

Erlang Programming

Erlang

编程指南



O'REILLY®



机械工业出版社
China Machine Press



Francesco Cesarini & Simon Thompson 著

慕尼黑Isar工作组 杨剑 译

Erlang 编程指南



本书是对Erlang语言的深入介绍。Erlang是任何必须并发、容错和快速响应的环境的理想编程语言。随着多核处理器及其针对并发的新的可扩展方式的发展，Erlang得到了广泛的使用。通过本书，你将学会如何使用Erlang编写复杂的并发程序，不管你是否具有编程背景和经验。

本书由国际知名的Erlang社区领导者根据他们的培训材料编写而成。本书的重点集中在解释Erlang的语法和语义，并且介绍了其模式匹配、规范列表、递归、调试、网络和并发性等内容。

本书帮助你：

- 理解Erlang的强大功能及其包含的特殊功能。
- 学习并发背后的概念以及Erlang处理并发的方式。
- 编写高效的Erlang程序并保持代码整洁和良好的可读性。
- 探究Erlang如何满足分布式系统的要求。
- 轻松添加简单的图形用户界面。
- 学习Erlang的跟踪机制以调试并发和分布式系统。
- 使用内置的Mnesia数据库和其他表存储功能。

本书每章末尾都提供了练习题，并且由简单的示例贯穿全书。

客服热线：(010) 88378991, 88361066

购书热线：(010) 68326294, 88379649, 68995259

投稿热线：(010) 88379604

读者信箱：hzjsj@hzbook.com

华章网站：<http://www.hzbook.com>

网上购书：www.china-pub.com



www.oreilly.com

O'Reilly Media, Inc. 授权机械工业出版社出版

此简体中文版仅限于在中华人民共和国境内（但不允许在中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

“即便我已经使用Erlang多年，在编程的时候仍然需要参考本书。不同层次的Erlang程序员都会发现本书是有价值的学习和参考资料。”

——Steve Vinoski,
《IEEE Internet Computing》
专栏作家

Francesco Cesarini 14年来一直向学生、开发人员、测试人员、项目和技术经理教授Erlang/OTP技术。他协助在爱尔兰、美国和英国建立起了Erlang开发中心。

Simon Thompson是一位肯特大学计算机实验室的逻辑和计算学教授，在过去25年里，他在那里教授本科生和研究生的计算学课程。

ISBN 978-7-111-30325-1



定价：79.00元

Erlang编程指南

Francesco Cesarini & Simon Thompson 著

慕尼黑Isar工作组 杨剑 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc. 授权机械工业出版社出版

机械工业出版社

图书在版编目 (CIP) 数据

O'Reilly精品图书系列

Erlang编程指南/ (美) 塞萨里尼 (Cesarini, F.) 等著; 慕尼黑Isar工作组等译.

—北京: 机械工业出版社, 2011.1

书名原文: Erlang Programming

ISBN 978-7-111-30325-1

I. E... II. ①塞... ②慕... III. 程序语言—程序设计 IV. TP312

中国版本图书馆CIP数据核字 (2010) 第061982号

北京市版权局著作权合同登记

图字: 01-2010-1515号

©2009 by O'Reilly Media, Inc.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and China Machine Press, 2011. Authorized translation of the English edition, 2011 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

本书英文原版由O'Reilly Media, Inc. 出版于2009年。

简体中文版由机械工业出版社出版于2011年。英文原版的翻译得到O'Reilly Media, Inc. 的授权。此简体中文版的出版和销售得到出版权和销售权的所有者——O'Reilly Media, Inc. 的许可。

版权所有, 未得书面许可, 本书的任何部分和全部不得以任何形式重制。

本书法律顾问

北京市展达律师事务所

书 名/ Erlang编程指南

书 号/ ISBN 978-7-111-30325-1

责任编辑/ 秦健

封面设计/ Karen Montgomery, 张健

出版发行/ 机械工业出版社

地 址/ 北京市西城区百万庄大街22号 (邮政编码 100037)

印 刷/ 北京京师印务有限公司

开 本/ 178毫米×233毫米 16开本 28.75印张

版 次/ 2011年3月第1版 2011年3月第1次印刷

定 价/ 79.00元 (册)

凡购本书, 如有倒页、脱页、缺页, 由本社发行部调换

客服热线: (010)88378991; 88361066

购书热线: (010)68326294; 88379649; 68995259

投稿热线: (010)88379604

读者信箱: hzsj@hzbook.com



O'Reilly Media, Inc.介绍

为了满足读者对网络 and 软件技术知识的迫切需求，世界著名计算机图书出版机构 O'Reilly Media, Inc. 授权机械工业出版社，翻译出版一批该公司久负盛名的英文经典技术专著。

O'Reilly Media, Inc. 是世界上在 Unix、X、Internet 和其他开放系统图书领域具有领导地位的出版公司，同时也是联机出版的先锋。

从最畅销的 *The Whole Internet User's Guide & Catalog*（被纽约公共图书馆评为20世纪最重要的50本书之一）到 GNN（最早的Internet门户和商业网站），再到 WebSite（第一个桌面PC的Web服务器软件），O'Reilly Media, Inc. 一直处于Internet发展的最前沿。

许多书店的反馈表明，O'Reilly Media, Inc. 是最稳定的计算机图书出版商——每一本书都一版再版。与大多数计算机图书出版商相比，O'Reilly Media, Inc. 具有深厚的计算机专业背景，这使得 O'Reilly Media, Inc. 形成了一个非常不同于其他出版商的出版方针。O'Reilly Media, Inc. 所有的编辑人员以前都是程序员，或者是顶尖级的技术专家。O'Reilly Media, Inc. 还有许多固定的作者群体——他们本身是相关领域的技术专家、咨询专家，而现在编写著作，O'Reilly Media, Inc. 依靠他们及时地推出图书。因为 O'Reilly Media, Inc. 紧密地与计算机业界联系着，所以 O'Reilly Media, Inc. 知道市场上真正需要什么图书。



译者序

伴随着并发多核CPU计算机系统近年来的迅速发展，作为有20多年历史的成熟系统语言，由于Erlang在语言级别上就提供的并发性和分布性等特性，所以它受到了大众越来越多的关注。

由于Web 2.0的爆炸式发展，以及Web 3.0的初步显现，网站已经开始由一个单纯提供信息的平台逐渐发展演变为一个错综复杂的交互软件系统平台。而云计算和NoSQL等新概念的一再升温也预示着并发系统时代的到来。这也对传统的软件开发方式提出新挑战。

很多曾经只有在计算机教科书中提到的概念，比如同步、线程、死锁、抢占和信号量等，现在越来越多地可以在现实系统中找到它们的影子。由于历史及其各种原因，传统编程语言在创建的初期并没有把并发性放到核心位置，它们并非在语言级别原生支持并发和多核系统。所以，用它们实现多核系统相当复杂且容易产生错误。而Erlang一开始就是为高并发性（一开始应用于电信系统中的高并发系统）设计的，所以它的并发特性是与生俱来的，并且在长达20多年的发展中其理论和实践都得到了不断完善。由此可见，Erlang语言能在多核并发系统中脱颖而出，绝对不是偶然。

并发系统中一个很重要的概念是进程通信。传统语言的处理方法是通过共享数据和内存实现的。而Erlang中的进程通信则采用其专有的进程消息模型，从而完全避免锁的使用，这避免了传统语言进程通信中会碰到的很多让人头疼的问题。这一点对于并发系统来讲可以说是具有革命性的意义。

本书由慕尼黑Isar工作组翻译，我们在翻译本书的过程中尽量忠实于原文，力求做到通俗易懂和言简意赅。自从翻译本书以来，我们感觉Erlang系统越来越棒了。由于介绍Erlang的中文资源相当匮乏，所以我们向大家推荐这本经典著作。我们希望本书对那些想了解现代并发系统和Erlang语言的朋友有所帮助。

本书第1~4章由刘姗姗翻译，第5~9章由张军翻译，第10~12章及附录由姜凯翻译，第13~16章由骆古道翻译，第17~20章由程帆翻译。杨剑负责最后全书校稿通读，并做了细致修改和调整工作。由于时间仓促，加之水平有限，翻译过程中难免有不妥或错误之处，敬请广大读者批评指正。

刘姗姗

于德国慕尼黑

目录

| | |
|----------------------|----|
| 序 | 1 |
| 前言 | 3 |
| 第1章 引言 | 9 |
| 为什么我应该使用Erlang | 9 |
| Erlang语言的历史及发展 | 11 |
| Erlang的特性 | 12 |
| Erlang和多核 | 17 |
| 案例研究 | 18 |
| 应该如何使用Erlang | 21 |
| 第2章 Erlang基础 | 22 |
| 整数 | 22 |
| Erlang终端 | 23 |
| 浮点数 | 23 |
| 基元 | 25 |
| 布尔类型 | 27 |
| 元组 | 28 |
| 列表 | 29 |

| | |
|-----------------------------|-----------|
| 项元比较 | 34 |
| 变量 | 36 |
| 复杂数据结构 | 38 |
| 模式匹配 | 40 |
| 函数 | 44 |
| 模块 | 46 |
| 练习 | 50 |
| 第3章 Erlang顺序编程 | 52 |
| 条件评估 | 52 |
| 保护元 | 57 |
| 内置函数 | 60 |
| 递归 | 66 |
| 运行时错误 | 75 |
| 处理错误 | 77 |
| 模块库 | 84 |
| 调试器 | 87 |
| 练习 | 90 |
| 第4章 并发编程 | 95 |
| 创建进程 | 96 |
| 消息传递 | 98 |
| 接收消息 | 100 |
| 注册进程 | 107 |
| 超时 | 110 |
| 性能基准测试 | 111 |
| 进程架构 | 112 |
| 尾递归和内存泄漏 | 113 |
| 面向并发程序设计的个案研究 | 115 |
| 竞争条件、死锁和饥饿进程 | 116 |
| 进程管理器 | 118 |
| 练习 | 119 |

| | |
|-------------------------|------------|
| 第5章 进程设计模式 | 121 |
| 客户端/服务器模型 | 122 |
| 进程模式实例 | 128 |
| 有限状态机 | 130 |
| 事件管理器和句柄 | 134 |
| 练习 | 141 |
| 第6章 进程错误处理机制 | 143 |
| 进程链接和退出信号 | 143 |
| 健壮性系统 | 152 |
| 练习 | 158 |
| 第7章 记录和宏 | 161 |
| 记录 | 162 |
| 宏 | 168 |
| 练习 | 172 |
| 第8章 软件升级 | 175 |
| 升级模块 | 175 |
| 幕后 | 178 |
| 升级过程 | 184 |
| .erlang文件 | 188 |
| 练习 | 188 |
| 第9章 更多数据类型和高级别构造 | 190 |
| 实践中的函数式编程 | 190 |
| Funs和高阶函数 | 191 |
| 列表解析 | 198 |
| 二进制类型和序列化 | 202 |
| 引用 | 211 |
| 练习 | 212 |

| | |
|------------------------------------|------------|
| 第10章 ETS和Dets表 | 214 |
| ETS表 | 214 |
| Dets表 | 228 |
| 移动用户数据库实例 | 231 |
| 练习 | 242 |
| 第11章 Erlang中的分布式编程 | 244 |
| Erlang中的分布式系统 | 244 |
| Erlang中的分布式计算：基础 | 246 |
| epmd进程 | 259 |
| 练习 | 260 |
| 第12章 OTP行为包 | 261 |
| OTP行为包介绍 | 261 |
| 通用服务器 | 264 |
| 监控进程 | 274 |
| 应用 | 278 |
| 版本发行的处理 | 284 |
| 其他行为包和更多阅读资源 | 287 |
| 练习 | 288 |
| 第13章 Mnesia介绍 | 290 |
| 何时使用Mnesia | 290 |
| 配置Mnesia | 292 |
| 事务处理 | 296 |
| 分区网络 | 301 |
| 扩展阅读 | 302 |
| 练习 | 303 |
| 第14章 图形用户界面编程wxErlang | 305 |
| wxWidgets | 305 |
| wxErlang：wxWidgets绑定到Erlang | 306 |
| 第一个实例：MicroBlog | 309 |

| | |
|--|------------|
| MiniBlog实例 | 313 |
| 获取和运行wxErlang | 316 |
| 练习 | 317 |
| 第15章 套接字编程 | 319 |
| 用户数据报协议 | 319 |
| 传输控制协议 | 323 |
| inet模块 | 328 |
| 扩展阅读 | 329 |
| 练习 | 330 |
| 第16章 Erlang与其他编程语言接口 | 332 |
| 交互运作概况 | 332 |
| 与Java交互运作 | 334 |
| C节点 | 339 |
| Unix终端的Erlang调用: erl_call..... | 343 |
| 端口程序 | 343 |
| 通信支持库 | 347 |
| 内联驱动程序和FFI | 349 |
| 练习 | 350 |
| 第17章 跟踪内置函数, dbg跟踪器以及匹配规则 | 351 |
| 引言 | 351 |
| 跟踪内置函数 | 352 |
| 用trace_pattern内置函数跟踪调用 | 358 |
| dbg跟踪器 | 362 |
| 匹配规则: fun语法 | 370 |
| 匹配规则: 螺母和螺栓 | 379 |
| 扩展阅读 | 387 |
| 练习 | 388 |
| 第18章 类型和文档 | 390 |
| Erlang中的类型 | 390 |

| | |
|--------------------------------|------------|
| TypEr: 成功类型和类型推断 | 394 |
| 使用EDoc生成文档 | 397 |
| 练习 | 405 |
| 第19章 EUnit和测试驱动开发 | 406 |
| 测试驱动开发 | 406 |
| EUnit | 407 |
| EUnit的基础架构 | 411 |
| 测试基于状态的系统 | 413 |
| 在Erlang中测试并发程序 | 414 |
| 练习 | 415 |
| 第20章 风格和效率 | 417 |
| 应用和模块 | 417 |
| 进程和并发 | 422 |
| 格式约定 | 425 |
| 编码策略 | 431 |
| 效率 | 433 |
| 最后 | 437 |
| 附录 使用Erlang | 439 |



序

Erlang是我们提供的解决方案，用于解决高并发和分布式“软实时系统”编程中三个方面的问题：

- 能够快速且高效地开发软件。
- 能够应对各种软件错误和硬件故障。
- 能够实现软件的热升级，也就是不停止运行而完成升级。

Erlang刚“发明”的时候主要应用于电信系统。但是，今天大量的应用系统也有这样的需求，并且Erlang已经应用于各种系统，比如各种不同的分布式数据库、财务系统和聊天服务器等。最近由于适合在多核处理器上使用，人们对Erlang的兴趣再次被点燃。当整个世界还在苦苦努力寻找方法以提高使用多核处理器效率的时候，使用Erlang的应用实际上不需要任何改动就可以享受到多核处理器的好处。

一开始Erlang的影响扩展得比较缓慢，也许是因为Erlang一下就大胆引进了函数式编程、轻量级并发性、异步消息传递和独特的故障处理方法等，并且集所有这些功能于一身。显而易见，这就是为什么Java作为一种和C++比较相近的语言，却可以相对容易地被大众接受。然而，因为已经达到了前面提到过的目标，我们认为我们的方法经受住了时间的考验。所以目前Erlang的用户也在迅猛增长。

这本书对Erlang进行了精彩和实用的讲解，并结合了大量逸闻趣事来阐述Erlang语言的发展背景和理念。

我相信，这会是一次愉快的、收获颇丰的阅读。

—— Mike Williams

Director of Traffic and Feature Software
Product Development Unit WCDMA, Ericsson AB
Erlang语言发明人之一

前言

促使我们写这本书的主要原因是出于对分享Erlang的热衷。希望我们的工作能对Erlang的学习者有所帮助，这也是我们对Erlang社区给予的一点儿回报。虽然我们两个是出于不同目的而开始学习Erlang的，但结果都是一样：通过付出比学习其他语言更多的努力而获得了更多的乐趣。最妙的是，它不仅仅是以娱乐为目的的工具，而是每天我们工作时都在使用它！

Francesco: 为什么使用Erlang

那是在1994年，我正在Uppsala大学学习计算机科学，我参加的课程之一是并行程序设计。讲师举着“Erlang并行程序设计”（Prentice Hall）第1版，说道：“阅读它吧。”然后他举起讲义又说道，“还有做练习。”之后Erlang就很少被提及。它很快隐没在线程、内存共享、信号量和死锁的理论中。

该课程的主要练习是设计一个模拟胡萝卜、兔子和狼共同居住的环境。在这个环境中，兔子随机在土地上寻找生长的胡萝卜。如果它们吃了足够多的胡萝卜，就会变胖并且由一只分裂为两只。在这个环境中，狼搜寻兔子，如果狼吃了足够多的兔子，也会变胖并且由一只分裂为两只。如果兔子和狼靠近到了一定的距离，就会彼此发布食物和掠食者的消息。如果一只兔子发现了一片生长胡萝卜的土地，其他兔子也会很快知道这个地方。如果狼一旦发现了兔子，马上就开始追逐它。

最后的结果很有趣。一只古怪的兔子直接跑进了狼群，而其他的兔子跑向了其他方向，有时在发现胡萝卜的地方停下来进食。每一块胡萝卜地、每一只兔子和狼都代表了一个Erlang进程，它们之间通过消息传递进行通信。

这个练习花了我大约40个小时。虽然当时我就很喜欢使用Erlang，并且惊讶于Erlang并发模型的简易性和每个进程对操作系统线程极少的占用，但是当时并没有很重视它。毕竟Erlang也只是我为了得到学位而学习的几门语言之一。我在函数式编程中使用了机器

语言，并且在实时程序设计中使用了ADA。Erlang对于我来说只是众多语言中的一种。在几个月后，当我开始学习面向对象程序设计时，这一观点得到了改观。

在面向对象程序设计课程中，我们需要模拟同一个环境，不过这一次用Eiffel语言——一种面向对象语言，我们的新讲师认为它极其适合此类模拟。尽管我已经解决过这个问题，可以直接重用其中某些算法，但我还是花费了大约120个人-时来解决它。

这使我惊讶地意识到Erlang的声明性语言和并行处理的特性是软件开发的新方向。当时，我并不确定编程语言是否会向Erlang模式转移，但是我相信，不管是哪种编程语言，将来一定会深受Erlang和它的先驱者的影响。于是我拿起电话打给Joe Armstrong——Erlang的创造者之一。一个星期之后，我拜访了爱立信的计算机科学实验室，从此我再没有回头。

Simon：为什么使用Erlang

我是在20世纪80年代早期开始进行函数式编程工作的，而在Erlang问世20年后才知道它的存在。Erlang最吸引我的地方是，它是为解决实际和复杂问题而设计的，并且通过一种优雅且极其有效的方式解决它们。这也正是近几年人们越来越多地把Erlang应用到的各种系统上的原因。

与Java、C++或Haskell语言相比，Erlang还是一种规模较小的语言，这也使得在Erlang上编写工具更具实用价值。这一点，还有我们在工作中所使用的库的质量，帮助在Kent工作的函数式程序设计组更加有效地为Erlang实现Wrangler重构工具。

谁应该阅读本书

我们写这本书的目的是向你介绍如何使用Erlang进行程序设计。我们并不要求你曾经使用过Erlang，或者熟悉其他函数式编程语言。

我们的确希望你已经拥有Java、C、Ruby或其他编程语言的经验，并且会特别指出Erlang与你所使用的其他语言的不同之处。

怎样阅读本书

我们把这本书分成两部分，第一部分需要顺序阅读，第二部分可以并行（或按照你喜欢的顺序）阅读，因为后一部分各章之间互相独立。

前11章包含了Erlang的核心部分：

- 第1章概括介绍了这门语言，其中包含Erlang 对于建立高效、健壮的并发系统的关键特性。我们还介绍了Erlang是如何发展到今天这种形式的，并且指出应用Erlang的成功实例，这些实例也就解释了为什么你也许会在你的项目中使用Erlang。
- 第2章和第3章包含了在Erlang中的顺序编程的基础知识。在这两章中，我们介绍了Erlang程序设计的核心之一——递归，并且还介绍了Erlang中的单一赋值与C和Java等语言中的变量处理的不同。
- 在包含顺序编程的同时，我们也介绍了Erlang的基本数据类型——数字、基元、字符串、列表和元组——并与其他语言的相似类型进行了比较。其他类型的介绍包含在稍后的各章中：记录类型在第7章，函数类型和二进制类型在第9章，ETS表中的大规模的存储在第10章中介绍。
- Erlang的特殊性质出现在第4~6章中，Erlang的并发性也是通过消息传递，引发在各自独立的内存中运行的并行进程之间的通信来实现。
- 系统可能实现代码的“热切换”，即支持在运行系统中的“软件升级”，这是第8章的主题。
- 这一部分的总结包含在第11章的分布式程序设计中。Erlang允许运行在同一个或不同主机上的Erlang运行时系统（或节点）协同工作，并且组成一个分布式系统。

在其余的章节中，我们将介绍互相独立的多个不同主题。这些包含以下几个方面：

- 开放电信平台（Open Telecom Platform, OTP）提供一系列库和设计原理来支持建立健壮和可扩展的Erlang系统，这是第12章的内容。
- Erlang分布式包含一些标准的应用，我们将在第13章介绍Mnesia数据库，在第14章介绍wxErlang图形用户界面程序设计库。
- Erlang分布式提供了把Erlang系统链接在一起的机制。第15章介绍了Erlang如何使用套接字设计跨越Internet的程序，第16章包含Erlang与其他语言（如C、Java、Ruby等）编写的系统进行交互的多种方式。
- 标准Erlang发布中包含许多有用的工具，我们将介绍其中的一部分。第17章深度解释了如何进行Erlang系统跟踪而不降低系统性能，第18章介绍了检验程序正确性和建立Erlang系统文档的工具，第19章介绍了如何通过EUnit来支持单元测试。
- 第20章介绍如何编写优美、可读性强和高效的Erlang程序，以及一些在Erlang社区中积累的经验。

附录包括如何启动Erlang，如何使用Erlang终端，比较流行的Erlang工具和如何查找关于Erlang更多的信息。

每章末尾均附有练习，你可以从以下网站下载本书中的所有代码：

<http://www.erlangprogramming.org>

该网站还包含其他更进一步的相关阅读材料，还有一些主要的Erlang社区的支持网站的链接。

本书是保持和Erlang Release 13 (R13-B) 版本兼容。本书所描述的大部分特性也适用于Erlang早期版本，已知的和近期版本冲突的细节会在网站详细说明。

本书的约定

本书应用了如下一些印刷排版上的约定：

斜体 (*Italic*)

用于文件名、文件扩展名、URL、应用程序名称、强调以及关键词。

等宽字体 (*Constant width*)

用于表明广义上的计算机代码，包括命令、选项、变量、属性、键、请求、函数、方法、数据类型、类、模式、参数、对象、事件、事件处理器、XML和XHTML标签、宏和关键词。

等宽粗体 (*Constant width bold*)

用于表明命令或者其他的用户需要逐字输入的文本内容。

等宽斜体 (*Constant width italic*)

用于表示需要使用用户提供的值或者由上下文决定的值来替代的文本内容。

使用代码示例

本书试图帮助读者在Erlang中编写程序和系统。你可以在自己的程序和文档中使用本书的代码。

你不必联系出版社即可获得使用本书代码的授权，除非你重写本书中大量的代码。比如说，如果你在编写程序时使用本书几个代码片段，那么你不需要获得我们的许可。引用本书的内容和示例代码来回答问题，同样不需要获得我们的许可。

在你的产品文档中包含大量我们的代码时需要获得我们的许可。出售或分发给O'Reilly书籍CD-ROM中的代码需要获得我们的授权。

如果你引用了本书的内容，我们对此表示感谢，但并不要求在引用时包含书的版权归属。这些版权归属通常包括标题、作者、出版社和ISBN，例如“*Erlang Programming*, by Francesco Cesarini and Simon Thompson. Copyright© 2009 Francesco Cesarini and Simon Thompson, 978-0-596-51818-9”。

如果你对是否需要获得示例代码的授权还不清楚，请随时联系我们：permissions@oreilly.com。

如何联系我们

请将对本书的评价和存在的问题通过如下地址告知出版者：

美国：

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

中国：

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

O'Reilly的每一本书都有专属网站，你可以在那找到关于本书的相关信息，包括勘误列表、示例代码以及其他的信息。本书的网站地址是：

<http://www.oreilly.com/catalog/9780596518189/>

或者：

<http://www.erlangprogramming.org>

对于本书的评论和技术性的问题，请发送电子邮件到：

bookquestions@oreilly.com

关于本书的更多信息、会议、资料中心和O'Reilly网络，请访问以下网站：

<http://www.oreilly.com/>

<http://www.oreilly.com.cn/>

致谢

我们感谢所有帮助本书出版的人。首先是Jan “Call Me Henry” Nyström，他帮助我们启动了个项目。

整个O'Reilly Media团队给予了我们无尽的帮助，特别是编辑Mike Loukides，他耐心地指导并鼓励我们，以确保每一章持续进行。在此特别感谢Audrey Doyle的审稿，还有Rachel Monaghan、Marlowe Shaeffer、Lucie Haskins、Sumita Mukherji和所有制作团队的工作人员。

然后是OTP团队，特别是Bjorn Gustavsson、Sverker Eriksson、Dan Gudmundsson、Kenneth Lundin、Håkan Mattsson、Raimo Niskanen和Patrik Nyblom，他们不仅帮助我们确保缺少文档和没有发布的特性跟最新的发行版本保持一致，同时保证其准确并正确。

其他需要特别提及的审阅人包括Thomas Arts、Zvi Avraham、Franc Bozic、Richard Carlsson、Dale Harvey、Oscar Hellström、Steve Kirsch、Charles McKnight、Paul Oliver、Pierre Omidyar、Octavio Orozio、Rex Page、Michal Ptaszek、Corrado Santoro、Steve Vinoski、David Welton、Ulf Wiger和Mike Williams。

即使我们在此并没有具体提及每个人的贡献，但正是由于他们每个人的贡献，才使得这本书更加出色。谢谢他们。

Francesco需要特别感谢Alison对她的支持和耐心。我并没有意识到在我同意写这本书后会遭遇到什么，我想她也没有。我保证在写下一本书之前，她会拥有一个摆脱笔记本电脑和手机的假期。还要感谢每一个Erlang培训和咨询部门的人对我的鼓励，还有Simon，他真是极好的合作者。我们将来一定还会一起合作，因为结果是如此美妙。但是现在——休息！

Simon需要特别感谢Jane、Alice和Rory在这几个繁忙月份里对他的支持和耐心：如果没有他们的鼓励，就没有这本书。同时感谢Francesco让我参加这个项目，与他合作非常愉快。我希望还有机会一起合作，只是别在近期……

我们为什么如此兴奋地向你介绍Erlang呢？这门语言真正让我们感到特别的地方是什么呢？Erlang的轻量级并发模型是最重要的亮点，该模型独立于底层操作系统，具有实现大规模进程的可扩展性。Erlang利用其避免数据共享的方式，很完美地适用于各种多核处理器。这在解决同步问题和各种性能瓶颈上得到了充分证实，而在其他很多的传统编程语言中，这些问题往往复杂而且很难解决。Erlang语言的声明性质决定了其程序简短紧凑，并且其许多的内置特性使它成为实现容错软实时系统的理想语言。Erlang还拥有非常强大的整合能力，它可以无缝地和其他各种大型系统融合。这意味着将Erlang引入现有系统并逐渐取代常规性能较差的语言，并不是一件不可能的事情。

虽然Erlang面世到现在已有些时日了，但是其语言自身、虚拟机及其类库也随着软件产业需求变化的脚步而快速变化。在一个专业、热情并专注的团队的推动下，在世界各地大学的计算机科学研究人员的帮助下，Erlang在不断地改善并前进着。

什么特点和特性决定了Erlang的成功？Erlang是在怎样的情况下设计的？而这些又如何影响到它现在的样子？本章将一一解答这些问题。另外，在这一章中，我们还将从商业项目、研究项目和开源项目等不同个案研究中，来全面剖析Erlang如何真正应用到实际当中去。同时还将阐述，和其他一些语言相比，使用Erlang的一些独特优势。最后我们还将与你分享实施Erlang的最佳实践。

为什么我应该使用Erlang

什么因素使Erlang成为项目的最佳选择呢？这取决于你所要完成的具体项目。如果你需要开发数值计算系统、图形密集型系统或者运行在手机等客户端上的系统，那么对不起，你可能买错了书。但是，如果你的目标系统是一个高水平、高并发、健壮和按需定制的软实时系统，而且要求在充分利用多核处理器的同时，也可以和其他语言编写的组件集成，那么Erlang是你明智的选择。Sun公司的网络技术首席工程师Tim Bray在2008年

7月OSCON的主题演讲上表示：

如果有人来找我，希望付给我很多钱用于构建一个大型的常年无误运行的信息处理系统，那么我会毫不犹豫地选择Erlang来实现它。

到目前为止，已经有许多公司正在将Erlang用于其真正的产品系统：

- Amazon使用Erlang实现SimpleDB数据库，用于作为亚马逊弹性计算云（EC2）服务的一部分提供数据库服务。
- 雅虎将Erlang用于它的社区书签服务Delicious，它有500多万的用户和1.5亿的书签网址。
- Facebook将Erlang用于其聊天系统服务的后台支持，它可以处理超过一亿数量的活跃用户信息。
- T-Mobile公司将Erlang用于SMS和认证系统（T-Mobile是世界上最大的移动电话公司之一）。
- 摩托罗拉公司在公共安全行业的呼叫处理产品中使用了Erlang。
- 爱立信在其支持节点产品中使用Erlang，这些节点用于世界各地的GPRS和3G移动网络通信。

现在让我们来看看最受欢迎的Erlang开源项目：

- 用于模型和纹理的多边形网格处理的三维建模软件Wings 3D。
- Ejabberd系统，它提供了一个基于即时消息（IM）应用服务器的可扩展消息处理现场协议（XMPP）。
- CouchDB 是一种“无模式”的面向文档的数据库。它提供跨越多核和多服务器集群的可扩展性。
- MochiWeb的类库提供了支持构建轻量级的HTTP服务器的功能。它用来支持强大的服务，如MochiBot和MochiAds，它们以动态方式生成的内容每天向数以百万计的用户提供服务。
- RabbitMQ，一个AMQP消息协议的实现。AMQP是一个新兴的高性能的企业消息传输标准。

虽然乌普萨拉大学多年来一直带头致力于Erlang高性能项目（HiPE）的研究，但是世界上许多其他的大学也并不落后。其中包括英国的肯特大学和匈牙利的Eötvös Loránd大学，他们都致力于研究Erlang软件的重构工具。西班牙的马德里理工大学与瑞典的查默斯大学和IT大学也正在一起研究开发基于Erlang特性的测试工具，而这些工具的问世将在很大程度上改变人们验证Erlang程序的方式。

随着这些公司、开源项目和世界各地的大学的研究开发，Erlang社区已经逐渐成为一个充满活力的蔓延六大洲的国际性社区。与此同时，博客、用户组、邮件列表以及专门的网站现在正帮助Erlang社区提升到下一个新的高度。

在历史上，Erlang最初是用于电信业的服务器端的编程，因此下面让我们追溯到Erlang最早被发明的20世纪80年代来了解Erlang的发展历史。

Erlang语言的历史及发展

在20世纪80年代中期，爱立信的计算机科学实验室接到一个任务：调查适合下一代电信产品的编程语言。在Joe Armstrong、Robert Virding和Mike Williams在Bjarne Däcker的带领下，他们花了两年时间用原型法测试了所有可能的编程语言。最终的结论是，虽然现有语言也有一些有趣的和相关的功能，但是没有一门独立的语言能够包容电信行业所需要的所有的特性。因此，他们决定自己开发一种全新的语言。从此Erlang诞生了，它受到了函数语言（比如ML和Miranda），并发语言（比如ADA、Modula、Chill）以及逻辑编程语言Prolog语言的启发和影响。与爱立信专有语言EriPascal和PLEX一样，Smalltalk语言的软件升级特性在Erlang中也得到了深刻的体现。

基于用Prolog开发的Erlang虚拟机，该实验室花了四年的时间用一种全新的语言为通信应用设计原型。经历无数次的尝试与失败，这种语言成为我们现在所知道的Erlang。1991年，Mike Williams终于写成了以C语言为基础的Erlang虚拟机的最初版本，一年之后，第一个应用Erlang的商业项目也在一个小团队的带领下诞生了。该项目是一个移动服务器，它允许DECT无线电话的用户漫游于私人办公室网络。该产品在1994年成功问世，并随后得到了用户很多关于技术改进和新特性的宝贵反馈，这些建议都在1995年的Erlang新的版本中得以实现。

Erlang是一种成熟的语言得到了证实，目前已经应用于数百个重大开发项目，包括：爱立信的宽带、GPRS和ATM交换解决方案系统。结合和总结这些项目，编程框架（OTP）在1996年问世。OTP提供了Erlang系统的结构化框架和一套实现健壮性和容错性的工具和类库。

了解Erlang的历史，对于理解它的设计理念很重要。虽然许多语言在开发之前并没有找到自己的具体定位，但Erlang是设计来解决一个问题的，那就是开发分布式、容错和大规模并发软实时系统的上市时间问题。实际上许多应用和电信系统有着相似的需求，比如Web服务、零售业和商业银行、计算机电话、消息系统和企业集成。这也就解释了为什么Erlang同样也能在这些领域取得巨大的发展和应用。

1998年12月爱立信做了一个重大决定，在由Mozilla公共许可证衍生的EPL许可证下把

Erlang作为开源代码发布，在此过程中，既没有预算和新闻发布又没有公司营销策划部门的帮助。1999年1月，erlang.org网站总共大约有36 000个网页。10年后，这一数字上升到280万。这个增长也反映了Erlang社区的不断扩大，是成功的商业、研究和开源项目、病毒营销、博客和书籍的组合，而所有这些都源于Erlang最初设计用于解决特定领域难题的需求的推动。

Erlang的特性

虽然Erlang本身是一种很有吸引力的编程语言，但当你把它与虚拟机（VM）、OTP中间件和类库相结合的时候，其真正的实力才能完全体现出来。其中的每一点都使Erlang软件开发变得如此特别。那么，具体有哪些功能体现出Erlang与其他类似语言的不同呢？

高级构造

Erlang是一种声明性的语言。声明性语言工作的原则是去描述应该计算什么，而不是去解释这个值是如何计算而来的。一个函数定义就像一组等式，尤其是当使用模式匹配从不同的情况中去选择和从复杂的数据结构中抽取组件的时候。下面是一个简单的例子：

```
area({square, Side}) -> Side * Side ;
area({circle, Radius}) -> math:pi() * Radius * Radius.
```

这个函数定义包含一个形状参数（这里是一个方形或者圆形），依靠它收到的形状类型，系统匹配正确的函数定义，并返回面积计算结果。

在Erlang中，不仅可以对高层数据进行模式匹配，而且同样可以对二进制序列数据进行匹配。下面是一个解码TCP数据包的函数开始部分：

```
decode(<< SourcePort:16, DestinationPort:16,
        SequenceNumber:32,
        AckNumber:32,
        DataOffset:4, _Reserved:4, Flags:8, WindowSize:16,
        Checksum:16, UrgentPointer:16,
        Payload/binary>>) when DataOffset>4 ...
```

在前面的代码中，每个长度的数字，比如在DataOffset:4中的4，给出了可以与变量相匹配的位数。相比之下，可以设想一下同样的效果如何在C或Java语言中实现呢？

Erlang的另一个特性是，函数（或者闭包）属于第一类数据。可以把它们绑定到一个变量上，也可以像其他数据一样处理它们：在一个列表中存储，以一个函数方式返回，或在不同的进程中传递它们。

Erlang中的列表解析（list comprehension）借鉴了函数式编程范例，它结合了列表生成器（list generator）和过滤器（filter），返回在使用过滤器后由一个列表生成器产生的部分元素的列表。下面是列表解析的一个例子，它只用几行代码就实现了快速排序算法。我们将在第9章中更详细地介绍列表解析。

```
qsort([]) -> [];  
qsort([X|Xs]) ->  
  
    qsort([Y || Y<-Xs, Y =< X]) ++ [X] ++ qsort([Y || Y<-Xs, Y > X]).
```

并发进程和消息传递

并发是Erlang成功的根本。Erlang不提供共享内存的线程，而是每个Erlang进程都在它自己的内存空间里执行，并拥有它自己的堆和栈。进程之间不能随意相互干扰，而这在线程模型中很容易发生，从而极易导致死锁和其他可怕的事情。

进程之间通过消息传递进行相互交流，而这个消息可以是Erlang中的任意数据。消息传递是异步的，因此一旦消息发送，这一进程就能够马上继续执行。消息是有选择性地取自进程信箱，因此没有必要按照消息的到达顺序来处理它们。特别是当进程处理发生在不同的计算机上，并且消息收取的顺序依赖于周围的网络环境的时候，这使并发更为健壮。图1-1给出了一个例子，使用一个“area server”进程为客户计算图形的面积，就像我们在“高级构造”一节中提到的那样。

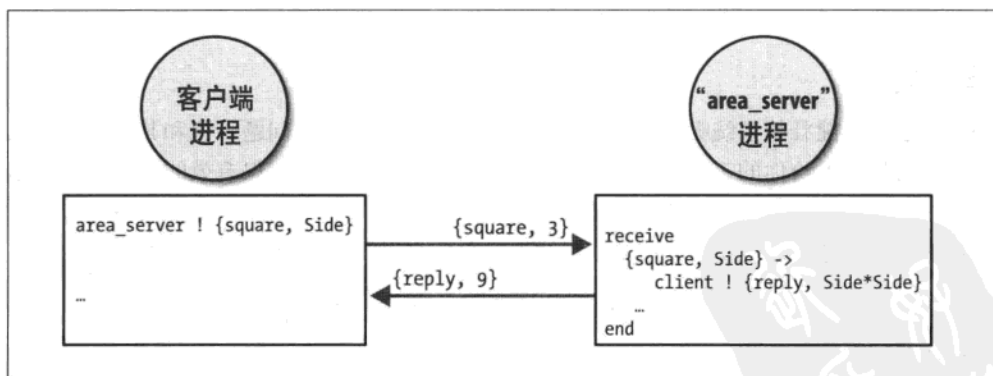


图1-1：进程之间的通信

可扩展、安全和高效的并发

Erlang的并发具有快速和可扩展的特性。它的进程是轻量级的，Erlang虚拟机不会为每一个已生成的进程创建一个操作系统线程。Erlang进程在虚拟机中生成、调度和处理，

而与底层的操作系统无关。因此，进程的生成时间是以微秒为单位的，并且独立于现存的进程的数量。比较而言，Java和C#为每一个进程生成一个底层的操作系统线程：由此你能得出一个有竞争性的结论是，Erlang在这方面以绝对的优势胜过其他两种语言。

Erlang进程相互之间的通信是通过消息传递进行的。不管在你的系统中有多少并发进程，系统的消息交换仅需要几微秒。消息传递的流程是，将数据从一个进程的内存空间复制到另一个的内存空间，这些都发生在同一个虚拟机中。这点不同于Java和C#中常使用的共享内存、信号量和操作系统线程。即便如此，性能测试结果显示Erlang同样超越其他语言，就像在进程生成时的优势一样。

你可能会认为把Erlang与C#和Java比较对这两种语言是不公平的，这就如同我们比较苹果和橘子一样。是的，你是正确的。但我们需要重点指出的是，如果要构建大规模的并发系统，那么你应该为此使用最合适的工具，而不必顾及底层的并发机制。因此，Erlang程序的并发模型与那些创建进程和消息传递所需时间较长的语言有着很大的区别。关于Erlang的并发处理方式我们将在第4~6章和第12章进行详细讲解。

软实时性

尽管Erlang是一种高级语言，但你同样可以利用它的软实时性。Erlang中的存储管理是自动的，垃圾收集的实现是以每个进程为基础。即使存在需要垃圾收集的内存，系统的响应时间也能以毫秒级计算。正因为如此，即使在持续高峰的时候，Erlang也能不降低吞吐量而高负荷运行。

健壮性

如何创建一个健壮的系统呢？虽然Erlang未必能解决你的所有问题，但和其他语言相比它在很大程度上给你的工作提供了便利。Erlang拥有一整套简单但有效的错误处理机制和异常监控机制，并且已经内置了大量通用库模块，其内核加入了健壮性的设计。通过针对正确分支进行编程和由类库来处理错误，程序变得简短易懂，而且错误往往更少。

类库统称为OTP中间件。它们包含了什么异常监控和错误处理机制呢？又有哪些类库建立在其基础上呢？

- Erlang进程可以链接在一起，如果一个进程崩溃，就会通知其他进程，然后（该进程）可以处理这个崩溃，或者选择结束自身。
- OTP提供了一些通用行为包，如服务器、有限状态机和事件句柄。这些工作进程具有内在的健壮性，它们处理所有这些模式的通用（因此很难）并发部分；而用户所需要做的只是对特定服务器的具体行为进行编程，这种编程显然比通用行为包更为直接。

- 这些通用行为包都与监控行为包链接在一起，而监控行为包唯一的任务是监控和处理进程终止。OTP把链接的构想引入框架的设计中，当一个进程监控其他工作进程和监控进程的同时，可能它自身也被其他进程所监控，所有这些都在一个分层结构中。图1-2展示了一个典型的监控树结构。
- 利用这种监控和链接方法，Erlang程序员可以专注于针对正确的情况编程，而在其他任何情况下可以使这一进程出错。避免防错性编程使程序员的工作容易了很多，也能更加直接了解程序的行为。

虽然在本书中我们集中阐述Erlang和它的错误处理机制以及异常监控特性，但同时，我们还会在第12章中介绍OTP设计模式。

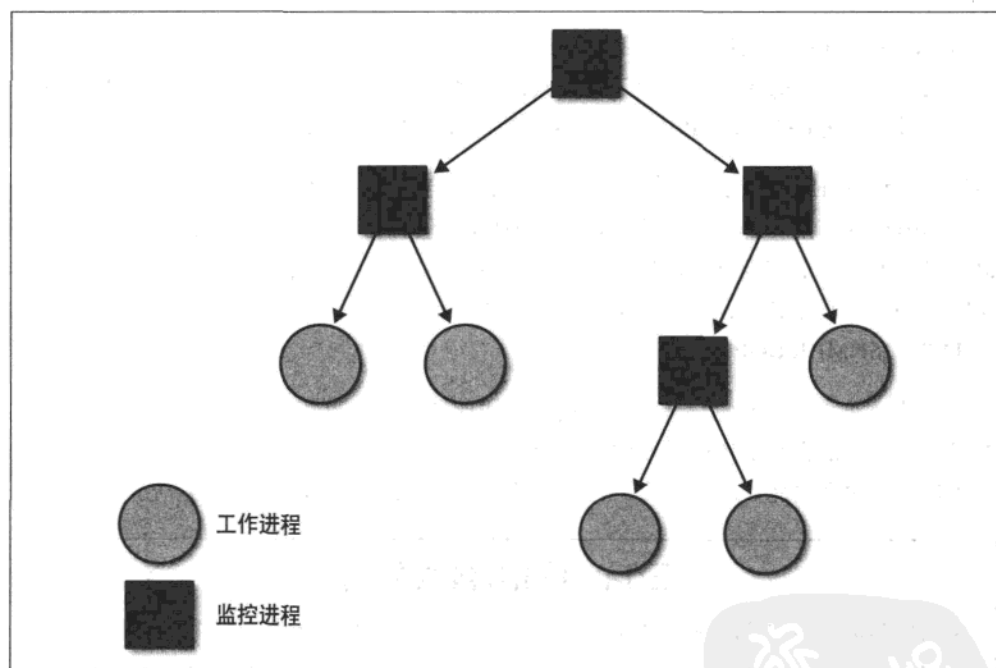


图1-2：一个监控树例子

分布式计算

Erlang已经把分布式纳入了语言的语法和语义中，它允许创建与位置无关的系统。默认的分布模式是基于TCP/IP协议的，它允许一个异构网络上的节点（或Erlang运行时系统）连接到任意一个运行任何操作系统的其他节点上。只要这些节点通过TCP/IP网络连接并且正确配置了防火墙，这样就形成了一个所有节点都能相互通信的全互连网络。

由于当初Erlang是设计在防火墙后面运行的，Erlang安全性是基于cookie (Cookie) 而很少限制访问权限。你可以使用网关 (gateway) 来创建更多不同的Erlang分布式网络节点，如果有必要，可以让它们使用安全的因特网通信协议 (例如SSL) 来通信。

Erlang程序由通过消息传递进行通信的进程所组成。当你开始用Erlang编写程序的时候，这些都在一个节点上，由于在节点内发送消息的语法和发送到远程节点上的语法是相同的，因此你可以轻松地在一组计算机上分布你的进程。由于分布式成为语言的一部分，因此操作 (例如集群、负载平衡、硬件和节点的增加)、通信和可靠性都只需要非常少的开销和相应非常少的代码就能实现。

集成与开放

为不同的工作你应该使用正确的工具。Erlang作为一门开放的语言，允许你保留遗留代码，或放入比Erlang更适合这项工作的其他编程语言的新代码。Erlang有专门的机制来和其他语言 (比如C、Java和Ruby，还包括Python、Perl和Lisp) 进行交互。

高层类库允许Erlang节点与Java或C的执行节点进行交流，这使得它们看上去表现得像分布式的Erlang节点一样。通过使用连接到Erlang运行时系统本身的驱动程序，其他外部语言可以更紧密地捆绑进来，例如一个设备驱动程序就可以如此，而且可以使用套接字让Erlang节点和用其他语言编写的系统进行通信，而它们一般使用比较流行的协议，比如HTTP、SNMP和IIOP。

Erlang的分布式特性使它与其他系统进行集成比其他语言更为自然。处理网络数据格式的能力是一门语言和类库的重要组成部分，这是事前应该考虑的。跟踪和记录能力也让你清楚地了解集成是如何工作的，从而可以让你更加有效地调试和调整系统。

Erlang和函数式编程

最近Erlang的成功也是函数式编程的成功，因为毫无疑问Erlang遵循了函数式编程的原则：它们从一开始就完全以正确的并发语言的基础来设计。

在20世纪80年代，Erlang社区流行的一个神话是：在不久的将来函数式语言将成为能在通用并行计算机上运行的唯一一种语言。当然二十年后的今天，我们看到这些并没有发生，但是现在我们看到这很可能正在进行中，如把Erlang普遍应用于大规模并发的服务器集群、云计算和各种基于多核处理器的台式计算机和笔记本电脑上。

Erlang和多核

向多核的转变是不可避免的。要让C和Java的遗留代码并行化是非常困难的，而调试并行化的C和Java则更难……那么有没有其他更好的选择呢？

Erlang的并发模式——不同进程之间不通过共享内存而是通过消息传递来通信——可以很自然地转换到多核处理器，而且这对于程序员而言大部分都是透明的，这确保了你可以在更强大的硬件上运行Erlang进程，而无须重新设计它们。

Erlang首次关于对称多处理（Symmetric multiprocessing，SMP）实验性的支持的研发始于20世纪90年代末，它现在已经成为标准发布中的一个组成部分。爱立信的Erlang/OTP开发团队的想法是运行SMP，测试其性能，然后找出瓶颈并优化。自从Erlang的第一个SMP功能版本发布之后，这一直是他们所坚持的工作方式。在最近发布版本中，虚拟机模型已经从一个单一的运行队列（进程可能运行在不同的处理器上），发展到在每个处理器上运行一个队列，这确保了运行队列不再是一个系统瓶颈，如图1-3所示。随着越来越多的复杂处理器的出现，运行时系统将与其一起发展。

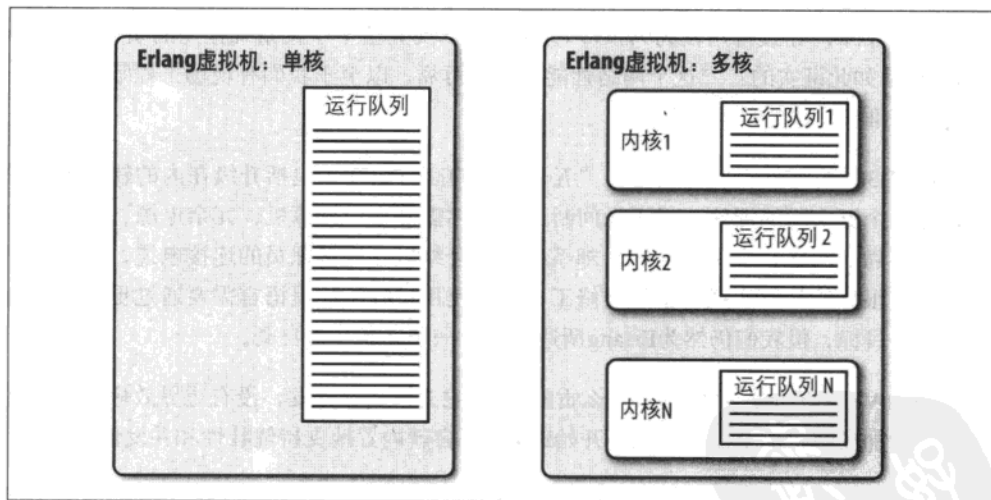


图1-3：一个多核处理器上的运行队列

Erlang SMP的目标是对程序员隐藏关于SMP的具体问题和实现细节。程序员只需要关心如何最优化并发操作，而无须担心隐藏其中的底层操作系统和硬件细节。因此，用Erlang编写的进程是与平台和内核或处理器数目无关的，它们能完全良好地运行在各个系统中。

案例研究

让我们来看一下刚才所描述的Erlang成功的几个因素。爱立信第一个主要的Erlang产品是AXD301 ATM交换机。而最近，Erlang是实现“无模式”面向文档的数据库CouchDB的关键所在。最后，我们通过一个摩托罗拉主导的研究项目来比较Erlang和C++的效率。

AXD301 ATM交换机

AXD301是一个电话级10-160 Gbps的ATM交换机，从设计到实施总共用了不到三年的时间。AXD301的核心包括了超过150万行的Erlang代码，它们负责处理所有复杂的控制逻辑，并监控运行和维护。它还集成了约50万行的C/C++代码来实现低级别的协议和设备驱动器，其中的大部分来自第三方库。

这种ATM交换机已经安装在世界各地的网络中，但最有名的是用在英国电信的一套系统，这也是迄今为止全世界最大的一套“Voice over ATM”的骨干系统。援引爱立信在试验最后阶段发表的新闻稿，“由于在2002年1月削减BT网络的第一节点时发生了一个轻微故障，导致可用性为99.999999%”，爱立信下一代系统程序的负责人Bernt Nilsson是如此证实的：“这个网络性能是如此可靠，以至于我们有现场工程师几乎没有学到任何维护技能的风险。”

根据AXD301的经验我们可以说，“五个九”的高可用性，包括升级在内的软件停机时间，是一个更实际的评估。对于无间断运行，你需要多台计算机、冗余电源、多个网络接口和可靠的网络，永不失效的冷却系统和不会绊倒系统管理员的连接电缆，毫无疑问还需要有很好的具有实践经验的维修工程师。使用传统的编程语言需要通过更多的努力达到这个目标，但我们仍然为Erlang所达到的这一切感到非常自豪。

Erlang为AXD301的成功作出了什么贡献呢？它支持增量开发，没有边界效应，让你更轻松轻松地添加或修改单个部件。从一开始Erlang语言就内置地支持健壮性和并发性。

Erlang很受编程团队的欢迎，因为他们发现它很适合编写更紧凑的代码，从而极大地提高他们的工作效率。根据项目经验，虽然不是科学的记录，Erlang代码4~10倍少于类似的用C/C++、Java或者PLEX（注1）编写的代码，而每千行代码的故障率是一样的。

爱立信已经将Erlang运用在公司的其他项目上，包括SIP电话协议栈、无线基站的控制软

注1： PLEX是一种专有语言，由爱立信开发并广泛应用于AXE-10交换机中。就像Erlang，它的许多特性都是超前的。PLEX从来没有对外发布过。

件、电话网关控制器、媒体网关、宽带解决方案、GPRS和3G的数据传输。而这些仅仅是其中几个我们所谈到的例子而已。

CouchDB

当Damien Katz决定开发CouchDB的时候，他希望能够成为开发“很酷的东西”的一员。他想看看是否自己足够棒，能将某些代码从零开始开发，并把它推到一个新的高度。CouchDB是一个开源数据库，它提供了一个“无模式”的文件存储仓库，以JSON格式存储对象，并通过一个RESTful接口来存取数据。

他写的CouchDB的第一个版本采用的是C++语言。这个系统包括三个部分：存储引擎、查看引擎和查询语言。组件的复杂性渐渐增加，而当开始触及并发问题的时候，他觉得自己处处碰壁。后来他偶然发现了Erlang，下载了它，并迅速意识到它完全可以解决他的那些问题。

从Damien的角度来看，Erlang一开始听起来非常复杂，他也认为，学习Erlang肯定也是非常艰难的。但当他接触到细节的时候，让他惊讶的是这门语言的简单性。与Java相比，用Erlang工作要耗费更多的精力，因为只有较少的工具和可以使用的集成开发环境（IDE），但他知道，要开发能可靠工作的系统，Erlang比使用其他语言需要更少的时间和精力。

Erlang给了Damien开发CouchDB所需的特性，而付出的代价只是使用传统编程语言所需代价的一小部分。当迁移CouchDB至Erlang的时候，他重点致力于并发方面并把它融入到现有的C++组件中。当Erlang具备了他所需要的数据库应用的所有特性的时候，他最终用Erlang取代了整个C++代码库。这包括支持密集的I/O、高可靠性和优雅处理故障的工具。代码的第一个性能测试，即使在它优化之前，已经允许超过20 000个同时连接。而同样情况下如果用C++来实现，只能达到500个。

CouchDB发布以后在开源社区中马上获得了大量的关注。Damien决定通过Apache许可证发布代码，这样他可以继续自由地发展它。今天，CouchDB已经成为当今全球产品级系统中使用的较为出名的Erlang开源软件之一。

Damien后来怎么样了？他在IBM得到一份工作，允许他将CouchDB作为一个开源项目继续发展。用Damien的话来说，他现在的确是为“很酷的东西”而工作的人了。你可以在<http://www.couchdb.org>中阅读更多关于CouchDB的内容。

比较Erlang与C++

最有经验的Erlang程序员证实，他们已经写过的Erlang程序大大短于其他对等竞争的业

界大规模使用的主流编程语言。在Erlang宣布开源之前，这的确在爱立信程序员中广为流传。但直到最近，也几乎没有科学证据来支持这些说法。但使用列表解析的快速排序算法（参见第9章）或者远程过程调用服务器的例子（参见第11章），这两种情况都出现在我们这本书上，且都证实了这种情况。当比较编程语言的时候，无论如何，你必须在这些语言针对的应用领域，对整个系统做性能评测，而不是以代码片段或简单的功能去衡量。

英国的Heriot-Watt大学得到了工程和自然科学研究理事会的赞助，用来研究分布函数式编程语言对电信业的影响。当我们第一次听到这个消息时，我们的反应是，为什么不和爱立信一起去研究呢？当我们意识到这个研究项目是在和爱立信的竞争对手——摩托罗拉实验室一起合作开展的时候，我们的想法就迅速改变了。虽然Heriot-Watt大学可能相信爱立信的话：Erlang适合电信应用编程，可是摩托罗拉可不这么认为。

研究的重点包括两个C++语言为基础的系统，涉及数据移动性（Data Mobility, DM）组成部分和派遣呼叫控制器（Dispatch Call Controller, DCC）。这些系统就像处理紧急呼叫服务一样为袖珍无线电系统处理数字通信流。开发DM时牢记了的容错性和可靠性。它的实现是由专业的C++编程人员在摩托罗拉专有类库的基础上进行开发的。DCC是一个内部研究的原型，它用于评估使用C++和CORBA来获得可扩展性的性能。

Erlang的重写由Jan Henry Nyström执笔，他是一位有着专业学术背景且经验丰富的Erlang程序员。对DM进行了两次Erlang的重写，而对DCC只进行了一次。第一次DM的实现与摩托罗拉的类库进行了连接，而第二次是一次纯粹的Erlang语言的实现。DCC是一个纯Erlang的实现。针对基于语言的性能、稳定性、生产力和编程语言结构的影响，进行了比较。

纯Erlang的实现给这个研究带来了有趣的结果。DM中的代码减少了85%。这一点事实可以证明：27%的C++代码为防错性程序而设计，11%用于内存管理，23%用于高级别的通信，而在Erlang中所有这些特性都是语义的一部分，或者在OTP类库中已经实现了。而DCC的代码只有和其相对应的C++代码的70%左右。

结果Erlang DM与C++的版本相比性能提升了100%，而后者当严重超载的时候就会崩溃。尽管最初的流量令人惊讶，但轻量级并发模型是解决任务的正确工具。与之相关的移动应用包括大量的并发和短消息，而关于复杂处理和数字运算方面则很少。C++一直没有针对它所处理的负载进行设计，因此最后的结论往往是，这些负载的结果可能是不相关。但无论如何，它们展示了Erlang语言作为基础系统的重要特性，那就是在超负荷的情况下保持稳定，并且当工作量下降时能够自动恢复。

尽管Erlang开拓者认为他们简短且紧凑的代码是建立在经验的基础上的，这个研究最后

提供了实例数据证实了这一点。一份题为“高级分布式在快速开发健壮电信软件中的应用：比较C++和Erlang”（注2）的报告充分肯定了“Erlang的优势”。

应该如何使用Erlang

开发Erlang的哲学同样适合于开发以Erlang为基础的系统。引用Erlang的三个发明人之一的Mike Williams的话：

找到设计原型的正确方法。

只有想法还不够，还必须实现它们并且懂得它们如何运行。

允许小范围内犯错，但绝不能发生在产品级项目上。

按照上面所引用的话，所有成功的Erlang项目都应从一个原型开始，而且Erlang完全支持获得原型所需的特性，并运行快速。工作原型通常包括一个功能性的子集，并允许端到端的系统测试。例如，如果你正在建立一个即时通信服务器（在本书中反复出现的一个主题），有效的功能测试包括有能力进行提示和发送消息到远程服务器，而不必担心如冗余、持久性和安全性等问题。

采用Erlang进行软件开发可以通过使用敏捷方法得到最好的实现，并在较短的循环周期内，逐步提供功能不断增加的系统。开发小组应该尽量小规模进行，如果条件允许，应该进行自动测试。本书的“集成与开放”小节所提到的Erlang可用工具，为软件开发提供了极为有利的支持和帮助。计算机辅助测试由EUnit进行单元测试，系统测试通过共同测试完成。其他工具包括cover（用于覆盖分析），还有Dialyzer（透析器），这是一个静态分析工具，用于确定软件的缺陷，如输入错误、死代码和不安全代码。

如果你打算把Erlang引入到你的组织中，那么从小处着手是一个很好的策略。引入小项目（或小系统），从而发挥Erlang的优势。这一策略对于Erlang尤其适合的原因，在于它从设计一开始就内置地支持分布式和集成，就如我们在“集成与开放”小节中所描述的那样，并且所有的产品级Erlang系统都虚拟地和其他语言与系统互联。一旦你在小范围内获得了成功，那么就可以考虑往更大规模发展了。

本书的网站和附录部分介绍了你可以去哪里了解Erlang的链接、协助程序开发的工具以及Erlang团体。但是现在，是我们开始工作的时间了。

注2: Nyström, J.H., P.W. Trinder, D.J. King. 《Concurrency and Computation: Practice & Experience》. 20(8),2008.

第2章

Erlang基础

从本章开始我们学习Erlang语言的基础。你可能会以为在这里我们只会讲到一些其他语言也有的基础知识。但是当你读了这一章以后，不管你以前使用过C/C++、Java、Python或者其他函数式语言，你只会感到惊讶。Erlang有赋值语句，但是它跟其他的命令式语言不同，因为在Erlang里面每个变量只会被赋值一次。Erlang里面的模式匹配不但确定控制流程，而且还用来绑定变量和解析复杂的数据结构。Erlang的模式匹配和其他的函数式语言相比也有着微妙的差别。因此，你需要仔细阅读！在这一章的最后，我们会展示如何在Erlang里定义函数，以及如何把它们放到模块中去创建程序，但是现在我们首先来看看Erlang的基本数据类型。

整数

Integers在Erlang里用来表示整数。它们可以是正整数或者负整数，也可以表示基数不为10的整数。最大整数值的概念在Erlang中不存在，因此在Erlang中可表达和使用任意大的整数。当一个整数大到一个word不能容纳的时候，Erlang内部会自动把它转换成用多个word表示的bignums类型。虽然用bignums类型能精确完成任意大整数的计算，但是相对于固定大小的整数类型它们的效率相对较低。一个整数究竟能有多大，在Erlang中它唯一的限制是实际使用的机器的可用内存。下面是一些整数的例子：

```
-234 0 10 100000000
```

Base#Value符号用于表示基数不是10的整数。Base是一个介于2~16的整数，Value就是基于Base的具体数值。例如2#1010表示整数10的二进制形式，而-16#EA表示整数-234的十六进制形式，因为在十六进制中字母A到F被用来表示10~15。

```
2#1010 -16#EA
```

为了表示字符的ASCII值，采用了\$Character的表示方法。表达式\$Character返回这个

Character的ASCII值。`$a`的值是整数97，而`$A`则代表整数65。换行符`$\n`的ASCII数值则是10：

```
$a $A $\n
```

Erlang终端

通过在Unix终端中输入`erl`来打开Erlang终端窗口，而在Windows中我们可以通过单击“开始”菜单中的Erlang运行图标来打开。你可以在附录中找到更多关于获取和运行Erlang的详细描述。当你得到Erlang的命令行提示符后（形式为`number>`），请尝试输入一些整数，在表达式的最后请别忘记用英文句点（注意：本书统一使用“句点”的翻译，而不是“句号”）（`.`）来终止，然后按回车键完成：

```
1> -234.  
-234  
2> 2#1010.  
10  
3> $A.  
65
```

如果你在输入结束时没有输入句点，那么Erlang终端就不会对你的输入值进行求值，并会继续接收你后续输入的信息，直到你输入句点并按下回车键：

```
4> 5-  
4>  
4> 4.  
1
```

`1>`和`2>`等都是命令行提示符，这表明Erlang已经准备好接受输入。当你按下回车键且结束行也使用了句点，那么Erlang就会对你所输入的内容进行求值，如果成功的话，就会显示出结果。请注意各种不同的整数表达式都转换和显示为十进制形式。如果你输入一个无效的表达式，你会得到一个错误，比如：

```
4> 5-.  
* 1: syntax error before: '.'  
5> q().
```

现在请忽略这些错误信息，它们的具体含义将在第3章中详细讲解。要从一个错误中恢复，只需要多按几次回车键，添加上句点，最后再按下回车键。如果你想退出Erlang终端窗口，那么请输入`q()`并加上句点。

浮点数

在Erlang中，浮点数用来代表实数。下面是一些浮点数的例子：

E-10是一种常规的浮点表示符号，用来表示十进制小数点必须向左移动10个位置。1.234E-10和 1.234×10^{-10} 是一样的，即0.0000000001234。Erlang中的浮点数的精确度是由IEEE 754-1985标准中的64位表示法来保证的。在开始下一节之前，请尝试着在Erlang命令行中输入几个浮点数。

注意：电信应用中的软实时应用系统软件很少依赖浮点数。因此从历史角度来看，高效率的浮点运算对于Erlang的虚拟机（VM）而言优先级很低。当Erlang虚拟机的维护者之一Björn Gustavsson开始从事一个三维图形建模的业余爱好项目，即Wings3D的时候，他对Erlang的浮点数运算效率不太满意。于是在他的努力下，实数运算也开始变得很有效率了。这一点对于任何一个用Erlang语言进行实数运算（比如图像处理）的人来说都是巨大的福音。

数学运算符

整数和浮点数的运算包括加、减、乘和除。正如先前所看到的，+和-可以作为一元运算符以`Op Expression`的格式用在表达式之前，如-12或+12.5。整数的运算结果一定是整数，浮点数除法的情况除外，它的结果是一个浮点数。使用div的结果是一个整数而不会返回余数，余数必须使用rem运算符单独计算得到。表2-1列举了所有的算术运算符。

表2-1：算术运算符

| 类型 | 描述 | 数据类型 |
|-----|---------|----------|
| + | 一元操作符 + | 整数 浮点数 |
| - | 一元操作符 - | 整数 浮点数 |
| * | 乘法 | 整数 浮点数 |
| / | 浮点除法 | 整数 浮点数 |
| div | 整数除法 | 整数 |
| rem | 整数取余 | 整数 |
| + | 加法 | 整数 浮点数 |
| - | 减法 | 整数 浮点数 |

所有的数学运算都是左结合的。在表2-1中，我们按照优先级把它们排列出来了。一元操作符+和-拥有最高的优先级；其次是乘、除和取余数操作；而加法和减法的优先级别最低。

举例来说，对 $-2+3/3$ 求值的顺序是，先对3除以3进行运算，得出浮点数1.0，然后把它和-2相加，最后得出-1.0的结果。在这里可以看到，把一个整数和一个浮点数相加是可以的：这是通过在进行加法运算前，先强制把整数转换成浮点数来进行的。

如果要改变默认的优先级，那就需要使用到圆括号： $(-2 + 3) * 4$ 最后得到4，然而 $-2 + 3 * 4$ 的结果是10，而 $-(2 + 3 * 4)$ 的值是-14。

现在，让我们把Erlang终端当做所谓的计算器来测试这些运算符。注意所得到的结果，特别是当浮点数和整数混合操作和有浮点数除法的时候。请自己尝试不同的计算组合：

```
1> +1.
1
2> -1.
-1
3> 11 div 5.
2
4> 11 rem 5.
1
5> (12 + 3) div 5.
3
6> (12+3)/5.
3.00000
7> 2*2*3.14.
12.5600
8> 1 + 2 + 3 + 5 + 8.
19
9> 2*2 + -3*3.
-5
10> 1/2 + (2/3 + (3/4 + (4/5))) - 1.
1.71667
```

在继续进入下一节之前，请尝试输入`2.0 rem 3`：

```
13> 2.0 rem 3.
** exception error: bad argument in an arithmetic expression
   in operator rem/2
   called as 2.0 rem 3
```

你尝试在一个整数和一个浮点数上执行一次运算，而Erlang运行时系统期望得到两个整数。因此错误发生了，这个错误是很典型的运行时系统抛出的错误。我们将会在第3章中详细讨论这种错误和其他类型的错误。如果你具有C或者Java的编程背景，你可能已经注意到，在进行浮点数除法之前并没有必要将整数转变为浮点数。

基元

在Erlang中用基元（atom）来表示文字常量。Erlang中的基元和其他语言中的枚举类型作用是一样的。对于初学者，有时你可以把它们想象成一个巨大的枚举类型。和其他语言相比，基元的作用相当于C和C++中的`#define`、Java中的“static final”和Ruby中的“enums”。

唯一可用于基元的操作是比较，而且这在Erlang中是以一种非常高效的方法实现的。和整数相比，Erlang中使用基元是为了使代码简洁而高效。和C或C++中宏定义仅仅由预处理器引入的情况不同，基元保留在目标代码中，这使得Erlang中的调试更加容易。

基元由小写字母开始或是由单引号界定。当基元用小写字母开始的时候，字母、数字、“at”符号(@)、英文句点(.)和下划线(_)都是有效的字符。如果一个基元通过单引号封装起来，则可以使用任意字符。下面是一些用小写字母开始的基元例子：

```
january fooBar alfa21 start_with_lower_case node@ramone true false
```

当使用单引号时，例子包括：

```
'January' 'a space' 'Anything inside quotes{}' #@ \n\012'  
'node@ramone.erlang-consulting.com'
```

注意：和语言中的其他一些特性一样，Erlang中基元的概念最初也是受到逻辑程序设计语言Prolog的启发。当然，它们也常出现在函数式编程语言中。

现在请尝试在终端中输入一些基元。如果在某个时候，终端停止了响应，那说明你可能使用了开始单引号，而忘记使用对应的结束单引号。请输入'.后按回车键，这样你就可以返回到终端命令行了。让我们使用一些空格、有趣的字符和大写字母来验证一下。尤其请注意引号是如何显示（和不显示）的，一个表达式是如何以及在何处由句点终结的。

```
1> abc.  
abc  
2> 'abc_123_CDE'.  
abc_123_CDE  
3> 'using spaces'.  
'using spaces'  
4> 'lowercaseQuote'.  
lowercaseQuote  
5> '\n\n'.  
'\n\n'  
6> '1  
6> 2  
6> 3  
6> 4'.  
'1\n2\n3\n4'  
7> 'funny characters in quotes: !"$%^&*()-='.  
'funny characters in quotes: !"$%^&*()-='  
8> '1+2+3'.  
'1+2+3'  
9> 'missing a full stop.'  
9> .  
'missing a full stop.'
```

布尔类型

Erlang中没有单独表示布尔类型的布尔值或者字符。基元true和false与布尔操作符一起使用，而不是布尔类型。它们用来表示测试的布尔返回值，尤其是用于比较操作：

```
1> 1==2.  
false  
2> 1<2.  
true  
3> a>z.  
false  
4> less<more.  
true
```

基元按照字典顺序来排序。本章的后面会更详细地讨论比较操作符。Erlang中有各种各样的内置函数，通常在Erlang社区中称为BIF。它们可以应用在程序和终端中。其中内置函数is_boolean用来测试一个Erlang值是否是布尔类型。

```
5> is_boolean(9+6).  
false  
6> is_boolean(true).  
true
```

复杂的测试可以使用表2-2描述的逻辑运算符来构建。

表2-2：逻辑运算符

| 运算 | 描述 |
|---------|---|
| and | 如果两个参数都是真，那么就返回真 |
| andalso | and的快捷计算形式：如果第一个参数是假，那么就返回假，而不需要计算第二个参数 |
| or | 如果两个参数的任何一个是真，那么就返回真 |
| orelse | or的快捷计算形式：如果第一个参数是真，那么就返回真，而不需要计算第二个参数 |
| xor | “异或”：如果两个参数的任何一个是真，并且另一个是假，那么就返回真 |
| not | 一元否定运算符：如果参数是假，那么就返回真，反之亦然 |

在下面的代码中，逻辑运算符置于参数之前或者放在两个参数之间。

```
1> not((1<3) and (2==2)).  
false  
2> not((1<3) or (2==2)).  
false  
3> not((1<3) xor (2==2)).  
true
```

元组

元组 (tuple) 是用来保存一组数据元素的复合数据类型，其中数据元素要求是Erlang数据类型，但并不一定要是相同的类型。元组使用封闭的花括号{...}来定义，而其中的元素由逗号隔开。下面是一些关于元组的例子：

```
{123, bcd} {123, def, abc} {abc, {def, 123}, ghi} {}  
{person, 'Joe', 'Armstrong'} {person, 'Mike', 'Williams'}
```

元组{123, bcd}包括了两个元素：整数123和基元bcd。元组{abc, {def, 123}, ghi}包括了三个元素，元组{def, 123}是其中的元素之一。空元组{}不包含任何元素。元组可以只包含一个元素，但由于你也可以直接使用“非元组”的元素自身，因此在代码中这样使用通常不是一个好主意。

当一个元组的第一个元素是一个基元时，称它为标记 (tag)。这一Erlang惯例用来表示不同类型的数据，同时通常在使用它的程序中有着特殊的意义。例如，在元组{person, 'Joe', 'Armstrong'}中，基元person是标记，这有可能也就表明第二个元素是这个人的名，然后第三个元素是这个人的姓。

使用第一个位置的标记是为了区分代码中使用不同的元组的不同目的。当错误元组当做一个参数被错误地传递，或者是作为一个函数调用的结果错误地返回的时候，这有助于找到错误的原因。这被认为是Erlang的最佳实践之一。

Erlang提供了一些内置函数用来设置和检索元组元素的内容，并得到元组的大小。

```
1> tuple_size({abc, {def, 123}, ghi}).  
3  
2> element(2,{abc, {def, 123}, ghi}).  
{def,123}  
3> setelement(2,{abc, {def, 123}, ghi},def).  
{abc,def,ghi}  
4> {1,2}<{1,3}.  
true  
5> {2,3}<{2,3}.  
false  
6> {1,2}=={2,3}.  
false
```

从第2条命令中我们可以看出，元组中的元素索引是从1开始而不是从0开始。第3个例子的结果是产生了一个新的元组，其第二个位置被换成了不同的数值def，其他位置上的元素则和之前的元组相同。这些函数都是通用的，它们可以适用于任意类型和任意大小的元组。

在开始讲解列表之前，请确保你在Erlang终端中通过练习很好地掌握了元组和Erlang中的元组内置函数。

列表

列表(List)和元组都是用来存储元素集合的，在这两种情况中，它们的元素都可以是不同的类型，而元素数目也可以是任意的。但在处理方式上列表和元组是非常不同的。在详细解释如何处理列表之前，让我们先来看看Erlang中列表是如何来表示的，并且在Erlang中字符串是如何作为一种特殊的列表来表示的。

列表使用封闭的方括号[...]来定义，而它们的元素由逗号隔开。列表中的元素和元组一样不一定是相同的数据类型，它们可以自由混合。下面是一些关于列表的例子：

```
[january, february, march]
[123,def, abc]
[a,[b,[c,d,e],f],g]
[]
[{person,'Joe','Armstrong'}, {person,'Robert','Virding'},
 {person,'Mike','Williams'}]
[72,101,108,108,111,32,87,111,114, 108,100]
$H,$e,$l,$l,$o,$,$W,$o,$r,$l,$d
"Hello World"
```

列表[a,[b,[c,d,e],f], g]的长度是3。第一个元素是基元a，第二个元素是列表[b,[c,d,e],f]，第三个是基元g。空列表表示为[]，而[{person,'Joe','Armstrong'}, {person,'Robert','Virding'}, {person,'Mike','Williams'}]是由带标记的元组组成的一个列表。

字符和字符串

字符由整数来表示，字符串（由字符组成）则由整数列表来表示。字符的整数值可以通过在字母前加上\$符号来得到：

```
1> $A.
65
2> $A + 32.
97
3> $a.
97
```

Erlang中没有字符串数据类型。在Erlang中，字符串是一个由ASCII值组成的整数列表，并使用双引号("")来表示。因此，字符串"Hello World"实际上就是列表[72,101,108,108,111,32,87,111,114,108,100]。如果你使用ASCII整数符号

\$Character来表示整数，那么你就会得到这样的列表[\$H,\$e,\$l,\$l,\$o,\$, \$w,\$o,\$r,\$l,\$d]。空字符串""则等价于空列表[]：

```
4> [65,66,67].
"ABC"
5> [67,$A+32,$A+51].
"Cat"
6> [72,101,108,108,111,32,87,111,114,108,100].
"Hello World"
7> [$H,$e,$l,$l,$o,$ , $w,$o,$r,$l,$d].
"Hello World"
```

字符串和二进制类型

电信应用程序并不依赖字符串操作，因此字符串从未成为Erlang中单独的数据类型。Erlang中的每一个字符在32位模拟器中占用8个字节（在64位模拟器中占用16个字节），这就造成了字符和字符串在内存中的存储不是很高效。

正如我们将要在第9章中讨论到的，Erlang中包括二进制类型，建议用它们来表示长字符串，特别是当应用程序只是传输它们而不是以任何方式分析它们的时候。最近发布的Erlang在二进制类型的处理速度方面又有所改善，而且看起来这一趋势将会继续下去。

然而，这方面的措施并没有阻止Erlang向字符串密集型应用的转变。我们已经实现了很多每秒处理数以千计的动态网页的网站，和一些内部通过XML API每秒要解析数以千计的SOAP请求的系统。这些系统运行在一些小型的硬件集群上，而这只需要几百美元就可以从eBay上购买到。

随着Erlang在新领域的渗透，目前的实现很有可能会成为一个问题，但迄今为止几乎没有产品系统受到这个实现结果的不良影响。

基元和字符串

基元和字符串的区别在哪里？首先它们处理的方式不同，唯一可用于基元的操作是比较操作，而你可以使用很多不同的方式处理字符串。例如，可以把字符串"Hello World"分解成["Hello","World"]的列表，而基元'Hello World'就不能这样处理。

你可以使用一个字符串来表示一个基元，即把字符串作为一个常量。基元和字符串之间的另一个区别是效率。字符串所占用的空间与字符串的大小成正比，而基元在系统表中表示，它仅仅需要几个字节用作索引，而与它的大小无关。当一个程序比较两个字符串（或者列表）的时候，需要遍历它们并一个一个地比较字符串中的每个字符。而在对基元进行比较的时候，运行时系统只需要一个比较操作来比较它们的内部标识符即可。

列表的组成和处理

正如我们前面所说，列表和元组的处理方式是不同的。元组的处理只能是提取的具体元素，而列表只要不为空，就可以把一个列表分成头部和尾部。列表的头部是指列表的第一个元素，它的尾部是一个包含所有剩余元素的一个列表，而这个列表可以继续这样分解下去，如图2-1所示。

正如列表可以如此这样分割下去一样，我们也可以用一個列表和一个元素组成一个新的列表。新的列表可以这样构造——[头部|尾部]，这是一个典型的构造器（constructor，简写为cons）。

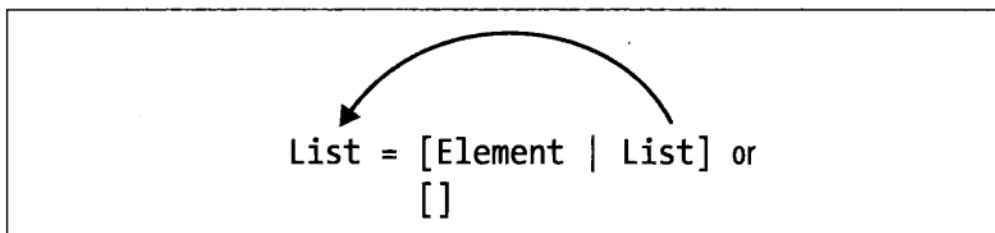


图2-1：列表的递归定义

因此，比如一个列表[1,2,3]，它的头部是1，而尾部是[2,3]。使用这个构造器，这个列表就可以表示成[1|[2,3]]。进一步分割尾部，那么你就可以得到[1|[2|[3]]]和[1|[2|[3|[]]]]。最终得到这个列表的有效表达式是[1,2|[3|[]]]，在采用构造器添加尾部之前，可以有一个以上的由逗号分隔的元素。这些列表都是和原来的列表[1,2,3]等价的。如果最后一个尾部项是一个空列表，那么这就是一个正确（proper）的列表或者结构良好（well-formed）的列表。

Erlang中列表的递归定义可能是我们学习Erlang的时候必须过的第一道坎。为了保险起见，我们再来看一些例子，所有这些列表在语义上都是等价的：

```
[one, two, three, four]
[one, two, three, four|[]]
[one, two|[three, four]]
[one, two|[three|[four|[]]]]
[one|[two|[three|[four|[]]]]]
```

请注意，你必须有一个元素在构造器的左边和一个列表在构造器的右边，它们都由方括号括起来，这样得到的列表就是一个结构良好的列表。

事实上，在Erlang中列表并不一定要是结构良好的，这意味着尾部并不一定要是一个列表。请尝试在终端中输入[[1,2]|3]。结果是什么呢？[1|2]和[1,2|foo]这样的表达式

在语法上是有效的Erlang数据结构，但它们只有有限的值（注1）。结构不良的列表在支持需求驱动或延迟（lazy）编程方面很有用，这些我们将在第9章再详细讨论。除了这些，在Erlang中应该尽量避免使用结构不良的列表。否则检查代码是错误的还是有意这样做的是非常困难的。例如用[2|3]来代替[2|[3]]，它们都是有效的Erlang表达式，在编译时都没有错误。但是当尾部需要一个列表而不是基元的时候，就会抛出一个运行时错误。

列表函数和操作

列表是Erlang中最有用的数据类型之一，特别是结合元组一起使用可以表示各种复杂的数据结构。具体来说，通常用列表来表示对象的集合，它们能够分割成其他的集合，来进一步组合和分析。

关于列表的很多操作都在lists库模块中定义，下面我们可以看到其中的一些例子。这些函数不是内置函数，它们通过在函数前加上模块的名字来调用，并用冒号隔开，例如lists.split。从这些例子可以清楚看出这些函数的作用。在第3章中，我们会看到如何自己定义像这样的函数：

```
1> lists:max([1,2,3]).
3
2> lists:reverse([1,2,3]).
[3,2,1]
3> lists:sort([2,1,3]).
[1,2,3]
4> lists:split(2,[3,4,10,7,9]).
[{3,4},{10,7,9}]
5> lists:sum([3,4,10,7,9]).
33
6> lists:zip([1,2,3],[5,6,7]).
[{1,5},{2,6},{3,7}]
7> lists:delete(2,[1,2,3,2,4,2]).
[1,3,2,4,2]
8> lists:last([1,2,3]).
3
9> lists:member(5,[1,24]).
false
10> lists:member(24,[1,24]).
true
11> lists:nth(2,[3,4,10,7,9]).
4
12> lists:length([1,2,3]).
** exception error: undefined function lists:length/1
13> length([1,2,3]).
3
```

注1： 而不是让以发现Erlang设计缺陷为使命的人们感到兴奋。

这些都不是内置函数，只有length函数除外，你可以从命令行12和命令行13看出这一点。

用于列表的操作符有3个。你已经看到了[...]操作符，还有++和--操作符，它们分别用来把列表相加以及把一个列表从另外一个列表中“减去”。这里举一些例子：

```
1> [monday, tuesday, wednesday].
[monday,tuesday,wednesday]
2>
2> [1|[2|[3|[]]]].
[1,2,3]
3> [a, mixed, "list", {with, 4}, 'data types'].
[a, mixed, "list", {with, 4}, 'data types']
4> [1,2,3] ++ [4,5,6].
[1,2,3,4,5,6]
5> [1,2,2,3,4,4] -- [2,4].
[1,2,3,4]
6> "A long string I have split "
    "across several lines."
"A long string I have split across several lines."
```

++运算符用来把两个列表相连组成了一个新的列表。因此，[1,2] ++ [3,4]将返回[1,2,3,4]。

--运算符用来分别把右边列表中的每个元素从左边的列表中减去。因此，[1,1] -- [1]返回[1]，而[1,2,3,4] -- [1,4]则返回[2,3]。而[1,2] -- [1,1,3]，则会得到列表[2]。这是因为如果右边的列表的元素在左边的列表中找不到，那么它们就会被忽略掉。

++和--操作符的运算顺序是右结合的，因此表达式[1,2,3]--[1,3]--[1,2]：

```
7> [1,2,3]--[1,3]--[1,2].
[1,2]
8> ([1,2,3]--[1,3])--[1,2].
[]
```

最后，请输入"Hello" "Concurrent" "World"，编译器会把它们三个连接起来并返回"Hello Concurrent World"。

如果你想在列表前添加一个元素，那么有两种方法：

- 直接使用构造器，例如[1|[2,3,4]]。
- 使用++运算符，例如[1] ++ [2,3,4]。

这两种方法产生同样的结果，但++运算符效率更低，并可能导致程序运行时速度大幅度减慢。因此当你想添加一个元素到一个列表头部的时候，应该尽量使用高效率的构造器方法([...|...])。

proplists模块中包含了和属性列表（property list）有关的函数。属性列表是有次序的

列表，它的元素可以是带有标记的元组，其中的第一个元素是用于寻找和插入的键值，也可以是基元（比如`blah`），这其实是元组`{blah, true}`的一种简写方式。

为了确保你已经熟练掌握了本节内容，请启动终端来测试你刚刚学到的有关列表和字符串的知识：

- 要特别注意列表生成的形式`[...|...]`，一些读者第一次使用可能有些困难。理解这些在Erlang中是如何运作的很重要，因为我们在第3章中讲到的递归就是依赖于列表的。
- 仔细观察结构不良的列表，因为下一次你可能会再次遇到它们，而且它们可能就是一个错误。
- 相加、相减和字符串连接运算符会使你的代码更加美观，因此一定要花费一些时间去掌握好它们的用法。
- 请记住，如果终端不返回你输入的字符串，那么你可能忘记了结束双引号。请输入`"`，然后按回车键。
- 此外请思考，当我们输入ASCII值的列表时会发生什么？终端会如何显示它们呢？

在本小节的最后，我们来看看Erlang中字符串的发展历史。

在Erlang语言支持字符串连接运算符之前，程序员往往会编写跨越多行的字符串。当代码变得不可读时，他们就把字符串分割成可管理的小块，然后使用`lists`库模块的`append`函数把它们连接起来。

当`++`运算符加入到Erlang语言之后，程序员就从使用`append`函数转变为滥用`++`运算符。其实`++`运算符和`append`函数都是代价很高的操作，因为表达式的左边的列表都要遍历一次。它们不仅仅是耗时的操作，而且常常是多余的，比如在Erlang中所有I/O函数（包括套接字操作）都接受未展开的字符串，比如`["Hello ", ["Concurrent "], ["World"]]`。

项元比较

在Erlang中比较项元需要两个表达式，它们位于比较运算符的左右两边。该表达式的结果是基元`true`或者`false`。等于（`==`）和不等于（`/=`）运算符忽略两边的具体数据类型，而只比较两边的值。比如`1==one`返回`false`，而`1==1`返回`true`。

`1==1.0`将返回`true`，`1/=1.0`将返回`false`，在这种情况下整数在比较前会先被转换成浮点数。如果不想这样，那么你可以使用精确等于和不精确等于运算符，它们不仅比较双方的值，而且也比较它们的数据类型。例如，`1:=1.0`和`1/=1`都将返回`false`，`1/=1.0`则返回`true`。

就像相等性比较一样，你也可以使用<（小于）、<=（小于或等于）、>（大于）和 >=（大于或等于）比较值之间的排序。表2-3列出了各个比较运算符。

表2-3：比较运算符

| 运算符 | 描述 |
|------|-------|
| == | 等于 |
| /= | 不等于 |
| ==:= | 精确等于 |
| ==/= | 精确不等于 |
| =< | 小于或等于 |
| < | 小于 |
| >= | 大于或等于 |
| > | 大于 |

如果表达式是比较不同的数据类型，请参考下面的等级排序：

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

这意味着，任何数字都比任何原子小，而任何元组也都小于任何列表：

```
3> 11<ten.  
true  
4> {123,345}<[].  
true
```

列表按字典顺序排列，就像是字典里面的词一样。第一个元素先比较，小的那个表明这个列表也较小；如果它们是相同的，则第二个元素将继续进行比较，如此下去。当其中一个列表先比较完了，那么它就是较小的列表。因此：

```
5> [boo,hoo]<[adder,zebra,bee].  
false  
6> [boo,hoo]<[boo,hoo,adder,zebra,bee].  
true
```

而另一方面，元组比较时，先会比较结构中的元素数目，然后再一个个比较各个元素的值：

```
7> {boo,hoo}<{adder,zebra,bee}.  
true  
8> {boo,hoo}<{boo,hoo,adder,zebra,bee}.  
true
```

能够对不同数据类型进行比较的特性，使得我们可以写出像sort这样的通用函数。这些

函数可以不管列表构成的多样性，总是能够对构成元素进行排序。目前我们暂且还不用考虑references、fun、ports和binaries类型，这些我们将会在第9章和第15章详细介绍。

使用精确等于和不精确等于运算符可以提供给编译器和类型工具更多信息，并产生更有效的代码。但是遗憾的是，`==`和`!=`不是很漂亮简洁的运算符，它们常常使得代码变得丑陋。结果在编程中，包括许多Erlang运行时的系统程序库中，经常使用等于和不同于运算符。

请启动Erlang终端，尝试一些比较运算符。请你尝试测试使用不同的数据类型和不同的等于运算符来比较结果。请确保你熟悉并掌握运算符是如何比较不同数据类型的：

```
1> 1.0 == 1.
true
2> 1.0 =:= 1.
false
3> {1,2} < [1,2].
true
4> 1 <= 1.2.
true
5> 1 /= 1.0.
true
6> (1 < 2) < 3.
false
7> (1 > 2) == false.
true
```

变量

变量用来存储简单和复合数据类型的值。在Erlang中，变量必须以大写字母开头（注2），后面的字符可以是大小写字母、整数和下划线。它们不能包含任何其他的“特殊”字符。下面是变量的一些例子：

```
A_long_variable_name Flag Name2 DbgFlag
```

Erlang中的变量不同于大多数的传统编程语言。在一个变量的生命周期内，包括在Erlang终端处理过程中，只容许给变量赋值一次，你就不能改变它了，这称为**单次赋值**。因此，当你需要对一个变量的值进行计算和操作的时候，可以把结果存储在另外一个新的变量当中。例如：

```
Double = 2,
Double = Double * Double
```

注2：变量也可以使用下划线开始，这在模式匹配中使用得比较多，我们会在“模式匹配”小节中具体讨论。

这会导致运行时错误，因为变量Double已经绑定到整数2了。尝试把它再次绑定到整数4就会出现错误。就像我们上面已经提到的，避免错误的方法就是把结果赋值给一个新的变量：

```
Double = 2,  
NewDouble = Double * Double
```

当你初次看到变量的单次赋值形式的时候可能会感觉到奇怪，但是你会很快习惯的。它促使你编写更简短的并且错误更少的代码。这也使得调试与值不正确相关的错误更加容易，因为我们在追查代码错误的根源到变量被赋值的地方时，只能有一个位置。

Erlang中的所有函数变量调用都是按值调用（call by value）的：在函数被求值之前，所有的函数调用参数都已经被求值。在Erlang中不存在引用调用的概念，这避免了引用调用会导致的一些边界效应（注3）。Erlang中所有变量都是局部的。全局变量是不存在的，这不仅使Erlang程序更容易调试，而且这也减少了出现错误的风险，避免了不良的编程习惯。

Erlang中变量的另一个有用的特性是我们不需要声明它们，只需要使用它们即可。如果程序员的编程背景是函数式编程，那么他可能已经习惯了这种用法，而如果是C或Java语言编程背景，那么他也会很快学会和欣赏这种用法。Erlang中变量不需要事前声明的原因是因为它具有动态类型系统（注4）。根据在变量上能够进行的运算操作，在运行时再确定数据类型。下面的代码尝试把一个基元和一个整数相乘，虽然它们在编译的时候不会出错（但是会有编译警告），但是在运行的时候就会抛出错误：

```
Var = one,  
Double = Var * 2
```

最初当我们使用变量必须以大写字母开头的时候，有可能会感觉到很不习惯，但我们会很快习惯的。当你使用Erlang编程好几年后，然后阅读C代码的时候，如果认为看到的任何人的代码使用的是基元而不是变量时，请不要感到惊讶。这种事情在我们身上就发生过。

在使用变量之前，请牢记：变量只能绑定一次！这可能在Erlang终端中会是个问题，因为程序有可能连续运行很多年，而使用相同的终端来跟程序进行交互。有两种措施可以作为一种变通方案来解决这个问题。使用函数f()可以对所有的变量绑定进行解绑定，而f(Variable)可以对具体变量Variable解绑定。不过只能在终端中使用它们。尝试在程序中使用它们是徒劳的，而且这还会导致一个编译错误：

注3：关于边界效应和破坏性操作我们会在本书的后面讨论。

注4：其他语言避免变量声明是因为有其他原因。例如Haskell使用类型演绎推理算法来推断变量类型。


```

1> A = (1+2)*3.
9
2> A + A.
18
3> B = A + 1.
10
4> A = A + 1.
** exception error: no match of right hand side value 10
5> f(A).
ok
6> A.
** 1: variable 'A' is unbound **

```

事实上，你在这里所看到的变量赋值是模式匹配的一种特殊情况，我们将在稍后详细讨论。

Erlang类型系统

在Erlang中没有更精细的类型系统，其中的一个主要原因是当时Erlang的创建者不知道如何去具体编写，所以它从未能得以实现。静态类型系统的优势是，错误能够在早期编译时发现，而不是等到运行时才发现，从而可以尽早地发现错误并以较低的成本修复。

有许多人都曾试图创建一个Erlang的静态类型系统。遗憾的是，由于在发明Erlang时作出的设计决定，使得没有一个项目能够完成一个全面的类型系统，因为Erlang代码热加载的特性使得这很难实现。引述Joe Armstrong在许多类型系统项目之一上的话来讲“看起来实现一个类型系统很简单，而事实上，几个星期的编程可以完成一个类型系统的95%。然后要花费几个人一年（这些都是计算机科学界最聪明的头脑之一），要不厌其烦地试图去解决剩下的5%的问题，但这真的很难。”

如果有合适的工具，就有可能不运行程序就可以发现程序中的许多错误。乌普萨拉大学开发了一款优秀的工具软件TypeEr，它可以推断Erlang的函数类型。TypeEr结合同样由乌普萨拉大学研发的另外一个软件Dialyzer，我们就能在程序编译的时候发现另外的错误，从而形成用来发现Erlang程序中错误的一个很强大的机制。我们会在第18章详细讨论这些工具。

复杂数据结构

当我们提到Erlang项元的时候，我们指的是合法的数据结构。Erlang项元可以是简单的数据值，但是我们经常使用它来描述任意复杂的数据结构。

在Erlang中，复杂数据结构由不同的数据类型复合嵌套而成。这些数据结构可能包含绑定变量或者简单和复合的值。下面是一个包含元组类型person的列表（使用基元person标记）的例子，它包括名字、姓以及包含一些属性的一个列表：

```
[{person,"Joe","Armstrong",
  [ {shoeSize,42},
    {pets,[{cat,zorro},{cat,daisy}}],
    {children,[{thomas,21},{claire,17}]}]
},
{person,"Mike","Williams",
  [ {shoeSize,41},
    {likes,[boats,wine]}]
}]
```

或者，我们可以通过如下几个步骤使用变量来做到。请注意：为了提高可读性，我们命名变量的时候可以把数据类型考虑进去：

```
1> JoeAttributeList = [{shoeSize,42}, {pets,[{cat,zorro},{cat,daisy}}],
1>                   {children,[{thomas,21},{claire,17}]}].
   [{shoeSize,42},
   {pets,[{cat,zorro},{cat,daisy}}],
   {children,[{thomas,21},{claire,17}]}]
2> JoeTuple = {person,"Joe","Armstrong",JoeAttributeList}.
   {person,"Joe","Armstrong",
   [{shoeSize,42},
   {pets,[{cat,zorro},{cat,daisy}}],
   {children,[{thomas,21},{claire,17}]}]}
3> MikeAttributeList = [{shoeSize,41},{likes,[boats,wine]}].
   [{shoeSize,41}, {likes,[boats,wine]}]
4> MikeTuple = {person,"Mike","Williams",MikeAttributeList}.
   {person,"Mike","Williams",
   [{shoeSize,41},{likes,[boats,wine]}]}
5> People = [JoeTuple,MikeTuple].
   [{person,"Joe","Armstrong",
   [{shoeSize,42},
   {pets,[{cat,zorro},{cat,daisy}}],
   {children,[{thomas,21},{claire,17}]}]},
   {person,"Mike","Williams",
   [{shoeSize,41},{likes,[boats,wine]}]}]
```

Erlang的一个优点是，它不需要明确地分配和释放内存。对于C程序员来说，这意味着不会再有不眠之夜来查找指针错误或者内存泄漏。用来存储复杂数据结构的内存存在需要的时候会被系统自动分配，而当不再引用它的时候，就会由垃圾收集器自动回收释放掉。

Erlang内存管理

当1993年开发第一个以Erlang为基础的产品的時候，批评者就说，对软实时系统使用垃圾收集器简直是太疯狂了（就像Java一样！）。虚拟机自动为系统处理分配内存，更重要的是，当它不再需要的时候它会自动回收内存（即术语“垃圾收集”）。由于垃圾收集器的精巧设计，软实时系统的效率并没有因此受到影响。

现有的Erlang虚拟机的实现使用了通用继承性垃圾收集器。垃圾收集针对每个并发进程各自独立进行：当一个处理器没有更多的内存用于存储的时候，就会自动触发垃圾收集器。

一个复制垃圾回收器有两个独立的区域（堆）来存储数据。当垃圾回收器工作的时候，它把使用的活动内存复制到其他堆栈，而其他的堆栈就会覆盖留下的垃圾内存。

垃圾回收器也是以代相传的，这意味着它有几代的堆（在Erlang中是两代）。垃圾回收器可以是浅的或者深的。一个浅的垃圾回收器只查看最新的数据，所有数据如果从三次的浅垃圾回收器中生存下来，就会把它移动到老一代中去。一个深垃圾回收器只会在一个浅的垃圾回收器回收不了足够的内存，或者执行了一定数目的回收以后，（这要看具体虚拟机版本而定）才会触发浅垃圾回收器。

模式匹配

Erlang的模式匹配可以用于：

- 变量赋值
- 控制程序的执行流程
- 从复合数据类型中提取值

通过这些功能的组合我们就能编写出简洁、可读性强而且功能强大的程序，特别是当用模式匹配来处理你所定义的一个函数的参数的時候。模式匹配可以这么写：

Pattern = Expression

正如我们前面所说的，这只是个一般化的形式。

模式（Pattern）由绑定或者未绑定的变量以及字符量（比如基元、整数或者字符串）所组成。一个绑定的变量是一个已经绑定值的变量，未绑定的变量是指一个尚未绑定值的变量。下面是一些模式的例子：

```
Double
{Double, 34}
{Double, Double}
[true, Double, 23, {34, Treble}]
```

表达式 (Expression) 可以包括数据结构、绑定变量、数学运算和函数调用。它不能包含未绑定的值。

当一个模式匹配执行的时候会发生什么呢？两种可能的结果是：

- 模式匹配成功，然后未绑定的变量因而变成绑定的变量了（会返回表达式的值）。
- 模式匹配失败，结果是没有绑定值。

是什么决定一个模式匹配的成功与否呢？一般会先计算=运算符右边的Expression，然后把它的值跟Pattern去比较：

- 表达式和模式必须有相同的形式：有三个元素的一个元组只可以匹配有三个元素的一个元组，一个列表[X|Xs]只可以匹配一个非空的列表等。
- 模式中的常量必须跟表达式相应位置上的值相等。
- 如果模式匹配成功，会把未绑定的变量绑定到相应的表达式的值上。
- 绑定变量的值必须和表达式相应位置上的值相等。

下面我们来看一个具体的例子，在Sum = 1+2中，一开始Sum是未绑定的变量，1和2相加求值后会跟变量Sum相比较。如果Sum是未绑定的，意味模式匹配成功后把Sum绑定到值3上。这里要澄清一点，Sum不是一开始绑定到1上，然后加上2。如果Sum已经绑定了，那么模式匹配只有当Sum本来就是3的情况下才会成功。

让我们看看在终端中运行的一些例子：

```
1> List = [1,2,3,4].
[1,2,3,4]
```

在命令行1中，模式匹配成功，并绑定列表[1,2,3,4]到List变量：

```
2> [Head|Tail] = List.
[1,2,3,4]
3> Head.
1
4> Tail.
[2,3,4]
```

在命令行2，模式匹配成功，因为List不是空的，所以它有一个头部和一个尾部，它们分别绑定到变量Head和Tail上。在命令行3和命令行4中你可以看到它们的输出。

```

5> [Head|Tail] = [1].
** exception error: no match of right hand side value [1]
6> [Head|Tail] = [1,2,3,4].
[1, 2, 3, 4]
7> [Head1|Tail1] = [1].
[1]
8> Tail1.
[]

```

命令行5错在哪里呢？看起来这应该运行成功，但是因为已经绑定了变量Head和Tail，所以这个模式匹配其实变成测试这个表达式是否跟[1,2,3,4]相等了，你可以看到在命令行6它就运行成功了。

如果你想提取列表[1]的头部和尾部，那么你需要使用未绑定的变量，我们可以看命令行7和命令行8，这样做就能运行成功。

```

9> {Element, Element, X} = {1,1,2}.
{1,1,2}
10> {Element, Element, X} = {1,2,3}.
** exception error: no match of right hand side value {1,2,3}

```

但是如果有一个变量就如命令行9一样反复在模式中出现会怎么样呢？变量第一次出现的时候是未绑定的，接着就绑定了它：在这里为值1。下次这个变量出现的时候它就是绑定变量了，因此只有相应的值是1的时候才会运行成功。你可以看到，命令行9运行成功，而命令行10却没有，因此“模式不匹配”的错误发生了。

```

11> {Element, Element, _} = {1,1,2}.
{1,1,2}

```

就如使用变量一样，在一个模式中我们也可以使用通配符,_。这将匹配任何东西，并且不产生任何绑定。

```

12> {person, Name, Surname} = {person, "Jan-Henry", "Nystrom"}.
{person, "Jan-Henry", "Nystrom"}
13> [1,2,3] = [1,2,3,4].
** exception error: no match of right hand side value [1,2,3,4]

```

为什么我们要使用模式匹配呢？下面我们以变量赋值为例。在Erlang中表达式Int=1是用来比较变量Int和整数1。如果Int是未绑定的，不管右边求值后如何都将绑定到Int上去，在这种情况下是值1。这就是变量赋值的实际工作流程。事实上我们不是赋值变量，而是通过模式来匹配它们。如果我们现在写Int=1，然后紧接着写Int=1+0，第一个表达式（假设Int是未绑定的）绑定变量Int到整数1。第二个表达式将把1和0相加然后和变量Int的内容相比较，而Int当前绑定值为1。由于结果是一样的，所以模式匹配将成功。如果我们写Int=Int+1，表达式的右边得到值2。试图把它的内容和变量Int比较将导致失败，这是因为变量Int已经绑定到1了。

模式匹配也可以用来控制一个程序的执行流程。接下来的章节中，我们将会介绍case语句、receive和函数语句。在上述每一种结构中，应该执行模式匹配来确定使用哪一个语句。实际上，我们测试一个模式匹配，其结果要么是成功要么是失败。例如，下面的模式匹配就会失败：

```
{A, A, B} = {abc, def, 123}
```

一开始的比较先确保表达式右边的数据类型和左边的数据类型相同，当然它们的大小也应是相同的。它们两者都是有3个元素的元组，因此到现在为止模式匹配是成功的。然后测试比较元组里面的各个元素。第一个未绑定变量A而先绑定到基元abc。由于第二个变量A也已经绑定到abc，然后把它和基元def比较，这会导致失败，因为它们的值不一样。

模式匹配[A,B,C,D] = [1,2,3]则会失败。虽然它们两个都是列表类型，但是左边的列表有4个元素，而右边一个列表只有3个元素。一个常见的误解是D可以设置为空列表，然后模式匹配应该成功。在这个例子中这是不可能的，因为C和D之间由逗号而不是构造器操作符分隔开。[A,B,C|D]=[1,2,3]的模式匹配就会成功，变量A、B和C分别绑定到整数1、2和3，变量D则绑定到尾部，在这里就是空列表。如果我们尝试[A,B|C]=[1,2,3,4,5,6,7]，A和B将分别绑定到1和2，而C则会绑定到一个包含[3,4,5,6,7]的列表。最后一个例子，[H|T]=[]会失败，因为[H|T]意味着这个列表至少要有有一个元素，而我们右边其实是一个空列表。

模式匹配的最后一个用途是用来从复合数据类型中提取值。例如：

```
{A, _, [B|_], {B}} = {abc, 23, [22, 23], {22}}
```

会成功地提取元组的第一个元素，即基元abc，并把它绑定到变量A上。同时也将成功提取第三个元素元组的第一个元素并把它绑定到变量B上。

看下面的例子：

```
14> Var = {person, "Francesco", "Cesarini"}.
{person, "Francesco", "Cesarini"}
15> {person, Name, Surname} = Var.
{person, "Francesco", "Cesarini"}
```

在第一个语句中我们把一个类型为person的元组绑定到变量Var上了，然后在第二个语句中我们分别提取了名字和姓氏。这将会成功地把字符串“Francesco”绑定到变量Name上并把字符串“Cesarini”绑定到变量Surname上。

我们刚才看到了变量可以用下划线开始，下划线是一个特殊的标记，它告诉编译器，这些变量是无关紧要的，它们只是作为程序中不需要的变量的占位符。“无关紧要的”变

量的功能跟普通变量一样，可以检查、使用和比较它们的值。唯一不同的是，普通变量的值如果从未使用过，编译器将会发出警告，而使用“无关紧要的”变量则不会。使用“无关紧要的”变量是一种很好的编程实践，这告诉阅读代码的人应该忽略掉这个值。为了提高可读性和维护性，程序员经常以“无关紧要的”变量的形式引入值和类型。下划线本身也是个“无关紧要的”变量，但不能访问其内容：因为它的值被忽略了而且从未绑定。

当模式匹配的时候，请注意“无关紧要的”变量的使用，让我们来看下面的具体例子：

```
{A, _, [B|_], {B}} = {abc, 23, [22, 23], {22}}
```

由于_变量从不绑定，因此它的值匹配不匹配都是无关的。但是，如果我们这么写：

```
{A, _int, [B|_int], {B}} = {abc, 23, [22, 23], {22}}
```

这就彻底改变了程序的语义。该变量_int先绑定到整数23，随后和一个包含整数23的列表相比较。这将导致这个模式匹配失败。

警告：使用带下划线的变量使代码更容易阅读，但是如果它在其他语句中不正确地重用，则有可能在代码中引入潜在的错误。自从为单例变量（变量仅在函数里面出现一次）引入编译器警告后，程序员就会机械地加入下划线，但往往忘记单次赋值原则和单例变量其实也会绑定值的事实。因此可以使用它们来增加代码的易读性和可维护性，但要小心确保你没有引入错误。

你可以看到就像我们说过的，模式匹配是一个很强大的机制，使用一些技巧我们可以只用一行或者两行代码就达到一些惊人的效果，如组合测试、赋值和流程控制。

尽管听起来有点啰嗦，请尝试在终端中使用模式匹配。你可以尝试定义列表以确保你已经真正掌握了这一概念，并使用模式匹配来解构你已建立的列表。尝试让模式匹配失败，然后检查和分析返回的错误（注5）。同时也请尝试使用绑定和未绑定的变量。由于模式匹配是编写简洁和美观的代码的关键，在继续学习之前弄懂它，这会让你在进步的同时尽可能地使用Erlang的特性。

函数

现在，我们已经介绍了数据类型、变量和模式匹配，那么你怎样使用它们呢？当然是在程序里面。Erlang程序包含相互调用的函数。函数组合在一起并在模块中定义。函数的

注5： Erlang运行时系统的不同版本的错误格式也是不同的。

名称是一个基元。一个函数的头包括名字，随后是一对括号，在里面包含多个形式的参数或者没有参数。在Erlang中，函数参数的数量叫做元数。使用箭头（->）来分隔函数头和函数主体。

在我们深入讨论之前，不要直接在终端中尝试输入函数。当然你也可以这么做，不过最终得到的将只会是各种语法错误。函数必须在模块中定义和单独编译。在第3章中，我们会详细讨论如何编写、编译和运行函数。

例2-1展示了一个用来计算形状面积的Erlang函数（注6）。Erlang函数是由分号分隔开的一个或者多个语句组成的，最后用句点来结束。每一个语句都有一个头部用来指定期望的参数模式，还有一个函数体，它由一个或多个以逗号分隔开的表达式组成，这些语句逐一按序执行，函数的返回值是最后一个表达式的结果。

例2-1：一个计算形状面积的Erlang函数

```
area({square, Side}) ->
    Side * Side ;
area({circle, Radius}) ->
    math:pi() * Radius * Radius;
area({triangle, A, B, C}) ->
    S = (A + B + C)/2,
    math:sqrt(S*(S-A)*(S-B)*(S-C));
area(Other) ->
    {error, invalid_object}.
```

当一个函数被调用的时候，通过检查传递给它的参数是否和在函数头中定义的模式匹配，它的语句被按序检查。如果模式匹配成功，就绑定变量，执行相应的函数体。如果没有匹配成功，就会模式匹配下一个语句。当定义一个函数的时候，应该确保每个参数都有一个语句匹配成功，这常常可以通过使用最后一个默认匹配语句来包含其他所有的（剩余）情况。

在例2-1中，area是计算正方形、圆和三角形面积的一个函数，它也可能会返回一个元组{error,invalid_object}。下面我们尝试调用这个面积函数：

```
area({circle, 2})
```

模式匹配在第一个语句失败了，因为尽管我们在变量和参数部分都有一个元数为2的元组，但是基元square和circle不匹配。然后执行第二个语句并且模式匹配成功，结果是变量Radius绑定到整数2。该函数的返回值将是这些函数语句中的最后一个表达式的值，math:pi()*2*2，即12.57（四舍五入）。当一个语句匹配成功了，余下的部分就不会再执行了。

注6： 在三角形的情况下，面积使用海伦公式计算，而math:sqrt/1是用来求一个浮点数的平方根。

以函数头`area(Other)->`开始的最后一个语句是一个默认匹配语句。因为变量`Other`没有绑定，当先前的其他三个语句模式匹配都失败时，它将匹配任何一个对`area`函数的调用：`{error, invalid_object}`。

一个常见的错误是影子语句永远不会被匹配。在下面例子的`flatten`函数中，它总是返回`{error, unknown_shape}`，因为对`Other`进行模式匹配总是成功的，所以`Other`会绑定到任何一个传递给函数`flatten`的参数上，这也包括`cube`和`sphere`：

```
flatten(Other) -> {error, unknown_shape};
flatten(cube) -> square;
flatten(sphere) -> circle.
```

让我们来看一个阶乘函数的例子：

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

如果我们调用`factorial(3)`，第一个语句模式匹配失败，因为`3`不匹配`0`。运行时系统会尝试匹配第二个语句，`N`是未绑定的，因此将成功地把`N`绑定到`3`。这一条语句返回`3 * factorial(2)`。运行时系统无法返回任何值，直到它执行完`factorial(2)`，然后它的值才能乘以`3`。调用`factorial(2)`将会进行第二次模式匹配，这次`N`绑定到`2`，并返回`2 * factorial(1)`，结果是继续调用`1 * factorial(0)`。`factorial(0)`和第一个语句模式匹配成功，它返回`1`。这意味着第三层的`1 * factorial(0)`返回`1`，第2层的返回`2 * 1`，第一层返回`factorial(3)`的结果，即`6`：

```
factorial(3).
Level 1: 3 * factorial(3 - 1)      (returns 6)
Level 2:    2 * factorial(2 - 1)   (returns 2)
Level 3:      1 * factorial(1 - 1) (returns 1)
Level 4:          1                (returns 1)
```

正如我们前面提到过的，模式匹配发生在函数的头部，如何匹配成功，就会绑定变量`N`的一个实例。对每个语句来讲变量都是本地的，它们不需要分配或释放，Erlang运行时系统会自动处理这些。

模块

函数组合在一起构成了模块。一个程序往往是分散在几个模块当中，每个模块包括按逻辑组合在一起的函数。模块文件以`.erl`后缀来结尾，文件名称和模块名称必须是相同的。模块可以这样直接命名`-module(Name)`，因此在例2-2中，`demo`模块将存储在一个名为`demo.erl`的文件中。

例2-2: 一个模块实例

```
-module(demo).  
-export([double/1]).  
  
% This is a comment.  
% Everything on a line after % is ignored.  
  
double(Value) ->  
    times(Value, 2).  
times(X, Y) ->  
    X*Y.
```

`export`指令以Function/Arity的格式, 包含了导出函数的一个列表。这些函数是全局性的, 这意味着可以从模块的外部调用它们。在Erlang中, 注释以百分号(%)开始直到该行结束。请务必在代码中多多使用它们!

