no starch press

·新·锐·编·程·语·言·集·萃·

Erlang趣学指南

Learn You Some Erlang for Great Good!

[加] Fred Hébert 著　　邓辉 孙鸣 译

中国工信出版集团　人民邮电出版社 POSTS & TELECOM PRESS

no starch press

·新·锐·编·程·语·言·集·萃·

Erlang趣学指南

Learn You Some Erlang for Great Good!

[加] Fred Hébert 著　　邓辉 孙鸣 译

人民邮电出版社
北京

图书在版编目（CIP）数据

Erlang趣学指南 /（加）弗莱德·赫伯特著 ；邓辉，孙鸣译. -- 北京 ：人民邮电出版社，2016.9
（新锐编程语言集萃）
书名原文：Learn You Some Erlang for Great Good!
ISBN 978-7-115-43190-5

Ⅰ. ①E… Ⅱ. ①弗… ②邓… ③孙… Ⅲ. ①程序语言－程序设计－指南 Ⅳ. ①TP312-62

中国版本图书馆CIP数据核字(2016)第200850号

内容提要

这是一本讲解 Erlang 编程语言的入门指南，内容通俗易懂，插图生动幽默，示例短小清晰，结构安排合理。书中从 Erlang 的基础知识讲起，融汇所有的基本概念和语法。内容涉及模块、函数、类型、递归、错误和异常、常用数据结构、并行编程、多处理、OTP、事件处理，以及所有 Erlang 的重要特性和强大功能。

本书适合对 Erlang 编程语言感兴趣的开发人员阅读。

◆ 著　[加] Fred Hébert
译　邓 辉 孙 鸣
责任编辑　杨海玲
责任印制　焦志炜

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路 11 号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京艺辉印刷有限公司印刷

◆ 开本：800×1000　1/16
印张：31
字数：693 千字　2016 年 9 月第 1 版
印数：1－3 000 册　2016 年 9 月北京第 1 次印刷

著作权合同登记号　图字：01-2012-7094 号

定价：79.00 元

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

版权声明

对本书的赞誉

作者 Fred Hébert 作为活跃在一线的有丰富实战经验的工程师，不仅把入门教程写得清晰易懂、深入浅出，更难能可贵的是从 Erlang 应用程序的完整生命周期角度把涉及设计、开发、测试、部署、调优的关键特性表现得淋漓尽致。跟着他的节奏，读者会很容易登堂入室，此外，书中配的插图幽默、诙谐、有爱，也为本书增色不少！

——余锋，阿里云研究员负责阿里云数据库

译者序

在我近 20 年的软件开发工作中，除了 Erlang，还使用过许多其他编程语言。有工作需要的 C/C++、Java，也有作为业余爱好使用的 Lisp、Haskell、Scala 等，其中我最喜欢的当属 Erlang。除了因为我的电信软件开发背景外，还有一个很重要的原因是 Erlang 独特的设计哲学和解决问题方式。

大家听说 Erlang，往往是因为其对高并发的良好支持。其实，Erlang 的核心特征是容错，从某种程度上讲，并发只是容错这个约束下的一个副产品。容错是 Erlang 语言的 DNA，也是和其他所有编程语言的本质区别所在。

我们知道，软件开发中最重要的一部分内容就是对错误的处理。所有其他的编程语言都把重点放在“防”上，通过强大的静态类型、静态分析工具以及大量的测试，希望在软件部署到生产环境前发现所有的错误。而 Erlang 的重点则在于“容”，允许软件在运行时发生错误，但是它提供了用于处理错误的机制和工具。如果把软件系统类比为人体，那么其他编程语言只关注于环境卫生，防止生病；而 Erlang 则提供了免疫系统，允许病毒入侵，通过和病毒的对抗，增强免疫系统，提高生存能力。

这个差别给软件开发带来的影响是根本性的。大家知道，对于大型系统的开发、维护来说，最怕的就是无法控制改动的影响。我们希望每次改动最好只影响一个地方，我们通过良好的模块化设计和抽象来做到这一点。但是如果这个更改不幸逃过了静态检查和测试，在运行时出了问题，那么即使这个改动在静态层面确实是局部的，也照样会造成整个系统的崩溃。而在 Erlang 中，不仅能做到静态层面的变化隔离，还可以做到运行时的错误隔离[①]，让运行时的错误局部化，从而大大降低软件发布、部署的风险。另外，分布式系统中错误出现的必然性更加凸显了 Erlang 容错哲学的价值。

Erlang 语言不仅内置了容错支持工具——完全隔离的进程、链接（link）和监控器（monitor），还提供了一套完整的系统级容错语义模型——监督树。基于这个语义模型，可以清晰、准确、声明性地表达出系统中数据状态的关键程度、系统部件之间的错误相关性和依赖关系、系统各部分之间的承诺保证等，整个系统的容错处理都固化到这个显式、一致、标准的程序结构中。用户只需编写处理正常情况的代码，这个程序结构会自动处理出现的错误。

基于这种独特的容错哲学和支持工具，用户就会以拥抱崩溃、拥抱失败、拥抱异常的态度构建自己的系统，这些原本令人恐惧的东西现在以一种受控的方式存在于系统中，它们不再是让人

① 借助操作系统的进程也可以做到运行时的错误隔离，不过粒度太大，也过于重量。

讨厌的破坏者，而是被转化为了一种简单、强大的工具，用来构建更大、更可靠的系统。

强大的并发支持也是 Erlang 的特色之一，在这一点上常常被其他语言争相模仿。不过，Erlang 和模仿者之间有个根本的不同点——公平调度。为了做到公平调度，Erlang 可谓“不择手段”，并做到“令人发指”的地步[①]。为什么要费劲做这些工作呢？对于一个高并发系统来讲，软实时、低延时、可响应性往往是渴求的目标，同时也是一项困难的工作。尤其是，在系统过载时，多么希望能具有一致的、可预测的服务降级能力。而公平调度则是达成这些目标的最佳手段，Erlang 也是目前唯一在并发上做到公平调度的语言。

由于 Erlang 在容错和并发的公平调度方面的独特性，可以说，这些年来 Erlang 一直被模仿，但是从未被超越。

从某种意义上讲，Erlang 不只是一门编程语言，更是一个系统平台。它不仅提供了开发阶段需要的支持，还提供了其他语言所没有的运行阶段的强大支持。其实，在静态检查和测试阶段发现的问题往往都是些“不那么有趣”的问题，那些逃逸出来的 bug 才是真正难对付的。特别是对于涉及并发和分布式的 bug，往往难以通过静态检查和测试发现，并且传统的调试手段也无法奏效[②]。而 Erlang 则提供了强大的运行时问题诊断、调试、解决手段。使用 Erlang 的 remote shell、tracing、自省机制以及强大的并发和容错支持，我们可以在系统工作时深入系统内部，进行问题诊断、跟踪和修正，甚至在需要时在线对其进行“高侵入性”的外科手术。一旦用户用这种方法解决过一个困难的问题，就再也离不开它了。如果要在静态类型和这项能力间进行选择，我会毫不犹豫地选择后者[③]。

对于 Erlang 存在的问题[④]，提得最多的有两个：一个是缺乏静态类型支持，另一个是性能问题。Erlang 是动态类型语言，往往会被认为不适合构架大型的系统。我自己也非常喜欢静态类型。一个强大的静态类型系统不但能够大大提升代码的可读性，而且提供了一个在逻辑层面进行思考、设计的强大框架。另外，还可以让编译器、IDE 等获取更深入的代码结构和语义信息，从而提供更高级的静态分析支持。

不过，在构建大型系统方面，我有些不同看法[⑤]。如果说互联网是目前最庞大的系统，相信没有人会反对，那么这个如此庞大的系统能构建起来的原因是什么呢？显然不是因为静态类型，根本原因在于系统的组织和交互方式。互联网中的每个部件都是彼此间隔离的实体，通过定义良好的协议相互通信，一个部件的失效不会导致其他部件出现问题。这种方式和 Erlang 的设计哲学是同构的。每个 Erlang 系统都是一个小型的互联网系统，每个进程对应一台主机，进程间的消息对应协议，一个进程的崩溃不会影响其他进程……Erlang 所推崇的设计哲学是面向崩溃

① 为了能够做到跨 OS 的高效调度，Erlang 放弃了基于时间片，采用了基于 reduction 的方式。几乎在系统的每个地方都会进行 reduction 计数，达到公平调度的目的。

② 例如，一个和竞争有关的 bug，一旦加上断点，竞争可能就不会出现了。

③ 其实可以兼得，Erlang 现在已经支持类型定义、标注和推导。

④ 我们不讨论那些主观性太强的问题，例如，有人觉得 Erlang 语法怪异。

⑤ 这些看法只针对 Erlang。对于其他的动态类型语言，在程序规模变大后，确实有难以理解和维护的问题。在 Erlang 中，由于其 let it crash 哲学，很多动态类型语言的问题可以在很大程度上被避免。著名的 AXD301 就是用 Erlang 开发的，规模达百万行代码，历时 3 年。而其前身 AXE-N 是用具有静态类型支持的 C++语言开发的，开发 7 年后失败了。这个例子充分说明，对于大型、复杂的系统来说，语言的语义模型是成败的关键。

（crash-oriented）以及面向协议（protocol-oriented）是架构大型系统的最佳方式[①]。

当然现在，鱼和熊掌可以兼得，Erlang 已经支持丰富的静态类型定义和标注功能[②]，并且可以通过 Dialyzer 工具进行一定程度的类型推导和静态检查。

再来说说性能问题。在计算密集型领域，Erlang 确实性能不高[③]。因此，如果要编写的是需要大量计算的工具程序，那么 Erlang 是不适合的。不过，如果说涉及计算的部分只是系统中的一个局部模块，而亟需解决的是一些更困难的系统层面的设计问题——并发、分布式、伸缩、容错、短响应时间、在线升级以及调试运维等，那么 Erlang 则是最佳选择。此时，可以用 Erlang 作为工具来解决这些系统层面的难题，局部的计算热点可以用其他语言（如 C 语言）甚至硬件来完成。Erlang 提供了多种和其他语言以及硬件集成的方法，非常方便，可以根据自己的需要（安全性、性能）进行选择。

我前段时间曾经开发过一款 webRTC 实时媒体网关[④]，就是采用了 Erlang + C 的方案。其中涉及媒体处理的部分全部用 C 语言编写，通过 NIF 和 Erlang 交互，系统层面的难题则都交给 Erlang 完成。系统上线几个月，用户量就达到数百万。其间，系统运行稳定，扩容方便，处理性能也不错（尤其是高负载时的服务降级情况令人满意）。不是说使用其他语言无法做到，不过要付出的努力何止 10 倍[⑤]！

前面做了这么多的铺垫，主要是为了激起读者对 Erlang 的兴趣。有了兴趣之后，下面当然就是要选择一本好的介绍 Erlang 知识的书籍进行深入、系统的学习。而读者手上拿着的这本，就是一本广受好评的关于 Erlang 的图书。这本书甚至力压 Erlang 之父 Joe Armstrong 的《Erlang 程序设计》，被公认为是学习 Erlang 的一本佳作。这不是没有原因的。

首先，本书对 Erlang 和 OTP 平台进行了非常全面、详细的介绍，不仅讲解了语言的语法、常见的数据结构、基本的函数式编程和并发编程、套接字编程等内容，还深入讲解了 OTP 中的每个关键组件以及整个系统的发布方法。不仅如此，对于真实系统开发中会用到的关键知识内容，分别独立成章进行介绍，包括 EUnit、ETS、Common Test、Mnesia、类型和 Dialyzer 等，甚至还用了一个专门的章节介绍分布式系统设计的核心困难所在和应对策略。

其次，对于一些关键主题，尤其是那些复杂的主题，作者并没有蜻蜓点水、一带而过，而是深入原理，辅以实例，进行了深入浅出的讲解。作者的讲解风格轻松、幽默，让读者在不知不觉中就理解了原本不那么容易理解的内容。讲解的过程中有大量的编程实例，这些例子都是以循序渐进的方式编写的。和很多其他书籍不同的是，这本书中的示例代码质量很高，有些甚至达到了产品级质量。

最后，也是最重要的一点。虽然本书中传递的知识点很多，但是并非只是讲解这些知识点是什么，而是把它们放到了具体的领域背景和时代背景中，让读者理解问题是什么，做出这些决策

① 目前火热的 micro-service 架构在某种程度上和 Erlang 的哲学类似。在 Erlang 中，微服务只是一个语言特性。
② 无论如何，给程序加上类型标注都是一项好的实践。
③ 这方面的性能大概是 C 语言的 1/7。
④ 主要作用是完成浏览器的 webRTC 媒体流和 IMS 网络媒体流之间的互通，需要大量转码和控制。
⑤ 这个是我自己对比的数字。我曾经用 C++语言开发过类似的系统。

的原因是什么，有什么局限性。有了这些背景内容，读者可以更深刻地理解这些知识，在应用这些知识时，可以做出更准确的判断和权衡。更为难得的是，作者把自己在真实产品开发中积累的真知灼见、遇到的语言“坑”和对策也都呈现在这本书中。从某种程度上讲，本书其实是一本关于并发、容错、分布式系统设计的书，只是碰巧使用了这个领域中的 DSL（也就是 Erlang 语言）进行设计的表达。

无论你是初学者还是 Erlang 老手，尤其是你想在产品系统开发中使用 Erlang 时，我都强烈推荐你阅读本书。你一定不会失望的！

最后，如果发现译文中有任何问题，欢迎来信指正（dhui@263.net）。祝大家阅读愉快！

邓辉

2016 年 8 月于上海

序

学习编程很有趣，至少应该是一件有趣的事情。如果没有趣的话，读者就不会喜欢上它。在我的程序员职业生涯中，我曾经自学了几种不同的编程语言，在这个过程中，有时也会觉得不是那么有趣。在学习一门编程语言的过程中是否觉得有趣，很大程度上取决于对这门语言的介绍方式。

开始学习一门新的编程语言时，表面上看起来，好像就是学习这门新语言本身。但是，深入思考后会发现，所要学习的其实是意义更加深远的东西——一种新的思考问题的方法。而令人兴奋的正是这种新的思考方法，不是语言中那些微小的标点符号细节，也不是这门语言和你最喜欢的编程语言在外观上的不同。

在编程领域，函数式编程一直背负着“难学”的名声（并发编程更是如此），因此，编写一本关于 Erlang 语言的书，并在其中同时介绍函数式编程和并发编程，想想都令人望而却步。毫无疑问，介绍函数式编程不是一件容易的事情，介绍并发编程也很困难。除非具有非常特殊的才能，否则根本无法以轻松、幽默的方式同时介绍这两方面的内容。

Fred Hebert 向我们展示了他的这种特殊才能。他总能把复杂的概念以简单的方式介绍给大家。

在学习 Erlang 时有一个最大的障碍，这个障碍并不在于 Erlang 中的概念本身非常难以理解，而在于这些概念和在其他大多数语言中遇到的概念非常不同。为了学习 Erlang，得先暂时忘掉在其他语言中学到的东西。Erlang 中的变量不能改变。不要进行防御性编程。进程非常、非常的轻量，如果愿意的话，可以创建上千甚至上百万个进程。哦，还有就是 Erlang 的语法比较奇怪。Erlang 和 Java 不同，它没有方法或者类，也没有对象……甚至等号的含义也不是“等于”——它的意思是“匹配这个模式”。

Fred 完全无惧这些问题，他在处理这些内容时采用了一种巧妙的冷幽默方式，并且在教授这些复杂的主题时，他所采用的方法完全让我们感受不到复杂性的存在。

这是到目前为止第 4 本主要的 Erlang 书籍，并作为一部重量级的作品加入 Erlang 丛书之列。但是，这本书不仅仅是关于 Erlang 的。Fred 在这本书中介绍的许多概念同样适用于 Haskell、OCaml 以及 F#语言。

我希望大家能够像我一样喜欢 Fred 的这本书。学习 Erlang 的过程是一个令人愉悦、发人深省的过程，我也希望大家在学习的过程中能体会到这一点。如果在阅读本书的过程中，把书中的程序输入电脑并运行，那么你会学到更多的东西。编写程序要比阅读程序困难得多，第一步要做的就是让你的手指习惯于程序的录入，并尝试着去修正那些不可避免的语法小错误。随着阅读的

深入，你会编写出对其他编程语言来说非常困难的程序——不过还好，你不会感觉到这一点。很快，你就可以编写分布式程序了。有趣的旅程就此开始了……

Fred，谢谢你写了一本好书。

Joe Armstrong

2012 年 11 月 6 日于瑞典斯德哥尔摩

前言

本书的写作最初是在网站上开始的，现在仍然能够访问 http://learnyousomeerlang.com/（感谢 No Starch 出版社在出版和技术素材方面的完全开放性）。从 2009 年公开本书第 1 章的内容开始，本书慢慢地从一份只有 3 章内容的微型教程（当时在 erlang-questions 邮件列表中请求大家对内容进行校正）成长为一份学习 Erlang 的官方推荐文档、一本书、一个我生命中的重大成就。对于这本书带给我的一切，从朋友到工作，再到 2012 年度 Erlang User 的名号，我既困惑又感恩。

致不了解 Erlang 的读者

当作为一个局外人从远处审视 Erlang 程序员时，会发现他们看起来像一个古怪的小团体，他们所信奉的原则几乎没有人愿意遵从。这些原则看起来不切实际、在应用方式上有诸多限制。更糟糕的是，Erlang 社区的成员看起来很像某个宗教派别的成员，完全认为他们掌握了软件世界中心的唯一真理。这种“唯一真理”和 Lisp 派、Haskeller、形式化证明派中的自命不凡者、Smalltalk 程序员、Forth 中的栈派等语言狂热派以前所鼓吹的完全一样。没什么特别的，还是那老一套。它们都承诺一定会成功，只是方式不同，但是程序员所编写的程序仍然错误百出，仍然代价高昂，仍然难以维护。

对 Erlang 来说，可能是它对并发或者并行的承诺吸引读者阅读本书的。也许吸引读者的是这门语言对于分布式计算的支持，或者是其不寻常的容错机制。当然，带着怀疑来学习 Erlang 是一件好事。它不能解决你的所有问题——毕竟，这是你的责任。Erlang 只是一个很好的工具箱，帮你解决问题。

致 Erlang 熟手

你可能已经了解 Erlang，或许还很深入。如果是这样的话，我希望本书阅读起来比较有趣，能够成为一本参考手册，或者其中的某些章可以让你更深入地了解 Erlang 语言的某些部分和工作环境，这些内容你以前可能不太熟悉。

也许，你对 Erlang 的了解在各个方面都比我好。那么，我希望本书能够拿来压压东西或者填充你的图书馆空间。

致已经阅读过在线内容的读者

谢谢你的支持，本书的内容经过了专业的编辑，书中 Erlang 的版本也提升到了 R15B+，希望你能喜欢。

致谢

感谢 Miran Lipovača 首先想到了 *Learn You a Language* 这个主意，并同意我在本书以及相关的网站中借用这个想法。

感谢 Jenn（我的女朋友）设计了最初的网站，为了能够让图片适合打印，她对本书中几乎全部图片进行了重新绘制，感谢她的辛苦工作。感谢她在我花费大量时间编写本书的过程中所给予的支持和耐心。

感谢所有花时间评审本书在线版本、寻找错误并提供帮助的人们（排名不分先后）：Michael Richter、OJ Reeves、Dave Pawson、Robert Virding、Richard O'Keefe、Ulf Wiger、Lukas Larsson、Dale Harvey、Richard Carlsson、Nick Fitzgerald、Brendon Hogger、Geoff Cant、Andrew Thompson、Bartosz Fabianowski、Richard Jones、Tuncer Ayaz、William King、Mahesh Paolini-Subramanya 以及 Malcolm Matalka。还有很多其他人也做了少量的评审工作，指出了一些拼写等方面的错误。

还要再次感谢 Geoff Cant，他是本书的正式技术评审。

感谢 No Starch 出版社的工作团队（Keith、Alison、Leigh、Riley、Jessica、Tyler 和 Bill）所做出的专业工作。

最后，感谢本书在线版本的所有读者，包括那些购买本书的读者和没有购买的读者。

引言

这是本书的开头部分。阅读本书应该是学习 Erlang 编程的第一步，所以，我们还是有必要说点什么。

我是在读了 Miran Lipovača 的 *Learn You a Haskell for Great Good!*（LYAH）教程后，才萌生了写这本书的想法。我觉得在引发读者兴趣和提供友好的学习体验方面，他做得非常棒。因为我早就认识他，我就去问他，如果我也按照这样的方式写一本关于 Erlang 的书，他会觉得怎样。他很喜欢这个想法，有部分原因应该是出于他对 Erlang 的兴趣吧。

于是我就开始写这本书。

当然，我之所以决定写这本书还有其他原因。当我开始学习 Erlang 时，我发现要找到这门语言的入门书很困难（Web 上的文档稀缺，书又比较贵），并且我认为如果存在类似 LYAH 这样的指南，对整个 Erlang 社区都是有好处的。此外，我还发现人们对于 Erlang 好坏的评价都非常的笼统。

本书的读者需要具备命令式语言（像 C/C++、Java、Python、Ruby 之类）编程的基础知识，但是并不要求读者具备函数式编程语言（像 Haskell、Scala、Clojure、OCaml 和 Erlang）方面的知识。我在编写本书时尽量做到客观、诚实，对 Erlang 实话实说，既肯定它的强项，也不避讳它的弱点。

Erlang 是什么

Erlang 是一门函数式编程语言。如果你曾经用过命令式语言，那么像 `i++`这样的语句对你来说再普通不过了，但是在函数式编程中，却不能这样使用。事实上，改变任何变量的值都是绝对不允许的。乍一听这似乎很奇怪，但是想想上过的数学课，你学到的内容是这样的：

```
y = 2
x = y + 3
x = 2 + 3
x = 5
```

如果我把以下内容加进去，你一定会觉得困惑。

```
x = 5 + 1
x = x
∴ 5 = 6
```

函数式编程认识到了这一点。如果我说 `x` 是 `5`，从逻辑上，我就不能说它也等于 `6`！这属于欺诈。这也是为什么每次用同样的参数去调用函数时，它都应该返回相同的值：

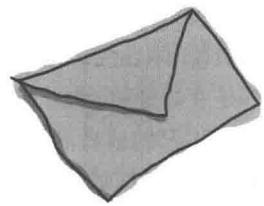
```
x = add_two_to(3) = 5
∴ x = 5
```

对于同样的参数，函数永远要返回同样的值，这个概念称为引用透明性（referential transparency）。正是因为这一点，才能够把 `add_two_to(3)` 替换成 5，因为 3+2 的结果就是 5。这意味着，为了解决更为复杂的问题，我们可以将很多函数粘合在一起，还能保证不破坏任何逻辑。既合乎逻辑，又整洁，不是吗？不过还有一个问题：

```
x = today() = 2013/10/22
    -- 等待一天后 --
x = today() = 2013/10/23
x = x
∴ 2013/10/22 = 2013/10/23
```

哦不！我美丽的等式！它们突然间全部出错了！为什么我的函数每天返回的值都不同呢？

显然，在某些情况下，不遵循引用透明性是有用的。Erlang 在函数式编程方面采用了一种非常注重实效的策略：遵守最纯粹的函数式编程原则（引用透明性、避免可变数据等），但是在遇到现实问题时，就打破这些原则。



Erlang 是一门函数式编程语言，它同时也非常重视并发和高可靠性。为了让几十个任务能同时执行，Erlang 采用了 actor 模型，每个 actor 都是虚拟机中的一个独立进程。简而言之，如果你是 Erlang 世界中的一个 actor，你将会是一个孤独的人，独自坐在一个没有窗户的黑屋子里，在你的邮箱旁等待着消息。当你收到一条消息时，会用特定的方式来响应这条消息：收到账单就要进行支付；收到生日卡，就回一封感谢信；对于不理解的消息，就完全忽略。

可以把 Erlang 的 actor 模型想象成这样一个世界，其中每个人都独自坐在屋子里，可以执行一些不同的任务。人和人之间只能通过写信进行交流，就是这样。这样的生活虽然听起来很乏味（但却是邮政服务的新时代），但这意味着，你可以要求很多人为你分担不同的任务，他们做错了事或者犯了错误绝对不会对其他人的工作造成任何影响。除了你之外，他们甚至不知道还有其他人的存在（这真是太棒了）。

事实上，在 Erlang 中，只能创建出相互之间完全没有共享、只能通过消息通信的 actor（进程）。每次消息交互都是显式的、可追踪的和安全的。

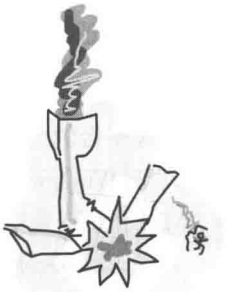
Erlang 不仅仅是一门语言，同时也是一个完整的开发环境。代码被编译成字节码，字节码运行在虚拟机中。所以，Erlang 很像 Java，也像患有多动症的孩子，在任何地方都能运行。下面是 Erlang 标准发布中的一些组件：

- 开发工具（编译器，调试器，性能分析器以及测试框架和可选的类型分析器）；
- 开放电信平台（OTP）框架；
- Web 服务器；
- 高级跟踪工具；
- Mnesia 数据库（一个键/值存储系统，能够在多台服务器上复制数据，支持嵌套事务，并

且可以存储任何类型的 Erlang 数据）。

Erlang 虚拟机和库还能让用户在不中断任何程序的情况下升级运行系统的代码，能轻易地将代码分布在多台计算机上，还能用一种简单但强大的方式去管理错误和故障。

在本书中，我们会对其中绝大部分工具的使用方法进行介绍，也会讲解如何实现安全的系统。



谈到安全性，你应该知道 Erlang 中与之相关的一个总方针——任其崩溃（let it crash），这说的可不是一架崩溃后会导致大量乘客死亡的飞机，它指的更像是在下方铺有安全网络的钢丝上的行走者。尽管应该避免犯错，但是也不用时刻去检查每一种可能的错误情况。

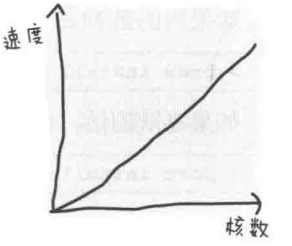
Erlang 提供了从错误中恢复的能力，用 actor 来组织代码的能力，以及用分布式和并发进行伸缩的能力，这些听起来都棒极了，那我们就赶快进入下一节吧……

保持冷静

本书中有很多放入小框框中的文字都以这个名字为标题（阅读的过程中读者会看到）。Erlang 目前之所以大受欢迎都源自于一些狂热的论点，这会导致人们对它过度信任。如果你也是一名狂热的 Erlang 学习者，那么下面的一些提醒有助于你保持清醒。

第一个提醒是，人们把 Erlang 强大的伸缩能力归结于它轻量的进程。没错，Erlang 的进程的确非常轻量，你可以同时拥有数十万个进程。但是，这并不等于说，因为能这样用 Erlang，所以一定要这样用。例如，在一个射击游戏中，让每颗子弹成为一个 actor，就很不明智。要是按照这种方式设计一个射击游戏，能射中的就只能是自己的脚丫了。在 actor 之间发送消息还是存在少量开销的，如果把任务划分得太细，效率会降低不少！

在我们学到后面，真正需要关心这个问题时，我会深入进行讲解。现在只需记住，随意地对问题进行并行化是不足以提升其效率的。（别灰心，有时候，使用上百个进程既是可行的，也是管用的！）



也有观点说，Erlang 的伸缩能力和计算机的核数成正比，但这通常不是真的。在某些情况下是可能的，但在绝大多数情况中，遇到的都是些无法让所有东西都同时运行的问题。

还有一件事情需要牢记，尽管 Erlang 在某些方面做得确实不错，但是，从技术上来说，用其他语言也是可能取得同样效果的。反之亦然。你应该仔细评估需要解决的问题，为其选择最佳工具和解决方案。Erlang 不是银弹，尤其不适合开发图像和信号处理、操作系统设备驱动之类的功能。Erlang 的强项在于：服务器端的大型软件（如队列中间件、Web 服务器、实时竞价系统和分布式数据库），协助其他语言完成一些困难的工作，高层协议实现等。至于中间地带的部分，你自行决定。

Erlang 也不是一定只能用来进行服务器端软件的开发。已经有人将 Erlang 用在很多意想不到的地方。一个例子就是 IANO，它是 Unict 团队（Catania 大学的 Eurobot 团队）制造的一个机器

人，其中的人工智能部分就是用 Erlang 开发的。在 2009 年度 Eurobot 大赛中，IANO 获得了银奖。还有一个例子是 Wings 3D，这是一个开源、跨平台的 3D 建模工具（不是 3D 渲染器），也是用 Erlang 实现的。

准备工作

学习 Erlang，只要一个文本编辑器和 Erlang 环境就足够了。可以从 Erlang 官网上获取源码和 Windows 平台的二进制可执行文件。

对于 Windows 操作系统来说，下载并运行二进制文件即可。别忘了把你的 Erlang 目录添加到 PATH 系统变量中去，这样就能从命令行直接访问它了。

对于基于 Debian 发布的 Linux 操作系统来说，要用下面的命令来安装 Erlang 包：

```
$ sudo apt-get install erlang
```

如果使用的是 Fedora 系统（假设已经安装了 yum），可以输入下面的命令来安装 Erlang：

```
# yum install erlang
```

不过，这些库中存放的通常都是些陈旧的 Erlang 安装包。使用旧的版本运行本书中的样例代码可能会得到和书中不一样的结果，对有些特定的应用，还可能会出现性能下降。因此，我建议从源代码中编译出安装包。参考包中的 README 文件，并且使用 Google 查找需要的所有安装细节。

在 FreeBSD 中，有很多可用的选项。如果用的是 portmaster，可以用下面的命令：

```
$ portmaster lang/erlang
```

如果是标准的 ports 系统，可以输入下面的命令：

```
$ cd /usr/ports/lang/erlang; make install clean
```

最后，如果想使用这个包，输入以下命令：

```
$ run pkg_add -rv erlang
```

如果用的是 Mac OS X 系统，可以用 Homebrew 来安装 Erlang：

```
$ brew install erlang
```

如果喜欢的话，也可以用 MacPorts：

```
$ port install erlang
```

> **注意** 在撰写本书时，我使用的 Erlang 版本是 R15B+，所以为获取最佳学习效果，读者应该使用这个版本或者更新的版本。不过，本书的绝大部分内容对像 R13B 这样老的版本也是有效的。

除了下载、安装 Erlang 外，读者还应当下载和本书配套的完整代码文件。其中包含有本书所有的程序和模块代码，都已经过测试。这些代码有助于修正读者自己程序中的问题。如果读者想

跳着阅读本书，那么也可以把它们当作学习后面章节的基础代码。这些文件全部被打包放在一个zip包中，可以从http://learnyousomeerlang.com/static/erlang/learn-you-some-erlang.zip下载。除此之外，本书的样例代码不再有任何其他外部依赖了。

从哪里寻求帮助

如果用的是Linux，就可以从man中找到很好的技术文档。例如，Erlang中有个lists模块（会在第1章中介绍）。要找到lists模块的文档，只需要输入以下命令：

```
$ erl -man lists
```

在Windows系统中，安装得到的文件中包括HTML文档。也可以随时从Erlang官网或其他的备选网站上下载这些文档。

如果希望把代码写得整洁些，可以从http://www.erlang.se/doc/programming_rules.shtml中学些到一些优秀的编码实践。本书中的代码也会尽量遵循这些指导原则。

有时，读者会发现仅仅了解一些技术细节并不足以解决手边的问题。此时，我主要从两个地方寻求帮助：Erlang的官方邮件列表（跟着它，就能学到不少东西）和irc.freenode.net网站上的#erlang频道。

目 录

第 1 章

启程

使用 Erlang 语言，可以在仿真器（emulator）中对绝大多数代码进行测试。在代码脚本被编译和部署后，不仅可以在仿真器中运行它们，还可以现场对它们进行修改。

在本章中，我们会介绍 Erlang shell 的使用方式，还会介绍一些基本的 Erlang 数据类型。

1.1 使用 Erlang shell

要在 Linux 或者 Mac OS X 系统上启动 Erlang shell，首先需要打开一个终端，输入 **erl**。如果之前已经正确安装了 Erlang，就会看到如下显示：

```
$ erl
Erlang R15B (erts-5.9) [source] [64-bit] [smp:4:4] [async-threads:0] [hipe]
[kernel-poll:false]

Eshell V5.9 (abort with ^G)
```

祝贺，你成功运行了 Erlang shell！

如果你是 Windows 用户，那么可以在命令行提示符中执行 erl.exe 启动 Erlang shell，但是一般建议使用 werl.exe，它存在于开始菜单中（选择 **All Programs → Erlang**）。werl 是一个专门用于 Windows 系统的 Erlang shell，它有自己的窗口，窗口具有滚动条，还支持一些行编辑快捷键（Windows 系统中的标准 cmd.exe shell 不支持这些快捷键）。然而，如果想重定向标准输入或者输出，又或者想使用管道，那么就只能使用 erl.exe 来启动 shell。

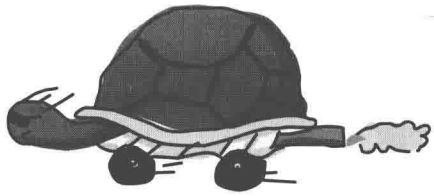
现在，我们可以在 shell 中输入代码，并在仿真器中运行它们。但是，我们先要熟悉一下 Erlang shell。

1.1.1 输入 shell 命令

Erlang shell 中内置了一个基于 Emacs 的功能子集构建的行编辑器，Emacs 是一款流行的文本编辑器，从 20 世纪 70 年代使用至今。如果你熟悉 Emacs，那就再好不过了。即便你对 Emacs 一无所知，也无妨。

首先，在 Erlang shell 中输入一些文本，然后，按下 Ctrl+A（^A）。光标会移至该行的开头。

同样地，按下 Ctrl+E（^E），光标会移至该行的末尾。你还可以使用左右箭头键让光标前后移动，用上下箭头键在之前写好的代码行之间来回切换。



我们来试一些其他输入。输入 `li`，然后按下 Tab 键。shell 会自动把刚才输入的 `li` 自动补全为 `lists:`。再次按下 Tab 键，shell 中会列举出 `lists` 模块下所有可用的函数信息。你可能会觉得这种表示法比较奇怪，不过放心，很快你就会熟悉了（第 2 章会对模块进行更多的介绍）。

1.1.2　退出 shell

至此，Elang shell 的大部分基本功能特性都已经介绍过了，除了一件非常重要的事情：我们还不知道如何退出 shell 呢！还好，有一种快速的寻求帮助的方法：在 shell 中输入 `help().`并按下 Enter 键。你会看到一系列命令的相关信息，包括进程查看函数、shell 工作方式控制函数等。其中大部分的命令都会在本书中用到，不过现在，我们最感兴趣的是下面这行描述：

```
q() -- quit - shorthand for init:stop()
```

嗯，这是退出 shell 的一种方法（实际上，一共有两种）。但是，当 shell 被冻结住时，是无法输入这条命令的！

如果刚才启动 shell 时，你留意观察过，可能会看到一句“aborting with ^G.”字样的注解。那就按下 Ctrl+G，然后再输入 `h` 寻求帮助。

```
User switch command
--> h
c [nn]            - connect to job
i [nn]            - interrupt job
k [nn]            - kill job
j                 - list all jobs
s [shell]         - start local shell
r [node [shell]]  - start remote shell
q                 - quit erlang
? | h             - this message
-->
```

如果你还带着单片眼镜，是时候扔掉它了！和其他语言中的简单 shell 完全不同，Erlang shell 是一组 shell 实例，每个实例运行着不同的作业（job）。此外，你可以像管理操作系统中的进程一样管理这些 shell 实例。如果输入 `k` *N*，*N* 是作业编号，就会终止掉承担这个作业的 shell 以及它运行的所有代码。如果不想杀死 shell，只想停止 shell 中运行的代码，可以使用 `i` *N* 命令。你还可以输入 `s` 创建一个新的 shell 实例，用 `j` 列举出所有的 shell 实例，以及用 `c` *N* 来连上一个 shell 实例。

尝试这些命令时，你可能会看到某个 shell 作业旁有个星号（*）：

```
--> j
1* {shell,start,[init]}
```

*表明这个 shell 是你最近一次使用的 shell 实例。如果在使用命令 `c`、`i` 或者 `k` 时，后面不带

任何数字，那么命令会默认作用到这个最近使用的 shell 实例上。

如果 shell 冻结住了，一个快捷的解冻指令序列是：按下 Ctrl+G、输入 i、按下 Enter 键、输入 c、按下 Enter 键（^G i ENTER c ENTER）。这个指令序列会先进入 shell 管理界面，中断当前 shell 作业，然后再重新建立连接。

```
Eshell V5.9 (abort with ^G)
1> "OH NO THIS SHELL IS UNRESPONSIVE!!! *hits ctrl+G*"
User switch command
  --> i
  --> c
** exception exit: killed
1> "YESS!"
```

在向 shell 中输入有“实际意义”的内容之前，有件重要的事情要说明一下：*表达式序列必须要以点号结尾，后面是空白符*（换行符、空格之类），否则，表达式不会被执行。你可以用逗号来分隔表达式，但是只会显示最后一个表达式的执行结果（其他表达式也都执行了）。大多数人都会觉得这个语法很奇怪，这是因为 Erlang 最初是直接在 Prolog 中实现的。Prolog 是一门逻辑编程语言。

现在，我们来做一些实践。就从学习 Erlang 的基本数据类型以及如何在 shell 中写一点 Erlang 程序开始吧。

1.2 Erlang 基础知识

虽然你刚刚看到的管理不同作业和 shell 会话的机制非常高级，但是 Erlang 仍然被认为是一门相对小巧、简单的语言（这里简单的含义与 C 比 C++简单中的含义相同）。Erlang 语言中只有少量内置的数据类型（相关的语法也不多）。我们先来看看数值类型。

1.2.1 数值类型

按照前面讲过的方法，打开 Erlang shell，并输入如下内容：

```
1> 2 + 15.
17
2> 49 * 100.
4900
3> 1892 - 1472.
420
4> 5 / 2.
2.5
```

可以看到，Erlang 并不关心输入的是浮点数还是整数。算术运算对两种类型都支持。

注意，如果你希望整数除法演算的结果还是整数，而不是浮点数，必须要使用 `div`。要获取整数除法的余数（模），可以使用 `rem`（remainder，余数）。

```
5> 5 div 2.
2
```

```
6> 5 rem 2.
1
```

在一个表达式中，可以使用多个操作符，算术计算遵循标准的优先级规则：

```
7> (50 * 100) - 4999.
1
8> -(50 * 100 - 4999).
-1
9> -50 * (100 - 4999).
244950
```

如果想以其他不为10的基数表示整数，只需以Base#Value这种形式输入数值即可（Base必须在2到36之间），像这样：

```
10> 2#101010.
42
11> 8#0677.
447
12> 16#AE.
174
```

在上面的例子中，我们把二进制数、八进制数、十六进制数转换成十进制数。太棒了！虽然语法有些奇怪，但是Erlang的计算能力完全可以和桌面角落里摆放的计算器媲美。真令人兴奋！

1.2.2　不变的变量

能做算术运算固然不错，但是如果不能保存计算结果的话，也没多大用处。你可以使用变量来存放结果。如果读过本书的引言部分，应该知道在函数式编程中，变量是不会变化的。

在Erlang中，变量必须以大写字符开始。下面的6个表达式展示了变量的基本用法：

```
1> One.
* 1: variable 'One' is unbound
2> One = 1.
1
3> Un = Uno = One = 1.
1
4> Two = One + One.
2
5> Two = 2.
2
6> Two = Two + 1.
** exception error: no match of right hand side value 3
```

从这些命令的执行中，我们首先看出的是，一个变量只能被赋值一次。之后，可以“假装”给一个变量再赋一次值，只要这个值和变量已有的值完全一样。如果这两个值不同，Erlang会报错。现象是这样，但是解释要略微复杂一些，和=操作符有关。负责比较操作并且在不相等时报错的是=操作符（不是变量）。如果相同，Erlang会返回这个值：

```
7> 47 = 45 + 2.
47
```

```
8> 47 = 45 + 3.
** exception error: no match of right hand side value 48
```

如果等号两边都是变量，并且左边是个未绑定的变量（没有任何值和它关联），Erlang 会自动把右边变量的值绑定到左边的变量上。这样，两个变量就有了相同的值。因此，两边的值比较，肯定相等，并且左边的变量会一直把这个值保存在内存中。

下面是另外一个例子：

```
9> two = 2.
** exception error: no match of right hand side value 2
```

这个命令失败了，因为单词 two 是以小写字符起始的。

> **注意** 严格来说，变量名还可以以下划线（_）起始，但是依照惯例，这种变量仅用于不想关心其值的情况。

=操作符的这种行为表现是模式匹配（pattern matching）的基础，许多函数式编程语言中都有这个特性，只是通常认为，Erlang 中的模式匹配要比其他语言中的更灵活、更完备。在本章的后面介绍其他数据类型时，我会更详细地介绍模式匹配。在后续的几章中，也会看到如何在函数中使用模式匹配。

注意，在 shell 中做测试时，如果不小心给某个变量赋错了值，可以使用 `f(Variable)`.函数把这个变量“删除”。如果想清除所有的变量，可以使用 `f()`.。这两个函数是专门用来辅助代码实验的，只能在 shell 中使用。当编写真实程序时，是不能用这种方法来销毁值的。如果你考虑到 Erlang 要适用于工业场景，就会觉得这个限制是合理的。一个 shell 极有可能不间断地运行很多年，在这期间某个给定变量名肯定会被重复使用。

1.2.3 原子

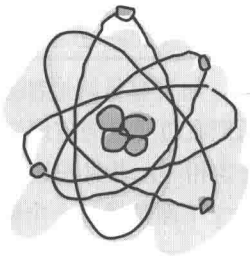
变量名不能以小写字符开头有一个原因：原子（atom）。原子是字面量，这意味着原子是常量，唯一的值就是自己的名字。换句话说，你看到的就是你能得到的——别想得到更多。原子 cat 代表“cat”，就这样。你不能操作它、不能改变它、也不能把它分成几部分。它就是 cat。只能接受它。

以小写字符开头的单词只是原子的一种写法，还有其他写法：

```
1> atom.
atom
2> atoms_rule.
atoms_rule
3> atoms_rule@erlang.
atoms_rule@erlang
4> 'Atoms can be cheated!'.
'Atoms can be cheated!'
5> atom = 'atom'.
atom
```

如果原子不以小写字符开头或者其中包含有除字母、数字、下划线以及@符号之外的其他字

符，那么必须被放到两个单引号（'）之间。第5行表明，原子加上单引号后和原来的原子相等。



我把原子比作名字，就是其值的常量。你以前编写过的代码中可能用过常量。例如，假设你用一些值来表示眼睛的颜色：1代表蓝色（blue），2代表棕色（brown），3代表绿色（green），4代表其他（other）。这些常量的名字必须要和底层的值进行匹配。但是，原子能够让你无须考虑那些底层的值。眼睛的颜色就是蓝色、棕色、绿色和其他。这些颜色可以用在代码的任何地方。底层的值绝对不会冲突，这种常量也绝不会出现没有定义的情况！（在第2章中，我们会介绍具有关联值的常量）。

因此，原子主要用来表达或者修饰和其放置在一起的数据，通常是在一个元组（tuple）（在1.2.5节进行介绍）中。有时（并不常见），单独使用原子也是有意义的。这也是为什么我们没有在此花过多篇幅介绍它的原因。在后面的例子中，会看到原子和其他类型数据的联合使用。

保持冷静

在消息发送和常量表示方面，原子确实是个不错的选择。然而，在其他许多方面，使用原子会带来一些问题。每个原子都会被引用到一个原子表中，原子表会消耗内存（在32位系统中，每个原子需要4字节；在64位系统中，每个原子需要8字节）。原子表不被垃圾回收，所以原子会一直累积，直到系统因为内存耗尽或者创建的原子个数超过了1 048 577而发生崩溃。

这意味着不能动态创建原子。如果系统需要可靠保证，但是用户输入部分动态创建了原子，那你就遇上大麻烦了，因为只要让其不停地创建原子就可以随意让系统崩溃。

原子应该被当作开发者使用的工具，坦白说，它本来就是用作此目的的。再重申一遍：日常编码中，只要原子都是你亲手输入的，那么完全可以安全地使用它们。只有动态生成原子才会存在风险。

注意　有些原子是保留字，这些原子不能使用，因为Erlang语言的设计者已经在一些特定的函数名、操作符、表达式等中使用了它们。这些保留字是`after`、`and`、`andalso`、`band`、`begin`、`bnot`、`bor`、`bsl`、`bsr`、`bxor`、`case`、`catch`、`cond`、`div`、`end`、`fun`、`if`、`let`、`not`、`of`、`or`、`orelse`、`query`、`receive`、`rem`、`try`、`when`和`xor`。

1.2.4　布尔代数和比较操作符

如果无法区分出大小或者真假，那就有大麻烦了。和其他语言一样，Erlang也有布尔操作和比较操作。

布尔代数非常简单：

```
1> true and false.
false
```

```
2> false or true.
true
3> true xor false.
true
4> not false.
true
5> not (true and true).
false
```

注意 布尔操作符 and 和 or 对操作符两边的参数都会去求值。如果你想要的是一个短路操作符（只在有必要时，才去求值右边的参数），可以使用 andalso 和 orelse。

相等性测试和不等性测试也很简单，只不过使用的符号和其他语言中稍微有些不同：

```
6> 5 =:= 5.
true
7> 1 =:= 0.
false
8> 1 =/= 0.
true
9> 5 =:= 5.0.
false
10> 5 == 5.0.
true
11> 5 /= 5.0.
false
```

在其他常见的语言中，通常使用==和!=来做相等性和不等性比较，但在 Erlang 中，使用的是=:=和=/=。最后 3 个表达式（第 9～11 行）展示了一个陷阱：在做算术计算时，Erlang 并不区分浮点数和整数，但是在做比较时，却会区分。不过别担心，如果你不想区分，可以使用==和/=操作符。因此，关键在于要想清楚是否需要精确的相等性。根据经验，你应该始终都用=:=和=/=，只有在明确知道确实不需要精确相等性时才换成==和/=。当数值的期望类型和实际类型不一致时，这种做法有助于避免一些错误的比较。

还有其他一些用于比较的操作符：<（小于）、>（大于）、>=（大于等于）和=<（小于等于）。最后一个是倒着写的（我的观点），我的代码中的许多语法错误都是由它产生的。请多留意一下这个=<。

```
12> 1 < 2.
true
13> 1 < 1.
false
14> 1 >= 1.
true
15> 1 =< 1.
true
```

如果输入 5 + llama 或者 5 =:= true 会怎样呢？要想知道答案，最好的办法当然是去

试验一下，然后被错误结果吓一跳！

```
12> 5 + llama.
** exception error: bad argument in an arithmetic expression
     in operator +/2
        called as 5 + llama
```

看来，Erlang 确实不喜欢我们将它的基本类型用错。仿真器返回一条出错消息，表明它不喜欢出现在+操作符两边的某个参数。

不过，对于用错类型，Erlang 也并不总是会生气：

```
13> 5 =:= true.
false
```

为什么在有些操作中拒绝接受不同的类型，而在另一些操作中却又允许呢？尽管 Erlang 不允许把两个不同类型的操作数加在一起，但是却允许对它们进行比较。这是因为 Erlang 语言的发明者把实用性的优先级排在理论前面，觉得如果能简单地写出像通用排序算法那样的程序，可以对任意数据排序，岂不是很棒！做出这个决定是为了简化编程工作，事实也基本上证实了这一点。

在进行布尔代数和比较操作时，还有最后一点要铭记在心：

```
14> 0 == false.
false
15> 1 < false.
true
```

如果你原来熟悉的是过程式语言或者流行的面向对象语言，那么很可能要抓狂了。第 14 行应当求值为 `true`，而第 15 行应当求值为 `false`！毕竟，`false` 代表 0，而 `true` 代表除 0 以外的任何东西！在 Erlang 中不是这样。因为我对你撒了谎。是的，我确实撒谎了。很惭愧。

Erlang 中并没有布尔值 `true` 和 `false` 之类的东西。数据项 `true` 和 `false` 都是原子，只要你始终认为 `false` 和 `true` 只能表示字面意思，那么它们其实和语言融合得是非常好的，不会带来问题。

> **注意** 比较操作中，数据类型之间的大小顺序是：number < atom < reference < fun < port < pid < tuple < list < bit string。其中有些类型你现在可能还不熟悉，随着对本书的学习，你会逐渐了解的。现在，只需要记住，正是因为有了这个顺序，才可以在任意数据类型之间进行比较。引用 Erlang 语言的发明者之一 Joe Armstrong 的一句话："具体的顺序并不重要——重要是定义明确的全局顺序。"

1.2.5 元组

使用元组可以把一组数据项组织到一起。在 Erlang 中，元组的书写形式为{*Element1*, *Element2*, ..., *ElementN*}。例如，如果你想告诉我笛卡儿坐标系中一个点的位置，就可以提供出它的坐标(x, y)。这个点可以被表示成一个具有两个数据项的元组：

```
1> X = 10, Y = 4.
4
2> Point = {X,Y}.
```

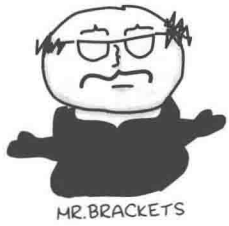
```
{10,4}
```

在这个例子中，一个点就是两个数据项。这样，就不用到哪儿都带着变量 X 和 Y，只要带着这一个元组就可以了。不过，如果拿到一个点，但是只想知道它的 x 坐标时，该怎么办呢？获取这个信息也没什么困难的。记住，在给变量赋值时，如果新旧值相同，Erlang 就不会报错。

我们来利用这个特性。（在输入下面的例子前，需要用 f() 清除以前设置的变量。）

```
3> Point = {4,5}.
{4,5}
4> {X,Y} = Point.
{4,5}
5> X.
4
6> {X,_} = Point.
{4,5}
```

从现在开始，X 就是元组中第一个元素的值了。是怎么做到的呢？一开始，X 和 Y 没有值，所以都是未绑定的变量。当把它们放在=操作符左边的元组{X,Y}中时，=操作符会去比较两边的值：{X,Y}和{4,5}。Erlang 非常聪明，它会把值从元组中取出，并把它们分配给左边的未绑定变量。接下来，就变成{4,5} ={4,5}的比较了，显然是成立的。还有很多其他种类的模式匹配形式，这只是其中的一种。



注意，第 6 行使用了无需关心型变量（_）。它的用法正如其名：把本来应该放置在这个位置的值丢弃掉，因为我们不会用到这个值。通常，_ 变量会一直被当作未绑定的，在模式匹配中充当通配符的作用。只要两边元组中元素的个数（元组的长度）相等，就可以通过模式匹配来提取元组中的元素。

```
7> {_,_} = {4,5}.
{4,5}
8> {_,_} = {4,5,6}.
** exception error: no match of right hand side value {4,5,6}
```

元组还可以配合单个值使用。例如，假设我们想保存下面的温度值：

```
9> Temperature = 23.213.
23.213
```

哇，这个温度太适合去海滩玩耍了！但是，等等，这个温度是开氏、摄氏还是华氏？我们可以用元组把温度值和单位放到一起：

```
10> PreciseTemperature = {celsius, 23.213}.
{celsius,23.213}
11> {kelvin, T} = PreciseTemperature.
** exception error: no match of right hand side value {celsius,23.213}
```

抛出异常了，不过这正是我们期望的。这同样是模式匹配所导致的。=操作符比较{kelvin, T}和{celsius, 23.213}，虽然变量 T 是未绑定的，但是 Erlang 发现了原子 celsius 和原子 kelvin 是不同的。因此产生了一个异常，并停止了代码执行。所以，期望开氏温度的程序代码

是不能处理摄氏温度的。这样，程序员就能方便地知道传入的是何种数据，并且还能用作一种辅助的调试手段。

如果一个元组中包含一个原子，原子后面只跟随着一个元素，那么这样的元组就称为带标记的元组（tagged tuple）。这种元组中的元素可以是任意类型，甚至是另外一个元组：

```
12> {point, {X,Y}}.
{point,{4,5}}
```

但是，如果想处理多个点时该怎么办呢？此时，可以使用列表（list）。

1.2.6　列表

列表是很多函数式语言的重要组成部分。列表可以用来解决各种各样的问题，毋庸置疑，列表是 Erlang 中使用最广泛的数据结构。列表中可以包含任何东西——数值、原子、元组以及其他列表——所有能想象得到的东西都可以放到这个数据结构中。

列表的基本形式是[*Element1*, *Element2*, ..., *ElementN*]，在列表中，可以混合放入多种数据类型：

```
1> [1, 2, 3, {numbers,[4,5,6]}, 5.34, atom].
[1,2,3,{numbers,[4,5,6]},5.34,atom]
```

很简单，对吧？再试另外一个：

```
2> [97, 98, 99].
"abc"
```

这是 Erlang 中最令人讨厌的东西之一：字符串。字符串就是列表，它们的表示方法完全一样。那为什么令人讨厌呢？原因如下：

```
3> [97,98,99,4,5,6].
[97,98,99,4,5,6]
4> [233].
"é"
```

在打印数值列表时，只有当其中至少有一个数字代表的不是字符时，Erlang 才会把它们作为数值打印出来。在 Erlang 中，根本就没有真正的字符串！毫无疑问，你将来肯定会为此而困扰，并会因此怨恨这门语言。不过，别失望，因为还有其他书写字符串的方法，我们会在 1.3.3 节中进行介绍。

保持冷静

这也是为什么会有程序员说 Erlang 不擅长字符串处理的原因：Erlang 中不具有其他大多数语言中的内置字符串类型。缺乏的原因是因为 Erlang 最初是由电信公司创建和使用的，他们从不（或者很少）使用字符串，所以没有把字符串作为一种独立类型正式增加到语言中。不过，随着时间的流逝，这个问题正逐步得到解决。现在，Erlang 虚拟机（VM）已经部分支持 Unicode，在字符串处理上也一直在提速。还可以把字符串存成一个二进制数据结构，更加的轻量，处理

速度也更快。我们会在 1.3.3 节进行介绍。

总之，标准库中字符串相关的函数还是不够多的。虽然用 Erlang 做字符串处理是可行的，但是如果要大量处理字符串时，可以选择更适合的语言，如 Perl 和 Python。

++操作符可以把列表粘合起来。--操作符可以从列表中删除元素。

```
5> [1,2,3] ++ [4,5].
[1,2,3,4,5]
6> [1,2,3,4,5] -- [1,2,3].
[4,5]
7> [2,4,2] -- [2,4].
[2]
8> [2,4,2] -- [2,4,2].
[]
```

++和--操作符都是右结合的。这意味着对于多个--或者++操作来说，操作是从右向左进行的，如下例所示：

```
9> [1,2,3] -- [1,2] -- [3].
[3]
10> [1,2,3] -- [1,2] -- [2].
[2,3]
```

在第一个例子中，从右向左，首先从[1,2]中移除[3]，剩下[1,2]。接着从[1,2,3]中移除[1,2]，只剩下[3]。在第二个例子中，首先从[1,2]中移除[2]，得到[1]，接着从[1,2,3]中拿走[1]，产生最后的结果[2,3]。

我们继续。列表中的第一个元素称为头（head），剩余的部分称为尾（tail）。可以用内建函数（BIF）获取它们：

```
11> hd([1,2,3,4]).
1
12> tl([1,2,3,4]).
[2,3,4]
```

注意 BIF 通常是一些不能用纯 Erlang 实现的函数，因此会用 C 或者任何其他实现 Erlang 的语言编写（在 20 世纪 80 年代是 Prolog）。也有一些 BIF，本可以用 Erlang 实现，但是为了让这些常用的操作性能更高，就用 C 来实现了。`length(List)` 函数就是一个例子，它返回参数列表的长度（你一定猜到了）。

访问列表的头元素或者给列表增加头元素都非常快捷高效。几乎所有需要处理列表的应用都是先操作列表头元素的。因为这种操作模式使用得非常频繁，所以 Erlang 通过模式匹配提供了一种简单的方式来分离列表的头和尾：`[Head|Tail]`。下例展示的是如何给列表增加一个新的头元素：

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

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

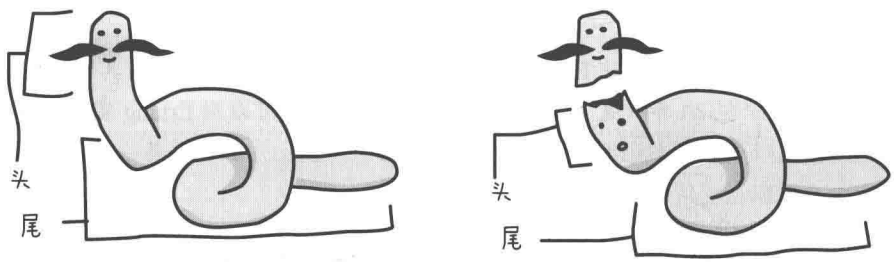
在处理列表时，如果有某种快捷的方法可以保存列表尾，以便以后继续处理，就非常方便了。如果你还记得元组的概念以及我们是如何使用模式匹配从点（{X,Y}）中提取值的，那么你就能够理解如何以类似的方式切下并得到列表中的第一个元素（头元素）：

```
15> [Head|Tail] = NewList.
[1,2,3,4]
16> Head.
1
17> Tail.
[2,3,4]
18> [NewHead|NewTail] = Tail.
[2,3,4]
19> NewHead.
2
```

这里使用的|操作符称为 cons 操作符（构造器）。事实上，仅凭 cons 操作符和值就能构建出任何列表：

```
20> [1 | []].
[1]
21> [2 | [1 | []]].
[2,1]
22> [3 | [2 | [1 | []]]].
[3,2,1]
```

换句话说，任何列表都可以用下面这个公式构建：[*Term1* | [*Term2*| [... | [*TermN*]]]]。所以，可以把列表递归地定义成一个头元素，后面跟着一个尾，而尾本身又是一个头元素后面跟着更多的头。从这个意义上说，可以把列表想象成是一条蚯蚓，你可以把它切成两半，然后，你就有了两条蚯蚓。



对那些没见过类似构造器的人来说，可能会对 Erlang 列表的构建方式感到困惑。为了帮助你熟悉这个概念，仔细阅读下面的所有例子（提示：它们都是等价的）：

```
[a, b, c, d]
[a, b, c, d | []]
```

```
[a, b | [c, d]]
[a, b | [c | [d]]]
[a | [b | [c | [d]]]]
[a | [b | [c | [d | []]]]]
```

理解了这一点，下一节中介绍的列表推导式（list comprehension）应该就好理解了。

注意 以[1|2]这种形式构建出来的列表称为非良构列表（improper list）。非良构列表可以用于[Head|Tail]这种形式的模式匹配，但是在 Erlang 的标准函数（即使是 length()）中使用时会失败。这是因为 Erlang 期望的是良构列表（proper list）。良构列表的最后一个元素是空列表。当定义一个像[2]这样的数据项时，这个列表会被自动地转换成良构形式。同样，[1|[2]]也会被转换成良构形式。尽管非良构列表是合乎语法的，但是除了某些用户自定义数据结构，它们的用途非常有限。

1.2.7 列表推导式

列表推导式（list comprehension）可以用来构建或者修改列表。和其他列表操作方式相比，它们会让程序既短小又易于理解。一开始，可能觉得这个概念不太好理解，但是，它们是值得花时间学习的。不要犹豫，请不断试验本节中的例子，直到理解为止！

列表推导式的思想来自于数学中的集合表示法，所以，如果你曾经学过一门和集合论有关的数学课，那么就会觉得列表推导式很亲切。通过规定成员必须要满足的属性，集合表示法描述了如何构建一个集合。我们来看一个简单的例子：$\{x \in \mathbb{R} : x=x^2\}$。它定义了一个集合，其中包含所有平方与自身相等的实数（这个结果集合是{0, 1}）。$\{x : x>0\}$是一个更简单的集合表示法示例。它定义了一个包含所有大于 0 的数的集合。

和集合表示法类似，列表推导式也是基于其他集合构建集合的。例如，考虑这样一个集合$\{2n : n \in L\}$，其中 L 是列表[1, 2, 3, 4]，我们可以这样理解："对列表[1, 2, 3, 4]中的每个 n，构建元素 n*2"。基于此构建出来的集合是[2,4,6,8]。对于这个集合，Erlang 的实现如下：

```
1> [2*N || N <- [1,2,3,4]].
[2,4,6,8]
```

对比一下数学表示法和 Erlang 表示法，会发现并没有大的区别：大括号（{}）变成了中括号（[]），冒号（:）变成了两个管道符（||），操作符∈变成了箭头（<-）。换句话说，只更改了符号，但是保留了同样的逻辑。在本例中，列表[1, 2, 3, 4]中的每个值会依次模式匹配到 N。箭头的作用和=操作符完全一样，只是它不会在不匹配时抛出异常。

还可以在列表推导式中使用具有布尔返回值的操作来增加约束条件。如果想得到 1 到 10 之间的所有偶数的集合，可以这样实现：

```
2> [X || X <- [1,2,3,4,5,6,7,8,9,10], X rem 2 =:= 0].
[2,4,6,8,10]
```

其中，X rem 2 =:= 0 用来检测一个数是不是偶数。

Erlang 中列表推导式的用法如下：

```
NewList = [Expression || Pattern <- List, Condition1, Condition2, ... ConditionN]
```

其中的 `Pattern <- List` 部分称为生成器表达式（generator expression）。

当想对列表中的每个元素都应用一个函数，要求它们都遵守某些约束时，列表推导式会很有用。例如，假设你有一家饭店，来了一个客人，看了菜单，问是否能够告诉他价格在3美元和10美元之间的所有菜品，消费税（大概百分之7）先不包含在内。

```
3> RestaurantMenu = [{steak, 5.99}, {beer, 3.99}, {poutine, 3.50}, {kitten, 20.99},
 {water, 0.00}].
[{steak,5.99},
{beer,3.99},
{poutine,3.5},
{kitten,20.99},
{water,0.0}]
4> [{Item, Price*1.07} || {Item,Price} <- RestaurantMenu, Price >= 3, Price =< 10].
[{steak,6.409300000000001},{beer,4.2693},{poutine,3.745}]
```

上面的小数点没做舍入，不太易读，不过，你应该知道这个例子想表达的要点。

列表推导式还有另外一个不错的地方，那就是可以同时使用多个生成器表达式，如下例所示：

```
5> [X+Y || X <- [1,2], Y <- [3,4]].
[4,5,5,6]
```

上面的表达式会执行1+3、1+4、2+3、2+4这4个操作。因此，如果想把列表推导式的用法描述得更通用些，可以写成这样：

```
NewList = [Expression || GeneratorExp1, GeneratorExp2, ..., GeneratorExpN,
Condition1, Condition2, ... ConditionM]
```

注意，也可以把生成器表达式和模式匹配放在一起当成过滤器使用：

```
6> Weather = [{toronto, rain}, {montreal, storms}, {london, fog},
6>            {paris, sun}, {boston, fog}, {vancouver, snow}].
[{toronto,rain},
 {montreal,storms},
 {london,fog},
 {paris,sun},
 {boston,fog},
 {vancouver,snow}]
7> FoggyPlaces = [X || {X, fog} <- Weather].
[london,boston]
```

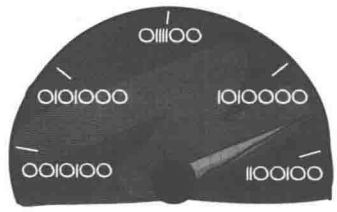
如果列表 Weather 中的一个元素不能匹配`{X, fog}`这个模式，在列表推导式中会被直接忽略，在=操作符中则会抛出异常。

在结束本章前，我们还要再学习一个基本数据类型。它是 Erlang 中一个不可思议的特性，有了它，解析二进制数据将易如反掌。

1.3　处理二进制数据

和其他大多数语言不同，在用模式匹配处理二进制数据方面，Erlang 提供了非常有用的抽象，

不需要使用那些专门的操作符进行老套的位操作。这种做法让原始二进制数据的处理变得有趣且容易（是真的），这对于 Erlang 所服务的电信应用来说是非常有必要的。乍一看，会觉得位操作的语法和惯用法非常奇怪，但是如果你知道位和字节一般的工作原理，就会觉得这些语法非常合理了。（如果你对二进制操作不熟悉的话，可以略过本章后面的内容）。



1.3.1 位语法

在 Erlang 的位（bit）语法中，会把二进制数据用<<和>>括起来，并把数据分隔成多个易理解的区段，区段之间用逗号分隔。一个区段就是一个二进制的位序列（不一定要在字节边界上，尽管默认情况下会这样）。

