{'EXIT', From, badarg}`。因为`123`不是原子式。

`case_clause`

> 缺少匹配的`case`语句分支。例如，若进程执行：
>
> ```
> M = 3,
> case M of
>     1 ->
>         yes;
>     2 ->
> ```

```
        no
end.
```

则进程将终止，并向所有链接进程发送`EXIT`信号`{'EXIT', From, case_clause}`。

`if_clause`

缺少匹配的`if`语句分支。例如，若进程执行：

```
M = 3,
if
     M == 1 ->
         yes;
     M == 2 ->
         no
end.
```

则进程将终止，并向所有链接进程发送`EXIT`信号`{'EXIT', From, if_clause}`。

`function_clause`

缺少能够匹配函数调用参数列表的函数首部。例如，对如下的`foo/1`定义调用`foo(3)`：

```
foo(1) ->
     yes;
foo(2) ->
     no.
```

则调用进程终止，并向所有链接进程发送`EXIT`信号`{'EXIT', From, function_clause}`。

`undef`

尝试执行未定义函数的进程将终止并向所有链接进程发送`{'EXIT', From, undef}`（参见第??节）。

`badarith`

执行非法算术表达式（如，`1 + foo`）将导致进程终止，并向所有链接进程发送`{'EXIT', Pid, badarith}`。

`timeout_value`

`receive`表达式中出现非法超时值；如超时值既不是整数也不是原子式`infinity`。

`nocatch`

执行了`throw`语句却没有对应的`catch`。

自定义默认的信号接收动作

BIF `process_flag/2`可用于自定义进程接收到`EXIT`信号时所采取的默认行为。如下所述，执行`process_flag(trap_exit,true)`将改变默认行为，而`process_flag(trap_exit,false)`重新恢复默认行为。

如前所述，`EXIT`信号的格式如下：

```
{'EXIT', Exiting_Process_Id, Reason}
```

调用了`process_flag(trap_exit,true)`的进程接收到其他进程发送的`EXIT`信号后不再会自动终止。所有`EXIT`信号，包括`Reason`为原子式`normal`的信号，都将被转换为消息，进程可以以接收其他消息同样的方式来接收这些消息。程序7.3说明了进程如何互相链接以及执行了`process_flag(trap_exit,true)`的进程如何接收`EXIT`信号。

```
-module(link_demo).
-export([start/0, demo/0, demonstrate_normal/0, demonstrate_exit/1,
         demonstrate_error/0, demonstrate_message/1]).

start() ->
    register(demo, spawn(link_demo, demo, [])).

demo() ->
    process_flag(trap_exit, true),
    demo1().

demo1() ->
    receive
         {'EXIT', From, normal} ->
             io:format(
                 "Demo process received normal exit from ~w~n",
                 [From]),
             demo1();
         {'EXIT', From, Reason} ->
             io:format(
                 "Demo process received exit signal ~w from ~w~n",
                 [Reason, From]),
             demo1();
         finished_demo ->
             io:format("Demo finished ~n", []);
         Other ->
             io:format("Demo process message ~w~n", [Other]),
             demo1()
    end.

demonstrate_normal() ->
    link(whereis(demo)).

demonstrate_exit(What) ->
    link(whereis(demo)),
    exit(What).

demonstrate_message(What) ->
    demo ! What.

demonstrate_error() ->
    link(whereis(demo)),
    1 = 2.
```

示例代码的启动方式如下：

```
> link_demo:start().
true
```

`link_demo:start()`以函数`demo/0`启动一个进程并用名字`demo`进行注册。`demo/0`关闭`EXIT`信号的默认处理机制并调用`demo1/0`等待新消息的到来。

我们来考察一次正常退出过程：

```
> link_demo:demonstrate_normal().
true
Demo process received normal exit from <0.13.1>
```

执行`demonstrate_normal/0`的进程（在这个例子中该进程由Erlang shell创建）寻找注册进程`demo`的进程标识并与之建立链接。函数`demostrate_normal/0`没有别的子句，它的执行进程无事可做因而正常终止，从而引发信号：

```
{'EXIT', Process_Id, normal}
```

该信号被发送到注册进程`demo`。注册进程`demo`正在等待`EXIT`信号，因此它将之转换为一条消息，该消息在函数`demo1/0`内被接收，并输出文本（参见图7.2）：

```
Demo process received normal exit from <0.13.1>
```

接着`demo1/0`继续递归调用自身。

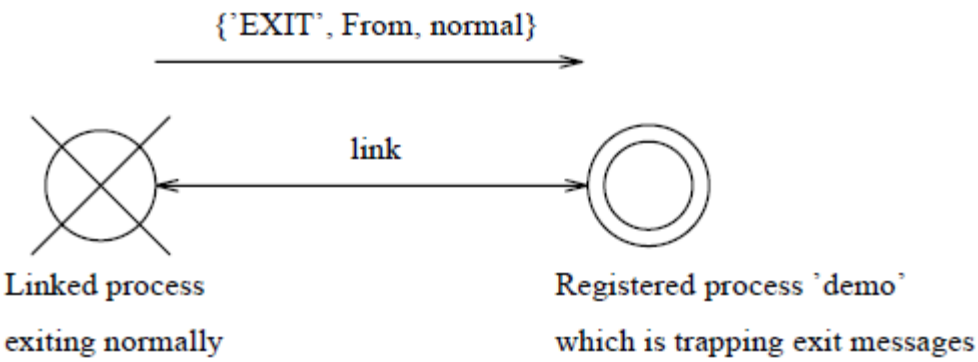

图7.2 正常退出信号

下面再来考察一次异常退出过程：

```
> link_demo:demonstrate_exit(hello).
Demo process received exit signal hello from <0.14.1>
** exited: hello **
```

和`demonstrate_normal/0`相同，`demonstrate_exit/1`创建一个到注册进程`demo`的链接。该例中，`demonstrate_exit/1`通过`exit(hello)`调用BIF `exit/1`。这导致`demostrate_exit/1`的执行进程异常终止，并将信号：

```
{'EXIT', Process_Id, hello}
```

发送给注册进程`demo`（参见图7.3）。注册进程`demo`将该信号转换为消息，并在函数`demo1/0`内被接收，从而输出文本：

```
```

```
Demo process received exit signal hello from <0.14.1>
```

接着`demo1/0`继续递归调用自身。

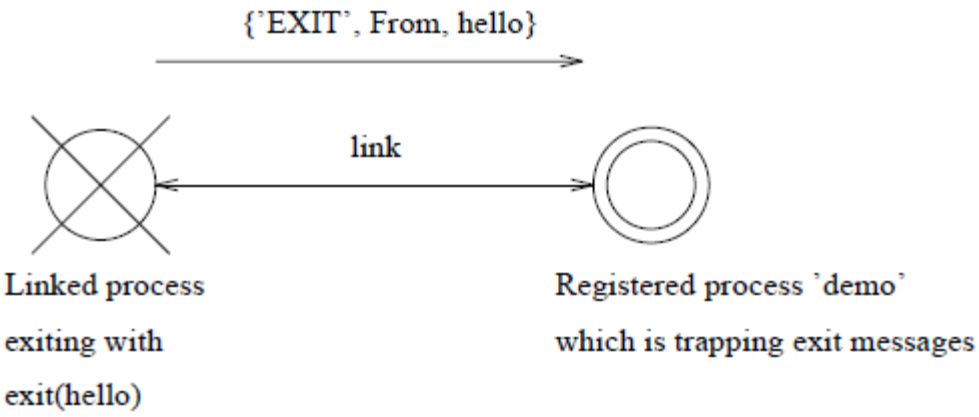

图7.3 执行`exit(hello)`

下一个案例中（如图7.4）我们将看到`link_demo:demonstrate_normal()`和`link_demo:demonstrate_exit(normal)`是等同的：

```
> link_demo:demonstrate_exit(normal).
Demo process received normal exit from <0.13.1>
** exited: normal **
```

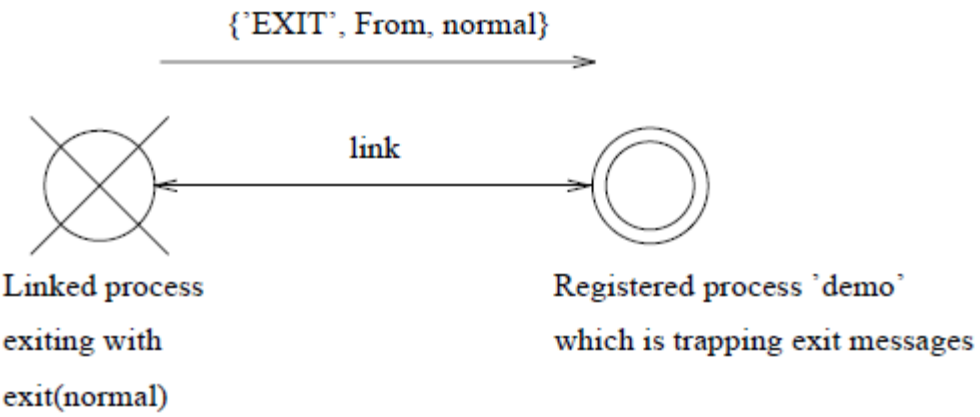

图7.4 执行`exit(normal)`

下一个案例将展示出现运行时错误时，会发生什么事：

```
> link_demo:demonstrate_error().
!!! Error in process <0.17.1> in function
!!!     link_demo:demonstrate_error()
!!! reason badmatch
** exited: badmatch **
Demo process received exit signal badmatch from <0.17.1>
```

向前面一样，`link_demo:demonstrate_error/0`创建一个到注册进程`demo`的链接。`link_demo:demonstrate_error/0`错误地试图匹配`1 = 2`。 该错误导致`link_demo:demonstrate_error/0`的执行进程异常终止，并发送信号`{'EXIT', Process_Id, badmatch}`至注册进程`demo`（参见图7.5）。