全局调用, 也称为完全限定的函数调用, 是在函数前加上模块名字作为前缀来实现的。因此, 在例2-2中, 调用`demo:double(2)`将返回4。局部函数只可以在模块内部使用。在模块里面使用模块名调用它们的时候, 会导致运行时错误。例2-1中的`math:sqrt/1`是什么意思呢? 它其实是调用Erlang标准版本中的`math`模块的`sqrt`(平方根)函数。

Erlang中的函数由它们的名字、元数和定义所在的模块唯一标示的。两个函数在同一模块中可能有相同的名字, 却有不同的元数。如果是这样, 它们就是不同的函数, 并且相互之间没有关系。只要函数是在模块中定义的, 那么就没有必要在调用之前声明它们。

Erlang的编译和虚拟机

若要执行从一个模块中导出的函数, 你必须编译代码, 这会在与模块相同的目录下面生成一个`module.beam`文件:

- 如果你使用的是Unix操作系统, 请在源代码目录下面打开Erlang终端。
- 在Windows环境下, 一种打开`werl`终端的方法是右击一个`.beam`文件, 然后从弹出的窗口菜单中选择Open With option的“`werl`”选项。从现在起, 双击任何源文件相同目录下的`.beam`文件, 就会打开一个Erlang终端了。

在这两种操作系统下, 你可以在Erlang终端中使用`cd(Directory)`进入到各个目录。一旦进入该目录, 你就可以在Erlang的终端中使用`c(Module)`并省略名称中的`.erl`后缀来进行编译。如果代码中没有错误, 编译就会成功。

大型的Erlang系统一般由多个松散耦合的Erlang模块组成, 它们都在独立的基础上各自编译。一旦你编译了代码, 在源代码目录下面我们就能找到一个与模块名称相同的文

件，但以后缀`.beam`结尾。这个文件包含了你可以从其他任何函数调用的字节代码。后缀`.beam`指的是Björn的Erlang抽象虚拟机，它是一个可以运行编译代码的抽象虚拟机。

一旦编译成功，你就可以使用完全限定名称来调用这些函数了。这是因为你是从模块外部调用这些函数的。调用没有导出的函数将会导致运行错误：

```
1> cd("/home/francesco/examples").
/home/francesco/examples
ok
2> c(demo).
{ok, demo}
3> demo:double(10).
20
4> demo:times(1,2).
** exception error: undefined function demo:times/2
```

模块指令

每个模块都拥有一个`-attribute(Value)`格式的属性列表。它们通常放在模块的开始，是由放在属性前面的`-`符号和结尾的句点来定义。`module`属性是强制性的，它定义了模块的名称。另一个我们介绍过的属性`export`包含一个函数/元数定义的列表。

编程时一个有用的指令是`-compile(export_all)`指令，它会在编译阶段导出模块中的所有函数。另一个方法是在编译这个文件的时候加上特定的选项：

```
c(Mod,[export_all]).
```

这个指令应该只用于测试目的。在代码被添加到产品中的时候，请不要像很多其他人一样忘记使用`export`指令来取代它！`compile`指令还包含其他很多只有在特殊条件下才会使用的选项。如果你很好奇，想知道有关它们的更多信息，那么请查看有关`compile`模块的手册。

另一个有用的指令是`-import(Module, [Function/Arity,...])`。它允许你从其他模块中导入函数然后本地调用它们。再来看看我们前面计算面积的那个例子，在模块中加入`-import(math,[sqrt/1])`指令将允许你调用系统函数来计算三角形的面积。这里再提醒一次，请不要忘记最后加上句点结束：

```
-import(math, [sqrt/1]).

area({triangle, A, B, C}) ->
  S = (A + B + C)/2,
  sqrt(S*(S-A)*(S-B)*(S-C));
```

使用import指令可能使你的代码难以理解。人们一开始看到sqrt/1的时候可能会认为这是个本地的函数，于是在该模块中寻找，这当然会失败了。另外一种情况，他可能检查文件的头部确定是否已经导入它们了。因此，尽量少用导入语句在Erlang社区里面是个约定的习俗。

你可以编写自己的模块属性。常见的例子包括：`-author(Name)`和`-date(Date)`。用户定义的属性只能有一个参数（不像其他的一些内置属性）。

所有的属性和模块的其他信息可以通过调用`Mod:module_info/0`或者选择性调用`Mod:module_info/1`函数得到。在终端中你可以使用`m(Module)`命令：

```
5> demo:module_info().
[{exports, [{double, 1}, {module_info, 0}, {module_info, 1}]},
 {imports, []},
 {attributes, [{vsn, [74024422977681734035664295266840124102]}]},
 {compile, [{options, []},
             {version, "4.5.1"},
             {time, {2008, 2, 25, 18, 0, 28}},
             {source, "/home/francesco/examples/demo.erl"}]}]

6> m(demo).
Module demo compiled: Date: February 25 2008, Time: 18.01
Compiler options: []
Object file: /home/francesco/examples/demo.beam
Exports:
    double/1
    module_info/0
    module_info/1

ok
```

如果你阅读过Erlang库，那么可能会遇到其他的一些属性，其中包括：`-behaviour(Behaviour)`、`-record(Name, Fields)`和`-vsn(Version)`。请注意，在demo模块中我们没有任何vsn属性，但其在前面的例子中出现过。当vsn没有定义的时候，编译器将会把它设置成模块的MD5码。还要注意module_info函数出现在导出函数的列表里面，这意味着可以在模块的外面访问到它们。现在请不要担心records、vsn和behaviour，我们还将第7章和第12章里详细介绍它们。

在本章中，我们介绍了Erlang的基础，看到了它的一些特有的东西：你可以赋值给变量，但只能一次；你可以模式匹配一个变量或者该匹配可能转化为测试和这个变量是否相等。还有该语言的其他一些更为熟知的功能，比如模块系统和基本数据类型等。

在本章的基础上，我们将在第3章详细讨论序列编程，在第4章介绍并发编程，它可能是Erlang语言最重要的一个特性。

练习

练习2-1：Erlang终端

在终端中输入如下Erlang表达式来查看它们的结果。它们将展示模式匹配的运行机制和本章描述的变量的单赋值形式。它们运行的时候会发生什么？它们的返回值会是什么呢？为什么呢？

A. Erlang表达式

```
1 + 1.  
[1|[2|[3|[]]]].
```

B. 通过模式匹配赋值

```
A = 1.  
B = 2.  
A + B.  
A = A + 1.
```

C. 列表递归定义

```
L = [A|[2,3]].  
[[3,2]|1].  
[H|T] = L.
```

D. 通过模式匹配来控制执行流程

```
B = 2.  
B = 2.  
2 = B.  
B = C.  
C = B.  
B = C.
```

E. 从复合数据类型中提取值

```
Person = {person, "Mike", "Williams", [1,2,3,4]}.  
{person, Name, Surname, Phone} = Person.  
Name.
```

练习2-2：模块和函数

请复制本章的demo模块例子。编译并尝试在终端中运行它。当调用demo:times(3,5)的时候会发生什么呢？如果省略模块名称，调用double(6)又会怎么样呢？

创建一个新的模块shapes，并在其中放入复制函数area。不要忘记包括所有的模块指令

和导出指令。在终端中编译并运行这个`area`函数。当编译它的时候，为什么会得到一个`Other`变量是未使用的警告呢？如果重命名这个变量为`_Other`会发生什么变化呢？

练习2-3：简单模式匹配

请编写一个模块`boolean.erl`，它接受逻辑表达式和布尔值（用基元`true`和`false`表示）并返回布尔值的结果。这些函数应该包括`b_not/1`、`b_and/2`、`b_or/2`和`b_nand/2`。编写过程中不可以使用逻辑结构`and`、`or`和`not`，但是可以使用模式匹配来达到目标。

在终端中测试模块。下面是一些在模块中调用导出函数的例子：

```
bool:b_not(false) => true
bool:b_and(false, true) => false
bool:b_and(bool:b_not(bool:b_and(true, false)), true) => true
```

`foo(X)=Y`指的是调用带有参数`X`的函数`foo`会返回值`Y`。请记住在Erlang中`and`、`or`和`not`是保留字，因此在你命名函数的时候请使用前缀`b_`。

提示：请使用`b_not/1`和`b_and/2`实现`b_nand/2`。



第3章

Erlang顺序编程

Erlang的设计受到了函数式和逻辑式编程语言的很大影响。在处理有序程序的时候，那些熟悉Prolog、ML或者Haskell编程语言的程序员会发现它们影响了Erlang的结构和编程技术。当用函数式编程语言编程的时候，可以使用递归编程技术来代替迭代结构，如while和for循环语句。

递归是函数式编程语言中最有用和最强大的工具。它允许程序员通过使用对相同函数的连续调用来遍历一个数据结构，同时函数调用的模式也就反映了数据的结构。由此产生的程序更紧凑和更容易理解与维护。函数式编程语言的一个重要特性：它是无边界效应的，除非是在打印或者访问外部存储的时候特别需要边界效应。

你可以通过各种条件结构控制递归，用于增强模式匹配的表达能力。例如在遍历数据结构的时候，使用不同的模式来形成不同的遍历：自下而上、自上而下和广度优先等。

本章还将介绍与顺序编程直接相关的其他一些特性。Erlang缺乏一个强类型系统，以及有很多灵活和动态的结构特点，使得很少需要处理运行时错误。通过异常处理机制，Erlang程序可以从错误中恢复然后继续执行。

在Erlang发布版本中包含很多模块，这些模块又包含了很多库、工具和实用程序以及完整的应用。在Erlang的每个新的发布版本中又会有新的模块加入，对现有的库来说往往会增加新的功能。其中的一些库构成了在Erlang世界中所谓的内置函数（BIF），因为它们是Erlang运行时系统的一部分。它们完成的功能要么是无法用Erlang编写，要么是如果它们由用户定义，则其执行效率会很低。

条件评估

Erlang拥有条件评估计算形式，它们是可以（至少部分）互换的。第一种形式在第2章我们已经遇到过：通过函数的参数进行模式匹配来进行函数的选择执行。第二种是和前

一种类似的case结构。第三种形式是if结构，你可以把它看做是case结构形式的简单表现。让我们先从case结构开始。

case结构

case结构依靠模式匹配来判断应该执行哪些语句，它跟通过模式匹配来选择执行函数很类似。不同的是，case不是把函数的真实参数和形式参数进行模式匹配，而是通过对一个表达式求值，然后对它的结果和一个由分号隔开的模式匹配列表进行匹配。

一般情况下case表达式有如下形式：

```
case conditional-expression of
  Pattern1 -> expression1, expression2, .. ;
  Pattern2 -> expression1, expression2, .. ;
  ... ;
  Patternn -> expression1, expression2, ..
end
```

使用的关键字是case、of和end。对conditional-expression求值，然后和Pattern1,..., Patternn进行匹配，直到第一个模式匹配成功。->符号把语句的模式或者头部与由逗号分隔的表达式列表组成的主体分开。一旦模式匹配成功，选中的语句表达式就会一个个按次序执行，case结构的结果就是最后一个表达式的运行结果。

在下面情况的case结构中，使用lists库模块中的member函数检查基元foo是否是列表List的一个成员元素，如果是，返回基元ok，否则返回元组{error,unknown_element}:

```
case lists:member(foo, List) of
  true -> ok;
  false -> {error, unknown_element}
end
```

case表达式总是返回一个值，因此总是可以把它的返回值绑定到一个变量。在Erlang终端中直接使用case语句也是可以的，但很少这样做。到目前为止，介绍的结构已经足够复杂了，因此现在有必要在模块中输入一些实验性质的函数，编译并运行它们来熟悉所学的内容。

与函数定义一样，case表达式的结果必须和一个模式匹配，否则将会得到一个运行时错误。如果在最后的模式中有_或者未绑定变量，那么它将匹配任何Erlang项元，这就如我们在第2章中讨论的catch-all语句一样。catch-all语句不是必需的，事实上，不鼓励把它作为一种防错性编程的形式来使用。（请参阅下面关于“防御性编程”的内容来获取更多关于catch-all语句的信息。）

防御性编程

假设程序把一个整数映射到一个表示一周中某天的基元。使用catch-all语句的防错性编程如下所示。

```
convert(Day) ->
  case Day of
    monday   -> 1;
    tuesday  -> 2;
    wednesday -> 3;
    thursday -> 4;
    friday   -> 5;
    saturday -> 6;
    sunday   -> 7;
    Other    -> {error, unknown_day}
  end.
```

我们强烈建议不要采取这种做法。更好的办法是使程序的convert函数带有语句错误信息（即没有语句相匹配）终止，因为这样很容易定位错误会发生的地方。

另一种处理方法是返回一个错误值，但随后每个调用convert/1的函数都必须处理这个错误，否则会有产生运算错误的危险，这是因为convert函数的结果有可能不是整数，而是一个元组。

曾经有一次让我们在200万行的源代码中寻找一个函数这样的防错性编程，它返回了一个error的元组，结果在系统完全不同的部分导致了一个匹配错误。

函数定义和case表达式有很多共同点。下面来看一个带一个参数的计算列表长度的函数的简单例子：

```
listlen([])      -> 0;
listlen([_|Xs]) -> 1 + listlen(Xs).
```

你也可以直接使用case表达式这样重写：

```
listlen(Y) ->
  case Y of
    []      -> 0;
    [_|Xs] -> 1 + listlen(Xs)
  end.
```

在带有多个参数的函数中，可以同时对所有参数进行模式匹配：

```
index(0,[X|_])      -> X;
index(N,[_|Xs]) when N>0 -> index(N-1,Xs).
```

另一方面，一个case表达式可以匹配单一的表达式。用参数构造的元组，可以让我们使用case语句来定义index：

```
index(X,Y) ->
    index({X,Y}).

index(Z) ->
    case Z of
        {0,[X|_]}      -> X;
        {N,[_|Xs]} when N>0 -> index(N-1,Xs)
    end.
```

同样，通过嵌套的case表达式，模式匹配可以分别匹配两个参数。

```
index(X,Y) ->
    case X of
        0 ->
            case Y of
                [Z|_] -> Z
            end;
        N when N>0 ->
            case Y of
                [_|Zs] -> index(N-1,Zs)
            end
    end.
```

因此，使用模式匹配定义函数要比使用case表达式代码更紧凑。但是要记得case表达式不单单只用于头部，它可以用于函数定义的任意地方，每种模式匹配都有在Erlang中发挥作用的地方。每个case分支只有一条语句，在Erlang中认为这是不好的做法。在我们的例子中，这样做只是为了简单阐明这一点。你应该使用模式匹配来避免出现这样的代码。

变量范围

变量的范围是指程序中变量可以使用的区域。相同的变量name可以在程序的许多地方使用，一些有可能指向同一个变量，而另一些则可能指向其他不同的变量，它们只不过是恰好有相同的名称而已。看看下面的例子：

```
f(X) -> Y=X+1,Y*X.
```

X在函数的起始语句中声明，因此它的范围是整个区域，也就是Y+X、Y*X，而Y的范围是在它声明后的余下部分，在这里是单一的表达式Y*X。

在接下来的例子中，有两个不用的变量，而名称都是Y。在函数f/1中定义和使用第一个变量。第一个变量在函数f/1中定义，并在最后一条语句中使用。第二个变量在函数g/1的第二种实现的起始语句中定义，它的有效范围是这个实现的整个区域：

```

f(X)      -> Y=X+1,Y*X.
g([0|Xs]) -> g(Xs);
g([Y|Xs]) -> Y+g(Xs);
g([])     -> 0.

```

正如之前所说，在Erlang中一个变量的范围是在相同函数中变量被绑定后的任意位置。不管这个变量是通过“=”明确绑定的，还是作为模式匹配的部分来绑定的。当只在case或者if结构的一些语句中绑定变量，而后面在同一个函数体中再一次使用它的时候，这就会产生问题。请看下面反映这个问题的小例子：调用unsafe(one)和unsafe(two)的结果是什么呢？事实上，我们不需要担心这些，因为编译器不会编译通过这些包括“不安全”变量的模块，也就是只在一条case或者if分支中定义变量，却在分支外使用该变量：

```

unsafe(X) ->
  case X of
    one -> Y = true;
    _   -> Z = two
  end,
  Y.

```

只有在所有的case或者if分支中都绑定一个变量的时候，才可以安全地使用它。这是一种不好的编程习惯，因为这样往往使代码更难阅读和理解。下面是我们安全使用变量的理想方法，对应的变量只绑定一次，而变量的值由一个case语句确定：

```

safe(X) ->
  case X of
    one -> Y = 12;
    _   -> Y = 196
  end,
  X+Y.

preferred(X) ->
  Y = case X of
    one -> 12;
    _   -> 196
  end,
  X+Y.

```

if结构

if结构就像一个没有conditional-expression和of关键字的case语句：

```

if
  Guard1 -> expression11, expression12, .. ;
  Guard2 -> expression21, expression22, .. ;
  ... ;
  Guardn -> expressionn1, expressionn2, ..
end

```

保护元表达式Guard1,...,Guardn按次序进行计算，直到其中的一个计算结果为true。如果结果是Guardi，那么就会执行下面这些语句：

```

expressioni1, expressioni2,..., expressionin

```

整个if表达式的结果是上面这一系列语句的返回值，它就被执行的分支中最后执行语句的计算值。

保护元表达式是Erlang中布尔表达式语句的子集，它只能调用有限的包含比较运算符和算术运算符的函数。在下一节中将详细了解保护元可能包括哪些。

如果没有保护元语句的计算值是基元true，那么运行时就会产生错误。要得到一个catch-all的语句，可以让最后一个保护元语句的值是基元true，当然这不是强制性的。

在下面的例子中，检查变量X，用以确定它的值是否小于、大于或等于1：

```
if
  X < 1 -> smaller;
  X > 1 -> greater;
  X == 1 -> equal
end
```

上面这个例子也可以使用catch-all语句实现，这样做可以清楚地表明它将始终返回一个结果：

```
if
  X < 1 -> smaller;
  X > 1 -> greater;
  true -> equal
end
```

特别是那些具有命令式编程语言背景的Erlang新手，他们往往会过度使用if语句，其实可以在case语句中使用模式匹配来更优雅地实现相同结果。下面的例子演示了当if语句有一系列保护元来测试特定表达式值的时候，如何使用case表达式来重写它：

```
if                                     case X rem 2 of
  X rem 2 == 1 -> odd;                  1 -> odd;
  X rem 2 == 0 -> even                 0 -> even
end                                    end
```

为了检查对if和case表达式的理解，请尝试在终端或者可以编译并运行小程序中使用这些表达式。要特别注意它们的返回值，请尝试绑定这个值到一个变量，并在随后的计算中使用它。也请尝试如下测试，写一个带有不安全变量的程序（在一些语句中定义，而另一些则没有），然后尝试编译它。

保护元

保护元（guard）是一个额外的限制条件，它应用于函数的case或者receive语句中（我们会在第4章中具体讲述receive表达式）。保护元应该放在“->”之前来分隔语句的主体和头部。

保护元由when关键字和紧跟其后的一个保护元表达式组成。只有在模式匹配和保护元表达式求值结果为基元true的情况下，这个语句才会执行。

接下来重写第2章中阶乘的例子：

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

这一次使用保护元：

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

我们对阶乘函数中的语句进行了重新排序。在以前的版本中，我们不得不使用factorial(0)作为第一个语句，用以确保该函数能正确结束。现在只有当参数N大于0的时候我们才选择递归语句（即一个调用它自身的阶乘函数）。

如果是由模式匹配和保护元一起来唯一确定哪些语句应该选中，那么这些语句的顺序就变得无关紧要了，正如现在所讲述的情况。我们应该注意的关于重写阶乘函数的最后一件事情是，当调用factorial(-1)的时候，因为-1小于0和不等于0，所以没有语句可以选择而发生运行时错误。在以前的阶乘函数版本中，该函数会永远循环下去而不返回一个值，这是因为factorial(-1)会调用factorial(-2)，依次类推。这个有可能会最终导致Erlang运行时系统由于内存溢出而终止。

单独的保护元表达式可以使用如下的结构获得：

- 约束变量
- Erlang常量的数据值，包括数字、基元、元组和列表等
- 类型测试语句，比如is_binary、is_atom、is_boolean和is_tuple等
- 第2章中所列出的项元比较运算符==、!=、<、>等
- 第2章中所列出的使用算术运算符组成的算术表达式
- 第2章中所列出的布尔表达式
- 保护元内置函数

导致运行时错误的保护元子表达式被视为返回false。

可以这样写如下一个例子：

```
my_add(X,Y) when not(((X>Y) or not(is_atom(X)) ) and (is_atom(Y) or (X==3.4))) ->
    X+Y.
```

这表明保护元可以是测试的复杂组合，但并不允许引用任何用户自定义的函数。

规定开发人员不能用语句实现自己的保护元函数的原因是限制它们的操作，从而确保保护元语句不会产生边界效应。在找到一个成功的语句执行之前，将会执行所有的测试保护元语句，这意味着假设在一个保护元里调用一个`io:format`，如果它失败了，你还是会看到打印输出，即使这个语句没有被选中执行。

注意：在Erlang语言中有一些类型测试内置函数的老版本，它们的名称就是类型名称：`atom/1`和`integer/1`等。不推荐使用这些内置函数，因为它们已经过时，它们的存在只是为了向后兼容。新的保护元函数是`is_atom/1`和`is_integer/1`等。

Erlang允许保护元进行简单的逻辑组合，以不同的方式实现：

- 用逗号(,)来分隔各个保护元语句，这是一种逻辑乘，因此只有在串行序列中所有表达式的值都是`true`的时候，它的结果才为`true`。
- 用分号(;)来分隔各个保护元语句，这是一种逻辑加（或者实际上是逗号分隔的逻辑乘），如果有一个表达式的值为`true`，则它的结果就是`true`。

作为一个使用“... ; ...,...”符号的例子，我们可以重写保护元函数（在应用几次德摩根定律之后）如下：

```
my_add2(X,Y) when not(X>Y) , is_atom(X) ; not(is_atom(Y)) , X/=3.4 ->
    X+Y.
```

只使用逗号或者分号的简单组合是优美的，在实践中我们不建议把分号和逗号混合使用，因为这样太容易产生逻辑错误了。

在结束本节之前，请复制下面的例子并在终端里运行它。函数`even`对一个整数除以2取余数，如果结果是0，它返回基元`true`；如果是1的话，说明这个整数不是偶数，那么它返回`false`。请尝试用一个浮点数或者基元作为参数调用`even`函数，看看到底会发生什么。第二个函数`number`，将根据传入的参数`number/1`返回基元`float`或者`integer`。如果传入的参数不是数字，函数就返回`false`：

```
-module(examples).
-export([even/1, number/1]).

even(Int) when Int rem 2 == 0 -> true;
even(Int) when Int rem 2 == 1 -> false.

number(Num) when is_integer(Num) -> integer;
number(Num) when is_float(Num)  -> float;
number(_Other)                  -> false.
```

内置函数

本节介绍一些较常用的根据函数类型分组的内置函数，我们会以实例来说明它们的使用方法。内置函数的简写是BIF，这在Erlang社区里面已经是一种习惯。在erlang模块的手册页中列出了标准和非标准的内置函数。

内置函数通常用C语言编写，并集成到虚拟机（VM）中，可用于操作、检查并获取数据，以及与操作系统进行交互。一个数据操作函数的例子是转换一个基元为一个字符串：`atom_to_list/1`。其他内置函数，例如`length/1`，它返回一个列表的长度，为了保证效率，它在运行时系统里面实现。

最初，所有的内置函数都属于erlang模块，目前在现实中因为考虑实用性和有效性有一些也在其他模块中实现。其中包含有内置函数的模块还有ets和lists。

虽然大多数内置函数视为Erlang的内部组成部分，但是其他一些则要看不同的虚拟机实现，它们并不一定存在于其他虚拟机实现中，或者现有虚拟机的特定操作系统版本中。标准的内置函数会自动包括进来，因此，调用它们不需要加模块前缀。但是非标准的内置函数，必须加上erlang模块前缀，如`erlang:function`。下面是一些非标准的内置函数的例子，如`erlang:hash(Term, Range)`，它返回Term在指定范围内的散列值，而`erlang:display(Term)`会打印出这个Term的标准输出，它主要用于调试目的。

对象存取及检查

Erlang中有很多的内置函数来处理内置类型，例如列表和元组：

`hd/1`

返回列表的第一个元素。

`tl/1`

返回删除第一个元素后的其余部分。

`length/1`

返回一个列表的长度。

`tuple_size/1`

返回元组元素的数目。

`element/2`

返回元组的第 n 个元素。

`setelement/3`

替换元组中的一个元素，并返回新的元组。

`erlang:append_element/2`

向元组添加一个元素作为最后的元素。

下面的代码展示了如何使用这些函数：

```
1> List = [one,two,three,four,five].
[one,two,three,four,five]
2> hd(List).
one
3> tl(List).
[two,three,four,five]
4> length(List).
5
5> hd(tl(List)).
two
6> Tuple = {1,2,3,4,5}.
{1,2,3,4,5}
7> tuple_size(Tuple).
5
8> element(2, Tuple).
2
9> setelement(3, Tuple, three).
{1,2,three,4,5}
10> erlang:append_element(Tuple, 6).
{1,2,3,4,5,6}
```

类型转换

类型转换必须是内置函数，因为它们改变了数据的内在表示方法。即使可行，但也不可能用Erlang语言编写出有效率的转换函数。类型转换函数有许多，它们不仅改变数值类型，而且同时也可以为基础类型和可打印类型（即字符串）之间相互转换。当转换浮点数为整数的时候，你可以选择四舍五入或者简单去掉尾部：

`atom_to_list/1`, `list_to_atom/1`, `list_to_existing_atom/1`

它们实现基元和字符串的相互转换。如果基元在运行时系统中的当前会话里没有使用过，那么调用函数`list_to_existing_atom/1`将会失败。

`list_to_tuple/1`, `tuple_to_list/1`

这两个函数实现元组类型和列表类型的相互转换。

`float/1`, `list_to_float/1`

这两个函数都产生一个`float`类型，一个是把一个整数参数转换成一个浮点数，另外一个是把一个字符串转换成一个浮点数。

`float_to_list/1`, `integer_to_list/1`

这两个函数都返回字符串。

round/1, trunc/1, list_to_integer/1

它们都返回整数。

让我们来看看它们是如何运行的：

```
1> atom_to_list(monday).  
"monday"  
2> list_to_existing_atom("tuesday").  
** exception error: bad argument  
   in function list_to_existing_atom/1  
   called as list_to_existing_atom("tuesday")  
3> list_to_existing_atom("monday").  
monday  
4> list_to_tuple(tuple_to_list({one,two,three})).  
{one,two,three}  
5> float(1).  
1.00000  
6> round(10.5).  
11  
7> trunc(10.5).  
10
```

进程字典

有一类内置函数可以允许函数把值和一个键关联并存储，以后可以在程序的其他地方重新获取它，它们称为进程字典。遗憾的是，检索和操纵这些值将引入Erlang中的全局变量。

使用进程词典可以在编程的时候提供一个快速的方法，但结果是代码非常难以调试和维护。大多数的Erlang函数是没有边界效应的，当程序崩溃的时候，传递给函数的参数通常包含了足够的信息来解决错误。引入进程字典后往往使这个过程大大复杂化，因为当程序崩溃的时候，进程字典的状态值也同时丢失了。我们将不在本书中包括这些内置函数，这是因为我们不鼓励坏习惯。尽管如此，如果你还是希望写丑陋的、难以调试的程序，或者准备参加一个Erlang的混乱编程竞赛的话，你可以阅读有关的Erlang发布版本的说明文件中的这些内置函数。这样至少你不能说，你的这个坏习惯是从我们这里学到的。

元编程

元编程是指某类计算机程序的编写，这类计算机程序编写或者操纵其他程序（或者自身）作为它们的数据，或者在运行时完成部分本应在编译时完成的工作。一个函数在运行时才确定将调用哪些函数的特性叫做元编程，也就是程序创建程序并运行。对于这样的用法，我们可以使用带三个参数的apply/3函数来实现，即模块名称、导出的函数名和参数列表。当调用时，它执行以参数给出名称的函数并返回其结果。

`apply/3`的美妙之处是模块、函数和参数无须在编译时就已知。它们可以当做变量传递给内置函数。因此，在下面的例子中，调用`apply/3`返回的值就是`examples:even(10)`的返回值：`true`。能够动态确定函数运行是编写通用代码必不可少的：

```
1> Module = examples.  
examples  
2> Function = even.  
even  
3> Arguments = [10].  
[10]  
4> apply(Module, Function, Arguments).  
true
```

一个初学者常犯的错误是使用`apply`的时候忘了把参数（即使里面只有一个）放入一个列表中。假设我们使用`apply`调用前面定义的函数`listlen`的时候忘记将参数放进一个列表中：

```
5> apply(sequential, listlen, [2,3,4]).  
** exception error: undefined function sequential:listlen/3
```

下面我们来看看如何正确使用`apply`：

```
6> apply(sequential, listlen, [[2,3,4]]).  
3
```

这正是我们想要的结果。

如果参数的个数是在编译时就已知的，那么你可以使用以下表示方法（如果有两个参数）：

```
Mod:Fun(Arg1, Arg2)
```

来取代更通用的`apply(Mod, Fun, [Arg1, Arg2])`。我们将在第9章的高阶函数里讨论动态创建函数的其他方法。

进程、端口、发布和系统信息

在讲解并发处理的章节中我们会讨论跟进程、进程审查和错误处理直接有关的几个内置函数。我们同时也会讨论端口处理和软件分发。对这些内置函数的讨论会在本书的相关章节里面出现。我们可能想知道各种各样的关于系统的信息，这些都可以通过内置函数得到。这些信息包括低级别的系统信息、跟踪栈以及当前时间和日期。这份清单会很长，但它们都可以在`erlang`模块的文档中找到。

函数`date/0`以元组`{Year, Month, Day}`的形式返回当前日期，函数`time/0`以元组`{Hour, Minute, Second}`的形式返回当前时间。而函数`now/0`则以元组`{MegaSeconds, Seconds,`

MicroSeconds}}的形式返回一个从1970年1月1号午夜算起的秒数。now/1内置函数会返回在特定的Erlang节点的唯一值，即使它们在同一微秒里被多次调用，因此它可以用作一个唯一的标识符。

输入和输出

io模块提供了Erlang程序的输入和输出功能。在本节中，我们将描述从标准输入设备读取和在标准输出设备中输出等主要功能。每个函数都可以接受文件句柄（属于类型io_device()）作为额外的（第一）参数：在file模块中定义文件操作。

要从标准输入读取一行，请使用io:get_line/1，它需要提示符字符串（或基元）作为它的输入：

```
1> io:get_line("gissa line>").
gissa line>lkdsjfljasdkjflkajsdf.
"lkdsjfljasdkjflkajsdf.\n"
```

也可以这样读取指定数量的字符：

```
2> io:get_chars("tell me> ",2).
tell me> er
"er"
```

最有用的输入函数是io:read/1，它可以从标准输入中读入一个Erlang项元：

```
3> io:read("ok, then>>").
ok, then>>atom.
{ok,atom}
4> io:read("ok, then>>").
ok, then>>{2,tue,{mon,"weds"}}.
{ok,{2,tue,{mon,"weds"}}}
5> io:read("ok, then>>").
ok, then>>2+3.
{error,{1,erl_parse,"bad term"}}
```

就像命令行5提醒我们的一样，项元是一个完全求过值的值，而不是任意的Erlang表达式，如2+3。

在Erlang中输出是由函数io:write/1所提供的，这将打印一个Erlang项元，但最常用的函数是提供格式化的输出io:format/2。io:format可以带有：

- 一个格式化的字符串（或二进制），它们可以控制参数的格式化；
- 一个需要打印的包含值的列表。

格式化字符串中包含了需要的印刷字符以及控制格式化的序列。

控制序列以波形字符开始 (~)，最简单的形式是一个单独的字符，说明如下：

~c

输出一个字符的ASCII码。

~f

输出一个有6个小数位的浮点数。

~e

输出一个以科学计数法表示的总共6位的浮点数。

~w

以标准语法输出任何项元。

~p

输出数据就如~w，但在“pretty printing”模式下，在适当的地方换行和缩进，在可能的情况下把列表作为字符串输出。

~W、~P

输出就如~w、~p，但限制结构深度为3。在数据列表中它有额外的一个参数来指明打印项元的最大深度。

~B

以10为基数输出一个整数。

在这里我们来看一些具体例子：

```
1> List = [2,3,math:pi()].
[2,3,3.141592653589793]
2> Sum = lists:sum(List).
8.141592653589793
3> io:format("hello, world!\n",[ ]).
hello, world!
ok
4> io:format("the sum of ~w is ~w.\n", [[2,3,4],ioExs:sum([2,3,4])]).
the sum of [2,3,4] is 9.
ok
5> io:format("the sum of ~w is ~w.\n", [List,Sum]).
the sum of [2,3,3.141592653589793] is 8.141592653589793.
ok
6> io:format("the sum of ~W is ~w.\n", [List,3,Sum]).
the sum of [2,3|...] is 8.141592653589793.
ok
7> io:format("the sum of ~W is ~f.\n", [List,3,Sum]).
the sum of [2,3|...] is 8.141593.
ok
```

完整的控制序列形式为~F.P.PadC，其中F为字段的输出宽度参数，P是它的精度，Pad是

填充字符，C是控制字符。关于这些的细节可以在io模块的文档中找到。现在，让我们来看两个例子：

```
8> io:format("the sum of ~W is ~.2f.~n", [List,3,Sum]).
the sum of [2,3|...] is 8.14.
ok
9> io:format("~40p~n", [{apply, io, format, ["the
sum of ~W is ~.2f.~n", [[2,3,math:pi()],3,ioExs:sum([2,3,math:pi()])]]}]}.
{apply,io,format,
  ["the sum of ~W is ~.2f.~n",
   [[2,3,3.141592653589793],
    3,8.141592653589793]]}
ok
```

我们看到使用~p很好地美化了输出，还有最后一个命令是值得尝试的，就是在格式化字符串中使用~w取代~40p。

注意：当打印整数组成的列表时，有时候使用~p的格式化输出只会引起混乱。漂亮的打印模式试图找出你想要的打印，并进行相应的格式化。但是如果你的整数列表的值刚好是有效的ASCII值时，你将会得到一个字符串。如果你想把整数打印出来，请使用~w：

```
1> List = [72,101,108,108,111,32,87,111,114].
"Hello Wor"
2> io:format("~p~n",[List]).
"Hello Wor"
ok
3> io:format("~w~n",[List]).
[72,101,108,108,111,32,87,111,114]
ok
```

递归

最好的解决编程问题的办法是使用久经测试的策略把问题分解成一系列的小问题。把几个简单问题的解决方案联系起来，就可以解决一个意想不到的大问题。让我们来尝试这样的方法，为一个整数列表的每个元素都加1。因为Erlang是单赋值语言，我们需要创建一个新的列表用来存储结果。

我们把这个函数叫做bump/1，然后把问题分成较小的多个容易解决的任务，一次实现一个任务。如果旧列表是空的，那么新的列表也应该是空的。下面的函数语句描述了这种情况：

```
bump([]) -> [];
```

第二种可能性是，列表至少包含了一个元素。如果是这样，我们把列表分成一个头部和一个尾部。我们使用头部创建一个新的列表，它的头部是旧列表的头部加1：

```
bump([Head | Tail]) -> [Head + 1 | ?].
```

现在的问题是我们如何着手处理列表的其余部分？我们要建造一个新列表，其所有的元素值都比旧列表的元素值大1。不过，这正是bump函数需要去实现的！解决的办法是使用列表尾部来递归调用我们所定义的函数：

```
bump([Head | Tail]) -> [Head + 1 | bump(Tail)].
```

这就是我们所寻找的。我们递归调用列表尾部，确保bump/1返回一个结构良好的列表，使它成为我们刚刚创建的新列表的尾部。因此这个解决方案是：

```
bump([]) -> [];  
bump([Head | Tail]) -> [Head + 1 | bump(Tail)].
```

这个函数真的能行吗？让我们一步步完成调用bump([1,2,3])：

```
bump([1, 2, 3]) => [1 + 1 | bump([2, 3])  
  1 + 1 => 2  
  bump([2, 3]) => [2 + 1 | bump([3])  
    2 + 1 => 3  
    bump([3]) => [3 + 1 | bump([])  
      3 + 1 => 4  
      bump([]) => []  
      [4 | []] => [4]  
    [4] <=  
    [3 | [4]] => [3, 4]  
  [3, 4] <=  
  [2 | [3, 4]] => [2, 3, 4]  
[2, 3, 4] <=
```

在这个bump函数的例子中，我们揭示了两个重要的问题。第一个是解决问题比较通用的技术：把问题分解成更小的问题。这在Erlang中是一种十分常见的Erlang递归编程模式。这种“魔法”是如何实现的呢？我们利用相同的变量调用同样的函数。它们是否已经绑定了呢？不，它们还没有。需要记住的是，在每一次的函数调用循环里面变量都是新的，而且是唯一的。每一次调用一个函数，一个新的帧在调用堆栈里建立，它带有返回点的信息，传递给函数的参数和局部变量。这一点非常重要，即使在运行时系统里对你来说是隐藏的，当我们在后面的章节中谈到尾递归函数的时候会回到这一点上来。

我们现在来看一个更详细的例子，一个类似类型的问题，这实际上是一种解决办法。我们要计算一个数字列表的平均值。因此，我们命名这个函数为average。什么是平均值呢？它是元素的总和除以列表的长度。我们可以定义average如下：

```
average(List) -> sum(List) / len(List).
```

这已经解决了问题！所以我们还需要做的是定义sum和len函数。为了计算sum，我们需要做和bump类似的案例分析，把它们分解成较小的问题。

让我们先从sum函数开始。如果列表是空的，那么其元素的总和显然是零：

```
sum([]) -> 0;
```

如果这个列表包含至少一个元素，我们就把列表分解成一个头部和一个尾部，添加头部（第一部分）到尾部的总和（列表的其余部分）。正如在bump的例子中的一样，我们已经定义了一个函数来解决这个问题，现在让我们来使用它：

```
sum([Head | Tail]) -> Head + sum(Tail).
```

下一步是编写len/1函数。我们调用长度函数len避免与内置函数发生冲突。在这里我们要为每个列表中的元素添加1，因此代码几乎和sum/1相同，不过有两个小的例外：我们不关心元素的值，而仅仅是添加1，从而得到下面的用法：

```
len([_ | Tail]) -> 1 + len(Tail).
```

我们使用“不关心”变量（_），用以说明对列表的头部值不感兴趣。当然如果为空列表的话，将返回0。现在我们把它们全部放在一起（注1）：

```
average(List) -> sum(List) / len(List).
```

```
sum([]) -> 0;  
sum([Head | Tail]) -> Head + sum(Tail).
```

```
len([]) -> 0;  
len([_ | Tail]) -> 1 + len(Tail).
```

在实践中，我们会使用length/1内置函数，而不是重新定义函数，因为如果可能的话最好重用代码，这也是因为内置函数的实现效率更高，但在这里让我们看看怎么定义递归是很有帮助的。

让我们通过一个例子来仔细看看average如何工作：

```
average([1, 2, 3]) => sum([1, 2, 3]) / len([1, 2, 3])  
    sum([1, 2, 3]) => 1 + sum([2, 3])  
        sum([2, 3]) => 2 + sum([3])  
            sum([3]) => 3 + sum([])  
                sum([]) => 0  
                    3 + 0 => 3
```

注1： 当一个函数碰到一个空列表的时候最好不要出现“除0”错误，那么你会如何修改这个函数的定义，让average函数在传入一个空列表的时候返回0值呢？

```

3 <=
2 + 3 => 5
5 <=
1 + 5 => 6
6 <=
len([1, 2, 3]) => 1 + len([2, 3])
len([2, 3]) => 1 + len([3])
len([3]) => 1 + len([])
len([]) => 0
1 + 0 => 1
1 <=
1 + 1 => 2
2 <=
1 + 2 => 3
3 <=
6 / 3 => 2.0
2.0 <=

```

这个例子中最让人惊讶的地方是函数`sum/1`和`len/1`是如此的类似。这是一个演变模式，到目前为止这在Erlang代码中十分常见。我们现在继续看一个这一模式的演变的例子。我们将遍历一个列表，筛选出不是偶数的元素。我们现在有三种情况需要考虑。第一种情况，该列表是空的，这意味着没有任何内容需要检查。这种情况下将返回空列表：

```
even([]) -> [];
```

第二种情况，如果列表中的第一个元素是偶数，我们希望在构建的列表中包括它，并让列表的剩余部分也包括那些是偶数的元素：

```
even([Head | Tail]) when Head rem 2 == 0 -> [Head | even(Tail)];
```

请注意，我们使用了保护元通过查看头部除以2后的余数是否为零来确定其是否为偶数。最后，第三种情况是列表的第一个元素是奇数。如果是这样，我们通过对尾部的递归调用，而且不在前面附加第一个元素来丢弃这个元素：

```
even([_ | Tail]) -> even(Tail).
```

请注意，我们在最后的语句里没有使用保护元来确保头部不是偶数。我们已经知道了这一点，因为先前的语句选择了所有的偶数，只留下了奇数。我们不需要检查表头，因此我们使用一个“不关心”变量：

```

even([]) -> [];
even([Head | Tail]) when Head rem 2 == 0 -> [Head | even(Tail)];
even([_ | Tail]) -> even(Tail).

```

让我们仔细看看它是如何计算的：


```

even([10, 11, 12]) => [10 | even([11, 12])] (10 rem 2 == 0)
  even([11, 12]) => even([12])           (11 rem 2 == 1)
    even([12]) => [12 | even([])]         (12 rem 2 == 0)
      even([]) => []
        [12 | []] => [12]
          [12] <=
            [12] <=
              [10 | [12]] => [10, 12]
                [10, 12] <=

```

在所有这些例子中，我们遍历整个列表，或者构建新的列表或者计算值。通过这些例子我们得出这样的结论，终止递归调用的条件称为一种基本情况。我们将编写一个 `member/2` 的功能，其中给定一个元素和一个列表，并且遍历该列表，如果元素是一个列表的成员则返回 `true`，否则返回 `false`。

没有元素属于空列表的成员。因此，如果发送空列表到这个函数，我们就返回 `false`：

```
member(_, []) -> false;
```

在这一点上，我们不关心我们要找什么元素，因为它们不存在，因此我们使用“不关心”变量。我们的第二种情况是，如果这个列表包含至少一个元素，那么检查第一个元素是否是我们正在寻找的。我们把列表分解成头部和尾部，然后模式匹配头部元素。如果它们相等，我们返回 `true`：

```
member(H, [H | _]) -> true;
```

通过和一个公共变量比较来确定这个元素和头部相等，而列表的其余部分已不再重要。最后，这个列表包含至少一个元素，但由于我们没有选择前面的语句，头部显然不符合我们正在寻找的，这意味着我们必须通过递归继续寻找列表的其余部分：

```
member(H, [_ | T]) -> member(H, T).
```

把这三个语句放在一起我们可以得到：

```

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

```

使用笔和纸，一步步来实现下面的例子。请找出基本情况并理解如何通过列表的递归定义把列表分解成头部和尾部：

```

1> c(recursion).
{ok, recursion}
2> recursion:member(friday, [monday, tuesday, wednesday, thursday, friday]).
true
3> recursion:member(sunday, [monday, tuesday, wednesday, thursday, friday]).
false

```

递归不仅在Erlang中是最基本的工具之一，在函数式编程中也是如此。请复制我们详细讲解过的递归的例子到一个叫做recursion的模式中，以不同的参数测试它们，你将会从中受益。

尾递归函数

我们在前面编写了sum函数：

```
sum([]) -> 0;
sum([Head | Tail]) -> Head + sum(Tail).
```

我们使用了直接递归样式。此样式意味着你可以把函数的定义作为一个列表的总和，如“列表[2,3,4]的结果是2加上列表[3,4]的总和”。或者你可以理解它为一个等式：

$$\text{sum}([2,3,4]) = 2 + \text{sum}([3,4])$$

另一种定义sum的方法是使用一个额外的函数参数，称为累加参数，在计算时用它来保存总和。

如果函数称为sum_acc，第二个参数保存“到目前为止的总和”，那么如何定义此函数呢？第一种情况是当列表为空时，你必须返回“到目前为止的总和”：

```
sum_acc([],Sum) -> Sum;
```

另外，如果列表不是空的，你从Head拿下添加到Sum，然后在Tail上调用sum_acc和新的“到目前为止的总和”：

```
sum_acc([Head|Tail], Sum) -> sum_acc(Tail, Head+Sum).
```

你如何调用函数去计算一个列表的总和呢？你从列表和“到目前为止的总和”为0开始：

```
sum(List) -> sum_acc(List,0).
```

为了了解这是如何工作的，让我们看看下面这个例子的计算步骤（隐藏了算术计算）：

```
sum([2,3,4])
=> sum_acc([2,3,4],0)
=> sum_acc([3,4],2)
=> sum_acc([4],5)
=> sum_acc([],9)
=> 9
```

sum_acc的定义称为尾递归（tail-recursive），因为函数体是调用函数本身。只要函数f的调用发生在f的函数体的最后一个表达式（即尾部）里，这个函数f就是尾递归，那这两个sum的定义区别在哪里呢？

- 直接定义更容易理解：你把定义可以理解为是列表求和的直接表述。
- 尾递归定义更像是用C或Java编写的程序：你必须了解该程序在执行时是如何展开的，了解第二个变量的最后值实际上是这个列表的总和。另外，在某些情况下，一个尾递归定义可以更有效地使用内存。

警告：关于Erlang性能的主要传说之一是（注2）：在Erlang中尾递归要比直接递归有效得多。在该语言的初期这也许是真的，但是在对发行版本7到版本12进行了优化之后，如果还认为尾递归会让你的程序更有效率，那就不再是正确的了。

Erlang系统开发者的建议：“现在选择的标准主要是看你的习惯问题。如果你真的需要最快的速度，你必须进行性能测量。再也不能绝对确保一个尾递归函数在任何情况下是最快的。”

为什么sum_acc会比原来的sum更有效率呢？线索就是在我们较早计算的时候，你可以看到计算像一个循环一样，它看起来只是在基本情况达到的时候改变了两个参数的值。换句话说，在它的实现中可以使用较少的内存，因此更有空间效率。另外，在编译器的优化中，不是尾递归定义的函数也可以达到高效率的实现。

另一个例子是重新实现bump/1函数。对于新的bump/1，我们将增加一个累加变量，用它来保存我们所要建立的列表：正如前面所说的，这个参数看起来更像一个命令式语言的变量，因为它的值在每次迭代里都发生变化！

我们不希望更改bump/1的接口，因此我们定义一个新的辅助函数：bump_acc/2，它使用两个参数：我们原始的列表和新构造的列表。

在最初调用bump_acc/2时，我们提供了最初的列表以及一个空列表。这个空列表是我们的累加器的初始值，在第一次迭代的时候它的值是空的，因为我们还没有建立任何东西：

```
bump(L) -> bump_acc(L, []).
```

现在我们就像前面的例子一样进行情况分析，但以不同的方式构造这个函数。如果旧的列表是空的，那么我们完成了，而新建立的列表就应是结果：

```
bump_acc([], Acc) -> Acc;
```

如果旧列表包含至少一个元素，那么我们将其拆分成头部和尾部，增量增加头部并将其插入我们正在建立的列表中，使用新的累加器和列表尾部调用bump_acc/2：

```
bump_acc([H | T], Acc) -> bump_acc(T, [H + 1 | Acc]).
```

注2：可以在Erlang文档的效率指南中找到。

在继续阅读之前，请尝试复制前面的代码并进行测试或者是用纸和笔来写出程序。你注意到什么错误了吗？你遍历列表并添加新的元素到列表开头。但是，这意味着第一个添加到这个列表的元素其实在最后面，这是因为结果是一个颠倒的列表。你可以在基本情况下颠倒列表结果来纠正它。这样生成的代码将是：

```
bump(List) -> bump_acc(List, []).  
  
bump_acc([], Acc) -> reverse(Acc);  
bump_acc([Head | Tail], Acc) -> bump_acc(Tail, [Head + 1 | Acc]).
```

要了解它跟原来的版本的区别，让我们来看一个求值的例子：

```
bump([1, 2, 3])  
=> bump_acc([1, 2, 3], [])  
=> bump_acc([2, 3], [1])  
=> bump_acc([3], [2, 1])  
=> bump_acc([], [3, 2, 1])  
=> reverse([3, 2, 1])  
=> [1, 2, 3]
```

当写**bump_acc/2**时，你可以看到使用累加器颠倒列表元素。你可以使用同样的原则，什么都不做而得到一个基于累加器的**reverse**函数：

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

尾调用递归优化

回想一下，一般来说，只要函数**f**的调用发生在**f**函数体的最后一个表达式（即尾部），这个函数**f**就是尾递归的。现在让我们来优化它们。

让我们来看看前面关于**bump_acc**的计算，很明显它可以通过覆盖参数的信息来实现，这些参数作为最初的函数调用保存在堆栈帧里，然后跳到了尾部的调用函数：在这种情况下是相同的函数。这样是在没有分配一个新的堆栈帧的情况下做到的。

同样的优化对间接尾递归的函数一样可行。例如，下面这个函数通过交错两个列表（同一长度）的值来进行合并：

```
merge(Xs, Ys) ->  
  lists:reverse(mergeL(Xs, Ys, [])).  
  
mergeL([X|Xs], Ys, Zs) ->  
  mergeR(Xs, Ys, [X|Zs]);
```

```

mergel([],[],Zs) ->
    Zs.

mergeR(Xs,[Y|Ys],Zs) ->
    mergel(Xs,Ys,[Y|Zs]);
mergeR([],[],Zs) ->
    Zs.

```

尾调用优化引出了尾递归定义，它对于需要一直运行的函数非常重要：这些构成了将在第4章讨论的并发进程的主体。

两个累加器例子

现在向你展示一个更详细的尾递归函数例子，以确保你真正理解了这个概念。我们将转换sum/1和len/1函数为尾递归。把列表进行两次遍历是没有意义的，即一次计算求总，第二次得到它的长度，你只需要遍历一次。为了达到这个目的，我们将添加一个辅助函数，同时做两个计算。

这个辅助函数有两个累加器，一个用于存储总值，另一个用于存储平均值。调用辅助函数将累加器初始化为0的示例如下：

```
average(List) -> average_acc(List, 0, 0).
```

我们现在使用分而治之的方法来解决这个问题。如果列表是空的，累加器包含总和及长度，那么可以进行除法：

```
average_acc([], Sum, Length) -> Sum/Length;
```

我们的第二个例子是，如果列表中至少包含一个元素，那么我们添加元素到总和累加器、增加长度累加器并通过累加器和列表尾部递归调用辅助函数：

```
average_acc([H | T], Sum, Length) -> average_acc(T, Sum + H, Length + 1).
```

这个average的代码速度更快，使用空间更少，而且一旦你习惯了累加器，其实它比原来的版本更具有可读性。在进入下一节内容之前，使用纸和笔来自己追踪一下average函数的尾递归版本的几个例子。你还应该考虑如何修改average函数，以便当它碰到空列表的时候不会返回一个错误：

```

average(List) -> average_acc(List, 0,0).

average_acc([], Sum, Length) ->
    Sum / Length;
average_acc([H | T], Sum, Length) ->
    average_acc(T, Sum + H, Length + 1).

```

迭代和递归函数的对比

前面讲到使用递归来进行迭代，为了明确表达这个概念，将向你展示如何在Erlang中重写C语言中的迭代，请尝试比较这些概念。

我们以一个简单的C迭代函数开始，计算从1到一个作为参数传递的整数边界的总和。C函数用两个局部整型变量来表示迭代器（或索引变量）和到现在为止的总和。求和运算使用for循环，下面是一个典型的迭代结构：

```
int sum(int boundary) {
    int i, sum = 0;

    for(i = 1; i <= boundary; i++)
        sum += i;
    return sum;
}
```

Erlang的版本使用含有一个累加器的辅助函数，结果得到类似的模式。sum函数通过带有初始值调用辅助函数来模仿C的sum函数。在定义中初始化sum变量对应的值，而在for循环里面初始化i变量。

辅助函数的基本情况和for循环的退出情况相对应，并明确返回了变量sum的值。

递归调用对应于for循环体，而循环破坏性地增量sum。Erlang函数通过递归调用做同样的事情，但通过Index参数来增加参数Sum。最后，迭代器变量i的增加模仿Index参数的递归调用并增加1：

```
sum(Boundary) -> sum_acc(1, Boundary, 0).

sum_acc(Index, Boundary, Sum) when Index <= Boundary ->
    sum_acc(Index + 1, Boundary, Sum + Index);
sum_acc(_I, _B, Sum)->
    Sum.
```

运行时错误

为了在Erlang中高效率工作，你应该了解可能发生运行时错误，并了解如何在终端里输出它们。Erlang运行时错误是由系统抛出的异常。在如下的列表例子中，我们会使用一个函数来生成运行时错误，然后可以看到在终端中调用这个函数的实际输出。

在这个例子中，我们使用一个test模块导出factorial/1、test1/1和test2/1函数。在最后的一些情形中，我们可以直接在终端中输出运行时错误。对于每一次运行时错误，我们都给出了它的简要原因说明：

function_clause

当已存在的函数模式无一匹配该调用函数的时候就会返回它。此错误通常发生在以下两种情况中：要么在条件分析中你忘记了一个条件；要么无意间使用了错误的参数调用函数。

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

1> test:factorial(-1).
** exception error: no function clause matching test:factorial(-1)
```

case_clause

当在case结构里没有与现有的模式相匹配时就会返回它。这方面最常见的原因是你忘记了一个或多个可能的情况。

```
test1(N) ->
    case N of
        -1 -> false;
        1 -> true
    end.
1> test:test1(0).
** exception error: no case clause matching 0
    in function test:test1/1
```

if_clause

当在if结构中没有现有的表达式求值是true的时候就会返回它。这是一个简化了的case结构，该错误通常是因为缺少一个模式。

```
test2(N) ->
    if
        N < 0 -> false;
        N > 0 -> true
    end.

1> test:test2(0).
** exception error: no true branch found when evaluating an if expression
    in function foo:test2/1
```

badmatch

错误发生的情况是模式匹配失败，也没有其他可供选择的语句。对于badmatch异常，很难找到单一的原因，但经常性的原因是你无意间尝试绑定已绑定过的变量，例如：

```
1> N=45.
45
2> {N,M}={23,45}.
** exception error: no match of right hand side value {23,45}
```

无法绑定23到N，因为N已绑定到45了。

另一种比较常见的原因是，你从一个函数调用中匹配获取部分结果。例如在一个元组列表中，使用库函数lists:keysearch/3搜索元组比较常见，成功时返回元组{value, Tuple}，其中Tuple就是要搜索的元组。这个函数现在通过下面这个方式调用：

```
1> lists:keysearch(1,1,[{1,2},{2,4}]).
{value,{1,2}}
2> lists:keysearch(3,1,[{1,2},{2,4}]).
false
3> {value, Tuple} = lists:keysearch(3,1,[{1,2},{2,4}]).
** exception error: no match of right hand side value false
```

因为我们希望立即使用检索到的元组。但是当没有发现带有匹配关键字的元组的时候，该函数返回false，从而导致出现badmatch：

```
1> Tuple = {1, two, 3}.
{1,two,3}
2> {1, two, 3, Four} = Tuple.
** exception error: no match of right hand side value {1,two,3}
```

badarg

当调用内置函数的时候若使用错误参数会返回它。在下面的例子中，length需要一个列表，但是调用时使用了一个基元。

```
1> length(helloWorld).
** exception error: bad argument
   in function length/1
   called as length(helloWorld)
```

undef

当没有定义或者导出的全局函数被调用时就会返回它。发生这种异常的原因往往是错误拼写函数名称，或者调用函数时在函数调用前面没有加上模块名称。

```
1> test:hello().
** exception error: undefined function test:hello/0
```

badarith

当算术运算时使用不适当的参数就会返回它，如非整数、浮点数或者试图除以0。

```
1> 1+a.
** exception error: bad argument in an arithmetic expression
   in operator +/2
   called as 1 + a
```

更多的错误类型，我们将在本书后面相关的内容里继续讨论。

处理错误

在前面部分已经介绍了在Erlang系统中一些可能发生的错误，以及对潜在发生原因的诊断。

当执行表达式时发生了运行时错误，你可能想捕获异常并且防止终止执行线程。此外，你可能想使其失效，然后让系统的另外部分来处理恢复：后一种涉及进程相连和监控的选择，这些我们将在第6、12章具体讨论。在本节中我们会讨论如何使用try...catch结构来捕获和处理错误。

使用try ...catch

try...catch结构后面隐藏的思想是对一个表达式求值，并提供处理表达式的正常结果以及异常终止的方法。更重要的是，这个结构允许你区分由Erlang的不同的异常结果处理机制抛出的不同的返回值，以及以不同的方式来处理它们。

在表达式求值之前，你插入保留字try。在case语句中你模式匹配（正常）结果，但不是以end马上终止语句，取而代之的是使用catch语句去处理异常。这些语句包括其头部的一个异常类型（也称为类）和异常模式，还有相应的返回表达式。

try...catch结构具有以下形式：

```
try Exprs of
  Pattern1 [when Guard1] ->
    ExpressionBody1;
  Pattern2 [when Guard2] ->
    ExpressionBody2
catch
  [Class1:]ExceptionPattern1
    [when ExceptionGuardSeq1] ->
      ExceptionBody1;
  [ClassN:]ExceptionPatternN
    [when ExceptionGuardSeqN] ->
      ExceptionBodyN
end
```

下面是这种形式的一个实例，在Erlang终端中通过一系列的命令来显示。在第一个命令中，X和2绑定，因此随后的任何尝试绑定X至3都将导致失败并产生badmatch错误：

```
1> X=2.
2
2> try (X=3) of
2>   Val -> {normal, Val}
2> catch
2>   _:_ -> 43
2> end.
43
```

在第二个命令中，所有类（匹配_:_）中的所有错误模式都映射到43，在这里返回了这个结果，这是因为只有error类（匹配error:_）模式是与第三个命令相匹配的：

```
3> try (X=3) of
3>   Val -> {normal, Val}
```

```
3> catch
3>   error:_ -> 43
3> end.
43
```

在第四个命令中，错误类型作为结果的一部分返回，并且我们可以看到这确实是一个表达式3（与X）的badmatch：

```
4> try (X=3) of
4>   Val -> {normal, Val}
4> catch
4>   error:Error -> {error,Error}
4> end.
{error,{badmatch,3}}
```

最后，throw语句允许我们在try...catch声明中抛出一个非正常的返回：

```
5> try (throw(non_normal_return)) of
5>   Val -> {normal, Val}
5> catch
5>   throw:Error -> {throw, Error}
5> end.
{throw,non_normal_return}
```

在try...catch中，你可以使用throw/1内置函数来调用一个非本地返回。绕过调用栈，try...catch表达式返回传给throw表达式的返回值。

想象一下，你需要解析一个非常大且深层嵌套的XML结构。在正常情况下，我们只需要处理正确的结构解析，而不需要检查每个递归的返回值是否有错误，这样就可以集中精力解析返回的结构。如果你碰到一个分析错误，函数就会抛出一个异常。try...catch会捕获这个异常，它可以绕过整个递归调用堆栈而成为表达式的返回值。

你应该避免使用throw，因为作为非本地的返回它会使你的代码很难追踪和调试。对这一指导原则的唯一例外是较早的解析器例子，其中在深层嵌套结构中，要是遇到错误的情况，应该使用throw退出。如果你真的必须使用catch和throw，为了对那些未来可能维护你的代码的同事表示同情，请确保在同一个模块中同时调用它们俩。这是因为试图找出在哪个模块中的哪些catch处理一个其他地方所定义的throw是一件很麻烦的事情。

那么有哪些错误类呢？

error

这是错误的主要种类，你已经在前面的内容中看到了不同类型的运行时错误，error也可以通过调用内置函数erlang:error(Term)来触发。

throw

这个是由显式调用throw抛出一个异常产生的类，try...catch表达式会捕获它。在Erlang中不推荐使用throw，因为它让理解程序的难度大大增加。

exit

这可以通过调用exit/1内置函数来触发，它包括一个终止的原因，exits也可以通过一个退出信号产生，我们将在第6章中详细介绍。

让我们使用函数return_error来看看它们的运行情况，定义如下：

```
-module(exception).
-export([return/1]).

return_error(X) when X < 0 ->
    throw({'EXIT', {badarith,
                    [{exception,return_error,1},
                     {erl_eval,do_apply,5},
                     {shell,exprs,6},
                     {shell,eval_exprs,6},
                     {shell,eval_loop,3}]}}});
return_error(X) when X == 0 ->
    1/X;
return_error(X) when X > 0 ->
    {'EXIT', {badarith, [{exception,return_error,1},
                        {erl_eval,do_apply,5},
                        {shell,exprs,6},
                        {shell,eval_exprs,6},
                        {shell,eval_loop,3}]}}).
```

取决于函数的参数是正、负或零，这个函数会有三种不同类型的行为。让我们定义try_return去捕获错误：

```
try_return(X) when is_integer(X) ->
    try return_error(X) of
        Val -> {normal, Val}
    catch
        exit:Reason -> {exit, Reason};
        throw:Throw -> {throw, Throw};
        error:Error -> {error, Error}
    end.

4> exception:try_return(1).
{normal,{'EXIT',{badarith,[{exception,return_error,1},
                           {erl_eval,do_apply,5},
                           {shell,exprs,6},
                           {shell,eval_exprs,6},
                           {shell,eval_loop,3}]}}}

5> exception:try_return(0).
{error,badarith}

6> exception:try_return(-1).
{throw,{'EXIT',{badarith,[{exception,return_error,1},
                           {erl_eval,do_apply,5},
```

```
{shell,exprs,6},
{shell,eval_exprs,6},
{shell,eval_loop,3}}}]}
```

在try...catch中你可以使用通配符，如果你不是在对返回值模式匹配，则可以省略of。输入下面的例子然后尝试使用不同的异常和通配符的匹配模式。不要忘记导出函数，但是请忽略最后两个catch语句产生的警告。把这些语句包括进来是用来示范语法并允许你进行尝试，但不会执行它们，因为所有的异常都在前面的情况中处理了：

```
try_wildcard(X) when is_integer(X) ->
    try return_error(X)
    catch
        throw:Throw -> {throw, Throw};
        error:_ -> error;
        Type:Error -> {Type, Error};
        _ -> other; %% Will never be returned
        _:_ -> other %% Will never be returned
    end.

7> exception:try_wildcard(-1).
{throw,{'EXIT',{badarith,[{exception,return_error,1},
                           {erl_eval,do_apply,5},
                           {shell,exprs,6},
                           {shell,eval_exprs,6},
                           {shell,eval_loop,3}}]}}}

8> exception:try_wildcard(0).
error
9> exception:try_wildcard(1).
{'EXIT',{badarith,[{exception,return_error,1},
                   {erl_eval,do_apply,5},
                   {shell,exprs,6},
                   {shell,eval_exprs,6},
                   {shell,eval_loop,3}}]}}
```

在离开这个例子之前，不要认为所有你用捕获错误可以做的就是以某种形式去传递给他们。相反，返回一些值来表明没有发生错误是可能的；这点在一个try_return的最终版本里说明：

```
try_return(X) when is_integer(X) ->
    try return_error(X) of
        Val -> {normal, Val}
    catch
        exit:_ -> 34;
        throw:_ -> 99;
        error:_ -> 678
    end.
```

请尝试这方面的例子，你将会看到没有捕获正值'EXIT'（我们会在第6章再讨论这种情况）。

使用catch

Erlang中最初的异常处理机制是catch。由于它有一些特别的行为，一个在乌普萨拉大学的高性能Erlang的团队成員Richard Carlson建议对Erlang的异常处理进行审查，结果最后引入了try...catch表达式。他得到了OTP团队的认可，他们把这个结构作为有文档说明的永久特性包括在Erlang R10B发布版本中。我们在这里讨论catch是因为大量遗留代码中使用到了这个结构。但是需要注意的是，它并不像try...catch那么优雅。

catch表达式允许你捕获出现的运行时错误。其格式就是“catch 表达式”，如果表达式计算结果正确就返回表达式的值。但是，如果一个运行时错误发生，它就返回元组{'EXIT', Error}，其中Error包含了运行时的错误信息。

通过数字字符串而不是数字进行调用，让我们尝试使用内置函数list_to_integer/1来捕获产生的异常：

```
1> list_to_integer("one").
** exception error: bad argument
   in function list_to_integer/1
      called as list_to_integer("one")
2> catch list_to_integer("one").
{'EXIT',{badarg,[{erlang,list_to_integer,[{"one"}],
                  {erl_eval,do_apply,5},
                  {erl_eval,expr,5},
                  {shell,exprs,6},
                  {shell,eval_exprs,6},
                  {shell,eval_loop,3}]}}}
```

看看这两个调用的结果。第一个调用我们得到了一个在终端中打印出来的运行时错误。如果这种情况发生在你的代码中，程序将会异常终止。第二个调用我们在catch区域内执行表达式。其结果是没有生成一个运行时错误，而返回一个格式为{'EXIT', {Reason, Stack}}的元组。Reason是描述错误类型的基元，在这里是一个badarg，而Stack是函数调用堆栈，它可以让你定位在哪使用错误参数调用了内置函数。

正如下面的例子所示，在Erlang中优先级有时候会有违直觉的。如果你在catch中绑定一个表达式的返回值，你需要用括号封装catch表达式，给它比赋值更高的优先级。如果你不这样做，编译器会返回一个语法错误：

```
3> catch 1/0.
{'EXIT',{badarith,[{erlang,'/',[1,0]},
                    {erl_eval,do_apply,5},
                    {erl_eval,expr,5},
                    {shell,exprs,6},
                    {shell,eval_exprs,6},
                    {shell,eval_loop,3}]}}}
```

```
4> X = catch 1/0.
* 1: syntax error before: 'catch'
```

```

4> X = (catch 1/0).
{'EXIT',{badarith,[{erlang,'/',[1,0]},
                    {erl_eval,do_apply,5},
                    {erl_eval,expr,5},
                    {erl_eval,expr,5},
                    {shell,exprs,6},
                    {shell,eval_exprs,6},
                    {shell,eval_loop,3}]}}}

5> X.
{'EXIT',{badarith,[{erlang,'/',[1,0]},
                    {erl_eval,do_apply,5},
                    {erl_eval,expr,5},
                    {erl_eval,expr,5},
                    {shell,exprs,6},
                    {shell,eval_exprs,6},
                    {shell,eval_loop,3}]}}}

```

内置函数throw/1可以跟catch一起使用。让我们看看下面的例子：

```

-module(math).
-export([add/2]).

add(X,Y) ->
    test_int(X),
    test_int(Y),
    X + Y.

test_int(Int) when is_integer(Int) -> true;
test_int(Int) -> throw({error, {non_integer, Int}})

```

让我们尝试在Erlang终端中与这个模块互动：

```

1> math:add(1,1).
2
2> math:add(one, 1).
** exception throw: {error,{non_integer,one}}
   in function math:test_int/1
3> catch math:add(one, 1).
{error,{non_integer,one}}

```

在catch范围内调用函数add/2结果得到元组{error,Reason}。而在catch范围外调用相同的函数就会得到一个运行时错误。

catch的另一个问题是，它不区分运行时错误的语义，不管是抛出、退出或者是一个函数的返回值，都同样对待它。它没有办法确定{'EXIT', Error}是如何返回得到的，它有可能是在catch中封装的调用执行时发生错误抛出{'EXIT', Error}；或者是一个运行时错误的结果；或者是调用内置函数exit/1的结果；或者仅仅是一个表达式返回了元组{'EXIT', Error}。

通过例子中返回的先前定义的return-error，这一点得到了最好的体现：

```
-module(exception).
-export([return/1]).

return(X) when is_integer(X) ->
    catch return_error(X).
```

和以下这些互动，你可以看到：

```
1> exception:return(-1).
{'EXIT',{badarith,[{exception,return_error,1},
                    {erl_eval,do_apply,5},
                    {shell,exprs,6},
                    {shell,eval_exprs,6},
                    {shell,eval_loop,3}]}}
```

```
2> exception:return(0).
{'EXIT',{badarith,[{exception,return_error,1},
                    {erl_eval,do_apply,5},
                    {shell,exprs,6},
                    {shell,eval_exprs,6},
                    {shell,eval_loop,3}]}}
```

```
3> exception:return(1).
{'EXIT',{badarith,[{exception,return_error,1},
                    {erl_eval,do_apply,5},
                    {shell,exprs,6},
                    {shell,eval_exprs,6},
                    {shell,eval_loop,3}]}}
```

你以为你已经明白一切了吗？请尝试在Erlang终端中输入：

```
catch exit({badarith, [{exception, return_error, 1}, {erl_eval, do_apply, 5},
                       {shell, exprs, 6}, {shell, eval_exprs, 6},
                       {shell, eval_loop, 3}]})
```

你会获得与例子中第四个语法相同但语义不同错误的方法。

Erlang程序员没有必要过多困扰于此，十多年来避免使用throw和不返回含有基元'EXIT'的元组，他们也能工作。尽管程序员可能没有太多困扰，但是try...catch表达式的提议解决了新程序的这些问题——能够和Erlang R10B完成的遗留代码一块工作。

模块库

大量的模块库与Erlang运行时系统一起发布。这些值得我们花一些时间来看看什么是可用的，从而可以对那些某些功能可能非常有用的模块有个大体的了解。对于不同的发布版本，我们有必要阅读它们的发布说明，其中包含了你应该知道库的主要改变，并指明了提供的新模块。如果对于你的问题找不到一个提供通用解决方案的模块，你应该尝试到开源社区里面寻找，因为很有可能已经有人在友好开源许可证下编写发表了你所需要

的库。本节我们讨论Erlang系统中最常用的库。但在看这些库之前，让我们来了解如何访问这些相关的文件。

文档

Erlang的发布版本带有文档，它同时有HTML和Unix手册页格式。文档往往是和Erlang发行版本捆绑在一起的，但也可以从<http://erlang.org>上单独下载或者在线访问（见图3-1）。你可以通过访问`file:///<erl_root_dir>/doc/index.html`来打开HTML文档的主页面，其中的`doc`子目录位于Erlang安装的根目录下。在Windows下，打开文档的快捷方式包含在Program Files菜单下Erlang/OTP的安装目录里。

如果你使用的是基于Unix的系统，“erl-man”命令是一种方便访问手册的方式。如果该命令不能工作，那么表明手册还没有安装好。最后，大多数拥有Erlang模式的编辑器都应该有直接方便查询手册的方法。

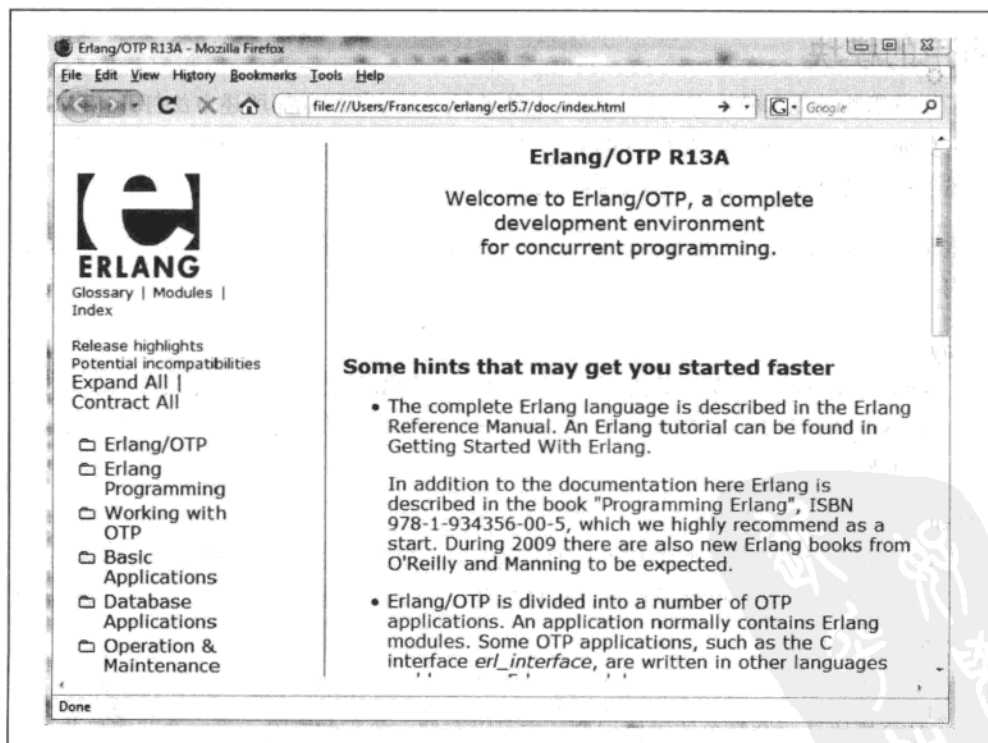


图3-1：Erlang在线文档的主页

你可以通过页面左边菜单来访问文档。在上方有以下链接：

Glossary

在Erlang中常用的术语清单。

Modules

Erlang发布版本中包含的按字母顺序排列的模块清单，每个模块都带有网络文档。这包括erlang和erl模块。许多模块有合适的描述性标题，在浏览器中对这个页面的搜索会让你寻找到所要的功能页面。

Index

一个对Erlang/OTP函数和命令的排列索引。浏览器搜索在这里再次派上用场，一旦你找到感兴趣的东西，它会提供给你相关模块的链接。

在左边其余的链接能展开为子菜单，它提供了系统各方面的信息。例如，浏览这些会给你带来关于工具的信息，以及关于Erlang/OTP构架方面的较高层的描述和如何开始熟悉系统。

有用的模块

为了知道哪些功能和模块在Erlang版本中是可用的，请按照前面所述方法打开HTML文档主页面。在左上方你应该可以找到一个列出了所有现存模块的页面的链接，那些文档描述了每个模块导出函数，以及它们的类型和相关类型定义。

那些最重要的模块在这里列出。请花一些时间浏览关于它们的手册并在终端中尝试它们：

array

array模块包含一个函数式的可扩展数组的抽象数据类型。它们可以有一个固定的大小或根据需要将其变大。这个模块包含设置和检查值，以及基于这些来定义递归的函数。

calendar

calendar模块提供函数来检索本地和全球时间以及提供星期、日期和时间转换。可以计算时间间隔，时间单位可以从日到微秒。calendar模块是基于公历和内置函数now/0的。

dict

dict模块是一个简单的键值词典，它可以存储、检索和删除元素，合并字典并遍历它们。

erlang

所有内置函数都认为在erlang模块中实现。关于这个模块的手册页列出了所有的Erlang内置函数，它在不同的虚拟机里是不一样的，因此有些虚拟机自动导入而有些则不是。

file

file模块提供了一个文件系统的接口，从而能够进行读、操作和删除文件。

filename

filename模块允许你写通用的文件操作和检查函数，而不用管其在底层操作系统中的文件符号表示。

io

io库模块封装了标准I/O服务的接口函数，它允许你读取和写入字符串到I/O设备，当然也包括标准输出。

lists

lists列表操作模块毫无疑问是在所有主要的Erlang系统中使用最多的库模块。它提供了检查、操作和处理列表的函数。

math

所有标准的数学函数，包括pi/0、sin/1、cos/1和tan/1都在math库模块中实现。

queue

queue模块为FIFO队列实现了一个抽象数据类型。

random

random模块基于提供的一个种子给出了一个伪随机数生成器。

string

string模块包含了一个字符串处理函数的数组。它和列表模块不同，它会考虑列表的内容实际上是ASCII字符这个事实。

timer

timer模块包含了时间相关的函数，它包括产生事件和转换不同的时间格式为毫秒，毫秒是在这个模块中主要使用的时间单位。

调试器

在Erlang中调试器是一个图形化工具，它提供调试有序代码和改变程序执行的机制。它允许用户一步一步运行程序的同时查看和操作变量。你可以设置断点停止执行以及检查递归栈和各个层次的变量绑定。我们这里只对调试器做一个简要的介绍，至少让你有足

够知识开始使用它。在此没有涵盖它的许多特性和细节，但所有这些你都可以在网上提供的调试器用户指南文档中找到。

请输入`debugger:start()`来启动调试器，之后会出现一个监视器窗口。这个窗口显示包含追踪编译模块、附加（追踪）进程和其他调试相关设置的一个列表（见图3-2）。

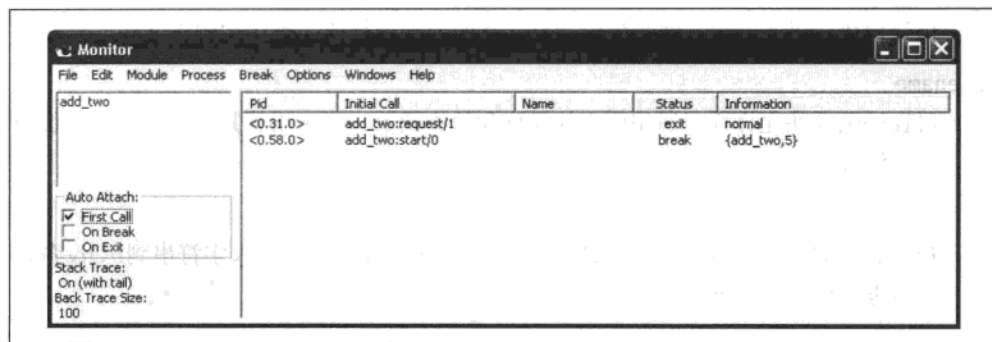


图3-2：监视器窗口

要追踪一个模块，你首先需要使用`debug_info`标志来编译它。在你的Unix终端中，你可以使用如下命令来执行：

```
erlc +debug_info Module.erl
```

在Erlang终端中，使用以下两个命令中的一个：

```
c(Module, [debug_info]).
```

```
compile:file(exception, [debug_info]).
```

然后通过打开模块菜单里的解释对话框来选择调试器里的模块。跟踪编译的模块和没有跟踪编译的模块（后者将在括号内显示）将一起在这个窗口上列出。单击你想追踪的模块，然后将在它的旁边出现一个*。只要在追踪模块下开始执行一个进程，在监视器窗口上将显示一个条目。你可以双击它，这将打开我们所说的附加窗口。此窗口（见图3-3）允许你一步一步执行代码、查看和处理变量，以及检查递归栈。另外打开附加窗口的一个方法是预先在跟踪窗口中选择附加选项。这些选择包括在代码中设置断点、运行解释模块和退出，或者只在第一次执行时调用解释模块。

你可以在break菜单中选择一个合适的菜单项或者通过单击监视器中的行或者模块窗口的行来设置断点。只有在可执行表达式上才可以设置断点，因此设置在函数头部、模式或者语句分隔符上的断点没有任何效果。断点可以有一个有效或无效的状态。当达到一个有效断点的时候，断点可以通过触发器删除、停用或者保持有效。

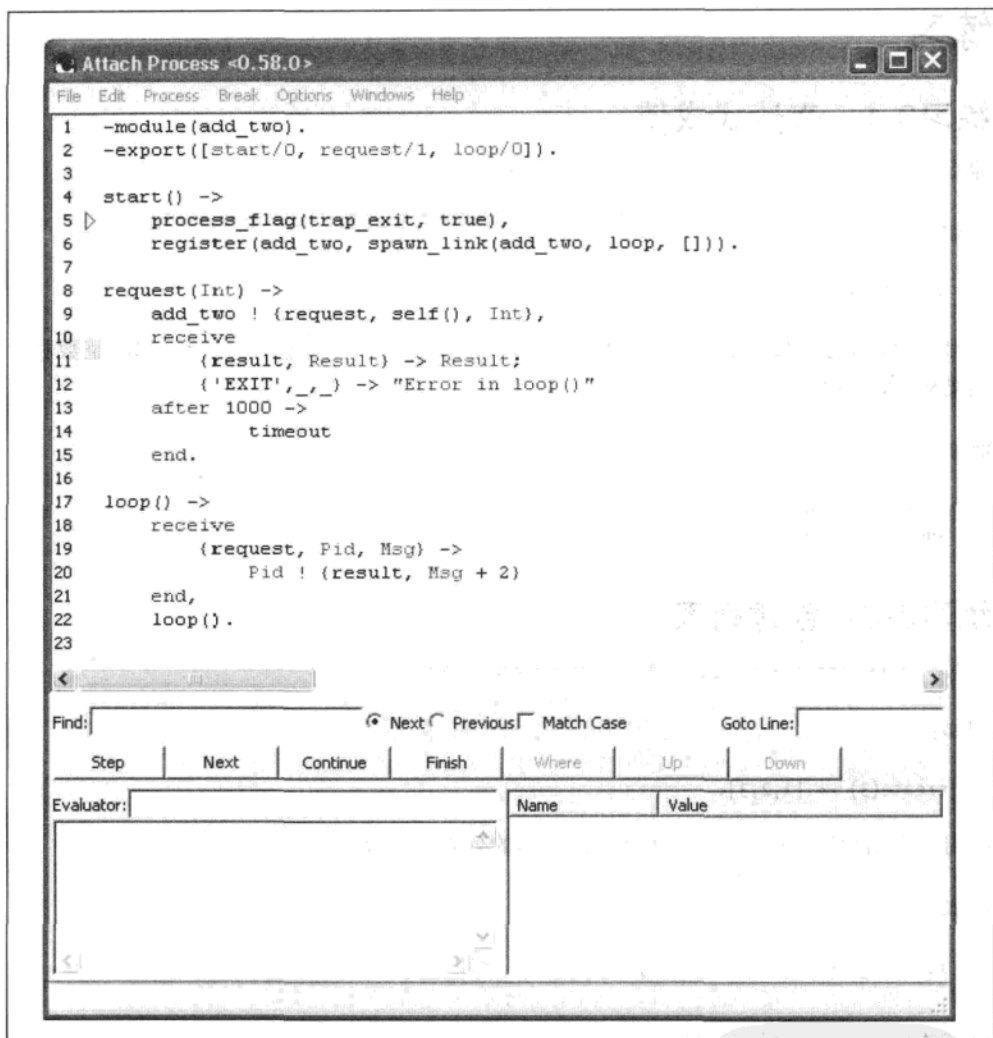


图3-3: 附加窗口

既然我们完成了顺序编程，是时候继续学习并发编程了。但在这样做之前，一定要掌握好关于递归的关键概念及其各种模式，因为只有当我们停止这一进程的时候，基于尾递归函数的并发才停止迭代。另外，请花一些时间阅读关于库模块的文档，要知道哪些是可用的。

下面的练习可以让你熟悉递归和它的各种不同用法。请特别注意我们在本章中涉及的各种递归模式的区别。如果你不能发现错误或者跟踪递归，请尝试使用调试器。

练习

练习3-1：表达式求值

编写一个函数`sum/1`，给定一个正整数`N`，其返回的是 $1 \sim N$ 的整数和。

例如：

```
sum(5) ⇒ 15.
```

编写一个`sum/2`函数，给定两个整数`N`和`M`，其中 $N \leq M$ ，其返回的是 $N \sim M$ 的整数和。

如果 $N > M$ ，进程异常终止。

例如：

```
sum(1,3) ⇒ 6.  
sum(6,6) ⇒ 6.
```

练习3-2：创建列表

编写一个返回格式为 $[1, 2, \dots, N-1, N]$ 的列表的函数。

例如：

```
create(3) ⇒ [1,2,3].
```

编写一个函数返回格式为 $[N, N-1, \dots, 2, 1]$ 的列表的函数。

例如：

```
reverse_create(3) ⇒ [3,2,1].
```

练习3-3：边界效应

编写一个打印出 $1 \sim N$ 的整数的函数。

提示：使用`io:format("Number:~p~n",[N])`。

编写一个打印出 $1 \sim N$ 的偶数的函数。

提示：使用保护元。

练习3-4：使用列表处理数据库

编写创建一个数据库的模块`db.erl`，它能够存储、检索和删除元素。`destroy/1`函数删

除数据库。考虑到Erlang有垃圾收集器，你不需要做任何事情。然而如果db模块把一切存储在文件中，那么你也应该删除这个文件。我们把destroy函数包括进来是为了接口的完整性。如果你不使用lists库模块，那么你就必须自己实现所有的递归函数。

提示：使用列表和元组作为你的主要数据结构。当测试程序的时候，请记住Erlang中变量是单次赋值的：

接口：

| | |
|----------------------------|--------------------------------------|
| db:new() | ⇒ Db. |
| db:destroy(Db) | ⇒ ok. |
| db:write(Key, Element, Db) | ⇒ NewDb. |
| db:delete(Key, Db) | ⇒ NewDb. |
| db:read(Key, Db) | ⇒ {ok, Element} {error, instance}. |
| db:match(Element, Db) | ⇒ [Key1, ..., KeyN]. |

例如：

```
1> c(db).
{ok,db}
2> Db = db:new().
[]
3> Db1 = db:write(francesco, london, Db).
[{francesco,london}]
4> Db2 = db:write(lelle, stockholm, Db1).
[{lelle,stockholm},{francesco,london}]
5> db:read(francesco, Db2).
{ok,london}
6> Db3 = db:write(joern, stockholm, Db2).
[{joern,stockholm},{lelle,stockholm},{francesco,london}]
7> db:read(ola, Db3).
{error,instance}
8> db:match(stockholm, Db3).
[joern,lelle]
9> Db4 = db:delete(lelle, Db3).
[{joern,stockholm},{francesco,london}]
10> db:match(stockholm, Db4).
[joern]
11>
```

注意：由于在Erlang中是单赋值，你需要每次给更新的数据库赋予新的变量。在终端中使用f()可以去除现有的变量绑定。

练习3-5：操作列表

编写一个函数，给定一个整数和一个整数列表，并且返回所有小于或等于该整数的整数。

例如：

```
filter([1,2,3,4,5], 3) ⇒ [1,2,3].
```

编写一个函数，给定一个列表，颠倒其中元素的顺序进行排列。

例如：

```
reverse([1,2,3]) ⇒ [3,2,1].
```

编写一个函数，给定一个列表的列表，将它们连接起来。

例如：

```
concatenate([[1,2,3], [], [4, five]]) ⇒ [1,2,3,4,five].
```

提示：你需要使用一个帮助函数并且通过多个步骤来串联列表。

编写一个函数，给定一个嵌套列表的列表，返回一个拉平的列表。

例如：

```
flatten([[1,[2,[3],[ ]]], [[4]], [5,6]]) ⇒ [1,2,3,4,5,6].
```

提示：使用concatenate来解决flatten。

练习3-6：列表排序

在列表上实现下列排序算法。

快速排序

我们使用列表的头部作为一个中轴，根据元素小于中轴或是其他的情况，列表被分割。这2个列表再分别递归调用quicksort，然后把它们和它们之间的中轴合并到一起。

合并排序

把列表分为两个（几乎）相同长度的列表。然后对它们分别进行排序，然后将其结果有序地合并到一起。

练习3-7：使用库模块

使用lists库函数实现练习3-4的数据库处理列表。保持相同的db模块接口，让你的这两个模块可以互换。你的解决方案会减少多少代码呢？

练习3-8：表达式求值和编译

这个练习要求你建立一个操作算术表达式的函数集合。以如下的表达式开始：

```
((2+3)-4) 4 ~((2*3)+(3*4))
```

这完全置于括号内，并使用波形符(~)来表示一元运算符负号。

首先，编写一个解析器，使得它们转换成Erlang的表达形式，比如：

```
{minus, {plus, {num, 2}, {num, 3}}, {num, 4}}
```

它表示了((2+3)-4)。我们叫它*exprs*。现在，编写一些函数：

- *evaluator*，它接受一个exp然后返回它的值，
- *pretty printer*，它可以把exp转换成一个字符串表示，
- *compiler*，它转换一个exp为堆栈机的系列代码，用以对exp求值，
- *simulator*，它为堆栈机实现表达式，
- *simplifier*，这将简化表达式，使0*e转化为0，1*e为e等。（有很多其他的想法！）

你还可以添加条件来扩展表达式的集合：

```
if ((2+3)-4) then 4 else ~((2*3)+(3*4))
```

如果“if”表达式的值为0，则返回“then”值，否则返回“else”值。

你还可以添加诸如以下的局部定义：

```
let c = ((2+3)-4) in ~((2*c)+(3*4))
```

或者，你可以添加变量，接着赋值，这样就可以在随后的表达式中使用。

对于所有的这些扩展，你需要考虑如何修改你已经编写过的函数。

练习3-9：索引

原始文档是一个行的列表（即字符串），而文档是一个单词的列表。编写一个函数读入文本到原始文档，然后再到文档。

编写一个文件的索引。它得到一个关于单词及其出现频率的列表，因此单词Erlang可能有以下项：

```
{ "Erlang", [1,1,2,4,5,6,6,98,100,102,102] }
```


编写一个函数，把它以一种可读的形式打印出来，例如：

```
"Erlang      1-2,4-6,98,100,102"
```

这样可以删除重复元素而把相邻的数字放到一个范围内。你可以通过一个函数来实现，通过一系列转换，先前的出现频率列表就可能如下所示：

```
[[{1,2},{4,6},{98,98},{100,100},{102,102}]]
```

练习3-10：文本处理

编写一个函数，它会使用未结构化的文本，如下所示：

```
Write a function that will print this in a readable form,
so that duplicates are removed and adjacent numbers are put into a
range. You might like to think of doing this via a function which turns
the earlier list of occurrences into a list like
[[{1,2},{4,6},{98,98},{100,100},{102,102}]]
through a sequence of transformations.
```

并将其转换为填补文本，比如：

```
Write a function that will print this
in a readable form, so that duplicates
are removed and adjacent numbers are put
into a range. You might like to think of
doing this via a function which turns
the earlier list of occurrences into a
list like
[[{1,2},{4,6},{98,98},{100,100},{102,102}]]
through a sequence of transformations.
```

当你按下下一个空白行时，停止填充。一个更需要技巧的练习是对齐如下的文本：

```
Write a function that will print this
in a readable form, so that duplicates
are removed and adjacent numbers are put
into a range. You might like to think of
doing this via a function which turns
the earlier list of occurrences into a
list like
[[{1,2},{4,6},{98,98},{100,100},{102,102}]]
through a sequence of transformations.
```

你不需要对最后一行进行对齐。



并发编程

并发是指多个不同的函数能够并行运行而不相互影响，除非程序员明显地想这么做。每个Erlang中的并发活动称为一个进程（process）。进程之间相互交流的唯一途径是通过消息传递，数据以这种方式从一个进程发送到另一个进程。Joe Armstrong总结的几条原则对Erlang及其并发模型的设计哲学作出了最好的阐述：

- 世界是并发的。
- 事物之间不共享数据。
- 事物通过消息进行通信。
- 事物会出现故障。

并发模型及其错误处理机制从一开始就内置于Erlang中。使用轻量级进程，几十万甚至上百万进程同时并行运行并不少见，而且经常仅仅使用很少的内存。运行时系统把并发性扩展到如此的级别直接影响到了程序的开发方式，这是Erlang语言区别于其他的并发编程语言的地方。

如果你要使用Erlang来编写一个即时消息服务器，就像Facebook那样，在系统中需要支持成千上万的用户进行消息传递，那么你应该怎么做呢？Erlang的设计哲学就是为每个事件生成一个新进程，这样，程序的结构就直接反映出了多用户交换消息的并发性。在即时消息服务器系统中，一个事件可能是一个位置更新，消息的发送或者消息的接收，或者是一个登录请求。每个进程服务于它处理的事件，并且当请求完成时终止。

使用C或Java你同样可以做到这些，但是当系统扩展到成千上万的并发事件规模时，你就会很麻烦。一个可能的选择是，生成一个处理特定事件或者特定用户的进程池，但肯定不是为每个事件生成一个新进程。Erlang没有使用这种方式，因为它不使用本地线程来表示进程。它在虚拟机中有自己的调度程序，在尽量减少内存占用的同时使进程创建非常高效率地进行。无论系统中并发进程有多少，都可以保证这种效率。同样的道理适

用于消息传递，不管进程的数量是多少，信息发送的时间都是可以忽略不计的并且是个常量。本章我们将介绍Erlang的并发编程，让你加入当今最强大的并发模型之一中来。

创建进程

到目前为止，我们已经研究了单一进程中的串行代码的执行。为了运行并发代码，必须创建多个进程。通过使用内置函数`spawn(Module, Function, Arguments)`可以生成一个新进程，并对Module模块中的导出函数Function以列表Arguments作为参数进行求值。内置函数`spawn/3`会返回一个进程标识符（process identifier），从现在起，我们简称它为`pid`。

在图4-1中，我们称之为`Pid1`的进程在程序的某处调用内置函数`spawn`。这个调用的结果是生成了一个进程标识符为`Pid2`的新进程。进程标识符`Pid2`作为调用`spawn`的结果返回，它通常绑定到如下格式的一个表达式变量中：

```
Pid2 = spawn(Module, Function, Arguments).
```

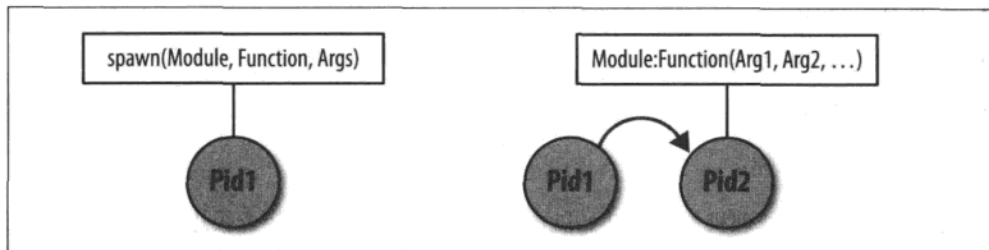


图4-1：调用spawn前和调用spawn后

此时只有在进程`Pid1`内部知道新进程的进程标识符`Pid2`，因为它是一个局部变量且没有和任何进程共享。这个生成的进程开始执行作为第二个参数传递给内置函数的导出函数，而这个函数的元数是由作为第三个参数传递给内置函数的列表长度决定的。

警告： 当在Erlang中开始编程的时候，一个常见的错误是忘记了`spawn`的第三个参数是一个参数列表，因此当使用参数`a`来对函数`m:f/1`生成一个进程的时候，应该调用：

```
spawn(m, f, [a])
```

而不是：

```
spawn(m, f, a).
```

一个进程一旦生成，它会持续执行和有效，直至终止它。如果一个进程没有更多的代码

可以执行，我们就说这个进程是正常终止的。反之如果发生一个运行时错误，比如错误匹配或者case语句错误，我们就说这个进程是异常终止的。

生成一个进程永远不会失败，即使使用了一个没有导出的或者甚至是不存在的函数。一旦这一进程生成了，spawn/3就返回进程标识符，新生成的进程将由于运行时错误而终止：

```
1> spawn(no_module, nonexistent_function, []).
<0.32.0>

=ERROR REPORT==== 29-Feb-2008::21:48:29 ===
Error in process <0.32.0> with exit value:
  {undef, [{no_module, nonexistent_function, []}]}
```

在前面的例子中，请注意错误报告是经过格式化的。它不同于你前面看到的那些，因为错误不是发生在终端里，而是在进程标识符为<0.32.0>的进程里。如果错误发生在一个新生成的进程中，它将由Erlang运行时系统中错误记录程序（error logger）的另一部分来监控处理，默认情况下它会在终端中使用前面演示的格式来输出一个错误报告。相反终端检测出的错误要格式化为一个更可读的形式。

内置函数processes()返回所有系统中运行的进程列表。在大多数情况下，使用这个内置函数应该没有什么问题，但在大型系统中从终端调用processes()有可能会出现极端的情况，从而产生运行时系统内存不足的错误（注1）！请不要忘记，在工业应用程序中，需要处理数以百万计同时运行的进程。运行时系统目前实现的绝对极限是数以亿计。具体请参考Erlang文档得到最新的数字。系统的这一默认数字要低得多，但是你可以很容易地通过终端命令erl +p MaxProcesses来改变它，其中MaxProcesses是一个整数。

你可以使用终端命令i()来查看当前运行时系统正在执行的进程。它会输出进程标识符，用来生成这个进程的函数，以及在本章后面会讲到的一些其他消息。让我们来看看下面这些终端输出的例子。你能找出作为错误记录程序运行的进程吗？

```
2> processes().
[<0.0.0>,<0.2.0>,<0.4.0>,<0.5.0>,<0.7.0>,<0.8.0>,<0.9.0>,
<0.10.0>,<0.11.0>,<0.12.0>,<0.13.0>,<0.14.0>,<0.15.0>,
<0.17.0>,<0.18.0>,<0.19.0>,<0.20.0>,<0.21.0>,<0.22.0>,
<0.23.0>,<0.24.0>,<0.25.0>,<0.26.0>,<0.30.0>]
3> i().
Pid           Initial Call           Heap   Reds   Msgs
Registered    Current Function      Stack
<0.0.0>        otp_ring0:start/2     987    2684   0
```

注1： 部分原因是终端中缓存了操作的返回值。

| | | | | |
|-----------------|------------------------|------|-------|---|
| init | init:loop/1 | 2 | | |
| <0.2.0> | erlang:apply/2 | 2584 | 61740 | 0 |
| erl_prim_loader | erl_prim_loader:loop/3 | 5 | | |
| <0.4.0> | gen_event:init_it/6 | 610 | 219 | 0 |
| error_logger | gen_event:fetch_msg/5 | 11 | | |
| <0.5.0> | erlang:apply/2 | 1597 | 508 | 0 |
| ... | | | | |

很多人和你一样感到奇怪，当你只生成一个在生成后会马上出错的进程，为什么内置函数 `processes()` 会返回远远超过20个进程呢。这是因为Erlang运行时系统的大部分进程都使用了Erlang语言来实现，`error_logger`和Erlang终端属于许多例子中的两个。你在本书的余下内容中还会碰到其他的系统进程。

消息传递

进程使用消息传递进行相互通信。消息使用 `Pid!Message` 构造来发送，其中 `Pid` 是一个有效的进程标识符，而 `Message` 是属于任意Erlang数据类型的一个值（见图4-2）。

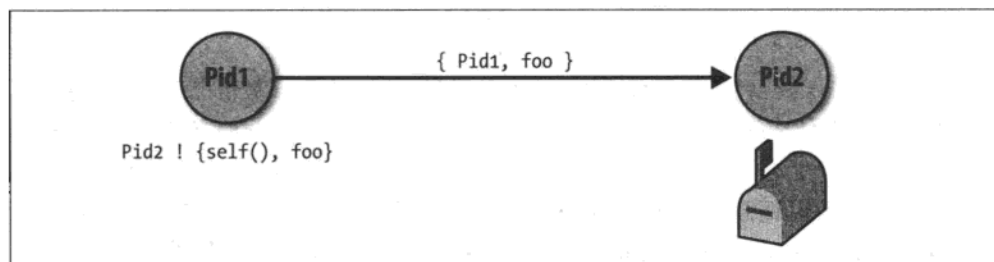


图4-2：消息传递

每个Erlang进程都有用来存储传入消息的信箱（mailbox）。当一个消息发送的时候，它会被从发送进程中复制到接收信箱以便于检索。消息会以它们到达的时间次序存储在信箱里面。如果两个消息从一个进程发送到另一个，那么消息接收到的次序和发送它们的次序肯定相同。不同的进程发送消息的时候没有这种保证，在这种情况下这个次序取决于虚拟机的具体实现。

发送消息永远不会失败，因此如果你尝试发送一个消息给一个不存在的进程，那么它只会被丢弃，但不会产生一个错误。最后，消息传递是异步的：一个发送进程不会在发送消息后被暂停，它会立即继续执行其代码中的下一个表达式。

为了在终端中测试发送信息，我们可以使用内置函数 `self/0`，它返回它所在的执行进程的标识符。Erlang终端就是一个处在read-evaluate-print循环中的Erlang进程，它等待你输入一个表达式。当你使用句点（.）终止一个表达式然后回车的时候，终端对你的

输入内容进行求值并输出结果。由于终端是一个Erlang进程，因此我们可以发送消息给它。你可以使用终端命令flush/0检索和显示所有发送到终端进程并保存在进程信箱中的消息，这个命令同时也删除（或刷新）信箱中的消息：

```
1> Pid = self().
<0.30.0>
2> Pid ! hello.
hello
3> flush().
Shell got hello
ok
4> <0.30.0> ! hello.
* 1: syntax error before: '<'
5> Pid2 = pid(0,30,0).
<0.30.0>
6> Pid2 ! hello2.
hello2
7> flush().
Shell got hello2
ok
```

前面的例子发生了什么呢？在命令行1中，内置函数self()返回一个进程标识符，它在终端中绑定到变量Pid并显示为<0.30.0>。在命令行2和命令行3中你可以看到消息发送到Pid，通过在终端使用flush()命令刷新信箱。

你不能在一个模块或者在终端中直接输入进程标识符，在这两种情况下，从终端的命令行4中我们都能看到，它们的结果都是一个语法错误。当内置函数如self和spawn返回一个进程标识符的时候，你需要把它绑定到一个变量，或者你使用终端函数pid/3产生一个进程标识符，就像命令行5所展示的和使用在命令行6里的一样。命令行7中的flush()在终端中显示消息。

PID!Message是一个有效的Erlang表达式，就像Erlang中的其他所有有效表达式一样，它返回一个值。在这种情况下这个值是被发送的消息。因此，如果你需要发送相同的消息到许多进程，你可以写一连串的消息发送，如Pid1!Msg,Pid2!Msg,Pid3!Msg，或者单一表达式，例如Pid3!Pid2!Pid1!Message，这相当于写Pid3!(Pid2!(Pid1!Message))，而Pid1!Message返回消息发送给Pid2，然后返回消息发送到Pid3。

正如我们前面已经说过的，把消息发送到不存在的进程也会永远成功。为了测试这个，让我们使用非法操作让终端进程崩溃。崩溃就如同一个异常进程的终止，这在Erlang中认为是正常的，Erlang提供了机制来处理它。我们在第5章中会更详细地讨论异常进程终止，所以在此之前请不要感到惊讶。使终端崩溃会自动产生一个新终端进程——在这个例子中运行时系统产生了一个进程标识符为<0.38.0>的新终端进程。

记住这一点，我们得到终端的进程标识符，终止终端进程，然后发送消息给它。基于消息传递的语义，这将导致消息丢弃：

```
7> self().
<0.30.0>
8> 1/0.
** exception error: bad argument in an arithmetic expression
    in operator '/'/2
       called as 1 /0
9> self().
<0.38.0>
10> pid(0,30,0) ! hello.
hello
11> flush().
ok
```

即使接收进程不存在或者生成的进程在创建时崩溃，消息传递和spawn也总是成功，其原因和进程依赖（process dependencies）有关，或者说它们故意没有依赖性。当进程B的终止会导致进程A的功能不正确的时候，我们就说进程A依赖于进程B。

进程依赖是非常重要的，而且往往会影响你的设计。在大规模并发系统中，你不会想要进程相互依赖，除非明确地想这么做，在这种情况下，你希望有尽可能少的依赖。让我们来看一个具体的例子，假设一个即时消息服务器同时处理数以千计条通过用户交换的消息。每个消息由一个专门处理此具体功能的进程来处理。如果因为一个错误，其中的一个进程终止了，那么你就会丢失这个具体的消息。确保它们之间很少的依赖，就可以让其他进程继续处理它们的消息，这保证了可以安全地处理这些消息和把它们正确传递给它们的接收者，而不用理会这个错误。

接收消息

消息可以使用receive语句从进程信箱中取得。receive语句是一个由保留字receive和end来确定界限的结构，它包含有若干语句。这些语句和case语句类似，其头部（箭头的左边）有一个模式，在语句体内（箭头的右边）有一系列的表达式。

在执行receive声明的时候，从信箱中的第一个（最早的）消息开始依次对在receive表达式里的每一个模式进行模式匹配：

- 如果成功匹配，这个消息从信箱中取出，在模式中的这个变量绑定到消息的匹配部分，并执行这个语句体。
- 如果没有匹配的语句，信箱随后的消息会逐一和所有语句模式匹配，直到一个消息匹配一个语句成功，或者所有的消息匹配所有可能的模式都失败了。

在下面的例子中，如果消息{reset, 151}发送到执行receive声明的进程中，第一条语句模式匹配成功，因此变量Board会绑定到整数151。这会导致调用函数reset(151)：

```
receive
  {reset, Board} -> reset(Board);
  _Other        -> {error, unknown_msg}
end
```

假设现在有两个新消息，restart和{reset, 151}，在执行进程中以这个次序依次接收到它们。一旦这个进程的执行流程到达了receive声明，它将尝试匹配信箱中最早接收到的消息restart。模式匹配在第一条语句失败，但在第二条语句中匹配成功，绑定变量_other到基元restart。如图4-3中的例子所示。

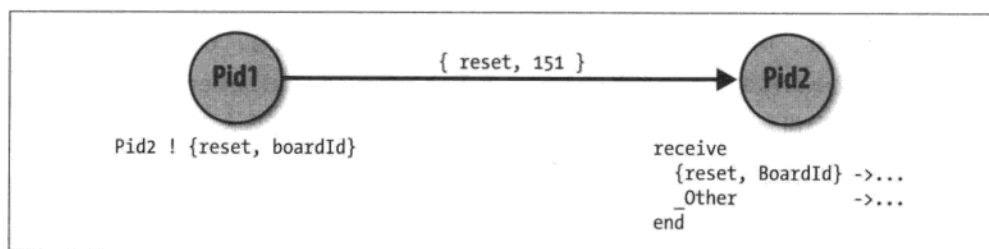


图4-3：选择性接收

一个receive声明返回匹配语句体的最后一个表达式的值。在这个例子中，这可以是reset/i调用的返回值或者是元组{error, unknown_msg}。虽然很少这样做，但是写下的表示式也可能是Result = receive Msg -> handle(Msg) end，其中一个变量（在这里指Result）绑定到receive语句的返回值了。让receive语句成为一个单独的函数体，并绑定返回值到这个变量，普遍认为这是一个更好的实践方法。

警告： 请花一分钟想想例子中的restart消息。虽然这不是一个很好的做法，但是我们依然可以发送一个格式为{restart}的消息。一个常见的误解是，以为消息必须是以元组的形式出现：其实并不是这样的，消息可以由任何有效的Erlang项元组成。

使用只有一个元素的元组是没有必要的，因为这样会消耗更多的内存而且处理得更慢。因此使用它们作为消息是不好的做法，在程序里面也是如此。一个准则是，如果元组只有一个元素，请使用元素自己，而不是用元组。

当一个case声明中没有有一个语句可以匹配的时候，就会抛出运行时错误。那么receive语句是怎么样的呢？receive的语法和语义与case非常相似的，它们的主要区别是，在receive语句中直到有一个消息匹配成功前这个进程都是被暂停的，而在一个case声明中则会发生运行时错误。

调度程序

现在来看看Erlang中的进程调度是如何工作的。让我们看看先前终端命令-:()的结果,你会发现一行的头部是Reds,它是规约(reduction)的一个缩写。程序中的每一个命令,无论它是一个函数调用,还是一个算术操作,或是内置函数,都会分配一定数量的规约步骤。虚拟机使用这个值来衡量一个进程的活动水平。

当派遣一个进程的时候,可以分配给它一定数量的允许其执行的规约(注2),数量在每个操作执行后都会减少。一旦这个进程进入receive语句,没有可以匹配的消息或者它的规约数量减少到零的时候,就是抢占(preempted)。只要内置函数没有得到执行,这一策略的结果就是进程之间公平(但不是相等的)分配执行时间。

谈到内置函数,例如所有的数学运算都分配有相同数目的规约,即使乘法和除法与加法或减法相比需要更长的运算时间。内置函数例如lists:reverse和lists:member,众所周知它们的执行时间会根据其输入的不同而变化很大,因此当需要的时候它们就会中断执行(这称为一个trap)来增加它们的规约数量。一般来说,新加入Erlang的内置函数实现默认都带有trap。过时的内置函数在需要的地方也可以重新实现。

你可以使用内置函数erlang:bump_reductions(Num)来增加一个进程的规约数量,而使用erlang:yield()来共同抢占进程。在标准对称多处理(SMP)模拟器中使用yield/0不会有太大影响。你不应该让调度程序的行为影响到如何设计你的系统和程序,因为这些行为可能在两个发布版本之间不知不觉地改变。但是,知道它们是如何工作的有助于在检测和配置系统时解释观察到的现象。

一般情况下, receive语句的形式如下:

```
receive
  Pattern1 when Guard1 -> exp11, .., exp1n;
  Pattern2 when Guard2 -> exp21, .., exp2n;
  ...
  Other                    -> expn1, .., expnn
end
```

限定receive语句范围的关键字是receive和end。每个模式由任何有效的Erlang项元,包括绑定的和未绑定变量以及可选保护元组成。表达式是有效的Erlang项元或者合法的求值为项元的表达式。receive语句的返回值是被执行的语句中最后一个表达式求值的结果,在这种情况下是expin。

注2: 规约数量在不同版本之间是不同的。在R12的版本中,规约数量从2000开始,然后进行一个操作就减少一个。在R13的版本中,规约的最初数量则取决于调度线程的数量。

为确保receive接收语句总是检索信箱中的第一条消息，你可以使用一个未绑定的变量（如在第一个例子中的Other）或者“无关紧要”变量，前提是你的确不关心它的值。

轻量级进程对比线程

Erlang进程是轻量级进程，它的生成、上下文切换和消息传递是由虚拟机管理的。操作系统线程和Erlang进程之间没有任何联系，这使并发有关的操作不仅独立于底层的操作系统，而且也是非常高效的和具有很强可扩展性。

Erlang并发的基准性能测试数值比它的竞争对手C#或者Java要好得多，尤其是当增加同时运行的有关进程创建和消息传递的进程数量的时候。在Erlang中我们每一个（或核心）处理器处理一个操作系统线程，而Java和C#则是每个进程代表一个操作系统线程。当然，有些人可能会认为比较这些并发模型就像比较苹果和橘子。而事实上，如果你需要写一个大规模的并发系统，无论你使用苹果或橘子都无关紧要，而最重要的是，你使用了正确的工具去工作。就结果来讲，这种比较是值得的。

选择性和非选择性接收

参见图4-4。你能否推断出传入消息的receive语句中的Pid已经绑定？调用decode_digit的时候变量Pid绑定到它传入的值，因此它已绑定到receive语句的模式部分了。当接收到一个消息，只有作为一个消息发送的元数为2的元组的第一个元素恰好和保存在变量Pid中的值相等（还记得:=结构吗），才会出现成功匹配。

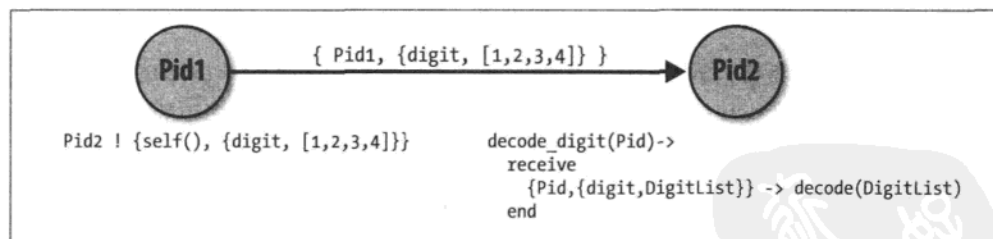


图4-4：带有绑定变量的选择性接收

我们把这种称为选择性接收（selective receive），基于一定的标准我们只检索明确表示感兴趣的信消息，而其他的消息留在信箱中。选择性接收往往基于进程标识符来进行选择，但顺序引用或其他标识符也是很常见的，例如标记元组和保护元。现在，让我们来比较图4-4中的绑定变量Pid和图4-5中的未绑定变量DigitList，当接收到一个消息的时候绑定哪一个呢？

在并发系统中，竞争状态是很常见的。当一个系统的行为取决于某些事件按一定次序发生的时候，竞争状态就出现了：这些事件的“竞争”影响了这些行为。由于并发的不确定性，导致无法确定由不同的进程发送的消息到达接收进程时的次序。此时选择性接收就变得至关重要了。

在图4-5中，不管它们发送的次序，Pid3会在消息foo之后收到消息bar。当进程在一个会合处同步，或者从几个数据源来的数据需要在处理前先收集起来的时候，选择性接收多个消息就非常有用。把这个和另外一门编程语言对比，在这个语言中可以根据消息到达的次序来处理消息：这当中的代码就必须处理bar先于foo或者foo先于bar的可能性问题。因此代码会变得更加复杂，也更有可能包含潜在的缺陷。

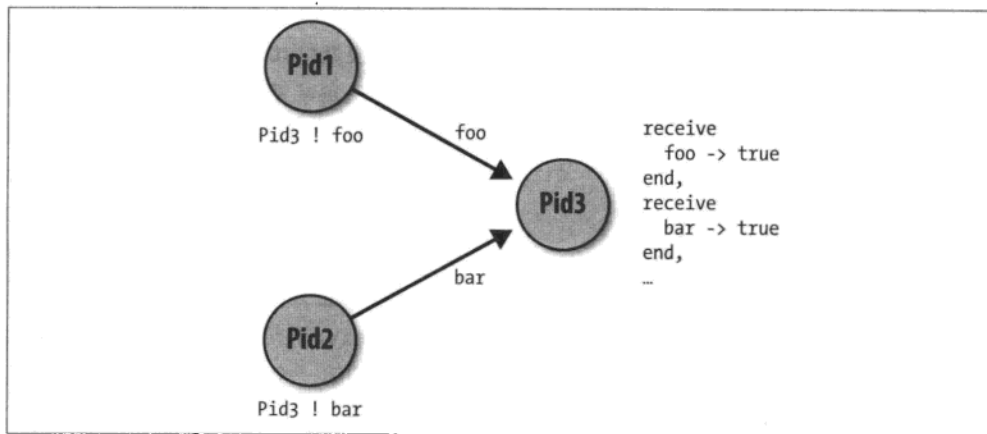


图4-5：多个消息的选择性接收

如果消息的次序不重要，就可以把它们绑定到一个变量上。在图4-6中，不用管消息的发送顺序，第一个到达进程Pid3的消息会马上得到处理。在receive语句中的变量Msg会根据谁第一个到达而绑定到基元foo或者基元bar。

图4-7展示了进程间是如何共享数据的。进程PidA发送一个包含通过调用内置函数self()得到自身的进程标识符的标记元组给进程PidB。PidB接收到它，绑定PidA的值到变量Pid。一个新的标记元组发送到PidC，它也在其接收的声明中模式匹配这个消息并绑定PidA的值到变量Pid。现在PidC使用Pid的绑定值发送消息foo返回给PidA。通过这种方式，进程彼此之间可以共享消息，而最初它们彼此都不知道。

由于进程间不共享内存，因此共享数据的唯一途径是通过消息传递。发送一个消息导致消息中的数据消息从发送进程堆中复制到接收进程堆中，因此这不会导致两个进程共享存储位置（每个都可能读取或写入），这是因为它们每个都有自己的数据副本了。

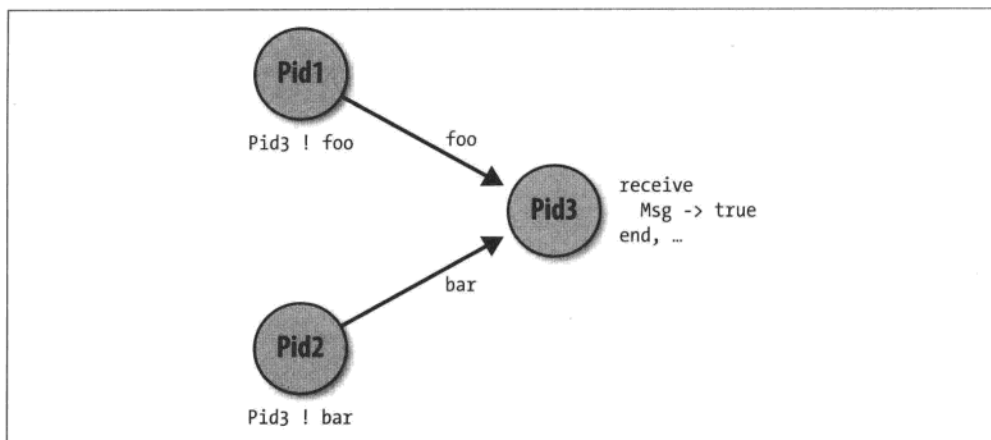


图4-6：无关发送次序的消息接收

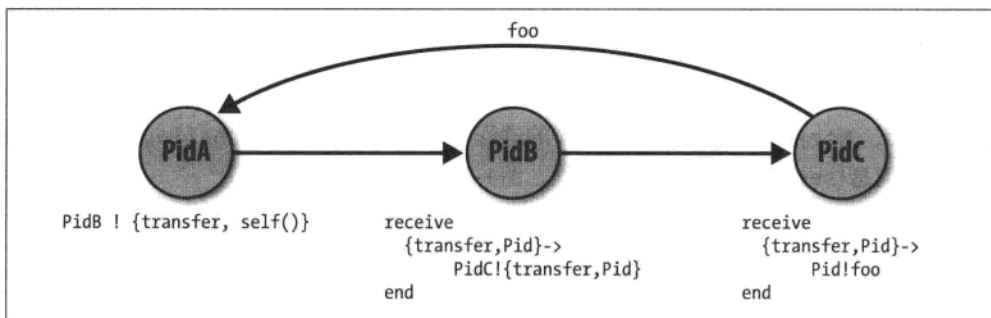


图4-7：进程间共享Pid数据

echo实例

既然我们已经讨论过进程生成和消息传递，现在让我们在一个小程序中使用`spawn`、`send`和`receive`。打开编辑器然后复制例4-1的内容或者从本书的网站下载。这样做的时候，请不要忘记导出用来创建进程的函数，在这里就是`loop/0`。在这个例子中，请特别注意这样一个事实，两个不同的进程之间是通过使用在同一模块中定义的代码来执行和互动的。

例4-1：echo进程

```

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

go() ->
  Pid = spawn(echo, loop, []),
  Pid ! {self(), hello},
  receive

```

```

{Pid, Msg} ->
    io:format("~w~n", [Msg])
end,
Pid ! stop.

loop() ->
    receive
        {From, Msg} ->
            From ! {self(), Msg},
            loop();
    stop ->
        true
end.

```

那么，这个程序是做什么的呢？调用函数`go/0`可以初始化一个进程，它的第一个动作是生成子进程。这个子进程开始执行`loop/0`函数，因为现在信箱是空的，所以立即在`receive`语句中暂停。还在`go/0`中执行的父进程，它通过绑定到内置函数`spawn`返回值上的子进程`pid`，发送给予进程一个包含父进程的进程标识符（通过调用`self()`得到）和基元`hello`的元组。

一旦消息发送了，父进程在`receive`语句中暂停。等待传入消息的子进程成功模式匹配元组`{Pid, Msg}`，其中`Pid`匹配父进程的进程标识符而`Msg`匹配基元`hello`。子进程使用`Pid`返回消息`{self(), hello}`到父进程。图4-8描述了这个过程。

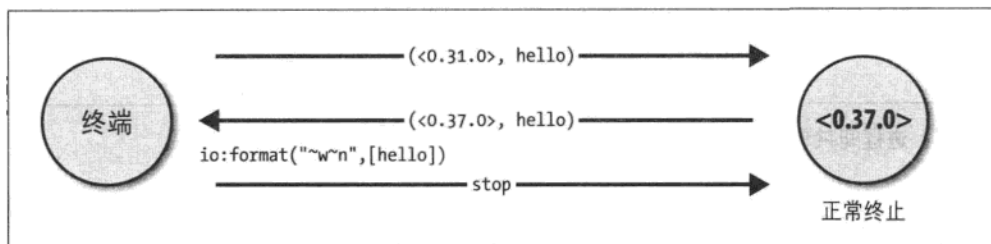


图4-8：例4-1的序列图

在这一点上，父进程在`receive`语句中暂停自己并等待一个消息。请注意，它只会模式匹配元组`{Pid, Msg}`，其中变量`Pid`已经绑定（作为内置函数`spawn`的结果）到子进程的进程标识符。这是一个不错（但并不是完全安全）的方式，用以确保收到的消息是事实上所期望的，而不仅仅是任何其他进程发送的由两个元素组成的一个元组。消息到达并且模式匹配成功。基元`hello`绑定到`Msg`变量，它作为一个参数传递给`io:format/2`调用，然后在终端中输出。一旦父进程在终端中输出基元`hello`，它就发送基元`stop`回子进程。

当父进程在忙着接收答复和输出它的时候，子进程在做什么呢？请记住，如果进程没有更多的代码可以执行的话就会终止，因此为了避免终止，子进程递归调用函数`loop/0`，

在receive语句中暂停。它接收到父进程发送的stop消息，返回基元true作为结果，然后正常终止。

请尝试在终端中运行程序，看看会发生什么：

```
1> c(echo).
{ok,echo}
2> echo:go().
hello
stop
```

基元hello显然是io:format/2调用的结果，但是基元stop从何而来呢？它是作为echo:go/0调用的结果返回的。为了让你自己更进一步熟悉并发，请尝试echo例子，将io:format/2声明放入loop/0进程然后发送不同的消息给它。你还可以尝试go/0进程，允许它发送和接收多个消息。在做实验的时候，很可能会遇到终端进程在receive语句中不匹配的地方被暂停的情况。如果发生了这种情况，你需要终止这个终端进程然后重新开始。

注册进程

进程之间直接使用进程标识符来进行通信并不总是可行的。为了使用一个进程标识符，一个进程需要知道并保存它的值。利用注册进程别名来提供特定的服务是很常见的，它是一个可用于代替进程标识符的名称。可以使用内置函数register(Alias, Pid)来注册一个进程，其中Alias是一个基元而Pid是进程标识符。可以不需要成为调用内置函数register的父进程或者子进程，当然只需要知道它的进程标识符。

一旦注册了一个进程，任何进程都可以将消息发送给它，而不需要知道它的标识符（见图4-9）。所有进程需要做的是使用结构Alias!Message。在程序里面，别名通常通过硬编码来使用。其他跟进程注册直接相关的内置函数包括unregister(Pid); registered(), 这个函数返回一个注册名称的列表，而whereis(Alias)则返回和Alias关联的进程标识符。

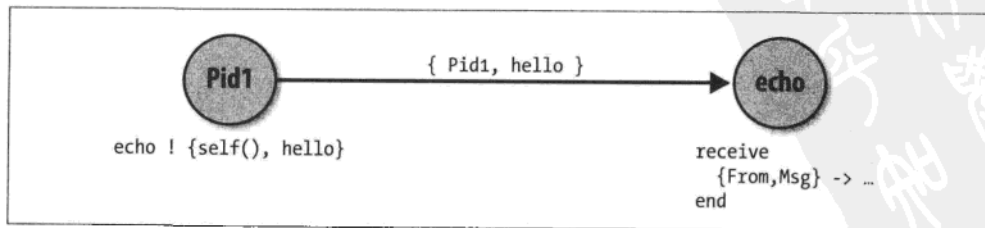


图4-9：发送一个消息到注册进程

看看例4-2，它是例4-1的一个变化。我们已经删除了go/0函数末尾的PID!stop表达式，取而代之的是绑定spawn/3的返回值，把它作为第二个参数传递给内置函数register。register的第一个参数是用来命名这个进程的基元echo。这个别名用来发送消息到新生成的子进程。

例4-2：注册echo进程

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

go() ->
  register(echo, spawn(echo, loop, [])),
  echo ! {self(), hello},
  receive
    {_Pid, Msg} ->
      io:format("~w~n", [Msg])
  end.

loop() ->
  receive
    {From, Msg} ->
      From ! {self(), Msg},
      loop();
    stop ->
      true
  end.
```

这虽然不是强制性的，但以进程定义所在的模块的名字来命名进程是一个很不错的做法。

使用刚刚讨论过的改变更新echo模块，请在终端中尝试刚刚读到的新的内置函数。测试echo的新实现并使用终端命令i()和regs()检查它的状态。请注意终端进程在不知道echo的进程标识符的情况下是如何发送stop消息给echo进程的，以及whereis/1如何在进程不存在的时候返回undefined的：

```
1> c(echo).
{ok,echo}
2> echo:go().
hello
ok
3> whereis(echo).
<0.37.0>
4> echo ! stop.
stop
5> whereis(echo).
undefined
6> regs().
```

```
** Registered procs on node nonode@nohost **
```

| Name | Pid | Initial Call | Reds | Msgs |
|-----------------------|---------|----------------|------|------|
| application_controlle | <0.5.0> | erlang:apply/2 | 4426 | 0 |

```

code_server      <0.20.0>      erlang:apply/2      112203      0
ddll_server      <0.10.0>      erl_ddll:init/1      32      0
erl_prim_loader  <0.2.0>      erlang:apply/2      206631      0
error_logger     <0.4.0>      gen_event:init_it/6      209      0
file_server      <0.19.0>      erlang:apply/2      12      0
file_server_2    <0.18.0>      file_server:init/1      25411      0
global_group     <0.17.0>      global_group:init/1      71      0
global_name_server <0.12.0>      global:init/1      60      0
inet_db          <0.15.0>      inet_db:init/1      103      0
init             <0.0.0>      otp_ring0:start/2      5017      0
kernel_safe_sup  <0.26.0>      supervisor:kernel/1      61      0
kernel_sup       <0.9.0>      supervisor:kernel/1      1377      0
rex              <0.11.0>      rpc:init/1      44      0
user             <0.23.0>      user:server/2      1459      0

** Registered ports on node nonode@nohost **
Name      Id      Command
ok