假设你想存储真彩（24 位）中的一个橙色像素。如果你曾经在 Photoshop 或者 Web CSS 样式表中调测过颜色，那么就应该知道颜色的十六进制表示格式是#RRGGBB。浅色调的橙色用这个记法表示就是#F09A29，在 Erlang 中可以展开成如下样子：

```
1> Color = 16#F09A29.
15768105
2> Pixel = <<Color:24>>.
<<240,154,41>>
```

大概的意思是，“把二进制数值#F09A29 以占用 24 位空间（红色占 8 位、绿色占 8 位、蓝色也占 8 位）的形式保存到变量 Pixel 中”。随后可以把这个值写入文件或者套接字中。这看起来也许不太像，但是一旦被写入文件中，这个值就会变成一串不可读的字符，在正确的上下文中，会被解码成一幅图片。

这种语法非常令人愉快，因为你可以用整洁、易读的文本去表达那些实际工作起来复杂混乱的逻辑。如果没有一个好的抽象，那么代码也会是复杂混乱的。更好的是：当把这些二进制数据从文件中读取回来时，Erlang 会把它们重新解释成漂亮的<<240,151,41>>格式！你可以在不同的表示方法之间自由地切换，根据需要，选择最有用的那种表示方法。

更有趣的是，还能使用模式匹配从二进制数据中提取内容：

```
3> Pixels = <<213,45,132,64,76,32,76,0,0,234,32,15>>.
<<213,45,132,64,76,32,76,0,0,234,32,15>>
4> <<Pix1,Pix2,Pix3,Pix4>> = Pixels.
** exception error: no match of right hand side value <<213,45,132,64,76,32,76,0,0,
234,32,15>>
5> <<Pix1:24, Pix2:24, Pix3:24, Pix4:24>> = Pixels.
<<213,45,132,64,76,32,76,0,0,234,32,15>>
```

在第 3 行中，我们定义了一个正好包含 4 个 RGB 颜色像素的二进制数据块。在第 4 行中，我们试图从这个二进制数据块中提取出这 4 个值。它抛出了异常，因为这个二进制数据中的区段个数多于 4 个——事实上，一共有 12 个区段。所以，我们要用 Pix1:24、Pix2:24 之类的告诉 Erlang，左边的每个变量都是 24 位的数据。然后，我们就可以把得到的第一个像素值进一步分解成单独的颜色值：

```
6> <<R:8, G:8, B:8>> = <<Pix1:24>>.
<<213,45,132>>
7> R.
213
```

“不错，非常棒。但是如果我一开始就只想要第一个颜色值该怎么办呢？难道每次都要把所有的值都提取出来吗？”别担心——为了解决此类问题，Erlang 提供了许多语法糖和模式匹配方法：

```
8> <<R:8, Rest/binary>> = Pixels.
<<213,45,132,64,76,32,76,0,0,234,32,15>>
9> R.
213
```

在上面的例子中，Rest/binary 是一种特别的表示法，能够让你表达无论二进制数据中还剩余什么，也无论它的长度是多少，都把它放到 Rest 变量中。所以，<<*Pattern*, Rest/binary>>对于二进制的模式匹配与 [*Head*|*Tail*]对于列表的模式匹配是一样的。

很棒，对吧？上面的表示法之所以可行，是因为对于一个二进制区段的描述，Erlang 允许有多种方式。下面的描述都是有效的：

```
Value
Value:Size
Value/TypeSpecifierList
Value:Size/TypeSpecifierList
```

其中，如果没有指定 *TypeSpecifierList*，那么 *Size* 的单位永远是位。*TypeSpecifierList* 由下面内容中的一个或者多个组成，之间用连字符（-）分隔。

类型

可能的取值为 integer、float、binary、bytes、bitstring、bits、utf8、utf16 和 utf32。如果不指定类型，Erlang 就会默认使用 integer 类型。

这个字段表示所使用的二进制数据的类型。注意，bytes 是 binary 的缩写，bits 是 bitstring 的缩写。

符号类型

可能的取值为 signed 和 unsigned。默认是 unsigned。只有当类型为 integer 时，这个字段才有意义。

字节序

可能的取值为 big、little 和 native。字节序的默认值是 big，因为它是网络协议编码中使用的标准字节序。

只有当类型为 integer、utf16、utf32 或者 float 时，字节序才有意义。这个选项和系统如何读取二进制数据有关。例如，BMP 图片文件的头格式中用 4 个字节的整数来表示文件大小。如果一个文件的大小是 72 个字节，那么一个小字节序（little-endian）系统会把它表示成<<72,0,0,0>>，而一个大字节序（big-endian）系统会把它表示成<<0,0,0,72>>。在 Erlang 中，前者会解释成 72，后者则会被解释成 1 207 959 552，所以，

要确保使用了正确的字节序。

还可以使用选项 native，会在运行时根据本地 CPU 使用的是小字节序还是大字节序进行选择。

单位

写成 unit:Integer 这样的形式。

单位指的是每个区段的大小。取值范围为1～256。对于integer、float和bitstring类型来说，单位默认设置为 1；对于 binary 类型，单位默认设置为 8。类型 utf8、utf16 和 utf32 无需定义 unit。Size 和单位的乘积等于要提取的区段中的位数，并且必须能被 8 整除。单位的大小通常用来确保字节对齐。

数据类型的默认长度可以通过组合不同的二进制区段描述加以改变。例如，<<25:4/unit:8>>会把数值 25 编码成一个 4 字节的整数，其形象的表示就是<<0,0,0,25>>。<<25:2/unit:16>>会得到同样的结果，<<25:1/unit:32>>也一样。一般来讲，Erlang 会接受 <<25:*Size*/unit:*Unit*>>这样的形式，并用 Size 乘以 Unit 算出表示这个值需要的空间。同样，空间的大小要能被 8 整除。

我们来看一些例子，帮助你理解刚才的这些定义：

```
10> <<X1/unsigned>> = <<-44>>.
<<"Ô">>
11> X1.
212
12> <<X2/signed>> = <<-44>>.
<<"Ô">>
13> X2.
-44
14> <<X2/integer-signed-little>> = <<-44>>.
<<"Ô">>
15> X2.
-44
16> <<N:8/unit:1>> = <<72>>.
<<"H">>
17> N.
72
18> <<N/integer>> = <<72>>.
<<"H">>
19> <<Y:4/little-unit:8>> = <<72,0,0,0>>.
<<72,0,0,0>>
20> Y.
72
```

可以看出，对于二进制数据的读取、存储和解释，存在多种方法。这有些令人困惑，不过与大多数语言中提供的常见工具相比，还是简单多了。

1.3.2 二进制数的按位操作

Erlang 中也提供了标准的二进制操作（向左移位、向右移位以及二进制的 and、or、xor 和 not）。相应的操作符是 bsl（按位左移）、bsr（按位右移）、band、bor、bxor 以及 bnot。

```
2#00100 = 2#00010 bsl 1.
2#00001 = 2#00010 bsr 1.
2#10101 = 2#10001 bor 2#00101.
```

使用这个表示法和通用的位语法，可以非常容易地对二进制数据进行解析和模式匹配。例如，可以使用如下代码解析 TCP 报文：

```
<<SourcePort:16, DestinationPort:16,AckNumber:32,
DataOffset:4, _Reserved:4, Flags:8, WindowSize:16,
CheckSum: 16, UrgentPointer:16,
Payload/binary>> = SomeBinary.
```

如果 SomeBinary 确实是网络处理代码发送过来的 TCP 报文，就可以用类似的方式提取出其中的内容。所有值都以位为单位（除了 Payload，它可以任意长），并且都是按照标准严格定义的。无论程序需要使用报文的哪个部分，直接使用相应的那个变量即可。

对于其他二进制数据，完全可以使用同样的匹配逻辑：视频编码、图像、其他协议实现等。

保持冷静

与 C 或者 C++相比，Erlang 要慢一些。除非你非常有耐心（或者是个天才），否则用 Erlang 去实现视频或者图像变换是不明智的，即使 Erlang 的二进制语法让这件工作变得异常有趣。Erlang 历来都不擅长数值密集型计算。

不过，请注意，对于那些不需要数值计算的应用，如事件响应、消息传递（使用原子时，会特别轻量）等，Eralng 通常是非常快的。它可以在毫秒（ms）内完成事件的处理，因此，它是构建软实时应用的极佳选择。

1.3.3　二进制字符串

还有另一个全新的二进制表示法：二进制字符串（binary string）。二进制字符串在语言中的实现方式，和用列表实现字符串使用的方式完全一样，只是在空间使用上更加高效。这是因为，普通列表与链表（一个字符一个“结点”，还有一个指向列表其余部分的引用）类似，而二进制字符串更像是 C 语言中的数组（一种紧凑填充的内存块）。

二进制字符串的语法是<<"this is a binary string!">>。与列表相比，二进制字符串的缺点是，它的模式匹配和操作方法没有列表那么简单。因此，通常当所存储的文本无需频繁操作或者空间效率是个实际问题时，才会使用二进制字符串。

注意　尽管二进制字符串非常轻量，还是不要用它们去标记数据。例如，使用字符串常量去表达{<<"temperature">>,50}是很有吸引力的，不过，对于这种情况，应该永远使用原子。如果使用原子，那么在对不同的值进行比较时几乎没有开销，并且不管原子多长，比较总可以在常量时间完成，而二进制字符串的比较则是线性时间的。另一方面，也不要因为原子更轻量，就用原子取代字符串。字符串是可以运算的（分隔、正则表达式等），而原子除了可以比较，什么都不能做。

1.3.4 二进制推导式

二进制推导式（binary comprehension）之于位语法就等同于列表推导式之于列表：在处理二进制数据时，可以让代码短小精悍。一般来讲，它的使用方式也和列表推导式完全一样：

```
1> << <<X>> || <<X>> <= <<1,2,3,4,5>>, X rem 2 == 0>>.
<<2,4>>
```

语法层面唯一的变化是：标准列表推导式中的<-变成了<=（用于二进制生成器），使用的是二进制数据（<<>>），而不是列表数据（[]）。

在本章的前面，你曾经看到过一个使用模式匹配从表示大量像素数据的二进制数据中提取RGB 值的例子。当时采用的技术在那个例子中是恰当的，但是对于更大型的结构，那样的做法会让代码变得难以理解和维护。使用二进制推导式，只需一行代码就可以完成同样的功能，更加整洁：

```
2> Pixels = <<213,45,132,64,76,32,76,0,0,234,32,15>>.
<<213,45,132,64,76,32,76,0,0,234,32,15>>
3> RGB = [ {R,G,B} || <<R:8,G:8,B:8>> <= Pixels ].
[{213,45,132},{64,76,32},{76,0,0},{234,32,15}]
```

把<-换成<=，我们就能把二进制数据当成生成器使用。上面的整个二进制推导式就是把二进制数据转换成元组中的整数。还可以通过另外一个二进制推导式来完成正好相反的功能：

```
4> << <<R:8, G:8, B:8>> || {R,G,B} <- RGB >>.
<<213,45,132,64,76,32,76,0,0,234,32,15>>
```

这样做时要非常小心，因为如果生成器返回的是二进制数，那么就必须要对结果二进制数中的元素给出明确的二进制类型定义。

```
5> << <<Bin>> || Bin <- [<<3,7,5,4,7>>] >>.
** exception error: bad argument
6> << <<Bin/binary>> || Bin <- [<<3,7,5,4,7>>] >>.
<<3,7,5,4,7>>
```

默认情况下，Erlang 会认为放入二进制数中以及从二进制数中提取的值都是整数（8 位无符号）。当写出<<Bin>>时，实际上是在说，我们想要一个二进制数，里面包含了一个存储在变量 Bin 中的整数。问题是，Bin 中保存的是另外一个二进制数，这对于 Erlang 来说完全不合理。我们承诺会提供一个整数，结果提供了一个二进制数。通过指定 Bin 的类型是 binary（见第 6 行），Erlang 就能够处理这个模式了，因为我们对 Bin 的描述和 Bin 中当前包含的完全吻合了。

还可以在二进制推导式中使用二进制生成器：

```
7> << <<(X+1)/integer>> || <<X>> <= <<3,7,5,4,7>> >>.
<<4,8,6,5,8>>
```

注意，在本例中把类型指定为 integer 是多余的，因为 integer 是 Erlang 中的默认类型。

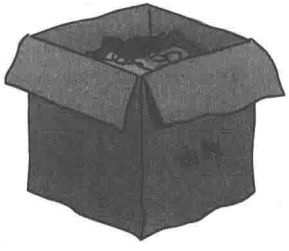
在本书中，我不会再介绍有关二进制数和二进制推导式的更多细节内容。如果你想更全面地了解位语法，可以去阅读位语法的规范定义白皮书，参见 http://user.it.uu.se/~pergu/papers/erlang05.pdf。

第 2 章

模块

使用交互式 shell 是动态语言编程中至关重要的一个部分。在交互式 shell 中，可以方便地测试代码和程序。在第 1 章中，我们甚至都没有使用文本编辑器或者保存文件，直接在交互式 shell 中就学习了 Erlang 语言的大部分基本数据类型。虽然你可以现在就放下这本书，跑出去打打球，并就此结束你的 Erlang 学习生涯，不过这样做会让你成为一个非常糟糕的 Erlang 程序员。代码当然需要存在某个地方才能被使用！在本章中，你将会看到，这就是模块要负责的事情。

2.1 什么是模块



模块（module）是一个具有名字的文件，其中包含一组函数。Erlang 中的所有函数都必须定义在模块中。你之前已经使用过模块了，不过可能没有意识到。在第 1 章中提到的那些 BIF 函数，如 `hd` 和 `tl`，实际上是属于 `erlang` 模块的。所有的算术、逻辑和布尔操作函数也都在 `erlang` 模块中。

`erlang` 模块中 BIF 函数和其他函数不同，因为在启动 Erlang 时，它们会被自动引入。模块中的其他所有函数都必须用 *`Module:Function(Arguments)`* 这样的形式进行调用，如下例所示：

```
1> erlang:element(2, {a,b,c}).
b
2> element(2, {a,b,c}).
b
3> lists:seq(1,4).
[1,2,3,4]
4> seq(1,4).
** exception error: undefined shell command seq/2
```

其中，lists 模块中的 seq 函数不会被自动引入，而 element 函数会被自动引入。之所以出现“undefined shell command”这条出错消息，是因为 shell 会像查找 f() 这样的 shell 命令一样去查找 seq，但是没有找到。erlang 模块中有些函数也没有被自动引入，不过它们使用的不是很频繁。

从逻辑上来说，应该把处理类似事情的函数放在同一个模块中。列表的通用操作函数被放在 lists 模块中，处理输入和输出的函数（如向终端或者文件写入东西）被组织在 io 模块和 file 模块中。唯一违反这个原则的是 erlang 模块，其中包含有数学计算、类型转换、多进程处理、虚拟机设置等函数。除了都是 BIF 函数，它们之间没有任何共同点。避免创建类似 erlang 这样的模块，要关注于整洁、有逻辑的模块划分。

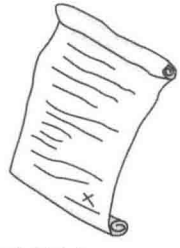
2.2 创建模块

编写模块时，可以定义两种东西：*函数*（function）和*属性*（attribute）。属性是元数据，用来描述模块自身，如模块的名字、外部可见的函数、模块的作者等。这类元数据很有用，因为它既可以为编译器提供工作提示，也可以让程序员从编译过的代码中获取信息，无需去查看源码。

目前，世界各地的程序员在 Erlang 代码中所使用的属性种类非常繁多。事实上，只要你愿意，甚至可以声明自己的专有属性。不过，有些预定义的模块属性在代码中的使用频率要远高于其他属性。

所有模块属性都采用 *-Name(Attribute).* 的形式。要让你的模块能够编译，下面的模块属性必须定义：

```
-module(Name).
```



这个属性永远是文件中第一个属性（也是第一条语句），理由很充分：它是当前这个模块的名字，其中 *Name* 是一个原子。你调用其他模块中的函数时，使用的就是这个名字。函数调用采用的形式是 M:F(A)，其中 M 是模块名，F 是函数名，A 是参数。

注意，-module 属性中定义的模块名必须和模块文件的名字一致。例如，如果模块名是 unimaginative_mame，那么文件名必须是 unimaginative_mame.erl（.erl 是标准的 Erlang 源文件扩展名）。如果名字不一致，那么模块将无法编译。

一切就绪，可以编码了！我们要写的第一个模块非常简单，也没啥用处。打开文本编辑器，输入以下内容，然后把文件保存为 useless.erl。

```
-module(useless).
```

只要有了这一行就是一个合法的模块。的确如此！当然，因为没有任何函数，所以也没什么用处。我们先来决定一下要从 useless 模块中导出（export）哪些函数。要想导出函数，需要使用另一个属性：

```
-export([Function1/Arity, Function2/Arity,..., FunctionN/Arity]).
```

这个属性用来定义模块中的哪些函数可以被其他模块调用。它接收一个带有各自元数（arity）的函数名列表。函数的元数是个整数，表示这个函数可以接收的参数个数。这是一个关键的信息，因为在同一个模块中定义的不同函数，只要它们的元数不同，就可以使用相同的函数名字。函数add(X,Y)和add(X,Y,Z)就是不同的函数，分别表示成add/2和add/3。

> **注意**　导出的函数相当于模块的接口。在定义接口时，尽量只暴露使用这个模块所必需的最少的信息，这一点非常重要。这样，你就可以随意更改内部实现细节，而不会影响那些依赖于你的模块的代码。

我们先让useless模块导出一个名为add的函数，这个函数有两个参数。在模块定义后面增加如下的-export属性：

```
-export([add/2]).
```

接着，我们来实现这个函数：

```
add(A,B) ->
    A + B.
```

函数定义的语法遵循 *Name*(*Args*)-> *Body*.这样的形式，*Name* 必须是一个原子，*Body* 可以是一个或者多个用逗号分隔的Erlang表达式。函数以一个句点结束。注意，和许多命令式语言不同，Erlang中没有return关键字。return毫无用处！无需显式说明，函数中最后一个表达式的执行结果会被自动作为返回值传递给调用者。

接下来，在文件中增加下面的函数。（是的，每个教程都需要一个“Hello,World”例子！）不要忘了把这个函数也加到-export属性中（现在，-export属性应该是这样：-export([add/2,hello/0]).）。

```
%%  显示欢迎语
%%  io:format/1 是标准的文本输出函数
hello() ->
    io:format("Hello, world!~n").
```

先来讲讲上面代码中的注释。在Erlang中，注释只能是单行的，以%符号起始。（在本例中使用了%%，这纯粹是风格问题）。hello/0函数展示了如何在自己的模块中调用其他模块的函数。在本例中，正如注释写的那样，io:format/1是一个标准的文本输出函数。

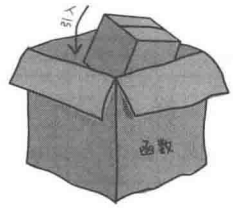
> **注意**　在Erlang社区中，习惯于在模块的概括性注释（模块是做什么的，许可证等）以及模块的区段分隔注释（公有代码、私有代码、辅助函数等）中使用3个百分号（%%%）。在所有其他需要放置在独立行中的注释使用2个百分号（%%），并和周边的代码采用同样的缩进。放在代码之后的行内注释，使用单个%。

我们来给这个模块增加最后一个函数，它同时使用了add/2和hello/0函数：

```
greet_and_add_two(X) ->
```

```
    hello(),
    add(X,2).
```

同样，不要忘了把 greet_and_add_two/1 添加到导出函数列表中。对 hello/0 和 add/2 的调用不需要在前面加上模块名，因为这两个函数就是定义在这个模块内的。



如果想用和 add/2 或者其他定义在当前模块中的函数一样的方式去调用 io:format/1，可以在文件的前面增加一个模块属性：-import(io,[format/1])..。然后，就可以直接调用 format("Hello,World!~n").了。-import 属性定义的一般形式如下：

```
-import(Module, [Function1/Arity, ..., FunctionN/Arity]).
```

因为引入函数的做法会降低代码的可读性，所以大多数程序员都强烈反对在代码中使用 -import 属性，尽管它确实能带来一些便捷。例如，以函数 io:format/2 为例，这个函数有一个同名函数，不过位于另外一个库 io_lib:format/2 中。此时，如果函数是被引入的，那么要判断出代码中使用的到底是哪个函数，就要到文件的最前面去查看这个函数是从哪个模块中引入的。因此，在函数调用中包含模块名被认为是一项好的实践，对于那些喜欢使用 grep 在工程间进行搜索的 Erlang 程序员来说，这种做法也是有帮助的。通常，只有 lists 模块中的函数会被引入，因为 lists 模块中函数的使用频率要远高于其他大多数模块中的函数。

现在，useless 模块的内容应该如下所示：

```
-module(useless).
-export([add/2, hello/0, greet_and_add_two/1]).

add(A,B) ->
    A + B.

%% 显示欢迎语
%% io:format/1 是标准的文本输出函数
hello() ->
    io:format("Hello, world!~n").

greet_and_add_two(X) ->
    hello(),
    add(X,2).
```

useless 模块编写完成了。保存 useless.erl 文件，接下来可以尝试编译它了。

2.3 编译代码

Erlang 代码会被编译成字节码，这样 VM 就能执行它了。调用编译器有多种方法。最常用的一种是在命令行中调用它，例如：

```
$ erlc flags file.erl
```

如果是在 shell 或者模块中，可以像这样编译代码：

```
compile:file(Filename)
```

还有一种方法，在开发代码时经常使用，就是在 shell 中编译：

```
c()
```

好了，是时候编译运行 useless 模块了。但是，先要告诉 Erlang shell 在哪里能找到我们的模块。打开 Erlang shell，输入下面的命令，要填上文件存放目录的全路径。

```
1> cd("/path/to/where/you/saved/the-module/").
"Path Name to the directory you are in"
ok
```

默认情况下，shell 只会在它的启动目录和标准库中去查找文件。cd/1 函数专用于 Erlang shell，可以更换 shell 的当前目录，这样寻找文件就会方便一些。

接下来，输入以下命令：

```
2> c(useless).
{ok,useless}
```

如果你看到的信息和上面不同——出现了像 userless.erl:Line: Some Error Message 之类的显示——请确认一下文件名是否正确，文件的路径是否正确，模块中是否有括号不匹配、遗漏了结束符（.）之类的错误，等等。

代码被成功编译后，你会发现在工作目录中，紧挨着 useless.erl 之后，多了一个 useless.beam 文件。这就是编译好的模块文件。

> **注意**　文件扩展名.beam 是 Bogdan/Björn's Erlang Abstract Machine 的缩写，也是虚拟机的名字。还有一些其他种类的 Erlang 虚拟机，但是基本上都不再使用了。例如，灵感来自于 Prolog 的 WAM 的 Joe's Abstract Machine （JAM）以及试图先把 Erlang 编译成 C，然后再编译成本地码的旧版 BEAM。基准测试表明这种做法收效甚微，所以就被放弃了。最近，还有人尝试把 Erlang 移植到 JVM 上，并发明了 Erjang 语言。尽管结果令人印象深刻，但是鲜有开发者把 Erlang 的开发工作迁移到 Java 平台上。

现在，调用一下我们的函数吧！

```
3> useless:add(7,2).
9
4> useless:hello().
Hello, world!
ok
5> useless:greet_and_add_two(-3).
Hello, world!
-1
6> useless:not_a_real_function().
** exception error: undefined function useless:not_a_real_function/0
```

函数的表现符合预期：add/2 对数字相加，hello/0 输出“Hello,world!”，greet_and_add_two/1 做了上面两件事。当然，你可能会问为什么 hello/0 在输出文本后返回了原子 ok。这是因为 Erlang 中的函数和表达式必须要有返回值，虽然在其他语言中可能并不需要。正因为如此，io:format/1 函数返回 ok 表示情况正常：没有错误发生。

第 6 行的调用抛出了一个异常，因为我们调用的函数并不存在于模块中。如果忘了把一个函数导出，那么当试图去调用它时，也会看到此类出错消息。

编译选项

Erlang 提供了许多编译选项，用来对一个模块的编译方式进行控制。你可以在 Erlang 文档中看到所有的编译选项。下面列出的是一些最常用的选项。

-debug_info

像调试器、代码覆盖率统计以及静态分析之类的 Erlang 工具都是使用模块中的调试信息来完成工作的。一般来说，建议这个编译选项一直开启。比起不开启这个选项所节省的那一点点字节码空间来说，你可能更需要这个选项所带来的好处。

-{outdir,Dir}

默认情况下，Erlang 编译器会将生成的.beam 文件放置到当前目录下。可以用这个选项指定编译文件的存放路径。

-export_all

这个选项会让编译器忽略文件中已定义的-export 模块属性，把文件中的所有函数都导出。这个选项在测试和开发新代码时特别有用，但在产品代码中严禁使用。

-{d,Macro}和{d,Macro,Value}

这个选项定义了一个可以在模块中使用的宏，其中 Macro 是个原子。这个选项在单元测试中用得最多，因为它能确保模块中的测试函数只在明确需要时才会被创建和导出。如果元组中没有定义第三个元素，Value 会被默认设置为 true。

在编译 useless 模块时，如果想使用编译选项，可以通过如下两种方式：

```
7> compile:file(useless, [debug_info, export_all]).
{ok,useless}
8> c(useless, [debug_info, export_all]).
{ok,useless}
```

还可以在模块内部通过模块属性来定义编译选项。要达到与前面第 7 行和第 8 行一样的编译效果，可以在模块中增加如下内容：

```
-compile([debug_info, export_all]).
```

注意 还有一个可以把 Erlang 代码编译成本地码的编译选项。并不是所有平台和操作系统中都能进行到本地码的编译，只能在支持这个特性的平台或者操作系统中进行，编译成本地码会让程序运行得更快（坊间传闻，大概能提速 20%）。要编译成本地码，需要使用 hipe

模块，然后调用 hipe:c(Module,OptionsList)来编译。也可以在 shell 中调用 c(Module,[native]).达到同样的效果。注意，这样编译出来的.beam 文件就不能跨平台了。一般来讲，在提升 CPU 密集型操作的性能时，用 hipe 进行编译往往都是最后的选择。

2.4 定义宏

Erlang 的宏和 C 语言的#define 语句类似，主要用来定义简短的函数和常量。在代码被编译成供 VM 使用的字节码之前，它们会被替换成所代表的文本表达式。这些宏主要用来避免四散在模块代码中的“魔数”。例如，如果你看到一段比较某个变量和一个硬编码数 3 600 的代码，就不清楚这个数字代表的是 1 h（3 600 s）还是 60 h（3 600 min），还是金钱数额之类的东西。然而，如果你看到的是?HOUR 这样一个 Erlang 宏，那么立即就能明白是什么意思。更好的是，如果你最后把这个表示从秒（3 600）更改为毫秒（3 600 000），那么只需要更改这个宏定义，代码中所有使用该宏的地方都会自动得到更新。

Erlang 中的宏是通过模块属性来定义的，如下所示：

```
-define(MACRO, some_value).
```

然后，你就可以在模块的任意函数中使用宏?MACRO 了，这个宏在代码编译前会被替换成 *some_value*。对于前面的小时例子，可以这样定义宏：

```
-define(HOUR, 3600).  % 单位是秒
```

“函数”宏的定义方法类似。下面是一个简单的函数宏，进行两个数的减法：

```
-define(sub(X,Y), X-Y).
```

要使用这个宏，像调用其他宏那样调用它就行了。例如，如果调用?sub(23,47)，这个调用会被编译器替换成 23-47。

Erlang 中有一些预定义的宏，下面列出了其中的一部分：

- ?MODULE，会被替换成当前模块的名字，是一个原子；
- ?FILE，会被替换成当前文件的名字，是一个字符串；
- ?LINE，会被替换成该宏所在的代码行的行号。

你还可以检测一下某个特定的宏是否已在代码中定义，并且根据这个结果有条件地定义其他宏。使用属性-ifdef(*MACRO*).、-else.和-endif.能够完成上述逻辑，如下例所示：

```
-ifdef(DEBUGMODE).
-define(DEBUG(S), io:format("dbg: "++S)).
-else.
-define(DEBUG(S), ok).
-endif.
```

在代码中使用时，这个宏看起来像这样：?DEBUG("entering some function")，只有在编译这个模块时定义了 DEBUGMODE 宏，才会打印出信息。否则，会被替换成原子 ok，什么都不做。

我们再来看一个例子，在这个例子中定义了一个只有在某个测试宏被定义时才会存在的测试函数：

```
-ifdef(TEST).
my_test_function() ->
    run_some_tests().
-endif.
```

然后，使用前面介绍的编译选项，我们可以选择是否定义 DEBUGMODE 或者 TEST，就像这样：c(Module, [{d,'TEST'},{d,'DEBUGMODE'}])。

2.5 模块的其他内容

接下来，我们会编写一些功能更强大的函数，不会再编写一些无用的代码，不过在此之前，我们先来了解和模块有关的其他一些杂项知识，以后也许会用得上。

2.5.1 元数据

在本章前面曾经提到过，模块属性是描述模块自身的元数据。如果不能访问源代码，那么在哪里才能找到这些元数据呢？嗯，编译器已经帮我们搞定了——在编译一个模块时，编译器会提取出大部分模块属性并把它们（还有其他一些信息）保存在 module_info/0 函数中。

可以用如下方式查看 useless 模块的元数据：

```
9> useless:module_info().
[{exports,[{add,2},
           {hello,0},
           {greet_and_add_two,1},
           {module_info,0},
           {module_info,1}]},
 {imports,[]},
 {attributes,[{vsn,[174839656007867314473085021121413256129]}]},
 {compile,[{options,[]},
 {version,"4.8"},
 {time,{2013,2,13,2,56,32}},
 {source,"/home/ferd/learn-you-some-erlang/useless.erl"}]}]
10> useless:module_info(attributes).
[{vsn,[174839656007867314473085021121413256129]}]
```

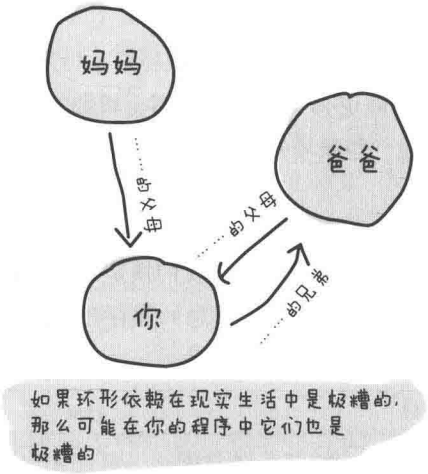
上面的示例中展示了模块另外一个函数 module_info/1，用它来获取一些特定信息。在上面的信息中，可以看到有导出的函数、引入的函数（本例中没有）、属性（自定义的元数据就在其中）以及一些编译选项和信息。如果你在module中增加了-author("An Erlang Champ").，那么它会出现在和 vsn 同样的区段中。

> **注意** vsn 是一个自动生成的唯一值，用来区分代码的不同版本，生成这个值时注释内容除外。它通常用在代码热加载（对运行中的应用进行升级，无需停止它）以及某些发布管理工具中。也可以通过在模块中增加-vsn(VersionNumber)属性来自行指定一个 vsn 值。

对于产品代码来说，模块属性的作用很有限，不过通过一些小技巧，它们也能用来解决问题。例如，在我为本书写的测试脚本中，就用它们来标注那些在单元测试时需要更严格对待的函数。脚本会查看模块属性，找出被标记的函数，并显示出关于它们的警告信息。如果你对这个脚本感兴趣，可以从 http://learnyousomeerlang.com/static/erlang/tester.erl 找到它。

2.5.2　环形依赖

关于模块设计，还有一点要牢记：一定要避免环形依赖。如果模块 B 调用了模块 A，那么模块 A 就不应当再去调用模块 B。这样的依赖关系最终会导致代码难以维护。



事实上，如果代码依赖于太多的模块——即使它们之间并不构成环形依赖——也会让代码变得难以维护。你肯定不希望哪天半夜醒来，发现一个疯狂的软件工程师正试图戳瞎你的眼睛，就因为你写的代码太烂了。

好了，说教的话到此为止。在第 3 章中，我们将继续 Erlang 的探索之旅，学习与函数有关的内容。

第3章

函数

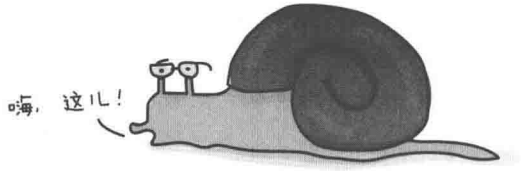
既然我们已经学会了如何保存和编译代码，接下来就可以编写功能更强大的函数了。之前所编写的函数都非常简单，有些乏味。现在，我们来学习更加有趣的内容。在本章中，我们将编写一些函数，它们会根据传入参数的不同而表现出不同的行为，还会学习几种表达式，可以用来基于不同的条件进行决策。

3.1 模式匹配

我们要编写的第一个函数会根据性别给出不同的问候语。在大多数过程式语言中，要实现这个功能，所写出的代码可能和下面的伪码类似：

```
function greet(Gender,Name)
    if Gender == male then
        print("Hello, Mr. %s!", Name)
    else if Gender == female then
        print("Hello, Mrs. %s!", Name)
    else
        print("Hello, %s!", Name)
    end
```

使用我们在第 1 章中介绍过的模式匹配，Erlang 可以帮你省去大量的样板代码。第 1 章中还展示了如何对列表和元组之类的结构中的变量进行比较和赋值（还记得`{point,{X,Y}}`这样的模式吧）。



在定义函数时，可以使用类似的模式。上面 `greet` 函数的 Erlang 版本实现如下所示：

```
greet(male, Name) ->
    io:format("Hello, Mr. ~s!", [Name]);
greet(female, Name) ->
```

```
    io:format("Hello, Mrs. ~s!", [Name]);
greet(_, Name) ->
    io:format("Hello, ~s!", [Name]).
```

当在 shell 中出现模式不匹配的情况时，Erlang 会很生气地抛出来一条出错消息。而当函数中的一个模式匹配失败时（如 greet(male,Name)），Erlang 会继续查找函数的下一个模式（这个例子中，就是 greet(female, Name)）进行匹配，如果匹配成功，就运行该匹配对应的语句。

greet 的两种实现版本之间的区别在于：在 Erlang 版本中，使用模式匹配同时完成了函数选择和变量绑定两件事情。无需先绑定变量，然后再去比较它们。所以，不必使用这种形式：

```
function(Args)
    if X then
        Expression
    else if Y then
        Expression
    else
        Expression
```

而是写成这样：

```
function(X) ->
    Expression;
function(Y) ->
    Expression;
function(_) ->
    Expression.
```

两种写法得到的结果相同，但是后一种写法更具有声明性。

其中的每一条函数声明都被称作一个函数子句（function clause）。函数子句之间必须用分号（;）分隔，所有函数子句一起形成一个完整的函数定义。可以把整个函数定义看作是一个更大的语句，这也是为什么最后一个函数子句以句点结束的原因。其中，确定工作流程所用的符号会有些奇怪，不过你会习惯的。你最好也这么认为，因为别无他法。

用 io:format 进行格式化输出

io:format 函数的格式化输出是通过替换字符串中的标记符来完成的。波浪号（~）字符用来指示一个标记符。有些标记符是内置的，如~n，会被替换成一个换行符。其他标记符几乎都是用来指示数据的格式化方法的。例如，函数调用 io:format("~s!~n",["Hello"]). 中包含了标记符~s，它接受字符串和二进制字符串作为参数。最终输出的消息是"Hello!\n"。另外一个广泛使用的标记符是~p，它会打印一个 Erlang 数据项，打印的方式和 Erlang shell 打印数据项的方式完全一样（增加必要的缩进和其他格式）。

在后续内容中，我们会看到更多 io:format 的用法，不过现在，你可以试试下面这些调用，看看它们都做了些什么：

```
io:format("~s~n",[<<"Hello">>])
io:format("~p~n",[<<"Hello">>])
io:format("~~~n")
io:format("~f~n", [4.0])
io:format("~30f~n", [4.0])
```

以上示例只是 io:format 函数众多特性的冰山一角。你可以去阅读在线文档，了解更多内容。

3.1.1 模式进阶

函数中的模式匹配可以非常复杂、强大。在第 1 章中曾经讲过，我们可以对列表进行模式匹配，从而获取它的头和尾。我们现在就来实现！

创建一个名为 fucntions 的新模块：

```
-module(functions).
-compile(export_all). % 理智起见，以后会替换成-export()!
```

在这个模块中，我们会编写一系列函数来探索模式匹配的各种用法。我们要写的第一个函数是 head/1，它的作用和 erlang:hd/1 完全一样。它以一个列表为参数，返回这个列表的第一个元素。借助于 cons 操作符（|）和“无需关心”型变量（_），可以这样实现这个函数：

```
head([H|_]) -> H.
```

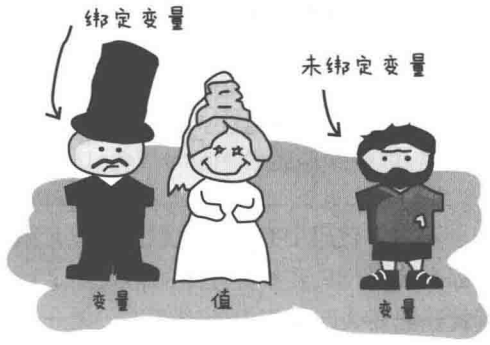
如果在 shell 中输入 functions:head([1,2,3,4]).（要先编译这个模块），会期望它的返回值是 1。同样，如果想得到列表的第二个元素，可以创建下面的函数：

```
second([_,X|_]) -> X.
```

Erlang 非常聪明，它会深入列表内部，提取出让模式匹配成功所需要的内容。在 shell 中试一下：

```
1> c(functions).
{ok, functions}
2> functions:head([1,2,3,4]).
1
3> functions:second([1,2,3,4]).
2
```

只要你愿意，列表中的所有值都可以通过模式匹配取出，不过，当列表的元素数量达到了数千个时，这种方法就不太现实了。要处理这种情况，一种更聪明的方法是使用递归函数，会在第 5 章中介绍。现在，我们还是集中精力了解更多关于模式匹配的内容。



3.1.2 绑定中的变量

在第 1 章中介绍的自由变量和绑定变量的概念同样适用于函数。我们用一个婚礼场景来复习一下绑定变量和未绑定变量。

图中，新郎有些沮丧，因为在 Erlang 中，变量

的值永远不能改变——没有自由！另一方面，未绑定的变量没有任何值附着在其上（就像右图中右边的那个小流浪汉一样）。变量绑定就是将一个值附着到一个未绑定的变量上。在 Erlang 中，如果给一个已经绑定的变量赋值，除非这个新值和变量原有的值相同，否则就会引发错误。我们来想象一下，左边的这个家伙娶了双胞胎姐妹中的一个。如果另外一个跑来拜访，他无法区分她们，所以行为正常。如果是别的女人闯了进来，他就会抱怨的。（如果你还不清楚这些概念，可以去复习一下 1.2.2 节）。

使用函数和模式匹配，可以比较出传给函数的两个参数是否完全一样。为此，我们创建一个名为 same/2 的函数，它有两个参数，返回两个参数是否一样：

```
same(X,X) ->
    true;
same(_,_) ->
    false.
```

就这么简单。

当调用 same(a,a) 时，第一个 X 未绑定，所以会自动获取值 a。然后，当 Erlang 处理第二个参数时，发现 X 已经绑定了。于是，Erlang 就把 X 已经绑定的值和作为第二个参数传递给函数的 a 进行比较，检查它们是否匹配。模式匹配成功，所以函数返回 true。如果两个值不同，模式匹配失败，就会进入第二个函数子句，这个子句不关心传递给它的参数（都已经是最后一个选择了，就别那么吹毛求疵了！），直接返回 false。注意，这个函数实际上能够接收任何类型的参数。它可以对任意类型的数据进行比较，不仅仅是列表或者单个变量。

现在来看一个更高级的例子。下面的函数只会把格式正确的日期打印出来。

```
valid_time({Date = {Y,M,D}, Time = {H,Min,S}}) ->
    io:format("The Date tuple (~p) says today is: ~p/~p/~p,~n",[Date,Y,M,D]),
    io:format("The time tuple (~p) indicates: ~p:~p:~p.~n", [Time,H,Min,S]);
valid_time(_) ->
    io:format("Stop feeding me wrong data!~n").
```

注意，在函数头中可以使用=操作符，这样可以在匹配元组内部元素（{Y,M,D}）的同时，匹配整个元组。可以这样来测试这个函数：

```
4> c(functions).
{ok, functions}
5> functions:valid_time({{2013,12,12},{09,04,43}}).
The Date tuple ({2013,9,6}) says today is: 2013/9/6,
The time tuple ({9,4,43}) indicates: 9:4:43.
ok
6> functions:valid_time({{2013,09,06},{09,04}}).
Stop feeding me wrong data!
ok
```

不过，代码中有一个问题。不管元组中的元素值是什么类型，即使是文本或者原子，这个函数都会接受，只要参数元组的格式满足{{A,B,C},{D,E,F}}。这是模式匹配的一个局限之处。模式匹配要么匹配指定的精确值，如一个已知的数或者原子，要么匹配一些抽象值，如列表的头

或者尾、具有 N 个元素的元组或者任意东西（使用_或者未绑定变量）。要解决这个问题，需要使用卫语句（guard）。

3.2 卫语句

*卫语句*是附加在函数头中的语句，能够让模式匹配更具表达力。正如前面提到的那样，模式匹配存在一定的限制，它不能表达像取值范围或者特定数据类型之类的东西。

计数（counting）是模式匹配无法表示的一个概念：这个 12 岁的篮球选手是不是太矮了，以至于不能和职业选手一起比赛？这段距离是不是太长了，不能倒立着走完？你的年龄是不是太老了或者太小了，从而不能开车？这些问题都无法通过简单的模式匹配回答。当然，你可以用下面这种非常不实用的方法来解决开车问题：

```
old_enough(0) -> false;
old_enough(1) -> false;
old_enough(2) -> false;
...
old_enough(14) -> false;
old_enough(15) -> false;
old_enough(_) -> true.
```

如果你愿意的话，可以这样做，不过，你只能永远独自维护自己的代码了。如果你还想交到朋友，就创建一个新的 guards 模块，输入下面的代码来解决开车问题：

```
old_enough(X) when X >= 16 -> true;
old_enough(_) -> false.
```

搞定了！可以看出，这个解决方案比前面的版本要简短、整洁很多。

卫表达式有一条基本规则，要想成功，它必须返回 true。如果返回了 false 或者抛出了异常，就表明卫语句失败。

假设现在禁止年龄超过 104 岁的人驾驶车辆。驾驶员的合法年龄范围是 16～104 岁。该如何处理这种情况呢？只要加入第二个卫语句即可：

```
right_age(X) when X >= 16, X =< 104 ->
    true;
right_age(_) ->
    false.
```

在卫表达式中，逗号（,）的作用和操作符 andalso 类似，分号（;）和 orelse（在第 1 章中介绍过）类似。因为 right_age/1 函数中使用了逗号，所以要让整个卫语句通过，两个卫表达式都必须成功。事实上，函数可以带有任意多个由逗号分隔的卫表达式，这些卫表达必须都成功，整个卫语句才能通过。

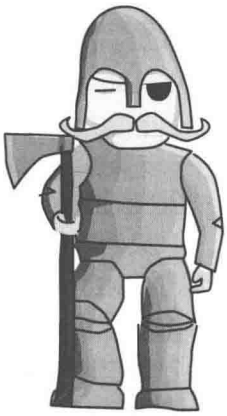
我们也可以从反方向来定义这个函数：

```
wrong_age(X) when X < 16; X > 104 ->
```

```
    true;
wrong_age(_) ->
    false.
```

这个方法也能得到正确的结果。愿意的话，可以测试一下。（你应该一直测试！）

> **注意**　我刚才把卫语句中的,和;比作操作符 andalso 和 orelse。其实，它们不完全相同。前面的那一对会捕获发生的异常，而后面的那一对不会。这就意味着，如果卫语句 X >= N; N >= 0 的前半部分在执行时抛出了异常，后半部分仍然会被求值，整个卫语句可能还会成功。如果 X >= N orelse N >= 0 的前半部分抛出了异常，那么后半部分就会被跳过，整个卫语句就会失败。然而（总会出现一个“然而”），在卫语句中，只有 andalso 和 orelse 可以嵌套使用。也就是说，(A orelse B) andalso C 是一个有效的卫语句，而(A; B), C 不是。鉴于它们的不同用途，最好的策略就是：在必要时，混合使用它们。



除了能在卫语句中使用比较和布尔操作，还可以使用算术运算符（例如,A*B/C >= 0)和判断数据类型的函数,如 is_integer/1、is_atom/1 等。（我们会在第 4 章中详细介绍这类函数）。

卫语句有一个缺点，考虑到副作用方面的原因，卫语句中不能使用用户自定义函数。Erlang 不是一个纯函数式编程语言（像 Haskell 那样），因为它大量依赖于副作用。你可以进行 I/O 操作，在 actor 之间发送消息，并且可以随时或者在需要时抛出异常。如果在卫语句中使用了一个函数，那么在对函数子句中的卫语句进行测试时，很难判断它是否会打印文本或者捕获重要的错误异常。所以，Erlang 只能选择不信任你。（这种选择可能才是正确的！）

嗯，现在，你所掌握的知识应该足以理解基本的卫语句语法，并且可以看懂它们了。

3.3　if 表达式

if 语句的作用和卫语句类似，和卫语句的语法也一样，但是它在函数子句头之外使用。实际上，if 语句又称为卫模式（guard pattern）。

Erlang 中的 if 和其他大多数语言中见到的 if 不同。和那些 if 语句相比，Erlang 的版本更像个怪物，如果换个别的名字的话，会更容易让人接受。一旦跨入 Erlang 的领土，就要把原先知道的关于 if 的一切知识都丢在门外。

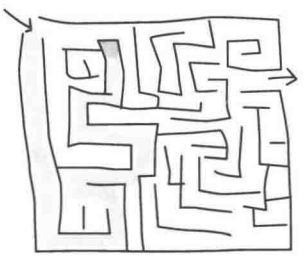
为了说明 if 表达式和卫语句的相似点，新建一个模块 what_the_if.erl，输入下面的内容：

```
-module(what_the_if).
-export([heh_fine/0]).

heh_fine() ->
    if 1 =:= 1 ->
```

```
    works
  end,
  if 1 =:= 2; 1 =:= 1 ->
    works
  end,
❶ if 1 =:= 2, 1 =:= 1 ->
    fails
  end.
```

保存模块，并在 shell 中运行它：

```
1> c(what_the_if).
./what_the_if.erl:12: Warning: no clause will ever match
./what_the_if.erl:12: Warning: the guard for this clause evaluates to 'false'
{ok,what_the_if}
2> what_the_if:heh_fine().
** exception error: no true branch found when evaluating an if expression
in function what_the_if:heh_fine/0
```



编译器给出警告，❶位置上的 if 语句永远不会匹配上，因为它的卫表达式只能求值为 false。记住，在 Erlang 中，每个表达式都必须有返回值，if 表达式也不例外。正因为如此，当 Erlang 无法让卫表达式成功时，就会崩溃，它不能不返回东西（这也解释了为什么 VM 在它抓狂时抛出了一个“no true branch found”的错误异常）。所以，无论如何，都需要增加一个捕获一切的分支。在大多数语言中，这称为 else 分支。在 Erlang 中，会使用 true，像这样：

```
oh_god(N) ->
  if N =:= 2 -> might_succeed;
  true -> always_does   %% 这是 Erlang if 的'else!'
end.
```

现在可以测试这个新函数了（原先那个函数会一直提示警告信息，可以忽略这些信息或者把它们当作一个提醒，告诫我们不要那样做）：

```
3> c(what_the_if).
./what_the_if.erl:12: Warning: no clause will ever match
./what_the_if.erl:12: Warning: the guard for this clause evaluates to 'false'
{ok,what_the_if}
4> what_the_if:oh_god(2).
might_succeed
5> what_the_if:oh_god(3).
always_does
```

下面是另外一个函数，展示了如何在 if 表达式中使用多个卫语句：

```
%% 注意，这个函数最好写成函数头模式匹配的形式！
%% 写成这样只是为了示例需要.
help_me(Animal) ->
  Talk = if Animal == cat -> "meow";
```

```
            Animal == beef -> "mooo";
            Animal == dog -> "bark";
            Animal == tree -> "bark";
            true -> "fgdadfgna"
        end,
    {Animal, "says " ++ Talk ++ "!"}.
```

这个函数再一次说明了任何表达式都必须有返回值。Talk 变量会被绑定上 if 表达式的结果，然后被串联到一个字符串并被放在一个元组中。在阅读这段代码时，可以容易地看出如果没有 true 分支，就会陷入困境，因为 Erlang 中没有像 null 值（如 Lisp 中的 nil，C 中的 NULL 或者 Python 中的 None 之类）这样的东西。

运行一下：

```
6> c(what_the_if).
./what_the_if.erl:12: Warning: no clause will ever match
./what_the_if.erl:12: Warning: the guard for this clause evaluates to 'false'
{ok,what_the_if}
7> what_the_if:help_me(dog).
{dog,"says bark!"}
8> what_the_if:help_me("it hurts!").
{"it hurts!","says fgdadfgna!"}
```

有许多 Erlang 程序员很好奇，为什么选择原子 true，而不是 else 来做控制流——毕竟，else 更为人所熟知，你可能也有此困惑。Richard O'Keefe 在 Erlang 的邮件列表中给出了如下回复，我在此直接引用，因为我无法回答得更好：

> "else" 确实可能更为人熟知，但这并不表示它是一个好的选择。我知道在 Erlang 中可以很容易地用 "; true->" 实现 "else"，但是，数十年的程序编程心理学研究表明这是一个糟糕的主意。我现在开始把

```
if X > Y -> a()
  ; true  -> b()
end
```

替换为

```
if X > Y  -> a()
  ; X =< Y -> b()
end
```

把

```
if X > Y -> a()
  ; X < Y -> b()
  ; true  -> c()
end
```

替换为

```
if X > Y -> a()
  ; X < Y -> b()
```

```
    ; X ==Y -> c()
  end
```

我发现，使用这种方式，在编写代码时稍微有些烦人，但是在阅读代码时能够带来极大的帮助。①

换句话说，无论是 else 还是 true 分支都应该极力避免。写 if 表达式时，如果能够覆盖所有的逻辑分支，而不是依赖于一个包罗一切的分支，通常会更容易理解。

> **注意** what_the_if.erl 文件中那些函数名称所表达出来的恐惧，可以被看成是从其他语言 if 表达式的视角来看 Erlang 语言中这个 if 构造时的感受。在 Erlang 中，if 是一个完美的逻辑构造，只是名字令人困惑。

正如前面提到的那样，在卫表达式中只能使用一组有限的函数(在第4章中会见到其他函数)。该轮到 Erlang 中真正的条件语句粉墨登场，大展宏图了。接下来，我将介绍 case 表达式！

3.4 case ... of 表达式

如果说 if 表达式像卫语句，那么 case ... of 表达式就像整个函数头。你可以对函数的每个参数使用复杂的模式匹配，还可以使用卫语句。

作为示例，我们编写一个集合（一组不重复的值）的 insert 函数，集合用一个无序列表表示。从效率角度来看，这可能是一个非常糟糕的实现，但是，我们目前关注的是语法。创建一个名为 cases.erl 的文件，并输入如下代码：

```
insert(X,[]) ->
    [X];
insert(X,Set) ->
   case lists:member(X,Set) of
      true -> Set;
      false -> [X|Set]
   end.
```

如果传入的参数是一个空集合（列表）和一个要加入的数据项 X，那么代码会返回一个只含有 X 的列表。否则，函数 lists:member/2 会去检查数据项 X 是否已经包含在列表中。如果是，就返回 true；否则，返回 false。如果集合中已经包含元素 X，就无需更改列表；否则，将 X 添加到列表头上。

在本例中，使用的模式匹配非常简单。然而，它可以变得更加复杂，如下例所示（仍然在 case 模块中）：

```
beach(Temperature) ->
   case Temperature of
      {celsius, N} when N >= 20, N =< 45 ->
         'favorable';
```

① http://erlang.org/pipermail/erlang-questions/2009-January/041229.html

```
        {kelvin, N} when N >= 293, N =< 318 ->
            'scientifically favorable';
        {fahrenheit, N} when N >= 68, N =< 113 ->
            'favorable in the US';
        _ ->
            'avoid beach'
    end.
```

这个函数会基于3种不同的温度体系来回答“是否适合去海滩玩耍？”的问题，这3种温度体系分别是摄氏、开氏和华氏。代码中联合使用了模式匹配和卫语句，无论使用何种温度体系，都能给出正确答案。

正如前面指出的那样，case ... of 表达式非常像一组带有卫语句的函数头。实际上，上面的代码也可以这样写：

```
beachf({celsius, N}) when N >= 20, N =< 45 ->
    'favorable';
...
beachf(_) ->
    'avoid beach'.
```

问题来了，在编写条件表达式时，到底该使用 if、case ... of 还是函数呢？

3.5　如何选择

在 if、case ... of 和函数这3个表达式中进行选择确实是个难题。函数调用和 case ... of 之间的差别很小。事实上，它们的底层表示是完全一样的，实际的性能开销也相同。不过，当需要对多个参数求值时，它们之间有一个明显的不同。例如，function(A,B)-> ... 可以有匹配到 A 和 B 的卫语句和值，但是要写成 case 表达式，却需要费点心思，要写成这样：

```
case {A,B} of
    Pattern Guards -> ...
end.
```

这种写法看上去有些奇怪。所以，对于类似的情况，使用函数调用会更加合适一些。另一方面，在前面编写的 insert/2 函数中，使用 case 表达式就比通过直接的函数调用在简单的 true 或者 false 分支之间进行选择要整洁多了。

既然说 case 表达式和函数已经足够灵活，甚至通过卫语句就可以完成 if 表达式的功能，那为什么还要使用 if 呢？if 背后的哲学非常简单：它的存在提供了一种卫语句的快捷使用方法，可以独立于整个模式匹配部分使用。

当然，前面所讲的基本上都是个人偏好，并没有一个放之四海而皆准的答案。实际上，Erlang 社区也会经常争论这个主题。只要写出来的代码容易理解，就没有人会因为你选择的方法而痛打你的。正如 wiki 的发明者 Ward Cunningham 曾经说过的那样：“整洁的代码就是那些和自己的期望完全一致的代码。”

第 4 章

类型

现代的函数式语言通常都因具有复杂、精致的类型系统而闻名。类型系统非常强大，可以让程序员非常容易地得到更多的安全性和更高的性能。静态类型系统之间的差别很大——涵盖范围从像 C 和 Java 之类的为编译器提供标注的简单类型系统，一直到使用高级数学概念保证程序永不崩溃的超级复杂类型系统。还有一些类型系统非常原始——完全没有静态类型，只有动态类型。它们不保证任何一段程序的安全性，只在运行时进行检查。

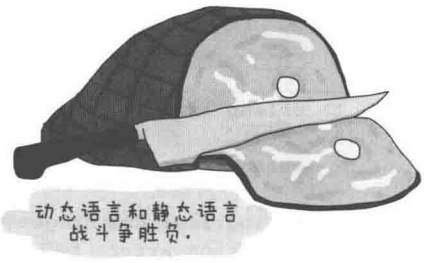
在本章中，我们将学习 Erlang 的类型系统，选择这种类型系统背后的原因，以及对于一个刚接触 Erlang 语言的新手来说会产生怎样的影响。

4.1 动态强类型

当我们在运行第 1 章中的例子，以及在第 2 章和第 3 章中创建模块和函数时，你可能已经注意到，根本不需要指定变量的类型或者函数的类型。在进行模式匹配时，所编写的代码也无需知道它会去匹配什么数据。元组`{X, Y}`可以匹配`{atom, 123}`，也可以匹配`{"A string", <<"binary stuff!">>}`、`{2.0, ["strings","and",atoms]}`，事实上，它可以匹配任何东西。

如果无法匹配，会直接抛出一个错误，但是这只会发生在代码运行时。这是因为 Erlang 是*动态类型语言*。所有错误都在运行时被捕获，在编译代码时，对于可能导致失败的问题，编译器并不总会给出警告，如第 1 章中`5 + llama`的例子。

静态类型和动态类型拥护者之间的主要分歧在于程序的安全性。有些程序员认为，好的静态类型系统能够在代码运行之前捕获绝大多数的错误。正因为如此，一般会认为静态类型语言比动态类型语言更加安全。虽然对于多数其他动态类型语言来说这种观点也许是对的，但是对于 Erlang，这个观点并不成立，并且有真实的证据可以证明这一点。



证明 Erlang 健壮性最好的例子是屡见报端的、由爱立信开发的具有 9 个 9（99.9999999%）可用性的 AXD 301 ATM 交换机，它是由超过 100 万行 Erlang 代码组成。请注意，这不等于说这款基于 Erlang 构建的系统中所有组件都不会失败，而是说，这款通用的交换机系统在 99.9999999% 的时间里都是可用的，包括计划内的停机时间。这样的可用性，部分得益于 Erlang 的基础理念：一个组件的失败不应该影响到整个系统。它对由程序员、硬件故障以及网络故障所造成的错误都进行了仔细的设计考量。语言中也包含了能够将程序分布到不同节点的特性。它能够处理意料之外的错误，并且永远不会停止运行。

简单来讲，尽管绝大多数语言和类型系统都旨在写出没有错误的程序，但是 Erlang 却认为错误肯定会发生，所以在语言中提供了一些特性，基于这些特性可以很容易对错误进行平滑处理，并且不会造成不必要的停机时间。所以，Erlang 的动态类型系统不是程序可靠性和安全性的障碍。这听起来很像是一种预言，不过在后续的章节中，你会看到强有力的证据。

> **注意**　从历史上看，Erlang 选择动态类型的原因很简单。Erlang 语言的早期实现者几乎都来自动态类型语言领域，因此，让 Erlang 成为一门动态类型语言就成了他们的一种自然选择。另外，动态类型也被间接证明是实现代码热加载（在不停止代码运行的情况下升级代码）的最简单方法。事实证明，在一个随时会替换任何部件的系统中进行静态类型检查，其难度要远远高于动态类型。

Erlang 同时还是一个强类型语言。弱类型语言会在不同的数据项之间做隐式的类型转换。例如，如果 Erlang 是弱类型的，就能支持这个操作：`6 = 5 + "1"`。但是，因为 Erlang 是强类型的，所以执行这样的操作会引发不正确参数（bad argument）异常：

```
1> 6 + "1"
** exception error: bad argument in an arithmetic expression
in operator +/2
called as 6 + "1"
```

当然，我们有时候也希望把一种类型的数据转换成另外一种类型。例如，你可能想把常规的字符串转换成二进制字符串以便存储，或者把整数转换成浮点数。Erlang 标准库提供了一组完成这项工作的转换函数。

4.2　类型转换

和许多语言一样，Erlang 也可以通过强制转换的方式改变数据的类型。因为大多数的类型转换都不能用 Erlang 直接实现，所以，这些操作都是 BIF 提供的。所有转换函数的命名都采用 *TypeA_to_TypeB* 这样的形式，都在 `erlang` 模块中实现。下面列出其中的几个：

```
1> erlang:list_to_integer("54").
54
2> erlang:integer_to_list(54).
"54"
3> erlang:list_to_integer("54.32").
** exception error: bad argument
```

```
in function list_to_integer/1
called as list_to_integer("54.32")
4> erlang:list_to_float("54.32").
54.32
5> erlang:atom_to_list(true).
"true"
6> erlang:list_to_binary("hi there").
<<"hi there">>
7> erlang:binary_to_list(<<"hi there">>).
"hi there"
```

我们无意中发现了语言中的一个小缺陷：因为函数命名采用 *Type_to_Type* 这样的形式，所以每当语言中增加一个新的数据类型时，OTP 团队就需要加入一整套 BIF 转换函数！

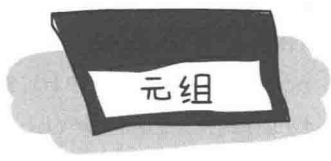
下面是目前提供的转换函数的完整列表：

```
atom_to_binary/2            integer_to_list/1          list_to_integer/2
atom_to_list/1              integer_to_list/2          list_to_pid/1
binary_to_atom/2            iolist_to_atom/1           list_to_tuple/1
binary_to_existing_atom/2   iolist_to_binary/1         pid_to_list/1
binary_to_list/1            list_to_atom/1             port_to_list/1
binary_to_term/1            list_to_binary/1           ref_to_list/1
binary_to_term/2            list_to_bitstring/1        term_to_binary/1
bitstring_to_list/1         list_to_existing_atom/1    term_to_binary/2
float_to_list/1             list_to_float/1            tuple_to_list/1
fun_to_list/1
```

转换函数确实不少。虽然本书的代码中可能不会用到所有这些函数，但是本书会介绍其中的绝大部分数据类型。

> **注意** BIF 函数 `binary_to_term/2` 和 `binary_to_term/1` 对数据反序列化的方式一样。它们之间最大的区别在于函数 `binary_to_term/2` 的第二个参数是一个选项列表。如果传入的是 `[safe]`，那么如果二进制数据中含有未知的原子或者匿名函数，就将不会被解码，因为这可能会耗尽节点的内存或者隐藏着一个安全风险。如果要解码的数据可能是不安全的，那么请使用 `binary_to_term/2` 函数，而不是 `binary_to_term/1` 函数。

4.3 数据类型检测函数



Erlang 的基本数据类型很容易识别：元组用大括号，列表用方括号，字符串用双引号括起来等。所以，我们可以使用模式匹配来强制要求一个确定的数据类型。例如，以列表为参数的函数 `head/1` 只能接受列表作为参数，否则，模式匹配（`[H|_]`）会失败。

然而，因为不能指定范围，在将模式匹配用于数值时会遇到问题。所以，在第 3 章中，在需要检测特定范围的函数中，如温度、年龄之类，使用了卫语句。现在，我们又遇到另外一个问题。如何编写卫语句才能保证只会让针对某个特定类型（如数值、原子或者二进制）

的模式匹配成功？

有一些专门负责检测数据类型的函数。它们接收一个参数，如果参数的数据类型正确，就返回 true，否则，就返回 false。它们是为数不多的、可以在卫表达式中使用的函数中的一部分，也称为类型检测 BIF。下面列出的是 Erlang 中的类型检测 BIF：

```
is_atom/1          is_function/1      is_port/1
is_binary/1        is_function/2      is_record/2
is_bitstring/1     is_integer/1       is_record/3
is_boolean/1       is_list/1          is_reference/1
is_builtin/3       is_number/1        is_tuple/1
is_float/1         is_pid/1
```

和所有其他卫表达式一样，这些函数可以用在任何允许使用卫表达式的地方。

你可能会想，为什么没有提供这样一个函数，它可以直接返回传给它的数据项的类型（就像这样：type_of(X) -> Type）。原因很简单：Erlang 只针对正确的情况编程。你的程序只需要处理那些你知道肯定会发生以及所期望的情况，对于除此以外的其他情况，都应该尽快抛出异常。因此，如果提供一个 type_of(X) 函数，就会怂恿人们在代码中写出条件分支，有点像这样：

```
my_function(Exp) ->
   case type_of(Exp) of
      binary -> Expression1;
      list -> Expression2
   end.
```

上面的代码等效于下面的写法：

```
my_function(Exp) when is_binary(Exp) -> Expression1;
my_function(Exp) when is_list(Exp) -> Expression2.
```

后一种形式更符合 Erlang 语言内在的声明性，我们通过在函数头中指定所期望的数据类型来进行分支处理，而不是根据某个类似 type_of(X) 的函数所返回的数据类型来做不同处理。

> **注意**　在所有可以在卫表达式中使用的函数中，类型检测 BIF 几乎占了大半。其余的也都是 BIF，只是不用于类型测试，包括 abs(Number)、bit_size(Binary)、byte_size(Binary)、element(N, Tuple)、float(Term)、hd(List)、length(List)、node()、node(Pid|Ref|Port)、round(Number)、self()、tl(List)、trunc(Number)和tuple_size(Tuple)。其中，函数node/1和self/0与分布式Erlang以及进程/actor有关。

Erlang 的数据结构看起来似乎很有限，但是，一般来讲，仅用列表和元组就足以构建其他复杂的数据结构了。例如，一棵二叉树的基本结点可以表示成{node, Value, Left, Right}，其中 Left 和 Right 要么是类似的结点，要么是空元组。我还可以把我自己表示成这样：

```
    {person, {name, <<"Fred T-H">>},
    {qualities, ["handsome", "smart", "honest", "objective"]},
    {faults, ["liar"]},
    {skills, ["programming", "bass guitar", "underwater breakdancing"]}}.
```