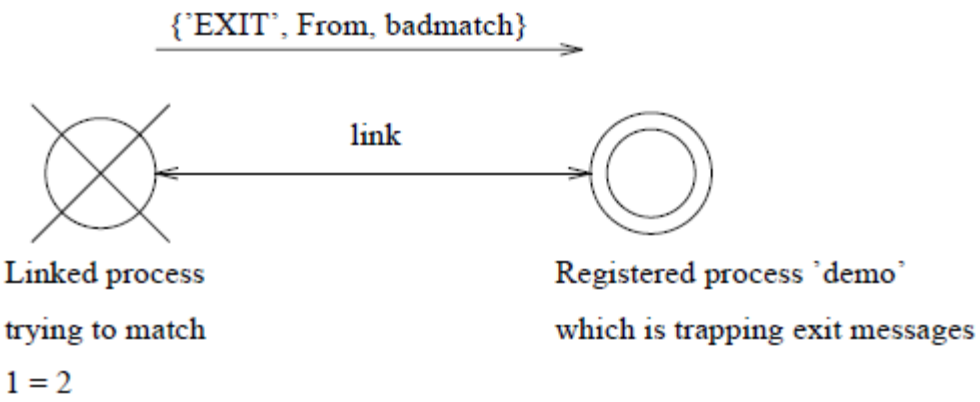

图7.5 匹配错误导致的进程失败

下一个案例中我们简单地向正在等待消息的注册进程`demo`发送消息`hello`：

```
> link_demo:demonstrate_message(hello).
Demo process message hello
hello
```

没有链接被创建，也就没有`EXIT`信号被发送或被接收。

通过以下调用来结束这个示例：

```
> link_demo:demonstrate_message(finished_demo).
Demo finished
finished_demo
```

未定义函数和未注册名称

最后一类错误关注的是当进程试图执行一个未定义的函数或者给一个未注册的名称发送消息时会发生什么。

调用未定义函数

如果进程尝试调用`Mod:Func(Arg0,...,ArgN)`，而该函数未被定义，则该调用被转换为：

```
error_handler:undefined_function(Mod, Func, [Arg0,...,ArgN])
```

假设模块`error_handler`已经被加载（标准发行版中预定义了`error_handler`模块）。`error_handler`模块可以被定义为程序7.4。

程序 **7.4**

```
-module(error_handler).
-export([undefined_function/3]).

undefined_function(Module, Func, Args) ->
    case code:is_loaded(Module) of
        {file,File} ->
```

```
            % the module is loaded but not the function
            io:format("error undefined function:~w ~w ~w",
                  [Module, Func, Args]),
            exit({undefined_function,{Module,Func,Args}});
        false ->
            case code:load_file(Module) of
                  {module, _} ->
                      apply(Module, Func, Args);
                  {error, _} ->
                      io:format("error undefined module:~w",
                      [Module]),
                      exit({undefined_module, Module})
            end
    end.
```

如果模块`Mod`已经被加载，那么将导致一个运行时错误。如果模块尚未加载，那么首先尝试加载该模块，若加载成功，再尝试执行先前调用的函数。

模块`code`了解哪些模块已被加载，同时也负责代码加载。

自动加载

编译过的函数无需再显式地编译或“加载”相关模块即可直接用于后续的会话。模块中的导出函数被第一次调用时，该模块将（通过上述的机制）被自动加载。

要实现自动加载，必须满足两个条件：首先，包含Erlang模块的源码文件必须与模块同名（扩展名必须为`.erl`）；其次，系统使用的默认搜索路径必须能定位到该未知模块。

向未注册名称发送消息

尝试向一个不存在的注册进程发送消息时会触发`error_handler:unregistered_name(Name,Pid,Message)`调用。其中`Name`是不存在的注册进程的名称，`Pid`是发送消息的进程标识，`Message`是发送给注册进程的消息。

自定义缺省行为

执行BIF `process_flag(error_handler, MyMod)`可以用模块`MyMod`替换默认的`error_handler`。这使得用户得以定义他们（私有）的错误处理器，用以处理针对未定义函数的调用以及以为注册进程名称为目标的消息发送。该功能仅对执行调用的进程自身有效。定义非标准的错误处理器时必须注意：如果你在替换标准错误处理器时犯了什么错误，系统可能会失控！

也可以通过加载一个新版本的`error_handler`模块来更改默认行为。这么做会影响到所有的进程（定义了私有错误处理器的进程出外），因此非常危险。

Catch和退出信号捕获

在`catch`作用域内求值和捕获进程退出信号是两种完全不同的错误处理机制。退出信号的捕获影响的是一个进程从其他进程处收到`EXIT`信号时的动作。`catch`只影响当前进程中由`catch`保护的表达式的求值。

执行程序7.5里的`tt:test()`会创建一个进程，这个进程匹配`N`（它的值是`1`）和`2`。这会失败的，引发信

号`{'EXIT',Pid,badmatch}`被发送到执行`tt:test()`并且正在等待一个信号的进程。如果这个进程没有正在捕获`exits`，它也会非正常终止。

程序 **7.5**

```
-module(tt).
-export([test/0, p/1]).

test() ->
    spawn_link(tt, p,[1]),
    receive
         X ->
              X
    end.

p(N) ->
    N = 2.
```

调用程序7.5中的`tt:test()`将创建一个以`2`对`N`（值为`1`）作匹配的链接进程。这会失败，并导致信号`{'EXIT',Pid,badmatch}`被发送至调用`tt:test()`的进程，该进程正在等待消息。要是这个进程不捕获退出信号，它就会异常退出。

如果我们执行的不是`tt:test()`而是`catch tt:test()`，结果一样：`catch`作用域外的另一个进程会发生匹配失败。在`spawn_link(tt,p,[1])`之前加上`process_flag(trap_exit, true)`，`tt:test()`就会将收到的`{'EXIT',Pid,badmatch}`信号转换为一条消息。

脚注

[1] 这不是bug或未录入文档的功能!
[2] 这个错误可能导致当前shell崩溃。如何避免这个错误是留给读者的练习。

第8章 编写健壮的应用程序

翻译：王飞
校对：连城

第7章讲解了Erlang的错误处理机制。这一章我们来看看怎样使用这些机制来构建健壮、容错的系统。

防范错误数据

回想一下在第??章（程序??.5）中描述的那个用来分析电话号码的服务程序。它的主循环包含了以下代码：

```
server(AnalTable) ->
    receive
        {From, {analyse,Seq}} ->
            Result = lookup(Seq, AnalTable),
            From ! {number_analyser, Result},
            server(AnalTable);
        {From, {add_number, Seq, Key}} ->
            From ! {number_analyser, ack},
            server(insert(Seq, Key, AnalTable))
        end.
```

以上的`Seq`是一个表示电话号码的数字序列，如`[5,2,4,8,9]`。在编写`lookup/2`和`insert/3`这两个函数时，我们应检查`Seq`是否是一个电话拨号按键字符[1]的列表。若不做这个检查，假设`Seq`是一个原子项`hello`，就会导致运行时错误。一个简单些的做法是将`lookup/2`和`insert/3`放在一个`catch`语句的作用域中求值：

```
server(AnalTable) ->
    receive
            {From, {analyse,Seq}} ->
                case catch lookup(Seq, AnalTable) of
                        {'EXIT', _} ->
                            From ! {number_analyser, error};
                        Result ->
                            From ! {number_analyser, Result}
                    end,
                    server(AnalTable);
            {From, {add_number, Seq, Key}} ->
                From ! {number_analyser, ack},
                    case catch insert(Seq, Key, AnalTable) of
                        {'EXIT', _} ->
                            From ! {number_analyser, error},
                                server(AnalTable); % Table not changed
                        NewTable ->
                            server(NewTable)
                    end
        end.
```

注意，借助`catch`我们的号码分析函数可以只处理正常情况，而让Erlang的错误处理机制去处理`badmatch`、`badarg`、`function_clause`等错误。

一般来说，设计服务器时应注意即使面对错误的输入数据，服务器也不会“崩溃”。很多情况下发送给服务器

的数据都来自服务器的访问函数。在上面的例子中，号码分析服务器获悉的客户端进程标识`From`是从访问函数获得的，例如：

```
lookup(Seq) ->
    number_analyser ! {self(), {analyse,Seq}},
    receive
        {number_analyser, Result} ->
            Result
    end.
```

服务器不需要检查`From`是否是一个进程标识。在这个案例中，我们（借助访问函数）来防范意外的错误情况。然而恶意程序仍然可以绕过访问函数，向服务器发送恶意数据致使服务器崩溃：

```
number_analyser ! {55, [1,2,3]}
```

这样一来号码分析器将试图向进程`55`发送分析结果，继而崩溃。

健壮的服务进程

讲解可靠服务进程设计的最好方法就是借助实例。

第??章（程序??.6）给出了一个资源分配器。对于这个分配器，如果一个资源被分配给了进程，而这个进程在释放资源之前终止（无论是出于意外还是正常终止），那么这个资源就无法被收回。这个问题可以通过以下的方法来解决：

- 令服务程序捕捉`EXIT`信号（`process_flag(trap_exit, true)`）。
- 在分配器和申请资源的进程之间建立连接。
- 处理由这些进程发出的`EXIT`信号。

正如图 8.1 所示。

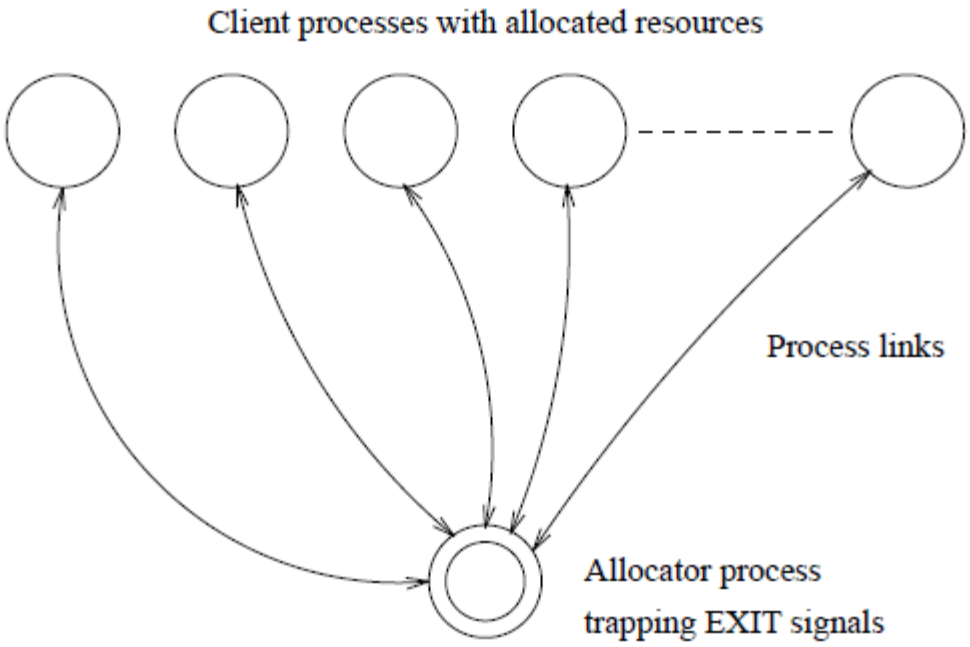

图8.1 健壮的分配器进程和客户进程

分配器的访问函数不变。通过以下方式启动分配器：

```
start_server(Resources) ->
    process_flag(trap_exit, true),
        server(Resources, []).
```