```

终端命令`regs()`会输出所有已注册的进程。当从一个拥有大量进程的系统里面检索系统信息的时候，它相对于`i()`也是另外一种选择。在前面的例子中，`echo`进程没有在罗列出来的进程里面出现，这是因为我们已停止它了。然而，你将看到所有注册过的系统进程。

警告：垃圾收集器不会收集基元是Erlang的一个内存管理特性。一旦你生成了一个基元，那么不管在代码中是否被引用它都会停留在基元表里。如果决定为一个瞬时的进程注册别名，而该别名是通过内置函数`list_to_atom`把一个字符串转换为一个基元而得到的，这就可能是一个潜在的问题。如果每天有数以百万计的用户登录系统，而你为它们的每个会话都建立一个已注册的进程，那么请不要惊讶最终将耗尽内存。

把用户到进程标识符的映射保存到一个会话表（session table）中对于你可能会更好些。最好是只注册生命周期长的进程，如果你真的必须将字符串转换为一个别名，请使用`list_to_existing_atom/1`，用以确保你的系统不会遭受内存泄漏。

发送一个消息到不存在的注册进程会导致`badarg`错误，从而使调用进程终止（见图4-10）。这种情况不同于发送消息到一个进程不存在的进程标识符，因为注册进程假设已经提供了一种服务，所以将注册进程的不存在当做是一个错误。如果你的程序希望发送消息到不存在的注册进程而不终止调用进程，那么请使用`try...catch`把这个调用保护起来。

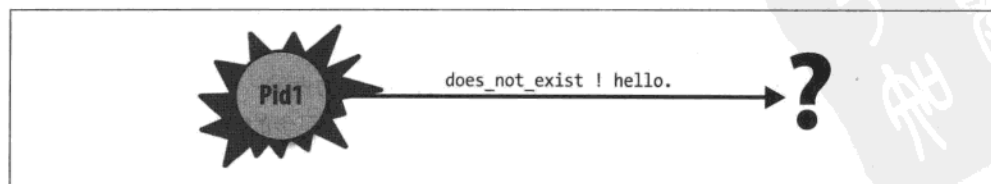


图4-10：发送信息到未注册的进程

超时

你可以看到，当进程进入一个receive语句而没有一个消息可以匹配的时候，这个进程就会暂停。这可能类似于你在家看信箱，发现没有任何邮件，那么被迫等待直到邮差到来。如果你是非常急迫地等待你的邮件或者没有任何更好的事情可做的时候，这可能是一种选择。但在大多数情况下，你要做的就是检查信箱，如果没有到达，你就继续做你的家务。Erlang进程也可以通过receive...after结构来这么做：

```
receive
  Pattern1 when Guard1 -> exp11, .., exp1n;
  Pattern2 when Guard2 -> exp21, .., exp2n;
  ...
  Other                    -> expn1, .., expnn
after
  Timeout -> exp1, .., expn
end
```

当一个进程到达receive语句而没有消息可以模式匹配的时候，它将会等待Timeout毫秒。如果超过Timeout毫秒后还是没有消息到达，那么after语句后的表达式就会执行。Timeout是一个整数，它表明一个以毫秒为单位的时间或者是基元infinity。使用infinity作为Timeout的值就跟不包括after结构一样。Timeout可以是变量，可以在每次调用函数的时候设置它，这样就可以在每次调用的时候让receive...after语句表现得和我们所期望的一样（见图4-11）。

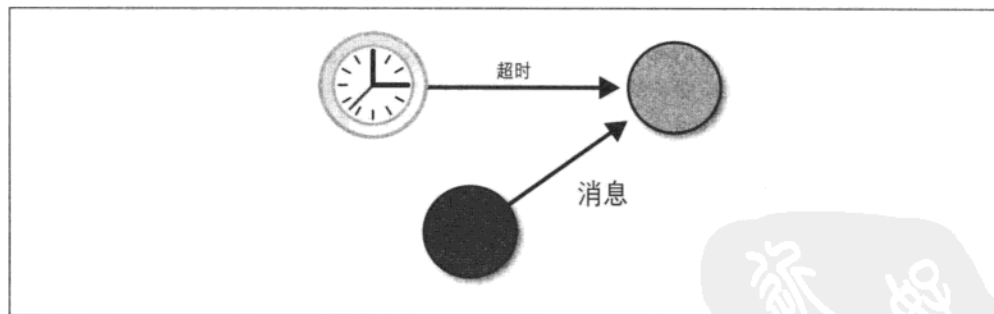


图 4-11：接收超时

假设你有个进程以别名db注册，它作为一个数据库服务器运行。每次你要查找一个项，你发送给这个数据库一个消息然后等待响应。然而在繁忙的时候，可能需要很长的时间才能处理请求，因此你可以通过使用receive...after结构返回一个超时错误。但是如果这样做，就冒着答复和发送到数据库服务器的串行请求不同步的风险，在timeout后最终将接收到来自服务器的响应而终止。下次发送给数据库一个请求，就会匹配最早出现在receive语句中的消息。这个消息在timeout后发送回答复，而不是响应刚刚发的请

求。当使用`receive...after`的时候，可以考虑刷新信箱并确保它是空的。这样代码如下所示：

```
read(Key) ->
  flush(),
  db ! {self(),{read, Key}},
  receive
    {read,R}          -> {ok, R};
    {error, Reason} -> {error, Reason}
  after 1000          -> {error, timeout}
end.

flush() ->
  receive
    {read, _}          -> flush();
    {error, _}         -> flush()
  after 0              -> ok
end.
```

`receive...after`语句的另一个用法是以毫秒级单位暂停一个进程，或者延迟一定时间后再发送消息。下面代码中的`sleep/1`定义直接取自`timer`库模块，而`send_after`会在`Time`毫秒后发送一个消息给调用进程：

```
-module(my_timer).
-export([send_after/2, sleep/1, send/3]).

send_after(Time, Msg) ->
  spawn(my_timer, send, [self(),Time,Msg]).

send(Pid, Time, Msg) ->
  receive
  after
    Time ->
      Pid ! Msg
  end.

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

性能基准测试

本章一直在讨论Erlang中很短的进程生成和消息传递时间。为了说明它们，先来运行一个性能基准程序，其中父进程生成一个子进程然后发送一个消息给它。当正在生成的时候，子进程生成一个新的进程并等待一个来自父进程的消息。当接收到消息的时候，它正常终止。子进程的子进程再生成另一个进程，从而导致数百、数以千计甚至上百万的进程。

这是一个顺序性能测试，它利用了SMP在多核系统上的优势，因为在任何时候，只有几个进程会并发执行：

```
-module(myring).
-export([start/1, start_proc/2]).

start(Num) ->
    start_proc(Num, self()).

start_proc(0, Pid) ->
    Pid ! ok;

start_proc(Num, Pid) ->
    NPid = spawn(myring, start_proc, [Num-1, Pid]),
    NPid ! ok,
    receive ok -> ok end.
```

让我们分别以十万、一百万和一千万的进程来测试前面的例子。为了测试这个程序，我们可以使用函数调用：

```
timer:tc(Module, Function, Arguments)
```

它需要一个函数及其参数来执行。它返回一个元组，其中包含运行这个函数花费的时间和这个函数的返回值。测试结果表明，这个程序需要花费0.48秒来生成100 000个进程，4.2秒生成百万进程，大约40秒生成千万进程。在计算机上可以尝试这些：

```
1> c(myring).
{ok,myring}
2> timer:tc(myring, start, [100000]).
{484000,ok}
3> timer:tc(myring, start, [1000000]).
{4289360,ok}
4> timer:tc(myring, start, [10000000]).
{40572800,ok}
```

进程架构

不管特定目的是什么，进程行为有一个共同的模式。必须生成进程并以它们的别名注册。新生成进程的第一个动作是初始化进程循环数据。循环数据通常是传递给内置函数 `spawn` 的参数和进程初始化的结果。它存储在一个我们称之为进程状态（process state）的变量中。把这个状态传递给接收求值函数，它收到一条消息，处理它和更新状态，然后作为尾递归调用的一个参数返回。如果它处理的消息之一是 `stop` 消息，接收进程就会在自身执行完后清理并结束。这个进程设计中的反复出现的主题我们称为设计模式（design pattern），不管进程的任务是什么它一定会出现。图4-12给出了这样的一个例子。

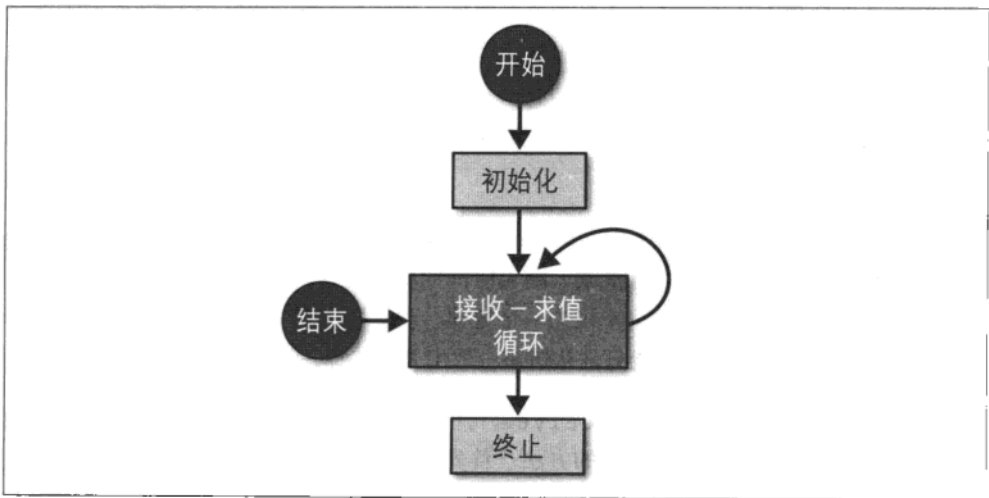


图4-12：进程架构

从这些反复出现的模式中来看看进程之间的区别：

- 各个进程传递到内置函数`spawn`调用的参数是不同的。
- 你必须决定是否要注册一个进程，如果你注册它，那么应该使用哪个别名。
- 在初始化进程状态的函数中，所采取的行动根据进程执行任务的不同而不同。
- 进程状态的存储有可能是通用的，但其内容根据不同的进程而有所不同。
- 在接收-求值循环里，进程会接收不同的消息和以不同的方式来处理它们。
- 最后，结束时各个进程的清理工作都不相同。

因此，即使存在通用行为的架构，这些行为也是以直接跟具体分配给进程的任务有关的方式完成的。

尾递归和内存泄漏

我们刚才提到，如果进程没有更多的代码可以执行，它们就会终止。假设你要编写一个`echo`进程，它无限期地持续回送接收到的消息（或者执行这些直到明确地发送消息来停止它）。你可以使用尾递归调用那个包含`receive`语句的函数来让Erlang进程一直活跃。我们常常称这个函数是进程的接受/计算循环。它的任务是接收一条消息，处理它，然后递归调用自身。

尾递归在并发编程中的重要性在这里变得很明显。因为你不知道将会调用多少次这个函数，你必须确保它在常量的内存空间内执行，当每次处理一个消息的时候不会增加递归调用堆栈。每分钟处理数千次以上的消息，且持续数小时、数天、数月或者数年都是很常见的！使用尾递归，接受/计算函数的最后一件事是调用自己，这样你就可以确保没有内存泄漏地不间断运行了。

当一个消息和receive声明中的所有语句都不匹配的时候会发生什么呢？它会无限期地停留在信箱中，这就造成了内存泄漏，随着时间的推移这还可能导致运行时系统内存溢出和崩溃。因此不处理未知消息应视为一个错误。要么刚开始就不应该发送这些消息到这个进程，要么处理它们，可能只是从信箱中取出并忽略掉。

忽视未知消息的防御性办法是在receive语句中使用“无关紧要”变量，虽然这个方法很方便，但它可能并不是一个最好的方法。首先没有被处理的消息也许就不应该发送到这个进程。如果它们是有意发送的，它们可能因为receive语句中的一个编程错误而不能匹配。抛弃这些消息只会把发现错误变得更困难。如果抛弃了未知的消息，请一定要记录它们的出现，这样做至少以后可以比较容易地发现和纠正这些错误。

并发相关的瓶颈

说进程是瓶颈指的是，随着时间的推移，它们发送消息的速度比处理信息的速度要快，这就导致了很长的信箱队列。信箱里有着很多消息的进程的性能如何呢？答案是：很糟糕。

首先，这个进程本身通过选择性receive可能只匹配某一个特定类型的消息。如果该消息是在信箱队列中的最后一个，在此消息成功匹配之前整个信箱必须先遍历一遍。这将导致性能上的问题，比如CPU时间的高占用性。

其次，给一个带有长消息队列的进程发消息的进程，将会导致发送消息的规约数量增加而受到惩罚。运行时系统总是首先试图放缓发送消息的进程，从而让带有长消息队列的进程赶上来。后者的瓶颈往往表现为系统整体吞吐量减少。

发现是否有瓶颈的唯一方法是在进行系统压力测试时观察它的吞吐量和消息队列的形成。消息队列问题的简单补救措施可以通过优化代码、微调操作系统和设置虚拟机来实现。

另一种减缓消息队列增长的方法是阻塞生成消息的进程，直到它们收到确认表明发出的消息已经收到并处理，从而有效地创建一个同步调用。当系统高负荷运行的时候，使用同步调用替换异步调用会降低这个系统的最高吞吐量，但是这也比造成消

息队列的增长付出的代价小很多。因此当我们知道某个地方发生瓶颈的时候，更安全的方法是引入同步调用降低吞吐量，这样就保证了高负荷运行时系统常量的请求吞吐量，而没有降低服务等级。

面向并发程序设计的个案研究

当我们在世界各地提供咨询服务的时候，碰到了许多背景为C++和Java的开发者，他们以自己的方式学习Erlang，他们碰到的一个很常见的问题是如何使用进程。不管开发者的经验水平达到了什么层次和他们的系统具体做什么，这个问题都经常出现。他们往往不是为系统中的每个真正的并发活动生成一个进程，而是常常为每个任务生成一个进程。Erlang中的并发应用程序需要对进程采取不同的策略，这反过来又意味着不同的方式与不同的推理习惯。Erlang和其他并发语言的主要区别是，在Erlang中使用进程是很方便的，因此最好在系统中为每个真正的并发活动生成一个进程，而不是为每个任务。本案例研究是在Erlang作为开源发布后不久，Francesco在爱立信以外承担的第一个咨询服务，他清楚地描述了任务和活动之间的差异。

他曾和开发Jabber即时消息代理的一组工程师一起工作。该系统通过套接字接收数据包和解码，然后根据其内容采取某些行动。一旦这个活动完成，它编码一个答复并发送到不同的套接字，然后转发到收件人。每一次只有一个包可以通过套接字，但许多套接字可以同时接收和处理数据包。

如图4-13所示，原有的系统没有真正为每一个并发活动——即处理端到端的数据包——准备一个进程，而是使用一个进程处理不同的任务——解码、编码等。在Erlang中每一个打开的套接字和一个进程相连，它通过套接字接收和发送数据。一旦接收到数据包，它就转发给一个进程去处理。一旦解码，解码进程向前转发给处理它的进程。结果被送往编码进程，该进程在格式化后，把回复向前转发给拥有属于接收者的打开链接的套接字进程。

在这个系统性能最佳的时候，它可以同时处理5个消息，无论并发连接套接字的数量是多少，解码、处理和编码在这里都有可能成为瓶颈。这里还有两个其他的进程，一个是当错误发生的时候用于处理错误；另一个是当读取、写入和删除数据的时候用于管理数据库。

重新回顾这个系统，我们明确了什么是我们认为的系统中的一个真正的并发活动。解码、处理和编码这些活动都不是答案，而是它们自己处理单独的数据包的活动。让每个数据包拥有一个进程，并使用这个进程进行数据包的解码、处理和编码，这意味着如果

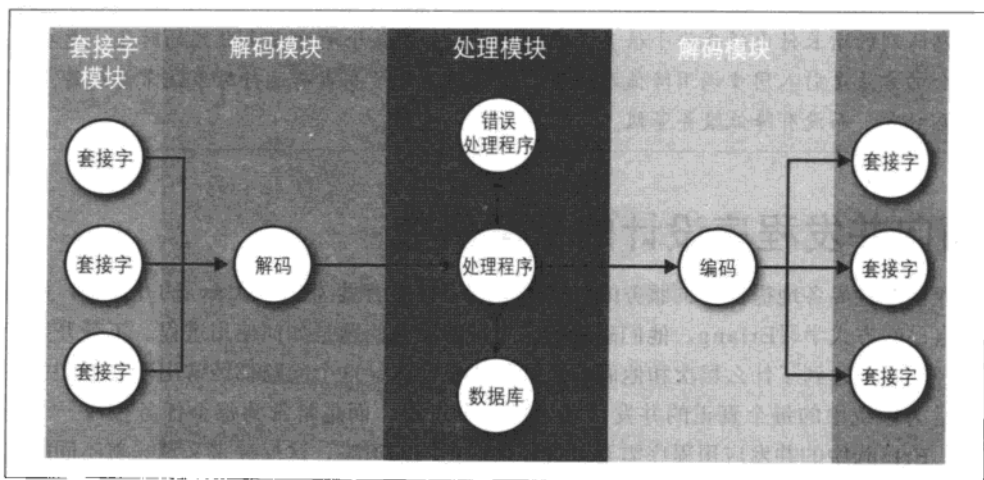


图4-13：一个设计糟糕的并发系统

有数千个数据包同时收到，它们都将会同时处理。认识到一个套接字在任何时间只可以接收一个数据包，意味着我们可以利用这个套接字进程来处理调用。一旦数据包到达，一个函数调用就确保了它被解码和处理。然后编码结果（可能是一个错误）发送给管理这个连接并属于最终接收者的套接字进程。我们不需要错误处理句柄和数据库进程，因为数据一致性通过数据库破坏性操作的序列化已经在处理进程中达到了，错误管理也是如此。

如果仔细查看图4-14，你会注意到，在套接字进程的上面，重写的程序添加了一个数据库进程。这是为了确保和维持数据一致性，因为许多进程访问相同的数据的结果是可能由于存在竞争而破坏数据。所有拥有破坏性的数据库操作，如写入和删除都会通过这一进程进行序列化。即使可以并发执行大部分活动，但最重要的是确定活动需要序列化和把它们放入属于自己的一个进程中。请小心查明在Erlang系统中真正的并发活动，并为它们每个生成一个进程，这样可以在确保最大化的吞吐量同时减少瓶颈发生的危险。

竞争条件、死锁和饥饿进程

任何人在使用Erlang之前如果已经对并发应用进行过编程，肯定都有他们自己的关于内存出错、死锁、竞争条件和饥饿进程的各种恐怖体验。内存共享和需要信号量才能进行访问的机制导致其中的一些情况出现。而其他是因为使用了优先权的结果。

进程间共享数据的唯一办法是通过从一个进程到另一个进程复制数据，使用一个“不共享数据”的办法，就消除了对锁的需要，结果也就消除了大部分与内存崩溃相关的错误、死锁和竞争条件。

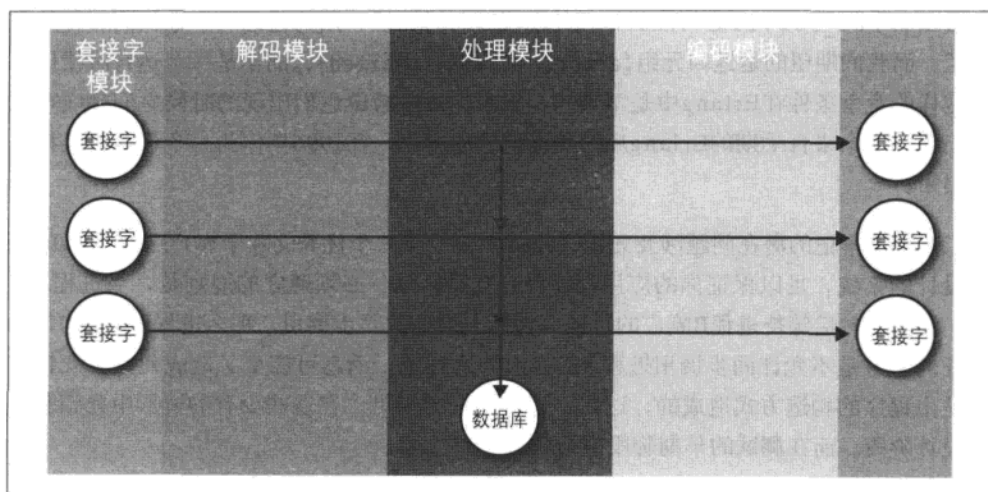


图4-14：为每个并发活动生成一个进程

并发程序中存在的问题也有可能是由于同步消息传递引起的，特别是当通信是通过网络传播的时候。Erlang通过异步消息传递来解决这个问题。最后，Erlang系统支持的调度程序、以进程为单位的垃圾收集机制以及大规模级别的并发，确保所有进程在执行时得到相对公平的时间片。在大多数系统中，你可以预计大多数进程在一个receive声明中暂停，它等待一个事件来触发一系列的活动。

Erlang也不是完全没有问题的。但是通过仔细和深思熟虑的设计你可以避免这些问题。让我们先从竞争条件开始。如果两个进程同时执行下面的代码，可能出现什么问题呢？

```

start() ->
  case whereis(db_server) of
    undefined ->
      Pid = spawn(db_server, init, []),
      register(db_server, Pid),
      {ok, Pid};
    Pid when is_pid(Pid) ->
      {error, already_started}
  end.
  
```

假设数据库服务器进程尚未开始，两个进程同时开始执行start/0函数。第一个进程调用whereis(db_server)，它返回基元undefined。模式匹配第一个语句，结果是一个新数据库服务器。它的进程标识符绑定到变量Pid。如果在创建数据库服务器之后，该进程使用完了规约并被抢占，这将允许第二个进程开始执行。第二个进程调用whereis(db_server)会返回undefined，因为第一个进程没有注册它。第二个进程创建数据库服务器并可能比第一个走得更远些，使用名字db_server来注册它。在这个阶段，第二个进程被抢占然后第一个进程在停止的地方继续运行。它试图为它创建的数据库服

务器以名字`db_server`进行注册，但是出现运行时错误，因为已经有一个相同名称的进程了。而我们期望的是返回元组`{error, already_started}`，而不是一个运行时错误。像这样的竞争条件在Erlang中是罕见的，但当你最不希望它们出现的时候它们确实可能会出现。一个来自早期的Erlang库的前面例子的变化，在1996年作为一个程序错误报告了出来。

第二个需要牢记的潜在问题涉及死锁（deadlocks）。一个优秀的基于客户端/服务器的原理设计的系统，足以保证你的应用程序没有死锁。唯一必须遵守的规则是，当进程A发送一个消息然后等待进程B响应的时候，实际上就是做同步调用，那么进程B在它的代码的任何地方是不允许同步调用进程A的，因为这样做，消息可能交叉造成死锁。死锁直接是由程序的构造方式造成的，这在Erlang中非常罕见。在这些少有的情形中死锁躲过了设计阶段，而在测试的早期阶段被捕获（注3）。

通过调用内置函数`process_flag(priority, Priority)`可以设置Priority为基元`high`、`normal`或者`low`，进而改变调度程序的行为，给予进程较高的优先级，进程调遣的时候就会优先处理它。你不仅仅应该谨慎地使用这个功能，事实上你根本不应该使用它！因为Erlang的大部分运行时系统是用Erlang语言编写的并以普通优先级运行，所以这样就会出现死锁和饥饿，在极端的情况下，一个调度程序会让低优先级别进程比它高优先级的竞争进程占用更多的CPU时间。使用SMP的系统，这个行为变得更加不确定。在邮件列表里面关于这种Erlang问题的争论很多，有关这一主题的内容甚至可以写整整一章。因此应该这样约束，即在任何情况下你都应该避免使用进程中的优先级别。一个正确的并发模式设计将确保系统是平衡和可确定的，而没有饥饿进程、死锁或者竞争条件。请谨记这一点！

进程管理器

进程管理器是一个调试工具，它用于检查Erlang系统中的进程状态。调试器主要用来跟踪程序的顺序方面的特性，而进程管理器则处理并发的方面。你可以在终端中输入`pman:start()`来启动进程管理器。这会打开一个窗口（见图4-15），显示的内容类似于你尝试使用`i()`命令的时候所看到的。双击任何的进程都会打开一个跟踪输出窗口。你可以通过选择文件菜单中的选项来设置你的选项。

每一个进程都有一个输出窗口，你可以跟踪所有的发送和接收消息。你可以跟踪内置函数和普通函数调用，以及并发相关的事件，例如进程正在生成或者终止。你也可以

注3： 在使用Erlang编写的数百万行代码的系统上工作了15年的历史当中，Francesco只有一次在系统集成阶段遇到了死锁。

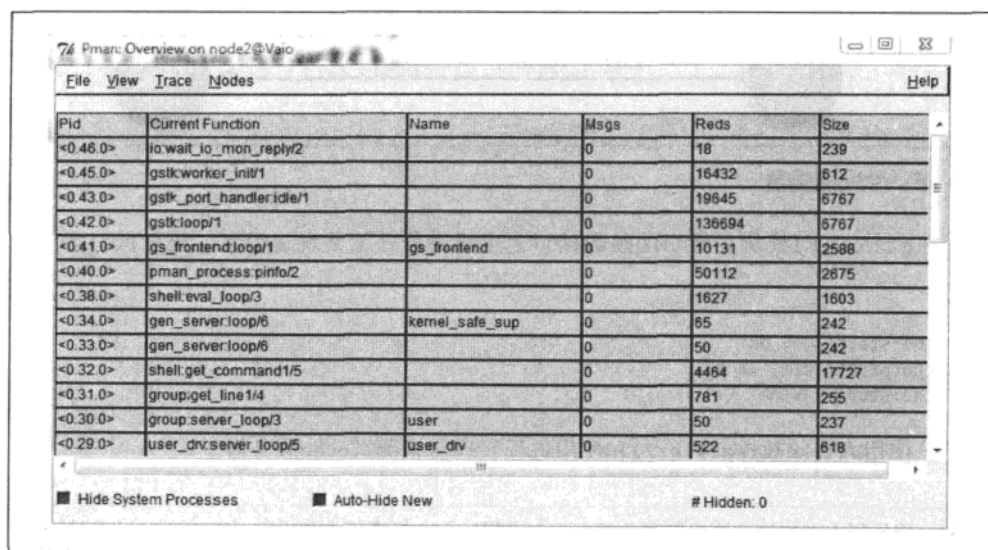


图4-15：进程管理器窗口

把你的跟踪输出从窗口重定向到一个文件里。最后，你可以选择继承层次来跟踪事件。Erlang发行版本中非常详尽的、精心编写的用户指南是我们推荐进一步阅读的资料。

警告：在撰写本书时，由于不再支持底层TCL/TK图形库，进程管理器运行在Microsoft Windows操作系统上的时候可能会不稳定。

本章介绍了Erlang中关于并发的基础知识，它是基于并发进程间互相传递消息而不是共享内存的。消息传递是异步的，并具有选择性receive的能力，它们可以独立于其所接收的次序而被处理，这些便于编写模块化的和简洁的并发程序。第5章将在这个基础上讨论以进程为基础的系统的的设计模式。

练习

练习4-1：echo服务器

编写如图4-16所示的服务器，它在接收循环里等待一个发送给它的消息。根据消息的不同，它可以输出内容或者再次循环，或者终止。如果你想隐藏正在处理进程的事实，那么可以通过一个函数接口来访问它的服务，这样你就可以在终端中调用它了。

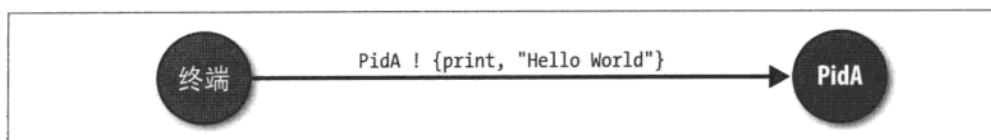


图 4-16: echo服务器

这个在`echo.erl`模块中导出的函数接口会生成进程并给其发送消息。函数接口如下所示：

```
echo:start() => ok  
echo:print(Term) => ok  
echo:stop() => ok
```

提示：使用内置函数`register/2`和使用进程管理器来测试echo服务器。

警告：请使用内部的消息协议以避免结束这个进程，例如当调用函数`echo:print(stop)`的时候。

练习4-2：进程环

编写一个程序，它生成 N 个进程并相连成一个环，如图4-17所示。一旦启动，这些进程会绕环发送 M 个消息，然后当收到退出消息的时候正常终止。你可以调用`ring:start(M, N, Message)`来启动环。

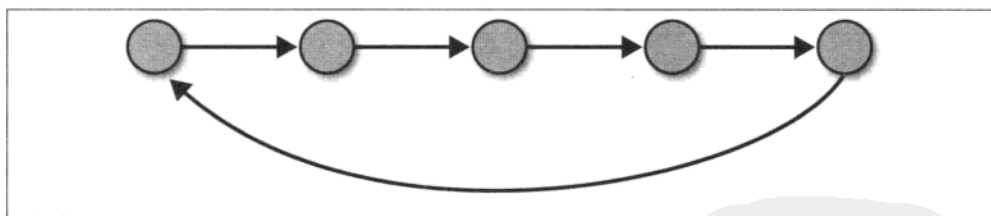


图4-17：进程环

有两种策略可以完成这个练习。第一种是通过一个中央进程，它设置环并启动发送消息。第二种方法是环里面的新进程产生下一个进程。使用这一策略，你必须找到一种方法来连接第一个进程到第二个进程。

无论你选择哪种方法，请确保你先使用笔和纸解决了这个问题，然后再开始编码。这不同于你以前解决的问题，因为有很多进程在同一时间执行相同模块中的相同函数。此外，进程之间会使用这个函数进行相互交互。在编写程序的时候，请确保你的代码在每一个循环迭代中都有很多`io:format`声明。这会给你一个到底发生了什么（或者没有发生）的完整概观，以便帮助你完成这个练习。

进程设计模式

在Erlang系统中，进程可以作为数据库网关、处理协议栈或者管理跟踪消息的日志记录。虽然这些进程可以处理不同的请求，但在如何处理这些请求方面是很相似的。我们把这些相似性叫做设计模式（design pattern）。在本章中，当我们使用Erlang进程的时候，将会讨论接触到的最常见模式。

客户端/服务器模型通常应用于负责资源分配的进程，比如一个房间列表，以及这些资源上的服务，比如房间预订或者查看它的可用性等。对服务器的请求将允许客户端（通常用Erlang进程实现）访问这些资源和服务。

另外一种很常见的模式是有限状态机（finite state machine, FSM）。想象一下，在一个即时消息（IM）会话内由进程处理事件。这个进程或称为有限状态机会处于三个状态中的一个。当远程ZM服务器会话正在建立的时候，它可能处于offline状态。为了使用户可以发送、接收邮件和更新状态，它有可能处于online状态。最后，当用户希望保持在线状态，但又不想收到任何消息或者状态更新的时候，它可能处于一个busy状态。状态的改变通过我们称之为事件（event）的进程消息来触发。一个IM服务器通知有限状态机用户成功登录，这会导致offline状态到online状态的状态转变（state transition）。有限状态机接收到事件不一定会触发状态转变。即时消息的接收或状态更新会保持有限状态机处于online状态，而退出事件会导致它从联机或者忙碌状态转变为offline状态。

最后一个我们会提到的模式是事件句柄（event handler）。事件句柄会接收特定类型的消息。这些可能是在你的程序中生成的跟踪消息或从外部得到的股票报价。在接收到这些事件的时候，你可能想实现一些操作比如发送SMS（手机短信）或者当某些条件合适的时候发送电子邮件，或者只是想把时间戳和股票价格记录在一个文件里面。

许多Erlang进程都属于这三种类别中的一种。在本章中，我们将研究进程设计模式的例子，解释如何应用它们到客户端/服务器程序、有限状态机和事件句柄来编程。经验丰富

的Erlang程序员会在项目设计阶段就认识到这些模式，以及使用属于OTP框架的库和模板。现在我们暂时不介绍使用OTP框架。我们将在第12章中具体介绍它。

客户端/服务器模型

Erlang进程可用来实现客户端/服务器解决方案，其中客户端和服务端都表示为Erlang进程。服务器可以是一个打印机的先进先出FIFO队列、窗口管理器或者文件服务器。它处理的资源可能是一个数据库、日历或者有限的物品列表，比如房间、书籍和电波频段等。客户端使用这些资源发送对服务器的请求：打印文件、更新窗口、预订房间或者使用电波频段。服务器收到请求并处理，如果请求成功就会响应确认消息并返回值，否则会返回一个错误消息（见图5-1）。

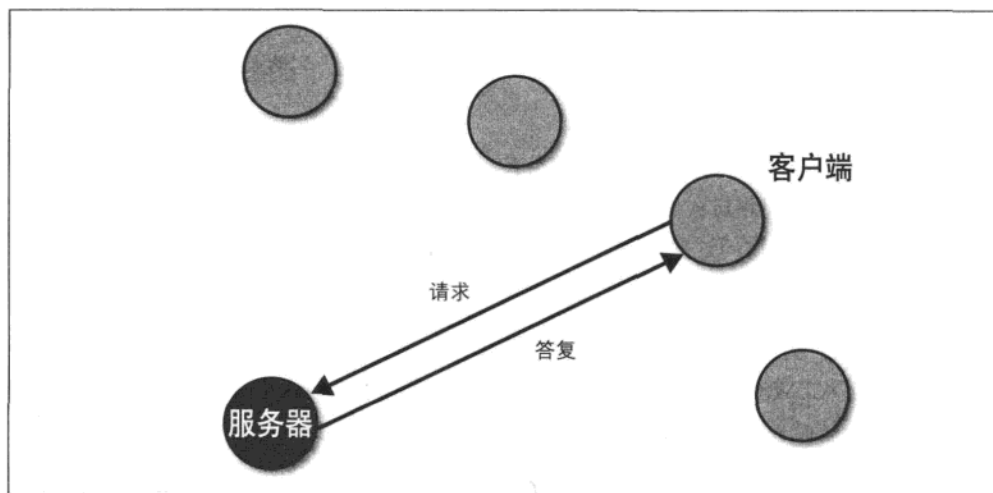


图5-1：客户端/服务器模型

在实现客户端/服务器行为的时候，客户端和服务端都表示为Erlang进程。它们之间通过发送和接收消息发生互动。消息传递往往隐藏在函数接口中，因此不是函数调用：

```
printerserver ! {print, File}
```

一个客户端会调用：

```
printerserver:print(File)
```

这是一种消息隐藏的形式，我们不会让客户端知道服务器是一个进程，这个进程其实是可以注册的而且有可能会驻留在远程计算机中。我们也不应该暴露客户端和服务端之间使用的消息协议，以保持它们之间的接口简单和安全。而所有的客户端需要做的就是调用函数并期待返回值。

对于把信息隐藏在函数接口之后，我们必须特别小心对待。当进程繁忙或运行在远程计算机上的时候，消息响应时间会有所不同。尽管在大多数情况下这不会引起任何问题，但是客户端需要了解它的存在和能够应付响应时间延迟的问题。你还需要知道函数调用有可能出错。其原因有可能是网络故障、服务器进程崩溃或者请求太多，以致服务器的响应时间变得让人不能接受了。

如果使用服务器管理资源或资源的客户端期望得到一个答复，那么对服务器的调用必须同步，如图5-2所示。如果客户端不需要答复，那么对服务器的调用可以异步。当你封装同步和异步调用到一个函数调用的时候，异步调用常常返回基元ok，表明该请求已发送到服务器。同步调用则会返回客户端的预期值。这些返回值通常遵循格式ok、{ok, Result}或者{error, Reason}。

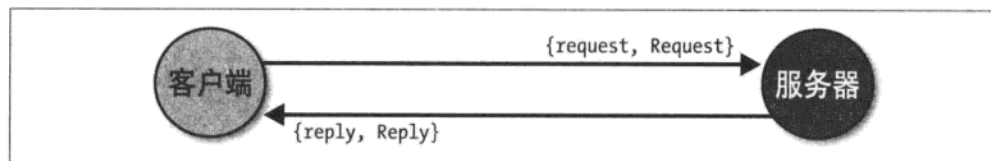


图5-2：同步客户端/服务器请求

客户端/服务器的实例

我们现在有足够的理论知识了！为了让你理解我们所讨论的，先来看一个客户端/服务器的例子，并在终端中进行测试。这个服务器帮助它的客户端负责管理电波频段以及连接移动电话到网络。电话连接的时候需要一个频段，然后在每次通话结束后释放它（见图5-3）。

当移动电话需要连接到另一个用户的时候，它调用客户端函数`frequency:allocate()`。这个调用会产生一个发送到服务器的同步消息。服务器端处理这个消息并可能回应一个包含可用频段的消息，但是如果当前所有的频段都占用的时候，它就会回应一个错误。因此`allocate/0`调用的结果要么是`{ok, Frequency}`，要么是`{error, no_frequencies}`。

通过函数接口我们隐藏了消息传递机制，以及这些消息的格式和实际的频段服务器作为注册Erlang进程来实现的这么一个事实。如果我们把服务器移动到远程主机上，那么就不需要更改客户端接口了。

当客户端完成了通话并准备释放这个连接的时候，它需要释放频段以使其他客户端可以重新使用它。这可以通过调用客户端函数`frequency:deallocate(Frequency)`实现。调用结果将作为一个消息发送给服务器。然后服务器可以把这个频段提供给其他客户并回

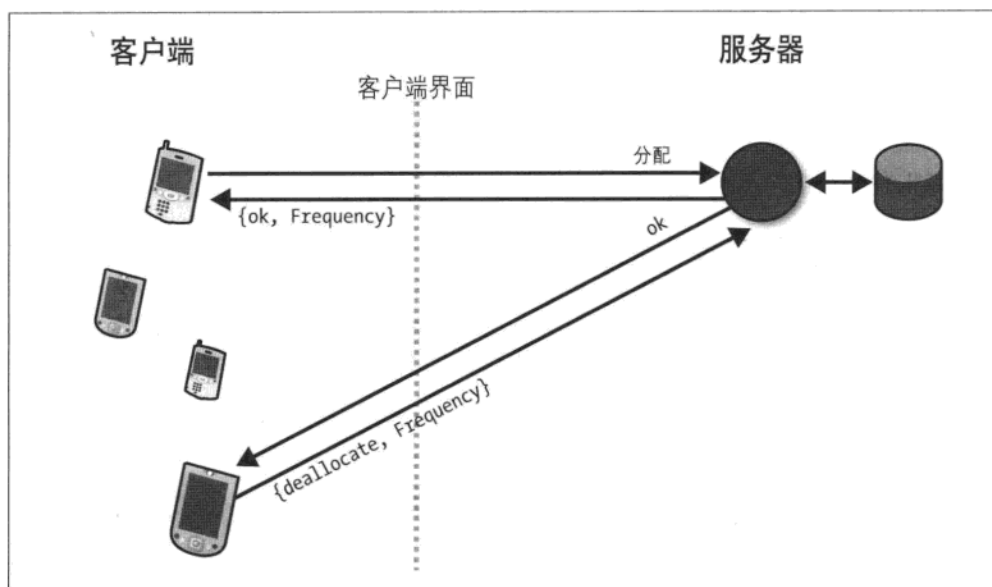


图5-3: 频段服务器

应基元`ok`。这个基元发送回客户端，并成为函数`deallocate/1`调用的返回值。图5-4显示了这个例子的消息序列图。

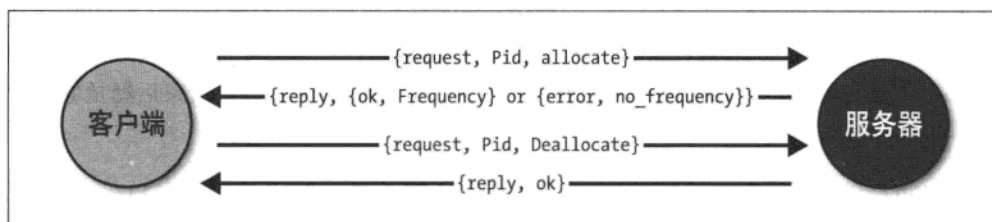


图5-4: 频段服务器消息序列图

服务器的代码定义在`frequency`模块中。下面是它的第一部分：

```
-module(frequency).
-export([start/0, stop/0, allocate/0, deallocate/1]).
-export([init/0]).

%% These are the start functions used to create and
%% initialize the server.

start() ->
    register(frequency, spawn(frequency, init, [])).
```

```

init() ->
    Frequencies = {get_frequencies(), []},
    loop(Frequencies).

% Hard Coded
get_frequencies() -> [10,11,12,13,14,15].

```

start函数创建一个新进程，它执行frequency模块中的init函数。spawn返回一个进程标识符，它是作为内置函数register的第二个参数进行传递的。第一个参数是基元frequency，这是注册进程的别名。这遵循了使用定义所在的模块的名称来注册一个进程的习惯约定。

注意：请记住，当创建一个新进程的时候，你必须导出init/0函数，因为内置函数spawn/3需要使用到它。我们已经把这个函数放在一个单独的export语句中，以区分应该从其他模块调用的客户端函数。你可以在代码的任何地方调用frequency:init()，但我们认为这不是好的做法，请避免使用。

新创建的进程从init函数开始运行。它创建了一个包含可以通过调用get_frequencies/0进行检索的可用频段，可以通过调用get_frequencies/0来获取可用频段的元组，还有一个在服务器刚刚启动时已分配频段的列表——起初是作为空列表给出的。我们称为状态或者循环数据的元组绑定到Frequencies变量上，然后作为参数传递给接收求值函数，这个函数在我们的这个例子中称为loop/1。

在init/0函数中，为了便于阅读我们使用了变量Frequencies，但是我们也可以直接调用loop/1来创建元组，比如调用loop({get_frequencies(), []})。

下面来看看客户端函数是怎么实现的：

```

%% The client Functions

stop()      -> call(stop).
allocate()   -> call(allocate).
deallocate(Freq) -> call({deallocate, Freq}).

%% We hide all message passing and the message
%% protocol in a functional interface.

call(Message) ->
    frequency ! {request, self(), Message},
    receive
        {reply, Reply} -> Reply
    end.

```

客户端和监控进程（注1）可以使用客户端函数和频段服务器进行交互。这些导出函数

注1：我们将在第6章详细讨论监控进程。

包括start、stop、allocate和deallocate。它们调用call/1函数，把传递的消息作为一个参数发送给服务器。这个函数会把服务器和它的客户端之间的消息协议封装，发送格式为{request, Pid, Message}的消息。基元request是元组中的标签，Pid是调用进程的进程标识符（在调用进程中通过调用内置函数self()得到），而Message是原来传递到函数call/1的参数。

当消息发送到进程的时候，客户端在receive语句中暂停等待一个格式为{reply, Reply}的响应，其中基元reply是一个标签，而变量Reply是事实上的响应。服务器的响应是模式匹配的，而变量Reply的内容成为客户端函数的返回值。

请特别注意消息是如何传递的和消息协议是如何抽象为独立于活动的一种格式，这其实就是我们前面提到过的消息隐藏，它允许我们对协议的细节和消息结构进行修改，而不会影响任何客户端的代码。

我们已经介绍了启动代码和与频段服务器互动的代码，现在来看看它的接收-求值循环：

```
%% The Main Loop

loop(Frequencies) ->
  receive
    {request, Pid, allocate} ->
      {NewFrequencies, Reply} = allocate(Frequencies, Pid),
      reply(Pid, Reply),
      loop(NewFrequencies);
    {request, Pid, {deallocate, Freq}} ->
      NewFrequencies = deallocate(Frequencies, Freq),
      reply(Pid, ok),
      loop(NewFrequencies);
    {request, Pid, stop} ->
      reply(Pid, ok)
  end.

reply(Pid, Reply) ->
  Pid ! {reply, Reply}.
```

receive语句可以接收来自客户端函数的三种请求，即allocate、deallocate和stop。这些请求遵循在函数call/1中定义的相同格式，即{request, Pid, Message}。Message在表达式中模式匹配，用来确定应该执行哪些语句。它反过来又决定了调用哪些内部函数。这些内部函数会返回新的循环数据，在我们的这个例子中它由可使用和已分配频段的新列表，以及如果需要还包括一个发送回客户端的答复组成。客户端的进程标识符作为请求的一部分发送，用于识别调用进程和reply/2调用。

假设客户端要初始化一次呼叫。为此，它需要通过调用函数frequency:allocate()请求一个频段。这个函数发送一个格式为{request, Pid, allocate}的消息给频段服务器，

在`receive`声明中模式匹配第一个语句。这个消息会导致调用服务器函数`allocate(Frequencies, Pid)`，其中`Frequencies`是循环数据，它包含一个由可使用和已分配频段组成的元组。`allocate`函数（我们会马上定义）会检查是否存在任何可用的频段：

- 如果有可用的频段，它返回更新的循环数据，从可用列表中移除最近分配的频段并和进程标识符共同保存到已分配列表中。发送到客户端的答复的格式是`{ok, Frequency}`。
- 如果没有可用的频段，则返回未经改变的循环数据，并返回消息`{error, no_frequency}`。

`Reply`发送给`reply(Pid, Message)`调用，它的格式为内部的客户端/服务器消息格式，并将其发送回客户端。然后这个函数递归调用`loop/1`，而新的循环数据作为一个参数传递给它。

重新分配（`Deallocation`）也以类似的方式运行。客户端函数发送消息`{request, Pid, deallocate}`，然后在`receive`声明中匹配第二个语句。这会调用`deallocate(Frequencies, Frequency)`，`deallocate`函数把`Frequency`这个频段从已分配列表中移动到再分配频段列表里，然后返回更新过的循环数据。它发送基元`ok`回客户端，并且使用更新过的循环数据递归调用函数`loop/1`。

如果收到`stop`请求，`ok`返回到调用进程并且终止服务器，就像不再有代码可以执行一样。在前面的两个语句中，在`case`语句的最后表达式中调用`loop/1`，但是在此并不是这样的。

让我们通过实现`allocate`和`deallocate`函数来完成这个系统：

```
%% The Internal Help Functions used to allocate and
%% deallocate frequencies.

allocate([[], Allocated], _Pid) ->
    {[[], Allocated], {error, no_frequency}};
allocate([[_Freq|Free], Allocated], Pid) ->
    {[Free, [{_Freq, Pid}|Allocated]], {ok, _Freq}}.
deallocate([Free, Allocated], Freq) ->
    NewAllocated=lists:keydelete(Freq, 1, Allocated),
    [{_Freq|Free}, ?NewAllocated].
```

在`frequency`模块中定义了`allocate/2`和`deallocate/2`函数，我们把它们称为内部帮助函数：

- 如果没有可用频段，`allocate/2`则会模式匹配第一个语句，这是因为元组中

包含可用的频段列表的第一个元素是空的。因此这一语句会返回元组 {error, no_frequency} 和未改变的循环数据。

- 如果至少有一个可用的频段，则第二个语句将匹配成功。频段和客户端的进程标识符从可用列表中删除，然后移动它们到已分配频段的列表中。

更新过的频段数据由allocate函数返回。最后，deallocate会使用库函数lists:keydelete/3把最新释放的频段从已分配的列表里面删除，并续接到可用频段列表中。

这个频段分配器的例子使用了迄今为止所有我们已经讲过的关键的顺序编程和并发编程的概念。它们包括了模式匹配、递归、库函数、进程创建和信息传递。请花一些时间好好理解它们。你应该使用调试器和进程管理器来测试这个例子，看看客户端和服务端之间的消息传递协议和循环函数的顺序特性。现在让我们来看一个频段分配器运行的例子：

```
1> c(frequency).
{ok,frequency}
2> frequency:start().
true
3> frequency:allocate().
{ok,10}
4> frequency:allocate().
{ok,11}
5> frequency:allocate().
{ok,12}
6> frequency:allocate().
{ok,13}
7> frequency:allocate().
{ok,14}
8> frequency:allocate().
{ok,15}
9> frequency:allocate().
{error,no_frequency}
10> frequency:deallocate(11).
ok
11> frequency:allocate().
{ok,11}
12> frequency:stop().
ok
```

进程模式实例

现在，让我们来看看刚才描述的客户端/服务器的例子和我们在第4章中介绍的进程框架之间的相似之处。设想一个应用程序，无论是网络浏览器还是字处理器，它都可以通过一个窗口管理器控制很多同时打开的窗口。为了拥有一个真正反映并发活动的进程，为每个窗口产生一个进程是一个可行的方法。这些进程可能没有注册，因为同一类型的许多窗口可能同时并发运行。

进程创建以后，每个进程都会调用`initialize`函数，它绘制并显示窗口和内容。`initialize`函数的返回值中包括对窗口中显示的窗口小部件的引用。这些引用保存在状态变量中并用于窗口的刷新。状态变量作为一个参数传递给实现接收求值循环的尾递归函数。

在这个循环函数中，进程会等待事件的发生或者关联到管理它的窗口。这可能是用户在表格内输入或者选择菜单项，或者是一个外部进程推送需要显示的数据。每一个与这个窗口相关的事件转换成一个Erlang信息发送给这个进程。当这个进程接收到消息的时候会使用状态和信息作为参数调用`handle`函数。如果事件是在表格内输入几个按键的结果，`handle`函数可能会显示它们。如果用户选择了菜单中的一个选项，`handle`函数就会根据菜单选项选择相应的动作来执行。或者，如果事件是由于外部进程推入数据引起的，比如从网络摄像机中得到一张图片或者收到一个警报信息，那么对应的窗口小部件就会更新。在Erlang中接收这些事件视为针对所有进程的通用模式。如何处理这些事件是需要特别考虑的，而且随着进程不同而不同。

最后，如果这个进程收到`stop`消息会怎么样呢？这个消息可能来自用户选择了退出菜单选项或者单击了删除按钮，或者从窗口管理器的广播中得到这个应用程序正在关闭的通知。无论是什么原因，一个`stop`消息发送给了这个进程。当接收到它的时候，这个进程调用一个`terminate`函数，它释放所有的窗口小部件，以确保不会再显示它们。在窗口关闭之后，由于不再有代码可以执行而终止进程。

看看下面的进程部分。你能在函数`initialize/1`、`handle_msg/2`和`terminate/1`中填入所有特定的代码吗？

```
-module(server).
-export([start/2, stop/1, call/2]).
-export([init/1]).

start(Name, Data) ->
    Pid = spawn(server, init,[Data])
    register(Name, Pid), ok.

stop(Name) ->
    Name ! {stop, self()},
    receive {reply, Reply} -> Reply end.

call(Name, Msg) ->
    Name ! {request, self(), Msg},
    receive {reply, Reply} -> Reply end.

reply(To, Msg) ->
    To ! {reply, Msg}.

init(Data) ->
    loop(initialize(Data)).
```

```

loop(State) ->
  receive
    {request, From, Msg} ->
      {Reply, NewState} = handle_msg(Msg, State),
      reply(From, Reply),
      loop(NewState);
    {stop, From} ->
      reply(From, terminate(State))
  end.

initialize(...) -> ...
handle_msg(...,...) -> ...
terminate(...) -> ...

```

使用前面的框架中介绍过的通用代码，让我们最后查看一遍这个GUI例子：

- `initialize`函数绘制窗口并显示它，返回绑定到`state`变量的窗口小部件的一个引用。
- 每当事件以Erlang消息的形式到达的时候，`handle_msg`函数就会处理它。这个调用带有消息和状态参数并返回一个更新过的`State`变量。这个变量会传递给递归`loop`调用，以确保这一进程持续进行下去。任何答复都会发回最初发出请求的进程中去。
- 如果收到`stop`消息，就会调用函数`terminate`，释放所有的窗口和所有与之相关的窗口小部件。它不会调用`loop`函数，以便允许进程正常终止。

有限状态机

Erlang进程可以用来实现有限状态机。有限状态机，简称FSM，是一个由有限的状态和事件组成的模型。你可以把FSM想象成关于世界的一个模式，它包含对真实世界细节的抽象。在任何一个时间点，FSM都处于一个特定的状态。根据到来的事件和现有FSM的状态，将会发生一系列的活动和过渡到一个新状态（见图5-5）。

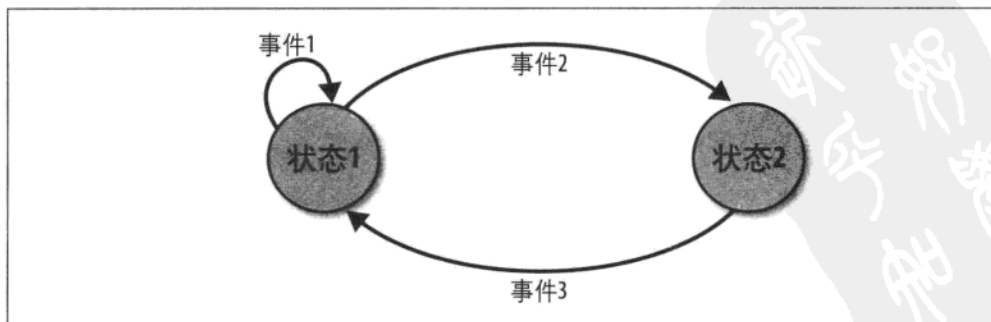


图5-5：有限状态机

在Erlang中，每个状态都表现为一个尾递归函数，每个事件又都表现为一条传入的消息。当接收到一条消息并且与receive语句匹配的时候，就会执行一系列的动作。在这些动作之后状态紧接着过渡到一个新状态。

有限状态机实例

作为一个例子，让我们把固定线路电话的建模看做一个有限状态机（见图5-6）。当等待电话打入或者等待用户提起的时候，这个电话可能处于空闲状态。如果你收到一个你姑妈的电话，电话开始振铃。一旦开始振铃，状态就从空闲转换到响铃，然后等待两个事件中的一个。你可以假装睡着了，希望你姑妈最后放弃，然后挂断电话。这将导致状态机回到空闲状态（然后你回去继续睡觉）。

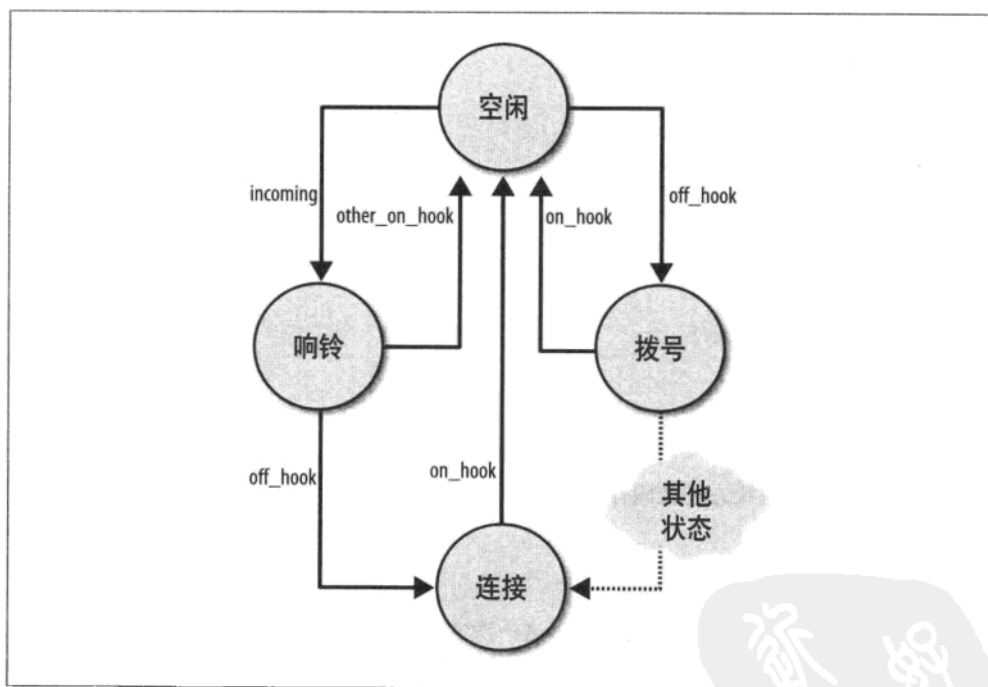


图5-6：固定线路电话有限状态机

相反，如果你不忽视它，接了电话，电话就停止振铃，然后FSM的状态就转换到连接状态，让你和姑妈通话。当你打完电话然后挂断，状态又恢复为空闲状态。

如果电话处于空闲状态，你拿起它，拨号音就开始响了。一旦拨号音开始响，FSM的状态就转换到拨号状态，然后输入你姑妈的电话号码。现在要么你可以挂断电话，FSM就会回到空闲状态，要么你姑妈接了电话，那就到了连接状态。

在各种进程应用软件中状态机很常见。在电信系统中，它们不仅可以如前面例子中介绍的一样用于处理设备的状况，而且也可以用在处理复杂的协议栈当中。事实上，Erlang能优雅地处理它们并不让人奇怪。在1987年和1991年Erlang早期版本的原型设计期间，开发团队就是使用我们本章中讲到的普通老式电话系统（POTS）有限状态机来测试他们的想法的，用来看Erlang到底应该是什么样子的。

每个状态都有一个尾递归函数，行动以函数调用的方式实现，事件使用消息表示，那么空闲状态的代码可能如下所示：

```
idle() ->
  receive
    {Number, incoming} ->
      start_ringing(),
      ringing(Number);
    off_hook ->
      start_tone(),
      dial()
  end.

ringing(Number) ->
  receive
    {Number, other_on_hook} ->
      stop_ringing(),
      idle();
    {Number, off_hook} ->
      stop_ringing(),
      connected(Number)
  end.
start_ringing() -> ...
start_tone() -> ...
stop_ringing() -> ...
```

你可以尝试编写其他状态的代码作为练习。

互斥信号量

让我们来看另一个有限状态机的例子，这一次实现一个互斥信号量。一个信号量是一个进程，它序列化访问对一个特定资源的访问，以确保相互排斥访问。互斥信号量可能不是使用Erlang的时候第一件会想到的事情，因为它们通常在共享内存的编程语言中使用。不过，它们其实可以作为一般的管理机制用来管理任何资源，而不仅仅是内存。

假设在一个时间点上只能有一个进程可以使用文件服务器，要保证没有两个进程同时对同一个文件进行读取或者写入。在作出任何文件服务器调用之前，想访问该文件的进程需要调用mutex:wait()函数给服务器上锁。当这个进程完成处理文件的时候，它调用函数mutex:signal()来移除锁（见图5-7）。

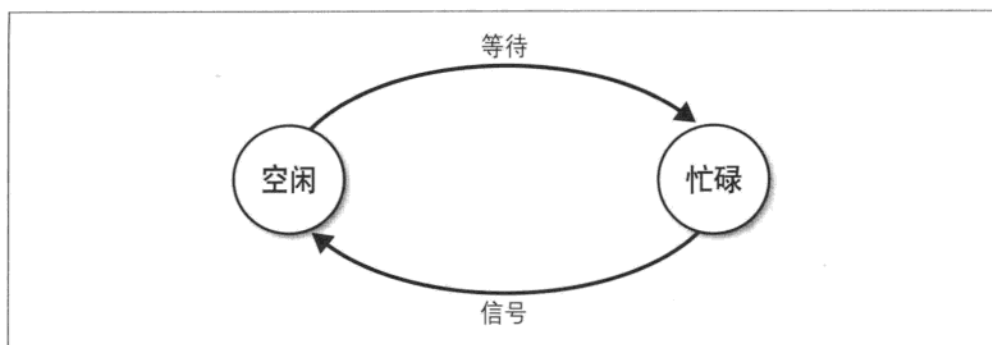


图5-7: 互斥信号量状态图

如果一个叫做PidB的进程试图调用`mutex:wait()`，而这个时候信号量正好由于进程PidA而繁忙，那么PidB会在`receive`语句暂停直到PidA调用`signal/0`。当信号量可使用了，信息队列中的第一个等待这个信息的进程，在我们的例子中就是PidB，就允许访问文件服务器了。消息序列图5-8说明了这一点。

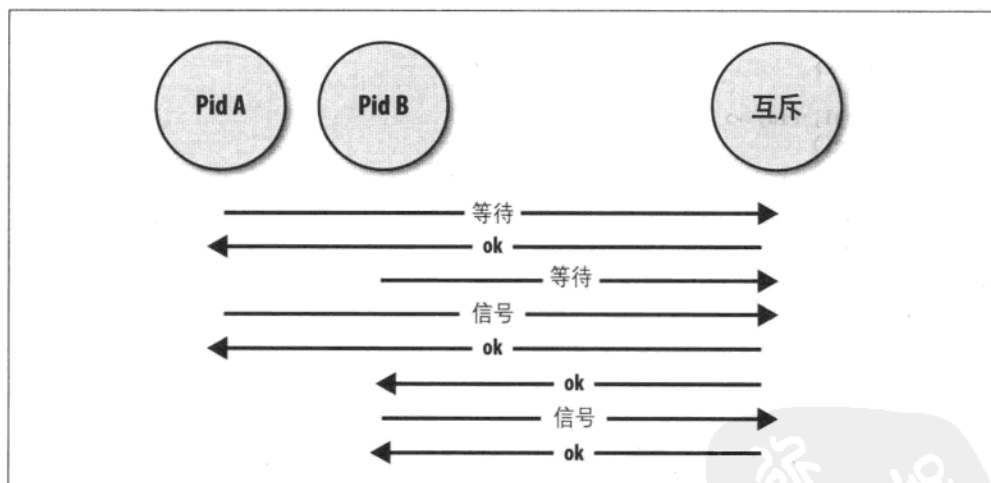


图5-8: 互斥信号序列图

看看下面的代码，了解它是如何使用尾递归函数表示状态和用信息表示事件的。在继续阅读之前，请尝试找出在互斥结束的时候`terminate`函数应该如何进行清理工作。

```
-module(mutex).
-export([start/0, stop/0]).
-export([wait/0, signal/0]).
-export([init/0]).
```



```

start() ->
    register(mutex, spawn(?MODULE, init,[])).

stop() ->
    mutex ! stop.

wait() ->
    mutex ! {wait, self()},
    receive ok -> ok end.

signal() ->
    mutex ! {signal, self()},ok.

init() ->
    free().

free() ->
    receive
        {wait, Pid} ->
            Pid ! ok,
            busy(Pid);
    stop ->
        terminate()
    end.

busy(Pid) ->
    receive
        {signal, Pid} ->
            free()
    end.

terminate() ->
    receive
        {wait, Pid} ->
            exit(Pid, kill),
            terminate()
    after
        0 -> ok
    end.

```

stop/0函数发送停止消息，在free状态就会处理它。在终止互斥进程之前，所有等待或者持有信号量的进程都允许完成它们的任务。但是，试图等待调用stop/0之后的信号量的任何进程会由terminate/0函数无条件地终止。

事件管理器和句柄

请想象下一个进程接收跟踪系统产生的事件。你可能想针对跟踪事件做很多事情，但是你没有必要在同一时间做所有这些事情。你可能想记录所有的事件到日志文件里面。如果你在控制台前面，你可能想打印输出它们到标准的I/O中。你可能对统计信息

有兴趣，想要确定某些错误发生的频率，或者根据事件的需要采取一定的行动，你可能想要发送短信或SNMP（注2）陷阱。

在某个时间，你可能不想执行所有的动作而只是想执行其中一些，然后在它们之间进行切换。当你离开办公桌的时候，你可能想关闭控制台，但是同时继续收集统计数据和保存记录到文件中。

事件管理器就可以做到我们刚才所描述的这一切。它是一个进程，它接收特定类型的事件并执行由事件类型确定的一系列行动。在进程的整个生命周期内可以动态地添加和删除这些动作，当实现进程的代码一开始编写的时候，并不一定需要定义或者知道它们。可以在我们称为事件句柄的模块中收集它们。

在大型系统中通常每一种事件类型都有一个事件管理器。事件类型通常包括警报、设备状态变化、错误和跟踪事件等。当它们接收到的时候，会对每个事件执行一个或多个动作。

在几乎所有的工业级系统里面都能发现的最常见的事件管理器形式是警报处理（见图5-9）。发生问题的时候触发警报，在问题清除时解除。它们可能自动或者需要人工干预，但这并非总是如此。如果两个设备之间的数据链路丢失了，警报就会触发，如果恢复了就会清除。其他的例子包括打开橱柜门、打破风扇或者TCP/IP连接丢失。

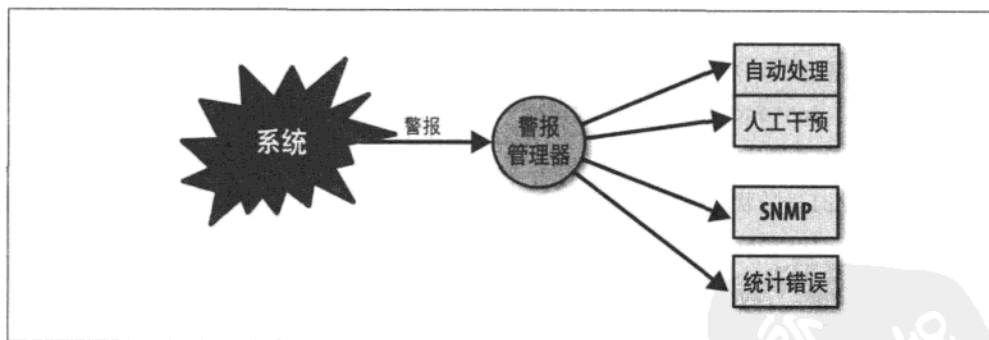


图5-9：由一个以事件句柄实现的警报管理器

警报句柄常常会记录这些警报，收集统计数据，过滤并转发给它们的代理。代理可能会接收事件并尝试自己解决这些问题。例如，如果一个通信链路断开了，一个代理就会自动尝试使用备用链路重新配置系统，只有在备用链路也发生故障的时候才会请求人的干预。

注2： SNMP是简单网络管理协议（Simple Network Management Protocol）的缩写。它是控制检测基于IP的网络系统的标准。

通用事件管理器实例

下面是一个事件管理器的例子，它允许你在运行时添加和删除句柄。这个代码是完全通用的，而且独立于个人的特定处理程序。处理程序可以在单独的模块里面实现并导出一些函数，我们称为回调函数。事件管理器可以调用这些函数。我们马上会介绍它们。让我们先来看看如何实现事件管理器，首先是它的客户端函数：

start(Name, HandlerList)

启动一个通用事件管理器，使用别名Name登记。HandlerList是格式为{Handler, Data}的一个元组列表，而其中Handler是句柄回调模块的名称，Data是传递给句柄init回调函数的参数。HandlerList在启动的时候可以为空，因为句柄可以随后通过调用add_handler/2来加入。

stop(Name)

终止所有的句柄和停止事件管理器进程。它会返回一个格式为{Handler, Data}的元素列表，其中Data是各个处理程序terminate回调函数的返回值。

add_handler(Name, Handler, Data)

增加在Handler回调模块中定义的句柄，Data作为句柄init回调函数的一个参数传递。

delete_handler(Name, Handler)

在Handler回调模块中删除定义的句柄。它会调用句柄的terminate回调函数，返回值就是这个调用的返回值。如果Handler不存在，那么这个调用就会返回元组{error, instance}。

get_data(Name, Handler)

返回Handler的状态变量的内容。如果Handler不存在，那么这个调用就会返回元组{error, instance}。

send_event(Name, Event)

向所有处理程序发送Event的内容。

下面是通用的事件管理器模块代码：

```
-module(event_manager).  
-export([start/2, stop/1]).  
-export([add_handler/3, delete_handler/2, get_data/2, send_event/2]).  
-export([init/1]).  
  
start(Name, HandlerList) ->  
    register(Name, spawn(event_manager, init, [HandlerList])), ok.  
  
init(HandlerList) ->
```

```

loop(initialize(HandlerList)).

initialize([]) -> [];
initialize([{{Handler, InitData}|Rest}] ->
  [{{Handler, Handler:init(InitData)}|initialize(Rest)}].

```

下面是对代码的具体解释：

- `start(Name, HandlerList)`函数创建事件管理器进程并以Name注册别名。
- 在`init/1`函数中开始运行新创建的进程，它需要格式为`{{Handler, Data}}`的`HandlerList`元组列表作为一个参数。
- 我们通过在`initialize/1`函数中调用`Handler:init(Data)`来遍历列表。
- 这个调用的结果存储在一个格式为`{{Handler, State}}`的列表中，其中State是`init`函数的返回值。
- 这个列表作为一个参数传递给事件管理器的`loop/1`函数。

当停止事件管理进程的时候，我们发出一个`stop`消息，然后在`loop/1`函数中接收它。你可以在模块的最后发现`loop/1`函数及其通用代码。收到`stop`消息后会引起`terminate/1`遍历句柄列表，并对每一个调用`Handler:terminate(Data)`。这些调用的返回值是格式为`{{Handler, Value}}`的一个列表，它会发送回最开始调用`stop/1`的进程中，然后成为这个函数的返回值：

```

stop(Name) ->
  Name ! {stop, self()},
  receive {reply, Reply} -> Reply end.

terminate([]) -> [];
terminate([{{Handler, Data}|Rest}] ->
  [{{Handler, Handler:terminate(Data)}|terminate(Rest)}].

```

现在，我们来看看用来添加、删除和检查事件句柄以及向它们转发事件的客户端函数。通过`call/2`函数可以发送请求给事件管理器进程，它会在`handle_msg/2`函数中处理它们。要特别注意`send_event/2`调用，它遍历句柄列表，调用回调函数`Handler:handle_event(Event,Data)`。这个调用的返回值取代了旧的Data，当下一次调用它的一个回调函数的时候，句柄就会使用它：

```

add_handler(Name, Handler, InitData) ->
  call(Name, {add_handler, Handler, InitData}).

delete_handler(Name, Handler) ->
  call(Name, {delete_handler, Handler}).

get_data(Name, Handler) ->
  call(Name, {get_data, Handler}).

```

```

send_event(Name, Event) ->
    call(Name, {send_event, Event}).

handle_msg({add_handler, Handler, InitData}, LoopData) ->
    {ok, [{Handler, Handler:init(InitData)}|LoopData]};

handle_msg({delete_handler, Handler}, LoopData) ->
    case lists:keysearch(Handler, 1, LoopData) of
        false ->
            {{error, instance}, LoopData};
        {value, {Handler, Data}} ->
            Reply = {data, Handler:terminate(Data)},
            NewLoopData = lists:keydelete(Handler, 1, LoopData),
            {Reply, NewLoopData}
    end;

handle_msg({get_data, Handler}, LoopData) ->
    case lists:keysearch(Handler, 1, LoopData) of
        false -> {{error, instance}, LoopData};
        {value, {Handler, Data}} -> {{data, Data}, LoopData}
    end;

handle_msg({send_event, Event}, LoopData) ->
    {ok, event(Event, LoopData)}.

event(_Event, []) -> [];
event(Event, [{Handler, Data}|Rest]) ->
    [{Handler, Handler:handle_event(Event, Data)}|event(Event, Rest)].

```

连同我们已经谈过的start和stop函数，下面的代码是一个直接取自进程模式的例子。现在，你应该已经发现了反复出现的主题，即通过遵循一个模式，进程以类似的方式处理完全不同的任务：

```

call(Name, Msg) ->
    Name ! {request, self(), Msg},
    receive {reply, Reply} -> Reply end.

reply(To, Msg) ->
    To ! {reply, Msg}.

loop(State) ->
    receive
        {request, From, Msg} ->
            {Reply, NewState} = handle_msg(Msg, State),
            reply(From, Reply),
            loop(NewState);
        {stop, From} ->
            reply(From, terminate(State))
    end.

```



事件句柄

在我们事件管理器的实现当中，我们的事件句柄必须导出以下三个回调函数：

`init(InitData)`

初始化句柄并返回一个值，在下次调用属于该句柄的回调函数时，将会使用这个返回值。

`terminate(Data)`

允许句柄进行清理。如果我们已经在回调函数`init/1`打开文件或者套接字，它们也将在这里关闭。`terminate/1`的返回值会传递回原来引起句柄删除的函数。在我们的事件管理器例子中，它们是`delete_handler/2`和`stop/1`调用。

`handle_event(Event, Data)`

当事件通过调用`send_event/2`传递到事件管理器的时候就会调用它。在下次调用该句柄的回调函数时，将会使用它的返回值。

现在使用这些回调函数来写两个句柄，一个在终端格式良好地输出事件，另一个记录事件到日志文件中。

这个`io_handler`事件句柄过滤出格式为`{raise_alarm, Id, Type}`和`{clear_alarm, Id, Type}`的事件，而忽略所有其他事件。在`init/1`函数中，我们设置一个计数器，当每次处理一个事件的时候它就会递增。

当每一次收到报警事件的时候，`handle_event/2`回调函数就会使用这个计数器，然后和报警信息一起在警报中显示：

```
-module(io_handler).
-export([init/1, terminate/1, handle_event/2])

init(Count) -> Count.

terminate(Count) -> {count, Count}.

handle_event({raise_alarm, Id, Alarm}, Count) ->
    print(alarm, Id, Alarm, Count),
    Count+1;
handle_event({clear_alarm, Id, Alarm}, Count) ->
    print(clear, Id, Alarm, Count),
    Count+1;
handle_event(Event, Count) ->
    Count.

print(Type, Id, Alarm, Count) ->
    Date = fmt(date()), Time = fmt(time()),
    io:format("#~w,~s,~s,~w,~w,~p~n",
              [Count, Date, Time, Type, Id, Alarm]).
```

```

fmt({AInt,BInt,CInt}) ->
  AStr = pad(integer_to_list(AInt)),
  BStr = pad(integer_to_list(BInt)),
  CStr = pad(integer_to_list(CInt)),
  [AStr,$:,BStr,$:,CStr].

pad([M1]) -> [$0,M1];
pad(Other) -> Other.

```

我们实现的第二个句柄把所有格式为{EventType,Id, Description}的事件记录到以逗号分隔的文件中，而忽略所有大小不为3的元组。

在init/1函数中打开文件，在handle_event/2中写入，并在terminate函数中关闭它。由于其他程序可能会读取和修改这个文件，我们会在写入的信息里面提供更多的细节，而在格式化方面花较少的精力。取代time()和date()，我们使用now()内置函数提供一个精确程度要高得多的时间戳。它返回一个包含从1970年1月1日开始算起的已经过去的毫微秒、秒和微秒的元组。当从事件管理器中删除log_handler的时候，调用terminate/2就可以关闭这个文件：

```

-module(log_handler).

-export([init/1, terminate/1, handle_event/2]).

init(File) ->
  {ok, Fd} = file:open(File, write),
  Fd.

terminate(Fd) -> file:close(Fd).

handle_event({Action, Id, Event}, Fd) ->
  {MegaSec, Sec, MicroSec} = now(),
  Args = io:format(Fd, "~w,~w,~w,~w,~w,~p~n",
    [MegaSec, Sec, MicroSec, Action, Id, Event]),
  Fd;
handle_event(_, Fd) ->
  Fd.

```

请在终端中尝试我们已经实现的事件管理器和两个句柄。我们使用log_handler来启动事件管理器，在这之后我们添加和删除io_handler。在这期间，我们产生一些警报和测试在事件管理器中实现的其他客户端函数：

```

1> event_manager:start(alarm, [{log_handler, "AlarmLog"}]).
ok
2> event_manager:send_event(alarm, {raise_alarm, 10, cabinet_open}).
ok
3> event_manager:add_handler(alarm, io_handler, 1).
ok
4> event_manager:send_event(alarm, {clear_alarm, 10, cabinet_open}).
#1,2009:03:16,08:33:14,clear,10,cabinet_open

```

```
ok
5> event_manager:send_event(alarm, {event, 156, link_up}).
ok
6> event_manager:get_data(alarm, io_handler).
{data,2}
7> event_manager:delete_handler(alarm, stats_handler).
{error,instance}
8> event_manager:stop(alarm).
[{io_handler,{count,2}}, {log_handler,ok}]
```

练习

练习5-1：一个数据库服务器

编写一个数据库服务器，把一个数据库存储到它的循环数据中。你应该注册服务器，然后通过函数接口来访问它的服务。模块my_db.erl的导出函数应该包括：

```
my_db:start() => ok.
my_db:stop() => ok.
my_db:write(Key, Element) => ok.
my_db:delete(Key) => ok.
my_db:read(Key) => {ok, Element} | {error, instance}.
my_db:match(Element) => [Key1, ..., KeyN].
```

提示：使用db.erl模块作为后端程序和使用第4章练习4-1的echo服务器的代码作为服务器框架。例如：

```
1> my_db:start().
ok
2> my_db:write(foo, bar).
ok
3> my_db:read(baz).
{error, instance}
4> my_db:read(foo).
{ok, bar}
5> my_db:match(bar).
[foo]
```

练习5-2：更改频段服务器

使用本章中的频段服务器例子，修改代码以确保只有分配了频段的客户才能释放它。确保当释放一个尚未分配的频段的时候，不会使服务器崩溃。

提示：在客户端调用的allocate和deallocate函数中使用内置函数self()。

扩展频段服务器，只有当没有频段可以分配的时候它才可以停止。

最后，测试你的改变，看看它们是否仍然允许个人客户分配一个频段超过一次。而这以前可以通过多次调用`allocate_frequency/0`来实现。请限制一个客户最多可以分配到3个频段。

练习5-3：交换句柄

如果你想在`log_handler`里面关闭并打开一个新的文件，这会发生什么呢？你也许应该在调用`event_manager:add_handler/2`之后紧接着调用`event_manager:delete_handler/2`。这样的风险是在这两个调用之间你可能会错过一个事件。因此，请实现以下函数：

```
event_manager:swap_handlers(Name, OldHandler, NewHandler)
```

它会自动交换句柄，确保不丢失任何事件。为了确保可以保持句柄的状态，请把`OldHandler:terminate/1`的返回值传递到`NewHandler:init/1`调用。

练习5-4：事件统计

编写一个`stats_handler`模块，在我们的例子中它带有事件元组`{Type, Id, Description}`的第一和第二个元素，并对`{Type, Description}`组合的发生次数进行统计。用户可以通过使用客户端函数`event_manager:get_data/2`来检索这些统计。

练习5-5：FSM电话

完成FSM电话例子的编程，然后使用事件句柄进程实现记录功能。应该确保记录足够多的信息，以便能够对使用手机进行计费。



进程错误处理机制

无论什么样的编程语言，构建分布式、容错、可扩展且具有高可靠性的系统都不是一件简单的事情。在处理系统的容错性和高可用性方面，Erlang有着它自己的独到之处，其根本原因是Erlang系统拥有构建于语言并发模型内的简单且功能强大的各种构造。这些构造允许进程相互监控并从软件故障中恢复。它们使Erlang相对于其他编程语言更具有竞争优势，因为通过隔离错误，确保不间断运行而提供的容错性，方便了复杂架构的开发。使用其他的编程语言尝试开发一个类似的框架要么失败，要么碰到一个巨大的难题，这是因为它缺少本章要描述的这种结构。

进程链接和退出信号

你可能听说过“让它崩溃，让别人来处理它”和“早崩溃”的办法。这就是Erlang的方式！如果出现错误，那么让你的进程尽快终止而让其他进程来处理。其他进程可以使用内置函数link/1监控、检测异常终止和对它们进行常用的处理。

内置函数link/1需要一个进程标识符作为参数，它会在调用进程和进程标识符所表示的进程之间建立一个双向联系。内置函数spawn_link/3会创建紧跟着调用spawn/3，这和调用link/1的结果是相同的，除了它是基元的（即调用在一个步骤里发生，因此调用要么都成功要么都失败）。在Erlang进程的图解中，你可以使用连线来表示进程之间的相互链接，如图6-1所示。

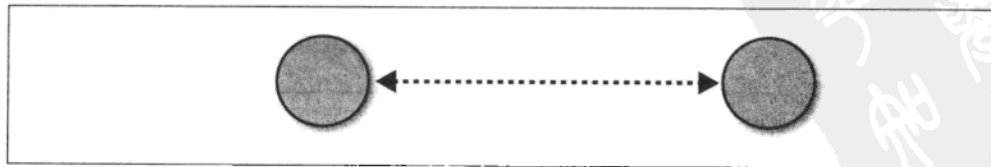


图6-1：链接进程

因为联系是双向的，所以进程A与B相连或进程B与A相连是无关紧要的，其结果都是相同的。如果一个链接进程异常终止，退出信号将会发送到与这个终止进程相连的所有进程。接收这一信号的进程将退出，并会把一个新的退出信号传递给与现在这个进程相连的所有进程（这个集合也称为链接集）。

退出信号是具有{'EXIT', Pid, Reason}格式的元组，它包括基元'EXIT'，终止进程的进程标识符Pid和终止的原因Reason。接收端的进程将以同样的原因终止，并产生一个新的带有它自己的进程标识符的退出信号，然后向它自己的链接集传送，如图6-2所示。

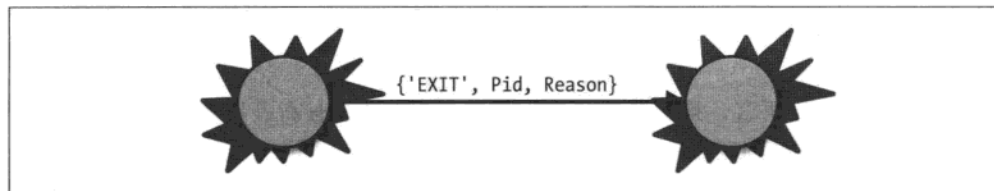


图6-2：退出信号

当进程A失败的时候，它的退出信号传递到进程B。进程B将以与A一样的理由终止，然后它的退出信号又会传递到进程C（见图6-3）。如果你在一个系统中有一个相互依赖的进程组，那么把它们链接起来是一个很好的设计实践，用以确保如果其中一个进程终止的时候，它们都会终止。

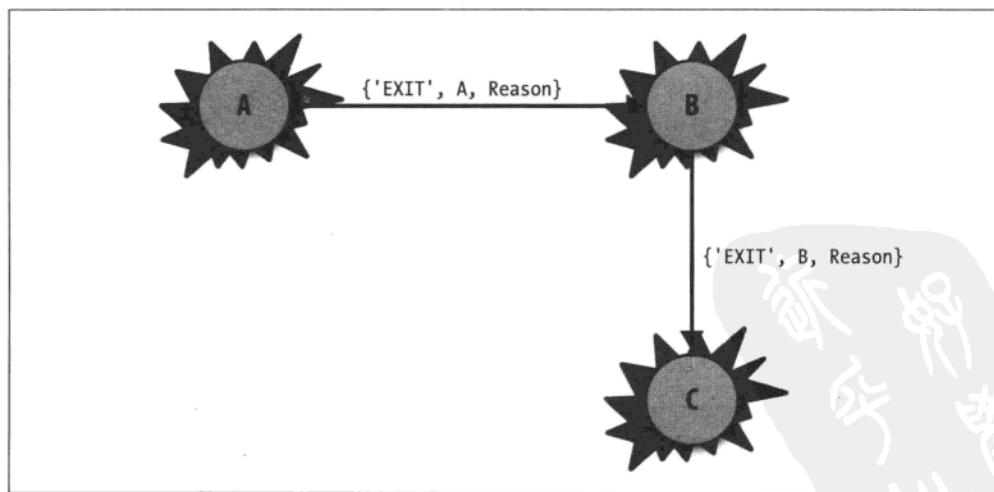


图6-3：退出信号的传播

在编辑器中输入下面的例子，或者在本书的网站下载。这是一个非常简单的程序，它生成一个与父进程相连的进程。当进程通过信息{request, Pid, N}传送一个整数N的时

候，进程就会在N上增加1并返回结果到进程标识符Pid。如果计算结果的时间超过1秒或者进程崩溃，函数request/1就会返回基元timeout。由于不会检查传递给request/1的参数，因此发送任何除了整数的参数，都会引起运行时错误，导致这个进程的终止。因为这个进程与父进程相连，所以退出信号会传送到父进程，然后以同样的原因终止：

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

start() ->
register(add_one, spawn_link(add_one, loop, [])).

request(Int) ->
add_one ! {request, self(), Int},
receive
  {result, Result} -> Result
  after 1000       -> timeout
end.

loop() ->
receive
  {request, Pid, Msg} ->
    Pid ! {result, Msg + 1}
end,
loop().
```

我们通过发送一个非整数来测试这个程序，看看终端的反应。在下面的例子中，我们发送基元one，它会导致进程崩溃。由于add_one进程与终端相连在一起，退出信号的传送也将会导致终端进程终止。在终端中显示的错误报告来自add_one进程，而输出exception exit来自终端本身。注意，在进程崩溃前和崩溃后，我们调用内置函数self()得到了不同的进程标识符，这表明已经重新启动了终端进程：

```
1> self().
<0.29.0>
2> add_one:start().
true
3> add_one:request(1).
2
4> add_one:request(one).

=ERROR REPORT==== 21-Jul-2008::16:29:38 ===
Error in process <0.37.0> with exit value: {badarith,[{add_one,loop,0}]}

** exception exit: badarith
    in function add_one:loop/0
5> self().
<0.40.0>
```

到目前为止一切顺利，现在你可能会问自己，如果当进程接收到退出信号的时候，唯一

可以做的是终止进程自身，那么它如何处理异常终止和恢复策略呢？答案是捕捉退出信号。

捕捉退出信号

进程可以通过设定进程标识`trap_exit`，然后执行函数调用`process_flag(trap_exit, true)`来捕获退出信号。通常在初始化函数中实现这个调用，它允许退出信号转换为{'EXIT', Pid, Reason}的格式。如果进程捕捉退出信号，这些消息会和其他消息一样保存在进程信箱中。你可以使用`receive`结构重新找回这些消息，当然你也可以像对待其他消息一样对它们进行模式匹配。

如果捕捉到了退出信号，它就不会再进一步传送。除了已经终止的进程，它链接集中的所有进程都不会受到影响。捕捉退出信号的进程通常标记为双圆，如图6-4所示。

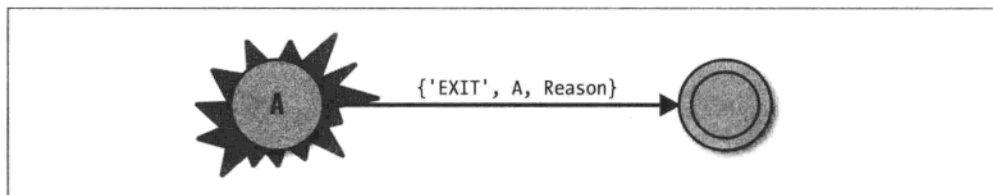


图6-4：捕捉退出信号

让我们来看一个具体的例子，如图6-5所示。标记有双圆的进程B捕捉了退出信号。如果一个运行时错误发生在进程A，它将终止并发送出一个格式为{'EXIT', A, Reason}的退出信号，其中A是失败进程的进程标识符，Reason是引起终止的原因。基元'EXIT'用来标记元组以便模式匹配。这个信号在不影响进程C的同时保存在进程B的信箱中。除非进程B明确通知进程C进程A已经终止，否则进程C永远都不会知道。

让我们重新来看看`add_one`的例子，现在让终端捕捉退出信号。结果是现在程序不再崩溃，而是把'EXIT'消息发送到终端。你可以使用`flush/0`命令来重新得到这个信号，因为它不会对`request`函数中的任何`receive`语句进行模式匹配：

```
1> process_flag(trap_exit, true).
false
2> add_one:start().
true
3> add_one:request(one).

=ERROR REPORT==== 21-Jul-2008::16:44:32 ===
Error in process <0.37.0> with exit value: {badarith,[[add_one,loop,0]]}
timeout
4> flush().
Shell got {'EXIT',<0.37.0>,{badarith,[[add_one,loop,0]]}}
ok
```

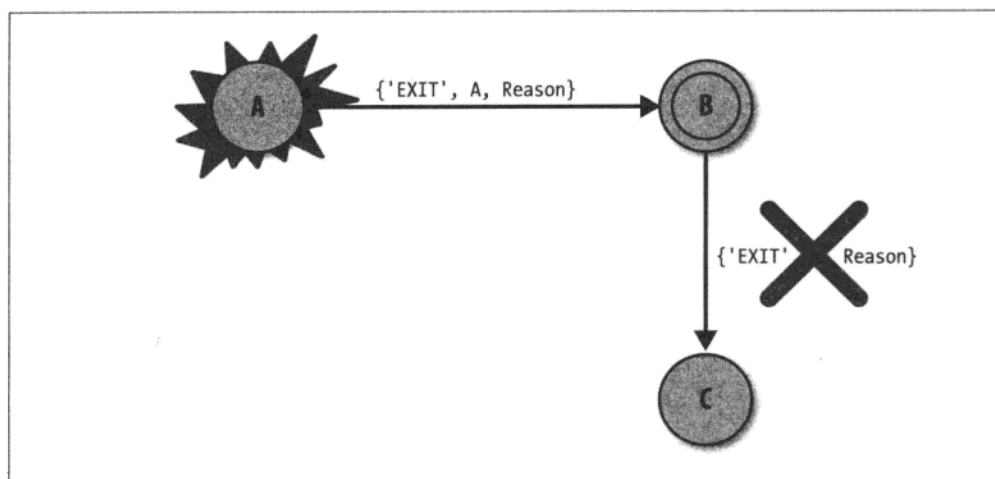


图6-5：退出信号的传播

它可以明确地捕捉发送到终端的退出消息。对于接下来这个前面介绍过的程序的变种，`request`函数有一个额外的基于{'EXIT',_,_}的模式匹配，使用它可以捕捉`loop()`进程中的退出消息：

```

-module(add_two).
-export([start/0, request/1, loop/0]).

start() ->
  process_flag(trap_exit, true),
  Pid = spawn_link(add_two, loop, []),
  register(add_two, Pid),
  {ok, Pid}.

request(Int) ->
  add_two ! {request, self(), Int},
  receive
    {result, Result}      -> Result;
    {'EXIT', _Pid, Reason} -> {error, Reason}
  after 1000
    -> timeout
  end.

loop() ->
  receive
    {request, Pid, Msg} ->
      Pid ! {result, Msg + 2}
  end,
  loop().
  
```

请注意我们如何在函数`start/0`中调用`process_flag(trap_exit, true)`。通过运行程序你应该可以看到如下的输出：

```
1> c(add_two).
{ok, add_two}
2> add_two:start().
{ok, <0.119.0>}
3> add_two:request(6).
8
4> add_two:request(six).
{error,{badarith,[{add_two,loop,0}]}}

=ERROR REPORT==== 24-Aug-2008::18:59:30 ===
Error in process <0.36.0> with exit value: {badarith,[{add_two,loop,0}]}
```

终端中命令行4的响应来自当`loop()`碰到基元`six`的时候匹配失败，然后对`{'EXIT',_,_}`进行匹配。

如果你想停止捕捉退出信号，可以使用`process_flag(trap_exit, false)`。切换`trap_exit`标签被认为是一种不好的软件实践，因为它使你的程序难以调试和维护。当一个进程生成的时候将`trap_exit`标签默认设置为`false`。

监控内置函数

链接是双向的。为了单向监控进程，可以把内置函数`erlang:monitor/2`添加到Erlang中。你可以如下调用：

```
erlang:monitor(process, Proc)
```

这样就生成了一个调用进程到另一个标记为`Proc`进程的监控进程，`Proc`可以是进程标识符也可以是注册的名字。当带有进程标识符的进程终止的时候，消息`{'DOWN', Reference, process, Pid, Reason}`会发送到监控进程。这个消息包含一个对监控进程的引用。我们将在第9章中对引用做更详细的讲解，它是（本质上）唯一的值，你可以使用它来识别实体，例如对特定请求的响应；你也可以比较引用的相等性，把它应用在模式匹配的定义中。

如果你尝试链接到不存在的进程，这个调用进程会因为运行时错误而终止。内置函数`monitor`的行为有所不同。如果调用`monitor`的时候`Pid`已经终止（或者根本不存在），则会立即发送带有`Reason`是`noproc`的`'DOWN'`消息。重复调用`erlang:monitor(process, Pid)`会返回不同的引用，从而建立多个独立监控。当`Pid`终止的时候它们都将发送出自己的`'DOWN'`消息。

监控进程可以通过调用`erlang:demonitor(Reference)`删除。在调用`demonitor`前`'DOWN'`信息可能已经发送，因此使用监控进程时不应该忘记刷新它的信箱。为了保险起见，你可以使用`erlang:demonitor(Reference, [flush])`，它在关闭监控的同时会删除由`Reference`提供的所有`'DOWN'`信息。

在下面的例子中，我们生成一个会立刻崩溃的进程，这是因为应该执行的模块并不存在。我们开始监控它，并立即收到'DOWN'的信息。当得到这个信息的时候，我们对Reference和Pid两者进行模式匹配，返回终止的原因。请注意原因是noprocs，它表明进程Pid不存在：请把这个和当我们试图链接到一个不存在的进程的时候得到的错误相比较：

```
1> Pid = spawn(crash, no_function, []).

=ERROR REPORT==== 21-Jul-2008::15:32:02 ===
Error in process <0.32.0> with exit value: {undef, [{crash,no_function,[]}]}

<0.32.0>
2> Reference = erlang:monitor(process, Pid).
#Ref<0.0.0.31>
3> receive
    {'DOWN', Reference, process, Pid, Reason} -> Reason
end.
noprocs
4> link(Pid).
** exception error: no such process or port
    in function link/1
        called as link(<0.32.0>)
```

什么时候你应该选择监控而不是选择链接呢？建立链接是永久的，如在监控树型结构中，或者当你希望退出信号的传送是双向的时候。监控器用来监测客户端调用的行为进程是比较理想的，如链接到另外一个进程而造成客户端中断，你不希望影响你所调用的进程状态，也不希望使其接收一个退出信号的时候。

内置函数exit

内置函数调用exit(Reason)来使调用进程终止，并把终止原因Reason作为参数传递给内置函数。终止进程会产生退出信号并发送到所有和它相连的进程。如果exit/1在try...catch构造（见图6-6）中调用，则可以在catch中捕获它。

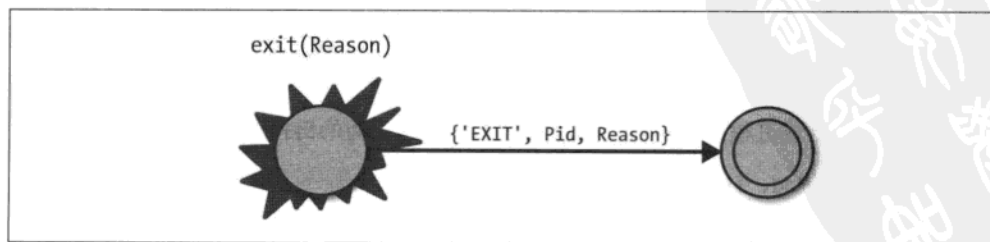


图6-6：内置函数exit/1

如果你想发送一个退出信号到一个特定的进程，可以通过使用exit(PidB, Reason)来调

用内置函数`exit`，如图6-7所示。它的结果和进程发送终止信息到其链接集几乎相同，除了`{'EXIT', PidA, Reason}`中的`Pid`是接收进程本身，而不是终止进程。如果接收进程捕获到退出信号，信号会转换为退出消息并保存到进程的信箱中。如果进程不捕获退出信号且原因不是`normal`，那么它将终止并产生和传递一个退出信号。发送给一个进程的退出信号不能通过`catch`捕捉，它会导致该进程终止。我们将在“传送语义”小节中详细讨论错误传送。

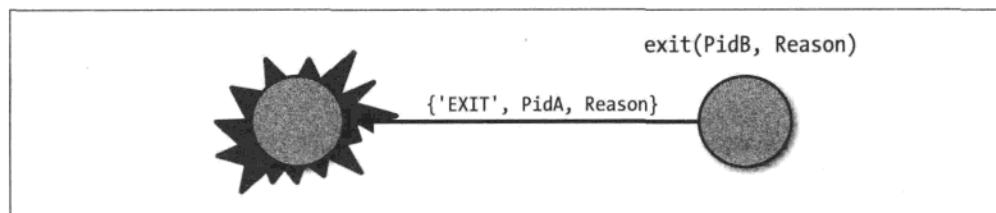


图6-7：内置函数`exit/2`

内置函数和术语

在尝试一些例子之前，先让我们更加深入地看看如下这些结构，回顾一下一些术语和最重要的终止处理的内置函数：

- 链接是在进程之间为退出信号建立的进行双向传送的路径。
- 退出信号是一个进程传输的信号，它包括表示进程结束的终止原因。
- 错误捕捉表明一个进程处理退出信号的能力，就仿佛它们只是一般消息一样。

与并发错误处理相关的内置函数包括：

`link(Pid)`

在调用进程和`Pid`之间设置一个双向链接。

`unlink(Pid)`

移除一个到`Pid`的链接。

`spawn_link(Mod, Fun, Args)`

原子性地生成进程并设置一个调用进程和新进程之间的链接。

`spawn_monitor(Mod, Fun, Args)`

原子性地生成进程并设置一个调用进程和新进程之间的监控。

`process_flag(trap_exit, Flag)`

把当前进程的退出信号转换为退出消息。`Flag`可以是基元`true`（开启捕捉退出信号）或者基元`false`（关闭捕捉退出信号）。

`erlang:monitor(process,Pid)`

生成一个针对Pid的单向监控。它返回给调用进程一个可以用来识别模式匹配的终止进程的引用。

`erlang:demonitor(Reference)`

清除监控从而使监测不再进行。不要忘记刷新信息，它可能已经在调用内置函数 `demonitor` 之前到达了。

`erlang:demonitor(Reference,[flush])`

和 `demonitor/1` 一样，但如果它是作为一个竞争条件的结果发送的时候会删除 `{_,Reference,_,_,_}` 消息。

`exit(Reason)`

导致调用进程以原因Reason终止。

`exit(Pid,Reason)`

发送一个退出信号到Pid。

即使在代码中使用 `monitor` 和 `link`，竞争条件还是可能发生。看看下面的代码片段。第一个语句生成一个把进程标识符绑定到Pid的进程，并允许运行此代码的进程链接到它：

```
link(Pid = spawn(Module, Function, Args))
```

第二条语句产生一个进程，链接到它，并绑定进程标识符到变量Pid：

```
Pid = spawn_link(Module, Function, Args)
```

乍一看，这两个例子似乎做同样的事情，除了 `spawn_link/3` 操作。原子性操作指的是操作在暂停之前必须完成。在处理并发情况的时候，这看起来是一个很小的细节，例如一个操作是否是原子性的，可以彻底改变我们编程的整个风格：

- 如果使用 `spawn_link/3`，那么将生成进程并链接到父进程。这个操作无法在 `spawn` 和 `link` 之间停止，因为所有内置函数都是基元不可分的。最早可能终止进程的时间是在执行内置函数之后。
- 如果取而代之把 `spawn` 和 `link` 作为两个单独的操作来运行，父进程在调用 `spawn` 生成进程并且绑定变量Pid之后，但在调用 `link/1` 之前，可能被暂停。新进程开始执行，遇到运行时错误并终止。父进程于是抢先执行，然后第一件事情是链接到一个不存在的进程。这将导致运行时错误，而不是接收到一个退出信号。

这个问题和我们在第4章中看到的关于竞争条件的例子很相似，它们的结果可能依赖于事件的次序，以及在何处暂停进程和在哪个核心上运行。经验法则是始终使用

`spawn_link`，除非你需要在链接和断开之间切换，或者它不是一个链接到进程的子进程。在进入下一节之前，请尝试使用内置函数`monitor`解决上述问题。

传送语义

我们已经介绍了最重要的术语和内置函数，现在来看看和退出信号相关的传送语义的细节，表6-1对它们进行了总结。当一个进程终止的时候，它会给它的链接集中的进程发送一个退出信号。这些退出信号可以是正常的或者非正常的。当没有更多的代码可以执行而使进程结束，或通过调用带有`normal`原因的内置函数`exit`的时候，产生正常退出信号。

当收到一个非正常退出信号的时候，一个没有捕捉退出信号的进程会终止。而忽略带有原因`normal`的退出信号。一个捕捉退出信号的进程会把所有进来的正常和非正常的退出信号，转化为常规的消息存储在信箱中，它们会通过`receive`语句处理。

在任何的内置函数的`exit`消息中，如果`Reason`是`kill`，不管`trap_exit`（注1）的值是什么，这个进程都将会无条件终止。一个带有`killed`原因的退出信号会传送到它的链接集。这会确保如果链接集合的任何一端无条件终止，那么也不会终止捕获退出信号的进程。

表6-1：传播语法

| Reason | 正常退出 <code>trap_exit=true</code> | 非正常退出 <code>trap_exit=false</code> |
|---------------------|-------------------------------------|--------------------------------------|
| <code>normal</code> | 收到{'Exit', Pid, normal} | 没有任何事情发生 |
| <code>kill</code> | 当Reason是 <code>kill</code> 时，则无条件终止 | 当Reason是 <code>kill</code> 时，则无条件终止 |
| <code>other</code> | 收到{'EXIT', Pid, Other} | 当Reason是 <code>other</code> 时，则无条件终止 |

健壮性系统

在Erlang中，通过分层建立健壮性系统。通过使用进程，你可以生成一棵树，而树的叶子由处理操作任务的应用层组成，树的内部节点监测它的叶子和在它们之下的其他节点，如图6-8所示。任何级别的进程都会捕捉比它们更低级别的进程的错误。如果进程的唯一任务是监控子进程，在这种情况下是指树的节点，称为监控进程（supervisor）。一个执行操作的叶进程（leaf process）称为工作进程。当提到子进程时，我们指的是监控进程和归属这个特定监控进程的工作进程。

注1： 你可以捕捉`exit(kill)`调用。只有使用`exit(Pid, kill)`的时候，才会无条件终止一个进程。

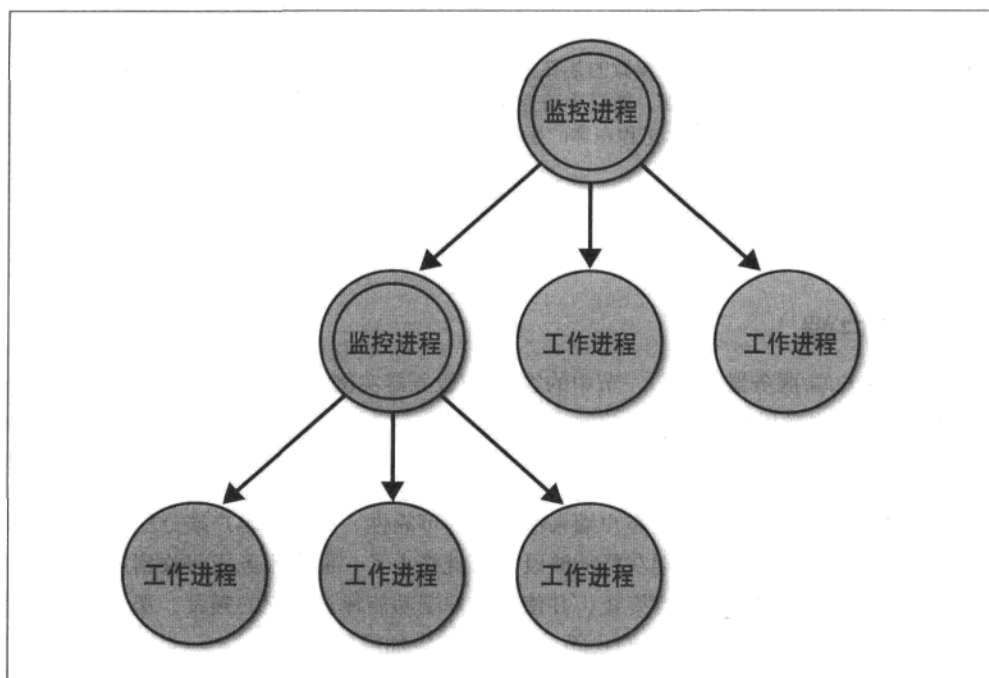


图6-8：监控树结构

在设计良好的系统中，应用程序员不需要担心错误处理部分的代码。如果个工作进程崩溃，退出信号发送到它的监控进程，监控进程会把它和系统的更高层次隔离开。基于预先设置的参数和终止的原因，监控进程决定这个工作进程是否需要重新启动。

然而监控进程可能不是唯一监控其他进程的进程。如果一个进程依赖于另一个进程，它并不一定是后者的子进程，它可以把自身和后一个进程链接。当发生异常终止的时候，这两个进程可以采取适当的行动。

在大型Erlang系统中，你永远都不应该允许有不属于监控树任何一部分的进程存在：所有的进程要么与监控进程相连，要么与其他工作进程相连。因为Erlang程序可以多年不重新启动而持续运行，即使不是数十亿，在系统运行生命周期内也可能生成数百万的进程。你需要对这些进程完全控制，并在必要的时候能关闭整个监控树。你永远不知道错误是通过什么形式表现出来的，我们想避免的是一个不正常的进程，由于它没有链接到监控进程，而最终导致我们无法对它进行终止。另外一个危险是处于暂停的进程，它可能是由于（但不限于）错误或超时，从而导致内存泄漏，这个可能需要数个月的时间去查明。

注意：想象一下在升级的时候，你不得不以特定类型的调用去终止所有进程。如果这些进程是监控树的一部分，那么你需要做的是终止高一层的监控进程，然后升级代码并重新启动。一想到我们不得不进入终端，然后手动查找并终止所有没有和父进程或者监控进程链接的进程。如果你不知道有什么进程在你的系统上运行，那么唯一可行的方法是重新启动终端，而这是违背高可用性原则的。

如果你很在乎系统的容错性和高可用性，那么请确保你所有的进程都与监控树相连。

监控客户端

还记得“客户端/服务器模型”一节中的客户端/服务器实例吗？服务器是不可靠的！如果客户端在其发出频段释放信息之前就崩溃了，服务器就不会释放此频段并且不允许其他客户再次使用。

让我们重写服务器，通过监控客户端来保证它的可靠性。当一个客户端分配了波段，服务器会链接上它。如果客户端在释放波段之前就终止了，那么服务器将收到退出信号并自动释放它。如果客户端没有终止，并使用客户端功能释放分配的频段，那么服务器会移除此链接。这里是来自第5章的代码，新的代码会加粗显示：

```
-module(frequency).
-export([start/0, stop/0, allocate/0, deallocate/1]).
-export([init/0]).

%% These are the start functions used to create and
%% initialize the server.

start() ->
    register(frequency, spawn(frequency, init, [])).

init() ->
    process_flag(trap_exit, true),
    Frequencies = {get_frequencies(), []},
    loop(Frequencies).

% Hard Coded
get_frequencies() -> [10,11,12,13,14,15].

%% The client Functions
stop()      -> call(stop).
allocate()  -> call(allocate).
deallocate(Freq) -> call({deallocate, Freq}).

%% We hide all message passing and the message
%% protocol in a functional interface.
call(Message) ->
    frequency ! {request, self(), Message},
    receive
```

```

    {reply, Reply} -> Reply
end.

reply(Pid, Message) ->
    Pid ! {reply, Message}.

loop(Frequencies) ->
    receive
        {request, Pid, allocate} ->
            {NewFrequencies, Reply} = allocate(Frequencies, Pid),
            reply(Pid, Reply),
            loop(NewFrequencies);
        {request, Pid, {deallocate, Freq}} ->
            NewFrequencies=deallocate(Frequencies, Freq),
            reply(Pid, ok),
            loop(NewFrequencies);
        {'EXIT', Pid, _Reason} ->
            NewFrequencies = exited(Frequencies, Pid),
            loop(NewFrequencies);
        {request, Pid, stop} ->
            reply(Pid, ok)
    end.

allocate([], Allocated, _Pid) ->
    {[], Allocated}, {error, no_frequencies}};
allocate([_Freq|Frequencies], Allocated, Pid) ->
    link(Pid),
    {[Frequencies,[_Freq,Pid]|Allocated]}, {ok, Freq}}.

deallocate([Free, Allocated], Freq) ->
    {value, {Freq, Pid}} = lists:keysearch(Freq, 1, Allocated),
    unlink(Pid),
    NewAllocated=lists:keydelete(Freq, 1, Allocated),
    {[Freq|Free], NewAllocated}.

exited([Free, Allocated], Pid) ->
    case lists:keysearch(Pid, 2, Allocated) of
        {value, {Freq, Pid}} ->
            NewAllocated = lists:keydelete(Freq, 1, Allocated),
            {[Freq|Free], NewAllocated};
        false ->
            {Free, Allocated}
    end.
end.

```

请注意在函数exited/2中，我们是如何保证包含客户端Pid和已经分配频段的数对的。这是为了避免潜在的竞争条件，也就是客户端正确地释放了的频段，但在服务器能够释放消息并解除自己和客户端的链接之前，客户端终止了。结果是，服务器会收到来自客户端的退出信号，即使它已经释放了此频段。

```

1> frequency:start().
true
2> frequency:allocate().
{ok, 10}

```

```
3> exit(self(), kill).
** exception exit: killed
4> frequency:allocate().
{ok,10}
```

在这个例子中，我们使用双向链接代替了单向监控。这个设计决定是基于如下的事实，如果频段服务器异常终止，我们希望所有已经被分配的频段的客户端也同样终止。

监控进程实例

监控进程的唯一任务是启动工作进程并监控。现实中它们是如何实现的呢？工作进程要么在监控进程的初始化阶段启动，要么在监控进程启动之后动态启动。一个监控进程可以捕捉退出信号，并在生成工作进程的时候链接到它们。如果一工作进程终止，那么监控进程会收到退出信号。然后监控进程可以使用工作进程在退出信号中的Pid来识别这个进程并重新启动它。

监控进程应当管理进程的终止，使用统一的方式来重新启动它们和决定采用何种行动。这些行动可能包括不做任何行动、重新启动该进程、重新启动或终止整个子树，让它的监控进程来解决这个问题。

不论系统如何，监控进程的行为应该一致。它与客户端/服务器、有限状态机和事件句柄一起，被认为是一个进程设计模式：

- 监控进程的通用行为是启动工作进程，监控它们，并在它们万一终止时重新启动。
- 监控进程的特定行为由工作进程组成，包括何时、如何启动和重新启动它们。

在下面的例子中，我们实现的监控进程需要一个以元组形式{Module, Function, Arguments}组成的子列表。通过给出启动工作进程需要调用的函数，这个列表描述了监控进程将要监控的工作进程：比如在本章开始时提到的一个例子{add_two, start, []}。我们假设子进程通过使用内置进程spawn_link/3启动，如果成功则这个函数返回元组{ok,Pid}，你可以验证这是add_two:start/0的情况。

在执行函数init/1时监控进程启动，它也是通过调用内置函数spawn_link/3开始的，但是它必须与它的父进程相连。它开始捕捉退出信号，并通过调用函数start_children/1生成所有的子进程。如果apply/3调用生成子进程成功，函数返回{ok, Pid}，那么传递到loop/1函数中的{Pid,{Module, Function, Arguments}}会添加到所生成的子进程的列表中：

```
-module(my_supervisor).
-export([start_link/2, stop/1]).
-export([init/1]).
```

```

start_link(Name, ChildSpecList) ->
  register(Name, spawn_link(my_supervisor, init, [ChildSpecList])), ok.

init(ChildSpecList) ->
  process_flag(trap_exit, true),
  loop(start_children(ChildSpecList)).

start_children([]) -> [];
start_children([_M, F, A | ChildSpecList]) ->
  case (catch apply(M,F,A)) of
    {ok, Pid} ->
      [{Pid, {M,F,A}} | start_children(ChildSpecList)];
    _ ->
      start_children(ChildSpecList)
  end
end

```

在receive语句中，监控进程循环等待EXIT和stop消息。如果一个子进程终止，那么监控进程会收到EXIT信号，并且重新启动已终止的子进程，替换其存储在子进程的ChildList变量中的列表项：

```

restart_child(Pid, ChildList) ->
  {value, {Pid, {M,F,A}}} = lists:keysearch(Pid, 1, ChildList),
  {ok, NewPid} = apply(M,F,A),
  [{NewPid, {M,F,A}} | lists:keydelete(Pid,1,ChildList)].

loop(ChildList) ->
  receive
    {'EXIT', Pid, _Reason} ->
      NewChildList = restart_child(Pid, ChildList),
      loop(NewChildList);
    {stop, From} ->
      From ! {reply, terminate(ChildList)}
  end.

```

我们通过调用同步客户端函数stop/0来停止监控进程。一旦收到stop消息，监控进程就会遍历ChildList逐个终止子进程。当所有的子进程都终止后，基元ok就会返回到发出stop调用的进程：

```

stop(Name) ->
  Name ! {stop, self()},
  receive {reply, Reply} -> Reply end.
terminate([_Pid, _ | ChildList]) ->
  exit(Pid, kill),
  terminate(ChildList);
terminate(_ChildList) -> ok.