上面的例子说明，通过嵌套元组和列表，并在其中填充数据，可以得到复杂的数据结构，并且还能构建出操作这些数据结构的函数。

4.4 致静态类型爱好者

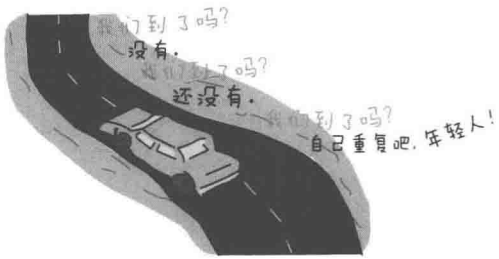
如果你是一个没有静态类型系统就完全无法忍受的程序员，那么我邀请你去看看第 30 章的内容，其中介绍了 Dialyzer。

在第 30 章中，我简单介绍了 Erlang 中用来做静态类型分析的工具，通过这些工具，你可以定义自己的类型并获得更多的安全性。类型完全是可选的，虽然有用，但它们绝不是编写优秀 Erlang 程序所必需的。

第 5 章

递归

有些习惯于命令式和面向对象编程语言的读者可能会觉得奇怪，为什么到现在还没有介绍循环方面的内容呢。这个问题的答案是另一个问题：什么是循环？事实上，函数式编程语言中通常没有类似 `for` 和 `while` 这样的循环结构。相反，函数式程序员会使用一种称为*递归*（recursion）的概念，这也是本章的主要内容。



5.1 递归的工作原理

回想一下在第 1 章中是怎么介绍不变的变量的（要是想不起来，就去重新看一下 1.2.2 节）。我们同样可以借助于一些数学概念和函数来解释递归。

一个简单的数学函数，如求一个数值的阶乘，就是可以用递归表达的好例子。数值 n 的阶乘就是序列 $1 \times 2 \times 3 \times \cdots \times n$ 的乘积，也可以表示成 $n \times (n–1) \times (n–2) \times \cdots \times 1$。在数学表示法中，数字的阶乘可以表示成数字后面跟着一个感叹号（!）。例如，3 的阶乘就是 $3! = 3 \times 2 \times 1 = 6$，4 的阶乘是 $4! = 4 \times 3 \times 2 \times 1 = 24$。这个函数在数学中表示如下：

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n((n-1)!) & \text{if } n > 0 \end{cases}$$

它的意思是，如果 n 的值是 0，就返回结果 1。对其他大于 0 的值，就返回 n 乘以 $n–1$ 的阶乘，逐级展开直到为 1：

$4! = 4 \times 3!$

$4! = 4 \times 3 \times 2!$

$4! = 4 \times 3 \times 2 \times 1!$

$4! = 4 \times 3 \times 2 \times 1 \times 1$

怎么把这个函数从它的数学表示转换成 Erlang 实现呢？这个转换非常简单。仔细看一下上面数学表示中的这些部分：$n!$、1 和 $n((n-1)!)$，然后再观察一下那个 `if` 表达式。它们不就是函数名称（`n!`）、卫语句（`if`）和函数体（`1` 和 `n((n-1)!)`）吗？我们把函数名 `n!` 更改为 `fac(N)` 来满足语法要求，然后就得到了下面的函数：

```
-module(recursive).
-export([fac/1]).

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

这个阶乘函数现在已经完成了！它和上面的数学定义非常相似。借助于模式匹配，我们还可以把这个定义再做些简化：

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

我们通过让函数调用自己实现了循环！你知道吗？“调用自己的函数”就是递归的一种定义。

不过，仅仅让函数调用自己还是不够的。如果函数永远不停地调用自己，毫无疑问，它会一直运行下去。我们还需要一个停止条件，又称为基本情形（base case），也就是一个函数子句，这个函数子句会返回一个值而不是再次调用自己。在上面的例子中，停止条件就是当 n 等于 0 时。此时，我们不再让函数去调用自身，而是返回 1，从而停止了函数的执行。

5.1.1 列表的长度

现在来编写一个更实际些的例子。我们要实现一个可以算出列表中元素个数的函数。从一开始，我们就知道实现中要包含如下 3 方面的内容：

- 基本情形；
- 调用自己的函数；
- 用于测试的列表。

对于绝大多数递归函数来说，我发现先写出基本情形会让实现变得容易一些。在计算列表长度时，最简单的输入是什么呢？毫无疑问，空列表就是最简单的输入，它的长度是 0。所以，让我们在头脑中建立一个关于列表长度的备忘：`[] = 0`。下一个最简单的是长度为 1 的列表：`[_] = 1`。似乎这些已经足够用来写出函数定义了。把它写下来：

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

太棒了！对于长度为 0 或者 1 的列表，这个函数可以计算它的长度。确实太有用了！事实上，它没什么用处，因为它还不是递归函数，现在我们要面对最困难的部分：对函数进行扩展，让它能够在列表长度大于 1 或者 0 的时候调用自己。

我在第 1 章中曾经说过，可以把列表递归定义为`[1 | [2| ... [n| []]]]`。这意味着，可以使用模式`[H|T]`去匹配含有一个或者多个元素的列表，因为长度为 1 的列表可以定义成

[X|[]]，长度为2的列表可以定义成[X|[Y|[]]]。注意，其中的第二个元素本身也是个列表。这样，我们只需要对表的第一个元素计数，然后让函数在第二个元素上调用自己。由于列表中的每个元素的长度都计数为1，所以可以按照下面的方式重写函数：

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

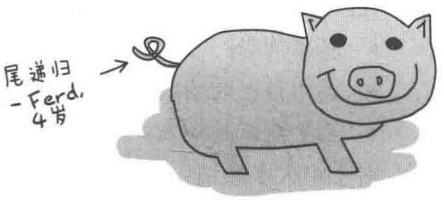
现在，我们自己实现了一个计算列表长度的递归函数。为了演示一下函数运行时的行为，我们拿列表[1,2,3,4]来试一下：

```
len([1,2,3,4]) = len([1 | [2,3,4])
               = 1 + len([2 | [3,4]])
               = 1 + 1 + len([3 | [4]])
               = 1 + 1 + 1 + len([4 | []])
               = 1 + 1 + 1 + 1 + len([])
               = 1 + 1 + 1 + 1 + 0
               = 1 + 1 + 1 + 1
               = 1 + 1 + 2
               = 1 + 3
               = 4
```

结果正确。祝贺你的首个有用的Erlang递归函数完成！

5.1.2　列表长度的尾递归实现

你也许已经注意到，对一个4元素列表来说，函数调用会被展开成一条由5个加法操作构成的链。对于短列表来说，这种做法还是不错的，但是如果列表中有几百万个元素，就会出现问题。你不希望只是为了做个简单的计算，就把几百万个数字放到内存中。这太浪费了。有一种更好的方法，那就是*尾递归*。



尾递归可以把前面的线性过程（操作链的长度会增长到和元素个数相等）转换为迭代过程（操作链的长度不会增加）。要让一个函数变成尾递归，这个函数必须是“孤立的”，这儿需要做一点解释。

前面的调用链之所以会增长，是因为前面部分的结果依赖于后面部分的求值。要计算出1+len(Rest)，需要先计算出len(Rest)。而计算出len(Rest)又需要得到另一个函数调用的结果。这些加法操作会被叠加起来，直至得到最后一个结果，只有在这时才会进行最终结果的计算。尾递归旨在消除这种操作的叠加，方法是在它们出现时就对其进行归约。

为了做到这一点，需要给函数增加一个额外的临时变量参数。我会以阶乘函数为例来阐明这个概念，不过这次会把它定义成尾递归的。前面提到的那个临时变量有时称为*累加器*（accumulator），为了限制调用链的增长，需要在调用发生时就把结果计算出来，累加器就是用来保存这个计算结果的：

```
tail_fac(N) -> tail_fac(N,1).
```

```
tail_fac(0,Acc) -> Acc;
tail_fac(N,Acc) when N > 0 -> tail_fac(N-1,N*Acc).
```

上面代码中，我们定义了 tail_fac/1 和 tail_fac/2 两个函数。这样做是必要的，因为 Erlang 不允许在函数中使用默认参数（不同的参数个数意味着不同的函数），所以只能手工做这个事情。在这个例子中，函数 tail_fac/1 充当了尾递归函数 tail_fac/2 之上的抽象。没有人会对 tail_fac/2 函数中那个隐藏的累加器细节感兴趣，所以我们只会把 tail_fac/1 从模块中导出。当这个函数运行时，会展开成下面这样：

```
tail_fac(4)       = tail_fac(4,1)
tail_fac(4,1)     = tail_fac(4-1, 4*1)
tail_fac(3,4)     = tail_fac(3-1, 3*4)
tail_fac(2,12)    = tail_fac(2-1, 2*12)
tail_fac(1,24)    = tail_fac(1-1, 1*24)
tail_fac(0,24)    = 24
```

看到差别了吗？这个版本中，程序占用的内存不会超过两个数据项，所以对空间的使用是常量的。计算 4 的阶乘和计算 1 000 000 的阶乘所耗费的空间是一样的（如果不考虑 4!在实际表示上比 1 000 000!小很多的话）。

有了将阶乘函数改成尾递归的经验，你也许能够看出如何把这个模式应用在 len/1 函数上。我们要把递归调用变成独立的。如果你喜欢形象一点的说法，可以这样想象：给函数增加一个参数，把那个+1 的部分放到函数调用中去。于是，下面的写法：

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

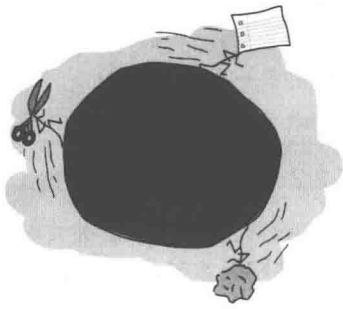
就变成

```
tail_len(L) -> tail_len(L,0).

tail_len([], Acc) -> Acc;
tail_len([_|T], Acc) -> tail_len(T,Acc+1).
```

现在，计算列表长度的函数就是尾递归的了。

5.2 更多递归函数

为了让你习惯递归的思考方式，我们会再多编写几个递归函数。毕竟，递归是 Erlang 中仅有的循环结构（列表推导式除外），所以是需要深刻理解的重要概念之一。同时，在所有其他函数式编程语言中，递归也很有用，所以，请认真学习！



5.2.1 duplicate 函数

我们要写的第一个函数是 duplicate/2。这个函数的第一个参数是个整数，第二个参数可以是任意数据项。这个函数会创建一个包含多个该数据项副本的列表，副本的个数由第一个整数参数指定。

同样，首先考虑基本情形有助于下一步的实现。对 duplicate/2 函数来说，它能做的最简单的事情就是把某个东西重复零次。此时，无论传入什么样的数据项，只需返回一个空列表即可。所有其他情况都要通过调用函数自身设法收敛到这个基本情形。第一个整数参数不允许为负数，因为无法把某个东西复制–n 次。下面就是相应的函数实现：

```
duplicate(0,_) ->
    [];
duplicate(N,Term) when N > 0 ->
    [Term|duplicate(N-1,Term)].
```

一旦实现了基本的递归函数，再将它转换成尾递归形式就容易多了，只需将列表构建移至一个临时变量中：

```
tail_duplicate(N,Term) ->
   tail_duplicate(N,Term,[]).

tail_duplicate(0,_,List) ->
   List;
tail_duplicate(N,Term,List) when N > 0 ->
   tail_duplicate(N-1, Term, [Term|List]).
```

成功！

递归就像一个 while 循环

在我刚开始学习递归时，在尾递归和 while 循环之间进行比较给我带来了很大帮助。前面的 tail_duplicate/2 函数具有 while 循环的所有常见组成部分。如果在一个具有类似 Erlang 语法的虚构语言中提供了一种假想的 while 循环，那么这个函数会被实现成如下的样子：

```
function(N, Term) ->
   while N > 0 ->
      List = [Term|List],
      N = N-1
   end,
   List.
```

注意，这段虚构代码中的所有语法元素 Erlang 中都有，只是位置不同。这就说明，一个完全的尾递归函数和像 while 循环这样的迭代过程非常相似。如果递归让你感到困惑，而你又习惯于 while 循环，那么直接把递归理解成 while 循环可能会有帮助。一定要小心——一旦使用了一段时间 Erlang，你最终会以递归的方式进行思考，这时，如果你又要返回去使用命令式语言，就需要做反方向的思维转换了！

5.2.2　reverse 函数

在本节中，将编写一个 reverse/1 函数，它会反转一个列表，在比较这个函数的递归版本和尾递归版本时，我们会发现另外一个有趣的特性。对这个函数来说，基本情形就是一个空列表，此时，没有什么需要反转的，只需返回一个空列表即可。在所有其他情况中，都要通过调用自身

设法收敛到基本情形，就像 duplicate/2 函数一样。这个函数会通过模式匹配[H|T]来遍历列表，先把列表的剩余部分反转，然后再把 H 放到其后：

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

对于长列表来说，这种做法是一个噩梦。我们不仅堆积了所有的列表追加操作，而且对每个追加操作都要遍历整个列表直到最后一个追加操作完成！对于喜欢形象化的读者，这么多的核对工作可以表示成下图：

```
reverse([1,2,3,4]) = [4]++[3]++[2]++[1]
                      ↑   ↵
                   = [4,3]++[2]++[1]
                      ↑ ↑   ↵
                   = [4,3,2]++[1]
                      ↑ ↑ ↑   ↵
                   = [4,3,2,1]
```

这正是尾递归可以解决的问题。因为我们会使用一个累加器，而且每次都会给它增加一个新的列表头，所以最后得到的列表就是自动反转过的。

首先来看一下代码实现：

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

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

如果用和呈现普通版本类似的方式来呈现这个版本的运行情况，可以得到下面的序列：

```
tail_reverse([1,2,3,4]) = tail_reverse([2,3,4], [1])
                        = tail_reverse([3,4], [2,1])
                        = tail_reverse([4], [3,2,1])
                        = tail_reverse([], [4,3,2,1])
                        = [4,3,2,1]
```

这就表明，为了反转而进行的元素访问次数现在是线性的了。我们不但避免了栈的增长，而且操作效率也更高了！

5.2.3 sublist 函数

我们要实现的另外一个函数是 sublist/2，它接收一个列表 L 和一个整数 N，返回列表 L 中的前 N 个元素。例如，sublist([1,2,3,4,5,6], 3)返回[1,2,3]。

同样的，本例的基本情形就是从列表中获取 0 个元素。但是这次需要小心，因为 sublist/2 函数略微有些不同。这个例子还有第二个基本情形，那就是传入的列表为空时！如果我们不检测空列表的情况，那么在调用 recursive:sublist([1],2)时会抛出一个异常，而此时，我们实际希望得到的结果是[1]。一旦定义好了基本情形，函数的递归部分只需要循环遍历这个列表，不断保存遇到的元素，直到进入某个基本情形，如下所示：

```
sublist(_,0) -> [];
sublist([],_) -> [];
sublist([H|T],N) when N > 0 -> [H|sublist(T,N-1)].
```

使用和之前同样的方式，也可以把它转换成尾递归形式：

```
tail_sublist(L, N) -> tail_sublist(L, N, []).

tail_sublist(_, 0, SubList) -> SubList;
tail_sublist([], _, SubList) -> SubList;
tail_sublist([H|T], N, SubList) when N > 0 ->
   tail_sublist(T, N-1, [H|SubList]).
```

这个函数中有个缺陷——一个致命的缺陷！我们使用了一个列表作为累加器，用法和反转列表函数中的完全一样。如果编译上述函数，那么调用 `tail_sublist ([1,2,3,4,5,6],3)` 不会返回`[1,2,3]`，而是返回`[3,2,1]`。我们唯一要做的就是拿到这个最后结果，自己去反转它。只需要修改 `tail_sublist/2` 函数，所有递归逻辑保持不变：

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

最终结果的顺序正确了。似乎在尾递归调用后再反转列表会浪费时间，应该说这个观点部分正确（因为这么做还是节省了内存）。正是因为这个原因，对于短列表来说，可能会发现普通递归版本的运行速度要快于尾递归版本，但是随着列表规模增大，反转列表占据的时间比重会越来越小。

> **注意**　不要自己去编写 `reverse/1` 函数，应当使用 `lists:rerverse/1` 函数。由于这个函数在尾递归调用中使用得非常频繁，所以 Erlang 的维护和开发人员就决定把它变成一个 BIF。现在，列表反转的速度非常快（这要感谢那些 C 语言函数），反转的劣势也就不再那么突出了。本章其余的代码会继续使用我们自己编写的反转函数，但是学完本章后，你就不应该再使用它了。

5.2.4　zip 函数

为了进一步理解递归，我们再写一个列表拼合函数（zipping function）。这个函数以两个长度相同的列表为参数，把它们合并成一个元组列表，每个元组中有两个数据项。我们要编写的 `zip/2` 函数的行为如下：

```
1> recursive:zip([a,b,c],[1,2,3]).
[{a,1},{b,2},{c,3}]
```

考虑到两个列表参数的长度要一样，所以基本情形就是拼合两个空列表：

```
zip([],[]) -> [];
zip([X|Xs],[Y|Ys]) -> [{X,Y}|zip(Xs,Ys)].
```

不过，如果希望限制条件宽松一点，可以在其中一个列表为空时就结束。此时，应该有两个基本情形：

```
lenient_zip([],_) -> [];
lenient_zip(_,[]) -> [];
lenient_zip([X|Xs],[Y|Ys]) -> [{X,Y}|lenient_zip(Xs,Ys)].
```

注意，对于所有的基本情形，函数的递归部分都是相同的。

尾递归优化

这里编写的尾递归不会引起内存消耗的增长，因为当 VM 看到一个函数在尾部（函数中最后一个被求值的表达式）调用了自身，它会清除当前的栈帧。这称为尾递归优化（tail recursion optimization，TRO），它是一个更为一般的优化方法的特例，这种更为一般的优化方法称为尾调用优化（last call optimization，LCO）。

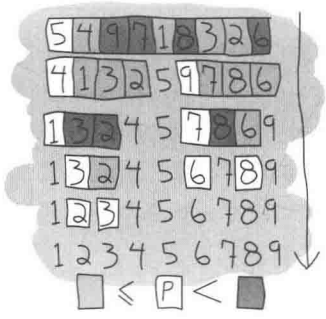
当函数体中的最后一个被求值的表达式是另外一个函数调用时，就会进行 LCO。在进行 LCO 时，和 TRO 一样，Erlang VM 会避免存储栈帧。因此，尾递归也适用于多个函数的情况。例如，函数调用链 `a() -> b(). b() -> c(). c() -> a().` 实际上导致了一个无限循环，但不会耗尽内存，因为 LCO 避免了栈溢出。这条原则，再加上累加器的使用，成就了尾递归的可用性。

为了确保你完全理解了尾递归函数的构建方式，我建议你尝试自己编写一下 `zip/2` 和 `lenient_zip/2` 的尾递归版本。尾递归是构建大型应用的核心概念之一，应用的主循环就是以尾递归的方式编写的。

如果想核对一下自己的答案，可以参见我在 recursive.erl 中的实现（http:/learnyousomeerlang.com/static/erlang/recursive.erl），特别是 `tail_zip/2` 和 `tail_ lenient_ zip/3` 这两个函数。

5.2.5 快速排序

为了保证你理解了递归和尾递归的含义，我们再进一步编写一个更加复杂的例子：快速排序（quicksort）。是的，就是那个“嘿，看，函数式代码可以写得如此简短”的经典范例。



快速排序的简单实现是，取列表中的第一个元素，把它作为支点，然后将小于等于它的元素放到一个新列表中，再将大于它的元素放到另外一个列表中。接下来对这两个列表做同样的操作，直到每个列表变得越来越小。这个过程会一直持续进行，直到没有任何东西需要排序，只剩下一个空列表，也就是我们的基本情形。之所以说这是个简单的实现，是因为更聪明的快速排序实现会尝试挑选最佳的支点，来让整个排序更快些。这不是本例关心的重点。

我们需要两个函数来完成快速排序：一个分割函数，用来将列表分成小于支点和大于支点两个部分，另外一个函数将分割函数应用于分割后的每个新列表上，并将它们拼接在一起。我们先来编写拼接函数（可以写在 recursive.erl 文件中）：

```
quicksort([]) -> [];
quicksort([Pivot|Rest]) ->
    {Smaller, Larger} = partition(Pivot,Rest,[],[]),
    quicksort(Smaller) ++ [Pivot] ++ quicksort(Larger).
```

上述代码中展示了基本情形，原列表被 paritition 函数分割成大、小两部分的列表，以及前后追加了快速排序过的大小两个列表的支点。这个函数完成了列表的组装。

下面，我们来编写分割函数：

```
partition(_,[], Smaller, Larger) -> {Smaller, Larger};
partition(Pivot, [H|T], Smaller, Larger) ->
    if  H =< Pivot -> partition(Pivot, T, [H|Smaller], Larger);
        H > Pivot -> partition(Pivot, T, Smaller, [H|Larger])
    end.
```

好了，现在可以运行快速排序函数了。

如果你在网上搜索过 Erlang 的样例代码，那么可能看到过另外一种快速排序实现方法——它更加简单和易读，不过使用了列表推导式。上述代码中比较容易替换的是创建新列表的代码，也就是 partition/4 函数：

```
lc_quicksort([]) -> [];
lc_quicksort([Pivot|Rest]) ->
    lc_quicksort([Smaller || Smaller <- Rest, Smaller =< Pivot])
    ++ [Pivot] ++
    lc_quicksort([Larger || Larger <- Rest, Larger > Pivot]).
```

这个版本更易读，但是作为交换，在把列表分成两部分时，它必须两次遍历列表。这是一场代码清晰性和性能之间的较量，不过真正的失败者是你，因为已经存在 lists:sort/1 函数了。请直接使用 lists:sort/1 函数。

保持冷静

所有这种简洁性都只是为了教学目的，但是在性能方面存在问题。许多函数式编程教程中对此根本没有提及！首先，这里展示的两种快速排序实现都需要多次处理和支点值相等的元素。我们可以考虑返回 3 个列表，分别是小于、大于以及等于支点的元素列表，这样程序会更加高效。

另外一个问题是，在把所有分割后的列表和支点合并在一起时，会多次遍历这些列表。如果把列表分割成 3 个部分，那么在做列表连接拼接时，是能够减少一些这方面的开销的。如果你对具体实现感兴趣，可以把 recursive.erl 文件中的最后一个函数（bestest_qsort/1）作为参考示例。

所有这些快速排序版本有一个共同的优点，就是它们可以应用于任意数据类型的列表，甚至是元组列表。用各种数据类型尝试一下，就会发现工作正常。

5.3 不仅仅是列表

至此，你可能会认为 Erlang 中的递归主要用于列表处理。虽然列表确实是一个可以被递归定义的数据结构的好例子，但是除了列表处理外，递归还有很多其他用途。考虑到多样性，在本节中，我们以二叉树的构建和数据访问为例。

首先，必须给出树的定义。在本例中，一棵树就是一组自顶向下的结点。结点是一个元组，里面含有一个键、一个和该键关联的值，以及另外两个结点。在这两个结点中，我们要求其中一个结点的键比包含它们的结点的键小，另外一个结点的键比包含它们的结点的键大。这就是递归！一棵树就是一个包含一组结点的结点，每个被包含的结点又包含了另外一组结点。这个过程不可能一直持续下去（我们没有无穷多的数据要存储），所以结点中也可以包含空结点。

要表示出结点，元组是一个合适的数据结构。对于我们的实现来说，可以把元组定义成`{node, {Key, Value, Smaller, Larger}}`（带标记的元组！），其中 `Smaller` 和 `Larger` 可以是类似的结点，也可以是一个空结点（`{node, nil}`）。这就是本例中最复杂的概念了。

我们来编写一个模块，实现这个非常简单的树。第一个函数是 `empty/0`，它返回一个空结点。空结点是一棵新树的开始结点，也称为根结点（root）。

```
-module(tree).
-export([empty/0, insert/3, lookup/2]).

empty() -> {node, 'nil'}.
```

通过这个函数，以及后续以同样的方法把结点的表示全部封装起来，可以把树的实现隐藏起来，这样，树的使用者就无需知道树的构建细节了。所有的实现细节都被包含在这个单独的模块中。你可以随时更改结点的表示方式，不会对外部依赖代码造成任何影响。

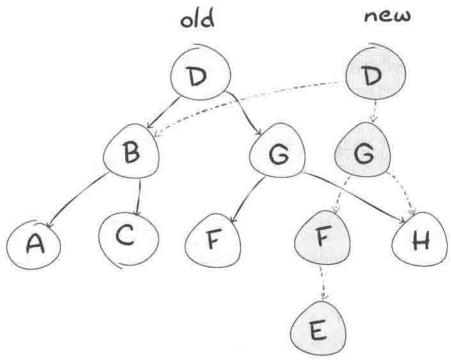
要想向树中增加内容，首先必须要弄清楚如何递归地浏览一棵树。我们仍旧采用在之前的递归示例中采用的方法：先来找出基本情形。

因为一棵空树就是一个空结点，所以从逻辑上讲，基本情形就是一个空结点。因此，每当遇到一个空结点时，就说明这是可以增加新键/值的地方。对于其他结点，代码都必须顺着树游走，设法找到一个可以放置内容的空结点。

要想从根结点开始找到一个空结点，必须要利用这样一个事实，那就是 `Smaller` 和 `Larger` 结点的存在，使得我们可以通过比较要插入的新键和当前结点中的键进行树的游历。如果新键比当前结点的键小，我们会尝试在 `Smaller` 中寻找空结点，如果新键比当前结点的键大，我们会在 `Larger` 中查找。不过，还有第三种情况：如果新键和当前结点的键相等时该怎么办呢？有两种选择：让程序失败或者用新的键替换原先的那个。我们选择后者。把所有这些逻辑放到函数中，实现如下：

```
insert(Key, Val, {node, 'nil'}) ->
    {node, {Key, Val, {node, 'nil'}, {node, 'nil'}}};
insert(NewKey, NewVal, {node, {Key, Val, Smaller, Larger}}) when NewKey < Key ->
    {node, {Key, Val, insert(NewKey, NewVal, Smaller), Larger}};
insert(NewKey, NewVal, {node, {Key, Val, Smaller, Larger}}) when NewKey > Key ->
    {node, {Key, Val, Smaller, insert(NewKey, NewVal, Larger)}};
insert(Key, Val, {node, {Key, _, Smaller, Larger}}) ->
    {node, {Key, Val, Smaller, Larger}}.
```

注意，insert 函数返回的是一棵完整的新树。这是只允许赋值一次的函数式语言的典型特征。虽然这种做法被认为不太高效，但是在更新一棵树或者向树中增加元素时，通常只需要修改受到变化影响的结点。其他结点都会在新旧两个版本的树之间共享，极大地降低了内存开销。在下图中，“E”结点被加入树中，它的所有父结点都需要更新。然而，树的整个左边部分（从“B”结点起始）并没有改变，可以在两个版本间共享。对函数式程序员来说，这是个众所周知的概念，称为*持久化数据结构*（persistent data structure）。



在本例中，还有最后一个函数 lookup/2 需要编写，给定一个键，这个函数可以从树中找到和这个键关联的值。此处的逻辑和向树中增加新内容的逻辑非常类似：逐步遍历每个结点，检查要查找的键是等于、小于还是大于当前结点的键。有两个基本情形：一个是结点为空（说明树中没有这个键），还有一个是在树中找到了这个键。因为我们不希望在查找某个不存在的键时程序崩溃，所以会返回原子 undefined。否则，返回{ok, Value}。如果只返回 Value，那么当结点的值就是原子 undefined，此时则无法在返回了正确的值和查找失败之间进行区分。通过把成功情况包装在一个这样的元组中，就可以容易地区分这两种情况了。函数实现如下：

```
lookup(_, {node, 'nil'}) ->
    undefined;
lookup(Key, {node, {Key, Val, _, _}}) ->
    {ok, Val};
lookup(Key, {node, {NodeKey, _, Smaller, _}}) when Key < NodeKey ->
    lookup(Key, Smaller);
lookup(Key, {node, {_, _, _, Larger}}) ->
    lookup(Key, Larger).
```

实现完成了。我们用一个小型的电子邮件地址簿来测试一下这个程序。编译文件，启动 shell：

```
1> T1 = tree:insert("Jim Woodland", "jim.woodland@gmail.com", tree:empty()).
{node,{"Jim Woodland","jim.woodland@gmail.com",
       {node,nil},
       {node,nil}}}
2> T2 = tree:insert("Mark Anderson", "i.am.a@hotmail.com", T1).
{node,{"Jim Woodland","jim.woodland@gmail.com",
       {node,nil},
       {node,{"Mark Anderson","i.am.a@hotmail.com",
              {node,nil},
              {node,nil}}}}}
3> Addresses = tree:insert("Anita Bath", "abath@someuni.edu",
3>     tree:insert("Kevin Robert", "myfairy@yahoo.com",
3>         tree:insert("Wilson Longbrow", "longwil@gmail.com", T2))).
{node,{"Jim Woodland","jim.woodland@gmail.com",
       {node,{"Anita Bath","abath@someuni.edu",
              {node,nil},
              {node,nil}}},
       {node,{"Mark Anderson","i.am.a@hotmail.com",
              {node,{"Kevin Robert","myfairy@yahoo.com",
                     {node,nil},
                     {node,nil}}},
              {node,{"Wilson Longbrow","longwil@gmail.com",
                     {node,nil},
                     {node,nil}}}}}}}
```

现在可以用它来查找电子邮件地址了：

```
4> tree:lookup("Anita Bath", Addresses).
{ok, "abath@someuni.edu"}
5> tree:lookup("Jacques Requin", Addresses).
undefined
```

上面的显示表明，我们这个实用的地址簿示例中使用的并不是列表，而是其他递归数据结构！

现在，可以去放松一下了……

> **注意** 我们实现的树非常简单。它不支持一些常见的操作，例如删除结点，或者重新平衡使得后续查找更快。如果你对如何实现这些操作感兴趣或者想了解一下，去学习一下 Erlang 中 gb_trees 模块的实现（你的 Erlang 安装路径为/lib/stdlib/src/gb_trees.erl）是个不错的选择。当自己的代码中需要树时，应当使用 gb_trees 模块，不要重新制造轮子。

5.4 递归思维

如果你已经理解了本章中的所有内容，递归思维对你来说可能就是一件自然的事情了。递归与其命令式语言中的对应物相比（通常是 while 或者 for 循环），一个不同之处在于，它不是采用逐步执行的方式（“先做这个，然后做那个，再做这个，最后完成”），递归方法是更加声明性

的（“如果输入是这个，这么做，否则，那么做”）。借助于函数头中的模式匹配，这个区别就变得更加明显。

如果你还不能领会递归的工作机制，阅读一下这段话，可能会对你有所帮助。

言归正传，递归加上模式匹配，往往是编写简洁、易读算法的最佳选择。把问题的每个部分都分解到不同的函数中，直到不能再被简化为止，这样，算法本身的工作就只是把这些简短函数的结果组装在一起（有点像我们在快速排序中的做法）。这种抽象思维也适用于一般的循环实现，但是我相信用递归实现起来会更容易。看法因人而异吧。

女士们、先生们，请听作者和他自己的对话

自己：嗯，我想我已经理解递归了。我明白递归的声明性特点了。我知道它的思想来源于数学，就像不变的变量那样。我也知道，在有些情况下，使用递归会更容易一些。还有别的吗？

作者：它遵循一个规范的模式。找出基本情形，把它们写下来。然后，所有其他情形都应当设法收敛到这些基本情形，从而得到结果。按照这种方式来写递归函数就相当容易了。

自己：哦，我明白了。你已经重复多次了。循环也可以做同样的事情。

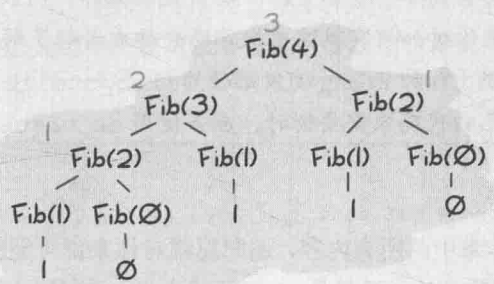
作者：是的，循环也可以。我没有否认这一点。

自己：还有一件事，如果说非尾递归版本比不上尾递归版本，那为什么还要写这些非尾递归的函数呢？

作者：嗯，那仅仅是因为这种方式更易于理解。从更漂亮和易理解的普通递归开始，再到理论上更加高效的尾递归，似乎是个全面学习的好方法。

自己：是的，也就是说，除了教学目的，普通递归没有什么用处。我明白了。

作者：也不尽然。在实践中，你会发现尾递归和普通递归在性能上的差别很小。那些会永远循环的函数需要小心对待，例如主循环。还有一些类型的函数，如果不把它们改成尾递归的形式，就会产生非常大的调用栈，会变慢，还可能会早早崩溃。斐波那契（Fibonacci）函数就是这类函数的最好例子，如果不把它写成迭代或者尾递归的方式，栈就以指数级增长。你应该对代码进行性能分析，看看慢在哪里，然后修正它。



自己：但是循环就是迭代的呀，所以不会出现这个问题。

作者：是的，只是……只是……我们漂亮的 Erlang……

自己：嗯，也没有那么好，对吧？就是因为 Erlang 中没有 while 或者 for，才需要学这么多东西。非常感谢你。不过，我还是准备用 C 语言来编写烤箱程序了！

作者：别太着急！函数式编程语言中还有一些其他宝藏！如果我们都能发现递归函数之间存在的一些基本共同点（累加器、结束时反转列表等），那么那些聪明人肯定早就已经发现大量的共同点和模式了。事实上，他们发现了相当多的公共模式，并且把使用最频繁的操作提炼出来放到库中。所以，你几乎很少需要亲自去写递归函数。如果你坚持学习，就会知道这些抽象是如何被构建出来的。不过，要达成这个目的，我们还需要更多的力量。下面让我来给你讲讲高阶函数……

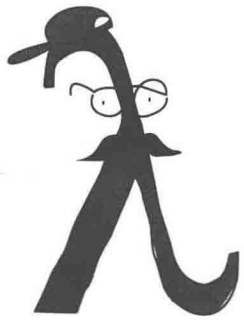
第 6 章

高阶函数

所有函数式语言都具有一个重要的特性：把自定义函数作为参数传递给另外一个函数。这个函数参数会被绑定到一个变量上，在函数内部可以像使用其他变量一样使用这个变量。如果一个函数的参数是以这种方式传过来的其他函数，则称之为高阶函数（higher-order function）。在本章中，你将会看到，高阶函数是一种强有力的抽象手段，也是 Erlang 提供的众多出色工具中需要熟练掌握的一个。

6.1 一切都是函数

函数可以携带并且可以作为参数传给高阶函数的思想起源于数学，主要来自 λ 演算。在本质上，纯 λ 演算中的所有东西都是函数——就连数字、操作符和列表也都是函数。因为所有东西都被表示成函数，所以，函数必须能接收其他函数作为参数，还必须能用函数来操作函数！（如果你想对 λ 演算有个简单、清楚的了解，可以参见维基百科上的相关词条内容。）



这个思想也许有些难以理解，我们先来看一个例子（这个例子和真正的 λ 演算没有任何关系，但是它阐明了要点）。

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

one() -> 1.
two() -> 2.

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

打开 Erlang shell，编译模块，运行下面的指令：

```
1> c(hhfuns).
```

```
{ok, hhfuns}
2> hhfuns:add(one,two).
** exception error: bad function one
in function hhfuns:add/2
3> hhfuns:add(1,2).
** exception error: bad function 1
in function hhfuns:add/2
4> hhfuns:add(fun hhfuns:one/0, fun hhfuns:two/0).
3
```

困惑吗？一旦你知道了这个程序的工作原理（难道不总是这样吗？），就不太会感到困惑了。在第 2 行中，原子 one 和 two 被传给了 add/2 函数，然后把它们当成函数名使用（X() + Y()）。如果只写函数名字，不写参数列表，那么函数名字会被解释成原子，而原子不可能是函数，所以调用失败。第 3 行中的表达式抛出异常也是这个原因：值 1 和 2 也不能当作函数调用，而函数才是我们真正需要的！

要解决这个问题，需要在语言中加入一个新的表示法，用于在模块外部传递函数。Module:Function/Arity 表示法正是用于此目的的：它告诉 VM 去使用这个指定的函数，并把这个函数绑定到一个变量上。

那么，以这种方式使用函数会有哪些好处呢？嗯，一个小例子可能有助于回答这个问题。我们会在 hhfuns 模块中增加一些函数，这些函数递归地遍历一个列表，将列表中的每个整数加 1 或减 1。

```
increment([]) -> [];
increment([H|T]) -> [H+1|increment(T)].

decrement([]) -> [];
decrement([H|T]) -> [H-1|decrement(T)].
```

你观察到这两个函数的相似之处了吗？它们做的事情大致相同：循环遍历列表，在每个元素上应用一个函数（+或者-），然后再调用自身。代码几乎完全一样，只有被应用的函数和递归调用是不同的。像上面这种针对列表的递归调用，它们的核心流程总是一样的。所以，我们将流程中相同的部分抽取出来放在一个单独的函数（map/2）中，这个函数接收另外一个函数作为参数。

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

incr(X) -> X + 1.
decr(X) -> X - 1.
```

现在，在 shell 中测试一下这个函数。

```
1> c(hhfuns).
{ok, hhfuns}
2> L = [1,2,3,4,5].
[1,2,3,4,5]
3> hhfuns:increment(L).
[2,3,4,5,6]
```

```
4> hhfuns:decrement(L).
[0,1,2,3,4]
5> hhfuns:map(fun hhfuns:incr/1, L).
[2,3,4,5,6]
6> hhfuns:map(fun hhfuns:decr/1, L).
[0,1,2,3,4]
```

可以看出，两者的结果是完全相同的，不同的是我们刚刚创造了一个聪明的抽象！每当我们想把一个函数应用于列表中的每个元素上时，只需以这个函数为参数调用 map/2 函数即可。然而，对于每个希望作为参数传入 map/2 的函数，都要把它放在一个模块中，给它取名，从模块中导出，然后再编译等有些麻烦。事实上，这种做法明显不实用。我们需要的是无需这些麻烦步骤就可以直接定义使用的函数，这种函数我们会在下一节介绍。

6.2 匿名函数

匿名函数（anonymous functions）又称 funs，是在行间定义的一种特殊函数，无需给其取名，从而解决了函数作为参数的那些麻烦问题。正常函数能做的事情，匿名函数基本上也都可以完成，除了不能递归调用自己。（它们是匿名的，怎么可能做到这点呢？）

匿名函数的语法如下：

```
fun(Args1) ->
    Expression1, Exp2,..., ExpN;
   (Args2) ->
    Expression1, Exp2,..., ExpN;
   (Args3) ->
    Expression1, Exp2,..., ExpN
end
```

下面是一个使用匿名函数的例子：

```
7> Fn = fun() -> a end.
#Fun<erl_eval.20.67289768>
8> Fn().
a
9> hhfuns:map(fun(X) -> X + 1 end, L).
[2,3,4,5,6]
10> hhfuns:map(fun(X) -> X - 1 end, L).
[0,1,2,3,4]
```

上面的例子反应了人们这么喜欢函数式编程的一个原因：可以对非常低层次的代码进行抽象。因此，可以完全忽略像循环这种基础概念，从而聚焦在做什么上，而不是怎么做。

6.2.1 匿名函数的其他用途

匿名函数非常适合做此类抽象，但是它们还有一些隐藏的威力。我们来看另外一个例子：

```
11> PrepareAlarm = fun(Room) ->
11>       io:format("Alarm set in ~s.~n",[Room]),
11>       fun() -> io:format("Alarm tripped in ~s! Call Batman!~n",[Room]) end
11>  end.
```

```
#Fun<erl_eval.20.67289768>
12> AlarmReady = PrepareAlarm("bathroom").
Alarm set in bathroom.
#Fun<erl_eval.6.13229925>
13> AlarmReady().
Alarm tripped in bathroom! Call Batman!
ok
```

等等，蝙蝠侠！这里到底发生了什么？嗯，首先，我们定义了一个匿名函数，并把它赋给 ParseAlarm。这个函数并没有运行。仅当调用 PrepareAlarm("bathroom")时，才会执行这个函数。就在那时，调用了函数 io:format/2，它打印出了“Alarm set in”文本。函数的第二个表达式（另一个匿名函数）被返回给调用者，并赋值给 AlarmReady。注意，在返回的这个匿名函数中，Room 变量的值来自于它的“父”函数（PrepareAlarm）。这涉及一个称作闭包（closure）的概念。不过，在介绍闭包之前，我们需要先弄清楚什么是作用域。

6.2.2 函数的作用域和闭包

可以把函数的作用域想象成存放所有变量及这些变量对应的值的地方。例如，在函数 base(A)-> B = A + 1 中，A 和 B 都是 base/1 函数作用域的一部分。这意味着，在 base/1 函数中的任何地方，都可以引用 A 和 B，并得到它们所绑定的值。当我说“任何地方”时，我不是在开玩笑，其中也包括匿名函数。

```
base(A) ->
    B = A + 1,
    F = fun() -> A * B end,
    F().
```

在这个例子中，B 和 A 仍然在 base/1 的作用域范围之内，所以函数 F 可以访问到它们。这是因为，F 继承了 base/1 的作用域。和现实生活中的大多数继承一样，父母无法继承孩子们所拥有的东西。

```
base(A) ->
    B = A + 1,
    F = fun() -> C = A * B end,
    F(),
    C.
```

在这个函数版本中，B 仍然等于 A+1，F 仍然可以顺利执行。然而，变量 C 只存在于 F 中的那个匿名函数的作用域中。最后一行，当 base/1 函数试图访问变量 C 的值时，会发现它只是一个未绑定变量。事实上，如果你尝试编译这个函数，编译器就会给出错误警告。继承权只能是单向的。

有必要提醒一下，不管匿名函数在哪里，这个被继承的作用域会一直跟随着它，即使把这个匿名函数传递给另外一个函数。下面是一个例子：

```
a() ->
    Secret = "pony",
```

```
    fun() -> Secret end.

b(F) ->
    "a/0's password is "++F().
```

编译、运行这个函数。

```
14> c(hhfuns).
{ok, hhfuns}
15> hhfuns:b(hhfuns:a()).
"a/0's password is pony"
```

谁泄露了a/0函数的密码？没错，就是a/0函数。由于定义在a/0中的匿名函数继承了a/0的作用域，所以，根据前面的解释，当在b/1中执行这个匿名函数时，它仍然携带着a/0的作用域。这个特性非常有用，因为它能够让我们把参数及其值带出它们的原始上下文，原来的整个上下文都不再需要了（前一节中的蝙蝠侠例子正好说明这一点）。

如果函数具有多个参数，但是其中有一个参数一直保持不变，那么此时就很适合使用匿名函数来携带状态了，如下面的例子所示：

```
16> math:pow(5,2).
25.0
17> Base = 2.
2
18> PowerOfTwo = fun(X) -> math:pow(Base,X) end.
#Fun<erl_eval.6.13229925>
17> hhfuns:map(PowerOfTwo, [1,2,3,4]).
[2.0,4.0,8.0,16.0]
```

通过把对math:pow/2函数的调用包装在一个匿名函数中，并把Base变量绑定到这个匿名函数的作用域里，就能让hhfuns:map/2函数中的每个PowerOfTwo调用，在进行以列表中的整数值为幂的计算时，所针对的都是Base变量。

编写匿名函数时，如果想对作用域进行重定义，就可能会落入一个小陷阱，如下所示：

```
base() ->
    A = 1,
    (fun() -> A = 2 end)().
```

这段代码定义了一个匿名函数，接着就运行了它。因为匿名函数继承了base/0的作用域，所以试图用=操作符去比较2和变量A（已绑定值1），是肯定会失败的。不过，如果我们在内嵌函数的函数头中进行定义，那么就可以重新定义这个变量了：

```
base() ->
    A = 1,
    (fun(A) -> A = 2 end)(2).
```

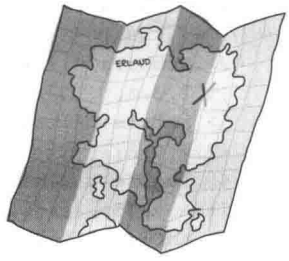
这种写法是合法的。如果对它进行编译，会得到一个变量被屏蔽的警告消息："Warning: variable 'A' shadowed in 'fun'" 屏蔽（shadowing）这个术语用来描述这样一种行为：定义了一个和父作用域中的某个变量同名的新名字。这里的警告是为了防止某些错误（通常也的确如此），

所以出现这种警告时，可以考虑对变量进行重命名。

介绍完了作用域，再来看看闭包。闭包指的就是可以让函数引用到它所携带的某些环境（作用域中的值部分）。换句话说，当匿名函数、作用域的概念以及可以携带变量的能力结合在一起时，闭包就出现了。

现在先把匿名函数理论放在一边，我们来学习一些其他的通用抽象，通过它们就不用编写递归函数了，这可是我在第 5 章结尾就承诺过的。

6.3 映射、过滤器、折叠以及其他



在本章的开始，我们简单介绍了一下如何对两个相似的函数进行抽象，从而得到 map/2 函数。

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

这个函数可被用于任何列表，只要我们想对其中的每个元素做同样的操作。另外，还可以从其他一些经常出现的递归函数中构建出许多类似的抽象。

6.3.1 过滤器

首先，我们来看一下过滤器（filter）。考虑下面的两个函数：

```
%% 只保留偶数
even(L) -> lists:reverse(even(L,[])).

even([], Acc) -> Acc;
even([H|T], Acc) when H rem 2 == 0 ->
    even(T, [H|Acc]);
even([_|T], Acc) ->
    even(T, Acc).

%% 只保留年龄大于 60 的男性
old_men(L) -> lists:reverse(old_men(L,[])).

old_men([], Acc) -> Acc;
old_men([Person = {male, Age}|People], Acc) when Age > 60 ->
    old_men(People, [Person|Acc]);
old_men([_|People], Acc) ->
    old_men(People, Acc).
```

第一个函数接收一个整数列表，只返回其中的偶数。第二个函数遍历一个形如{Gender,Age}的人员列表，只返回年龄超过 60 的男性。

和前面那个例子相比，找出这两个函数之间的相似性要困难一些，但是我们还是能发现一些共同点。这两个函数都是对列表操作，并且有着同样的目标：保留列表中通过测试（又称谓词）的元素，丢弃其他的元素。基于这个归纳，我们可以把需要的所有有用信息提取出来，并且对它们进行抽象，如下所示：

```
filter(Pred, L) -> lists:reverse(filter(Pred, L,[])).

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

要使用这个 filter 函数，只需给它传入一个谓词。编译、运行 hhfuns 模块。

```
1> c(hhfuns).
{ok, hhfuns}
2> Numbers = lists:seq(1,10).
[1,2,3,4,5,6,7,8,9,10]
3> hhfuns:filter(fun(X) -> X rem 2 == 0 end, Numbers).
[2,4,6,8,10]
4> People = [{male,45},{female,67},{male,66},{female,12},{unkown,174},{male,74}].
[{male,45},{female,67},{male,66},{female,12},{unkown,174},{male,74}]
5> hhfuns:filter(fun({Gender,Age}) -> Gender == male andalso Age > 60 end, People).
[{male,66},{male,74}]
```

通过这两个例子可以看出，借助于 filter/2 函数，程序员只需关心谓词的定义和列表就可以了，无需考虑诸如循环遍历列表以及丢弃不需要的数据之类的事情。这也是函数式代码抽象要达成的一个重要目标：提炼出一直不变的部分，让程序员提供出变化的部分。

6.3.2　折叠一切

在第 5 章中，我们看到了另外一种列表递归操作，其中把某个操作依次作用于列表的每个元素，最后把所有元素归约成一个单一值。这种操作称为折叠（fold），可以用它来简化下面这些函数：

```
%% 找出列表中的最大值
max([H|T]) -> max2(T, H).

max2([], Max) -> Max;
max2([H|T], Max) when H > Max -> max2(T, H);
max2([_|T], Max) -> max2(T, Max).

%% 找出列表中的最小值
min([H|T]) -> min2(T,H).

min2([], Min) -> Min;
min2([H|T], Min) when H < Min -> min2(T,H);
min2([_|T], Min) -> min2(T, Min).

%% 计算列表所有元素的和
sum(L) -> sum(L,0).

sum([], Sum) -> Sum;
sum([H|T], Sum) -> sum(T, H+Sum).
```

要了解 fold 函数的使用方式，我们需要找出上面 3 个函数所做操作之间的所有共同点。正如前面提到的那样，这 3 个函数都是将一个列表归约成一个单一值。因此，可以认为 fold 函数只需要对列表进行遍历，同时保留一个数据项即可——无需构建任何其他列表。我们也无需考虑卫语句，因为只有某些函数用到了它们，并非所有函数。卫语句需要包含在传递给 fold 的函数中。在这一点上，fold 函数看起来很像 sum。

这 3 个函数之间还有一个不那么明显的共同点：每个函数在开始计算时，都需要一个初始值。在 sum/2 中，使用的是 0，因为做的是加法，考虑到 X=X+0，0 是一个中性值，所以从 0 开始，加法计算就不会出错。如果做的是乘法，就会使用初始值 1，因为 X=X*1。

min/1 和 max/2 函数不能使用默认的起始值。如果列表中只有负数，而我们又从 0 开始，那么答案就会出错。所以需要使用列表中的第 1 个元素作为起始值。很遗憾，我们不能总是以这种方式来决定起始值，所以我们把这个决策权留给程序员。

综合考虑以上所有因素，我们可以构建出下面的抽象：

```
fold(_, Start, []) -> Start;
fold(F, Start, [H|T]) -> fold(F, F(H,Start), T).
```

运行一下吧。

```
6> c(hhfuns).
{ok, hhfuns}
7> [H|T] = [1,7,3,5,9,0,2,3].
[1,7,3,5,9,0,2,3]
8> hhfuns:fold(fun(A,B) when A > B -> A; (_,B) -> B end, H, T).
9
9> hhfuns:fold(fun(A,B) when A < B -> A; (_,B) -> B end, H, T).
0
10> hhfuns:fold(fun(A,B) -> A + B end, 0, lists:seq(1,6)).
21
```

你能想到的所有把列表归约成一个元素的函数，基本上都能用 fold 表达出来。

不可思议的是，你还可以用一个元素（或者一个变量）去表示累加器。累加器也可以是一个列表。因此，我们可以用 fold 去构建一个列表。这意味着，所有其他列表处理递归函数几乎都可以基于折叠实现出来，甚至包括映射和过滤器，从这个意义上讲，折叠操作是普适的。下面是一些示例：

```
reverse(L) ->
    fold(fun(X,Acc) -> [X|Acc] end, [], L).

map2(F,L) ->
   reverse(fold(fun(X,Acc) -> [F(X)|Acc] end, [], L)).

filter2(Pred, L) ->
   F = fun(X,Acc) ->
```

```
            case Pred(X) of
                true -> [X|Acc];
                false -> Acc
            end
        end,
    reverse(fold(F, [], L)).
```

这些函数的运行效果和之前自己写的完全一样。真是强有力的抽象！

6.3.3　其他抽象

Erlang 标准库中提供了许多基于列表的抽象，映射、过滤器和折叠只是其中几个（参见 `lists:map/2`、`lists:filter/2`、`lists:foldl/3` 和 `lists:foldr/3`）。此处还有 `all/2` 和 `any/2`，这两个函数都以一个谓词为参数，分别用来测试列表中是否所有元素都为 `true` 以及列表中是否至少有一个元素为 `true`。

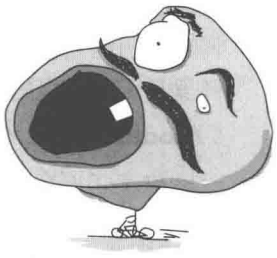
其中还有一个 `dropwhile/2` 函数，这个函数会略去列表前面的所有元素直到某个不满足谓词的元素。与之相反的是 `takewhile/2` 函数，这个函数会保留列表前面的所有元素直到某个不满足谓词的元素。作为这两个函数的补充，还有一个 `partition/2` 函数，它接收一个列表，返回两个列表，其中一个列表中的元素都满足给定的谓词，另外一个则包含剩余不满足的元素。

其他常用的列表函数还有 `flatten/1`、`flatlength/1`、`flatmap/2`、`merge/1`、`nth/2`、`nthtail/2` 和 `split/2`。如果想进一步了解它们，可以参见文档中对这些函数的介绍。

库中还包含像 `zip` 函数（在第 5 章中介绍过的）、`unzip` 函数、`map` 和 `fold` 的合并函数之类的其他函数。我建议你去阅读关于 `lists` 模块的文档，看看到底能完成哪些工作。你会发现，只要充分利用这些聪明人所提炼出来的抽象，基本上就无需自己去编写递归函数了。

第 7 章

错误和异常



在本书中，本章的地位有些尴尬。到目前为止，你已经看到了各种各样的错误，但是却没有看到多少错误处理机制。这多少是因为 Erlang 是一门具有两种主要范型的语言：函数式和并发。从本书开始以来，我所讲的都是函数式部分的内容：引用透明性、递归、高阶函数等。而让 Erlang 闻名于世的是其并发部分的内容：actor 模型、数百万个并发进程、监督树等。

虽然 Erlang 中提供了处理函数式代码中错误的方法，但是你时常听到的却是任其崩溃（let it crash）。这种错误处理机制位于 Erlang 语言的并发部分。不过，由于理解函数式部分的内容是学习并发部分的基础，所以本章只会介绍语言的函数式部分内容。要想管理错误，必须先理解错误。

7.1 错误编译

错误有很多种：编译期错误、逻辑错误以及运行时错误。我们先来看看编译期错误。

7.1.1 编译期错误

编译期错误通常是一些语法错误。例如，函数名、语言中的专用符号（方括号、圆括号、句点和逗号）、函数的参数个数等方面的错误。

下面列出了常见编译期出错消息以及可能的解决办法，可以在遇到这些错误时参考。

```
module.beam: Module name 'madule' does not match file name 'module'
```

在-module 属性中输入的模块名和文件名不匹配。

```
./module.erl:2: Warning: function some_function/0 is unused
```

函数没有被导出，或者它的调用者使用了错误的名字或者参数个数，还可能是之前编写的函数现在不再需要了。请检查代码！

./module.erl:2: function some_function/1 undefined

函数不存在。在-export 属性中或者函数定义时写错了函数名或者参数个数。当某个函数无法编译时也会显示这个错误，这通常是因为语法错误。例如，忘记了用句点结束函数定义。

./module.erl:5: syntax error before: 'SomeCharacterOrWord'

出现这个错误的原因有许多。常见的原因有括号、元组缺少结束符号，或者表达式结束符错误（例如，在 case 语句的最后一个分支结束时使用了逗号）。其他原因包括代码中使用了保留原子，或者在不同编码之间不正确地转换了 unicode 字符（我遇到过这种情况！）。

./module.erl:5: syntax error before:

这条出错消息的信息含量要少于上一条。通常在行结束符不正确时出现这条错误。它是上一条出错消息的特例，要保持警惕。

./module.erl:5: Warning: this expression will fail with a 'badarith' exception

虽然 Erlang 是动态类型语言，但是别忘了它是强类型的。出现这个出错消息，是由于编译器聪明地发现有一个算术表达式会失败（如 `1lama+5`）。不过对于更复杂的类型错误，编译器是无法发现的。

./module.erl:5: Warning: variable 'Var' is unused

定义了一个变量但是从来没有使用过它。这可能是代码 bug 造成的，因此请仔细检查编写的代码。对于其他情况，可以把变量名改成_，或者如果觉得名字能让代码更易读，那么可以在变量名前增加一个下划线前缀。

./module.erl:5: Warning: a term is constructed, but never used

在某个函数中，做了某些事情，如构建了一个列表或者定义了元组或匿名函数，但是没有把它绑定到变量或者返回。这个警告表明你做了一些无用的工作或者犯了错误。

./module.erl:5: head mismatch

函数可能会有多个头定义，每个头的参数个数可能不同。别忘了不同的参数个数意味着不同的函数，并且不能把不同参数个数的函数头交叉定义。同样地，当把一个函数的定义放到另外一个函数的头子句之间时，也会导致这个错误。

./module.erl:5: Warning: this clause cannot match because a previous clause at line 4 always matches

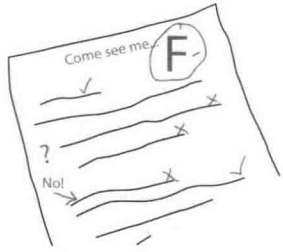
在模块中定义函数时，把一个特定的子句放在了一个匹配一切的子句之后。此时，编译器会给出警告，告知永远不会执行到这个子句。

./module.erl:9: variable 'A' unsafe in 'case' (line 5)

在 `case ... of` 的某个分支内定义了一个变量，但是在这个分支外面使用了这个变量。这被认为是不安全的。如果想使用这种变量，最好这样做：`MyVar = case ... of`。这个列表基本上覆盖了当前所有的编译期错误。错误有很多，并且通常来说，最困难的工作

往往是找到导致其他函数发生大量级联错误的那个错误。在解决编译错误时，最好按照报告的顺序进行，这样可以避免被那些实际上根本不是错误的错误所误导。

7.1.2 逻辑错误



逻辑错误最难被发现和调试。逻辑错误通常是由程序员引入的：条件语句（如 `if` 和 `case`）的分支没能考虑到所有情况、在本该使用除法的地方使用了乘法等。它们不会让程序崩溃，但是会导致隐蔽的数据错误以及不期望的程序行为。

在解决逻辑错误时，常常只能依靠自己。不过 Erlang 中有不少工具，可以给你提供帮助，这些工具包括测试框架、TypEr 和 Dialyzer 工具以及跟踪调试模块。编写测试也许是最好的防御手段。很不幸，每个程序员在职业生涯中都会遇到许多的错误，多到足以用几十本书来进行描述。在这里，我们将关注那些会导致程序崩溃的错误，因为它们就出现在事发地，不会传播到相离错误发生地很远的其他代码中。注意，这基本上就是我之前提到过的“任其崩溃”思想的起源。

7.1.3 运行时错误

运行时错误非常具有破坏性，因为它们会导致程序崩溃。虽然 Erlang 中有一些处理运行时错误的方法，但是能够识别这些错误肯定是有帮助的。下面会介绍一些常见的运行时错误以及产生这些错误的代码示例。

1. 函数子句错误

发生函数子句（function clause）错误最可能的原因是，函数的所有卫语句或者所有模式匹配都失败了，如下例所示：

```
1> lists:sort([3,2,1]).
[1,2,3]
2> lists:sort(fffffff).
** exception error: no function clause matching lists:sort(fffffff) (lists.erl, line
414)
```

2. `case` 子句错误

当忘记了一个特定情况、传入的数据类型错误或者需要一个匹配一切的子句时，会发生 `case` 子句（`case` clause）错误。下面是一个例子：

```
3> case "Unexpected Value" of
3>   expected_value -> ok;
3>   other_expected_value -> 'also ok'
3> end.
** exception error: no case clause matching "Unexpected Value"
```

3. `if` 子句错误

`if` 子句（`if` clause）错误的原因和 `case` 子句类似。当 Erlang 找不到一个可以求值为 `true` 的分支时，会引发这个错误。

```
4> if 2 > 4 -> ok;
4>    0 > 1 -> ok
```

```
4> end.
** exception error: no true branch found when evaluating an if expression
```

要确保考虑了所有情况，或者增加一个捕获一切的 true 子句。

4. 不正确匹配错误

当模式匹配失败时，就会出现不正确匹配（bad match）错误。这通常意味着你试图进行不可能的模式匹配（如下例所示）、对一个变量进行二次绑定或者在=操作符两边放置了不相等的东西（和重新绑定变量导致的错误一样！）。

```
5> [X,Y] = {4,5}.
** exception error: no match of right hand side value {4,5}
```

注意，有时出现这个错误是因为程序员认为*_MyVar*形式的变量和_完全一样。带下划线前缀的变量是正常变量，只不过在它们没有被使用时，编译器不会给出警告。不能对它们绑定多次。

5. 不正确参数错误

不正确参数（bad argument）错误和函数子句错误类似，因为它们都和使用不正确的参数调用函数有关。

```
6> erlang:binary_to_list("heh, already a list").
** exception error: bad argument
     in function binary_to_list/1
        called as binary_to_list("heh, already a list")
```

它们之间的主要区别在于，这个错误通常是由程序员在函数内部引发的，出现在卫语句之外，参数校验之后。BIF 以及其他任何用 C 语言编写的函数也都会选择抛出这个错误。在 7.2 节中，我会介绍如何引发这种错误。

6. 未定义函数错误

当调用了一个不存在的函数时，会发生未定义函数（undefined function）错误。

```
7> lists:random([1,2,3]).
** exception error: undefined function lists:random/1
```

要确保函数从模块中导出了，并且参数个数正确，仔细检查函数的名字和模块的名字，保证输入正确。

当模块不在 Erlang 的搜索路径上时，也会发生这个错误。默认情况下，Erlang 的搜索路径是当前目录。可以通过 code:add_patha("/some/path/")或者 code:add_pathz("some/path")把路径增加到搜索列表中。如果还不能解决问题，看看模块是否被编译过！

7. 不正确算术计算错误

当试图进行不正确的算术计算时，会发生不正确算术计算（bad arithmetic）错误，如除以 0 或者在原子和数值之间进行算术计算。

```
8> 5 + llama.
** exception error: bad argument in an arithmetic expression
     in operator +/2
        called as 5 + llama
```

8. 不正确函数错误

导致不正确函数（bad function）错误最常见的原因是把变量当成函数使用，但是变量的值并不是函数。下面的例子使用了第 6 章中的 hhfuns 函数，并把两个原子当成函数使用。这是不正确的，抛出了不正确函数错误。

```
9> hhfuns:add(one,two).
** exception error: bad function one
     in function hhfuns:add/2 (hhfuns.erl, line 7)
```

9. 不正确元数错误

不正确元数（bad arity）错误是不正确函数错误的特殊情况。当使用高阶函数时，给它们传递的参数个数多于或者少于实际参数个数时会出现这个错误。

```
10> F = fun(_) -> ok end.
#Fun<erl_eval.6.13229925>
11> F(a,b).
** exception error: interpreted function with arity 1 called with two arguments
```

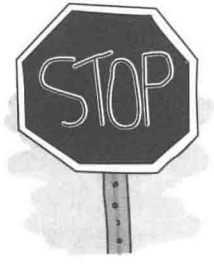
10. 系统限制错误

出现系统限制错误的原因有很多，下面是其中的一些：

- 进程太多；
- 原子太长；
- 函数参数个数太多；
- 原子太多；
- 连接的节点数太多。

在Erlang 高效编程指南的系统限制小节中，包含有这类错误的完整列表和细节信息，网址为 http://www.erlang.org/doc/efficiency_guide/advanced.html#2265856。注意，其中有些错误非常严重，会导致 VM 崩溃。

7.2 引发异常



要想监控代码的执行并防止逻辑错误，引发运行时崩溃，从而尽早定位问题往往是一种好方法。

在 Erlang 中有 3 种异常：出错（error）、退出（exit）和抛出（throw）。它们分别有各自的用处，见下面几节中的介绍。

7.2.1 出错异常

调用 erlang:error(Reason) 会结束当前进程的执行，如果你捕获了这个异常，那么其中会包括最后几次函数调用的栈跟踪和参数列表。出错异常（error exception）会引发运行时错误。

当代码对当前情况无法处理时，可以使用出错异常结束执行。如果出现了一个 if 子句异常，该如何处理呢？更改代码并重新编译——这就是你可以做的（而不是简单地显示一条漂亮的出错消息）。

1. 不适合出错异常的情形

第 5 章中的 tree 模块是不适合使用出错异常的一个例子。在进行查询时，tree 模块不一定总能在树中找到一个特定的主键。此时，让调用者去处理这个未知的结果是合理的。调用者可以使用一个默认值、插入一个新值、删除整棵树或者做其他处理。在这种情况下，返回{ok, Value}元组或者一个像 undefined 这样的原子，要比引发错误异常更为合适。

2. 自定义错误

出错异常并不局限于 Erlang 语言所提供的那些。你可以自定义出错异常，下面是一个示例：