为了接收`EXIT`信号，我们将“服务器”循环改为：

```
server(Free, Allocated) ->
    receive
            {From,alloc} ->
                allocate(Free, Allocated, From);
            {From,{free,R}} ->
                free(Free, Allocated, From, R);
            {'EXIT', From, _ } ->
                check(Free, Allocated, From)
        end.
```

为了跟申请资源（如果还有资源可用）的进程建立连接，还需要修改`allocate/3`。

```
allocate([R|Free], Allocated, From) ->
        link(From),
        From ! {resource_alloc,{yes,R}},
        server(Free, [{R,From}|Allocated]);
allocate([], Allocated, From) ->
        From ! {resource_alloc,no},
        server([], Allocated).
```

`free/4`更复杂些：

```
free(Free, Allocated, From, R) ->
    case lists:member({R, From}, Allocated) of
            true ->
                From ! {resource_alloc, yes},
                    Allocated1 = lists:delete({R, From}, Allocated),
                    case lists:keysearch(From, 2, Allocated1) of
                        false ->
                            unlink(From);
                        _ ->
                            true
                    end,
                    server([R|Free], Allocated1);
            false ->
                From ! {resource_alloc, error},
                server(Free, Allocated)
        end.
```

首先我们检查将要被释放的资源，的确是分配给想要释放资源的这个进程的。如果是的话，`lists:member({R, From}, Allocated)`返回`true`。我们像之前那样建立一个新的链表来存放被分配出去的资源。我们不能只是简单的`unlink From`，而必须首先检查`Form`是否持有其他资源。如果`keysearch(From, 2, Allocated1)`（见附录??）返回了`false`，`From`就没有持有其他资源，这样我们就可以`unlink From`了。

如果一个我们与之建立了link关系的进程终止了，服务程序将会收到一个`EXIT`信号，然后我们调用`Check(Free, Allocated, From)`函数。

```
check(Free, Allocated, From) ->
    case lists:keysearch(From, 2, Allocated) of
        false ->
            server(Free, Allocated);
        {value, {R, From}} ->
            check([R|Free],
            lists:delete({R, From}, Allocated), From)
    end.
```

如果`lists:keysearch(From, 2, Allocated)`返回了`false`，我们就没有给这个进程分配过资源。如果返回了`{value, {R, From}}`，我们就能知道资源`R`被分配给了这个进程，然后我们必须在继续检查该程序是否还持有其他资源之前，将这个资源添加到未分配资源列表，并且将他从已分配资源列表里删除。注意这种情况下我们不需要手动的与该进程解除连接，因为当它终止的时候，连接就已经解除了。

释放一个没有被分配出去的资源是可能一个严重的错误。我们应当修改程序??.6中的`free/1`函数，以便杀死试图这样干的程序：[2]。

```
free(Resource) ->
    resource_alloc ! {self(),{free,Resource}},
        receive
            {resource_alloc, error} ->
                exit(bad_allocation); % exit added here
            {resource_alloc, Reply} ->
                Reply
        end.
```

用这种方法杀死的程序，如果它还持有其他资源，同时还与服务程序保持着连接，那么服务程序因此将收到一个`EXIT`信号，如上面所述，处理这个信号的结果会是资源被释放。

以上内容说明了这么几点：

- 通过设计这样一种服务程序接口，使得客户端通过访问函数（这里是`allocate/0`和`free/1`）访问服务程序，并且防止了危险的“幕后操作”。客户端和服务程序之间的连接对用户来说是透明的。特别是客户端不需要知道服务程序的进程ID，因此也就不能干涉它的运行。
- 一个服务程序如果捕获`EXIT`信号，并且和它的客户端建立连接以便能监视它的话，就可以在客户端进程死亡的时候采取适当的处理行为。

分离计算部分

在一些程序里，我们可能希望将计算部分完全隔离出来，以免影响其它程序。Erlang shell就是这样一个东西。第??章那个简单的shell是有缺陷的。在它里面运行的一个表达式可能通过这几种方式影响到进程：

- 它可以发送进程标示符给其他进程（`self/0`），然后就可以与这个进程建立连接，给它发送消息。
- 它可以注册或注销一个进程

程序8.1用另外一种方法实现了一个shell：

程序**8.1**

```
-module(c_shell).
```

```
-export([start/0, eval/2]).

start() ->
    process_flag(trap_exit, true),
    go().

go() ->
    eval(io:parse_exprs('-> ')),
    go().

eval({form, Exprs}) ->
    Id = spawn_link(c_shell, eval, [self(), Exprs]),
    receive
        {value, Res, _} ->
        io:format("Result: ~w~n", [Res]),
            receive
            {'EXIT', Id, _ } ->
                true
        end;
    {'EXIT', Id, Reason} ->
        io:format("Error: ~w!~n", [Reason])
    end;

eval(_) ->
    io:format("Syntax Error!~n", []).

eval(Id, Exprs) ->
    Id ! eval:exprs(Exprs, []).
```

shell进程捕获`EXIT`信号。命令在一个与shell进程连接的单独的进程（`spawn_link(c_shell, eval, [self(), Exprs])`）中运行。尽管事实上我们把shell进程的进程ID给了`c_shell:eval/2`，但是因为对于作为实际执行者的`eval:exprs/2`函数，并没有给它任何参数，因此也就不会对造成影响。

保持进程存活

一些进程可能对系统来说是非常重要的。例如，在一个常规的分时系统里，常常每一个终端连接都由一个负责输入输出的进程来服务。如果这个进程终止了，终端也就不可用了。程序8.2通过重启终止的进程来保持进程存活。

这个注册为`keep_alive`的服务程序保有一个由`{Id, Mod, Func, Args}`模式元组构成的列表，这个列表包含了所有正在运行的进程的标识符、模块、函数和参数。 它使用BIF `spawn_link/3`启动这些进程，因此它也和每一个进程建立连接。然后这个服务程序就开始捕获`EXIT`信号，当一个进程终止了，它就会收到一个`EXIT`信号。在搜索了那个由元组构成的列表之后，它就能重启这个进程。

不过程序8.2当然也需要改进。如果从进程列表里移除一个进程是不可能的话，那么当我们试图用一个并不存在的`module:function/arity`来创建进程，程序就会进入死循环。建立一个没有这些缺陷的程序，就作为练习留给读者来完成。

讨论

当进程收到了一个“原因”不是`normal`的信号，默认行为是终止自己，并通知与它相连接的进程（见第??节）。通过使用连接和捕捉`EXIT`信号建立一个分层的系统是不难的。在这个系统最顶层的进程（应用进

`EXIT` `EXIT`

程）并不捕获　　信号。具有依赖关系的进程相互连接。底层进程（操作系统进程）捕获　　并且和需要监视的应用进程（见图8.2）建立连接。使用这种操作系统结构的例子是交换机服务器和电话应用程序，将在第??章讲述，第??章是它们的文件系统。

一个因为`EXIT`信号导致异常的应用进程，将会把信号发送给所有跟它处在通一进程集内的进程，因此整个进程集都会被杀死。连接到该进程集内应用程序的操作系统进程也会收到`EXIT`信号，并且会做一些清理工作，也可能重启进程集。

程序 **8.2**

```
loop(Processes) ->
    receive
        {From, {new_proc, Mod, Func, Args}} ->
            Id = spawn_link(Mod, Func, Args),
            From ! {keep_alive, started},
            loop([{Id, Mod, Func, Args}|Processes]);
        {'EXIT', Id, _} ->
            case lists:keysearch(Id, 1, Processes) of
                false ->
                    loop(Processes);
                {value, {Id, Mod, Func, Args}} ->
                    P = lists:delete({Id,Mod,Func,Args},
                    Processes),
                    Id1 = spawn_link(Mod, Func, Args),
                    loop([{Id1, Mod, Func, Args} | P])
            end
    end.

new_process(Mod, Func, Args) ->
    keep_alive ! {self(), {new_proc, Mod, Func, Args}},
    receive
        {keep_alive, started} ->
        true
    end.
```