```

在我们的例子中，监控进程和子进程相互链接。你能想到一个你不应该使用内置函数monitor的原因吗？因此选择这样设计的原因和频段服务器的那个例子很类似。如果监控进程终止，无论这听起来有多少可怕我们都希望它能终止所有的子进程！

让我们在终端中运行例子并确保它能正常工作：

```
1> my_supervisor:start_link(my_supervisor, [{add_two, start, []}]).
ok
2> whereis(add_two).
<0.125.0>
3> exit(whereis(add_two), kill).
true
4> add_two:request(100).
102
5> whereis(add_two).
<0.128.0>
```

这个监控进程的例子相对来讲比较简单。我们会在后面的练习中继续对它进行扩展。

练习

练习6-1：链接Ping Pong服务器

修改第4章的练习4-1中A和B的进程，使它们彼此链接。当调用stop函数的时候，不要发送stop信息，而是使第一个进程异常终止。这个应该导致EXIT信号向其他进程继续传送，从而导致它们也终止。

练习6-2：一个可靠的互斥信号量

假设在“有限状态机”一节中的互斥信号量不可靠。如果当前拥有信号量的进程在释放它之前就终止了，这将会发生什么呢？或者如果一个等待执行的进程由于一个退出信号而终止了，又会发生什么情况呢？通过捕捉退出信号和链接到目前持有信号量的进程，这会让你的互斥信号量变得可靠。

在这项练习的第一个版本中，当调用link(Pid)的时候请使用try ... catch。你必须使用catch封装它，以防止使用Pid标记的进程在你处理其请求前已经被终止了。

在这项练习的第二个版本中，请使用erlang:monitor(type, Item)。比较这两种解决方案的差异。其中的哪个解决方案你更喜欢呢？

练习6-3：监控进程

我们在例子中提到的监控进程是很基础的。我们想要扩展它的特征并使其能够处理更多的通用功能。在迭代测试和开发周期中，一次添加一个以下功能：

- 如果一个子进程同时正常和异常终止，监控进程会收到退出信号并重新启动子进程。我们希望扩展子进程元组{Module, Function, Argument}，让它包括一个Type

参数，它可以设置为permaent或者transient。如果子进程的Type是transient，当正常终止的时候不会重新启动它。只有当异常终止的时候它才会被重新启动。

- 当监控进程试图去生成一个子进程，而它的模块不可用的时候，将会发生什么呢？这个进程会崩溃，它的EXIT信号会发送到监控进程并立刻重新启动它。监控进程不会处理无限重新启动的情况。为了避免这种情况的发生，使用一个计数器使一个子进程每分钟最多重新启动五次。当达到这个界限的时候，把这个子进程从子进程列表中移除并送出一个错误消息。
- 你的监控进程甚至在它自己已经启动后也应该能够启动子进程。在子进程列表中添加一个唯一的标识符，实现这个函数start_child(Module, Function, Argument)，它会返回唯一的Id和子进程的进程标识符Pid。不要忘记实现stop_child(Id)调用来终止子进程。为什么当终止它们的时候，我们选择通过它的Id而不是通过Pid来确定子进程呢？

请以本章例子中的监控进程为基础。你可以从网站上下载本书的代码来开始练习。

要测试你的监控进程，请启动互斥信号量和数据库服务器进程（见图6-9）。

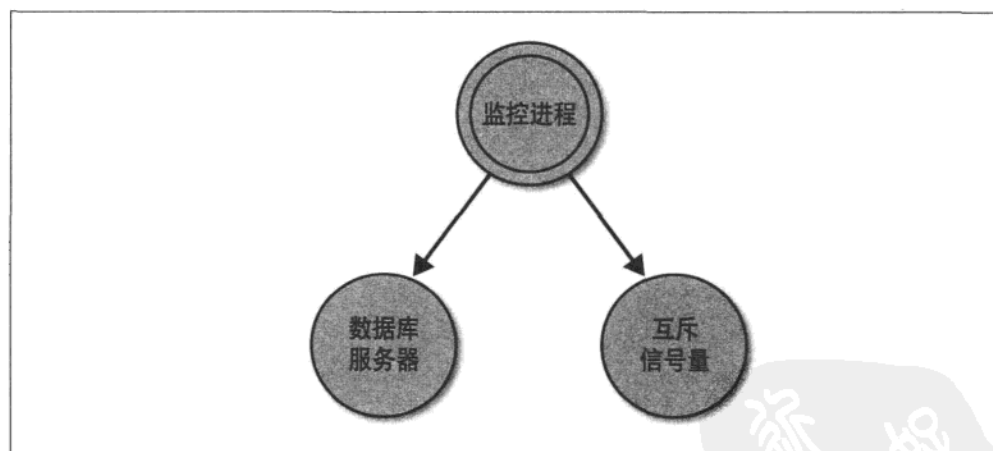


图6-9：监控树

- 你不得更改start函数，用以确保进程间的链接及其链接到它们的父进程正常，然后返回{ok,Pid}。
- 使用exit(whereis(ProcName), kill)终止你的进程。
- 通过调用whereis(ProcName)来看看此进程是否已经重新启动了，确保你每次调用得到不同的进程ID。

- 如果进程是未注册的，那么通过调用`exit(Pid, kill)`来终止它。从`start_child`函数的返回值中你可以获得进程标识符`Pid`（然后你可以启动同一类型的多个进程）。
- 一旦终止了一个进程，请调用终端中的`i()`帮助函数来检查此进程是否重新启动了。



记录和宏

一旦你的第一个Erlang产品投放到市场，部署到世界各地，那么为增强第二个版本功能的工作就应该开始了。设想一个大约有15 000行代码的程序，其大小恰巧与爱立信公司发行的第一个Erlang产品——移动服务器的代码量一样。在你的代码库中，你的元组包含和现有功能有关的数据和很多已经被硬编码的常量。当你添加新功能的时候，需要给这些元组添加新的字段。但问题是，对代码中的这些字段的更新，不仅是在你增加功能的地方，还有在15 000行代码中其他你没有添加过它们的地方。缺少更新一个元组就会致使运行时错误。假设你的常量也需要更新，那么你需要在用到它们的每个地方更改硬编码值。这方面的消耗甚至可能比我们实现这个软件的成本还高，因为我们还需要对整个代码库进行重新测试，以确保没有新的缺陷出现或者没有遗漏更新某个地方的字段和常量。

计算机中最常见的构造之一是集合若干数据片段作为单一项。Erlang元组提供了集合数据的基本机制，但它们确实也有一些缺点，特别是当大量的数据项集合作为一个对象时。在本章的前一部分，你将学习到有关记录的知识，记录将能克服我们前面提到的大部分缺点，并使代码易于升级。而最关键的是，记录提供数据抽象，当访问它的时候数据的实际形式被隐藏起来了。

宏允许我们使用缩写，运行之后由Erlang预处理器来展开。使用宏可以让程序更具有可读性，方便扩展语言和编写调试代码。最后，我们通过描述include声明语句来结束本章，它引入的头文件包含了我们在Erlang项目中要使用到的记录和宏的定义。

虽然这两种构造不是Erlang程序所必需的，但它们使程序更易于阅读、修改和调试，增强了源代码功能和支持部署产品。显然，在爱立信的移动服务器投入应用，开发人员开始技术支持并开发后续的增强特性之后不久，本章所描述的这两个构造——记录和宏就添加到了Erlang语言中，这并非巧合，

记录

为了解记录的优势，我们先来看一个关于people信息的小例子。假设你要存储关于一个人的基本信息，包括他的姓名、年龄和电话号码。你可以使用由三个元素组成的元组{Name, Age, Phone}:

```
-module(tuples1).
-export([test/1, test/2]).

birthday({Name, Age, Phone}) ->
    {Name, Age+1, Phone}.

joe() ->
    {"Joe", 21, "999-999"}.

showPerson({Name, Age, Phone}) ->
    io:format("name: ~p age: ~p phone: ~p~n", [Name, Age, Phone]).

test1() ->
    showPerson(joe()).

test2() ->
    showPerson(birthday(joe())).
```

在程序中每次使用到person的时候，它必须以一个完整的元组出现：{Name, Age, Phone}。如果加入新的字段，那就意味着你必须更新每个使用此元组的地方，甚至包括根本没有用到新字段的地方，虽然对于一个拥有三个元素的元组看起来没有太大的问题。但是如果你忘记更新其中一个，当模式匹配这个元组的时候就会出现运行时错误badmatch。另外，当处理大小为30甚至是10个元素的时候，元组就不能很好地扩展了，使用它产生误解和错误的可能性很大。

介绍记录

类似于C语言中的structure或者Pascal语言中的record，记录是一个有着固定数目字段的数据结构，这些字段通过名称来访问。记录不同于元组，元组的字段通过位置信息来访问。在上面person的例子中，我们可以定义一个记录类型如下：

```
-record(person, {name, age, phone}).
```

这就引入了记录类型person，每个记录实例包含如下三个名字的字段：name、age和phone。字段名称被定义为基元。下面是一个关于这种类型的记录实例：

```
#person{name="Joe",
        age=21,
        phone="999-999"}
```

在前面的代码中，`#person`是`person`记录的构造函数。在这个例子中我们列出字段的顺序和定义中的顺序碰巧相同，但这不是必需的。使用下面的表达式会得到相同的值：

```
#person{phone="999-999",
        name="Joe",
        age=21}
```

在这两个例子中，我们定义了所有的字段，但也可以在记录的定义中为字段提供默认值，如下所示：

```
-record(person, {name,age=0,phone=""}).
```

现在有一个如下给出的`person`记录：

```
#person{name="Fred"}
```

它包含年龄值为0和一个为空的电话号码，但一个默认值都没有给出的时候，“默认的默认”是基元`undefined`。

一个包含`field1`到`fieldn`字段的`name`记录的一般定义采取如下格式：

```
-record(name, {field1 [ = default1 ],
               field2 [ = default2 ],
               ...
               fieldn [ = defaultn ] })
```

方括号中的部分是可选字段默认值的声明。相同的字段名称可以用于一个以上的记录类型，事实上，两个记录可以共享相同的名字列表。记录的名称只可用于一个定义，不管怎样它是用来识别记录的。

操作记录

假设你得到一个记录值。那么如何读取这个记录的字段，如何描述修改过的记录呢？让我们看看下面的例子：

```
Person = #person{name="Fred"}
```

你可以这样读取记录的字段：`Person#person.name`、`Person#person.age`，等等。它们的值会是什么呢？访问字段的一般形式是：

```
RecordExp#name.fieldName
```

其中的`name`和`fieldName`不能是变量，而`RecordExp`是一个用来表明这是一个记录的表达式。通常这是一个变量，但它也可能是一个应用程序函数的结果或对另一个记录类型的字段访问。

假设你要修改一个记录的一个字段。你可以这样直接写：

```
NewPerson = Person#person{age=37}
```

在这种情况下记录语法的真正优势就体现出来了。你只需提到需要修改值的字段，那些从Person到NewPerson没有改变的字段值就不需要在定义中说明了。事实上，记录机制允许更新任何选择字段，例如：

```
NewPerson = Person#person{phone="999-999",age=37}
```

修改记录字段的一般形式是：

```
RecordExp#name{..., fieldNamei=valuei, ... }
```

字段更新可以以任何顺序出现，但每个字段名称最多只能出现一次。

基于记录的函数和模式匹配

基于记录的模式匹配可以提取字段值和影响计算的控制流程。假设你想定义birthday函数，给这个人的年龄增加1。你就可以使用字段选择，然后更新定义这个函数：

```
birthday(P) ->  
  P#person{age = P#person.age + 1}.
```

如果我们使用模式匹配显然会更简单明了：

```
birthday(#person{age=Age} = P) ->  
  P#person{age=Age+1}.
```

从前面的代码中我们可以看到，这个函数应用到person记录，并提取age字段到变量Age中。它也可以和字段值匹配，让你只增加Joe的年龄，而保持其他人的年龄不变：

```
joesBirthday(#person{age=Age,name="Joe"} = P) ->  
  P#person{age=Age+1};  
joesBirthday(P) -> P.
```

重温一下本节开头的例子，你可以使用记录给出如下定义：

```
-module(records1).  
-export([birthday/1, joe/0, showPerson/1]).  
  
-record(person, {name,age=0,phone}).  
  
birthday(#person{age=Age} = P) ->  
  P#person{age=Age+1}.
```

```
joe() ->
  #person{name="Joe",
    age=21,
    phone="999-999"}.

showPerson(#person{age=Age,phone=Phone,name=Name}) ->
  io:format("name: ~p age: ~p phone: ~p~n", [Name, Age, Phone]).
```

虽然用在这里的符号有些冗长，但这能更清楚地理解代码，它把我们操作`person`记录的用意以及有关的详细信息明确地表达了出来：从`birthday`定义中我们可以清楚看到，它只对`age`字段进行操作，而不改变其他字段。最后，如果想改变或扩展记录的组成部分，其代码更容易修改；在本章的结尾部分的第一个练习中，你可以证实它们。

记录字段可以包含任何有效的Erlang的数据类型。因为记录是有效的数据类型，所以字段也可以包含其他的记录，这就产生了嵌套记录。例如，在一个`person`记录中，`name`字段的内容本身也可以是一个记录：

```
-record(name, {first, surname}).

P = #person{name = #name{first = "Robert",
                        surname = "Virding"}}
First = (P#person.name)#name.first.
```

此外，嵌套字段的字段选择可以通过一个单一的表达式给出，就如前面的`First`定义一样。

终端中的记录

Erlang中的记录只是在编译期有效，在虚拟机中没有它们的类型。正因为如此，终端对它们的处理与对其他构造的处理不同。

在终端中可以使用命令`rr(moduleName)`来加载模块`moduleName`中的所有记录定义。或者，你可以在终端中通过使用命令`rd(name, {field1, field2, ... })`来直接定义包含有`field1`、`field2`等的`name`记录。这个对测试和调试，或者当你无权访问记录定义所在的某个模块的时候，是非常有用的。最后，使用命令`rl()`可以列出目前终端中所有可见的记录定义。请在终端中尝试：

```
1> c("/Users/Francesco/records1", [{outdir, "/Users/Francesco/"}]).
{ok,records1}
2> rr(records1).
[person]
3> Person = #person{name="Mike",age=30}.
#person{name = "Mike",age = 30,phone = undefined}
4> Person#person.age + 1.
31
```



```

5> NewPerson = Person#person{phone=5697}.
#person{name = "Mike",age = 30,phone = 5697}
6> rd(name, {first, surname}).
name
7> NewPerson = Person#person{name=#name{first="Mike",surname="Williams"}}.
#person{name = #name{first = "Mike",surname = "Williams"},
      age = 30,phone = undefined}
8> FirstName = (NewPerson#person.name)#name.first.
"Mike"
9> rl().
-record(name,{first,surname}).
-record(person,{name,age = 0,phone}).
ok
10> Person = Person#person{name=#name{first="Chris",surname="Williams"}}.
** exception error: no match of right hand side value
      #person{name = #name{first = "Mike",surname = "Williams"},
      age = 30,phone = undefined}

```

在前面的例子中，我们从records1模块中加载了person记录定义，创建了一个实例，并提取age字段。在命令行6中，我们创建了一个name类型的新记录，它包含first字段和surname字段。我们通过一个操作绑定保存在变量Person中的记录的名字字段到一个建立的记录实例。最后，在命令行8中，我们通过保存在变量NewPerson中的类型为person的记录的名字字段来提取名称，所有这些都在一个操作中完成。

让我们来看看命令行10发生了什么。这是一个初学者和经验丰富的老程序员都会犯的很常见的错误，即忘记了Erlang的变量是单赋值的，而且=运算符是非破坏性的。在命令行10中，你可能认为你在设置name字段为一个新的名字，但你其实是在对一个你刚刚在右边创建的记录和左边已绑定变量Person的内容进行模式匹配。模式匹配会失败，因为左边的name记录包含的字段"Mike" 和 "Williams"与右边的字段"Chris"和"Williams"不匹配。

最后，终端命令rf(RecordName)和rf()删除当前终端中可见的一个或所有的记录定义。

记录实现

我们现在要告诉你一个众所周知的秘密。我们宁可告诉你，但是，当你从终端开始测试记录，或者使用调试工具来对你的代码进行纠错，或者打出内部数据结构的时候，你一定会碰到它。Erlang编译器在程序运行之前使记录生效实现。把记录转换成元组，并且把藉于记录的函数转换为藉于对应元组的函数和内置函数。你可以从终端的交互中看到这些：

```

11> records1:joe().
#person{name = "Joe",age = 21,phone = "999-999"}
12> records1:joe() == {person,"Joe",21,"999-999"}.
true

```

```
13> Tuple = {name,"Francesco","Cesarini"}.
#name{first = "Francesco",surname = "Cesarini"}
14> Tuple#name.first.
"Francesco"
```

从上面的代码中，我们可以推断出`person`是一个4位元组，第一个元素是用基元`person`来“标记”的元组，其余部分是类似的在记录的声明中给出的元组字段。这个`name`记录是一个3位元组，其中第一个元素是基元`name`，第二个是`first`名称字段，第三个是`surname`字段。

注意默认情况下终端是如何假设一个元组是一个记录的。遗憾的是，在你的程序中也同样如此，因此无论你做什么，永远也不要你的程序中使用元组来表现记录。如果你这样做，我会断绝和你的关系并拒绝承认曾帮助过你学习Erlang语言。我是认真的！

警告：为什么你不应该使用记录的元组的表现形式呢？因为使用这种表现形式会破坏数据抽象，你对记录类型的任何修改在使用元组的时候都不会反映到代码中。如果你在记录中添加一个字段，编译器创造的元组的大小将会改变，在尝试模式匹配记录到你的元组时会导致`badmatch`错误（显然你忘记了添加新元素）。如果你正在使用记录，交换记录中的字段顺序不会影响你的代码，因为你通过名字访问字段。然而如果在某些地方，你使用一个元组，却忘了交换所有的具体数值，那么你的程序可能会运行错误或者更糟糕，程序可能会表现出意想不到的行为。最后，即使这不是你最需要担心的，在Erlang未来的发行版本中内部记录的表现形式也可能会发生改变，从而使你的代码向后不兼容。

想查看在代码转换时有关记录的生成代码，编译模块的时候请使用“E”选项。结果会产生一个以E为后缀的文件。让我们以编译`records1`模块为例：使用`compile:file(records1, ['E'])`或终端命令`c(records1, ['E'])`会产生一个名为`records1.E`的文件。包含目标代码的`beam`文件没有被生成。我们不会讨论各种命令的细节，因为它们依赖不同的实现，并且也超出了本书的范围。但我们仍然可以有趣地看到：

```
-file("/Users/Francesco/records1.erl", 1).

birthday({person,_,Age,_} = P) ->
begin
    Rec0 = Age + 1,
    Rec1 = P,
    case Rec1 of
        {person,_,_,_} ->
            setelement(3, Rec1, Rec0);
        _ ->
            erlang:error({badrecord,person})
    end.
end.

joe() ->
{person,"Joe",21,"999-999"}.
```

```

showPerson({person,Name,Age,Phone}) ->
    io:format("name: ~p age: ~p phone: ~p~n", [Name,Age,Phone
])).

module_info() ->
    erlang:get_module_info(records1).

module_info(X) ->
    erlang:get_module_info(records1, X).

```

记录内置函数

内置函数`record_info`给我们提供一个记录类型及其代表的信息。函数调用`record_info(fields, recType)`返回`recType`中的字段名称列表，而函数调用`record_info(size, recType)`返回元组的大小，即字段的数目加上1。一个字段在元组中的位置由`#recType.fieldName`来确定，而`recType`和`fieldName`都是基元：

```

15> #person.name.
2
16> record_info(size, person).
4
17> record_info(fields, person).
[name,age,phone]
18> RecType = person.
person
19> record_info(fields, RecType).
* 1: illegal record info
20> RecType#name.
* 1: syntax error before: '.'

```

请注意命令行19是如何失败的。如果你在一个模块中输入相同的代码，作为一个函数的一部分并且编译，编译也会失败。原因很简单，内置函数`record_info/2`和`#RecordType.Field`计算可以不包含变量，但必须包含文字基元。这是因为它们是在代码运行和变量被绑定之前由编译器来处理的，它们会先各自被转换为所表达的值。

一个你可以在保护元中使用的内置函数是`is_record(Term, RecordTag)`。这个内置函数将证实项元是一个元组，它的第一个元素是`RecordTag`，而且其元组的大小是正确的。这个内置函数返回基元`true`或`false`。

宏

宏（macro）允许你在Erlang中使用结构缩写，然后Erlang的预处理程序（EPP）在编译的时候会扩展它们。宏可以使程序更具有可读性和实现语言本身以外的特性。使用有条件的宏，就有可能写出可以在不同的方式下定制的程序，在调试和生产方式中或者在不同的系统架构中相互切换。

简单宏

最简单的宏可以用来定义一个常量，如：

```
-define(TIMEOUT, 1000).
```

宏是通过在宏的名字前放置一个?来使用的，如：

```
receive
    after ?TIMEOUT -> ok
end
```

当宏在预处理程序中扩展时，前面的代码就是如下的Erlang语句：

```
receive
    after 1000 -> ok
end
```

一个简单宏定义的一般形式为：

```
-define(Name, Replacement).
```

大写Name是一种习惯，但不是必需的。在前面的例子中，Replacement是常量1000，事实上它可以是任何Erlang的标记序列，也就是说，它可以是一系列“单词”，比如变量、基元、符号或标点符号。其结果不一定是完整的Erlang表达式或顶级的形式（即函数定义或编译指令）。通过宏扩展生成新的关键字是不可能的，请看如下例子：

```
-define(FUNC, X).
-define(TION, +X).

double(X) -> ?FUNC?TION.
```

在这里，你可以看到，置换TION不是一个表达式，但是可以通过扩展一个合法的函数定义（或顶级形式）产生。请注意，当追加宏的时候，默认用来间隔的空格将添加到结果中去：

```
double(X) -> X + X.
```

宏参数化

宏可以带有用变量名标识的参数。带参数的宏的一般形式表示为：

```
-define(Name(Var1, Var2, ..., VarN), Replacement).
```

在这里，作为正常的Erlang变量，变量Var1、Var2、...、VarN需要大写第一个字母。下面是一个例子：

```
-define(Multiple(X,Y),X rem Y == 0).

tstFun(Z,W) when ?Multiple(Z,W) -> true;
tstFun(Z,W)                -> false.
```

宏定义在这里使得保护元表达式更具有可读性；需要使用一个宏而不是一个函数，因为保护元的语法排除函数调用。宏在扩展后，调用就变成“内联”：

```
tstFun(Z,W) when Z rem W == 0 -> true;
tstFun(Z,W)                -> false.
```

宏参数化的另一个例子是用于诊断打印输出。在代码中出现两个已经定义的宏，而注释掉其中一个并不罕见：

```
%-define(DBG(Str, Args), ok).
-define(DBG(Str, Args), io:format(Str, Args)).

birthday(#person{age=Age} = P) ->
    ?DBG("in records1:birthday(~p)~n", [P]),
    P#person{age=Age+1}.
```

在开发系统的时候，你在代码中调试打印所有信息。当你想将它关闭的时候，你需要做的只是在重新编译代码之前注释掉DBG的第二个定义，并去掉第一个定义的注释。

调试和宏

Erlang中的宏的主要用途之一是允许代码可以以不同的方式引入。宏方法的优点是可以使用条件宏（我们将在本节中讲述），它可以生成不同版本的代码，如调试版本和生产版本。

第一个方面，宏有能力将宏的参数作为字符串保留，它由参数标记组成。你可以在变量前加前缀??（如??Call）来达到这一目的：

```
-define(VALUE(Call),io:format("~p = ~p~n",[??Call,Call])).
test1() -> ?VALUE(length([1,2,3])).
```

第一次使用Call参数的是??Call，它将作为一个字符串参数扩展到文本，第二次使用将在终端中继续扩展，你会看到如下内容：

```
36> macros1: test1().
"length ( [ 1 , 2 , 3 ] )" = 3
```

第二个方面，有一组常用在调试代码中的预定义的宏：

?MODULE

扩展为它所在模块的名称。

?MODULE_STRING

扩展为由它所在的模块的名称组成的字符串。

?FILE

扩展为它所在的文件名。

?LINE

扩展为它所在的行号。

?MACHINE

扩展使用到的虚拟机，到目前为止，唯一可能的值是BEAM。

最后，定义条件宏是可能的，它根据传递给编译器的不同标记，以不同的方式来扩展。与前面例子中的?DBG相比，条件宏是一种更优雅和有效的方法。下面的语句使这成为可能：

```
-undef(Flag).
```

复位Flag。

```
-ifdef(Flag).
```

如果设置了Flag，接下来的语句就会被执行。

```
-ifndef(Flag).
```

如果没有设置Flag，接下来的语句就会被执行。

```
-else.
```

这提供了另外一种可能性，如果这个条件满足了，接下来的语句就会被执行。

```
-endif.
```

终止条件构造。

下面是一个使用它们的实例：

```
-ifdef(debug).  
    -define(DBG(Str, Args), io:format(Str, Args)).  
-else.  
    -define(DBG(Str, Args), ok).  
-endif.
```

在代码中可以使用如下：

```
?DBG("~p:call(~p) called~n",[?MODULE, Request])
```

如果想要打开系统的调试，你需要设置调试标签。可以在终端中使用以下命令：

```
c(Module,[{d,debug}]).
```

或者，你可以使用`compile:file/2`以编程的形式来做到这一点。你可以使用`c(Module, [{u, debug}])`重置标签。

需要正确地嵌套类似的条件宏的定义，并且不能在函数定义中出现它们。

要调试宏定义，可以让编译器把`ep`处理完文件的结果保存到一个文件中。你可以在终端中使用`c(Module, ['P'])`，而在程序中可以使用函数`compile:file/2`。这些命令保存结果到文件`Module.P`中。`'P'`标签不同于`'E'`标签，因为记录类型必要的代码转换不是由`'P'`进行的。

include文件

习惯上把记录和宏定义放入一个`include`文件中，使它们能够在整个项目的多个模块中共享，而不仅仅是在一个模块中。为了使现有的定义可以在多个模块中使用，可以把它放在一个单独的文件中，然后在模块中使用`-include`指令，其通常放置在`module`和`export`指令之后：

```
-include("File.hrl").
```

在前面的指令中，文件名两边的双引号`"..."`是强制性的。包含的文件带有后缀`.hrl`，但这不是强制的。

编译器拥有一个路径列表来搜索包含文件，其中第一个路径是当前的目录，其次是包含要被编译的源代码的目录。你可以在编译的时候使用`i`选项把其他的路径列表包含进来：`c(Module, [{i, Dir}])`。可以指定多个目录，最后指定的目录会被首先搜索。

练习

练习7-1：扩展记录

扩展`person`记录类型，让它包括一个`person`的地址字段。试确定哪些`person`的现有功能必须修改，哪些可以保留不变。

练习7-2：记录保护元

使用`record`内置函数`record(P, person)`就可以检查变量`P`是否包含`person`记录。解释你将如何使用它来修改函数`foobar`，定义如下：

```
foobar(P) when P#person.name == "Joe" -> ...
```

使得碰到非记录类型也不会出错失败。

练习7-3: db.erl练习回顾

重新回顾我们在第3章练习3-4中关于数据库*db.erl*的例子。用记录代替元组重写它。作为一个记录,你可以使用如下定义:

```
is_record{data, {key, data}}.
```

你应该记得把这个定义放在一个包含文件中。请使用在第5章练习5-1中开发的数据库服务器测试你的结果。

练习7-4: 记录和形状

定义一个记录类型来表示圆,再定义一个来表示矩形。你应当做如下几点假设:

- 圆的半径。
- 矩形的长度和宽度。

定义一些能接收这些类型的函数,计算这些几何图形的周长和面积。一旦完成,将三角形的代码添加到你的类型定义和函数中,其中你可以假设三角形是由其三条边的长度来定义的。

练习7-5: 二叉树记录

定义一个记录类型来代表二叉树状结构,内部节点和树叶上带有数值。实例如图7-1所示。定义记录类型的函数可以实现如下功能:

- 计算树的总值。
- 在树中寻找最大值(如果有的话)。

如果树已经排好序,那么对所有节点在下面的左边子树的节点的值要小于或等于该节点的值,而这个节点的值要小于或者等于其下面的右边子树的节点的值。如图7-2所示:

- 定义一个函数来检查一个二叉树是否已经排序。
- 定义一个函数在一个有序树中插入一个值,而且维持原有排序。

练习7-6: 参数化宏

定义一个参数化宏SHOW_EVAL,当显示模式被关掉时它将简单地返回表达式的结果,但当显示模式被打开时会打印表达式和它的数值。无论在什么情况下你都应该确保计算表达式只有一次被求值。

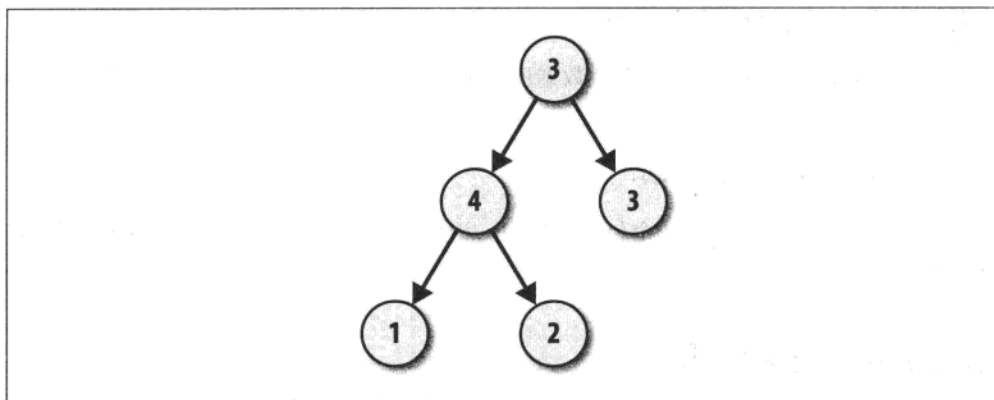


图7-1：一个二叉树例子

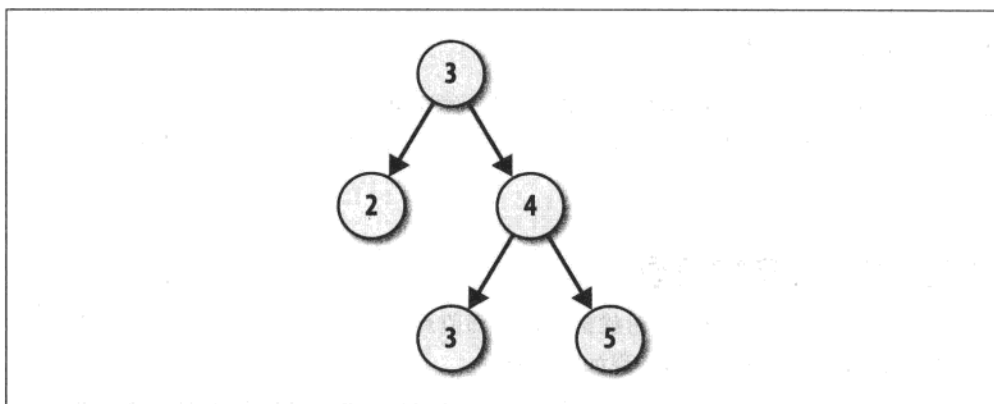


图7-2：一个有序二叉树

练习7-7：呼叫计数

在一个特定的模块功能中你如何使用Erlang的宏工具来计算打电话的次数？

练习7-8：枚举类型

枚举类型是由有限个元素组成的，例如一星期中的天数，或者一年中的月份。你如何使用宏来帮助实现Erlang中的枚举类型？

练习7-9：调试db.erl练习

扩展练习7-3中的数据库例子，使它们执行的时候可以包括可选的调试代码，用来报告它们执行的时候对数据库的请求行为。

软件升级

你收到了一个错误报告，说在你的一个即时消息（IM）服务器中，欧元符号传递到最终目的地时变成了乱码。你在映射特殊字符的库模块中发现了错误，并纠正了这个错误，然后重新编译代码并进行测试。当它们通过验证后，你传送这些补丁到运行着的服务器并在Erlang的运行时系统中加载它。下一次当接收到一个欧元符号的时候，就会使用这个补丁模块，从而可以正确地映射欧元符号。

你不需要通过复杂或者奇怪的解决方案来实现这一切，不必重新启动系统，最重要的一点是，这不会影响任何其他目前正在处理的与即时消息相关的事件。所有这一切似乎听起来很简单，但事实的确如此。这不仅简单，而且非常酷！最初的灵感来自Smalltalk语言，这门语言的软件升级能力是现代编程语言中非常罕见的。

这种能够在运行时加载新的模块和升级模块，使得不仅可以在修改错误的时候，也可以在添加新功能的时候，让系统不间断地运行。它缩短了错误修改周期并且便于测试，因为在大多数的情况下，系统不必再为补丁程序的验证和部署而重新启动。这个软件升级机制依赖于一套简单且强大的结构，而在此基础之上建立了更多强大的工具。这些升级工具广泛地用于各个基于Erlang的系统，在这些系统中的停机时间必须减小到最小。

升级模块

你在尝试本书的例子和做练习的时候，你可能遇到加载新模块到运行时系统中的情况，而你自己可能没有意识到到底发生了什么事情。让我们思考一下在Erlang中递增的软件开发方法。在本章的其余部分我们会讨论update工作的细节和其他Erlang代码处理，在此之前我们先详细讲解一个简单的例子来研究软件更新在实际中是如何进行的。

我们从编写一个类似在第3章练习3-4中描述的数据库模块开始。我们使用在dict模块中定义的键值字典库来创建各种变量，并引入一项新的类库。请查看它的用户手册。我们

要创建一个模块，它导出两个函数：`create/0`，返回一个空数据库，以及`write/3`，插入一个`Key`和`Element`到数据库中。

此代码的描述可以在`db`模块中找到。还记得我们在第2章中讨论的`-vsn(1.0)`属性吗？虽然它不是强制性的，但这有助于我们在运行的任何时候追踪已加载到运行时系统中的模块的版本：

```
-module(db).
-export([new/0,write/3,read/2, delete/2,destroy/1]).
-vsn(1.0).

new()                -> dict:new().

write(Key, Data, Db) -> dict:store(Key, Data, Db).

read(Key, Db) ->
  case dict:fetch(Key, Db) of
    error      -> {error, instance};
    {ok, Data} -> {ok, Data}
  end.

delete(Key, Db) -> dict:erase(Key, Db).

destroy(_Db)  -> ok.
```

让我们现在编译和测试我们所写的代码，它在数据库中加入两个元素，然后寻找还没有插入的一个。在开始的时候`dict`模块返回的数据结构可能看起来比较奇怪。它在这里是可见的，因为我们正在终端中测试模块，并且把数值和传递到字典函数的一系列的变量绑定。在正常情况下，`Db`变量将传递到`receive-eval`循环，而且它是不可见的：

```
1> c(db).
{ok,db}
2> Db = db:new().
{dict,0,16,16,8,80,48,
  {[[],[],[],[],[],[],[],[],[],[],[],[],[],[],[]],
  {[[],[],[],[],[],[],[],[],[],[],[],[],[],[],[]]}}
3> Db1 = db:write(francesco, san_francisco, Db).
{dict,1,16,16,8,80,48,
  {[[],[],[],[],[],[],[],[],[],[],[],[],[],[],[]],
  {[[],[],[],[],[],[],[],[],[],[],[],[],[],[],[]],
  [[francesco|san_francisco]]}}
4> Db2 = db:write(alison, london, Db1).
{dict,2,16,16,8,80,48,
  {[[],[],[],[],[],[],[],[],[],[],[],[],[],[],[]],
  {[[],[],[],[],[],[],[],[],[],[],[],[],[],[],[]],
  [[alison|london]],
  [[francesco|san_francisco]]}}
5> db:read(francesco, Db2).
** exception error: no case clause matching san_francisco
   in function db:read/2
```

```
6> dict:fetch(francesco, Db2).
san_francisco
```

且慢！这里出现错误了。当调用`read/2`时，它没有返回`{ok, san_francisco}`，而是我们得到了一个case语句错误。检查一下我们的实现和字典模块的手册，很快就看到，我们使用了`dict:fetch/2`，而不是`dict:find/2`。这是一个`dict:fetch/2`的直接调用，该条目在字典中将返回Data（not `{ok, Data}`），否则就会抛出一个异常错误。另一方面是`dict:find/2`函数，如发现该条目则返回元组`{ok, Data}`，否则返回基元`error`。

让我们来改正这个错误，用如下代码替换`read`函数，与此同时，提高这个模块的版本到1.1：

```
...
-vsn(1.1).
...
read(Key, Db) ->
  case dict:find(Key, Db) of
    error      -> {error, instance};
    {ok, Data} -> {ok, Data}
  end.
...
```

开始用终端处理任何事情之前，我们先使用`module_info/0`函数来获取1.0版本的代码。我们把修改保存到`db`模块，在运行前一个版本的同一个终端中编译它，然后用我们之前在数据库中插入的相同条目来继续测试这个模块。现在`read/2`运行正常了。当我们调用`module_info/1`的时候，我们在属性列表中得到了新的模块版本：

```
7> db:module_info().
[{exports, [{new, 0},
            {write, 3},
            {read, 2},
            {destroy, 1},
            {delete, 2},
            {module_info, 0},
            {module_info, 1}}],
 {imports, []},
 {attributes, [{vsn, [1.0]}]},
 {compile, [{options, [{outdir, "/Users/Francesco/"},
                      {version, "4.5.2"},
                      {time, {2008, 8, 11, 3, 9, 42}},
                      {source, "/Users/Francesco/db.erl"}]}]}]

8> c(db).
{ok, db}
9> db:read(francesco, Db2).
{ok, san_francisco}
10> db:read(martin, Db2).
{error, instance}
11> db:module_info(attributes).
[{vsn, [1.1]}]
```

在例子中我们修正了一个错误，其实也可以增加新的功能，或者两者同时进行。在编写和测试我们的代码的时候，虽然你可能没有意识到，实际上已经在使用软件的升级功能了。当我们这样做的时候，进程中的数据存储（例子中的终端中的Db变量数据）不会受到升级的影响，它们在加载新模块之后仍然可以使用。

幕后

那么软件升级功能在幕后是如何工作的呢？在任何时候，一个模块的两个版本都可以加载到运行时系统。我们称它们为旧的和当前的版本。在我们开始解释软件升级的具体细节之前，先快速地浏览一下一个模块中的函数是如何指引到另外一个模块的函数的。

- 你可能不记得以下格式的函数调用：

```
Module:Function(Arg1, .., ArgN)
```

该Module名称添加到一个导出Function名称之前，这通常称为完全限定的函数调用。这是方法之一，让一个模块中的一个函数（称为A）指引到另一个模块中定义的函数（我们称之为B）。

- 其他为了使模块A可以指引模块B的函数的相关机制，可以使A导入一些B的函数：

```
-import(B, [f/1]).
```

这样在A里面就可以直接调用f，就像B:f一样。

在一个模块中可能可以直接或者通过一个完全限定的名称来调用同一模块中的另一个函数。我们很快会再回到这两者的区别上来。

现在，我们解释一下软件的升级过程，首先是为了模块间调用，然后是为了模块内调用。每个指向模块A中函数的正在运行的进程，将链接到模块A的一个版本。当模块A的一个新版本加载的时候，它就成为当前版本，以往版本变成了旧版本。

如果模块A中定义的一个进程P直接或通过一个完全限定的函数调用（见图8-1a），从模块B中调用一个函数，从而加载模块B的一个新版本（第2版），这个进程将仍然与B的相同版本链接，其现已成为B的旧版本（见图8-1b）。在B中的下一个函数调用，不管是直接或完全限定的形式，该链接将切换到新的版本2（见图8-1c）。这将适用于B的所有函数，而不仅仅是触发切换的那些函数。

对正在运行的进程，其定义模块升级的情况将更加复杂，特别是，它取决于模块中哪个函数直接或通过完全限定函数调用被调用了：

- 如果函数调用不是完全限定的，该进程将继续运行模块的旧版本。

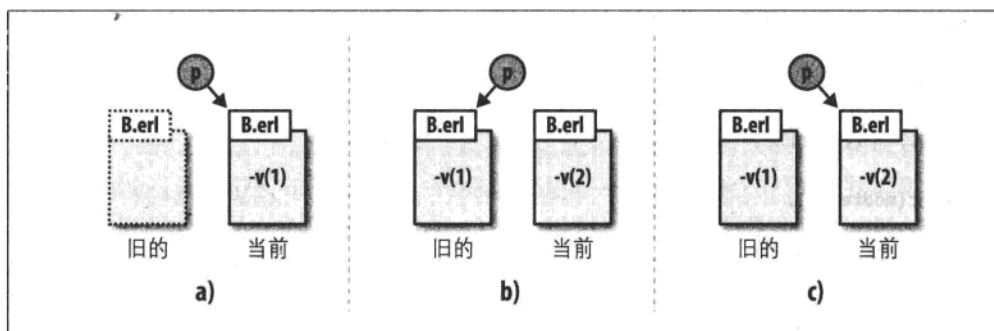


图8-1：升级模块B

- 然而当加载了一个新版本的时候，对旧模块的全局调用将是不可能的了，然而其本地调用可以通过递归循环中的进程来完成。

为了更详细地说明这一点，让我们来看一个例子，了解模块和软件升级对它的影响：

```

1  -module(modtest2).
2
3  -export([main/0,loop/0,a/1,do/1]).
4
5  main() ->
6    register(foo,spawn(modtest2,loop,[])).
7
8  loop() ->
9    receive
10   {Sender, N} ->
11     Sender ! a(N)
12   end,
13   loop().
14
15  do(M) ->
16    foo ! {self(),M},
17    receive Y ->
18      Y
19    end.
20
21  a(N) -> N+2.
```

主程序创建一个命名进程foo，它运行函数loop()。loop函数的结果是提供几个值给函数a/1：值通过do/1函数发送到loop进程。下面是该程序的代码：

```

9> c(modtest2).
{ok,modtest2}
2> modtest2:main().
true
3> modtest2:do(99).
99
```

假设你现在升级a/1函数的第21行的定义如下：

```
a(N) -> N.
```

重新编译，效果如下：

```
4> c(modtest2).
{ok,modtest2}
5> modtest2:do(99).
99
```

很明显没有发生任何变化。另外，如果你把调用a(N)修改成一个完全限定的函数调用：

```
loop() ->
receive
{Sender, N} ->
    Sender ! modtest2:a(N)
end,
loop().
```

重新编译后对同一软件升级的影响将显而易见：

```
6> c(modtest2).
{ok,modtest2}
7> modtest2:do(99).
99
```

最后一个例子，当递归调用是完全限定的函数调用的时候它有可能升级一个正在运行的循环：

```
loop() ->
receive
{Sender, N} ->
    Sender ! a(N)
end,
modtest2:loop().
```

如果你插入打印语句：

```
loop() ->
receive
{Sender, N} ->
    Sender ! a(N)
end,
io:put_chars("boo!~n"),
modtest2:loop().
```

你可以在下面的交互中看出变化的效果：

```
1> c(modtest2).
{ok,modtest2}
```

```

2> modtest2:main().
true
3> modtest2:do(99).
99
4> c(modtest2).
{ok,modtest2}
5> modtest2:do(99).
99
6> modtest2:do(99).
bool
99

```

在前面的数据库例子中，我们总是运行db模块中的最新版本，因为所有终端到类库的调用都是完全限定的函数调用。

由于在任何时候运行时系统的一个模块中只能存在两个版本，当加载第三个版本的时候，最古老的版本就会被清除（删除）掉，而当前版本成为旧版本，如图8-2所示。任何被清除模块的最老版本的运行进程将被终止。任何现已成为旧版本的运行进程将继续运行，直到它执行一个完全限定的函数调用。

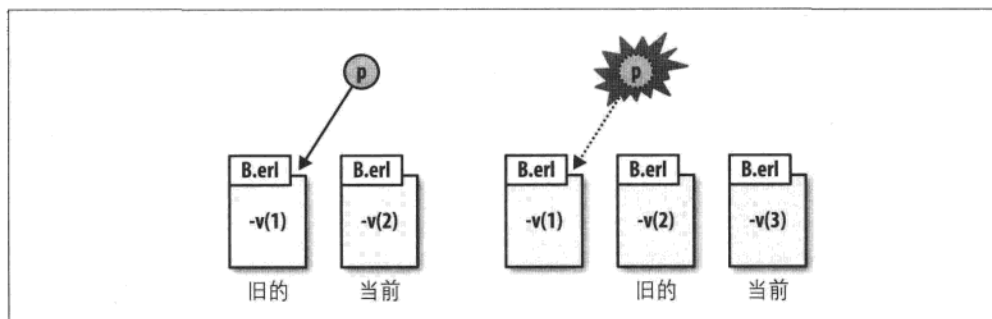


图8-2：连接到一个旧版本

载入代码

在Erlang的运行时系统中可以通过几种不同的方式实现代码加载。第一种是通过调用一个尚未加载模块中的函数。代码服务器是作为Erlang核心的一部分的一个进程，它将为那个模块搜索编译文件（.beam），如果找到了，就会加载它。注意，如果该beam文件没有找到，则不会自动编译。调用这个模块的进程就可以调用这个函数了。

另一种加载模块的方法是编译它。你可以使用c(Module)终端命令，compile:file(Module)，或它的派生函数之一，它们都可以在compile模块的文档中找到。在我们的例子中，在每次编译它之后，我们就可以在终端里加载db模块的最新版本。

最后，你可以通过调用`code:load_file(Module)`显式地加载一个模块。这个调用是有用的，因为它可以出现在程序里或者在终端中使用。不过在终端中，可以使用等价的终端命令`l(Module)`。

所有的这些调用会清除模块的最老版本（如果有的话），当前版本成为旧版本，而最新加载的成为当前的版本。请注意词语旧的和当前指的不是编译时间或`vsn`属性里的更高版本号，而是你在运行时系统里加载模块的顺序。

有很多方法可以查看是否已经加载一个模块：

- 尝试在Erlang终端中输入该模块名称的一部分且按Tab键；如果名称自动补齐，那么模块已经被加载了。
- 按Tab键将列出当前所有加载的模块。
- 另一种查看模块是否已经加载的方法是调用函数`is_loaded(Module)`，如果模块已经加载，它会返回beam文件的位置；否则，它返回基元`false`。

代码服务器

我们已经在前面的内容中简要地提到了代码服务器。在此，我们将更详细地分析它。代码服务器的主要任务是管理动态加载和清除模块。`code.erl`模块中的一组库函数为程序员提供了灵活管理和配置服务器的系统代码库。

加载模块

调用一个未加载模块，或者显式地通过`load_file/1`函数调用，结果都会触发Erlang中代码的动态加载。代码服务器将在搜索路径中搜索模块编译版本的代码，当代码服务器发现它们时，就会在虚拟主机上加载它们。

该代码搜索路径由一个目录列表组成。当查找你要加载模块的编译版本时，它会顺序地搜索这些目录。通过使用`get_path()`函数可以查看运行时系统的默认搜索路径。默认目录包括当前的工作目录和所有作为Erlang/OTP发布一部分的默认类库应用程序的路径。你可以在`$ERLANGROOT/lib`目录下找到所有这些库应用程序。若要找出你安装Erlang的根目录，可以使用`root_dir()`函数。在你的代码搜索路径中，相同的模块可能会有多个版本。当你加载一个新的版本时将会按顺序扫描目录，结果是该模块第一次出现的时候会被选中。因此，创建一个补丁目录，使其首先出现在代码搜索路径中的做法并非罕见。在这个目录中的任何补丁都将会首先选择，它们会覆盖其修补的原始版本的模块。

粘目录

你是否曾经尝试过创建名称为lists.erl的模块？如果是这样，当你尝试编译它的时候，几乎可以肯定会遇到错误提示：运行时系统无法加载属于粘目录中的模块。

如果这种情况是伴随模块出现而不是list，你一定选择了内核、标准库（stdlib）或者系统结构体系支持库（sasl）中的应用程序使用过的名称作为一个模块的名称。这三个目录已标记为默认的粘目录，因此任何打算覆盖它们所定义的模块的做法都会失败。

你可以通过函数调用code:stick_dir(Dir)创建自己的粘目录。如果你想覆盖粘目录中的一个模块，可以使用code:unstick_dir(Dir)函数，或者在启动Erlang的时候使用-nostick标签。

操控代码搜索路径

你可以添加目录到代码搜索路径中，通过使用code:add_patha(Dir)在列表的开头增加目录，而使用code:add_pathz(Dir)在末端追加目录。在下面的例子中，注意当前工作目录如何成为第二个元素。该代码模块还提供了很多功能来删除、替换和重写目录：

```
3> code:add_patha("/tmp").
true
4> code:get_path().
["/tmp",".", "/usr/local/lib/erlang/lib/kernel-2.12.3/ebin",
"/usr/local/lib/erlang/lib/stdlib-1.15.3/ebin",
"/usr/local/lib/erlang/lib/xmerl-1.1.9/ebin",
"/usr/local/lib/erlang/lib/wx-0.97.0718/ebin",
"/usr/local/lib/erlang/lib/webtool-0.8.3.2/ebin",
"/usr/local/lib/erlang/lib/typer-0.1.3/ebin",
.....]
```

在这个例子中，我们只显示了其中一些库模块目录。在你的机器上尝试命令并进入任何列出的目录中去。在检查内容时，你应该找到所有与特定应用有关的beam文件。

前面文字中关于代码服务器如何查找模块的解释，阐述了为什么你必须把Erlang终端的当前工作目录改成含有你的beam文件的目录。在code:get_path/0的例子中，除非你修改了代码搜索路径的结构，否则将可以看到当前工作目录（.）是当尝试加载一个模块时代码服务器搜索的第一个目录。最后，在启动Erlang的终端的时候，你可以通过erl -pa Path或者erl -pz Path来指示erl命令在路径的开始（a）和最后（z）加上这些路径，你也可以添加目录。

终端模式：互动式和嵌入式

在默认情况下，我们称Erlang的终端为互动模式。这意味着在启动的时候只有模块运行时系统需要加载。其他代码是在当一个完全限定的函数调用发生时动态加载。

在嵌入式模式下，所有在一个二进制启动文件中列出的模块都会在启动时加载。启动后，对尚未加载模块的调用会导致一个运行时错误。嵌入式模式强制实施严格的版本控制，因为它要求所有模块在启动时可用。最后，通过停止、搜索和加载一个模块不会影响系统的软实时方面，因为它可能是互动模式的情况。你可以在启动Erlang时通过`erl -mode Mode`指令选择你的模式，其中Mode是`embedded`或者`interactive`。

清除模块

代码服务器可以通过调用`code:purge(Module)`去掉或清除模块的旧版本，但如果任何程序正在运行这些代码，它们将首先被终止，之后删除旧版本代码。调用的结果是，如果有任何程序被终止了该函数返回`true`，否则返回`false`。

如果你不希望终止任何正在运行的旧版本代码的程序，请使用`code:soft_purge(Module)`。这个调用只有在没有进程运行这些代码时，才会删除该模块的旧版本。如果有任何进程仍在运行这个代码，它会返回`false`，其他什么也不做。如果成功删除了旧的模块，它会返回`true`。

我们将在第12章中介绍的OTP框架提供了一种监督机制，旨在有组织地处理进程的终止。这种监督类型很显然的一个应用是在软件更新后终止一些进程。

升级过程

既然我们已经很详细地了解了软件升级，现在来讨论一个实例，其中循环数据的格式需要在运行循环中改变。

我们实现了一个`db_server`模块，它提供了一个以1.1版本的`db`模块的字典格式存储数据的进程。除了导出的客户端函数，请特别注意`upgrade/1`函数。关于这方面你马上会了解更多：

```
-module(db_server).  
-export([start/0, stop/0, upgrade/1]).  
-export([write/2, read/1, delete/1]).  
-export([init/0, loop/1]).  
-vsn(1.0).
```

```

start() ->
    register(db_server, spawn(db_server, init, [])).

stop()->
    db_server ! stop.

upgrade(Data) ->
    db_server ! {upgrade, Data}.

write(Key, Data) ->
    db_server ! {write, Key, Data}.

read(Key) ->
    db_server ! {read, self(), Key},
    receive Reply -> Reply end.

delete(Key) ->
    db_server ! {delete, Key}.

init() ->
    loop(db:new()).

loop(Db) ->
    receive
        {write, Key, Data} ->
            loop(db:write(Key, Data, Db));
        {read, Pid, Key} ->
            Pid ! db:read(Key, Db),
            loop(Db);
        {delete, Key} ->
            loop(db:delete(Key, Db));
        {upgrade, Data} ->
            NewDb = db:convert(Data, Db),
            db_server:loop(NewDb);
    stop ->
        db:destroy(Db)
    end.

```

upgrade函数需要一个变量作为参数，并将其转发到db_server进程。这个变量传递给db:convert/2函数，它返回一个可能的更新格式的数据库。在db模块的1.1版本中没有包括convert/2函数，因为所有我们所做的就是解决一个错误，并没有要求我们更改数据的内部格式。仔细阅读一遍db_server的代码，并确保你已经完全明白了。如果有任何不清楚的地方，请复制它并在终端中测试，并且请阅读dict库的手册。

现在让我们来创建一个新的db模块，这一次使用gb_trees库模块建立在通用平衡树基础上。当实现它的时候，我们包括了convert/2函数。指定由dict模块返回的数据结构，这个函数从字典中提取所有的元素，然后把它们插入二叉树中，返回一个可以由gb_trees模块使用的数据结构：

```

-module(db).
-export([new/0, destroy/1, write/3, delete/2, read/2, convert/2]).
-vsn(1.2).

```

```

new() -> gb_trees:empty().

write(Key, Data, Db) -> gb_trees:insert(Key, Data, Db).

read(Key, Db) ->
    case gb_trees:lookup(Key, Db) of
        none -> {error, instance};
        {value, Data} -> {ok, Data}
    end.

destroy(_Db) -> ok.

delete(Key, Db) -> gb_trees:delete(Key, Db).

convert(dict,Dict) ->
    dict(dict:fetch_keys(Dict), Dict, new());
convert(_, Data) ->
    Data.

dict([Key|Tail], Dict, GbTree) ->
    Data = dict:fetch(Key, Dict),
    NewGbTree = gb_trees:insert(Key, Data, GbTree),
    dict(Tail, Dict, NewGbTree);
dict([], _, GbTree) -> GbTree.

```

现在，我们可以把db模块从1.1版本升级到1.2版本，这改变了db_server循环数据的内部格式。为此，我们需要在代码搜索路径的顶部加一个补丁目录。通过使用-pa patches标签我们启动Erlang运行时系统（或者使用code:add_patha/1动态地添加目录）。下一步，在补丁目录中放置db模块的1.2编译版本。最后，我们加载新的db模块，（软件方式地）清除旧的版本，并调用升级的客户端函数。例8-1充分显示了这些交互过程。

例8-1：软件升级活动

```

1> cd("/Users/Francesco/database/").
/Users/Francesco/database
ok
2> make:all([load]).
Recompile: db
Recompile: db_server
up_to_date
3> db:module_info().
[{exports, [{new, 0},
             {write, 3},
             {read, 2},
             {destroy, 1},
             {delete, 2},
             {module_info, 0},
             {module_info, 1}}],
 {imports, []},
 {attributes, [{vsn, [1.1]}]},
 {compile, [{options, []},
             {version, "4.5.2"},
             {time, {2008, 8, 11, 16, 34, 48}},
             {source, "/Users/Francesco/database/db.erl"}]}]}

```

```

4> db_server:start().
true
5> db_server:write(francesco, san_francisco).
{write,francesco,san_francisco}
6> db_server:write(alison, london).
{write,alison,london}
7> db_server:read(alison).
{ok,london}
8> db_server:read(martin).
{error,instance}
9> code:add_patha("/Users/Francesco/patches").
true
10> code:load_file(db).
{module,db}
11> code:soft_purge(db).
true
12> db_server:upgrade(dict).
{upgrade,dict}
13> db:module_info().
[{exports,[{new,0},
            {write,3},
            {read,2},
            {destroy,1},
            {delete,2},
            {convert,2},
            {module_info,0},
            {module_info,1}}],
 {imports,[]},
 {attributes,[{vs_n,[1,2]}]},
 {compile,[{options,[{outdir,"/Users/Francesco/patches/"}}],
            {version,"4.5.2"},
            {time,{2008,8,11,16,30,33}},
            {source,"/Users/Francesco/patches/db.erl"}]}]}
14> db_server:write(martin, cairo).
{write,martin,cairo}
15> db_server:read(francesco).
{ok,san_francisco}
16> db_server:read(martin).
{ok,cairo}

```

服务器依然以相同的关键字元素数据在运行，但是以不同的格式保存而且使用的是db模块的升级版本。正如你所看到的，它是如此简单且强大。

在软件升级中其他需要记住的重要问题包括非向后兼容的模块，升级失败后降级的功能，在分布式环境中升级的同步。虽然运行时软件升级的基本原理很简单，但如果你的系统很复杂而且很重要，那么你升级的步骤和过程可能就不那么简单了。请确保你彻底测试过升级步骤，并考虑到所有可能的情况。作为OTP中间件一部分的SASL应用程序拥有建立在我们讨论过的原理基础上的处理软件升级的复杂工具。

实际的软件升级

在生产环境中，你显然不会从终端中升级一个模块，特别是使用例8-1中的命令行9、10和11。你必须确保它们会自动执行，当一个客户加载db模块的1.2版本后，但在调用upgrade/1函数前调用客户端read或write，这将导致字典的数据结构传递给gb_trees模块。

你将不得不暂停客户端的功能，或者（最好）在你处理升级命令的循环中放置代码的加载，并最终原子性地执行。然而如果你不需要升级内部数据结构，而只是修复错误或增加不影响现有结构的功能，那么你需要做的只是加载新的模块。

.erlang文件

现在是时候介绍.erlang文件了。这个文件放置在用户的主目录或Erlang的根目录中。它应该包含有效的Erlang表达式，所有这些都在启动时读取和执行。它对于设置你正在使用的开发环境的模块和工具的路径非常有用。例如：

```
code:add_patha("/home/cesarini/patches").
code:add_patha("/home/cesarini/erlang/buildtools-1.0/ebin").
```

.erlang文件也可以为配置环境或启动所用。在第11章我们将提供更多的例子。

练习

练习8-1：运行时的软件升级

使用本章基于通用平衡树的db.erl模块，添加一个额外的函数调用code_upgrade/1。此函数接受采用第3章练习3-4中数据库列表版本格式的数据库。它将创建一个EST表，然后保存所有传递给它的元素。它的返回值应为被创建的表。

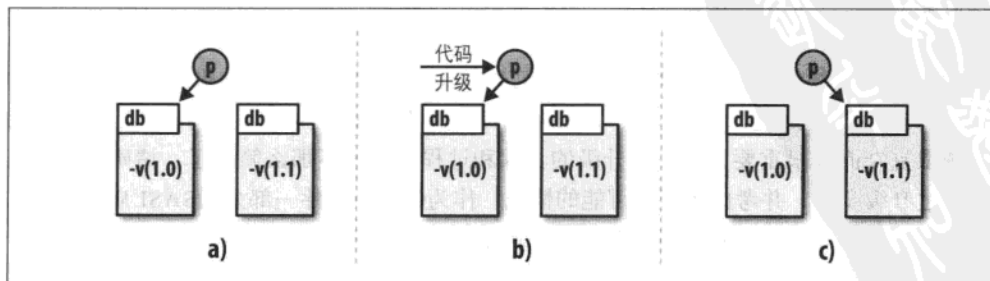


图8-3：升级数据库服务器

接口：

```
db:code_upgrade([RecordList]) -> gb_tree().
```

1. 在终端中测试函数，并将beam文件和源代码文件放置在一个叫做*patches*的子目录中。在本练习的第二部分，继续工作在你当前的工作目录，在那里你保存了采用列表的db.erl模块。注意这两个模块的不同编译版本，防止把它们混淆。
2. 添加客户端函数code_upgrade()到my_db.erl服务器模块中。这个函数应该将消息发送到服务器，服务器将加载新的db模块，用旧的数据库调用函数db:code_upgrade/1，然后使用新的ETS数据库表达式继续循环，如图8-3所示。在代码升级后，所有操作之前的数据都应该仍然可用，客户端应该好像什么都没有发生过一样能够插入、删除和查询元素。
3. 启动时使用db模块的列表版本来测试你的程序。在数据库中插入一些新的元素并切换到db模块的树版本。要切换到新版本，你必须首先把它加载到系统中。（在比较大的系统中，有处理升级的工具。由于这个数据库服务器不是太大，你可以手动加载。）一旦加载db.erl模块的ETS版本，就调用my_db:code_upgrade()。然后你就可以读取元素、写入新的元素和删除旧的元素。请确保服务器稳定运行。

下面是一个关于交互的例子：

```
1> my_db:start().
ok
2> my_db:write(bob, handyman).
ok
3> my_db:write(chris, projectleader).
ok
4> my_db:read(bob).
{ok, handyman}
5> code:add_patha("/home/cesarini/erlang/patches/").
true
6> my_db:code_upgrade().
ok
7> my_db:read(bob).
{ok, handyman}
8> my_db:delete(bob).
ok
9> my_db:write(bob, manager).
ok
10> code:soft_purge(db).
true
```



更多数据类型和高级别构造

到本章为止，我们已经涵盖了Erlang所有的基础部分：顺序编程、并发性、容错和错误恢复。Erlang语言，特别是它的众多类库，为程序员提供了尽可能多有效的帮助。本章概括了各种语言的性能，其中许多来自函数式编程语言，这些是很好的工具并将会大大提高Erlang程序员的工作效率。

实践中的函数式编程

Erlang是一门函数式的编程语言——相对于其他语言——这是什么意思呢？Erlang突出的特点是函数定义，Pascal、C等其他许多语言也是如此。一个真正的函数式编程语言是其函数可以像任何其他类型的数据一样被处理。Erlang中函数式的数据类型称为*fun*s。它们可以作为参数传递给其他函数，而且可以存储在像元组和记录类型的数据结构中或作为消息发送给其他进程。最重要的是，它们可以作为其他函数的结果，以至于函数可以作为数据传递，可以在程序中动态创建，而不仅仅指静态定义的函数。这可以让你写出简明、抽象、可再次使用的函数，而且它可以通过特定的行为进行参数化，即作为函数的参数“包装起来”。结果就是，你的代码不但变得更紧凑，而且也更容易编写、理解和维护。

列表解析是另一个强大的结构，其根源在于函数式编程语言。列表解析允许生成列表，并把它们合并起来，然后基于一系列谓词进行筛选结果。其结果是一个由生成器产生的谓词评价为true的元素列表。就像*fun*s，列表解析可以产生紧凑强大的代码，从而提高程序员的效率。

二进制类型是另一种Erlang的数据类型，虽然与函数式编程没有直接关联，但是在Erlang中它也有不小的影响。二进制类型只不过是一个1和0的序列，是一块存储在内存中的没有类型化的内容。所有套接字和端口通信都是基于二进制的，就像所有文件相关的I/O。在Erlang中使用二进制的威力在于位级别的模式匹配，这样使用很少的努力和代码就能提取相关的位和字节。这使得Erlang非常适合处理协议栈和与网际协议相关的传

输，主要是编码和解码消息帧，只需要很少的代码就可以把它们作为协议栈处理的结果发送或接收。

最后，引用数据类型，其中的元素通常称为*refs*，可以在分布式环境中给你提供跨进程使用的独特标记。特别是，我们使用引用数据值进行比较，而其中许多是跟消息传递有关的。

Funs和高阶函数

为了理解关于funs的全部内容，先从一个例子开始讲解。在Erlang终端中输入下面的赋值语句，绑定变量Bump到fun：

```
Bump = fun(Int) -> Int + 1 end.
```

这个fun需要一个变量作为参数，绑定它到变量Int并把它数值增加1。你可以通过在其后的括号中带上它的参数来调用fun，就好像调用函数一样。如果给它分配了一个变量，你就可以使用它的名称：

```
1> Bump = fun(Int) -> Int + 1 end.  
#Fun<erl_eval.6.13229925>  
2> Bump(10).  
11
```

或者，你可以直接调用：

```
3> (fun(Int) -> Int + 1 end)(9).  
10
```

一个fun就是一个函数，但不是使用模块、函数名字和元数来唯一识别它，而是使用它所绑定的变量或它的定义来确定。在接下来的内容中，我们将解释为什么fun是如此相关和有效。

函数作为参数

列表中最常见的操作之一是访问每个元素，并以某种方式将其转换。例如，下面的代码把一个数字列表中所有的元素都乘以2，并颠倒每一个列表的元素：

| | |
|---|--|
| <pre>doubleAll([]) -> [];</pre> | <pre>revAll([]) -> [];</pre> |
| <pre>doubleAll([X Xs]) -> [X*2 doubleAll(Xs)].</pre> | <pre>revAll([X Xs]) -> [reverse(X) revAll(Xs)].</pre> |

你能看到两个函数之间的通用模式吗？这两个例子的不同之处是影响了元素X转换方式，在例子中用斜体表示；一个函数可以捕获这种转换，给出一个map函数，其第一个参数F是应用到列表中每个元素的函数：

```
map(F,[]) ->
[];
map(F,[X|Xs]) ->
[F(X) | map(F,Xs)].
```

另一种常见的列表操作是过滤拥有特别属性的元素，例如，偶数或者回文的列表（当列表颠倒后也相同）：

```
3> hof1:evens([1,2,3,4]).
[2,4]
4> hof1:palins([[2,2],[1,2,3],[1,2,1]]).
[[2,2],[1,2,1]]
```

以下的代码展示了这两个函数的例子：

```
evens([]) ->
[];
evens([X|Xs]) ->
case X rem 2 == 0 of
true ->
[X| evens(Xs)];
- ->
evens(Xs)
end.

palins([]) ->
[];
palins([X|Xs]) ->
case palin(X) of
true ->
[X| palins(Xs)];
- ->
palins(Xs)
end.
```

在前面的代码中，`palin/1`在模块`hof1`中是如此定义的：

```
palin(X) -> X == reverse(X).
```

`filter`函数把这种“过滤”行为作为一个定义包装起来，其中函数`P`包含了要测试的属性：如果`X`有此属性，`P(X)`会返回`true`，否则返回`false`：

```
filter(P,[]) ->
[];
filter(P,[X|Xs]) ->
case P(X) of
true ->
[X| filter(P,Xs)];
- ->
filter(P,Xs)
end.
```

像`P(X)`返回`true`或`false`的函数称为谓词。到目前为止，一切都很顺利，但实际上你如何使用`map`函数和`filter`函数得到想要的行为呢？为了实现它们，你需要写出这些作为参数传递的fun函数。

编写函数：函数表达式

当你编写函数定义的时候，很可能习惯于这样做：你必须给出函数的参数，也许可以使

用模式匹配来区分不同的情况，并在最后一个被执行的表达式中返回结果。除了不会给出函数名字，一个fun表达式可以做同样的事情。让我们先看一些例子。

参数双倍化的函数如下：

```
fun(X) -> X*2 end
```

两个数字相加的函数：

```
fun(X,Y) -> X+Y end
```

得到一个列表的表头（如果为空则为null）的函数如下：

```
fun([]) -> null;  
([X|_]) -> X  
end
```

正如你所看到的，这些非常类似于函数定义，除了表达式需要单独使用fun...end围起来之外。有时你会看到end后面紧跟点号。这是因为点号表示定义的结束或终端输入行的结束。它不是函数自身表达式的一部分。很多情况下使用的语法和case类似：关键字fun不会在每个不同的模式匹配情况之前出现。但要记住，参数使用括号括起来，就如(...)，即使是只有一个变量作为参数的情况也是如此。

通过使用函数表达式，你现在就可以使用map和filter来定义doubleAll和palins了：

```
doubleAll(Xs) ->  
  map( fun(X) -> X*2 end , Xs).  
  
palins(Xs) ->  
  filter( fun(X) -> X == reverse(X) end , Xs).
```

在每一种情况下，你可以看到函数表达式准确地“封装”了特殊行为映射或者过滤属性。

函数表达式也可以封装边界效应，看下面的例子：

```
fun(X) -> io:format("Element: ~p~n",[X]) end
```

函数将打印有关它的参数消息到标准输出，而下面这个：

```
fun(X) -> Pid ! X end
```

函数将它的参数作为消息发送到Pid。它们自己都可以作为参数传递给foreach，它对列表中的每个元素执行一个操作：

```
foreach(F,[]) ->  
  ok;
```

```
foreach(F,[X|Xs]) ->
  F(X),
  foreach(F,Xs).
```

以下是foreach的执行:

```
5> hof1:foreach(fun(X) -> io:format("Element: ~p~n",[X]) end, [2,3,4]).
Element: 2
Element: 3
Element: 4
ok
6> hof1:foreach(fun(X) -> self() ! X end, [2,3,4]).
ok
7> flush().
Shell got 2
Shell got 3
Shell got 4
ok
```

请注意, foreach不仅只在hof1例子中定义, 而且也在lists模块中定义。把一个fun作为参数的函数称为高阶函数。lists模块中有一个它们的集合, 我们很快就会一起介绍。

函数作为结果

你已经看到函数如何作为其他函数的参数, 以及如何使用fun表达式描述“匿名”函数。现在我们来了解一下在编写的时候如何结合它们, 以及使用以函数作为其结果的函数。让我们从一个例子开始:

```
times(X) ->
  fun (Y) -> X*Y end.
```

这是一个接收一个参数X的函数,

```
times(X) -> ...
```

而其结果是如下的表达式:

```
fun (Y) -> X*Y end.
```

当上述函数应用到Y, 它将乘上X。运行结果如下:

```
8> Times = hof1:times(3).
#Fun<hof1.3.75238199>
9> hof1:times(3)(2).
* 1: syntax error before: '('
10> (hof1:times(3))(2).
6
11> Times(2).
6
```

对hof1:times(3)的求值给出了有些神秘的结果，这个值是一个函数。因此，我们可以尝试把它应用到2，通过在命令9中使用一般的函数。在第一次尝试中，我们得到了一个语法错误，但是在命令10中正确地使用了括号，你会注意到，和我们期望的一样，times(3)应用到2得到结果6。

利用这一点，我们还可以给出doubleAll的另一个定义：

```
double(Xs) ->
  map(times(2),Xs).
```

函数map的第一个参数是times(2)，它是应用times到2的结果的一个函数。另一个例子由以下函数给出：

```
sendTo(Pid) ->
  fun (X) ->
    Pid ! X
  end.
```

这个例子是对给定的Pid返回这个函数，它发送参数X作为消息传递给Pid。

注意：在本章的开始提到Erlang中的函数就像任何其他数据值一样。这里有一个例外的情况：对于一个动态创建的函数你所不能做的一件事情就是生成它，因为它不属于任何特定的模块。

利用已定义的函数

Erlang还提供了如何使用那些已经定义作为其他函数的参数，或者作为变量的值的函数。在一个模块M中，元数为n局部函数F可以表示为fun F/n。在这个模块之外，我们把它标记为fun M:F/n，就如下面的例子：

```
12> hof1:filter(fun hof1:palin/1,[[2,2],[2,3]]).
[[2,2]]
13> Pal = fun hof1:palin/1.
#Fun<hof1.palin.1>
14> hof1:filter(Pal,[[2,2],[2,3]]).
[[2,2]]
```

而palin/1在模块hof1中如下定义：

```
palin(X) -> X == reverse(X).
```

fun foo/2只是表示法fun(A1, A2) ->foo(A1,A2) end的语法糖（注1）。但是，“完全

注1：短语语法糖由Peter Landin发明，被用作速记表示法，它的使用使语言更加“甜美”，但不增加其表现力。

限定”表示法fun M:F/N正如我们在第8章解释的对软件更新有特别的意义。当调用fun的时候，它将始终使用该模块当前版本中的定义，而其他fun调用会到其fun定义所在的模块中，但这可能会是模块的旧版本。

请记住，当把fun传递到Erlang其他节点的时候，如果局部及全局函数调用是在fun内完成的，那么定义这些调用函数定义所在的模块必须存在于远程Erlang节点的代码搜索路径中。

注意：在Erlang的遗留代码中，你可能会遇到表示法{Module,Function}，这个元组表示一个fun。fun的元数在运行中根据调用时参数的个数确定。这种做法现在不推荐使用而且也不应该再使用。

函数和变量

所有在fun表达式中引入的变量都认为是新变量，所以它们会遮盖任何一个已经存在的且具有相同名称的变量。这意味着不能再访问在fun表达式内被遮盖的变量，而退出fun的时候它们仍然可以访问。所有其他变量可以在fun内访问到。让我们用一个例子证明这一点：

```
foo() ->
  X = 2,
  Bump = fun(X) -> X+1 end.
  Bump(10).

15> funs:foo()
11
```

```
bar() ->
  X = 3,
  Add = fun(Y) -> X+Y end,
  Add(10)

1> funs:bar()
13
```

在foo中，Bump中的局部变量X遮盖了较早的X，而且foo()的结果表明在Bump(10)中，X通过参数传递获得值10。在这个bar的例子中，很显然X的原始定义可以在Add内访问。

预定义高阶函数

lists模块包含一个高阶函数的集合，也就是说，函数把fun作为参数并且把它们应用到一个列表。这些高阶函数允许隐藏函数调用中的递归模式，并隔离所有的边界效应和在fun中对列表元素的操作。以下列表包含了在lists模块中定义的高阶函数的最常见的递归模式：

all(Predicate, List)

如果Predicate，也就是一个返回一个布尔值的fun，当把它应用到列表的每个元素上返回值都是true的时候返回true，否则返回false。

`any(Predicate, List)`

如果谓语句表中的任意一个元素对Predicate的返回值为true则返回true，否则返回false。

`dropwhile(Predicate, List)`

如果去除列表的头部和递归尾部Predicate，则返回true。

`filter(Predicate, List)`

移除列表中Predicate是false的所有元素，返回一个剩余元素的列表。

`foldl(Fun, Accumulator, List)`

带有两个参数的函数Fun。这些参数是来自List和Accumulator的一个元素。fun返回新的Accumulator，一旦列表已经遍历完毕，这也是该调用的返回值。与它的姐妹lists:foldr/3功能不同，lists:foldl/3由左到右以尾递归方式遍历列表：

```
foldl(F, Accu, [Hd|Tail]) ->
  foldl(F, F(Hd, Accu), Tail);
foldl(F, Accu, []) when is_function(F, 2) -> Accu.

foldr(F, Accu, [Hd|Tail]) ->
  F(Hd, foldr(F, Accu, Tail));
foldr(F, Accu, []) when is_function(F, 2) -> Accu.
```

`map(Fun, List)`

带一个Fun参数，它应用于列表中的每个元素。它返回一个包含fun应用结果的列表。

`partition(Predicate, List)`

把一个列表分成两个列表，其中一个包含Predicate返回true的元素，另一个列表包含Predicate返回false的元素。

打开lists模块的手册，你可以查看到那些刚才我们已经列出的但在本章没有讨论过的高阶函数。最重要的是在继续阅读之前，请在终端中尝试它们：

```
16> Bump = fun(X) -> X+1 end.
#Fun<erl_eval.19.120858100>
17> lists:map(Bump, [1,2,3,4,5]).
[2,3,4,5,6]
18> Positive = fun(X) -> if X < 0 -> false;
                        X >= 0 -> true
                        end
                        end.
#Fun<erl_eval.19.120858100>
19> lists:filter(Positive, [-2,-1,0,1,2]).
[0,1,2]
20> lists:all(Positive, [0,1,2,3,4]).
true
21> lists:all(Positive, [-1,0,1]).
```



```
false
22> Sum = fun(Element, Accumulator) -> Element + Accumulator end.
#Fun<erl_eval.12.113037538>
23> lists:foldl(Sum, 0, [1,2,3,4,5,6,7]).
28
```

延迟求值和列表

在第2章中我们给出了列表的递归定义，你可能会记起一个适当或格式正确的列表要么是一个空表，要么其头部是一个元素而尾部是一个适当或者格式正确的列表的列表。Erlang的求值机制意味着当一个列表传递给一个函数，在函数本身执行之前该列表会被完全求值。其他函数式的语言，特别是Haskell，实现的是按需求驱动（或延迟）求值。在延迟求值的情况下，参数只有在必要的时候才在函数体内求值；对于数据结构，如列表，只有那些有需要的部分才会求值。

在Erlang中建立一个延迟列表是有可能的。要做到这一点，可以使用这样的列表，尾部是一个fun，它返回一个新的头部和一个递归fun。这将避免生成一个大的列表，然后遍历它。相反，你可以减少内存的使用并只在你需要的时候生成后续的值。

作为一个例子，下面的列表包含从零开始的整数的无穷序列。头部是第一个序列数，而尾部是一个构造，它将递归生成下一个序列数和新的尾部：

```
next(Seq) ->
  fun() -> [Seq|next(Seq+1)] end.
```

在终端中运行调用或一个递归函数时，它可能这样发送序列数：

```
24> SeqFun0 = sequence:next(0).
#Fun<sequence.0.31154838>
25> [Seq1|SeqFun1] = SeqFun0().
[0|#Fun<sequence.0.31154838>]
26> [Seq2|SeqFun2] = SeqFun1().
[1|#Fun<sequence.0.31154838>]
```

建立一个函数库，实现类似map/2和foldl/3操作延迟列表的函数是有可能的。我们把这个留给读者作为练习。

列表解析

典型的列表操作是映射——应用一个函数到列表中的每一个元素，还有过滤——选择一个列表中具有特定属性的元素。通常这些操作一起使用，并且列表解析表示法为你提供了强大和简洁的方式来编写这样的列表构造。我们首先来看一些简单的例子，然后以一个简单的数据库例子来了解更多的细节。

第一个实例

在下列代码的列表解析中包括生成器 `X <- [1,2,3]`，这意味着X依次应用于值1、2和3。每个输出为`X+1`，也就是符号`||`之前的表达式使1得到2，等等。最后的结果是`[2,3,4]`。

```
1> [X+1 || X <- [1,2,3]].  
[2,3,4]
```

在接下来的代码段中，有一个过滤器：`X rem 2 == 0`，只选择那些通过测验的Xs。在这里，X是偶数：

```
2> [X || X <- [1,2,3], X rem 2 == 0].  
[2]
```

下面的代码结合了这两个：选择`[1,2,3]`中的偶数元素，然后它们中的每个值都加1。

```
3> [X+1 || X <- [1,2,3], X rem 2 == 0].  
[3]
```

这些是列表解析的基础知识。在我们看更大的例子之前先了解列表解析的通用形式。

通用列表解析

通常来说，一个列表解析有三个组成部分：

```
[ Expression || Generators, Guards, Generators, ... ]
```

生成器

生成器的形式为`Pattern <- List`，其中Pattern是一种与List表达式中的元素相匹配的模式。符号`<-`可以理解为“来自”。它也像数学符号 \in ，表达意思“是一个元素来自”。

保护元

保护元（Guards）就像函数定义中的保护元，其结果是true或者false。保护元中的变量是指那些在保护元左边出现的发生器（任何其他变量在外部级别定义）。

表达式

表达式会指定结果元素的形式。模式匹配执行两个任务，正如它在函数定义中的一样：它允许一个复杂的元素（如元组）匹配成员组成部分，而且它只选择那些和模式相匹配的元素，如下面的例子所示：

```
1> Database = [ {francesco, harryPotter}, {simon, jamesBond},  
                {marcus, jamesBond}, {francesco, daVinciCode} ].  
...  
2> [Person || {Person,_} <- Database].
```

```

[francesco,simon,marcus,francesco]
3> [Book || {Person,Book} <- Database, Person == francesco].
[harryPotter,daVinciCode]
4> [Book || {francesco,Book} <- Database].
[harryPotter,daVinciCode]
5> [Person || {Person,daVinciCode} <- Database].
[francesco]
6> [Book || {Person,Book} <- Database, Person != marcus].
[harryPotter,jamesBond,daVinciCode]
7> [Person || {Person,Book} <- Database, Person != marcus].
[francesco,simon,francesco]
8> [{Book, [Person || {Person,B} <- Database, Book==B ]} || {_,Book} <- Database].
[{harryPotter,[francesco]},
 {jamesBond,[simon,marcus]},
 {jamesBond,[simon,marcus]},
 {daVinciCode,[francesco]}]
9> [{Book,[ Person || {Person,Book} <- Database ]} || {_,Book} <- Database].
[{harryPotter,[francesco,simon,marcus,francesco]},
 {jamesBond,[francesco,simon,marcus,francesco]},
 {jamesBond,[francesco,simon,marcus,francesco]},
 {daVinciCode,[francesco,simon,marcus,francesco]}]

```

在命令4中，我们有生成器{francesco,Book} <- Database，该模式只匹配那些第一个元素是francesco的对。因此列表解析的结果是所有francesco借出而没有返回的Book。例子中的命令5、命令6和命令7中展示了使用生成器和保护元给出更复杂的结果。

在命令8中，一个列表解析作为结果表达式的一部分出现。主要的生成器遍历数据库中所有的书籍，并为其中的每个产生一个配对[Person || {Person,B} <- Database, Book==B]，它是一个所有借图书的人的列表。

命令9用作对比并展示了每个在生成器中出现的变量都是一个新变量。因此，在内部生成器中出现的变量book是新变量，而且不会如你所料的那样模式匹配外部Book变量的值。从命令8和命令9的结果可以明显看出差异。

多个生成器

一般来说，一个列表解析有一个以上的生成器，中间可以散布一些保护元。在下面一个例子中，我们使用lists:seq(N,M)表示从N到M的整数的列表（例如，lists:seq(1,4)表示为[1,2,3,4]）：

```

10> [ {X,Y} || X <- lists:seq(1,3), Y <- lists:seq(X,3) ].
[{1,1},{1,2},{1,3},{2,2},{2,3},{3,3}]
11> [ {X,Y} || X <- lists:seq(1,4), X rem 2 == 0, Y <- lists:seq(X,4) ].
[{2,2},{2,3},{2,4},{4,4}]
12> [ {X,Y} || X <- lists:seq(1,4), X rem 2 == 0, Y <- lists:seq(X,4), X+Y>4 ].
[{2,3},{2,4},{4,4}]

```

生成器是嵌套的，所以结果是在命令10中，X赋值1，然后Y将遍历整个列表[1,2,3]，这是当X为1时生成器列表的值，然后X给定一个值而Y遍历[2,3]运行，如此循环下去。

在命令11中，一个保护元跟随在X的生成器之后，它只允许X为偶数值。在命令12中，最后的保护元展示了在保护元中如何使用多个变量。

标准函数

你可以使用列表解析来定义在列表模块中出现的一些标准列表处理函数。看一下列表模块的自身实现，你会发现，在一些情况下它使用了列表解析：

```
map(F,Xs)    -> [ F(X) || X <-Xs ].
filter(P,Xs) -> [ X || X <-Xs, P(X) ].
append(Xss)  -> [ X || Xs <- Xss, X <- Xs ].
```

下面是一个更长点的例子，即查找一个列表的所有排列的函数：

```
perms([]) ->
  [[]];
perms([X|Xs]) ->
  [ insert(X,As,Bs) || Ps <- perms(Xs),
    {As,Bs} <- splits(Ps) ].

splits([]) ->
  [{[]},[]];
splits([X|Xs] = Ys) ->
  [ {[X],Ys} | [ { [X|As] , Bs } || {As,Bs} <- splits(Xs) ] ].

insert(X,As,Bs) ->
  lists:append([As,[X],Bs]).
```

该算法通过找到所有Xs的排列构建了所有[X|Xs]的排列，而且所有X可以插入的排列的方法由splits函数给出。另一种替代的算法是通过产生列表中的元素Y和所有去除Y的列表的排列P返回[Y|P]（编码留给读者作为练习）。

通过使用列表解析，实现一个三行代码的快速排序的例子，以此来结束这个讨论。给定一个列表，我们将其拆分成头部和尾部。使用头部作为一个支点，产生两个新的列表。第一个列表包含所有小于或等于支点的元素，第二个列表包含所有大于支点的元素。然后我们对新生成的列表进行递归调用，并把支点放在中间把它们合并起来：

```
qsort([]) -> [];
qsort([X|Xs]) ->
  qsort([Y || Y<-Xs, Y <= X]) ++ [X] ++ qsort([Y || Y<-Xs, Y > X]).
```

Fun和列表解析的诞生

早在Erlang成为开放源代码发布之前，Joe Armstrong向我（Francesco）挥手示意到他的办公室并且热情地向我介绍一些刚刚添加到语言中的新功能。他描述了如何抽象出递归的模式——所有的边界效应（指在函数定义的边界处容易发生的编译错误——译者注）都封装在fun函数里——并使用高阶函数遍历列表。他使用了本书中介绍的相同的快速排序的例子，继续展示列表解析的强大。Joe让我在我的程序中使用fun和列表解析，但是不要告诉任何人关于这方面的事情，关于这一事实当我离开他的办公室时并没有多想！

几个月后，当Erlang 5.4发布版本投入生产时，一封电子邮件发送到爱立信的内部邮件列表中。有人偶然发现了++结构，然后想知道发行版中是否还包含其他“没有文档的功能”。Erlang的产品经理开始也感到惊讶，因为一个将加入fun和列表解析的要求已经由于其他优先的事项在几个月前搁置了。通过调查编译器使用的解析器，他很快了解了真相。然后他的外交答复是，任何Erlang中没有文档说明的功能，包括++，不会得到官方的支持，而且不能保证其成为未来版本中的一部分。

然而，根据邮件列表上的回应，很明显我不是唯一一个和Joe谈话的人。主要项目已经（没有告诉任何人）开始使用这些“没有文档描述的特性”，它们创造了很多关键的程序，这确保了列表解析和fun成为Erlang中长期存在（并带有文档的）的一部分。

二进制类型和序列化

有时大量的结构化数据必须在计算机之间转移或储存。应该如何使协议确保数据能够尽可能快速和高效地产生和传播呢？答案是利用一切可能的储存位，字中的每个位尽量包含可能多的信息。Erlang的二进制类型为操作二进制数据结构提供了一个模式匹配表示法，使这类较低级别的编程更简单和可靠，而且比使用元组或列表更节省空间。除了二进制类型，在接下来的内容中，我们也会用树（tree）作为例子，以便查看高级别的数据结构如何高效地进行序列化和反序列化。

二进制类型

二进制类型是指向一个原始、无类型的内存块的索引。最初它被Erlang运行时系统用作通过网络下载代码，但很快就应用在一个更通用的基于套接字的通信设置中。通过内置函数提供了编码、解码和操作二进制的功能，二进制类型能更有效地传输大量数据：

```

1> Bin1 = term_to_binary({test,12,true,[1,2,3]}).
<<131,104,4,100,0,4,116,101,115,116,97,12,100,0,4,116,114,
  117,101,107,0,3,1,2,3>>
2> Term1 = binary_to_term(Bin1).
{test,12,true,[1,2,3]}
3> Bin2 = term_to_binary({cat,dog}).
<<131,104,2,100,0,3,99,97,116,100,0,3,100,111,103>>
4> Bin3 = list_to_binary([Bin1, Bin2]).
<<131,104,4,100,0,4,116,101,115,116,97,12,100,0,4,116,114,
  117,101,107,0,3,1,2,3,131,104,2,100,...>>
5> Term2 = binary_to_term(Bin3).
{test,12,true,[1,2,3]}
6> {Bin4,Bin5} = split_binary(Bin3,25).
<<131,104,4,100,0,4,116,101,115,116,97,12,100,0,4,116,
  114,117,101,107,0,3,1,2,3>>,
<<131,104,2,100,0,3,99,97,116,100,0,3,100,111,103>>
7> Term4 = binary_to_term(Bin5).
{cat,dog}
8> is_binary(Term4).
false
9> is_binary(Bin4).
true

```

在前面的代码中，相反用途的内置函数`term_to_binary/1`和`binary_to_term/1`把项元编码和解码为二进制类型，即字节序列。函数`is_binary/1`是一个用来测试二进制类型的保护元。当处理八进制的字节流时，即整数列表，你可以使用`list_to_binary/1`和`binary_to_list/1`。

注意：最好使用`term_to_binary`把在单一二进制中的一连串的元素编码为一个单一的列表。把元素单独编码，使用函数`list_to_binary`连接结果，然后通过解码给出意外结果。例如，在前面代码的命令5中，程序`binary_to_term(Bin3)`解码第一段中的第一个元素。要想把它们全部解码，首先是分割，然后分别对它们进行解码（如命令6和命令7）。

这里讨论的内置函数工作于字节级别，并且可以看到，相对于使用模式匹配方式就可用于处理任何长度位串的位语法而言，内置函数相对较麻烦（包括连接和分裂这样的串）。

位语法

本节中介绍的位语法允许把二进制看做一些段，它是位的序列但不必是字节（不必以字节边界对齐）。我们使用术语`bitstring`来表示任意长度的位序列，而术语`binary`是指字符串、它的长度是可被8整除，所以可以看做一个字节序列。

你可以使用下面的位语法来构建Bin：

```
Bin = <<E1, E2, ..., En>>
```

你可以像这样来对它们进行模式匹配：

```
<<E1, E2, ..., En>> = Bin
```

以下是一些位语法的实例：

```
1> Bin1 = <<1,2,3>>.
<<1,2,3>>
2> binary_to_list(Bin1).
[1,2,3]
3> <<E,F>> = Bin1.
** exception error: no match of right hand side value <<1,2,3>>
4> <<E,F,G>> = Bin1.
<<1,2,3>>
5> E.
1
6> [B|Bs] = binary_to_list(Bin1).
[1,2,3]
7> Bin2 = list_to_binary(Bs).
<<2,3>>
```

在前面的代码中，内置函数binary_to_list/1和list_to_binary/2提供了列表和二进制之间的相互转换，这样方便二进制以列表的方式进行操作。

二进制真正的长处在于，Bin的每个表达式都可以通过一个size和/或者type表达式来进行限定：

```
Expr:Size/Type
```

这些限定允许对数字格式进行精确控制，包括整数和浮点数，并意味着位程序可以当做高层次的协议的规范进行阅读，而不是协议的低级别（和不透明）实现。我们从接下来的TCP段操作的实例中可以看到这一点。

现在我们具体看一下size和type。

Size

指定的大小以位为单位。一个整数的默认大小为8，一个浮点数的大小是64（8字节）。

Type

type是一个类型说明符列表，并以连字符分隔。类型说明符可以是以下任何内容：

type

有效的类型是整数、浮点数、二进制、字节、位和位串。

sign

有效的sign包括signed和unsigned（默认值）。在signed的情况下，第1位确定sign：0为正数，1为负数。

endian值

endian值取决于CPU。endian值可以是big（默认）、little或native。在big endian的情况下，第一个字节表示最低位；在little endian的情况下，第一个字节表示最高位。只有当在不同的CPU架构中传输二进制时才需要使用endian值。如果你希望endian值在运行时确定，请使用native。

指定单位，如unit:Val

条目使用的位的数目是Val*N，其中N是该值的大小。位（bit）和位串（bitstring）缺省的单位是1；对于二进制（binary）和字节（byte）则是8。

下面的代码片段展示了这些类型的运用：

```
1> <<5:4, 5:4>>.
<<"U">>
2> <<Int1:2, Int2:6>> = <<128>>.
<<128>>
(foo@Vaio)3> Int1.
2
(foo@Vaio)4> Int2.
0
5> <<5:4/little-signed-integer-unit:4>>.
<<5,0>>
6> <<5:4/big-signed-integer-unit:4>>.
<<0,5>>
7> <<5:2,5:8>>.
<<65,1:2>>
```

请注意<<5:4,5:4>>如何会返回<<"U">>。整数5用等价的四个位表示为0101。在我们的二进制中，我们把它们两个放到一起，01010101，就是整数85，表示ASCII中的U。把85放入列表，你就又得到大写字母U的字符串表示法。<<"Hello">>和<<\$H,\$e,\$l,\$l,\$o>>是一样的，或等值的ASCII表示法：<<72,101,108,108,111>>。

按位模式匹配

匹配二进制的时候可以使用相同的语法，特别是在模式匹配中使用size和type。当类型省略的时候，默认类型是一个整数：

```
1> A = 1.
1
2> Bin = <<A, 17, 42:16>>.
<<1,17,0,42>>
3> <<0:16,E,F/binary>> = Bin.
```



```
<<1,17,0,42>>
4> [D,E,F].
[273,0,<<"*>>]
```

正如你所看到的，一个二进制序列解包的方式和构造的方式完全不同。这是复杂协议处理的数据机制之一，因此可以对帧进行编码和解码，并且Erlang使这些变得很简单：

```
5> Frame = <<1,3,0,0,1,0,0,0>>.
<<1,3,0,0,1,0,0,0>>
6> <<Type, Size, Bin:Size/binary-unit:8, _/binary>> = Frame.
<<1,3,0,0,1,0,0,0>>
7> Type.
1
8> Size.
3
9> Bin.
<<0,0,1>>
```

可以对任何长度的位串进行匹配，如下所示：

```
10> <<X:7/bitstring,Y:1/bitstring>> = <<42:8>>.
<<"*>>
11> X.
<<21:7>>
12> Y.
<<0:1>>
```

在二进制模式匹配中有一些可能的陷阱：

- 系统不会理解表达式`B=<<1>>`，因为它会被读作`B = <<1>>`，所以重要的是永远把符号`<<`和`=`用一个空格隔开。
- 系统不会明白表达式`<<X+1:8>>`。诀窍是给算术运算式加上括号`<<(X+1):8>>`。
- `<<X:7/binary,Y:1/binary>>`永远不会匹配成功，因为每一个二进制序列在模式匹配中的长度必须是8的倍数。
- 不能使用一个未分配的变量作为一个片段的大小，如同在函数定义中：`foo(N, <<X:N, ...>>) -> ...`。这里你需要使用一个已定义的值。但是，如果变量已定义，那么使用它来指定段的大小是可以的。

位串解析

列表解析表示法`[... || X <- List, test, ...]`是一种有力的方式，可以用它来描述通过“生成和测试”方法从其他列表中生成的列表。位串内涵符号为位串实现一个类似的作用。如下列所示：

```
13> << <<bnot(X):1>> || <<X:1>> <= <<42:6>> >>.
<<21:6>>
```

位串解析由<< ... || ... >>分隔，而且<=用于生成器（而不是<-）。最重要的是，所有的位串实体都封装在<<...>>中。在前面的例子中，变量X是位变量，将依次分配到位101010（42的进制格式）。它们中的每一个在输出时由bnot取反，得到字符串010101，从而得到21的进制表示。

位串内涵的原理与列表相同：模式匹配语法用于生成器（在<=符号的左侧），并且正如前面所解释的那样，结果可以用位语法描述。

位语法示例：解码TCP段

当你为传输控制协议（TCP）编码或解码段时，正是位语法发挥作用的时候（注2）。

TCP段包括一个紧跟着数据的头部（使用定义的结构）。头部有10个强制性的字段——其中每一个由8个1位的标签组成——和一个可选项。数据头部的大小（以32位字为单位）是通过（4位）Data Offset（数据偏移量）指定，由此可以计算出头部可选部分的大小。此项必须最少为5（最大为15）。

下面的代码展示了解码函数和两个数据包样本：

```
decode(Segment) ->
  case Segment of
    << SourcePort:16, DestinationPort:16,
      SequenceNumber:32,
      AckNumber:32,
      DataOffset:4, _Reserved:4, Flags:8, WindowSize:16,
      Checksum:16, UrgentPointer:16,
      Payload/binary>> when DataOffset>4 ->

      OptSize = (DataOffset - 5)*32,
      << Options:OptSize, Message/binary >> = Payload,
      <<CWR:1, ECE:1, URG:1, ACK:1, PSH:1, RST:1, SYN:1, FIN:1>> = <<Flags:8>>,
      %% Can now process the Message according to the
      %% Options (if any) and the flags CWR, ..., FIN.
      binary_to_list(Message);

    _ ->
      {error, bad_segment}
  end.

seg1() ->
  << 0:16, 0:16,
    0:32,
    0:32,
    5:4, 0:4, 0:8, 0:16,
```

注2：在互联网工作任务组的一系列RFC(Requests For Comment)中具体描述TCP。RFC 4614提供了它的一个概述，即“A Roadmap for Transmission Control Protocol (TCP) Specification Documents”，参见<http://tools.ietf.org/html/rfc4614>。

```

0:16, 0:16,
"message">>.

seg2() ->
<< 0:16, 0:16,
0:32,
0:32,
7:4, 0:4, 0:8, 0:16,
0:16, 0:16,
0:64,
"message">>.

```

这个定义让人感到高兴的是，case语句中的模式是关于段看起来像什么的一种可读的定义（还不是正式的）。它以由32位项确认的顺序和数量为开始的两个16位字代表源和目标。

到目前为止，我们在字节边界进行了匹配，下一步将匹配4个变量：

```
DataOffset:4, _Reserved:4, Flags:8, WindowSize:16
```

这里给出了DataOffset，4个保留的位（所以匹配“无关紧要”变量），表示标签的8个位，等等。在匹配更多的字段之后，二进制的余数和Payload/binary相匹配。这个匹配把DataOffset至少为5的检查作为保护元。

该分支的语句里也使用了模式匹配。在下列语句中：

```
<< Options:OptSize, Message/binary >> = Payload,
```

任何Option都是从Payload中取得。假设没有，那么Option将和空的二进制<<>>相匹配，而且Payload将会是这个Message。无论何种情况，下面的模式匹配：

```
<<CWR:1, ECE:1, URG:1, ACK:1, PSH:1, RST:1, SYN:1, FIN:1>> = <<Flags:8>>,
```

将同时从Flag字节中提取8个1位标签。

这两个片段：seg1/0和seg2/0，显示出该段的结构和解码函数中的模式匹配完全一样。这些数据可用于测试这里显示的存根解码器的功能。

位运算符

Erlang中的位级的运算符可用于整数，而结果也返回整数。表9-1列出了位运算符。

表9-1：位运算符

| 运算符 | 描述 |
|------|------------------|
| bond | 按位与 |
| bor | 按位或 |
| bxor | 按位异或 |
| bnot | 按位非 |
| bsl | 按位左移，第2个参数给出移动位数 |
| bsr | 按位右移，第2个参数给出移动位数 |

在下面的例子中，运算符应用于 $17 = 10001_2$ 和 $9 = 1001_2$ ，其中包括：

```
1> 9 band 17.  
1  
2> 9 bor 17.  
25  
3> 9 bxor 17.  
24  
4> bnot 9.  
-10  
5> bnot (bnot 9).  
9  
6> 6 bsr 1.  
3  
7> 6 bsl 4.  
96  
,
```

序列化

在本节中，我们将展示如何在Erlang中序列化二叉树。你可以用嵌套的元组表示二叉树：用{node, ..., ...}代表一个节点和用{leaf, ...}代表一片叶子，如图9-1所示。一种用来序列化数据结构的方法是把这个结构格式良好地打印为一个完全置于括号内的字符串；图9-1中的例子将以"{node,{node,{leaf,cat ...}"开始。通过解析可以来反序列化像这样的字符串，但这既不是一个有效率的存储形式，也不是一个有效率的反序列化机制。

当序列化结构的时候，你可以知道所产生的表达式的大小。因此，你可以把这些信息集成到序列化中从而避免进行结构解析。最初的时候，你可以把此结构转换为一个流，其中在每个子树之前你记录下树的表达式的大小。

在这个例子中，你会得到[11,8,2,cat,5,2,dog,2,emu,2,fish]，其中，11是整个表达式的长度，8是列表[8,2,cat,5,2,dog,2,emu]的长度，它代表顶级节点的左子树，等等。使用这个长度信息重建原始的树相对容易（我们把这个作为练习留给读者）。

然而，在序列格式里面有多余的信息。如果你有一个段代表一个节点的子树，它足以推断出（代表的）左子树的大小，而右子树由余下的给出。所以，你可以使用段[8,6,2,cat,2,dog,emu,fish]来表示图9-1中的树。下面的代码显示了转换为这一表达形式的过程：

```
treeToList({leaf,N}) ->
  [2,N];
treeToList({node,T1,T2}) ->
  TTL1 = treeToList(T1),
  [Size1|_] = TTL1,
  TTL2 = treeToList(T2),
  [Size2|List2] = TTL2,
  [Size1+Size2|TTL1++List2].
```

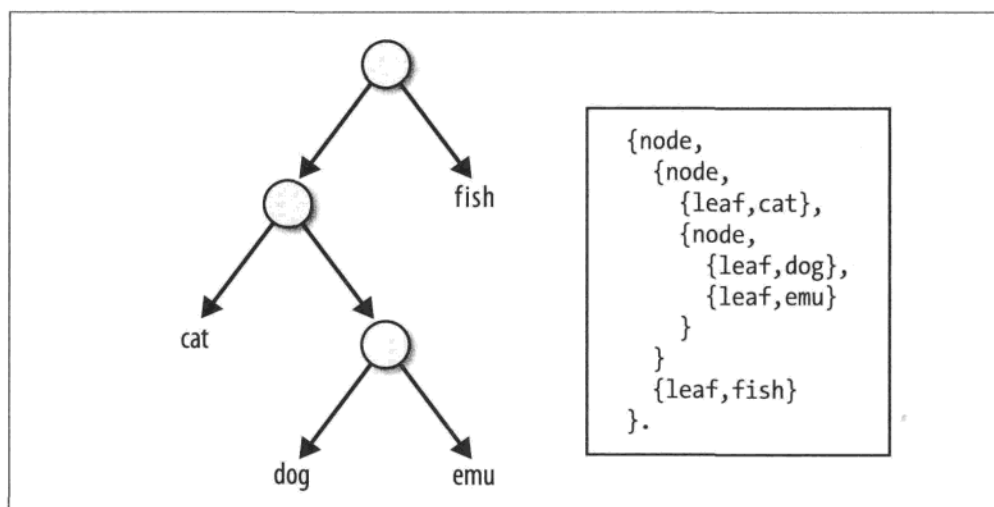


图9-1：用于序列化的二叉树

下一个代码片段展示了反序列化：

```
deserialize([_|Ls]) ->
  listToTree(Ls).

listToTree([2,N]) ->
  {leaf,N};
listToTree([N]) ->
  {leaf,N};
listToTree([M|Rest] = Code) ->
  {Code1,Code2} = lists:split(M-1,Rest),
  {node,
   listToTree(Code1),
   listToTree(Code2)
  }.
}
```

这里最重要的一点是ListToTree的最后子句，其中表达式中的长度M用作一个点，在转换两个半分支之前分割这个表达式。

一旦有一个列表表达式，你就可以直接将它转换成一个字节流。你还可以直接删除整个构造过程中的中间列表，直接将树转换为比特流。反之亦然。

引用

还记得第5章的频率服务器的例子吗？客户端发送一个分配频率的请求，然后等待{reply, Reply}格式的响应。你怎样才能确定这消息确实来自频率服务器，而不是任何其他决定以这种格式发送一个消息到客户端的进程？答案是使用引用。

你可以使用内置函数make_ref()创建引用，它在一个节点的生命周期内是（几乎）唯一的，它的值在2²⁸调用之后才会重复。它们对一个节点来说也是唯一的，所以两个不同的节点的两个引用永远不可能是相同的（注3）。

你可以比较引用是否相等，而且通过使用一个索引标记一个消息，而后在回复中返回同一个索引，用这种方式可以在一个协议中匹配调用和响应。要在频率服务器的例子中这样做，你可以使用下面的代码：

```
call(Message) ->
  Ref = make_ref(),
  frequency ! {request, {Ref, self()}, Message},
  receive
    {reply, Ref, Reply} -> Reply
  end.

reply({Ref, Pid}, Message) ->
  Pid ! {reply, Ref, Message}
```

注意我们如何把变量Ref绑定到引用的。当收到答复时，其中包括了绑定变量Ref，使得只接受模式匹配正确的信息。这保证了答复来自频率服务器，因为它是客户端唯一发送过索引的进程。

注3： 在实践中，唯一得到两个相同的引用的办法是把一个引用保存到Erlang项元的文件中，停止节点，然后重新启动节点后重新从文件加载引用；一个在新启动的节点中创建的引用可能等于那个保存的值，因为对于创建的每个节点，引用分配总是从0开始的整数。

练习

练习9-1：高阶函数

使用fun和高阶函数编写一个函数，打印出介于1和N的整数。

提示：使用`?lists:seq(1, N)`。

使用fun和高阶函数编写一个函数，给定一个整数列表和一个整数，它将返回所有小于或等于该整数的整数。

使用fun和高阶函数编写一个函数，打印出介于1和N的偶数。

提示：用两个步骤来解决这个问题，或在fun中使用两个子句。

使用fun和高阶函数编写一个函数，给定一个列表的列表，将它们连接起来。

使用fun和高阶函数编写一个函数，给定一个整数列表，返回所有整数之和。

提示：使用`lists:foldl`并尝试找出我们更喜欢使用`foldl`而不是`foldr`的原因。

练习9-2：列表解析

使用列表解析创建一个1~10的整数集，并使其能被3整除（例如，`[3,6,9]`）。

使用列表解析删除一个多态列表中的所有非整数，并返回整数的平方的列表：
`[1,hello,100,boo,"boo",9]` 应返回`[1,10000,81]`。

使用列表解析并给定两个列表，然后返回一个新列表，它是两个列表的交叉集合（例如
`[1,2,3,4,5]`和`[4,5,6,7,8]`应返回`[4,5]`）。

提示：假设列表不包含重复的元素。

使用列表内涵给定两个列表，然后返回一个新的列表，它是两个列表的对称差。使用
`[1,2,3,4,5]`和`[4,5,6,7,8]`应返回`[1,2,3,6,7,8]`。

提示：假设列表不包含重复的元素。

练习9-3：压缩打包函数

定义函数`zip`，把一对列表转换成一个成对元素的列表：

```
zip([1,2],[3,4,5]) = [{1,3},{2,4}]
```

采用这个例子定义一个函数zipWith，它以固定的步长应用一个二元函数于两个列表参数上：

```
add(X,Y) -> X+Y.  
zipWith(Add, [1,2], [3,4,5]) = [4,6]
```

请注意，在这两种情况下，比较长的列表结果将被截去。

练习9-4：现有的高阶函数

Erlang的列表模块中包含了许多高阶函数，即把函数作为参数的函数。写出你自己对这些函数的定义。

练习9-5：列表解析中的长度规格

当从整数中得到位串的时候，研究长度规格的作用。<<42:6>>和<<42:5>>的结果是什么？把这个和下列形式的模式匹配作对比：

```
<<X:4,Y:2>> = <<42:6>>.
```

此外，研究如下形式的模式匹配：

```
<<C:4,D:4>> = << 1998:6 >>.  
<<C:4,D:2>> = << 1998:8 >>.
```

练习9-6：位串

使用位串构造和模式匹配，给出“序列化”一节中序列化算法的一个位级别的实现。你需要想想各种部件需要多少存储空间，包括组成序列化部分的大小的信息。



第10章

ETS和Dets表

许多实际的系统都需要在有限的时间里存储和检索大量的数据。举例来说， 移动手机应用程序访问和操作用户信息来处理通话的同时， 还需要计费和服务支持。在软实时系统中， 搜索时间与被搜索数据量成正比是不能接受的。查询时间不仅需要是常量，而且要很短。

在程序中主要使用的一个数据组合类型是一些项（或者是元素或者是对象）的聚合。Erlang的列表提供了实现聚合的一种方法，但是如果列表中的项超过一定数量，存取元素过程就会变慢。平均来说，我们需要校验聚合中50%的元素来确定一个给定元素的存在，而需要遍历所有的元素来确定一个给定值还存在。

为处理快速检索，Erlang使用两种机制。本章介绍Erlang项元存储（Erlang Term Storage, ETS）机制和Erlang磁盘项元存储（Disk Erlang Term Storage, Dets）机制，分别提供内存高效和磁盘高效的大量数据聚合的存储和检索。Erlang同时也提供一个完整的数据库应用程序Mnesia，这将在第13章介绍。

ETS表

ETS表存储元组，通过元组中的关键字字段存储元素。这些表是通过散列表和二叉树实现的，不同的表现形式提供不同类型的聚合。

集合和袋

在数学上，集合和袋是两种不同的聚合。它们都包含元素，但不同点是集合每个元素只包含一次，袋可以包含某个元素多次。集合和袋在建模和编程中都很实用。让我们用两个例子来比较它们的不同。

你可以用集合对参加生日聚会人员的聚合建模，因为在这个例子中某人只会来或不来。另外，你可以用袋模拟在生日聚会（注1）上收到的礼物，因为你有可能收到同样的礼物。因此，你需要存储每件礼物，不仅是保证每人都会收到你的感谢信，而且可以了解哪些可以放到ebay上卖。

Erlang语言有四种不同的ETS表，如下简要列出。为了讨论它们的不同，我们将举一个索引的例子，在索引元组中包含一组成对的单词（比如一个字符串）和数字，例如{"refactorings",4}。在本章的其余部分，我们将把元组中的第一项（包含的单词）作为关键字。

集合 (Set)

在集合中每个关键字只出现一次，所以使用这种表作为索引的例子，意味着每个词只对应表中的一个元素。

有序集合 (Ordered set)

一个有序集合与集合具有同样的属性，但其元素按关键字的字典顺序存储。比如说，记录"refactorings"会位于记录"replay"之前的位置。在第2章中我们已经介绍了对于不同数据类型的存储顺序。

袋 (Bag)

袋允许不同的记录对应同一个关键字，就像{"refactorings",4}和{"refactorings",34}是允许的。元素必须间隔开，在索引的例子中这意味着只有一个项元对应于在特定行上的一个特定单词。

复袋 (Duplicate bag)

一个复袋允许包含重复的元素，也可以有重复的关键字，所以在本章实例中，表中可能包含项元{"refactorings",4}两次或多次。

针对数学上的工具“集合和袋”，在Erlang语言中可以表达为对关键字的处理：集合和有序集合只包含每个关键字一次，而袋和复袋对于关键字的重复给以两种不同的变化。

实现和平衡

对于这些聚合的实现，除了对有序集合中的元素访问时间相对聚合大小有对数级别变化外，其他的检索时间都是恒定的。两种情况的表现都要远好于基于列表形式的线性访问时间。

集合、袋和复袋都被存储为散列表形式，其中存储元组的位置是通过函数 (hash

注1： 我们的生日在1月12日和3月26日。

function，散列函数）把元组的关键字映射到存储这个值的内存地址来确定，如图10-1所示。假设我们的散列表对10个项元分配了空间，那么散列函数会返回10个独一无二的内存位置，每个项元一个。如果我们需要插入一个附加项元，此表会重新计算散列值（重新组织），为更多的项元创建空间并且返回更多互不相同的内存空间作为关键字的散列结果。这给我们一个从关键字到对应元组的、常量的存取时间，但当我们写入一个项元和重新计算散列表时，这个时间将会变化（注2）（通常，这有可能是因为两个数据的关键字散列为同一个值）。

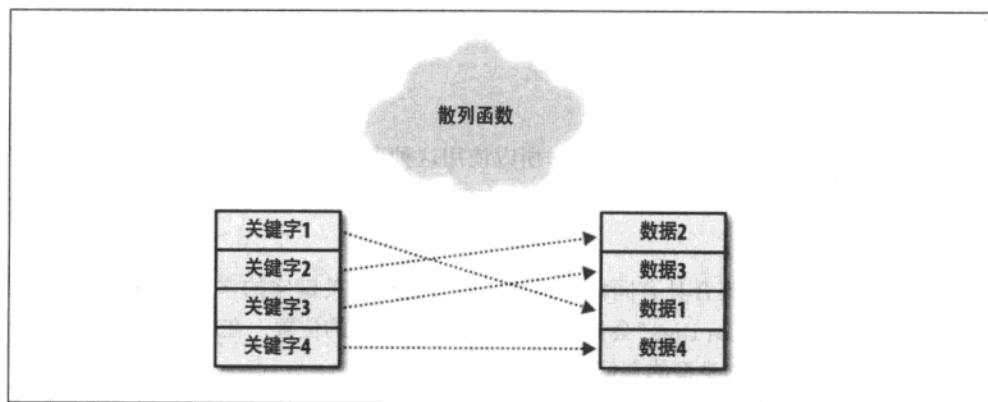


图10-1：散列表

一个有序集合存储在一个AVL平衡二叉树中，其顺序由关键字字段内在的顺序确定。这意味着分支的长度决定存取的时间复杂度，它是存储在二叉树中聚合大小的对数。图10-2 显示一个用数字表示关键字的值的平衡二叉树的例子。

既然集合和袋提供一个恒定的存取时间，那为什么还需要访问特定元素更慢的有序集合？这是因为有序集允许聚合按照键的顺序来遍历，然而其他表现形式只是简单按照存储顺序来遍历。在集合中，顺序依赖于散列函数的实现细节，同时依赖于它们在内存中的存储顺序。这显然不对应于数据的自然顺序。

选择使用哪种形式的表取决于特定的应用。举例来说，如果要写出一个复杂的索引，需要按字母（关键字）顺序遍历索引，但是要从在线索引中查找单个词汇，只需要存取某个特定的元组，这样一个无序的聚合就已经足够了。

注2： 你可以从Scott Lystig Fritchie (Erlang Workshop '03, ACM Press, 2003; <http://doi.acm.org/10.1145/940880.940887>) 的文章“A Study of Erlang ETS Table Implementations and Performance.”中获得关于实现ETS表的细节。

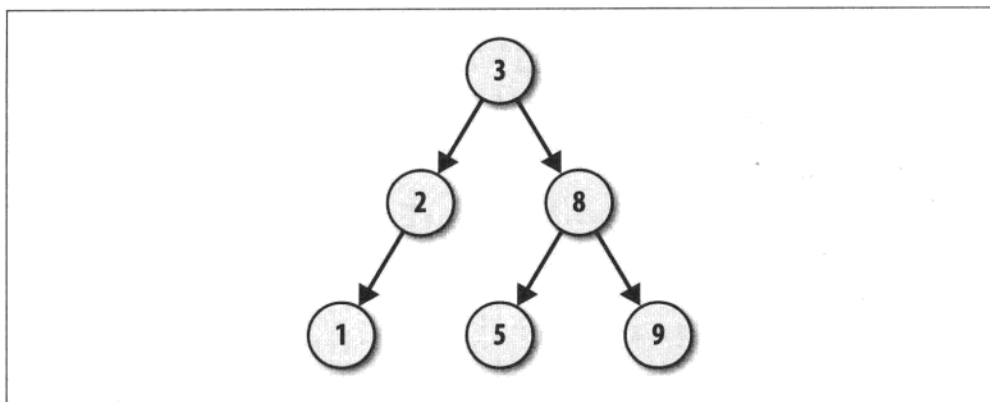


图10-2：一个平衡二叉树

Erlang发行版中ETS表的实现非常灵活，允许关键字字段可以是任何类型，包括复杂数据结构。而且已经把它高度优化，因此它构成了实现Erlang的Mnesia数据库的基础，我们将在第13章中介绍。

创建表

ets模块包含一系列创建、操作和删除ETS表的函数。通过调用`ets:new/2`创建一个表：第一个参数是表的名字，第二个参数由可选项的列表组成。函数调用`ets:new(myTable, Opts)`返回表的标识符来用作表的引用。

在默认设置中，如果传递一个空选项表给`ets:new/2`函数，则创建一个集合，其关键字在第一个位置，并提供表中值的受保护访问允许所有进程读取表，但只有表的所有者可以写入。其他选项包括：

`set`, `ordered_set`, `bag`, `duplicate_bag`

如果在选项列表中包括这些中的任意一个，则创建相应类别的ETS表。

`{keypos, Pos}`

创建一个表，其关键字在位置Pos上。

`public`, `protected`, `private`

所有进程可读写一个公共表，仅表的所有者可读写一个私有表。我们在前面已经描述过受保护表。

`named_table`

静态注册表的名字，并且可以将其在ETS操作中用作表的引用。

在介绍ets模块的文档中有关于其他设置参数的描述。你只需传入表的标识符就可以通过使用ets:info/1函数获取关于表的信息。

```
1> TabId = ets:new(myTable, []).
15
2> ets:info(TabId).
[{memory,301},
 {owner,<0.31.0>},
 {name,myTable},
 {size,0},
 {node,node@nohost},
 {named_table,false},
 {type,set},
 {keypos,1},
 {protection,protected}]
```

如果表是通过named_table的选项创建的，那么你既可以通过名字又可以通过标识符对表进行存取。

警告： 即使每个表都是通过ets:new/2创建的，且表的名字通过表的信息可以查到，也不可以把表的名字用于表的存取，除非把named_table 选项设为enabled。

如果选项没有enabled，试图通过这种方式存取表将导致坏参数运行错误信息。

当不再有程序引用表的时候，该表使用的存储空间是会被自动回收的（也就是说，它们无法进行“垃圾回收”）。因此，你需要通过调用ets:deleted(TabId)来手动删除这个表。然而，表与创建它的进程相连，如果该进程终止了，那么该表会自动被删除。

警告： 因为ETS表是和创建它的进程相连，所以你在终端测试它们时需要特别小心。如果你引发了运行时错误，那么终端进程将崩溃并重启。结果是你将会失去所有的ETS表和与它们相关的数据。如果这种情况发生了，那么可以使用终端命令f()清除所有与你的表引用有关的变量并重新启动。

处理表元素

你可以使用ets:insert/2函数插入元素到表格，并且通过ets:lookup/2函数根据关键字来存取它们：

```
3> ets:insert(TabId,{alison,sweden}).
true
4> ets:lookup(TabId,alison).
[{alison,sweden}]
```

在这个例子中，TabId是一个集合，插入以alison为关键字的第二个元素会覆盖第一个

元素。当处理集合时一个通常的错误做法是，在插入一个更新前先删除一个元素。这个删除操作是多余的，因为插入会自动覆盖旧的项元。

```
5> ets:insert(TabId,{alison,italy}).
true
6> ets:lookup(TabId,alison).
[{alison,italy}]
```

如果你删除了现存的表并重建这个表为袋，你将会看到一些不同的变化：

```
7> ets:delete(TabId).
true
8> TabId2 = ets:new(myTable,[bag]).
16
9> ets:insert(TabId2,{alison,sweden}).
true
10> ets:insert(TabId2,{alison,italy}).
true
11> ets:lookup(TabId2,alison).
[{alison,sweden},{alison,italy}]
```

保留袋中元素的插入顺序。在结果中元素的顺序即是它们加入袋中的顺序，最先进入的最先出来。

因为这个ETS表与复袋相比更像是一个袋，所以第二次插入同一个元组没有任何效果。

```
12> ets:insert(TabId2,{alison,italy}).
true
13> ets:lookup(TabId2,alison).
[{alison,sweden},{alison,italy}]
```

实例：建立索引第一部分

作为一个ETS表的应用，接下来介绍为一个文本文档设计索引的例子。当在后面的内容中讨论如何存取和遍历表中的数据的时候，我们还会提到这个例子。

说明：这个文档是文本文件，索引需要记录某个特定单词在文档的哪些行出现过。忽略少于4个字母的单词，并且在索引中，文档中所有的单词都规范为非缩写形式。把连续行使用范围的形式来显示。

对于前面一个段落的索引的第一部分如下所示：

| | |
|-------------|------|
| appears | 2. |
| consecutive | 3. |
| document | 1,3. |
| file | 1. |
| form | 3-4. |
| ignored | 2. |
| index | 1,3. |
| | |

这个程序的运行分为两个阶段，第一个阶段从文本文件中收集单词出现的数据 {Word,LineNumber}，然后转换为索引，就像在图10-3中显示的那样。

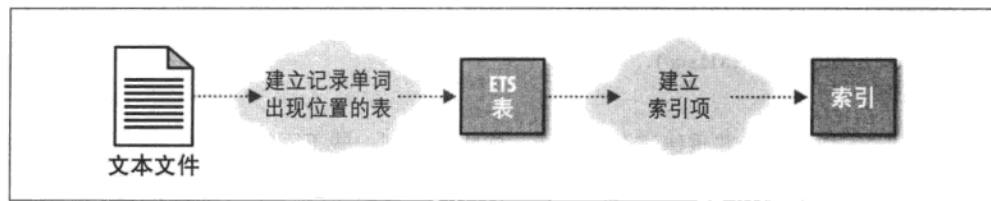


图10-3: 建立一个索引

你该使用什么类型的ETS表来存储单词出现的信息？

- 如果索引只是简单地用于特定单词的查找，你可以使用一个无序的结构。如果表的关键字是一个单词，这就需要关键字的可重复性，因为对于不同的行的记录可能对应相同的关键字。另外，不需要建立一个索引来记录在同一行重复出现的项元：在这种情况下，袋是一个恰当的数据结构选择。
- 如果完整的索引需要以字母表顺序打印，需要表是一个有序集合，那么在这种情况下关键字是什么？
 - 如果是单词，每个单词只允许有一个条目：在这种情况下，存储的记录需要包含一个行号的列表。我们将这个方法作为练习放在本章最后。
 - 我们在这里选择的方式是把关键字成对表示，使得记录在表中表示为{{Word, LineNumber}}形式的元组，即单字段元组，其唯一的字段是它自身的一个数对。

索引的上层程序包含3个独立的步骤：创建表，在表中加入数据和从表中建立索引。看示例的时候请记住，如果遇到一个不理解的新的库函数，你应该在Erlang的在线文档中查找该模块：

```
index(File) ->
    ets:new(indexTable, [ordered_set, named_table]),
    processFile(File),
    prettyIndex().
```

一旦打开文件，可以通过processlines函数每次处理一行，它将记录当前的行号：

```
processFile(File) ->
    {ok, IoDevice} = file:open(File, [read]),
    processLines(IoDevice, 1).

processLines(IoDevice, N) ->
    case io:get_line(IoDevice, "") of
```

```

eof ->
  ok;
Line ->
  processLine(Line,N),
  processLines(IoDevice,N+1)
end.

```

把每一行分割成单词，可以使用模块`regexp`中的函数和一个隐藏分割单词的符号细节的宏定义：

```

-define(Punctuation,"[ |,|\\.|;|:|\\t|\\n|\\(|\\)|]+").

processLine(Line,N) ->
  Words = re:split(Line, ? Punctuation, [{return, list}]),
  processWords(Words,N) ;

```

`processWords`函数把单词插入表中，忽略短的单词，并通过`string:to_lower/1`把其余单词变为小写：

```

processWords(Words,N) ->
  case Words of
    [] -> ok;
    [Word|Rest] ->
      if
        length(Word) > 3 ->
          Normalise = string:to_lower(Word),
          ets:insert(indexTable,{{Normalise , N}});
          true -> ok
      end,
      processWords(Rest,N)
  end.