```
1> erlang:error(badarith).
** exception error: bad argument in an arithmetic expression
2> erlang:error(custom_error).
** exception error: custom_error
```

其中，custom_error 不能被 Erlang shell 识别，也不具有像“bad argument in...,”之类的惯常解释，不过它的用法完全一样，可以被程序员以同样的方式进行处理（会在 7.3 节中介绍）。

7.2.2　退出异常

Erlang 中有两种退出异常（exit exception）。

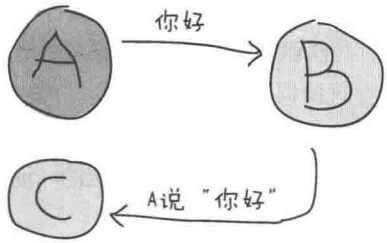
- 内部退出：这种异常在调用函数 exit/1 时触发，会导致当前进程结束执行。
- 外部退出：这种异常在调用函数 exit/2 时触发，在多进程环境中使用（Erlang 并发部分内容）。

这里，我们只关注内部退出。我们会在第 12 章中介绍外部退出。

内部退出和出错异常类似。事实上，它们在历史上曾经完全一样，当时只存在 exit/1 函数。出错和退出的应用场景大致一样。那么如何在两者之中进行选择呢？嗯，不太好选择。为了知道何时该使用哪一个，需要理解 Erlang 进程背后最一般的原则。

进程之间可以互发消息。进程也可以监听消息，也就是等待消息。

进程可以选择要接收哪条消息。进程可以丢弃掉一些消息，忽略消息，等待一段时间后放弃接收，等等。



基于这些基本的概念，Erlang 的实现者们使用一种专门消息（退出信号）在进程之间传递异常。这有点像进程的最后一口气，它们一旦被发出，进程就死亡了，进程代码也会停止运行。其他正在监听这条专门消息的进程就会知道所发生的事件，并根据需要进行相应的处理，包括记录日志、重启死亡进程等。

解释了这个概念之后，就很容易理解 `erlang:error/1` 和 `exit/1` 之间的区别了。虽然两者的使用方式非常类似，但是它们在意图上存在区别。那只不过是一个错误呢，还是一种值得杀死当前进程的严重状况呢？下面的事实会让这种意图更加明显：`erlang:error/1` 会返回调用栈而 `exit/1` 不会。如果当前函数的调用栈很深，参数很多，把这个退出消息复制给每一个监听进程意味着要复制很多数据。在有些情况下，这并不现实。

7.2.3 抛出异常

当期望程序员处理所发生的异常时，可以抛出（throw）异常。与错误异常和退出异常不同，抛出异常并没有“让进程崩溃!”的意思，只是为了改变控制流。

要抛出异常，具体语法如下：

```
1> throw(permission_denied).
** exception throw: permission_denied
```

可以把 `permission_denied` 替换成任何想要的东西（包括`'everything is fine'`，不过这没什么好处，还会让你失去朋友）。

> **注意** 如果使用了 `throw`，并期望程序员去处理它们，在抛出这些异常的模块中对其进行文档描述通常是一种很好的做法。

在深度递归中，抛出异常可以用于非局部返回（nonlocal return）。`ssl` 模块就是一个例子，它用 `throw/1` 把元组`{error, Reason}`返回给顶层函数。顶层函数会把这个元组直接返回给用户。这样，实现者就只需针对成功的情况编写代码，把所有的异常情况都交给一个顶层函数处理。

另外一个使用抛出异常的例子是 `array` 模块，其中有一个查询函数在找不到所需元素时会返回用户提供的默认值。当找不到元素时，会抛出一个 `default` 值作为异常，顶层函数会处理这个异常，并把它替换成用户提供的默认值。这样，编写这个模块的程序员就不需要把默认值作为参数传递给查询算法中的每一个函数，同样只需关注成功的情况。

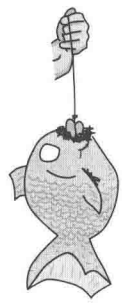
一般来说，在用抛出异常进行非局部返回时，尽量把它限制在一个模块中，这样调试起来就比较容易。这样做还可以让你在更改模块内部实现时无需更改接口。

7.3　处理异常

我曾经说过，抛出异常、出错异常和退出异常都可以被处理。处理的方式是使用 try ... catch 表达式。

try ... catch 是一种表达式的求值方法，它可以在处理成功情况的同时处理出现的错误。下面是这种表达式的一般语法形式：

```
try Expression of
    SuccessfulPattern1 [Guards] ->
        Expression1;
    SuccessfulPattern2 [Guards] ->
        Expression2
catch
    TypeOfError:ExceptionPattern1 ->
        Expression3;
    TypeOfError:ExceptionPattern2 ->
        Expression4
end.
```



> **注意**　在上面的语法中，[*Guards*] 中的方括号只用来表示卫语句是可选的，无需把卫语句放置到列表中。

try 和 of 之间的 *Expression* 是被保护的对象。这意味着，发生在这个表达式中的任何异常都会被捕获到。

try ... of 和 catch 之间的模式和表达式的行为方式与 case ... of 完全一样。它们不会被保护，在其中允许出现模式匹配、变量绑定以及卫语句。

在 catch 部分中，可以针对不同的异常类型把 *TypeOfError* 替换为 error、throw 或者 exit。如果没有提供任何类型，会默认为 throw 类型。下面我们来进行一些实验。

7.3.1　处理不同类型的异常

我们先创建一个名为 exceptions 的模块（这里，我们追求简单）。

```
-module(exceptions).
-compile(export_all).
throws(F) ->
    try F() of
        _ -> ok
    catch
        Throw -> {throw, caught, Throw}
    end.
```

编译这个模块，并用不同的异常类型来进行实验。

```
1> c(exceptions).
{ok,exceptions}
2> exceptions:throws(fun() -> throw(thrown) end).
```

```
{throw,caught,thrown}
3> exceptions:throws(fun() -> erlang:error(pang) end).
** exception error: pang
     in function exceptions:throws/1 (exceptions.erl, line 5)
```

可以看出，这个 try ... catch 只处理抛出异常。这是因为没有指定任何异常类型，会默认为 throw 类型。我们可以编写另外几个函数，每个都捕获一种异常类型：

```
errors(F) ->
    try F() of
        _ -> ok
    catch
        error:Error -> {error, caught, Error}
    end.

exits(F) ->
    try F() of
        _ -> ok
    catch
        exit:Exit -> {exit, caught, Exit}
    end.
```

我们来试试新的代码。

```
4> c(exceptions).
{ok,exceptions}
5> exceptions:errors(fun() -> erlang:error("Die!") end).
{error,caught,"Die!"}
6> exceptions:exits(fun() -> exit(goodbye) end).
{exit,caught,goodbye}
```

接下来，我们展示一下如何在单个 try ... catch 中捕获所有类型的异常。首先定义一个会产生所有类型异常的函数。

```
sword(1) -> throw(slice);
sword(2) -> erlang:error(cut_arm);
sword(3) -> exit(cut_leg);
sword(4) -> throw(punch);
sword(5) -> exit(cross_bridge).

black_knight(Attack) when is_function(Attack, 0) ->
    try Attack() of
        _ -> "None shall pass."
    catch
        throw:slice -> "It is but a scratch.";
        error:cut_arm -> "I've had worse.";
        exit:cut_leg -> "Come on you pansy!";
        _:_ -> "Just a flesh wound."
    end.
```

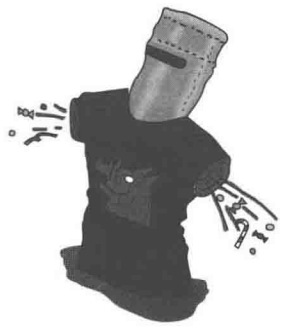
其中，is_function/2 是一个 BIF 函数，会确保变量 Attack 是一个参数为 0 的函数。然后再另外增加一行代码：

```
talk() -> "blah blah".
```

这次的结果完全不同。

```
7> c(exceptions).
{ok,exceptions}
8> exceptions:talk().
"blah blah"
9> exceptions:black_knight(fun exceptions:talk/0).
"None shall pass."
10> exceptions:black_knight(fun() -> exceptions:sword(1) end).
"It is but a scratch."
11> exceptions:black_knight(fun() -> exceptions:sword(2) end).
"I've had worse."
12> exceptions:black_knight(fun() -> exceptions:sword(3) end).
"Come on you pansy!"
13> exceptions:black_knight(fun() -> exceptions:sword(4) end).
"Just a flesh wound."
14> exceptions:black_knight(fun() -> exceptions:sword(5) end).
"Just a flesh wound."
```

第 9 行展示了函数正常执行时，黑武士的正常行为。随后的每一行分别展示了依照异常类型(throw、error 和 exit)和原因(slice、cut_arm 和 cut_leg) 进行模式匹配的结果。



第 13 行和第 14 行展示了匹配所有异常的子句的结果。需要使用_来确保捕获了任意类型、任意原因的异常。在实际中，要小心使用捕获一切的模式。在对代码进行保护时，尽量只处理那些能够处理的异常。对于无法处理的其余异常，Erlang 提供了另外的处理机制。

7.3.2　catch 后的 after 语句

在 try ... catch 之后还可以增加一个子句，这个子句始终会执行，如下：

```
try Expression of
    Pattern -> Expr1
catch
    Type:Exception -> Expr2
after
    Expr3
end
```

这个子句和很多其他语言中的 finally 块等价。无论是否出现错误，after 部分中的表达式都一定会运行。

不过，after 子句不会返回任何值。因此其中通常会运行带有副作用的代码。无论是否发生异常，都要确保打开的文件被关闭，是这个子句的标准应用之一。

7.3.3　尝试多个表达式

我们已经介绍了如何使用 catch 语句块处理 Erlang 中的 3 种异常。不过，有一点我没有讲：

实际上，在 try 和 of 之间可以有多个表达式！

```
whoa() ->
    try
        talk(),
        _Knight = "None shall pass!",
        _Doubles = [N*2 || N <- lists:seq(1,100)],
        throw(up),
        _WillReturnThis = tequila
    of
        tequila -> "Hey, this worked!"
    catch
        Exception:Reason -> {caught, Exception, Reason}
    end.
```

调用 exceptions:whoa()时，显然会返回{caught, throw, up}，因为其中调用了 throw(up)。可以看出，在 try 和 of 之间可以有多个表达式。

exceptions:whoa/0 中有一点很明显，但是你可能没有注意到，那就是当使用了多个表达式时，我们有时就可能不太会在意返回值了。因此 of 部分就变得没什么用了。嗯，很好——可以直接把它去掉：

```
im_impressed() ->
    try
        talk(),
        _Knight = "None shall pass!",
        _Doubles = [N*2 || N <- lists:seq(1,100)],
        throw(up),
        _WillReturnThis = tequila
    catch
        Exception:Reason -> {caught, Exception, Reason}
    end.
```

现在精简多了！

保护正确的东西

异常中被保护的部分是无法做到尾递归的。在执行时，VM 必须要一直保持一个相关的引用，以防异常出现。因为不带有 of 部分的 try ... catch 语句中只有被保护的部分，所以对于需要长时间运行的程序（Erlang 的擅长领域）来说，在其中进行递归调用非常危险。经过充分数量的迭代之后，会导致内存耗尽，或者让程序的运行速度变慢。如果把递归调用放在 of 和 catch 之间，就不会受到保护，此时可以对其进行尾调用优化（在第 5 章中介绍过）。不过，如果在 try 表达式中使用了 after 语句，那么就不能进行这种优化了，因为 after 语句中的表达式会被最后执行，所以需要在函数调用列表中对其进行记录。

为了避免这种意料之外的行为，有人会默认使用 try ... of ... catch 而不是 try ... catch，那些明显无此问题的非递归代码除外。至于如何选择，你得自行决策！

7.3.4　更多选择

如果说前面的错误处理机制还不足以让 Erlang 和其他大多数语言势均力敌，那么 Erlang 中还提供了另外一种错误处理机制。这个机制是由关键字 `catch` 提供的，基本上可以捕获所有类型的异常，也能返回正常结果。它显示异常时使用了不同的表示方式，所以显得有些奇怪。下面是一个例子：

```
1> catch throw(whoa).
whoa
2> catch exit(die).
{'EXIT',die}
3> catch 1/0.
{{'EXIT',{badarith,[{erlang,'/',[1,0],[]},
                    {erl_eval,do_apply,6,[{file,"erl_eval.erl"},{line,576}]},
                    {erl_eval,expr,5,[{file,"erl_eval.erl"},{line,360}]},
                    {shell,exprs,7,[{file,"shell.erl"},{line,668}]},
                    {shell,eval_exprs,7,[{file,"shell.erl"},{line,623}]},
                    {shell,eval_loop,3,[{file,"shell.erl"},{line,608}]}]}}}
4> catch 2+2.
4
```

可以看出，抛出异常和之前一样，但是退出异常和出错异常都被表示为`{'EXIT', Reason}`。这是由于出错是在退出之后增加到语言中的（Erlang 的实现者为了向后兼容使用了类似的表示方法）。

我们来尝试另外一个例子：

```
5> catch doesnt:exist(a,4).
{'EXIT',{undef,[{doesnt,exist,[a,4],[]},
                {erl_eval,do_apply,6,[{file,"erl_eval.erl"},{line,576}]},
                {erl_eval,expr,5,[{file,"erl_eval.erl"},{line,360}]},
                {shell,exprs,7,[{file,"shell.erl"},{line,668}]},
                {shell,eval_exprs,7,[{file,"shell.erl"},{line,623}]},
                {shell,eval_loop,3,[{file,"shell.erl"},{line,608}]}]}}
```

错误的类型是 undef，意思是调用的函数没有定义。

紧跟着错误类型后面的列表是调用栈跟踪。下面是对调用栈跟踪的解释。

- 调用栈跟踪的顶层元素表示最后一个调用函数（*{Module, Function, Arguments}*）。也就是那个没有定义的函数。
- 之后的元组是在错误出现之前所调用的函数。此时，它们的格式是*{Module, Function, Arity, Details}*。
- `Details` 字段是一个元组列表，其中包含着文件名和文件行号。在本例中，文件名是 `erl_eval.erl` 和 `shell.erl`，因为这两个文件负责解释在 Erlang shell 中输入的代码。

关于调用栈跟踪的解释就这些了，真的！

> **注意**　对于 R15B 之前的 Erlang 发布版，在调用栈跟踪中没有 `Details` 字段。在 20 年的时间里，Erlang 程序员都是靠简短的函数名和超强的推理能力查找错误的源头。

在崩溃的进程中手动调用 erlang:get_stacktrace/0，也可以获得调用栈跟踪信息。

catch 常常会被写成如下方式（目前还是在 exceptions.erl 文件中）：

```
catcher(X,Y) ->
   case catch X/Y of
      {'EXIT', {badarith,_}} -> "uh oh";
      N -> N
end.
```

当按照如下方式运行时，会得到预期的结果：

```
6> c(exceptions).
{ok,exceptions}
7> exceptions:catcher(3,3).
1.0
8> exceptions:catcher(6,3).
2.0
9> exceptions:catcher(6,0).
"uh oh"
```

这种捕获异常的方式看起来更加紧凑、简单，不过在使用的过程中会有几个问题。下面的例子展示了其中的一个问题：

```
10> X = catch 4+2.
* 1: syntax error before: 'catch'
10> X = (catch 4+2).
6
```

我们期望第一种输入形式和第二种有同样的行为。但是，Erlang 并没有如我们所愿。这是由语言中操作符的优先级造成的。catch 和=之间存在冲突，唯一避免冲突的办法就是把 catch 包裹在括号中。这有些违反直觉，因为大部分表达式都不会以这种方式被放到括号中。

catch 的另外一个问题是，无法分辨出异常的底层表示和真正异常之间的区别，如下例所示：

```
11> catch erlang:boat().
{'EXIT',{undef,[{erlang,boat,[],[]},
                {erl_eval,do_apply,6,[{file,"erl_eval.erl"},{line,576}]},
                {erl_eval,expr,5,[{file,"erl_eval.erl"},{line,360}]},
                {shell,exprs,7,[{file,"shell.erl"},{line,668}]},
                {shell,eval_exprs,7,[{file,"shell.erl"},{line,623}]},
                {shell,eval_loop,3,[{file,"shell.erl"},{line,608}]}]}}
12> catch exit({undef,[{erlang,boat,[],[]},
12>                    {erl_eval,do_apply,6,[{file,"erl_eval.erl"},{line,576}]},
12>                    {erl_eval,expr,5,[{file,"erl_eval.erl"},{line,360}]},
12>                    {shell,exprs,7,[{file,"shell.erl"},{line,668}]},
12>                    {shell,eval_exprs,7,[{file,"shell.erl"},{line,623}]},
12>                    {shell,eval_loop,3,[{file,"shell.erl"},{line,608}]}]}).
{'EXIT',{undef,[{erlang,boat,[],[]},
                {erl_eval,do_apply,6,[{file,"erl_eval.erl"},{line,576}]},
```

```
{erl_eval,expr,5,[{file,"erl_eval.erl"},{line,360}]},
{shell,exprs,7,[{file,"shell.erl"},{line,668}]},
{shell,eval_exprs,7,[{file,"shell.erl"},{line,623}]},
{shell,eval_loop,3,[{file,"shell.erl"},{line,608}]}]}}
```

同样，也无法分辨出错误和正常退出之间的区别，因为两者的结果完全一样。使用 throw 也可以产生出上面的异常。事实上，在 catch 中使用 throw/1 也会造成问题，来看另外一个例子：

```
one_or_two(1) -> return;
one_or_two(2) -> throw(return).
```

现在，看看这个致命的问题：

```
13> c(exceptions).
{ok,exceptions}
14> catch exceptions:one_or_two(1).
return
15> catch exceptions:one_or_two(2).
return
```

因为代码处在 catch 之中，所以我们根本无法知道函数是抛出了异常还是正常返回了一个值！这种情况在实际中可能不会经常发生，但是这个问题非常严重，因此在 Erlang/OTP R10B 发布版中增加了 try ... catch 机制。

7.4　在树中使用 try 语句

为了实践一下异常处理，我们用第 5 章中的 tree 模块来做一些练习。我们要编写一个查询函数，用于判断某个值是否存在于树中。因为树是按照主键排序的，但是在这个练习中，我们不关心主键，所以我们会遍历整棵树，直到找到了这个值。

对树遍历的方法和我们之前在 tree:lookup/2 中的做法大致类似，不过这次会总是先搜索左子树，然后再搜索右子树。在编写这个函数时，别忘了树结点可以是{*node*, {*Key*, *Value*, *NodeLeft*, *NodeRight*}}，或者{node, 'nil'}（在树为空时）。基于这些认识，我们可以实现一个没有异常处理的基本实现：

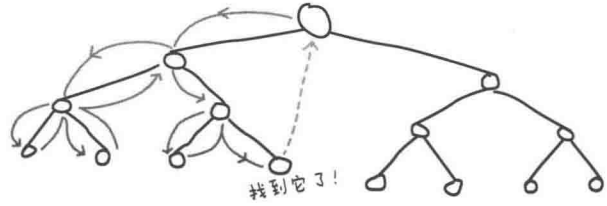
```
%% 在树中查找给定值'Val'
has_value(_, {node, 'nil'}) ->
    false;
has_value(Val, {node, {_, Val, _, _}}) ->
    true;
has_value(Val, {node, {_, _, Left, Right}}) ->
    case has_value(Val, Left) of
        true -> true;
            false -> has_value(Val, Right)
    end.
```

这个实现有一个问题，对于树分支中的每一个结点，都要对前一个分支的结果进行检查。

这有点令人讨厌。使用 throw，我们可以减少一些比较操作：

```
has_value(Val, Tree) ->
    try has_value1(Val, Tree) of
        false -> false
    catch
        true -> true
    end.

has_value1(_, {node, 'nil'}) ->
    false;
has_value1(Val, {node, {_, Val, _, _}}) ->
    throw(true);
has_value1(Val, {node, {_, _, Left, Right}}) ->
    has_value1(Val, Left),
    has_value1(Val, Right).
```

这个实现的执行逻辑和前一个版本类似，只是不再需要对返回值进行检查了——根本就不用关心它。在这个实现版本中，只有执行到了 throw 语句才说明在树中找到了值。此时，对树的遍历就结束了，直接返回到顶层的 catch 语句中。否则，会一直执行，直到返回最后的 false。调用者会看到这个值。



当然，这个实现的代码长度要比上一个长一些。针对正在进行的操作，是可以通过使用基于 throw 的非局部返回获得速度和清晰性方面的收益的。这个例子中只是进行了简单的比较，看不出能得到多少好处，但是，对于更加复杂的数据结构和操作来说，这个实践是很有价值的。

学习了这些内容后，我们现在已经可以使用 Erlang 语言中的顺序化部分去解决实际的问题了。

第 8 章

用函数式思维解决问题

现在，我们所学到的 Erlang 知识足以用来做一些实际的工作了。在本章中，我们会用前面几章介绍的技术解决一些有趣的问题。

本章中的问题取自 Miran Lipovača 的 *Learn You a Haskell for Great Good!*一书（No Starch，2011，可在线阅读 http://learnyouahaskell.com）。我决定使用同样的问题，这样有好奇心的读者可以自行对 Erlang 和 Haskell 的解决方案进行对比。如果你这样做了，就会发现，虽然这两种语言的语法差异很大，但是最终代码却非常相似。这是因为，一旦理解了函数式编程的概念，就能很容易地把这种理解应用到其他函数式语言中。

8.1 逆波兰式计算器

大部分人在学习算术表达式时都是把操作符放在数的中间（`(2 + 2) / 5`）。这也是大部分计算器所接受的数学表达式格式，这种表示法也是学校中所教授的。这种表示法有一个缺点，你必须要知道操作符的优先级。例如，乘法和除法要比加法和减法更加重要（优先级更高）。

还有另外一种表示法，称为*前缀表示法*或者*波兰表示法*，其中操作符在操作数前面。在这种表示法中，`(2 + 2) / 5` 会被表示成`(/ (+ 2 2) 5)`。如果我们决定让+和/永远只能接收两个参数，那么`(/ (+ 2 2) 5)`可以被简化为`/ + 2 2 5`。

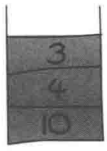
不过，我们将要使用的不是波兰表示法，而是*逆波兰表示法*（RPN），它和前缀表示法正好相反：操作符跟在操作数后面。在 RPN 中，上面的例子表达式会被写成`2 2 + 5 /`。表达式`9 * 5 + 7`会变成`9 5 * 7 +`，`10 * 2 * (3 + 4) / 2`会被转换成`10 2 * 3 4 + * 2 /`。这种表示法在早期的计算器中被广泛应用，因为它使用的内存很少。事实上，现在还有人在使用 RPN 计算器。我们也要编写一个。

8.1.1 RPN 计算器的工作原理

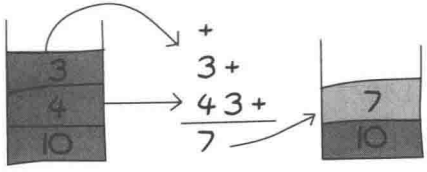
我们先来考虑一下如何读取 RPN 表达式。一种方法是逐个寻找操作符，然后根据参数个数把它们和操作数组织在一起：

```
10 4 3 + 2 * -
10 (4 3 +) 2 * -
10 ((4 3 +) 2 *) -
(10 ((4 3 +) 2 *) -)
(10 (7 2 *) -)
(10 14 -)
-4
```

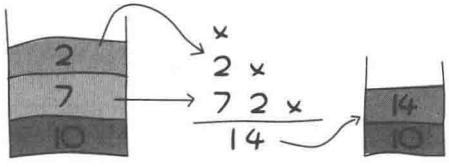
不过，对于计算机或者计算器来说，有一种更简单的读取 RPN 表达式的方法，可以把读取到的所有操作数都放到一个栈中。例如，在数学表达式 10 4 3 + 2 * -中，第一个读取到的操作数是 10。把它放入栈中。接着读取到的是 4，也放到栈顶部。第三个读取到的是 3——再次放到栈中。现在，栈的情况如下：



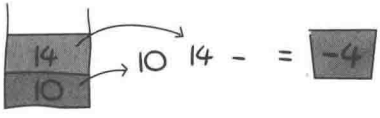
下一个读入的字符是+。这是一个有两个参数的函数。为了调用它，需要给它提供两个操作数，可以从栈中获取：



我们得到了结果 7，并把它放回到栈顶部（嗯，我们不想到处乱放这些数字！）。栈的内容现在是[7,10]，表达式的剩余部分是 2 * -。可以读取 2，并把它放到栈顶部。然后读取到*，它需要两个操作数才能进行计算。同样，从栈中获取：



把 14 放回到栈顶部。现在只剩下-了，它也需要两个操作数。啊，运气太好了！栈中正好有两个操作数。就用它们！



这样就得到了计算结果。这种基于栈的方法相对简单可靠，在开始计算结果之前需要的解析工作也比较少，因此那些老的计算器都使用这种方法。

8.1.2　实现 RPN 计算器

一旦弄清楚了复杂的部分，用 Erlang 去实现一个 RPN 计算器就不太困难了。这个问题中的复杂部分是得到最终结果所依赖的那些步骤，我们刚刚已经弄清楚了这些步骤。因此，我们可以开始实现了，创建一个名为 calc.erl 的文件。

首先要考虑的是如何表示一个数学表达式。为了简单起见，我们用字符串来表示输入的表达式："10 4 3 + 2 * -"。字符串中含有空格，空格不是我们解决问题的过程所必需的，只是用来对字符串进行简单的词法分析。词法分析之后，得到的才是一个有用的数据项列表，其格式为["10","4","3","+","2","*","-"]。函数 string:tokens/2 正好可以完成这项工作：

```
1> string:tokens("10 4 3 + 2 * -", " ").
["10","4","3","+","2","*","-"]
```

这对于我们的表达式来说是一个不错的表示形式。

接下来要实现栈。如何实现呢？你也许已经注意到了，Erlang 的列表和栈的行为很像。[*Head*|*Tail*]中的 cons 操作符（|）的实际行为效果和把 Head 压入一个栈（也就是 Tail）完全一样。用列表来实现栈完全可行。

对于表达式的读取，只需要像我们手工解决问题那样去实现就行了。从表达式中读取一个值，如果它是一个数值，就把它放到栈中。如果它是一个函数，就把它需要的所有值从栈中弹出，然后把计算结果再放回栈中。概括来说，就是我们需要一次性地遍历完整个表达式，并累积结果。听上去，这正是折叠（fold）操作做的事情！

现在需要考虑 lists:foldl/3 对表达式中的每个操作符和操作数要应用的函数是什么。由于这个函数运行在一个折叠操作中，因此它需要两个参数：第一个参数是要处理的表达式元素，第二个参数是栈。

可以开始在 calc.erl 文件中编写代码了。先来编写负责循环和去除表达式中空格的函数：

```
-module(calc).
-export([rpn/1]).

rpn(L) when is_list(L) ->
    [Res] = lists:foldl(fun rpn/2, [], string:tokens(L, " ")),
    Res.
```

下面我们来实现 rpn/2。注意，因为表达式中的每个操作数和操作符演算的结果都会被放到栈顶部，所以整个表达式的求值结果也会在栈中。我们需要把这个最后的结果值先从栈中取出来，然后再返回给调用者。因此，我们用[Res]进行模式匹配，并只返回了 Res。

现在来编写比较困难的部分。对接收到的所有值，函数 rpn/2 都需要进行栈操作。这个函数的头看起来像 rpn(Op,Stack)，其返回值看起来像[NewVal|Stack]。当得到的是一个普通数值时，会执行如下操作：

```
rpn(X, Stack) -> [read(X)|Stack].
```

其中，read/1 函数会把字符串转换成整数或者浮点数。遗憾的是，Erlang 中没有完成这项

工作的内置函数（只有能完成一种类型转换的函数）。因此，我们要自己来实现这个函数，如下：

```
read(N) ->
    case string:to_float(N) of
        {error,no_float} -> list_to_integer(N);
        {F,_} -> F
    end.
```

在上面的代码中，string:to_float/1 会把像"13.37"之类的字符串转换成等价的数值。不过，如果读取到的不是浮点值，就会返回{error,no_float}。此时，需要再次调用 list_to_integer/1。

现在我们回到 rpn/2。读取到的所有数值都被增加到栈中了。但是，由于我们使用的模式会匹配到所有东西（参见第 5 章中关于模式匹配的介绍），所以操作符也会被压入栈中。为了避免这种情况，我们把对于操作符的匹配放到前面的子句中。先来增加加法操作符：

```
rpn("+", [N1,N2|S]) -> [N2+N1|S];
rpn(X, Stack) -> [read(X)|Stack].
```

可以看出，当遇到"+"字符串时，会从栈顶部取出两个数值（N1,N2），把它们相加的结果放回到栈中。这个逻辑和我们之前的手工计算逻辑完全一样。运行一下程序，可以看出它工作正常：

```
1> c(calc).
{ok,calc}
2> calc:rpn("3 5 +").
8
3> calc:rpn("7 3 + 5 +").
15
```

剩余的代码很简单，只要把其他操作符都加进来就行了：

```
rpn("+", [N1,N2|S]) -> [N2+N1|S];
rpn("-", [N1,N2|S]) -> [N2-N1|S];
rpn("*", [N1,N2|S]) -> [N2*N1|S];
rpn("/", [N1,N2|S]) -> [N2/N1|S];
rpn("^", [N1,N2|S]) -> [math:pow(N2,N1)|S];
rpn("ln", [N|S])    -> [math:log(N)|S];
rpn("log10", [N|S]) -> [math:log10(N)|S];
rpn(X, Stack) -> [read(X)|Stack].
```

注意，有些函数（如对数）只有一个参数，只能从栈中取出一个元素。另外，还可以增加 sum 和 prod 之类的函数，这两个函数分别返回到目前为止读取到的所有元素的和以及乘积，这两个函数的实现作为练习留给读者。为了能够给你提供些帮助，这两个函数已经在我编写的 calc.erl 中实现了。

8.1.3 代码测试

为了保证一切正常，我们会编写一些非常简单的单元测试。可以把 Erlang 的=操作符当作一种断言函数使用。只要遇到了不期望的值，断言都会崩溃，这正是我们需要的。当然，Erlang 中还有其他更高级的测试框架，包括 EUnit 和 Common Test。我们会在第 25 章和第 28 章中学习

它们，现在，简单的=就够用了。

```
rpn_test() ->
    5 = rpn("2 3 +"),
    87 = rpn("90 3 -"),
    -4 = rpn("10 4 3 + 2 * -"),
    -2.0 = rpn("10 4 3 + 2 * - 2 /"),
    ok = try
        rpn("90 34 12 33 55 66 + * - +")
    catch
        error:{badmatch,[_|_]} -> ok
    end,
    4037 = rpn("90 34 12 33 55 66 + * - + -"),
    8.0 = rpn("2 3 ^"),
    true = math:sqrt(2) == rpn("2 0.5 ^"),
    true = math:log(2.7) == rpn("2.7 ln"),
    true = math:log10(2.7) == rpn("2.7 log10"),
    50 = rpn("10 10 10 20 sum"),
    10.0 = rpn("10 10 10 20 sum 5 /"),
    1000.0 = rpn("10 10 20 0.5 prod"),
    ok.
```

这个测试函数尝试了所有操作。如果没有出现异常，就认为测试成功。前面的4个测试检查基本的算术函数是否正确。在第5个测试中，`try ... catch` 期望出现一个不正确匹配错误，因为这个表达式不正确：

```
90 34 12 33 55 66 + * - +
90 (34 (12 (33 (55 66 +) *) -) +)
```

在 `rpn/1` 结束时，由于缺少一个操作符，因此值-3947和90会被留着栈中。有两种解决这个问题的方法：一种是忽略它，只把栈顶部的值取出（就是那个最后计算的结果），另一种是崩溃，因为算术计算是错误的。由于Erlang的哲学是让其崩溃，因此我们在此也选择了这种方法。实际导致崩溃的地方是 `rpn/1` 中的`[Res]`。这个模式要求栈中只能存在一个元素——计算结果。

其中还有几个形如 `true = FunctionCall1 == FunctionCall2` 的测试，这是因为=操作符的左边不能是函数调用。它实际上还是一个断言，因为我们把两个函数比较的结果和 `true` 进行匹配。

我还增加了几个关于 `sum` 和 `prod` 操作的测试用例，这样你就可以在实现这两个函数后对其进行测试了。如果所有的测试都成功了，应该看到如下显示：

```
1> c(calc).
{ok,calc}
2> calc:rpn_test().
ok
3> calc:rpn("1 2 ^ 2 2 ^ 3 2 ^ 4 2 ^ sum 2 -").
28.0
```

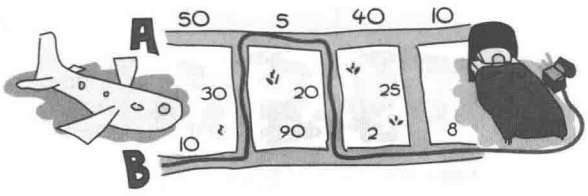
在上面的例子中，28.0确实等于 $\mathrm{sum}(1^2 + 2^2 + 3^2 + 4^2) - 2$。你可以根据需要尝试更多的计算。

注意 有一种方法可以改进我们的计算器，当由于未知操作符或者栈中遗留有多余的值而导致计算器崩溃时，让计算器抛出 badarith 错误，而非 badmatch 错误。这可以让 calc 模块的使用者更容易进行调试。

8.2 从希思罗到伦敦

接下来的问题也取自 *Learn You a Haskell* 一书。你正在一架几个小时后就要降落在希思罗机场的飞机上。你需要尽快赶到伦敦。因为你一个叔叔去世了，而你想第一个跑过去与他告别。

从希思罗到伦敦有两条路，这两条路之间有很多小的街道。由于限速和交通方面的原因，某部分路段和小街道花得时间要更长一些。在飞机降落前，你决定找到一条到达你叔叔住所的最优路径，从而最大化自己的机会。下面是你在笔记本电脑上找到的一幅地图。



你阅读过一些在线书籍，对 Erlang 非常喜欢，因此决定用 Erlang 来解决这个问题。为了更容易使用地图，你可以把如下数据输入一个名为 road.txt 的文件中：

```
50
10
30
5
90
20
40
2
25
10
8
0
```

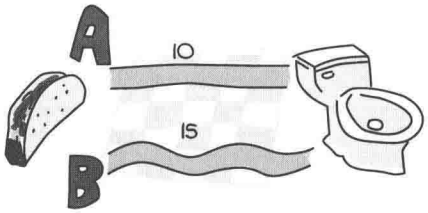
路径的罗列方式是 A1, B1, *X*1, A2, B2, *X*2, ..., A*n*, B*n*, X*n*，其中 *X* 是一条把地图中 A、B 两侧连接起来的路段。我们增加了一个 0 作为最后一个 *X* 路段，因为无论如何，此时已经到达了目的地。可以用形如{A,B,X}的三元组来表示这些数据。

接着你意识到，如果不知道如何手工解决这个问题，那么根本谈不上用 Erlang 去解决这个问题。我们先用学到的递归知识来手工解决这个问题。

8.2.1 递归地解决问题

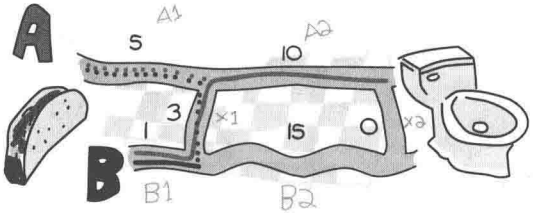
在编写一个递归函数时，首先要做的是找到基本情形（base case）。对手边的这个问题来说，基本情形是只有一个元组要分析的情形，也就是说，只需要在 A 和 B 之间进行选择（以及穿越 X，

在本例中无需穿越 X，因为已经到达了目的地）。



此时，只需选择路径 A 和路径 B 中最短的那条即可。我们理解递归的工作原理，知道应该尝试着向基本情形收敛。这意味着，在处理的每一步中，都要把问题化简成一个 A、B 选择问题，供下一步使用。

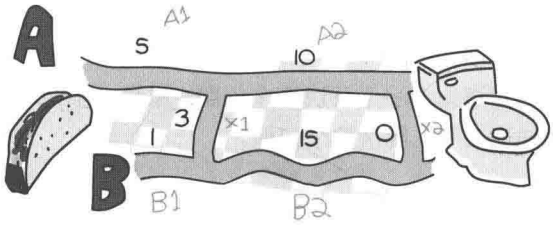
我们把地图扩展一下，重新开始计算。



啊！变得有趣了！如何把三元组{5, 1, 3}化简成一个严格的 A、B 选择问题呢？我们来看看对于 A 来说有多少种选择。要到达 A1 和 A2 的交界处（我们称这个点为 A1），可以从 A1 路段直接到达（5），也可以先经 B1（1），然后穿越 X1（3）到达。在本例中，第一个选择（5）要比第二个选择（4）长。如果选 A，最短路径是[B,X]。那么 B 有哪些选择呢？可以经 A1（5），然后穿越 X1（3），也可以直接经 B1 到达（1）。

现在，我们有了一条长度为 4 的路径[B,X]到达第一个 A 交叉点，以及一条长度为 1 的路径[B]到达 B1 和 B2 的交叉点。接下来必须决定如何到达第二个 A 点（A2 与终点或者 X2 的交叉点）以及第二个 B 点（B2 和 X2 的交叉点）。为了做出决策，我建议还是按照前面的做法进行（你没有选择只能遵从，因为我是本书的作者）。开始！

到达下一个 A 点有两条路径：一条是从[B,X]出发经 A2 到达，这条路径的长度是 14（14=4+10），还有一条是从[B]出发，再经 B2、X2 到达，路径长度为 16（16=1+15+0）。此时，[B,X,A]要优于[B,B,X]。



到达下一个 B 点也有两条路径：一条是从[B,X]出发经 A2，然后再穿越 X2，这条路径的长

度为 14（14=4+10+0），还有一条是从[B]出发，经 B2 直接达到，路径长度为 16（16=1+15）。这里，第一条是最优路径：[B,X,A,X]。

当整个过程完成时，剩下两条路径 A 或者 B，长度都是 14。每一条都是正确的选择。由于最后一个 X 路段的长度为 0，所以最后总会有两条同样长度的路径。通过递归地解决问题，我们可以保证最后一定能得到最短的路径。还不错，对吧？

通过这种巧妙的方法，我们得到了编写递归函数所需要的基本逻辑部件。我们可以来实现了，不过我保证，需要自己编写的递归函数只有非常少的几个。我们会使用 fold 操作。

> **注意** 虽然前面展示的 fold 操作的使用和构造都是基于列表的，但是 fold 操作代表的其实是一个更为广泛的概念，即使用累加器对一个数据结构进行遍历。因此，fold 操作可以在树、字典、数组、数据库表等上面实现。有时，尝试使用 map 和 fold 这样的抽象是有益的，因为有了这些抽象，在对代码逻辑所使用的数据结构进行更改时，就容易多了。

8.2.2 编写代码

现在进展到哪一步了？啊，是的！我们已经有了一个作为输入的文件。file 模块是最好的文件处理工具。它包含有许多在其他编程语言中常见的文件处理函数（设置权限、移动文件、重命名文件、删除文件等）。

file 模块中也包含有一些常见的文件读写函数，如 file:open/2 和 file:close/1——它们的功能如其名字所示（打开和关闭文件！）：file:read/2——读取文件的内容（可以作为字符串，也可以作为二进制），file:read_line/1——读取一行以及 file: position/3——把打开文件的位置指针移动到一个指定位置。

这个模块中还包含一组快捷函数，如 file:read_file/1（打开并以二进制方式读取整个文件内容）、file:consult/1（打开并把文件解析成 Erlang 数据项）、file:pread/2（改变位置指针，然后读取内容）以及 file:pwrite/2（改变位置指针并写入内容）。

有了这些可用的函数，找到一个读取 road.txt 文件的函数就很容易了。因为我们知道，路径信息相对而言比较少，所以选择使用 file:read_file("road.txt").：

```
1> {ok, Binary} = file:read_file("road.txt").
{ok,<<"50\r\n10\r\n30\r\n5\r\n90\r\n20\r\n40\r\n2\r\n25\r\n10\r\n8\r\n0\r\n">>}
2> S = string:tokens(binary_to_list(Binary), "\r\n\t ").
["50","10","30","5","90","20","40","2","25","10","8","0"]
```

注意，在这个例子中，我们把空格（" "）和制表符（"\t"）也作为有效的分词符号，这样可以把文件写成这种格式："50 10 30 5 90 20 40 2 25 10 8 0"。

接下来，我们需要把列表中的字符串转换成整数。

```
3> [list_to_integer(X) || X <- S].
[50,10,30,5,90,20,40,2,25,10,8,0]
```

我们创建一个新模块 *road.erl*，在其中写入如下代码：

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

main() ->
    File = "road.txt",
    {ok, Bin} = file:read_file(File),
    parse_map(Bin).

parse_map(Bin) when is_binary(Bin) ->
    parse_map(binary_to_list(Bin));
parse_map(Str) when is_list(Str) ->
   [list_to_integer(X) || X <- string:tokens(Str,"\r\n\t ")].
```

函数 main/0 负责读入文件的内容，并把内容传给函数 parse_map/1。由于我们使用了函数 file:read_file/1 来读取文件 road.txt 的内容，所以得到的是二进制数据。因此，我们让函数 parse_map/1 同时匹配了列表和二进制类型。如果是二进制类型，就把它转换成字符串（列表）后再次调用这个函数（分隔字符串的函数只接受列表）。

接下来，要把解析后的数据重新组织成之前描述过的格式：{A,B,X}。很不幸，并没有一种通用的方法可以一次从列表中取出 3 个元素，因此需要我们自己在递归函数中通过模式匹配来完成这项工作：

```
group_vals([], Acc) ->
    lists:reverse(Acc);
group_vals([A,B,X|Rest], Acc) ->
    group_vals(Rest, [{A,B,X} | Acc]).
```

这是一个标准的尾递归函数，并没有什么复杂的逻辑。下面需要修改 parse_map/1 去调用这个函数：

```
parse_map(Bin) when is_binary(Bin) ->
    parse_map(binary_to_list(Bin));
parse_map(Str) when is_list(Str) ->
    Values = [list_to_integer(X) || X <- string:tokens(Str,"\r\n\t ")],
    group_vals(Values, []).
```

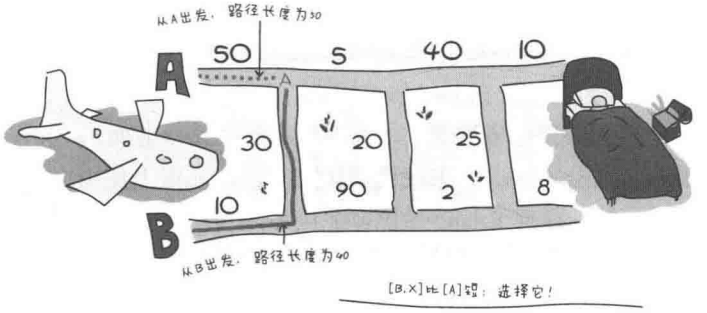
我们来编译一下，并运行一下看看是不是一切正常。

```
1> c(road).
{ok,road}
2> road:main().
[{50,10,30},{5,90,20},{40,2,25},{10,8,0}]
```

很好，结果看起来是正确的。现在，fold 操作中使用的函数要操作的数据部分已经得到了。接下来，要找到一个好的累加器。

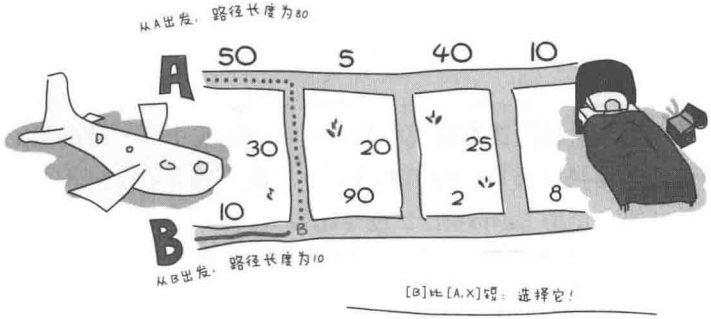
在累加器的选择方面，我发现有一个最容易的思考方法，就是假想自己处于一个正在运行的算法过程的中间。对这个特定的问题来说，可以假想自己正在试图寻找第 2 个三元组（{5,90,20}）的最短路径。为了决定出最优路径，需要知道前一个三元组的结果。很幸运，我

们知道如何得到这个结果，因为此时不需要累加器，并且所有的计算逻辑也都已经知道了。对于A点来说，计算如下：



我们选择两条路径中更短的那一条。

对于B来说，选择方法一样：



现在，我们知道A侧的当前最优路径是[B, X]。我们也知道它的长度是40。B侧的最优路径就是[B]，长度是10。我们可以使用这些信息，用同样的逻辑寻找A、B的下一个最短路径，只要把之前的路径长度值算进来就行了。

我们还需要已经走过的路径信息，展示给用户看。那么，现在就需要两条路径信息（一条A路径，一条B路径）和两个累加的路径长度，因此累加器可以采用这种格式：{{*DistanceA*, *PathA*}和{*DistanceB*, *PathB*}}。这样，`fold`操作的每次迭代都会访问所有这些状态，在迭代结束时，我们会把计算的结果展示给用户。

现在函数所需要的参数已经完整了：{A,B,X}元组和格式为{{*DistanceA*,*PathA*}, {*DistanceB*,*PathB*}}的累加器。

我们来实现这个函数，它会返回一个累加器，如下：

```
shortest_step({A,B,X}, {{DistA,PathA}, {DistB,PathB}}) ->
    OptA1 = {DistA + A, [{a,A}|PathA]},
    OptA2 = {DistB + B + X, [{x,X}, {b,B}|PathB]},
    OptB1 = {DistB + B, [{b,B}|PathB]},
    OptB2 = {DistA + A + X, [{x,X}, {a,A}|PathA]},
    {erlang:min(OptA1, OptA2), erlang:min(OptB1, OptB2)}.
```

其中，OptA1是A侧的第一条路径选择（走路段A），OptA2是A侧的第二条路径选择（走

路段 B，然后再穿越路段 X）。变量 OptB1 和 OptB2 类似，计算的是 B 侧的路径选择。最后，函数返回一个累加器，其中包含着选择的最优路径。

在代码中保存路径信息时，我决定使用[{x,X}]这种形式而不是 [x]，这只是因为可以让用户知道每条路段的长度，感觉会更好一些。同时，我们是以反向的方式累加路径信息的（{x,X}放在{b,B}前面）。这是因为我们处在 fold 操作中，它是一个尾递归。由于使用了这种累加方式，整个列表都是反向的，因此必须把最近走过的路段放在其他路段前面。

最后，我们使用 erlang:min/2 来找出最短路径。在元组上使用这样一个比较函数听起来会有些奇怪，但是，请记住，任何 Erlang 数据项之间都是可以比较的！因为路径长度是元组的第一个元素，所以可以用这个函数对其进行排序。

剩下来要做的就是把这个函数放到一个折叠操作中：

```
optimal_path(Map) ->
    {A,B} = lists:foldl(fun shortest_step/2, {{0,[]}, {0,[]}}, Map),
    {_Dist,Path} = if hd(element(2,A)) =/= {x,0} -> A;
                      hd(element(2,B)) =/= {x,0} -> B
                   end,
    lists:reverse(Path).
```

fold 操作结束时，两条路径应该具有同样的长度，不过其中有一条的最后路段是{x,0}。代码中的 if 语句会检查两条路径中最后走过的路段，并返回不包含{x,0}的那一条。选择路段数最少的那条路径也是可以的（使用 length/1 进行比较）。找到了最短路径后，会把它反转（它是以尾递归的方式构建的，必须要反转）。然后，你可以把它公示于众，也可以选择保密，并最先向叔叔告别。当然，我们得先修改 main 函数，让它调用 optimal_path/1。修改完后进行编译。

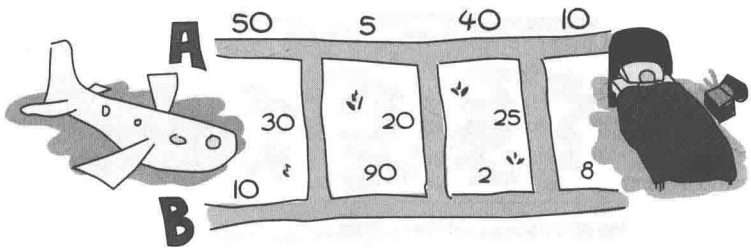
```
main() ->
    File = "road.txt",
    {ok, Bin} = file:read_file(File),
    optimal_path(parse_map(Bin)).
```

现在，按如下方式进行测试：

```
1> c(road).
{ok,road}
2> road:main().
[{b,10},{x,30},{a,5},{x,20},{b,2},{b,8}]
```

啊，看！答案正确。太棒了！

把路径可视化出来是这样的：



8.2.3 不使用 Erlang shell 运行程序

怎样做才能让程序更好用呢？答案是能够脱离 Erlang shell 运行程序。为了做到这一点，需要再次更改 mian 函数：

```
main([FileName]) ->
    {ok, Bin} = file:read_file(FileName),
    Map = parse_map(Bin),
    io:format("~p~n",[optimal_path(Map)]),
    erlang:halt().
```

现在，main 函数有了一个参数，需要从命令行接收这个参数。我们还增加了一个函数调用 erlang:halt/0，这个调用会关闭 Erlang VM。我们把调用 optimal_path/1 放到 io:format/2 中，因为这是唯一能在 Erlang shell 之外显示文本的方法。

更改完成后，road.erl 文件的内容如下（去掉了一些注释）：

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

main([FileName]) ->
    {ok, Bin} = file:read_file(FileName),
    Map = parse_map(Bin),
    io:format("~p~n",[optimal_path(Map)]),
    erlang:halt(0).

%% 把字符串转换成易读的地图三元组列表
parse_map(Bin) when is_binary(Bin) ->
    parse_map(binary_to_list(Bin));
parse_map(Str) when is_list(Str) ->
    Values = [list_to_integer(X) || X <- string:tokens(Str,"\r\n\t ")],
    group_vals(Values, []).

group_vals([], Acc) ->
    lists:reverse(Acc);
group_vals([A,B,X|Rest], Acc) ->
    group_vals(Rest, [{A,B,X} | Acc]).

%% 选择最优路径
optimal_path(Map) ->
    {A,B} = lists:foldl(fun shortest_step/2, {{0,[]}, {0,[]}}, Map),
    {_Dist,Path} = if hd(element(2,A)) =/= {x,0} -> A;
                      hd(element(2,B)) =/= {x,0} -> B
                   end,
    lists:reverse(Path).

%% 问题解决的核心逻辑
%% 把三元组{A,B,X}转换成{DistanceSum, PathList}，其中A,B,X是距离，a,b,x是可能的路径
shortest_step({A,B,X}, {{DistA,PathA}, {DistB,PathB}}) ->
    OptA1 = {DistA + A, [{a,A}|PathA]},
    OptA2 = {DistB + B + X, [{x,X}, {b,B}|PathB]},
```

```
    OptB1 = {DistB + B, [{b,B}|PathB]},
    OptB2 = {DistA + A + X, [{x,X}, {a,A}|PathA]},
    {erlang:min(OptA1, OptA2), erlang:min(OptB1, OptB2)}.
```

可以这样来运行代码：

```
$ erlc road.erl
$ erl -noshell -run road main road.txt
[{b,10},{x,30},{a,5},{x,20},{b,2},{b,8}]
```

很好，得到了正确答案！在 Erlang shell 之外运行程序需要的工作就是这些。当然，你还可以编写一个 bash 批处理脚本把这个命令行包装成一个可执行文件，也可以用 escript 命令（提供了脚本支持）完成这项工作。

从这两个例子中可以看出，把问题分解成一些小的、可以独立解决的部分，之后再把它们合并在一起，可以更容易地解决问题。还有一点也很重要，就是在真正理解问题之前，不要立即去解决它们，因为从长远来看，这通常会造成很多浪费。最后，编写一些测试总是有用的。测试可以保证一开始工作正常的代码，即使在以后对代码的实现细节进行了更改，还能保持正常工作。

使用 escript

Erlang 的 escript 命令提供了一种在不直接启动 erl 的情况下运行 Erlang 程序的简单方法。简单来讲，escript 命令以一个模块为参数，然后无需编译即可解释执行这个模块。

模块的结构和前面介绍的类似，不过需要更改一下模块的头部说明。不能再使用 -module(*Name*) 属性，而是要使用如下格式：

```
#!/usr/bin/env escript
%% -*- erlang -*-
%%! -pa 'ebin/' [Other erl Arguments]
main([StringArguments]) ->
    ...
```

在脚本启动时，会自动调用 main/1 函数，脚本的启动命令可以是 ./script-name.erl 或者 escript script-name.erl（后者更适用于 Windows 平台）。模块会像一个普通脚本一样运行。

如果你想得到 escript 的便利性，但是不想解释执行代码（会慢一些），更希望能够执行编译后的结果，那么只需在文件中的某个地方增加 -mode(compile). 模块属性即可。

关于 escript 的更多信息，请参阅 Erlang 自带的相关文档，也可以在线阅读：http://erlang.org/doc/man/escript.html。

第 9 章

常用数据结构简介

现在，你可能对 Erlang 中函数部分的内容非常了解了，能够毫无问题地读懂许多程序。并且，我们在第 8 章中也学习了如何用函数式思维解决问题，但是，如果现在让你去考虑如何构建一个真实、有用的程序，我敢说还是有点困难的。嗯，这也是我自己在学习 Erlang 过程中的感受——如果你做得比我好，那么祝贺你了！

到目前为止，我们介绍了不少内容，包括大部分的基本数据类型、shell、模块和函数（含递归）的编写、各种编译方法、程序的控制流、异常处理以及如何提炼一些通用的操作。我们还介绍了如何用元组、列表以及一棵没有完整实现的二叉搜索树来存放数据。不过，Erlang 标准库中还有一些可供程序员使用的其他数据结构没有介绍。本章会填补这个空白，我们会介绍记录、键/值存储、集合、有向图以及队列。

9.1 记录

首先，记录（record）是一种拼凑物。它们是在语言实现完毕后临时添加上去的，因此使用起来有些不方便。但是，如果数据结构比较小，并且想直接通过名字去访问属性字段，那么使用记录还是很合适的。记录的这种使用方式和 C 语言中的结构很像。

9.1.1 定义记录

记录以模块属性的形式声明：

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

-record(robot, {name,
                type=industrial,
                hobbies,
                details=[]}).
```

在上面的代码中，我们定义了一个表示机器人的记录，有4个字段：name、type、hobbies和details。其中，type和details字段具有默认值，分别是industrial和[]。

下面的代码展示如何在records模块中创建一个记录实例：

```
first_robot() ->
    #robot{name="Mechatron",
           type=handmade,
           details=["Moved by a small man inside"]}.
```

运行一下代码：

```
1> c(records).
{ok,records}
2> records:first_robot().
{robot,"Mechatron",handmade,undefined,
       ["Moved by a small man inside"]}
```

哇！看来记录是一种拼凑实现了！Erlang记录只是元组之上的语法糖。还好，有一种方法可以维持记录存在的错觉。Erlang shell提供了一条命令rr(*Module*)，可以加载*Module*中定义的记录。用它来试试我们的records模块：

```
3> rr(records).
[robot]
4> records:first_robot().
#robot{name = "Mechatron",type = handmade,
       hobbies = undefined,
       details = ["Moved by a small man inside"]}
```

> **注意**　rr()函数除模块名之外还可以接收其他参数。它可以以通配符（如rr("*")）作为参数，还可以以一个列表作为第二个参数，用来指定要加载哪些记录。

啊，很好！这样记录用起来就更容易了。注意，在first_robot/0中，我们没有给hobbies字段赋值，在记录定义中也没有给它提供默认值。Erlang会默认把它的值设置成undefined。

为了展示一下robot记录定义中默认值的行为，请编写如下函数：

```
car_factory(CorpName) ->
   #robot{name=CorpName, hobbies="building cars"}.
```

现在来运行一下：

```
5> c(records).
{ok,records}
6> records:car_factory("Jokeswagen").
#robot{name = "Jokeswagen",type = industrial,
       hobbies = "building cars",details = []}
```

现在我们拥有了一台工业机器人，它喜欢把时间花在制造汽车上。

Erlang shell 中的其他记录相关函数

除了 rr() 外，Erlang 还提供了其他一些函数，可以在 shell 中处理记录。

- rd(*Name*, *Definition*)，定义一个记录，方式和模块中使用的-record(*Name*, *Definition*)类似。
- rf()，“卸载”所有记录。
- rf(*Name*)或者 rf([*Names*])，删除指定的记录定义。
- rl()，把 shell 中当前定义的所有记录打印出来，打印采用的记录格式非常易于复制、粘贴到模块中。如果只想打印某些特定的记录，可以使用 rl(*Name*)或者 rl([*Names*])。

9.1.2 读取记录字段值

只能向记录中写入值并不是非常有用。我们需要能够从记录中读取值。总的来说，有两种读取记录值的方法：专用的点语法以及模式匹配。假设已经加载了描述机器人的 robot 记录定义，我们先来看看如何使用点语法。

```
5> Crusher = #robot{name="Crusher", hobbies=["Crushing people","petting cats"]}.
#robot{name = "Crusher",type = industrial,
       hobbies = ["Crushing people","petting cats"],
       details = []}
6> Crusher#robot.hobbies.
["Crushing people","petting cats"]
```

嗯——语法不是很漂亮。这是因为记录在本质上就是元组的缘故。因为记录是通过编译器技巧实现的，所以必须要包含记录名关键字以指定变量对应的具体记录，也就是 Crusher#robot.hobbies 中的#robot 部分。这很糟糕，但是没有其他办法。更糟糕的是，嵌套的记录更加丑陋：

```
7> NestedBot = #robot{details=#robot{name="erNest"}}.
#robot{name = undefined,type = industrial,
       hobbies = undefined,
       details = #robot{name = "erNest",type = industrial,
                        hobbies = undefined,details = []}}
8> (NestedBot#robot.details)#robot.name.
"erNest"
```

其中的括号并不是必需的。输入 NestedBot#robot.details#robot.name 也是合法的。不过，为了向后兼容（R14A 之前的版本），以及满足我个人的偏好，我倾向于使用带有括号的版本，因为这种方式可以使代码更加易读。

下面的例子进一步说明了记录和元组之间的依赖关系：

```
9> #robot.type.
3
```

这条语句输出了 type 在底层元组中的元素位置编号。

记录还提供了一种补偿性的特性，可以在函数头中对它们进行模式匹配，也可以用在卫语句中。为了展示一下这种特性的用法，请在文件顶部定义一个新记录，然后在定义下面增加如下函数：

```
-record(user, {id, name, group, age}).

%% 使用模式匹配进行过滤
admin_panel(#user{name=Name, group=admin}) ->
    Name ++ " is allowed!";
admin_panel(#user{name=Name}) ->
    Name ++ " is not allowed".

%% 可以随意扩展 user 记录，函数无需修改
adult_section(U = #user{}) when U#user.age >= 18 ->
    %% 显示不适合写出来的内容
    allowed;
adult_section(_) ->
    %% 重定向到 Sesame Street 教育网站
    forbidden.
```

函数 admin_panel/1 中展示了把变量和记录字段进行绑定的语法(也可以同时把多个变量绑定到多个字段)。

注意，在函数 adult_section/1 中，为了把整条记录绑定到一个变量，需要使用 *SomeVar = #some_record{}* 这种方式。

接下来，我们照常进行编译。

```
10> c(records).
{ok,records}
11> rr(records).
[robot,user]
12> records:admin_panel(#user{id=1, name="ferd", group=admin, age=96}).
"ferd is allowed!"
13> records:admin_panel(#user{id=2, name="you", group=users, age=66}).
"you is not allowed"
14> records:adult_section(#user{id=21, name="Bill", group=users, age=72}).
allowed
15> records:adult_section(#user{id=22, name="Noah", group=users, age=13}).
forbidden
```

这个例子表明，我们不需要去匹配元组中的所有元素，甚至在编写函数时也不必知道元组中到底有多少元素。如果我们只需要 age 和 group 字段，就可以只去匹配它们，不用关心元组中的其他剩余元素。如果我们使用的是一个普通的元组，函数需要定义成 function({record, _, _, ICareAboutThis, _, _}) -> ...的形式。那么，每当有人决定向元组中增加一个元素时，其他人（可能会对此很生气）就必须更新所有使用这个元组的函数。

9.1.3 更新记录

下面的函数说明了如何更新一条记录（如果不能更新，就不太有用）。

```
repairman(Rob) ->
    Details = Rob#robot.details,
    NewRob = Rob#robot{details=["Repaired by repairman"|Details]},
    {repaired, NewRob}.
```

对它进行编译：

```
16> c(records).
{ok,records}
17> records:repairman(#robot{name="Ulbert", hobbies=["trying to have feelings"]}).
{repaired,#robot{name = "Ulbert",type = industrial,
                 hobbies = ["trying to have feelings"],
                 details = ["Repaired by repairman"]}}
```

可以看出，robot 记录被修改了。这里的记录修改语法有些特殊。看起来记录好像被就地更改了（`Rob#robot{Field=NewValue}`），不过，这只是一种为了调用底层 `erlang:setelement/3` 函数的编译技巧。

9.1.4 共享记录定义

如果记录很有用，但是又想避免重复定义记录，那么可以使用头文件在多个模块间共享记录定义，这是 Erlang 程序员常用的方法。Erlang 的头文件和 C 中的类似。头文件就是会被增加到模块中的一小段代码，就像一开始就把它写在那里一样。

创建一个名为 *records.hrl* 的文件，其中包含如下内容：

```
%% 这是一个.hrl（头）文件
-record(included, {some_field,
                   some_default = "yeah!",
                   unimaginative_name}).
```

要在模块 `records.erl` 中包含这个头文件，只需在其中增加如下一行代码：

```
-include("records.hrl").
```

然后，为了测试，增加如下函数：

```
included() -> #included{some_field="Some value"}.
```

像往常一样对其进行编译。

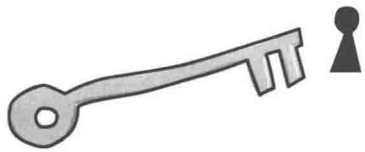
```
18> c(records).
{ok,records}
19> rr(records).
[included,robot,user]
20> records:included().
#included{some_field = "Some value",some_default = "yeah!",
          unimaginative_name = undefined}
```

太好了！关于记录的内容讲完了。正如你所看到的，记录的语法不是很漂亮，它们只是拼凑

出来的语法糖，但是，对于代码的可维护性来说，它们还是比较重要的。

> **警告**　在开源软件中，常常会看到它们使用这里介绍的方法把记录定义在一个项目范围可见的.hrl 文件中，在多个模块间共享。虽然我觉得有义务正式介绍一下这种用法，但是我强烈建议把所有记录定义局部化在一个模块中。如果其他模块想查看某个记录的内部结构，可以编写函数来访问这个记录的字段，尽量保持记录内部实现的私有。这有助于防止名字冲突，避免代码升级时出现问题，并且可以从总体上提升代码的可读性和可维护性。

9.2　键/值存储



在第5章中，我们编写了一棵树，并用它作为通讯录的键/值存储。这个通讯录很难用。我们无法进行删除，也不能把它转换成其他有用的结构。它作为一个递归示例还不错，但是也就仅此而已。

现在，我们来学习一组有用的数据结构和模块，它们可以以键/值的方式进行数据存储。我不会给出每一个函数的定义，也不会展示完整的例子或者详细介绍所有模块，因为可以在 Erlang 文档中容易地找到这方面的信息。你可以把我当成“肩负唤起大家对于 Erlang 键/值存储意识使命的那个人”（这个名字听起来不错——我正好需要一条这样的丝带）。

9.2.1　小数据量存储

如果要存储的数据量比较小，那么有两种主要的数据结构可供使用：*属性列表*（proplist）和*有序字典*（orddict）。

1. *属性列表*

属性列表就是形如`[{Key,Value}]`的元组列表。属性列表是一种比较奇怪的数据结构，因为它们只需要满足这一条规则。事实上，这个规则非常宽松，列表中还可以包含布尔值、整数以及任何其他类型值。在此，我们只关心列表中的元素是由一个键和一个值组成的元组的情况。

可以通过 `proplists` 模块来处理属性列表。这个模块中包含 `proplists:delete/2`、`proplists: get_value/2`、`proplists:get_all_values/2`、`proplists:lookup/2` 和 `proplists:lookup_all/2` 之类的函数。你可以从 Erlang 文档中找到关于它们的详细定义。

你会在文档中发现，该模块中没有向列表中增加元素或者更新列表元素的函数。这表明属性列表是一个非常不严格的数据结构。事实上，属性列表通常更适合于表示某种东西的属性描述列表。例如，我们可以用属性列表`[{name, buddy}, {race,husky}, friendly]`来描述一条狗，其中值 `friendly` 等同于`{friendly, true}`。