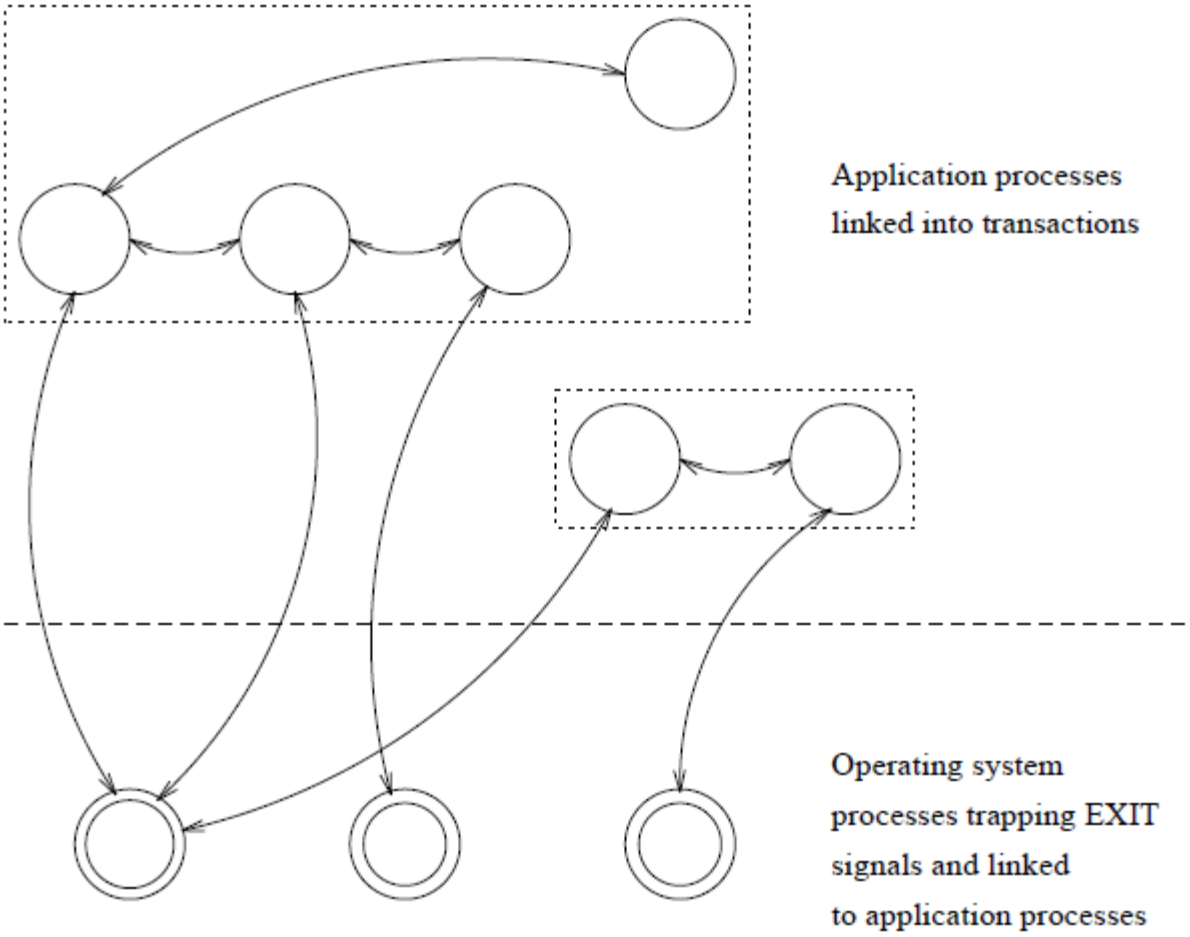

图8.2 操作系统和应用程序进程

脚注

[1] 即数字`0`到`9`和`*`以及`#`。
[2] 这可能是一个好的编程练习，因为它将强制程序的编写者更正这些错误。

第9章 杂项

翻译：连城

本章包含：

- 末尾调用优化——一种令尾递归程序得以在常数空间内执行的优化技术。
- 引用——提供确保在所有节点上都唯一的名称。
- 代码替换——在嵌入式实时系统中必须做到代码的运行时替换，即是说，系统不能停机。
- 端口——提供和外部世界通讯的机制。
- 二进制数据——用于操作无类型内存区域的内建数据类型。
- 进程字典——用于破坏性地存取进程全局数据。[*]
- 网络内核——网络内核用于协调分布式Erlang系统中的所有网络操作。
- 散列——一种将项式映射到唯一的整数用以实现高效的表查询操作的方法。
- 效率——我们将讨论如何编写高效的Erlang程序。

[*] 译者注：此处的“破坏性”指的是进程字典可被修改，从而破坏了Erlang函数式语法的变量不变性。

末尾调用优化

Erlang支持末尾调用优化，从而使得函数得以在固定大小的空间内执行。存储持久数据的主要手法是将之存储于由服务器进程操纵的结构中（典型实例参见第??节）。为了令这种手法得以正常工作，服务器必须利用末尾调用优化。

如果不这么做，服务器最终将会耗尽内存空间从而无法正常工作。

尾递归

我们通过展示同一个函数的两种不同风格的写法来引入尾递归的概念，其中一种写法是尾递归的形式。考察定义如下的`length`函数：

```
length([_|T]) ->
    1 + length(T);
length([]) ->
    0.
```

我们不妨对`legth([a, b, c])`求值。`length`的第一个子句将问题归结为对`1 + length([b, c])`求值。不幸的是，`+`运算无法立即执行，而是得延迟到`length([b, c])`求值完毕为止。系统必须记住这个`+`运算并在后续的某个阶段（此时已知`length([b,c])`的值）系统回溯至该执行这个`+`运算时再实际执行运算。

未决的运算被保存于局部数据区。这块区域的包含至少`K * N`个位置（其中`K`是一个常数，代表对`length`进行一次全新求值所需空间的大小，`N`是未决的运算数量）。

现在我们再写一个等价的求列表长度的函数，其中使用了一个累加器（参见第??节）。该函数仅占用固定大小的空间（为避免混淆，我们将之记为`length1`）：

```
length1(L) ->
    length(L, 0).

length1([_|T], N) ->
    length1(T, 1 + N);
length1([], N) ->
    N.
```

要求`length1([a, b, c])`，我们首先求`length1([a, b, c], 0)`。再归结为`length1([b, c], 1 + 0)`。现在`+`运算可以立即执行了（因为所有参数都已知）。于是，计算`length1([a, b, c])`的函数求值过程为：

```
length1([a, b, c])
length1([a, b, c], 0)
length1([b, c], 1 + 0)
length1([b, c], 1)
length1([c], 1 + 1)
length1([c], 2)
length1([], 1 + 2)
length1([], 3)
3
```

尾递归函数就是在递归调用前不累计任何未决运算的函数。如果函数子句中函数体的最后一个表达式是对自身的调用或者是个常数，那么它就是尾递归子句。如果一个函数的所有子句都是尾递归子句，那么它就是一个尾递归函数。

例如：

```
rev(X) -> rev(X, []).

rev([], X) -> X;
rev([H|T], X) -> rev(T, [H|T]).
```

该函数就是尾递归函数，但：

```
append([], X) -> X;
append([H|T], X) -> [H | append(T,X)].
```

就不是尾递归函数，因为第二个子句的最后一个表达式（`[H | append(T,X)]`中的`|`）既不是对`append`的调用，也不是常数。

末尾调用优化

尾递归是更泛化的末尾调用优化（Last Call Optimisation，LCO）的一个特例。末尾调用优化可应用于任何函数子句最后一个表达式为函数调用的情况。

例如：

```
g(X) ->
    ...
    h(X).

h(X) ->
    ...
    i(X).
```

```
i(X) ->
    g(X).
```

上述代码定义了一组三个相互递归的函数。LCO使得对`g(X)`的求值可以在常数空间内完成。

仔细翻阅本书的所有服务程序示例代码会发现，这些程序都可以在常数空间[1]内执行。

引用

引用是全局唯一的对象。BIF `make_ref()`返回全局唯一的对象，该对象与系统中以及所有其他（可能存在的）运行着的节点中的所有对象都不相等。针对引用的唯一运算就是相等比较。

例如，我们可以在客户端—服务器模型中采用如下的接口函数：

```
request(Server, Req) ->
    Server ! {R = make_ref(), self(), Req},
        receive
            {Server, R, Reply} ->
                Reply
        end.
```

`request(Server, Req)`向名称为`Server`的服务器发送请求`Req`；请求中包含一个唯一引用`R`。在接收服务器返回的应答时会校验是否存在该唯一引用`R`。与服务器端的这种“端对端”的通讯方法可用于确认请求是否已被处理。

代码替换

在嵌入式实时系统中，我们希望在不停机的情况下进行代码升级。比如我们希望在不影响服务的情况下修复某台大型交换机中的软件错误。

在运营过程中进行代码替换是“软”实时控制系统的普遍需求，这些系统往往运营时间很长，代码体积也很大。而在特殊处理器上运行或烧录在ROM里的硬实时系统则往往没有这种需求。

代码替换实例

考察程序9.1。

我们首先编译并加载`code_replace`的代码。然后我们启动程序，并向创建出来的进程发送消息`hello`、`global`和`process`。

程序**9.1**

```
-module(code_replace).
-export([test/0, loop/1]).

test() ->
    register(global, spawn(code_replace, loop, [0])).
```

```
loop(N) ->
    receive
        X ->
            io:format('N = ~w Vsn A received ~w~n', [N, X])
    end,
    code_replace:loop(N+1).
```

最后我们再次编辑程序，将版本号从`A`改为`B`，重新编译、加载程序，并向进程发送消息`hello`。

会话结果如下：

```
%%% start by compiling and loading the code
%%%   (this is done by c:c)
> c:c(code_replace).
...
> code_replace:test().
true
> global ! hello.
N = 0 Vsn A received hello
hello
> global ! global.
N = 1 Vsn A received global
global
> global ! process.
N = 2 Vsn A received process
%%% edit the file code_replace.erl
%%% recompile and load
> c:c(code_replace).
....
> global ! hello.
N = 3 Vsn B received hello
```

这里我们看到，在`loop/1`的执行过程中，虽然我们重新编译、加载了它的代码，但作为`loop/1`的参数的局部变量`N`的值仍被保留了下来。

注意服务器循环的代码是以如下形式编写的：

```
-module(xyz).

loop(Arg1, ..., ArgN) ->
    receive
        ...
    end,
    xyz:loop(NewArg1, ..., NewArgN).
```

这与下面这样的写法有细微的差异：

```
-module(xyz).

loop(Arg1, ..., ArgN) ->
    receive
        ...
    end,
    loop(NewArg1, ..., NewArgN).
```

第一种情况中调用`xyz:loop(...)`意味着总是使用模块`xyz`中最新的`loop`版本。第二种情况中（不显式指定模块名）则只调用当前执行模块中的`loop`版本。

显式使用模块限定名（`module:func`）使得`module:func`动态链接至运行时代码。对于使用完整模块限定名的调用，系统每次都会使用最新版本的可用代码进行函数求值。模块中本地函数的地址解析在编译期完成——它们是静态的，不能在运行时改变。

上述会话示例中`c:c(File)`编译并加载`File`中的代码。在第??节对此有详细讨论。

端口

端口提供了与外部世界通讯的基本机制。用Erlang编写的应用程序往往需要与Erlang系统之外的对象交互。还有一些现存的软件包，例如窗口系统、数据库系统，或是使用C、Modula2等其他语言的程序，在使用它们构建复杂系统时，也往往需要给它们提供Erlang接口。

从程序员的视角来看，我们希望能够以处理普通Erlang程序的方式来处理Erlang系统外的所有活动。为了创造这样的效果，我们需要将Erlang系统外的对象伪装成普通的Erlang进程。端口（Port），一种为Erlang系统和外部世界提供面向字节的通讯信道的抽象设施，就是为此而设计的。