```

这样就完成了建立索引的第一个阶段，我们将在讨论如何遍历ETS表和从中提取信息后再回到这个例子上。

表的遍历

你已经看到如何使用`ets:lookup/2`函数来查找与单个关键字相关的信息，它将返回包含某个关键字的所有元组。比如说，我们有一个索引，你需要一次一个关键字，一步一步地查看表中所有的数据。通过`ets:first/1`函数得到表中的第一个关键字，表中的下一个关键字通过`ets:next/2`得到。假设在索引的例子中，我们已经创建并填充了那个索引例子里的有序集合类型的表，我们就可以这样遍历它：

```

3> First = ets:first(indexTable).
{"appears",2}
4> Second = ets:next(indexTable,First).
{"consecutive",3}

```


如果这样做，我们可以看到元素是以数对{Word, Number}的字典顺序排序的。

如果用袋代替表，用Word作为关键字，我们会看到表在创建后完全不同的行为：

```
3> First = ets:first(indexTable).
"words"
4> Second = ets:next(indexTable,First).
"form"
```

现在关键字的顺序是由关键字的散列值的顺序确定的，既不取决于关键字自己的顺序，又不取决于值插入表中的顺序。在本质上，顺序是任意的，但对我们来说唯一可靠的是，能够通过使用first和next一步一步地遍历表来访问所有的关键字。

ETS表和并发更新

EST对并发更新只提供非常有限的支持。如果你正在使用ets:first/1和ets:next/2函数遍历集合或袋类型的ETS表，而同时又有其他进程试图写入或删除其中的元素，这将会发生什么？

如果其他进程正在写入一个新元素，而这将导致表的重新散列，完全把所有记录重新排序，这会发生什么？在最好的情况下，你的next/2会引发一个运行时错误。在最坏的情况下，这种行为将是未定义的，将返回任何元素或'\$end_of_table'基元，或者引发badarg错误。

如果你知道在你使用ets:first/1和ets:next/2函数遍历ETS表时，其他的进程执行破坏性的调用，可以使用ets:safe_fixtable/2函数。它保证在遍历的时候，每个元素你只访问一次。如果一个新的元素在遍历开始后加入（由遍历表的进程加入或者另外一个进程），那么将无法保证这个元素可以访问到。然而，所有其他的元素将只遍历一次。通过把Flag设置为true，一个进程调用ets:safe_fixtable(TableRef, Flag)锁定一个表。通过把Flag设置为false或者在进程终止时，Flag被释放。如果几个进程锁定同一个表，直到所有进程释放了这个锁定或者全部终止，锁定才被解除。

如果你锁定了一个表，不要忘记释放它，因为只要表处在锁定状态中，被删除的对象就不会从表中移除。这不仅会导致没有释放被删除对象使用的内存，还会导致表的运算性能的下降。

ets:last/1给出有序集合的最后一个元素。对于其他类型的表，ets:last/1将返回它的第一个元素。如果Last是表中的最后一条记录，那么调用next会返回'\$end_of_table'。对于最初那个索引的例子，结果是：

```

5> Last = ets:last(indexTable).
{"words",3}
6> ets:next(indexTable,Last).
'$end_of_table'

```

实例：建立索引第二部分

在这一节中，我们将向你展示如何通过遍历包含有单词出现频率的ETS表，以便创建文本文件的索引。为了建立这个索引，我们需要写一个函数来遍历表并执行两个操作：

- 收集一个特定单词的一些记录，并与其出现的行号配对，例如{"form",[4,3]}。
- 把这些元组中的每一个格式良好地输出，删除重复的行号，并把连续的行号显示为一个范围。

由prettyIndex/0函数完成遍历，它读取第一个字段{Word,N}，并且建立元组{Word,[N]}，其中保存了到目前为止所收集的关于这个Word的信息。这个部分的索引项和当前字段，以及表的标识符将一起都传入工作函数prettyIndexNext/2，它将执行遍历的操作：

```

prettyIndex() ->
  case ets:first(indexTable) of
    '$end_of_table' ->
      ok;
    First ->
      case First of
        {Word, N} ->
          IndexEntry = {Word, [N]}
        end,
        prettyIndexNext(First,IndexEntry)
      end.

```

函数prettyIndexNext/3将读取下一个记录，如果没有记录存在，那么把当前IndexEntry输出。如果有下一个元组{NextWord, M}，以下情况之一将会发生：

- 如果NextWord与IndexEntry中的单词相同，需要把M加入到包含Word的行的列表中，并以递归方式调用prettyIndexNext。
- 如果NextWord不同，在用一个新的部分索引项{NextWord, [M]}递归调用前，需要把IndexEntry输出。

```

prettyIndexNext(Entry,{Word, Lines}=IndexEntry) ->
  Next = ets:next(indexTable,Entry),
  case Next of
    '$end_of_table' ->
      prettyEntry(IndexEntry);
    {NextWord, M} ->
      if

```

```

NextWord == Word ->
  prettyIndexNext(Next,{Word, [M|Lines]});
true ->
  prettyEntry(IndexEntry),
  prettyIndexNext(Next,{NextWord, [M]})
end
end.

```

prettyEntry函数的定义将留作练习（可以参考本章结束部分的练习10-1）。

提取表的信息：匹配

你已经看到如何从ETS表中通过一个给定关键字字段来提取元组和遍历一个表，本节将向你展示如何通过模式匹配来从表中提取元素。你可以通过ets:match/2函数或更一般的调用ets:select函数完成这些，在后一种情况中，匹配规则会以更简单的形式给出，或者通过一个函数定义使用ets:fun2ms/1编译得到。

为说明match操作，让我们看一个包含3个元组的例子：

```

1> ets:new(countries, [bag,named_table]).
countries
2> ets:insert(countries,{yves,france,cook}).
true
3> ets:insert(countries,{sean,ireland,bartender}).
true
4> ets:insert(countries,{marco,italy,cook}).
true
5> ets:insert(countries,{chris,ireland,tester}).
true

```

表的元素是3元元组，所以与这个表匹配的模式也是3元元组。此模式包含3种元素：

- `'_'`是通配符，它会匹配任何一个在这个位置上的值。
- `'$0'`和`'$1'`是变量，它会匹配任何一个在这个位置上的值。
- 一个值，在这种情况下可能会是ireland或cook。

匹配的结果是给出结果的列表，它包含表中每一个成功的匹配。通配符和变量匹配的区别是，如果值与变量匹配，则会在结果中显示，而通配符匹配则不会。值对变量的匹配会在列表中以这些变量的升序给出。

这可以通过一个例子说明：

```

6> ets:match(countries,{'$1',ireland,'_'}).
[[sean],[chris]]
7> ets:match(countries,{'$1','$0',cook}).
[[france,yves],[italy,marco]]

```

在命令6中，模式要求第二个字段是ireland，而对其他字段没有限制，并且显示每个成功匹配的第一部分的值，这里有两个。

在命令7中，要求第三个字段是cook。每个成功匹配都会显示第一字段和第二字段。你可以预测下面的匹配的结果吗？请特别注意结果返回的顺序：

```
8> ets:match(countries,{'$2',ireland,'_'}).
???
9> ets:match(countries,{'_',ireland,'_'}).
???
10> ets:match(countries,{'$2',cook,'_'}).
???
11> ets:match(countries,{'$0','$1',cook}).
???
12> ets:match(countries,{'$0','$0',cook}).
???
```

你可以通过在Erlang终端中键入ETS表的创建命令（在之前代码的命令1~5）来检验你的结果（注3）。可以用match_object函数来返回全部匹配模式的元组，也可以通过match_delete删除匹配的对象：

```
13> ets:match_object(countries,{'_',ireland,'_'}).
[{sean,ireland,bartender},{chris,ireland,tester}]
14> NewTab = ets:match_delete(countries,{'_',ireland,'_'}).
true
15> ets:match_object(countries,{'_',ireland,'_'}).
[]
```

警告： 使用匹配操作要非常小心，因为它们会改变系统的实时行为。这是因为所有的匹配操作都是作为内置函数来实现的，而内置函数是原子性操作的。因此，对于一个巨大表的一个匹配操作会阻碍其他进程的执行，直到完成对整个表的遍历。

为了避免这个问题，正如前面展示的，最好使用first和next。虽然它可能需要更多的时间，但是它不会破坏系统的实时特性。

提取表的信息：筛选

一个匹配规约是一个Erlang的项元，它描述了一个小程序。它是一个模式的概括，允许以下的形式：

- 在匹配变量进行求值的保护元。
- 返回的表达式并不仅仅是一个简单的绑定的表。

注3： 或者你也可以这样检查它们：[[sean],[chris]]；[[[]],[[]]]；[]；[[yves,france],[marco,italy]]；[]。

下面是一个例子，与前面的countries ETS表一起运行：

```
16> ets:select(countries,  
    [{{'$1','$2','$3'},[{ '/=','$3',cook}],[{ '$2','$1' }]}]).  
[[ireland,sean],[ireland,chris]]
```

匹配规约的是一个3元组列表，其中每一元都大致对应于一个函数语句。这里仅仅举出一个作为例子。在这个元组中有3部分：

`{ '$1', '$2', '$3' }`

这是一个模式，与前面在ets:match函数中使用的一样。

`[{ '/=','$3',cook }]`

这是一个保护元表达式的列表，用前缀形式写成。这里的单一保护元检查条件\$3 /= cook是否满足。只有保护元判断为true，匹配才会成功。

`[{ '$2','$1' }`

这是返回表达式。

我们在本章介绍匹配规约并不是因为它的优美（你一定同意它们是很难看的构造），而是因为它的快速和与内部Erlang运行时系统的紧密结合。

由于语法的不灵活，Erlang实现支持以闭包来描述的匹配规约。函数ets:fun2ms/1把fun作为参数，描述了我们想要在ETS表中执行的比较（或）模式匹配，还有我们想要通过select返回的返回值。幸运的是，fun2ms返回一个匹配规约，我们可以把它用作参数调用select函数，免除了我们要理解和书写匹配规约的困难。

```
17> MS = ets:fun2ms(fun({Name,Country,Job}) when Job /= cook ->  
    [Country,Name] end).  
[[{'$1','$2','$3'},[{ '/=','$3',cook}],[{ '$2','$1' }]]]  
18> ets:select(countries, MS).  
[[ireland,sean],[ireland,chris]]
```

这允许书写匹配描述时采用通常的顺序来书写变量的命名和子表达式。注意fun必须是一个文字函数，而不可以作为一个变量传递给fun2ms。所谓的文字函数，我们是指在ets:fun2ms/1调用中输入的函数，而不是绑定到一个变量的那个。fun函数自己也需要一个元组作为参数。最后，如果在模块中使用，需要包含一个头文件：

```
-include_lib("stdlib/include/ms_transform.hrl").
```

函数ets:select/2会把所有成功匹配的结果返回表。你可以通过函数ets:select/3进行“批量”匹配，函数的第三个参数限制了匹配的数量。ets:select/3返回匹配并且附加一个延续，通过把它传递给ets:select/1来调用并进一步匹配。

我们将在第17章提供匹配描述的完整和惊人的细节。如果你需要一个详细的解释，或者对fun2ms到底返回了什么语法和语义感兴趣，你可以现在就到那一章去看看。

在表上的其他操作

还有其他许多关于表的操作，这里列出了其中最有用的一部分：

`ets:tab2file(TableId | TableName, FileName)`

`tab2file/2` 把表转存到文件，返回`ok`或`{error, Reason}`。

`ets:file2tab(FileName)`

`file2tab`把一个转存的表读回到系统，返回`{ok, Tab}`或`{error, Reason}`。

`ets:tab2list(TableId | TableName)`

`tab2list` 返回一个包含表中所有元素的列表。

`ets:i()`

这会列出当前进程可见的所有关于ETS表的汇总信息。

`ets:info(TableId | TableName)`

前面已讨论过`info`函数，它返回给定表的属性。

关于这些和其他函数的详细信息都包含在介绍ets模块的文档中。

记录和ETS表

我们已经带领你了解了Erlang记录的内部实现，现在你应该可以推断出也可以在ETS表中插入记录。但是这里有一个小小的例外。还记得吗，在ETS表中默认关键字位置是元组的第一个元素。在记录中，这个位置保留给了记录的类型，除非明确指定关键字的位置，否则你无法获得想要的行为。通过表达式`#RecordType.KeyField`获取`RecordType`里`KeyField`的位置，可以把`{keypos, #RecordType.KeyField}`加入到`ets:new/2`函数调用的选项列表中。

让我们试着通过终端把记录加入表中。我们先定义它们，然后创建ETS表，并且插入和查找几个元素。你也许可以从第7章回忆起，当在终端中使用记录的时候，它们必须预先定义，或者其定义已经从声明它的模块或头文件中载入了。

```
1> rd(capital, {name, country, pop}).
capital
2> ets:new(countries, [named_table, {keypos, #capital.name}]).
countries
3> ets:insert(countries, #capital{name="Budapest", country="Hungary",
                                pop=2400000}).
true
```

```

4> ets:insert(countries, #capital{name="Pretoria", country="South Africa",
                                pop=2400000}).

true
5> ets:insert(countries, #capital{name="Rome", country="Italy",
                                pop=5500000}).

true
6> ets:lookup(countries, "Pretoria").
[#capital{name = "Pretoria",country = "South Africa",
           pop = 2400000}]
7> ets:match(countries, #capital{name='$1',country='$2',_='_'}).
[["Rome","Italy"],
 ["Budapest","Hungary"],
 ["Pretoria","South Africa"]]
8> ets:match_object(countries, #capital{country="Italy",_='_'}).
[#capital{name = "Rome",country = "Italy",
           pop = 5500000}]
9> MS = ets:fun2ms(fun(#capital{pop=P, name=N}) when P < 5000000 -> N end).
[#{capital{name = '$1',country = '_',pop = '$2'},
  [{<','$2',5000000}],
  ['$1']}]
10> ets:select(countries, MS).
["Budapest","Pretoria"]

```

看看我们在命令7和命令8中是如何匹配记录的。为了保证记录字段的变化不会影响到ETS表上的操作，这些操作中并没有使用这些字段，一个格式为`#capital{name = '$1', country = "Italy", _ = '_'}`的表达式作为一个模式传递到匹配函数。构造`_ = Expression`把所有没有绑定的记录字段替换为`Expression`。在我们的例子中，我们选取`'_'`，这意味着“所有没有明确被提及的字段”和匹配通配符`'_'`匹配。

表的可视化

Erlang系统提供一个工具使得当前ETS和Mnesia表的状态变得可视化。当可视化工具表通过调用`tv:start()`启动时，将会显示当前节点和连接的节点拥有的表。启动时显示了一个表的列表，如图10-4所示。

单击其中的一个表可以打开它的可视化窗口，如图10-5所示。可视化窗口允许编辑表的内容，可以了解对表进行过的改动，同时还可以在可视化窗口中创建新表。

Dets表

Dets表为Erlang项元提供了一种有效的、基于文件的存储形式。当需要快速存取并保持一致性时，它和ETS表一起使用。Dets表有与ETS表类似的函数集合，包括检索、匹配和选择函数。但当函数调用磁盘搜索和读操作时，它与在ETS表上做同样的操作相比要慢得多。在R13发布版本中，一个Dets文件的大小不能超过2G。如果你需要保存超过2G的数据，那么，你需要将它们分割到多个Dets表中。

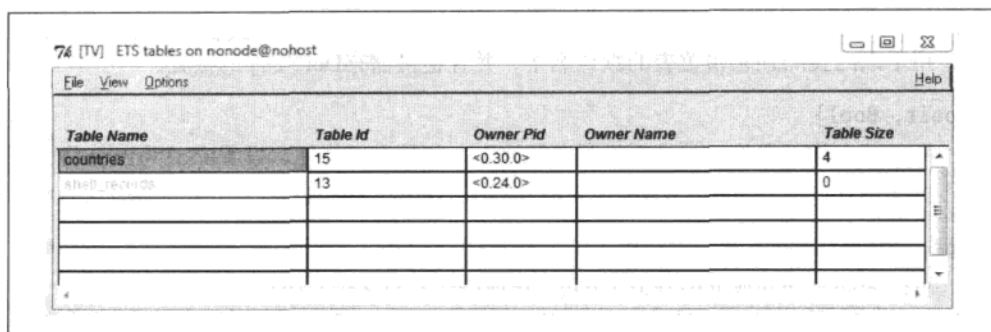


图10-4：可视化表的主窗口

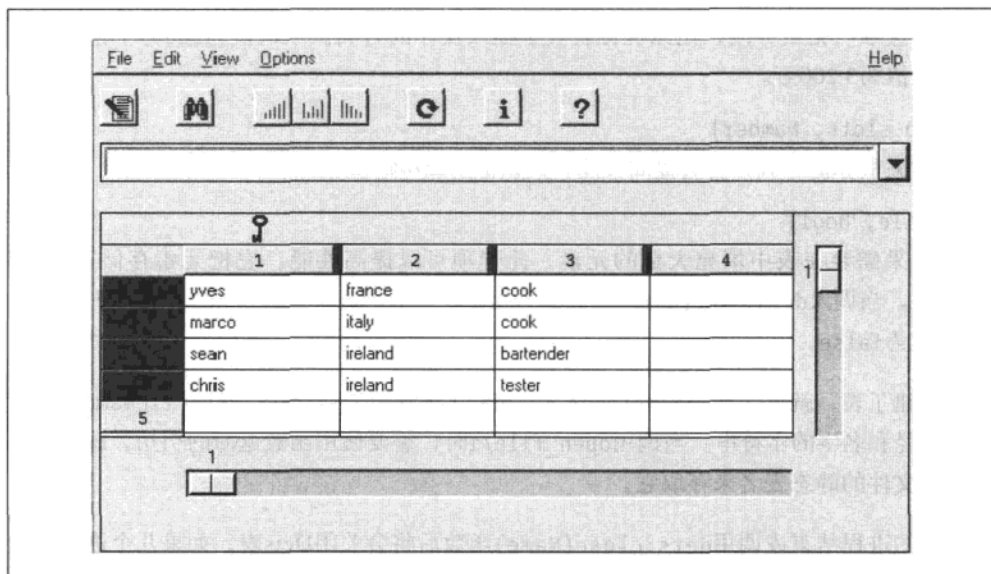


图10-5：可视化一个表

Dets表包含set、bag和duplicate bag。为了使用它们，你需要用dets:open_file(TableName, Options)打开它们，其Option参数是关键字元组的一个列表，包含下列这些：

{auto_save, Interval}

设置表周期刷新的时间间隔。刷新表意味着即使表非正常关闭，也不需要修复。间隔是以毫秒为单位的整数（默认值为180 000，即3分钟）。如果你不希望刷新文件，可以使用基元infinity。

`{file, FileName}`

用于以FileName来覆盖表的默认名字，并且提供储存Dets文件的位置。

`{repair, Bool}`

指示当表非正常关闭时是否需要进行修复。如果需要修复，设置Bool为true将引发自动修复，设置为false将返回元组{error, need_repair}。

`{type, TableType}`

可以为set、bag或duplicate bag。目前Dets表不支持有序集合。

以下是用于优化的选项：

`{max_no_slots, Number}`

此选项将把表分段，用以优化表关于插入操作的时间。默认值为200万个记录，最大值为3 200万。

`{min_no_slots, Number}`

如果估算值正确，将会提高性能。默认值为256项元。

`{ram_file, Bool}`

如果需要在表中填充大量的元素，此选项可以提高性能。它把元素存储在RAM中，当调用dets:sync(Name)函数或关闭表时，输出这些元素到文件。标签flag默认设为false。

一旦创建了表，就可以使用dets:open_file(FileName)来打开它，其中FileName是一个包含路径和名字的字符串。当调用open_file/1时，需要使用函数返回的引用，此时将无法使用文件的静态表名来存取它。

在自己的进程结束或调用dets:close(Name)函数后将会关闭Dets表。如果几个进程打开同一个表，这个表会保持打开的状态，直到所有的进程终止或者明确地关闭这个表。如果在Erlang运行时系统终止之前没有关闭表，这将导致在下次表打开时自动修复。这可能是一个很耗时的任务，而时间取决于表的大小。

在接下来的例子中，我们将创建一个表格，用于测试表的插入、选择和关闭操作：

```
1> dets:open_file(food, [{type, bag}, {file, "/Users/Francesco/food"}]).
{ok, food}
2> dets:insert(food, {italy, spaghetti}).
ok
3> dets:insert(food, {sweden, meatballs}).
ok
4> dets:lookup(food, china).
[]
5> dets:insert(food, {italy, pizza}).
ok
```

```

6> NotItalian = ets:fun2ms(fun({Loc, Food}) when Loc /= italy -> Food end).
[{'$1','$2'},[{'/','=',' $1',italy}],['$2']]
7> dets:select(food, NotItalian).
[meatballs]
8> dets:close(food).
ok
9> {ok, Ref} = dets:open_file("/Users/Francesco/food").
{ok,#Ref<0.0.0.173>}
10> dets:lookup(Ref, italy).
[{italy,spaghetti},{italy,pizza}]
11> dets:info(Ref).
[{type,bag},
 {keypos,1},
 {size,3},
 {file_size,5920},
 {filename,"/Users/Francesco/food"}]
12> dets:lookup(food, italy).
** exception error: bad argument
   in function dets:lookup/2
      called as dets:lookup(food,italy)
13> dets:info(Ref).
Undefined

```

请特别注意命令9~12的dets:open_file/1函数，其中表是通过引用来进行存取的，而不是通过名字。也请注意命令13，只有在打开表的进程崩溃（或终止）后，表才变成不可用。

注意： 当需要存储大小可预估的关键数据时，就需要使用ETS和Dets表。如果系统需要事务、查询、复制和一致性等功能，你应该使用Mnesia，这个数据库应用程序作为OTP中间件的一部分发布。它把ETS和Dets表结合Erlang分布式一起使用，提供一个关系数据模型，它支持原子性的分布式事务、检查点、备份、分段、运行时模式配置改变。更多关于Mnesia的信息，请参考第13章和Erlang发行包中的参考指南。

移动用户数据库实例

你的手机服务商需要用来管理你的账户的构架。每次发送短信时，服务器必须进行数据库检索，以确保你是一个及时付费的用户。每次需要移动终端发起一个数据会话时，也需要进行一个类似的检索过程，以确保你已订阅了数据包服务。每次当你希望使用手机的定制服务短信功能，比如说，在电视歌唱比赛中为你喜欢的选手投票，将你想要投票的艺术家的编号写入短信，当移动服务运营商收到这个信息，底层的系统需要执行用户检索，确保你的账户有权发送和接收定制服务短信功能。在这个例子中，运营商会在很短的时间内收到成千上万的短信，系统需要处理大量的突发数据，同时不能降低服务的水平。类似的，想象一下在竞选活动中，运营商尽可能快地把一个短信发给数百万用户，每个请求都需要在短信发送之前进行一个检索，以确认接收者是一位客户，这样可

以避免内部网络的费用，所以用户数据库不仅每天需要处理数百万的查询，还需要支持海量持续的突发请求。不用多说，Erlang完美地适用于此类应用。既然我们已经详细介绍了ETS和Dets表，那么让我们用它建立一个移动用户数据库，并提供可以用来查询的Erlang接口。

很明显，我们需要使用ETS表来确保常量的和快速的查询时间。由于还需要进行数据备份，因此我们将使用Dets表作为ETS表的镜像。我们提供两个接口，一个供数据库配置使用，可以进行用户的添加、更新和删除操作。在这个接口中，操作必须迅速完成，因为有可能我们需要处理上千万的用户，但他们不需要软实时性。第二个接口是为移动网络中的服务准备的，在网络中它们需要查询用户的信息。这些检索不依赖于配置接口而独立运行，这样它们就不会互相干扰，或者在有突发数据时造成阻塞。假设这些服务使用之前描述的用Erlang编写的数据库，给用户数据库提供Erlang API，那么会进一步加快请求的处理速度，这是因为不需要在不同的系统中进行数据变换。

我们把用户的数据保存在文件*usr.hrl*定义的记录类型*usr*中：

```
%%% File : usr.hrl
%%% Description : Include file for user db

-record(usr, {msisdn,          %int()
              id,              %term()
              status = enabled, %atom(), enabled | disabled
              plan,            %atom(), prepay | postpay
              services = []}). %atom(), service flag list
```

msisdn（注4）字段是指与GSM SIM卡关联的用户手机号码，我们将用整数把它存储在ETS表中，并把顶头的0丢掉。这样，一个格式为071234567的号码就可以表示为71234567。这种表示方法在配置和查询数据库时，都需要用到。

*id*域字段对应于移动运营商的内部用户ID。可以用它来管理特定的配置接口，它也许与*msisdn*不同，因为使用ETS和Dets表时，无法对第二个关键字进行索引，我们需要建立一个单独的表，包含从用户*id*到*msisdn*的映射。当启动或重新启动系统时，数据库服务器会遍历Dets表，并在内存中生成ETS和索引映射表的一些记录。

数据库后台操作

我们会通过很小的开发和测试迭代来实现我们的例子。把所有直接依赖于Dets和ETS的数据库操作在它们各自的模块中独立出来，如此操作具有好处，可以保证以后改变和操

注4：MSISDN代表移动用户综合服务数字网络号码（Mobile Subscriber Integrated Services Digital Network Number）。

作存储媒体时不影响系统的其他部分。如果有一天，你决定把用户的数据转移到Mnesia或其他数据库时，你可以用很小的代价完成这些。

我们把数据库模块称为usr_db.erl。它将操作的3个表是：

UsrRam

一个已命名的ETS表，用于存储usr记录和快速存取用户数据。

UsrIndex

一个已命名的ETS表，用于将用户id映射到msisdn。

UsrDisk

为了冗余和持久性目的，用一个Dets表镜像usrRam表。

后台服务器的第一个开发迭代步骤将包含打开和关闭这些表格的功能。当你添加更多的功能时，请记住把它们添加到usr_db模块的export语句中。

```
%%% File      : usr_db.erl
%%% Description : Database API for subscriber DB

-module(usr_db).
-include("usr.hrl").
-export([create_tables/1, close_tables/0]).

create_tables(FileName) ->
    ets:new(usrRam, [named_table, {keypos, #usr.msisdn}]),
    ets:new(usrIndex, [named_table]),
    dets:open_file(usrDisk, [{file, FileName}, {keypos, #usr.msisdn}]).

close_tables() ->
    ets:delete(usrRam),
    ets:delete(usrIndex),
    dets:close(usrDisk).
```

请特别注意这个表的属性。usrRam表需要指定关键字的位置，因为它需要储存usr记录。不这样操作，它将会自动选择第一个位置作为默认的关键字的位置，它在记录的元组表达式中映射为记录的名字。这个表的默认类型是集合，并确保每个关键字的唯一性。表是受保护的，允许所有的进程读取usr的记录，但只允许进程拥有者操作它们。usrIndex表将存储一个把usr id映射到msisdn的元组。

usrDisk文件是一个Dets表，文件名可作为选项设置，允许我们出于不同的测试目的而使用不同的文件名。因为它需要使用用户记录来镜像usrRam表，所以也需要指定它的关键字的位置：

```
1> c(usr_db).
{ok,usr_db}
2> usr_db:create_tables("UsrTabFile").
{ok,usrDisk}
```

```

3> ets:info(usrIndex).
[{memory,308},{owner,<0.29.0>},{name,usrIndex},{size,0},{node,node@nohost},
 {named_table,true},{type,set},{keypos,1},{protection,protected}]
4> ets:info(usrRam).
[{memory,308},{owner,<0.29.0>},{name,usrRam},{size,0},{node,node@nohost},
 {named_table,true},{type,set},{keypos,2},{protection,protected}]
5> dets:info(usrDisk).
[{type,set},{keypos,2},{size,0},{file_size,5432},{filename,"UsrTabFile"}]
6> usr_db:close_tables().
ok
7> dets:info(usrDisk).
undefined
8> ets:info(usrRam).
undefined
9> ets:info(usrIndex).
undefined

```

到目前为止一切顺利，一切似乎都能正常运行。现在引入两个函数，用于在数据库中插入一个新用户，在索引和ETS表中建立一个新的条目，同时在Dets表中建立一个备份。如果是第一次配置用户，我们将调用add_usr/1函数，如果用户已经存在，我们将调用update_usr/1函数，并且更新它的status、plan或services：

```

add_usr(#usr{msisdn=PhoneNo, id=CustId} = Usr) ->
    ets:insert(usrIndex, {CustId, PhoneNo}),
    update_usr(Usr).

update_usr(Usr) ->
    ets:insert(usrRam, Usr),
    dets:insert(usrDisk, Usr),
    ok.

```

研究一下前面的函数。注意，如果调用update_usr/1函数，那将覆盖已存在的表项。我们这样做是因为假设用户id不变，并且该用户已经存在。如果这些前提条件不成立，那么数据库会遭到破坏，因为usrIndex表不再把用户id映射到msisdn。在Erlang中，你应该信任你的内部接口。这些前提条件的满足是调用进程的职责。我们将会告诉你如何和何时编写这些函数。

既然我们能够插入数据，也应该能够使用msisdn或用户id进行查找。实现这个功能的函数是lookup_id/1和lookup_msisdn/1。如果使用id，那么需要调用一个本地get_index/1函数，用于把id映射到msisdn：

```

lookup_id(CustId) ->
    case get_index(CustId) of
        {ok,PhoneNo} -> lookup_msisdn(PhoneNo);
        {error, instance} -> {error, instance}
    end.

lookup_msisdn(PhoneNo) ->
    case ets:lookup(usrRam, PhoneNo) of

```

```

    [Usr] -> {ok, Usr};
    [] -> {error, instance}
end.

get_index(CustId) ->
    case ets:lookup(usrIndex, CustId) of
        [{CustId,PhoneNo}] -> {ok, PhoneNo};
        [] -> {error, instance}
    end.

```

让我们在终端里测试这些程序。为此，我们需要建立ETS表并且重新打开Dets表。因为我们已经在前面的调用里创建了Dets表，这个没有任何记录的文件应该仍然存在。所以我们将打开它而不是重新创建它。

我们还需要在终端中载入用户记录的定义，因为接下来我们将读和写这些记录。在所有3个表中插入100 000个用户，这一项操作可能会花费几秒钟，因为把Dets的条目写入文件这一操作，会对I/O造成很大负担。这些可以提供给我们充足的数据，用来保证数据库的生产效率对一个小型的移动通信运营商来说是相称的：

```

1> c(usr_db).
{ok,usr_db}
2> rr("usr.hrl").
[usr]
3> usr_db:create_tables("UsrTabFile").
{ok,usrDisk}
4> usr_db:lookup_id(1).
{error,instance}
4> Seq = lists:seq(1,100000).
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,
23,24,25,26,27,28,29|...]
5> Add = fun(Id) -> usr_db:add_usr(#usr{msisdn = 700000000 + Id,
                                id = Id,
                                plan = prepay,
                                services = [data, sms, lbs]})
                                end.
#Fun<erl_eval.6.13229925>
6> lists:foreach(Add, Seq).
ok
7> ets:info(usrRam).
[{memory,2214643}, {owner,<0.29.0>}, {name,usrRam}, {size,100000},
{node,nonode@nohost}, {named_table,true}, {type,set}, {keypos,2},
{protection,protected}]
8> {ok, UsrRec} = usr_db:lookup_msisdn(700000001).
{ok,#usr{msisdn = 700000001,id = 1,status = enabled,
        plan = prepay,
        services = [data,sms,lbs]}}
9> usr_db:update_usr(UsrRec#usr{services = [data, sms], status = disabled}).
ok
10> usr_db:lookup_msisdn(700000001).
{ok,#usr{msisdn = 700000001,id = 1,status = disabled,
        plan = prepay,
        services = [data,sms]}}

```

现在让我们来终止这个终端的进程。所有属于这个进程的表将被清除，以致在内存中储存用户的记录和索引也会丢失。我们使用create_tables/1来重建这些表：

```
11> exit(self(), kill).
** exception exit: killed
12> usr_db:lookup_msisdn(700000001).
** exception error: bad argument
    in function ets:lookup/2
       called as ets:lookup(usrRam,700000001)
    in call from usr_db:lookup_msisdn/1
13> usr_db:create_tables("UsrTabFile").
{ok,usrDisk}
14> usr_db:lookup_msisdn(700000001) .
{error,instance}
15> dets:lookup(usrDisk, 700000001).
[#usr{msisdn = 700000001,id = 1,status = disabled,
    plan = prepay,
    services = [data,sms]}]
```

这里发生了什么？我们遗漏了什么？数据库没有存储在基于RAM的表中，我们得到了返回值{error, instance}。用户信息只有在我们直接从Dets表中读取时，才可以得到。很简单：我们已经打开了Dets usrDisk，在关闭终端前可以存取所有我们已经存在的记录。我们也重建了usrIndex和usrRam表，在这个过程中，我们没有使用usr记录和从usr id到msisdn的映射的RAM副本拷贝来填充它们。我们需要一个恢复备份功能，当创建索引和RAM中的项时，遍历Dets表并使用其内容。幸运的是，我们可以调用从dets模块中输出的traverse函数：

```
restore_backup() ->
  Insert = fun(#usr{msisdn=PhoneNo, id=Id} = Usr) ->
    ets:insert(usrRam, Usr),
    ets:insert(usrIndex, {Id, PhoneNo}),
    continue
  end,
  dets:traverse(usrDisk, Insert).
```

traverse函数使用Dets表名和一个应用到每个元素的fun函数作为参数。我们传递的fun函数在usrRam和usrIndex都创建一个记录，并把用户数据库恢复到最初的状态：

```
16> c(usr_db).
{ok,usr_db}
17> usr_db:restore_backup().
[]
18> usr_db:lookup_msisdn(700000001).
{ok,#usr{msisdn = 700000001,id = 1,status = disabled,
    plan = prepay,
    services = [data,sms]}}
19> usr_db:lookup_id(1).
{ok,#usr{msisdn = 700000001,id = 1,status = disabled,
    plan = prepay,
```

```
services = [data,sms]}}
```

最后，我们需要一个有效率的方式来遍历整个数据库，清除所有终止服务、欠费或已更换服务商的用户。我们将通过用户的状态字段来识别，并设置为disabled。

我们将在usrRam表的RAM副本上调用first/1和next/2函数遍历移动用户。你是否还记得我们曾经说过，当遍历时需要使用safe_fixtable/2锁定表，因为在遍历，具有破坏性的操作会引起运行时错误，或者更糟，将导致无法预计的行为？使用safe_fixtable/2可以保证只遍历每个元素一次，而不会受到任何在遍历开始后执行的破坏性操作的影响。把遍历操作封装在catch语句中是个好主意，因为我们需要保证在发生运行时错误事件时释放表。不这样做，如果在其他地方捕获了异常，那将会导致内存泄漏。使用catch而不是try catch的原因是，我们不关心loop_delete_disabled/1函数的返回值，只是想确定它没有使进程崩溃：

```
delete_disabled() ->
    ets:safe_fixtable(usrRam, true),
    catch loop_delete_disabled(ets:first(usrRam)),
    ets:safe_fixtable(usrRam, false),
    ok.

loop_delete_disabled('$end_of_table') ->
    ok;
loop_delete_disabled(PhoneNo) ->
    case ets:lookup(usrRam, PhoneNo) of
        [usr{status=disabled, id = CustId}] ->
            delete_usr(PhoneNo, CustId);
        _ ->
            ok
    end,
    loop_delete_disabled(ets:next(usrRam, PhoneNo)).
```

随着delete_disabled/0函数的就位，我们有了所有需要的数据库后台功能。之前我们在命令9中设置一个用户的状态为disabled，所以遍历表和删除状态为disabled的项，结果应返回99 999个用户。接下来测试一下，一旦满意它的功能，我们将开始定义和实现服务器的代码：

```
20> c(usr_db).
{ok,usr_db}
21> usr_db:delete_disabled().
ok
22> ets:info(usrRam).
[{memory,2214621}, {owner,<0.182.0>}, {name,usrRam}, {size,99999},
 {node,nonode@nohost}, {named_table,true}, {type,set}, {keypos,2},
 {protection,protected}]
```

在我们加入所有函数后，usr_de模块的export语句如下所示：


```
-export([create_tables/1, close_tables/0, add_usr/1, update_usr/1, delete_usr/1,
        lookup_id/1, lookup_msisdn/1, restore_backup/0, delete_disabled/0]).
```

数据库服务器

数据库服务器输出3个函数集合。第一个以操作和维护为目的，用于启动和停止服务器；第二个作为用户服务程序的接口，以用户及其授权使用的服务来配置数据库；第三个作为服务使用的接口来向用户请求信息。

下面我们将介绍数据库服务器的接口：在函数描述中，我们使用Erlang类型标记法，更多的细节请查看第18章。举例来说，`delete_usr(CustId) -> ok | {error, timeout}`表示`delete_usr`含有一个参数（是一个客户标识符），而且它返回基元`ok`或者元组`{error, timeout}`。竖线（|）用于分隔可选的返回类型。

首先，我们看看启动和停止服务器的操作和维护函数：

`start() -> ok`

使用`usrDb`作为默认的Dets文件名启动数据服务器。它在这里只用作测试目的。

`start(FileName) -> ok`

用任何一个有效的字符串覆盖文件名`usrDb`。同时调用`start/0`和`start/1`，并且仅当Dets文件已完成遍历且其内容已写入ETS表时，返回`ok`。

`stop() -> ok | {error, Reason}`

删除ETS表并关闭Dets文件，终止数据库服务器。它返回调用`dets:close/1`的结果。

客户服务软件将使用以下函数来配置用户和它们允许使用的服务。所有改变数据库状态的函数会通过数据库服务器同步执行，用以保证数据的一致性：

`add_usr(PhoneNum, CustId, Plan) -> ok | {error, timeout}`

用于配置用户。`PhoneNum`和`CustId`是整数型，`Plan`是`prepay`和`postpay`中的一个基元。

`delete_usr(CustId) -> ok | {error, timeout}`

用于删除一个指定用户和与其相关的所有索引。

`set_service(CustId, Service, Flag) -> ok | {error, instance | timeout}`

用于对某个特定的用户添加和删除一个Service。Service是一个基元的列表，包括（并不受限于）`data`、`lbs`和`sms`。基元`data`确保用户制定了一个数据计划，`sms`允许用户发送和接收定制服务短信，`lbs`允许第三方对某个特定用户的位置进行查询。Flag代表基元`true`或`false`，取决于是否添加还是删除Service。

```
set_status(CustId, Status) -> ok | {error, instance | timeout}
```

用于设置一个特定CusId的状态为enabled或者disabled。

```
delete_disabled()-> ok | {error, timeout}
```

遍历表并删除所有状态为disabled的用户。

```
lookup_id(CustId) -> {ok, #usr{}} | {error, instance}
```

基于CustId查找一个用户，并返回usr类型的记录，因为它已在包含文件usr.hrl中有定义。这个函数没有改变用户数据，而且结果是它在调用进程的作用域内执行，而不是服务器上执行。

最后，我们有了服务应用程序使用的函数。因为所需数据通过受保护的ETS可以获取，所以这两个函数都在调用进程的作用域里执行。在客户端和进程之间没有消息交换，避免了在大负荷量情况下的瓶颈和减少了总体的响应时间。因为大多数的服务只会存取msisdn而不是usr id，所以把msisdn作为关键字：

```
lookup_msisdn(PhoneNo) -> {ok, #usr{}} | {error, instance}
```

基于它的PhoneNo查找用户并返回一个usr类型的记录。这个函数的用途主要是服务应用程序记录中的数据，以确认用户是否有权使用某项服务。比如说，在被允许发送和接收一个定制服务短信之前，可能要向一个预付费用户收费以确保他有良好的信用。或者我们只是想要发送一条带有经度和纬度的短信，只要用户设置了sms和lib标记。

```
service_flag(PhoneNo, Service) -> true | false | {error, instance | disabled}
```

检测某个用户是否存在或是否处于enable状态。如果是，它会遍历服务表来确定用户是否允许使用一个特定请求中的Service。这是lookup_msisdn/1调用的一个变化，其逻辑测试在usr模块中完成。

注意我们是如何提取出了代码中的服务器循环，我们通过模式匹配每一次调用的第一个参数，在单独的函数中处理所有消息。所有的客户端/服务器之间的通信也同样通过函数call/1和reply/2进行提取。启动服务器是一个同步操作，并且只有Dets表被载入并在usrIndex与usrRam表中完成镜像后，函数start/0和start/1的调用才返回。超时设置为30秒。这个值可以进行微调，取决于服务器的负荷和数据库的大小。实际的超时值通常是系统压力测试时确定的：

```
%% File      : usr.erl
%% Description : API and server code for cell user db

-module(usr).
-export([start/0, start/1, stop/0, init/2]).
-export([add_usr/3, delete_usr/1, set_service/3, set_status/2,
        delete_disabled/0, lookup_id/1]).
```

```

-export([lookup_msisdn/1, service_flag/2]).

-include("usr.hrl").
-define(TIMEOUT, 30000).

%% Exported Client Functions
%% Operation & Maintenance API

start() ->
    start("usrDb").

start(FileName) ->
    register(?MODULE, spawn(?MODULE, init, [FileName, self()])),
    receive started-> ok after ?TIMEOUT -> {error, starting} end.

stop() ->
    call(stop).

%% Customer Service API

add_usr(PhoneNum, CustId, Plan) when Plan==prepay; Plan==postpay ->
    call({add_usr, PhoneNum, CustId, Plan}).

delete_usr(CustId) ->
    call({delete_usr, CustId}).

set_service(CustId, Service, Flag) when Flag==true; Flag==false ->
    call({set_service, CustId, Service, Flag}).

set_status(CustId, Status) when Status==enabled; Status==disabled->
    call({set_status, CustId, Status}).

delete_disabled() ->
    call(delete_disabled).

lookup_id(CustId) ->
    usr_db:lookup_id(CustId).

%% Service API

lookup_msisdn(PhoneNo) ->
    usr_db:lookup_msisdn(PhoneNo).

service_flag(PhoneNo, Service) ->
    case usr_db:lookup_msisdn(PhoneNo) of
        {ok, #usr{services=Services, status=enabled}} ->
            lists:member(Service, Services);
        {ok, #usr{status=disabled}} ->
            {error, disabled};
        {error, Reason} ->
            {error, Reason}
    end.

%% Messaging Functions

call(Request) ->
    Ref = make_ref(),
    ?MODULE! {request, {self(), Ref}, Request},
    receive

```

```

{reply, Ref, Reply} -> Reply
after
?TIMEOUT -> {error, timeout}
end.

reply({From, Ref}, Reply) ->
    From ! {reply, Ref, Reply}.

%% Internal Server Functions

init(FileName, Pid) ->
    usr_db:create_tables(FileName),
    usr_db:restore_backup(),
    Pid ! started,
    loop().

loop() ->
    receive
        {request, From, stop} ->
            reply(From, usr_db:close_tables());
        {request, From, Request} ->
            Reply = request(Request),
            reply(From, Reply),
            loop()
    end.

%% Handling Client Requests

request({add_usr, PhoneNo, CustId, Plan}) ->
    usr_db:add_usr(#usr{msisdn=PhoneNo,
                        id=CustId,
                        plan=Plan});

request({delete_usr, CustId}) ->
    usr_db:delete_usr(CustId);

request({set_service, CustId, Service, Flag}) ->
    case usr_db:lookup_id(CustId) of
        {ok, Usrc} ->
            Services = lists:delete(Service, Usrc#usr.services),
            NewServices = case Flag of
                true -> [Service|Services];
                false -> Services
            end,
            usr_db:update_usr(Usrc#usr{services=NewServices});
        {error, instance} ->
            {error, instance}
    end;

request({set_status, CustId, Status}) ->
    case usr_db:lookup_id(CustId) of
        {ok, Usrc} ->
            usr_db:update_usr(Usrc#usr{status=Status});
        {error, instance} ->
            {error, instance}
    end;
end;

```

```
request(delete_disabled) ->
  usr_db:delete_disabled().
```

在终端测试usr服务器代码，结果可能如下所示：

```
1> c(usr).
{ok,usr}
2> rr("usr.hrl").
[usr]
3> usr:start().
ok
4> usr:add_usr(700000000, 0, prepay).
ok
5> usr:set_service(0, data, true).
ok
6> usr:lookup_id(0).
{ok,#usr{msisdn = 700000000,id = 0,status = enabled,
        plan = prepay,
        services = [data]}}
7> usr:set_status(0, disabled).
ok
8> usr:service_flag(700000000,lbs).
{error,disabled}
```

当检查数据库服务器代码的时候，试着通过相反的测试用例负检验和破坏性的数据来让它崩溃。使用usr模块中提供的接口，你是否能够通过不更新usrIndex表改变用户ID来破坏数据？看看你能否发现代码中的缺陷，它可能导致运行时错误并造成服务器终止。

还有许多的方法可以用纯函数的方式建立更有效率的集合，在Chris Okasaki的《Purely Functional Data Structures》中有更多的描述（由Cambridge University Press出版）。

练习

练习10-1：格式良好的打印

这个练习要求你完成在本章前面部分出现的索引例子中的prettyEntry的定义。

在定义prettyEntry时，你可能发现定义如下函数会有用途：

accumulate/1

这个函数应该输入一个降序排列行号的列表，并且生成一个包含范围和去掉重复行号的列表。例如：

```
accumulate([7,6,6,5,3,3,1,1]) = [{1},{3},{5,7}]
```

prettyList/1

这个函数将打印`accumulate`的输出，如把列表`[{1},{3},{5,7}]` 输出为
`1,3,5-7.`。

pad/2

用数字`N`和字符串`word`作为参数调用这个函数，将返回长度为`N`的字符串，如果长度小于`N`，右边以空格补齐（假设`word`不会长于`N`）。

练习10-2：重新索引

修改索引程序，使得ETS表是这样的一个有序集，它以`Word`元组`{Word, LNs}`中`Word`字段为关键字，其中的`LNs`是一个包含`Word`所出现的行号的列表。

练习10-3：ETS表的系统日志应用

一个ETS表可以用于记录一个通信系统中的日志，通过它来进行分析和报告错误。以一个简单的消息系统为例，其中的每个消息都期望收到一个回执。内置函数`now()`消息可以由发送时间来标识。假设消息以它们的发送时间作为时间戳，这样它们可以与其回复进行匹配。

设计一个或多个ETS表来包含消息和回执，并且使用这些表编写一个程序来完成下面的功能：

- 检查是否每个消息都收到了唯一的一个回执。
- 通过时间单位为秒变化的发送窗口来监听收到回执的平均时间。



Erlang中的分布式编程

编写容错系统，你至少需要两台计算机（注1），而且需要它们分布在你的程序中。分布式系统属于现代计算技术的核心内容。在服务器编程中，这是例外，而不是用于衡量单个计算机对不同难度任务的表现的准则；相反，一定数量的计算机共同提供一个健壮、高效和可扩展的平台，并在此基础上建立应用程序。

Erlang分布式是语言内建的，从用户角度看，它是完全透明的。通过进程ID存取进程，不管是引用在本地计算机上的一个进程，还是在世界其他地方的系统上的一个进程，操作都一样。在本章中，我们会介绍分布式系统背后的理论部分，以及如何把它应用在基于Erlang的系统中。

Erlang中的分布式系统

分布式系统的实质是通过一些用网络连接起来的计算机、处理器或者内核以一种透明的方式提供某种服务。服务可以是特定的，比如分布式文件系统或数据库提供的存储功能；也可以是通用的，比如分布式操作系统通过计算机网络提供通常操作系统所具有的功能。分布式可以看做是连接紧密的并行处理器，或者更明确地说是松散耦合的电子科学系统网格（grids of e-science systems）。Erlang提供分布式程序设计能力，使得Erlang系统可以在跨越Erlang节点联网运行。

安装一个Ejabberd，它是一个Erlang开源的基于Jabber的即时消息（IM）服务器。它实现分布于两个或更多Erlang节点组成的集群。这些节点位于同一或不同的机器上，通过信息和事件负载均衡来互相支援。如果其中的一个节点因为软件或硬件错误，或简单的因为内存泄漏而终止，其他的节点将接管其流量，并且对最终用户隐藏这个错误。最坏的情况是，最终用户在套接字重新连接新节点时，可以意识到他们经历了一个网络小故障，但他们所察觉到的只是其他用户的退出和登录。

注1：根据Joe Armstrong的说法是至少两台，但是，如果你要问Leslie Lamport的话就是至少3台。

Erlang Web框架是一个基于Erlang的Web应用的开源应用程序，通过使用分布式获取可伸缩性和可靠性。一个典型的集群由前端和后端节点组成。前端节点包含Web服务器（在Erlang节点中运行）、一个高速缓存层和为内层处理请求的XML分解。它还包含处理XHTML动态生成的功能。两个或更多的后端节点包含数据库和所有生成动态内容所需的粘合剂和逻辑性。真正的负载在前端，因为它处理套接字连接和大部分的解析。为了扩展系统，你所需要做的只是添加硬件和前端节点，只有在必需的时候才增加对后端的支持。如果任何节点失效了，负载均衡器都会自动把流量转接到仍然存活的节点上。

如果你想要通过在一些节点上分布功能来扩展一个Erlang系统，其中一件你需要考虑的事情是如何在节点上均衡负载。你可以通过随机或者轮询的方式来分派任务，这两种方法在任务规模较小时都可行。否则，你需要估算一下被分布任务的大小。最后，你可以使用主从模式，在请求时分派任务。

无论你使用哪种方式，监控你的系统行为和调整你的分布式策略——或以实时方式或通过代码升级——来响应系统变化的需求是至关重要的。

另一个分布式系统的例子是Erlang旗舰产品中的一个——AXD301 ATM切换器。最小的Erlang集群由两个节点组成，一个称为呼叫建立节点，另一称为操作维护（O&M）节点。如果O&M节点失效，会产生一个故障切换，并且O&M应用程序会在呼叫建立节点重启。当O&M节点恢复正常，通过自动恢复和手动干涉，会产生一次接管，O&M应用程序会重新转移回原节点继续运行。

呼叫建立节点的失效属于紧急情况，因为它们会影响ATM通信。如果一个呼叫建立调用设置节点失效，那么一个故障切换会把所有呼叫建立应用程序转移到O&M节点。分布式数据保证，任何呼叫的建立是在故障切换之前初始化的，呼救都不会丢失。在O&M节点上运行的新的呼叫建立应用程序会接管它们。当原始节点被重启，为保证不会扰乱通信并且没有呼叫建立请求丢失，一个O&M应用程序的接管会把O&M的功能迁移到新的已重启的节点上，并且呼叫建立功能保留在以前的O&M节点中。

并发性是所有分布式系统的核心，因为运算和通信会并行地运行在处理器和网络所组成的系统中。分布式系统主要的挑战是在错误事件发生时系统的健壮性。这个领域的先锋人物Leslie Lamport总结的重要结论是：

“分布式系统是一个会因为另一个你甚至不知道其存在的计算机的错误，而导致你的计算机无法使用的系统。”

但是如果你应用正确，构建分布式系统有很多好处：

- 可以提供按需增长的性能。一个典型的例子是Web服务器：如果你计划发布一个软

件的新的部分版本，或者计划用实时流媒体传送足球比赛视频，那么可以把这些服务器分布到许多机器上，分布式使之成为可能并且没有问题。

这种性能可以通过服务的复制来保障——在这种情况下是一个Web服务器——这通常会在分布式系统架构中出现。

- 复制也可以提供容错性：如果其中一个被复制的Web服务器失效或因为某些原因无法使用，HTTP请求仍可以由其他服务器处理，尽管是以一个较低的速率实现。这种容错性使得系统更具健壮性、更可信赖。
- 分布式允许透明地存取远程资源，并且以此为基础，通过把不同的系统联合为一个整体来对全体用户提供服务。现代电子商务系统提供这种方法集合，例如Amazon.com网站。
- 最后，分布式系统架构使系统具有可扩展性，并且通过远程存取使其他服务变为可能。

电信系统包含所有以上这些内容。但若以更开阔的视角观察，并不只有这些。贸易系统、零售银行业务、航空和铁路运输调度和Web服务，这些高度相互影响、关键任务的系统只是从Erlang分布式中获益的领域的一部分。

Erlang中的分布式计算：基础

一个Erlang节点是已命名的正在运行的Erlang运行时系统。多个节点可以运行在同一主机上，但是也可以在不同主机上，如图11-1所示，三个节点运行在各自子网中的主机STC和主机FCC上。

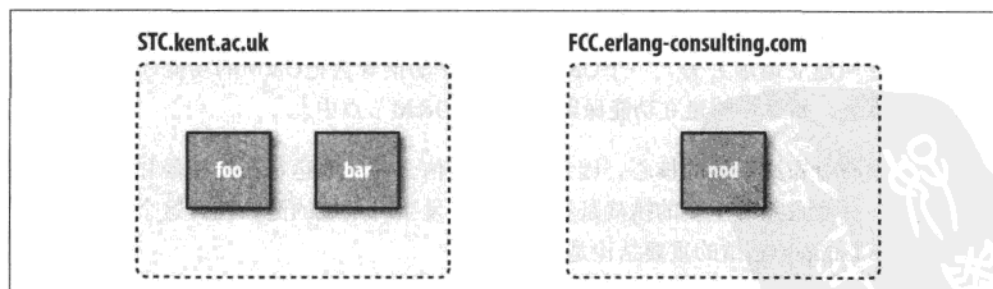


图11-1：三个节点运行在两个主机上

作为第一个例子，我们来看看两个节点运行在同一机器——STC上的情况，如图11-2所示。为运行一个节点，`erl`命令需要`sname`标签（也可以使用`name`标签，我们会很快讨论到这一点）。举例来说：

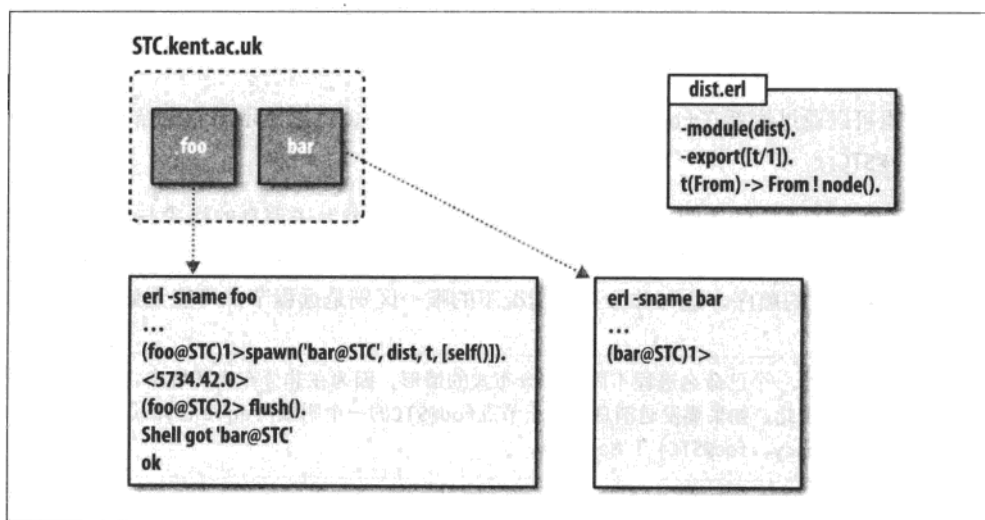


图11-2：第一个例子：两个节点在同一主机上

```
erl -sname foo
Erlang (BEAM) emulator version 5.6.3 [source] [64-bit] [smp:2]
[async-threads:0] [kernel-poll:false]

Eshell V5.6.3 (abort with ^G)
(foo@STC)1>
```

注意Erlang终端中的提示信息显示了节点名和主机名foo@STC。这被称为节点的（唯一）标识符。

一个类似的命令erl-sname bar会在STC系统上建立第二个节点。

为了理解下面会发生什么，你需要查看一下dist.erl模块。它包含函数t：

```
t(From) -> From ! node().
```

这个函数把进程标识符From作为一个参数，并且它的单一动作是发送一个消息到这个标识符代表的进程。这个消息是调用node()的结果，该函数返回该调用所在的节点的标识符。

下面，我们看看用户在foo@STC中的输入。这里使用了spawn/4函数，它的第一个参数spawn应该发生的节点。其余的类似于spawn/3：模块、方程和初始化参数。其效果如下所示：

1. 进程在节点bar@STC上生成，并且开始执行函数t，其他以在foo@STC运行的终端的进程标识符作为参数。

2. 这样的效果是发送node()函数的值（即当前节点的标识符，在此处是bar@STC）到进程标识符，函数t然后终止。
3. 这里可以通过刷新在foo@STC余留的消息进行测试，这里显示出它已经发送标识符bar@STC。

这个例子显示了通信的透明性。在两个节点之间通过命令发送消息的格式与在非分布式的情况下是完全一样的：Pid!Message。此外，消息从一个节点到另一个节点的传输保持与发送时相同的顺序。这与非分布式情况下的唯一区别是远程节点可能无效。

注意：发送消息给一个已命名进程不同于非分布式的情形，因为在非分布式环境中，命名是针对本地的。因此，如果要发送消息给位于节点foo@STC的一个叫做frequency的进程，需要采用{frequency, foo@STC} ! Message。

节点名和可见性

就像我们前面所说的，一个节点是一个Erlang可执行系统，如果一个节点可以与其他节点通信，它被称为存活节点；另一种说法是这个节点是可命名的，所以可以参与通信。

函数erlang:is_alive()会测试本地运行时系统是否为存活状态。在下面的例子中，使用net_kernel模块函数可以改变一个正在运行的运行时系统的存活状态，也可以使用node/0内置函数找出当前节点的名字。尝试下面的代码：

```
1> erlang:is_alive().
false
2> net_kernel:start([foo]).
{ok,<0.33.0>}
(foo@STC.local)3> erlang:is_alive().
true
(foo@STC.local)4> node().
'foo@STC.local'
(foo@STC.local)5> net_kernel:stop().
ok
6> erlang:is_alive().
false
```

任何存活的节点都要命名：在那台主机上，这些名称必须是唯一的，但是在不同的主机上，名字可以重复。这种名字/主机对称称为节点的标识符，用来唯一识别网络中的节点。

名字有两种形式。你已经见过了第一种形式，第二种是新形式：

短名字：erl -sname foo ...

sname会在局域网中命名一个主机，并以name@host的形式（比如foo@STC）给出。

长名字: `erl -name foo ...`

这个名字给出主机的完整IP地址: 它可以是`foo@192.168.1.11`或者(在局域网中)`foo@STC.local`。就像你在前一个例子中的命令4看到的, 如果使用`net_kernel:start`函数来启动一个分布式节点, 这个节点会被赋予一个长名字。

注意长名字节点只能与其他具有长名字的节点通信, 短名字节点只能与其他具有短名字的节点通信。

注意: 为了使用`server.kent.ac.uk`形式的主机名, 而不是原始IP地址如`foo@192.168.1.11`, 需要把主机名解析为IP地址。一个域名服务器(DNS)可以做到这个, 但若不能访问DNS服务器, 名字可以通过使用包含在主机文件的信息而在本地解析。具体完成的步骤取决于不同的平台。参考你的特定操作系统文档来了解它具体是怎样工作的。

通信和安全

对于两个节点之间的通信, 不仅需要两个节点同为存活节点, 还需要在被称为安全标识信的基元中包含共享信息。每个节点在任意时刻只有一个cookie, 共享同一值的节点可以进行通信。

每个节点在启动时可以指定一个确切的cookie, 比如:

```
erl -sname foo -setcookie blah
```

如果启动时没有预设值, Erlang运行时系统会选择存储在`.erlang.cookie`文件中的一个值。如果该文件不存在, 它会在用户账户的主目录中创建一个。一个随机产生的安全cookie会存储在那里。这样, 用同一用户账户创建的节点在默认情况下共享同一cookie。如果你已经在分布式Erlang做过试验而没有设置cookie, 查看一下`erlang.cookie`文件。你可以用你需要的值来编辑它。

cookie为了演示应用中的安全信标和分布式, 通过在同一网络中的分离主机上运行的节点, 我们重复使用了图11-2中的例子, 如图11-3所示。

注意图11-3中的两个节点如何明确采用同一cookie值启动。这个值会覆盖在其他主机的`erlang.cookie`文件中包含的任何值。

分布Erlang代码: 一个警告

你第一次尝试分布Erlang代码时也许会失败: 为什么呢? 以下的调用:

```
(foo@STC)1> spawn('bar@FCC', dist, t, [self()]).
```

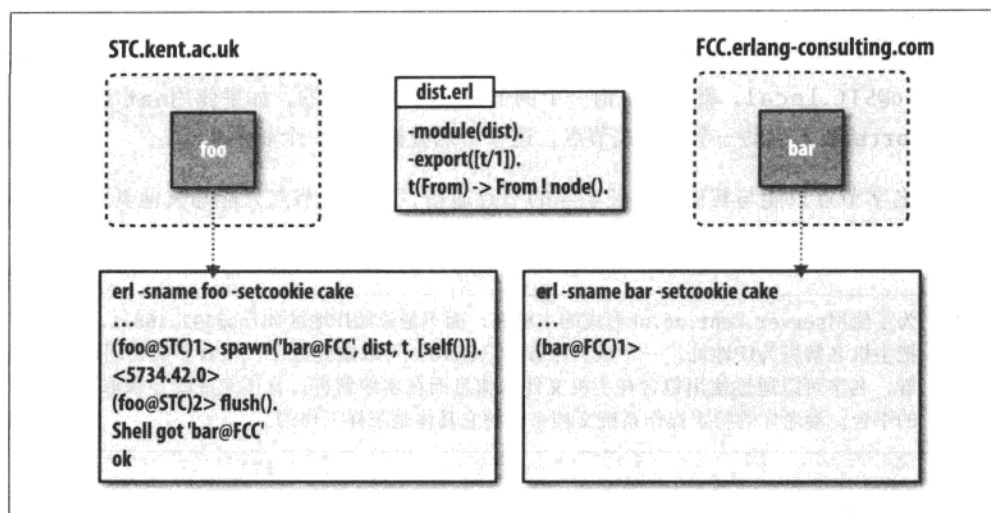


图11-3：在不同主机上的节点的例子

是在主机STC的foo节点上，但是产生的代码试运行在FCC上。模块dist.erl不会在FCC上执行，除非它可以自动地从STC传送到FCC。

然而此次调用仍会失败，因为你需要在主机FCC上有一个代码的已编译版本，它包含远程Erlang节点上的代码搜索路径。

Erlang分布式和安全

当你的计算机上的一个分布式节点通过共享cookie与另一个远程节点建立连接，远程节点的拥有者同样拥有获得和你本地Erlang节点运行所用的账号相同的用户访问权限。

在一个封闭的电信系统或在防火墙后运行的银行系统中，这是可以接受的，但是如果你用私人账户运行这个节点，任何连接到它的人都可以阅读和删除文件，执行命令并且劫持你的机器。然而调用以下函数：

```
spawn(YourNode, os, cmd, ["rm -rf *"])
```

就会有趣地发现，你不能享受该调用带来的和平和宁静的迹象了（注2）。

所以，你应该永远不公开你的Erlang节点名和cookie给任何人，除非你已经调整了网络核心来应对严重的安全问题，或者你明确地相信这个人不会做任何恶意的事情。

注2： 除非你的账户需要一些严格的保管措施。

通信和消息

最基本的通信是一个节点测试其是否可以与其他节点建立连接，也就是非正式地说 *pinging* 其他节点（见图11-4）。

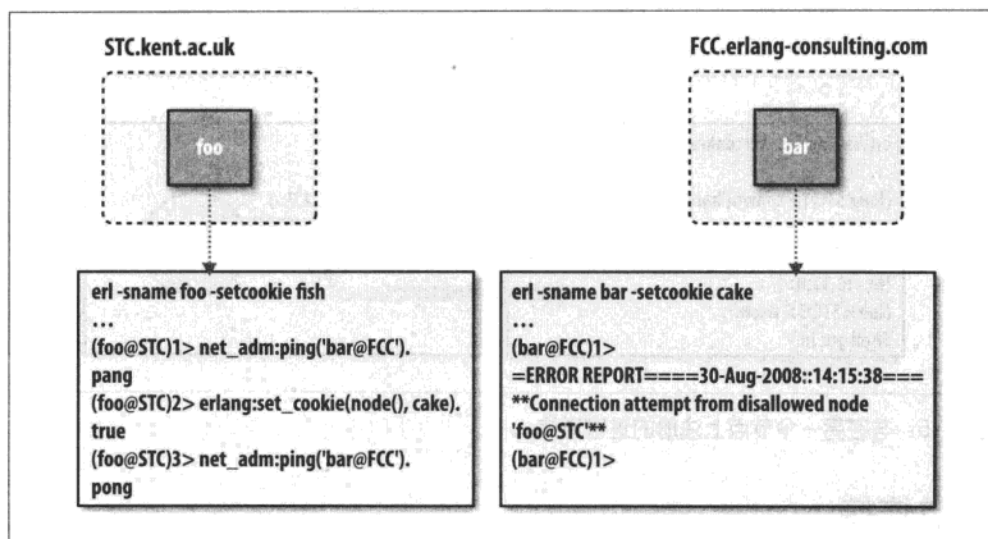


图11-4: ping 一个节点

图11-4中两个节点采用不同的值初始化cookie，这将阻止它们互相通信。foo节点试图通过net_adm:ping/1:进行通信，pong回复表示通信失败。这个被ping的节点也会给出一个错误报告来给连接请求发信号，通过警告的方式表示一个潜在的安全问题发生了。在改变foo节点的cookie值为cake后，通过pong结果指示ping成功了；这样一个成功的尝试不会通知bar节点。

下面，我们将看一个例子，其中调用函数spawn/4在远程节点注册一个进程，然后与它通信，如图11-5所示。

第一个在foo节点中的命令生成dist:s/0进程。结果是在server名下注册循环loop/0。该循环的作用是重复接受采用格式为{M, Pid}的消息并且返回消息M到Pid。在foo的第二个命令中，消息hi与foo运行的终端的Pid一同被发送。这会被server接收，并且hi消息会传回到foo，作为flush()函数的结果你会在foo的收信箱中看到。

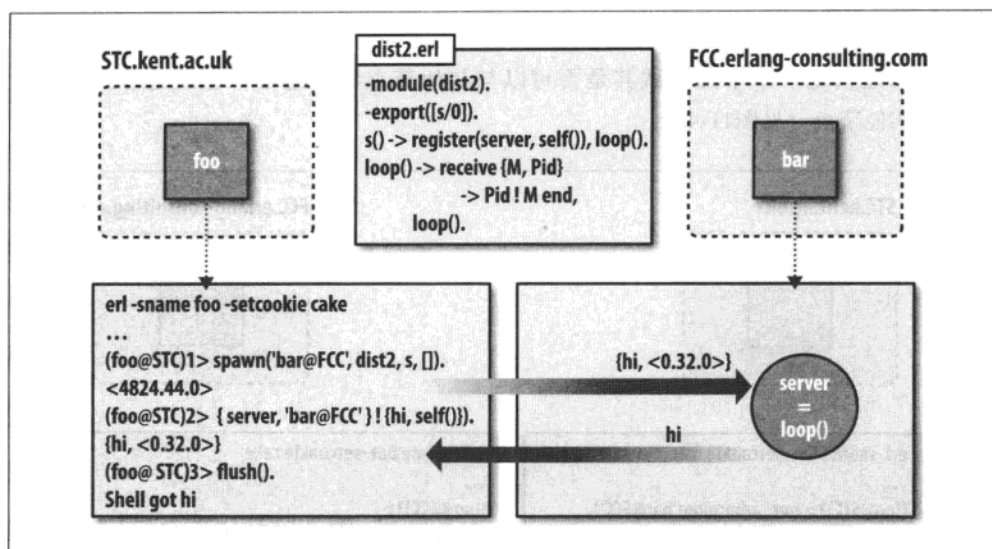


图11-5：与在另一个节点上注册的进程通信

节点连接

只要分布式Erlang节点共享相同的cookie信息，它们之间就可以进行通信，但是你没有看到这些节点之间的连接是如何建立的。这是因为Erlang运行时系统在一个节点第一次被引用时自动建立连接。这可以通过调用`net_adm:ping/1`或通过发送一个消息到它的一个注册过的进程上来做到。连接在一起的节点，信息默认是共享的。所以，如果A知道B，B知道C，那么A也能够找到C。

每个节点在任意一个时刻都有一个cookie。在分布式Erlang中安全基于共享cookie信息：如果cookie值改变会发生什么？下面的例子展示了这一点：

```

(foo@STC)1> net_adm:ping('bar@STC').
pong
(foo@STC)2> erlang:set_cookie(node(),cake).
true
(foo@STC)3> net_adm:ping('bar@STC').
pong
(foo@STC)4> erlang:set_cookie(node(),fish).
true
(foo@STC)5> net_adm:ping('bar@STC').
pong

```

节点foo和bar开始有不同的cookie值，因此对第一个命令进行否定回复。第二个命令中，foo的cookie设置为bar的值，所以在第三个命令中由于一个连接已建立，ping成功。

功了。在命令4中，cookie被改回初始值，但命令5显示尽管两个节点现在有了不同的cookie值，但两个节点仍保持连接。

连接的“包含”模型也许并不是所需要的，而且使用net_kernel模块中的函数可以手动控制连接，也可以使用带有-connect_all false标签的erl命令防止节点全局地相互连接。

每个节点的net_kernel进程协调分布式Erlang节点的操作。内置函数如spawn/4被net_kernel进程转换为消息发送到远程节点的net_kernel上。net_kernel进程还处理cookie的认证。简单地说，这是因为net_kernel只是一个Erlang进程，用户可以修改它来提供其他行为，例如改变认证方案或者不允许进程从其他的节点产生进程。

甚至对于最无安全意识的读者也可能意识到，仅仅基于安全cookie的安全机制并不可靠。由于电信集群趋向于在防火墙后运行，在它的分布式模型中提高安全性从来就不是问题。早期的cookie甚至不经加密就通过网络发送。

考虑一下分布式Erlang底层的安全机制，你怎样在Erlang中创建一个安全的分布式系统？这个问题有两个答案：

- 如果为了可伸缩性和健壮性来建立分布式系统，你很可能工作在一个封闭且安全的网络环境中。在这种情况下，Erlang分布式模型通过一种透明且有效的方式直接支持你的需求。
- 如果你想建立一个地理上的分布式系统，最好是使用已有的安全机制来保证节点之间的通信，如在TCP/IP上采用SSL加密。Erlang分布式的库支持许多协议，包含安全协议如SSL。我们将在第15章中介绍在Erlang中使用TCP/IP通信的基础知识。

给定进程所有的行为和你要求达到的安全级别，你可以通过编写自己的net_kernel进程来增强安全性。

隐藏节点

当节点之间建立了连接后，它们为创建一个完全的网状网络开始互相监视。如果你有4个Erlang节点，全网格化的网络会在这些节点中建立6个TCP/IP连接。使用公式 $N * (N - 1) / 2$ ，我们可以快速计算出10个节点需要45个连接。这不仅会增加在这些节点之间发送监控消息的开销，而且我们也许一开始不希望所有的节点相互连接。节点数目超过100或更多会使情况变得更糟，特别是在其中许多节点与其他节点并没有关联的时候。

解决方法是使用隐藏节点和明确地只建立必要的连接。通过如下方法启动Erlang你可以启动一个隐藏节点：

```
erl -sname foo -hidden
```


一旦开始，可以调用`net_kernel:connect(NodeName)`建立与其他节点的连接。使用`nodes/0`内置函数不会返回任何的隐藏节点。为了查看它们，你需要带一个基元参数调用函数`nodes/1`，例如调用`nodes(hidden)`会列出你所希望与之建立连接的隐藏节点，并且`nodes(connected)`会给出所有节点的一个合计列表，包含隐藏和非隐藏节点。

在下面的例子中，我们启动了三个节点，分别命名为`alpha`、`beta`和`gamma`，其中`gamma`是一个隐藏节点。启动后，我们从`alpha@STC`连接到`beta@STC`和`gamma@STC`。一旦三个节点开启后，结果如下：

```
(alpha@STC)1> net_kernel:connect('beta@STC').
true
(alpha@STC)2> net_kernel:connect('gamma@STC').
true
(alpha@STC)3> nodes().
['beta@STC']
(alpha@STC)4> nodes(hidden).
['gamma@STC']
(alpha@STC)5> nodes(connected).
['beta@STC', 'gamma@STC']
```

一旦我们在`alpha`中执行了所有的命令，我们可以检查一下检验节点连接是如何拓展到`beta`和`gamma`的。让我们来看一看：

```
UNIXSHELL> erl -sname beta
Erlang (BEAM) emulator version 5.5
Eshell V5.5 (abort with ^G)
(beta@STC)1> nodes().
['alpha@STC']
(beta@STC)2> nodes(connected).
['beta@STC']
```

就像你在上面代码中看到的，隐藏节点`gamma`并没有出现。对`gamma`自己来说，没有可见节点，除非你用`hidden`或`connected`的标签来查看它们。如果这样做，唯一可见的节点是`alpha`，因为当连接建立时，`beta`上的信息没有被传播：

```
UNIXSHELL> erl -sname gamma -hidden
Erlang (BEAM) emulator version 5.5
Eshell V5.5 (abort with ^G)
(gamma@STC)1> nodes().
[]
(gamma@STC)2> nodes(hidden).
['alpha@STC']
```

注意：隐藏节点可以用作网关来把小的分布式集群连接到一起。这是一种允许网格中成百的节点松散连接在一起的技术，而且不会为了互相监视使它们过载。隐藏节点通常也可用于操作和维护，也可以作为跟踪节点。跟踪节点不需要承担任何网络通信，对于系统操作也不是必需的，但它必须能够获得信息并且与远程节点交互。

远程程序调用

在分布式计算中一个典型的结构是远程程序调用（Remote Procedure Call, RPC），其中一个对本地的程序调用被替换为在远程节点上运行的相同程序的调用。在Erlang上实现RPC很简单，而且Erlang的实现避免了许多在其他语言中实现RPC的缺陷。

在基本的Erlang RPC实现中，如下的（本地）函数调用：

```
Val = fac(N)
```

被一个消息的发送和接收代替，如这里的图11-6所示：

```
remote_call(Message, Node) ->
{facserver, Node} ! {self(), Message},
receive
  {ok, Res} ->
    Res
end.
```

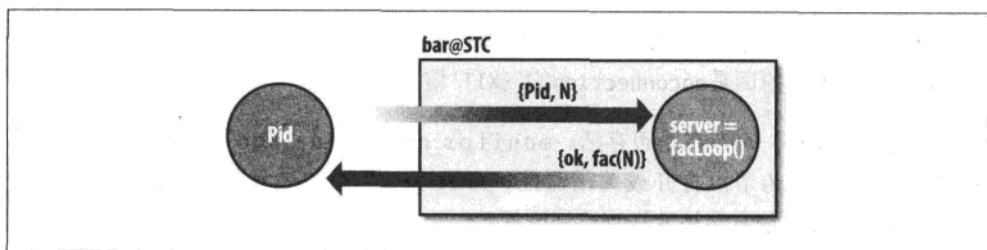


图11-6：远程程序调用

在下列代码中，facserver进程运行在bar@STC节点上并且运作在facLoop/0循环函数中：

```
server() ->
  register(facserver, self()),
  facLoop().

facLoop() ->
  receive
    {Pid, N} ->
      Pid ! {ok, fac(N)}
  end,
  facLoop().
```

本地调用和远程调用的主要不同点是事实上的远程节点可能会失效。可以通过几种不同的方式来解决这个问题。比如说，你可以在客户端代码中添加一个超时：

```
remote_call(Message, Node) ->
{facserver, Node} ! {self(), Message},
receive
```

```

{ok, Res} ->
    Res
after 1000 ->
    {error, timeout}
end.

```

如果在1秒内没有收到回复，那么就返回元组{error, timeout}。你应该小心使用超时，因为在超时后仍可能收到消息并存储在进程邮箱中。远程服务器也许非常繁忙或网络非常拥堵。如果你不刷新消息，下一次remove_call/2会被激活并且发送一个新的请求到阶乘服务器，而最终你取出的是队列中的第一个消息，而队列可能包含以前调用的响应。

或者，你可以链接到服务器进程，如果失败，客户端进程也同样失效。你可以使用spawn_link/4而不是spawn/4来启动服务器进程来做到这点：

```

setup() ->
    process_flag(trap_exit, true),
    spawn_link('bar@STC', myrpc, server, []).

```

如果远程进程终止，你会接收到通常的'EXIT'信号。如果两个节点之间的网络连接中断，网络内核发送原因是noconnection的'EXIT'信号。

最终，还可以监控一个节点是否存活。monitor_node(Node, Bool)内置函数会通过布尔型标签Bool来为节点打开或关闭监控，如在以下代码中展示的。当激活监控时，{nodedown, Node}消息会被发送到监视进程：

```

remote_call(Message, Node) ->
    monitor_node(Node, true),
    {facserver, Node} ! {self(), Message},
    receive
        {ok, Res} ->
            monitor_node(Node, false),
            Res;
        {nodedown, Node} ->
            {error, node_down}
    end.

```

当完成某个节点的监控之后不要忘记取消监控，因为在每次调用内置函数monitor_node(Node, true)后都会产生一条nodedown消息。

rpc模块

rpc库提供的服务的实现与远程过程调用相似，同时也有助于广播和对RPC调用的平行评估。最常用的函数是：

```
rpc:call(Node, Module, Function, Arguments)
```

此函数会在远程节点上执行。模块必须在远程节点的代码搜索路径中，并且节点必须相互连接的或者共享cookie。结果是调用的返回值或者错误{badrpc, Reason}。

如果你的应用程序将采用分布式形式，那么请花点时间仔细阅读rpc模块的手册并且熟悉它。你会找到对同步、异步和阻塞调用有帮助的函数。还有对一个节点池同步和异步的广播调用的调用。你永远不知道你何时会用到这些函数，所以现在复习它们会避免你编写已有的重复的代码。

I/O 组长

试试调用`rpc:call(Node, io, format, ["Hello World~n"])`，你会很惊讶地发现Hello World不是打印在远程节点上，而是在本地节点上。

这种行为可以通过称为组长（group leader）的概念解释。每个Erlang进程都有一个组长，其负责处理所有该进程的I/O。这个组长进程是可继承的，从而导致在子进程中，包括在远程节点上，和它们父进程一样共享的同一组长。当执行远程程序调用，组长随调用传给远程节点。

组长可以在运行时重新指派。内置函数`group_leader()`返回调用进程的组长的进程标识符，同时当调用`group_leader(LeaderPid, Pid)`时，把组长进程LeaderPid分配给以Pid为名的进程。

基本分布式编程模块

许多关键的模块支持Erlang中的分布式编程。其中一些我们已经接触过，还有一些是新的：

erl

这个模块包含`erl`命令，用来启动Erlang运行时系统。你可以通过在启动时设置不同的标记来改变运行时系统的行为。这些包括：

`-connect_all false`

通过这个标记，系统将不会保留一个连接节点的全局列表，这样可以阻止全局声明。

`-hidden`

它的作用是启动一个隐藏节点，这通常是出于操作和维护的目的。

`-name Name/-sname Name`

这些标记赋予节点一个长或短的名字，即Name。

`-setcookie Cookie`

这个标记把节点cookie的值设为Cookie。

erlang

这个erlang模块是Erlang内置函数的集合，其中很多是自动输入的，因此可以不需添加erlang前缀而调用。这里那些没有自动输入的函数，需要加上erlang模块的前缀：

`disconnect_node(Node)`

这个函数断开与作为参数传入的Node的连接。

`erlang:get_cookie()`

如果本地节点存活，则返回本地节点的当前cookie值，否则返回nocookie。

`monitor_node(Node, Flag)`

这个函数打开或关闭对Node节点的监控，其行为取决于Flag为真或假。这里还有一种变体`monitor_node/3`，它是非自动载入的。

`node()`

它返回本地节点的名字Name@Host，或如果节点非存活则返回nonode@nohost。

`node(Arg)`

它返回Arg所在的节点：Arg可以是一个进程标识符、一个引用或是一个端口。

`nodes()`

它返回在系统中可见节点的一个列表，但不包含本地节点。`node (type)`会返回特定节点的列表，其中类型是基元hidden或connected。

`erlang:set_cookie(Node, Cookie)`

设置Node节点的cookie值为Cookie。

`spawn(Node, Module, Function, ArgumentList)`

这个函数在节点Node上执行 `spawn(Module, Function, ArgumentList)`函数。

`spawn_link/4` 与`spawn_link/3`类似。

net_kernel

这个模块包含对手动启动、停止、连接和监控节点的构造。这些函数会由运行时系统自动调用，但用户可以通过修改来使用。

net_adm

这个模块包含各种有帮助的函数，其中包括ping（前面提到过）和用以检验本地主机文件的函数等。

epmd进程

当运行本章中分布式Erlang的示例代码时，你也许会注意到运行了一个名为epmd的命令的操作系统线程。epmd的命令是Erlang运行时系统的一部分，它为Erlang分布式的节点扮演了端口映射看守程序的角色。无论有多少分布式节点运行在它上面，每个机器只会启动一个epmd看守程序进程，该看守程序进程会监听来自端口4369的所有连接请求，并且把它们映射到被接节点的监听端口。如果还没有开始运行，当启动你的第一个分布式Erlang节点时，epmd会自动运行。然而通过手动开启，可以传递一系列的命令和设置参数。

在解决分布式有关的故障，配置分布式Erlang以穿透防火墙或模拟繁忙的网络时，你会发现epmd命令非常有用。可执行程序位于Erlang根目录下，与虚拟机的二进制文件在一起。以下是可以传递给epmd的标记：

-help

打印一个调试命令的列表。这些命令并不总是在用户手册中列出。

-port PortNumber

改变监听端口。这对于处理防火墙中的特殊端口很有用。

-names

列出本地节点的名字。在Erlang作为后台进程运行而没有终端来查找命名冲突时，这个选项很有用。

-daemon

启动epmd为一个看守进程。

-kill

终止empd进程。以连接的进程保持连接，但是与主机建立新的尝试连接会失败。重启epmd会导致所有连接节点的信息丢失。新的节点将可以互联，但旧的节点不会。

-packet_timeout

设置在epmd超时和关闭连接前，连接无效时间的秒数。连接通过keepalive机制保持开放；如果没有其他的通信，消息tic被发送且以一个tok作为响应。

-delay_accept and -delay_write

用于在测试环境里模拟服务器繁忙和网络阻塞。

在防火墙后的分布式Erlang

当在防火墙后运行分布式Erlang节点的时候，你需要打开epmd监听的端口，默认是

4369, 但可按你的喜好改变它, 只需端口在你的节点集群中是一致的。你还需要为个别节点的互联打开端口。你可以通过运行以下命令指定端口范围:

```
application:set_env(kernel, inet_dist_listen_min, 9100)
application:set_env(kernel, inet_dist_listen_max, 9105)
```

这些命令强制Erlang分布式使用从9100到9105的端口。你可以按需要的范围替换这些值。

练习

练习11-1: 分布式相联存储器

设计一个相连存储器的分布式版本, 其中值和标签相关联。可以通过储存标签/值对, 并查找与标签相关联的值。例如Email的地址簿, 其中Email地址(值)与昵称(标签)相关联。

在同一主机的两个节点上复制这个存储器, 向其中一个节点(随机选择或交替选择)发送查找命令, 然后向两个节点发送更新命令。

重新实现你的系统, 使其存储节点在不同主机上(从彼此或从前端)。如果这样做你需要注意什么?

怎样重新实现你的系统, 使其包含3个或4个存储节点?

设计一个系统来测试你对这个练习的答案。它应该能生成随机存储和检索请求。

练习11-2: 系统监控

在测试条件下设计一个系统监控你的分布式存储系统的行为。这个系统——可以是全局系统中的另外一个节点——应该记录带宽和负载均衡的信息。如果系统过载, 它的行为会是怎样的?

OTP行为包

在前面的内容中，我们已经介绍了当程序使用Erlang并发模型时反复出现的模式。我们曾经讨论过并发系统的共同功能，你也看到了进程会以类似的方式处理非常不同的任务。因此我们强调了特殊情况和潜在的问题，当处理并发时必须处理它们。

例如，让我们考虑一个拥有50位开发人员的项目，他们跨越了多个地理区域。如果该项目没有进行适当协调，也没有提供模板，那么该项目可能结果会有多少不同的客户端/服务器的实现来结束呢？更危险的是，这样的实现中有多少能够正确地处理特殊的边界情况和与并发相关的错误呢？要是没有代码检查，是否存在一种跨系统的统一方式来处理客户端发出请求到服务器后导致服务器崩溃的情况呢？或者能否保证来自一个请求的响应确实是该响应，而不是任何其他恰好兼容内部协议的消息呢？

通过提供对最常见的并发设计模式的库模块的实现，OTP行为包（OTP behavior）能够应对所有这些问题。程序员并没有意识到，在其背后的库模块能确保以一致的方式处理错误和特殊情况。因此，OTP行为包提供了标准化的构建板块（building block）的一个集合，用于设计和构建工业级的系统。OTP行为包及其相关中间件的主题很广泛。在本章中，我们将提供一个你起步时所需要的概述。

OTP行为包介绍

OTP行为包是一种进程设计模式的形式化。以库模块方式实现它们，这些模块是与标准的Erlang发行版本一起提供的。这些库模块完成所有通用进程的功能和错误处理。由程序员所编写的具体代码存放在一个单独的模块中，而且通过一组预定义的回调函数来调用。

OTP行为包包括完成实际处理的工作进程和监控进程，其任务是监控工作进程和其他监控进程。工作进程行为包通常在图上以圆形表示，包括服务器、事件句柄和有限状态

机。监控进程在图示中以正方形表示，用于监控它们的子进程，包括工作进程和其他监控进程，所有这些构成一种我们称为监控树的图（见图12-1）。

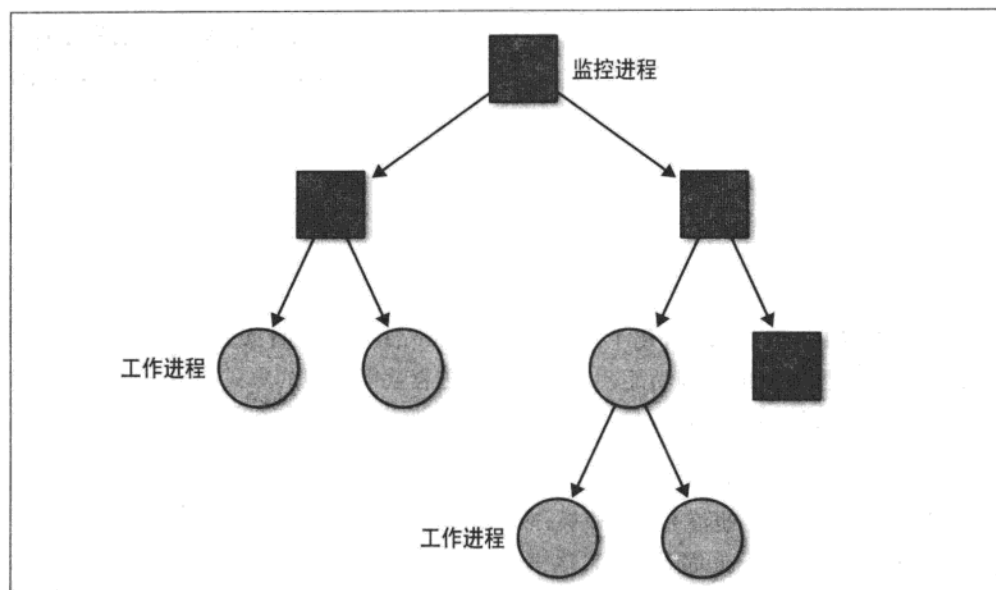


图12-1：应用软件的监控树

监控树可以被打包成一个称为应用程序的行为包。OTP应用程序不仅是Erlang系统的构建板块，而且也是打包可重用组件打包的一种方法。工业级系统包括了一组松耦合和分布式的应用程序。这些应用程序是标准Erlang发行版本的一部分，或者是由像你这样的程序员所开发的特定的应用程序。

不要把一个OTP应用程序与应用程序的更一般概念混淆了，它通常是指用一个更完整的系统解决高层次的任务。OTP应用程序的例子包括Mnesia数据库，这将在第13章中讨论，一个SNMP代理，或者在第10章中所介绍的移动用户数据库，在本章后面部分将它转化作为行为包应用程序。OTP应用程序是一种可重用的组件，该组件把库模块与监控进程和工作进程捆绑在一起。从现在起，当我们提到一个应用程序时指的是OTP应用程序。

行为包模块包括了所有的通用代码。虽然也可以实现自己的行为包模块，但是这样做很罕见，这是因为作为Erlang/OTP发行版本一部分的行为包模块能够应对多数在你的代码中使用的设计模式。行为包模块提供操作系统所提供的如下通用功能：

- 生成或者注册进程。
- 以同步或异步调用方式来发送和接收客户消息，包括定义内部的消息协议。

- 存储循环数据和管理循环进程。
- 终止进程。

虽然Erlang提供了行为包模块，但是程序员必须开发回调模块（见图12-2）。在第5章中我们介绍了回调模块的概念。回调模块包含了所有具体的代码来提供所要求的功能。这些代码通过一个针对每个行为而标准化的接口来激活。

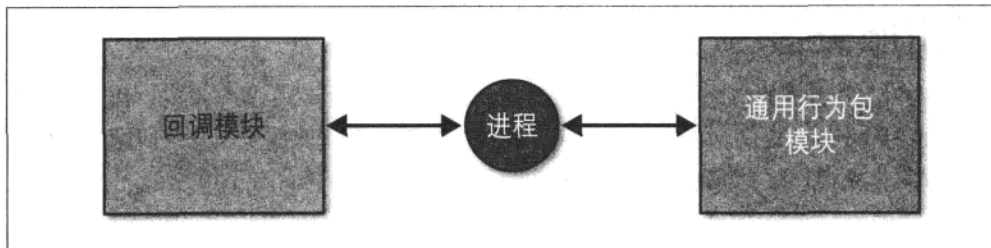


图12-2：把代码分解成通用模块和具体模块

循环数据（loop data）是一个包含数据的变量，该数据是行为包需要在调用之间存储的数据。在调用后将会返回循环数据的更新版本。此更新的循环数据通常称为新循环数据（new loop data），它将在下一次调用时作为参数进行传递。循环数据通常也称为行为包状态（behavior state）。

回调模块中包含的用来提供所要求的特定行为的功能如下所示：

- 初始化进程循环数据，如果这一进程已注册了，那么就初始化进程名称。
- 处理特定的客户端请求，如果是同步，则发送应答给客户端。
- 在进程请求之间处理和更新进程数据。
- 清除循环数据直到终止。

把代码分解成通用行为包库和具体的回调模块，这样做有许多好处：

- 由于许多可能发生的特殊情况和错误已经被稳定的和经过验证的行为包处理，所以在你的产品中错误会更少。
- 正因为如此，而且也因为你编写好了如此多的代码，你可以期望产品的面市时间更短。
- 它强制程序员使用一种避免并发应用程序中的典型错误的方式编写代码。
- 最后，你的整个团队将具有共同的编程风格。在阅读别人的代码的同时，已经具备了对现有行为包的基本理解，这对于了解客户端/服务器协议，查找进程在何处和

如何启动或终止，或如何处理循环数据等，都会十分容易。所有这一切都是通用行为包库来管理的。由于是在回调模块里编码，你不必专注于每一件事情是如何做的，而可以专注于在这种情况下正在做的特定的事情。

在接下来的内容中，我们将说明一些最重要的行为包，包括通用服务器和监控进程，以及如何把它们打包成应用程序。

通用服务器

实现客户端/服务器的行为包的通用服务器是在`gen_server`行为包里定义的，作为标准库应用程序的一部分提供。本章将从第10章的移动客户数据库的例子来介绍回调的工作原理。如果你不记得这个例子，那么在继续阅读之前最好快速浏览一下。

我们将重写 `usr.erl` 模块，把它从Erlang进程迁移到`gen_server`行为包。这样做的目的是：我们将不会接触`usr_db`模块，同时保持后台数据库。当学习这个例子时，如果你对细节感兴趣，那么可以参阅`gen_server`模块手册。

启动服务器

利用`gen_server`行为包时，不是使用`spawn`和`spawn_link`内置函数，而是可以使用`gen_server:start/4`和`gen_server:start_link/4`函数。

`spawn`和`start`之间的主要区别是调用的同步性。使用`start`代替`spawn`，使得工作进程的启动更加确定，以防止意外的竞争状况发生，因为调用将不会返回工作进程的标识符，直到初始化完成。调用这些函数的方法如下（对于这两个函数的每一个，我们显示两个版本）：

```
gen_server:start_link(ServerName, CallbackModule, Arguments, Options)
gen_server:start(ServerName, CallbackModule, Arguments, Options)
gen_server:start_link(CallbackModule, Arguments, Options)
gen_server:start(CallbackModule, Arguments, Options)
```

在上面的调用中：

ServerName（服务器名称）

是格式为`{local, Name}`或者`{global, Name}`的元组，如果已经注册，那么它们就表示进程的局部或全局名称。如果你不想注册该进程，而是使用它的进程标识符来引用它，那么就省略该参数，并取而代之调用函数`start_link/3`或`start/3`。

CallbackModule（回调模块）

是模块的名称，其中存放了具体的回调函数。

Arguments (参数)

是一个有效的Erlang项元，它将传递到init/1回调函数。你可以选择传递哪种类型的项元：如果你有许多参数需要传递，那么使用列表或元组；如果没有，那么传递基元或空列表，在回调函数中忽略它。

Options (选项)

是一个列表，允许你设置内存管理标记 `fullsweep_after`和`heapspace`，以及跟踪和调试标记。大多数的行为包实现只是传递一个空列表。

函数`start`将生成新的进程，该进程使用已经提供的参数调用`CallbackModule`模块中的`init(Arguments)`回调函数。函数`init`必须初始化服务器的`LoopData`，并返回一个格式为`{ok, LoopData}`的元组。`LoopData`包含循环数据的第一个实例，并且将在回调函数之间进行传递。如果你想存储一些传递到`init`函数的参数，那么你就要在`LoopData`变量中这样做。

`start_link`和`start`函数之间的明显不同点是，`start_link`链接到其父进程，而`start`并不是。不过，这里需要特别一提的是，把其自身链接到监控进程是OTP行为包的责任。`start`函数经常被用来从终端测试行为包，因为导致终端进程崩溃的输入错误并不会影响行为包。所有`start`和`start_link`的变化版本都会返回`{ok, Pid}`。

在继续介绍这个例子之前，让我们快速回顾一下到目前为止所讨论的内容。借助`gen_server:start_link`调用，你可以启动`gen_server`行为包。其结果是生成调用`init/1`回调函数的一个新进程。这个函数初始化了`LoopData`并返回元组`{ok, LoopData}`。

在例子中，我们调用`start_link/4`，用与回调模块相同的名称注册该进程，并调用`MODULE`宏。我们传递一个参数，即`Dets`表的文件名。保存该选项列表为空：

```
start_link(FileName) ->
  gen_server:start_link({local, ?MODULE}, ?MODULE, FileName, []).

init(FileName) ->
  usr_db:create_tables(FileName),
  usr_db:restore_backup(),
  {ok, null}.
```

尽管监控进程可能会调用`start_link/4`函数，但`init/1`回调函数却被不同的进程调用：刚刚生成的进程。在我们的服务器中并不真正地需要`LoopData`变量，因为ETS和`Dets`表是已命名的。尽管如此，当返回`{ok, LoopData}`结构时，仍然必须包含一个值，因此我们通过返回基元`null`绕开它。如果ETS和`Dets`表没有`named_tables`，那么我们将在这里传递它们的引用。

只做必要的事情并尽量减少在init函数中的操作，而调用init是同步调用，它会阻止其他所有的序列化进程启动，直到它返回。

传递消息

如果你想把消息发送到你的服务器，那么请使用下列调用：

```
gen_server:cast(Name, Message)
gen_server:call(Name, Message)
```

在上面的调用中：

Name（名称）

要么是以`local`名称注册的服务器，要么是元组`{global, Name}`。这也可能是服务器的进程标识符。

Message（消息）

是一个有效的Erlang项元，它包含了传递到服务器的消息。

对于异步消息请求，你可以使用`cast/2`。如果你使用进程标识符，那么这样的调用直接返回基元`ok`，无论你发送到`gen_server`的信息是否存活。这些语义与标准的`Name ! Message`结构没有什么不同，而如果该注册的进程`Name`不存在时，那么调用进程终止。

在收到该消息时，`gen_server`将在回调模块中调用回调函数`handle_cast(Message, LoopData)`。`Message`是传递给`cast/2`函数的参数，而`LoopData`是原本由`init/1`回调函数返回的参数。`handle_cast/1`回调函数处理该消息的细节，并在完成时返回元组`{noreply, NewLoopData}`。在未来服务器的调用中，当一个消息发送到服务器时将会把最近所返回的`NewLoopData`值作为参数进行传递。

如果你想发送一条同步消息给服务器，那么可以使用`call/2`函数。在收到此消息时，进程使用回调模块中的函数`handle_call(Message, From, LoopData)`。它包含特定服务器的具体代码，并在完成时才返回元组`{reply, Reply, NewLoopData}`。直到现在，`call/3`函数调用才同步地返回值`Reply`。如果你正在发送信息的目标进程不存在，那么不论它是否已经注册，触发函数调用的进程都将会终止。

让我们从服务API中的两个函数开始介绍，稍后将提供整个程序。它们被客户端进程调用，并且导致一条同步消息发送给与回调模块相同的名字注册的服务器进程。注意我们如何在客户端验证数据。如果客户端发送不正确的信息，那么它会终止。

```
set_status(CustId, Status) when Status==enabled; Status==disabled->
    gen_server:call(?MODULE, {set_status, CustId, Status}).
```

```
delete_disabled() ->
    gen_server:call(?MODULE, delete_disabled).
```

在收到该消息时，`gen_server`进程调用`handle_call/3`回调函数，该函数以与发送信息相同的顺序处理这一消息：

```
handle_call({set_status, CustId, Status}, _From, LoopData) ->
    Reply = case usr_db:lookup_id(CustId) of
        {ok, Ustr} ->
            usr_db:update_usr(Ustr#usr{status=Status});
        {error, instance} ->
            {error, instance}
    end,
    {reply, Reply, LoopData};

handle_call(delete_disabled, _From, LoopData) ->
    {reply, usr_db:delete_disabled(), LoopData}.
```

请注意回调函数的返回值。元组包含控制基元`reply`，告诉`gen_server`的通用代码，该元组的第二个元素是发送到客户端的`Reply`。元组的第三个元素是新的`LoopData`，在一个服务器的新迭代中，它作为`handle_call/3`函数的第三个参数进行传递，在这里的两种情况下它是不变的。`_From`参数是一个元组，它包含一个唯一的信息引用和客户端的进程标识符。在库函数中把元组作为一个整体使用，对此我们将不会在本章进行讨论。在大多数情况下，你不需要它。

幕后`gen_server`库模块存在大量的机制和保障措施内置于函数中。如果你的客户发送同步消息到你的服务器，并且你在5秒钟内没有得到响应，那么运行`call/2`函数的进程将会终止。你也可以通过使用以下代码重写它：

```
gen_server:call(Name, Message, Timeout)
```

其中`Timeout`是以毫秒为单位的值或基元`infinity`。超时机制的出现原本是为了预防死锁，确保意外互相调用的服务器在默认超时之后终止。崩溃报告将会被记录下来，并希望由此产生相应的补丁。大多数应用程序将超时5秒才正常工作，但在沉重的负荷条件下，你可能需要微调该值，甚至可能使用`infinity`，这种选择非常依赖于应用程序。所有Erlang/OTP的核心代码都使用`infinity`。

在使用`gen_server:call/2`函数时，其他的保障措施包括：把一条消息发送到根本不存在的服务器，或者在发送其响应前，服务器已经崩溃了。在这两种情况下，调用进程将会终止。在Erlang中，若发送一条在`receive`语句中永不模式匹配的消息是一个错误，这可能导致内存泄漏。

如果你对服务器发出一次调用或进行一个转换，但没有在`handle_call/3`和`handle_cast/2`调用里处理消息，你认为会发生什么？在OTP中，当进行一次调用或转

换时，该信息总是可以从进程邮箱中提取出来，且激活各自的回调函数。如果没有回调函数模式匹配作为第一个参数进行传递的消息，那么这个进程将会以函数语句错误而崩溃。结果，这些问题可以在测试的早期阶段被捕获，并分别处理。

停止服务器

你如何停止服务器？在你的`handle_call/3`和`handle_cast/2`回调函数中，不是返回`{reply, Reply, NewLoopData}`或者`{noreply, NewLoopData}`，你可以分别返回`{stop, Reason, Reply, NewLoopData}`或者`{stop, Reason, NewLoopData}`。有些东西需要触发这个返回值，往往是一条停止消息发送到服务器。当收到包含`Reason`和`LoopData`的`stop`元组时，通用代码执行`terminate(Reason, LoopData)`回调函数。

`terminate`函数是插入所需要代码理所当然的地方，该代码清理服务器的`LoopData`，以及系统使用的其他持久性数据。在这个例子中，这将意味着关闭ETS和Dets表。`stop`调用不必在一个同步调用中出现，所以当实现它时可以使用`cast`：

```
stop() ->
    gen_server:cast(?MODULE, stop).

handle_cast(stop, LoopData) ->
    {stop, normal, LoopData}.

terminate(_Reason, _LoopData) ->
    usr_db:close_tables().
```

请记住，`stop/0`将由客户端进程调用，而`handle_cast/2`和`handle_call/2`由行为包进程调用。在`handle_cast/2`回调函数中，我们在`stop`结构里返回`normal`。任何不是`normal`的理由都将生成错误报告。

由于每一秒钟可能生成和终止数以千计的通用服务器，它们的每一个输出错误报告并不是出路。只有当这是不应该发生时，并且你没有办法恢复时，才可以返回一个非正常值。一个被关闭的套接字或一条从外部端口收到的已损坏的消息，不是引发非正常终止的理由。另外，非正常终止可能是内部数据的损坏或配置文件的丢失造成的。

如果你的服务器因为一个运行时错误崩溃，那么将调用`terminate/2`。但是如果你的行为包收到来自其父进程的退出信号，那么当你设置为捕获退出时，将会调用`terminate`。提防这种特殊的情况，因为我们已经遇到很多次，特别是在从终端使用`start_link`启动该行为包时。

警告： 把行为包回调函数用作库函数，并从程序的其他部分激活它们，这是一个极坏的做法。例如，你决不应该从另一个模块调用usr_db:init(FileName)来创建和填充数据库。行为包回调函数的调用，应该是作为系统中发生事件的结果，由行为包库模块发出，而永远不应该直接由用户发出。

完整实例

这是由第10章的usr.erl模块作为一个gen_server行为包改写的：

```
%% File : usr.erl
%% Description : API and gen_server code for cellphone user db

-export([start_link/0, start_link/1, stop/0]).
-export([init/1, terminate/2, handle_call/3, handle_cast/2]).
-export([add_usr/3, delete_usr/1, set_service/3, set_status/2,
        delete_disabled/0, lookup_id/1]).
-export([lookup_msisdn/1, service_flag/2]).
-behavior(gen_server).

-include("usr.hrl").

%% Exported Client Functions
%% Operation & Maintenance API

start_link() ->
    start_link("usrDb").

start_link(FileName) ->
    gen_server:start_link({local, ?MODULE}, ?MODULE, FileName, []).

stop() ->
    gen_server:cast(?MODULE, stop).

%% Customer Services API

add_usr(PhoneNum, CustId, Plan) when Plan==prepay; Plan==postpay ->
    gen_server:call(?MODULE, {add_usr, PhoneNum, CustId, Plan}).

delete_usr(CustId) ->
    gen_server:call(?MODULE, {delete_usr, CustId}).

set_service(CustId, Service, Flag) when Flag==true; Flag==false ->
    gen_server:call(?MODULE, {set_service, CustId, Service, Flag}).

set_status(CustId, Status) when Status==enabled; Status==disabled->
    gen_server:call(?MODULE, {set_status, CustId, Status}).

delete_disabled() ->
    gen_server:call(?MODULE, delete_disabled).

lookup_id(CustId) ->
    usr_db:lookup_id(CustId).

%% Service API
```



```

lookup_msisdn(PhoneNo) ->
    usr_db:lookup_msisdn(PhoneNo).

service_flag(PhoneNo, Service) ->
    case usr_db:lookup_msisdn(PhoneNo) of
        {ok, #usr{services=Services, status=enabled}} ->
            lists:member(Service, Services);
        {ok, #usr{status=disabled}} ->
            {error, disabled};
        {error, Reason} ->
            {error, Reason}
    end.

%% Callback Functions

init(FileName) ->
    usr_db:create_tables(FileName),
    usr_db:restore_backup(),
    {ok, null}.

terminate(_Reason, _LoopData) ->
    usr_db:close_tables().

handle_cast(stop, LoopData) ->
    {stop, normal, LoopData}.

handle_call({add_usr, PhoneNo, CustId, Plan}, _From, LoopData) ->
    Reply = usr_db:add_usr(#usr{msisdn=PhoneNo,
                                id=CustId,
                                plan=Plan}),

    {reply, Reply, LoopData};

handle_call({delete_usr, CustId}, _From, LoopData) ->
    Reply = usr_db:delete_usr(CustId),
    {reply, Reply, LoopData};

handle_call({set_service, CustId, Service, Flag}, _From, LoopData) ->
    Reply = case usr_db:lookup_id(CustId) of
        {ok, User} ->
            Services = lists:delete(Service, User#usr.services),
            NewServices = case Flag of
                true -> [Service|Services];
                false -> Services
            end,
            usr_db:update_usr(User#usr{services=NewServices});
        {error, instance} ->
            {error, instance}
    end,
    {reply, Reply, LoopData};

handle_call({set_status, CustId, Status}, _From, LoopData) ->
    Reply = case usr_db:lookup_id(CustId) of
        {ok, User} ->
            usr_db:update_usr(User#usr{status=Status});
        {error, instance} ->
            {error, instance}
    end

```

```

end,
{reply, Reply, LoopData};

handle_call(delete_disabled, _From, LoopData) ->
{reply, usr_db:delete_disabled(), LoopData}.

```

运行gen_server

在终端中测试gen_server实例时，你将得到使用你自己编写代码的服务器进程完全相同的行为。但是这个代码更稳定，因为死锁、服务器崩溃、超时以及其他并发编程相关的错误，都是在幕后处理的：

```

1> c(usr).
/Users/Francesco/otp/usr.erl:11: Warning: undefined callback
function code_change/3 (behaviour 'gen_server')
/Users/Francesco/otp/usr.erl:11: Warning: undefined callback
function handle_info/2 (behaviour 'gen_server')
{ok,usr_db}
2> c(usr_db).
{ok,usr_db}
3> rr("usr.hrl").
[usr]
4> usr:start_link().
{ok,<0.86.0>}
5> usr:add_usr(700000000, 0, prepay).
ok
6> usr:set_service(0, data, true).
ok
7> usr:lookup_id(0).
{ok,#usr{msisdn = 700000000,id = 0,status = enabled,
plan = prepay,
services = [data]}}
8> usr:set_status(0, disabled).
ok
9> usr:service_flag(700000000,lbs).
{error,disabled}
10> usr:stop().
ok

```

你是否注意到模块里的-behavior(gen_server)指令？这是告诉编译器，你使用的模块是gen_server回调模块，因此，它期望许多回调函数。如果所有的回调函数没有实现，那么你会获得警告信息，这是第一个命令行中的compile操作结果。不要编写你自己的代码，以避免出现这些警告。如果你的服务器没有异步调用，那么你显然不需要handle_cast/2。可以忽略这些警告。

注意：英国或加拿大的读者：不要绝望或摇头！欢迎在你的指令中使用你的英式英语拼写：-behaviour(gen_server)。编译器是双语的，可以同时处理美式英语和英式英语。

如果你使用Pid!Msg格式的原始Erlang消息传递来给服务器发送一条消息，那会发生什么？这是可能的，因为gen_server是一个Erlang进程，它能够像其他任何的进程一样发送和接收消息。不要害羞，请尝试一下：

```
11> {ok, Pid} = usr:start_link().
{ok,<0.119.0>}
12> Pid ! hello.
hello

=ERROR REPORT==== 24-Jan-2009::18:08:07 ===
** Generic server usr terminating
** Last message in was hello
** When Server LoopData == null
** Reason for termination ==
** {'function not exported',[{usr,handle_info,[hello,null]],
                             {gen_server,handle_msg,5},
                             {proc_lib,init_p,5}}]
** exception exit: undef
    in function usr:handle_info/2
       called as usr:handle_info(hello,null)
    in call from gen_server:handle_msg/5
    in call from proc_lib:init_p/5
```

哎呀！有些事情并没有按计划进行。查看错误，试着找出发生了什么。Pid!Msg格式不兼容OTP的内部消息协议。在收到一条不兼容规范的消息时，gen_server进程会尝试调用函数usr:handle_info(hello, null)，其中hello是消息，而null是循环数据。

每当进程收到一条无法识别消息的时候，就会调用回调函数handle_info/2（注1）。其中可能包括来自你所监测节点的“节点失去”（node down）消息，或者来自你所连接进程的退出信号，或者仅仅是使用...!...结构发送的消息。如果你期待这样的消息，但对此并不感兴趣，那么把以下定义添加到你的回调模块中，同时不要忘记把它导出：

```
handle_info(Msg, LoopData) ->
    {noreply, LoopData}.
```

另外，如果你想处理这些消息，那么你应该以调用的第一个参数来模式匹配它们。如果你的服务器不期望收到非OTP兼容的消息，那么不要添加handle_info/2调用，它将忽略到达的消息，“以防万一”。这样做被认为是防御性编程，它可能使你隐藏的任何错误更难以检测。

注1： 你是否在实例的编译器警告里注意到它了？

警告：OTP的缺点之一是各种行为包模块的层面，这将会影响性能。在尝试从它们的调用中节省几微秒时，开发人员已经知道使用Pid!Msg结构，而不是一个gen_server的case，并且在handle_info/2回调函数中处理它们的消息。

不要这样做！你将会使得你的代码无法得到支持和维护，并且首先失去了使用OTP的许多优点。如果你对节省几个微秒还痴迷，那么尽量克制，当且仅当你知道你的程序运行不够快时才去优化。我们将会在第20章中讨论优化和涉及真正影响代码性能的内容。

在我们看下一个行为包之前，这里列出了gen_server输出的API导致的回调函数及它们期望的返回值的一个小结：

启动

请看下列调用：

```
start(Name, Mod, Arguments, Opts)
start_link(Name, Mod, Arguments, Opts),
```

其中Name是一个可选的参数，它将生成新的进程。这一进程会导致应该返回{ok, LoopData}或者{stop, Reason}中的一个的回调函数init(Arguments)调用。如果init/1返回{stop, Reason}，那么将不调用terminate/2“清理”函数。

同步通信

使用call(Name, Msg)发送一条同步消息到你的服务器。这将导致服务器进程调用回调函数handle_call(Msg, From, LoopData)。期望的返回值包括：

```
{reply, Reply, NewLoopData}
{stop, Reason, Reply, NewLoopData}.
```

异步通信

如果你想发送一条异步消息，那么请使用cast(Name, Msg)。在handle_cast(Msg, LoopData)回调函数里处理它并返回{noreply, NewLoopData}或者 {stop, Reason, NewLoopData}。

非OTP兼容信息

在收到非OTP兼容消息时，gen_server将执行handle_info(Msg, LoopData)回调函数。该函数应返回{noreply, NewLoopData}或者{stop, Reason, NewLoopData}。

终止

当从回调函数中的一个收到stop结构时（init除外），或当捕获存在而发生了异常进程终止时，就会触发terminate(Reason, LoopData)回调函数。在terminate/2中，你可以取消在init/1中所做的事情。这将忽略它的返回值。

监控进程

监控进程行为包的任务是监控它的子进程，并且基于一些预设定的规则，当子进程终止时采取行动。组成监控树的子进程包括监控进程和工作进程，工作进程是OTP行为包，它们包括`gen_server`、`gen_fsm`（支持有限状态机行为包）和`gen_event`（提供事件处理功能）。

工作进程必须把自身链接到监控进程行为包，并且处理不透露给程序员的特殊系统消息。这和原本在Erlang中一个进程链接到另一个进程的情况不同，因此我们不能混淆这两种机制。基于这个原因，将Erlang进程用你所知道的形式加到监控树中是不可能的。因此，在本章的剩下部分中，我们将重点描述OTP框架内的监控。

使用`start_link`函数启动一个监控进程：

```
supervisor:start_link(ServerName, CallbackModule, Arguments)
supervisor:start_link(CallbackModule, Arguments)
```

在上面的调用中：

ServerName

是在监控进程中注册的名字，它是一个格式为`{local, Name}`或`{global, Name}`的元组。如果你不想注册监控进程，那么可以使用二元函数。

CallbackModule

是放置`init/1`回调函数的模块的名字。

Argument

是一个有效的Erlang项元，当`init/1`回调函数被调用时传递给它。

注意，不像`gen_server`那样，监控进程并不接收任何选项。`start_link`函数会生成一个新进程，用来调用`init/1`回调函数。在初始化监控进程时，`init`函数必须返回如下格式的元组：

```
{ok, {SupervisorSpecification, ChildSpecificationList}}
```

其中，监控进程规范是一个元组，它包含关于如何处理进程崩溃和重新开始的信息。子进程规范列表（`child specification list`）指定监控进程要启动和监控的子进程，同时也包含如何终止和重新开始它们的信息。

监控进程规范

监控进程规范是一个包括3个元素的元组，若用于描述子进程终止，那么这个监控进程应该如何应对：

```
{RestartStrategy, AllowedRestarts, MaxSeconds}
```

重新启动策略决定了当一个子进程的兄弟进程终止，其他的子进程会如何受到影响。它可能是下面当中的一个：

`one_for_one`

将重新启动终止的子进程，而且不影响任何其他子进程。如果这棵监控树同一级别的所有进程都不依赖于其他进程，你应该采用这种策略。

`one_for_all`

会终止所有的子进程并且重新启动它们。如果不考虑所有子进程的开始顺序，在子进程之间存在很强的依赖性时，你应该使用这种策略。

`rest_for_one`

会终止所有在子进程崩溃后启动的子进程，并且重新启动它们。这种策略假设进程以依赖的顺序启动，其中被生成的进程依赖于它们已经开始的兄弟进程。

如果你的进程进入一个重新启动的循环会发生什么？它崩溃了之后又重新启动，但是又碰到相同的已损坏数据，因此它又崩溃了。不能这样一直运行下去。这时候就该引入 `AllowedRestarts` 了。它定义了监控进程允许的异常终止的最大数量，并且进程可以处理 `MaxSeconds` 秒。如果异常终止发生的数量超过了允许的数量，就假定监控进程不能解决这个问题，它自己会终止。这个监控进程的监控进程将收到一个退出信号，并根据它的配置决定该如何继续。

找到 `AllowedRestarts` 和 `MaxSeconds` 合理的值并不容易，因为它们依赖于应用程序。在实际应用中，我们使用的值可以从每秒重新启动10次一直到一小时重新启动一次。你的选择不得不依赖于你的子进程要做什么，你期望你的监控进程管理多少子进程，以及你如何设置你的监控策略。

子进程规范

在 `init/1` 函数返回的结构中，第二个参数是一个子进程规范列表。子进程规范为监控进程提供每一个子进程的特性，包括如何启动的指令。每一个子进程规范有如下格式：

```
{Id, {Module, Function, Arguments}, Restart, Shutdown, Type, ModuleList}
```

在先前的代码中：

`Id`

是监控进程中特定子进程的唯一标识符。因为子进程可能崩溃并重新启动，它的进程标识符可能会改变。这个时候，这个标识符就可以取代进程标识符使用。

监控进程使用元组{Module, Function, Arguments}启动一个子进程。这个监控进程最后必须为特定的OTP行为包调用start_link函数，然后返回{ok, Pid}。

Restart

是基元transient、temporary或permanent当中之一。短暂进程（transient process）永远不会重新启动。暂时进程（temporary process）只在异常终止的时候重新启动，而永久进程（permanent process）总是重新启动，不管它是异常终止还是正常终止。

Shutdown

指定一个毫秒数值，一个设置捕获退出的行为包在从它的监控进程接收到shutdown信号后，必须在规定的毫秒数内执行它的terminate回调函数。监控进程发出shutdown信号的原因可以是它达到了允许重新启动子进程的最大数量，或者由于rest_for_one或one_for_all的重新启动策略。如果子进程没有在这个时间里自己终止，那么监控进程会无条件地终止它。Shutdown也可以采用基元infinity，如果进程为监控进程，则必须选择一个值；如果进程被无条件地终止，也可以采用基元brutal_kill。

Type

指定这个子进程是一个工作进程还是一个监控进程。

ModuleList

是执行这个进程的模块列表。发行句柄用它来决定在软件升级过程中，它应该暂停哪些进程。根据经验，应该总是包含行为包的回调模块。

在一些情况下，可以从配置文件中动态地创建子进程规范。但是在大多数情况下，它们都是在监控进程的回调模块中静态编码的。Init/1函数是唯一需要导出的回调函数。

子进程规范列表中很容易引入有关句法和语义的错误，因为它们趋向于变得相当复杂。监控模块的帮助函数check_childspecs/1接收一个子进程规范列表，然后返回ok或者元组{error, Reason}。在“监控进程实例”一节中将给出一个移动用户数据库的子进程规范的实例。为了确保你理解发生了什么，我们将把所有的记录映射到它们各自在子进程规范结构中的字段。

监控进程实例

在这个例子中，usr_sup模块是一个监控进程行为包，用于监控在前面的内容中提到的gen_server的usr例子中的一个子进程。

我们调用start_link/0来启动监控进程。注意，我们省略了为Dets表传递一个文件名的选项，因为包含它原本只是出于测试的目的。特别注意子进程和由init/1返回的监控进程规则：

```
-module(usr_sup).
-behavior(supervisor).

-export([start_link/0]).
-export([init/1]).

start_link() ->
    supervisor:start_link({local, ?MODULE}, ?MODULE, []).

init(FileName) ->
    UsrChild = {usr,{usr, start_link, []},
                permanent, 2000, worker, [usr, usr_db]},
    {ok,{one_for_all,1,1}, [UsrChild]}.
```

现在你可以在终端里试试看。不要只测试肯定的情况，也试着终止子进程，确保它已经被重新启动。最后，在MaxSeconds定义的时间内终止服务器超过MaxRestart次（在这个例子中是每秒两次），然后看看监控进程是否终止：

```
13> c(usr_sup).
{ok,usr_sup}
14> usr_sup:start_link().
{ok,<0.149.0>}
15> whereis(usr).
<0.150.0>
16> exit(whereis(usr), kill).
true
17> whereis(usr).
<0.156.0>
18> usr:lookup_id(0).
{ok,#usr{msisdn = 7000000000,id = 0,status = disabled,
        plan = prepay,
        services = [data]}}
19> exit(whereis(usr), kill).
true
20> exit(whereis(usr), kill).
** exception exit: shutdown
```

注意： 当一个进程终止时，所有它创建的ETS表都被销毁。如果你让ETS表在进程重新启动后幸存下来，而不想引起处理Dets表或文件系统的负担，并且让ETS表在进程重新启动时幸存下来，一个技巧是让你的监控进程在它的init/1函数中产生这些表，而不是在生成的进程中产生。

动态子进程

到目前为止，我们只涉及了静态子进程。如果你需要监控进程能够动态生成子进程，用

于处理特定的事件、管理任务并在完成后终止，你该怎么办？每条接收到的即时消息，或进入即时消息服务器的同伴的更新都是这样的例子。你不能在init回调函数中指定这些子程序，因为它们是动态生成的。相反，你需要使用对函数`supervisor:??_child/2`的调用：

```
supervisor:start_child(SupervisorName, ChildSpec)
supervisor:terminate_child(SupervisorName, Id)
supervisor:restart_child(SupervisorName, Id)
supervisor:delete_child(SupervisorName, Id).
```

在上面的调用中：

SupervisorName

可以是监控进程的进程标识符或它的注册名。

ChildSpec

是一个单一的子进程规范元组，就像“子进程规范”一节中描述的一样。

Id

是在ChildSpec中定义的唯一子进程标识符。

在ChildSpec元组中特别重要的是子进程Id。即使在终止后，ChildSpec仍然由监控进程存储，并且通过它的Id引用，这样就允许进程可以停止和重新开始子进程。只有在删除时才能将子进程规范永久移除。

注意：如果你已经浏览了监控进程行为包手册，你也许会发现它没有导出一个stop函数。这是因为除了它们的父监控进程，其他任何进程都不能停止这些监控进程，这个函数并没有实现。

通过在你的监控进程回调模块中添加下列代码，可以更容易地增加你的stop函数。但是，只有在父进程调用stop时，它才有效：

```
stop() -> exit(whereis(?MODULE), shutdown).
```

如果你的监控进程没有注册，那么可以使用它的进程标识符。

应用

应用行为包用来将Erlang模块包裹成可以重复使用的组件。一个Erlang系统由一组松散结合的应用组成。其中一些应用是由程序员或开源的团体开发，另一些则是OTP发行包的一部分。Erlang运行时系统及其工具会平等对待所有应用，不管它们是不是Erlang发行包的一部分。

这里有两种应用。最常见的应用形式叫做正常应用（normal application），它们可以启动监控进程和所有相关的静态工作进程。另一种是库应用（Library application），例如标准库（它是Erlang发行包的一部分）包含库模块，但不启动监控树。这不是说代码不包含进程或监控树，只是意味着它们作为属于其他应用的监控树的一部分启动。

在本节中，我们会谈到将移动用户系统封装到一个OTP应用所需要的所有函数功能，并且启动它的高层次的监控进程。一旦完成，这个应用的行为将会像其他正常应用一样。不要忘记，在本章当我们谈到应用时指的都是OTP应用。

应用作为一个单元被加载、开始和停止，与每一个应用相关联的资源文件不仅要描述它，还要指定它的模块、注册的进程和其他配置数据。应用要遵守特殊的目录结构，它规定了beam文件、模块、资源以及包含文件所存放的位置。许多存在的工具都需要这种结构来正确构建行为包和正常工作。你可以使用`application:which_applications()`来找出你的Erlang运行时系统中运行着哪些应用：

```
1> application:which_applications().
[{stdlib,"ERTS CXC 138 10","1.15.2"},
 {kernel,"ERTS CXC 138 10","2.12.2"}]
```

标准库和内核是基本的Erlang应用的一部分。在启动运行时系统时，它们一起构成了最小的OTP子集。应用元组中的第一项是应用的名字，第二项是一个描述字符串，第三项是应用的版本号，如果你不知道在先前例子中的描述字符串是什么意思，其实还有很多人和你一样不知道，因为它是Ericsson内部产品编码机制。

在本章的后面我们会向你展示在哪儿配置应用的描述。

目录结构

在你的Erlang终端中输入代码`get_path()`，其实，在我们解释如何在代码服务器中操纵代码搜索路径时，你曾经这么做过。那时你也许还没有发现，每一个代码路径都指向一个OTP应用的一个特殊结构的目录。

让我们以Inets应用为例，详细检查它的内容。在Mac OS X中，这个特殊的Erlang安装路径如下：

```
/usr/local/lib/erlang/lib/inets-5.0.12/
```

在其他操作系统中，只需从Erlang根目录切换（`cd`）到`lib`目录下，典型情况比如：`/usr/local/lib/erlang/lib`或`C:/Program Files/erl5.6.2/lib/`，然后寻找最近的Inets版本。在一个应用的所有子目录中，下面的这些组成了正在讨论的OTP发行包里的应用：

src

包含应用中所有Erlang模块的源代码。

ebin

包含所有编译过的beam文件和应用的资源文件。在这个例子中是*inets.app*。

Include

包含所有为了应用外使用的Erlang头文件。通过使用下面的指令：

```
-include_lib("Application/include/Name.hrl")
```

其中，*Application*是没有版本号的应用目录名（在这个例子中是*inets*），*Name.hrl*是包含文件的名称，编译器会自动通过代码搜索路径选择应用所指向的版本。

Priv

是一个可选的目录，它包含必要的脚本、图片、配置文件或者其他的非Erlang相关的资源，不需知道应用的版本，你也可以使用*priv_dir(Application)*调用来存取它。

你会发现*Inets*（和其他的）应用也许会有更多的目录，包括*docs*和*examples*。这些目录在运行时对系统没有影响，只是为了方便。在一些应用中，你也许找不到*priv*目录。如果你不使用它，省略它并没有问题，但是有些人认为这不是一个好的实践经验。在真实的系统中，唯一的强制目录是*ebin*，这是因为当你把系统转交给用户时，你也许并不想包括你的源代码。

用脚本来创建这些目录是很常见的，并且还可以使用make文件在编译完你的代码后，把beam文件移动到*ebin*目录。你如何设置这些，这依赖于你的应用中的操作系统、构建系统、储存库和许多其他非Erlang相关的情况。尽管为小型项目手动设置这些是可行的，但对于大型项目，你也许想使用模板并自动完成这个任务。

应用的资源文件

应用的资源文件也称作*app*文件，包含关于你的应用的资源和依赖性的信息。切换到*Inets*应用的*ebin*子目录下并查找*inets.app*文件。这是*Inets*应用的资源文件。仔细查看，你会发现其他的所有应用也都有一个*app*文件，这个应用资源文件由一个元组组成，其中第一个元素是应用标签，第二个元素是应用的名字，第三个是特性列表。

让我们分别来看一看这些特性。注意，出于对版面空间的考虑，我们省略了例子中的一些模块：

```
{application,inets,  
 [{description,"INETC CXC 138 49"}],
```

```

{vsn,"5.0.5"},
{modules,[inets_sup,inets_app,inets_service,
          %% FTP
          ftp, ftp_progress,ftp_response,ftp_sup,
          %% HTTP client:
          http,httpc_handler,httpc_handler_sup,httpc_manager,
          %% TFTP
          tftp,tftp_binary,tftp_engine,tftp_file,tftp_lib,tftp_sup
          ]},
{registered,[inets_sup, httpc_manager]},
{applications,[kernel,stdlib]},
{mod,[inets_app,[[]]]}.