如果想向属性列表中增加一个元素，只能使用 `cons` 操作符手工插入元素（`NewList = [NewElement| OldList]`）。这种做法甚至可以用来进行元素更新，因为 `proplists` 模块会按照顺序遍历列表，一旦找到匹配的元素就停止查找。如果更新操作比较频繁，那么可以使用 `lists:keyreplace/4` 之类的函数，从而避免属性列表随着时间变得越来越长。在这么小的一个数据结构上同时使用两个模块不是太好，不过，由于属性列表的定义很不严格，所以一般仅用

于处理配置列表。

2. 有序字典

如果在存储小量数据时想使用更完善些的键/值存储，可以考虑使用 orddict 模块。有序字典是具有一些规范要求的属性列表。每个键都只能出现一次。这个列表是排过序的，因此平均来说，查找速度要快一些。所存储的数据项必须是严格的{*Key*, *Value*}形式。不能像属性列表那样，把有序字典当成列表来操作，必须使用 orddict 模块的函数接口来进行所有需要的操作。

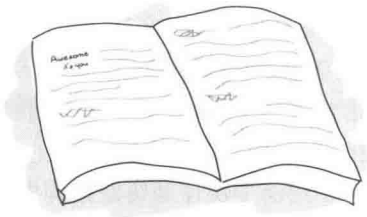
常用的 CRUD（创建、读取、更新和删除）函数包括 orddict:store/3、orddict:find/2（当不知道字典中是否包含给定主键时使用）、orddict:fetch/2（当知道主键存在或者必须存在时使用）以及 orddict:erase/2。可以使用 orddict:new/0 或者 orddict:from_list/1 创建一个有序字典。同样，你可以在 Erlang 文档中找到这些函数的定义。

> **警告** 在创建和操作有序字典时，可能会试图直接去修改键/值列表，但是为了避免出现顺序错误，应该永远使用 orddict 模块所提供的函数进行相应操作。

一般来讲，对于小于 75 个元素的数据量来说，有序字典在复杂性和效率之间做到了很好的平衡（参见我做的基准测试 keyval_benchmark.erl，这个文件在本书附带的源代码中。）。如果超过了这个数据量，应该切换到后面介绍的其他键/值存储。

9.2.2 大数据量存储：字典和通用平衡树

Erlang 提供了两种用于大数据量存储的键/值数据结构：字典（dict）和通用平衡（GB）树。字典的接口和有序字典完全一样：dict:store/3、dict:find/2 以及 dict:fetch/ 2、dict:erase/2。字典中也具有有序字典中的所有其他函数，如 dict:map/2 和 dict:fold/2（在处理整个数据结构时很有用！）。因此，当有序字典需要向上伸展时，字典是非常好的选择。



GB 树是由 gb_trees 模块实现的，其中包含的函数要比 dict 中多，在数据结构的使用上提供了更多的直接控制手段。gb_trees 有两种主要的工作模式：一种针对彻底了解自己数据的情况（称之为智能模式），还有一种针对不能对数据做假设的情况（称之为简单模式）。在简单模式中，数据操作函数为 gb_trees:enter/2、gb_trees:lookup/2 和 gb_trees:delete_any/2。在智能模式中，相应的函数为 gb_trees:insert/3、gb_trees:get/2、gb_trees:update/3 和 gb_trees:delete/2。还有一个 gb_trees:map/2 函数，它和 lists:map/2 等价，不过操作的对象是树（在需要时，它非常好用）。

简单模式函数相比智能模式函数存在劣势的原因在于 GB 树是平衡树，每当插入一个新元素时（或者删除一批元素时），树都需要平衡自己。这需要时间和内存（即使是些无用的检查，最后发现无需变化，也要花时间去确定树已经处于平衡状态）。智能模式函数默认已知晓键值在树中的存在情况。有了这种假设，就可以略过所有的安全检查，从而达到更快的执行速度。

保持冷静

如果需要只以数值为主键的数据结构时该怎么办呢？对此，在大部分语言中通常会使用数组。Erlang 中也有数组。在 Erlang 数组中，可以使用数值索引来访问元素，并且可以对整个数组进行折叠（fold）操作，同时忽略那些未定义的数组位置。不过，基本上没人使用 Erlang 数组。

和命令式语言中的数组不同，Erlang 数组的插入和查询操作无法在常量时间内完成。相反，Erlang 数组是一种持久化的（persistent）数据结构，因为它们不允许破坏性更新。因此，它们通常会比那些允许破坏性赋值语言中的数组慢一些。了解并使用那种数组的人常常会以一种明确的风格把它用在一些特定的算法中。Erlang 数组不能做到这一点。因此，人们基本上不使用数组。

当需要进行矩阵处理或者其他需要数组的工作时，Erlang 程序员往往会使用端口技术来让其他语言去做这种高计算量的工作，也可以使用 C 节点、链入式驱动（linked-in driver）或者本地实现函数（NIF）。更多的细节请参见 Erlang 文档。

什么时候使用 `gb_trees` 模块而不是 `dict` 呢？嗯，对此，很难有一个明确的判断。基于我编写的基准测试模块（keyval_benchmark.erl）的结果，GB 树和字典在很多方面的性能相似。不过，基准测试表明，字典的读取性能最好，GB 树在其他操作方面要更快一些。

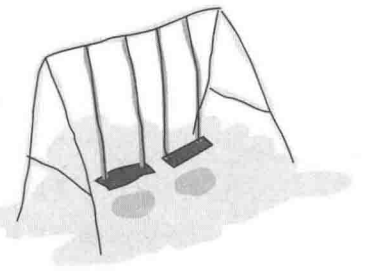
还要注意，字典有一个折叠操作函数，而 GB 树没有。不过，GB 树有一个迭代操作函数，它会返回树的一小部分信息，可以基于此调用 `gb_trees:next(Iterator)` 来按照顺序得到后面的值。这意味着，在访问整棵 `gb_trees` 树时，没有一个通用的折叠操作，只能自己来编写递归函数。另一方面，使用 `gb_trees` 可以通过函数 `gb_trees:smallest/1` 和 `gb_trees:largest/1` 快速地访问结构中的最小和最大元素。这是因为 GB 树保留了其中所有元素的顺序，从最小到最大。而字典中没有保留这种顺序。因此，如果想按照顺序遍历键/值存储，那么 GB 树是一个好的选择。

因此，应用的需要决定了键/值库类型的选择。你需要考虑诸多因素，例如，要存储的数据量的大小以及需要做的操作。度量、性能分析、基准评测之后再做决定。

注意　还有些特殊的键/值存储，用于处理更大规模的数据。这些存储库有 ETS 表、Dets 表以及 Mnesia 数据库。它们往往应用于多进程和并发的场景，因此我们会在第 25 章中对它们进行介绍。我在这里提起它们只是为了激发你的求知欲，同时也给那些有兴趣的读者一个参考。

9.3　集合

如果你曾经在数学课中学习过集合论，那么应该大概知道集合能做什么。如果没有学习过，你可能会想略过这一节的内容。



集合是值唯一的一组元素，集合之间可以进行比较和其他操作——判断元素是否属于两个集合、不属于任何一个集合或者只在其中一个集合中等。还有一些高级操作，用来定义关系、在这些关系上进行操作以及其他更多的功能。在此，我不会对集合理论进行深入的介绍，只会简单讲讲有哪些可用的模块。

Erlang 中有 4 个主要的集合处理模块。一开始可能会感觉有些奇怪，不过这是因为实现者们觉得并不存在一种“最好的”集合实现方法。这 4 个模块分别是：

ordsets

ordsets 模块集合被实现为一种有序列表。它们主要适用于小集合，是最慢的一种集合，不过它的实现是所有集合中最简单、最容易理解的一种。该模块中的一些标准函数有 ordsets:new/0、ordsets:is_element/2、ordsets:add_element/2、ordsets:del_element/2、ordsets:union/1 和 ordsets:intersection/1。

sets

sets（模块名）实现使用的底层数据结构和 dict 使用的相似。sets 模块的接口和 ordsets 完全一样，不过支持的数据规模要大一些。和 dict 一样，sets 更擅长读密集型的处理，如检查某个元素是否在集合中。

gb_sets

gb_sets 模块的底层实现结构是一棵 GB 树，和 gb_trees 模块使用的类似。gb_sets 和 sets 的关系与 gb_trees 和 dict 的关系一样：在非读取操作方面，gb_sets 要更快一些，提供的控制手段也更多一些。gb_sets 在实现了和 sets、ordsets 同样接口的同时，还增加了其他一些函数。和 gb_tress 一样，gb_sets 中也分智能模式函数和简单模式函数，也有迭代器以及对最小值和最大值的快速访问函数。

sofs

可以使用 sofs 模块创建集合的集合。这个模块使用有序列表实现，这个列表被放置在一个包含元数据信息的元组中。如果想完全掌控集合和集合族之间的关系，强制集合类型或者有其他类似要求时，可以使用这个模块。当想使用数学意义上的集合概念，而又不仅仅是值唯一的元素组时，这个模块非常有用。

有这么多的选择会让人有点困惑。来自 Erlang/OTP 团队的 Björn Gustavsson，同时也是 Wings 3D 项目的程序员，建议在大多数情况下尽量使用 gb_sets，当需要一种清晰的表示，想在自己的代码中操纵这种表示时，使用 ordsets，当需要=:=操作符时，使用 sets（参见 http://erlang.org/pipermail/erlang-questions/2010-March/050332.html）。

总之，和键/值存储一样，最好的方法通常都是去做基准测试，看看哪种选择最适合自己的应用。

保持冷静

虽然说有这么多种不同的集合实现应该是一件好事，但是其中有些实现细节却令人非常沮丧。例如，gb_sets、ordsets 和 sofs 都使用==操作符来进行值的比较，如果有两个数 2 和 2.0，它们会被认为是相同的数。

但是，在 sets 模块中使用了=:=操作符，这意味着，不能随意在不同实现之间进行切换。在有些情况下，你需要一种精确的行为，此时就无法拥有多重实现的好处了。

9.4 有向图

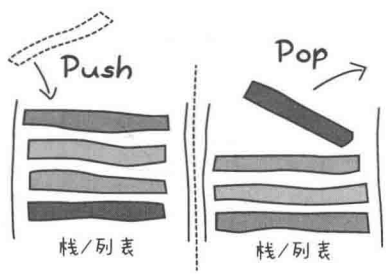
另外一个和数学关系密切的数据结构是有向图（directed graph）。有向图在 Erlang 中被实现为两个模块：digraph 和 digraph_utils。digraph 模块主要实现了有向图的构造和修改功能——操作边和顶点、寻找路径和环等。digraph_utils 模块则实现了图的遍历功能（后序和前序），环、树形图[①]以及树性质检测，寻找邻居顶点等功能。

由于有向图和集合论关系密切，因此 sofs 模块中包含了一些函数，用来在集合族和有向图之间进行双向转换。

因为有向图的构建方式，在对图论或者集合论不了解的情况下很难真正用好这个模块。如果你对自己的问题很了解，并且也有兴趣更深入地学习这些模块，在 Erlang 文档中可以很容易地找到相关的内容。

9.5 队列

queue 模块实现了一个双向、先进先出（FIFO）的队列（queue）。队列的实现方式在这里可以简单说明一下：使用了两个列表（在这个上下文中是栈），分别用来进行元素的快速入队和出队。



因为单个列表无法同时在两端做到高效的元素增加和删除（只能在头部做到快速增加和删除），所以 queue 模块采用了这样的实现思路：如果有两个列表，就用一个列表来增加元素，另一个列表用来删除元素。于是，一个列表变成了队列的一端，可以向其中增加元素，另一个列表变成了队列的另外一端，可以从中取出元素。当第二个列表为空时，可以把第一个列表反转，并把它变成提取元素的新列表。对于队列生存期间的所有操作的平均性能来说，这种实现是比较高效的。

queue 模块中的函数被分成 3 组具有不同复杂性的接口（或者 API）。

原始 API

原始 API 中包含了队列实现的基础函数，包括创建空队列的 new/0，插入新元素的

① 树形图（arborescence）和树为图论术语，表示具有特定性质的图，详见 Erlang 文档。——译者注

in/2，以及移除元素的 out/1。其中也包含有像把队列转换成列表、反转队列、检查某个特定值是否在队列中之类的函数。

扩展 API

扩展 API 主要增加了一些内省能力和灵活性。可以用它来进行一些诸如在不移除第一个元素的情况下查看队列的头元素（get/1 或者 peek/1）、直接移除元素而不关心它的值（drop/1）之类的操作。虽然这些函数并不是队列必要的操作，但是总体来讲它们还是很有用的。

Okasaki API

Okasaki API 有点奇怪。它的实现来自 Chris Okasaki 的 *Purely Functional Data Structures*（Cambridge University Press，1999）一书。这个 API 所提供的操作函数和其他 API 中的类似，不过，其中有些函数名是反着写的，整个实现也有些奇怪。除非有特殊的原因，否则不要使用这个 API。

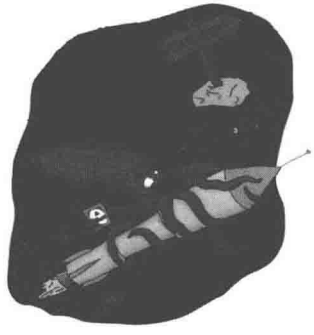
一般来讲，当需要保证先来的元素必须要先被处理时，可以使用队列。到目前为止，我所展示的例子主要是用列表来作为累加器，用完后需要进行反转操作。对于那些不能一次反转所有元素，以及元素增加频繁的情况，queue 模块是非常适用的。（嗯，应该先测试和度量。永远要先测试和度量！）

9.6 小结

Erlang 中最常用的几个数据结构就介绍到这里。感谢你专注地全程参与。Erlang 中还有一些解决其他问题的数据结构。在本章中，我们学习了在常见的 Erlang 应用场景中，你最可能会用到的那些数据结构。我希望你能够去浏览一下标准库和其他扩展库，以了解更多的信息。

如果知道我们已经完成了 Erlang 的顺序（函数式）编程之旅，你也许会感到高兴。我知道，很多人接触 Erlang 是为了了解那些和并发、进程等相关的内容。这种想法可以理解，因为这正是 Erlang 真正闪光之处。Erlang 提供了监督树、漂亮的错误处理、分布式支持，以及其他更多的东西。我自己已经迫不及待地想编写这方面的内容了，因此我猜有些读者肯定也等不及要阅读这些内容。

但是，在学习 Erlang 的并发特性之前，熟练掌握 Erlang 的函数式编程部分是很必要的。现在，我们可以去学习新概念了。出发！



第 10 章

并发编程漫游指南

在 21 世纪开端的那片未曾标明的寂静虚空中，存在着人类知识文明的一个小片段。在这个小片段中，有一个非常微不足道的、沿袭了冯诺依曼架构的知识领域，这个领域非常的原始，以至于仍然认为 RPN 计算器是一种极好的设计。

这个知识领域中有一个问题一直没有得到解决：这个领域的研究人员一直没能找出令人满意的并行软件编写方法。虽然提出了不少解决方案，但是这些方案基本上都是在考虑如何使用一小段称为锁（lock）或者互斥量（mutex）的逻辑块，这有些奇怪，因为从总体上来说，真正需要并行的并不是这一小段逻辑块。

因此，这个问题就一直遗留了下来。这令不少人感觉不快，大部分人感觉痛苦，甚至包括那些拥有 RPN 计算器的人。

越来越多的人开始觉得向编程语言中加入并行机制本身就是一个大的错误，认为程序根本就该永远运行在其初始线程中。

> **注意** 模仿《银河系漫游指南》的语气非常有趣。如果还没有看过这本书，建议看一下。非常好的一本书！

10.1 不必惊慌

嗨。今天（或者随便哪一天，你正在读这段话就行——甚至是明天），我会给大家介绍 Erlang 的并发特性。你之前也许已经了解过或者解决过并发相关的问题。或者对多核编程的出现也很好奇。无论如何，你之所以看这本书，有很大的可能是因为最近关于并发方面的讨论太多了。

不过，要先提醒一下大家：本章基本上全是理论。如果你现在头疼，或者很讨厌编程语言方面的历史，或者现在就想编程，那么最好略过这部分内容直接跳到本章的末尾，或者干脆直接去看下一章（下一章的内容实践性强一些）。

在本书一开始的引言部分中，我说过，Erlang 的并发是基于消息传递和 actor 模型的，并以

人和人之间只通过书信进行通信为例进行了说明。在本章的后面，我们会对并发进行更详细的介绍，不过，我们要先搞清楚并发（concurrncy）和并行（parallelism）之间的区别，这一点很重要。

虽然在许多地方，这两个词的含义相同，但是在 Erlang 中，并发指的是有许多独立运行的 actor，但是并不要求它们同时运行，而并行指的是多个 actor 在同时运行。我们会在本书中使用这种定义，不过如果看到其他地方或者其他人对这两个词使用了不同的定义，也不要觉得奇怪。计算机科学领域中对此也没有共识的定义。

Erlang 从一开始就支持并发，甚至可以追溯到一切都在单核处理器上运行的 20 世纪 80 年代。每个 Erlang 进程都被分配一个运行时间片，和多核系统出现之前的桌面应用很像。那时也是可以进行并行计算的，只要把代码跑到另外一台计算机上，并和第一台通信即可。不过，在这种环境中，只能有 2 个 actor 并行运行。现在，多核系统允许在单台计算机上进行并行计算（某些工业芯片可以有数十个核），Erlang 可以充分利用这种能力。

保持冷静

分清楚并发和并行之间的区别非常重要，因为很多程序员认为 Erlang 很早就支持多核计算机了，而实际上并没有那么早。Erlang 真正支持对称多处理（SMP）是在 2000 年中期，到了 2009 年才在 R13B 发布中完成了正确的实现。在这之前，通常需要禁用 SMP 以防止性能损失。为了在多核计算机上不使用 SMP 进行并行计算，需要启动多个 VM 实例。

一个有趣的事实是，因为 Erlang 的并发是通过隔离的进程实现的，所以向语言中增加真正的并行机制并不需要任何语言层面概念的改变。所有的更改都是在 VM 中透明完成的，程序员完全感受不到。

10.2 并发概念

让我们回到 Erlang 语言诞生的那段时间，当时 Erlang 语言本身的开发速度非常之快，并频繁地从使用 Erlang 开发电话交换机的工程师那里获取反馈。这种交互证明了基于进程的并发以及异步的消息传递机制可以很好地对工程师所面临的问题进行建模。并且，在 Erlang 之前，电话软件领域中就已经存在了某种趋向于并发的文化。这种文化来自 PLEX，爱立信公司早期创建的一门编程语言，以及 AXE，一款用 PLEX 实现的交换机。Erlang 沿袭了这个趋势，并试图对之前使用的工具进行改进。

Erlang 之所以得到认可，是因为它满足了一些需求。其中主要的几个是：能够向上伸展（scale up），可以跨多个交换机支持数千用户，具有很高的可靠性——达到永不停止运行的程度。

10.2.1　伸缩性

有些属性被认为是实现伸缩能力所必需的。因为可以把用户表示成只处理某些特定事件（例如，收到一个呼叫或者挂机）的进程，所以，理想中的系统应该可以支持众多小计算量的进程，当事件到来时，能够非常快速地在进程之间进行切换。为了让系统更加高效，进程的启动和消亡就必须非常快速。让进程保持轻量是实现这种高效所必需的。必须保持轻量还有一个原因，就是不想使用像进程池（一组数量固定的进程，运行所分配的工作）之类的东西。相反，如果需要多少进程就可以使用多少进程，那么程序设计会容易许多。

> **注意**　关于伸缩性，还有另外一个重点内容，就是要能够摆脱硬件限制。有两种方法：一种是升级硬件，还有一种是增加硬件。第一种方法在某个界限之下是有用的，超过了这个界限就会异常昂贵。第二种方法通常要便宜一些，是通过增加更多的计算机来完成工作。此时，如果语言支持分布式就很有用了。

由于电话应用可靠性要求很高，因此决定采用最彻底的做法，禁止进程之间共享内存。在出现崩溃之后，共享内存会导致系统中的状态不一致，使问题复杂化。与共享内存的方式不同，进程之间只能通过发送消息进行通信，所有的消息数据都是复制的。这种方法效率会低一点，但是更安全。

10.2.2　容错

Erlang 的初始设计者们一直认为错误是不可避免的。你可以竭力去防止 bug，但是在大多数情况下，还是会有 bug 悄悄溜进你的系统。即使出现了奇迹，你的代码中确实没有任何 bug，但是无论如何硬件最终一定会出现故障。因此，Erlang 的观点是找出好的处理错误和问题的方法，而不是企图防止错误的出现。

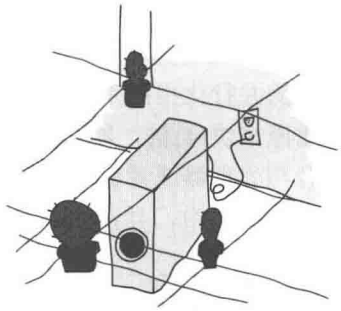
结果表明，基于进程间消息传递的设计方法是一个好主意，因为可以相对容易地向其中增加错误处理机制。以轻量进程（可以快速重启和关闭）为例，有研究表明，大型软件系统中的主要故障来源都是一些间歇性和瞬时性的 bug（参见 http://dslab.epfl.ch/pubs/crashonly/）。此外，还有一个原则：如果系统中的某个部分出现了错误，造成了数据破坏，那么这个部分应该尽快死亡以防止错误和坏数据传播到系统的剩余部分。

还有另外一个概念值得提及：系统可能有多种不同的终止方式，正常关闭和崩溃（因意料之外的错误终止）是其中的两种。

崩溃显然是最差的情况。一种更安全的解决办法是，确保让崩溃和正常关闭具有同样的效果。这种效果可以通过像*无共享*（shared-nothing）（系统各个部分的内存完全隔离）和单一赋值（可以进一步隔离进程的内存）、避免使用锁（如果在发生崩溃时有部分数据是被加锁的，那么会导致其他进程无法访问这部分数据，或者导致这部分数据状态不一致）等实践和其他保护措施达成，而 Erlang 语言的设计中已经包含了这些内容。

因此，Erlang 中最理想的做法就是尽可能快地杀死进程以避免数据破坏和暂态的 bug。轻量进程是这种方案的关键所在。Erlang 语言还提供了进一步的错误处理机制，可以让进程去监控其

他进程（会在第 12 章中介绍），这样就可以知道进程何时死亡，并决定如何处理。



如果说快速重启进程足以应对进程崩溃，那么下一个问题就是处理硬件故障。当有人踢坏了正在运行程序的计算机时，如何保证程序能够继续运行？虽然可以想出像使用激光检测以及有策略地在机器周边放置仙人掌等有想象力的防护机制，解决一时的问题，但是这种做法无法持续。同时让程序运行在多台计算机上是一种更好的解决方案——这也是系统能够伸缩的基础。这是仅使用消息传递机制进行独立进程间通信的另外一个好处。无论进程是在同一台计算机上还是在不同的计算机上，它们的工作方式完全一样，使得这种通过分布式实现的容错对程序员来说几乎透明。

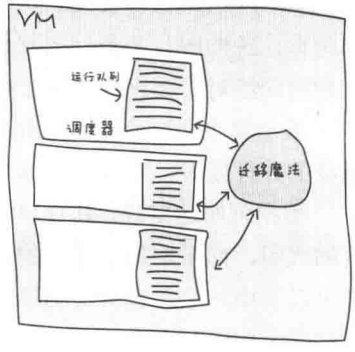
分布式对于进程之间的通信方式有直接的影响。分布式造成的最大问题是，不能因为在发起函数调用时节点（一台远程计算机）是活着的就认为在整个调用执行期间节点会一直活着，也不能认为它一定会正确执行这个调用。如果有人碰掉了网线或者拔掉了机器的电源，那么你的应用就会被挂住，或者崩溃。谁知道呢？

嗯，结果再次表明，采用异步消息传递的设计是一个好的选择。在基于异步消息传递的进程模型中，消息从一个进程发送到另外一个进程，并被保存在接收进程的邮箱中，直到被读取出来为止。值得说明的是，消息在发送时不会检查接收进程是否存在，因为这样做没有用处。正如上一段中所表明的，我们无法知道在消息发送和被收到之间进程是否会崩溃。即使消息被收到了，也无法知道这条消息是会被处理，还是接收进程会在处理前死亡。异步消息可以让远程函数调用更加安全，因为没有对会出现的情况做任何假设，这个责任被转移到程序员身上。如果需要对消息的传送进行确认，那么必须给原始进程再发送一条回应消息。这条消息基于同样的安全语义，基于这个原则构建的任何程序或者库也都是如此。

10.2.3 并发实现

现在，我们知道了 Erlang 为何决定采用基于异步消息传递的轻量进程的方法。但是，Erlang 的实现者们是如何做到这一点的呢？

首先，不能依靠操作系统来实现进程。操作系统有多种进程实现方式，这些方式之间的性能差异也很大。不过，所有这些实现方式基本上都很慢、很重，难以满足 Erlang 应用的需要。通过在 VM 中实现进程，Erlang 的实现者们可以对优化和可靠性进行完全掌控。现在，一个 Erlang 进程大概占用 300 个字的内存空间，创建时间只有几微秒——当前主流的操作系统基本上不可能做到。



为了管理程序所创建的所有进程，VM 会为每个核启动一个线程来充当一个调度器（scheduler）。每个调度器有一个运行队列（run queue），也就是一个 Erlang 进程列表，会给其中的每个进程分配一小段运行时间片。当某个调度器的运行队列中任务过多时，会把一部分任务迁移到其他队列中。这意味着，每个 Erlang

VM 都会进行负载均衡操作，程序员无需关心。VM 还会进行其他优化工作，例如，对发向过载进程的消息进行限速，以调节和均衡负载。

所有这些困难的工作都已经在 Erlang 中完成。因此，让 Erlang 程序并行化会比较容易。这里的并行化指的是，当增加第二个核时，程序的速度应该是原来的 2 倍，如果有 4 个核，要比原来快 4 倍，以此类推，真的是这样吗？这要视情况而定。程序的这种表现称为在速度提升与处理器或者核的数量之间存在*线性伸缩*（linear scaling）关系（参见下一节中的图示）。在现实世界中，并没有免费的午餐（嗯，也许在葬礼上有，但是总归要有人，在某个地方为此付账）。

10.3　并非完全不能线性伸缩

达成线性伸缩的困难并不在于语言本身，而在于要解决问题本身的内在属性。通常称那些伸缩性非常好的问题为*易并行*（embarrassingly parallel）问题。如果你在网上搜索“embarrassingly parallel problems”，所找到的大部分可能都是诸如光线跟踪（一种创建 3D 图像的方法）、密码学中的暴力搜索，以及天气预测之类的例子。

聊天室、论坛和邮件列表中也不时会有人问是否可以用 Erlang 来解决这类问题，或者是否可以用 Erlang 来对图形处理器（GPU）编程。答案几乎肯定是不行。原因很简单：所有这类问题通常都需要对大量数据进行数值计算。Erlang 对此并不擅长。

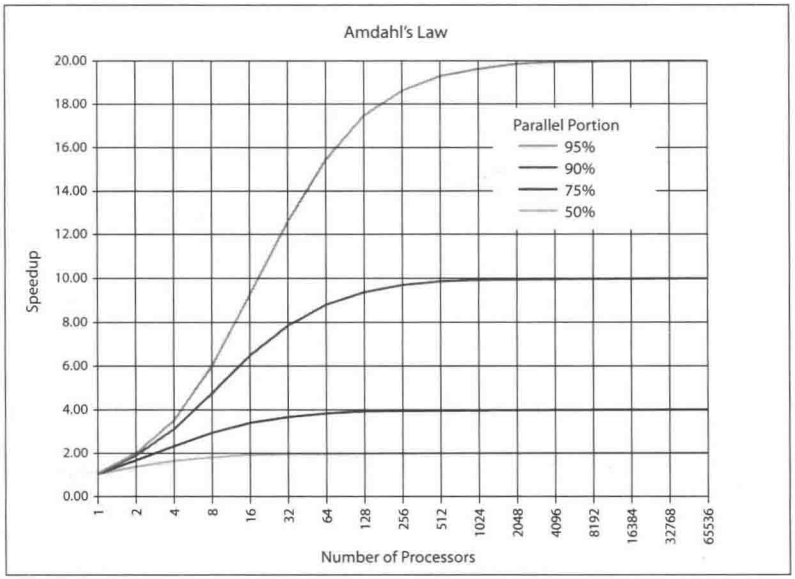
Erlang 中的易并行问题所处的层次要更高一些。一般来讲，它们都是些像聊天服务器、电话交换机、Web 服务器、消息队列、Web 爬虫之类的应用，可以把这类应用中工作的执行体表示成一组相互独立的逻辑实体（actor，是谁呢？）。这类问题可以被有效地解决，基本上接近线性伸缩的程度。

有许多问题根本不可能展现出这种伸缩特性。事实上，只要有一个中心顺序操作序列就足以让这种特性完全丧失。*并行程序的最快运行速度是由其最慢的顺序部分决定的*。任何时候你去商场时，都可以观察到这种现象。数百人可以同时购物，不太会相互干扰。但是，一旦开始付账，只要收银员的人数少于结账顾客的人数，就要开始排队。可以增加收银员的数量，让每个顾客一个收银员，但是之后还得给每个顾客准备一扇门，因为购物者无法同时进出商场。

换句话说，即使顾客可以并行地选购商品，并且可以做到无论商场中只有一个顾客还是有上千顾客，选购所花费的时间都是一样的，但是在结账时，他们仍然需要等待。因此，他们的整个购物时间绝对不会比排队结账所花时间更短。

关于这个原则有个一般化的概括，称为*阿姆达定理*（Amdahl's law）。这个定理给出了在提升系统的并行度时，可以期望系统以多少速度、以何种比例提升。

根据阿姆达定理，具有 50%并行度的代码运行速度绝不能提升到原来的 2 倍，具有 95%并行度的代码，如果增加足够多的处理器，理论上可以期望运行速度提升到原来的 20 倍。从这幅图中可以看出一个比较有趣的现象，程序中最后被移除的那小部分顺序代码在速度提升方面的贡献要远高于程序并行度并不高时所移除的等量顺序代码。



（本图修改自 Daniel 的一幅原创图，在 Creative Commons 许可下使用。可以从 http://en.wikipedia.org/wiki/File:AmdahlsLaw.svg 找到原图。）

保持冷静

并行化并不适用于所有问题。在有些情况下，并行化甚至会降低应用的运行速度。当程序是 100%串行时，如果使用了多个进程，速度就会降低。

这方面最好的示例之一是进程环基准测试（ring benchmark）。进程环基准测试是一种测试方法，会创建数千个进程，以环形的方式在进程间逐个传递数据。可以把它看作一个传话游戏。在这个基准测试中，虽然同时只有一个进程在工作，但是 Erlang VM 仍然要花费时间在多个核之间分配负载，并让每个进程都能有时间运行。

这种做法使得一些通常的硬件优化机制无法实施，也让 VM 做了很多无用功。

这种负载分配工作通常会让纯粹的串行应用在多核上的运行速度比单核更慢。如果这种算法是系统的中心执行路径，那么禁用 SMP（`$ erl -smp disable`）是一个好的选择。不过，在有些情况中，那些并非整个程序中心执行路径的串行算法所占的比例要远小于其他事件处理。此时，禁用 SMP 不会有太大的影响。

10.4 再见，谢谢你的鱼

当然，如果不介绍 Erlang 语言中并发编程所需要的 3 个原语，那么本章的内容肯定是不完整的，这 3 个原语是创建（spawn）进程、发送消息以及接收消息。在实际中，编写真正可靠的应用还需要一些其他机制，不过现在，这 3 个就足够了。

10.4.1　创建进程

我讲了许多关于进程的话题，但是还没有解释过进程到底是什么。其实，进程就是一个函数。进程运行一个函数，一旦运行结束，进程就消失了。从技术上讲，进程还具有一些隐藏状态（如消息邮箱），但是现在我们只关注于进程执行的函数。

要启动一个新进程，可以使用 Erlang 提供的函数 spawn/1，这个函数以一个函数为参数，并运行它：

```
1> F = fun() -> 2 + 2 end.
#Fun<erl_eval.20.67289768>
2> spawn(F).
<0.44.0>
```

spawn/1 的返回值（<0.44.0>）称为进程标识符，在 Erlang 社区中，通常写成 pid（我在本书中使用这种写法）、Pid 或者 PID。pid 是一个随意设定的值，用来表示虚拟机运行期间的某个时间点上存在（或者曾经存在）的某个进程。可以用 pid 作为地址进行进程间的通信。

注意，我们无法得到函数 F 的返回值。我们只能得到它的 pid。因为进程不会返回任何东西。

那么如何得到 F 的返回值呢？嗯，有两种方法。其中比较容易的一种方法是直接把结果打印出来：

```
3> spawn(fun() -> io:format("~p~n",[2 + 2]) end).
4
<0.46.0>
```

虽然不能在真实程序中使用这种方法，但是用来看看 Erlang 是如何调度进程的还是可以的。还好，只要使用函数 io:format/2 就足以进行我们的实验了。我们会快速启动 10 个进程，并让每个进程都暂停一会，暂停是由函数 timer:sleep/1 完成的，该函数以一个整数值 *N* 为参数，会让调用进程等待 *N* 毫秒后再继续执行。暂停时间过后，进程会把赋给它的值打印出来：

```
4> G = fun(X) -> timer:sleep(10), io:format("~p~n", [X]) end.
#Fun<erl_eval.6.13229925>
5> [spawn(fun() -> G(X) end) || X <- lists:seq(1,10)].
[<0.273.0>,<0.274.0>,<0.275.0>,<0.276.0>,<0.277.0>,
<0.278.0>,<0.279.0>,<0.280.0>,<0.281.0>,<0.282.0>]
2
1
4
3
5
8
7
6
10
9
```

打印的顺序是混乱的。欢迎来到并行世界！因为进程是同时运行的，所以事件的顺序不能再得到保证。这是因为，Erlang VM 使用了很多技巧来决定要运行哪个进程，并保证每个进程都能

得到平等的运行时间。许多 Erlang 服务被实现为进程，包括交互式 shell。你创建的进程必须要和系统本身需要的进程之间进行平衡，这就是出现混乱顺序的原因。

对称多处理

无论是否禁用 SMP，上面实验的结果都一样。为了证明这一点，只要用$ erl -smp disable 命令启动 Erlang VM 进行测试就可以了。

要检查你的 Erlang VM 在运行时是否打开了 SMP 开关，可以不用任何选项启动一个新的 VM，并看看第一行打印内容。如果能在其中找到[smp:2:2]，说明 SMP 开关打开了，VM 运行在 2 个核上，使用了 2 个运行队列。如果找不到，就说明 SMP 被禁用了。

[smp:2:2]的意思是说，有两个可用的核，并且有两个调度器（每个调度器拥有一个运行队列）。在以前的 Erlang 版本中，虽然也可以有多个调度器，但是所有的调度器共享一个运行队列。从 R13B 之后，每个调度器具有自己的运行队列，并行性能会好很多。

为了证明 shell 本身也是一个常规的进程，可以使用 BIF self/0，这个函数会返回当前进程的 pid：

```
6> self().
<0.41.0>
7> exit(self()).
** exception exit: <0.41.0>
8> self().
<0.285.0>
```

可以看到，因为进程被重启了，所以 pid 变了。

没有人愿意一直通过打印的方式得到进程的结果值，然后再把这个值手动传递给其他进程（至少我不愿意），因此接下来我们学习一下如何发送消息。

10.4.2 发送消息

现在我们来介绍消息传递原语——操作符!，也称为 bang 符号。该操作符的左边是一个 pid，右边可以是任意 Erlang 数据项。这个数据项会被发送给左边的 pid 所代表的进程，这个进程就可以访问它了。下面是一个例子：

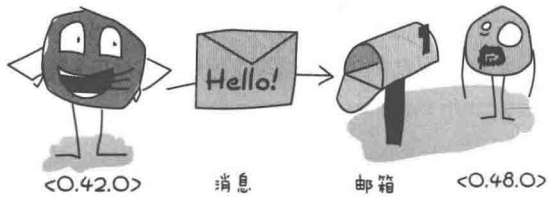
```
9> self() ! hello.
hello
```

消息会被放到接收进程的邮箱中，但是并没有被读取。上面例子中出现的第二个 hello 是这个发送函数的返回值。这意味着，可以用如下方式给多个进程发送同样的消息：

```
10> self() ! self() ! double.
double
```

上面的调用等价于 self() ! (self() ! double)。

关于进程邮箱，有一点要注意，消息是按照接收顺序保存在邮箱中的。每当读取一个消息时，就会把它从邮箱中取出。同样地，这和人们通过信件通信的方式有点类似。



要想看到当前邮箱中的内容，如果在 shell 中，可以使用 flush() 命令：

```
11> flush().
Shell got hello
Shell got double
Shell got double
ok
```

flush/0 函数只是一种输出所收到的消息的快捷方法。这意味着，仍然无法把输出结果绑定到变量上，不过至少我们知道如何把消息从一个进程发送给另外一个进程，也知道如何检查是否收到消息。

10.4.3　接收消息

发送一些无人会看的消息的用处就和写一些表达自我情绪的诗一样（换句话说，就是不大有用）。因此，我们还需要 receive 表达式。下面我们将不再在 shell 中进行试验，而是编写一个短小的海豚（dolphin）示例程序来展示如何接收消息。下面是我们的新程序：

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

dolphin1() ->
    receive
        do_a_flip ->
            io:format("How about no?~n");
        fish ->
            io:format("So long and thanks for all the fish!~n");
        _ ->
            io:format("Heh, we're smarter than you humans.~n")
    end.
```

可以看出，receive 的语法和 case ... of 非常相似。事实上，它们的模式匹配部分的工作原理完全一样，只是 receive 模式中变量会绑定到收到的消息，而不是 case 和 of 之间的表达式。receive 表达式也可以有卫语句。下面是其总体的语法形式：

```
receive
    Pattern1 when Guard1 -> Expr1;
    Pattern2 when Guard2 -> Expr2;
    Pattern3 -> Expr3
end
```

了解了这些内容之后，现在开始编译这个模块，运行它，开始和海豚进行交流：

```
11> c(dolphins).
```

```
{ok,dolphins}
12> Dolphin = spawn(dolphins, dolphin1, []).
<0.40.0>
13> Dolphin ! "oh, hello dolphin!".
Heh, we're smarter than you humans.
"oh, hello dolphin!"
14> Dolphin ! fish
fish
```

在上面的测试中，引入了一个新的进程创建函数 spawn/3。不再只以一个函数为参数，spawn/3 函数有 3 个参数：模块、函数和函数参数。函数运行起来后，会发生如下事件。

（1）函数执行到了 receive 表达式。因为进程邮箱为空，所以海豚进程会一直处于等待消息状态。

（2）收到了消息"oh, hello dolphin!"。函数会先对 do_a_flip 进行模式匹配。失败了。然后再尝试模式 fish，也失败了。最后遇到了匹配一切的子句（_），匹配成功。

（3）进程打印消息“Heh, we're smarter than you humans.”

注意，如果发送的第一条消息被匹配上了，那么进程<0.40.0>不会对随后发送的消息做任何反应。这是因为，一旦函数打印了“Heh, we're smarter than you humans”，它就结束了，进程也随之结束。我们需要重启海豚进程：

```
8> f(Dolphin).
ok
9> Dolphin = spawn(dolphins, dolphin1, []).
<0.53.0>
10> Dolphin ! fish.
So long and thanks for all the fish!
fish
```

这次，fish 消息得到了处理。

如果不使用 io:format/2，而是从海豚进程那里得到一个回应不是更好吗？当然更好！（这还用问吗？）

在本章的前面我曾经说过，要想知道进程是否收到了消息，唯一的方法是让它发送一条回应。我们的海豚进程需要知道要把回应发送给谁。这里采用的方法和邮政服务中的完全一样。如果我们想让某人给我们回信，就必须在信中添加我们的地址。在 Erlang 中，可以通过把进程的 pid 打包在一个元组中完成这项工作，如果不这样做，那么消息就都是匿名的。打包的结果是一条类似{Pid, Message}的消息。我们来编写一个新的海豚函数，让它接受这种消息：

```
dolphin2() ->
    receive
        {From, do_a_flip} ->
            From ! "How about no?";
        {From, fish} ->
            From ! "So long and thanks for all the fish!";
        _ ->
            io:format("Heh, we're smarter than you humans.~n")
```

```
    end.
```

可以看出，在原先接受的 do_a_flip 和 fish 消息中都增加了一个 From 变量。这个变量就是前面说的回应消息要发往的 pid。

```
11> c(dolphins).
{ok,dolphins}
12> Dolphin2 = spawn(dolphins, dolphin2, []).
<0.65.0>
13> Dolphin2 ! {self(), do_a_flip}.
{<0.32.0>,do_a_flip}
14> flush().
Shell got "How about no?"
ok
```

看起来工作得非常好。我们可以接收到对我们发送消息的回应（需要在每条消息上添加一个地址，有点像电子邮件中的 Reply To 字段），但是每调用一次，我们都还得启动一个新的进程。递归是解决这个问题的方法。我们需要做的就是让函数调用自身，这样它就不会结束，并一直等待更多的消息。下面的 dolphin3/0 函数就是增加了递归调用后的版本：

```
dolphin3() ->
    receive
        {From, do_a_flip} ->
            From ! "How about no?",
            dolphin3();
        {From, fish} ->
            From ! "So long and thanks for all the fish!";
        _ ->
            io:format("Heh, we're smarter than you humans.~n"),
            dolphin3()
    end.
```

上面的代码中，匹配一切的子句和 do_a_flip 子句都会循环调用 dolphin3/0。注意，这个调用不会导致栈溢出，因为它是尾递归的。只要发送的消息匹配上了这两个模式，海豚进程就会一直循环。不过，如果发送了 fish 消息，进程就会停止：

```
15> Dolphin3 = spawn(dolphins, dolphin3, []).
<0.75.0>
16> Dolphin3 ! Dolphin3 ! {self(), do_a_flip}.
{<0.32.0>,do_a_flip}
17> flush().
Shell got "How about no?"
Shell got "How about no?"
ok
18> Dolphin3 ! {self(), unknown_message}.
Heh, we're smarter than you humans.
{<0.32.0>,unknown_message}
19> Dolphin3 ! Dolphin3 ! {self(), fish}.
{<0.32.0>,fish}
20> flush().
```

```
Shell got "So long and thanks for all the fish!"
ok
```

dolphin.erl 的代码编写完了。从上面的测试中可以看出，它的行为确实符合我们的预期，对于收到的每一条消息都会进行一次回应，然后继续运行，当然 `fish` 消息除外。最后，海豚对我们人类疯狂古怪的行为感到厌烦，永远离我们而去。

讲完了。这就是 Erlang 全部并发特性的核心内容。我们学习了进程和基本的消息传递方面的知识。要编写出真正有用且可靠的程序，还有不少概念需要了解。我们会在后面几章中对其中的一些概念进行介绍。

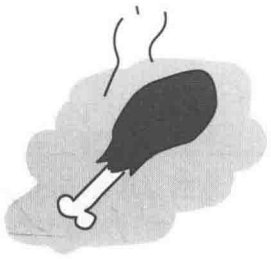
第 11 章

深入多重处理

第 10 章中的例子作为阐明概念来说是非常适合的，但是它们不能进一步指导真实的项目开发。这并不是因为那些例子本身比较糟糕，而是因为如果进程和 actor 只是一些能收发消息的函数，并不会带来多少好处。为了能够得到更大的好处，需要在进程中持有状态。

在本章中，我们会把并发的概念和原语应用到实际的例子中，在这些例子中，进程会持有状态。

11.1 定义进程状态



我们在一个新模块 kitchen.erl 中创建一个函数，执行这个函数的进程类似于冰箱。这个冰箱进程有两个操作：向冰箱中存放食物以及从冰箱中取出食物。在取食物时，只能取那些之前放进去的食物，所取出的食物数量最多只能是放进去的食物数量。冰箱进程可以基于下面这个函数实现：

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

fridge1() ->
    receive
        {From, {store, _Food}} ->
            From ! {self(), ok},
            fridge1();
        {From, {take, _Food}} ->
            %% 哦....
            From ! {self(), not_found},
            fridge1();
        terminate ->
            ok
    end.
```

上面的代码存在错误。当向冰箱中存放食物时，进程回应了 ok，但是实际上并没有存放任何食物。调用了 fridge1()函数，接着又重头开始了，没有任何状态。同样，在调用进程从冰箱中取出食物时，也无物可取，因此只能返回 not_found。为了能够存放和获取食物，需要给这个函数增加状态。

借助于递归函数的帮助，进程的状态可以全部存放到递归函数的参数中。对于冰箱进程来说，可以把所有的食物存放到一个列表中，在某人想要从冰箱中取东西吃时，可以从这个列表中查询：

```
fridge2(FoodList) ->
    receive
        {From, {store, Food}} ->
            From ! {self(), ok},
            fridge2([Food|FoodList]);
        {From, {take, Food}} ->
            case lists:member(Food, FoodList) of
                true ->
                    From ! {self(), {ok, Food}},
                    fridge2(lists:delete(Food, FoodList));
                false ->
                    From ! {self(), not_found},
                    fridge2(FoodList)
            end;
        terminate ->
            ok
    end.
```

注意，fridge2/1 函数有一个参数 FoodList。可以看出，当我们发送一个和{From, {store, Food}}匹配的消息时，在递归调用前，这个函数会把 Food 增加到 FoodList 中。一旦进行了递归调用，就可以获取到之前存放进去的食物了。事实上，上面的代码实现了这个功能。

函数中使用 lists:member/2 检查 FoodList 中是否包含有 Food。根据结果，要么把找到的食物发回给调用进程（并从 FoodList 中移除），要么返回 not_found：

```
1> c(kitchen).
{ok,kitchen}
2> Pid = spawn(kitchen, fridge2, [[baking_soda]]).
<0.51.0>
3> Pid ! {self(), {store, milk}}.
{<0.33.0>,{store,milk}}
4> flush().
Shell got {<0.51.0>,ok}
ok
```

存放食物的操作看起来正常。现在再存放一种食物，然后再把它从冰箱中取出来：

```
5> Pid ! {self(), {store, bacon}}.
{<0.33.0>,{store,bacon}}
6> Pid ! {self(), {take, bacon}}.
{<0.33.0>,{take,bacon}}
```

```
7> Pid ! {self(), {take, turkey}}.
{<0.33.0>,{take,turkey}}
8> flush().
Shell got {<0.51.0>,ok}
Shell got {<0.51.0>,{ok,bacon}}
Shell got {<0.51.0>,not_found}
ok
```

和期望的一样，可以从冰箱中取出熏肉（bacon），因为之前它与牛奶（milk）和小苏打（baking soda）一起放入过，但是冰箱中却无火鸡（turkey）可取。这也是为何最后一条消息是{<0.51.0>,not_found}的原因。更有趣的是，进程邮箱的工作方式会确保即使有 1 000 个人突然同时从冰箱中获取最后一块火鸡，也只有一个人能够得到。

11.2　隐藏消息实现

上面的例子有一点很讨厌，使用冰箱进程的程序员必须要知道冰箱进程自身使用的协议。这是一个无意义的负担。一种好的方法是，使用函数来处理消息的接收和发送，从而把消息隐藏起来：

```
store(Pid, Food) ->
    Pid ! {self(), {store, Food}},
    receive
        {Pid, Msg} -> Msg
    end.

take(Pid, Food) ->
    Pid ! {self(), {take, Food}},
    receive
        {Pid, Msg} -> Msg
    end.
```

现在和进程交互起来整洁多了：

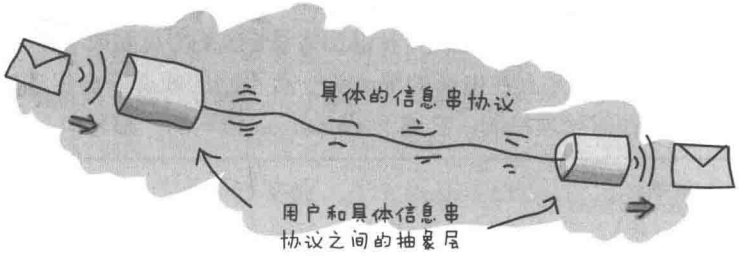
```
9> c(kitchen).
{ok,kitchen}
10> f().
ok
11> Pid = spawn(kitchen, fridge2, [[baking_soda]]).
<0.73.0>
12> kitchen:store(Pid, water).
ok
13> kitchen:take(Pid, water).
{ok,water}
14> kitchen:take(Pid, juice).
not_found
```

现在不需要再关心消息的实现细节了。如果现在再想发送 self()以及某个像 take 或者 store 这样的具体原子，只要知道一个 pid 和要调用哪个函数就行了。这隐藏了所有的实现细节，并使得冰箱进程更易于使用。

现在把整个进程启动部分也隐藏起来。我们进行了消息隐藏，但是使用者还是得自己创建进程。我们来增加函数 start/1，如下：

```
start(FoodList) ->
    spawn(?MODULE, fridge2, [FoodList]).
```

代码中，?MODULE 是一个宏，它的值是当前模块的名字。猛一看，编写这样一个函数似乎没什么好处，其实是有好处的。最根本的好处是保持与调用 take/2 和 store/2 的一致性——所有关于冰箱进程的操作现在都通过 kitchen 模块进行。如果我们想在冰箱进程启动或者启动另外一个进程（冰柜进程）时，进行日志操作，那么可以非常容易地在 start/1 函数中实现。但是，如果进程的创建是由使用者自己调用 spawn/3 函数完成的，那么每个启动冰箱进程的地方都要增加一个新的调用。这很容易出错，而错误令人讨厌。



我们来使用一下这个函数：

```
15> f().
ok
16> c(kitchen).
{ok,kitchen}
17> Pid = kitchen:start([rhubarb, dog, hotdog]).
<0.84.0>
18> kitchen:take(Pid, dog).
{ok,dog}
19> kitchen:take(Pid, dog).
not_found
```

不错！热狗从冰箱里取出来了，我们的抽象很完美！

11.3 超时

我们用命令 pid(A,B,C) 来做一个试验，这个命令可以把 3 个整数转换成一个 pid。在这个试验中，我们故意给 kitchen:take/2 传递一个假的 pid：

```
20> kitchen:take(pid(0,250,0), dog).
```

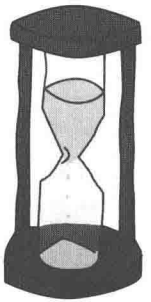
哇。shell 挂住了。这种情况是由 take/2 的实现方式造成的。为了理解问题的原因，我们先来回顾一下正常情况下的处理流程。

（1）食物存放消息从你这里（shell）发送给冰箱进程。

（2）你的进程切换到消息接收模式，等待新消息的到来。

（3）冰箱进程存放好食物并给你的进程发送 ok 回应。

（4）你的进程收到这条消息，并继续执行。



下面是当 shell 挂住时的处理流程。

（1）食物存放消息从你这里（shell）发送给一个未知进程。

（2）你的进程切换到消息接收模式，等待新消息的到来。

（3）那个未知进程要么根本不存在，要么根本不期望这条消息，所以不做任何处理。

（4）shell 进程停滞在消息接收模式。

这很讨厌，尤其是此时不能进行任何错误处理。没有任何异常情况出现，程序就等待在那里——永远，这是一种*死锁*。一般来讲，任何异步处理操作（就是 Erlang 中的消息发送）在一段确定的时间之内没有收到任何消息时，都需要一种退出方法。当某个页面或者图片一直无法加载时，Web 浏览器就会选择放弃，当某人在电话的那一头一直不回应或者开会迟到时，你也会放弃。Erlang 中当然也有恰当的超时处理机制，它属于 receive 语句的一部分：

```
receive
    Match -> Expression1
after Delay ->
    Expression2
end.
```

receive 和 after 之间的部分和之前见到的完全一样。当过了 *Delay*（单位是毫秒）时间后还没有收到和 *Match* 模式相匹配的消息，就会执行 after 部分。此时，会执行 *Expresson2*。

我们要编写两个新的接口函数，store2/2 和 take2/2，它们与 store/2 和 take/2 完全一样，除了会在 3 秒后停止等待：

```
store2(Pid, Food) ->
    Pid ! {self(), {store, Food}},
    receive
        {Pid, Msg} -> Msg
    after 3000 ->
        timeout
    end.

take2(Pid, Food) ->
    Pid ! {self(), {take, Food}},
    receive
        {Pid, Msg} -> Msg
    after 3000 ->
        timeout
    end.
```

当等待的时间过长时，会返回 timeout。我们并没有处理等待时间过长的情况，回应消息可能随后会以非期望的方式发送给我们，但是至少如果其他进程死亡了，我们不会陷入死锁。

Erlang 中有个叫监控器的机制可以解决这里的问题，并让代码变得更加健壮，我们会在第 12 章中介绍。不过，现在只需要按下 Ctrl+G 解冻 shell，然后试试新的接口函数：

```
User switch command
--> i
--> s
--> c
Eshell V5.7.5 (abort with ^G)
1> c(kitchen).
{ok,kitchen}
2> kitchen:take2(pid(0,250,0), dog).
timeout
```

计时器起作用了。

> **注意** 我说过 after 只接收毫秒值，但是实际上还可以使用原子 infinity。虽然这个值在多数情况下不是很有用（此时可以直接去掉整个 after 语句），不过当程序员想把等待的时间传递给某个接收结果的函数时，这个值可能会有用。此时，如果程序员想让函数一直等待，可以传递这个值。

计时器不是只能用于放弃过长的等待。在 timer:sleep/1 函数的实现中还有另外一种用法，我们在第 10 章中使用过这个函数。下面是它的实现代码（把它放到一个新模块 multiproc.erl 中）：

```
sleep(T) ->
    receive
    after T -> ok
    end.
```

在这个特定的情况中，receive 语句中不会进行任何匹配，因为没有任何模式。相反，一旦延时 T 过后，就会执行 after 部分的语句。

另外一种特殊情形是把超时值设置为 0：

```
flush() ->
    receive
      _ -> flush()
    after 0 ->
       ok
    end.
```

当执行这个函数时，Erlang VM 会尽力寻找一个和所提供的模式匹配的消息。在上面的代码中，可以匹配所有的消息。只要找到了消息，flush/0 函数就会递归地调用自己直到邮箱为空。此后，会执行 after 0 -> ok 部分，并返回。

11.4 选择性接收

基于上面 flush 函数的思想，可以实现一种选择性接收，能够通过嵌套调用对接收到的消息进行优先级排序：

```
important() ->
    receive
        {Priority, Message} when Priority > 10 ->
            [Message | important()]
    after 0 ->
        normal()
    end.

normal() ->
    receive
        {_, Message} ->
            [Message | normal()]
    after 0 ->
        []
    end.
```

这个函数会构建出一个包含所有消息的列表，其中优先级高于 10 的排在前面：

```
1> c(multiproc).
{ok,multiproc}
2> self() ! {15, high}, self() ! {7, low}, self() ! {1, low}, self() ! {17, high}.
{17,high}
3> multiproc:important().
[high,high,low,low]
```

因为函数中使用了 `after 0`，所以每个消息都会被取出来，直到邮箱中没有任何消息，不过进程会在获取其他消息之前先提取那些优先级大于 10 的消息，其他消息会在 `normal/0` 函数中收集。这项实践称为*选择性接收*（selective receive）。它看起来很有吸引力，不过要注意，由于 Erlang 的处理方式，选择性接收有时使用起来不太安全。

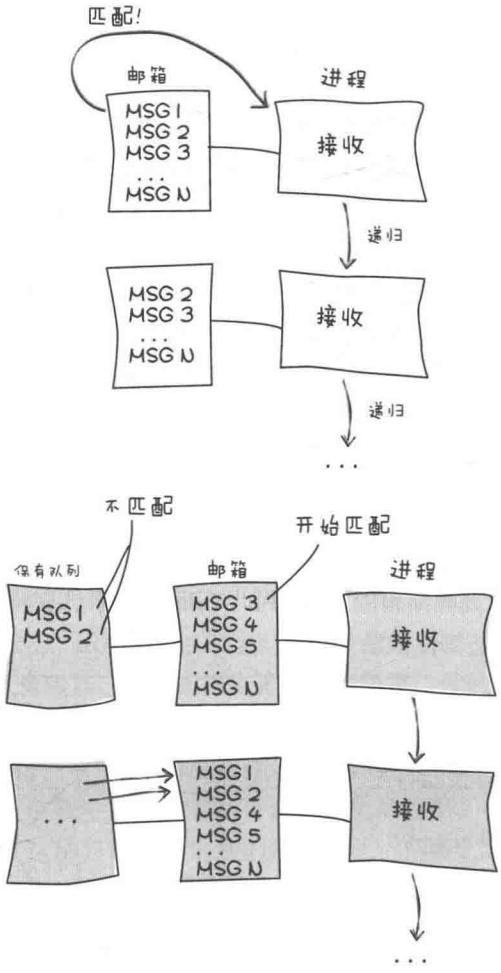
11.4.1　选择性接收的风险

当消息发送到一个进程时，即使发送进程已经死亡，也会把它们保存在接收进程邮箱中，直到进程去读取消息并且匹配上了某个模式为止。消息的保存顺序和接收到的顺序一致。这意味着，每当执行一次 `receive` 语句去匹配一个消息时，都要扫描一遍邮箱，从收到的第一条（也就是最老的）消息开始。

会先用这条最老的消息去匹配 `receive` 中的每一个模式，直到匹配上某个模式。如果匹配上了，就会从邮箱中移除这条消息，进程的代码会正常执行直到遇到下一条 `receive` 语句。当执行下一条 `receive` 语句时，VM 会使用当前邮箱中最老的消息（就是排在刚刚移除掉的消息后面的消息）进行匹配，以此类推。

当某条消息没有被匹配上时，会把它放到一个备用*保存队列*（save queue）中，然后尝试下一条消息。如果下一条消息匹配上了，那么第一条消息会被放回到邮箱的顶部，供下次尝试使用。

这种机制可以让你只关心有用的消息。以上述方式忽略掉一些消息，以后再处理它们是选择性接收的本质所在。虽然这很有用，但是选择性接收的问题在于，如果进程邮箱中有很多你根本不关心的消息，那么读取那些有用消息花费的时间会越来越长（进程占用内存也会增加）。



在前面关于消息匹配的说明中，假设我们想要第 367 条消息，前面 366 条消息都会被程序作为垃圾忽略。为了得到第 367 条消息，进程需要尝试匹配这 366 条垃圾消息。做完匹配后，这些消息会被放到备用的保存队列中，然后取出第 367 条消息，之后前面的 366 条消息还会被放回到进程邮箱中。下一条有用的消息可能会被埋藏得更深，需要更长的时间才能找到。

这种消息接收方式是 Erlang 性能问题的一种常见原因。如果你的应用运行缓慢，并且你知道有大量的消息发来发去，那么就可能是这个原因导致的。如果真的是这种选择性接收造成了系统性能的严重下降，首先要做的是问问自己为何会收到那些不想要的消息。消息发给正确的进程了吗？匹配模式正确吗？消息格式正确吗？是不是在应该使用多个进程的地方只用了一个进程呢？对这些问题的回答可能会解决你的问题。

消息交互优化

从 R14A 之后，Erlang 编译器中增加了一个新的优化手段。对于某些非常特定的进程间消息传递情况，它会对选择性接收进行简化。multiproc.erl 中的 optimized/1 函数就是满足这种情况的一个例子。

为了让优化起作用，函数中必须要创建一个引用（可以使用 make_ref() 或者创建监控器，会在第 12 章中介绍），然后把引用放在消息中发送出去。还是在这个同样的函数中，再进行一个选择性接收操作。如果只有当消息中包含相同的引用才能被匹配上时，编译器会自动保证 VM 在扫描时略过这个引用创建之前收到的所有消息。

注意，你不能试图强行更改自己的代码使之满足这种优化条件。Erlang 语言的开发者只会寻找那些经常被用到的模式，然后把它们变得更快。如果你编写的代码满足语言的惯用法，那么自然就能享受到优化的结果，反之不然。

11.4.2　邮箱使用的其他风险

由于存在无用消息污染进程邮箱的风险，因此 Erlang 程序员有时会使用一种防御性的手段来应对这种情况。一种标准的防御性做法如下：

```
receive
    Pattern1 -> Expression1;
    Pattern2 -> Expression2;
    Pattern3 -> Expression3;
    ...
    PatternN -> ExpressionN;
    Unexpected ->
        io:format("unexpected message ~p~n", [Unexpected])
end.
```

这种做法可以保证任何消息都会匹配到至少一个模式语句。Unexpected 变量会匹配所有消息，把非期望的消息从邮箱中取出，并打印一条警告消息。根据应用的不同，可能也会把非期望消息保存在某种日志工具中，供以后查看使用。如果消息发给了错误的进程，就永远地丢失了这些消息，非常遗憾，以后也很难查出其他进程没有收到该收到的消息的原因，因为这很可能是一个 bug。

如果确实需要以不同的优先级对消息进行处理，从而不能使用这种匹配一切的语句，一种更聪明的方法是实现一个最小堆（min-heap，参见 https://secure.wikimedia.org/wikipedia/en/wiki/Min-heap）或者使用 gb_trees 模块（在第 9 章中介绍过），把收到的每条消息都取出来放到其中（确保把优先级数字作为键值的前面部分，这样可以用来对消息排序）。然后，就可以根据需要在数据结构中搜索最小或者最大的元素了。

在大多数情况下，这种技术的消息接收效率要高于选择性接收。不过，如果收到的消息基本上都是最高优先级的，那么效率会低一些。同样，在优化之前请先测试和度量。

现在我们已经学习了如何在进程中保存状态，接下来需要学习一下如何在多进程环境中高效地处理错误，这是第 12 章的主要内容。

第 12 章

错误与进程

在大多数语言中，异常都是在程序执行流内处理的，我们在前面的例子中使用的 `try ... catch` 就是这种方式。这种常见的做法存在一个问题，要么必须在正常代码逻辑的每一层中处理尚未被捕获的异常错误，要么只好把错误处理的负担一直推到程序的最顶层 `try ... catch` 中。这样做虽然可以捕获所有错误，但是却再也无法知道错误出现的原因了。虽然真实世界中的情况可能没这么复杂，但是大体上也就是这个样子。我们在前面已经看到，Erlang 也支持这种模型。

不过，Erlang 还支持另外一种层次的异常处理，可以把异常处理逻辑从程序的正常执行流中移出来，放到另外一个并发进程中。这种方法会让代码更加整洁，只用考虑那些“正常的情况”。在本章中，我们会介绍实现这种异常处理方法的基本工具：链接、监控器以及命名进程，并且还会介绍一些高效使用这些工具的通用实践方法。

12.1 链接

链接（link）是两个进程之间的一种特殊关系。当在两个进程间建立了这种关系后，如果其中一个进程由于意外的抛出、出错或者退出（参见第 7 章）而死亡时，另外一个进程也会死亡，把这两个进程独立的生存期绑定成一个关联在一起的生存期。

从尽快失败阻止错误蔓延的角度来说，这是一个非常有用的概念。如果某个进程由于错误崩溃了，但依赖于它的进程却继续运行，那么所有这些依赖进程都必须要处理依赖缺失情况。让它们死亡，然后重启整个进程组通常是一种可以接受的替代方案。链接就是实现这种功能的。

Erlang 中有一个原生函数 `link/1`，用于在两个进程间建立一条链接，它的参数是进程的 pid。当调用它时，会在当前进程和参数 pid 标识的进程之间建立一条链接。要去除链接，可以使用 `unlink/1`。

当链接进程中的一个死亡时，会发送一条特殊的消息，其中含有死亡原因相关的信息。如果进程正常死亡了（函数正常执行完毕），就不会发送这条消息。

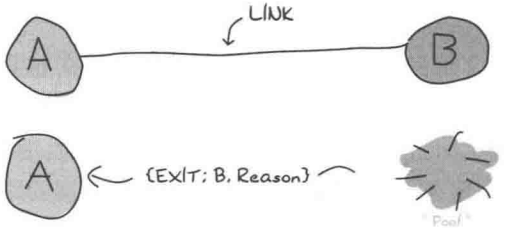
我们来看看这个新函数是如何工作的，下面给出 linkmon.erl 文件中的代码片段。

```
myproc() ->
    timer:sleep(5000),
    exit(reason).
```

如果你进行如下调用（在两个 spawn 命令之间等待 5 s），会发现仅当在进程间建立了链接时，shell 进程才会由于 reason 原因崩溃：

```
1> c(linkmon).
{ok,linkmon}
2> spawn(fun linkmon:myproc/0).
<0.52.0>
3> link(spawn(fun linkmon:myproc/0)).
true
** exception error: reason
```

下图是相应的工作原理图。



不过，这个{'EXIT', B, Reason}消息是不能像通常那样被 try ... catch 捕获的。需要使用其他机制来进行捕获，我们会在 12.1.1 节进行介绍。