执行`open_port(PortName, PortSettings)`可以创建一个端口，其行为与进程类似。执行`open_port`的进程称为该端口的连接进程。需要发送给端口的消息都应发送至连接进程。外部对象可以通过向与之关联的端口写入字节序列的方式向Erlang系统发送消息，端口将给连接进程发送一条包含该字节序列的消息。

系统中的任意进程都可以与一个端口建立链接，端口和Erlang进程间的`EXIT`信号导致的行为与普通进程的情况完全一致。端口只理解三种消息：

```
Port ! {PidC, {command, Data}}
Port ! {PidC, {connect, Data}}
Port ! {PidC, close}
```

`PidC`必须是一个连接进程的`Pid`。这些消息的含义如下：

`{command, Data}`

> 将`Data`描述的字节序列发送给外部对象。`Data`可以是单个二进制对象，也可以是一个元素为`0..255`范围内的整数的非扁平列表[2]。没有响应。

`close`

> 关闭端口。端口将向连接进程回复一条`{Port, closed}`消息。

`{connect, Pid1}`

> 将端口的连接进程换位`Pid1`。端口将向先前的连接进程发送一条`{Port, connected}`消息。

此外，连接进程还可以通过以下方式接收数据消息：

```
receive
    {Port, {data, Data}} ->
        ... an external object has sent data to Erlang ...
    ...
end
```

在这一节中，我们将描述两个使用端口的程序：第一个是在Erlang工作空间内部的Erlang进程；第二个是在Erlang外部执行的C程序。

打开端口

打开端口时可以进行多种设置。BIF `open_port(PortName, PortSettings)`可用于打开端口。`PortName`可以是：

`{spawn, Command}`

> 启动名为`Command`的外部程序或驱动。Erlang驱动在附录E中有所描述。若没有找到名为`Command`的驱动，则将在Erlang工作空间的外部运行名为`Command`的外部程序。

`Atom`

> `Atom`将被认作是外部资源的名称。这样将在Erlang系统和由该原子式命名的资源之间建立一条透明的连接。连接的行为取决于资源的类型。如果`Atom`表示一个文件，则一条包含文件全部内容的消息会被发送给Erlang系统；向该端口写入发送消息便可向文件写入数据。

`{fd, In, Out}`

> 令Erlang进程得以访问任意由Erlang打开的文件描述符。文件描述符`In`可作为标准输入而`Out`可作为标准输出。该功能很少使用：只有Erlang操作系统的几种服务（`shell`和`user`）需要使用。注意该功能与仅限于UNIX系统。

`PortSettings`是端口设置的列表。有效的设置有：

`{packet, N}`

> 消息的长度将以大端字节序附在消息内容之前的`N`个字节内。`N`的有效取值为`1`、`2`或`4`。

`stream`

> 输出的消息不附带消息长度——Erlang进程和外部对象间必须使用某种私有协议。

`use_stdio`

> 仅对`{spawn, Command}`形式的端口有效。令产生的（UNIX）进程使用标准输入输出（即文件标识符`0`和`1`）与Erlang通讯。

`nouse_stdio`

> 与上述相反。使用文件描述符`3`、`4`与Erlang通讯。

`in`

> 端口仅用于输入。

`out`

> 端口仅用于输出。

`binary`

端口为二进制端口（后续将详述）。

`eof`

到达文件末尾后端口不会关闭并发送`'EXIT'`信号，而是保持打开状态并向端口的连接进程发送一条`{Port, eof}`消息，之后连接进程仍可向端口输出数据。

除了`{spawn, Command}`类型的端口默认使用`use_stdio`外，*所有*类型的端口默认都使用`stream`。

Erlang进程眼中的端口

程序9.2定义了一个简单的Erlang进程，该进程打开一个端口并向该端口发送一串消息。与端口相连的外部对象会处理并回复这些消息。一段时间之后进程将关闭端口。

程序**9.2**

```
-module(demo_server).
-export([start/0]).

start() ->
    Port = open_port({spawn, demo_server}, [{packet, 2}]),
    Port ! {self(), {command, [1,2,3,4,5]}},
    Port ! {self(), {command}, [10,1,2,3,4,5]},
    Port ! {self(), {command, "echo"}},
    Port ! {self(), {command, "abc"}},
    read_replies(Port).

read_replies(Port) ->
    receive
        {Port, Any} ->
            io:format('erlang received from port:~w~n', [Any]),
            read_replies(Port)
    after 2000 ->
            Port ! {self(), close},
            receive
                {Port, closed} ->
                    true
            end
    end.
```

程序9.2中的`open_port(PortName, PortSettings)`启动了一个外部程序。`demo_server`是即将运行的程序的名字。

表达式`Port ! {self(), {command, [1,2,3,4,5]}}`向外部程序发送了五个字节（值为1、2、3、4、5）。

为了让事情有意思一点，我们令外部程序具备一下功能：

- 若程序收到字符串“echo”，则它会向Erlang回复“ohce”。
- 若程序收到的数据块的第一个字节是10，则它会将除第一个字节以外的所有字节翻倍后返回。
- 忽略其他数据。

运行该程序后我们得到以下结果：

```
> demo_server:start().
erlang received from port:{data,[10,2,4,6,8,10]}
erlang received from port:{data,[111,104,99,101]}
true
```

外部进程眼中的端口

程序**9.3**

```
/* demo_server.c */
#include <stdio.h>
#include <string.h>

/* Message data are all unsigned bytes */
typedef unsigned char byte;

main(argc, argv)
int argc;
char **argv;
{

    int len;
    int i;
    char *progname;
    byte buf[1000];

    progname = argv[0];         /* Save start name of program */

    fprintf(stderr, "demo_server in C Starting \n");

    while ((len = read_cmd(buf)) > 0){
        if(strncmp(buf, "echo", 4) == 0)
          write_cmd("ohce", 4);
        else if(buf[0] == 10){
          for(i=1; i < len ; i++)
            buf[i] = 2 * buf[i];
          write_cmd(buf, len);
        }
    }
}

/* Read the 2 length bytes (MSB first), then the data. */
read_cmd(buf)
byte *buf;
{
    int len;

    if (read_exact(buf, 2) != 2)
        return(-1);

    len = (buf[0] << 8) | buf[1];
    return read_exact(buf, len);
}

/* Pack the 2 bytes length (MSB first) and send it */
write_cmd(buf, len)
byte *buf;
int len;
{
    byte str[2];
```

```
    put_int16(len, str);
    if (write_exact(str, 2) != 2)
        return(-1);
    return write_exact(buf, len);
}

/* [read|write]_exact are used since they may return
 * BEFORE all bytes have been transmitted
 */
read_exact(buf, len)
byte *buf;
int len;
{
    int i, got = 0;

    do {
        if ((i = read(0, buf+got, len-got)) <= 0)
          return (i);
        got += i;
    } while (got < len);
    return (len);
}

write_exact(buf, len)
byte *buf;
int len;
{
    int i, wrote = 0;

    do {
        if ((i = write(1, buf+wrote, len-wrote)) <= 0)
          return (i);
        wrote += i;
    } while (wrote < len);
    return (len);
}

put_int16(i, s)
byte *s;
{
    *s = (i >> 8) & 0xff;
    s[1] = i & 0xff;
}
```

程序9.3通过表达式`len = read_cmd(buf)`读取发送至Erlang端口的字节序列，并用`write_cmd(buf, len)`将数据发回Erlang。

文件描述符0用于从Erlang读取数据，而文件描述符1用于向Erlang写入数据。各个C函数的功能如下：

`read_cmd(buf)`

从Erlang读取一条命令。

`write_cmd(buf, len)`

向Erlang写入一个长度为`len`的缓冲区。

`read_exact(buf, len)`

读取`len`个字节。

`write_exact(buf, len)`

写入`len`个字节。

`put_int16(i, s)`

将一个16位整数打包为两个字节。

函数`read_cmd`和`write_cmd`假设外部服务和Erlang间的协议由一个指明数据包长度的双字节包头和紧随的数据构成。如图9.1所示。

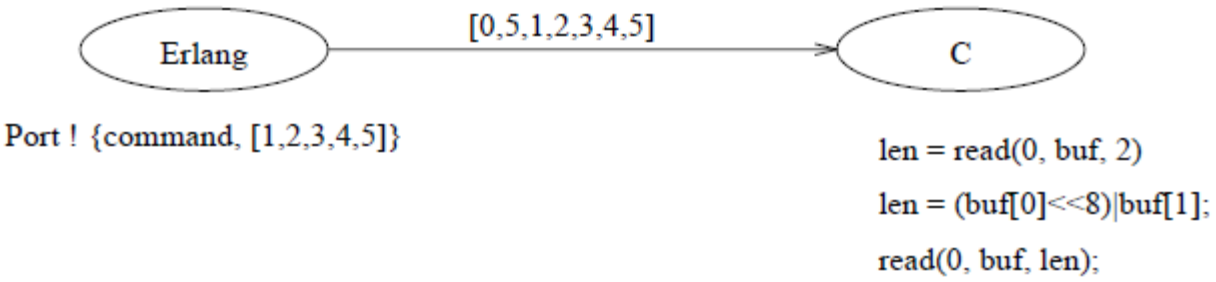

图9.1 端口通讯

之所以使用这种协议（双字节包头加数据）是由于端口是以如下方式打开的：

```
open_port({spawn, demo_server}, [{packet, 2}])
```

二进制类型

二进制类型是一种用于存储无类型内存区域的数据类型。若`open_port/2`的最后一个参数`Settings`列表中包含原子式`binary`，则打开的端口便是二进制端口。来自二进制端口的消息都是二进制类型的数据。

为了说明二进制端口和普通端口的区别，我们用“双字节包头加数据”协议从外部进程向Erlang发送字符串`"hello"`。外部程序将输出如下字节序列：

```
0 5 104 101 108 108 111
```

若与Erlang进程相连的端口是普通端口，则会向向进程发送消息`{Port, {data, [104, 101, 108, 108, 111]}}`。若是二进制端口，消息则是`{Port, {data, Bin}}`，其中`Bin`是长度为5的二进制数据对象，内容即为消息中的字节数据。注意，在这两种情况下，向端口发送数据的外部进程没有区别。