```

在上面的代码中，`description`是一个字符串，显示为`application:which_application/0`函数调用的结果。`vsn`属性是表示应用的版本的字符串。这应该和应用目录的后缀相同。在一些更大型的构建系统中，应用的版本一般通过在提交你的代码时执行私有的脚本自动更新。

`modules`标签列出所有属于这个应用的模块。列出它们的目的有两个：第一个是确定在构建系统的时候它们都存在，并且没有和其他任何应用的名字冲突；第二个是可以在启动或者加载应用的时候加载它们。对于每一个模块都应该有一个对应的`beam`文件。为了确保注册名没有与其他应用冲突，我们在这个字段列出所有已注册进程。当创建你的`boot`文件时，发行工具会检测模块和注册进程名是否冲突。我们会在下一节中看到`boot`文件。单单只是把它们包含在应用的资源文件中并没有用，除非使用那些工具。

大部分的应用都必须在它们依赖的其他应用之后启动。如果包含在`app`文件里的`application`列表中的应用还没有启动，你的应用也不会开启。`kernel`和`stdlib`是所有其他应用都依赖的基本的标准应用。除此之外，一些特定的依赖性取决于应用自身的特性。

最后，`mod`参数是一个元组，包含了回调模块和传递给`start/2`回调函数的参数。

环境变量也许对`Inets`应用没有必要，但对一般的应用却非常重要。`env`标签说明了一个关键字的值的元组列表，在应用中可以使用下面的函数调用来存取这个列表：

```

application:get_env(Tag)
application:get_all_env().

```

要存取属于其他应用的环境变量，只需在两个函数调用中加入应用的`Name`，如下所示：

```

application:get_env(Name,Tag)
application:get_all_env(Name).

```

我们的移动用户服务数据库的应用资源文件`usr.app`将包含四个模块、两个已注册进程和对`stdlib`以及`kernel`应用的依赖。让我们也为环境变量中的`Dets`表加上文件名：

```
{application, usr,
  [{description, "Mobile Services Database"},
   {vsrn, "1.0"},
   {modules, [usr, usr_db, usr_sup, usr_app]},
   {registered, [usr, usr_sup]},
   {applications, [kernel, stdlib]},
   {env, [{dets_name, "usrDb"}]},
   {mod, {usr_app, []}}}]}
```

开始和结束应用

你可以使用下列命令来开始和结束应用：

```
application:start(ApplicationName).
application:stop(ApplicationName).
```

在上面的代码中，ApplicationName是一个基元，它代表应用的名字。

应用控制器加载属于这个应用的环境变量，并且通过一组回调函数启动高层次的监控进程。当调用start/1时将激发在应用回调模块中的start(StartType, Arguments)函数。StartType通常为基元normal，但是，如果你处理的是分布式应用（注2），也许会碰到start类型为takeover和failover。Argument是任何有效的Erlang数据类型的值，它和回调模块一起在应用资源文件中定义。

Start必须返回元组{ok, Pid} 或 {ok, Pid, Data}。Pid表示顶层监控进程的进程标识符。Data是一个有效的Erlang数据类型，用于储存在终止应用时的数据。

如果你结束应用，那么将会给顶层监控进程发送一个关闭消息。这会导致它所有的子进程以与启动相反的顺序结束，并且通过监控树传播退出路径。一旦监控树终止，应用的回调模块里回调函数stop(Data)会被调用。Data最初以start/2回调函数的{ok, Pid, Data}结构返回。如果你的start/2函数没有返回任何数据，那就忽略这个参数。如果你一定要在终止监控树前调用一个回调函数，那么可以在你的回调模块中导出函数prep_stop(Data)。

那么，具备了所有先前介绍的信息，你如何将你的usr服务器数据库包裹为一个应用呢？目录结构会是什么样的？app文件的内容会是什么？

让我们从应用的回调文件开始，先导出start/2和stop/1函数：

```
-module(usr_app).
-behaviour(application).
-export([start/2, stop/1]).
```

注2： 在本章中我们并不会涉及分布式应用。有关它们的更多信息，你需要参考OTP文档。

```

start(_Type, StartArgs) ->
    usr_sup:start_link().
stop(_State) ->
    ok.

```

就像你看到的一样，应用回调模块相对简单。尽管我们还没有在我们的例子中完成它，但是把监控进程和应用行为包组合成一个整体并不常见。你可以使用两个紧挨着的`-behavior`命令，如果与回调函数没有冲突，那么编译器将不会产生任何警告。

这里留下了一个`usr.erl`模块中需要做的微小的改变，在模块里我们调用`start_link/0`来读取环境变量：

```

start_link() ->
    {ok, FileName} = application:get_env(dets_name),
    start_link(FileName).

```

所有这些都就位了，剩下的部分就是我们的应用目录结构了，把相关的文件放在那里：

```

usr-1.0/src/usr.erl
    usr_db.erl
    usr_sup.erl
    usr_app.erl
    /ebin/usr.beam
    usr_db.beam
    usr_sup.beam
    usr_app.beam
    usr.app
    /priv/
    /include/usr.hrl

```

编译所有的模块，并用它们来进行一轮测试。把beam文件移动到ebin目录，通过告诉系统它们的路径，确保它们可被存取。你可以在Erlang开始的时候用`erl -pa Dir`命令来完成它，也可以在内核中直接使用`code:add_path(Dir)`。

在下面的交互中，我们开启了一个应用，并且在停止它之前在客户端设置上运行了一些操作。这么做的同时，我们检查到`supervisor`和`gen_server`进程已经不存在：

```

1> code:add_path("usr-1.0/ebin").
true
2> application:start(usr).
ok
3> application:start(usr).
{error,{already_started,usr}}
4> usr:lookup_id(10).
{error,instance}
5> application:get_env(usr, dets_name).
{ok,"usrDb"}
6> application:stop(usr).

=INFO REPORT==== 27-Jan-2009::22:14:33 ===
    application: usr

```

```
        exited: stopped
        type: temporary
ok
6> whereis(usr_sup).
undefined
```

注意我们如何从系统环境中读取`dets_name`的环境变量。在`usr`例子中，我们从应用里调用函数，结果是，我们不必指定定义应用的名字。浏览应用模块的手册，尝试读取应用环境变量的不同选项，以便更好地理解哪些是可用的。

应用监控器

应用监控器是一个工具，提供所有运行中应用的一个概括。通过`appmon:start()`调用启动它，你会看到在所有分布式节点上运行的全部应用的一个列表，不同的菜单允许你操作节点和显示。左边的条栏显示的是监视下的节点负载（见图12-3）。

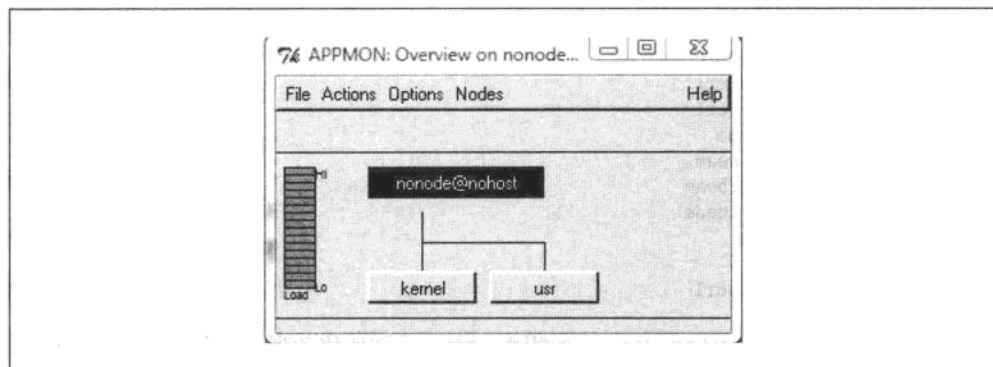


图12-3：应用的监控器窗口

在图12-3中，注意`stdlib`应用为何不显现出来。只显示包含一棵监控树的应用。双击应用会打开一个新窗口，用于展示它的监控树的视图（见图12-4）。菜单和按钮允许你操作不同的进程。链接到`usr_sup`的顶层进程是应用控制器的一部分。它们是那些启动、监控和停止顶层监控进程的进程。

版本发行的处理

从行为中可以看出，我们已经创建了一个监控树。监控树封装在一个应用程序中，它能作为一个实体被加载、启动和停止。Erlang系统由一系列松散耦合的在发布（release）文件中指定的应用程序的集合组成。这包括你已经运行过的Erlang基本安装包。从你的Erlang的根目录进入发行目录，随后是发行的子目录之一。我们的例子中采用的是`R12B`（见图12-5）。

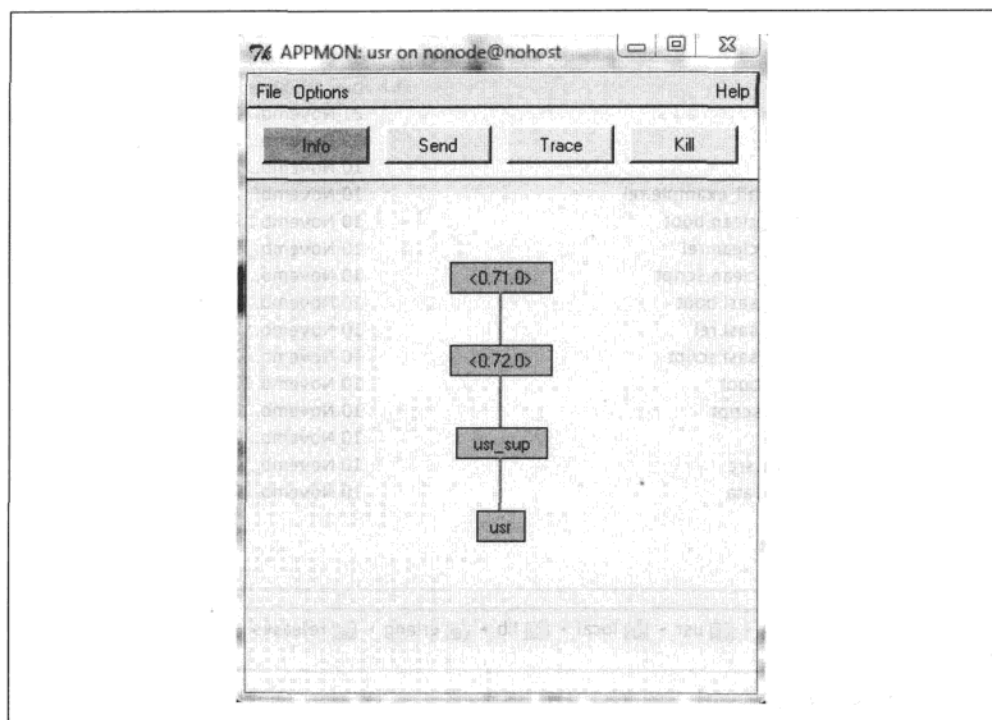


图12-4：应用程序监视器中的监控树

在里面你会发现以`rel`为后缀的发行文件的一个列表，如图12-5所示。选择`start_clean.rel`并查看：

```
{release, {"OTP ?APN 181 01", "R12B"}, {erts, "5.6.2"},
 [{kernel, "2.12.2"},
 {stdlib, "1.15.2"}]}.
```

它由第一个元组组成，其中这个元组的第一个元素是`release`标签，第二个元素是一个由发行版的名称和发行版本号组成的元组。第三个元素是一个含有Erlang运行时系统版本的元组。元组中的最后一个元素是一个应用及其版本号的列表，并以它们启动的顺序来定义。

列表中的每个应用都指向一个应用资源文件。当你调用函数`sys_tools:make_script(Name, Options)`的时候，将会读取并检查这些应用文件。检查模块和注册进程名字的冲突，如果一切正常，就会生成`Name.boot`和`Name.script`文件。

`Name.boot`是一个二进制文件，它包含用来加载应用模块和启动顶层监控进程的指令。`Name.script`文件是它对应的二进制版本的一个文本版本。`Options`参数是一个列表，其中包括最重要的数据`{path, [Dir]}`，它描述了代码服务器不知道的所有指示应用的`ebin`

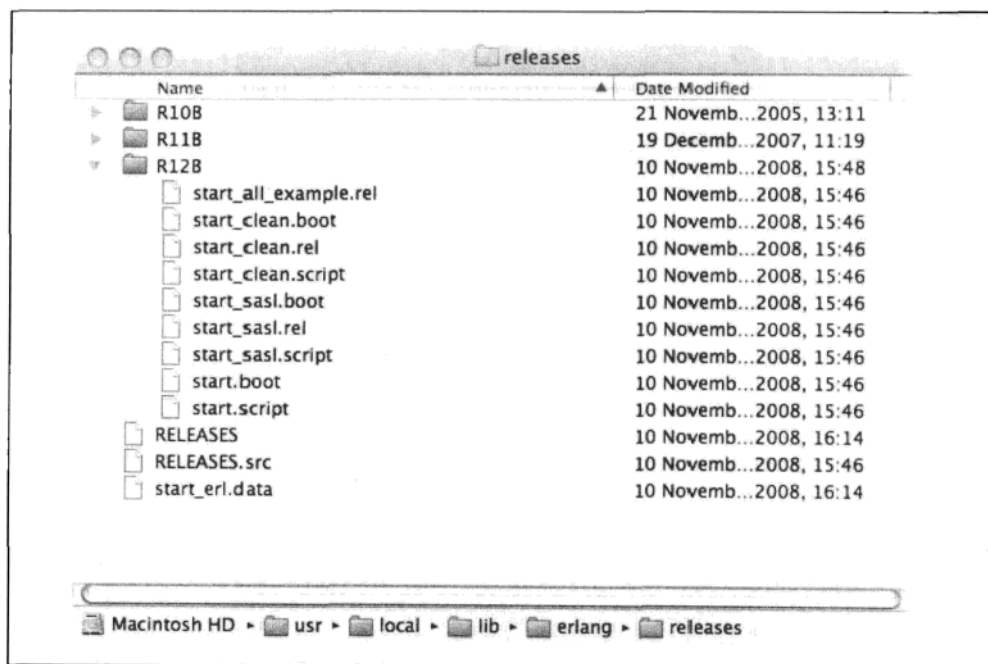


图12-5：发布版本目录

目录的路径。local指令是make_script/2另一种选择，它表示不应该假设所有应用都会在Erlang根目录的lib下找到。如果你想把Erlang的安装和应用程序分开，后者很有用。

在一个经过部署的系统中，你只能发布与你的系统有关的应用程序。这些应用包括了你的应用和你所需要的OTP发行版本的一个子集。它们应该存储在Erlang根目录的lib目录下，但你可以很容易通过在代码服务器中加入代码搜索路径来替代这个建议。在我们的例子中实际上是使用选项列表中的本地指令来覆盖它。

当启动文件已经创建的时候，你就可以使用以下命令启动你的系统：

```
erl -boot Name
```

这可以确保加载boot文件中指定的所有模块和应用程序，并且正确启动应用和它们各自的监控树。启动时如果有任何失败，那么都不会启动Erlang节点。

移动用户数据库的发行文件usr.rel将包括kernel和stdlib，这两个是任何OTP发行版本都必需的应用，其中还包括我们的usr应用的1.0版本：

```
{release, {"Mobile User Database", "R1"}, {erts, "5.6.2"},
 [{kernel, "2.12.2"},
  {stdlib, "1.15.2"},
  {usr, "1.0"}]}
```

因为在这个例子里使用了终端，前面我们已经在里面添加了usr-1.0/sbin的路径，因此创建了一个和现有的代码路径匹配的boot文件。如果我们没有在终端中设置路径，那么将不得不把选项{dir, ["usr-1.0/ ebin"]}]加入到发行文件中：

```
7> systools:make_script("usr", [local]).
ok
8> ls().
usr-1.0
usr.boot
usr.rel
usr.script
usrDb
```

我们现在可以使用“erl -boot usr”来启动系统。

注意：当创建boot文件时看一看生成的usr.script文件。我们不会在本书中解释它，因为它的多数指令其实是相当直接的。你可以编辑这些文件，通过调用systools:script2bootfile/1生成一个新的boot文件。

想一想OTP的早期开拓者。在1996年它的第一个版本中，脚本文件必须手动生成，因为当时还没有实现make_script/2。

其他行为包和更多阅读资源

本章的描述应当覆盖了您使用OTP行为包的时候会碰到的大多数情况。但是，当您使用通用服务器、监控进程和应用程序的时候，您可以了解得更详细。我们没有具体讨论而只是简要介绍的行为包，包括有限状态机、事件句柄和特殊进程。所有这些行为包库都有手册可供您参考。此外，Erlang文档有一节专门描述了OTP的设计原则，并提供了更多的细节和这些行为包的一些例子。

有限状态机是电信系统的一个重要组成部分。在第5章中，我们介绍了把一部手机用一个有限状态机建模的想法。如果手机未使用，它就处于闲置状态。如果有呼叫传入，它就会进入振铃状态。这并不一定必须是一个手机，它可以是ATM交叉连接或者在协议栈中的数据处理。gen_fsm模块提供了一个有限状态机行为，您可以使用它来解决这些问题。把状态定义为回调函数，它返回一个包含下一个状态和更新过的循环数据的元组。您可以同步或异步地发送事件给这些状态。有限状态机的回调模块也应该导出标准的回调函数，比如init、terminate和handle_info。由于gen_fsm是一个标准的OTP行为包，所以它可以链接到监控树。

事件句柄和管理器是在gen_event库模块中实现的另外的行为包。它的想法是创建一个集中点，用于接收特定类型的事件。事件可以同步和异步地传送，并附带当收到它们时

预定义的一系列行动。对事件可能的响应，包括在文件中记录它们，以短信的形式发送一个警报或者收集统计信息。每个行动在一个单独的回调模块中定义，带有自己的LoopData，它们将在调用之间被保持。针对每一个具体的事件管理器，句柄可以添加、删除或更新。因此，在实践中，针对每个事件管理器会有很多回调模块，这些回调模块的不同实例可能存在于不同的管理器中。

有时你想添加到监控树的进程，可能不是一般的OTP行为包。这可能出于效率的考虑，其中你使用了纯粹的Erlang代码来实现进程。你可能想把那些在OTP开发之前就存在的遗留代码加入监控树里，或者你可能抽象出一个设计模式并实现你自己的行为包。

编写自己的行为包是非常直接的。主要的区别在于你如何生成进程和处理系统调用。你应该使用proc_lib库生成进程，它可以导出spawn和start函数。使用proc_lib函数存储进程的启动数据，提供了同步启动进程的方法，并在异常终止时产生错误报告。要和OTP兼容，进程需要处理系统消息和事件，产生SYS库模块的循环控制。它们还必须与其父进程相连，如果它们设置捕获退出，父进程终止的时候它们也要终止。你可以在sys和proc_lib库模块中找到编写自己的OTP行为包的更多信息。

练习

练习12-1：重看数据库服务器

使用gen_server行为包模块，重写在第5章练习5-1。使用lists后台数据库模块，在循环数据中保存你的列表。你应该通过函数接口来注册服务器并访问其服务。my_db_gen.erl模块中的导出函数应包括以下内容：

```
my_db_gen:start() => ok.  
my_db_gen:stop() => ok.  
my_db_gen:write(Key,Element) => ok.  
my_db_gen:delete(Key) => ok.  
my_db_gen:read(Key) => {ok,Element}|{error,instance}.  
my_db:match(Element) => [Key1, ..., KeyN].
```

提示：如果你正在使用Emacs或者Eclipse，请使用gen_server模板：

```
1> my_db:start().  
ok  
2> my_db:write(foo, bar).  
ok  
3> my_db:read(baz).  
{error, instance}  
4> my_db:read(foo).  
{ok, bar}
```

```
5> my_db:match(bar).  
[foo]
```

练习12-2：监控数据库服务器

实现一个监控进程，用来启动和监控练习12-1中的`gen_server`。你的监控进程应该能够每小时处理5次崩溃。你的子进程应该是持久的，并给予至少30秒的终止时间，因为它可能需要一些时间来关闭一个巨大的Dets文件。

练习12-3：数据库服务器作为一个应用程序

在一个应用中封装练习12-2中的监控树，设置正确的目录结构，并利用应用资源文件来完成。



第13章

Mnesia介绍

设想一个Erlang节点的集群，转发请求到分布式的五六台电脑上。数据必须是跨越集群易于访问和最新的，而且数据库的破坏性操作必须在事务中执行，以避免竞争条件导致的数据不一致性，即使这种情况很少发生。你需要能够在运行时添加和删除节点，并且提供持续性以确保从所有可能的故障情况中快速恢复。

解决的方案是把ETS和Dets表的高效和简洁与Erlang分布合并起来，并在顶层添加一个事务层。这个解决方案称为Mnesia，它是一个功能强大的数据库，作为标准Erlang分布包的一部分发布出来。Mnesia是Claes Wikstroem（我们也称呼他为Klacke，注1）在爱立信的计算科学实验室工作那段时间的想法。Håkan Mattsson最终接手，并通过产品化和增加大量新功能把Mnesia提高到一个新水平。

Mnesia可以根据你的愿望变得容易或复杂。本章目标是向你介绍Mnesia及其功能，而不让你迷失在太多的细节中。

何时使用Mnesia

Mnesia最初是开发集成在带有高可靠性需求的分布式、大规模并发的软实时系统（如电信行业）等中。如果你的系统有如下的需求，你可以采用Mnesia：

- 针对潜在的复杂数据，进行快速键-值查找；
- 跨越节点集群分布和复制数据，并支持位置透明；
- 支持带有快速数据访问的数据持续性；
- 表位置和表特性的运行重配置；
- 支持事务处理，跨越节点合理的分布式集群；

注1：我们同样要感谢Klacke提供的ASN.1编译器、第一代的垃圾收集器、ETS、Dets、Erlang分布式、位语法和YAWS。我相信，他会很惊喜地收到你的错误报告。

- 数据索引；
- 与典型Erlang系统一样水平的容错性；
- 把你的数据模型紧密耦合到Erlang的数据类型和Erlang本身；
- 没有软实时限定时间的关系查询。

如果你的系统有如下需求，那么你就不要使用Mnesia：

- 简单键-值查找；
- 巨大的二进制的存储介质，如图片或音频文件；
- 一个持续的日志；
- 一个必须存储GB级数据容量的数据库；
- 永远不会停止增长的大型数据档案。

对于简单键-值查找，你可以使用ETS表或dict库模块。对于二进制的大容量文件，你可以更好地把每个记录与相关的各种文件脱离；为了处理核查和跟踪日志，disk_log库模块应该是你的第一选择。

如果你正在寻找存储未来的Web 2.0社会网络杀手级应用，而且该应用必须日夜不断地扩充到数亿用户，那么Mnesia可能不是你的正确选择。对于海量数据记录，并且要能够随时访问，使用CouchDB、MySQL、PostgreSQL或者Berkeley DB等可能会更好一些，它们都有大量的开源Erlang的驱动程序和API可用。一个Dets表的上限是2GB。这意味着，如果存储类型是光盘，那么一个Mnesia表的上限是2GB。对于其他的存储类型，上限视系统架构而定。在32位系统中，上限是4GB (4×10^9 字节)，而在64位系统是16EB (16×10^{18} 字节)。如果你需要存储更大的数据量，那么你将不得不把你的表进行分段，并尽可能把它们发布到若干个节点上。Mnesia具有对分段表的内置支持。

虽然Mnesia可能不是你的所有Web 2.0用户数据的首选，但它是缓存所有用户会话数据的最佳选择。一旦用户登录，数据就可以从持久存储介质读取，并且为了备份跨越计算机集群进行复制。当你检索用户数据时，登录可能需要较长的时间，但一旦完成，所有的会话活动将会非常快速。当用户注销或者会话过期时，你可以删除该记录并更新持久性数据库中的用户配置文件。

如同已经讲过的，Mnesia因为处理数千万用户的实时数据而出名。由于它非常快速且可靠，因此从操作和维护的角度看，使用正确的设置将带来极大的好处。现在设想一下，你的应用程序数据库伴随外部API的数据格式化和解析一起的胶水代码逻辑，所有这些都运行在相同的内存空间，并由一个Erlang系统统一控制。你的应用程序变得不仅有效而且便于维护。

配置Mnesia

Mnesia封装为一个OTP应用程序。要使用它，你通常需要创建一个空模式并存储在磁盘上。但你也可以把Mnesia用作一个内存数据库，它只把模式保存在内存中。创建了一个模式之后，你就需要启动Mnesia并且创建表。一旦创建好了表，你就可以读取和操作自己的数据。你需要创建一次自己的模式和表，通常是在安装你的系统的时候进行。当你完成了这一切，就可以连同Mnesia一起启动系统，所有持久数据就可以使用了。

设置模式

模式是一个描述数据库的表定义聚合。它涵盖了你存储于内存、磁盘或两者皆有的表，还有其配置特征和它将包含的数据的格式。这些特征可能随节点不同而不同，因为你可能希望你的表在操作和维护节点上具有它的磁盘拷贝，而在事务节点上只具有内存拷贝。在Erlang中，模式是以持久Mnesia表存储的。在配置数据库时，你可以创建一个空模式表，随着时间推移，它随着你的表定义的增加而增加。

为了创建模式（注2），启动分布式Erlang节点并和它们连接。如果你不希望分发Mnesia，那么只需启动一个非分布式Erlang节点。在此之前，有一点是很重要的，即要确保没有旧模式存在，并且确保Mnesia尚未启动。

在接下来的例子中，我们将讲解两个节点switch和om上的数据库：

```
(om@Vaio)1> net_admin:ping(switch@Vaio).
pong
(om@Vaio)2> nodes().
[switch@Vaio]
(om@Vaio)3> mnesia:create_schema([node()|nodes()]).
ok
(om@Vaio)4> ls().
Mnesia.om@Vaio      Mnesia.switch@Vaio  include
lib
ok
```

命令`mnesia:create_schema(Nodes)`只能在相互连接的节点之一上运行。通过建立列表`[node()|nodes()]`，我们得到所有连接的Erlang节点，它们碰巧就是我们要创建模式表所在的相同节点。你可能还记得在第11章中我们讨论了分布式Erlang，内置函数`node()`返回了本地节点，而`nodes()`返回了所有其他连接节点。命令`mnesia:create_schema/1`会自动传播到其他节点。

注2：那些熟悉数据库的人可能会奇怪，我们为什么称为“模式创建”，因为它把数据库本身的创立形象地类比为后续创建表的表操作隐含地创建模式，其中的模式也就是数据库所包含的表的细节。

在创建模式时，每个节点都将创建一个目录。就我们的情况而言，由于两个分布式Erlang节点共享同一个根目录，它们的模式目录Mnesia.om@Vaio和Mnesia.switch@Vaio将会出现在同一个位置，其中Vaio是我们执行命令的计算机主机名。该目录的其他内容则取决于你先前对Erlang所做的工作，但不会影响你的模式。

此时计算机上的节点还没有被连接，那么仅仅是该计算机所运行的节点的模式目录被创建。如果你不打算在分布式环境中运行Mnesia，那么模式目录名称将是Mnesia.nonode@nohost。只是把[node()]作为参数传递到create_schema/1进行调用。

通过下列指令启动Erlang，你也可以覆盖根目录的位置：

```
erl -mnesia dir Dir
```

其中用你所要存储的模式目录取代Dir。

开始Mnesia

一旦创建你的模式，可以通过以下命令启动应用程序：

```
application:start(mnesia).
```

如果你使用的是第12章所讲的启动脚本，那么你应该在发布文件中包括Mnesia应用程序。在测试环境中，如果你不使用OTP行为包，也可以使用mnesia:start()。在带有发布处理的工业项目中，一般认为，分隔应用程序并单独启动它们是最佳的做法。

如果你启动没有模式的Mnesia，那么将会创建仅限于内存的数据库。重新启动后它们将不复存在，所以每次启动系统，你都必须重新创建仅限于内存的表。要启动具有内存模式的Mnesia，你需要做的就是确保特定的节点不存在模式目录，然后就可以启动Mnesia。

通过调用application:stop(mnesia)或mnesia:stop()，你可以停止Mnesia。

Mnesia表

Mnesia表包含Erlang记录。在默认情况下，该Erlang记录类型的名称就成为表的名称。通过使用下列函数的调用，你可以创建一个表：

```
mnesia:create_table(Name, Options)
```

在此函数调用中，Name是记录类型，而Options是一个格式为{Item, Value}元组的列表。下列的Item和Value都是最常用的：

`{disc_copies, Nodelist}`

提供节点列表，其中含有你所需光盘和内存的表副本。

`{disc_only_copies, Nodelist}`

`Nodelist`包含你仅想要光盘的特定表副本的节点。这通常是备份节点，因为在这些节点上本地内容读取将是缓慢的。

`{ram_copies, Nodelist}`

指定你想拥有特定表的备份内存副本的节点。此属性的默认值为`[node()]`，所以省略它将会创建本地Mnesia内存副本。

`{type, Type}`

描述这个表是集合、有序集合还是袋。默认值设置为集合。

`{attributes, AtomList}`

是一个代表所谓记录字段名称的基元列表。它们主要用于索引或使用查询列表解析。请不要把它们硬编码，可以使用函数`record_info(fields, RecordName)`调用来生成它们。

`{index, List}`

是一组属性（即记录字段名称列表），每当访问表中元素时，它们就可作为辅助键。

在默认情况下，键的位置在该记录的第一个元素。在Mnesia表中，每一个记录的实例称为对象（object）。对象的键与表名一起作为对象标识符（object identifier）。

一旦创建了模式，我们就要在所有节点上启动Mnesia，进行一次性创建表的操作。这通常是在安装和配置Erlang节点时进行的。出于备份和性能方面的原因，我们想在两个叫做om和switch的远程节点上运行数据库。在节点om上，我们想要以内存和硬盘方式存储数据，而在节点switch上，我们只希望维持一个内存副本。当查看这个例子时，请记住终端命令`rr/1`，它读取一个文件并提取其记录的定义，使得在终端中可以使用这些定义：

```
(om@Vaio)5> rr(usr).
[usr]
(om@Vaio)6> Fields = record_info(fields, usr).
[msisdn,id,status,plan,services]
(om@Vaio)7> application:start(mnesia).
ok
(om@Vaio)8> mnesia:create_table(usr, [{disc_copies, [node()]},
    {ram_copies, nodes()}], {type, set}, {attributes, Fields}, {index, [id]}].
{atomic,ok}
```

在这个Mnesia例子中，请注意我们是如何只创建了一个表，并表明它已使用了索引和

必须有内存副本和文件备份。简单说明一下在这个场景背后发生了什么。很明显，在移动用户数据库的后端模块采用了ETS和Dets表，我们在那里创建了三个表：磁盘副本的表、内存副本的表和索引内容的表。

无须打开Mnesia表。当启动Mnesia应用程序时，将创建或打开在模式中所配置的一切表。这是一个相对快速且无阻塞的操作，与其他应用程序的启动平行完成。

对于大型持久表，或者被错误关闭的表而且其备份文件需要维护，即使没有正确加载它们，其他应用程序也可能会尝试访问。如果发生这种情况，那么该进程将以错误no_exists崩溃。为了避免这种情况，你应该调用：

```
mnesia:wait_for_tables(TableList, TimeOut)
```

在你的进程或OTP行为包的初始化阶段，其中TableList是该进程所使用的Mnesia表的名字的列表（包括持久性和易变性），而TimeOut是基元infinity或以毫秒为单位的整数。

在处理包含数百万行记录的大型表时，如果你没有使用infinity作为超时，那么你必须确保TimeOut值至少是几分钟，如果不是针对巨大的、分段的和基于磁盘的表所需的几个小时的话。需要这些表的进程独立于Mnesia的启动来调用wait_for_tables/2。通过模式匹配返回值，可以加载确保该表。如果发生超时，那么返回值将导致一个错误匹配，并被记录。你最不想遇到的事情是调用wait_for_tables/2而返回{timeout, TableList}，忽略此值，并继续假定已正确地加载该表。

在移动用户的例子中，最初我们需要一个进程，由它创建和拥有ETS和Dets表并且序列化所有破坏性（写入和删除）操作。事实并非如此。Mnesia将为我们处理此事。因此，我们将丢弃模块usr_db.erl，并且用usr.erl模块代替所有功能。我们也将删除所有与进程相关的程序，例如start、spawn、init和stop，或者如果我们采用行为包例子，那么就删除gen_server调用和回调。相反我们将增加调用create_tables/0和ensure_loaded/0，并保留全部客户端API：

```
-module(usr).
-export([create_tables/0, ensure_loaded/0]).
-export([add_usr/3, delete_usr/1, set_service/3, set_status/2,
         delete_disabled/0, lookup_id/1]).
-export([lookup_msisdn/1, service_flag/2]).

-include("usr.hrl").

%% Mnesia API

create_tables() ->
    mnesia:create_table(usr, [{disc_copies, [node()]}], {ram_copies, nodes()}),
```

```

{type, set}, {attributes, record_info(fields, usr)},
{index, [id]})}.
ensure_loaded() ->
ok = mnesia:wait_for_tables([usr], 60000).

```

事务处理

由于许多可能位于不同节点的并发进程可以同时访问和操作对象，你需要保护数据避免竞争条件。为了做到这一点，可以用Fun封装这些操作，并且在事务中执行它们。事务处理保证数据库从一个一致状态转换到另一个一致状态，这些遍及所有节点的改变是持久和原子性的，并且并行运行的事务不会相互干扰。在Mnesia中可以使用下列命令来执行事务：

```
mnesia:transaction(Fun)
```

其中Fun包含了如读取、写入和删除操作。如果操作成功，那么该调用将返回一个元组{atomic, Result}，其中Result是Fun中执行的最后一个表达式的返回值。如果该事务失败，那么就返回{aborted, Reason}。总是模式匹配{atomic, Result}，因为你的事务永远不应该失败，除非从事务内调用mnesia:abort(Reason)。

除了你的Mnesia操作外，确保这些Fun没有边界效应。当在事务中执行Fun时，Mnesia将锁住被操控的对象。如果对于一个对象另一个进程持有一个与之冲突的锁，那么该事务首先将释放它目前所有的锁，然后重新启动。边界效应如一个io:format/2调用或发送一条消息，可能导致打印输出或消息发送被执行上百次。

写入操作

为了在表中写入对象，你需要使用函数mnesia:write(Record)，把它封装到Fun中，并且封装在事务中执行。该调用返回基元ok。Mnesia会加一把写锁到此对象的所有副本（包括远程节点上的）。试图给已锁定的对象加锁将会失败，这将促使事务释放所有的锁并且重新开始。

在终端中直接尝试这些函数，这将会让你更好地感受它是如何工作的：

```

(om@Vaio)9> Rec = #usr{msisdn=700000003, id=3, status=enabled,
(om@Vaio)9>          plan=prepay, services=[data,sms,lbs]}.
#usr{msisdn=700000003, id=3, status=enabled,
      plan=prepay, services=[data,sms,lbs]}
(om@Vaio)10> mnesia:transaction(fun() -> mnesia:write(Rec) end).
{atomic,ok}

```

还记得第一次出现在第10章中这个例子的ETS和Dets的版本，你是如何不得不为每个插

入到数据库的用户建立三个表的记录吗？还有如果你想在多个节点上分布这些数据呢？由于usr Mnesia表包含分布式内存和基于磁盘的副本以及索引，所以你只需要做一次写入操作。在幕后Mnesia为你完成其余的工作。

在移动用户的例子中要写入或更新记录，你需要在usr.erl模块中封装写操作，具体如下：

```
add_usr(PhoneNo, CustId, Plan) when Plan==prepay; Plan==postpay ->
    Rec = #usr{msisdn = PhoneNo,
               id      = CustId,
               plan    = Plan},
    Fun = fun() -> mnesia:write(Rec) end,
    {atomic, Res} = mnesia:transaction(Fun),
    Res.
```

由于在前面的ETS和OTP行为包例子中，add_usr/1返回ok，你将可以保证这些新函数向后兼容。

读取和删除操作

为了读取对象，你可以使用函数mnesia:read(ObjId)，其中ObjId是格式为{TableName, Key}的对象标识符。如果该对象不存在，那么这个函数调用将返回空列表；或者如果该对象存在且表是集合或袋，那么返回一个或多个记录列表。你需要在事务范围内执行该函数。若没有这样做将会导致运行时错误。

注意我们正从哪个节点读取记录。这没什么关系。我们只需要确保在这个节点上的Mnesia已经启动，并且当创建表时也是如此。

要删除一个对象，你可以在事务中使用mnesia:delete(ObjId)调用。该调用返回基元ok，不管该对象是否存在。

我们的模式假设分布在两个节点上。让我们启动第二个节点上的Mnesia，并且在前面例子中的节点om上查看一下我们所写入的记录：

```
(switch@Vaio)1> application:start(mnesia).
ok
(switch@Vaio)2> usr:ensure_loaded().
ok
(switch@Vaio)3> rr(usr).
[usr]
(switch@Vaio)4> mnesia:transaction(fun() -> mnesia:read({usr, 700000003}) end).
{atomic,[#usr{msisdn = 700000003,id = 3,status = enabled,
              plan = prepay,
              services = [data,sms,lbs]}]}
(switch@Vaio)5> mnesia:read({usr, 700000003}).
```

```

** exception exit: {aborted,no_transaction}
    in function mnesia:abort/1
(switch@Vaio)6> mnesia:transaction(fun() -> mnesia:abort(no_user) end).
{aborted,no_user}
(switch@Vaio)7> mnesia:transaction(fun() -> mnesia:delete({usr, 700000003}) end).
{atomic,ok}
(switch@Vaio)8> mnesia:transaction(fun() -> mnesia:read({usr, 700000003}) end).
{atomic,[]}

```

正如你所看到的，执行破坏性的操作如写入或删除，将会在所有节点上重复该操作。注意第5行命令的错误，在那里我们在事务范围以外执行一次读取。另外，从第6行命令的返回值可以看出，在那里我们中止了一次事务。

索引

当创建usr表时，我们传递给调用的选项之一就是元组{index, AttributeList}。这将为该表建立索引，允许我们使用在AttributeList中所列出的任何一个辅助字段（或键）来查找和操作对象。要使用索引，你必须执行以下调用：

```
index_read(TableName, SecondaryKey, Attribute).
```

对我们这个实例中为客户提供数据的所有函数使用属性CustomerId。如果你想删除订户记录，那么你就必须检查它是否存在。如果这个字段不存在，那么函数delete_usr/1返回{error, instance}。如果该记录存在，那么你可以找到它的主键并用它删除该字段：

```

delete_usr(CustId) ->
  F = fun() -> case mnesia:index_read(usr, CustId, id) of
    [] -> {error, instance};
    [Usr] -> mnesia:delete({usr, Usr#usr.msisdn})
  end
end,
{atomic, Result} = mnesia:transaction(F),
Result.

```

采用类似的方式，如果你想添加或删除订户有权使用的服务，那么你可以查看usr记录，如果这个条目存在，那么就使用msisdn更新其状态：

```

set_service(CustId, Service, Flag) when Flag==true; Flag==false ->
  F = fun() ->
    case mnesia:index_read(usr, CustId, id) of
    [] -> {error, instance};
    [Usr] ->
      Services = lists:delete(Service, Usr#usr.services),
      NewServices = case Flag of
        true -> [Service|Services];
        false -> Services
      end,
      mnesia:write(Usr#usr{services=NewServices})
    end
  end
end

```

```

        end
    end,
    {atomic, Result} = mnesia:transaction(F),
    Result.

```

同样的原则也适用于启用和禁用指定的订户：

```

set_status(CustId, Status) when Status==enabled; Status==disabled->
    F = fun() ->
        case mnesia:index_read(usr, CustId, id) of
            [] -> {error, instance};
            [Usr] -> mnesia:write(Usr#usr{status=Status})
        end
    end,
    {atomic, Result} = mnesia:transaction(F),
    Result.

```

注意首先所有这些函数是如何查看对象，并且如果存在，那么它们可以删除或操纵它。当用户改变状态或服务时，任何其他进程在index_read/3和write/1函数调用之间来删除记录就没有风险。这是因为这两个操作是在同一个事务中运行的，在它们正在操纵的对象上设置了锁，并且确保搁置其他试图访问这些对象的事务。因此，在同一个对象上的任何两个事务之间不能互相干扰。需要记住的是，竞争条件可能在不同节点的进程之间发生。让我们在终端上尝试刚刚定义的函数，看看它们是如何工作的：

```

(switch@Vaio)9> usr:add_usr(700000001, 1, prepay).
ok
(switch@Vaio)10> usr:add_usr(700000002, 2, prepay).
ok
(switch@Vaio)11> usr:add_usr(700000003, 3, postpay).
ok
(switch@Vaio)12> usr:delete_usr(3).
ok
(switch@Vaio)13> usr:delete_usr(3).
{error,instance}
(switch@Vaio)14> usr:set_status(1, disabled).
ok
(switch@Vaio)15> usr:set_service(2, premiumsms, true).
ok
(switch@Vaio)16> mnesia:transaction(fun() -> mnesia:index_read(usr, 2, id) end).
{atomic,[#usr{msisdn = 700000002,id = 2,status = enabled,
    plan = prepay,
    services = [premiumsms]}]}

```

如果你创建一个表，并想要在运行时添加或删除索引，那么你可以使用模式操作函数add_table_index(Tab, Attribute)和del_table_index(Tab, Attribute)。

脏操作

有时在事务的范围以外而且没有设置任何锁来执行一次操作，是可以接受的。这种操作

称为脏操作（dirty operation）。在Mnesia中，脏操作的速度比在事务中执行同样操作快10倍左右，这使得它们对软实时系统成为一个可行的选择。如果你能保证你的表的一致性、隔离性、持久性和分布式特性，那么脏操作将大大提高你的程序的性能。

几个最常见的Mnesia脏操作如下所示：

```
dirty_read(ObjId)
dirty_write(Object)
dirty_delete(ObjectId)
dirty_index_read(Table, SecondaryKey, Attribute)
```

所有这些操作都将和在事务中执行同样的操作一样。如果你需要实现满足带宽要求的软实时系统，那么事务处理很快会成为一个主要瓶颈。以我们的移动用户为例，这些时间限制严格的函数是与服务相关的。如果你需要发送十万条短信，其中每个短信都需要查找以确保用户不仅存在并启用，而且准许接受优质级手机短信，那么速度就变得至关重要。如果是在脏读取之前或之后改变用户数据，那么它不会影响短信的发送，因为这些函数仅包含一个非破坏性的读操作：

```
lookup_id(CustId) ->
  case mnesia:dirty_index_read(usr, CustId, id) of
    [Usr] -> {ok, Usr};
    []      -> {error, instance}
  end.

%% Service API

lookup_msisdn(PhoneNo) ->
  case mnesia:dirty_read({usr, PhoneNo}) of
    [Usr] -> {ok, Usr};
    []      -> {error, instance}
  end.

service_flag(PhoneNo, Service) ->
  case lookup_msisdn(PhoneNo) of
    {ok, #usr{services=Services, status=enabled}} ->
      lists:member(Service, Services);
    {ok, #usr{status=disabled}} ->
      {error, disabled};
    {error, Reason} ->
      {error, Reason}
  end.
```

使用脏操作同时确保数据的一致性的一种常见方法是在一个单一进程中序列化所有破坏性操作。虽然在这个进程之外另一个进程可能被允许执行脏读取，但是通过发送请求到进程并以接收顺序来执行，所有涉及写入或删除元素的操作都是序列化的。

在Mnesia版本的usr例子中，我们一起去除了中心进程，并且使用事务。如果我们一直

保留这个进程，那么可以用Mnesia的脏操作来取代所有的ETS和Dets读、写和删除操作。如果我们把表遍及om和switch节点分布，我们将不得不把所有破坏性操作重定向到其中的一个节点，否则我们会有在两个位置上同时运行更新相同对象的风险。

如果你需要在分布式环境中使用脏操作，那么关键是要确保某些键子集的更新是通过单一节点的进程进行序列化的。如果你的键是在1~1 000的范围，那么你可以在一个节点上更新所有偶数键而在另外一个节点上更新所有奇数键，从而解决我们刚才所描述的竞争条件。

不一致的表

如果你想看看Mnesia表是如何通过使用脏操作变得不一致的，那么可以启动在不同计算机上的两个分布式Erlang节点，并且使它们共享同一个Mnesia表。在一个节点上，键入具有键和一个或多个字段的`mnesia:dirty_write/1`，但不要按Enter键。在另外一个节点上也这样做，保持相同的键，但改变字段的值。然后，你需要很快地断开两台计算机之间的网络网线，然后在两个终端上按下Enter键，再重新连接网线。如果你足够快，在节点之间的TCP/IP连接超时之前你将会重新连接网线。

读取两个节点的记录，并且你可能会发现不一致的表，例如值可能会有所不同。执行`dirty_write/1`后会发生的事情是，你在本地更新本地对象的副本，之后将其发送到远程节点。由于TCP/IP连接暂时断开，所以此记录被缓冲。这个缓冲将会在两个节点上发生，所以一旦你重新插入网线，就产生了一个竞争条件，覆盖了对等节点的记录。如果你使用了事务，那么这个竞争条件就不会发生。

分区网络

在分布式环境使用Mnesia时，最大的一个问题是分区网络的存在。虽然这个问题没有直接关系到任何分布式事务数据库，特别是Mnesia，但是迟早你一定会遇到它。假设你有共享一个Mnesia表的两个Erlang节点。如果一些像小的网络故障那样的事情发生在节点之间，并且表的两个副本彼此独立地更新，那么当网络再次恢复正常时，你就拥有了一个不一致的共享表（只要一个节点已经得到更新，Mnesia就有能力恢复）。与脏操作的例子不一样，Mnesia知道表是被分区的，并且会报告此事件以便你可以对此采取行动。

你会怎么做？你会选择两个表副本的哪一个？你能用某种方式把两个数据库合并在一起吗？从分区网络中恢复数据库，是目前还没有发现“银弹”（silver bullet）（译注1）解决方案的研究领域。在Mnesia中可以通过以下函数调用选择主节点：

译注1：喻指新技术，尤指人们寄予厚望的某种新科技。


```
mnesia:set_master_nodes(Table, Nodes).
```

如果网络变为分区的，那么Mnesia将会自动获得主节点的内容，通过以同步方式把它复制到分区的节点并且再复制回来。分区时所有在表中而不是在主节点上的更新都会丢弃。

最常见的Mnesia部署是将表复制到两个或三个节点。一旦你开始增加这个数目，那么分区网络的风险将以指数增加。无论你的测试多么宽泛，分区数据库都将很难证明自己，直到你进入真实的应用而且你的系统在高负荷情况下运行。当设计分布式数据库时，手头始终要有一个从分区数据库恢复的计划。

扩展阅读

我们几乎完成了用户数据库模块。只缺少一个操作：遍历列表并删除所有禁用用户。在Mnesia中有许多遍历和搜索数据的方法。你可以使用first和next，查询列表解析，甚至还有select和match。

我们选择了调用mnesia:foldl/3，没有特别的原因，只是因为它是一个值得一提的有趣的函数。其行为和它在lists模块同样的部分一样，它不是遍历一个列表，而是遍历一个表：

```
delete_disabled() ->
F = fun() ->
    FoldFun = fun(#usr{status=disabled, msisdn = PhoneNo},_) ->
        mnesia:delete({usr, PhoneNo});
        (_,_) ->
            ok
    end,
    mnesia:foldl(FoldFun, ok, usr)
end,
{atomic, ok} = mnesia:transaction(F), ok.
```

虽然现在我们可能已经完成了移动用户的例子，但是几乎还没有揭开Mnesia所提供的内容的神秘面纱。在工业系统中最常用的一些功能包括：表分段、备份、故障恢复、Mnesia事件和无盘节点，这里仅提到了很少的几个。所有更详细的内容都包括在Mnesia用户指南和Mnesia参考手册中，两者都是OTP文档的一部分。不过，我们所谈及的已经足够让你能够有效地开始使用Mnesia。

练习

练习13-1：设置Mnesia

在下列循序渐进的练习中，你将创建一个分布式的Mnesia数据库Muppet。

首先启动两个节点：

```
erl -sname foo
erl -sname bar
```

在第一个节点中声明Muppet的数据结构：

```
foo@localhost 1> rd(muppet, {name, callsign, salary}).
```

接下来创建一个模式以便你可以使表保持持久：

```
foo@localhost2> mnesia:create_schema([foo@localhost, bar@localhost]).
```

现在，你需要在两个节点上启动Mnesia：

```
foo@localhost 3> application:start(mnesia).
bar@localhost 1> application:start(mnesia).
```

数据库开始运行了！创建一个分布式表：

```
foo@localhost 4> mnesia:create_table(muppet, [
                                {attributes, record_info(fields, muppet)},
                                {disc_copies [foo@localhost, bar@localhost]})].
```

请注意disc_copies属性如何指定那些你想要持久保留副本的节点。检查一切是否正确：

```
foo@localhost 5> mnesia:info().
```

现在，全部检查一遍并输入你当前的Muppets例子：

```
foo@localhost 6> mnesia:dirty_write(#muppet
{name = "Francesco" callsign="HuluHuluHulu", salary = 0}).
```

看看到目前有多少个Muppet：

```
foo@localhost 7> mnesia:table_info(muppet, size).
```

列出我们本章没有涉及的函数名称，但你可以在Mnesia手册中查到它们：

```
foo@localhost 8> mnesia:dirty_all_keys(muppet).
```

太好了，现在进入其他节点，并查看Muppet：

```
bar@localhost 2> mnesia:dirty_read({muppet, "Francesco"}).
```

练习13-2：事务处理

编写一个函数读取Muppet工资，并给把它增加10%。使用一个保证不存在竞争条件的事务处理。

练习13-3：Mnesia脏操作

用Mnesia脏操作来实现第10章的usr_db.erl模块。该模块应该完全向后兼容基于ETS和Dets的解决方案。在终端上测试它，并且如果成功的话，再利用usr.erl模块在一个进程中序列化这些操作。你可以调用create_tables/1 和 close_tables/0启动和停止Mnesia应用程序，并且restore_backup/0调用会引发wait_for_tables/2调用。所有其他函数应返回和原来的例子一样的值。



图形用户界面编程wxErlang

图形用户界面编程（Graphical User Interface, GUI）并不是Erlang的明显优势之一，但正在进行的一项工作为Erlang提供了跨平台和最先进的GUI编程系统：wxErlang，它是Erlang系统绑定的wxWidgets系统。

wxWidgets是由许多C++库组成的，并提供了一系列构件，如菜单、按钮、文本和图形显示等。wxWidgets还提供了一个构建跨平台软件的通用框架，其中包括支持国际化和底层工具，例如内存管理。由于wxErlang的庞大和复杂性，本章不可能对它进行全面概述。不过，本章讲述了底层工具的基本原理以及其最常用的相关内容的一些尝试。我们所讲述的范围可以使你从零开始，并给予一定的基础知识来对该库更加深入探索。

本章将介绍wxWidgets并且解释Erlang绑定背后的一些原理。在描述wxErlang的事件处理机制后，我们将会以两个阶段展示一个微型博客的例子。本章最后将提出更多关于学习wxWidgets和wxErlang的指导性建议，以及一系列用于提高和扩展可运行例子的练习。

wxWidgets

在20世纪90年代初，wxWidgets开源项目由Julian Smart发起，而现在它由一个大约20个开发人员的小组和广泛的贡献者来提供技术支持。wxWidgets是一个C++库，但它已经能够与其他许多语言进行绑定，包括Haskell、Java、Perl和Python，为使用这些语言的编程人员提供了访问最先进的图形用户界面构建工具包直接且高级的方法。最近，由Dan Gudmundsson和Mats-Ola Persson所领导的工作，为我们提供了Erlang版本的wxWidgets。

大多数具有GUI的系统将部署到各种不同的平台（多种硬件和操作系统的组合）上。为了避免编写你所开发的系统的不同实现，你就必须使用能够运行于多个平台的GUI工具包。一种是使用与平台无关的工具包，这可以使应用程序在多种平台上具有相同的外观。这种方法的缺点是，该程序不可能为每一个平台提供应用程序本地化的外观和感受

（并且我们可以肯定的是，大家都经历过这样的使用应用程序的挫折感，即它不符合你最喜爱的UI准则）。

wxWidgets工具包不仅支持多种平台，而且它的设计支持本地化的外观和感受，包括Windows、Linux和Mac OS X等各种不同风格。这使得wxWidgets工具包特别适合与平台独立的Erlang语言一起使用。

wxWidgets具有面向对象的架构。每个图形实体是一个C++对象，并且属于一个C++类。通过多重继承，这些类和一些更复杂的图形实体如“文字输入对话框”相关联。本章的例子包括事件句柄和对话类，以及wxWindow和wxObject基类。

图形用户界面是事件驱动（event-driven）的：通过把事件和调用的（成员）函数关联，GUI对象处理事件（既有用户发起的也有内在的）。你可以使用事件表（event table）创建这样的关联，把特定的事件绑定到它们的句柄函数，并且把它们作为一个类的构造函数的一部分静态地构建。或者，你也可以创建一个动态关联，通过在一个一个对象的基础上把句柄函数连接到事件，并且允许在对象生命周期中修改这些处理。

可以在栈中创建C++的对象，但总的来说，主要的图形用户界面实体是在堆中创建的。一旦对象不能访问，任何存储GUI实体而分配到的内存都需要被显式释放。wxWidgets提供了一些协助完成这一功能的机制。

在网站<http://www.wxwidgets.org/>以及《Cross-Platform GUI Programming with wxWidgets》一书中，你可以在那些广泛的联机文档中找到关于wxWidgets的更多信息。

wxErlang：wxWidgets绑定到Erlang

为了把wxWidgets绑定到Erlang，必须确定如何用Erlang来提供它的面向对象结构和事件处理机制，以便使它尽可能紧密地和语言底层的设计原理相结合。本节将介绍相关的顶层机制，更具体的细节在随后的部分说明。

wxErlang技术文档和每个模块的EDoc页面一起提供一个概述页面，它给出了每一个函数的类型信息以及连接到wxWidgets联机文档的相应网页的链接。

对象和类型

在wxErlang绑定中，每个类由一个模块表示，而每个对象是由一个对象引用表示。例如下面的wxErlang函数调用：

```
File = wxMenu:new(),
```

它创建了一个新的wxMenu对象，并且把这个对象的引用赋给变量File，效果类似于在C++代码中调用构造函数：

```
wxMenu *File = new wxMenu;
```

要调用该方法，如以下这个C++的例子：

```
File->Append(NEW,wxT("New\\tCtrl-N"));
```

wxErlang提供了一个具有三个参数的append函数，其中append的第一个参数是对象的引用File：

```
wxMenu:append(File,?NEW,"New\\tCtrl-N"),
```

采用额外的“this”或者“self”值作为第一个值的wxErlang函数模式用于整个绑定。以类似的方式，类的构造函数由相同元数的new函数取代。一些wxWidgets函数采取了可选参数。在wxErlang里，所有的可选参数通过一个单一无序的属性列表的进行传递，如同proplists模块支持的。

这里值得重申的是，在前面的代码中File变量是固定的，它的值是一个对象的引用。另一方面，它所指的对象可变；这个变化是由wxErlang操作来完成——在这种情况下是wxMenu模块中的那些操作。

前面的“append”的例子另外还说明了两点：

- 在wxWidgets代码中，wxT构造函数把字符串封装起来，wx字符串假定都是以本地架构格式采用UTF-32编码的。在Erlang中，默认的是ASCII字符串，但针对其他字符集都将需要明确处理。
- 在wx中大量使用C++的宏机制，而不仅是在定义对象标识符时，如NEW。引入头文件wx.hrl通过Erlang预处理器机制，在wxErlang中这些仍然可用。

最后，虽然大多数wxWidgets的类有相应的wxErlang表达形式，但是还有一些表达数据类型的类被直接映射到Erlang的数据类型。例如，wxPoint由一个数对{Xcoord, Ycoord}表示，而wxGBPosition由{Row, Column}元组表示。所有数据类型对应的全部细节都保存在wxErlang文件的概述部分中。

事件处理、对象标识符和事件类型

在Erlang中的wx绑定事件有两种处理方法。它们可以被如在wxWidgets中那样的回调函数处理，或者把它们当做Erlang消息接收，这样就与Erlang的并发编程模型集成了。这里我们将讨论后面一种机制。

要了解消息的结构，首先要了解wx事件的其他三个方面内容：

标识符 (Identifiers)

标识符是用于唯一标识符GUI的组件的整数，如窗体、按钮和菜单项等。wxWidgets是包含了各种通用组件的标准标识符的一个集合，比如wxID_OPEN和wxID_ABOUT（用作文件打开菜单项和关于窗口）。作为wxErlang里wx.hrl头文件中的宏定义，这些标识符都是可用的并可以用?wxID_ABOUT来引用。

这些标准的使用值得鼓励。第一，系统可以识别与特定的标识符相关联的默认行为，第二，与某些标识符相关联的对象可以一种用平台敏感的方式来处理。在本章的后面部分，我们将会以一个博客例子说明这一点。

在wxErlang发行包的wx.hrl头文件中，你可以找到一个完整的标准标识符列表，与在wxWidgets文档或者书籍中的一样。

事件类型 (Event types)

wxErlang事件以不同的形式和大小出现：菜单选择、通过树命令的导航以及由时间的推移或者用户用鼠标或键盘完成的操作来触发的事件。这些不同种类的事件由一组基元集合表示，包括command_menu_selected、enter_window、close_window和其他一百多个称为事件类型的基元。

依赖于其类型，一个事件将由不同类型的信息与之相关联。例如，close_window将没有任何信息与之关联（除了它的类型外），然而事件发生时（在其他事件中），enter_window事件将与鼠标位置关联。这些关联信息用一个记录（record）表示，其类型取决于事件的类型。在任何情况下，记录的类型字段包含事件类型。

在wxEvtHandler.erl模块及其相关文档中，可以获得完整的事件类型定义以及关联的记录。

连接 (Connection)

一个图形组件为了接收消息，它必须和特定的事件类型关联；通过指定一系列标识符，也可以把它限定为一组特定的对象。

例如，对于连接到事件的窗体（其引用来自Frame变量），其中事件是由菜单所选择的命令产生的，你可以使用下面的表达式：

```
wxFrame:connect(Frame, command_menu_selected)
```

connect操作对于任何从wxEvtHandler继承的类都是可用的。这个实现使用了wxWidgets的动态事件句柄的连接机制。

有了这些定义，我们现在可以采取如下格式解释wxErlang消息：

```
#wx{id=Id, obj=Obj, userData=T, event=Rec}
```

其中`Id`是接收事件的图形对象的标识符，`Obj`是一个对建立连接的对象的引用（在较早的连接例子里，这应该是`Frame`对象），而`Rec`是一个包含信息的记录，它依赖事件的特定类型。在任何情况下，`Rec`的类型字段都包含了发出信息的类型。

消息的处理采用Erlang中标准的方式，因此，一个“添加”菜单项选中的消息将进行如下处理：

```
receive
  #wx{id=?APPEND, event=#wxCommand{type=command_menu_selected}} ->
    ... handler code for APPEND ...
  ... other messages ...
end
```

进一步的连接的细节包括回调函数连接的方式和多个处理程序处理事件的方式，可以在`wxErlang`文档的概述以及`wxWidgets`的在线文档和书籍中找到。

综上所述

若要使用`wxWidgets`，`wxErlang`应用程序需要启动和停止一个`wx-server`，比如这样：

```
wx:new().
...
wx:destroy().
```

各自独立创建服务器的两个进程将不能共享对象。相反，使用`wx:get_env/0`可以获取正在运行进程的环境信息，并且使用`wx:set_env/1`在一个新的进程中设置这样的环境，这样就允许进程共享对它们对象的访问。

当调用`destroy`函数时，将收回所有由`wx`应用程序使用的内存。在对象上使用函数`wxClass:destroy/1`是可以收回分配给`Class`小窗口部件的内存。这特别适合于一些临时对象，如表示一个对话框体的对象，因为这些是在`wxWidgets`系统的栈中被分配的，而不是在`wxErlang`中。

第一个实例：MicroBlog

我们将开发的第一个例子是最小的：一个“微型”博客应用程序。简单地说，它所有能够做的就是显示一个关于对话框，但它会告诉你创建`wxErlang`应用程序的基本原理。在“MiniBlog实例”一节中，你将看到如何添加博客的功能。

我们的程序包含在一个文件里面：*microblog.erl*，开头是这样的（注1）：

```
%% 微型博客程序，它创建一个带有菜单的窗体，
%% 并允许显示“about”对话框。

-module(microblog).
-compile(export_all).

-include_lib("wx/include/wx.hrl").

-define(ABOUT,?wxID_ABOUT).
-define(EXIT,?wxID_EXIT)
```

这表明了*wx.hrl*头文件的引入（位于特定安装位置），包含了标准标识符和类型的定义。**ABOUT**和**EXIT**的本地宏定义把它们连接到与菜单项相对应的标准**wx**标识符。这将允许应用程序以平台特有的方式处理它们。

在我们的这个实例中，顶层函数就是`start/0`：

```
%% 顶层函数：创建wx-server，图形对象，
%% 显示应用程序、进程和终止时清理。

start() ->
    wx:new(),
    Frame = wxFrame:new(wx:null(), ?wxID_ANY, "MicroBlog"),
    setup(Frame),
    wxFrame:show(Frame),
    loop(Frame),
    wx:destroy().
```

这个函数首先创建一个**wx-server**的实例，并且在终止之前确保已经销毁而且回收了内存。主要的图形对象是**wxFrame**，这是由无父类的对象（`wx:null()`）、一个任意标识符（`?wxID_ANY`）和标题为“MicroBlog”所创建的对象。

函数**setup**设置窗体内的图形对象，然后在进入主进程**loop**前，使用**wxFrame:show**来显示它们。下面是创建该应用程序的步骤：

```
%% 顶层框架：产生一个菜单栏、两个菜单、两个菜单项
%% 和状态栏。把窗体连接到句柄事件。

setup(Frame) ->

MenuBar = wxMenuBar:new(),
File = wxMenu:new(),
Help = wxMenu:new(),

wxMenu:append(Help,?ABOUT,"About MicroBlog"),
wxMenu:append(File,?EXIT,"Quit"),
```

注1：在版本R13之前，这需要一个明确的对于*wx.hrl*文件的 `include`。

```

wxMenuBar::append(MenuBar,File,"&File"),
wxMenuBar::append(MenuBar,Help,"&Help"),

wxFrame::setMenuBar(Frame,MenuBar),

wxFrame::createStatusBar(Frame),
wxFrame::setStatusText(Frame,"Welcome to wxErlang"),

wxFrame::connect(Frame, command_menu_selected),
wxFrame::connect(Frame, close_window).

```

函数`setup`创建一个菜单栏和两个菜单：File（文件）和Help（帮助）。把About（关于）与Exit（退出）项添加到菜单，然后把菜单附加到为窗体设置的菜单栏。同时把状态栏也添加到窗体。最后，把两种类型的事件关联到窗体进行处理：也就是那些选择菜单项（`command_menu_selected`）和`close_window`发出的信号。

图14-1显示了在Mac OS X中运行的应用程序。它们具有Mac应用程序的外观和感受，菜单出现在屏幕上方的菜单栏里，而不是出现在主窗口的顶部。标准的Mac OS X菜单——Erlang（即应用程序菜单）、File、Window和Help——呈现在我们面前了，它们包含了在所有OS X应用程序可以看到的标准菜单项，这不需要显式地创建MicroBlog里的Erlang或者Window菜单。

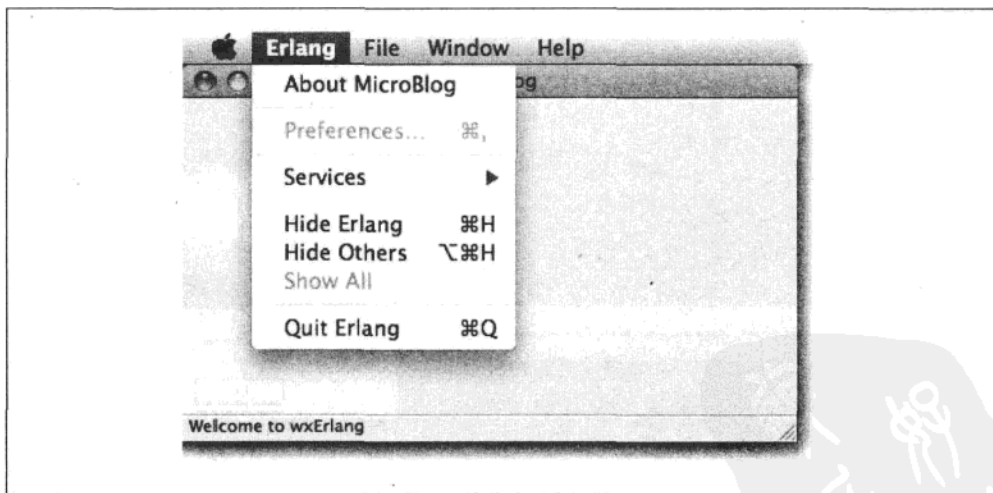


图14-1：Mac OS X中的MicroBlog

而且，About MicroBlog菜单项出现在应用程序的菜单上，这符合OS X GUI的图形用户界面的指导原则，也可以通过把它添加到`setup.wx`的Help菜单里做到这一点，这是因为About菜单项是用标准标识符`wxID_ABOUT`来标识的。与Windows XP版本的作下对比，其中About菜单项出现在Help菜单里，如图14-2所示。

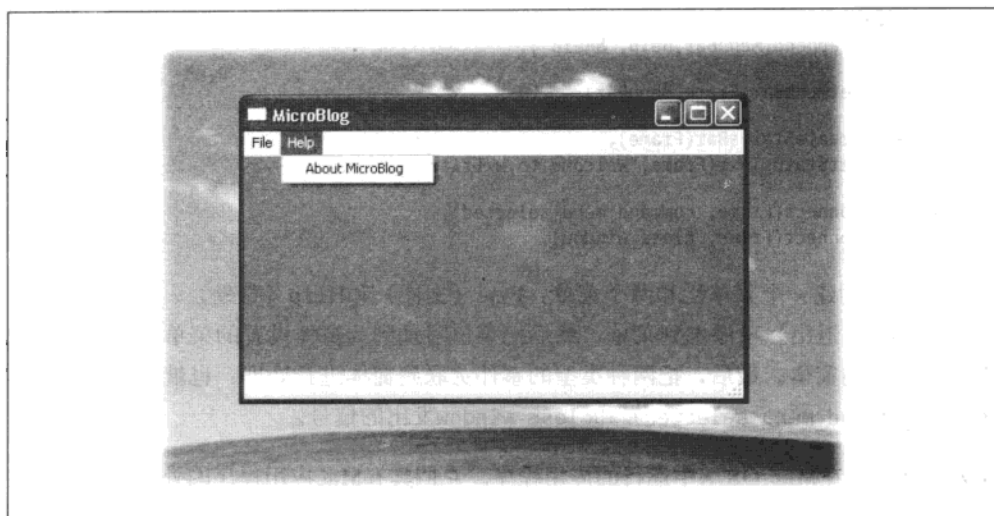


图14-2: Windows XP中的MicroBlog

图14-3显示了在Windows XP中的MicroBlog里选择About的效果图。

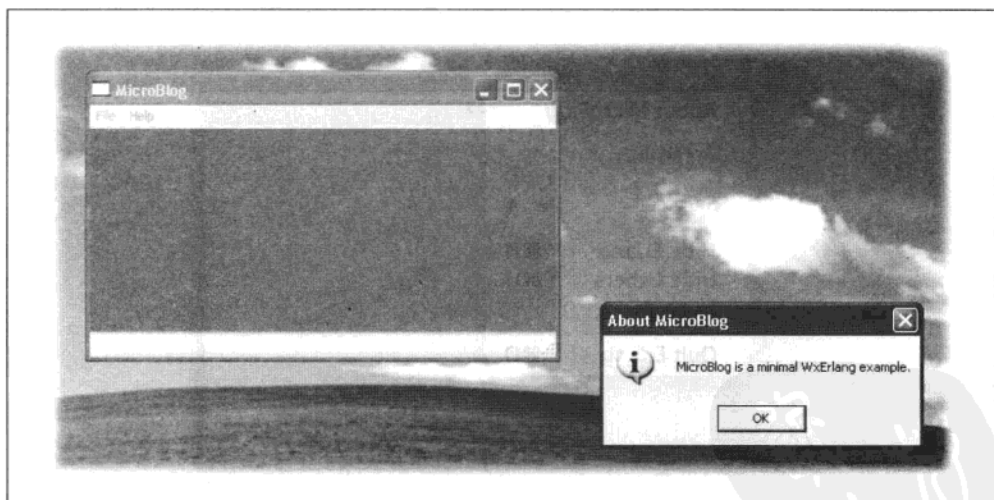


图14-3: 在Windows XP中的About MicroBlog

代码的最后部分给出了loop主函数:

```
loop(Frame) ->
  receive
    #wx{id=?ABOUT, event=#wxCommand{}} ->
      Str = "MicroBlog is a minimal WxErlang example.",
```

```

MD = wxMessageDialog:new(Frame,Str,
                        [{style, ?wxOK bor ?wxICON_INFORMATION},
                         {caption, "About MicroBlog"}]),
wxDialog:showModal(MD),
wxDialog:destroy(MD),
loop(Frame);

#wx{id=?EXIT, event=#wxCommand{type=command_menu_selected}} ->
wxWindow:close(Frame,[])
end.

```

这表明了这两种消息与进程代码一起进行了处理。选择About菜单项产生一个消息对话框，它在对话框中显示了关于应用程序的信息，可以通过OK按钮关闭它。该对话框是以模态形式显示的，使得其他窗口的交互被中断，而仅仅显示该对话框；对话框也可以采用非模态形式显示。还需要注意，在对话框MD显示以后要被显式地销毁，以便允许在这个时候收回内存。

MiniBlog实例

这个例子扩展了前面的例子，给出了“微型博客”的基本实现。这构成了一系列所建议的扩展的基础，并带来了使用wxErlang编程的机会。

微型博客是带有日期的记录列表，每一条记录各占一行，就像Facebook状态消息或一条Twitter Tweet。GUI提供了包括About和Exit选项在内的针对博客的操作：

New (创建)

创建一个新的、空的微型博客。

Open (打开)

打开保存在BLOG文件中的博客。

Save (保存)

保存在BLOG文件中的当前的博客，如果它已经存在，那么覆盖它的内容。

Add entry (添加记录)

在博客的结尾添加一个记录。该记录将自动标上日期。

Undo latest (撤销最新)

撤销最新的“添加项”，这可以通过回退进行操作。

图14-4显示了该程序的屏幕截图。

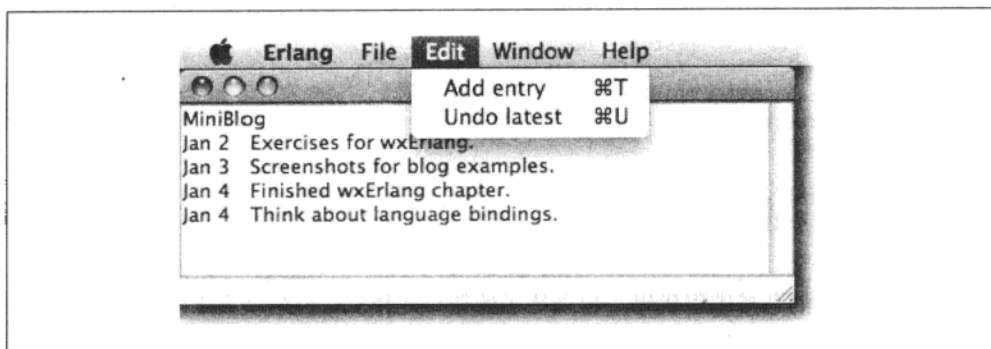


图14-4：微型博客应用

为了说明关于MiniBlog的Erlang代码，我们将解释如何修改MicroBlog的代码。miniblog.erl模块头部用大量的标识符宏定义扩展了microblog.erl，并且给出了每个菜单命令的唯一标识符：

```
-define(APPEND,131).
-define(UNDO,132).
-define(OPEN,133).
-define(SAVE,134).
-define(NEW,135).
```

主函数miniblog:start/0扩展了前面的函数，添加了包含记录的文字控件（wxTextCtrl），然后将这个Text对象传递给setup和loop函数：

```
start() ->
    wx:new(),
    Frame = wxFrame:new(wx:null(), ?wxID_ANY, "MiniBlog"),
    Text = wxTextCtrl:new(Frame, ?wxID_ANY,
                          [{value,"MiniBlog"},
                           {style,?wxTE_MULTILINE}]),
    setup(Frame,Text),
    wxFrame:show(Frame),
    loop(Frame,Text),
    wx:destroy().
```

注意，在文本控件的创建中Frame作为父对象（parent object）被传递，并且有两个可选的参数在最后的变量列表中被传入：该控件的初始值和一个把该控件设置为多行风格的参数。

在建立图形用户界面的时候，需要明确指定一些额外的菜单和菜单项，并且设置文本控件使得它不能被直接修改（下面代码中将省略没有改变的部分）：

```
setup(Frame,Text) ->
    ...,
```

```

Edit = wxMenu:new(),
...,
wxMenu:append(File,?NEW,"New\tCtrl-N"),
wxMenu:append(File,?OPEN,"Open saved\tCtrl-O"),
wxMenu:appendSeparator(File),
wxMenu:append(File,?SAVE,"Save\tCtrl-S"),
wxMenu:append(Edit,?APPEND,"Add en&try\tCtrl-T"),
wxMenu:append(Edit,?UNDO,"Undo latest\tCtrl-U"),

wxMenuBar:append(MenuBar,Edit,"&Edit"),
...,

wxTextCtrl:setEditable(Text,false),
....

```

请注意，在菜单项的字符串中包含一个助记符（在字母前增加一个“&”）和一个快捷方式（在这字符串前增加“\t”）。快捷方式的解释是与平台相关的，如在Mac OS X中，“Undo（撤销）”快捷方式变为⌘U。

主要的处理循环在receive语句中添加了多条子句：

```

loop(Frame,Text) ->
  receive
    #wx{id=?APPEND, event=#wxCommand{type=command_menu_selected}} ->
      Prompt = "Please enter text here.",
      MD = wxTextEntryDialog:new(Frame,Prompt,
                                [{caption, "New blog entry"}]),
      case wxTextEntryDialog:showModal(MD) of
        ?wxID_OK ->
          Str = wxTextEntryDialog:getValue(MD),
          wxTextCtrl:appendText(Text,[10]++dateNow()++Str);
        _ -> ok
      end,
      wxDialog:destroy(MD),
      loop(Frame,Text);

    #wx{id=?UNDO, event=#wxCommand{type=command_menu_selected}} ->
      {StartPos,EndPos} = lastLineRange(Text),
      wxTextCtrl:remove(Text,StartPos-2,EndPos+1),
      loop(Frame,Text);

    #wx{id=?OPEN, event=#wxCommand{type=command_menu_selected}} ->
      wxTextCtrl:loadFile(Text,"BLOG"),
      loop(Frame,Text);

    #wx{id=?SAVE, event=#wxCommand{type=command_menu_selected}} ->
      wxTextCtrl:saveFile(Text,[{file, "BLOG"}]),
      loop(Frame,Text);

    #wx{id=?NEW, event=#wxCommand{type=command_menu_selected}} ->
      {_,EndPos} = lastLineRange(Text),
      StartPos = wxTextCtrl:xYToPosition(Text,0,0),
      wxTextCtrl:replace(Text,StartPos,EndPos,"MiniBlog"),

```

```
    loop(Frame,Text)
end.
```

通过触发wxTextCtrl模块中的适当操作来处理这些事件。系统把文本控件中的记录以文字方式保存，并在文件*BLOG*中保存这些状态。图14-5显示了文本输入对话框，用于在Mac OS X中输入一条新的博客记录。可以看到，对话框里同时也出现了标准按钮OK（接受）和Cancel（取消）。图14-6显示了在Windows XP中相同的文字输入对话框。

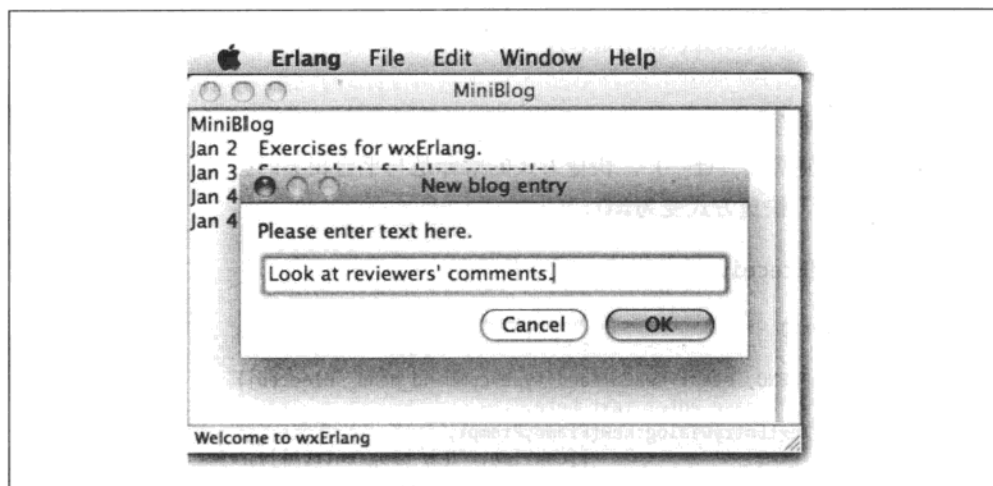


图14-5：制作博客中

我们选择这个例子来说明wxErlang的基本操作，你可以使用许多方法来改进和扩充它，在本章结束部分的练习当中，我们将会给出一些建议。

获取和运行wxErlang

wxErlang是Erlang/OTP标准发行包的一部分，其中包含了入门资料，并带有编译好的Mac和Windows二进制文件，或者也可以用源代码编译构建系统。EDoc格式的wxErlang API文档也包含在发行包中（注2）。

警告： 当运行wxErlang时，你需要通过-smp标记，让Erlang支持对称多进程的方式运行。

注2： 如果你想在Erlang R13发行包之前的版本上运行wxErlang，可以从SourceForge的网站上获取：<http://wxerlang.sourceforge.net>。

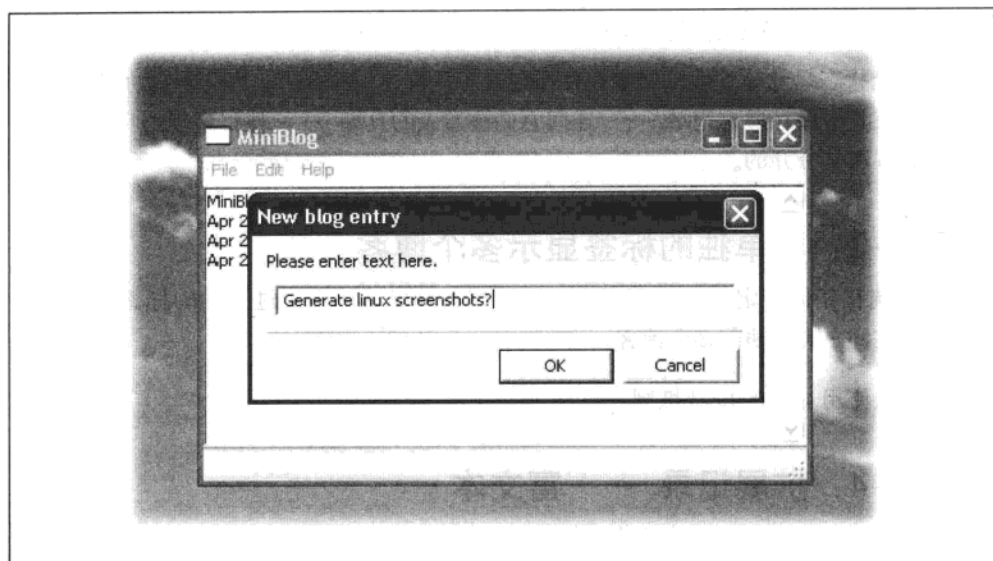


图14-6：制作运行于Windows XP的博客

wxErlang发行包包含了更多实质性的例子，其中包括Sudoku的一个版本、一个XRC的演示以及用wxErlang实现的etop和lerled。

wxWidgets更深入的技术文档可以在线获得 —— wxErlang技术文档包含的链接 —— 和在《Cross-Platform GUI Programming with wxWidgets》一书中找到的。

练习

wxWidgets是一个庞大且复杂的图形用户界面工具包，而本章只触及及其复杂性的表面。这些练习扩充了本章所包含的可运行实例，并且要求查阅wxErlang和wxWidgets文档，以便学习这些控件和其他组件的细节来完成所设置的任务。

练习14-1：选择博客文件

现有系统允许当前的博客只可以在一个固定的文件中保存。添加控件以允许用户选择文件，并在里面保存他的博客文件，还可以选择在博客加载的时候打开文件。

当加载文件时，如果文件不存在，那么该程序应当能处理这种情况。

练习14-2：单独保存博客记录

现有的系统状态只是简单地以文本控件所包含单一文字块来保存。给系统添加一个单独的后端，在其中保存博客记录——连同它们被写入的日期——使得呈现给用户的数据模型及其视图是分开的。

练习14-3：以单独的标签显示多个博客

现有的系统在任何特定时间都只能访问一个单独的博客。请通过不同的标签扩展该系统，可以在同一时间访问多个博客。

提示：可以使用 wxNotebook 机制。

练习14-4：扩展记录——富文本

这里博客的记录都只是一个简单的单行文本。探索如何能够输入多行文字，并且可以包括样式（例如使用 Rich Text Format）。

练习14-5：标记记录

提供一个可以用来标记博客的记录机制，使这些匹配特定关键字的记录可以显示出来。

练习14-6：多用户和评论

现有的系统是为单个用户设计的。研究如何可以管理用户的身份（使用密码），以及如何扩展程序，以便容纳博客记录的评论。

练习14-7：布局和wxErlang整理器

针对练习14-4、14-5和14-6的答案，需要有一个复杂的布局，研究一下 wxErlang 的整理器，你可以使用它来布置多个图形控件，而不需要明确设置所包含的各个部件的尺寸。

套接字编程

虽然分布式Erlang可能是向允许远程计算机上运行的程序之间能够相互通信迈出的第一步，但是我们有时不得不依靠更底层的机制和标准化的协议。Socket（套接字）允许用任何语言进行编写程序，通过使用互联网协议（IP）套件的协议传输字节流，以便在不同的计算机之间交流数据。

尽管套接字用于在不同的机器上运行的程序之间创建一个面向字节的通信流，但是我们将在第16章提及的端口（port）也会为同一台机器上运行的程序提供相同的功能。Erlang中的字节流可以看做二进制或整数列表，它往往遵循应用程序级的标准协议，允许彼此独立编写的程序进行交互。

基于套接字通信的实例包括有Web浏览器和服务器之间的通信、即时消息（IM）客户端、电子邮件服务器和客户端以及点对点应用程序。Erlang发行包本身是基于通过套接字相互通信的节点。

Erlang可以对用户隐藏原始的数据包，提供用户数据报协议（UDP）和传输控制协议（TCP）的用户友好的API。它们包含在两个库模块里：一个是`gen_udp`，它是一个无连接、不可靠和基于数据包通信的协议；另一个是`gen_tcp`，它提供了面向连接的通信信道。这两个协议都是通过IP进行通信的。

用户数据报协议

用户数据报协议（UDP）是无连接协议。如果已发送出了一个UDP数据包，并且恰好有一个套接字在另一端监听，那么它将会接收到这个数据包。UDP几乎不提供错误恢复功能，而是把数据包留给应用程序，由应用程序来确保数据包的接收和一致性。UDP数据包可以采取不同的路由，因此接收数据包可能与发送的顺序不同。它们也可能在路由途中丢失，并且由于接收端并没有确认它们，它们的丢失是“默默”进行的。虽然该协议可能不可靠，但是使用它的开销很小，它非常适合这样的传输，既你宁愿放弃数据包也

不愿等待它重新发送数据包。例如，错误和报警器被广播出去，以希望另一端的套接字能接到它们。

在Erlang中，UDP是通过gen_udp模块实现的。让我们通过一个例子来熟悉它。在同一主机上启动两个Erlang节点，并确保按以下顺序执行命令：

1. 在第一个Erlang节点，打开端口为1234的UDP套接字。
2. 在第二个Erlang节点，打开端口为1235的UDP套接字。
3. 使用第二个节点套接字发送二进制<<"Hello World">>到在本地主机IP地址127.0.0.1的端口1234上监听的套接字。
4. 使用第二个节点的套接字，发送字符串"Hello World"到相同的IP地址和监听套接字。
5. 第一个节点中已打开（且拥有）套接字的进程应收到两条"Hello World"消息。使用终端命令flush()可以读取它们。
6. 关闭这两个套接字，从而释放这些端口号。

在第一个节点的Erlang终端上，命令和输出是这样的：

```
1> {ok, Socket} = gen_udp:open(1234).
{ok, #Port<0.576>}}
2> flush().
Shell got {udp, #Port<0.576>, {127,0,0,1}, 1235, "Hello World"}
Shell got {udp, #Port<0.576>, {127,0,0,1}, 1235, "Hello World"}
ok
3> gen_udp:close(Socket).
ok
```

你应该记住，一旦打开套接字，你需要从第二个节点发送消息到第一个节点。在第二个节点的Erlang终端上，命令是这样的：

```
1> {ok, Socket} = gen_udp:open(1235).
{ok, #Port<0.203>}}
2> gen_udp:send(Socket, {127,0,0,1}, 1234, <<"Hello World">>).
ok
3> gen_udp:send(Socket, {127,0,0,1}, 1234, "Hello World").
ok
4> gen_udp:close(Socket).
ok
```

要特别注意发送给拥有套接字进程的UDP消息的格式，而事实上，即使第一个消息是以二进制发送的，但它同样收到了作为列表的两条消息。在更详细地探讨所涉及的函数后，我们将解释这一切。

如果你在不同的计算机上尝试这个例子，那么你应该用发送消息的目标计算机地址取代本地计算机的IP地址，并且确保没有防火墙阻止相关的端口。

正如你在图15-1看到的，其他主机上的客户端把它们的UDP数据包发送到监听套接字，而这个套接字又把它们转发到了一个Erlang进程。在任何时候只有一个进程允许接收来自某个特定套接字的数据包。这个进程称为控制进程。

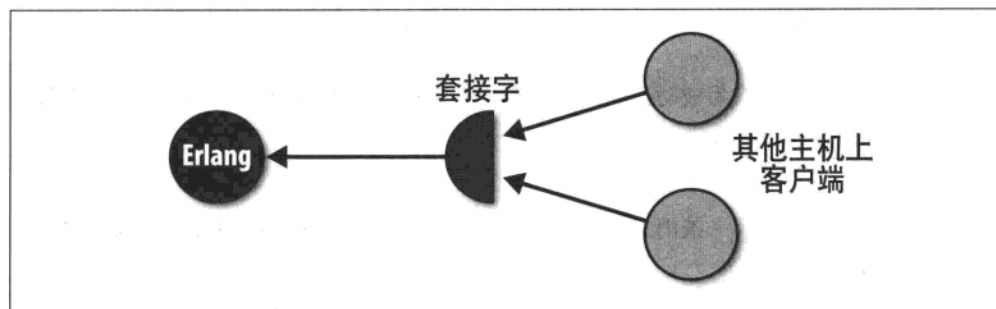


图15-1：UDP监听套接字

要在客户端和服务端上打开套接字，可以调用下面的函数：

```
gen_udp:open(Port)
gen_udp:open(Port, OptionList)
```

端口（Port）是一个整数，代表套接字的监听端口号。若把消息发送到套接字的客户端使用该端口。其中OptionList包含允许你取代默认值的配置选项。最有用的参数包括：

list（列表）

不管所有消息是如何发送的，把所有数据包中的消息作为一个整数列表转发。如果没有其他选项，那么这就是默认值。

binary（二进制）

把所有数据包中的消息作为二进制数据包转发。

{header, Size}

如果数据包是作为二进制接收，那么可以使用该选项。它把消息分割成一个大小为Size的列表、头部（header）和消息（以二进制格式出现）。正如在第9章所描述的，在引入位语法和二进制模式匹配之前，此选项特别有用。重复一下前面的两个节点的UDP例子，不过这次使用以下代码打开第一个套接字：

```
{ok, Socket} = gen_udp:open(1234,[binary,{header,2}]).
```

接着发送[0,10|"Hello World"]，结果接收的第一条消息如下：

```
2> flush().
Shell got {udp,#Port<0.439>,{127,0,0,1},1235,[0,10]<<"Hello World">>}}
ok
```

在前面的代码中把该消息分割成了（两个整数）头部和消息。`{active, true}`确保从套接字接收的所有消息经Erlang消息格式`{udp, Socket, IP, PortNo, Packet}`转发到拥有该套接字的进程。`Socket`是接收套接字，`IP`和`PortNo`是IP地址和发送端口号，而`Packet`是消息本身。当打开一个套接字时，默认是主动模式。

`{active, false}`

可以把套接字设置为被动模式。使用`gen_udp:recv/2`和`gen_udp:recv/3`从套接字接收消息，而不是发送。

`{active, once}`

将发送它接收的第一条消息到套接字，但随后的消息都必须使用`recv`函数取出。

`{ip, ip_address()}}`

在一台定义了多个网络接口的计算机上打开一个套接字时使用。此选项指定套接字应使用哪些接口。

`inet6`

设置IPv6套接字时使用。`inet`用来设置IPv4套接字，并且也是默认值。

调用函数`open`返回`{ok, Socket}`或者`{error, Reason}`，其中`Socket`是已打开的套接字标识符，而`Reason`是作为基元返回的几个POSIX错误代码之一。它们可以在Erlang运行时系统文档的`inet`手册页面上找到。遇到的最常见的错误首先是`eaddrinuse`，如果这个地址已在使用；其次是`eaddrnotavail`，如果你使用的端口号已超出了操作系统所预留的范围；最后是`eaccess`，如果你没有权限打开这个套接字。

调用`gen_udp:close(Socket)`将关闭套接字，并且释放分配给它的端口号，它返回基元`ok`。

如果你想发送消息，可以使用下列函数：

```
gen_udp:send(Socket, Address, Port, Packet)
```

其中的`Socket`是在本地计算机上发出消息的UDP套接字。这里的`Address`可以是输入的一个包含主机名或IP地址的字符串，一个包含本地主机名的基元或一个包含组成IP地址的整数的元组，在高吞吐量的应用程序中通过元组的形式比采用字符串更有效。`Port`是接收主机上的端口号，并且`Packet`是作为一个字节序列的消息内容，它可以是一个整数或二进制列表。

当以被动模式打开套接字时，连接进程必须使用下面这些函数调用，明确地从该套接字接收数据包：

```
gen_udp:recv(Socket, Length)
gen_udp:recv(Socket, Length, Timeout)
```

这里的Length只与TCP原始传输模式相关，所以在这种情况下可以忽略它。如果在超时之前收到一个数据包，那么就返回{ok, {Ip, PortNo, Packet}}。如果在Timeout毫秒内没有收到字节，那么将会返回{error, timeout}。如果不是被动模式，而接收进程调用gen_udp:recv，那么将看到{error, einval}错误，这是一个表明参数无效的POSIX错误代码。

UDP最常见的用途是实现简单网络管理协议（SNMP）。SNMP是一个经常用来监控基于IP网络的设备和系统的协议标准。你可以在运行时系统提供的文档中阅读有关Erlang的SNMP应用程序。

传输控制协议

传输控制协议（Transmission Control Protocol，TCP），简称TCP，它是一个允许点对点数据流交换的面向连接协议。不同于UDP，TCP数据包的接收是有保证的，并且以与发送相同的顺序接收这些数据包。常见的TCP用途包括HTTP请求、点对点的应用和IM客户端/服务器端连接。Erlang发行包建立在TCP基础之上。与UDP一样，无论是客户端还是服务器端不必都是在Erlang中实现的。

在体系结构层面上，TCP和UDP之间的主要区别是，一旦你打开了一个使用TCP连接的套接字，它就始终保持打开状态，直至任何一方关闭它或因为一个错误而终止。在建立一个连接时，你往往为每一次请求产生一个新进程，只要有请求被处理，就保持这个进程的存活。

这实际是如何工作的？假设你有一个监听进程，其任务是等待传入的TCP请求。只要一个请求到达，响应该连接请求的进程就变成了接收进程。有两种机制来定义这种接收进程：

- 第一个选择是产生新的进程并成为接收进程，而监听者返回并继续监听下一个新的连接请求。
- 第二个选择是使监听进程成为接收进程，并使产生的新进程成为新的监听者，如图15-2所示。

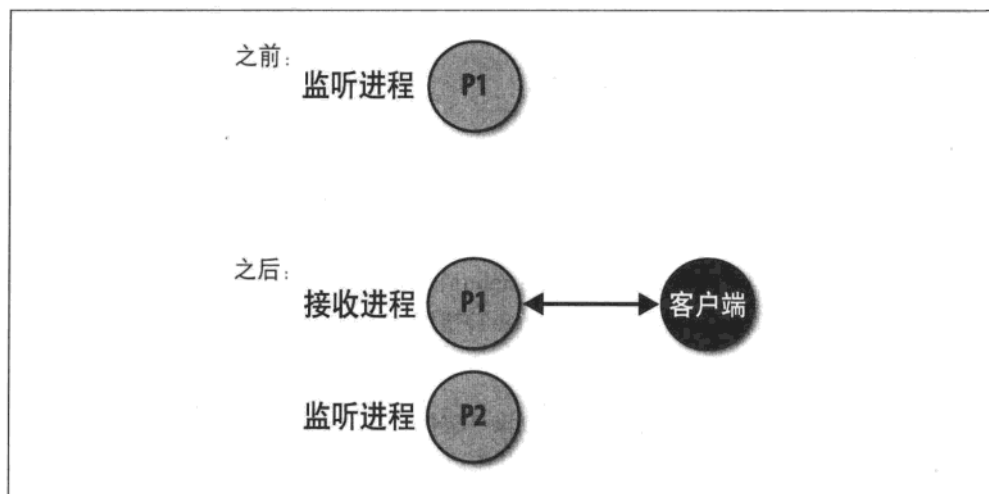


图15-2：监听和接收进程

如果套接字以主动模式打开，那么拥有套接字的进程将收到形式为{tcp, Socket, Packet}的消息，其中Socket是接收的套接字而Packet是消息本身。

如果是在被动模式下工作，就像使用UDP，你需要使用以下调用：

```
gen_tcp:recv(Socket, Length)
gen_tcp:recv(Socket, Length, Timeout)
```

调用将返回一个格式为{ok, Packet}的元组。在这些调用中，非零值Length在返回消息前将会一直等待这个套接字的字节数。如果该值为0，则返回所有已收到的数据。如果关闭发送套接字，并且少于Length的字节被已缓冲，那么它们将被丢弃。只有当数据包类型是原始的（raw）时，选项Length才有意义。

警告：使用被动模式是确保你的系统不会因请求而溢出的一个好方法。为每一个接收到的消息产生一个新的进程来处理请求，这是一种常见的设计模式。在持续繁忙的流量的极端情况下，系统因请求而溢出（进程也是这样），虚拟机将有内存不足的风险。通过使用被动模式的套接字，底层的TCP缓冲区可用于抑制请求，并拒绝客户端的消息。了解是否需要在TCP一层进行抑制，以及在网络流量突发时内存大小是否是一个问题，最好的办法是对你的系统进行广泛的压力测试。

TCP实例

让我们先从一个如何使用TCP套接字的简单例子开始。给定一个主机和一个二进制数，

客户端打开一个端口为1234的套接字连接。然后使用位语法，把二进制分解成100字节的块，并且以单独的数据包方式把它们发送过去。

```
client(Host, Data) ->
  {ok, Socket} = gen_tcp:connect(Host, 1234, [binary, {packet, 0}]),
  send(Socket, Data),
  ok = gen_tcp:close(Socket).
```

你可能还记得第9章中的关于二进制的描述，表达式<<Chunk:100/binary, Rest/binary>>将这个二进制字符串的第一个100字节绑定到变量Chunk，而其余的字节留在Rest中。当二进制字符串少于100字节时，模式匹配调用函数send/2的第一个语句将会失败。无论留下部分是什么，甚至可能是空的二进制字符串，都将匹配第二个语句，并将其内容发送到服务器，然后关闭该Socket连接。

```
send(Socket, <<Chunk:100/binary, Rest/binary>>) ->
  gen_tcp:send(Socket, Chunk),
  send(Socket, Rest);
send(Socket, Rest) ->
  gen_tcp:send(Socket, Rest).
```

服务器端有一个等待客户端连接的监听进程。当请求到达时，监听进程转换为接收进程，并准备以被动模式接收二进制字符串。一个新的监听进程生成并且等待下一个连接请求。接收进程继续接收来自客户端的数据，并把它添加到一个列表，直至关闭套接字，之后会将数据保存到一个文件中。

```
server() ->
  {ok, ListenSocket} = gen_tcp:listen(1234, [binary, {active, false}]),
  wait_connect(ListenSocket, 0).

wait_connect(ListenSocket, Count) ->
  {ok, Socket} = gen_tcp:accept(ListenSocket),
  spawn(?MODULE, wait_connect, [ListenSocket, Count+1]),
  get_request(Socket, [], Count).

get_request(Socket, BinaryList, Count) ->
  case gen_tcp:recv(Socket, 0, 5000) of
    {ok, Binary} ->
      get_request(Socket, [Binary|BinaryList], Count);
    {error, closed} ->
      handle(lists:reverse(BinaryList), Count)
  end.

handle(Binary, Count) ->
  {ok, Fd} = file:open("log_file_"+integer_to_list(Count), write),
  file:write(Fd, Binary),
  file:close(Fd).
```

请注意get_request/3函数是如何以100字节为大小接收二进制块的。一旦已经收到所有块并且关闭了套接字，你就需要把保存它们的列表颠倒过来，因为第一个应该写入的块

此时是列表的最后一个元素。你把这些块写到一个文件，并在完成后关闭套接字，释放该文件描述符。

若要运行该实例，你需要做的就是使用`tcp:start()`来启动服务器，并且使用如下命令运行客户端：

```
tcp:client({127,0,0,1}, <<"Hello Concurrent World">>).
```

你可以看到很多命令，它们都类似于我们在前面的UDP实例中使用过的命令。主要区别在于以下调用：

```
gen_tcp:listen(PortNumber, Options)
```

这将启动一个监听套接字，然后等待传入的连接。这个调用采用与前面所述的函数`gen_udp:open/2`同样的选项，以及下面TCP特定的选项：

{active, true}

确保所有从套接字接收的消息都作为Erlang消息转发到拥有这个套接字的进程。当打开一个套接字时，默认值是主动模式。

{active, false}

设置套接字为被动模式。套接字收到的消息被缓存起来，进程必须通过函数`gen_tcp:recv/2`和`gen_tcp:recv/3`调用读取它们。

{active, once}

将设置套接字为主动模式，但是一旦收到第一条消息，就将其设置为被动模式，并使用`recv`函数来读取后续的消息。

{keepalive, true}

当没有转移数据时，确保所连接的套接字发送保持活跃（keepalive）的消息。因为“关闭套接字”消息可能会丢失，如果没有接收到保持活跃消息的响应，那么该选项可确保这个套接字能被关闭。在默认情况下，该标签是关闭的。

{nodelay, true}

数据包直接发送到套接字，不管它多么小。在默认情况下，此选项处于关闭状态，并且与之相反，数据被聚集而以更大的数据块进行发送。

{packet_size, Integer}

设置数据包允许的最大长度。如果数据包比Size更大，那么将认为这个数据包无效。

还有其他的标记，你可以在关于`gen_tcp`和`inet`模块的手册中读到所有相关的内容。

gen_tcp:listen/2调用将会立即返回。它返回一个套接字标识符Socket，其将被传递给下列函数：

```
gen_tcp:accept(Socket)
gen_tcp:accept(Socket, Timeout)
```

这些调用会阻塞进程，直到有一个连接请求发送到该IP地址上的套接字。Timeout是以毫秒为单位的值，它会引起没有连接到该端口的请求返回结果{error, timeout}。通过以下的调用产生请求连接：

```
gen_tcp:connect(Address, Port, OptionList)
```

Address是你所连接机器的IP地址，Port是相应套接字的端口号。OptionList类似于在gen_tcp:listen/2调用中定义的那个，它包含gen_udp:open/2的选项，与前面所讨论的TCP特定的keepalive、nodelay和packet_size在一起。

由于这个例子的套接字是在被动模式下运行的，可以调用函数gen_tcp:recv/1和gen_tcp:recv/2读取套接字的消息。如果在主动模式下运行套接字，那么消息会以{tcp, Socket, Packet} 和 {tcp_error, Socket, Reason}的格式发送到这一进程。

如果调用gen_tcp:close(Socket)，则将关闭套接字。这可在客户端或服务器端上实现。在两者之中的任何一种情况下，{tcp_closed, Socket}消息将发送到另一端的套接字，从而有效地关闭套接字。

控制进程一般是这样的进程，它通过调用gen_tcp:accept或者gen_tcp:connect之一来建立一个连接。为了在别处重定向消息，并把控制传递给其他进程，该控制进程得调用gen_tcp:controlling_process(Socket, Pid)。

在我们前面的例子中，调用gen_tcp:accept的进程成为控制进程，然后我们生成了新的监听进程。相反，如果我们要生成一个将成为控制进程的新进程，并保持监听进程不变，其代码是这样的：

```
server() ->
  {ok, ListenSocket} = gen_tcp:listen(1234, [binary, {active, false}]),
  wait_connect(ListenSocket, 0).

wait_connect(ListenSocket, Count) ->
  {ok, Socket} = gen_tcp:accept(ListenSocket),
  Pid = spawn(?MODULE, get_request, [Socket, [], Count]),
  gen_tcp:controlling_process(Socket, Pid),
  wait_connect(ListenSocket, Count+1).
```

在最新的Erlang/OTP发行包中允许使用同一监听套接字对应多个接收器。可以预料，比

起每次创建一个新的接收器，这将提供更大的带宽。我们把这个实例的修改留给你作为一个练习！

inet模块

inet模块包含和套接字一起使用的通用函数，无论你是否使用TCP或UDP。它们提供了套接字通用的访问方法以及有用的库函数。不需要太多考虑哪些是可用的，在本节中，我们通过终端里展示它们的用途来演示最常用的函数。如果需要更多的信息，你可以在inet模块手册上查看。该手册还包含了所有的套接字操作将会返回的POSIX错误定义。

一旦建立了套接字，如果需要更改套接字选项，那么可以调用`inet:setopts(Socket, OptionList)`，其中`OptionList`是一个标记元组的列表，它含有本章前面已经描述的选项和其他的一些选项，在inet模块手册上已经列出了一些不太常用的选项。

要得到现有套接字的配置参数，可以使用`inet:getopts(Socket, Options)`，其中`Options`表示一个你感兴趣的选项值的基元列表。该函数返回一个带有标记的列表，如果底层操作系统或所使用的套接字类型不支持那个特殊的选项，那么将会在结果中忽略它们。

```
1> {ok, Socket} = gen_udp:open(1234).
{ok, #Port<0.468>}}
2> inet:getopts(Socket, [active, exit_on_close, header, nodelay]).
{ok, [{active,true},{exit_on_close,true},{header,0}]}
```

套接字将收集统计有关发送和接收的数据。接收计数器以`recv_`为前缀，而发送计数器以`send_`为前缀。使用如下可以得到这些统计信息：

`avg`

该数据包的平均大小。

`cnt`

已发送或接收的数据包数量。

`dvi`

套接字已发送或者接收字节的数据包大小的偏差。

`max`

最大数据包的大小。

`oct`

套接字已发送的或接收的字节数。

在这个例子中，我们的UDP套接字接收4个数据包，并且没有发送任何数据包。其输出为：

```
3> flush().
Shell got {udp,#Port<0.468>,{127,0,0,1},1235,"Hello World"}
Shell got {udp,#Port<0.468>,{127,0,0,1},1235,"Hello World"}
Shell got {udp,#Port<0.468>,{127,0,0,1},1235,"Hello World"}
Shell got {udp,#Port<0.468>,{127,0,0,1},1235,"Hello World"}
ok
4> inet:getstat(Socket).
{ok,{recv_oct,44},
    {recv_cnt,4},
    {recv_max,11},
    {recv_avg,11},
    {recv_dvi,0},
    {send_oct,0},
    {send_cnt,0},
    {send_max,0},
    {send_avg,0},
    {send_pend,0}}}
```

你可能会发现一些有用的函数，可以在终端里尝试一下。其中一些将会返回`hostent`的记录，它已在`inet.hrl`包含文件中进行了定义。请记住，你可以使用终端命令`rr("../lib/kernel-2.13/include/inet.hrl")`加载记录定义。

```
inet:peername(Socket).
inet:gethostname().
inet:getaddr(Host, Family).
inet:gethostbyaddr(Address).
inet:gethostbyname(Name).
```

最后，需要知道一个有用的命令，特别是在你试图从套接字打开、发送或接收数据遇到问题时，它是`inet:i()`。它列出了包括那些Erlang运行时系统使用的以及那些你已经创建的所有的TCP和UDP套接字。

在我们的例子启动了一个分布式Erlang节点。运行命令后显示了两个套接字——等待传入连接的TCP监听套接字和连接到`epmd`端口映射后台程序的套接字：

```
(bar@vaio)1> inet:i().
Port Module Recv Sent Owner Local Address Foreign Address State
108 inet_tcp 0 0 <0.62.0> *:54843 *.* ACCEPTING
110 inet_tcp 4 18 <0.60.0> localhost:54844 localhost:4369 CONNECTED
Port Module Recv Sent Owner Local Address Foreign Address State
ok
```

扩展阅读

本章介绍了一些底层的机制，用于建立更复杂的协议和层次。作为OTP发行包一部分的

Inets应用程序，是以IP为基础的协议实现的容器。它包括一个称为Inets的Web服务器以及HTTP和FTP客户端。它也有一个普通文件传输协议（TFTP）客户端和服务端。关于Inets应用程序的更多信息，请参阅其用户指南和参考手册。

分布式Erlang的其中一部分是安全套接字层（SSL）应用软件，其提供了套接字之上的加密通信。Erlang的SSL应用软件基于开放源代码的OpenSSL工具包。在Erlang发行包的用户指南和参考手册中，你可以了解关于此应用软件更多的信息。

如果你有兴趣阅读有关其他互联网协议的实现，那么推荐两本好书：由Eric Hall所著的《Internet Core Protocols》（O'Reilly出版社）和由W. Richard Stevens所著的《TCP Illustrated》（Addison-Wesley出版社）。

练习

练习15-1：探听HTTP请求

在本地计算机上打开一个监听套接字。启动Web浏览器，并发送了一个网页请求给它。打印该请求的内容并研究。在关闭套接字连接之前需要多长时间？如果你关闭浏览器，那么将会发生什么情况？

练习15-2：简单的HTTP代理

更改浏览器代理设置为指向你本地计算机的端口1500（注1）。在该端口上启动监听套接字，并接收任何到达它的连接。从你的Web浏览器尝试下载任何网页。这项请求应该转发到你的套接字连接。侦听请求的内容，并提取你的浏览器正试图加载网页的URL。

使用Inets应用程序里的HTTP客户端，读取你正要加载的网页的内容，并且没有变化地把它发送到打开的套接字连接。提示：如果你不是在代理或防火墙后面，那么`http:start()`和`http:request("http://www.erlang.org")`应该可以工作。不过，在使用它们之前，你一定要阅读与Erlang发行包一起提供的HTTP手册。

练习15-3：点对点

编写一个模块，它包含点对点传输层的代码。你需要一个进程，在启动后该进程等待到达端口1234的套接字连接，或者等待函数`peer:connect(IPAddress)`被调用。如果是后

注1：根据正在运行Erlang节点的用户权限，你可能无法打开要保留或已经采取的端口。如果是这样，那么可以选择一个较大的数字。

者被调用，那么它将尝试连接到该地址的端口1234。一旦连接建立成功，你就应该能够使用函数`peer:send(String)`把数据发送给对方。记录发送的内容到文件，并且把它打印到终端上。你应该导出的函数是：

```
peer:start() -> ok | {error, already_started}
peer:connect(IPAddress) -> ok | {error, Reason}
peer:send(String) -> ok | {error, not_connected}
peer:stop() -> ok | {error, not_started}
```

这项工作的困难之处在于这样一个事实，你的进程将和它自身在另外一台计算机上的副本通信，这将需要一些认真的思考。对于这一点，我们指的那两个进程将运行相同的代码基础。

如果你担心有人（Big Brother）在窥视你，那么你可以借助`crypto`模块来加密所要发送的数据包。



Erlang与其他编程语言接口

对于建立任何规模的现代计算机系统，使用一种以上的编程语言是普遍的现象。设备驱动程序通常是用C编写的，而许多集成开发环境（IDE，如Eclipse）和其他以GUI为主的系统是用Java或C#编写的。轻量级Web应用程序可以使用Ruby和PHP开发，而Erlang可以提供轻量级和容错并发性。如果你需要有效地操纵或解析字符串，那么Perl或Python是这一方面的规范。为你解决了一个特定问题的库可能并不是用你最喜欢的语言编写的，这样你必须选择使用外来库，或者硬着头皮自己使用Erlang来全部重新编写这个东西（注1）。

在自然语言或编程语言中，语言之间通信从来就不是一件简单的事情。在自然语言中，我们必须了解不同语言的工作方式。它们是否包含冠词？它们是否代表词性？动词出现在句子哪里？我们还必须了解单词是如何翻译的。例如，在葡萄牙语中的动词ser和英语的动词“to be”是一样的含义？（它不是的。）对于编程语言也同样如此。它们来自哪一种典范？这些语言是函数的、面向对象的、并发的或结构的？在Java中一个整数是否与Erlang的整数指的是同一样东西？（这肯定也是不一样的！）

互操作性不仅是语言之间的通信事情，而且Erlang/OTP也借助于XML、ODBC、CORBA、ASN和SNMP支持这种通信。这些有助于Erlang作为“分布式胶水”把单线程遗留程序联合在一起。

交互运作概况

Erlang为语言之间的互动提供了一些机制：基于Erlang分布式节点的更高层模型、通过端口允许与外部程序通信的更低层模式以及连接程序到虚拟机本身的一种机制，这称为内联驱动（linked-in drivers）。

注1：这样做是针对Wrangle（Erlang重构工具）的重复代码检测算法。一个现有的高效C库用来识别在Erlang软件里的“克隆”。

Erlang分布式编程模型提供了一种简单而灵活的解决方案，该方案解决了Erlang如何与其他语言进行工作协作的高层问题：在相同或不同计算机的其他节点上运行这一语言，使得它像一个分布式Erlang节点，你可以和节点之间来回传递消息。这些节点提供了一个外来程序可以运行的环境，但也与Erlang节点进行通信。为了使这个通信能够正常工作，可以使用端口，或者在其他语言里提供一个Erlang通信原语的更高水平模型。在任何情况下，存在如何处理基本类型以及支持在两种语言的复杂数据之间转换处理不同程度的问题。

在本章中，我们将讨论构建在Java和C语言上的节点，使得它们可以与Erlang相互协作，而且还将介绍`erl_call`，它允许Unix终端和构建在`erl_interface`库之上的分布式Erlang节点通信。这些连同JInterface Java包都来自标准的Erlang发行包。这些库提供了代码和结构的稳定性，但损失了一些绝对速度。之后我们将介绍如何通过端口进行通信，而且我们会给出如何利用`erlectricity`库与Ruby交互的实例。

与其他语言一起工作

在本章中，我们将介绍Java、C和Ruby。不过，Erlang也可以与其他一些编程语言连接，其中包括：

- OTP.NET，它通过JInterface代码端口提供了与.NET平台的连接。
- Py接口，这是一个Erlang节点的Python实现，它允许在Python和Erlang之间通信。
- Perl Erlang端口，它允许Perl代码与Erlang通过一个端口进行通信。
- PHP/Erlang，目的是成为一个PHP扩展，利用一组简单功能，将一个PHP线程转变成一个Erlang C节点。
- Haskell /Erlang FFI，使得在Haskell和Erlang编写的程序之间实现双向通信。从Haskell发送至Erlang的消息看起来像函数调用，从Erlang发送至Haskell的消息传递到MVars。
- 一个Erlang/Gambit接口，它允许在Scheme和Erlang的程序之间进行通信。
- Distel支持Erlang和Emacs Lisp的相互操作，在Emacs中提供增强的Erlang的模式。

为了获得互操作的最大效率，你可以定义一个内联驱动。问题在于一个错误的内联驱动将导致整个Erlang运行时系统内存泄漏、暂停或崩溃，所以应谨慎使用内联驱动。

与Java交互运作

JInterface Java包提供了在Java里的Erlang风格的进程和通信的高层模型。你可以独立地使用这个软件包，在Java语言里提供Erlang风格的并发性，或者也可以用作混合Java/Erlang分布式系统的一部分，这将允许Java系统包含Erlang的组件，反之亦然。

一个像JInterface这样的Java包由一组Java类的集合组成，其中大部分是以前缀Otp开始。本节将介绍其中最常用来提供在Erlang和Java之间通信时传递信息和处理数据类型的例子。你可以在Erlang/OTP文档的“Interface and Communication Applications”一节中找到关于JInterface的补充资料及其Erlang类。本节运行的例子是第11章的远程进程调用（RPC）实例的修改版本。

节点和邮箱

在第11章中我们描述了Erlang节点，也介绍了Erlang中的分布式编程。Erlang节点通过它的名称来标识：它由带主机名（以短的或长的形式）的标识符组成。如果它们的名称不同，则每一台主机可以运行多个节点。

类OtpNode给出了Erlang节点的JInterface表示：

```
OtpNode bar = new OtpNode("bar");
```

这将创建Java对象bar —— 我们称之为一个节点 —— 它表示Erlang节点bar，运行于执行该语句的主机上。

通过创建一个邮箱，你可以在该节点上创建一个进程，它可以由一个标识符表示或者也可以用名称注册。要创建这个进程，可以使用如下语句：

```
OtpMbox mbox = bar.createMbox();
```

在创建时它给这个进程一个名称。接着，在创建时把名称作为一个字符串传入：

```
OtpMbox mbox = bar.createMbox("facserver");
```

使用以下语句，你也可以单独创建这一邮箱：

```
mbox.registerName("facserver");
```

这将创建名为facserver的进程。这一进程将用作一个“阶乘服务器”，把它接收到的整数的阶乘发送到原来发送该整数的进程。

一旦你命名了邮箱，你就可以使用其名称来访问。如果也需要它的标识符 —— 也许是一个远程Erlang节点需要 —— 可以使用邮箱的函数self获得：

```
OtpErlangPid pid = mbox.self();
```

表达Erlang类型

软件包JInterface包含了各种不同的类，它们用Java语言代表了各种Erlang类型。这些方法允许在基本Java类型和这些代表类型之间的转换，也支持两门语言之间的值转换，这是它们可以有效地协同工作的本质。

在前面的Java语句中曾经介绍过这样的实例：类OtpErlangPid给出了Erlang进程标识符的Java表达。Erlang类型，如二进制文件、列表、进程、端口、引用、元组和项元，可以映射为相应的Java类，如OtpErlangAtom、……、OtpErlangTuple、OtpErlangObject。

Erlang的浮点类型可以转换为OtpErlangFloat或OtpErlangDouble，整数类型可以转换为OtpErlangByte、OtpErlangChar、OtpErlangShort、OtpErlangInt、OtpErlangUInt、或OtpErlangLong，这取决于具体的整数值和符号。

为了表示两个特殊基元true和false，可以使用OtpErlangBoolean类，而Erlang字符串——它们在Erlang中是整数的列表——是由OtpErlangString来描述的。

在JInterface文档中有这些类的细节，我们将在“C节点”一节以及RPC的例子中使用这些类。

通信

Erlang进程发送和接收消息，而在JInterface中这些操作是通过邮箱的发送和接收方法实现的。互换消息是Erlang项元，而在Java中由OtpErlangObjects表示。在Erlang中发送消息：

```
Pid ! {ok, M}
```

而在mbox进程中由下面的语句给出：

```
mbox.send(pid,tuple);
```

其中Java中的变量pid对应Erlang中的变量Pid，而tuple（注2）代表Erlang项元{ok, M}。

接收一个消息：

注2： 如果你已经编写Erlang程序好几年了，看到如同pid和tuple这样的变量你的反应会是：它们都不是大写开头的。其实很多人也和你一样。最重要的是在这个过程中，你没有公开地提出疑问：函数让基元作为参数是如何实际工作的，从而确保没有人注意到你的失误。

```
OtpErlangObject o = mbox.receive();
```

这个语句不同于Erlang的函数receive，因为它没有对消息进行模式匹配。它单独解构和分析消息，我们将在接下来的例子中说明这一点。

合而为一：重写RPC

下面的Erlang代码在主机STC上叫做bar的节点建立了一个阶乘服务器：

```
setup() ->
    spawn('bar@STC',myrpc,server,[]).

server() ->
    register(facserver,self()),
    facLoop().

facLoop() ->
    receive
    {Pid, N} ->
        Pid ! {ok, fac(N)}
    end,
    facLoop().
```

服务器接收到格式为{Pid, N}的消息，并且把结果{ok, fac(N)}发送回Pid。在Java中，下面的示例代码可完成同样的事情：

```
1 import com.ericsson.otp.erlang.*; // For the JInterface package
2 import java.math.BigInteger;       // For factorial calculations
3
4 public class ServerNode {
5
6     public static void main (String[] _args) throws Exception{
7
8         OtpNode bar = new OtpNode("bar");
9         OtpMbox mbox = bar.createMbox("facserver");
10
11         OtpErlangObject o;
12         OtpErlangTuple msg;
13         OtpErlangPid from;
14         BigInteger n;
15         OtpErlangAtom ok = new OtpErlangAtom("ok");
16
17         while(true) try {
18             o = mbox.receive();
19             msg = (OtpErlangTuple)o;
20             from = (OtpErlangPid)(msg.elementAt(0));
21             n = ((OtpErlangLong)(msg.elementAt(1))).bigIntegerValue();
22             OtpErlangObject[] reply = new OtpErlangObject[2];
23             reply[0] = ok;
24             reply[1] = new OtpErlangLong(Factorial.factorial(n));
25             OtpErlangTuple tuple = new OtpErlangTuple(reply);
26             mbox.send(from,tuple);
```

```

27
28     }catch(OtpErlangExit e) { break; }
29 }
30 }

```

在前面的例子中，并发特征以粗体字在第8、9、18和26行上显示，其余的代码用来分析、解构和重构数据值，以及提供循环控制。

通过运行节点**bar**且在该节点上运行进程**facserver**，启动主程序。在主循环的第18~26行，一条消息被接收并回复。收到的消息是一个Erlang项元，即一个**OtpErlangObject**。在第19行有一个**OtpErlangTuple**类型转化，并从中可以获取发送者的进程标识符（第20行）和所发送的整数（第21行）。在第21行中，Erlang的值作为一个长整数被获取，但被换成Java的**BigInteger**，以便能够精确计算阶乘。

代码的其余部分（第22~26行）构造了回复元组并将其发送。第22行构造一个对象数组，包含基元**ok**（第23行）和返回值**factorial(n)**（第24行）。第26行中在最后发送回客户端之前，把它转化成一个元组（第25行）。

互动

与运行Java的节点进行互动，你可以使用下面的代码在终端中调用**myrpc:f/1**：

```

-module(myrpc).
...
f(N) ->
    {facserver, 'bar@STC'} ! {self(), N},
    receive
        {ok, Res} ->
            io:format("Factorial of ~p is ~p.~n", [N,Res])
    end.