> **注意**　如果想在 shell 中杀死一个进程，可以使用函数 exit/2，这个函数会使用这种机制杀死进程。它的调用方法是 exit(Pid, Reason)。如果想的话，可以调用一下试试。

可以使用链接建立一个大型的进程组，这个组中的进程应该一起死亡。下面就是一个例子：

```
chain(0) ->
    receive
        _ -> ok
    after 2000 ->
        exit("chain dies here")
    end;
chain(N) ->
    Pid = spawn(fun() -> chain(N-1) end),
    link(Pid),
    receive
        _ -> ok
    end.
```

这个函数以整数 N 为参数，启动 N 个链接在一起的进程。每次调用时，会把 N-1 作为参数

传递给 chain 函数，创建下一个进程（调用 spawn/1），spawn 中使用了匿名函数把 chain 调用包装起来，因此无需额外的参数。调用 spawn(?MODULE, chain, [N-1])，效果相同。

这个函数会创建许多链接在一起的进程，并会随着后续进程的死亡而死亡。

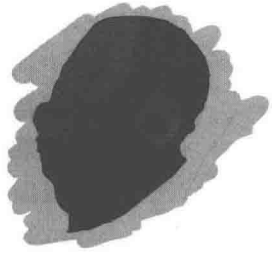
```
4> c(linkmon).
{ok,linkmon}
5> link(spawn(linkmon, chain, [3])).
true
** exception error: "chain dies here"
```

可以看到，shell 确实收到了从其他进程发来的死亡信号。下面展示了所创建进程的链接和死亡关系：

```
[shell] == [3] == [2] == [1] == [0]
[shell] == [3] == [2] == [1] == *dead*
[shell] == [3] == [2] == *dead*
[shell] == [3] == *dead*
[shell] == *dead*
*dead, error message shown*
[shell] <-- restarted
```

当运行 linkmon:chain(0)的进程死完后，错误顺着进程链传播，直到 shell 进程，shell 进程也因此死亡。也可以让链条中间的某个链接进程崩溃。因为链接是双向的，所以只要一个死亡，其他的就都会死亡。

> **注意** 链接不会堆叠。如果在同样的两个进程之间调用了 15 次 link/1，那么这两个进程之间只会存在一条链接，只需一次 unlink/1 调用就可以解除这个链接。



注意，link(spawn(Function))或者 link(spawn(M,F, A))并不是一个原子操作。有时，进程会在链接建立成功之前死亡，从而导致不期望的行为。因此，Erlang 中增加了 spawn_link/1-3 函数。这个函数的参数和 spawn/1-3 完全一样，创建一个进程，并和它建立链接，就像使用了 link/1 一样，不过这是一个原子调用（两个操作被合并成一个操作，要么成功，要么失败，不会出现其他情况）。这个调用通常更安全，并且可以少敲几个括号。

12.1.1 捕获退出信号

跨进程的错误传播对进程来说和消息传递类似，不过使用的是一种称为信号（signal）的特殊消息。退出信号是一种“秘密”消息，会自动作用到进程上并杀死它们。

我之前曾经多次说过，为了可靠起见，应用必须能够快速地杀死和重启进程。现在，链接可以完成快速杀死进程的工作。还缺少快速重启的部分。要重启一个进程，首先需要知道它已经死了。有一种称为系统进程的概念，可以完成这项工作，它的做法是在链接之上再增加一个层次（蛋糕上美味的糖霜）。

系统进程就是一般的进程，只是它们可以把退出信号转换成普通的消息。进程可以通过调用

process_flag (trap_exit, true)实现这一点。一例胜千言。我们来重写一下 chain 示例，让最开始的那个进程成为系统进程。

```
1> process_flag(trap_exit, true).
true
2> spawn_link(fun() -> linkmon:chain(3) end).
<0.49.0>
3> receive X -> X end.
{'EXIT',<0.49.0>,"chain dies here"}
```

啊！现在有趣多了。还用图示来说明的话，现在看起来是这样：

```
[shell] == [3] == [2] == [1] == [0]
[shell] == [3] == [2] == [1] == *dead*
[shell] == [3] == [2] == *dead*
[shell] == [3] == *dead*
[shell] <-- {'EXIT,Pid,"chain dies here"} -- *dead*
[shell] <-- still alive!
```

使用这种机制就可以做到进程的快速重启了。通过在应用程序中使用系统进程，可以很容易地编写这样一个进程，它唯一的任务就是检查是否有进程死亡了，如果死亡了就重启它。在第 13 章实际使用这些技术时，我们会对其进行详细的介绍。

12.1.2　老异常，新概念

现在回到第 7 章中介绍的异常函数，看看针对捕获了退出信号的进程，它们在行为上有什么变化。我们会用一个非系统进程进行对比试验。来看看那些未捕获的 throw、error 和 exit 在与它链接的那个进程中会有什么表现。

1. 异常和退出信号捕获

进程死亡的原因有很多。我们只会研究其中的一部分，看看当捕获了退出信号时，这些原因看起来是什么样的。

异常源： spawn_link(fun() -> ok end)

未捕获时的结果： 无任何结果

捕获时的结果： {'EXIT', <0.61.0>, normal}

进程正常退出，没有出现问题。注意，这个结果和 catch exit(normal)的结果有点儿像，只不过元组中多了一个 pid，标识哪个进程死亡了。

异常源： spawn_link(fun() -> exit(reason) end)

未捕获时的结果： ** exception exit: reason

捕获时的结果： {'EXIT', <0.55.0>, reason}

进程由于一个自定义的原因终止了。如果没有捕获退出信号，和它链接的进程就会崩溃。如果捕获了退出信号，就会收到一条消息。

异常源： spawn_link(fun() -> exit(normal) end)

未捕获时的结果： 无任何结果

捕获时的结果： {'EXIT', <0.58.0>, normal}

这个调用成功地模拟了进程正常退出的情况。有时，你想把杀死进程当作程序正常流程的一部分，并且不想引发任何异常，那么可以使用这种方法。

异常源：`spawn_link(fun() -> 1/0 end)`

未捕获时的结果：`Error in process <0.44.0> with exit value: {badarith, [{erlang, '/', [1,0]}]}`

捕获时的结果：`{'EXIT', <0.52.0>, {badarith, [{erlang, '/', [1,0]}]}}`

错误（`{badarith, Reason}`）没有用 `try ... catch` 块进行捕获，从而升级成了一个`'EXIT'`。此时，它和`exit(reason)`完全一样，不过多了一个描述所发生情况细节的栈跟踪信息。

异常源：`spawn_link(fun() -> erlang:error(reason) end)`

未捕获时的结果：`Error in process <0.47.0> with exit value: {reason, [{erlang, apply, 2}]}`

捕获时的结果：`{'EXIT', <0.74.0>, {reason, [{erlang, apply, 2}]}}`

这里的错误情况和 1/0 非常相似。这很正常——`erlang:error/1` 就是用来做这种事的。

异常源：`spawn_link(fun() -> throw(rocks) end)`

未捕获时的结果：`Error in process <0.51.0> with exit value: {{nocatch, rocks},[{erlang, apply, 2}]}`

捕获时的结果：`{'EXIT', <0.79.0>, {{nocatch, rocks}, [{erlang, apply, 2}]}}`

因为没有用 `try ... catch` 去捕获 `throw`，所以它先升级成一个错误，随后又升级成一个 `EXIT`。没有捕获退出信号时，进程死亡了；当捕获退出信号时，则可以很好地处理这个错误。

常见的异常就这些了。当正常退出时，一切正常。当出现异常情况时，进程死亡，并发送相应的信号消息。

2. 完全不同的 `exit/2`

接下来试试 `exit/2` 函数。这个函数等同于 Erlang 进程中的枪支。进程可以用它来远程杀死其他进程。下面是一些可能的调用。

异常源：`exit(self(), normal)`

未捕获时的结果：`** exception exit: normal`

捕获时的结果：`{'EXIT', <0.31.0>, normal}`

当没有捕获退出信号时，`exit(self(), normal)`和 `exit(normal)`一样。否则，会收到一条消息，消息的格式和有链接关系的进程正常死亡时该进程收到的消息格式一样。

异常源：`exit(spawn_link(fun() -> timer:sleep(50000) end), normal)`

未捕获时的结果：无任何结果

捕获时的结果：无任何结果

这基本上等同于 exit(Pid, normal)调用。这个命令不会有任何效果，因为不能以 normal 作为原因参数远程杀死一个进程。

异常源： exit(spawn_link(fun() -> timer:sleep(50000) end), reason)

未捕获时的结果： ** exception exit: reason

捕获时的结果： {'EXIT', <0.52.0>, reason}

这个调用会导致链接着的另外一方进程因为 reason 原因终止。结果等同于另外一方进程自己调用 exit(reason)。

异常源： exit(spawn_link(fun() -> timer:sleep(50000) end), kill)

未捕获时的结果： ** exception exit: killed

捕获时的结果： {'EXIT', <0.58.0>, killed}

很奇怪，死亡进程发送给创建者进程的消息有所变化。现在创建者进程收到的消息是 killed 而不是 kill。这是因为 kill 是一个特殊的退出信号，会在下一节进行介绍。

异常源： exit(self(), kill)

未捕获时的结果： ** exception exit: killed

捕获时的结果： ** exception exit: killed

哇，看看。这个调用似乎根本不可能被捕获。下面的异常也很厉害。

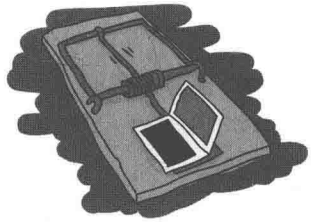
异常源： spawn_link(fun() -> exit(kill) end)

未捕获时的结果： ** exception exit: killed

捕获时的结果： {'EXIT', <0.67.0>, kill}

这就有些令人困惑了。当进程用 exit(kill)杀死自己时，如果我们的进程没有捕获退出信号，它死亡的原因是 killed。但是，当捕获了退出信号时，收到的消息中的原因却是 kill。

3. kill 的含义



虽然进程可以捕获退出信号，但是有时还是想强行杀死（kill）这个进程。也许某个捕获了退出信号的进程陷入了一个死循环，不能再接收任何消息了。kill 原因是一种特殊的信号，它不能被捕获。这可以保证任何被它终止的进程都肯定会死掉。通常，当其他所有手段都无效时，可以使用 kill。

由于 kill 原因无法被捕获，因此当其他进程接收到这个消息时需要被更改为 killed。如果不进行更改，那么所有和它链接在一起的其他进程都会因为同样的 kill 原因相继死亡，并会导致和这些进程具有链接关系的其他进程死亡，从而发生死亡的连锁反应。

这也解释了为何在没有捕获退出信号时，链接着的进程如果调用 exit(kill)，本进程收到的是 killed(信号被更改，防止连锁反应)，但是在本地捕获了退出信号时，看起来仍然是 kill 的原因。

如果你感到困惑，不要担心。很多程序员都有这个感受。退出信号是有些奇怪复杂。不过，

除了这里介绍的之外，再没有其他特别的情况了。一旦你理解了这里的内容，基本上就可以理解Erlang的所有并发错误管理了。

12.2 监控器

也许杀死进程并不是你想要的。也许你并不希望自己离开后整个世界都随你而去。也许你只想当一个跟踪者。如果是这样，那么监控器（monitor）可能就是你想要的，因为它们不会杀死进程。监控器是一种特殊类型的链接，有如下两点不同：

- 监控器是单向的；
- 在两个进程之间可以设置多个监控器（监控器可以叠加，每个监控器有自己的标识）。

如果一个进程想知道另外一个进程的死活，但是这两个进程之间并没有强的业务关联时，可以使用监控器。另外，监控器具有独立标识且可叠加的特性也非常有用。一开始可能感觉不到它的用处，但是当编写需要知道其他进程死活的库时，就会知道它的用途。那为何不能使用链接呢？因为链接不能堆叠，所以当某个库设置了两个进程间的链接，然后又去除了该链接时，就会破坏其他不相关的库在这两个进程间所设置的重要链接。监控器（以及可堆叠性）可以让开发库的程序员把自己使用的监控和其他不相关的监控分离开来。由于每个监控器都有一个唯一标识，因此可以对它们进行单独的识别和处理。

链接是比监控器更具组织性的概念。当设计自己的应用架构时，要确定哪个进程完成哪项任务，以及哪个进程要依赖于哪个进程。有些进程会监督其他一些进程，有些进程不能独立存活，等等。通常，这个结构要事先明确。对于这种情况，链接机制是适用的，但是在其他情况中，不必使用它。

但是，如果你调用了几个不同的库，它们都需要知道某个进程的生死时该怎么办呢？如果此时使用了链接，那么在需要解除进程链接时，马上就会出现问题。链接是不能叠加的，因此当解除两个进程间的链接时，其他库在这两个进程间建立的链接就全部被解除了，从而破坏了这些库对链接关系的假设。因此，你需要可叠加的链接，而监控器可以解决这个问题，因为它们可以单独地移除。另外，监控器的单向特性对库来说也很方便，因为其他进程不需要感知到这些库的死活。

那么监控器是什么样呢？我们来创建一个看看。创建监控器的函数是`erlang:monitor/2`，它的第一个参数永远是原子`process`，第二个参数是进程的pid。

```
1> erlang:monitor(process, spawn(fun() -> timer:sleep(500) end)).
#Ref<0.0.0.77>
2> flush().
Shell got {'DOWN',#Ref<0.0.0.77>,process,<0.63.0>,normal}
ok
```

每当被监控的进程死亡时，监控进程都会收到一条消息，格式是{'DOWN', *MonitorReference*, *process*, *Pid*, *Reason*}。其中的引用可以用来解除对一个进程的监控。记住，监控器是可叠加的，因此会收到多条DOWN消息。引用可以唯一确定一条DOWN消

息。还要说明一点，和链接一样，监控器也有一个具有原子性质的函数，可以在创建进程的同时监控它：spawn_monitor/1-3。

```
3> {Pid, Ref} = spawn_monitor(fun() -> receive _ -> exit(boom) end end).
{<0.73.0>,#Ref<0.0.0.100>}
4> erlang:demonitor(Ref).
true
5> Pid ! die.
die
6> flush().
ok
```

在上面的例子中，我们在进程死亡前解除了对它的监控，因此无法跟踪到它的死亡消息。还有另外一个函数 demonitor/2，它的功能会多一点。这个函数的第二个参数是一个选项列表。不过，只有两个可用选项：info 和 flush。

```
7> f().
ok
8> {Pid, Ref} = spawn_monitor(fun() -> receive _ -> exit(boom) end end).
{<0.35.0>,#Ref<0.0.0.35>}
9> Pid ! die.
die
10> erlang:demonitor(Ref, [flush, info]).
false
11> flush().
ok
```

info 选项用来指示某个监控器在被解除时是否还存在。这也是为何在第 10 行的调用中返回了 false。flush 选项会把邮箱中存在的 DOWN 消息都清除掉，因此 shell 中的 flush() 命令没有从 shell 进程的邮箱中找到任何消息。

12.3　命名进程

介绍完链接和监控器之后，还有另外一个问题仍没有解决：在检测到某个我们所依赖的进程死亡后，要做些什么呢？我们以 linkmon.erl 模块中的如下函数为例：

```
start_critic() ->
    spawn(?MODULE, critic, []).

judge(Pid, Band, Album) ->
    Pid !{self(), {Band, Album}},
    receive
        {Pid, Criticism} -> Criticism
    after 2000 ->
        timeout
    end.

critic() ->
    receive
        {From, {"Rage Against the Turing Machine", "Unit Testify"}} ->
            From ! {self(), "They are great!"};
```

```
        {From, {"System of a Downtime", "Memoize"}} ->
            From ! {self(), "They're not Johnny Crash but they're good."};
        {From, {"Johnny Crash", "The Token Ring of Fire"}} ->
            From ! {self(), "Simply incredible."};
        {From, {_Band, _Album}} ->
            From ! {self(), "They are terrible!"}
    end,
    critic().
```

假设我们现在正在商店里购物，想买些音像制品。有几张唱片感觉不错，但是我们不太确定。因此，决定联系一下我们的乐评人朋友。

```
1> c(linkmon).
{ok,linkmon}
2> Critic = linkmon:start_critic().
<0.47.0>
3> linkmon:judge(Critic, "Genesis", "The Lambda Lies Down on Broadway").
"They are terrible!"
```

由于太阳风暴（要假装得更真实些），通信的连接中断了。

```
4> exit(Critic, solar_storm).
true
5> linkmon:judge(Critic, "Genesis", "A trick of the Tail Recursion").
timeout
```

真令人讨厌。我们无法得到对唱片的评论了。为了让 critic 一直活着，我们编写了一个简单的“监督者”进程，它的唯一任务就是在 critic 死亡时重启它。

```
start_critic2() ->
    spawn(?MODULE, restarter, []).

restarter() ->
    process_flag(trap_exit, true),
    Pid = spawn_link(?MODULE, critic, []),
    receive
        {'EXIT', Pid, normal} -> % 正常死亡
            ok;
        {'EXIT', Pid, shutdown} -> % 手动终止，不是崩溃
            ok;
        {'EXIT', Pid, _} ->
            restarter()
    end.
```

上面代码中的 restarter 是一个独立的进程。它会启动 critic 的进程，如果 critic 由于异常原因死亡，会循环调用 restarter/0 并启动一个新的 critic 进程。注意，为了能够在需要时手工杀死 critic 进程，我们增加了一条{'EXIT', Pid, shutdown}处理语句。

这种实现存在一个问题，无法得到 critic 进程的 pid，因此我们无法调用它，得到它的意见。Erlang 中提供了一种解决方法：给进程命名。通过给进程起一个名字，可以用一个原子而不是一个不可理解的 pid 来标识一个进程。可以使用这个原子名给进程发送消息，和 pid 完全一样。

可以使用函数 erlang:register(Name,Pid)为进程命名。如果进程死亡了，它会自动

失去自己的名字。也可以使用函数 unregister/1 手工解除进程的名字注册。可以调用 registered/0 得到所有已注册进程的列表，或者通过 shell 命令 regs() 得到更详细的信息。可以重新编写 restarter/0 函数，代码如下：

```
restarter() ->
    process_flag(trap_exit, true),
    Pid = spawn_link(?MODULE, critic, []),
    register(critic, Pid),
    receive
        {'EXIT', Pid, normal} -> %正常死亡
            ok;
        {'EXIT', Pid, shutdown} -> %手动终止，不是崩溃
            ok;
        {'EXIT', Pid, _} ->
            restarter()
    end.
```

可以看出，不管重新创建的 critic 进程的 pid 是什么，register/2 都会给它命名为 critic。接下来，需要把 judge/3 函数的 pid 参数去掉。如下：

```
judge2(Band, Album) ->
    critic ! {self(), {Band, Album}},
    Pid = whereis(critic),
    receive
        {Pid, Criticism} -> Criticism
    after 2000 ->
        timeout
    end.
```

上面代码中，Pid = whereis(critic) 那一行用来得到 critic 的 pid，并用它在 receive 表达式中进行模式匹配。在模式匹配中使用这个 pid 是为了确保匹配到正确的消息。（此时邮箱中可能会有 500 条消息！）不过，这可能会带来问题。这段代码假定 critic 的 pid 在函数的头两行语句执行之间保持不变。但是，完全有可能发生如下情况：

```
1. critic ! Message
                        2. critic receives
                        3. critic replies
                        4. critic dies
5. whereis fails
                        6. critic is restarted
7. code crashes
```

下面这种情况也可能发生：

```
1. critic ! Message
                        2. critic receives
                        3. critic replies
                        4. critic dies
                        5. critic is restarted
6. whereis picks up
   wrong pid
7. Message never matches
```

一个进程中所发生的错误，如果处理不当会导致另外一个进程出现问题。在本例中，critic原子所代表的进程 pid 值可以被多个进程看到。这就是大家熟知的*共享状态*。问题是，多个不同的进程几乎可以同时访问、修改 `critic` 的值，导致信息不一致，发生软件错误。对这种情况有一个常用术语：*竞争条件*（race condition）。

竞争条件非常危险，因为它们的出现依赖于事件的时序。在几乎所有现存的并发和并行语言中，这种时序都和一些不可预测的因素有关，如处理器的繁忙程度、进程运行的位置以及程序所处理的数据类型。

保持冷静

你也许听说过 Erlang 中一般不存在竞争条件或者死锁，因此可以让并行程序更加安全。在很多情况下，这是正确的，但是也只是因为通过邮箱进行的消息传递机制限制了某些事件的顺序，并且 Erlang 语言严格限制了共享状态的数量。一般来讲，你绝对不能假设自己的代码中完全没有竞争条件。

上面的命名进程问题只是并发代码众多可能错误中的一种。

其他可能的出错情况包括多个不同进程同时访问计算机上的文件（修改它们），或者同时更新同一个数据库中的记录。

幸运的是，如果我们不认为命名进程一直保持不变，那么这个示例代码中的问题就很容易被修正。我们可以通过引用（用 `make_ref()` 创建）来作为识别消息的唯一值，并用它来保证从正确的进程收到了正确的消息。我们需要把函数 `critic/0` 重写为 `critic2/0`，把 `judge/3` 重写为 `judge2/2`：

```
judge2(Band, Album) ->
    Ref = make_ref(),
    critic ! {self(), Ref, {Band, Album}},
    receive
        {Ref, Criticism} -> Criticism
    after 2000 ->
        timeout
    end.

critic2() ->
    receive
        {From, Ref, {"Rage Against the Turing Machine", "Unit Testify"}} ->
            From ! {Ref, "They are great!"};
        {From, Ref, {"System of a Downtime", "Memoize"}} ->
            From ! {Ref, "They're not Johnny Crash but they're good."};
        {From, Ref, {"Johnny Crash", "The Token Ring of Fire"}} ->
            From ! {Ref, "Simply incredible."};
        {From, Ref, {_Band, _Album}} ->
            From ! {Ref, "They are terrible!"}
    end,
    critic2().
```

接下来，需要更改 restarter/0 函数，让它用 critic2/0 而不是 critic/0 创建进程。

更改完成后，其他函数仍然可以正常工作，代码的使用者也感觉不到差别。嗯，他们还是会感觉到的，因为我们更改了函数名和函数的参数个数，不过他们并不知道更改了哪些实现细节，也不知道为什么要这么更改。他们所感受到的只是代码变得更简单了，也无需在函数调用中传递 pid 了。下面是一个例子：

```
6> c(linkmon).
{ok,linkmon}
7> linkmon:start_critic2().
<0.55.0>
8> linkmon:judge2("The Doors", "Light my Firewall").
"They are terrible!"
9> exit(whereis(critic), kill).
true
10> linkmon:judge2("Rage Against the Turing Machine", "Unit Testify").
"They are great!"
```

现在，如果我们杀死了 critic 进程，会立即创建一个新的来继续回答我们的问题。这就是命名进程的有用之处。如果注册进程不存在，那么在调用 linkmon:judge2/2 时，函数内部的！操作符会抛出一个不正确参数错误，要确保当所依赖的命名进程不存在时，那些依赖进程不能运行。

在第 13 章中，我们会把所学到的 Erlang 并发编程知识付诸实践，编写一个真实的应用。

名有所值

记住，原子的个数是有限的（虽然很多）。绝对不要动态创建原子。这意味着，命名进程应该保留给那些单个 VM 实例中唯一的、重要的并且在整个应用运行期间都要一直存在的服务。

如果需要为那些暂时的或者 VM 中并不唯一的进程命名，就意味着可能需要把它们看成一个群组。明智的做法是把它们链接在一起，让它们共存亡，而不是试图使用动态的名字。

第 13 章

并发应用设计

一切都很完美。所有的概念你都理解了。但是，到目前为止，本书列举的所有例子都非常简单：计算器、树、从希思罗到伦敦问题等。是时候编写一些更有趣、更具教育意义的程序了。在本章中，我们将用 Erlang 的并发特性编写一个小型应用。这个应用不大，代码行数也不多，但是很有用，并且具有不错的扩展性。

我是一个做事不太有计划的人。功课、需要出门办理的事情、本书的编写、工作、会议、约会等事情，经常会把我弄得焦头烂额。结果，我的房间里到处都是清单，上面写着又忘记做的事情。我希望，你有时也会想要一种待办事宜提醒功能（不过你肯定不像我这样健忘），因为我们将编写一个事件提醒器应用，可以对要做的工作进行提醒，以防你忘记了约会时间。

13.1 理解问题

首先，需要弄清楚我们到底要做什么。“一个提醒器应用，”你回答到。“当然，”我说。但是这还不够。我们打算如何和软件进行交互？我们想让它为我们提供什么帮助？如何用进程来表达这个程序？我们如何知道要发送什么消息？

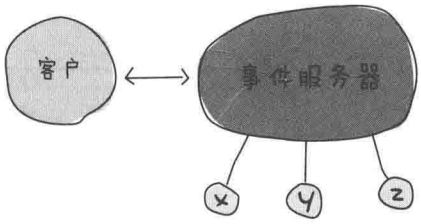
正如 Edward V. Berard 所说，“在水面上行走和基于规格说明开发软件可以非常容易，但前提是它们都先被冰冻起来”。所以，我们先来确定一份规格说明，并严格遵守它。

我们要编写的这个小软件将允许我们做如下事情。

- 增加一个事件。事件中包含最后期限（给出警告的时间）、事件名称以及事件描述。
- 当事件的最后期限到达时，显示警告。
- 根据名字取消事件。
- 通过命令行和系统交互，也可以扩展为其他方式（如 GUI、Web 页面、即时消息软件或者电子邮件）。

这个应用不需要持久化磁盘存储。在展示本章所涵盖的架构级概念时，也不需要它。不过，我们会介绍一下可以将持久化存储的逻辑放在哪里，以便你在需要将它添加到真实应用时参考，还会介绍一些帮助函数。由于没有持久化存储，因此必须能够在运行时更新代码。

程序的结构如下图所示，其中客户（client）、事件服务器（event server）、x、y 以及 z 都是进程：



事件服务器的任务如下：

- 接收来自客户进程的订阅；
- 把来自事件进程的消息转发给每个订阅者进程；
- 接收增加事件的消息（需要时会启动 x、y 和 z 进程）；
- 接收取消事件的消息，随后杀死事件进程。

事件服务器可以被客户进程终止，事件服务器的代码可以通过 shell 重新加载。

客户进程的任务如下：

- 向事件服务器发起订阅，并接收通知消息；
- 请求服务器增加一个包含具体内容的事件；
- 请求服务器取消一个事件；
- 监控服务器（为了知道服务器进程是否死亡）；
- 在需要时，关闭事件服务器。

要能够容易地支持多个客户进程对事件服务器进行订阅的情况。每个客户进程都可以充当与不同交互端点（GUI、Web 页面、即时消息软件、电子邮件等）通信的网关。

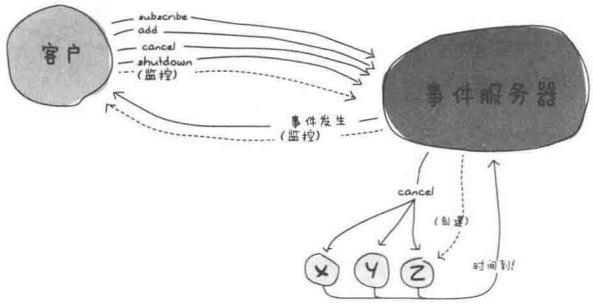
进程 x、y 和 z 代表着等待触发的通知（本质上，它们就是和事件服务器链接在一起的计时器）。它们的任务如下：

- 当计时器到时，给事件服务器进程发送一条消息；
- 接收事件取消消息，然后死亡。

注意，所有事件都会通知给所有客户（即时消息、电子邮件以及其他在本例中没有实现的系统），取消事件不用通知客户进程。本章所编写的软件仅供你我自己使用，我们假设只有一个用户在使用它。

下面是一幅更加复杂的图示，其中包含了所有可能的消息：

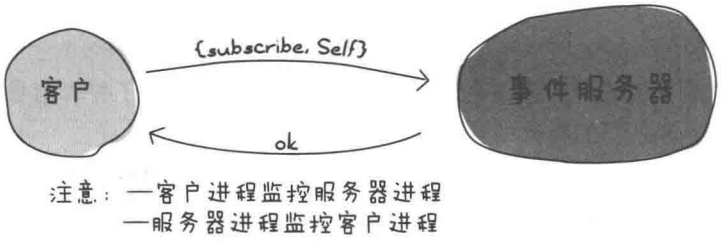
图中包含了所有的进程。其中的箭头表示消息，通过这种方式我们定义了一个高层协议，起码也算是一个高层协议的骨架。



在真实的应用中，把每个待提醒的事件都表示为一个进程的做法可能有些过度了，并且难以扩展到大量事件的场合。不过，因为这个应用只有你一个用户，所以这种设计没有问题。还可以使用如 `timer:send_after/2-3` 之类的函数避免创建过多的进程。

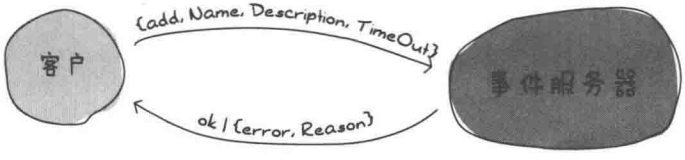
13.2 设计协议

既然已经知道了每个组件要完成的任务以及通信的内容，那么把所有要发送的消息都罗列出来，并明确消息的格式就是一个不错的主意。我们先来看看客户和事件服务器之间的消息通信：

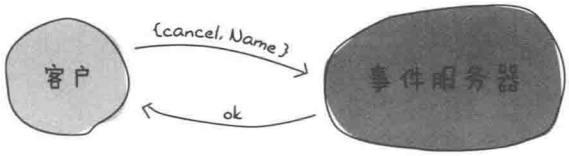


此处使用了两个监控器，因为客户进程和服务器进程之间没有明显的依赖关系。当然，如果没有服务器，客户进程就无法工作，但是服务器却可以在没有客户进程的情况下正常工作。这里本可以使用一个链接，但是因为系统可能会扩展到支持多个客户进程的情况，所以不能认为当服务器死亡时，其他客户进程也都想死亡。也不能认为一定可以把客户进程转变成一个系统进程并捕获服务器死亡的退出信号。

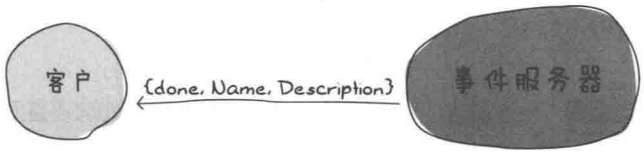
现在来看看下一组消息：



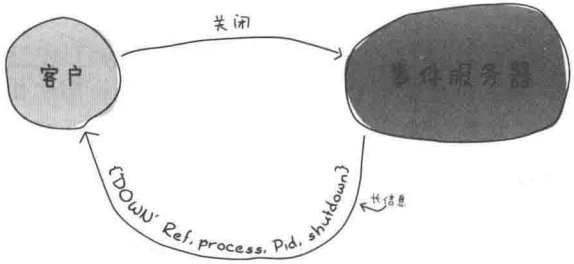
这条消息是向事件服务器中增加一个事件。除非出现问题（也许 `TimeOut` 的格式不对），否则会回应一条确认消息，格式为原子 `ok`。与之相反的操作，删除事件，如下图所示：



之后，到了事件的预定时间，事件服务器会发送一条通知消息：



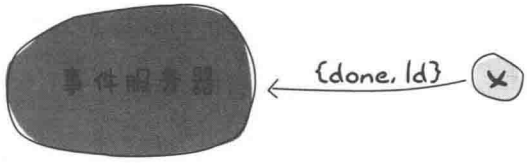
接下来，只剩下两个特殊情况下的消息了，即关闭服务器和服务器崩溃，如下：



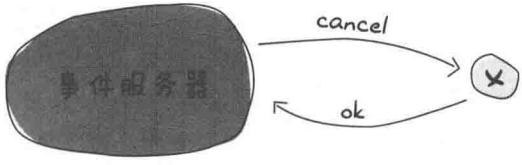
当服务器死亡时，并没有直接发送确认，因为监控器已经提供了警告消息。这些基本上就是客户进程和事件服务器之间的所有消息协议。

下面我们来定义事件服务器和事件进程之间的消息。在开始之前，有一件事需要注意：让事件服务器和事件进程链接起来是非常有用的。因为，如果服务器死亡了，那么我们会希望所有事件进程都死亡，因为没有了服务器，它们也没有存在的意义。

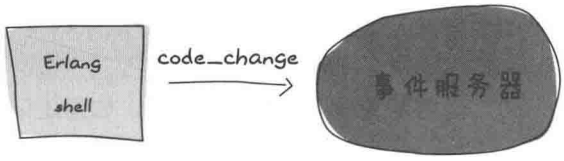
当事件服务器启动事件进程时，会给它们提供一个特殊的标识（事件名）。一旦某个事件的时间到了，它就发送一条消息，如下所示：



另一方面，事件进程也要留意来自事件服务器的取消消息：



还剩最后一条协议消息——升级服务器：



这条消息无需回应。在真正实现这个特性时，你会明白这样做的道理。

现在协议定义完了，并且进程的层级关系也基本弄清楚了，接下来可以动手实现这个项目了。

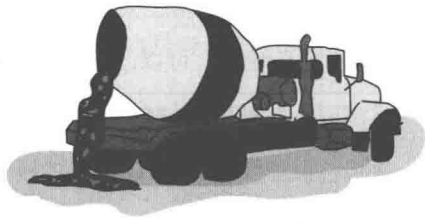
13.3 目录结构

在开始前，应该先要创建一个标准的 Erlang 项目目录结构，如下所示：

```
ebin/
include/
priv/
src/
```

这些目录中存放的文件如下：

- ebin/目录存放编译过的源代码；
- include/目录存放可以被其他应用包含的.hrl 文件（私有的.hrl 文件通常放在 src/目录中）；
- priv/目录存放需要和 Erlang 交互的可执行文件，如特定的驱动库等。在本项目中并没有用到这个目录；
- src/目录存放所有的.erl 文件。



在标准 Erlang 项目中，目录结构之间会有少许差异。可以增加一个 conf/目录来存放特定的配置文件，增加一个 doc/目录来存放文档，还可以增加一个 lib/或者 deps/目录存放应用运行需要的第三方库。正式发布的商用 Erlang 产品通常会使用一些不同的目录名，不过一般都会包含我们项目中定义的 4 个目录名，因为这是标准 OTP 实践的一部分。

13.4 事件模块

我们先来编写事件模块，因为它需要的依赖最少。它可以在没有事件服务器和客户函数的情况下独立运行。

进入 `src/`目录，并在其中创建一个 event.erl 文件，会用这个文件来实现应用中的 `x`、`y` 和 `z` 事件。

在编写代码前，我必须得提醒一下，目前的协议是不完整的。协议中虽然说明了在进程之间发送的数据是什么，但是并没有明确一些复杂的细节：如何寻址，使用引用还是名字，等等。大部分的消息都被包装进形如`{Pid, Ref, Message}`的元组中，其中 `Pid` 是发送者进程，`Ref` 是唯一的消息标识符，用来唯一确定一条消息。如果没有这个引用，那么当先连续发送多条消息

再等待回应时，就不知道哪个回应对应哪条消息了。

13.4.1　事件和循环

event.erl 中的 loop/1 函数是进程运行的核心代码，如果还记得协议内容，应该可以写出它的大概骨架：

```
loop(State) ->
    receive
        {Server, Ref, cancel} ->
            ...
    after Delay ->
            ...
    end.
```

上面代码展示出了实现事件到期通知所需的超时处理，以及对服务器取消事件消息的处理。loop 函数中有一个 State 变量。State 变量中保存着计时器的超时值（单位是秒）以及事件名称（为了发送消息{done, Id}）之类的数据。其中还需要保存事件服务器的 pid，在发送通知时使用。

这就是循环状态中的所有内容了。现在我们来定义 state 记录，放置在文件顶部：

```
-module(event).
-compile(export_all).
-record(state, {server,
                name="",
                to_go=0}).
```

注意，使用-compile(export_all).是为了避免每次都更改导出函数列表。一旦模块开发完毕，请把它替换成一个真正的-export([...]).序列。

状态定义好后，可以改进一下主循环的代码：

```
loop(S = #state{server=Server}) ->
    receive
        {Server, Ref, cancel} ->
            Server ! {Ref, ok}
    after S#state.to_go*1000 ->
        Server ! {done, S#state.name}
    end.
```

其中，乘以 1 000 的操作是为了把 to_go 的值从秒变换成毫秒。也可以调用 timer:seconds/1，这个函数会把秒换算成毫秒，得到的结果一样。

保持冷静

注意，语言缺陷！需要在 loop 函数头中绑定变量 Server，因为要在 receive 语句中用它来进行模式匹配。别忘了记录是拼凑出来的！表达式 S#state.server 会被秘密地转换成 element(2, S)，这不是一个合法的可匹配模式。

不过，after 语句中的 S#state.to_go 没有问题，因为它是一个可以以后再求值的表达式。

现在，我们来测试一下 loop 函数：

```
6> c(event).
{ok,event}
7> rr(event, state).
[state]
8> spawn(event, loop, [#state{server=self(), name="test", to_go=5}]).
<0.60.0>
9> flush().
ok
10> flush().
Shell got {done,"test"}
ok
11> Pid = spawn(event, loop, [#state{server=self(), name="test", to_go=500}]).
<0.64.0>
12> ReplyRef = make_ref().
#Ref<0.0.0.210>
13> Pid ! {self(), ReplyRef, cancel}.
{<0.50.0>,#Ref<0.0.0.210>,cancel}
14> flush().
Shell got {#Ref<0.0.0.210>,ok}
ok
```

我们先用 rr(Mod) 命令导入 event 模块中的记录。然后把 shell 进程当成服务器（self()）启动事件进程。5 s 后会触发这个事件。在 3 s 后，执行了第 9 条命令，在 6 s 后，执行了第 10 条命令。可以看到，第二次尝试时确实收到了{done, "test"}消息。

之后，我们又尝试了取消功能（把时间设置为 500 s，这样可以有充足的时间在 shell 中输入取消命令）。我们创建了引用、发送了消息，并得到了具有同样引用的回应消息，因此我们知道这条 ok 回应消息就是来自这个进程而不是系统中的其他任何进程。

取消消息中包含着引用，但是在 done 消息中没有使用引用，这只是因为我们并没有期望它来自某个特定的进程（任何进程都行，我们不会在 receive 中进行匹配），也不想对它进行回应。

再试试另外一个测试。如果有个事件明年才发生会怎么样呢？

```
15> spawn(event, loop, [#state{server=self(), name="test", to_go=365*24*60*60}]).
<0.69.0>
16>
=ERROR REPORT==== DD-MM-YYYY::HH:mm:SS ===
Error in process <0.69.0> with exit value: {timeout_value,[{event,loop,1}]}
```

哎哟。看起来我们遇到了一个实现层面的限制。Erlang 的超时值最大只能设置为 50 天（以毫秒为单位）。这也许不是那么重要，但是我之所以在这里提及它，有以下 3 个原因。

- 在本章写到一半时，这个问题给我的模块编写和测试带来过麻烦。
- Erlang 并非在所有方面都是完美的。这个问题就是没有按照 Erlang 实现者的意图使用计时器造成的。
- 这并不是一个真正的问题。我们可以绕过它。

对这个问题，我们的解决方案是，编写一个函数，该函数会在超时值过长时把它分成多个片

段。这需要对 loop/1 函数做些修改以支持这种方案。分段的方法是，把超时时间值等分成多个 49 天（因为限制是 50 天），然后再和剩余时间一起放到一个列表中。列表中时间的总秒数和原始时间相等：

```
%%  由于 Erlang 受 49 天（49*24*60*60*1000 毫秒）的时间限制，
%%  因此需要使用这个函数
normalize(N) ->
    Limit = 49*24*60*60,
    [N rem Limit | lists:duplicate(N div Limit, Limit)].
```

函数 lists:duplicate/2 会按照第一个参数指定的份数，复制第二个参数指定的表达式（[a,a,a] = lists:duplicate(3, a)）。如果给 normalize/1 传入值 98*24*60*60+4，就会返回[4,4233600,4233600]。

采用这个新形式的 loop/1 函数的代码如下：

```
%% loop 使用一个时间列表绕过~49 天的超时限制
loop(S = #state{server=Server, to_go=[T|Next]}) ->
    receive
        {Server, Ref, cancel} ->
            Server ! {Ref, ok}
    after T*1000 ->
        if Next =:= [] ->
            Server ! {done, S#state.name};
           Next =/= [] ->
            loop(S#state{to_go=Next})
        end
    end.
```

这段代码会取出 to_go 列表的第一个元素，并把这个元素的值设定为超时时间。超时后，会接着使用列表中的下一个值。如果列表为空，就说明全部超时结束，要给服务器发送通知消息。否则，loop 函数会一直处理列表中的剩余值，直到完成。

现在可以测试这个修订过的 loop 函数。它应该和之前一样工作正常，不过现在可以支持以年为单位的超时时间了。

13.4.2　增加接口

每当启动一个事件进程时，都要手工调用 event:normalize(N)，非常烦人，我们关于超时的解决方案绝对不能干扰到使用代码的程序员。解决这个问题的标准做法是，定义一个 init 函数，由它来初始化所有数据，供 loop 函数使用。在实现这个函数时，我们也顺便实现一下标准的 start 和 start_link 函数：

```
start(EventName, Delay) ->
    spawn(?MODULE, init, [self(), EventName, Delay]).

start_link(EventName, Delay) ->
    spawn_link(?MODULE, init, [self(), EventName, Delay]).
```

```
%%% 事件模块的内部实现
init(Server, EventName, Delay) ->
    loop(#state{server=Server,
                name=EventName,
                to_go=normalize(Delay)}).
```

现在，接口整洁多了。不过在测试之前，最好给我们唯一需要发送的那条消息——cancel消息——定义出自己的接口函数。

```
cancel(Pid) ->
    %% 设置监控器，以免进程已经死亡了
    Ref = erlang:monitor(process, Pid),
    Pid ! {self(), Ref, cancel},
    receive
        {Ref, ok} ->
            erlang:demonitor(Ref, [flush]),
            ok;
        {'DOWN', Ref, process, Pid, _Reason} ->
            ok
    end.
```

噢，一个新技巧！这里使用了一个监控器来检查进程是否还活着。如果进程已经死了，就可以避免无谓的等待，直接返回协议中规定的 ok。如果进程发回了带有引用的回应，那么我们知道它就快死了，于是就移除了引用，避免我们不再关心进程是否死亡后还能收到 DOWN 消息。注意，其中使用了 flush 选项，这个选项会将调用 demonitor 之前发送过来的 DOWN 消息清除掉。

我们来测试一下这些函数：

```
17> c(event).
{ok,event}
18> f().
ok
19> event:start("Event", 0).
<0.103.0>
20> flush().
Shell got {done,"Event"}
ok
21> Pid = event:start("Event", 500).
<0.106.0>
22> event:cancel(Pid).
ok
```

一切正常！

事件模块中还有一件比较讨厌的事情，必须要以秒为单位输入事件发生的时间。如果能够使用像 Erlang 中的 datetime（{{*Year*, *Month*, *Day*}, {*Hour*, *Minute*, *Second*}}）那样的标准格式就舒服多了。在模块中增加如下函数，它会计算出当前计算机时间和你指定时间之间的差值。

```
time_to_go(TimeOut={{_,_,_}, {_,_,_}}) ->
    Now = calendar:local_time(),
    ToGo = calendar:datetime_to_gregorian_seconds(TimeOut) -
```

```
        calendar:datetime_to_gregorian_seconds(Now),
    Secs = if ToGo > 0 -> ToGo;
              ToGo =< 0 -> 0
    end,
    normalize(Secs) .
```

哦哟，calendar 模块中的函数名字太另类了。这个函数会计算出当前时间和事件触发时间之间的秒数。如果触发时间已经过去，就返回 0，这样可以立即通知服务器。现在把 init 函数中对 normalize/1 的调用替换成这个函数。把 Delay 变量重命名为 DateTime，这样更具说明性。

```
init(Server, EventName, DateTime) ->
    loop(#state{server=Server,
                name=EventName,
                to_go=time_to_go(DateTime)}).
```

事件模块编写完了，我们可以休息一会儿。请启动一个事件，去享用一品脱（半升）牛奶或者啤酒吧，别忘了及时回来看看是否收到了事件通知消息。

13.5　事件服务器

现在来编写事件服务器模块。根据协议，它的骨架应该如下所示：

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

loop(State) ->
    receive
        {Pid, MsgRef, {subscribe, Client}} ->
            ...
        {Pid, MsgRef, {add, Name, Description, TimeOut}} ->
            ...
        {Pid, MsgRef, {cancel, Name}} ->
            ...
        {done, Name} ->
            ...
        shutdown ->
            ...
        {'DOWN', Ref, process, _Pid, _Reason} ->
            ...
        code_change ->
            ...
        Unknown ->
            io:format("Unknown message: ~p~n",[Unknown]),
            loop(State)
    end.
```

和之前一样，那些需要回应的消息，采用{*Pid*, *Ref*, *Message*}的格式。

服务器在其状态中需要保存两样东西：向它发起订阅的客户列表以及它创建的所有事件进程

的列表。

协议中规定，当事件发生时，事件服务器会收到{done, *Name*}消息，但是发送给客户进程的消息是{done, *Name*, *Description*}。这样做的目的是为了尽量最小化消息开销，并让事件进程只关注对它必要的东西。客户列表和事件列表的定义如下：

```
-record(state, {events,      %%#event{}记录列表
                clients}). %%pids 列表

-record(event, {name="",
                description="",
                pid,
                timeout={{1970,1,1},{0,0,0}}}).
```

可以把这个记录定义放到 loop 函数的函数头中：

```
loop(S = #state{}) ->
    receive
        ...
    end.
```

可以把事件和[客户都定义成有序字典。这两者的数量短时间内都不太可能达到数百个之多。在第 9 章中曾经讲过，有序字典最适合这种场景。下面是进行初始化的 init 函数：

```
init() ->
    %% 从静态文件中加载事件的逻辑可以放在这里
    %% 需要给 init 函数传递一个参数，用来指定从哪个文件中寻找事件。然后即可进行加载
    %% 还可以通过这个函数直接把事件传递给服务器
    loop(#state{events=orddict:new(),
                clients=orddict:new()}).
```

完成了骨架和初始化工作，接下来可以逐个实现消息处理。

13.5.1 处理消息

第一条是订阅消息。我们希望把所有订阅者都保存在一个列表中，这样当事件发生时，就可以通知它们。同时，协议中也提到要对订阅者进行监控。这样做是有意义的，因为我们不想保存那些已经崩溃的客户进程，并继续给它们发送没用的消息，处理代码如下：

```
{Pid, MsgRef, {subscribe, Client}} ->
    Ref = erlang:monitor(process, Client),
    NewClients = orddict:store(Ref, Client, S#state.clients),
    Pid ! {MsgRef, ok},
    loop(S#state{clients=NewClients});
```

loop/1 函数在处理这个消息时，创建了一个监控器，并把客户进程信息以 Ref 为主键保存在一个有序字典中。这样做的原因很简单：当我们收到了监控器的 EXIT 消息，其中包含着这个引用，仅在此时，才需要再次使用这个客户 ID（用来从有序字典中删除这个客户进程的数据项）。

下一个要关注的是事件增加消息。对它的处理可能会返回一条出错消

息。我们只会检查收到的时间数据是否有效。虽然对于是否满足{{*Year*,*Month*,*Day*},{*Hour*,*Minute*,*seconds*}}这种格式的检查比较容易，但是还要保证订阅的事件不能发生在如非闰年的 2 月 29 日之类不存在的日期上。此外，我们也不想接收不可能的日期值，如“5 小时，负 1 分钟，75 秒”。这些验证工作放在一个函数中就可以了。

在验证时，要使用的工具函数是 calendar:valid_date/1。正如函数名字所表达的那样，这个函数会检查日期的有效性。遗憾的是，calendar 模块的另类不仅体现在前面见过的函数名上，这个模块中根本就没有检查{H,M,S}有效性的函数。我们得自己编写一个，当然也要遵循另类的命名方式。

```
valid_datetime({Date,Time}) ->
    try
        calendar:valid_date(Date) andalso valid_time(Time)
    catch
        error:function_clause -> %% 不符合{{D,M,Y},{H,Min,S}}格式
            false
    end;
valid_datetime(_) ->
    false.

valid_time({H,M,S}) -> valid_time(H,M,S).
valid_time(H,M,S) when H >= 0, H < 24,
                       M >= 0, M < 60,
                       S >= 0, S < 60 -> true;
valid_time(_,_,_) -> false.
```

现在，可以把 valid_datetime/1 函数用在增加事件的消息处理中了。

```
{Pid, MsgRef, {add, Name, Description, TimeOut}} ->
    case valid_datetime(TimeOut) of
        true ->
            EventPid = event:start_link(Name, TimeOut),
            NewEvents = orddict:store(Name,
                                      #event{name=Name,
                                             description=Description,
                                             pid=EventPid,
                                             timeout=TimeOut},
                                      S#state.events),
            Pid ! {MsgRef, ok},
            loop(S#state{events=NewEvents});
        false ->
            Pid ! {MsgRef, {error, bad_timeout}},
            loop(S)
    end;
```

如果时间有效，就创建一个新的事件进程，接着把事件数据保存在事件服务器的状态中，然后给调用者发送一条确认消息。如果时间格式错误，并没有默不作声地接受这个错误或者让服务器崩溃，而是直接通知调用者，还可以增加一些额外的检查，如名字冲突或者其他限制等。（别忘了更新协议文档！）

协议中定义的下一条消息是事件取消消息。客户在取消一个事件时绝不会失败，因此处理代码比较简单。只要检查一下事件是否在进程状态记录中就可以了。如果在状态记录中，使用之前定义的 event:cancel/1 函数杀死事件进程并返回 ok。如果不在，通知调用者一切正常——事件进程没有运行，这正是调用者希望的。

```
{Pid, MsgRef, {cancel, Name}} ->
    Events = case orddict:find(Name, S#state.events) of
                 {ok, E} ->
                     event:cancel(E#event.pid),
                     orddict:erase(Name, S#state.events);
                 error ->
                     S#state.events
             end,
Pid ! {MsgRef, ok},
loop(S#state{events=Events});
```

现在，由客户主动发给事件服务器的消息处理已经实现完毕。接下来处理服务器和事件进程之间的交互。共有两个消息要处理：取消事件（这个已经完成）和事件超时。事件超时消息很简单，就是{done, Name}：

```
{done, Name} ->
    case orddict:find(Name, S#state.events) of
        {ok, E} ->
            send_to_clients({done, E#event.name, E#event.description},
            S#state.clients),
            NewEvents = orddict:erase(Name, S#state.events),
            loop(S#state{events=NewEvents});
        error ->
            %%事件取消的同时，超时也被触发了，会进入这个分支
            loop(S)
    end;
```

函数 send_to_clients/2 的功能如其名字描述，定义如下：

```
send_to_clients(Msg, ClientDict) ->
    orddict:map(fun(_Ref, Pid) -> Pid ! Msg end, ClientDict).
```

loop 函数的核心代码基本上就是这些了。剩余的都是对各种状态消息的处理：客户进程死亡了、关闭服务器、代码升级等。实现如下：

```
shutdown ->
    exit(shutdown);
{'DOWN', Ref, process, _Pid, _Reason} ->
    loop(S#state{clients=orddict:erase(Ref, S#state.clients)});
code_change ->
    ?MODULE:loop(S);
Unknown ->
    io:format("Unknown message: ~p~n",[Unknown]),
    loop(S)
```

上面代码中第一条消息（shutdown）的含义非常明显。收到了关闭消息，让进程死亡。如果想把状态存储到磁盘中，可以把逻辑放在这个地方。如果想实现更加安全的存储/退出语义，就要在每条 add、cancel 和 done 消息处增加存储逻辑。从磁盘加载事件的逻辑可以放在 init 函数中，事件加载完毕，就可以创建事件进程了。

对'DOWN'消息的处理也非常简单。收到它说明一个客户进程死亡了，因此需要把该客户从进程状态的客户列表中删除掉。

现在，未知的消息只要使用 io:format/2 打印出来供调试使用就可以了。在实际的产品应用中，应该要使用专门的日志模块。否则，所有有用信息就都被浪费掉了，因为在生产环境中无人关注这种输出方式。

接下来是代码变更消息。这个消息非常有意思，所以放在单独一节中。

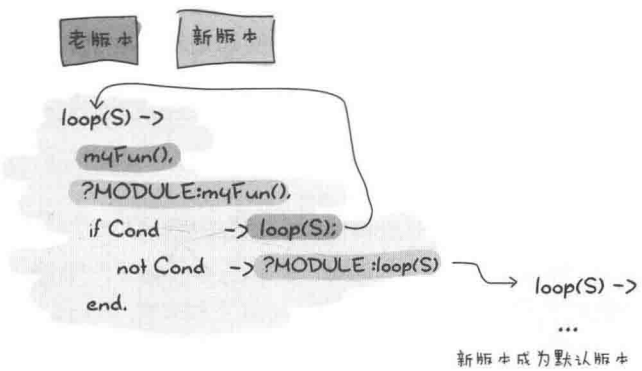
13.5.2　代码热升级

为了热加载代码，Erlang 内置了*代码服务器*（code server）。简单来讲，代码服务器是一个 VM 进程，管理一张 ETS 表（内存数据表，内置于 VM 中，会在第 25 章中介绍）。代码服务器在内存中为每个模块保存两个版本，这两个版本可以同时运行。当用 c(Module) 编译了一个模块，并用 l(Module) 或者 code 模块（可以查阅 Erlang 文档了解 code 模块）中的函数加载该模块时，会自动加载这个模块的新版本。

Erlang 中函数有本地和外部两种调用方式，这是一个很重要的概念。本地调用指的是那些可以针对非导出函数进行的调用。本地调用的格式为 *Name*(*Args*)。外部调用只能用于导出函数，格式为 *Module*:*Function*(*Args*)。外部调用更准确的名字是*完全限定调用*（fully qualified call）。

当 VM 中加载了一个模块的两个版本时，所有本地调用都针对进程中当前运行的模块版本。不过，完全限定调用则是永远针对代码服务器中模块的最新版本。因此，如果在完全限定调用中进行了本地调用，那么这些本地调用使用的都是最新版本的代码。

由于 Erlang 中的每个进程/actor 为了改变状态，都需要进行递归调用，因此可以通过一个外部递归调用加载进程的新版本。



> **注意** 如果在进程仍在运行模块的第一个版本时又加载了这个模块的第三个版本，那么这个进程会被 VM 杀死。VM 会认为这个进程是一个没有监督者的孤儿进程，或者没有提供自身升级的方法。如果没有进程运行在最旧的版本上，那么会直接把最旧的版本移除，把最新的放进来。

Erlang 中提供了多种把自己的代码和系统模块绑定在一起的方法，每当加载了模块的新版本时，系统模块会发送相应的消息。基于此，可以在收到这种消息时触发一个模块重加载动作，这个重加载动作最好始终用一个代码升级函数完成，如 `MyModule:Upgrade(CurrentState)`，这个函数会根据新版本的要求对状态数据结构进行转换。OTP 框架中已经实现了这种“订阅”处理机制，我们会在第 14 章中进行介绍。对于提醒器应用，我们不使用代码服务器，而是直接从 shell 上发送一条自定义的 `code_change` 消息，实现最基本的代码重加载功能。这些内容已经足以用来实现代码热加载了。下面是一个一般化的示例：

```
-module(hotload).
-export([server/1, upgrade/1]).

server(State) ->
    receive
        update ->
            NewState = ?MODULE:upgrade(State),
            ?MODULE:server(NewState); %% 进入模块的新版本循环中
        SomeMessage ->
            %% 完成一些处理
            server(State) %% 保持在进程当前运行的模块版本中
    end.

upgrade(OldState) ->
%% 在这里转换并返回状态
```

可以看出，上面的`?MODULE:loop(S)`符合我们讲的调用模式。

13.5.3 隐藏消息细节

隐藏消息细节！如果你期望他人能基于你的代码和进程构建应用，就必须把消息细节隐藏在接口函数中。`evserv` 模块的接口函数如下：

```
start() ->
    register(?MODULE, Pid=spawn(?MODULE, init, [])),
    Pid.

start_link() ->
    register(?MODULE, Pid=spawn_link(?MODULE, init, [])),
    Pid.

terminate() ->
    ?MODULE ! shutdown.
```

我们对服务器进程进行了注册，因为目前同时只能运行一个服务器进程实例。如果想让提醒

器应用可以支持多个用户，那么用 global 模块来注册名字是一个不错的主意，当然使用 gpoc 库进行注册是更好的方案。对这个示例应用来说，上面使用的方法就足够了。

> **注意**　gproc 库是一个 Erlang 进程字典，提供了许多内置进程字典所没有的有用特性，例如，可以用任意数据项充当进程别名、一个进程可以有多个名字、等待其他进程注册完毕、具备原子语义的名字移交以及计数器。如有需要，可以从 http://github.com/uwiger/gproc 获取实现代码。

接下来要隐藏的是之前编写的第一条消息：订阅消息。协议规范中要求有一个监控器，可以放在这里创建。如果创建监控器返回的引用包含在收到的 DOWN 消息中，调用者就知道服务器进程死亡了。

```
subscribe(Pid) ->
    = erlang:monitor(process, whereis(?MODULE)),
    ?MODULE ! {self(), Ref, {subscribe, Pid}},
    receive
        {Ref, ok} ->
            {ok, Ref};
        {'DOWN', Ref, process, _Pid, Reason} ->
            {error, Reason}
    after 5000 ->
            {error, timeout}
    end.
```

下一个要隐藏的消息是事件增加消息：

```
add_event(Name, Description, TimeOut) ->
    Ref = make_ref(),
    ?MODULE ! {self(), Ref, {add, Name, Description, TimeOut}},
    receive
        {Ref, Msg} -> Msg
    after 5000 ->
        {error, timeout}
    end.
```

注意，上面代码中把可能收到的{error, bad_timeout}消息转给了客户进程。我们还可以调用 erlang:error(bad_timeout) 让客户进程崩溃。到底是让客户进程崩溃还是转发出错消息目前还是 Erlang 社区中存在争议的一件事情。下面是让客户进程崩溃的函数实现：

```
add_event2(Name, Description, TimeOut) ->
    Ref = make_ref(),
    ?MODULE ! {self(), Ref, {add, Name, Description, TimeOut}},
    receive
        Ref, {error, Reason}} -> erlang:error(Reason);
        {Ref, Msg} -> Msg
    after 5000 ->
        {error, timeout}
    end.
```

接下来是事件取消消息，它只有一个名字参数：

```
cancel(Name) ->
    Ref = make_ref(),
    ?MODULE ! {self(), Ref, {cancel, Name}},
    receive
        {Ref, ok} -> ok
    after 5000 ->
        {error, timeout}
    end.
```

最后，再为客户提供一个小功能——我们来实现一个函数，累积给定时间段内收到的所有消息。如果收到了消息，就把它们全部从进程邮箱中取出，这个函数会尽可能快地返回。

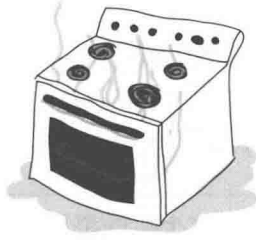
```
listen(Delay) ->
    receive
        M = {done, _Name, _Description} ->
            [M | listen(0)]
    after Delay*1000 ->
        []
    end.
```

如果客户要轮询消息更新，那么这个函数非常有用，但是如果应用需要一直监听消息，就可以使用推送方式，此时不需要这样的函数。

13.6 测试

现在可以编译并测试这个应用了。方便起见，我们会编写一个 Erlang makefile 来进行工程构建。创建一个名为 Emakefile 的文件，放到工程的根目录中。这个文件中的内容是 Erlang 数据项，指示 Erlang 编译器编译出.beam 文件。

```
{'src/*', [debug_info,
           {i, "src"},
           {i, "include"},
           {outdir, "ebin"}]}.
```



上面的元组告诉编译器在编译文件时增加 debug_info 选项（最好带上这个选项），在编译 src/目录中的模块时，从 src/和 include/目录中寻找头文件，并把编译的结果文件放到 ebin/目录中。

打开一个命令行，在工程的根目录中运行 erl -make 命令，文件会被编译并放到 ebin/目录中。输入命令 erl-pa ebin/启动 Erlang shell。-pa directory 选项告诉 Erlang VM 把这个目录增加到模块搜索路径中。

还可以先启动 Erlang shell，然后调用 make:all([load])。这个调用会在当前目录中寻找名为 Emakefile 的文件，重新进行编译（如果发生了变化），并加载新模块。

现在可以跟踪数以千计的事件了。试试看。

```
1> evserv:start().
<0.34.0>
```

```
2> evserv:subscribe(self()).
{ok,#Ref<0.0.0.31>}
3> evserv:add_event("Hey there", "test", FutureDateTime).
ok
4> evserv:listen(5).
[]
5> evserv:cancel("Hey there").
ok
6> evserv:add_event("Hey there2", "test", NextMinuteDateTime).
ok
7> evserv:listen(2000).
[{done,"Hey there2","test"}]
```

运行得很好。基于我们创建的几个基本接口函数，客户程序的编写应该非常简单了。

13.7　增加监督功能

为了让我们的示例应用更加健壮，需要编写一个像第 12 章中那样的“重启器”。创建一个名为 sup.erl 的文件来充当我们的监督者：

```
-module(sup).
-export([start/2, start_link/2, init/1, loop/1]).

start(Mod,Args) ->
    spawn(?MODULE, init, [{Mod, Args}]).

start_link(Mod,Args) ->
    spawn_link(?MODULE, init, [{Mod, Args}]).

init({Mod,Args}) ->
    process_flag(trap_exit, true),
    loop({Mod,start_link,Args}).

loop({M,F,A}) ->
    Pid = apply(M,F,A),
    receive
        {'EXIT', _From, shutdown} ->
            exit(shutdown); % 会杀死子进程
        {'EXIT', Pid, Reason} ->
            io:format("Process ~p exited for reason ~p~n",[Pid,Reason]),
            loop({M,F,A})
    end.
```

这个实现和第 12 章中的 restarter 有些相似，不过更通用一些。它可以接收任何实现了 start_link 函数的模块。它会一直重启它所监督的进程，除非监督者自己因为收到一个 shutdown 退出信号而终止。下面是它的用法：

```
1> c(evserv), c(sup).
{ok,sup}
2> SupPid = sup:start(evserv, []).
<0.43.0>
3> whereis(evserv).
<0.44.0>
```

```
4> exit(whereis(evserv), die).
true
Process <0.44.0> exited for reason die
5> exit(whereis(evserv), die).
Process <0.48.0> exited for reason die
true
6> exit(SupPid, shutdown).
true
7> whereis(evserv).
undefined
```

可以看出，杀死监督者时，会连带杀死它的子进程。

注意 我们会在第 18 章中介绍更高级、更灵活的监督者实现。当人们提到监督树时，所指的都是那些监督者。这里展示的只是监督者最基本的形式，和真正的监督者相比，并不适用于生产环境。

13.8 命名空间

因为 Erlang 的模块结构是扁平的（没有层级），所以有些应用中会出现模块名冲突的情况。user 模块就是一个常见的例子，几乎每个项目都会多次定义这个名字。这会和 Erlang 自带的 user 模块冲突。可以使用函数 code:clash/0 进行冲突检测。

由于可能存在冲突，因此一种常见的方法是把项目名作为前缀加在每个模块名的前面。对于本例来说，我们的提醒器应用模块应该被重命名为 reminder_evserv、reminder_sup 和 reminder_event。

有些程序员会想再增加一个模块，名字和应用名一样，把其他程序员使用这个应用时所需的常见调用都封装到这个模块中。这些函数可以是启动带有监督者的应用、向服务器发起订阅、增加和取消事件等。此外，还要注意其他可能冲突的命名空间，如进程注册名、数据库表等。

对于简单的 Erlang 并发应用来说，这些内容已经足够了。在这个例子中，我们很轻松地就得到了一大堆并发进程：监督者、客户、服务器、计时器进程（可以有数千个）等。无需在进程之间进行同步，没有锁，也没有真正的主处理循环。基于消息传递的方式可以容易地把应用划分成一些含有分离的关注点和任务的模块。

现在，你可以使用 evserv.erl 中的几个基本调用来构造客户程序，这些客户程序可以在 Erlang VM 之外和事件服务器进行交互，从而让我们的应用真正有用。

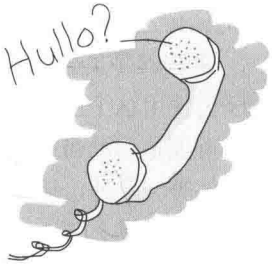
不过，在做这件事之前，我建议你先认真研究一下 OTP 框架。在接下来的几章中，我们会介绍 OTP 框架中的一些构建块，基于它们可以构建出更加健壮、更加优雅的应用。Erlang 的大部分威力都来自 OTP 框架的使用。OTP 框架是一个精工细作、设计优良的工具。每一个对自己有高要求的 Erlang 程序员都必须要了解它。

第 14 章

OTP 简介

在本章中，我们将学习 Erlang 的 OTP 框架。虽然 OTP 是开放电信平台的简称，但是现在它更多的是指那些具有电信应用属性的软件，而非电信软件本身。如果说 Erlang 的巨大优势一部分来自于其并发和分布式特性，还有一部分来自其错误处理能力，那么 OTP 框架则是第三部分。

在前面几章中，我们看到了一些例子，这些例子展示了使用 Erlang 语言的内置机制（包括链接、监控器、服务器、超时、捕获退出信号等）编写并发应用的常见实践方法。我们介绍了并发编程中一些需要注意的“陷阱”：要保证执行的顺序、要避免竞争条件，以及进程可能会随时死亡。还介绍了代码热加载、给进程命名、增加监督者等其他技术。



自己动手实现所有这些功能既耗时又容易出错，还可能会忘记某些边界条件，或者不小心掉入某些陷阱中。OTP 框架通过把这些核心实践组织成一组库来帮助我们处理这些问题，这些库都是经过仔细设计和多年实战检验的。每个 Erlang 程序员都应该使用它们。

同时，OTP 框架也是一组模块和标准，为构建应用提供帮助。由于大多数 Erlang 程序员都会使用 OTP，因此你在现实中遇到的 Erlang 应用基本上也都遵循这些标准。

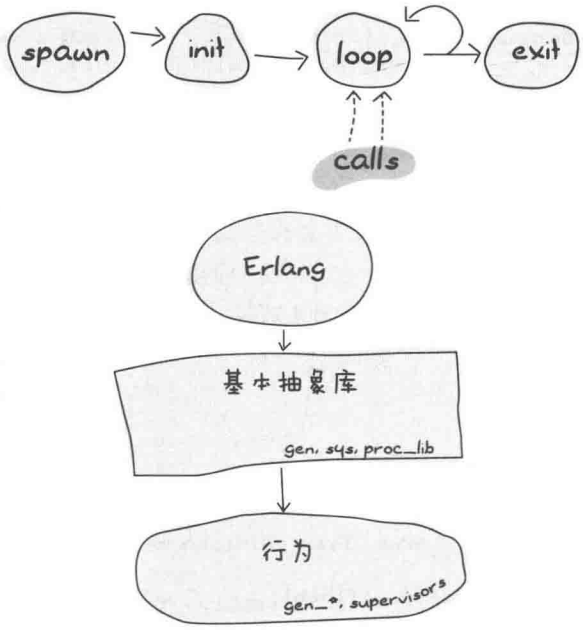
14.1 提炼通用进程

在前面的进程例子中，我们经常会根据非常具体的任务划分工作。几乎所有的进程中都会有一个函数负责创建新的进程，一个函数负责为进程提供初始值，一个函数负责主循环等。事实证明，不管这些进程是做什么用的，这些内容通常都会出现在所有并发程序中。

负责 OTP 框架的工程师和计算机科学家们发现了这些模式，把它们放到一组公共的库中。

OTP 库代码中所使用的抽象和我们使用的大部分抽象完全一样（如使用引用来标记消息），只不过相比我们的实现来说，它们经过了多年实践检验，编写得也更加仔细。OTP 库中包含有用于安全创建和初始化进程的函数、具备容错功能的消息发送函数，以及许多执行其他任务的函数。

不过，基本上不需要直接去使用这些函数库。因为它们所包含的抽象非常的基本和通用，所以在这些抽象之上又构建了许多更有意思的东西，称为行为（behavior）。



从本章开始，我们会研究一些进程的公共用法，以及如何把这些公共用法提炼出来并通用化。对于每项公共用法，我们都会讲讲 OTP 框架行为中的实现方式。

14.2 基础服务器

本章要介绍的通用模式我们之前已经使用过了。我们在第 13 章实现了一个事件服务器，其中使用了客户/服务器模型。事件服务器接收来自客户的消息，处理这些消息，然后根据协议回复客户。

14.2.1 kitty 服务器

在本章中，我们会编写一个非常简单的服务器，这样就可以关注其核心属性了。下面是 `kitty_server` 的代码实现：

```
%%%%% 简单实现版本
-module(kitty_server).
-export([start_link/0, order_cat/4, return_cat/2, close_shop/1]).