令端口发送二进制对象而非列表的好处在于，相对于长列表，构造和发送二进制数据的速度要快很多。

下列BIF可用于二进制操作：

`term_to_binary(T)`

将项式`T`转为二进制。得到的二进制数据对象为该项式的外部项式格式表示。

`binary_to_term(Bin)`

与`term_to_binary/1`相反。

`binary_to_list(Bin)`

将二进制对象`Bin`转为证书列表。

`binary_to_list(Bin, Start, Stop)`

将二进制对象从`Start`到`Stop`的部分转为整数列表。二进制对象的位置下标从1开始计算。

`list_to_binary(Charlist)`

将`Charlist`转为二进制数据对象。与`term_to_binary(Charlist)`不同，该BIF构造的是一个包含`Charlist`所包含的字节序列的二进制对象，而前者是针对项式`Charlist`构造一个外部项式格式的二进制对象。

`split_binary(Bin, Pos)`

将`Bin`从`Pos`处切分为两个新的二进制对象。得到的是包含两个新二进制对象的元组。例如：

```
1> B = list_to_binary("0123456789").
#Bin
2> size(B).
10
3> {B1,B2} = split_binary(B,3).
{#Bin,#Bin}
4> size(B1).
3
5> size(B2).
7
```

`concat_binary(ListOfBinaries)`

构造一个串接二进制对象列表`ListOfBinaries`中的所有二进制对象的新二进制对象。

另外，保护式`binary(X)`在`X`为二进制数据对象时返回成功。二进制对象主要用于网络中的代码加载，但也可用于那些需要处理大量音视频数据等原始数据的应用。通常可以高效地通过端口输入大量二进制数据，完成数据处理后，再输出到另一个或原先的端口。

进程字典

每个进程都拥有一个字典。通过下列BIF可以操作该字典：

`put(Key, Value)`

将与键`Key`相关联的新值`Value`加入进程字典。若与`Key`相关联的值已经存在则该值将被删除并被新值`Value`替代。该BIF返回原先与`Key`关联的值，若原先没有值与`Key`相关联，则返回`undefined`。`Key`和`Value`可以是任意的Erlang项式。

`get(Key)`

`Key` `Key` `undefined`

返回进程字典中与　关联的值。若没有值与　相关联则返回　　。

`get()`

以`{Key, Value}`元组列表的形式返回整个进程字典。

`get_keys(Value)`

返回一个列表，包含进程字典中值为`Value`的所有的键。

`erase(Key)`

返回整个进程字典后将至删除。

对于各个进程而言进程字典是局部的。进程刚被创建时进程字典为空。任何函数都可通过调用`put(Key, Value)`向字典中添加`{Key, Value}`键值对，而后再通过调用`get(Key)`取出。在`catch`作用域内，若在调用`put`后调用`throw`或出现错误，放入字典的值不会被撤回。

借助`get()`和`erase()`可以获取或删除整个字典。删除单个条目可用`erase(Key)`。

有时候我们希望在多个不同函数中访问同一块全局数据，而将之作为进程中所有函数的参数来进行传递又不太方便。小心使用`put`和`get`就可以避免这个问题。

`get`和`set`在语言中引入了破坏性操作，令程序员写出具有副作用的函数。这些函数的调用结果可能跟它们的调用次序相关。对进程字典的使用应该非常小心。`get`和`set`就好比传统命令式语言里的*goto*。`get`和`set`在某些特定场景下很有用，但使用它们会造成不清晰的代码，应该尽可能地避免使用。鉴于不鼓励使用进程字典，本书的所有程序都不使用进程字典——为了内容完整，只在此处和附录中包含相关内容。

网络内核

`net_kernel`进程被用于协调分布式Erlang系统。运行时系统会自动向`net_kernel`发送某些消息。在该进程中执行的代码决定对于不同的系统消息应该采取何种动作。

Erlang系统可在两种模式下运行。它可以作为一个不与其他Erlang系统通讯的封闭系统运行，也可以同其他系统进行通讯，这时我们认为它存活着。通过调用BIF　`alive/2`可以令系统活过来。通常这是由Erlang操作系统而不是用户完成的。以下调用：

```
erlang:alive(Name, Port)
```

将通知网络命名服务一个Erlang系统已经启动并可以参与分布式计算了。

`Name`是一个用于标识该Erlang系统的本地名称。该Erlang系统的外部名称为`Name@MachineName`，其中`MachineName`是节点所在的机器名，而字符“`@`”用于分隔本地名称与机器名。例如，在名为`super.eua.ericsson.se`的主机上调用`erlang:alive(foo,Port)`将会启动一个名为`foo@super.eua.ericsson.se`的Erlang系统，该名称全局唯一。在同一台机器上可以同时运行多个本地名不同的Erlang系统。

`Port`是一个Erlang端口。外部端口程序必须遵从Erlang分布式系统的内部协议。该程序负责所有的网络操

作，如建立与远程节点间的通讯信道以及向这些节点的字节缓冲区读写数据。不同版本的端口程序允许Erlang节点采用不同的网络技术进行通讯。

执行`alive/2`将使执行该表达式的进程被加入一个可参与分布式计算的Erlang节点池。执行`alive/2`的进程必须以`net_kernel`为名进行注册。否则，该BIF调用会失败。要将一个节点从网路中断开，可以关闭分布式端口。

BIF `is_alive()`可用于检测一个节点是否存活。该BIF返回`true`或`false`。

一旦有新节点出现，`net_kernel`就会收到一条`{nodeup, Node}`消息；一旦有节点失败，`net_kernel`也相应会收到一条`{nodedown, Node}`消息。所有调用`spawn/4`或`spawn_link/4`的进程创建请求以及所有采用`{Name, Node} ! Message`结构向远程注册进程发送消息的请求都会经过`net_kernel`进程。这使得用户可以通过自定义`net_kernel`代码来达成多种目的。例如，BIF `spawn/4`实际上是用Erlang自身实现的。在远程节点创建进程的客户端代码为：

```
spawn(N,M,F,A) when N /= node() ->
    monitor_node(N, true),
    {net_kernel, N} ! {self(), spawn, M, F, A, group_leader()},
    receive
        {nodedown, N} ->
            R = spawn(erlang, crasher, [N,M,F,A,noconnection]);
        {spawn_reply, Pid} ->
            R = Pid
    end,
    monitor_node(N, false),
    R;
spawn(N,M,F,A) ->
    spawn(M,F,A).

crasher(Node,Mod,Fun,Args,Reason) ->
    exit(Reason).
```

这段代码的效果是向远程节点上的`net_kernel`进程发送一条消息。远程的`net_kernel`负责创建新进成，并告知客户端新进程的Pid。

认证

Erlang系统采用“magic cookies”的方式内建了认证支持。Magic cookie是分配给各个节点的一个保密原子式。每个节点在启动时都会被自动分配一个随机cookie。节点`N1`要想和节点`N2`通讯，就必须知道`N2`的magic cookie。这里不讨论`N1`如何找出`N2`的cookie。为了令`N1`得以和`N2`通讯，`N1`必须执行`erlang:set_cookie(N2, N2Cookie)`，其中`N2Cookie`是`N2`的cookie值。另外，要令`N1`能够收到来自`N2`的响应，`N2`也必须执行`erlang:set_cookie(N1, N1Cookie`，其中`N1Cookie`是`N1`的cookie值。

Erlang运行时系统会将cookie插入到发送给所有远程节点的所有消息中。若一条消息抵达某节点时携带着错误的cookie，则运行时系统会将这条消息转换为以下格式：

```
{From,badcookie,To,Message}
```

其中`To`是消息接收方的Pid或注册名而`From`是发送方的Pid。所有未认证的消息发送请求和进程创建请求都会被转为`badcookie`消息并发送至`net_kernel`。`net_kernel`可以任意处置`badcookie`消息。

以下两个BIF可用于cookie操作:

`erlang:get_cookie()`

返回自己的magic cookie。

`erlang:set_cookie(Node,Cookie)`

将节点`Node`的magic cookie设置为`Cookie`。获得`Node`的cookie后可以使用该BIF。它将令后续发送给`Node`的所有消息都包含`Cookie`。如果`Cookie`确实是`Node`的magic cookie，则消息将直接被发送至`Node`上的接收进程。如果包含的cookie有误，该消息将在接收端被转为`badcookie`消息，再被发送至那里的`net_kernel`。

默认情况下，所有节点都假定所有其他节点的cookie是原子式`nocookie`，因此初始时所有的远程消息都包含cookie `nocookie`。

若调用`erlang:set_cookie(Node, Cookie)`时`Node`的值为本地节点的名字，则本地节点的magic cookie将被设置为`Cookie`，同时，其他所有cookie值为`nocookie`的节点都会变为`Cookie`。如果所有节点都在启动时执行:

```
erlang:set_cookie(node(), SecretCookie),
```

则它们将自动互相认证以便协作。应用如何获取到`SecretCookie`是一个实现问题。保密cookie应保存于一个仅能由用户读取或仅能由用户组读取的文件中。

在UNIX环境下，节点启动后的默认行为是读取用户HOME目录下名为`.erlang.cookie`的文件。首先将会对文件的保护权限进行检查，然后便会调用`erlang:set_cookie(node(), Cookie)`，其中`Cookie`是包含cookie文件内容的原子式。之后，同一用户就可以安全地与其他所有在相同用户ID下运行的Erlang节点进行通讯了（假设所有节点都在同一文件系统下运行）。如果节点驻留在不同的文件系统中，用户只须保证涉及到的文件系统中的cookie文件的内容相同即可。

net_kernel消息

以下是可以发送给`net_kernel`的消息的列表:

- `{From,registered_send,To,Mess}` 向注册进程`To`的发送消息`Mess`的请求。
- `{From,spawn,M,F,A,Gleader}` 创建新进程的请求。`Gleader`是请求发起方进程的group leader。
- `{From,spawn_link,M,F,a,Gleader}` 创建新进程并向新进程建立链接的请求。
- `{nodeup,Node}` 当系统中有新节点接入时，`net_kernel`就会收到该消息。这种情况既可能是某远程节点来联络我们，也可能是本地节点上的某个进程向该远程节点首次完成了一次远程操作。
- `{nodedown,Node}` 当某节点失败或从本地节点无法联络到某远程节点时，`net_kernel`就会收到该消息。
- `{From,badcookie,To,Mess}` 当有未认证请求发送到本节点时，`net_kernel`就会收到一条可表征该请求性质的消息。例如，某未认证节点发起了一个进程创建请求，`net_kernel`就会收到消息:

  ```
  {From,badcookie, net_kernel, {From,spawn,M,F,A,Gleader}}
  ```