```

客户端代码和用来与Erlang节点进行交互的代码是完全一样的，并且在节点之间发送与接收消息的“图灵测试”（注3）将无法分清Java节点和Erlang节点的差异。

小细节

在本小节中，我们将介绍如何让**JInterface**的程序在你的计算机上正常运行。

注3：图灵测试曾经是由数学家和计算机先驱阿兰·图灵（1912—1954）作为机器智能测试提出来的。这个想法翻译为现代技术就是，一个测试人员与两个“人”在线聊天，其中一个是人，另外一个机器，如果测试人员不能可靠地确定谁是人谁是机器，那么就可以说该机器具有智能。

首先，为了建立和管理Java和Erlang节点之间的连接，当创建一个节点时，运行epmd（Erlang Port Mapper Daemon）是有必要的。你应该还记得第11章的epmd。你可以简单地输入epmd（Windows系统中是epmd.exe）来运行它，你可以通过输入以下命令来测试它是否已经运行：

```
epmd -names
```

这将列出在主机上运行的所有Erlang节点的名称。此命令可用于检查你认为应该运行的节点实际上是否在运行。

如果节点启动时没有提供cookie，那么该系统将使用默认的cookie来创建一个节点。这可能行得通，但是如果你需要创建一个带有指定cookie的节点，那么就需要使用以下命令：

```
OtpNode bar = new OtpNode("bar", "cookie-value");
```

如果需要使用一个特定的端口，那么它就是有三个参数的构造函数的第三个参数。

回到“合而为一：重写RPC”一节的程序，该程序的第1行已经导入JInterface Java代码，但由于它包含在OTP发行包中，而不是标准Java，这里有必要给Java编译器和运行环境指出它在哪里，具体如下：

```
<otp-root>/jinterface-XXX/priv/OtpErlang.jar
```

在前面的代码中，<otp-root>是发行包的根目录，在运行节点上输入code: root_dir()可以得到，其中XXX是版本号。在Mac OS X中的完整路径是：

```
/usr/local/lib/erlang/lib/jinterface-1.4.2/priv/OtpErlang.jar
```

这个值通过下列方式提供给编译器：

```
javac -classpath ".:usr/local/lib/erlang/lib/  
jinterface-1.4.2/priv/OtpErlang.jar" ServerNode.java
```

对Java系统而言：

```
java -classpath ".:usr/local/lib/erlang/lib/  
interface-1.4.2/priv/OtpErlang.jar" ServerNode
```

更上一层楼

该JInterface库有比你迄今为止所看到的更丰富的功能：

- 使用OtpMbox中的link和unlink函数可以与Java进程连接和断开。

- 这个实例依赖于在节点之间的自动连接。你可以在节点上使用ping函数来测试一个远程节点是否存在，如果存在，这个连接就自动建立。
- 使用二进制可以在节点之间传送任意数据，这是由OtpErlangBinary类操纵的。
- OtpConnection类提供了RPC的更高层的机制，正如rpc模块为Erlang所做的一样。
- OtpConnection类的方法还支持跟踪控制。

在线文档对这些以及其他特性有详细描述。

C节点

erl_interface库提供了C语言一端的功能，包括构建、操作和访问Erlang二进制项元的C编码。还包括用于处理项元构造（和析构）时的内存分配、访问全局名称和报告错误的功能。具体细节如下：

`erl_marshall`、`erl_eterm`、`erl_format`和`erl_malloc`

处理Erlang项元格式，包括内存管理。特别是它们提供了类似Erlang项元的C结构之间的转换，也允许更高级别的数据操作。

`erl_connect`和`ei_connect`

通过分布式Erlang节点提供与Erlang的连接。

`erl_error`

打印错误信息。

`erl_global`

提供对全局注册名称的访问。

`registry`

提供存储和备份键-值对的功能。这就提供了ETS表的一些功能，并且可以备份到或恢复来自在相连的Erlang节点上的Mnesia表。

此外，Erlang外部项元格式是作为字节序列的Erlang项元的一种表达方式：二进制。使用内置函数`term_to_binary/1`和`binary_to_term/1`，你可以在Erlang中的这两种表达方式之间转换。在“端口程序”一节中，我们将会更详细地讨论这个内容。

在本节中，我们将重写已经给出的Java阶乘服务器实例，这次是采用C语言，它是基于Erlang的在线互操作性教程的例子：

```
1 /* fac.c */
2
3 #include <stdio.h>
```

```

4 #include <sys/types.h>
5 #include <sys/socket.h>
6 #include <netinet/in.h>
7
8 #include "erl_interface.h"
9 #include "ei.h"
10
11 #define BUFSIZE 100
12
13 int main(int argc, char **argv) {
14     int fd; /* file descriptor of Erlang node */
15
16     int loop = 1; /* Loop flag */
17     int got; /* Result of receive */
18     unsigned char buf[BUFSIZE]; /* Buffer for incoming message */
19     ErlMessage emsg; /* Incoming message */
20
21     ETERM *fromp, *argp, *resp; /* Representations of Erlang terms */
22     int res; /* Result of the fac call */
23
24     /* initialize erl_interface (once only) */
25     erl_init(NULL, 0);
26
27     /* initialize the connection mechanism */
28     if (erl_connect_init(1, "mycookie", 0) == -1)
29         erl_err_quit("erl_connect_init");
30
31     /* connect to a running Erlang node */
32     if ((fd = erl_connect("blah@STC")) < 0)
33         erl_err_quit("erl_connect");
34
35     while (loop) {
36         /* message received */
37         got = erl_receive_msg(fd, buf, BUFSIZE, &emsg);
38
39         if (got == ERL_TICK) {
40             /* ignore */
41         } else if (got == ERL_ERROR) {
42             loop = 0;
43         } else {
44             if (emsg.type == ERL_REG_SEND) {
45                 /* unpack message fields */
46                 fromp = erl_element(1, emsg.msg);
47                 argp = erl_element(2, emsg.msg);
48
49                 /* call fac and send result back */
50                 resp = erl_format("{ok, ~i}", fac(ERL_INT_VALUE(argp)));
51                 erl_send(fd, fromp, resp);
52
53                 /* free the term storage used */
54                 erl_free_term(emsg.from); erl_free_term(emsg.msg);
55                 erl_free_term(fromp); erl_free_term(argp);
56                 erl_free_term(resp);
57             } } }
58

```

```

59 int fac(int y) {
60     if (y <= 0)
61         {return 1;}
62     else
63         {return (y*fac(y-1));};
64 }

```

C代码的一般形态类似于前面的Java节点，但C代码有更多底层操作，如下所示：

- C库的引用代码（第3~6行）和Erlang接口的引用代码（第8和9行）；
- 输入缓冲区的内存分配（第11和18行）；
- 释放C代码中分配给Erlang项元的存储空间（第53~56行）。

在第24~33行建立并连接一个Erlang节点：

- `erl_init(NULL, 0)`初始化`erl_interface`库，在任何程序中只须调用一次。
- `erl_connect_init(1, "mycookie", 0)`初始化连接机制，包括该节点的识别号码（这里是1）与所使用的cookie。
- `fd = erl_connect("blah@STC")`连接到Erlang节点`blah@STC`并返回连接的一个文件描述符。

在第35~57行中的`loop`将会永远循环下去，可以使用下面一行代码来读取一条消息（第37行）：

```
got = erl_receive_msg(fd, buf, BUFSIZE, &emsg);
```

上面的一行代码将收到缓冲区`buf`中的一条消息，并且把它解码成一个Erlang项元`emsg`。

（第40行）忽略检查该节点是否还有效的`ERL_TICK`消息，并且在收到`ERL_ERROR`消息时，该循环（第42行）终止。否则，`loop`循环体的功能部分将执行下列操作：

- 提取消息发件者的进程标识符`fromp`和载荷`argp`（第46和47行）。
- 把`argp`转换成C语言整数，将其传递给阶乘函数，并把Erlang项元`{ok, fac(...(argp))}`返回到`fromp`进程（第51行）。`erl_format`调用使用字符串格式以一种可读的方式构建Erlang项元。若要手动地构建同样的项元，可以进行如下操作：

```

arr[0] = erl_mk_atom("ok");
arr[1] = erl_mk_integer(fac(ERL_INT_VALUE(argp)));
resp = erl_mk_tuple(arr, 2);

```

- 最后，释放Erlang项元和子项元使用的存储空间（第54~56行）。

为了编译这个C程序，你必须确保`erl_interface.h`文件和`liberl_interface.a`和`libei.a`库已经存在。你可以使用以下命令做到这一点（在Mac OS X系统中）：

```
gcc -o fac -I/usr/local/lib/erlang/lib/erl_interface-3.5.9/include \
-L/usr/local/lib/erlang/lib/erl_interface-3.5.9/lib fac.c -lerl_interface -lei
```

在上面的代码中，斜体字的目录给出了你的系统上最新版本`erl_interface`的路径。它把C代码编译成当前目录中可执行的`fac`。

下面给出连接到C节点的Erlang代码。一般来说，C节点的名称是`cN`，其中N是该节点的标识数字，所以在这里我们使用`c1@STC`：

```
-module(fac).
-export([call/1]).

call(X) ->
  {any, 'c1@STC2'} ! {self(), X},
  receive
    {ok, Result} ->
      Result
  end.
```

在Erlang终端上调用上面对代码，结果如下：

```
% erl -sname "blah" -setcookie "mycookie"
..... at this point the C executable should be called .....
(blah@STC2)1> c(fac).
{ok,fac}
(blah@STC2)2> fac:call(7).
5040
(blah@STC2)3> fac:call(0).
1
```

在这个例子中，C节点作为一个客户端，Erlang节点必须首先启动，所以当C节点尝试连接到Erlang节点时，它已经在运行了。

扩展阅读

你刚看到的C节点是作为客户端运行的：它可以连接到Erlang节点。它也可以以服务器模式运行，这首先需要该程序创建一个套接字——在特定的端口号码上监听——然后利用`epmd`程序发布这个套接字。这个程序也可以接受来自Erlang节点的连接。

`erl_interface`库提供了其他各种不同的工具，如针对传入消息的模式匹配、用于存储键-值数对的注册系统以及一个全局命名方案。所有这些再加上服务器风格的节点，在`erl_interface`的互操作性教程和用户指南里有详细介绍。

Unix终端的Erlang调用：erl_call

“在OTP中隐藏的宝石”之一是`erl_call` Unix命令，可以使用`erl_inter`来提供与一个分布式Erlang节点的通信。此外，也为了启动节点并且与之通信，你可以使用这些从命令行中编译和对Erlang求值。可以读取`stdin`的功能允许其他脚本使用，如在CGI bin中的那些。

你可以通过一系列标志来展示这个命令的参数和选项。完整的集合在手册页面里有描述，或者通过无标志的命令`erl_call`得到一个概要。其中`-n`、`-name`或者`-sname`的标志是必需的，因为这些标志用于指定所调用的节点的名称（或简称）。通常，`-s`标志也伴随着一起使用，如果节点尚未运行的话，那么它将启动这个节点。

标志`-a`类似于带有类似格式的参数的`apply/3`，其中标记`-e`是计算来自标准输入的内容（一直到`Ctrl-D`为止）。以下是`erl_call`命令的具体操作过程：

```
% erl_call -s -a 'erlang date' -n blah
{2009, 3, 21}
% erl_call -s -e -n blah
X=3,
Y=4,
X+Y.
Ctrl-D
{ok, 7}
% erl_call -a 'erlang halt' -n blah
%
```

端口程序

Erlang端口允许在Erlang节点与外部程序之间进行通信，这是通过运行在节点上的Erlang进程——所谓的端口已连接的进程——和外部程序之间发送或接收二进制信息，外部程序运行于操作系统的一个单独的线程中（见图16-1）。如第14章所述，`wxErlang`捆绑到`wxWidgets`需要使用的端口。

端口最简单的应用程序之一是`os:cmd/1`函数，该函数可以在Erlang终端内调用一个操作系统命令：

```
1> os:cmd("date").
"Sat 21 Mar 2009 18:11:24 GMT\n"
```

这使得端口表现得像不捕获退出的Erlang进程一样。已连接的进程可以链接到它，并发送和接收Erlang的消息和退出信号。二进制通信底层的机制依赖于操作系统，例如，在Unix的系统上是通过管道通信。在Erlang一方基于端口互动的模板如下所示：

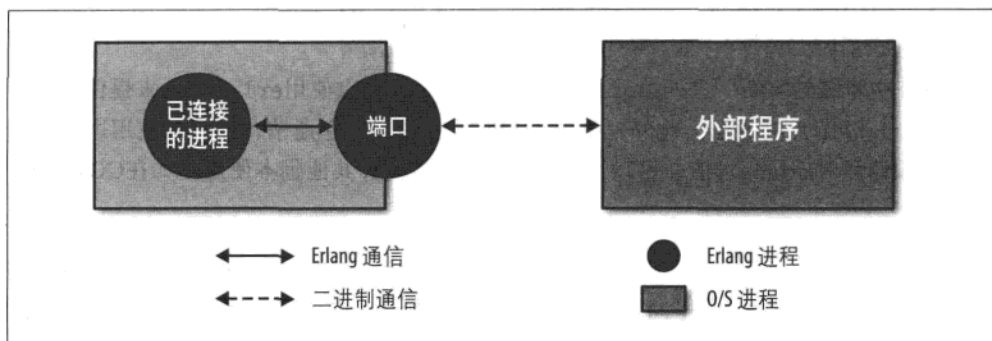


图16-1: Erlang端口及其所连接的进程和外部程序

```
Port = open_port({spawn, Cmd}, ...),
...
port_command(Port, Payload),
...
receive
  {Port, {data, Data}} ->
...

```

在这段代码中，通过调用`open_port/2`打开端口，它返回端口标识符`Port`。在这个已连接的进程中，通过调用`port_command(Port,...)`把数据发送到`Port`（和外部程序中），并且在以`{Port, {data, Data}}`格式匹配数据的`receive`语句中接收来自`Port`的数据。

我们刚刚介绍了端口如何工作的顶层摘要。在本节的剩下部分将会更详细地讲解Erlang命令，这些命令用于与外部程序通信，以便控制端口和针对数据编码和解码的方式。最后，我们将向你展示如何采用Ruby和C语言编写外部程序来通过端口与Erlang通信。

Erlang端口命令

若要打开一个Erlang端口，可以使用命令`open_port/2`：

```
open_port({spawn, Cmd}, Options)
```

这将把命令`Cmd`作为外部程序运行，外部程序是由给定的`Options`列表产生的。下面是主要的可用选项清单（在`erlang`模块文档中，你可以找到完整清单）：

`{packet, N}`

这里给出了此端口所使用的二进制数据包的大小。`N`可以采用值1、2或4。根据这个选项，其数据包前面是数据包大小。你如果需要发送可变大小的数据包，那么应该使用选项`stream`代替。

binary

从端口来的所有I/O产生二进制数据对象，而不是字节。

use_stdio

这将使用（在Unix系统中）标准输入和输出与所生成的进程进行通信，为了回避这种情况，可使用选项`nouse_stdio`。

exit_status

在外部程序退出时，这可确保把消息发送到该端口，详情可参阅联机文档。

例如，要运行一个Ruby程序`echoFac.rb`，需要执行下面的命令：

```
Cmd = "ruby echoFac.rb",  
Port = open_port({spawn, Cmd}, [{packet, 4}, {use_stdio, exit_status, binary}]),
```

执行这些命令后，变量`Port`包含所生成的Ruby程序的端口标识符。

所连接的进程可以使用`port_command/2`与`Port`进行通信。在所连接的进程中执行下面的命令将会把数据发送到`Port`：

```
port_command(Port, Data)
```

所发送的消息具有这样的格式`{Port, {data, Data}}`。

警告： 端口标识符如`Port`可以让任何Erlang进程访问该端口，并且也可以由此访问附属于该端口的外部程序。任何进程都可以使用这一点，但强烈建议不要直接使用`Port!...`进行通信！如果从不是端口所有者的任何其他进程调用它都将会导致失败，所有的通信应该使用`port_command/2`。

为了把进程标识符为`Pid`的进程连接到端口`Port`，必须执行下面的调用：

```
port_connect(Port, Pid)
```

这可以由任何进程调用，但是旧的端口拥有者将仍然保持与端口的连接，拥有者将需要调用`unlink(Port)`来删除这个链接。

要关闭一个端口 —— 从而终止与外部程序的通信 —— 所连接的进程需要执行命令`port_close(Port)`。

为了具体说明这些例子，下面是一个计算23的阶乘的Erlang小程序，通过把参数发送到Ruby并且在Ruby程序`echoFac.rb`中进行计算，这是以`erlectricity`库的`echo`实例为基础，它将在“通信支持库”一节中讨论。

在下面的例子中，**粗体字**是通信的基础代码：

```

-module(echoFac).
-export([test/0]).

test() ->
  Cmd = "ruby echoFac.rb",
  Port = open_port({spawn, Cmd}, [{packet, 4}, use_stdio, exit_status, binary]),
  Payload = term_to_binary({fac, list_to_binary(integer_to_list(23))}),
  port_command(Port, Payload),
  receive
    {Port, {data, Data}} ->
      {result, Text} = binary_to_term(Data),
      Blah = binary_to_list(Text),
      io:format("~p~n", [Blah])
  end.

```

运行函数test/0，其结果如下：

```

1> echoFac:test().
"23!=25852016738884976640000"
ok
2>

```

在“通信支持库”一节中，我们将会解释该程序的其余部分。

端口发送或接收的通信数据

通过端口的通信使用二进制数据，因此，在通信之前需要通过某种方式把程序数据转换成二进制形式，然后在接收端解码。在这里可以使用第9章介绍的位语法，因为可以使用大量erlang模块提供的编码和解码函数，如下所示：

term_to_binary/1

根据Erlang二进制项元格式对参数编码，从而把参数转换为二进制数据对象。如果以选项binary创建端口，那么这些数据可以通过Port进行通信。

binary_to_term/1

这是term_to_binary/1的逆向函数。

list_to_binary/1

这将返回一个由参数里的整数和二进制数组成的二进制结果。例如：

```

> list_to_binary([<<1,2,3>>,1,[2,3,<<4,5>>],4|<<6>>]).
<<1,2,3,1,2,3,4,5,4,6>>

```

binary_to_list/1

这是list_to_binary/1的逆向函数。

回到前面的例子：

```

test() ->
  Cmd = "ruby echoFac.rb",
  Port = open_port({spawn, Cmd}, [{packet, 4}, use_stdio, exit_status, binary]),
  Payload = term_to_binary({fac, list_to_binary(integer_to_list(23))}),
  port_command(Port, Payload),
  receive
    {Port, {data, Data}} ->
      {result, Text} = binary_to_term(Data),
      Blah = binary_to_list(Text),
      io:format("~p~n", [Blah])
  end.

```

粗体字的行显示了两个方向的转换。在创建Payload时，`integer_to_list`把整数23转换为字符串"23"，然后转化为二进制<<"23">>，这就与基元`fac`搭配成一对，这一对被编码为二进制。

在`receive`语句中，以标准形式`{Port, {data, Data}}`从Port中接收这个消息。把Data解码成为格式`{result, Text}`的项元，然后可以使用`binary_to_list`来解码Text本身。这允许把数据输出到终端。

文件描述符、端口以及I/O

想象一个Erlang节点，它接收嵌入XML的由内联驱动程序解析的HTTP POST消息。如果开始你每秒接收数以千计的请求，那么你很快就会达到允许同时打开文件描述符的限制。

在你启动脚本（或在运行Erlang进程的环境中）时，运行命令`ulimit -n Max`，其中Max是允许同时打开文件描述符的最大数目。默认值和最大值都依赖于操作系统。请记住，每个端口是由两个文件描述符所组成的，一个用于读而另一个用于写。

另一个优化是，当开始启动模拟器时，把标记`+A Size`传递给命令`erl`。通过增加虚拟机中处理文件I/O的异步线程数，这将提高应用程序的速度。Size是一个整数，介于默认值0~1024。

通信支持库

正如“端口程序”一节的例子所示，自己去编码和解码数据是相当繁琐的。当然，你可以灵活地选择在你的外部程序和Erlang节点之间有效率的通信协议，但为每个程序发明一套编码并不是解决问题的最好方法。

如果你不希望走这条路，那么Erlang提供了与外部世界通信的库，特别是C和Java。在本章开头我们已经讨论了Java库，在本章的剩余部分，我们将讨论与Ruby语言的接口。

在Ruby中工作：erlectricity

erlectricity是一个Ruby库——一个“宝石”——你可以下载并和Ruby一起使用。它的源代码来自<http://github.com/mojombo/erlectricity>，你可以在Ruby中通过如下命令安装：

```
$ gem install mojombo-erlectricity -s http://gems.github.com
```

通过在文件的头部使用require声明，你可以在Ruby程序中包括erlectricity库。

这个软件包的核心是receiver.rb程序，它提供了Ruby程序接收和发送消息的功能。这也取决于port.rb的端口实现和matcher.rb里的匹配；对于各种不同的Erlang类型，编码和解码数据格式由encoder.rb和decoder.rb提供。

这个库带有一个测试套件和例子，还包括把Erlang连接到Campfire的一个后台程序，这是通过Campfire API的Ruby实现Tinder完成的。

使用erlectricity的实例

Ruby的erlectricity库支持通过端口与Erlang进程通信。以下是该系统的Ruby一方，该程序echoFac.rb与echoFac.erl通信：

```
require 'rubygems'
require 'erlectricity'
require 'stringio'

def fac n
  if (n<=0) then 1 else n*(fac (n-1)) end
end

receive do |f|
  f.when(:fac, String) do |text|
    n = text.to_i
    f.send!(:result, "#{n}!=#{(fac n)}")
    f.receive_loop
  end
end
```

这个程序的工作部分是revice函数：在接收消息f时，与{fac, text}进行匹配，其中text是一个字符串。如果匹配成功，那么通过调用to_i函数，把text转换为一个整数，并且把下面的消息发送到发送该消息的端口：

```
{:result , "#{n}!=#{(fac n)}"}
```

在Ruby字符串中，结构`#{...}`包含了所要计算的表达式。例如，如果变量`n`的值为6，那么`"#{n}!=#{(fac n)}"`将是字符串`"6!=720"`。

正如我们前面所说的，运行`echoFac:test()`的结果是：

```
1> echoFac:test().
"23!=25852016738884976640000"
ok
```

内联驱动程序和FFI

在默认情况下，连接到Erlang的外部程序将在一个单独的操作系统进程里运行。这样可以隔离系统的两个部分，以便外部程序的崩溃不会影响Erlang程序，但它的缺点是，难以满足某些更底层的实时性的需求。为了满足这些需求，可以把外部程序在与Erlang系统同样的线程里运行，这就是所谓的内联驱动程序。

建立内联驱动程序以及这样的驱动程序需要实现的回调函数可以通过在线互操作性教程中的第6章了解这些细节。与内联驱动程序的通信使用端口，并且与在端口程序中使用相同的内置函数。

警告：当使用内联驱动程序时，你的程序将会执行得非常快。但是，这一速度是要付出代价的，因为当事情出错的时候，它们会变得非常糟糕。内联驱动程序的崩溃或内存泄漏将导致Erlang虚拟机崩溃。端口恰恰与之相反，它将关闭端口并且发送EXIT信号到所连接的进程。

因此使用内联驱动程序需要非常小心，应该尽可能让它们保持简单，并且只有在性能是关键问题时才把它们整合进来——例如，当和一个外部系统如Berkeley数据库集成或集成系统调用时，比如YawsWeb服务器上的`sendfile`。

Erlang增强提案（注4）（Erlang Enhancement Proposal, EEP）流程是一种机制，它方便Erlang社区提出语言的改进提案，并且把提案融入到标准发行版本中。EEP7是外部函数接口（Foreign Function Interface, FFI）的一个提案，它保证了开发更可靠的内联驱动程序。与此同时，还存在一些工具包，如Erlang驱动程序工具包（Erlang Driver Toolkit, EDTK）和Dryverl，其目的都是为了支持Erlang语言的内联驱动程序开发。

注4： 网址：<http://www.erlang.org/eeeps/>。

练习

练习16-1：使用端口的C语言阶乘

使用端口而不是分布式Erlang节点，用C语言编写前面阶乘服务器实例的实现。

练习16-2：另一种语言的阶乘服务器

使用以下语言之一的接口：Scheme、Haskell、C#、Python或者PHP，重新实现阶乘服务器实例。



跟踪内置函数，dbg跟踪器 以及匹配规则

任何一个有影响力的程序设计语言，如果要使其数百万行的代码可以在全球成千上万套设备上顺利运行，就必须提供内置的底层跟踪机制，用于生成实时的调试。不提供这种机制的语言会给开发人员和系统工程师带来巨大的负担，因为他们要么必须自己从头开发这一框架，要么在黑盒环境里调试他们的系统。

在Erlang中，爱立信跟踪实时电话交换机的经验反映在跟踪内置函数上。从成为这个语言第1版的一部分开始，跟踪内置函数经过这些年的发展，已经逐渐成为能够提供系统变化状态完全可视化的一整套工具的基础，也因此大大减少了解决错误的时间和调试的工作量。

引言

想象一下，当模式匹配调用`ets:lookup(msgQ,MsgId)`的结果时，你将从一个实时系统收到一个“badmatch”错误报告。你希望程序可以模式匹配一个表示消息类型的基元，然而当它碰到元组`{error,unknown_msg}`的时候就终止了。

你可以很快确定已损坏的数据库，然后着手寻找写入ETS表的`ets:insert/2`调用。由于进入系统的消息在系统的边界进行测试，因此这条消息一直到调用`insert`时才会产生。你可以添加一个`case`语句，确保从未有错误元组插入表中，但这样做会被看做是防御性编程，因为此消息要么标记不正确，要么首先就不应该这么深入系统。不仅如此，你也不知道是哪个特定的`ets:insert/2`调用引发了这个问题。在大型系统中，你将会找到非常多的相关调用。

在一个庞大且复杂的系统中，如果不知道在哪个模块里损坏了记录，你可能不得不通过搜索数百万行的代码来找到元组`{error,unknown_msg}`。就算你找到了元组，插入`io:format/2`语句来打印错误及进程信息也解决不了问题，因为在实时系统中，进程来来去去，每小时ETS表中都要插入和删除数百万条的记录。此外，由于这是一个有严格版本

控制的实时系统，代码变更必须在部署之前经过测试和批准。即使这个选项很具有吸引力，但它会导致漫长的周转周期。然而，不要为质量保证小组的流程和缓慢的周转时间而紧张，因为你可以做得更好。

在Erlang中，开发人员或技术支持工程师考虑要做的第一件事情就是为所有的`ets:insert/2`函数调用打开跟踪功能。你可以同时跟踪局部和全局调用——也就是调用同一模块和其他模块里的函数——而不需要跟踪编译（即重新编译）代码。跟踪事件可以包括函数调用本身、函数的参数和结果、该调用的调用函数和一个时间戳。启用跟踪后，每当跟踪函数被调用时你都可以产生一个跟踪事件，这个跟踪事件要么在终端里输出，要么通过管道输送到一个套接字上，在这个套接字的另一端的程序接收它，将它格式化并存储在一个可读的日志文件里。

但仍然有一个问题，因为在一个实时系统里，每小时都会有数百万次对函数`ets:insert/2`的调用。数以百万计的跟踪消息会给你带来很多不必要的信息，并且可能会影响性能。而你只对那些第二个参数（即消息类型）是无效的跟踪事件感兴趣。那么你应该怎么办呢？你可以实现一个匹配规则，对每一次调用都会检查传递的参数。你可以定义这样的匹配规则：只有当你的参数检查到元组`{error,unknown_msg}`时才生成一个跟踪事件，而不是对每个有效消息类型都产生。在这个跟踪事件中，你确保调用函数被显示出来，它可以告诉你`ets:insert/2`调用是在哪儿产生的。然后，你可以开始跟踪这个调用函数，检查它的参数，找到关于无效消息来源的最终线索，最后定位错误。更重要的是，在既不影响性能也不用重新编译代码的条件下，你可以在一个有着数以千计的同步事务处理同时执行的实时系统里完成所有的事情。

如果这一切听起来好得不够真实，请继续阅读。在局部和全局调用中检查参数，基于特定的触发器来打开和关闭进程的跟踪，以及对进程状态改变的完全可视化，这些都可能发生且不给开发者带来额外的负担。在Erlang中，你可以使用`dbg`跟踪器工具来得到这些，这个工具以用户友好的方式封装了跟踪内置函数以及匹配规则。这是另外一个简单却功能强大并且精心设计的架构例子，它可以减少开发和技术支持的难度以及错误的周转时间。

本章中的第一节涵盖了底层的跟踪内置函数，以及在Erlang中基于消息跟踪的基础。接下来的部分会介绍建立在内置函数上的`dbg`跟踪工具。也许`dbg`跟踪工具才是你真正需要的，但是第一节会向你介绍在Erlang中如何进行跟踪，以及用于跟踪的术语。

跟踪内置函数

内置函数`erlang:trace/3`用于启用和禁用Erlang运行时系统中的底层跟踪机制。它提供了一种用来监控并发、代码运行以及内存使用情况的手段。像调试器和进程管理器这样

的工具使用这个内置函数来收集和显示跟踪事件。因为它不是自动载入，所以你需要在调用时使用包含它们的erlang这个模块名称作为前缀。

跟踪内置函数的优点在于不必再跟踪编译代码。你可以在实时系统里面使用，得到一个令人难以置信的强大的故障诊断工具，从而使所发生的事件完全可视化。

跟踪事件将作为消息用以下格式发送：

```
{trace,Pid,Tag,Data1[,Data2]}
```

这里的[,Data2]表示依赖跟踪消息类型的可选字段。生成的跟踪事件如下所示：

- 全局和本地的函数调用
- 垃圾收集和内存使用情况
- 进程有关的活动和消息传递

在任何时候，只有一个进程可以接收到另一个进程的跟踪事件。这个进程被称为跟踪器进程（tracer process），接收跟踪事件的跟踪器进程是调用了以下函数的进程：

```
erlang:trace(PidSpec,Bool,TraceFlags).
```

在这个调用中，PidSpec定义了你想要跟踪的进程，它可以是一个进程标识符或者是基元existing、new和all中的任意一个：

- existing基元为所有存在的进程启动跟踪，但跟踪器进程和所有在调用后生成的进程排除在外。
- 如果你传递一个new，那么这个调用将会跟踪在跟踪内置函数进程后产生的所有进程。
- all基元将会对除跟踪进程自身以外的所有进程进行跟踪，无论它们是在跟踪调用之前还是之后创建的。

调用erlang:trace(self(),Bool,TraceFlags)会产生一个坏参数（bad argument）错误，因为跟踪器进程不能被跟踪。接收跟踪消息的进程不会被跟踪是为了避免无限循环。设想一下，你正在跟踪接收到的消息。每次你的进程收到一条消息，将会生成一条跟踪消息，这将导致生成另一条跟踪消息。而且，这个过程的速度没有程序能够控制。

如果你想要其他进程而不是调用trace/3内置函数的那个进程来接收跟踪事件，可以把元组{tracer,Pid}当做TraceFlags列表里的一个元素传递（我们将会简单说明）。在这个元组里，Pid必须是一个进程标识符或者一个端口标识符。

跟踪内置函数的第二个参数是基元`true`或`false`，用来指定你是否要启用或者禁用跟踪的某个特定属性。这些都在一个跟踪标志列表里用基元（注1）描述，并且根据第二个参数的布尔值来指定你要生成或禁止哪些跟踪事件：`true`则生成，`false`就禁止。我们在随后的内容中会详细描述跟踪标志。`erlang:trace/3`调用返回值是一个表示被跟踪进程的数量整数。

进程跟踪标志

`send`标志跟踪这个进程发送的所有消息，而当添加消息到跟踪进程的信箱里时，`'receive'`会生成事件。因为在Erlang中`receive`是一个保留字，你必须把它括在单引号里面，这样就产生一个基元。设置`send`和`'receive'`标志为`true`产生以下跟踪事件：

```
{trace,Pid,send,Message,To}
{trace,Pid,send_to_non_existing_process,Message,To}
{trace,Pid,'receive',Message}
```

到目前为止，你一定非常希望能在终端中试试跟踪内置函数。在下面的部分中，我们将使用这个程序作为例子：

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

start()->spawn_link(ping,loop,[]).

send(Pid)->
  Pid ! {self(),ping},
  receive pong-> pong end.

loop()->
  receive
    {Pid,ping}->
      spawn(crash,do_not_exist,[]),
      Pid ! pong,
      loop()
  end.
```

`start`函数生成一个在接收计算（receive-evaluate）循环里等待的子进程。只要接收到一条`ping`消息，就发送调用`ping:send/1`的结果。因为执行子进程的模块并不存在，所以该子进程产生一个新的进程会立即异常终止。

这种异常的终止将会生成一份崩溃报告并输出。（请记住，生成一个进程永远不会失败（注2）。相反，将会失败的是那个最新生成的进程，因为它应当执行的是未定义的函

注1： 所有的跟踪标志，除了在上一段中讲到的元组`{tracer,Pid}`外，剩下的全都是基元。

注2： 除非你超过了系统允许的最大进程数。

数。)最后,子进程发送回一个pong消息作为响应。调用ping:send/1接收这个消息并且作为函数的结果返回。

在终端运行前面的例子,并且打开所有send和receive事件的跟踪,我们可以得到:

```
1> Pid=ping:start().
<0.55.0>
2> erlang:trace(Pid,true,[send,'receive']).
1
3> ping:send(Pid).
pong
=ERRORREPORT====6-Sep-2008::19:16:00===
Error in process <0.40.0> with exit value: {undef,[[{crash,do_not_exist,[]}]]}

4> flush().
Shell got {trace,<0.55.0>,'receive',{<0.39.0>,ping}}
Shell got {trace,<0.55.0>,send,pong,<0.39.0>}
ok
5> erlang:trace(Pid,false,[send,'receive']).
1
```

请注意第2行和第5行如何返回整数1,即正在被跟踪的进程的数目。你可以在任何时候对进程设置跟踪标志:要么在进程创建前使用all或new标志,要么在进程创建后使用all、existing标志或者该标志的进程标识符。由于跟踪消息将被发送到终端进程(即执行trace/3调用的进程),你可以用终端指令flush/0读取它们。

在任何特定时间,一个进程可以处于三种状态之一(见图17-1)。它可能正在执行(running)代码、可能被暂停(suspended)等待执行的时机,也可能在一个receive语句中等待(waiting)一个消息的到来。当一个进程的运行状态从执行变为暂停时,称为被抢占(preempt),而将它从暂停队列移动到准备执行队列时,称为被调用(schedule)。

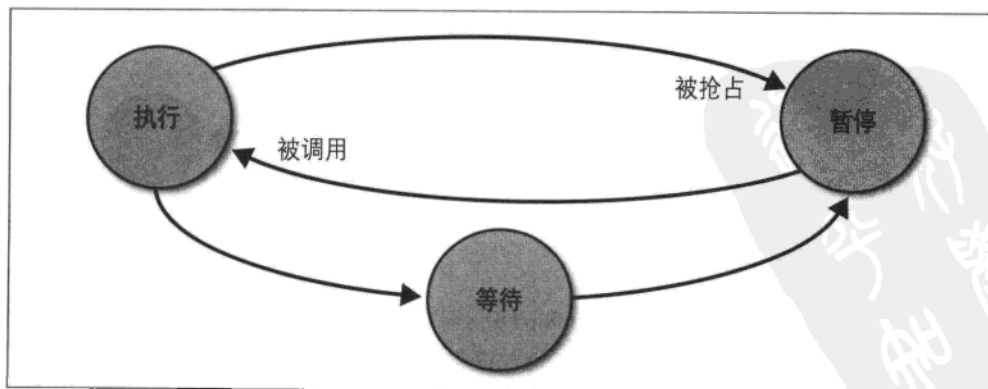


图17-1: 进程状态

打开执行标志，你就可以从进程开始（in）到消亡（out），全程跟踪模块的状态转变、函数、元数和该进程进程标识数。跟踪事件总是成对产生，并保持以下格式：

```
{trace,Pid,in,{M,F,Arity}}
{trace,Pid,out,{M,F,Arity}}
```

进程管理器Pman用procs标志跟踪与进程有关的事件，例如进程的生成和中止、进程连接和注册。procs标志产生的跟踪事件包括：

```
{trace,Pid,spawn,Pid2,{M,F,Args}}
{trace,Pid,exit,Reason}
{trace,Pid,link|unlink,Pid2}
{trace,Pid,getting_linked|getting_unlinked,Pid2}
{trace,Pid,register|unregister,Pid2}
```

当然，也可以用调用标志跟踪函数调用，我们会在“用trace_pattern内置函数跟踪调用”一节中详细讨论。

继承标志

使用set_on_spawn标志的规定：任何子进程将继承父进程的标志，包括set_on_spawn标志本身（见图17-2）。因此，子进程产生的任何进程也将继承子进程的所有标志。相反，如果你使用set_on_first_spawn标志，你就指定了所有产生的进程将只继承除set_on_first_spawn外的标志。子进程产生的任何进程将不会继承该标志。换句话说，set_on_spawn具有传递性，而set_on_first_spawn不具有。

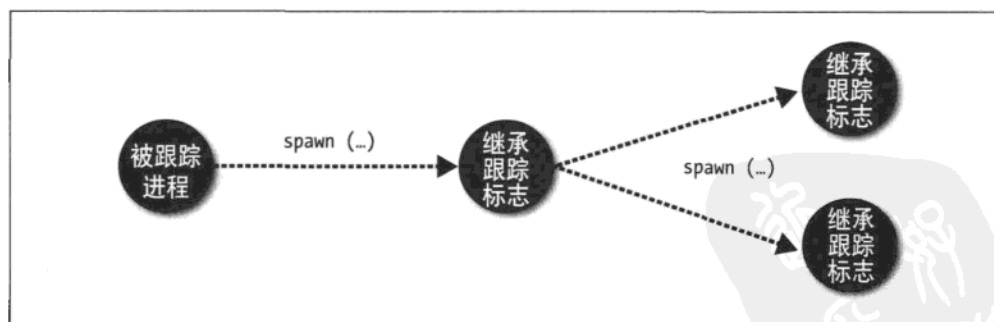


图17-2：set_on_spawn跟踪标志

让我们用ping例子来看set_on_spawn和procs标志是如何起作用的。发送ping消息到进程，它会产生一个立即终止的子进程。追踪进程会从第一个进程收到生成跟踪事件，还会从第二个进程收到带有undef运行错误的退出消息：

```

1> Pid=ping:start().
<0.31.0>
2> erlang:trace(Pid,true,[set_on_spawn,procs]).
1
3> ping:send(Pid).
pong
=ERROR REPORT===6-Sep-2008::19:50:33===
Error in process <0.40.0> with exit value: {undef,[{crash,do_not_exist,[]}]}}

4> flush().
Shell got {trace,<0.31.0>,spawn,<0.34.0>,{crash,do_not_exist,[]}}
Shell got {trace,<0.34.0>,exit,{undef,[{crash,do_not_exist,[]}]}}
ok

```

跟踪标志`set_on_link`和`set_on_first_link`与`set_on_spawn`标志类似，但它们控制的是连接发生时继承的标志。

垃圾收集和时间戳

内存使用和垃圾收集所花费的时间通常难以预测。在第2章中我们说过，Erlang系统使用的是世代性的垃圾收集机制。收集器有两代数据：“当前的”与“旧的”。从一次垃圾收集幸存下来的“当前的”数据会变成“旧的”。这个方法的前提是许多存储在堆中的数据是短暂的，而且不会在第一次垃圾收集存活下来；反之，在第一次收集保存下来的数据将趋向于保持更长时间的寿命。因此，在当前的数据中比在旧的数据中进行更频繁的垃圾收集更有意义。

使用`garbage_collection`标志，你可以接收到与垃圾收集开始和终止有关的跟踪事件。产生这些事件的格式如下：

```

{trace,Pid,gc_start,Info}
{trace,Pid,gc_end,Info}

```

这里的`Info`是一个带标签的元组的列表，它包含以下这些：

`heap_size`

当前堆使用的部分。

`heap_block_size`

用来存储堆和栈的内存块大小。

`old_heap_size`

旧堆的使用部分。

`old_heap_block_size`

用于存储旧堆的内存块大小。

`stack_size`

实际栈大小。

`recent_size`

从上一次垃圾收集集中幸存下来的数据大小。

`mbuf_size`

消息缓冲区（或邮箱）的大小。

带标签的元组中提供的所有堆和消息缓冲区的大小都以字为单位（注3）。

现在，如果你想计算垃圾收集所花费的时间，或者你想精确记录一个特定跟踪事件产生的时间，你应该怎么做？你应该使用时间戳标志，它将为所有的消息增加一个时间戳。其格式和内置函数`now()`返回的格式相同，即`{MegaSeconds,Seconds,Microseconds}`，时间从1970年1月1日开始。所有跟踪消息现在都有一个`trace_ts`标签，并在结尾处包含一个额外的字段来写入时间戳：

```
1> Pid=ping:start().
<0.31.0>
2> erlang:trace(Pid,true,[garbage_collection,timestamp]).
1
3> ping:send(Pid).
Pong
=ERROR REPORT====6-Sep-2008::20:30:41===
Error in process <0.40.0> with exit value: {undef,[{crash,do_not_exist,[]}]}
```

```
4> flush().
Shell got {trace_ts,<0.31.0>,gc_start,
  [{mbuf_size,0},{recent_size,0},{stack_size,2},{old_heap_size,0},
   {heap_size,231}],{998,578633,990000}}
```

```
Shell got {trace_ts,<0.31.0>,gc_end,
  [{mbuf_size,0},{recent_size,228},{stack_size,2},{old_heap_size,0},
   {heap_size,228}],{998,578633,990001}}
```

`cpu_timestamp`标志使所有跟踪的时间戳相对于CPU时间，而不是挂钟时间。要使用此标志，目标计算机需要支持高精度的CPU测量，在跟踪内置函数里面的`Pid`参数也必须是“all”。

用`trace_pattern`内置函数跟踪调用

你可以用内置函数`erlang:trace_pattern/3`启动局部和全局函数调用的跟踪。你必须连

注3： 如果你记不起来什么是我们所指的堆和旧堆了，你可以复习一下第2章的“Erlang内在管理”部分。我们在第3章介绍了进程的栈。

同内置函数`erlang:trace/3`一起使用这个内置函数，并且使用`call`和`return_to`标志来调用。这将会在这两个调用产生的集合的交集上启动跟踪（见图17-3）。采用`trace/3`，你可以定义要监视哪些进程，而用`trace_pattern/3`，你可以定义要跟踪的函数的子集。只有被跟踪进程执行被跟踪函数时，才会产生一个事件。

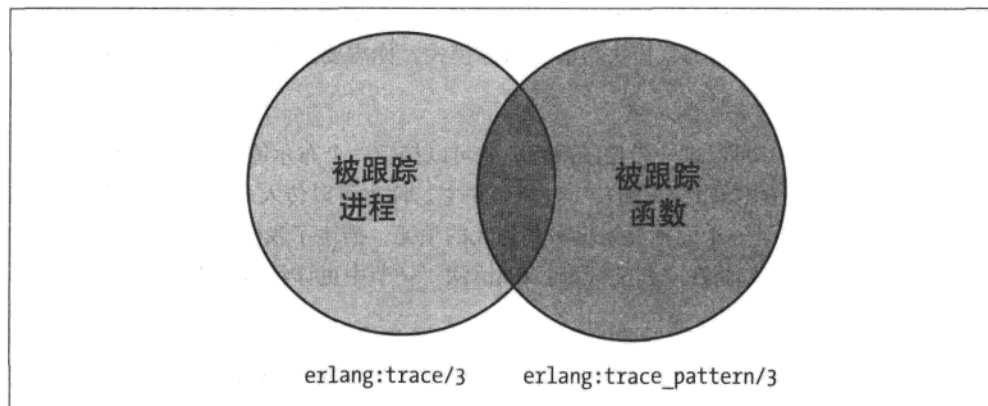


图17-3：内置函数`trace`和`trace_pattern`的交集

由这两个内置函数的交集所产生的跟踪消息具有以下格式：

```
{trace,Pid,call,{M,F,Args}}  
{trace,Pid,return_to,{M,F,Args}}
```

其中当以参数`Args`调用模块`M`中的函数`F`时，生成`call`。当调用返回并且执行的函数完成时，生成`return_to`事件。当跟踪递归调用时，只有在递归的终止条件满足时才把`return_to`事件发送出去，而不是每一个函数的循环都发送。你只能和`call`标志一起使用`return_to`标志。

和`call`标志一同使用`arity`标志，所有包含一个`{M,F,Args}`元组的事件标签都将直接返回函数的元数信息，即`{M,F,Arity}`。如果你对传递给一个函数的参数不感兴趣，并且想减少作为调用跟踪的结果而传递的消息的大小，这种方式是非常有用的。

当调用下列代码时：

```
erlang:trace_pattern(MFA,Condition,FlagList)
```

你可以在第一个参数里定义你想跟踪的函数，其中可以指定模块、函数以及你想要产生一个跟踪事件的调用元数。这个MFA的格式可以是下面其中之一：

```
{Module,Function,Arity}  
{Module,Function,'_'}
```

```
{Module, '_', '_'}  
{ '_', '_ ', '_ '}
```

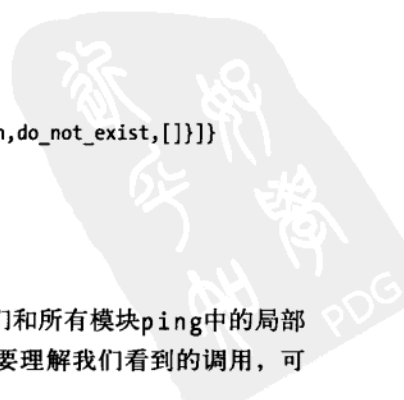
通配符允许使用基元'_'，因此{Module, '_', '_'}将启动在模块里定义的所有任意元数的函数。但是通配符也要遵循前面的模式，类型组合{'_', Function, '_'}是不允许的，我们传递给函数erlang:trace_pattern/3的模块必须在调用内置函数之前加载。设置标志后重新编译代码或重新加载模块将清除跟踪模式。你需要重新运行跟踪命令内置函数来重新启用跟踪。

条件 (Condition) 可以进一步控制跟踪。它可以接受一个布尔值：赋值为true将启用在MFA中定义的所有函数跟踪，而false将禁用它。你也可以传入一个匹配规则 (match specification)，它是一个同样丑陋但极其强大的项元，描述了我们在第10章中讨论过的一个简单的程序。我们将在“匹配规则：fun语法”一节中更详细地讨论它们。

最后，FlagList定义了跟踪函数的类型，进一步过滤了在MFA中定义的东西。传入全局 (global) 则只跟踪导出函数，而传入局部 (local) 则导出和非导出的函数都跟踪。

当把Erlang与其他编程语言相比较，通过这些非常强大的内置函数获得的对你的系统的可视性，将大大减少开发补丁的周转时间。你是否很好奇，也很想体验从终端调用跟踪函数呢？让我们来看一个例子，体会传递给内置函数的可能的范围值。这么做的同时，不要忘记在内置函数调用前面加上erlang模块名字定义内置函数调用。让我们回过头来看看在“继承标志”小节中解释过的ping的例子：

```
1> l(ping).  
{module,ping}  
2> erlang:trace(all,true,[call]).  
25  
3> erlang:trace_pattern({ping, '_', '_'},true,[local]).  
5  
4> Pid=ping:start().  
<0.120.0>  
5> ping:send(Pid).  
pong  
=ERROR REPORT====4-Apr-2009::19:33:25===  
Error in process <0.122.0> with exit value: {undef, [{crash, do_not_exist, []}]}
```



```
6> flush().  
Shell got {trace,<0.120.0>,call,{ping,loop,[]}}  
Shell got {trace,<0.120.0>,call,{ping,loop,[]}}  
Ok
```

在这个例子中，跟踪所有进程的每一个函数调用。我们把它们和所有模块ping中的局部函数调用相交。请记住，此选项包含局部和非局部的调用。要理解我们看到的调用，可以给ping.erl模块中的命令行标上数字：

```

1 -module(ping).
2 -export([start/0,send/1,loop/0]).
3
4 start()->spawn_link(ping,loop,[]).
5
6 send(Pid)->
7   Pid ! {self(),ping},
8   receive pong-> pong end.
9
10 loop()->
11   receive
12     {Pid,ping}->
13       spawn(crash,do_not_exist,[]),
14       Pid ! pong,
15       loop()
16   end.

```

我们跟踪的函数是：`start/0`、`send/1`和`loop/0`，在模块里面也有其他的函数调用（例如第4行的`spawn_link/3`和第7行的`self/0`），但是这些都是在调用不在`ping`模块里定义的函数。

调用`ping:start()`有什么效果呢？它将以`Pid`作为进程标识符启动一个执行`loop/0`的进程，这个进程会第一次调用在被跟踪的模块的一个函数，生成第一条跟踪消息并且显示在命令第6行`flush()`的结果里。这个调用是全局调用，因为它作为内置函数`spawn/3`的结果是从模块外部调用的。

当我们接着调用`send(Pid)`的时候，第7行代码会发送一个消息到`Pid`（这也将顺带包含一个对`self/0`的调用），这个消息在`loop/0`的主体中被处理。在接收到这条消息后，它将调用`spawn/3`，最后调用第15行的`loop/0`。对`loop/0`的调用是一个局部调用，当执行`flush()`时，它将产生第二个跟踪消息。

如果你在命令第3行中用`global`替代`local`，然后进行如下相同的跟踪，你将期待全局调用会是怎样的呢？

```
erlang:trace_pattern({ping,'_', '_'},true,[global]).
```

执行相同的测试你会得到一条跟踪消息，这是内置函数`spawn/3`所产生的结果。但如果你跟踪所有进程的所有全局调用，为什么`ping:send/1`没有产生一个跟踪事件呢？这毕竟是一个全局调用。原因是：不能跟踪接收跟踪消息的进程。如果你想为调用跟踪内置函数的进程产生跟踪事件，你必须把这些消息重定向到另一个进程。你可以这么做，通过传递选项`{tracer,Pid}`给内置函数`trace/3`来指定你想发送消息的目标进程的标识符。

dbg跟踪器

你可能意识到，尽管跟踪内置函数非常强大和方便，但是它们也太底层了，并且对用户不是很友好。毕竟，它们为建立诸如dbg跟踪器之类的其他工具提供了一个基础。

dbg跟踪器是一个基于文本的调试器，它给内置函数trace和trace_pattern提供了一个友好的用户界面，但是使用了“用trace_pattern内置函数跟踪调用”一节引入的跟踪原则和机制。你可以把dbg工具用作“进程管理器”小节中我们讨论过的进程管理器的一个补充，尤其是当你在基于文本的终端上跟踪，而你又不具备访问显示的权限或你得转移输出时。由于dbg工具对系统性能的影响很小，它已经成为跟踪大型实时系统的恰当选择。

dbg入门

dbg:h()调用为你提供了一个有用函数的清单。想要找出这些函数中任何一个的更多信息，你可以向dbg:h/1传递它们的名称。假设你们正在加紧调试一个实时系统，当半夜里的一个技术支持电话过后，你最不想做的事情就是去查看Erlang手册。帮助函数总能派上用场：

```
1> dbg:h().
The following help items are available:
  p, c
    - Set trace flags for processes
  tp, tpl, ctp, ctpl, ctpg, ltp, dtp, wtp, rtp
    - Manipulate trace patterns for functions
  n, cn, ln
    - Add/remove traced nodes.
  tracer, trace_port, trace_client, get_tracer, stop, stop_clear
    - Manipulate tracer process/port
  i
    - Info

call dbg:h(Item) for brief help a brief description
of one of the items above.

ok
2> dbg:h(p).
p(Item) -> {ok, MatchDesc} | {error, term()}
  - Traces messages to and from Item.
p(Item, Flags) -> {ok, MatchDesc} | {error, term()}
  - Traces Item according to Flags.
  Flags can be one of s,r,m,c,p,sos,sol,sofs,
  soft,all,clear or any flag accepted by erlang:trace/3

ok
```

dbg:p(PidSpec,TraceFlags)调用允许指定想要跟踪的进程，以及你想要它们产生哪些跟踪事件。PidSpec可以是以下的其中一个：

Pid

一个特定的进程ID。

all

将跟踪所有进程，无论它们是在调用调试器之前还是之后产生的。

new

将跟踪在调用调试器后产生的所有进程。

existing

将跟踪在调用调试器前存在的所有进程。

Alias

一个注册进程的别名，除all、new或existing之外。

{X,Y,Z}

指示由进程ID <X.Y.Z>所表示的进程，也可以用"<X.Y.Z>"，它是内置函数pid_to_list的结果。

TraceFlags是一个单一的基元或标志列表。你可以传递跟踪内置函数所能接受的任何标志，包括下列标志的缩写和组合：

s

跟踪发送消息。

r

跟踪接收消息。

m

跟踪发送和接收消息。

p

跟踪与进程有关的事件。

c

根据dbg:tp/2调用里设定的跟踪模式来跟踪全局和局部调用。

sos和sofs

指示set_on_spawn和set_on_first_spawn标志，我们已经在“继承标志”小节中提到过。

sol和sofl

指示set_on_link和set_on_first_link标志，我们已经在“继承标志”小节中提到过。

all

设置所有的标志。

clear

清除所有设定的跟踪标志。

你肯定再次渴望尝试dbg跟踪器和生成跟踪事件。在下面的例子中，我们将跟踪来自ping模块的所有与消息相关的事件，ping模块在“继承标志”小节中已经介绍过。

我们启动ping进程和一个单独的跟踪进程。这个跟踪进程将接收和显示所有的跟踪事件，包括那些在终端里面产生的进程。在命令第5行，我们为ping进程发送和接收的所有消息打开跟踪事件，之后调用函数ping:send/1。在之后的跟踪输出里，你可以看到，发送一个ping消息后，终端进程马上就回复了一个pong消息。

```
3> Pid = ping:start().
<0.41.0>
4> dbg:tracer().
{ok,<0.43.0>}
5> dbg:p(Pid, m).
{ok,[{matched,nonode@nohost,1}]}
6> ping:send(Pid).
pong
(<0.41.0>) << {<0.29.0>,ping}
7> (<0.41.0>) <0.29.0> ! pong
7>
=ERROR REPORT==== 6-Sep-2008::21:40:31 ===
Error in process <0.47.0> with exit value: {undef,[{crash,do_not_exist,[]}]}

7> dbg:stop().
ok
```

从第6行的跟踪输出可以看出，括号内的进程就是跟踪事件起源的地方。在跟踪信息中由标志<<指示，进程<0.41.0>接收到消息{<0.29.0>,ping}，然后发送一个pong的消息给进程<0.29.0>做出响应。

可以用这个例子试验不同的跟踪标志，这样做的时候，注意如何启动和停止跟踪器。是不是在用dbg:stop()停止跟踪器后，在dbg:p/2中设置的标志就会自动清除呢？

使用相同的ping进程，现在我们用set_on_spawn标志追踪所有和进程相关的活动。因此，进程<0.55.0>继承了所有的标志，结果是它生成了退出跟踪事件：

```
8> dbg:tracer().
{ok,<0.51.0>}
9> dbg:p(Pid, [p, sos]).
{ok,[{matched,nonode@nohost,1}]}
10> ping:send(Pid).
pong
(<0.41.0>) spawn <0.55.0> as crash:do_not_exist()
```

```

=ERROR REPORT=== 6-Sep-2008::21:43:26 ===
Error in process <0.55.0> with exit value: {undef, [{crash, do_not_exist, []}]}

(<0.55.0>) exit {undef, [{crash, do_not_exist, []}]}
{ok, [{matched, nonode@nohost, 1}]}
11> dbg:stop().
ok

```

请注意，在前面的两个例子中，没有记录关于跟踪进程本身的活动的跟踪信息。并且，如果你还没弄清楚，那么`dbg:stop/0`将不会清除跟踪模式。为此，同样需使用`dbg:stop_clear/0`清除跟踪标志。

跟踪和性能分析函数

函数`dbg:c(Mod, Fun, Args, TraceFlags)`是在终端执行跟踪和性能分析函数的理想选择。如果省略`TraceFlags`参数，将会设置`all`标志。在下面的例子中，我们跟踪所有和`io:format/1`调用有关的活动，在这个例子里，它是`io:format("hello~n")`。

这个跟踪展示了导入/导出机制的内部运作，在这种运作环境下，消息发送给“组长”。当运行到`receive`语句时，调用进程被暂停，一旦“组长”的响应返回，将会调度这个进程继续执行。

```

1> dbg:c(io, format, ["Hello World~n"]).
Hello World
(<0.53.0>) <0.23.0> ! {io_request, <0.53.0>, <0.23.0>,
                        {put_chars, io_lib, format, ["Hello World~n", []]}}
(<0.53.0>) out {io, wait_io_mon_reply, 2}
(<0.53.0>) << {io_reply, <0.23.0>, ok}
(<0.53.0>) in {io, wait_io_mon_reply, 2}
(<0.53.0>) << timeout
ok

```

如果你想在某一个特定函数里监视内存的使用和垃圾收集所花费的时间，使用`dbg:c/3`是很理想的，因为它把调用隔离在一个单独的进程中。使用`set_on_link`和`set_on_spawn`标志来跟踪边际效应不是最好的办法，因为函数一旦返回就会清除所有的标志。

跟踪局部和全局函数调用

到目前为止一切顺利，但跟踪内置函数的一个真正强大也有可能是最常用的特性，就是它们能为局部和全局函数调用生成跟踪事件。在`dbg`模块中，你可以使用下列语句启用对全局调用的跟踪：

```
dbg:tp({Mod, Fun, Arity}, MatchSpec)
```

我们可以如下使用进行局部调用：


```
dbg:tpl({Mod, Fun, Arity}, MatchSpec)
```

你连同`dbg:p/2`调用一起使用上面的语句，其中用`c`标志一同指定要跟踪的进程。通过这两个集合的交集产生的跟踪事件与在“用`trace_pattern`内置函数跟踪调用”一节中提到的相同。

与跟踪内置函数一样，你可以定义`Module`、`Function`和`Arity`为`'_'`，但是其中`{'_', Function, '_'}`的格式是不允许的，因为通配符出现在一个非通配符的前面将会很快涉及`dbg`的匹配规则，因此我们将使用`[]`。

`tp/2`和`tpl/2`返回值的格式是`{ok, Matches}`，其中`Matches`是一个元组列表，用来报告已经启用跟踪的Erlang节点，以及在每一个节点上启用跟踪的函数数量。

用`dbg:ctpg({Mod, Fun, Arity})`可以禁用调用跟踪。调用这个函数可以禁用函数跟踪而不管调用是局部的还是全局的。这也是一个可能会经常使用的调用。然而如果你只想禁用一个使用`dbg:tp/2`设置的特定全局调用模式上的跟踪，那么可以使用`dbg:ctpg({Mod, Fun, Arity})`：禁用局部跟踪可以使用`dbg:ctpl({Mod, Fun, Arity})`。

所有用来启用和禁用函数跟踪的调用都会返回一个格式为`{error, Reason}`或`{ok, MatchDescription}`的元组。`MatchDescription`是一个格式为`{matched, Node, Number}`的元组列表，其中`Number`代表在特定节点上启用或禁用多少函数调用。在下面的例子里面你可以清楚地看到这点。

另外请注意我们现在如何在终端生成跟踪事件。在“用`trace_pattern`内置函数跟踪调用”一节的使用跟踪内置函数的例子中，不能启用终端上的跟踪，因为终端本身就是那个接收跟踪事件的进程。现在消息发往不在终端中的跟踪进程，因此，诸如`start/0`和`send/1`的全局调用所产生的跟踪事件也变得可见了：

```
1> dbg:tracer().
{ok,<0.100.0>}
2> dbg:p(all,[c]).
{ok,[{matched,nonode@nohost,25}]}
3> dbg:tp({ping, '_', '_'}, []).
{ok,[{matched,nonode@nohost,5}]}
4> Pid = ping:start().
<0.105.0>
(<0.97.0>) call ping:start()
(<0.105.0>) call ping:loop()
5> ping:send(Pid).
(<0.97.0>) call ping:send(<0.105.0>)
pong
=ERROR REPORT==== 7-Sep-2008::12:47:07 ===
Error in process <0.107.0> with exit value: {undef,[{crash,do_not_exist,[]}]}

6> dbg:ctpg({ping, '_', '_'}).
{ok,[{matched,nonode@nohost,5}]}
```

你应该花一些时间熟悉跟踪器及其界面，因为这是Erlang实时分布式中可用的最强有力的工具之一。当阅读dbg的手册时，你会发现很多函数tp、tpl、ctp、ctpl和ctog的不同版本。除非你喜欢使用快捷方式，否则你不用担心它们。要记清楚我们在本章讨论过的函数可能已经很辛苦了，更不用说所有它们的变形。好消息是我们讨论过的函数都是你所需要的，它们允许你去表达所有其他变体中覆盖的组合。这是一个权衡：拥有很好的记忆力并记住所有的变形，还是额外输入一些字符。而作为备用，所有你要求助的是dbg:h()。

警告： 当在跟踪实时系统时，使用dbg跟踪器要特别小心。如果在一个繁忙的系统里面启用太多的跟踪事件，可能会有生成太多终端输出的风险，以致连输入必要的停止跟踪的命令都变得不可能了。结果，跟踪器进程不能跟上产生的进程消息。它们排队等待，而且Erlang的实时系统很快会耗尽内存。支持工程师（还有一位是这本书的作者，注4）曾由于dbg而引起美国固定电话和移动数据网络的服务中断。在一个安全的环境中你可以随意使用跟踪，但是当调试一个实时系统时，一定要确定你知道自己在做什么，十分小心地使用跟踪。

分布式环境

跟踪可以在分布式环境中进行，这要把所有的跟踪输出都重新定向到一个单独的跟踪器。当你启用或者清除一个跟踪时，你一定从例子中已经注意到调用返回值的格式如下：

```
{ok,[{matched,node@nohost,5}]}.
```

由于我们没有运行分布式节点，所有的局部跟踪都在节点node@nohost上。在一个分布式环境中运行跟踪时，你反而会获得一个包含所有跟踪被启用的节点的列表。若跟踪器不能和一个远程节点连接，这个特殊节点的结果会是：`{matched,Node,0,RpcError}`。

调用dbg:n(Node)可以增加一个分布式的Erlang节点到跟踪列表上，而dbg:cn(Node)可以将这个节点删除。要列出你正在上面运行跟踪的节点，可以用dbg:ln()。一旦节点被添加，设置跟踪项、标志和函数的模式将会影响所有被跟踪节点。传给dbg:p/2的进程标识符也可以在其他节点上。dbg:p/2唯一没有处理的事是全局注册的名字。

重定向输出

到现在为止，在我们看过的所有dbg跟踪器的例子中，跟踪事件发送给一个跟踪进程，然后这个进程会将事件格式化并且在终端输出。这也许并不总是一个处理调试消息最好

注4： 我们已经把它留作练习，让你来找出这是哪一个作者负责的。

的方法。相反，你可能想要收集统计数据、测量垃圾收集或者把输出转发到一个套接字或文件中。

这里有一个好消息是，dbg允许你定义自己的fun函数来处理由跟踪内置函数产生的跟踪消息。如果你用dbg:tracer(process,{HandlerFun,Data})启动一个跟踪器工具，所有的进程会作为参数传递给HandlerFun，它是一个用户定义的二元fun函数。第一个参数是跟踪消息，第二个是用户定义的Data。这个fun返回的数据会传给它的下一个迭代，因此可以把这个值看做一个累加器，例如它可以用来了解总资源的使用情况。

在下面的例子中，我们将会监控终端进程的内存使用情况。我们的HandlerFun会在开始收集垃圾的时候保存内存的使用数据，并且提供是分配还是释放内存的信息。我们通过运行一些内存密集型应用来测试跟踪器。一个delta正值意味着释放当前堆里和旧堆中的内存，而一个负值表示分配内存。所有的大小都用字来表示。注意在当前堆中释放内存时会有多少内存存在旧堆中被分配。这将是寿命更长的数据，它在当前堆的垃圾收集中幸存下来，并且转移到旧堆中。

下面的例子很长，任何拼写错误都将导致你不得不从头开始输入。比较聪明的办法是：用一个编辑器输入这个例子，然后粘贴到终端中，要么从本书的网站下载：

```
1> HandlerFun =
  fun({trace, Pid, gc_start, Start}, _) ->
    Start;
    ({trace, Pid, gc_end, End}, Start) ->
      {_, {_,OHS}} = lists:keysearch(old_heap_size, 1, Start),
      {_, {_,OHE}} = lists:keysearch(old_heap_size, 1, End),
      io:format("Old heap size delta after gc:~w~n",[OHS-OHE]),
      {_, {_,HS}} = lists:keysearch(heap_size, 1, Start),
      {_, {_,HE}} = lists:keysearch(heap_size, 1, End),
      io:format("Heap size delta after gc:~w~n",[HS-HE])
    end.
#Fun<erl_eval.12.113037538>
2> dbg:tracer(process, {HandlerFun, null}).
{ok,<0.32.0>}
3> dbg:p(self(), [garbage_collection]).
{ok,[{matched,nonode@nohost,1}]}
4> List = lists:seq(1,1000).
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,
 23,24,25,26,27,28,29|...]
Old heap size delta after gc:0
Heap size delta after gc:6020
5> RevList = lists:reverse(List).
[1000,999,998,997,996,995,994,993,992,991,990,989,988,987,
 986,985,984,983,982,981,980,979,978,977,976,975,974,973,972|...]
Old heap size delta after gc:-676
Heap size delta after gc:3367
```

重定向到套接字和二进制文件

前面介绍了如何编写一个自己的fun函数来处理跟踪消息。使用相同的机制，你可以将跟踪输出重定向到一个套接字或二进制文件中。这将又带来一个好处，就是可以减少跟踪节点的负载，尤其是作为终端中任何I/O的结果。正是这个I/O变得非常密集，从而导致正确输入停止跟踪器所需的命令都十分困难。

你如何重新定位输出呢？在`dbg:tracer/2`调用中，我们传递的不是在前面内容中展示的进程参数，而是基元Port。调用：

```
dbg:tracer(port, PortFun)
```

可以启动一个跟踪器，它将会把跟踪消息传递给一个Erlang的端口。PortFun是一个fun函数，封装被打开的端口，并由调用`dbg:trace_port(ip, Port)`返回，这里的Port既可以是端口号也可以是元组{PortNumber, QueueSize}。QueueSize限制了没有发送的消息队列的大小，如果远程套接字用户没有接收跟踪事件，那么将丢弃它们。此选项将打开一个监听端口、缓冲消息，一旦连接客户端就把它们发送出去。

要连接客户端，需调用`dbg:trace_client(ip, Arg)`，这里的Arg可以是端口号（如果跟踪器在相同的主机上运行），也可以是元组{HostName, PortNumber}（如果在不同的计算机上运行，当预计跟踪会带来沉重的负载时，这也是更好的选择）。这个调用将连接到跟踪运行的监听端口，并且检索跟踪事件。

如果你想把跟踪事件发送到一个二进制文件，那么需要把变量PortFun绑定到`dbg:trace_port(file, FileOptions)`的返回值。FileOptions描述了跟踪在文件系统中的存储方式。FileOptions可以定义一个文件名或一个交换文件规则，用来限制文件使用空间（wrap files specification）。在默认情况下，使用{FileName, wrap, FileSuffix}将产生8个交换文件，每个文件的大小为128KB。要修改默认值，可以使用{Filename, wrap, FileSuffix, WrapTrigger, WrapCount}，其中WrapTrigger是以千字节为单位的大小、或者是元组{time, Milliseconds}。而Suffix是文件后缀，WrapCount是在交换前创建的文件数量。

你可以调用`trace_client/2`检索消息，这里的第一个变量是基元file，用来读取到现在为止写入的所有跟踪事件；或者是follow_file来持续读取和处理在调用后写入的跟踪事件。第二个变量是FileName，它在你传给`dbg:trace_port/2`调用的FileOptions变量里被传入。

如果你想处理你自己的跟踪事件，而这些事件通过一个文件或一个套接字接收，那么你可以使用`dbg:trace_client(Type, Arg, {HandlerFun, Data})`调用。它会开启一个客户端，依次把HandlerFun应用到每一个到来的跟踪事件。Type和Arg与在先前提到的

trace_client/2中定义的一样，而{HandlerFun,Data}元组也和tracer/2中的相同。dbg:stop_trace_client(Pid)调用可以停止这个客户端，这里的Pid是tracer/2调用的返回值。

在接下来的例子中，我们开启了一个局部跟踪器，让它把跟踪事件重定向到IP端口为1234的在相同主机上运行的另一个Erlang节点。一个跟踪器客户端会接收事件并处理它们。在处理承载着实时流量的系统时，可以把你的跟踪事件重定向到一个文件，而你可以在另一台计算机上处理该文件；或者重定向到一个套接字，其中远程计算机上的客户端将接收消息流量，这是一个很好的办法：

```
1> PortFun = dbg:trace_port(ip, 1234).
#Fun<dbg.12.21848437>
2> dbg:tracer(port, PortFun).
{ok,<0.33.0>}
3> dbg:p(all, [c]).
{ok,[{matched,nonode@nohost,25}]}
4> dbg:tp({ping, '_', '_'}, []).
{ok,[{matched,nonode@nohost,5}]}
5> dbg:tpl({ping, '_', '_'}, []).
{ok,[{matched,nonode@nohost,5}]}
6> Pid = ping:start().
<0.39.0>
7> ping:send(Pid).
pong
=ERROR REPORT==== 14-Sep-2008::12:25:23 ===
Error in process <0.41.0> with exit value: {undef,[{crash,do_not_exist,[]}]}
```

输出被一个不同的Erlang终端的跟踪器客户端进程接收：

```
1> Pid = dbg:trace_client(ip, 1234).
<0.40.0>
2>(<0.30.0>) call ping:start()
(<0.39.0>) call ping:loop()
(<0.30.0>) call ping:send(<0.39.0>)
(<0.39.0>) call ping:loop()
2> dbg:stop_trace_client(Pid).
ok
```

匹配规则：fun语法

在第10章中我们介绍了一个强大的（但不美观的）匹配规则。你也许还能想起来，一个匹配规则由一个Erlang项元组成，它通过一个小程序来表述一个条件去匹配一系列的参数。匹配规则在功能上有限制，而在跟踪内置函数里使用，它们主要处理过滤和操纵跟踪事件。如果它们匹配成功，那么可以产生一个跟踪事件，并且执行事先定义好的程序。若把匹配规则编译成类似仿真器使用的格式，则使匹配规则比函数更有效率。但是，除了更加高效，规则编写起来也十分复杂，并且乍看起来，它们还很令人费解。

幸运的是，你可以使用`dbg:fun2ms/1`调用产生匹配规则，它涵盖了大部分简单且实用的情况。它把用`fun`语法定义的规则转为匹配规则，结果和手动写一个匹配规则一样有效，而且它们易于读写、修改和调试。我们使用`dbg:fun2ms/1`把匿名的`fun`转化为匹配规则，它在给局部和全局调用设定跟踪标志时使用。我们从介绍这个更高层的办法开始，但是对于那些需要利用跟踪内置函数全部功能的开发者，我们还是得遵从它们自己的匹配规则。

用fun2ms生成规则

还记得在本章开头描述的那个错误吗？就是那个我们用元组`{error,unknown_msg}`让ETS表损坏的错误？我们可以用下面的办法再次产生这个错误。首先，我们用`dp:fill()`调用来创建和损坏ETS表，调用函数`thedp:process_msg()`取用ETS表中的第一个元素并且处理表的时候会产生一个系统崩溃。在现实世界中，我们并不知道这些函数，所以我们必须依赖高层的测试，最终导致这些调用由一个工作进程产生。在这个例子里，我们从终端调用它们，因为这样便于分步演示那些用匹配规则解决错误的调试过程。

```
1> dp:fill().
true
2> dp:process_msg().
** exception error: no case clause matching [{2,{error,unknown_msg}}]
   in function dp:process_msg/0
```

错误消息将会立即告诉我们在`process_msg/0`调用中有一个`case`语句错误。利用在终端输出的异常错误，跟随函数`dp:process_msg()`的调用流程，我们可以立即发现在`case`语句中少了模式 `{_, {error, unknown_msg}}`：

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

process_msg() ->
  case ets:first(msgQ) of
    '$end_of_table' ->
      ok;
    Key ->
      case ets:lookup(msgQ, Key) of
        [{_, {event, Sender, Msg}}] ->
          event(Sender, Msg);
        [{_, {ping, Sender}}] ->
          ping(Sender)
      end,
      ets:delete(msgQ, Key),
      Key
  end.

event(_,_) -> ok.
```

```
ping(_) -> ok.
fill() ->
    catch ets:new(msgQ, [named_table, ordered_set]),
    dp:handle_msg(<<2,3,0,2,0>>).
```

记录{2, {error, unknown_msg}}不应该插入表中的第一个位置，因此我们假定case语句是正确的，而且增加{error, unknown_msg}也不能解决问题。我们需要找出是谁把它加入到表中，然后阻止它再次发生。损坏是由ets:insert/2调用产生的，但是因为系统中有很多ets:insert/2调用，我们通过创建一个匹配规则，在只有插入的元组元素是{error, unknown_msg}时产生了一个跟踪事件，以便查明究竟是哪一个。当这个匹配规则和模块的跟踪模式、函数以及参数一起使用的时候，任何时候只要带有这些参数的调用，都会触发一个跟踪事件。

函数dbg:fun2ms/1使用一个文字fun(literal fun)作为参数，并且返回一个匹配规则来描述fun函数里的属性。用“文字fun”在这里是指输入一个fun并且作为参数传递，而不绑定一个变量到fun函数，然后作为参数传递，或者这个fun是应用高阶函数得到的。我们也可以把这个叫做“明确的”或“明白的”fun。这个fun拥有一个参数，这个参数既不是变量也不是一个变量列表，所有这些可以被模式匹配或者用在保护元里。你可以在fun函数体里使用变量，但是最后一个表达式要么是一个转换为一个行动的预定义的调用，要么是一个会被忽略的项元素。

Erlang预编译器接受这个文字fun，然后把它翻译成一个匹配规则，你可以直接在终端中输入匹配规则，但是如果你要在模块里面包含它们，那你必须包含头文件ms_transform.hrl到文件中。这是标准库应用程序的一部分，最简便的包含方法如下：

```
-include_lib("stdlib/include/ms_transform.hrl").
```

有了这些知识，我们可以产生一个匹配规则，如果调用的第二个参数的格式是{_, {error, unknown_msg}}，当这个规则传递给dbg:tp/2函数时，就会触发一个事件。这包含调用ets:insert(msgQ, {1, {error, unknown_msg}}):

```
dbg:fun2ms(fun([_, {_, {error, unknown_msg}}]) -> true end).
```

你看出我们是如何在fun函数里面模式匹配的吗？忽略文字fun返回的基元true，而我们使用fun语法的模式匹配机制表示一个要求的模式。

让我们在终端中试试看。记住，你不能把fun绑定到一个变量上，它是一个由预编译器处理的文字fun表述，因此它必须作为一个参数显式地传递给调用。也要注意，db:fill/0调用只把一个错误的记录写入ETS表中，而它可以加入成千的记录，这些记录中的大多数都是正确的。作为我们实验的结果，我们只要生成一个跟踪事件，它在第二个参数的第二个元素是损坏我们表的元组时被触发。

```

3> dbg:tracer().
{ok,<0.58.0>}
4> Match1 = dbg:fun2ms(fun([_,_,{error, unknown_msg}]) -> true end).
[{{['_','_',{error,unknown_msg}}],[],[true]]]
5> dbg:tp({ets, insert, 2}, Match1).
{ok,[{matched,nonode@nohost,1},{saved,1}]}
6> dbg:p(all,[c]).
{ok,[{matched,nonode@nohost,25}]}
7> dp:fill().
true
(<0.54.0>) call ets:insert(msgQ,{2,{error,unknown_msg}})

```

现在我们知道调用进程的进程标识符，但是它没有让我们变得更聪明。由于为每一个进入的消息都创建进程，而且一旦这个消息开始排队，进程马上就会终止，因此没有跟踪它的节点。我们真正需要的是那个调用这个插入的函数（也就是调用者函数）。

幸运的是，我们可以通过让匹配规则产生一个包含调用函数信息的事件，以便得到这个信息。我们可以让fun返回一个预定义的文字函数的集合其中之一来实现这一点，所有这些函数由预编译器处理。在这种情况下，可以使用message(caller())：

```

dbg:fun2ms(fun([_,_,{error, unknown_msg}]) -> message(caller()) end)

```

文字调用message(Data)向跟踪器进程发送一个带Data的消息，使用display(Data)在终端中显示它。在这个例子中，我们像Data一样传递文字函数caller()。我们很快会简单介绍其他有效的数据选项，而现在，让我们聚焦在确定调用者上面，以及看它能否帮助我们解决这个bug：

```

8> Match2 = dbg:fun2ms(fun([_,_,{error, unknown_msg}]) ->
8>                                     message(caller())
8>                                     end).
[{{['_','_',{error,unknown_msg}}],[],[{message,{caller}}]}]
9> dbg:tp({ets, insert, 2}, Match2).
{ok,[{matched,nonode@nohost,1},{saved,2}]}
10> dp:fill().
true
(<0.34.0>) call ets:insert(msgQ,{2,{error,unknown_msg}}) ({dp,handle_msg,1})

```

注意：在ets:insert/2跟踪消息后，调用函数({dp,handle_msg,1})如何也被跟踪器显示出来。这是在我们的文字fun里调用message(caller())的结果。我们现在知道是dp:handle_msg/1函数以错误数据调用了ets:insert/2。查看代码，顺着调用流程，马上就能发现，{error,unknown_msg}元组在函数dp:handle/3中作为MsgType参数绝不是作为整数1或2的匹配结果产生：

```

handle_msg(<<MsgId, MsgType, Sender:16, MsgLen, Msg:MsgLen/binary>>) ->
  Element = handle(MsgType, Sender, Msg),
  ets:insert(msgQ, {MsgId, Element}).

```



```
handle(1, Sender, Msg) -> {event, Sender, Msg};
handle(2, Sender, _Msg) -> {ping, Sender};
handle(_Id, _Sender, _Msg) -> {error, unknown_msg}.
```

现在我们可以函数handle/3上运行一个跟踪，只在MsgType不是整数1或2的时候产生一个事件。我们在fun函数的头部绑定MsgType，并在保护元中对它进行测试。绑定变量只允许在函数头中进行：在你的匹配fun里面使用“=”会返回一个错误，并且匹配规则也不能编译成功。

我们可以很简单地添加一个保护元到文字fun函数：

```
dbg:fun2ms(fun([Id, Sender, Msg]) when Id /=1, Id /=2 -> trueend)
```

记住，保护元之间的分号表示至少有一个保护元必须成功，而逗号表示它们必须全部成功。在文字fun里面允许使用的保护元和常规的保护元一样。它们包含：

类型测试中使用的内置函数

```
is_atom、is_constant、is_float、is_integer、is_list、is_number、is_pid、
is_port、is_reference、is_tuple、is_binary、is_function、is_record
```

布尔运算符

```
not、and、or、and also、or else
```

关系运算符

```
>、>=、<、=<、:=、==、=/=、/=
```

算术运算符

```
+, -, *, div、rem
```

位运算符

```
band、bor、bxor、bnot、bsl、bsr
```

保护元里允许使用的其他内置函数

```
abs/1、element/2、hd/1、length/1、node/1、2、round/1、size/1、tl/1、
trunc/1、self/0
```

当在函数头部进行模式匹配时，或者因为在保护元操作里使用了无效的的参数时，如果一个参数错误导致了一个运行时错误，那么匹配就会失败。如果运行时错误发生在匹配规则体中，匹配规则会简单地返回基元'EXIT'。产生一个跟踪事件并且忽略任何的返回值。

使用不等于 (/=) 条件保护元可以在终端里测试无效的整数，我们将会得到解决bug的最后线索！

```

11> Match3 = dbg:fun2ms(fun([Id, Sender, Msg]) when Id /=1, Id /=2 -> true end).
[{'$1','$2','$3'},[{'/=',$1',1},{'/=',$1',2}],[true]]
12> dbg:tpl({dp, handle, 3}, Match3).
{ok,[{matched,node@nohost,1},{saved,3}]}
13> dp:fill().

(<0.44.0>) call dp:handle(3,2,<<>>)
(<0.44.0>) call ets:insert(msgQ,{2,{error,unknown_msg}}) ({dp,handle_msg,1})
true

```

查看前面的跟踪事件，我们现在知道dp:handle/3使用消息类型3调用，而发送器的硬件ID是2。看到在ID 2代表的硬件上软件做了修正，我们意识到软件不支持这个，从而导致数据被损坏。问题得到解决！

这个bug从哪里来

本小节中描述的崩溃也发生在我们的测试设备中。我们产生的跟踪事件存储在日志文件中，它允许我们去发现哪一个消息类型没有处理，以及追查那些硬件在起始时使用和例子中讲述的相似的一个策略。我们很快就会发现，原硬件的软件修改版本是比我们安装支持的版本更晚的一个发行版本。现实中有很多的替代方案，这个例子中的bug来自连接模块，它允许带有不支持的软件修改版本的硬件连接到系统。

紧跟在更正这个连接模块之后，我们删除在dp:handle调用中对未知消息的处理，也就是我们最初返回{error,unknown_msg}元组的地方。这是一种防错性编程的例子，它不应该首先被包括进来。可以让处理消息的进程以一个case语句错误终止运行，而终止只会影响那些处理消息的进程，其中这些消息源自不应该连接到系统的硬件。这会给我们一个较早问题的警报，以及一个错误发生在哪里的直接指示（连同硬件的ID和消息类型）。

你可以在文字fun里面包含调用，在预编译器解析变换时把它们翻译成动作。在这个例子中你已经看到一些动作了，但在这里，作为参考我们列出了所有的动作：

return_trace()

在完成跟踪调用时产生一个额外的事件，包括函数的返回值。当产生这个额外的跟踪事件时，跟踪函数调用的尾递归属性将丢失。

exception_trace()

和return_trace的行为方式一样，但是如果产生了一个运行时错误，将会产生一个exception_from消息。

`display(Data)`

把传递给它的数据打印输出产生一个边界效应。这个Data可以是在fun头部绑定的一个参数，也可以是本小节描述的其他文字函数中的一个返回值。

`message(Data)`

用Data产生一个跟踪事件。这个Data可以是在fun头部绑定的一个参数，也可以是本小节描述的其他文字函数中的一个返回值。

`message(false)`

是message/1的一个特例，它没有为call和return_to的跟踪标志产生跟踪事件。如果你对边界效应感兴趣，例如display/1文字函数产生的那些，那么这将很有用。

`message(true)`

是message/1的另一个特例，其中{Module, Function, Arity}的跟踪行为就如同没有匹配规则与它关联。它只用来覆盖message(Data | false)调用。

`enable_trace(TraceFlag)`

启用进程中的TraceFlag标志来触发匹配规则。每一次你只能传递一个标志，但是你仍然可以在fun函数体中用它们的跟踪标志进行多次调用。TraceFlag在跟踪中定义，被允许的跟踪器工具不使用缩写。在一个匹配规则里面执行这个调用和调用erlang:trace(self(), true, [TraceFlag])是等价的。

`enable_trace(Pid, TraceFlag)`

除了它会启用在Pid上的跟踪标志外，和enable_trace/1一样，Pid可以是一个进程标识符或一个注册的名称。

`disable_trace(TraceFlag)`

禁用一个特定进程中触发匹配规则的TraceFlag。enable_trace/1文字调用的限制同样适用于TraceFlag。

在一个匹配规则里面执行这个调用和调用erlang:trace(self(), false, [TraceFlag])是等价的。

`disable_trace(Pid, TraceFlag)`

和disable_trace/1一样，除了在指定的Pid上禁用TraceFlag。

trace(Disable, Enable)和trace(Pid, Disable, Enable)允许你同时启用和禁用许多标志。

`silent(true)`

关闭产生的所有调用跟踪消息，直到它自身里的一个匹配规则silent(false)匹配成功。

set_tcw(Int)

设置一个独一无二的跟踪控制字，你也可以使用`erlang:system_info(trace_control_word)`内置函数调用来读取它。这是一个被一些高级工具（比如说用户定义的）使用的自由使用状态值，允许这些工具根据它的值来影响采取的行动。

你可以把以下的调用当做参数传给`display/1`和`message/1`文字函数：

caller()

返回允许你标识调用函数的元组{Module, Function, Arity}。

get_tcw()

返回先前由`set_tcw/1`设置的跟踪控制字。

object()

返回一个包含所有传递给匹配规则参数的列表。

bindings()

返回一个包含所有在匹配头部中绑定变量的列表。

process_dump()

返回进程栈，该栈以二进制形式恰当地格式化为一个字符串。你可以把它当做参数传给`message/1`，其中你可以实现你自己的跟踪进程。把它传递给`display/1`不会成功，因为它不能输出二进制数据。

self()

返回调用进程的进程标识符。

函数fun2ms拾零

当使用`fun2ms`产生匹配规则的时候，你必须服从一些限制，因为匹配规则中的功能很有限。既然你知道了匹配规则是怎样工作的，这些限制也就更容易理解了。

你不能在`fun`函数体中调用Erlang函数。匹配主体中允许的保护元和函数将被翻译，这是因为即使编译正确，使用它们时还是会引入匹配规则返回错误。所有作为`fun`参数所定义的变量都被匹配规则变量以它们出现的顺序替代。你可以看一看那些我们已经生成的匹配规则，例如下面的这个`fun`头部：

```
fun([Id, Sender, Msg])when Id /=1, Id /=2 -> true end)
```

它翻译为`['$1', '$2', '$3']`。这些变量每出现一次都会在匹配规则的条件和函数体中被替换。因此在前面的例子中，保护元会翻译成一个格式为`[{'/=','$1',1},{'/=','$1',2}]`的条件，而像`is_integer(Id)`这样的保护元会翻译为一个类似`[{is_integer,'$1'}]`的条件。

在fun之外绑定以及那些没有在头部出现的变量，要么通过终端，要么通过产生匹配规则的函数。它们被翻译成格式为{const, Constant}的常量表达式，在这里，Constant是Constant的文字值。可以使用下列函数产生匹配规则：

```
foo(A) -> dbg:fun2ms(fun([B,C]) when B == A -> A end)
```

调用foo(10)将产生一个如下格式的规则：

```
[[{'$1','$2'}, [{'=='','$1',{const,10}}], [{const,10}]]
```

你只可以在fun的头部绑定变量，即使在那里，那样做也受到限制。例如，一个fun的格式为fun({A,[B|C]}=D)，允许其中的D在顶层和所有的变量绑定。但是fun({A,[B|C]}=D)是禁止的。在第一种情况下，如果你对输出或者返回所有参数感兴趣，你最好使用文字函数object()，它被翻译为'\$_'，可以成为一个扩展到所有参数的项。如果你只对变量的绑定感兴趣，并且你忽略那些无关紧要的变量，可以使用文字函数bindings()，在底层的匹配规则中它被翻译为'\$*'。

翻译文字项元结构是为了把它们变成有效的匹配规则，把元组转换为只有一个元素并且包含自身的元组，常数表达式则转换为{const, Constant}的格式。把记录 and 记录上的操作转换为元组和元素调用。把is_record/2保护元转换为{is_record, Var, Arity}，其中Var被格式为'\$0'的变量替代，而Arity是元组的大小。把列表模式匹配转换为使用{hd, List}和{tl, List}的结构。由于case、if和catch这样的条件结构在匹配规则中是不允许的，所以在fun中也禁用它们。

还记得吗？当处理一个模块中的fun2ms调用时，模块总是包含ms_transform.hrl头文件。如果不包含它，虽然代码可以成功编译却不会有任何警告，但是匹配规则不会被翻译，还有可能发生运行时错误。匹配规则的翻译在编译时完成，因此使用这些文字函数将不会影响运行时的性能。

ets和dbg匹配规则的区别

ets和dbg模块使用的匹配规则是不同的。如果触发翻译的伪代码函数是ets:fun2ms/1，那么fun的头部必须包含一个单独的变量，或者用一个保护元作为过滤器的单独元组。fun函数体以及匹配规则自身构成项元并返回select/2调用结果的值。这是在没有产生任何边界效应和绑定变量的情况下完成的。即使你可以在终端中使用ets匹配规则，通常情况下你也将发现它们内嵌在程序代码里。

另外，如果伪函数是dbg:fun2ms/1，那么fun的头部必须包含一个单独的变量或一个单独的列表。规则本身将会包含带来边界效应的强制命令，边界效应比如把自身作为输出显示，跟踪标志的操纵，或者添加多余的数据到跟踪事件中，产生规则的返回值将被忽

略。跟踪内置函数的规则通常的情况是在终端中使用，不过没有什么可以阻止你把它们集成到你的程序里。

连接这两者是对低负荷的高效过滤的需求，它不影响系统的实时特性。如果到现在为止你看到的东西还不足以让你感到惊讶，那么让我们来仔细地看一看这些匹配规则。给心脏脆弱的人一个忠告：我们将在“匹配规则：螺母和螺栓”一节中描述的东西尽管很强大，但是并不优美。

匹配规则：螺母和螺栓

当介绍帮助函数`dbg:fun2ms/1`时，我们对它的结果做了高层的解释，也就是匹配规则。规则是带有格式为`[Head, Conditions, Body]`的3个元素元组：

- *head*用于绑定及匹配变量和项元。
- 在*condition*中，逻辑测试应用到变量上。你可以定义你自己的逻辑测试或者使用预定义的保护元。
- 在规则的主体部分，我们列出了一个可能为空的预定义动作的集合，如果在头部匹配成功和全局条件满足，那么将执行这些动作。

本节的目的是确保你理解从`dbg:fun2ms/1`和`ets:fun2ms/1`调用产生的匹配规则，而且尽可能自己实现一些简单的例子。

头部

头部是一个变量、常量和复合数据类型的列表。所有在头部中的变量都符合格式`'$int()'`，其中`int()`被一个格式为`'0'`或`'1'`的整数替代，范围为`0~100 000 000`。基元`'_'`表示“无关紧要”的变量，当且仅当在你对一个特定参数的匹配部分不感兴趣的时候使用。然而，如果你想不用绑定任何变量来匹配所有的变量和元数，那么可以使用`"_"`给出一个格式为`['_', Conditions, Body]`的限定。你可能会忍不住写成`[[, Conditions, Body]`，但是你必须意识到，这将只匹配元数为0的函数。

在下面的例子中，我们建立了一个限定来匹配元数为2的函数。我们把第一个参数和变量`'$1'`绑定，第二个参数必须是一个元组，这个元组的第一个元素要和`'$2'`绑定，第二个元素必须是元组`{error, unknown_msg}`。在这种情况下下的匹配规则如下：

```
[[{'$1',{'$2',{error,unknown_msg}}],[,[]]]
```

在上面的代码中，条件和主体都用空列表表示，因为不存在逻辑检查和边界效应。回想“用`fun2ms`生成规则”小节中的例子，我们在其中寻找调试消息，并且查看`dbg:fun2ms/1`生成了什么。

在下面的例子中，我们使用`dbg:tp/2`来启用所有对函数`ets:insert/2`的调用跟踪。作为第二个参数传过去的匹配规则确保只有第二个参数是一个大小为2的元组，其中只有第二个元素是元组`{error, bad_day}`时，才会触发跟踪事件。

```
1> dbg:tracer().
{ok,<0.32.0>}
2> dbg:p(all,[c]).
{ok,[{matched,nonode@nohost,25}]}
3> dbg:tp({ets,insert,2}, [{['$1','$2',{error,bad_day}],[],[[]]}]).
{ok,[{matched,nonode@nohost,1},{saved,1}]}
4> ets:new(foo,[named_table]).
foo
5> ets:insert(foo, {1, monday}).
true
6> ets:insert(foo, {1, {error, bad_day}}).
true
(<0.30.0>) call ets:insert(foo,{1,{error,bad_day}})
```

我们可以使用“无关紧要”变量代替`'$1'`和`'$2'`，这是因为我们对条件和主体中的参数值不做任何改变。

在规则里变量是可以重复的。在规则`[['$1','$1'],[],[[]]]`中，我们匹配所有元数为2的参数相同的函数调用。当跟踪`foo(1,1)`的调用时，这个规则会触发一个跟踪事件。如果第一个参数是一个有两个元素的元组，而第二个参数是一个数列，这个数列中的头两个元素和元组中的相同，那么，匹配规则头部`[['$1','$2'],['$1','$2'|'_']]`将会匹配元数为2的函数。例如，如果跟踪调用`foo({1,2},[1,2,3])`，那么将触发一个事件。

Condition

`condition`列表允许组合变量上的多个逻辑测试。如果它们是正的，那么匹配成功且触发一个跟踪事件。必须绑定所有的变量，但是与头部中不同，常量和复合数据类型必须用特殊的语法描述。要表示一个元组，就必须使用`{const, Tuple}`或`{Tuple}`，这里的`Tuple`是一个代表元组的文字或者变量。可以使用`{hd, List}`和`{tl, List}`把列表分解为一个头部和一个尾部，其中`List`代表列表。

布尔函数包含的功能和内置函数相似，包括你可以在保护元以及布尔型的、关系型的、算术型的和逻辑型的运算符中使用的操纵。尽管匹配规则是Erlang项元，然而所有的运算符都要用基元表示，运算自身也要在元组中一起组合。

保护元的格式为`{Guard, Variable}`，其中`Guard`可以是下面中的任何一个：`is_atom`、`is_constant`、`is_float`、`is_integer`、`is_list`、`is_number`、`is_pid`、`is_port`、`is_reference`、`is_tuple`、`is_binary`、`is_function`、`is_seq_trace`。

例如，你可以使用以下格式的一个保护元去测试一个列表：

```
{is_list, '$0'}.
```

检查一个记录的保护元使用了项元is_record并符合以下格式：

```
{is_record, '$1', RecordType, record_info(size, RecordType)}
```

在上面的代码中，RecordType必须是一个硬编码的常量，以提供所需的记录类型。

条件表达式结构采用如下的格式：

```
{Construct, Exp1, Exp2, ...}
```

在上面的代码中，元组的第一个元素是逻辑结构，剩下的（可能是嵌套的）是条件表达式或保护元。如果Expression求值为false，那么逻辑结构{'not', Expression}求值为true。而任何其他的结果，求值均为false。

'and'、'or'、'xor'、andalso和orelse结构会采用一个大小为3或者更大的元组，其中第一个元素是结构，其他的是逻辑表达式。使用'and'或者'or'会计算结构内所有的表达式，然而andalso和orelse会分别在一个表达式返回false或者true后，立即结束计算。如果'xor'求值为true，那么它的表达式之一求值必须为true，而其他的为false。这里有一个条件表达式的例子，其中所有的3个表达式的计算值都必须为true：

```
{'and', {'not', '$1'}, '$2', {'or', '$3', '$4'}}.
```

比较运算符接受带有3个元素的元组，它的第一个元素是以下运算符：'>'、'>='、'<'、'<='、'=:='、'==='、'=/='或者'/'，剩余的两个包含其结果会被比较的表达式。结合保护元、条件表达式以及比较运算符会带来和下面格式相似的结果：

```
{'and', {is_integer, '$0'}, {is_integer, '$1'}, {'>=', '$0', '$1'}}.
```

下面的内置函数运算符在条件表达式里面也是允许的。它们与其对应的内置函数一样，由大小为2或3的元组组成，元组的大小取决于内置函数需要多少参数。运算符abs、hd、tl、length、node、round、size、trunc、'bnot'、'bsl'和'bsr'全都只用一个参数。使用element、'+', '-', '*', 'div', 'rem', 'band', 'bor'和'bxor'则要求一个带运算符和两个参数的元组。内置函数操作{self}和{get_tcw}返回调用进程的进程标识符或者跟踪控制字。经常在匹配规则体中使用它们，也允许把它们用作条件表达式的参数。

前面已经介绍了足够多的语法和语义，可能难以一下子全部消化。最好的解决办法是看一些例子，然后试着自己编写。你可以看出是什么触发了下面的匹配规则吗？在你继续阅读前，试着把它找出来：


```

[{'$1', '$2', '$3'},
 [{orelse,
   {':=', '$1', '$2'},
   {'and',
    {':=', '$1', {hd, '$3'}},
    {':=', '$2', {hd, {tl, '$3'}}}}
  ]],
[]
}]

```

让我们把上面的代码一点一点分开。头部告诉我们这个函数的元数为3。条件是 `orelse`，这里的第一个条件是 `'$1' ':= ' '$2'`，要求第一个参数必须和第二个参数完全相同。如果这个条件失败，那么我们会尝试联合（and）`{':=', '$1', {hd, '$3'}}`和`{':=', '$2', {hd, {tl, '$3'}}`。从这里我们可以推断出 `'$3'` 必须是一个列表，这个列表的第一个元素 `{hd, '$3'}` 要和第一个参数 `'$1'` 相同，第二个元素（尾部的头）要和第二个参数 `'$2'` 相同。在 Erlang 项元中，第三个参数必须是一个如 `'$3'=['$1', '$2' | '_']` 的列表。触发一个匹配的跟踪调用将包含 `foo(1,2,[1,2,3])` 和 `foo(true, true, false)`。作为一个练习，你可以试着定义一个函数，使用 `dbg:fun2ms/1` 把函数转换为这样的格式。

就像你在例子中看到的，允许使用许多运算符和结构。它们与相关的保护元、内置函数以及运算符采用同样的方法运作。经验告诉我们，如果它们在保护元中可以使用，那么它们也允许在匹配条件中使用。

在有坏参数的情况中，错误类型的数据和变量绑定，然后变量传递给这些操作，最后条件失败。执行匹配规则的进程并不会崩溃。

让我们再来多看一些例子。第一个例子有两个参数，如果 `'$1'` 大于等于0或者 `'$2'` 小于等于10，匹配成功。调用跟踪函数 `foo(0,10)` 或者 `foo(5,5)` 可以成功触发一个跟踪事件：

```

[{'$1', '$2'},
 [{and, {'>=', '$1', 0},
        {'<=', '$2', 10}}],
[]
}]

```

接下来一个例子是，如果元数为2的函数的第一个参数是基元 `enable` 或者 `disable`，第二个参数是一个元组，而这个元组的第一个元素为其他的元组 `{slot, 1, 3}`，那么匹配成功。注意在例子中是怎么使用 `{const, Term}` 结构来表示 `enable` 和 `disable` 的，然后把这些元组封装到另外一个元组中。

由于匹配规则的条件和主体由元组组成，因此不得不要么使用`{const, Term}`结构，要么插入一个元组（如`{{slot, 1, 3}}`），用它来把“真”的元组与一个条件结构、内置函数和比较运算区分开：

```
[[{'$1', '$2'},
  [{'and', {'orelse', {':=', '$1', {const, enable}},
                    {':=', '$1', {const, disable}}},
   {':=', {element, 1, '$2'}, {{slot, 1, 3}}}],
 []],
 []]
```

如果我们要在机架3的插槽1的板上跟踪特定的操作，我们就可以使用上面的规则。函数例子触发一个包括函数调用`board:action(enable, {{slot, 1, 3}, disabled})`和`board:action(disable, {{slot, 1, 3}, unknown})`的跟踪事件。

如同你将在“规则主体”小节看到的那样，在触发规则时，我们可以打开规则主体部分的一个特定的跟踪标志。在看到如何做到这一点之前，让我们先来处理上面的例子，添加第二个匹配规则到列表中。在一个实时的环境中，两个匹配规则都会得到相同的结果，但是下面的例子只允许大小为2的元组。使用`element`内置函数等效的办法，我们指定了第二个参数必须是至少带有两个元素的一个元组：

```
[[{enable, {{slot, 1, 3}, '_'}},
  [],
  []],
 [{disable, {{slot, 1, 3}, '_'}},
  [],
  []],
 []]
```

规则主体

如果匹配成功并且条件的计算值为`true`，那么可以指定一系列的行动，包括发送跟踪事件、输出项元、启用和禁用跟踪标志，以及返回跟踪信息。我们在前面的内容中提到的和这些相同的规则，在这里也适用于变量和常量；行为一定要在元组里定义，即使它们没有参数。先前相同的规则也可以应用到`'_'`和`'$$'`，即`'_'`返回整个变量列表，而`'$$'`返回所有绑定的变量列表。

在下面的例子中，如果第一个匹配规则匹配成功，那么会把`procs`标志加入到跟踪标志列表中；如果第二个匹配规则匹配成功，那么它将会把那个标志从列表中删除：

```
[[{enable, {{slot, 1, 3}, '_'}},
  [],
  [{enable_trace, procs}]],
```

```

    [{disable, {{slot, 1, 3}, '_'}},
     [],
     [{disable_trace, procs}]]
]

```

规则主体中启用和禁用调用的语法格式为：`{enable_trace, TraceFlag}`和`{disable_trace, TraceFlag}`，其中`TraceFlag`就像在“跟踪内置函数”一节中定义的一样，是一个单独标志。动作`enable_trace`和`disable_trace`与下面这个内置函数有相同的功能：

```
erlang:trace(self(), Flag, [TraceFlag])
```

其中项元`true`或者`false`会取代这里的`Flag`，分别代表启用和禁用这个`TraceFlag`。

在匹配规则主体中可以执行如下行为：

`{message, Args}`

把`Args`添加到跟踪事件中。`Args`可以是变量、参数或者其他行为比如`process_dump`的返回值。

`{message, false | true}`

是一种特殊的情况：如果标志位为`false`，则不会发送跟踪标志`call`和`return_to`的消息。传递`true`可以产生一个普通的跟踪消息，这与根本没有主体完全一样。

`{return_trace}`

在这个函数的返回是触发一个跟踪消息的发送。如果函数是尾递归的，那么此属性将丢失。

`{exception_trace}`

和`return_trace`的行为一样，除了由于运行时错误导致所跟踪的函数退出之外，都会生成一个`exception_from`跟踪消息。

`{silent, true | false}`

除了跟踪进程之外，还为进程开启和禁用跟踪消息。

`{display, Term}`

在标准输出中显示一个单独的`Term`，而这只被用作调试目的。

`{set_tcw, Value}`

设定一个节点的跟踪控制字，并且返回先前的值。调用`erlang:system_flag(trace_control_word, Value)`和在规则主体中执行命令效果相同。

`{enable_trace, TraceFlag}`

启用任何一个跟踪内置函数接收的跟踪标志。对于你想启用的每一个`TraceFlag`，

你需要一个单独的enable_trace调用，这与调用erlang:trace(self(), true, [TraceFlag])效果相同。

{disable_trace, TraceFlag}

禁用TraceFlag。在匹配规则中执行这个函数，与调用函数erlang:trace(self(), false, [TraceFlag])效果相同。

{trace, DisableList, EnableList}

将禁用在DisableList中定义的标志，同时也自动启用在EnableList中定义的标志。这个标志和在内置函数erlang:trace/3中使用的效果一样，但你不可以包含cpu_timestampflag标志。不过，你可以包含{tracer, Pid}来指定哪一个Pid应当接收跟踪事件。如果在两个列表中同时定义，那么在EnableList中的标志（包括跟踪器指令）优先级高于在DisableList中的标志。

{trace, Pid, DisableList, EnableList}

和有两个参数的函数相同，但是也允许你定义一个进程标识符或者一个注册的别名，而它们之上的标志应被启用或禁用。该标志的聚合是原子性的。

用下面的调用来返回数值，常见用法是作为前面描述的动作message和display的一个参数：

{process_dump}

返回进程上的文字消息，格式化为一个字符串并且储存为二进制。

{caller}

以格式{Module, Function, Arity}返回一个调用函数。如果函数不能确定，则跟踪事件返回undefined。

{get_tcw}

返回先前设定的跟踪控制字。这个调用和内置函数erlang:system_info(trace_control_word)有相同的作用。

{pid}

返回执行匹配规则的进程的进程标识符。

在下面的例子中，我们把进程的转存添加到跟踪消息中，在自定义的跟踪器fun中提取出它，把它从二进制转换为字符串并打印。注意，终端的进程栈更深。我们已经删掉了一些尾随的函数。而且也要注意我们是如何在匹配规则的头部传递'_'的，'_'代表任意一个元数的函数：

```
1> DbgFun = fun({trace, _Pid, _event, _data, Msg}, _Acc) ->
1>     io:format("~s~n", [binary_to_list(Msg)])
```

```

1>         end.
#Fun<erl_eval.12.113037538>
2> dbg:tracer(process, {DbgFun, null}).
{ok,<0.33.0>}
3> dbg:tp({io,format,1}, [{['_',[],[{message,{process_dump}}]}]}).
{ok,[{matched,nonode@nohost,1},{saved,1}]}
4> dbg:p(all,[c]).
{ok,[{matched,nonode@nohost,25}]}
5> io:format("Hello~n").
Hello
=proc:<0.30.0>
State: Running
Spawned as: erlang:apply/2
Spawned by: <0.24.0>
Started: Tue Oct 07 13:17:07 2008
Message queue length: 0
Number of heap fragments: 0
Heap fragment data: 0
Link list: []
Reductions: 8879
Stack+heap: 2584
OldHeap: 2584
Heap unused: 1271
OldHeap unused: 2584
Stack dump:
Program counter: 0x01b0e904 (shell:eval_loop/3 + 44)
CP: 0x01b02388 (erl_eval:do_apply/5 + 1304)

0x01473450 Return addr 0x01b0f15c (shell:exprs/6 + 368)
y(0)    [{ 'DbgFun',#Fun<erl_eval.12.113037538>}]
y(1)    []
y(2)    none

0x01473460 Return addr 0x01b0ec40 (shell:eval_exprs/6 + 80)
y(0)    []
y(1)    []
y(2)    [{ 'DbgFun',#Fun<erl_eval.12.113037538>}]
y(3)    {value,#Fun<shell.7.51306786>}
y(4)    {eval,#Fun<shell.24.79061235>}
y(5)    13
y(6)    []
y(7)    []
y(8)    []

```

在最后这个例子中可以看到格式为元组`io:format/2`的调用。我们看到格式为元组`{erl_eval,do_apply,5}`的输出，其紧跟在命令4的调用之后：

```

1> dbg:tracer().
{ok,<0.32.0>}
2> dbg:p(all, [call]).
{ok,[{matched,nonode@nohost,25}]}
3> dbg:tp({io, format, 2}, [{['_', [], [{display, {caller}}]}]}).
undefined
{ok,[{matched,nonode@nohost,1}, {saved,1}]}

```

```
4> io:format("Hello~n",[]).
{erl_eval,do_apply,5}
Hello
(<0.30.0>) call io:format("Hello~n",[])
ok
```

存储匹配规则

回顾一下前面的例子，特别留意dbg:tp/2调用的返回值，它返回了{ok,[{matched,node@nohost,1},{saved,1}]}。你已经知道了第一个内部组件{matched,node@nohost,1}，它告诉你在这个特定的节点上匹配一个函数。但是{saved,1}却告诉你，调用中匹配规则和这个标识符一起被保存了，这个标识符也就是数字1。dbg跟踪器给所有的匹配规则分配数字，允许你用这个数字代替规则本身。

用来检索和操纵这些调用的帮助函数包括以下这些：

dbg:ltp()

列出这个部分使用的所有匹配规则。

dbg:dtp()anddbg:dtp(Id)

删除储存的匹配规则（带一个特殊标识符）。

dbg:wtp(FileName)和dbg:rtp(Filename)

从一个文件读写匹配规则。

```
5> dbg:tp({io,format,2},{['_'],[],[{enable_trace,procs}]}).
{ok,[{matched,node@nohost,1},{saved,2}]}
6> dbg:ltp().
1: [{['_'],[],[{display,{caller}}]}]
2: [{['_'],[],[{enable_trace,procs}]}]
exception_trace: x
x: [{['_'],[],[{exception_trace}]}]
ok
```

从Erlang/OTP的R13发布版本开始，就有一个预存的匹配规则，它的标识符为exception_trace，别名x。它是作为匹配规则[['_'],[],[{exception_trace}]}预存的，并且可以用在调用如dbg:tp({M, F, A},x)中。它值得在此强调，因为它有可能是在调试的时候用最常使用的规则之一。

扩展阅读

也许你已经发现，尽管跟踪实时Erlang程序的功能很强大，但有时候也很复杂。在本章中，我们给你提供了一个全面的概述，它包括什么是可用的，哪些可以帮助你处理遇到

的大部分的情况。但是如果你处理高负载永不停止的实时系统，那么你永远也不知道会发生什么。

在`dbg:fun2ms/1`的`fun`主体调用中，我们没有提到两个函数`set_seq_token/2`和`get_seq_token/0`。它们在匹配规则主体中被翻译成顺序跟踪行为`set_seq_token`和`get_seq_token`。和`seq_trace`模块输出的API一起，它们允许使用者追踪进程之间的消息传播路径。如果进程A给进程B发送一条消息，并且导致给进程C和D也发送了消息，那么这个传播消息调用链就会被跟踪。如果有兴趣学习更多内容，可以阅读介绍`seq_trace`的手册页。单单这个题目就足以增加一节内容。

如果你想阅读更多有关跟踪内置函数的内容，首先应该想到的是`erlang`模块手册页，其中对所有的内置函数都有详细说明。如果你对探索匹配规则感兴趣，那么在《ERTS User's Guide》中用了一整章的篇幅来说明它们，其中包含一个完整的语法教程。如果你宁愿简单地停留在匹配规则及其转换上，`ms_transform`手册页也许是你的最佳选择。你传给`fun2ms`的匹配规则函数也可以在《ERTS User's Guide》中找到。最后，`dbg`的手册页介绍了`dbg`跟踪器工具。

练习

习题17-1：测量垃圾收集时间

用跟踪内置函数编写一个程序，它的作用是监控一个进程在特定函数中花在垃圾收集上的时间（以微秒计）。一旦函数结束运行，则输出时间。你必须确保读取不受事先分配的内存影响，并且为每一个读取都生成一个新进程。

用一个尾递归函数和一个非尾递归函数测试你的程序，它们做的是相同的事情。非尾递归函数处理大小不同的列表需要消耗时间，而这样做让你可以监测这种影响。

提示：使用`timer:now_diff/2`计算时间差。

你可以把以下代码用作一个尾递归函数：

```
average(List) -> sum(List) / len(List).  
  
sum([]) -> 0;  
sum([Head | Tail]) -> Head + sum(Tail);  
  
len([]) -> 0;  
len([_ | Tail]) -> 1 + len(Tail).
```

而且你也可以把以下代码用作一个非尾递归函数：

```
average(List) -> average_acc(List, 0,0).  
  
average_acc([], Sum, Length) -> Sum / Length;  
average_acc([H | T], Sum, Length) -> average_acc(T, Sum + H, Length + 1).
```

当你执行代码时，可能会得到如下结果：

```
1> List = lists:seq(1,1000).  
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,  
23,24,25,26,27,28,29|...]  
2> gc_mon:measure(gc_test, average, List).  
Gc monitoring terminated  
Microseconds:8  
ok
```

习题17-2：使用dbg的垃圾收集

用dbg和你自己的fun跟踪器重写习题17-1的答案。这么做的时候，添加一个计数器来测量进程运行时回收的内存。要特别注意处理不同的内存类型。

习题17-3：跟踪ETS表的条目

记录一个崩溃报告。进一步调查发现，一个叫做countries的ETS表被记录{'EXIT',Reason}损坏了。不是类型为countries的一个记录而是写入了元组。每天这个表要更新上千次，因此跟踪所有表的记录不是一个好办法，因为花费的时间和要过滤的数据量会很大。

创建一个匹配规则，使得每次元组{'EXIT',Reason}作为ets:insert/2函数的第二个参数传递时都触发这个匹配规则。当你让匹配规则正常工作以后，打开一个单独的Erlang节点，你的跟踪器会在这个节点上接收和输出所有的跟踪消息。

在这些正常运行以后，使用条件语句重写匹配规则，让它在{'EXIT',Reason}和{'EXIT',Pid,Reason}上都能触发。

习题17-4：谁是肇事者

判断本书中的两位作者，是谁曾经由于使用dgb跟踪器工具不慎而一手导致了全美国范围内的移动数据网络中断。

第18章

类型和文档

第2章介绍了Erlang中的基本类型：整数、浮点数、基元、字符串、元组和列表；第7章介绍了记录；第9章进一步介绍了二进制类型和引用。当我们声明函数和其他定义时，我们也会给出它们输入输出类型的非正式描述。

本章介绍如何在Erlang中使用Richard Carlsson开发的EDoc文档框架写出函数类型，并且作为它们函数的正式文档的一部分。为一个函数类型写下的东西可以用实现者Dialyzer开发的TypEr工具检查它与函数定义的一致性。TypEr无须任何用户输入就能推断类型，因此它可以成为理解程序的一个重要工具。TypEr和Dialyzer是Uppsala University高性能Erlang (HiPE) 研究小组的研究成果。所有的这些工具都是标准Erlang发行包的一部分。

Erlang中的类型

在本章中，我们将以一个例子开头，然后概括介绍Erlang的类型标识。

实例：有字段类型的记录

在本书前面的部分我们讨论过记录的定义。在第10章移动用户数据库的例子中介绍了一个记录的声明：为一个移动电话系统的特定用户保存信息。

```
-record(usr, {msisdn,           %int()
              id,               %term()
              status = enabled  %atom(), enabled | disabled
              plan,             %atom(), prepay | postpay
              services = []}).  %[atom()], service flag list
```

usr记录有5个字段，在注释里，每个字段类型都有说明。

但是，你可以更进一步把这些注释变为程序的一部分。首先，引入类型声明作为计划(plan)、用户状态(status)以及服务(service)的类型：

```
-type(plan()      :: prepay | postpay).  
-type(status()   :: enabled | disabled).  
-type(service()  :: atom()).
```

这3个类型声明定义了`plan`、`status`和`service`类型。由于是在常量函数中，所以它们的后面跟着一对括号：

`plan()`

有两个元素：基元`prepay`和`postpay`。可以使用“|”标志来定义选项（就像在正则表达式和语法中一样）。在这里，选项是两个可能的类型成员。

`status()`

也只有两个元素：基元`enabled`和`disabled`。

`service()`

是一个`atom()`的别名，但是使用它时可以更清楚地表示这个位置上的一个项元代表某种服务。

有了这些定义，你可以显式地把类型赋给这个记录的字段：

```
-record(usr, {msisdn      ::integer(),  
             id          ::integer(),  
             status = enabled ::status(),  
             plan       ::plan(),  
             services = []   ::[service()]  
}).
```

这更清楚地表明了`usr`元素的记录类型是什么，以及操作这些元素的程序该如何使用它们。更重要的是，它为工具提供了一些信息可以帮助你代码中检测类型错误。这个简短的例子可以让你体验一下Erlang类型的标识。现在让我们来进一步了解一些细节。

Erlang类型标识

Erlang预定义了许多类型，其中包括：

`any()`

包括所有的Erlang数据值（和它的别名函数`term()`作用一样）。

`atom()`

包括所有的Erlang基元。

`binary()`

包含了所有的二进制数值。

`boolean()`

包括基元`true`和`false`。

`byte()`

包括数字0~255。

`char()`

是`integer()`类型的Unicode子集。

`deep_string()`

是一个递归类型，等同于`[char+deep_string()]`。

`float()`

包括了所有的浮点数。

`function()`

包括了所有的`fun`。

`integer()`

包括了所有的整数，你应该还记得它们是“大整数”。

`list(T)`

是一个类型为`T`的列表的类型，也可以写作`[T]`。

`nil()`

只有一个元素，空列表`[]`。

`none()`

是一个没有元素的空类型，用来表示一个永不返回的函数的返回类型。

`number()`

由`float()`和`integer()`类型联合组成。

`pid()`、`port()`、`reference()`

都是自定义的。

`string()`

字符列表类型`[char()]`的同义词。

`tuple()`

包含所有的元组。

你可以在EDoc和Dialyzer文档中找到其他预定义类型的详细说明。除了上述提到的预定义类型之外，你还可以使用下列标识定义你自己的类型：

`atom`

可以把任何基元都当做一个类型，例如类型`ok`有一个元素`ok`。

|或+

你可以用它们去定义两个类型的联合，例如 `true|false`。

`#rec{}`

这是一个叫做`rec`的记录类型。

`{T1,T2,...}`

这是一个元组类型，它的第一个元素来自类型`T1`，第二个元素来自类型`T2`，以此类推。例如，`{error,atom()}`包括所有这样的元组，它们的第一个元素是基元`error`，第二个元素是任意一个基元。

`[T]`

这是一个列表类型，它的元素来自类型`T`。

`L..U`

这是一个整数范围，它的下限是`L`，上限是`U`。可以用它来定义类型`byte()`和`char()`，以及许多其他内置类型，包括类型`pos_integer()`。

函数类型的定义有如下的格式：

```
(Argument_types) -> Result_type
```

其中参数类型也可以包含相应参数的名字。

这里你可以用一条`-spec`语句给程序里的一个函数指定一个类型（`type`）或者原型（`prototype`）。为了使语句意义明确，用`-spec`来指定一个函数的类型，而用`-type`来定义一个类型。回顾第10章中`usr_db`的例子，你可以进行如下操作：

```
-spec(create_tables(FileName::string()) -> {ok, ref()} | {error, atom()}).
create_tables(FileName) ->
... .
-spec(close_tables() -> ok | {error, atom()}).
close_tables() ->
... .
```

这里`create_tables`类型表明参数`FileName`是一个字符串，并且返回值有两种可能：

- 如果操作成功，将返回一个元组，其第一个元素为`ok`，第二个元素是所创建的表的引用。

- 如果失败, 将返回另一个元组, 其第一个元素为error, 第二个元素是一个基元, 大概地指出可能的错误类型。

这里FileName的使用是可选的。下面的语句同前面的意义一样:

```
-spec(create_tables(string()) -> {ok, ref()} | {error, atom()}).
```

然而, 包含参数名(假定它的字面意思反映出它的用途)可以提高程序文档的易读性(注1)。

TypEr: 成功类型和类型推断

由Tobias Lindahl和Kostis Sagonas建立的TypEr系统(注2)用来检查-spec注释的有效性, 并且可以在没有类型注释的情况下推断模块中函数的类型。

TypEr可以在命令行中使用。例如查看所有的选项, 你可以输入:

```
typer --help
```

以第10章中移动用户数据库为例, 输入如下命令:

```
typer --show usr.erl usr_db.erl
```

它将输出以下内容(为了简洁进行了缩减):

```
Unknown functions: [{ets,safefixtable,2}]

%% File: "usr.erl"
%% -----
-spec start() -> 'ok' | {'error','starting'}.
-spec start(_) -> 'ok' | {'error','starting'}.
-spec stop() -> any().
-spec add_usr(_,_,_) -> any().
-spec delete_usr(_) -> any().
...

%% File: "usr_db.erl"
%% -----
-spec create_tables(_) -> any().
-spec close_tables() -> any().
```

注1: 本章后面将会介绍的EDoc系统会自动地在其类型文档中包含参数名称, 即使它没有出现在-spec语句中出现。

注2: 2005年在塔林(Tallin)和2007年在弗莱堡(Freiburg)召开的Erlang研讨会的论文中有关于TypEr的描述(<http://doi.acm.org/10.1145/1088361.1088366>和<http://doi.acm.org/10.1145/1292520.1292523>)。

```
-spec add_usr(#usr{}) -> 'ok'.
-spec update_usr([tuple()] | tuple()) -> 'ok'.
-spec delete_usr(_) -> 'ok' | {'error','instance'}.
...
```

在如Haskell这样的静态类型语言中，由类型检查器推断的函数类型可以确保应用到输入类型的参数时，函数不会出错。Erlang是一种动态类型语言，因而TypEr工具将会使用另外的方法。

注意：TypEr能够推断成功类型，它封装了函数能够成功应用的所有类型。一般来说，这种方法并不精确，但它在任何时候都好过近似（over approximation）。因而，用任何其他的方式调用该函数都会失败。

为了解释清楚，举一个例子：如果一个函数 f 给定一个成功类型 $S \rightarrow T$ ，并且 E 是任意一个Erlang表达式，使得 $f(E)$ 成功求值为 V ，这样 E 就必须是类型 S ， V 必须是类型 T 。

在前面的usr_db.erl例子中，create_tables/1不会推断出任何有用的信息，因为它的输入类型和输出类型都是any()。说得更清楚一些，TypEr是一个应用到usr_db的不带类型注解的版本。如果你添加了这些，应用TypEr的结果可能会不同：

```
-spec(create_tables(string()) -> {ok, ref()} | {error, atom()}).

create_tables(FileName) ->
    ets:new(subRam, [named_table, {keypos, #usr.msisdn}]),
    ets:new(subIndex, [named_table]),
    dets:open_file(subDisk, [{file, FileName}, {keypos, #usr.msisdn}]).

-spec(close_tables() -> ok | {error, atom()}).

close_tables() ->
    ets:delete(subRam),
    ets:delete(subIndex),
    dets:close(subDisk).

-spec(add_usr(#usr{}) -> ok).

add_usr(#usr{msisdn=PhoneNo, id=CustId} = Usrc) ->
    ets:insert(subIndex, {CustId, PhoneNo}),
    update_usr(Usrc).

-spec(update_usr(#usr{}) -> ok).

update_usr(Usrc) ->
    ets:insert(subRam, Usrc),
    dets:insert(subDisk, Usrc),
    ok.

-spec(delete_usr(integer()) -> ok | {error, atom()}).
```

```

delete_usr(CustId) ->
  case get_index(CustId) of
    {ok,PhoneNo} ->
      delete_usr(PhoneNo, CustId);
    {error, instance} ->
      {error, instance}
  end.

-spec(delete_usr(integer(),integer()) -> ok|{error,atom()}).

delete_usr(PhoneNo, CustId) ->
  dets:delete(subDisk, PhoneNo),
  ets:delete(subRam, PhoneNo),
  ets:delete(subIndex, CustId),
  ok.

...