-record(cat, {name, color=green, description}).

%%% 客户 API
start_link() -> spawn_link(fun init/0).

%% 同步调用
order_cat(Pid, Name, Color, Description) ->
```

```
    Ref = erlang:monitor(process, Pid),
    Pid ! {self(), Ref, {order, Name, Color, Description}},
    receive
        {Ref, Cat} ->
            erlang:demonitor(Ref, [flush]),
            Cat;
        {'DOWN', Ref, process, Pid, Reason} ->
            erlang:error(Reason)
    after 5000 ->
        erlang:error(timeout)
    end.

%% 这个调用是异步的
return_cat(Pid, Cat = #cat{}) ->
    Pid ! {return, Cat},
    ok.

%% 同步调用
close_shop(Pid) ->
    Ref = erlang:monitor(process, Pid),
    Pid ! {self(), Ref, terminate},
    receive
        {Ref, ok} ->
            erlang:demonitor(Ref, [flush]),
            ok;
        {'DOWN', Ref, process, Pid, Reason} ->
            erlang:error(Reason)
    after 5000 ->
        erlang:error(timeout)
    end.

%%% 服务器函数
init() -> loop([]).

loop(Cats) ->
    receive
        {Pid, Ref, {order, Name, Color, Description}} ->
            if Cats =:= [] ->
                Pid ! {Ref, make_cat(Name, Color, Description)},
                loop(Cats);
               Cats =/= [] -> % 减少库存
                Pid ! {Ref, hd(Cats)},
                loop(tl(Cats))
            end;
        {return, Cat = #cat{}} ->
            loop([Cat|Cats]);
        {Pid, Ref, terminate} ->
            Pid ! {Ref, ok},
            terminate(Cats);
        Unknown ->
            %% 做些日志
```

```
            io:format("Unknown message: ~p~n", [Unknown]),
            loop(Cats)
    end.
%%% 私有函数
make_cat(Name, Col, Desc) ->
    #cat{name=Name, color=Col, description=Desc}.

terminate(Cats) ->
    [io:format("~p was set free.~n",[C#cat.name]) || C <- Cats],
    ok.
```

上面就是 kitty 店铺服务器的实现。其逻辑极其简单：只要给出猫咪的描述，就可以得到它。如果有人退还了一只猫咪，就会把它加到列表中，之后不管用户实际给出什么样的订单描述，都会自动返回这只猫咪（我们开这个 kitty 店铺就是为了赚钱）。

```
1> c(kitty_server).
{ok,kitty_server}
2> rr(kitty_server).
[cat]
3> Pid = kitty_server:start_link().
<0.57.0>
4> Cat1 = kitty_server:order_cat(Pid, carl, brown, "loves to burn bridges").
#cat{name = carl,color = brown,
description = "loves to burn bridges"}
5> kitty_server:return_cat(Pid, Cat1).
ok
6> kitty_server:order_cat(Pid, jimmy, orange, "cuddly").
#cat{name = carl,color = brown,
description = "loves to burn bridges"}
7> kitty_server:order_cat(Pid, jimmy, orange, "cuddly").
#cat{name = jimmy,color = orange,description = "cuddly"}
8> kitty_server:return_cat(Pid, Cat1).
ok
9> kitty_server:close_shop(Pid).
carl was set free.
ok
10> kitty_server:close_shop(Pid).
** exception error: no such process or port
in function kitty_server:close_shop/1
```

再审视一下这个模块的源代码，从中你会发现一些之前曾经应用过的模式。那些设置和解除监控器、使用计时器、接收数据、主循环处理、使用 `init` 函数等代码片段都不陌生。应该把这些一直重复编写的东西提炼出来。我们先从客户 API 开始。

14.2.2 通用化同步调用

在源代码中，首先注意到的是那两个同步调用非常相似。应该能够把这些调用放到之前提到过的抽象库中。现在，我们暂时先把它们提炼到一个新模块中，这个模块存放了 kitty 服务器的所有通用部分的代码。

```
-module(my_server).
```

```
-compile(export_all).

call(Pid, Msg) ->
    Ref = erlang:monitor(process, Pid),
    Pid! {self(), Ref, Msg},
    receive
        {Ref, Reply} ->
            erlang:demonitor(Ref, [flush]),
            Reply;
        {'DOWN', Ref, process, Pid, Reason} ->
            erlang:error(Reason)
    after 5000 ->
        erlang:error(timeout)
    end.
```

这个函数以一条消息和一个 pid 为参数，并以安全的方式转发了这条消息。

此后，可以把所有的消息发送都替换成调用这个函数。如果基于提炼出来的 my_server 重写一个新的 kitty 服务器，看起来应该像这样：

```
-module(kitty_server2).
-export([start_link/0, order_cat/4, return_cat/2, close_shop/1]).

-record(cat, {name, color=green, description}).

%%% 客户 API
start_link() -> spawn_link(fun init/0).

%% 同步调用
order_cat(Pid, Name, Color, Description) ->
    my_server:call(Pid, {order, Name, Color, Description}).

%% 异步调用
return_cat(Pid, Cat = #cat{}) ->
    Pid ! {return, Cat},
    ok.

%% 同步调用
close_shop(Pid) ->
    my_server:call(Pid, terminate).
```

14.2.3　通用化服务器循环

下一个比较大的通用代码块并不像 call/2 函数那样明显。注意，迄今为止我们所编写的每一个进程都有一个循环，这个循环的作用是对所有的消息进行模式匹配。这部分有点麻烦，不过我们只需要分离模式匹配逻辑和循环逻辑即可。一种快速的实现方法如下：

```
loop(Module, State) ->
    receive
        Message -> Module:handle(Message, State)
    end.
```

而专用模块的处理代码可以这样编写：

```
handle(Message1, State) -> NewState1;
handle(Message2, State) -> NewState2;
...
handle(MessageN, State) -> NewStateN.
```

这已经不错了，不过还可以把它变得更整洁些。

如果你认真阅读了 kitty_server 模块的源代码（我希望你这样做了!），应该会发现同步调用和异步调用分别有自己专门的调用方式。如果通用服务器实现能够提供一种清晰的方法来区分调用的类别那就更好了。

为了实现这个目标，需要在 my_server:loop/2 中匹配不同类型的消息。这意味着，需要对函数 call/2 做点更改，让同步调用变得更明显一点。我们采用的方法是在函数第二行的消息中增加一个原子 sync，如下所示：

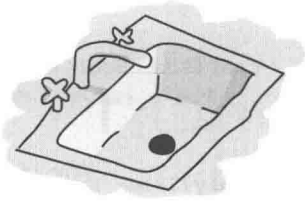
```
call(Pid, Msg) ->
    Ref = erlang:monitor(process, Pid),
    Pid! {sync, self(), Ref, Msg},
    receive
        {Ref, Reply} ->
            erlang:demonitor(Ref, [flush]),
            Reply;
        {'DOWN', Ref, process, Pid, Reason} ->
            erlang:error(Reason)
    after 5000 ->
        erlang:error(timeout)
    end.
```

然后，再为异步调用提供一个新函数 cast/2。代码如下：

```
cast(Pid, Msg) ->
    Pid ! {async, Msg},
    ok.
```

现在，主循环的实现如下：

```
loop(Module, State) ->
    receive
        {async, Msg} ->
            loop(Module, Module:handle_cast(Msg, State));
        {sync, Pid, Ref, Msg} ->
            loop(Module, Module:handle_call(Msg, Pid, Ref, State))
    end.
```



接下来，还可以增加针对非同步/异步消息（那些不小心发过来的消息）的处理分支，或者用于调试和代码热加载之类的其他处理分支。

主循环中有一个问题不太令人愉快，那就是存在抽象泄露。使用 my_server 的程序员在回应所发送的同步消息时，必须要知道

引用的信息。这就降低了抽象的效果。在使用时，必须理解所有讨厌的细节。下面是一个快速的修正：

```
loop(Module, State) ->
    receive
        {async, Msg} ->
            loop(Module, Module:handle_cast(Msg, State));
        {sync, Pid, Ref, Msg} ->
            loop(Module, Module:handle_call(Msg, {Pid, Ref}, State))
    end.
```

我们把 Pid 和 Ref 放到一个元组中，并给这个元组起一个类似于 From 之类的名字，然后将其作为一个参数变量传递给其他函数。使用者并不需要了解这个变量的内容细节。我们会再提供一个发送回应的函数，这个函数知道 From 的细节：

```
reply({Pid, Ref}, Reply) ->
    Pid ! {Ref, Reply}.
```

14.2.4　启动函数

接下来，还需要为启动函数（start、start_link 和 init）指定专用模块的名字和状态信息。增加这些函数后，模块代码如下：

```
-module(my_server).
-export([start/2, start_link/2, call/2, cast/2, reply/2]).

%%% 公共 API
start(Module, InitialState) ->
    spawn(fun() -> init(Module, InitialState) end).

start_link(Module, InitialState) ->
    spawn_link(fun() -> init(Module, InitialState) end).

call(Pid, Msg) ->
    Ref = erlang:monitor(process, Pid),
    Pid ! {sync, self(), Ref, Msg},
    receive
        {Ref, Reply} ->
            erlang:demonitor(Ref, [flush]),
            Reply;
        {'DOWN', Ref, process, Pid, Reason} ->
            erlang:error(Reason)
    after 5000 ->
        erlang:error(timeout)
    end.

cast(Pid, Msg) ->
    Pid ! {async, Msg},
    ok.

reply({Pid, Ref}, Reply) ->
```

```
    Pid ! {Ref, Reply}.

%%% 私有函数
init(Module, InitialState) ->
    loop(Module, Module:init(InitialState)).

loop(Module, State) ->
    receive
        {async, Msg} ->
            loop(Module, Module:handle_cast(Msg, State));
        {sync, Pid, Ref, Msg} ->
            loop(Module, Module:handle_call(Msg, {Pid, Ref}, State))
    end.
```

14.2.5 通用化 kitty 服务器

接下来，我们要重新实现 kitty 服务器，新的服务器模块是 kitty_server2，它是一个遵从 my_server 定义的接口的回调模块。该模块的公共接口和之前实现的完全一样，只是现在所有调用都转发给 my_server 了。

```
-module(kitty_server2).

-export([start_link/0, order_cat/4, return_cat/2, close_shop/1]).
-export([init/1, handle_call/3, handle_cast/2]).

-record(cat, {name, color=green, description}).

%%% 客户 API
start_link() -> my_server:start_link(?MODULE, []).

%% 同步调用
order_cat(Pid, Name, Color, Description) ->
    my_server:call(Pid, {order, Name, Color, Description}).

%% 异步调用
return_cat(Pid, Cat = #cat{}) ->
    my_server:cast(Pid, {return, Cat}).

%% 同步调用
close_shop(Pid) ->
    my_server:call(Pid, terminate).
```

注意，在模块的顶部又多加了一个-export()属性。其中的函数都是 my_server 模块需要的：

```
%%% 服务器函数
init([]) -> []. %% 此处无需做信息处理!

handle_call({order, Name, Color, Description}, From, Cats) ->
    if Cats =:= [] ->
        my_server:reply(From, make_cat(Name, Color, Description)),
```

```
        Cats;
        Cats =/= [] ->
        my_server:reply(From, hd(Cats)),
        tl(Cats)
    end;

handle_call(terminate, From, Cats) ->
    my_server:reply(From, ok),
    terminate(Cats).

handle_cast({return, Cat = #cat{}}, Cats) ->
    [Cat|Cats].
```

下面我们需要重新插入私有函数：

```
%% 私有函数
make_cat(Name, Col, Desc) ->
    #cat{name=Name, color=Col, description=Desc}.

terminate(Cats) ->
    [io:format("~p was set free.~n",[C#cat.name]) || C <- Cats],
    exit(normal).
```

别忘了把之前 `terminate/1` 中的 `ok` 改为 `exit(normal)`，否则服务器会一直运行。

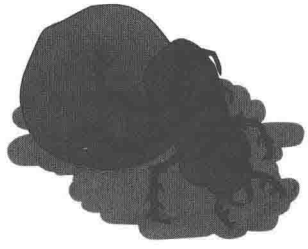
现在可以编译并测试新的代码了，并用和上一个版本完全一样的方式运行它。代码很相似，我们来看一下发生了哪些变化。

14.3　专用与通用

kitty 服务器的例子展示了 OTP 的精髓（从概念上讲）。这正是 OTP 的根本所在：找到所有通用的组件，把它们提炼到库中，保证它们可以良好运行，然后尽可能地重用它们。接下来就只需要关注那些专用部分，也就是那些总是随应用变化而变化的部分。

显然，对于 kitty 服务器这个例子来说，这样做看不出有多少好处。反而会有点为了抽象而抽象的感觉。如果要发布给客户的应用就是像 kitty 服务器这样，那么第一个版本就很好了。不过，对于更大型的应用，在分离代码的通用部分和专用部分上所付出的努力就是值得的。

假设一台服务器上运行着一些 Erlang 软件。其中有几个 kitty 服务器进程、一个兽医进程（给它发送受伤的猫，它会把治好的猫返还给你）、一个 kitty 美容进程和一个宠物食物服务器进程等。这些进程基本都是客户/服务器方式的。随着时间的流逝，系统会变得越来越复杂，充满了各种不同的服务器进程。



增加服务器进程，不仅会增加代码层面的复杂性，还会增加测试、维护以及理解方面的复杂性。每个服务器的实现都是不同的，由不同的人采用不同的风格进行编写。不过，如果所有这些服务器都共享通用的 my_server 抽象，那么可以极大地降低这些复杂性。你可以迅速理解模块的基本概念（“哦，它是个服务器！”），也只需要对这

个单一的通用实现进行测试和文档编写，从而可以把剩余的精力投入到每个服务器的专用部分实现上。

这意味着可以节省大量跟踪和解决 bug 的时间（对所有的服务器进程，只需在一个地方做这些工作），并且可以减少引入 bug 的数量。如果你总是在重写 `my_server:call/3` 或者进程的主循环代码，这不但非常耗时，而且漏掉步骤的可能性会激增，从而引入 bug。更少的 bug 意味着更少的夜间故障电话，这对所有人都绝对是一件好事（我敢打赌，你肯定也不喜欢下班后再回到办公室修复 bug）。

把通用部分和专用部分分离还会带来另外一个有价值的效果：单个模块的测试立即变得更容易了。如果你想对老的 kitty 服务器实现进行单元测试，每个测试都要启动一个进程，为它提供恰当的状态，发送消息，并希望收到想要的回应。而另一方面，在第二版的 kitty 服务器实现中，我们可以直接调用 `handle_call/3` 和 `handle_cast/2` 函数，检查它们返回的新状态就行了。根本不需要启动服务器进程，只要把状态作为参数传递给函数就行了。注意，这也意味着服务器的通用部分也更容易测试了，因为只需要实现一些非常简单的函数，而这些函数没有任何逻辑，我们可以只关注于想观察的行为。

以这种方式使用通用抽象还有一个不那么明显的优点，如果所有人在进程中都使用同样的后端实现，那么当有人对一个后端进行了优化使之更快时，所有使用它的进程就都会运行得更快一些。要想让这个原则在实际中有效，通常要求所有人都使用同样的抽象，并花时间去改进它们。幸运的是，在 Erlang 社区中，这正是 OTP 框架的做法。

在 kitty 服务器模块中，还有一些问题我们没有解决：进程命名、超时配置、调试信息、非期望消息处理、代码热加载的配合、特殊错误的处理、公共回复代码的提炼、服务器关闭的处理、保证服务器和监督者的配合等。对于本书来说，要解决所有这些问题太占篇幅了，但是这些内容在需要发布的真实产品中是必需的。这再次说明，自己动手解决这些问题是一件有风险的事情。不过，你（以及那些将要维护你应用的人）很幸运，Erlang/OTP 团队已经在 `gen_server` 行为中解决了所有这些问题。`gen_server` 有点像 `my_server` 的增强版，此外，它还经受过年复一年的测试和产品使用。

14.4 面向未来的回调

和本章中我们自己设计的接口类似，OTP 的 `gen_server` 行为也会要求提供一些进程的初始化和结束、基于消息发送的同步和异步处理以及其他任务的处理函数。

14.4.1 `init` 函数

第一个回调函数是 `init/1`。它和我们在 `my_server` 中实现的类似。它负责初始化服务器的状态，并完成服务器需要的所有一次性任务。这个函数可以返回`{ok, State}`、`{ok, State, TimeOut}`、`{ok, State, hibernate}`、`{stop, Reason}`以及 `ignore`。

常规的`{ok, State}`返回值无需解释，只要记住 `State` 会直接传给进程的主循环，并作为

进程的状态一直保存在那里就行了。当期望服务器在某个时间期限之前能收到一条消息时，可以使用 TimeOut 变量。如果到期没有收到任何消息，那么会给服务器发送一条特殊的消息（原子 timeout），可以在 handle_info/2（会在本章后面介绍）中处理这条消息。很少会在产品代码中使用这个选项，因为不能总是知道会收到哪条消息，而任意一条消息都会重置计时器。通常，更好的方法是使用 erlang:start_timer/3 之类的函数，可以获取更好的处理控制。

另一方面，如果确实觉得进程在很长一段时间内不会有什么消息要处理，并且担心内存问题，那么可以在元组中使用 hibernate 原子。一般来讲，hibernate 选项会缩减进程的状态，直到它收到一条消息，不过会多耗费些处理能力。如果在是否使用 hibernate 选项时存有疑惑，就说明不太需要它。

如果在初始化的过程中出现了错误，可以返回{stop, Reason}。

进程休眠细节

如果你有兴趣，这里有一个更技术性的关于进程休眠（hibernation）的定义。当调用 BIF 函数 erlang:hibernate(M,F,A) 时，当前运行进程的调用栈会被丢弃（这个函数不会返回）。接着会进行垃圾回收，把进程内存收缩为仅够存放进程数据的一块连续的内存堆。这基本上压缩了所有的数据，因此进程所占空间也会少很多。

当进程收到一条消息时，会用参数 A 调用函数 M:F，进程会重新执行。

注意　当执行 init/1 函数时，创建服务器的进程会被阻塞。这是因为它在等待一条“就绪”消息，这条消息由 gen_server 模块自动发送以确认一切正常。

14.4.2　handle_call 函数

函数 handle_call/3 用于处理同步消息。它有 3 个参数：Request、From 以及 State。它和我们在 my_server 中自己实现的 handle_call/3 非常相似。最大的不同在于回应消息的方式。在我们自己的服务器抽象中，必须要使用 my_server:reply/2 进行回应。在 gen_server 中，有 8 种不同的返回值可供选择，这些返回值都是元组形式，具体如下：

```
{reply,Reply,NewState}
{reply,Reply,NewState,TimeOut}
{reply,Reply,NewState,hibernate}
{noreply,NewState}
{noreply,NewState,TimeOut}
{noreply,NewState,hibernate}
{stop,Reason,Reply,NewState}
{stop,Reason,NewState}
```

在这些返回值中，TimeOut 和 hibernate 的工作方式和 init/1 中的一样。Reply 中的内容会被原封不动地发回给调用服务器的进程。

注意，共有 3 种不同的 noreply 选项。当使用 noreply 时，服务器的通用部分会认为你将自己发送回应消息。可以调用 gen_server:reply/2 发送回应，用法和 my_server:

reply/2 完全一样。

在绝大部分情况下，只需要使用 reply 元组。不过，有些情况确实需要使用 noreply，例如，希望由另外一个进程来替你发送回应，或者想先发送一条确认消息（“嗨！我收到消息了！”），然后要继续处理（处理完无需再回应）。如果这是所需要的场景，那么只能使用 gen_server:reply/2，否则，调用会超时然后崩溃。

14.4.3 handle_cast 函数

handle_cast/2 回调函数的工作方式和 my_server 中的也很相似。它的参数是 Message 和 State，用于异步消息的处理。和 handle_call/3 类似，其中也可以进行任何处理。不过，它只能返回 noreply 元组：

```
{noreply,NewState}
{noreply,NewState,TimeOut}
{noreply,NewState,hibernate}
{stop,Reason,NewState}
```

14.4.4 handle_info 函数

我曾经说过，我们自己实现的服务器不能处理和接口不相容的消息。嗯，handle_info/2 就是解决这个问题的。它和 handle_cast/2 非常相似，事实上，返回值也完全一样。它们之间的区别在于，这个回调函数只用来处理直接通过！操作符发送的消息，以及如 init/1 中的 timeout、监控器通知或者 EXIT 信号之类的特殊消息。

14.4.5 terminate 函数

当上面 3 种 handle_something 函数返回形如{stop, Reason, NewState}或者{stop, Reason,Reply, NewState}的元组时，会调用 terminate/2 函数。它有两个参数：Reason 和 State，分别对应 stop 元组中的同名字段。

当父进程（创建服务器的进程）死亡时，也会调用 terminate/2 函数，不过这只会发生在 gen_server 捕获了退出信号的时候。

> **注意** 如果在调用 terminate/2 时，原因不是 normal、shutdown 或者{shutdown, Term}，那么 OTP 框架会把这当成故障，并会把进程的状态、故障原因、最后收到的消息等记入日志。这让调试变得更加容易，可以帮助你快速定位问题。

这个函数和 init/1 正好相反，因此所有在 init/1 中做的动作都应该在 terminate/2 中有对应的取消动作。它是服务器进程的看门人——在确信所有人都离开后，负责锁好大门。当然，VM 本身也会完成一些清理工作，在进程退出时，VM 会删除所有 ETS 表（参见第 25 章）、关闭所有套接字（参见第 23 章），还会完成其他一些任务。注意，这个函数的返回值不是那么重要，因为调用它之后，代码就停止执行了。

14.4.6 code_change 函数

函数 code_change/3 用于代码升级。它的调用形式是 code_change(Previous Version, State, Extra)。其中，变量 PreviousVersion 在升级时是版本数据项本身（如

果忘记了，可以参考第 2 章），在降级时（就是加载老代码）是{down, Version}。State 变量中保存着服务器当前的所有状态数据，可以对其进行转换。

假如一开始我们使用有序字典来存储所有的数据。一段时间之后，有序字典变得越来越慢，我们决定用常规的字典把它替换掉。为了避免进程在接下来的调用中崩溃，可以在这个函数中安全地进行数据结构的转换。所要做的就是用{ok, NewState}返回新的状态。我们在第 22 章中介绍发布升级时会使用这个特性，也会介绍 Extra 变量。现在先不用关心这些内容。

现在，所有的回调函数都介绍完毕。如果感到有些困惑也不用担心。OTP 框架有时是存在一些环形依赖：要理解框架中的 A 部分，得先理解 B 部分，但是理解 B 部分又要求先理解 A 部分。解决这种困惑最好的方法就是自己去实际实现一个 gen_server。

14.5　gen_server 实践

现在，我们来构建 kitty_gen_server。它和 kitty_server2 相似，只在 API 方面有很少的改变。

首先，创建一个新的模块，并增加如下两行内容：

```
-module(kitty_gen_server).
-behavior(gen_server).
```

> **注意**　关键字 behavior 和 behaviour 对 Erlang 编译器来说都是合法的。

试着编译它。应该会出现如下信息：

```
1> c(kitty_gen_server).
./kitty_gen_server.erl:2: Warning: undefined callback function code_change/3
(behavior 'gen_server')
./kitty_gen_server.erl:2: Warning: undefined callback function handle_call/3
(behavior 'gen_server')
./kitty_gen_server.erl:2: Warning: undefined callback function handle_cast/2
(behavior 'gen_server')
./kitty_gen_server.erl:2: Warning: undefined callback function handle_info/2
(behavior 'gen_server')
./kitty_gen_server.erl:2: Warning: undefined callback function init/1
(behavior 'gen_server')
./kitty_gen_server.erl:2: Warning: undefined callback function terminate/2
(behavior 'gen_server')
{ok,kitty_gen_server}
```

编译通过了，不过有一些关于缺少回调函数的警告信息。这是因为我们使用了 gen_server 行为。简单来讲，行为（behavior）提供了一种手段，通过它可以让一个模块指定一组期望另外一个模块实现的函数。行为是一种契约，把高质量的通用部分代码和易出错的专用部分代码（你编写的）之间的交互协议封装起来。

自定义行为

定义自己的行为非常简单。只需要导出一个名为 behavior_info/1 的函数即可(译者注:现在推荐的做法是在模块中用-callback 标注回调函数类型),实现如下:

```
-module(my_behavior).
-export([behavior_info/1]).

%% init/1, some_fun/0 和 other/3 是所期望的回调函数。
behavior_info(callbacks) -> [{init,1}, {some_fun, 0}, {other, 3}];
behavior_info(_) -> undefined.
```

这样就定义完毕了。只要在模块中使用-behavior(my_behavior).即可。如果在实现行为的模块中忘记了某个回调函数,编译器会给出警告信息。

在原来的 kitty 服务器实现中,第一个函数是 start_link/0。它现在的实现如下:

```
start_link() -> gen_server:start_link(?MODULE, [], []).
```

第 1 个参数是回调模块名,第 2 个参数是要传递给 init/1 的数据项,第 3 个参数是服务器运行需要的调试选项。还可以在第一个位置处填加第 4 个参数{local, Name},这个参数用来给服务器注册名字。注意,这个函数之前的版本只是返回一个 pid,而这里返回的是{ok, Pid}。

接下来的函数如下所示:

```
%% 同步调用
order_cat(Pid, Name, Color, Description) ->
    gen_server:call(Pid, {order, Name, Color, Description}).

%% 异步调用
return_cat(Pid, Cat = #cat{}) ->
    gen_server:cast(Pid, {return, Cat}).

%% 同步调用
close_shop(Pid) ->
    gen_server:call(Pid, terminate).
```

这几个函数都和 my_server 中的等价。注意,可以给 gen_server:call 传递第 3 个参数,用于设定一个超时时间,单位是毫秒。如果没有提供超时时间(或者原子 infinity),默认会被设置为 5 s。如果超时时间结束,还没有收到任何回应,调用会崩溃。5 s 的默认值完全是随意设置的,很多 Erlang 老手会告诉你应该把默认值设置为 infinity。从我自己的经验来看,我通常会希望在 5 s 之内收到回应,有了这个强迫崩溃的超时设置,往往会帮助我诊断更为严重的问题。

现在,可以增加 gen_server 回调函数了。表 14-1 展示了调用和回调之间的关系。

表 14-1　调用和回调函数之间的关系

gen_server	YourModule
start/3-4	init/1
start_link/3-4	init/1
call/2-3	handle_call/3
cast/2	handle_cast/2

还有其他几个回调函数如 handle_info/2、terminate/2 和 code_change/3，这些回调函数主要处理一些特殊情况。

我们先来更改原先编写过的函数，使之满足 gen_server 模型：init/1、handle_call/3 和 handle_cast/2。

```
%%% 服务器函数
init([]) -> {ok, []}. %% 此处无需做信息处理

handle_call({order, Name, Color, Description}, _From, Cats) ->
    if Cats =:= [] ->
        {reply, make_cat(Name, Color, Description), Cats};
       Cats =/= [] ->
        {reply, hd(Cats), tl(Cats)}
    end;
handle_call(terminate, _From, Cats) ->
    {stop, normal, ok, Cats}.

handle_cast({return, Cat = #cat{}}, Cats) ->
    {noreply, [Cat|Cats]}.
```

同样只需很少的更改。事实上，由于有了更聪明的抽象，因此现在的代码更短了。

现在来实现新的回调函数。第一个是 handle_info/2。由于这只是一个示例模块，并且也没有预定义的日志系统，因此直接把非期望的消息打印出来就行了。

```
handle_info(Msg, Cats) ->
    io:format("Unexpected message: ~p~n",[Msg]),
    {noreply, Cats}.
```

作为经验之谈，请一定要在 handle_cast/2 和 handle_info/2 中对非期望的消息进行日志记录。也许，你还想在 handle_call/3 中记录这种消息，不过通常来讲，对这种调用不做出回应（加上默认的 5 s 超时）就足以达到相同的效果了。

下一个是 terminate/2 回调函数。它和我们之前使用的 terminate/1 非常相似。

```
terminate(normal, Cats) ->
    [io:format("~p was set free.~n",[C#cat.name]) || C <- Cats],
    ok.
```

最后一个回调函数是 code_change/3：

```
code_change(_OldVsn, State, _Extra) ->
```

```
%% 没什么要更改的。这个函数仅仅是行为的需要，并没有被使用。
{ok, State}.
```

别忘了把 make_cat/3 私有函数加进来：

```
%%% 私有函数
make_cat(Name, Col, Desc) ->
    #cat{name=Name, color=Col, description=Desc}.
```

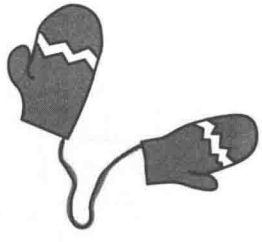
现在可以试试这个全新的代码了：

```
1> c(kitty_gen_server).
{ok,kitty_gen_server}
2> rr(kitty_gen_server).
[cat]
3> {ok, Pid} = kitty_gen_server:start_link().
{ok,<0.253.0>}
4> Pid ! <<"Test handle_info">>.
Unexpected message: <<"Test handle_info">>
<<"Test handle_info">>
5> Cat = kitty_gen_server:order_cat(Pid, "Cat Stevens",
5>       white, "not actually a cat").
#cat{name = "Cat Stevens",color = white,
     description = "not actually a cat"}
6> kitty_gen_server:return_cat(Pid, Cat).
ok
7> kitty_gen_server:order_cat(Pid, "Kitten Mittens",
7>   black, "look at them little paws!").
#cat{name = "Cat Stevens",color = white,
     description = "not actually a cat"}
```

由于我们把 Cat 返还给了服务器，所以当我们再次购买时，服务器就直接把它返回给我们，并没有产生任何新的。如果再试一次，应该可以得到我们想要的：

```
8> kitty_gen_server:order_cat(Pid, "Kitten Mittens",
8>   black, "look at them little paws!").
#cat{name = "Kitten Mittens",color = black,
     description = "look at them little paws!"}
9> kitty_gen_server:return_cat(Pid, Cat).
ok
10> kitty_gen_server:close_shop(Pid).
"Cat Stevens" was set free.
ok
```

太好了，一切正常！



对于这次通用化旅程的冒险，我们能说些什么呢？大概还是之前那些通用的道理：把通用部分和专用部分分离从任何方面来说都是个不错的主意。维护会变得更简单。复杂性也得到消减。代码更安全、更易于测试，并且 bug 更少。即使存在 bug，也很容易修正。

虽然通用服务器只是众多可用抽象中的一种，但是它们是最经常使用的一种。在随后的几章中，我们会介绍更多的抽象和行为。

第 15 章

令人愤怒的有限状态机

有限状态机是工业界使用的众多重要协议实现的核心部分。使用有限状态机，程序员能够以一种易于理解的方式表达复杂的流程和事件序列。

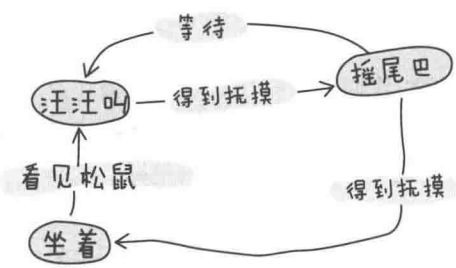
大多数喜欢数学的读者可能知道有限状态机有着极为严格的数学定义，不过 Erlang 中使用的有限状态机并不是这种定义的直接实现，只是受了它的启发。一个典型的 Erlang 有限状态机会被实现为一个进程，运行一组给定的函数（状态），接收一些消息（事件）以驱动状态迁移。

有限状态机广泛应用于电信领域，以至于 OTP 工程师专门为它们编写了一个行为——`gen_fsm`。

本章会介绍 Erlang 世界中有限状态机的概念以及 OTP 中的对应物。作为实践，我们会设计一个全异步的、基于消息的协议，可以用在一个客户到客户的交易系统中，这个交易系统可以作为一款视频游戏的组成部分。

15.1 什么是有限状态机

有限状态机（finite-state machine，FSM）并不是一台真正的机器，不过它确实只有有限个状态。我发现，使用图示可以让 FSM 更容易理解一些。例如，下面是一幅简单的图示，把一条（很蠢的）狗表示成状态机。



这条狗有 3 个状态：坐着（sits）、汪汪叫（barks）和摇尾巴（wag tail）。不同的事件或输入

会驱使狗改变状态。如果狗很平静地坐着，看到了一只松鼠，它就开始汪汪叫，直到你去抚摸它为止。不过，如果狗坐着时，你去抚摸它，那就不知道它会做出什么动作了。在 Erlang 的世界中，狗会死掉（并且最终会被它的监督者重启）。在现实世界中，让狗死而复生非常罕见（并且有点反常），意味着狗被汽车碾压后还能活过来，不过，如果能这样也不算坏事。

作为对比，下面是一幅关于猫的状态图：

这只猫只有一个状态，任何事件都不能改变它。用 Erlang 来实现猫的状态机既有趣又简单：

```
-module(cat_fsm).
-export([start/0, event/2]).

start() ->
    spawn(fun() -> dont_give_crap() end).

event(Pid, Event) ->
    Ref = make_ref(), % 这里不使用监控器
    Pid ! {self(), Ref, Event},
    receive
        {Ref, Msg} -> {ok, Msg}
    after 5000 ->
        {error, timeout}
    end.

dont_give_crap() ->
    receive
        {Pid, Ref, _Msg} -> Pid ! {Ref, meh};
        _ -> ok
    end,
    io:format("Switching to 'dont_give_crap' state~n"),
    dont_give_crap().
```

我们可以运行一下这个模块，验证一下这只猫的对人不理不睬：

```
1> c(cat_fsm).
{ok,cat_fsm}
2> Cat = cat_fsm:start().
<0.67.0>
3> cat_fsm:event(Cat, pet).
Switching to 'dont_give_crap' state
{ok,meh}
4> cat_fsm:event(Cat, love).
Switching to 'dont_give_crap' state
{ok,meh}
5> cat_fsm:event(Cat, cherish).
```

```
Switching to 'dont_give_crap' state
{ok,meh}
```

可以用同样的方式编写狗的 FSM，不过状态要多一些：

```
-module(dog_fsm).
-export([start/0, squirrel/1, pet/1]).

start() -> spawn(fun() -> bark() end).

squirrel(Pid) -> Pid ! squirrel.

pet(Pid) -> Pid ! pet.

bark() ->
    io:format("Dog says: BARK! BARK!~n"),
    receive
        pet ->
            wag_tail();
        _ ->
            io:format("Dog is confused~n"),
            bark()
    after 2000 ->
        bark()
    end.

wag_tail() ->
    io:format("Dog wags its tail~n"),
    receive
        pet ->
            sit();
        _ ->
            io:format("Dog is confused~n"),
            wag_tail()
    after 30000 ->
        bark()
    end.

sit() ->
    io:format("Dog is sitting. Gooooood boy!~n"),
    receive
        squirrel ->
            bark();
        _ ->
            io:format("Dog is confused~n"),
            sit()
    end.
```

只要简单地对照图示中的状态和迁移就可以完成代码的编写。下面是这个 FSM 的运行情况：

```
6> c(dog_fsm).
{ok,dog_fsm}
7> Pid = dog_fsm:start().
Dog says: BARK! BARK!
<0.46.0>
Dog says: BARK! BARK!
Dog says: BARK! BARK!
Dog says: BARK! BARK!
8> dog_fsm:pet(Pid).
pet
Dog wags its tail
9> dog_fsm:pet(Pid).
Dog is sitting. Goooooood boy!
pet
10> dog_fsm:pet(Pid).
Dog is confused
pet
Dog is sitting. Goooooood boy!
11> dog_fsm:squirrel(Pid).
Dog says: BARK! BARK!
squirrel
Dog says: BARK! BARK!
12> dog_fsm:pet(Pid).
Dog wags its tail
pet
❶ 13> %% wait 30 seconds
Dog says: BARK! BARK!
Dog says: BARK! BARK!
Dog says: BARK! BARK!
13> dog_fsm:pet(Pid).
Dog wags its tail
pet
14> dog_fsm:pet(Pid).
Dog is sitting. Goooooood boy!
pet
```

如果愿意的话，你可以遵照状态图来运行 FSM（我通常会这么做，因为这样做可以确保没有错误）。注意，在❶处所输入的命令是一个供读者参考的注释，Erlang shell 接受这种输入。

上面介绍的就是用 Erlang 进程实现 FSM 的核心方法。代码中有些地方还可以稍做变化。我们可以像在服务器的主循环中做的那样，把进程状态数据作为参数放到状态函数中。还可以增加 `init` 和 `terminate` 函数，处理代码升级等。

猫 FSM 和狗 FSM 之间有一个不同之处，猫状态机的事件是同步的，而狗的是异步的。在真正的 FSM 中，会混合使用这两种方式，不过，完全是出于懒惰，我采用了最简单的形式。

还有另外一种事件在上述例子中没有展示：全局事件可以发生在任何状态中。狗嗅到食物的味道就是这种事件的案例之一。一旦触发了“嗅到食物”这个事件，不管狗当前处于什么状态，它都会循着气味寻找食物。

我们不在“玩具”FSM 的编写上多花时间了。接下来，我们直接去学习 gen_fsm 行为。

15.2　通用有限状态机

gen_fsm 行为和 gen_server 有点类似，因为 gen_fsm 是 gen_server 行为的一个专用版本。它们之间最大的区别在于，gen_fsm 中不再处理调用（call）消息和投掷（cast）消息，而是处理同步和异步事件。与前面狗和猫的例子类似，每个状态都用一个函数来表示。下面，我们来介绍一下模块正常工作所需要实现的回调函数。

15.2.1　init 函数

FSM 中的 init 函数和通用服务器中使用的 init/1 完全一样，除了返回值多一些，可接受的返回值为{ok, StateName, Data}、{ok, StateName,Data, Timeout}、{ok, StateName, Data, hibernate}以及{stop, Reason}。stop 元组的工作原理和 gen_server 中的完全一样，hibernate 和 Timeout 的语义也保持不变。

StateName 是一个新出现的变量。StateName 是原子类型，表示下一个被调用的回调函数。对于狗的例子来说，就是 bark 状态。

15.2.2　StateName 函数

函数 StateName/2 和 StateName/3 是占位名字，由你来决定它们的内容。假设 init/1 函数返回元组{ok, sitting, dog}，这意味着 FSM 会处于 sitting 状态。这个状态和我们在 gen_server 中看到的状态不同；更像是狗在 FSM 中的 sit、bark 和 wag_tail 状态。这些状态设定了事件处理的上下文。

例如，考虑一下有人给你打电话的场景。如果你的状态是“周六早上酣睡中”，你的反应会是对着电话大喊大叫；如果你的状态是“等待工作面试中”，你会拿起电话，礼貌地应答；但是，如果你的状态是“已死亡”，那么我很好奇你竟然还能读到这里。

在上面的 FSM 中，init/1 函数的返回值表明我们应该处于 sitting 状态。当 gen_fsm 进程收到一个事件时，函数 sitting/2 或者 sitting/3 会被调用。对于异步事件，会调用 sitting/2 函数，对于同步事件，会调用 sitting/3 函数。

函数 sitting/2（或者更一般地说，StateName/2）有两个参数：一个是 Event，作为事件发送来的实际消息；一个是 StateData，调用携带的数据。sitting/2 函数可以返回以下几种元组：{next_state,NextStateName, NewStateData}、{next_state, NextStateName, NewStateData, Timeout}、{next_state, NextStateName, hibernate}以及{stop, Reason, NewStateData}。

函数 sitting/3 的参数与此类似，只是在 Event 和 StateData 之间多了一个 From 参数。From 参数和 gen_fsm:reply/2 的用法与 gen_server 中的完全一样。函数 StateName/3 可以返回如下元组：

```
{reply, Reply, NextStateName, NewStateData}
```

```
{reply, Reply, NextStateName, NewStateData, Timeout}
{reply, Reply, NextStateName, NewStateData, hibernate}

{next_state, NextStateName, NewStateData}
{next_state, NextStateName, NewStateData, Timeout}
{next_state, NextStateName, NewStateData, hibernate}

{stop, Reason, Reply, NewStateData}
{stop, Reason, NewStateData}
```

注意，这些函数的数量不受限制，只要被导出就行。元组中的原子 NextStateName 决定了下一次会调用哪个函数。

15.2.3 handle_event 函数

之前曾经提到过，无论当前在哪个状态中，全局事件都会触发一个特定反应（狗闻到食物的味道时，会丢下手头正做的任何事情去寻找食物）。由于这类事件在每个状态中都会以同样的方式处理，因此 handle_event/3 回调函数正好满足需要。这个函数的参数和 StateName/2 类似，不过它在中间多了一个参数 StateName（handle_event(Event, StateName, Data)），这个参数表明了收到事件时所处的状态。它的返回值和函数 StateName/2 一样。

15.2.4 handle_syn_event 函数

回调函数 handle_sync_event/4 和 StateName/3 的关系与 handle_event/2 和 StateName/2 的关系一样。这个函数处理同步全局事件，参数和所返回的元组种类都和 StateName/3 一样。

现在，正好可以来解释一下我们是如何知道一个事件是全局的还是针对某个特定状态的。要知道这一点，可以去看看向 FSM 发送事件所使用的函数。被 StateName/2 函数处理的异步事件是用函数 gen_fsm: send_event/2 发送的，而被 StateName/3 函数处理的同步事件是用函数 gen_fsm:sync_send_event/2-3 发送的（第三个可选的参数是超时）。

两个对等的用来发送全局事件的函数为 gen_fsm:send_all_state_event/2 和 gen_fsm: sync_send_all_state_event/2-3（名字很长）。

15.2.5 code_change 和 terminate 函数

code_change 函数的工作方式和 gen_server 中的完全一样，只是多出一个额外的状态参数，如 code_change(OldVersion, StateName, Data, Extra)，并且所返回的元组格式为{ok, NextStateName, NewStateData}。

同样地，terminate 的行为和通用服务器中的也类似，terminate(Reason, StateName, Data)函数做的工作应该和 init/1 相反。

15.3 交易系统规格说明

是时候来实践一下上面学习的 FSM 知识了。很多 Erlang 教程在介绍 FSM 时，都会使用像电话交换机之类的例子。根据我的猜测，大部分程序员基本上都不会为了学习状态机而去了解电话交换机。因此，我们会选择一个更加适合大多数开发者的例子：为一个假想的视频游戏设计和实

现一个物品交易系统。

我所选择的设计具有一定的挑战性。我们并没有用一个中心代理来路由玩家的物品和确认信息（说实话，这种设计比较简单），而是要实现一个服务器，让两个玩家可以直接相互通信（具有易于分布的优点）。

15.3.1　操作定义

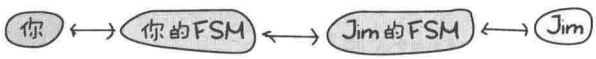
首先，我们要定义出在交易时玩家可以进行的操作。第一个操作是请求建立一个交易。另外一个用户也应该能够接受这个交易请求。我们不给玩家拒绝交易的权利，因为这样可以简单一些。以后增加这个特性也不困难。

交易建立成功后，用户应该能够相互协商。这意味着，用户既可以提供交易物品，也可以撤销这些物品。当交易双方都对报价满意时，可以宣布自己做好了确定交易的准备。此时，交易双方都会保存交易数据。在任何时间点上，玩家都可以取消整个交易。有些“穷”玩家可能只提供一些对方（这个玩家可能很忙）看来肯定没有价值的物品，因此应该可以通过取消交易来还击这类玩家。

简而言之，有如下可用的操作：

- 请求一个交易；
- 接受一个交易；
- 提供交易物品；
- 撤销交易物品；
- 宣布自己就绪；
- 强行取消交易。

每当执行其中的某个操作时，对方玩家的 FSM 应该可以感知到，这是合理的。因为在你玩游戏时，如果 Jim 让他的 FSM 发送了一个物品给你，你的 FSM 必须要知道这件事。这意味着，玩家双方都可以和自己的 FSM 交互，并且自己的 FSM 会和对方的 FSM 交互。具体如下图所示。

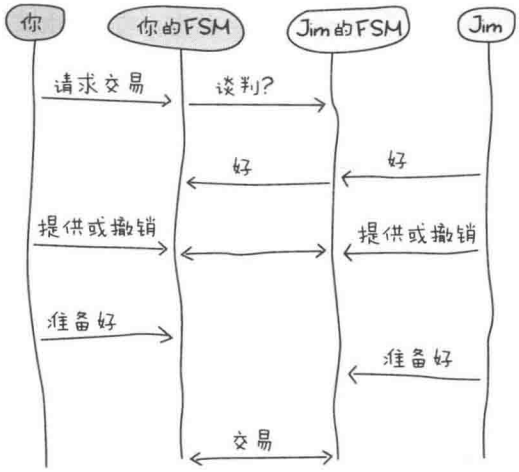


当有两个对等的进程相互通信时，首先要注意的一点就是尽量避免同步调用。如果 Jim 的 FSM 给你的 FSM 发送了一条消息，然后等待这条消息的回应，与此同时，你的 FSM 也给 Jim 的 FSM 发送了一条消息，也在等待所需的回应，那么你和 Jim 都会因为互相等待而无法回应。导致双方 FSM 都被冻结。*死锁*就出现了。

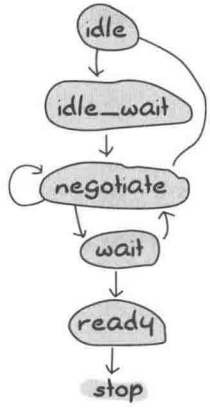
一种解决方案是等待超时后继续往下执行，不过这样的话，双方进程的邮箱中就会有一些残留的消息，导致协议混乱。这显然是一个麻烦的问题，因此我们希望避免它。

处理这个问题最简单的方法是采用全异步的方式。注意，Jim 仍旧对自己的 FSM 发起一个同步调用，这个调用并没有风险，因为 FSM 不需要向 Jim 发起调用，因此不会在它们之间发生死锁。

当两个 FSM 互相通信时，整个消息交换过程如下图所示：



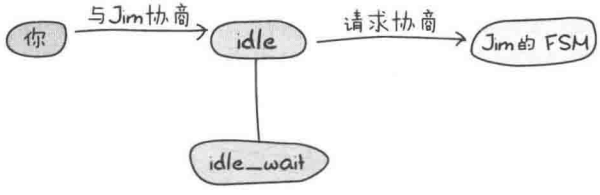
两个 FSM 都处于空闲状态。当你向 Jim 请求交易时，Jim 首先要接受，然后才能向下进行。接下来，你和 Jim 可以提供物品或者撤销所提供的物品。当你们双方都宣布自己就绪时，交易就发生了。上图所示只是所有可能情况的一个简化版本。在实现这个交易系统的过程中，我们会考虑所有可能的情况。



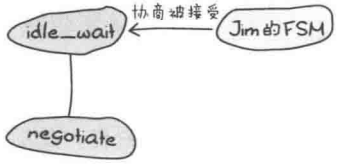
15.3.2 定义状态图和状态迁移

现在我们要完成最困难的任务：定义状态图以及状态迁移。这项工作通常需要一点思考，因为需要考虑可能会出错的所有细枝末节（有时，即使已经对定义审查了多次，有些地方还是会出现问题）。我决定实现的状态图如右图所示。

一开始，两个 FSM 都处于 `idle` 状态。此时我们可以做的就是请求和另外一个玩家进行交易协商。



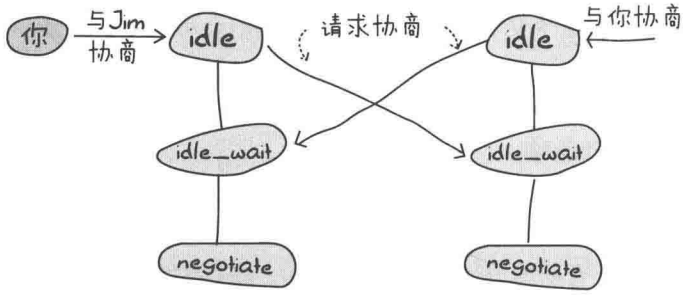
在 FSM 发送完这条消息后，我们进入 `idle_wait` 状态以等待结果回应。一旦对方 FSM 发送了回应，我们的 FSM 就切换到 `negotiate` 状态：



此后，对方玩家 FSM 也应该处于 negotiate 状态。显然，如果我们可以邀请另一个玩家，那么另外一个玩家也可以邀请我们。如果一切正常，结果状态图应该如下所示：

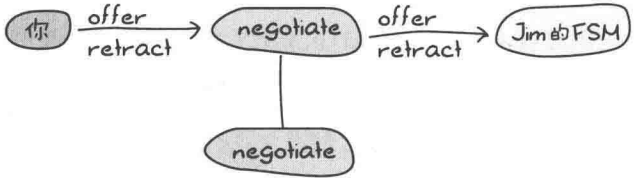
这幅图正好是前面两幅图合在一起再反过来的样子。注意，在本例中我们希望玩家接受交易请求。

如果碰巧在我们请求和另外一个玩家交易的同时，他也向我们发起了交易请求，出现这种情况该怎么办呢？



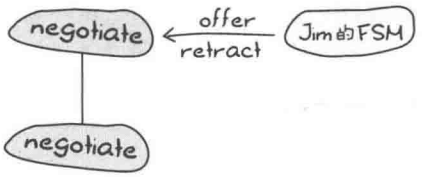
在这种情况下，交易双方都向自己的 FSM 发起和对方进行协商的请求。协商请求消息一发出，双方的 FSM 就立即切换到 idle_wait 状态。接下来，它们就可以处理协商问题了。仔细审查一下前面的状态图，会发现这种巧合出现的情况只有一种可能，那就是在 idle_wait 状态中收到协商请求消息。因此，我们知道，在 idle_wait 状态中收到这个消息意味着出现了竞争条件，可以认为交易双方都想和对方进行协商。可以把交易双方都切换到 negotiate 状态。

现在，可以进行协商了。很好！根据前面列出的操作列表，必须要支持用户提供交易物品 (offer)，还要可以撤销交易物品 (retract)：



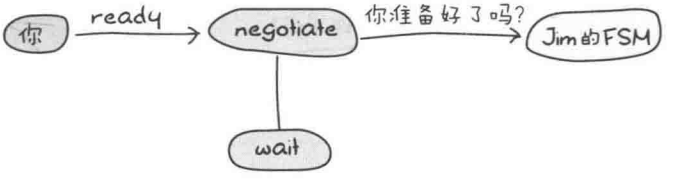
所要做的就是把用户的消息转发给另外一个 FSM。双方的 FSM 都要保存任一方已经提供的交易物品列表，当收到这种消息时，可以更新这个列表。此后，会一直处于 negotiate 状态，

也许另外一个玩家也想提供交易物品：

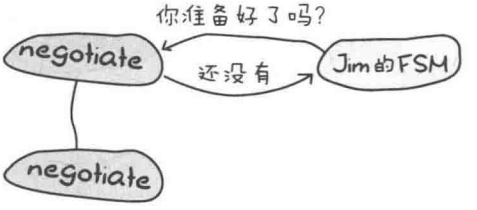


此时，FSM 的行为大体上一样，都一直处于 negotiate 状态。这是一种正常情况。

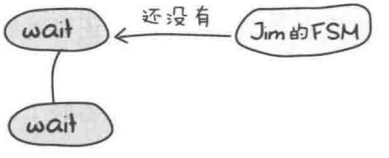
一旦我们觉得自己已经够大方的了，不想再提供交易物品，那么就需要宣布自己准备正式化这个交易。因为必须要同步玩家双方，所以需要一个中间状态，就像 idle 和 idle_wait 状态那样：



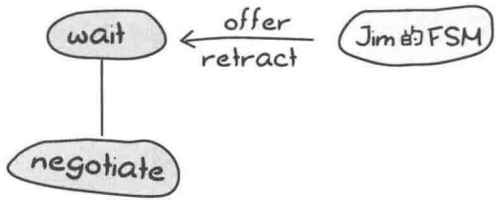
一旦就绪，我们的 FSM 就会向 Jim 的 FSM 询问 Jim 是否就绪。在等待回应期间，我们的 FSM 会进入 wait 状态。我们得到的回应会依赖于 Jim 的 FSM 状态。如果它处在 wait 状态，它会告诉我们它准备好了。否则，它会告诉我们它还没有准备好。在 Jim 询问我们是否准备好时，如果我们处在 negotiate 状态，那么我们的 FSM 也会自动地给 Jim 发送这个回应：



我们的 FSM 会保持在 negotiate 状态，直到我们告诉它已经准备好了。假设我们现在宣布就绪，那就进入 wait 状态。然而，Jim 还没有准备好。这意味着，当我们宣布自己就绪时，要去问问 Jim 是否也已经准备好，而他的 FSM 会回答“还没有”：



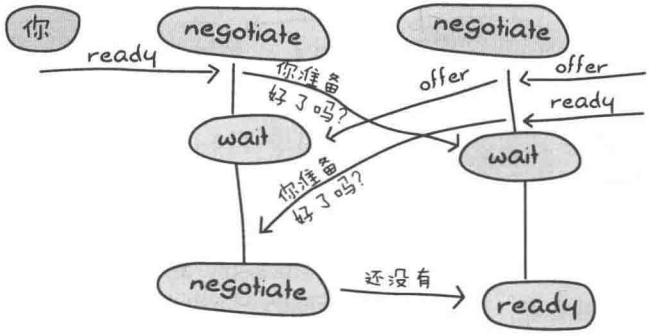
他还没有准备好，但是我们已经准备好了。我们什么也不能做，只能一直等待。在等待 Jim 期间（他仍处在协商中），Jim 可能会向我们发送更多的物品，也可能会撤销一些之前提供的物品：



我们当然不希望 Jim 此时把自己所有的物品都删除，然后单击“我准备好了”，在交易过程中欺骗我们。他一更改所提供的物品，我们就会立即回到 `negotiate` 状态，这样我们也可以修改自己提供的物品，或者检查一下当前的物品情况再决定是否就绪。如此往复。

在某个时刻，Jim 也准备确定这个交易。此时，他的 FSM 会询问我们的 FSM 我们是否就绪：

我们的 FSM 会回复说我们已经就绪。不过，我们会保持在 `wait` 状态，不会迁移到 `ready` 状态。为什么呢？因为这里有一个潜在的条件竞争！设想一下如果我们不这样做，出现如下事件序列时会有什么样的后果：

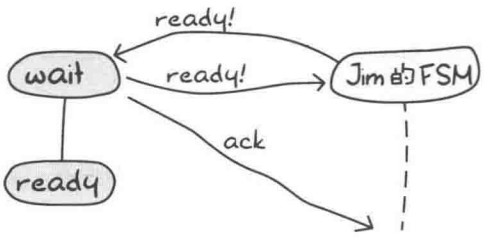


根据消息接收的方式，在我们宣布自己就绪之后，还可以处理来自 Jim 的物品提供消息，即使此时 Jim 也宣布自己准备好了。这意味着，我们一接收到 Jim 提供的物品消息，就立即切回到 `negotiate` 状态。在这个过程中，Jim 会回应我们他准备好了。如果此时 Jim 直接把状态迁移到 `ready`（如上图所示），那么他就会永远处于等待状态，而我们也不知道该做些什么。这种情况反过来也会发生在我们身上。

可以通过增加一个间接层次来解决这个问题（谢谢 David Wheeler）。这就是我们为何要保持 `wait` 状态，并发送“ready!”的原因（见前面的状态图）。

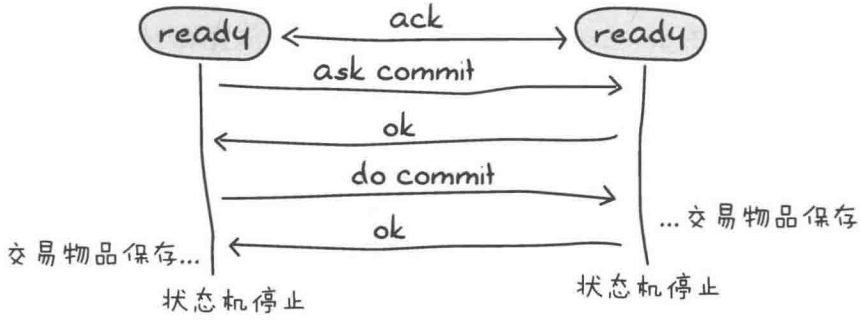
> 注意 David Wheeler，计算机科学家（http://en.wikipedia.org/wiki/David_Wheeler_(computer_scientist)），他有一句话经常被引用，“计算机科学中的所有问题都可以通过增加一个间接层次得以解决……除了过多间接层次本身所导致的问题。”

下图展示了“ready!”消息的处理方式，图中假设 FSM 处于 `wait` 状态，因为之前我们告诉过 FSM 我们已经准备好了。



当我们收到来自其他 FSM 的“ready!”消息时，也会回送一个“ready!”消息。这是为了避免出现前面提到过的双重竞争条件。这会导致某一方的 FSM 多出一条“ready!”消息，不过在本例中，我们忽略它即可。接下来，在迁移到 `ready` 状态之前，我们会发送一条“ack”消息（Jim 的 FSM 会做同样的事情）。之所以发送“ack”消息，是由于同步交易双方时某些实现细节的需要，在本章的后面会对此进行介绍。哎呀——终于到了玩家同步这个环节了。

现在来介绍 `ready` 状态。这个状态有点特殊。两个玩家都已经就绪，且所有控制权基本上也都交给了 FSM。基于此，我们可以实现一个两阶段提交协议的简陋版本来保证在正式交易时一切正常。



我们的版本（如上图所示）非常简单。相比理解 FSM 需要的代码量来说，编写一个完全正确的两阶段提交协议所需要的代码量要多得多。（关于两阶段提交协议的更多信息，请参见 http://en.wikipedia.org/wiki/ Two_phase_commit。）

最后，还要允许在任意时刻取消交易。这意味着——不管当前处于什么状态——都要接收来自对方的“cancel”消息，并退出交易。礼貌起见，在离开前，应该通知一下对方。

到目前为止，我们已经介绍了不少内容。要完全理解这些概念，需要花点时间，所以如果一时不能理解，也不必担心。有不少人对我的协议进行了检查，看看它是否正确，即便如此，当初

还是遗漏了几个竞争条件，直到后面审查代码时才发现。多读几遍代码是一件很正常的事情，当你对异步协议不熟悉时，更是如此。如果你属于这种情况，那么我强烈建议你去设计自己的协议。然后问自己以下问题。

- 如果协议双方非常快速地做同样的操作会出现什么情况呢？
- 如果它们非常快地连发两条消息时该如何处理呢？
- 在迁移状态时，如何处理那些没来得及处理的消息？

你会发现，复杂性会迅速增长。你也许会得到一个和我类似的解决方案，或者更好（如果是这样，一定要通知我！）。不管结果如何，这都是一个非常有趣的问题，并且 FSM 本身也不会变得复杂。

如果你已经理解了所有这些内容（没理解也行，也许你就是一个喜欢对着干的读者），就可以进入下一节了。在下一节中，我们会实现这个游戏系统。现在，如果愿意的话，可以去喝一杯美味的咖啡，休息一会儿。

15.4　游戏交易

现在我们来用 OTP 的 gen_fsm 实现交易系统的协议。首先要做的是创建接口。

15.4.1　公共接口

我们的模块有 3 个调用者：玩家、gen_fsm 行为以及其他玩家的 FSM。不过，我们只需要导出玩家使用的函数和 gen_fsm 行为回调函数。因为其他的 FSM 也运行在 trade_fsm 模块中，所以可以从内部访问需要的函数。

```
-module(trade_fsm).
-behavior(gen_fsm).

%% 公共 API
-export([start/1, start_link/1, trade/2, accept_trade/1,
         make_offer/2, retract_offer/2, ready/1, cancel/1]).

%% gen_fsm 回调函数
-export([init/1, handle_event/3, handle_sync_event/4, handle_info/3,
         terminate/3, code_change/4,
         % 定制的状态名
         idle/2, idle/3, idle_wait/2, idle_wait/3, negotiate/2,
         negotiate/3, wait/2, ready/2, ready/3]).
```

这就是我们的 API 定义。可以看出，其中有些函数既有同步版本，也有异步版本（如 idle/2 和 idle/3）。主要是因为，在有些情况下，我们希望玩家同步地调用 FSM，但是又希望其他 FSM 是异步调用的。让玩家同步调用可以极大地简化 FSM 的逻辑，因为这限制了可以连续发送的冲突消息的数量。在后面增加 gen_fsm 回调函数时，会介绍这一部分内容。现在，我们先来根据前面的协议定义实现公共 API。

```
%%% 公共 API
start(Name) ->
    gen_fsm:start(?MODULE, [Name], []).

start_link(Name) ->
    gen_fsm:start_link(?MODULE, [Name], []).