散列

Erlang提供了一个可从任意项式产生一个整数散列值的BIF：

`hash(Term, MaxInt)`

返回一个在`1..MaxInt`范围内的整数。

借助`hash` BIF我们可以编写一个高效的字典查询程序。该程序的接口与第??节的二叉树实现的字典几乎完全一样。

程序**9.4**

```
-module(tupleStore).
-export([new/0,new/1,lookup/2,add/3,delete/2]).

new() ->
    new(256).

new(NoOfBuckets) ->
    make_tuple(NoOfBuckets, []).

lookup(Key, Tuple) ->
    lookup_in_list(Key, element(hash(Key, size(Tuple)), Tuple)).

add(Key, Value, Tuple) ->
    Index = hash(Key, size(Tuple)),
    Old   = element(Index, Tuple),
    New   = replace(Key, Value, Old, []),
    setelement(Index, Tuple, New).

delete(Key, Tuple) ->
    Index = hash(Key, size(Tuple)),
    Old   = element(Index, Tuple),
    New   = delete(Key, Old, []),
    setelement(Index, Tuple, New).

make_tuple(Length, Default) ->
    make_tuple(Length, Default, []).

make_tuple(0, _, Acc) ->
    list_to_tuple(Acc);
make_tuple(N, Default, Acc) ->
    make_tuple(N-1, Default, [Default|Acc]).

delete(Key, [{Key,_}|T], Acc) ->
    lists:append(T, Acc);
delete(Key, [H|T], Acc) ->
    delete(Key, T, [H|Acc]);
delete(Key, [], Acc) ->
    Acc.

replace(Key, Value, [], Acc) ->
    [{Key,Value}|Acc];
replace(Key, Value, [{Key,_}|T], Acc) ->
    [{Key,Value}|lists:append(T, Acc)];
replace(Key, Value, [H|T], Acc) ->
    replace(Key, Value, T, [H|Acc]).
```

```
lookup_in_list(Key, []) ->
    undefined;
lookup_in_list(Key, [{Key, Value}|_]) ->
    {value, Value};
lookup_in_list(Key, [_|T]) ->
    lookup_in_list(Key, T).
```

该程序与程序??.4的唯一区别就在于函数`new/1`，我们需要向该函数传入散列表的大小。

程序??.4是传统散列查找程序的一个简单实现。散列表`T`由一个定长元组表示。为了查找项式`Key`对应的值，需要计算出一个介于`1..size(T)`之间的散列索引`I`。`element(I, T)`返回一个列表，包含散列索引相同的所有`{Key, Value}`键值对。在该列表中可以搜索到所需的`{Key, Value}`对。

向散列表中插入数据时，首先计算出`Key`的散列索引整数`I`，再向`element(I, T)`返回的列表中插入新的`{Key, Value}`对。原先与`Key`关联的值将被丢弃。

`tupleStore`模块提供了高效的字典。为了提高访问效率散列表的大小必须大于表中所插入的元素的数目。从这种结构中进行查询非常高效，但插入就逊色些。这是因为大部分Erlang视线中BIF `setelement(Index, Val, T)`每次都会创建一个新的元组`T`。

效率

最后我们来讨论一下效率。这并不是说我们认为这个主题不重要，而是因为我们相信过早关注效率问题会导致不良的程序设计。关注重点应该一直放在程序的正确性上，为了达到这个目的，我们提倡开发简练漂亮且“明显”正确的算法。

作为示例，我们将展示如何将低效的程序改造为高效的程序。

作为练习，我们从一个包含某假象公司员工信息元组的文件开始，该文件的内容为：

```
{202191,'Micky','Finn','MNO','OM',2431}.
{102347,'Harvey','Wallbanger','HAR','GHE',2420}.
... 2860 lines omitted ...
{165435,'John','Doe','NKO','GYI', 2564}.
{457634,'John', 'Bull','HMR','KIO', 5436}.
```

我们要写一个程序来输入这些数据、将每个条目都放入字典、访问所有条目一遍，再将数据写回文件。这个程序将频繁执行，因此我们得让它尽可能地快。

文件访问

从上述的元组文件中读入数据的最简单的方法就是使用`file:consult(File)`读取文件（参见附录C）——这个方法很耗时，因为每一行都会被读取和解析。一个好一点的做法是将输入文件从文本格式改为二进制格式。通过以下函数可以实现：

```
reformat(FileOfTerms, BinaryFile) ->
    {ok, Terms} = file:consult(FileOfTerms),
    file:write_file(BinaryFile, term_to_binary(Terms)).
```

要读入二进制文件并恢复原始数据，执行：

```
read_terms(BinaryFile) ->
    {ok, Binary} = file:read(BinaryFile),
    binary_to_term(Binary).
```

读取二进制文件并将结果转换为项式要比读取并解析一组项式要快得多，从下表便中可见一斑：

文本大小**(bytes)**	二进制大小**(bytes)**	`file:consult` **(ms)**	`read_terms` **(ms)**	耗时比例
128041	118123	42733	783	54.6
4541	4190	1433	16	89.6

对于4.5K的文件，二进制文件读取要快90倍；对于128K的文件要快55倍。注意二进制文件要被文本文件小一些。

字典访问

我们使用了不同的方法来构建和更新雇员字典。这些方法包括：

`lists`

所有雇员记录都保存在一个列表中。在表头进行首次插入，其余更新对列表进行线性扫描。

`avl`

采用第??节描述的AVL树插入算法。

`hash`

采用程序9.4的散列算法。

为了检验不同方法的效率，我们对我们的每一条雇员数据都进行一次插入和查找，得到以下的计时结果：

条目数	**AVL**插入	**AVL**查找	列表插入	列表查找	散列插入	散列查找
25	5.32	0.00	0.00	0.64	1.32	0.00
50	1.32	0.32	0.00	1.00	0.32	0.00
100	2.00	0.50	0.00	1.50	0.33	0.16
200	9.91	0.50	0.00	3.00	2.08	0.17
400	28.29	0.46	0.04	5.96	4.25	0.09
800	301.38	0.54	0.02	11.98	1.77	0.15
1600	1060.44	0.61	0.02	24.20	4.05	0.14

上表中每次插入或查询的时间单位都是毫秒。我们看到对于大小超过800的数据表，散列表的查询效率是最高的。

上面我们看到使用二进制文件和散列查询算法要比使用`file:consult`和简单列表查询方法快六千倍。和传统命令式语言一样，决定程序效率的最重要因素还是良好的算法设计。

脚注

[1] 当然，要除去服务器用于存储本地数据结构的空间。

[2] 非扁平列表就是不含有子列表的列表。（译者注：也就是说当`Data`是一个整数列表时，既可以是`[1,2,3]`也可以是`[1,[2,3]]`，在这里二者是等价的。）

附录A Erlang 语法参考

这部分语法参考是 LALR 语法的改编版本。

此语法和严格的 LALR 语法对 `match_expr` 有不同理解。 `match_expr` 中等号左边可以是一个模式或者表达式， Erlang 编译器会在语义分析时确定其含义。

类型	优先级	运算符
Nonassoc	0	`'catch'.`
Right	200	`'='.`
Right	200	`'!'.`
Left	300	`add op.`
Left	400	`mult op.`
Nonassoc	500	`prefix op.`

编号	非终结符	表达式
1	`add op`	`:= "+" | "-" | "bor" | "bxor" | "bsl" | "bsr"`
2	`comp_op`	`:= "==" | "/=" | "=<" | "<" | ">=" | ">" | "=:=" | "=/="`
3	`mult_op`	`:= "*" | "/" | "div" | "rem" | "band"`
4	`prefix_op`	`:= "+" | "-" | "bnot"`
5	`basic_type`	`:= "atom" | "number" | "string" | "var" | "true"`
6	`pattern`	`:= basic_type | pattern_list | pattern_tuple`
7	`pattern_list`	`:= "[" "]" | "[" pattern pattern tail "]"`
8	`pattern_tail`	`:= "|" pattern | "," pattern pattern_tail | ε`
9	`pattern_tuple`	`:= "{" "}" | "{" patterns "}"`
10	`patterns`	`:= pattern | pattern "," patterns`
11	`expr`	`:= basic_type | list | tuple | function_call | expr add op expr | expr mult_op expr | prefix_op expr | "(" expr ")" | "begin" exprs "end" | "catch" expr | case_expr | if_expr | receive_expr | match_expr | send expr`
12	`list`	`:= "[" "]" | "[" expr expr_tail "]"`
13	`expr_tail`	`:= "|" expr | "," expr expr_tail | ε`
14	`tuple`	`:= "{" "}" | "{" exprs "}"`
15	`function_call`	`:= "atom" "(" parameter_list ")" | "atom" ":" "atom" "(" parameter_list ")"`
16	`parameter_list`	`:= exprs | ε`
17	`case_expr`	`:= "case" expr "of" cr_clauses "end"`
18	`cr_clause`	`:= pattern clause_guard clause_body`
19	`cr_clauses`	`:= cr_clause | cr_clause ";" cr_clauses`
20	`if_expr`	`:= "if" if_clauses "end"`
21	`if_clause`	`:= guard clause_body`
22	`if_clauses`	`:= if_clause | if_clause ";" if_clauses`
23	`receive_expr`	`:= "receive" "after" expr clause_body "end" | "receive" cr_clauses "end" | "receive" cr_clauses "after" expr clause_body "end"`
24	`match_expr`	`:= expr "=" expr`
25	`send expr`	`:= expr "!" expr`
	`exprs`	`:= expr | expr "," exprs`

26		
27	guard_expr	:= basic_type | guard_expr_list | guard_expr_tuple | guard_call | "(" guard_expr ")" | guard_expr add_op guard_expr | guard_expr mult_op guard_expr | prefix_op guard_expr
28	guard_expr_list	:= "[" "]" | "[" guard_expr guard_expr_tail "]"
29	guard_expr_tail	:= "|" guard_expr | "," guard_expr guard_expr_tail | ε
30	guard_expr_tuple	:= "{" "}" | "{" guard_exprs "}"
31	guard_exprs	:= guard_expr | guard_expr "," guard_exprs
32	guard_call	:= "atom" "(" guard_parameter_list ")"
33	guard_parameter_list	:= guard_exprs | ε
34	bif_test	:= "atom" "(" guard_parameter_list ")"
35	guard_test	:= bif_test | guard_expr comp_op guard_expr
36	guard_tests	:= guard_test | guard_test "," guard_tests
37	guard	:= "true" | guard_tests
38	function_clause	:= clause_head clause_guard clause_body
39	clause_head	:= "atom" "(" formal parameter_list ")"
40	formal parameter_list	:= patterns | ε
41	clause_guard	:= "when" guard | ε
42	clause_body	:= "->" exprs
43	function	:= function_clause | function_clause ";" function
44	attribute	:= pattern | "[" farity_list "]" | "atom" "," "[" farity_list "]"
45	farity_list	:= farity | farity "," farity_list
46	farity	:= "atom" "/" "number"
47	form	:= "-" "atom" "(" attribute ")" | function