```

在注释文件上运行TypEr会检查指定类型，然后给出以下结果：

```

%% File: "usr_db.erl"
%% -----
-spec create_tables(string()) -> {'ok',ref()} | {'error',atom()}.
-spec close_tables() -> 'ok' | {'error',atom()}.
-spec add_usr(#usr{}) -> 'ok'.
-spec update_usr(#usr{}) -> 'ok'.
-spec delete_usr(integer()) -> 'ok' | {'error',atom()}.
-spec delete_usr(integer(),integer()) -> 'ok' | {'error',atom()}.
...

```

进行这些的时候，TypEr会对比推断类型来检查指定类型，并报告不一致的情况。例如，如果你像下面这样改变add_usr的-spec：

```

-spec(add_usr(#usr{}) -> integer()).

```

TypEr会这样报告：

```

typer: Error in contract of function usr_db:add_usr/1
The contract is: (#usr{}) -> integer()
but the inferred signature is: (#usr{}) -> 'ok'

```

另外，如果你改变create_tables/1的spec，那么将不报告错误，这是因为这个函数的推断类型和任何一个参数的函数都符合。

Dialyzer：Erlang程序的差错分析器

在Erlang程序中，TypEr提供了类型分析。Dialyzer则将它进一步扩展，能对Erlang程序进行静态分析，用于分辨软件的差错，包括冗余测试和无法实现的代码，以及明显的类型错误。

为了提高其操作的速度，Dialyzer可以使用--build_plt选项创建一个持续查找表（Persistent Lookup Table，PLT）。当你包含了内核、标准库以及Mnesia时，例如：

```
dialyzer --build_plt -r <erl-lib>/kernel-2.12.5/ebin <erl-lib>/
stdlib-1.15.5/ebin <erl-lib>/mnesia-4.4.7/ebin
```

它需要花几分钟来产生PLT表和如下报告：

```
Creating PLT /Users/simonthompson/.dialyzer_plt ...
re.erl:41: Call to missing or unexported function unicode:characters_to_binary/2
re.erl:134: Call to missing or unexported function Unicode:characters_to_list/2
re.erl:200: Call to missing or unexported function re:compile/2
re.erl:226: Call to missing or unexported function unicode:characters_to_binary/2
re.erl:245: Call to missing or unexported function unicode:characters_to_list/2
re.erl:505: Call to missing or unexported function unicode:characters_to_list/2
re.erl:545: Call to missing or unexported function unicode:characters_to_binary/2
Unknown functions:
  compile:file/2
  compile:forms/2
  compile:noenv_forms/2
  compile:output_generated/1
  crypto:des3_cbc_decrypt/5
  crypto:start/0
done in 16m43.44s
done (warnings were emitted)
```

之后，作为运行实例针对文件调用Dialyzer将会给出如下报告（用时不到1秒）：

```
dialyzer -c usr.erl usr_db.erl
Checking whether the PLT /Users/simonthompson/.dialyzer_plt is up-to-date... yes
Proceeding with analysis...
usr.erl:110: The pattern [] can never match the type {'error','instance'}
usr_db.erl:69: Call to missing or unexported function ets:safefixtable/2
done in 0m0.33s
done (warnings were emitted)
```

从联机文档中你可以找到更多关于Dialyzer的信息。

使用EDoc生成文档

有一种说法是函数式程序可以自动文档化。遗憾的是，尽管函数式编程语言可以产生更易读的程序，但是复杂的程序不能自动文档化，更别说浅显易懂了。如果持续更新程序模块中的自由文本注释，它们将是描述一个程序的第一步。但是它们也具有缺乏结构性的缺点，而且难以独立于程序文本进行扫描、搜索以及读取。

EDoc为Erlang提供了一种文档框架，它可以克服上述缺点并且根据你在模块里插入的信息生成文档：

- EDoc为注释提供一个结构，包括type、spec的信息以及关于函数的文字的注释。
- EDoc是一个文档生成器：一个HTML文件的标准样式将会根据每个模块里的结构化信息生成。
- EDoc提供一种框架，添加覆盖整个模块集合的信息（例如：在应用程序或软件包中），并且提供一个大型结构的概括或者对整个系统的假设。

EDoc是标准Erlang发行包的一部分，它和很多相似的系统有很多的共同点，如Haddock（针对Haskell）、Javadoc、pydoc和RDoc（针对Ruby）。在本节中，我们将为第10章中的移动用户数据库的例子生成文档，以便为你介绍许多EDoc的特性。

注意：在稍后发布的Erlang/OTP版本中，EDoc计划将和TypeR共享信息，这样可以使-type和-spec声明共享类型信息。同时，这个类型信息将以一种不同的格式给出，这将在本节中详细介绍。

为usr_db.erl生成文档

EDoc可以从内嵌在注释中格式为@tag text的标签生成文档。每一个标签可以持续许多行，直到下一个标签或非注释的行出现。你可以以不同的方式使用不同种类的标签：

模块标签

提供模块级别的文档。必须在模块声明本身之前出现。

函数标签

与那些跟随它们的函数关联，并且给出特定函数的信息。

通用标签

可能包含“要去做”的信息或者一个类型定义，它可以在一个文件的任何位置出现。

我们将会以usr_db为例依次介绍以上标签的使用方法。

模块标签

在这个例子中，模块标签如下所示：

```
%% @author Francesco Cesarini <support@erlang-consulting.com>
%% @author Simon Thompson [http://www.cs.kent.ac.uk/~sjt/]
%% @doc Back end for the mobile subscriber database.
%% The module provides an example of using ETS and DETS tables.
%% @reference <a href="http://oreilly.com/catalog/9780596518189/">
    Erlang Programming</a>,
%% <em> Francesco Cesarini and Simon Thompson</em>,
```

```
%% O'Reilly, 2009.  
%% @copyright 2009 Francesco Cesarini and Simon Thompson
```

标签给出以下信息：

@author

模块的作者和一些可选的联系信息（email或HTML）。

@copyright

版权声明。

@doc

模块的描述，采用合适良好的XHTML文本。这个文本的第一个句子用作模块的概要。

@reference

提供进一步信息的一个引用，它可以包含XHTML链接。

你可以使用其他的标签来给出其他信息，如版本号（**@version**）、把模块引进到系统中的时间（**@since**），以及文档是否可见（**@hidden**或**@private**）。

函数标签

函数的主要文档由其类型和功能的描述给出：

@spec

给出一个函数的类型。先前已经给出类型描述的格式。预计在以后的版本中将使用 **-spec** 声明中的信息取代它。

@doc

表示函数的一般文档，采用XHTML标签。

其他可用的标签，包括交叉引用其他对象的文档（**@see**），描述哪种例外的类型可以抛出（**@throws**），还有是否不赞成使用一个函数（**@deprecated**）；也可以使用先前介绍的 **@hidden**、**@private**和**@since**。

对于usr_db模块片段，文档如下：

```
%% @doc Create the ETS and DETS tables which implement the database. The  
%% argument gives the filename which is used to hold the DETS table.  
%% If the table can be created, an 'ok' tuple containing a  
%% reference to the created table is returned; if not, it returns an 'error'  
%% tuple with an atom describing the error.  
  
%% @spec create_tables(string()) -> {ok, reference()} | {error, atom()}
```

```

-spec(create_tables(string()) -> {ok, ref()} | {error, atom()}).

create_tables(FileName) ->
    ets:new(subRam, [named_table, {keypos, #usr.msisdn}]),
    ets:new(subIndex, [named_table]),
    dets:open_file(subDisk, [{file, FileName}, {keypos, #usr.msisdn}]).

%% @doc Close the ETS and DETS tables implementing the database.
%% Returns either 'ok' or an 'error'
%% tuple with the reason for the failure to close the DETS table.

%% @spec close_tables() -> ok | {error, atom()}

-spec(close_tables() -> ok | {error, atom()}).

close_tables() ->
    ets:delete(subRam),
    ets:delete(subIndex),
    dets:close(subDisk).

%% @doc Add a user (of the 'usr' record type) to the database.

%% @spec add_usr(#usr{}) -> ok

-spec(add_usr(#usr{}) -> ok).

add_usr(#usr{msisdn=PhoneNo, id=CustId} = Usr) ->
    ets:insert(subIndex, {CustId, PhoneNo}),
    update_usr(Usr).

```

在先前的代码中，我们使用@spec来提供类型，在Erlang未来的发布版本中，预计-spec声明可以取代它，那时，@spec标签就是多余的了。

通用标签

通用标签可以在模块的任何地方出现：

@type

将会给出类型定义，EDoc将识别这种类型定义来生成文档。预计在今后的发布版本中将使用-type声明取代它。

@todo

用它来表明“要做的”注释。它将不会在产生的文档中出现，除非激活todo选项。

运行EDoc

主要的EDoc函数都在edoc模块中。一个对edoc:application/1的调用可以为一个应用程序产生文档，而对edoc:files/1的调用会给一组文件产生文档；这些函数都使用EDoc默认的选项。函数的双参数版本允许一个选项的集合当做第二个参数传入。

模块页

图18-1给出运行`edoc:files(["usr_db.erl", "usr.erl"])`时为`usr_db`模块生成的页面。这也显示出了一个典型Erlang模块的EDoc页结构。

页面开始部分是它的主要内容的链接，然后是模块和其他模块标签的一句话概述，接着是模块的完整描述。

在函数的索引中，函数按字母顺序排列，而不是按照它们在文件中的次序。把每一个函数都链接到它的详细信息上，并且给出一句话的概述：它的`@doc`标签的第一个句子。

每一个函数都有关于它的类型的详细信息。注意信息不仅包含参数和结果的类型，还包括从源代码自动提取的参数名字。例如，`create_tables/1`的`@spec`包含以下信息：

```
%% @spec create_tables(string()) -> {ok, reference()} | {error, atom()}
```

这里是生成的文档：

```
create_tables(File::string()) -> {ok, reference()} | {error, atom()}
```

这提供了一些额外的有用信息：字符串参数代表一个文件名。在没有任何`@spec`信息的情况下，EDoc仍然会在产生的信息中包含参数的名称。

概述页

对每一个项目都会生成一个概述页，用于提供这个项目中模块的索引。更多的信息可以查看`overview.edoc`文件，它通常出现在`doc`子目录下，这里也包含其他的文档。

`overview.edoc`文件和模块头部有相同的内容标签，但是没有必要把每一行包含在注释里。其这样运行例子的一个结果如图18-2所示，它由如下开头的一个文件产生：

```
@author Francesco Cesarini <support@erlang-consulting.com>
@author Simon Thompson [http://www.cs.kent.ac.uk/~sjt/]
@reference <a href="http://oreilly.com/catalog/9780596518189/">Erlang Programming</a>,
    <em> Francesco Cesarini and Simon Thompson</em>,
    O'Reilly, 2009.
```

EDoc中的类型

`usr_db.erl`模块不包含类型定义，但是`usr.hrl`包含一些，并且在`usr.erl`中可以引用它们。文档中的类型采用如下方式：首先给出定义，然后给出可选的说明：

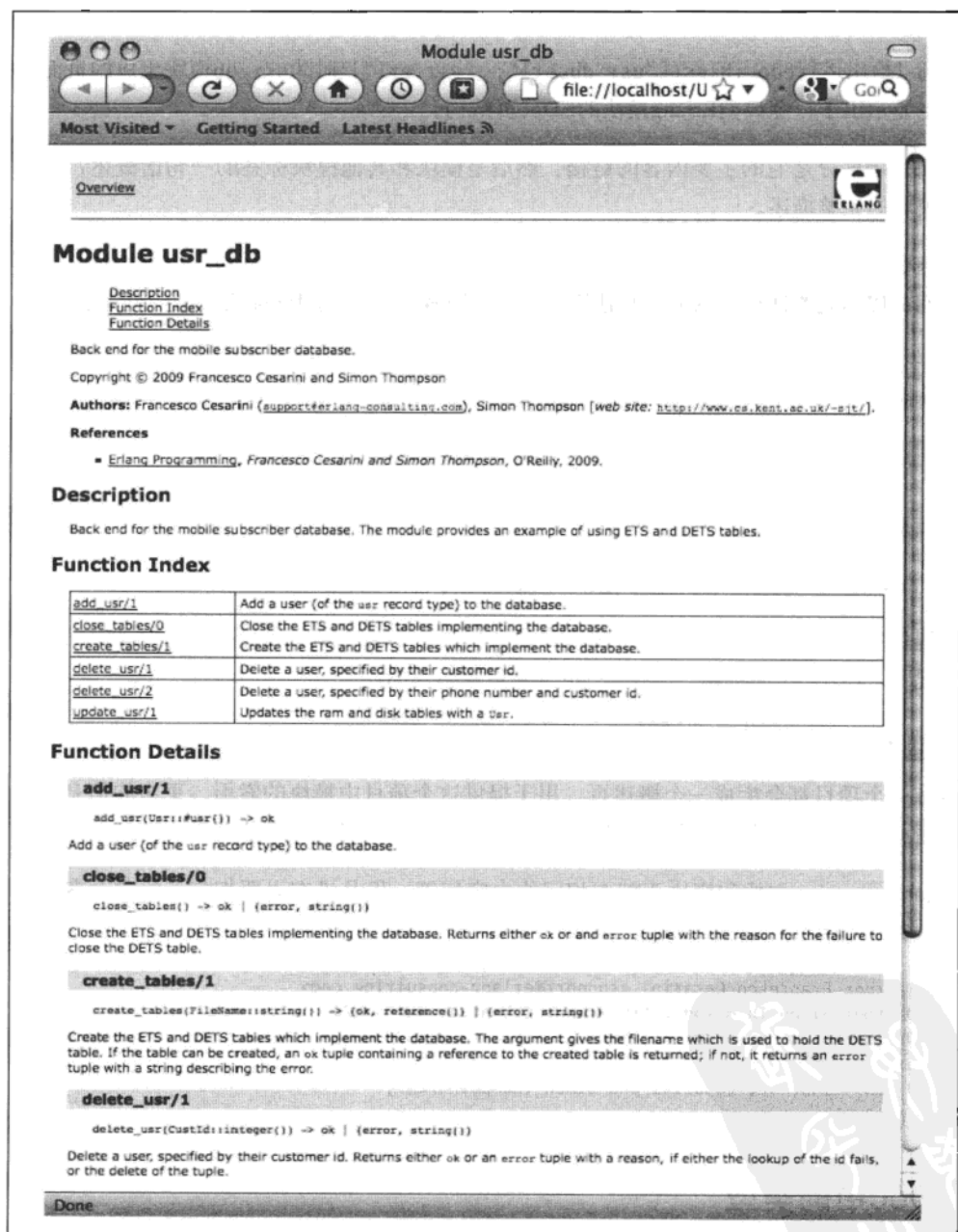


图18-1: `usr_db.erl`的EDoc页（其中的一个片段）

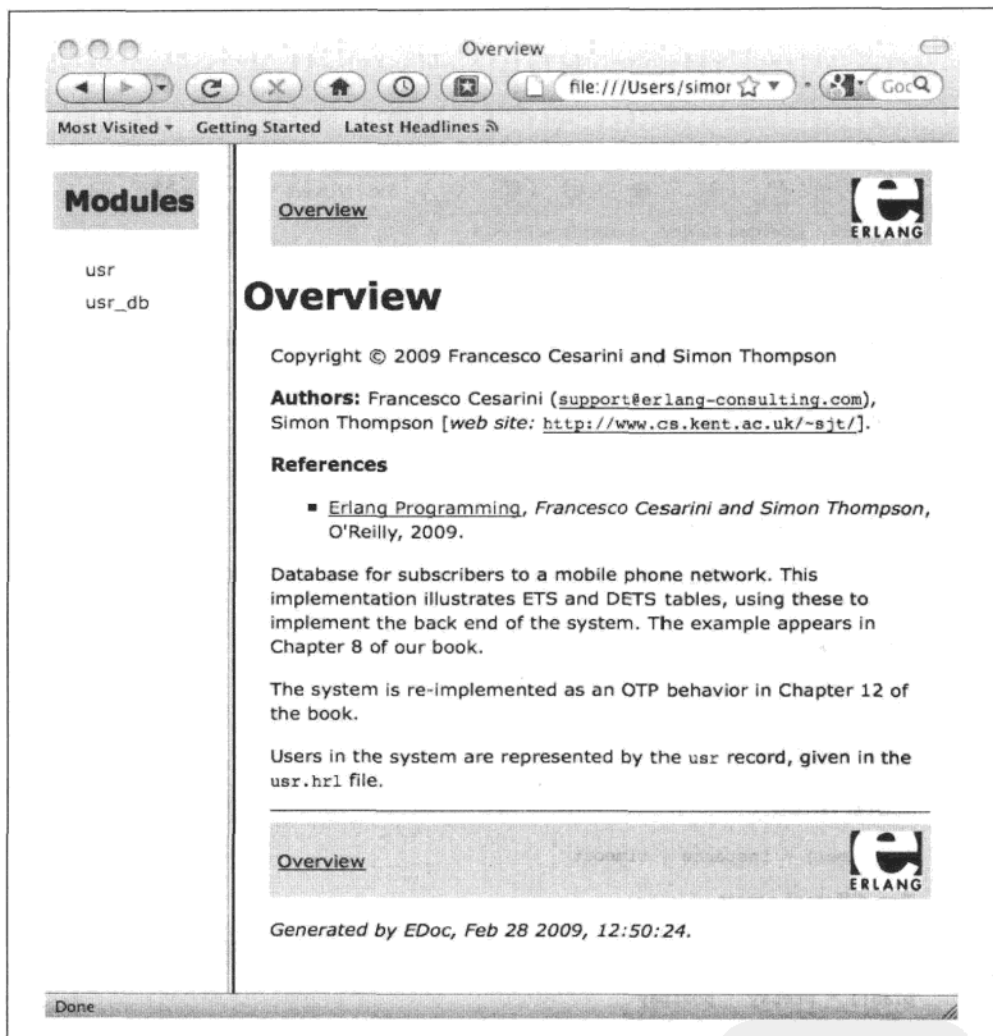


图18-2: EDoc数据库的概括页

```
%% @type plan() = prepay|postpay. The two payment types for mobile subscribers.
%% @type status() = enabled | disabled. The status of a customer can be enabled
%% or disabled.
%% @type service() = atom(). Services are specified by atoms, including
%% (but not limited to) 'data', 'lbs' and 'sms'. 'Data' confirms the user
%% has subscribed to a data plan, 'sms' allows the user to send and receive
%% premium rated smses, while 'lbs' would allow third parties to execute
%% location lookups on this particular user.
```

这些是从usr模块的文档中提取出来的。如果在函数的@spec中使用一个定义的类型，那么会把它链接到它的定义上。有关类型的信息会出现在文档的一般性描述之后，并在函

数索引之前。图18-3显示了usr.erl的文档片段，其中数据类型定义出现在模块描述之后。

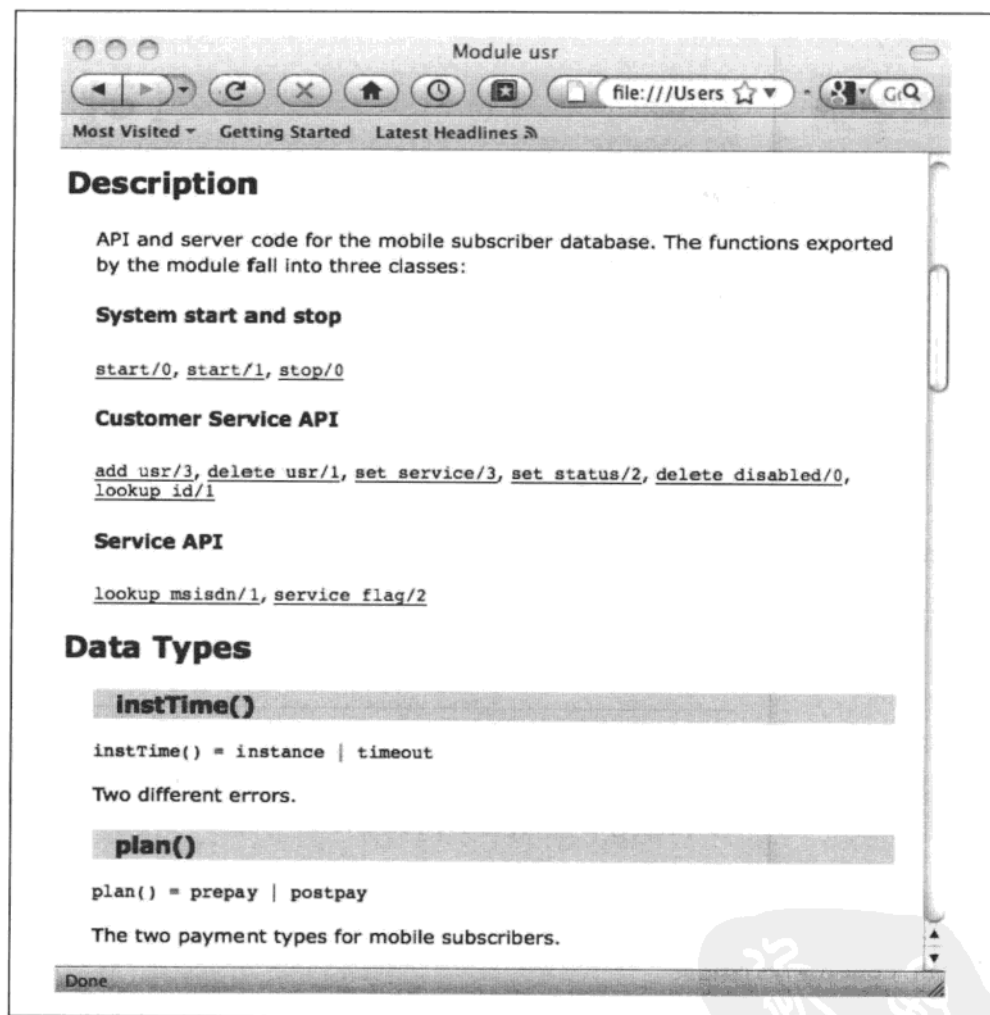


图18-3: usr.erl的网页片段

进一步了解EDoc

如果你想为应用程序和其他项目生成完整的文件，EDoc提供了一些工具来进行格式化、交叉引用以及预处理。

EDoc带有一组预定义的宏，可以把它们放在括号里来调用它们，如{@name}和{@name argument}。它们包括@date、@time、@module、@version，以及{@link reference}。

description}], 它为reference标识的对象创建一个链接link, description则给出链接的锚的文本。

References包含这个模块中函数的name/arity和模块Mod中函数的Mod:name/arity。typename()是指在同一模块中的类型, 而Mod:typename()则指在其他模块中的类型。因此下面的链接:

```
{@link set_status/2. 'set_status/2'}
```

给出了在相同模块中函数set_status/2的文档链接。

在这个例子中, 该链接也标记为set_status/2: 你可以使用另一个EDoc工具来逐字引用。对文本'...'的引用会把封装好的文本放入逐字(代码)格式, 正确地忽略掉文本中任何XHTML的重要符号例如<、=和>。图18-3也给出了一个使用@link结构的例子, 其中文档的描述部分给出函数的索引, 该索引按照函数而不是字母顺序划分。

为了方便作者, EDoc支持维基风格的格式(wiki-style formatting)而不是XHTML。空行用来分隔段落(例如: <p> ... </p>元素)。头可以自动从==Heading==中生成, 类似的也可以从===中生成, 但是, 不能使用=。这个头标记会自动生成一个锚点(和相同), 这个锚点被Edoc的宏{@section Heading}引用。有关EDoc中格式化和交叉引用的更多详细信息请参见在线文档。

练习

我们并没有给出本章的正式练习, 而只是鼓励你在本章之前的练习的答案上试用TypEr、Dialyzer和EDoc。



EUnit和测试驱动开发

当编写程序的时候，你怎么知道这个程序将如何运行呢？你也许会在脑子里面形成一个模型，这个模型定义了该程序会做什么。但是只有使用你的程序或者与它以某种方式互动时，才能确定这个程序真正的功能。第18章展示了如何使用-spec来表示预想的一个函数的输入和输出的类型是什么，TypeR可以检查它是否和代码本身一致。

类型并不能告诉你一个程序如何运行，但是测试是了解你的代码如何工作的最好方法之一。其实本书从头到尾我们都在非正式地这样做。每一次给出一些定义，我们马上就到Erlang的终端中尝试它们。在Erlang中进行开发时，你的代码和测试循环往往都很小。你写完一些函数然后测试它们。你再添加一些函数，然后再次测试它们。每次在终端中重复所有这些测试往往会变得既耗时且容易出错。

在本章中，我们将会介绍EUnit工具，它给出了一个在Erlang中进行单元测试的框架。我们会说明如何在实践中使用它，也会讨论它是如何支持软件工程领域的测试驱动开发（test-driven development, TDD）。

测试驱动开发

软件开发中的瀑布模型把一个软件系统的开发看做几个步骤：首先说明系统的需求。然后，在这个基础上进行系统设计。只有这些都完成的时候，才能开始实现。难怪这么多的软件项目未能兑现用户的需求！不用说，对于Erlang以及其他任何编程语言，瀑布模型都是灾难性的。

测试驱动的方法将上述开发步骤颠倒了顺序。实现是从第一天开始的，而且有一个焦点：每一个要实现的特性都通过一个测试集合表明其特性。增量的开发被添加到代码里，并且只有在通过测试之后才会把它添加到代码里。同时，早期的测试不能因此而失效，而且它也必须通过回归测试。

这种方法会使开发人员获益，因为他在每个阶段都有明确的目标：通过测试。同时也会使用户满意，因为他可以持续地与系统交互或者测试这个系统：每一次不断演变的系统得到构建，他都可以了解到它是否如预想的那样工作。

测试驱动开发（TDD）已经和敏捷编程（agile programming）结合到了一起，但是把这两者等同是错误的。从很早的LISP系统开始，函数编程就具有一个非正式的测试驱动方法的特点了：大多数的函数编程系统在它们的顶层都有一个读取－评估－打印（read-evaluate-print）循环，它鼓励基于测试或实例的方法。Erlang从这个传统里产生，它的根源是函数编程和Prolog，Erlang终端中相互作用的例子展示了实践中的这个非正式的测试驱动方法。

测试驱动开发并没有被限制在单人的项目中。有业界传闻证实说，测试驱动开发在更大型的软件项目中效率也很高，主要是因为它意味着开发团队更早地忙于处理潜在的问题，处在一个能够开发出比采用传统方法更全面的测试套件的流程中。因此，这将导致交付更好的软件。毫无疑问，Erlang和测试驱动开发是携手并进的。在本章的剩下部分中，我们将讨论EUnit系统，它将提供对更正式的测试驱动方法的支持，EUnit系统也包含在标准Erlang发布包中。

EUnit

EUnit提供了一个定义和执行单元测试的框架，它可以用来测试一个特定的程序单元（在Erlang中是一个函数或函数的集合）是否像预想的一样。这个框架可以表示许多不同种类的测试或者一组让EUnit测试更容易编写的宏。

对于一个没有边界效应的函数，测试主要用来检测输入输出的行为是否和预想的一样。而且，它也可以进一步测试函数是否（只有）在要求时产生了一个异常。

有边界效应的函数要求更复杂的基础架构。边界效应包括这样的操作，它们可能影响持久性数据结构，例如ETS表、Dets表，甚至操作系统结构（例如文件和I/O），也包括另一些操作，它们包含了并发进程之间传递的消息。需要测试的这些程序的基础架构包括以下内容：

- 测试带有边界效应的程序一般在检查一个特定操作的行为前，需要一些设置和数据的初始修改。在这之后需要进行程序状态的清理。
- 在并发程序的测试单元中一般需要一个测试台，在这个测试台中，在其他组件期望和被测试单元互动的地方编写了一些模拟对象或占位程序。

EUnit支持测试带有边界效应的和单行的程序。让我们从一个最简单的函数测试的例子入手。

如何使用EUnit

EUnit提供了一个详细的测试表达式和一个让测试更容易编写的宏定义层。它也提供一种收集在一个模块中所有的测试的机制，执行后可以提供关于结果的一份报告。对于一个使用EUnit的模块，它必须包含eunit库：

```
-include_lib("eunit/include/eunit.hrl").
```

在EUnit中，一个单独的测试由一个名为`name_test`的函数标识；一个测试生成函数会被命名为`name_test_`（如用一个`_`结尾）。我们在“EUnit的基础架构”一节中定义了一些测试生成函数。

在包含了EUnit的头文件后，模块(`mod`, `say`)的编译会导入函数`test/0`，这个模块中的所有测试都需要通过调用`mod:test()`来运行。

把测试函数从模块(`mod`, `say`)中分离出来放到模块`mod_tests.erl`中是可能的，模块中也包含了`eunit.hrl`头文件。测试也用完全一样的方法激活，即使用`mod:test()`。

你有可能想在自己的代码上使用EUnit（例如使用`?assert`宏），但是你并不想产生测试。在这种情况下，在包含`eunit.hrl`库的代码行之前，你需要插入以下代码：

```
-define(NOTEST, true)
```

如果这个宏出现在应用的所有模块包含的一个头文件中，它会给出一个单一点来控制是否启用测试。现在我们将在前面介绍的例子的基础上介绍如何编写测试。

函数的测试实例：树的序列化

在第9章的“序列化”小节中，我们看到如何把一棵二叉树序列化为一个列表，然后再从列表重新构造出这棵树。我们演示了一个优化了的版本，而把原来的版本作为练习。在这个没有优化的版本中，表达式的第一个元素递归地给出了遍历整棵树后树的大小：

```
treeToList({leaf,N}) ->
    [2,N];

treeToList({node,T1,T2}) ->
    TTL1 = treeToList(T1),
    [Size1_] = TTL1,
    TTL2 = treeToList(T2),
    [Size2_] = TTL2,
    [Size1+Size2+1|TTL1++TTL2].

listToTree([2,N]) ->
    {leaf,N};
```

```
listToTree([_|Code]) ->
  case Code of
    [M|_] ->
      {Code1,Code2} = lists:split(M,Code),
      {node,
        listToTree(Code1),
        listToTree(Code2)
      }
  end.
```

函数treeToList/1把树转换为一个列表，而listToTree/1应该把那个表达式转换回原先的树。现在我们可以以一些树为例子来测试这些函数：

```
tree0() ->
  {leaf, ant}.

tree1() ->
  {node,
    {node,
      {leaf,cat},
      {node,
        {leaf,dog},
        {leaf,emu}
      }
    },
    {leaf,fish}
  }.
```

我们要求这两个函数轮流应用返回原来的值：

```
leaf_test() ->
  ?assertEqual(tree0() , listToTree(treeToList(tree0()))).
node_test() ->
  ?assertEqual(tree1() , listToTree(treeToList(tree1()))).
```

这里我们使用eunit宏assertEqual来测试两个Erlang的项元值是否相等。

我们也可以测试函数treeToList的特定值：

```
leaf_value_test() ->
  ?assertEqual([2,ant] , treeToList(tree0())).
node_value_test() ->
  ?assertEqual([11,8,2,cat,5,2,dog,2,emu,2,fish] , treeToList(tree1())).
```

在listToTree的测试中，我们的做法有点不一样。这个函数是局部的，我们可以检查发布它在treeToList范围外是未定义的：

```
leaf_negative_test() ->
  ?assertError(badarg, listToTree([1,ant])).
node_negative_test() ->
  ?assertError(badarg, listToTree([8,6,2,cat,2,dog,emu,fish])).
```

这些测试都使用了`assertError`宏，它可以捕获一个引发的异常，并且检查它是否和第一个参数（这里是一个`badarg`）指定的格式相同。

当文件中包含了这些测试后，在终端中运行函数`test/0`将给我们一份简单的报告，说明所有的测试都成功了：

```
2> serial:test().
All 6 tests successful.
ok
```

当测试成功时无须多说，但不是所有的测试都能成功。当测试失败时，EUnit会报告些什么呢？在真实的测试驱动开发的风格中，作为一个例证，我们可以用相同的测试集来回归测试优化版本的序列化，以便查看会发生什么：

```
11> serial2:test().
serial2:leaf_negative_test...*failed*
::error:{assertException_failed, [{module, serial2},
                                   {line, 66},
                                   {expression, "listToTree ( [ 1 , ant ] )"},
                                   {expected, "{ error , badarg , [...] }"},
                                   {unexpected_success, {leaf, ant}}]}
in function serial2:'-leaf_negative_test/0-fun-0-' / 0

serial2:node_value_test...*failed*
::error:{assertEqual_failed, [{module, serial2},
                              {line, 72},
                              {expression, "treeToList ( tree1 ( ) )"},
                              {expected, [11, 8, 2, cat, 5, 2 | ...]},
                              {value, [8, 6, 2, cat, 2 | ...]}}]
in function serial2:'-node_value_test/0-fun-0-' / 1

serial2:node_negative_test...*failed*
... details similar ...

=====
Failed: 3. Aborted: 0. Skipped: 0. Succeeded: 3.
error
```

这份报告表明，6个测试中有3个失败了，并且给出了关于失败原因的详细反馈。在`leaf_negative_test`的例子中，特定函数意外成功了，而不是引发一个`badarg`异常。在第二个例子中，由于实际结果和实际值不同，所以两者都在报告中输出了。

失败的测试要么包含导致原函数失败的值，要么对新实现中的改变很敏感，其中列表表达式的细节被改变了。起始的两个测试检查轮流地应用函数是否返回原来的参数，结果证实，函数的关键属性仍然保有。

注意：如果你想把测试函数放到一个独立的`serial_tests`模块中，你可以使用一个导入指令来包含测试，而不用对串行模块做任何改变：

```
-module(serial_tests).
-include_lib("eunit/include/eunit.hrl").
-import(serial,
      [treeToList/1, listToTree/1
       tree0/0, tree1/0,]).

leaf_test() ->
    ?assertEqual(tree0() ,
                  listToTree(treeToList(tree0()))).
... etc ...
```

EUnit的基础架构

在本节中，你将学习EUnit系统的基础部分，可以用它来建立测试和测试集合。

断言宏

EUnit的基本构建块是一个单一的测试，由函数`..._test()`给出。在之前例子的代码的右边，我们可以使用`assertEqual`和`assertError`来检查数值和异常。其他的断言宏如下所示：

`assert(BoolExpr)`

不仅可以在测试中使用，也可以在程序的任何地方检查在那个点的一个布尔表达式的值。

`assertNot(BoolExpr)`

与`assert(not(BoolExpr))`等价。

`assertMatch(GuardedPattern, Expr)`

将对`Expr`求值，如果它和被保护的的模式匹配失败，将会在`test()`上报告异常。

`assertExit(TermPattern, Expr)`和`assertThrow(TermPattern, Expr)`

将会测试一个程序的退出或一个异常的抛出，功能与`assertError`相似。

在例子中我们使用`assertEqual(E, F)`而不是`assert(E == F)`，这是因为当测试失败时，`assertEqual`会产生更多有用的信息。

测试生成函数

除了一个单一测试，你可以定义测试生成函数来把许多测试组合到一个单一的函数里。一个测试生成器会返回一个表达式，它代表着一组将要在EUnit中执行的测试。

表示一个测试最简单的办法是一个没有参数的fun表达式：

```
leaf_value_test_() ->
  fun () -> ?assertEqual([2,ant] , treeToList(tree0())) end.
```

在先前的代码中，已经加粗表示与leaf_value_test定义中的不同的地方。宏的库允许把它写得更简洁：

```
leaf_value_test_() ->
  ?_assertEqual([2,ant] , treeToList(tree0())).
```

在这段代码中，宏_assertEqual与assertEqual扮演着同样的角色，但_assertEqual是测试表达式而不是测试本身。

一般来说，一个测试生成函数会返回一组测试。例如，下面的代码会把两个测试封装到一个单一的函数中：

```
tree_test_() ->
  [?_assertEqual(tree0() , listToTree(treeToList(tree0()))),
   ?_assertEqual(tree1() , listToTree(treeToList(tree1())))].
```

当EUnit运行这个测试时，将执行所有在列表中的测试。

EUnit测试表示法

EUnit用许多种不同的方法表示测试和测试集合。这里列出了一些最有用的。完整的描述可以在EUnit文档中找到。通过调用eunit:test(TestRep)执行一个测试表达式TestRep：

简单的测试对象

最简单的测试对象是一个空fun，也就是说，一个不带参数的fun。一个简单的测试对象也可以由一对基元{Module, Function}给出，它指模块中的一个函数。

测试集合

测试集合由列表和深度列表给出。在模块中也可以用模块的名字（基元）来代表这些测试。

原语

原语不包含作为参数的嵌入测试集合，它是测试的描述，这些测试存在于一个模块中，如{module, Module}，路径中如{dir, Path::string()}，或者应用及文件中。生成器也可以用{generator, GenFun::(() -> Tests)}嵌入。

控制

可以控制测试执行的方式和地点。

```
{spawn, Tests}
```

将在一个单独的子程序中运行测试，而测试进程将等待直到测试结束。

```
{timeout, Time::number(), Tests}
```

将运行测试Time秒，时间一到马上终止所有没有结束的测试。

```
{inorder, Tests}
```

将按照严格的顺序执行测试。

```
{inparallel, Tests}
```

将在可能的地方并行地运行测试。

固定装置

固定装置为一系列要运行的特定测试提供设置和清理的支持，我们会在“测试基于状态的系统”一节详细讨论它们。

测试基于状态的系统

要解释这个题目，我们需要回到移动用户数据库的例子中。当在第10章中介绍这个例子时，我们在Erlang终端测试了它（参见“移动用户数据库实例”一节）。本节将介绍如何将这种测试风格包含到EUnit中。我们也将更详细地讨论表达测试的方法。

固定装置：设置和清理

基于状态的系统有一个特性：你只能在设置完全正确的配置后才能测试特定的性质。一旦你设置好，测试就能进行。但是，你需要清理系统，为任何的进一步测试做准备。

我们要写的第一个测试在create_tables("UsrTabFile")创建表格后进行，测试在一个空数据库中的搜索：

```
?_assertMatch({error,instance}, lookup_id(1))
```

测试运行后，我们删除文件UsrTabFile进行清理。这可以编写一个固定装置来完成：它是一个测试描述，允许设置和清理。最简单的固定装置格式如下所示：

```
{setup, Setup, Tests}
{setup, Setup, Cleanup, Tests}
```

这里，对于一种类型T：

```
Setup   :: (() -> T)
Cleanup :: (T -> any())
```


Setup函数在Tests之前执行，返回一个类型为T的值X。测试后执行Cleanup(X)。这允许那些在设置时得到的信息（如有关进程标识符或表的信息）能和清除阶段相互交流。

对于我们的例子可以这样写：

```
setup1_test_() ->
{spawn,
 {setup,
  fun () -> create_tables("UsrTabFile") end,      % setup
  fun (_) -> ?cmd("rm UsrTabFile")                end, % cleanup
  ?_assertMatch({error,instance}, lookup_id(1))
 }
}.
```

注意在清除时我们可以使用?cmd宏调用外部的Unix指令删除文件，另外测试是通过调用eunit:test/1来执行。要测试数据库正常工作，我们需要更详细的设置。在生成的进程终止时，程序创建的ETS表将被销毁：

```
setup2_test_() ->
{spawn,
 {setup,
  fun () ->
    create_tables("UsrTabFile"),
    Seq = lists:seq(1,100000),
    Add = fun(Id) -> add_usr{#usr{msisdn = 700000000 + Id,
                                id = Id,
                                plan = prepay,
                                services = [data, sms, lbs]}}
    end,
    lists:foreach(Add, Seq)
  end,
  fun (_) -> ?cmd("rm UsrTabFile") end,
  ?_assertMatch({ok, #usr{status = enabled}} , lookup_msisdn(700000001) )
 }
}.
```

我们可以通过调用Mod:test()或eunit:test(Mod)来运行这两个测试，其中Mod是包含这些测试的模块名称。

在Erlang中测试并发程序

当一个程序包含许多并发演变对象时，例如EUnit这样的单元测试框架很难看出有直接的帮助。尽管EUnit在系统演变中为一个表达式应用到一个特殊点提供了基架，但监控系统本身的演变变化却很困难。测试并发系统的最大的挑战是竞争条件。运行你的测试，你可以毫不费力地再现错误。但当你启用这些进程的跟踪，然后突然所有东西都像预期的那样运行了，这是因为额外的I/O导致你的进程以不同的顺序执行。

一些EUnit的功能是很有用的。在程序中的任何一个点`?assert`一个属性是可能的，这样可以在先决条件或后决条件以及系统恒定性被打破时进行监控。EUnit也提供对调试的支持，它用消息向Erlang终端报告而不是使用标准输出。这些宏包括以下内容：

`debugVal(Expr)`

将输出源代码和当前的Expr值。结果总为Expr的值，因此这个宏可以出现在程序的任何表达式周围，而且不影响其功能。

`debugTime(Text, Expr)`

将输出Text，然后输出Expr的（经过的）执行时间。

如果在模块包含eunit头文件之前设置NODEBUG宏（设置为true），那么在那个模块里面的宏都不起作用了。

其他支持并发程序测试的系统包括Quviq Quick-Check（Quviq快速检查，它为Erlang执行基于属性的随机测试）（注1）；McErlang是用Erlang写的一个Erlang的模块检查器（注2）；还有通用测试，它基于OTP测试服务应用的系统测试框架，也是标准Erlang发布包的一部分）。

练习

练习19-1：测试顺序函数

回到第3章的练习中，为你的解决方案设计EUnit测试：你的解决方案是不是通过了所有的测试呢？是你的解决方案有错误还是你定义的测试方法有问题（如果测试失败的话）？

练习19-2：测试并发系统

在EUnit中为第4章介绍的echo进程设计测试。当把实现修改为注册进程时，你必须如何改变测试？

注1： 参见网址：<http://www.quviq.com/>。关于工具的一个先前版本及其在测试通信软件中的应用可以参见在<http://doi.acm.org/10.1145/1159789.1159792>中的讨论。

注2： 参见网址：<https://babel.ls.fi.upm.es/trac/McErlang/>。工具的一个先前版本可以参见在<http://doi.acm.org/10.1145/1291220.1291171>中的讨论。

练习19-3：软件更新

你会如何为第8章中的db_server例子的一个软件升级定义EUnit测试？

练习19-4：测试OTP行为

把第12章中给出的测试例子包含到EUnit的框架中。

练习19-5：为OTP行为制定测试

为第12章中练习的答案设计EUnit测试。



风格和效率

贯穿全书，我们已经介绍了在Erlang编程中该做的和不该做的。我们介绍了良好的实践经验和有效率的`结构`，同时也指出了错误的实践经验、低效的方式及瓶颈问题。在这些准则中，你可能会发现一些和通用的计算有关，一些与Erlang相关，另外一些依赖于虚拟机。学习编写既有效率又优美的Erlang代码不是一蹴而就的。在本章中，我们会总结在开发Erlang系统时所使用的`设计准则`和`编程策略`。我们也将讨论常见的错误和低效的方法，还会涉及内存的处理和程序性能分析。

应用和模块

在Erlang中相互紧密作用的模块集合称为一个应用。Erlang系统由一系列松散联结的应用组成。一个很好的实践是：设计一个应用，它`为其他应用发出的调用提供一个单一的进入点`。把所有向外导出的函数集中到一个模块中，这`为维护代码基础提供了灵活性`，因为任何对“内部”模块的修改对外部的使用者都是不可见的。这个模块的文件给应用的接口提供了一个完整描述。这个单一的进入点也有助于跟踪和调试应用里的`呼叫流程`。在大型系统中，用一个简短的首写字母给应用中的模块加上前缀是一个好的实践经验。这样可以确保在不同应用模块名称的选择永远不会重复，否则，两个应用在任何大型系统中一起使用时将会引出问题。

模块是你基本的建设块。当设计你的模块时，应该尽可能地减少导入的函数的数量。只要涉及模块的使用者，模块的复杂度就和导出的函数数量成比例关系，因为模块的使用者只需要理解导出的函数。拥有尽可能小的接口也给应用的维护者提供很大的灵活性，因为它使重构内部代码变得更容易。

你应该尽可能地减少模块之间的依赖性。维护一个可以调用许多模块的模块比维护只对部分模块进行调用的模块要难。降低相互依赖性不仅对被调用模块的维护有帮助，同时也使对它们的重构更简单，因为只需维护少量的外部调用。

模块间的相互依赖性应该形成一个非循环图（见图20-1），也就是说，不应该存在模块X，它（在整个依赖性链中）在整个依赖性链中依赖于另一个模块，而这个模块又依赖于模块X。

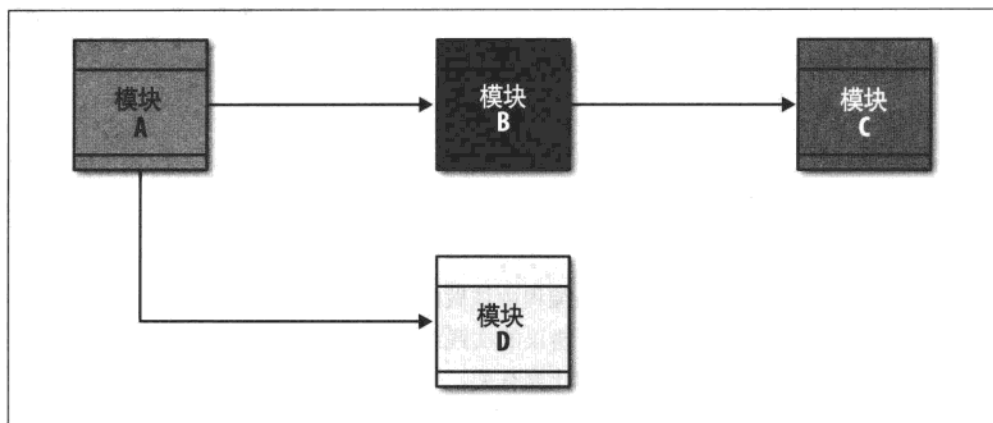


图20-1：交互模块的依赖性

在一个特定的模块中，应该把相关的函数放在一起（注1）。例如，你应该把诸如start、stop、init和terminate这样的函数放在一起，把所有的消息控制函数放在模块的相同部分。把相关的函数放在一起让你的代码更容易理解和检查，特别是你在一个函数中做一些事情或者打开一些东西，而在另一个函数中不做或关闭它们时，这样做的效果更为明显。

库

你应该把共同使用的代码都放到库中。库应该是相关联的函数的集合，这些函数可能操作相同类型的数值。如果可能，应确保库函数没有边界效应，因为这样做可以增强它们的可重用性。而那些有边界效应的函数，如消息传递、破坏性的数据库操作以及导入/导出，你应该尽量保证所有带边界效应的相关操作（例如所有那些处理一个特定ETS表的操作）包含在一个单一的库模块中。

脏代码

脏代码由那些“不应该做”但是无论如何又迫不得已做的东西组成。避免写脏代码有时候非常困难，有时甚至不可能。

注1：在Chris Ryder的博士论文“Software Measurement for Functional Programming”（University of Kent, 2004）中，研究表明这是程序员必然会做的事情。

脏代码包含对进程字典的使用，内置函数`process_info/2`的使用，或者进行了关于内部数据类型或其他内部结构假设的代码的使用。例如，你可以使用内置函数`process_info/2`查看消息队列的长度或者偷看信箱中的第一条消息。你可能会想在进程字典里面存储一个全局变量，或者有一个查看记录类型的动态函数，而你一般用它转发一个请求到一个回调模块。

把复杂代码和脏代码隔离在单独的模块中非常重要，并且要竭尽所能地减少和避免使用这种代码。为脏代码编写文档，并且说明脏代码以什么方式为脏，以便任何修改你代码的人对你在写脏代码时的假设一目了然。这些是至关重要的。

接口

为你所有的导出接口编写文档。文档应该包含参数值、可能的范围以及返回值。如果你的函数有边界效应，如消息传递、数据库更新、导入/导出，你也应该包含这些信息。要为说明和协议提供索引，并且为所有主要的数据结构编写文档。注释要清楚，提供有用的信息，不要有多余的东西。你应该描述函数能做什么，而不是怎么做。在“使用EDoc生成文档”一节中提供了一个例子，例子的完整版本可以在本书的网站上找到。

决定为什么导出一个函数。一个很好的经验是，把导出指令根据下面的函数类型划分为不同种类：

- 使用者的接口函数；
- 模块之间的函数（在相同应用里使用，但不导出到其他地方）；
- 内部导出（在相同模块的内置函数中使用，例如`apply/3`和`spawn/3`）；
- 标准行为的回调函数（例如`init`、`handle_call`等）。

显而易见，这取决于你有多少函数要导出。根据以往经验：如果你导出的语句多于一行，那么就该将它分解。你可以在一个导出指令中包含一些客户端函数，如`start/0`、`stop/0`、`read/1`和`write/2`，而在其他地方包含一些回调函数，如`init/1`、`terminate/2`和`handle_call/3`。这让任何一个读代码而不是读文档的人能够区分被导出接口和内部导出或回调函数。

只有当你开发代码时才能使用`-compile(export_all)`指令（即使是在这种时候，你也必须在迫不得已的情况下才能使用）。在开发工作完成后，不要忘记删除这个指令，这往往不会发生（注2）。另一个替代的办法是把如下信息作为一个模块的指令包含进来，让它成为一个如下所示的可选项：

注2： 即使是在本书中！

```
compile:file(foo, [export_all,...]).
```

返回值

如果一个函数不是一直成功，那么给它的返回值添加标签。如果你不这么做，一个正的结果可能会被理解为一个负的。在下面的例子中，如果你使用列表[{0,true}, {1,false}, {2,false}]来对关键字1进行关键字搜索会发生什么？你怎么区分这个关键词的返回值：当找不到这个条目的时候，基元false作为终止条件返回。

```
keysearch(Key, [{Key, Value}|_]) -> Value;
keysearch(Key, [_|Tail])         -> keysearch(Key, Tail);
keysearch(_key, [])              -> false.
```

Erlang的方法是使用标准的返回值的格式：ok、{ok, Result}或{error, Reason}。在我们的情况中可以把使用{ok, false}的一个成功的查寻和使用基元false的一个失败的查询区分开。keysearch/2正确的实现为：

```
keysearch(Key, [{Key, Value}|_]) -> {ok, Value};
keysearch(Key, [_|Tail])         -> keysearch(Key, Tail);
keysearch(_key, [])              -> false.
```

另一种解决方法是，在没有找到对应的关键字的情况下产生一个异常。采用这个办法，每次都需要在try ... catch结构中使用keysearch，以便处理异常的情况。

如果你知道这个函数总是成功，并且返回一个单一的值，如true或false，或者返回一个整数、基元或合成的数据类型。这将允许这个函数的返回值作为参数传递给另一个函数，而不用检查是成功或失败。

你应该总是选择那些使调用者任务简单化的返回值。这么做可以使代码更紧凑、更易读。例如，如果你知道get_status/1总是成功，当你只需要返回Status时，为什么还要写一个带标签的返回值，你还得从中提取返回值：

```
{ok, Status} = get_status(BladeId),
NewStatus = reset(BladeId, Status)
```

当你所要做的只是返回Status呢？这样做可以允许函数调用作为参数传递给reset/2调用。

```
NewStatus = reset(BladeId, get_status(BladeId))
```

不要为调用者将会对函数结果所做的事情做任何假设。在下面的例子中，我们假设函数的调用者要输出一条错误消息，说明我们想要提高税收的人不是瑞典人：

```
tax_to_death(Person) ->
  case is_swede(Person) of
    true ->
```

```

    {ok, raise_taxes(Person)};
{error, Nationality} ->
    io:format("Person not Swedish:~p~n",[Nationality]),
    error
end.

```

现在不做假设，相反地，我们允许这个函数的调用者根据这个调用的返回值做出决定：

```

tax_to_death(Person) ->
    case is_swede(Person) of
    true ->
        {ok, raise_taxes(Person)};
    {error, Nationality} ->
        {error, Nationality}
    end.

```

另一种看待这个例子的方式是，通常，一个函数做一件事情是最好的：在这种情况下执行更新或输出一个特定格式的错误消息。如果你发现有一个函数在做两件事情，你可以把它重构为两个单独的函数，每一个都可以单独地重复使用。

内部数据结构

不要让私有数据泄漏。私有数据结构的所有细节应该从接口全部抽象出来。在抽象化接口时，封装函数的使用者不需要的信息。设计要有灵活性，这样引入内部数据表达的任何变化都不会影响到导出的函数接口。下面的队列模块：

```

-module(q).
-export([add/2, fetch/1]).

add(Item, Q) -> lists:append(Q, [Item]).

fetch([H|_]) -> {ok, H, _};
fetch([]) -> {error, empty}.

```

可以像这样使用：

```

NewQ = [],
Queue1 = q:add(joe, NewQ),
Queue2 = q:add(klacke, Queue1).

```

在先前的代码中，我们泄漏了这样一个事实：当把变量NewQ绑定到一个空队列上时，队列的数据结构是一个列表。这样并不好。q模块应该已经导出了函数empty() -> [], 我们可以使用这个函数来创建需要绑定到变量NewQ的空队列：

```

Newq = q:empty(),
Queue1 = q:add(joe, Newq), ...

```


现在可以重写队列的实现了，例如，作为一对列表来分别保存队首和队尾，而不用重写使用队列模块的任何代码。这其中也有一个例外：如果你想判定两个队列是否相等。在这种情况下，你将需要为队列表达定义一个相等性测试函数，因为相同的队列在这个新的实现中可以以不同的方式表达。

进程和并发

进程是一个Erlang系统中的基本构成元素。Erlang程序设计的一个基本原则，是在Erlang程序中的并发进程和建立模型的系统中的并发活动之间，建立一对一的映射。这样会积累非常多的同步进程，因此要尽可能地避免不必要的进程互动和不必要的并发。确保你为每一个并发活动创建一个进程，而不仅仅是针对两个或三个进程。如果你有面向对象编程的经验，这并不等同于为每一个对象或该对象的每一个方法创建一个进程。或者，如果你正在管理登录到一个系统的用户，这可能不等同于为每一个会话创建一个进程。相反，你将会为进入系统的每一个事件创建一个进程。这等同于大量的短暂进程，同时只有非常有限数量的持久进程。

你应该总是在一个模块里实现一个进程循环和它的外围函数。避免把多于一个的进程循环的代码放在相同模块中。因为当你想理解是谁在执行什么的时候，上述做法将让人非常容易混淆。你也许会有许多执行相同代码基础的进程实例，但是应确保在模块中的循环是唯一的。

在一个函数接口上隐藏所有的消息传递是为了更大的灵活性以及信息隐藏。在你的客户端代码不使用下列表达式：

```
resource_server ! {free, Resource}
```

而用下面的函数调用代替：

```
resource_server:free(Resource)
```

像进程一样在相同模块中放置客户端函数。客户端函数是这样的函数：它们由其他进程调用，产生一条消息并发送到在模块里定义的进程中。这样做让人们可以很容易地理解消息流而不用在模块之间跳转，更不用找出接收消息的语句是在哪个模块中编写。从你的角度看，它减少并简化了你必须生成的文档，因为你只需要描述函数的API而不是消息流。你也会得到更灵活的代码，因为你隐藏了以下信息：

- 资源服务器是一个进程；
- 代码使用别名resource_server注册；
- 客户端和服务端端之间的消息协议；

- 调用是异步的事实。

如果你使用一个函数接口，你可以灵活地修改所有这些而不会影响客户端代码。

注册进程应该尽可能和实现它们的模块有相同的名字。这可以使实时系统的故障诊断更容易，并提高代码的可读性和可维护性。注册进程应该有一个长生命期，你永远不应该在注册和注销它们之间来回切换。由于基元使用的存储空间是不被垃圾回收的，因此应避免给注册进程动态创建基元，这样做会导致内存泄漏。

系统的进程应该有定义良好的行为和角色。你应该在使用12章中介绍的OTP行为包时慎重考虑，这些包括：服务器、事件句柄、有限的状态机、监控进程以及应用。

当使用消息传递时，所有的消息应带有标签，这会使在receive声明中的语句顺序变得不重要，结果它使新消息的添加变得简单，并且不会改变现有的行为，这样也降低了出错的风险。

避免在receive语句中只对非捆绑的变量进行模式匹配。在下面的例子中，如果你想让你的进程也可以处理消息{get, Pid, Variable }会怎么样？

```
loop(State) ->
  receive
    {Mod, Fun, Args} ->
      NewState = apply(Mod, Func, Args),
      loop(NewState)
  end.
```

可以很容易看出，你必须用模式{Mod, Fun, Args}来匹配上面的消息。但如果你的receive声明和更多的消息匹配又会发生什么呢？答案并不那么显而易见。或者如果你想模块get中应用一个函数会得到什么结果？通过给消息打上标签，你完全会具有很大的机动性来重新安排消息的次序，并且避免在错误的语句中进行不正确的模式匹配：

```
loop(State) ->
  receive
    {apply, Mod, Fun, Args} ->
      NewState = apply(Mod, Func, Args),
      loop(NewState);
    {get, Pid, Variable} ->
      Pid ! get_variable(Variable),
      loop(State)
  end.
```

当使用receive语句时，你不应该接收未知消息。如果已经收到，这些消息应当被当做错误处理，并且你应该允许使你的系统崩溃，就像一个OTP应用的做法一样。如果你不想让系统崩溃，那么请确保你把未预料到的消息写入日志。这么做让你可以查明它们从

哪儿来，并且诊断如何处理它们。你也许会发现这些消息源自端口、套接字或在单元测试过程中应该获得的错误。

如果你不处理未知消息，你会注意到系统的CPU使用率开始增加，并且响应时间在减少，直到Erlang运行时系统内存耗尽并且崩溃。增加的CPU使用率和反应时间可以这样解释：无论何时一个进程接收到一条消息，在匹配满足之前，它必须遍历信箱中潜在的成百上千的消息。不匹配的消息越多，泄漏的内存也就越多。

当你收发消息时，你应该对客户端隐藏内部消息协议，并且一直使用客户端函数的名字来给消息打上标签，因为这使得你从客户端函数跳转到处理该消息的进程循环变得轻而易举：

```
free(Resource) ->
    resource_server ! {free, Resource}.
```

应该使用引用。在类似的响应和请求源自不同的进程的复杂系统中，可以使用引用来唯一地把响应从指定的请求中识别出来：

```
call(Message) ->
    Ref = make_ref(),
    resource ! {request, {Ref, self()}, Message},
    receive
        {reply, Ref, Reply} -> Reply
    end.

reply({Ref, Pid}, Message) ->
    Pid ! {reply, Ref, Message}.
```

小心对待超时设定。如果你使用它们，总是刷新最新到达的消息。如果你不这么做，你的下一个请求将会导致前一个的超时响应。你可以这样来避免这个问题：使用引用并且假设没有其他的进程会发送格式为{reply, Ref, Reply}的消息。只刷新你没有识别的消息：

```
call(Message) ->
    Ref = make_ref(),
    resource ! {request, {Ref, self()}, Message},
    wait_reply(Ref).

wait_reply(Ref) ->
    receive
        {reply, Ref, Reply} -> Reply;
        {reply, _, Reply} -> wait_reply(Ref)
    end.
```

当一个进程可能会终止时，使用链接或监控器比使用超时设置可以更好地处理这种情况。

要对捕捉退出进行严格限制，这么做的时候，不要来回切换。一定要记得刷新退出信号，并且在链接和取消链接进程时清楚知道竞争条件。当你链接到一个进程时，也许它已经崩溃了；当你取消链接时，它可能已经终止，而它的EXIT信号将还在进程信箱中等待被读取。

把错误修复代码和正常代码分开，因为把它们结合在一起会增加代码的复杂度。这也确保了崩溃和修复策略的处理是一致的。永远不要试图解决一个本不应该发生并继续存在的错误。如果一个意外的错误发生了，那么就让你的进程崩溃并且把它留给监控进程处理。在你试图处理错误的时候，很有可能产生比你认为解决了的更多的错误。在下面的例子中，如果变量List不是一个列表，你会怎么做？

```
bump(List) when is_list(List) ->
    lists:map(fun(X) -> X+1 end, List);
bump(_) ->
    {error, no_list}.
```

在调用bump/1的每一个地方，你必须在一个case语句中满足两个返回值，然而这也不能让读代码的人更聪明地弄清楚：为什么List会被损坏。不要用防御性编程，用下面代码代替：

```
bump(List) ->
    lists:map(fun(X) -> X+1 end, List).
```

如果List不包含列表，那么在lists:map/2中会发生运行时错误，并且会导致你的进程终止。让你的管理程序处理退出信号，并让它决定与它管理的其他进程一致的恢复策略。确保记录了崩溃并且事后调试时可以使用它。在一个单独的日志进程中记录这些错误和崩溃。

由于我们应该隐藏在函数接口后面传递的进程和消息，确保依赖性形成一个非循环图，同时也确保进程之间没有死锁。在Erlang中死锁非常少见，但是如果并发没有经过深思熟虑或者没有正确设计，它也会发生。确保你的模块依赖性没有循环是朝正确方向迈出了一步。

格式约定

程序写出来不仅是给计算机执行的，也是给它们的作者和程序员阅读和理解的。用一种大家都容易阅读和理解的方法来写程序，不仅可以帮助你在6个月没有碰它之后，仍然记得你的程序做什么，还可以帮助其他要使用或修改你的程序的程序员。它也可以让其他人员查找错误或其他问题以及理解调试信息变得更容易。因此，用一致的方法写程序会使每一个人受益。在本节中，我们将给出一系列Erlang编程中常用的格式约定。

首先，避免写深度嵌套的代码（注3），在case、if、receive和fun语句中，永远不要存在多于两层嵌套的代码，下面是如何不去这样做的例子：

```
reset(BladeId, AdminState, OperState) ->
  case AdminState of
    enabled ->
      case OperState of
        disabled ->
          enable(BladeId);
        enabled ->
          disable(BladeId),
          enable(BladeId)
      end;
    disabled ->
      {error, admin_disabled}
  end.
```

一个常用的减少缩进的技巧是创建暂时的组合数据类型。如果要嵌套if和case语句，那么把它们一起加入到一个元组中，并且在一个语句中模式匹配它们：

```
reset(BladeId, AdminState, OperState) ->
  case {AdminState, OperState} of
    {enabled, disabled} ->
      enable(BladeId);
    {enabled, enabled} ->
      disable(BladeId),
      enable(BladeId);
    {disabled, _OperState} ->
      {error, admin_disabled}
  end.
```

你也可以通过在函数头部中插入模式匹配来减少缩进。通过在函数语句里对AdminState和OperState进行模式匹配，我们不仅可以让代码更容易阅读，还把嵌套语句的层数减少到零，并且减少了总代码的长度：

```
reset(BladeId, enabled, disabled) ->
  enable(BladeId);
reset(BladeId, enabled, enabled) ->
  disable(BladeId),
  enable(BladeId);
reset(_BladeId, disabled, _OperState) ->
  {error, admin_disabled}.
```

当case语句更适合时，避免使用if语句。这对于有命令式语言背景，而对模式匹配还感觉不自在的程序员很常见。常常问问自己，是否可以用模式匹配和避免在基元true和

注3：Wrangler的重组工具将指出这点，以及许多其他Erlang代码中的不良风气。Wrangler可以在这里找到：<http://www.cs.kent.ac.uk/projects/protest/>。

false上模式匹配的保护元来把if语句重写为case语句。如果这样做，代码会变得更易读、更紧凑吗？

```
get_status(A,B,C) ->
  if
    A == enabled ->
      if
        B == enabled ->
          if
            C == enabled ->
              enabled;
            true ->
              disabled
          end;
        true ->
          disabled
      end;
    true ->
      disabled
  end.
```

上面的例子是一个错误使用if语句的极端事件。通过创建一个有三个变量A、B、C的组合数据类型，并用case语句代替if语句，我们减少了缩进的层数，使代码更易阅读、更紧凑：

```
get_status(A,B,C) ->
  case {A,B,C} of
    {enabled, enabled, enabled} -> enabled;
    {_status1, _status2, _status} -> disabled
  end.
```

我们的目的在于让你的模块保持在可管理的大小。简短的模块让维护和调试，甚至代码的理解都变得容易。一个可管理的模块不包括注释应该不超过400行代码。可以把很长的模块按逻辑一致的方式分成几个模块。记住，很长的代码并不能解决你的问题。

代码行不应该太长，或者不要越过页面。一行代码永远不要超过80个字符。有很多次，开发者试图通过加宽他们的编辑器窗口直到覆盖整个屏幕来向说服我们它们很长的代码是可读的。这并不能解决你的问题。

如果你的代码越过了页面，请花几分钟来重新格式化一下，这将为下一个维护和调试它的人省下几个小时。下面的代码格式是一个典型的例子：当一个程序员写好了代码，完成了测试，却从不回过头来检查时，那将会发生什么？

```
name(First, Second) ->
  case person_exists(First, Second) of
    true ->
      Y = atom_to_list(First) ++ [$ |atom_to_list(Second)] ++
        [$ |get_nickname(First,
```

```

                                Second)],
    io:format("true person:~s~n",[Y]);
false ->
    ok
end.

```

你可以很简单地像下面这样重写：

```

name(First, Second) ->
case person_exists(First, Second) of
true ->
    Y = atom_to_list(First) ++
        [$ |atom_to_list(Second)] ++
        [$ |get_nickname(First, Second)],
    io:format("true person:~s~n",[Y]);
false ->
    ok
end.

```

经过了第二次迭代再想一想，先前的代码可以转换成下面这样：

```

name(First, Second) ->
case person_exists(First, Second) of
true ->
    NickName = get_nickname(First, Second),
    io:format("true person:~w ~w ~s~n",[First, Second, NickName]);
false ->
    ok
end.

```

如果代码越过了页面，用下面的方法解决这个问题：

- 选择更短的变量和函数名称；
- 使用临时变量把语句和表达式划分为多行代码；
- 在case、if和receive语句中减少缩进的层数；
- 将可能的重复代码转移到不同的函数中。

选择有意义的函数名和变量名。如果名字中包含多个单词，可以用大写字母或者下划线来分隔它们。为了不让两者混淆，一些人会对变量采用一种格式而对函数采用另一种。无论如何，一旦你采用一种格式就坚持在你的整个代码中一直使用它。所有这里讨论的约定都有这样一个相同点：一致性让代码易读，而格式间的突变分散了阅读者的注意力，从而难以理解所发生的事情。

避免使用长名字，因为它们是代码越过页面的主要原因。一个长名字在函数头部看来并没有错，但是想象一下，在语句的第二层和一个很长的变量名称一起调用它会是什么情

况。如果你使用更长的名字，那么在函数内使用它们（这些函数可能会在全部的代码库中使用），比在那些使用范围限制在一个特定函数中的变量中使用更有意义。

使用缩写词和首写字母来缩短你的名字，但是要确保名称有意义并且容易理解。一个常见的隐患是使用相似的名字，因此很容易混淆：Name、Names和Named看起来很相似。如果你使用前缀，那么采用那些可以提供变量类型或函数返回值的线索的前缀。找到有意义的函数名和变量名需要练习和耐心，因此不要小看了这项任务。

当使用“无关紧要”变量时，避免为了使用而使用下划线。即使你不使用这个变量，它的内容也可以对任何读代码的人有意义。在未使用的变量前不使用下划线将导致编译器警告。但要注意，格式为_name的变量是常规的单一赋值变量，因此一旦绑定了它们，它们就不能被当做“无关紧要”变量重新使用。

在下面的例子里，如果变量AdminState绑定到基元disabled，_State也会绑定到disabled上。如果OperState绑定到enabled，第二个case语句就会因为case语句错误而失败，因为没有东西与之模式匹配：

```
restart(BladeId, AdminState, OperState) ->
  case AdminState of
    enabled ->
      disable(BladeId);
      _State ->
        ok
    end,
  case OperState of
    disabled ->
      ok;
      _State ->
        stop(BladeId)
    end.
```

使用记录作为主要的数据类型来存储复合数据类型。如果记录只在一个模块内使用而不用导出时，其定义应该放在模块开头来防止别人在他们的模块中使用它。在许多模块中使用的记录应该放在一个包含文件中，并且正确地用文档说明它们。

使用记录选择器存取一个字段，并使用模式匹配来存取多于一个字段的值：

```
Cat = #cat{name = "tobby", owner = "lelle"},
#cat{name = Name, owner = Owner} = Cat,
Name2 = Cat#cat.name.
```

永远不要使用记录的内部元组表示，因为这违背了记录给表带来的灵活性：

```
Cat = #cat{name = "tobby", owner = "lelle"},
{cat, Name, _owner} = Cat
```


如果像这样写代码，然后在记录中加入一个字段，你就不得不更新使用这个元组表示的代码，即使这个函数不会被这个字段影响。

只有需要的时候使用变量。使用变量有两个原因。第一，当需要使用这个变量时，对它的绑定不止一次；第二，为了让代码更清楚，并且缩短行的长度。你可能想要编写这样的代码：

```
Sin = sin(X),  
Cos = cos(X),  
Tan = Sin / Cos
```

但是直接传递函数返回值到另一个函数会让你的代码更清楚，也更紧凑：

```
Tan = sin(X) / cos(X)
```

总是问问你自己，这个变量是否真的使你的代码更清楚。你在Erlang中编程越多，你会发现你使用变量的次数越少。

使用catch和throw的时候要小心，比起使用catch和throw，使用try...catch的结构总是更可取的。试着减少使用它们，并且确保任何的throw都在相同模块中像catch一样被捕获。就像你在第3章里面看到的一样，只在当你可以安全地忽略函数的返回值时，使用catch才是有意义的。

如果使用catch和throw，一定要确保你也处理了可能被捕获的运行时错误。例如，下面的代码出现在即将发送到产品的代码中：

```
Value = (catch get_value(Key)),  
ets:insert(myTable, {Key, Value})
```

如果一个运行时错误在get_value/1调用中发生了会怎么样？你会抓到实时错误{'EXIT', Reason}，把它绑定到变量Value上，并且储存在ETS表中。

对于使用进程字典要非常小心。事实上，要像避免瘟疫一样避免它。内置函数put和get具有破坏性的操作。它们会让你的函数变得不确定并且难以调试，因为在进程崩溃后，要分析进程字典的内容，尽管有可能，但依然非常困难。可以在你的函数中引入新的参数来代替进程字典。

如果你不知道进程字典是什么（注4），这儿什么也没有介绍，请继续往后阅读。

小心使用指令-import(mod, [fun/arity...])。它会使代码难以理解，并且在开始时

注4： 我们仅仅在这本书中由于这个原因提到它。

让最有经验的Erlang程序员混淆。一个使用它的诱惑是可以缩短行的长度，但是这牺牲了可理解性。在下列这些情况中使用它是有意义的：

- `lists`模块中的公共函数，如`map`、`foldl`和`reverse`，对任何读你代码的人这些都是可理解的。
- 如果想在不变它们的情况下，把EUnit测试从模块`foo`移动到模块`foo_tests`上，将会需要从`foo`中向`foo_tests`导入所有的定义。

形成你自己的Erlang编程风格可能需要许多年。确保你在任何单独的系统中保持一致的风格。这包括你对缩进和空间的使用，你对变量名、函数名和模块名的选择。在大型项目中，通常都会提供给你风格准则。

编码策略

用户在使用系统时应该能够预知将会发生什么。这种决定论的基础来源于你的函数返回结果的可预见性和一致性。一个一致的系统中的模块做相似的事情。这要比其中一个模块用非常不同的方法做类似的事情的系统更容易理解和维护。

减少有边界效应的函数数量，把它们集中到特定的模块中。这些模块可以处理文件或封装数据库操作。有边界效应的函数会引起在系统状态中永久的变化。使用和调试这些代码的强制性知识，使得代码的重用和维护变得困难。试图减少有边界效应函数数量的方法有：最大化纯函数的数量；把边界效应隔离到原子性函数里，而不是把它们和函数的变形结合在一起。

一些最具有挑战性的错误是由竞争条件引起的，特别是那些真正并行地运行在多核系统中的代码。这儿有一个常见的情景：你运行一个测试，并且每次总是可以重现相同的错误。一旦你打开跟踪输出，那些开销会使进程进行得更缓慢，并改变了执行的顺序。突然间，你的错误再也不能重现了。如何才能避免这种情况？让你的代码尽可能地具有确定性。

一个确定的程序是指无论执行顺序如何，总是以相同方式运行，并且提供相同结果或总是出现相同错误的程序。是什么让一个解决方案具有确定性的呢？假设一个监控进程必须并行启动5个工作进程，并且这些进程的启动顺序并不重要。一个非确定的解决方案会生成所有的进程并检查它们是不是正确地启动了。一个有确定性的解决方案会每次只生成一个进程，在确保每个进程正确启动后才继续产生下一个。尽管两者可能产生相同的结果，但非确定性的解决方案可以使初始错误很难再产生，并且基于进程生成的顺序产生不同的结果。虽然确定性并不能保证消除竞争条件，但它确实可以减少竞争条件的数量，让你的系统更具有可预见性，并且也让调试更加直接。

可以把常见的设计模式抽象出来。如果你在两个地方有相同的代码，那么请把它们分离到一个共同函数中（注5）。如果你有模式相似的代码，那么试着通过一个变量或一个单独的函数来定义差异，并在一个通用的调用中把这些模式结合起来。在适合的地方用lists模块中的高阶函数调用来代替列表上的递归函数。你可以把对列表元素的操作封装在一个fun函数中，并且使用lists模块提供的高阶函数选择一种递归模式。要看列表中的元素会发生什么，你只需要检查fun；要看采用了哪种递归模式，可以检查调用了lists模块中的哪个函数。

当你正在设计一个属于自己的库时，你会发现许多函数都遵循相似的模式，例如列表中的fold或map。然后你可以用自己的高阶函数来封装这些模式，并且允许你和其他模块用户抽象他们的函数。随着高阶函数封装了通用的计算模式，使得带有多重子句和终止条件的递归函数变得更紧凑、更易读、更易维护。

要避免防御性编程。如果你的输入数据源于系统内部，请信任它，只有在数据通过外部接口进入时才测试它。一旦数据已经进入运行时系统，那么调用函数就应该负责确保输入数据是正确的，而不是被调用函数。如果用错误的输入数据调用你的函数，建议是让它崩溃。

下面是一个防御性编程的例子，展示了一个函数将基元表示的月份转化为它们各自的数字：

```
month('January') -> 1;
month('February') -> 2;
...
month('December') -> 12;
month(_other) -> {error, badmonth}.
```

增加最后一句，用它作为一个默认匹配的情况可能有很大的诱惑力。如果用错误的基元调用month/1，那么它会返回元组{error, badmonth}。这个返回值会迫使函数的使用者测试和处理这个错误值，或者会在系统的某个地方引起一个运行时错误，其中函数期待一个整数却收到了元组。删除最后的防御性语句也会引起函数语句错误，并且提供调用链和崩溃报告中拼错的基元。

不要让你的代码适应未来的发展。不要尝试写出在系统发展时可以处理每一种可能事件的代码。这样做会使你的代码难以理解和维护，并且附加了许多不必要的复杂度。最好的情况是，无论如何你都不可能直到结束一直使用你所添加的东西。请善待那些在将来几年会维护和支持你的代码的技术员。不要预计在你的代码投入产品后会发什么。仅仅实现需要的东西。

注5： Wrangler系统可以帮助你找出重复的代码，并把它构建到一个共同函数中。

这里还有两个最后的建议，它们并不直接与Erlang相关，而是面向一般的编程：避免剪切—复制编程，并且不要把不再使用的代码注释掉，而是应该直接删除它们，因为当你将来需要它们的时候，你总是可以从你的资料库里面把它们找回来（注6）。

效率

Erlang的虚拟机是在不断优化和改进的。在先前发布的版本中是低效的结构和必要的变通方案，但在当前版本中也许不再是一个问题。因此，当你在旧的性能指南、日志记录，特别是在新闻组和邮件列表档案中的旧邮件里，读到效率、替代方法和优化时要特别小心。如果有疑问，请你参考发布说明、发布的笔记以及正在使用的运行时系统的文档。因此，最重要的是基准测试（benchmark）、压力测试和相应的对系统的分析。

顺序编程

一个关于效率最常见的错误概念与fun和列表内涵有关。列表内涵允许产生列表和过滤元素，而fun允许你把一个函数参数绑定到一个变量上。如今，编译器把列表内涵转换为普通的递归函数而Fun在很久以前就已经优化过了，它们已经摆脱了性能在常规函数调用与apply/3的使用之间的极度低效问题。

在Erlang中并不能有效率地实现字符串。在32位的表达式中，每一个字符由4个字节组成，另外还有4个额外的字节指向下一个字符。在64位的表达中，这些加倍为8个字节。好的一个方面是，采用统一编码（Unicode）的使用并不是一个问题。坏的一个方面是，如果在处理大型数据集合和内存确实成为一个问题时，你必须将字符串转换为二进制，并且使用位语法匹配它们。

如果速度和带宽很重要，那么可以使用re库模块来处理正则表达式。这个库支持Erlang中PCRE风格的正则表达式，它的实现比regexp库模块在效率上有实质性的提高，现在都不赞成使用者。

曾经有人鼓励你把最常见的模式放在最上方来对函数语句以及receive语句和case语句重新排序。现在，编译器会帮你重新排列语句，大多数情况下它会使用有效率的二进制查找直接跳到正确的字句，而不管语句的数量。例外情况是，你有下面这种形式的代码：

```
month('January') -> 1;
month('February') -> 2;
month(String) when is_list(String) -> {error, badmonth};
month('March') -> 3;
```

注6：当然，这需要假设你使用了版本控制。

```
...
month('December') -> 12;
month(_other) -> {error, badmonth}.
```

由于变量String总是匹配和被绑定，语句可能在保护元上失败，编译器需要单独处理这种情况。它会先尝试使用二分查找来匹配'January'或'February'，然后把参数绑定到变量String上并且计算保护元的值。如果保护元失败了，那将会在接下来的月份中执行一个新的二分查找。要么把保护元的语句转移到开头：

```
month(String) when is_list(String) -> {error, badmonth};
month('January') -> 1;
month('February') -> 2;
...
```

要么放在最后一个语句之后，month(_other) -> ...之前，这样做可以稍微提高一下效率。

如果你在设置或重新设置许多定时器，应该避免使用计时器模块，因为它通过一个进程把所有的请求序列化，这将导致产生瓶颈。相反，尝试使用Erlang的计时器内置函数erlang:send_after/3、erlang:start_timer/2、erlang:cancel_timer/1或erlang:read_timer/1来代替它。

如果可能，尽量使用元组而不是列表。元组的大小是两个字加上每个元素的大小。列表会为每个元素消耗一个字。因此，元组消耗更少的内存并且存取更快。

请记住，基元是不被垃圾收集的。如果你在动态数据上使用内置函数list_to_atom/1动态地生成基元，你也许最终会内存耗尽或达到允许的基元的极限（稍微多于一百万）。如果把外部数据转换为基元，这个内置函数会使你的系统对拒绝服务攻击（DoS攻击）敞开大门。其中有一种可能的方法是，你使用list_to_existing_atom/1来代替。

列表

经常问问你自己是否真的需要用扁平列表来工作，因为函数lists:flatten/1是一个代价很昂贵的操作。通过端口和套接字的I/O操作接收非扁平列表，包括那些由二进制块组成的列表，因此没有必要把列表扁平化。

这个道理同样适用于内置函数iolist_to_binary/1和list_to_binary/1。如果你的列表很深，那么请使用lists:append/1来代替：

```
1> Str = [$h,$e,[$l,$l],$o]].
[104,[101,"ll",111]]
2> io:format("~s~n",[Str]).
hello
```

```
ok
3> lists:append([[1,2,3],[4,5,6]]).
[1,2,3,4,5,6]
```

左关联的连接效率很低。使用下面的方式连接字符串：

```
lines(Str) ->
  "Hello " ++ Str ++ " World".
```

这会导致多次遍历++左边的字符串。不使用++，你可以通过下列方式让编译器帮你连接：

```
lines(Str) ->
  ["Hello ", Str, " World"].
```

不要以List ++ [Element]或lists:append(List, [Element])的形式使用++向列表的尾部添加元素。你每增加一个元素，都会遍历++左边的列表。做一次，你可能会得逞。递归地做，然后对每个递归循环都这么做，你将会面临严重的性能问题：

```
double([X|T], Buffer) ->
  double(T, Buffer ++ [X*2]);
double([], Buffer) ->
  Buffer.
```

把元素添加到列表头上会更有效率。当递归调用达到终止条件时，反转它。这样，你将只遍历这个列表两次：

```
double([X|T], Buffer) ->
  double(T, [X*2|Buffer]);
double([], Buffer) ->
  lists:reverse(Buffer).
```

如果你在处理接收的非扁平列表的函数，可以使用[List, Element]在非扁平列表尾部添加元素，创建一个格式为[[[[[1,2],3],4],5]的列表。这样可以节省你完成时反转列表的时间。

因此，在添加列表时，如果你知道这个列表不是由单个元素组成，并且结果必须是平的，你应该使用++。而不是下面的代码：

```
List1 ++ List2 ++ List3
```

比这样的代码更优美吗？

```
lists:append([List1, List2, List3])
```

尽管只遍历列表一次。对于小型列表，你也许觉察不出执行时间有任何的不同，但是一旦你的代码投入产品中，代码行的长度增加了，性能将成为一个显著问题。在下面的例

子中，我们采用一个整数列表，把所有的偶数提取出来，用倍数乘以它们，然后把它们全部加在一起：

```
even_multiple(List, Multiple) ->
  Even = lists:filter(fun(X) -> (X rem 2) == 0 end, List),
  Multiple = lists:map(fun(X) -> X * Multiple end, Even),
  lists:sum(Multiple).
```

我们在这个`even_multiple`的定义中遍历了3次列表。我们也可以这样做：封装过滤，在一个`fun`中添加整数，使用一个高阶函数遍历列表。这样可以更高效地执行相同的操作，只需遍历列表一次。

```
even_multiple(List, Multiple) ->
  Fun = fun(X, Sum) when (X rem 2) == 0 ->
    X + Sum;
    (X, Sum) ->
      Sum
  end,
  Multiple * lists:foldl(Fun, 0, List) .
```

尾递归和非尾递归

非尾递归函数常常比尾递归函数更优美，但是在处理大型数据集合时，要小心内存使用的大量爆发。这些爆发会不经意间导致你的Erlang运行时系统内存耗尽，并因此而终止。以非尾递归形式建立一个大的数据结构会有效率，但是其他在大型结构数据上的操作迫使使用深度递归调用时，使用尾递归可能更高效。这是因为它让最后调用的优化成为可能，而结果上，它允许函数在恒定数量的内存中执行。

最后，我们关于本小节的意见是：确保衡量了系统的性能，以便你明白系统内存的使用和行为。

并发

需要许多内存的操作会影响性能，因为它会经常触发垃圾收集器。在处理内存密集型操作时，生成一个单独的进程，一旦你完成了要做的事情就终止它。垃圾收集的时间会减少，同时它所有的内存区域都会马上得到释放。

你可以更进一步用下列语句生成一个进程：

```
spawn_opt(Module, Function, Args, OptionList)
```

其中的`OptionList`包含元组`{min_heap_size, Size}`。`Size`是一个整数，代表生成进程时分配的堆的大小（以字为单位）。默认分配的堆的大小是233个字，是为允许大量并发设置的一个保守的值。

在调整系统的性能时，增加堆的大小将减少垃圾收集的次数，可以潜在地加速某些操作。要小心使用`min_heap_size`，保证在使用之前和之后测试系统的性能，以确保你做的改变得到了预期的效果。如果你不小心，系统最终可能多使用了不必要的内存，并且由于更糟的数据局部性运行变得更慢。在启动Erlang运行时系统时，你可以通过对`erl`用`+h Size`标志来设置所有进程堆的大小。

你要对全部的垃圾收集进行协调。如果你记得第3章中的讨论，内存堆分为新堆和旧堆，新堆在垃圾收集器的清理中幸存下来的数据将被移动到旧堆中，选项`{fullsweep_after, Number}`可以指定，无论旧堆是否已满，在旧堆被清理前应该进行的垃圾收集的次数。

将值设置为0会强制每次都全部清理。在嵌入式系统中，内存是有限的，因此这个选项很有用。如果你使用大量短周期的数据，特别是大的二进制数，看看把值设置介于10~20是否有差别。只有在你知道进程存在内存消耗的时候才需要采用`fullsweep_after`选项，并确保使用它们时会有差别。你可以使用内置函数`erlang:system_flag(Flag, Value)`为所有新生成的进程设置全局的`fullsweep_after`和`min_heap_size`标志选项。

当一个进程在`receive`语句中被暂停等待一个事件时将不会触发垃圾收集，直到接收到一个消息和需要使用更多的内存时。它不管这个进程等了多久，也无视在堆中未使用的数据数量。要克服这点，你可以在调用进程中使用内置函数`garbage_collect/0`，或者在一个特殊进程中使用`garbage_collect/1`，强迫进行一次垃圾收集。使用`erlang:hibernate/3`会节省更多的内存，在`gen_server`、`gen_fsm`和`gen_event`中使用可以得到支持。

请使用二进制来编码大的消息。进程间的消息发送会引起消息从发送进程的栈复制到接收进程的堆。避免不必要的并发，可能的话，保持消息尽量的小。

如果你需要向许多进程发送一个大的消息，那么请将它转化为二进制。大于64位的二进制（引用计数的二进制）作为指针传送，避免了大量数据在进程间复制。如果你有很多收件人或者要转发没有改变的消息到许多进程，把数据类型转换为二进制，将比把发送进程栈中的数据复制到每一个接收进程的堆中的花费要少。当发送大量数据到端口和套接字上时，使用二进制也会更有效率。特别是，把你的输出转换为整数列表是没有必要的。

最后

让别人审查你的代码，因为他们会给你一些关于风格和优化的反馈和评论，总是试着写清晰并且易懂的代码。指定你的接口类型，并且模型化你的数据结构。要用心选择算法

和结构，以便能有效地扩展。你必须在一开始就考虑实时效率，因为它不是你稍后能轻松解决的问题。最后，永远不要在开发的第一步就优化你的代码，相反，编程时应牢记可维护性和可阅读性。当你完成了你的应用时，只在必要的时候剖析和优化你的代码。

初学者常见的错误有：

- 一个函数占用了很多页。
- 深度嵌套的if、case和receive语句。
- 拙劣的类型定义和没有标签的返回值。
- 选择不当的函数名和变量名。
- 不必要的或者多余的进程。
- 不恰当缩进或者没有缩进不缩排的代码。
- 对put和get的使用。
- 错误使用catch和throw。
- 拙劣的、多余的或缺失的注释。
- 使用元组表示记录。
- 拙劣的和充分的模式匹配使用。
- 试图用不必要的优化来让程序变得运行快速。

一个程序员只有在把一个问题解决了至少一次之后，才能完全理解它。当他找到一个解决办法时，他会想出比原来更好的办法。常回过头查看，并且重写你的代码，记住前面说过的初学者的错误。你会很快发现你的重组程序由更优美、更有效的代码基础组成，因此它变得容易测试、调试和维护（注7）。在重写你的第一个主要的Erlang程序时，可以期望达到50%代码缩减。随着你经验越来越丰富，并且发展了你的程序风格，这种缩减会减少，因为你在第一轮的时候就开始优化代码了。

在用Erlang工作时，黄金条例总是如下这些：

首先让它可以工作；

然后让它变得优美；

最后，只有在你不得不这样时，保证代码快速，同时保持代码优美。

你会很快发现，在大多数情况下，你的代码已经足够快了。祝你Erlang编程愉快！

注7：如果你在编写工业级应用程序，那么你绝对不要相信你是最后一个碰你代码的人。请善待那些在你之后接管你的代码的人！

使用Erlang

首先，本附录将帮助你开始使用Erlang。其次，我们推荐一些工具帮助你更有效率地开发以Erlang为基础的系统。最后，我们告诉你在哪里可以得到更多关于Erlang的资源，特别是许多网络资源。

Erlang入门

本节将告诉你如何开始使用Erlang：首先，你如何安装它；其次如何运行Erlang程序；最后，向你展示Erlang终端的不同命令，从而帮助你成为一个终端高手，包括使用历史记录机制和行编辑命令。

安装系统

Erlang发行包可以从Erlang网站：<http://erlang.org/download.html>，或者其中的一个镜像站点上获取。你也可以使用软件BitTorrent下载Erlang，而且还会发现它已经绑定到主要的Linux发行版本中。在Unix操作系统中，包括Linux和Mac OS X，都提供可以编译的源代码；在Windows系统中需要使用二进制安装程序。当从源代码构建时，按照发行包的指示进行即可。

运行Erlang终端

在Unix、Linux和Mac OS X操作系统中可以通过输入`erl`命令来从命令行运行Erlang终端，并设置任何你需要的选项。

在Windows系统中打开Erlang文件，需要双击文件图标，这将确保该系统是在正确的目录下打开。在Windows中，这个系统有两个选择，当你右击一个Erlang源文件时，它们在“打开方式”的菜单中提供了：

Erl

在命令提示程序中的Erlang终端里打开文件。

Werl

在一个比标准命令提示程序更易于支持复制和粘贴操作的窗口中打开一个Erlang终端。

要运行带有选项集的Erlang —— 例如，选项-smp要求运行wxErlang —— 你可以在“运行窗口”里运行这些命令，或者在命令提示符中作为命令输入。此选项允许你在发出命令之前更换到适当的目录，如下所示：

```
C:\Documents and Settings\Administrator>cd Desktop\programming\wxex
C:\...\wxex>"c:\Program Files\erl5.7\bin\erl.exe" -smp miniblog.erl
Eshell V5.7 (abort with ^G)
1> miniblog:start().
```

通过右键单击该文件图标和选择属性，你也可以改变这些特性。在快捷方式选项卡上，你可以把你的目标改变为包括你的启动选项，也可以把“从目录开始”更改到指向你的源代码位置。

在Unix下的Erlang终端和在Windows下的Werl中，对于输入命令存在一组标准编辑操作：

向上和向下箭头

取出上一个和下一个命令行，这可能是一个命令的一部分，因为在一般情况下命令是可以跨行的。

Ctrl+P和Ctrl+N

具有与向上和向下箭头相同的效果。

向左和向右箭头

将光标向左边和右边移动一个字符。

Ctrl+B和Ctrl+F

具有与向左和向右箭头相同的效果。

Ctrl+A

把光标移动到该行开始。

Ctrl+E

把光标移动到该行末尾。

Ctrl+D

删除光标下的字符。

正如你在正文中已经看到的，在Erlang终端中存在一组命令。其中常用命令如下所示：

c(File)

编译并加载File，清除旧版本的代码。

b()

打印当前变量绑定。

f()

“忘记”所有的当前变量绑定。

f(X)

“忘记”对变量X的绑定。

命令的历史及其结果是被记录的：

h()

将打印历史列表（其默认长度是20，但可以改变）。

e(N)

将重复命令数N。

e(-N)

将重复前面第n个命令，例如，**e(-1)**就是前一个命令。

v(N)

第N个命令的返回值。

v(-N)

前面第n个命令的返回值，例如，**v(-1)+v(-2)**将返回前两个值的总和。

其他终端命令，包括那些在终端中处理记录定义的命令，其详细描述可以在介绍模块shell的文档中找到。

Erlang工具

如果你想开始使用新的语言编写程序，那么你最不想做的一件事就是也要学习一种新的编辑器。幸运的是，许多通用编辑器和集成开发环境（IDE）支持编写Erlang程序。在本节中，我们将讨论这些编辑器和IDE，以及其他一些有用的工具。有些工具是作为Erlang发行版本的一部分提供，在伴随发行包的文档里有对这些工具的全面描述。

编辑器

Erlang程序包含于文本文件中，因此它们可以在任何文本编辑器中创建。不过，大量的编辑器支持Erlang关联的操作，并仿效Java和C++社区，越来越多完全成熟的集成开发环境支持Erlang：

- 根据Erlang用户的最新调查（注1），专业Erlang程序员的主要开发工具是Erlang模式的Emacs，在Erlang的联机文档中已经有该工具的参考手册。它给Erlang进行语法着色，以及上下文关联格式化，这样可以帮助你展开程序，以便你和其他人都可以阅读它。该模式也将检查你的模块名称和文件名是否匹配，以及为共同的OPT行为提供框架。
- *Distel*或分布式Emacs Lisp，把Emacs对Erlang的支持提升到另外一个层次，因为它允许Emacs与运行着的Erlang节点进行互动，正如我们在第16章中为其他编程语言描述的那样。*Distel*为函数和模块名称提供了补全功能；在Emacs中运行Erlang代码；提供了一些有限的重构支持；存在一个有特色的交互式调试器。
- 对于Vim也有一个Erlang插件包，它支持缩进和语法高亮显示，以及折叠和部分全方位补全功能。这个邮件包可以从Vim的网站上获取：http://www.vim.org/scripts/script.php?script_id=1584。
- 对于很多Java和C++程序员而言，Eclipse是他们的首选工具，Eclipse现在是通过*Erlide*（为Erlang开发的Eclipse插件）来支持Erlang，它可以在Sourceforge.net上获得。*Erlide*实现了语法高亮显示和缩进，而且还提供了在集成开发环境中对Erlang表达式的计算，并在保存文件时自动进行编译。其结构还规定了Erlang项目的范围，并支持调试。*Erlide*还提供*Wrangler*（稍后讨论）进行重构。
- 其他的集成开发环境和编辑器的支持包括*ErlyBird*，一个基于NetBeans的Erlang集成开发环境和*UltraEdit*，它被有些Windows开发人员用来使Erlang程序语法高亮显示。

其他工具

除了前面介绍的工具之外，我们找到其他一些有用的工具，包括：

- 虽然EUnit提供了单元测试的框架，但是通用测试为完全以Erlang为基础的系统提供了一个框架。通用测试是Erlang标准发行版本的一部分。

注1：可以在<http://www.protest-project.eu/publications/survey.pdf>上获取，这曾经作为项目ProTest的一部分。

- 传统测试允许你在特定的输入下检查系统的行为。有一种`QuickCheck`包含的替代办法，要求测试人员描述系统特性，然后针对这些随机生成的输入值测试属性。利用有限状态机来训练它们的行为，`QuickCheck`还能够测试并行系统的特性，并把不能满足特性的反例数据缩减到最小。`QuickCheck`是Quviq AB的一个产品。
- 除了简单的测试之外，静态分析可以检查死代码以及类型的异常情况。`Dialyzer`作为Erlang标准发行版本的一部分提供。
- Erlang的重构是通过`Wrangler`工具提供支持的，它已嵌入到Emacs和Erlide中，并由英国肯特大学提供。`RefactorErl`也支持一些重构，它与作为Erlang标准发行版本一部分的`Distel`和`syntax_tools`模块一样。
- 虽然测试工具可以在一组选定的输入下检查系统的行为，但是模型检查使得用户能够检查被测试系统所有可能的行为。采用Erlang编写的Erlang模型检查器`McErlang`是在马德里的加尔达海纳理工大学研制的。模型检查的另一种方法把Erlang转化成 μ CRL，然后紧接进行模型化地检查结果。

了解更多

当你想了解更多有关Erlang的内容时，最好的起始点是Erlang主页：<http://www.erlang.org>。在那里你可以了解即将举行的活动、书籍、课程和工作机会，以及访问系统文档。

系统文档——你可以访问：<http://www.erlang.org/doc/>，并在下载Erlang时下载到你的计算机——开始时可能使人不知所措，但是它包含了很多有用的信息：

- 在发布版本中每个Erlang模块的文档是由<http://www.erlang.org/doc/>提供的，可以单击主页左上角的模块链接来访问。
- 要获得关于某个主题或潜在功能的其他信息，你可以使用由文档生成的索引，它可以从主页的左上角访问到。

至于Erlang主要的应用程序和工具，在主页左边下方的标题页给出了它们的链接。其中特别有用的是：

- 安装指南（在Erlang/OTP标签下）。
- 迷你入门教程（在Erlang编程下）。
- Erlang参考手册（在Erlang编程下）。
- 常见问题回答，在<http://www.erlang.org/faq.html>里面。

在下面这些地方你可以找到Erlang的其他信息：

- Erlang社区网站Trapexit.org, 网址: <http://www.trapexit.org>。
- 本书网站<http://www.erlangprogramming.org>, 其中包含所有本书所谈及的网站链接, 以及更多的背景资料。
- Erlang邮件列单, 可从<http://erlang.org/faq.html>得到, 在Nabble.com和其他地方可以得到其归档的格式。
- Erlang博客。只要在搜索引擎中搜索“Erlang”, 就会发现网站<http://planet.trapexit.org>和Erlang星球网站<http://www.planeterlang.org>聚集了很多博客。
- 两个每年度一次的Erlang重要事件: 一个是Erlang研讨会, 是由ACM SIGPLAN赞助并和Erlang研讨会与函数编程国际会议在同一地点召开 (<http://www.erlang.org/workshop/>), 另一个是Erlang用户大会, 它每年在斯德哥尔摩举行 (<http://www.erlang.org/euc/>)。
- Erlang工厂, 主要运作Erlang商业会议, 其网站是: <http://www.erlang-factory.com/>, 网页上有许多在会议上所做报告的幻灯片和视频。



作者简介

Francesco Cesarini是Erlang语言培训和咨询公司创始人 (<http://www.erlang-consulting.com>)。从1995年开始,他差不多每天都在使用Erlang语言,而他第一次接触Erlang是当他在Erlang的诞生地——爱立信计算机科学实验室组实习的时候。他在爱立信参与了Erlang的旗舰项目并持续了4年,其中包括了OTP中间件R1的发布版本。他热衷于对参与软件开发周期各个环节的各方人员包括开发工程师、支持工程师、测试人员、项目经理和技术经理传授关于Erlang/OTP方面的相关知识。2003年,他开始在哥德堡大学的计算机系教授本科生。

在Erlang开源发布不久,他就成立了Erlang培训和咨询公司。公司在英国、瑞典和波兰(很快在美国)都设有办事处,并已成为有关Erlang咨询、承包合同、支持、培训和系统开发方面的世界领先者。Francesco在Erlang社区非常活跃,他不仅仅出现在Erlang的各种定期讲座、研讨会和多次在全球会议上演讲,而且还参与国际性的一些研究项目。他组织Erlang的地方用户组,并在同事的帮助下,运行维护trapexit.org这个Erlang社区网站。

Simon Thompson是一位在肯特大学计算实验室工作的逻辑和计算学教授,他教授本科生和研究生的计算学课程已经有25年了,而且6年前他成为了计算机系的部门负责人。他的研究工作主要集中在函数式编程:程序验证、类型系统以及最近关于函数编程语言方面的软件开发工具的开发。他的团队已经开发出了关于Haskell语言的重构工具HaRe,现在他们正在开发类似的关于Erlang语言的重构工具Wrangler。

很多机构(包括自然科学基金委员会和欧洲框架计划组织)都资助Simon的研究工作。他拥有剑桥大学的数学硕士学位和牛津大学的数理逻辑博士学位。他在感兴趣的领域著有3本书:《Type Theory and Functional Programming》、《Miranda: The Craft of Functional Programming》和《Haskell: The Craft of Functional Programming》(Second Edition),这3本书已由Addison - Wesley出版社出版。

译者简介

杨剑 (Jacky Yang) 2004年硕士毕业于华中科技大学计算机系, 现就职于爱立信上海公司, 任资深工程师。致力于接入网、无线网、VoIP终端和核心网IMS相关产品的研发。
邮箱: yangjian_pro@sohu.com。

慕尼黑 Isar 工作组由以下几位组成:

骆古道 (Gudao Luo) 网名Cnruby, 20世纪80年代初毕业于西北工业大学数理力学系, 1988年公派留学德国, 从事组合最优化理论研究, 从20世纪90年代初期起一直致力于计算机领域软件开发、设计和管理等方面工作, 个人博客为“道喜技术日记”和“天天红玉世界”。

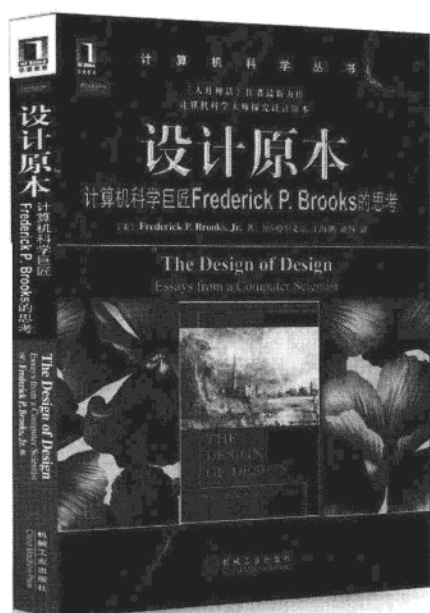
张军 (Jun Zhang) 2000年留学德国, 专注于计算机人工智能研究。不来梅大学毕业后他从事和Java有关的软件开发工作, 兴趣爱好及关注方面包括云计算、物联网、Web 2.0及手机软件。

刘姗姗 (Shanshan Liu) 2008年毕业于德国科隆莱茵技术大学工程管理专业。现从事于企业客户项目管理工作, 对软件开发有浓厚兴趣, 专注于Scrum项目管理、GUI及Web 2.0。

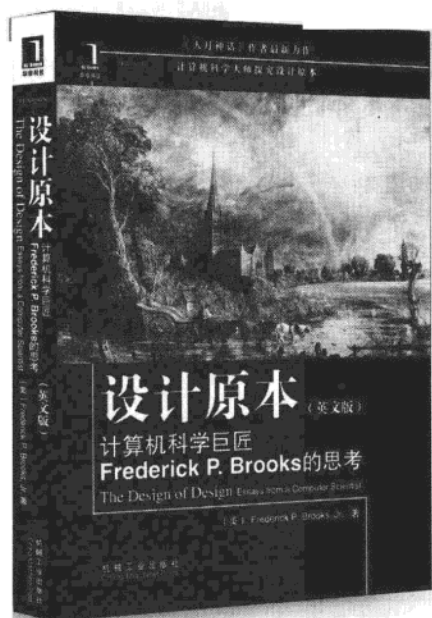
姜凯 (Jiang Kai) 2006年留学德国, 慕尼黑工业大学电子系在读, 方向为微处理系统及数据和图像处理。

程帆 (Fan Cheng) 于2007年同济大学计算机科学与技术系本科毕业后, 留学德国攻读硕士学位, 现在就读于慕尼黑工业大学计算科学与工程专业。专业研究主要集中在机器人方向, 特别是人工智能以及自动控制方面。





ISBN: 978-7-111-32557-4
定价: 55.00

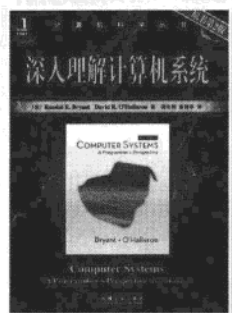


ISBN: 978-7-111-32503-1
定价: 69.00

《人月神话》作者最新力作 计算机科学大师探究设计原本

本书包含了多个行业设计者的特别领悟。Frederick P. Brooks, Jr. 精确发现了所有设计项目中内在的不变因素，揭示了进行优秀设计的过程和模式。通过与几十位优秀设计者的对话，以及他自己在几个设计领域的经验，作者指出，大胆的设计决定会产生更好的结果。

本书几乎涵盖所有有关设计的议题：从设计哲学谈到设计实践，从设计过程到设计灵感，既强调了设计思想的重要性，又对沟通中的种种细节都做了细致入微的描述，并且谈到了因地制宜做出妥协的具体准则。



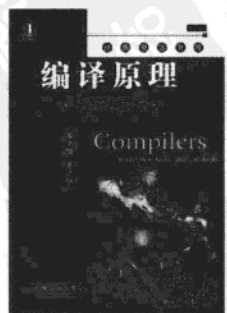
深入理解计算机系统（原书第2版）
ISBN: 978-7-111-32133-0
定价: 99.00



深入理解计算机系统（英文版·第2版）
ISBN: 978-7-111-32631-1
定价: 128.00



编译原理（第2版）
ISBN: 978-7-111-25121-7
定价: 89.00



编译原理（英文版·第2版）
ISBN: 978-7-111-32674-8
定价: 78.00



专业成就人生
立体服务大众

www.hzbook.com

填写读者调查表 加入华章书友会 获赠精彩技术书 参与活动和抽奖

尊敬的读者：

感谢您选择华章图书。为了聆听您的意见，以便我们能够为您提供更优秀的图书产品，敬请您抽出宝贵的时间填写本表，并按底部的地址邮寄给我们（您也可通过www.hzbook.com填写本表）。您将加入我们的“华章书友会”，及时获得新书资讯，免费参加书友会活动。我们将定期选出若干名热心读者，免费赠送我们出版的图书。请一定填写书名书号并留全您的联系信息，以便我们联络您，谢谢！

书名：

书号：7-111-()

| | | | |
|-------|---|---------|-----|
| 姓名： | 性别： <input type="checkbox"/> 男 <input type="checkbox"/> 女 | 年龄： | 职业： |
| 通信地址： | | E-mail： | |
| 电话： | 手机： | 邮编： | |

1. 您是如何获知本书的：

☐ 朋友推荐 ☐ 书店 ☐ 图书目录 ☐ 杂志、报纸、网络等 ☐ 其他

2. 您从哪里购买本书：

☐ 新华书店 ☐ 计算机专业书店 ☐ 网上书店 ☐ 其他

3. 您对本书的评价是：

| | | | | |
|------|-----------------------------|-----------------------------|-----------------------------|----------------------------------|
| 技术内容 | <input type="checkbox"/> 很好 | <input type="checkbox"/> 一般 | <input type="checkbox"/> 较差 | <input type="checkbox"/> 理由_____ |
| 文字质量 | <input type="checkbox"/> 很好 | <input type="checkbox"/> 一般 | <input type="checkbox"/> 较差 | <input type="checkbox"/> 理由_____ |
| 版式封面 | <input type="checkbox"/> 很好 | <input type="checkbox"/> 一般 | <input type="checkbox"/> 较差 | <input type="checkbox"/> 理由_____ |
| 印装质量 | <input type="checkbox"/> 很好 | <input type="checkbox"/> 一般 | <input type="checkbox"/> 较差 | <input type="checkbox"/> 理由_____ |
| 图书定价 | <input type="checkbox"/> 太高 | <input type="checkbox"/> 合适 | <input type="checkbox"/> 较低 | <input type="checkbox"/> 理由_____ |

4. 您希望我们的图书在哪些方面进行改进？

5. 您最希望我们出版哪方面的图书？如果有英文版请写出书名。

6. 您有没有写作或翻译技术图书的想法？

☐ 是，我的计划是_____ ☐ 否

7. 您希望获取图书信息的形式：

☐ 邮件 ☐ 信函 ☐ 短信 ☐ 其他_____

请寄：北京市西城区百万庄南街1号 机械工业出版社 华章公司 计算机图书策划部收

邮编：100037 电话：(010) 88379512 传真：(010) 68311602 E-mail: hzjsj@hzbook.com