非终结符	编号
add_op	*1 11 27
attribute	*44 47
basic_type	*5 6 11 27
bif_test	*34 35
case_expr	11 *17
clause_body	18 21 23 38 *42
clause_guard	18 38 *41
clause_head	38 *39
comp_op	*2 35
cr_clause	*18 19
cr_clauses	17 *19 19 23
expr	*11 11 12 13 17 23 24 25 26
expr_tail	12 *13 13
exprs	11 14 16 *26 26 42
farity	45 *46
farity_list	44 *45 45
form	*47
formal_parameter_list	39 *40
function	*43 43 47
function_call	11 *15

function_clause	*38 43
guard	21 *37 41
guard_call	27 *32
guard_expr	*27 27 28 29 31 35
guard_expr_list	27 *28
guard_expr_tail	28 *29 29
guard_expr_tuple	27 *30
guard_exprs	30 *31 31 33
guard_parameter_list	32 *33 34
guard_test	*35 36
guard_tests	*36 36 37
if_clause	*21 22
if_clauses	20 *22 22
if_expr	11 *20
list	11 *12
match_expr	11 *24
mult_op	*3 11 27
parameter_list	15 *16
pattern	*6 7 8 10 18 44
pattern_list	6 *7
pattern_tail	7 *8 8
pattern_tuple	6 *9
patterns	9 *10 10 40
prefix_op	*4 11 27
receive_expr	11 *23
send_expr	11 *25
tuple	11 *14

附录B 内置函数

附录C 标准库

翻译：赵卫国

附录C 描述了 Erlang 标准库模块的一些函数。

io

Io 模块提供了基本的输入输出。这儿的所有函数都有可选参数 `Dev` ，它是一个用于输入输出的文件描述符。默认值是标准输入输出。

`format([Dev],F,Args)`	按格式 `F` 输出参数 `Args` 。
`get_chars([Dev],P,N)`	输出提示 `P` 并读出 `Dev` 的前 `N` 个字符。
`get_line([Dev], P)`	输出提示 `P` 并读出 `Dev` 的一行。
`n1([Dev])`	输出新的一行。
`parse_exprs([Dev], P)`	输出提示 `P` 并从 `Dev` 中读出 Erlang 表达。如果成功返回 `{form, ExprList}` , 否则返回 `{error, What}` 。
`parse_form([Dev], P)`	输出提示 `P` ，并把 `Dev` 读成一个 `Erlang` 表。如果成功返回 `{form, Form}` ，否则返回 `{error, What}` 。
`put_chars([Dev], L)`	输出列表 `L` 中的字符。
`read([Dev], P)`	输出提示 `P` 并且从 `Dev` 中读一项式。如果成功则返回 `{term,T}` 否则返回 `{error,What}` 。
`write([Dev],Term)`	输出 `Term` 。

file

`file` 模块提供了与文件系统的标准接口。

`read file(File)“`	返回 `{ok,Bin}` ，其中 `Bin` 是一个包含文件 `File` 内容的二进制数据对象。
`write file(File, Binary)`	把二进制数据对象 `Binary` 中的内容写入到文件 `File` 中。
`get_cwd()`	返回 `{ok,Dir}` ，其中 `Dir` 是当前工作目录。
`set cwd(Dir)`	把当前工作目录设为 `Dir` 。
`rename(From, To)`	把文件名 `From` 改为 `To` 。
`make dir(Dir)`	创建目录 `Dir` 。
`del dir(Dir)`	删除目录 `Dir` 。
`list dir(Dir)`	返回 `{ok,L}` ，其中 `L` 是目录 `Dir` 中的所有文件列表。
`file info(File)`	返回 `{ok,L}` ，其中 `L` 是包含文件 `File` 信息的元组。
`consult(File)`	如果正确返回 `{ok,L}` ，这里的 `L` 是文件 `File` 。
`open(File, Mode)`	打开文件 `File` 的模式 `Mode` 有三种，分别是 `read` 、 `write` 和 `read_write` 。如果成功打开返回 `{ok,File}` ， 失败则返回 `{error,What}` 。
`close(Desc)`	关闭文件 `Desc` 。
`position(Desc, N)`	把文件 `Desc` 的当前位置设为 `N` 。

`truncate(Desc)`	把文件 `Desc` 在当前位置截断。

lists

`list` 模块提供了标准列表进程函数.下面的参数中以 `L` 开头的都代表是列表。

`append(L1, L2)`	返回 `L1+L2` 。
`append(L)`	把 `L` 中所有子列表附加起来的。
`concat(L)`	把列表 `L` 中的所有原子式合并形成一个新的原子。
`delete(X, L)`	返回把 `L` 中第一个出现的 `X` 删除后的列表。
`flat_length(L)`	和 `length(flatten(L))` 等价。
`flatten(L)`	返回对L进行扁平化处理后的列表。
`keydelete(Key, N, LTup)`	返回列表 `LTup` 删除它的第一个元组中第 `N` 个元素是 `Key` 的元组后的列表。
`keysearch(Key, N, LTup)`	遍历元组列表 `LTup` ,查找一个第 `N` 个元素是 `Key` 的元组,若找到返回 `{value, X}` ;否则返回 `false` 。
`keysort(N, LTup)`	返回有 `LTup` 中一系列元组的分类的版本,这其中的第 `N` 个元素用来作关键字。
`member(X, L)`	若 `X` 是列表 `L` 中的成员返回 `true`, 否则返回 `false` 。
`last(L)`	返回 `L` 的最后一个元素。
`nth(N, L)`	返回 `L` 的第 `N` 个元素。
`reverse(L)`	把 `L` 中最上层的元素反转。
`reverse(L1, L2)`	和 `append(reverse(L1), L2)` 等价。
`sort(L)`	对 `L` 进行排序。

code

`code` 模块用于载入或操纵编译过的代码。

`set_path(D)`	把代码服务器查询的路径设为目录 `D` 。
`load_file(File)`	在当前路径上加载文件 `File.erl` 。加载成功返回 `{module, ModuleName }` ; 失败返回: `{error, What}` 。
`is_loaded(Module)`	检验模块 `Module` 是否已经加载.若已加载返回 `{file, AbsFileName}` , 否则返回 `false` 。
`esure_loaded(Module)`	加载之前未加载的模块,它的返回值和 `load_file(File)` 一样。
`purge(Module)`	清楚模块 `Module` 中的代码。
`all_loaded()`	返回所有载入模块的元组 `{Module, AbsFileName}` 。

附录D Erlang的错误处理

翻译：赵卫国

本附录提供了Erlang错误处理机制的细致总结。

匹配错误

当我们调用一个传入错误参数的内建函数时，参数不匹配的函数时，匹配错误就会产生。

当遇到匹配错误时,系统的行为可以描述成以下几种情形:

```
if(called a BIF with bad args)then
    Error = badarg
elseif(cannot and a matching function)then
    Error = badmatch
elseif(no matching case statement)then
    Error = case_clause
    ...
if(within the scope of a “catch”)then
    Value of “catch” = {'EXIT', Error}
else
    broadcast(Error)
    die
endif
```

其中“broadcast(Error)”可以描述为:

```
if(Process has Links)then
    send {'EXIT', self(), Reason} signals to all linked
    processes
endif
```

异常抛出

函数 `throw(Reason)` 的行为可以描述如下:

```
if(within the scope of a “catch”)then
    Value of “catch” = Reason
else
    broadcast(nocatch)
    die
endif
```

退出信号

当接收到 `{'EXIT', Pid, ExitReason}` 信号时， Erlang 的行为可以描述成如下代码:

```
if(ExitReason == kill)then
```

```
    broadcast(killed) % note we change ExitReason
    die
else
    if(trapping exits)then
        add {'EXIT', Pid, ExitReason}
        to input mailbox
    else
        if(ExitReason == normal) then
            continue
        else
            broadcast(ExitReason)
            die
        endif
    endif
endif
```

如果进程表示符为 `Sender` 的进程运行一个简单的函数 `exit(Pid,Why)` ，那么进程 `Pid` 就会收到一个代表进程 `Sender` 好像死亡的消息 `{'EXIT', Source, Why}` 。

如果进程正常终止，把信号 `{'EXIT', Source, normal}` 发送到所有的链接进程.

函数 `exit(Pid, kill)` 产生一个无法销毁的消息，它使的接收进程无条件死亡，把退出的原因改为 `killed` 并把退出的原因发送给所有的链接进程（如若不然,可能使服务器意想不到的崩溃）.

未定义函数

当涉及到未定义函数或注册进程，错误的最后一级就会发生。

如果在调用函数 `Mod:Func(Arg0,...,ArgN)` 但代码中没有这个函数时，就会触发 `error_handler:undefined_function(Mod, Func, [Arg0,...,ArgN])` 。

error_logger

Erlang运行时系统生成的错误消息都转化为下面这种形式:

```
{emulator,GroupLeader,Chars}
```

并把它发送给一个名为 `error_logger` 下的注册进程。由于所有用户代码都可以在 `error_logger` 中运行,因此可以很容易的把错误信息发送到其他结点上处理。这儿的变量 `GroupLeader` 是错误产生进程的进程表示符。有了它， `error_logger` 就可以把错误返回给这个产生错误的进程，以便让连接这个结点的终端打印出错误信息。

附录E 驱动

翻译：连城

本附录描述了如何编写所谓的Erlang内链驱动。任意代码都可以被连入Erlang运行时系统并在某Erlang端口的外端执行。

Erlang进程可以借助端口来收发普通消息。运行时系统与通过端口链入的软件之间通过传递指针来通讯。对于IO非常密集的端口软件来说这样更合适。