%% 请求开始交易会话。当/如果对方接受时返回。
trade(OwnPid, OtherPid) ->
    gen_fsm:sync_send_event(OwnPid, {negotiate, OtherPid}, 30000).

%% 接受某个玩家的交易请求
accept_trade(OwnPid) ->
    gen_fsm:sync_send_event(OwnPid, accept_negotiate).

%% 从物品表中选择一个物品进行交易
make_offer(OwnPid, Item) ->
    gen_fsm:send_event(OwnPid, {make_offer, Item}).

%% 撤销某个交易物品。
retract_offer(OwnPid, Item) ->
    gen_fsm:send_event(OwnPid, {retract_offer, Item}).

%% 宣布自己就绪。当对方也宣布自己就绪时，交易就完成了。
ready(OwnPid) ->
    gen_fsm:sync_send_event(OwnPid, ready, infinity).

%% 取消交易。
cancel(OwnPid) ->
    gen_fsm:sync_send_all_state_event(OwnPid, cancel).
```

这段代码非常规范，在本章前面已经对这些 gen_fsm 函数（除了 start/3-4 和 start_link/3-4，我相信你能自己弄明白这两个函数的意思）做过介绍。

15.4.2 FSM 到 FSM 的函数

下面，我们来实现 FSM 到 FSM 的函数。首先来实现完成交易初始化工作的函数，当我们想邀请其他玩家和我们交易时，可以使用这些函数。

```
%% 向另外一个 FSM 发起交易会话请求
ask_negotiate(OtherPid, OwnPid) ->
    gen_fsm:send_event(OtherPid, {ask_negotiate, OwnPid}).

%% 转发玩家的交易接受消息
```

```
accept_negotiate(OtherPid, OwnPid) ->
    gen_fsm:send_event(OtherPid, {accept_negotiate, OwnPid}).
```

第一个函数向对方 pid 发起交易会话请求，第二个函数用于回应交易请求（毫无疑问，它们是异步的）。

接下来可以编写提供和撤销物品的函数。根据协议，这些函数实现如下：

```
%% 转发玩家的交易物品提供消息
do_offer(OtherPid, Item) ->
    gen_fsm:send_event(OtherPid, {do_offer, Item}).

%% 转发玩家的交易物品撤销消息
undo_offer(OtherPid, Item) ->
    gen_fsm:send_event(OtherPid, {undo_offer, Item}).
```

下一个调用和交易是否准备就绪有关。同样地，基于协议，共有 3 条消息。第一条是 `are_you_ready`，这条消息会有两种回应消息：`not_yet` 或者`'ready!'`。

```
%% 询问对方是否就绪
are_you_ready(OtherPid) ->
    gen_fsm:send_event(OtherPid, are_you_ready).

%% 回复未就绪
%% 也就是说，不在'wait'状态
not_yet(OtherPid) ->
    gen_fsm:send_event(OtherPid, not_yet).

%% 通知对方玩家当前处于等待进入 ready 状态。
%% 状态会迁移到'ready'
am_ready(OtherPid) ->
    gen_fsm:send_event(OtherPid, 'ready!').
```

还有一些其他函数，供交易双方 FSM 在 `ready` 状态提交事务时使用。它们的准确用法，本章后面会详细介绍，不过结合函数的名字以及前面的事件序列/状态图应该可以明白它们的用途，你也可以据此写出自己的 `trade_fsm` 实现。

```
%% 确认 fsm 处于 ready 状态
ack_trans(OtherPid) ->
    gen_fsm:send_event(OtherPid, ack).

%% 询问是否可以提交
ask_commit(OtherPid) ->
    gen_fsm:sync_send_event(OtherPid, ask_commit).

%% 开始同步提交
do_commit(OtherPid) ->
    gen_fsm:sync_send_event(OtherPid, do_commit).
```

哦，还有一个礼貌性的函数，用于在取消交易时通知对方的 FSM：

```
notify_cancel(OtherPid) ->
```

```
    gen_fsm:send_all_state_event(OtherPid, cancel).
```

15.4.3 gen_fsm 回调函数

现在，可以进入真正有趣的部分：gen_fsm 回调函数。第一个回调函数是 init/1。在本例中，希望每个 FSM 都持有它所代表的玩家的名字（这样的话，输出可以漂亮些），这个名字会保存在状态数据中，状态数据会作为每个回调函数的最后一个参数传递。还有其他想保存在内存中的东西吗？在本例中，我们希望保存对方玩家（Jim）的 FSM 进程的 pid、我们提供的交易物品、对方玩家 FSM 提供的交易物品。我们还要增加一个监控器的引用（这样如果对方死了，我们可以退出）以及一个 from 字段，用来实现延迟回复。

```
-record(state, {name="",
                other,
                ownitems=[],
                otheritems=[],
                monitor,
                from}).
```

init/1 函数只需要玩家名字。注意，起始状态是 idle。

```
init(Name) ->
    {ok, idle, #state{name=Name}}.
```

接下来要考虑的回调函数是状态函数。到目前为止，我们介绍了状态迁移以及可以使用的函数调用，但是还需要一种方法来让我们知道一切运行正常。先来编写几个工具函数。

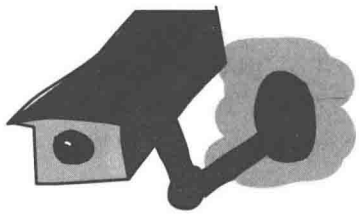
```
%% 给玩家发送一条通知，可以是一条发给玩家进程的消息，
%% 不过在此处，打印到 shell 上就足够了
notice(#state{name=N}, Str, Args) ->
    io:format("~s: "++Str++"~n", [N|Args]).

%% 记录非期望的消息
unexpected(Msg, State) ->
    io:format("~p received unknown event ~p while in state ~p~n",
              [self(), Msg, State]).
```

我们从 idle 状态开始。为了方便起见，首先来实现异步版本的函数。idle 状态的异步回调函数只需关心其他玩家的交易请求即可。这是因为根据 API 函数定义，本方玩家会使用同步版本的调用，所以需要另一个有 3 个参数的回调函数。

```
idle({ask_negotiate, OtherPid}, S=#state{}) ->
    Ref = monitor(process, OtherPid),
    notice(S, "~p asked for a trade negotiation", [OtherPid]),
    {next_state, idle_wait, S#state{other=OtherPid, monitor=Ref}};
idle(Event, Data) ->
    unexpected(Event, idle),
    {next_state, idle, Data}.
```

代码中使用了监控器，这样就可以处理对方进程死亡的情况，将监控器的引用和对方 FSM 的 pid 一起保存在 FSM 的数据中，然后迁移到 idle_wait 状态。注意，我们会报告所有不期望

的消息，忽略它们，并一直保持在原来的状态。由于竞争条件的存在，我们可能会在处理流程中收到一些带外消息。一般来说，忽略它们不会造成什么问题，不过要把它们去掉也不太容易。所以，在收到这些未知但是意料之中的消息时，只要不让整个 FSM 崩溃就可以了。



当本方玩家请求 FSM 联系另一个玩家进行交易时，它会发送一个同步事件。此时，需要 idle/3 回调函数。

```
idle({negotiate, OtherPid}, From, S=#state{}) ->
    ask_negotiate(OtherPid, self()),
    notice(S, "asking user ~p for a trade", [OtherPid]),
    Ref = monitor(process, OtherPid),
    {next_state, idle_wait, S#state{other=OtherPid, monitor=Ref, from=From}};
idle(Event, _From, Data) ->
    unexpected(Event, idle),
    {next_state, idle, Data}.
```

这里的处理方式和异步版本中的类似，只是多了一个询问对方是否愿意和我们进行交易的调用。你应该注意到，我们并没有回应本方玩家。这是因为，现在还不能提供任何有价值的信息，我们希望玩家进程先挂起，等待交易请求被接受后再继续工作。只有在 idle_wait 状态收到对方的接受消息时才会给玩家发送回应。

此时，我们需要处理两种情况：一种是对方玩家接受了我们的邀请，同意协商；另一种是在我们发起交易请求的同时，他也发起了交易请求（竞争条件，协议中描述过）。

```
idle_wait({ask_negotiate, OtherPid}, S=#state{other=OtherPid}) ->
    gen_fsm:reply(S#state.from, ok),
    notice(S, "starting negotiation", []),
    {next_state, negotiate, S};
%% 对方接受了我们的请求，迁移到 negotiate 状态。
idle_wait({accept_negotiate, OtherPid}, S=#state{other=OtherPid}) ->
    gen_fsm:reply(S#state.from, ok),
    notice(S, "starting negotiation", []),
    {next_state, negotiate, S};
idle_wait(Event, Data) ->
    unexpected(Event, idle_wait),
    {next_state, idle_wait, Data}.
```

上面代码中有两种情况都要迁移到 negotiate 状态，不过，别忘了调用 gen_fsm:reply/2 通知玩家一切正常，可以开始交易了。还有一种情况要处理，本方玩家接受来自对方的交易请求。

```
idle_wait(accept_negotiate, _From, S=#state{other=OtherPid}) ->
    accept_negotiate(OtherPid, self()),
    notice(S, "accepting negotiation", []),
    {reply, ok, negotiate, S};
idle_wait(Event, _From, Data) ->
    unexpected(Event, idle_wait),
    {next_state, idle_wait, Data}.
```

这种情况同样也会迁移到 negotiate 状态。negotiate 状态中，需要处理来自本方玩家和对方 FSM 增加、删除交易物品的异步请求。不过，我们还没有想好如何存储这些物品。可能我们比较懒惰，并且交易物品的数量也不会太多，所以简单地使用列表来存储就足够了。不过，我们以后可能需要更改存储的方法，因此把对交易物品的操作封装成单独的函数是个不错的主意。把下列函数和 notice/3 以及 unexpected/2 函数一起放到文件的末尾：

```
%% 向物品列表中增加一件物品
add(Item, Items) ->
    [Item | Items].

%% 从物品列表中移除一件物品
remove(Item, Items) ->
   Items -- [Item].
```

这两个函数很简单，不过它们起到了将操作（增加、删除物品）和操作的实现（使用列表）隔离开的作用。我们可以在不破坏其他代码的情况下，轻易地将列表换成属性列表（proplist）、字典或者其他任何数据结构。

使用这两个函数，就可以实现提供和移除交易物品的功能了：

```
negotiate({make_offer, Item}, S=#state{ownitems=OwnItems}) ->
   do_offer(S#state.other, Item),
   notice(S, "offering ~p", [Item]),
   {next_state, negotiate, S#state{ownitems=add(Item, OwnItems)}};
%% 本方撤销一件交易物品
negotiate({retract_offer, Item}, S=#state{ownitems=OwnItems}) ->
   undo_offer(S#state.other, Item),
   notice(S, "cancelling offer on ~p", [Item]),
   {next_state, negotiate, S#state{ownitems=remove(Item, OwnItems)}};
%% 对方提供一件交易物品
negotiate({do_offer, Item}, S=#state{otheritems=OtherItems}) ->
   notice(S, "other player offering ~p", [Item]),
   {next_state, negotiate, S#state{otheritems=add(Item, OtherItems)}};
%% 对方撤销一件交易物品
negotiate({undo_offer, Item}, S=#state{otheritems=OtherItems}) ->
   notice(S, "Other player cancelling offer on ~p", [Item]),
   {next_state, negotiate, S#state{otheritems=remove(Item, OtherItems)}};
```

这段代码显示出了双方使用异步消息通信时丑陋的一面。一侧使用的消息形式是“make”和“retract”，另一侧则采用“do”和“undo”的形式。这种形式完全是随意选择的，只是为了区分玩家到 FSM 的消息和 FSM 到 FSM 的消息。注意，在处理来自本方玩家的消息时，要把自己做出的改变告诉对方。

还有一项任务是要处理协议中的 are_you_ready 消息。这是在 negotiate 状态中需要处理的最后一个异步事件：

```
negotiate(are_you_ready, S=#state{other=OtherPid}) ->
   io:format("Other user ready to trade.~n"),
   notice(S,
```

```
            "Other user ready to transfer goods:~n"
            "You get ~p, The other side gets ~p",
             [S#state.otheritems, S#state.ownitems]),
    not_yet(OtherPid),
    {next_state, negotiate, S};
negotiate(Event, Data) ->
    unexpected(Event, negotiate),
    {next_state, negotiate, Data}.
```

正如协议中描述的那样，只要不处于 wait 状态，在收到这条消息时，都必须回复 not_yet。我们还把当前的交易明细打印出来，供玩家决策使用。

当玩家做完决策，准备就绪时，会发送 ready 事件。这个事件是同步的，因为我们不希望玩家在宣布他准备好后还不停地增加物品、改动交易。

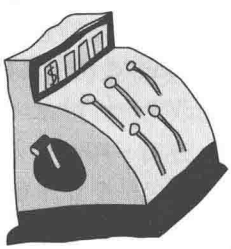
```
negotiate(ready, From, S = #state{other=OtherPid}) ->
    are_you_ready(OtherPid),
    notice(S, "asking if ready, waiting", []),
    {next_state, wait, S#state{from=From}};
negotiate(Event, _From, S) ->
    unexpected(Event, negotiate),
    {next_state, negotiate, S}.
```

此时，要把状态迁移到 wait。注意，这时除了等待对方玩家的回应什么也做不了。我们把 From 变量保存起来，这样在需要时，就可以通过 gen_fsm:reply/2 通知本方玩家了。

wait 状态既有趣又复杂。由于对方玩家可能还没有就绪，他还会提供和撤销交易物品。这种情况下，我们的 FSM 应该能自动地回退到 negotiate 状态。否则，如果对方玩家移除了和我们交易的大宗物件，然后宣布自己就绪，我们的战利品岂不白白损失，那真是太不幸了。所以，返回到 negotiate 状态是一个正确的决策。

```
wait({do_offer, Item}, S=#state{otheritems=OtherItems}) ->
    gen_fsm:reply(S#state.from, offer_changed),
    notice(S, "other side offering ~p", [Item]),
    {next_state, negotiate, S#state{otheritems=add(Item, OtherItems)}};
wait({undo_offer, Item}, S=#state{otheritems=OtherItems}) ->
    gen_fsm:reply(S#state.from, offer_changed),
    notice(S, "Other side cancelling offer of ~p", [Item]),
    {next_state, negotiate, S#state{otheritems=remove(Item, OtherItems)}};
```

这种情况下，我们有必要给玩家回应了，可以使用存储在 S#state.from 中的进程标识给玩家发送回应。



接下来要考虑的消息是关于双方 FSM 同步的，这样它们就都能够迁移到 ready 状态并确认交易了。在本例中，我们着重关注之前定义的协议。

有 3 个消息需要处理：are_you_ready（对方玩家刚刚宣布就绪）、not_yet（我们询问对方玩家是否就绪，他回复没有就绪）以及'ready!'（我们询问对方玩家是否就绪，他回复就绪了）。

先来处理 are_you_ready 消息。在介绍协议时我曾经说过，这里隐藏着一个竞争条件。此

时唯一能做的就是用 am_ready/1 发送'ready!'消息，其余的事情以后再处理。

```
wait(are_you_ready, S=#state{}) ->
    am_ready(S#state.other),
    notice(S, "asked if ready, and I am. Waiting for same reply", []),
    {next_state, wait, S};
```

我们会停在等待状态，因此没必要通知本方玩家。同样地，当收到我们的请求，对方回应了 not_yet 时，也不用通知本方玩家。

```
wait(not_yet, S = #state{}) ->
    notice(S, "Other not ready yet", []),
    {next_state, wait, S};
```

另一方面，如果对方玩家准备好了，我们也会发送一个'ready!'消息给对方 FSM，然后回应本方玩家，最后迁移到 ready 状态。

```
wait('ready!', S=#state{}) ->
    am_ready(S#state.other),
    ack_trans(S#state.other),
    gen_fsm:reply(S#state.from, ok),
    notice(S, "other side is ready. Moving to ready state", []),
    {next_state, ready, S};
%% 不关心这些消息!
wait(Event, Data) ->
    unexpected(Event, wait),
    {next_state, wait, Data}.
```

你也许已经注意到，我们使用了函数 ack_trans/1。事实上，双方 FSM 都要使用它。为什么呢？要理解这一点，需要看看在 ready 状态中都发生了些什么。

处于 ready 状态时，双方玩家的操作都会变得无效（除了取消交易操作）。我们不再关心新的物品提供消息。这为我们的处理带来了一定的自由度。总的来说，双方 FSM 可以自由地交互，无需担心其他的事情。这样我们完全可以实现一个简陋的两阶段提交协议。任何一方玩家都不参与的情况下，要能启动交易的提交，我们需要一个事件去触发 FSM。ack_trans/1 中的 ack 事件就是这个用途。一旦处于 ready 状态，就会处理这条消息并进行相应的动作，事务就开始了。

两阶段提交协议要求通信是同步的。这意味着，无法让双方的 FSM 同时开始事务，因为这会导致死锁。关键在于寻找一种方法，可以选定一个 FSM 发起事务提交，另外一个则不动，等待前一个 FSM 发来的指令。

事实证明，设计 Erlang 的工程师和计算机科学家非常聪明（嗯，这一点我们早就知道了）。任何进程的 pid 都是可以相互比较和排序的。不论进程是何时创建的、现在是死是活，以及是否来自另外一个虚拟机，都可以做这个操作（在第 26 章介绍分布式 Erlang 时会看到更多这方面的内容）。

两个进程的 pid 可以进行比较，并且一个会比另外一个大，基于此我们就可以实现一个函数

priority/2，它接收两个 pid，返回值表明当前进程是否被选中。

```
priority(OwnPid, OtherPid) when OwnPid > OtherPid -> true;
priority(OwnPid, OtherPid) when OwnPid < OtherPid -> false.
```

调用这个函数，就可以让一个进程发起提交，另外一个进程执行这个进程的命令。

下面是在 ready 状态中收到 ack 消息时的代码实现：

```
ready(ack, S=#state{}) ->
    case priority(self(), S#state.other) of
        true ->
            try
                notice(S, "asking for commit", []),
                ready_commit = ask_commit(S#state.other),
                notice(S, "ordering commit", []),
                ok = do_commit(S#state.other),
                notice(S, "committing...", []),
                commit(S),
                {stop, normal, S}
            catch Class:Reason ->
                %% 退出！ ready_commit 或者 do_commit 失败了
                notice(S, "commit failed", []),
                {stop, {Class, Reason}, S}
            end;
        false ->
            {next_state, ready, S}
    end;
ready(Event, Data) ->
    unexpected(Event, ready),
    {next_state, ready, Data}.
```

这个庞大的 try ... catch 表达式，就是主导 FSM 中的事务提交流程。ask_commit/1 和 do_commit/1 都是同步的。因此，主导 FSM 可以随意调用它们。可以看出对方 FSM 只能等待。它会收到来自主导 FSM 进程的命令。它收到的第一条消息应该是 ask_commit。这条消息只是为了保证双方 FSM 都还在那里——没发生什么糟糕的情况，并且它们都想完成任务。

```
ready(ask_commit, _From, S) ->
    notice(S, "replying to ask_commit", []),
    {reply, ready_commit, ready, S};
```

收到这条消息后，主导 FSM 进程会通过 do_commit 要求确认事务。此时必须提交数据。

```
ready(do_commit, _From, S) ->
    notice(S, "committing...", []),
    commit(S),
    {stop, normal, ok, S};
ready(Event, _From, Data) ->
    unexpected(Event, ready),
    {next_state, ready, Data}.
```

一旦完成了这项工作，FSM 就退出了。主导 FSM 收到 ok 回应，就可以开始在自己这一侧

进行提交了。这也解释了为何需要那个庞大的 try ... catch：如果对方 FSM 死亡了或者它的玩家取消了交易，同步调用会由于超时而崩溃。此时提交应该被取消。

请注意，commit 函数的实现如下：

```
commit(S = #state{}) ->
    io:format("Transaction completed for ~s. "
              "Items sent are:~n~p,~n received are:~n~p.~n"
              "This operation should have some atomic save "
              "in a database.~n",
               [S#state.name, S#state.ownitems, S#state.otheritems]).
```

很无趣，对吧？一般来讲，只有两个参与者的情况下是不能完成真正安全的事务提交的，通常会需要一个第三方实体来判断玩家双方是否一切正常。一个真正的提交函数应该代表双方玩家去和这个第三方实体交互，然后要么对数据库进行安全的写入，要么回退整个交易。在此，我们不会涉及这种细节，当前的 commit/1 函数对于本例来说已经足够了。

还没有结束。还有两种类型的事件没有介绍：玩家取消交易以及对方玩家 FSM 进程崩溃。前者可以在回调函数 handle_event/3 和 handle_sync_event/4 中处理。当对方玩家取消交易时，我们会收到一条异步通知消息。

```
%% 对方玩家发送了取消事件
%% 停止正在做的工作，终止进程
handle_event(cancel, _StateName, S=#state{}) ->
    notice(S, "received cancel event", []),
    {stop, other_cancelled, S};
handle_event(Event, StateName, Data) ->
    unexpected(Event, StateName),
    {next_state, StateName, Data}.
```

当我们取消交易时，不要忘记通知对方玩家：

```
%% 本方玩家取消了交易。必须通知对方玩家我们退出了
handle_sync_event(cancel, _From, _StateName, S = #state{}) ->
    notify_cancel(S#state.other),
    notice(S, "cancelling trade, sending cancel event", []),
    {stop, cancelled, ok, S};
%% 注意：不要回复非期待的消息，让调用者崩溃
handle_sync_event(Event, _From, StateName, Data) ->
    unexpected(Event, StateName),
    {next_state, StateName, Data}.
```

最后一个需要处理的是对方 FSM 崩溃事件。还好，我们在 idle 状态函数中设置了一个监控器。可以匹配这条消息，并做出相应的反应：

```
handle_info({'DOWN', Ref, process, Pid, Reason}, _, S=#state{other=Pid, monitor=Ref}) ->
    notice(S, "Other side dead", []),
    {stop, {other_down, Reason}, S};
handle_info(Info, StateName, Data) ->
    unexpected(Info, StateName),
    {next_state, StateName, Data}.
```

注意，即使在提交时收到了 cancel 或者'DOWN'事件，一切也都是安全的，并且玩家仍旧拥有自己的物品。在这个例子中，不存在可以窃取他人物品的漏洞。

> **注意**　对于大部分消息，我们都使用 io:format/2 进行打印，这使得 FSM 的使用者可以看到 FSM 的运行情况。在真实的应用中，你也许想使用更为灵活的方法。一种做法是让使用者传递一个 pid 给 FSM，这个进程可以接收发送给它的通知消息。这个进程可以连接一个 GUI 或者其他任何可以让玩家观察到事件的系统。选择 io:format/2 只是因为简单，使得我们可以把重点放在 FSM 和异步协议上。

现在，只剩下两个回调函数没有介绍了：code_change/4 和 terminate/3。code_change/4 中目前不需要做任何事情。只要把它导出，使得在加载 FSM 的新版本时可以调用它即可。terminate 函数也很简短，因为在本例中并没有涉及真正的资源。

```
code_change(_OldVsn, StateName, Data, _Extra) ->
    {ok, StateName, Data}.

%% 交易结束。
terminate(normal, ready, S=#state{}) ->
    notice(S, "FSM leaving.", []);
    terminate(_Reason, _StateName, _StateData) ->
    ok.
```

哎哟，终于完成了。

现在，可以试一下我们的交易系统了。嗯，做这件事有点讨厌，因为得启动两个进程让它们相互通信。为了解决这个问题，我在文件 trade_calls.erl 中编写了一些测试（可以从 http://learnyousomeerlang.com/static/erlang/trade_calls.erl 获得），使用这些测试，可以实现如下 3 种场景。

- main_ab/0 会运行一个标准的交易流程，并输出所有细节信息。
- main_cd/0 会在中途取消交易。
- main_ef/0 和 main_ab/0 非常相似，只是它包含另外一种竞争条件。

这些测试中，第一个和第三个中的交易会成功，第二个会失败（会打印大量出错消息，不过这些出错消息展示了工作流程）。

15.5　为自己骄傲

本章的内容要比其他章难一些，我必须承认，用这么有难度的例子来讲解通用 FSM 行为确实不太理智。如果你感到困惑，可以想想如下问题。

- 能理解系统是如何根据进程所处的状态来处理不同事件的吗？
- 能理解系统是如何从一个状态迁移到另外一个状态的吗？

- 知道何时使用 send_event/2 和 sync_send_event/2-3 而不是 send_all_state_event/2 和 sync_send_ all_state_event/3 吗？

如果你的回答都是肯定的，那么你已经理解了 gen_fsm。

其余的部分——异步协议、延迟回复以及保存 From 变量、同步调用时给进程排优先级、简陋的两阶段提交等——不是理解 gen_fsm 所必需的内容。它们的存在只是为了表明可以做这些事情，并且凸显编写真正并发软件的困难，即便使用的是像 Erlang 这样的语言。Erlang 并不能取代计划和思考，也不会帮你解决问题。它只是给你提供了工具。

尽管如此，如果你理解了所有这些内容，那么可以为自己感到骄傲（尤其是之前从没编写过并发软件的）。现在，你可以开始真正地并发思考了。

15.6 适用于真实世界吗

一个真实游戏的逻辑要复杂得多，因此交易也要更复杂一些。正在交易的物品可能穿在游戏角色身上，并且被敌人损坏了。在交易时，还可能出现物品被移出和移入清单的情况。玩家在同一台服务器上吗？如果不在，如何同步提交到不同数据库上？

在理想环境中，我们的交易系统是健全的。在把它应用到真实游戏之前（如果你敢的话），要确保一切正常。测试，测试，再次测试。你会发现测试并发和并行的代码非常痛苦。你会掉头发，失去朋友甚至失去部分理智。即便经历了这些，你也必须牢记，系统的健壮性永远是由其最弱的一个环节决定的，因此系统还是可能非常脆弱。

> **警告** 虽然这个交易系统的模型是健全的，但是在代码编写完很长时间之后，甚至代码已经运行了几年，还是会时不时出现一些微妙的并发 bug 和竞争条件。虽然我的代码通常是防弹的（是的），不过有时你还是必须要面对刀和剑。小心那些蛰伏的 bug。

太好了，我们终于可以把所有这些令人抓狂的内容抛到身后了。接下来，我们将学习如何使用 OTP 中的 gen_event 行为来处理各种事件，如警告和日志。

第 16 章

事件处理器

在第 13 章的创建备忘提醒（reminder）应用中，我曾经提到可以通过即时消息、电子邮件或是其他方式去通知客户。在第 15 章中，交易系统通过 `io:format/2` 函数来告知人们当前正在发生的事情。

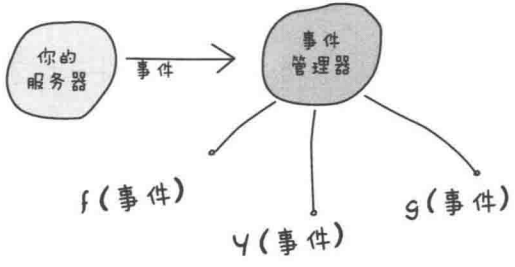
你可能已经看到这些例子之间的相同点：它们都是让人（进程或应用）知道某个事件在某个时间点发生了。有时，我们只需要输出结果；有时，我们则要先得到事件订阅者的 pid，然后向其发送一条消息。

本章会介绍 OTP 事件处理器，它是众多通知处理策略中的一种。下面会先介绍一下事件处理器的相关知识，然后把这些知识付诸实践，实现一个播报体育事件的通知系统。

16.1 处理它！*泵式散弹枪*

我们前面所使用的基于打印的通知方法极其简单，但是难以扩展。而使用订阅者则是一种很好的方法。事实上，对于那种当订阅者收到事件后，相应的操作处理需要执行很长时间的情况，这种方法非常有用。在一些简单的情况下，可能会觉得没有必要为每个回调专门启动等待事件的进程，此时，可以采用第三种方法。

在第三种方法中，我们只启动一个进程。首先，它接受一些函数，在收到事件时执行这些函数。这个进程通常称作事件管理器（event manager），它看上去有点像下图这样：



这种方法的优点如下。

- 如果服务器有很多订阅者，那么它会保持运行状态，不会由于转发事件而阻塞。因为所有的事件，它都只需要转发一次——转给管理器。
- 如果转发事件时需要传送大量数据，那么这些数据的传送也只有一次，并且所有的回调操作使用同一份数据实例。
- 无需为短生命周期的任务启动进程。

当然，这个方法也有一些缺点，具体如下。

- 如果所有的函数都要运行很长时间，那么它们之间会互相阻塞。为了避免阻塞的发生，可以让函数将事件转发给一个进程，也就是把事件管理器当成事件转发器使用（和我们在第 13 章备忘提醒录应用中的做法类似）。
- 如果一个回调函数进入了无限循环，那么除非它崩溃了，否则任何新事件都不会得到处理。

这些问题其实很好解决。在本质上，就是将事件管理器方式转换成订阅者方式。由于事件管理器方法本身比较灵活，所以转换也相对比较容易，在本章中，你会看到如何进行这个转换。

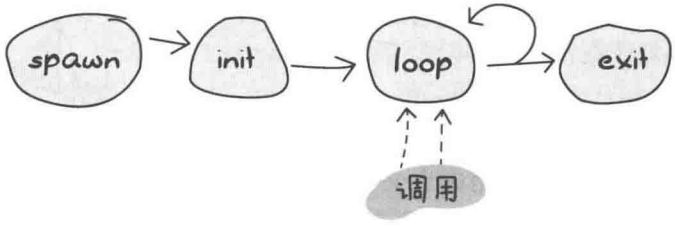
一般我都会先用纯 Erlang 实现一个 OTP 行为的简单版本。不过这次，我会直接使用 OTP 的行为。下面首先介绍 `gen_event`。

16.2 通用事件处理器

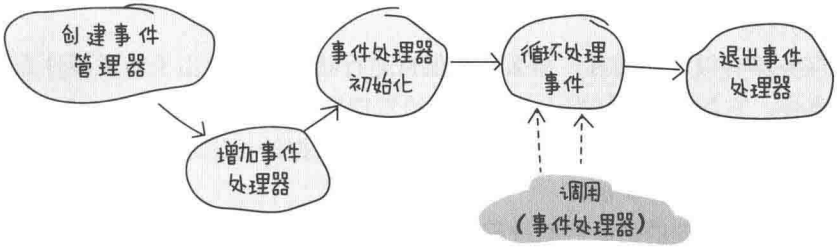
`gen_event` 行为与 `gen_server` 以及 `gen_fsm` 有很大的不同，它根本不需要实际启动一个进程。之所以不需要进程是因为它的工作方式是“接受一组回调函数”。

简单来讲，`gen_event` 行为运行这个接受并调用回调函数的事件管理器进程，而你只需要提供包含这些回调函数的模块即可。这意味着你无需关心事件分派，只需按照事件管理器要求的格式放置回调函数即可。所有的事件管理就自然都有了，你只需提供应用特定的东西。这没啥可惊讶的，因为 OTP 的设计哲学就是将处理逻辑的通用部分和专用部分分离。

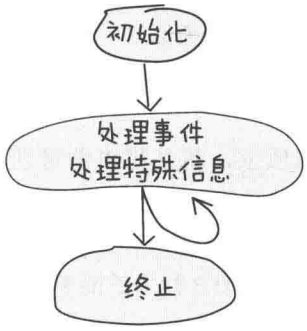
不过，这种分离意味着，标准模式中的进程创建、初始化、循环、终止函数只能应用到事件处理器上面。回忆一下，事件处理器只是运行在管理器中的一组函数。也就是说，程序员要把当前模型从：



切换到如下图所示的样子。



每个事件处理器都可以拥有自己的状态，这些状态由事件管理器进程携带。每个事件处理器都具有如下形式。



现在，我们来看看事件处理器的回调函数。

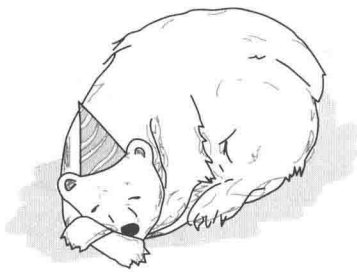
16.2.1　init 和 terminate 函数

init 和 terminate 函数与我们前面看到的 gen_server 和 gen_fsm 行为中的类似。init/1 函数接收列表参数，返回{ok，State}。在 init/1 中创建的东西，要在 terminate/2 函数中有对应释放操作。

16.2.2　handle_event 函数

handle_event(Event,State) 函数可以说是 gen_event 回调模块的核心函数。和 gen_server 中的 hanle_cast/2 一样，handle_event/2 函数也是异步的。不过，它的返回值有所不同：

- {ok, NewState};
- {ok, NewState, hibernate}，让事件管理器进程进入休眠状态，直到收到下一个事件；
- remove_handler;
- {swap_handler, Args1, NewState, NewHandler, Args2}。



返回值{ok, NewState}元组的含义和 gen_server: handle_cast/2 函数中的一样。它只更新自己的状态，不做任何回应。返回{ok, NewState, hibernate}则会使整个事件管理器进入休眠状态。记住，事件处理器和其管理器运行在同一个进程中。

返回 remove_handler 则会导致事件处理器从事件管理器中删除。当某个事件处理器知道自己已经完成工作并且无其他任务时，可以使用这个返回值。

最后一个是返回值{swap_handler, Args1, NewState, NewHandler, Args2}，这个返回值不太常用。它移除当前事件处理器并用一个新的替代它。事件管理器首先调用 CurrentHandler:terminate(Args1,NewState)函数，并移除当前的事件处理器。接着调用 NewHandler:init(Args2,ResultFromTerminate)函数，添加新的事件处理器。当你知道某个特定事件发生在何处，需要将控制权转交给一个新事件处理器时，可以返回这个值。什么时候该返回这个值呢？一般来讲，需要的时候就知道了，到时直接用就行了。

所有的事件都是通过 gen_event:notify/2 函数触发的，和 gen_server:cast/2 一样，它也是异步的。还有另外一个函数 gen_event:sync_notify/2，它是同步的。由于 handle_event/2 是异步的，因此存在这么一个同步函数感觉有点奇怪。这里的同步指的是，当所有的事件处理器都收到这个事件并且处理完毕后，sync_notify 函数才会返回。在那之前，事件管理器会一直阻塞调用进程，不予响应。

16.2.3 handle_call 函数

handle_call 函数和 gen_server 的 handle_call 回调函数类似，不同之处在于，它可以返回{ok, Reply, NewState}、{ok, Reply, NewState, hibernate}、{remove_handler, Reply}以及{swap_handler, Reply, Args1, NewState, Handler2, Args2}。使用 gen_event:call/3-4 函数可以发起该调用。

这就引出了一个问题：如果有 15 个不同的事件处理器，它该如何工作呢？是希望有 15 个回应呢？还是希望只有一个回应，其中包含了所有内容呢？嗯，事实上，我们只能选择一个事件处理器做出回应。具体的工作原理，我们会在 16.3.2 节向事件管理器中增加处理器时进行详细介绍，不过，如果读者已经等不及了，可以查阅 gen_event:add_handler/3 函数的文档了解详情。

16.2.4 handle_info 函数

handle_info 回调和 hanle_event 回调非常类似（有着同样的返回值和含义），唯一的不同在于，handle_info 只处理带外消息，如退出信号或使用！操作符直接向事件管理器进程发送消息。它的使用场景和 gen_server 以及 gen_fsm 中 handle_info 的使用场景类似。

16.2.5 code_change 函数

code_change 函数的工作方式和 gen_server 中的一样，不过它仅仅针对单独的事件处理器。它的参数为 OldVsn、State、Extra，分别表示版本号、当前事件处理器的状态，最后这个参数——Extra，目前可以不用关心。这个方法只要返回{ok, NewState}即可。

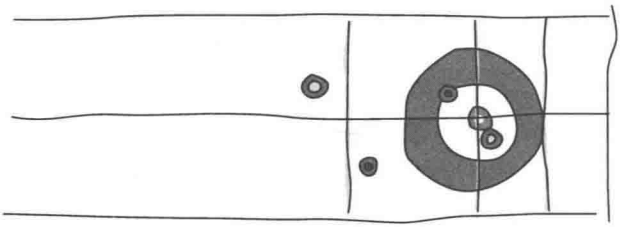
16.3 现在是冰壶比赛时间

是时候来看看我们能用 gen_event 做点什么了。在本例中，我们会创建一系列的事件处理器来追踪赛事更新，所追踪的是世界上最有趣的体育赛事之一：冰壶比赛。

对那些从来没有看过或者玩过冰壶运动的人来说（这无疑是一个遗憾），游戏规则还是比较

简单的。两队都尽力推着冰壶石（一块厚厚的大石饼，重量约在 38～44 磅，也就是 17～20 千克，上面还有一个提手），让它在冰面上滑动，尽力把它送到红圈的中央。

两队一共有 16 块冰壶石。每个回合结束时（称为一次*终局*），如果某队有 1 块最靠近中心的冰壶石，那这个队就得 1 分。如果有 2 块最靠近中心的冰壶石，就得 2 分；如果有 3 块最靠近中心的冰壶石，就得 3 分，以此类推。一场比赛一共 10 次终局，在 10 次终局完成后，得分最高的团队赢得比赛。



还有其他一些让这个游戏更加引人入胜的规则。但这是一本关于 Erlang 的书，而不是关于引人入胜的冬季运动项目的书。如果你想了解更多关于冰壶游戏的知识，我建议你去浏览一下维基百科上的相关文章。

假设我们正在为下一届冬季奥运会工作，这是一个完全真实的场景。已万事俱备，比赛使用的运动场已经准备完成，工作人员正在调试记分牌。此时我们发现需要一个软件系统，可以让官方工作人员录入比赛事件。例如，抛出了一块冰壶石，一个回合结束了以及比赛结束了——接着，这些事件会被分别路由到记分牌上、某个统计系统中、新闻播报员的提示器中等。

聪明的读者，既然本章是关于 gen_event 的，你们一定能推断出我们会用 gen_event 来完成这项任务。在本例中，我们不会实现比赛的所有规则，但是在完成这个样例应用之后，你可以随时完成这项工作——我保证不会太复杂。

16.3.1　记分牌

从记分牌开始。因为工作人员目前正在安装记分牌，所以我们可以使用一个仿造（fake）模块。一般来讲，仿造模块可以让我们进行一些交互测试，不过对于本例来说，只需在标准输出上显示一些当前发生的事情就可以了。这是一种模拟（mock）实现，它用来帮助我们开发与系统中尚不存在的部分进行交互的代码。于是，就有了下面的 curling_scoreboard_hw.erl 文件：

```
-module(curling_scoreboard_hw).
-export([add_point/1, next_round/0, set_teams/2, reset_board/0]).

%% 这是一个“哑”模块，只是用于代替真正的硬件控制器要做的事情。
%% 一个真实的硬件控制器要能保存某些状态并且确保一切工作正常，但是这个模块无需关心这些

%% 在记分牌上显示参赛队伍
set_teams(TeamA, TeamB) ->
    io:format("Scoreboard: Team ~s vs. Team ~s~n", [TeamA, TeamB]).

next_round() ->
    io:format("Scoreboard: round over~n").
```

```
add_point(Team) ->
    io:format("Scoreboard: increased score of team ~s by 1~n", [Team]).

reset_board() ->
    io:format("Scoreboard: All teams are undefined and all scores are 0~n").
```

以上就是记分牌模块的全部功能。通常，记分牌还会有一个计时器以及其他一些复杂的功能。不过，奥林匹克委员会似乎不希望我们在一个教程中实现那些不重要的东西。

16.3.2 比赛事件

基于这个硬件接口，我们可以做一点自己的设计了。目前，我们知道这几个事件是需要处理的：增加参赛队伍、进入下一回合以及设置分数。仅当开始新的比赛时，我们才会调用 reset_board 函数，所以不需要把它放到协议中。在我们的协议中，所需事件可以采用如下的格式：

- {set_teams, TeamA, TeamB}，这个事件会被转换成对 curling_scoreboard_hw:set_teams (TeamA, TeamB) 函数的一次调用。
- {add_points, Team, N}，这个事件会被转换对函数 curling_scoreboard_hw:add_point(Team) 的 *N* 次调用。
- next_round，这个事件会被转换成对同名函数的一次调用。

以下面这个简单的事件处理器骨架为基础，开始实现：

```
-module(gen_event_callback).
-behavior(gen_event).
-export([init/1, handle_event/2, handle_call/2, handle_info/2, code_change/3,
terminate/2]).

init([]) -> {ok, []}.

handle_event(_, State) -> {ok, State}.

handle_call(_, State) -> {ok, ok, State}.

handle_info(_, State) -> {ok, State}.

code_change(_OldVsn, State, _Extra) -> {ok, State}.

terminate(_Reason, _State) -> ok.
```

对于所有的 gen_event 回调模块，都可以用上面的代码骨架。目前，记分牌事件处理器无需做什么特别的事情，只要把调用转交给硬件模块即可。我们希望用 gen_event:notify/2 函数来触发事件，这样对协议的处理就应该在 handle_event/2 中完成。文件 curling_scoreboard.erl 展示了对代码骨架文件的修改，具体如下：

```
-module(curling_scoreboard).
-behavior(gen_event).
```

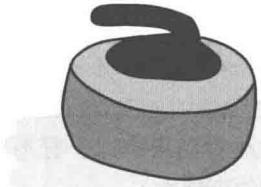
```
-export([init/1, handle_event/2, handle_call/2, handle_info/2, code_change/3,
terminate/2]).

init([]) ->
    {ok, []}.

handle_event({set_teams, TeamA, TeamB}, State) ->
    curling_scoreboard_hw:set_teams(TeamA, TeamB),
    {ok, State};
handle_event({add_points, Team, N}, State) ->
     [curling_scoreboard_hw:add_point(Team) || _ <- lists:seq(1,N)],
    {ok, State};
handle_event(next_round, State) ->
    curling_scoreboard_hw:next_round(),
    {ok, State};
handle_event(_, State) ->
    {ok, State}.

handle_call(_, State) ->
    {ok, ok, State}.

handle_info(_, State) ->
    ok, State}.
```

从中，你可以看到对 handle_event/2 函数所做的更改。我们来试一下：

```
1> c(curling_scoreboard_hw).
{ok,curling_scoreboard_hw}
2> c(curling_scoreboard).
{ok,curling_scoreboard}
3> {ok, Pid} = gen_event:start_link().
{ok,<0.43.0>}
4> gen_event:add_handler(Pid, curling_scoreboard, []).
ok
5> gen_event:notify(Pid, {set_teams, "Pirates", "Scotsmen"}).
Scoreboard: Team Pirates vs. Team Scotsmen
ok
6> gen_event:notify(Pid, {add_points, "Pirates", 3}).
ok
Scoreboard: increased score of team Pirates by 1
Scoreboard: increased score of team Pirates by 1
Scoreboard: increased score of team Pirates by 1
7> gen_event:notify(Pid, next_round).
Scoreboard: round over
ok
8> gen_event:delete_handler(Pid, curling_scoreboard, turn_off).
ok
9> gen_event:notify(Pid, next_round).
Ok
```

有几件事情需要说明一下。首先，gen_event 是作为一个独立进程启动的。其次，使用

gen_event: add_handler/3 函数把事件处理器动态地添加进去。可以根据需要多次调用这个操作。不过，在前面介绍 handle_call 函数时提到过，这种做法在你想使用某个特定的事件处理器时，可能会引发一些问题。



如果某个处理器有多个实例，当你想去调用、添加或者删除某个特定实例时，就需要找到一种能够唯一地标识这个实例的方法。在处理这个问题上，我最喜欢的方法（如果你没有什么特殊要求，这个方法非常适用）是通过 make_ref()来产生一个唯一的值。为了把这个值和处理器对应起来，你可以通过调用 add_handler/3 函数：gen_event:add_handler(Pid, {Module, Ref}, Args)来实现。此后，可以使用{Module, Ref}来标识这个特定的事件处理器，问题就解决了。

接下来，我们给事件处理器发送消息，这个事件处理器成功地调用了硬件模块。然后，删除这个事件处理器。这里，turn_off 会作为参数传递给 terminate/2 函数，目前的实现中忽略了这个参数。事件处理器虽然被删除了，但我们仍然能够给事件管理器发送消息。万岁！

前面的命令行调用中，有个地方比较粗糙，我们直接调用了 gen_event 模块，并把实现协议暴露给了所有人。比较好的做法是，在此之上提供一个抽象模块，它对所有需要的调用进行封装。这样，会让每个使用代码的人感觉舒服些，还可以在需要时去更改实现。同时，这样做还能指定标准冰壶比赛中必须要包含的事件处理器。

```
-module(curling).
-export([start_link/2, set_teams/3, add_points/3, next_round/1]).

start_link(TeamA, TeamB) ->
    {ok, Pid} = gen_event:start_link(),
    %% 记分牌是必需的。
    gen_event:add_handler(Pid, curling_scoreboard, []),
    set_teams(Pid, TeamA, TeamB),
    {ok, Pid}.

set_teams(Pid, TeamA, TeamB) ->
    gen_event:notify(Pid, {set_teams, TeamA, TeamB}).

add_points(Pid, Team, N) ->
    gen_event:notify(Pid, {add_points, Team, N}).

next_round(Pid) ->
    gen_event:notify(Pid, next_round).
```

现在，可以来运行一下。

```
1> c(curling).
{ok,curling}
2> {ok, Pid} = curling:start_link("Pirates", "Scotsmen").
Scoreboard: Team Pirates vs. Team Scotsmen
```

```
{ok,<0.78.0>}
3> curling:add_points(Pid, "Scotsmen", 2).
Scoreboard: increased score of team Scotsmen by 1
Scoreboard: increased score of team Scotsmen by 1
ok
4> curling:next_round(Pid).
Scoreboard: round over
ok
```

从运行结果中似乎看不出修改后的优势，但是它确实提升了代码的易用性（并且还减少了消息写错的可能性）。

大功告成，官方工作人员现在可以使用这些代码了。不过，奥委会要求我们再做些工作，还得满足新闻界的需求。

16.3.3　通知新闻界

我们希望国际记者们能够从负责更新比赛消息的官方工作人员那里获取到实时的比赛数据。因为这是一个示例程序，所以这里不打算涉及创建套接字、编写消息更新协议等内容，只是在系统中放置一个中间进程来完成这项工作。

一般来讲，当某个新闻组织希望得到比赛消息时，它可以注册自己的事件处理器来转发自己所需要的数据。事实上，就是要把 gen_event 服务器变成一种消息集线器（message hub），把消息路由给需要的人。

首先，我们要修改 curling.erl 模块，为它增加新的接口。出于易用性考虑，只增加两个函数：join_feed/2 和 leave_feed/2。想要获取比赛消息，只需提供正确的事件管理器 pid 和事件要发往的 pid 即可。这样做后会返回一个唯一的值，以后需要撤销订阅时，可以把这个值作为参数传递给 leave_feed/2 函数。

```
%% 为进程 ToPid 订阅比赛消息。
%% 特定于本次订阅的事件处理器的唯一标识被返回，
%% 可以在取消订阅时使用
join_feed(Pid, ToPid) ->
    HandlerId = {curling_feed, make_ref()},
    gen_event:add_handler(Pid, HandlerId, [ToPid]),
    HandlerId.

leave_feed(Pid, HandlerId) ->
    gen_event:delete_handler(Pid, HandlerId, leave_feed).
```

注意，我们使用了前面介绍的用于多处理器实例的技术（{curling_feed, make_ref()}）。可以看出，这个函数需要一个名为 curling_feed 的 gen_event 回调模块。如果只用该模块的名字作为 HandleId，当然也能工作得不错，但是当某个事件处理器实例完成工作时，我们没有办法删除这个处理器实例。此时，事件管理器会以未定义的方式从中选择一个删除。使用引用作为标识，可以确保来自 *Head-Smashed-In Buffalo Jump* (Alberta，加拿大) 报社的某个家伙的离开不会切断来自 *The Economist*（谁知道这个杂志为什么要报道冰壶比赛呢）的记者

的比赛消息源。curling_feed 模块的代码实现如下：

```
-module(curling_feed).
-behavior(gen_event).

-export([init/1, handle_event/2, handle_call/2, handle_info/2, code_change/3,
terminate/2]).

init([Pid]) -> {ok, Pid}.

handle_event(Event, Pid) ->
    Pid ! {curling_feed, Event},
    {ok, Pid}.

handle_call(_, State) -> {ok, ok, State}.

handle_info(_, State) -> {ok, State}.

code_change(_OldVsn, State, _Extra) -> {ok, State}.

terminate(_Reason, _State) -> ok.
```

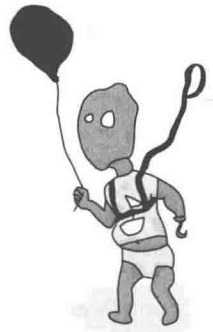
代码中唯一有趣的地方仍是 handle_event/2 函数，该函数直接将所有事件都转发给订阅进程。

现在，我们来使用一下这个新模块。

```
1> c(curling), c(curling_feed).
{ok,curling_feed}
2> {ok, Pid} = curling:start_link("Saskatchewan Roughriders",
2>                                 "Ottawa Roughriders").
Scoreboard: Team Saskatchewan Roughriders vs. Team Ottawa Roughriders
{ok,<0.165.0>}
3> HandlerId = curling:join_feed(Pid, self()).
{curling_feed,#Ref<0.0.0.909>}
4> curling:add_points(Pid, "Saskatchewan Roughriders", 2).
Scoreboard: increased score of team Saskatchewan Roughriders by 1
ok
Scoreboard: increased score of team Saskatchewan Roughriders by 1
5> flush().
Shell got {curling_feed,{add_points,"Saskatchewan Roughriders",2}}
ok
6> curling:leave_feed(Pid, HandlerId).
ok
7> curling:next_round(Pid).
Scoreboard: round over
ok
8> flush().
ok
```

这里，我们用 shell 进程订阅比赛消息、获取赛事更新，然后离开并停止接收消息。你可以

试着多次加入多个订阅者，也会一切正常。



不过，这引入了一个问题。上例中，如果冰壶比赛订阅者进程崩溃了会怎样呢？还需要继续留着这个处理器吗？理想情况下，我们不想这样做；实际上，我们也不需要做什么。只需要把调用 gen_event:add_handler/3 更改为调用 gen_event:add_sup_handler/3 即可。如果崩溃了，那么事件处理器就自行消失了。相反地，如果 gen_event 事件管理器崩溃了，那么订阅者进程会收到{gen_event_EXIT, Handler, Reason}消息，这样就可以进行相应处理。很简单，不是吗？仔细想想！

保持冷静

童年时，你可能会去阿姨或者祖母家里参加聚会或者其他活动。在那种场合下，除了父母外，还会有其他一些长辈时刻监视着你。一旦你犯了错，这些人会一拥而上地加以指责，其中有妈妈、爸爸、阿姨、祖母等。在你已经清楚地认识到自己确实做错了之后的很长一段时间，他们还在絮絮叨叨个不停。嗯，gen_event:add_sup_handler/3 函数跟这有点像——不是开玩笑。

当调用 gen_event:add_sup_handler/3 函数时，会在你的进程和事件管理器进程之间建立一个链接，这样这两个进程就都被监督着，并且处理器可以知道其父进程是否死亡。在 12.2 节中我曾经说过，监控器很适合用来编写需要知道其他进程运行情况的库。和链接（link）不同，监控器是可以堆叠的（stacked）。不过，在 Erlang 中，gen_event 早于监控器出现，而对向后兼容的强制要求导致了这个非常糟糕的缺陷。一般来说，某个进程可以是多个事件处理器的父进程，以防万一，事件管理器进程根本不会解除和处理器父进程之间的链接（除非管理器进程永久终止）。监控器可以从根本上解决这个问题，但是却无法用在这里。

这意味着，当你自己的进程崩溃时，一切正常——被监督的处理器会被终止（会调用 YourModule:terminate({stop, Reason}, State)）。当你的处理器代码崩溃时（事件管理器没有问题），也一切正常——你的进程会收到{gen_event_EXIT, HandlerId, Reason}消息。但是，当事件管理器进程停止时，会出现如下两种情况：

- 你的进程会收到{gen_event_EXIT, HandlerId, Reason}消息，然后因为没有捕获到退出信号而崩溃。
- 你的进程会先收到{gen_event_EXIT, HandlerId, Reason}消息，然后再收到一条标准的'EXIT'消息，这第二条消息不是多余，就是令人困惑。

这确实是一个缺陷，但是至少你已经了解它了。如果愿意，你可以尝试把事件处理器切换成受监督的事件处理器。尽管在某些情况下会有些烦人，但是确实会安全一些。毕竟安全第一。

还没结束呢！如果某些媒体记者没有及时订阅比赛消息该怎么办呢？我们要能让他们从接口中获取当前比赛的状态信息。因此，我们要编写另外一个事件处理器 curling_accumulator 来专门处理这种情况。同样，在完整实现它之前，先在 curling 模块中增加一些我们希望的调

用，如下：

```
-module(curling).
-export([start_link/2, set_teams/3, add_points/3, next_round/1]).
-export([join_feed/2, leave_feed/2]).
-export([game_info/1]).

start_link(TeamA, TeamB) ->
    {ok, Pid} = gen_event:start_link(),
    %% 记分牌是必需的
    gen_event:add_handler(Pid, curling_scoreboard, []),
    %% 启动比赛状态累加器
    gen_event:add_handler(Pid, curling_accumulator, []),
    set_teams(Pid, TeamA, TeamB),
    {ok, Pid}.
...

%% 返回当前比赛状态
game_info(Pid) ->
    gen_event:call(Pid, curling_accumulator, game_data).
```

注意，在 game_info/1 函数中，只把 curling_accumulator 当作事件处理器的标识。但是，如果某个事件处理器存在很多版本，前面讲的使用 make_ref()（或者其他方法）来确保找到正确的事件处理器的方法仍然是适用的。还要注意，curling_accumulator 事件处理器和记分牌一样，都是自动启动的。

现在，我们来实现 curling_accumulator 模块。它需要保存冰壶比赛的状态。到目前为止，需要跟踪的信息包括参赛团队、得分以及比赛回合。可以把这些信息全部保存在一个 state 记录中，每当收到事件时，就对它进行更新。接下来，就只需要实现对 game_data 事件的回应了，代码如下：

```
-module(curling_accumulator).
-behavior(gen_event).

-export([init/1, handle_event/2, handle_call/2, handle_info/2, code_change/3,
terminate/2]).

-record(state, {teams=orddict:new(), round=0}).

init([]) -> {ok, #state{}}.

handle_event({set_teams, TeamA, TeamB}, S=#state{teams=T}) ->
    Teams = orddict:store(TeamA, 0, orddict:store(TeamB, 0, T)),
    {ok, S#state{teams=Teams}};
handle_event({add_points, Team, N}, S=#state{teams=T}) ->
    Teams = orddict:update_counter(Team, N, T),
    {ok, S#state{teams=Teams}};
handle_event(next_round, S=#state{}) ->
    {ok, S#state{round = S#state.round+1}};
handle_event(_Event, State=#state{}) ->
    {ok, State}.
```

```
handle_call(game_data, S=#state{teams=T, round=R}) ->
    {ok, {orddict:to_list(T), {round, R}}, S};
handle_call(_, State) ->
    {ok, ok, State}.

handle_info(_, State) -> {ok, State}.

code_change(_OldVsn, State, _Extra) -> {ok, State}.

terminate(_Reason, _State) -> ok.
```

可以看到，我们只是不断地更新比赛状态，直到有人请求比赛的详细信息，此时，就把保存的信息发送给他们。这个实现非常简单。也许，一种更聪明的做法是：把比赛中发生的所有事件直接保存在一个列表中，每当有新的进程订阅比赛信息时，就将这个列表发给它。不过，这个例子中，不必这么做，现在来实验一下新程序。运行新程序：

```
1> c(curling), c(curling_accumulator).
{ok,curling_accumulator}
2> {ok, Pid} = curling:start_link("Pigeons", "Eagles").
Scoreboard: Team Pigeons vs. Team Eagles
{ok,<0.242.0>}
3> curling:add_points(Pid, "Pigeons", 2).
Scoreboard: increased score of team Pigeons by 1
ok
Scoreboard: increased score of team Pigeons by 1
4> curling:next_round(Pid).
Scoreboard: round over
ok
5> curling:add_points(Pid, "Eagles", 3).
Scoreboard: increased score of team Eagles by 1
ok
Scoreboard: increased score of team Eagles by 1
Scoreboard: increased score of team Eagles by 1
6> curling:next_round(Pid).
Scoreboard: round over
ok
7> curling:game_info(Pid).
{[{"Eagles",3},{"Pigeons",2}],{round,2}}
```

太令人振奋了！奥林匹克委员会肯定会爱上这个程序。现在我们可以洋洋得意，拿着大把现金支票，整晚打视频游戏了。

在本章中，我们并没有介绍 gen_event 模块的所有功能。事实上，我们甚至没有介绍事件处理器最常见的用途：日志和系统警告。之所以这么做，是因为专门讲解这方面内容的 Erlang 资料已经非常多了。如果你对这些用法感兴趣，可以去看看标准库中 error_logger 模块的相关内容。

虽然我们没有介绍 gen_event 最常见的用法，但是我们探讨了理解这些用法、构建自己的实现并把它们集成到自己的应用中所必需的所有概念。更为重要的是，我们已经学习了实际代码开发中会用到的 3 种主要 OTP 行为。还有一些行为有待我们学习——这些行为会作为黏合剂把工作进程黏合在一起，如监督者（supervisor），这是第 17 章的主要内容。

第 17 章

谁来监督监督者

监督者是 OTP 中最有用的部分之一。在第 12 章和第 13 章中，我们已经用到了一些简单的监督者，它们提供了一种方法，仅通过重启出错的进程，就能让软件在出现错误时继续运行。在本章中，我们会介绍 OTP 中的监督者，它远胜于我们自己编写的版本。

在前面的例子中，我们实现的监督者会启动一个工作者进程，和它链接在一起，并用 `process_flag(exit, true)` 来捕获它的退出信号，在得知工作者进程死亡后重启它。如果我们要做的只是重启，这样的工作方式固然不错，但是未免有些愚蠢。假如你正在用遥控器开启电视。第一次没成功，你可能会再试一次或者两次，以防万一出现按错按钮或者信号发送错误的情况。但换做我们的监督者，即使遥控器没电了或者和那台电视不配套，它也会一直不停地尝试打开电视。真是个蠢到家的监督者。

在我们实现的监督者中，还有一个比较愚蠢的地方，它们一次只能监控一个工作者进程。尽管某些情况下，为每个工作者进程安排一个监督者是合适的，但在大型应用中，这样的做法意味着这些监督者只会形成一条链，而不是一棵树。当一个任务需要 2 个或者 3 个工作者进程才能完成时，该如何监督呢？我们的实现方式根本做不到这一点。

幸运的是，OTP 中的监督者非常灵活，可以解决上面的问题（以及更多其他类型的问题）。在本章中你将看到，通过它们我们可以定义在一个指定的周期内重启工作者进程的最大次数。它们能让一个监督者监督多个工作者，甚至还提供了一些模式供你选择，能够让你决定工作者进程之间的失败依赖关系。

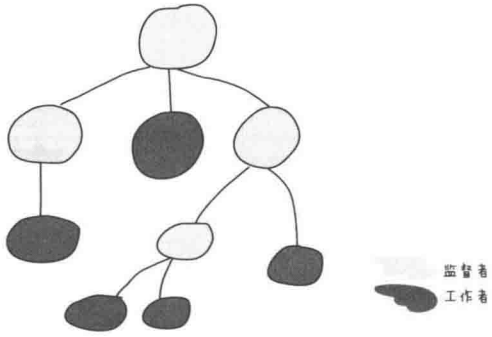
17.1 监督者中的概念

在所有 OTP 行为中，监督者是最容易使用和理解的一个，但也是最难设计好的一个。与监督者以及应用设计相关的策略多种多样，不过在啃这些硬骨头之前，我们有必要先来了解一些基本概念。

在本书的前面，我曾经使用过一些术语，但并没有给出明确的定义，工作者（worker）就是

其中的一个。可以从监督者相反的角度定义工作者。如果把监督者定义为一个除了保证在其子进程死亡后重启它们之外，不做其他任何事情的进程，那么工作者就是那些负责完成实际的工作，并且在工作的过程中可能会死掉的进程。工作者进程通常都被认为是不可靠的。

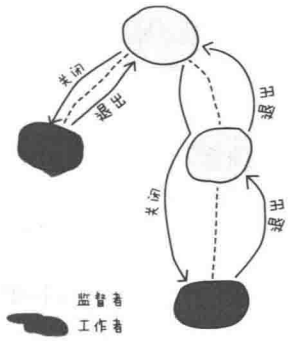
监督者可以监督工作者和其他监督者。工作者只能放置在一个监督者下面：



为什么每个进程都要被监督呢？嗯，道理很简单：如果创建了未受监督的进程，如何才能确认它们死了呢？如果某个东西无法被度量，那么它就是不存在。如果一个进程悄无声息地游离在所有的监督树之外，如何知道它是否真的存在？它是怎么产生的？这样的事情未来还会发生吗？

如果真出现了这种情况，就说明内存在缓慢地泄露——非常慢，以至于 VM 会因为没有可用内存而突然死亡；非常慢，以至于虽然不断地出现问题，却无法轻易定位它。当然，你可能会说，“如果我当心一点，想清楚要做的事情，就不会出问题。”可能不出问题，但是也可能出问题。在一个生产系统中，你肯定不想冒险。这也是为什么 Erlang 语言一开始就有垃圾回收的原因。把东西都监督起来非常有用。

此外，监督机制还可以让你以恰当的顺序终止应用。虽然没想让自己编写的 Erlang 软件永远运行，但还是会希望它能够干净地结束。如何知道所有的东西都已经做好了关闭的准备呢？使用监督者，就很容易知道了。当想终止一个应用时，只需去终止虚拟机中最顶层的那个监督者（调用 `init:stop/1` 函数就可以了）。然后这个监督者会接着要求它的子进程停止运行。如果某个子进程也是一个监督者，它会做同样的事情：



如果没有用树形结构来组织所有的进程，那么很难让 VM 以井然有序的方式终止。当然，有时也会出现进程因某种原因被卡住而不能正常终止的情况。此时，监督者可以强行杀死这个进程。

现在，我们有了工作者、监督者、监督树、指定依赖关系的方法以及指定监督者何时放弃重启子进程或者等待子进程结束的方法等。监督者所能做的并不仅限于这些事情，不过现在，这些知识足以让我们来看看如何使用它们了。

17.2 使用监督者

这应该是目前为止最为暴力的一节：父亲们花费时间把它们的孩子都绑到树上，孩子们被强迫工作，最终还得被强行杀死。但我们并不是真正的暴力狂，不会自己动手做这些事情。

我曾经说过，监督者使用起来很简单，这不是一句玩笑话。我们只需要提供一个回调函数：`init/1`。麻烦的地方在于这个函数的返回值非常复杂。下面是一个返回值的例子：

```
{ok, {{one_for_all, 5, 60},
      [{fake_id,
        {fake_mod, start_link, [SomeArg]},
        permanent,
        5000,
        worker,
        [fake_mod]},
       {other_id,
        {event_manager_mod, start_link, []},
        transient,
        infinity,
        worker,
        dynamic}]}}.
```

这些内容是什么意思？下面给出的是这个返回值的一般形式定义，更易于解释一些：

```
{ok, {{RestartStrategy, MaxRestart, MaxTime},[ChildSpec]}}.
```

我们来逐条讲解。

17.2.1 重启策略

上述定义中的 `RestartStrategy` 的值可以为 `one_for_one`、`one_for_all`、`rest_for_one` 以及 `simple_one_for_one`。

1. `one_for_one`

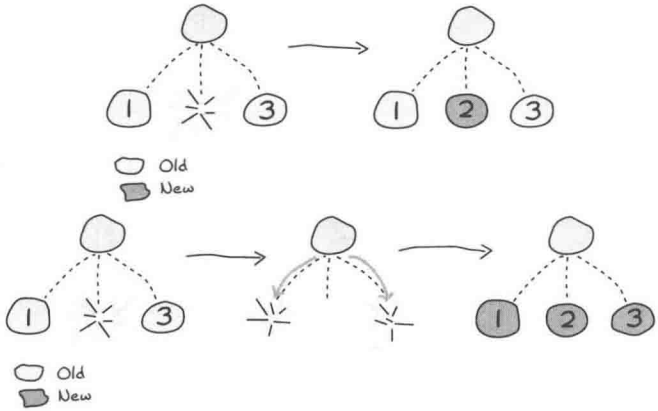
`one_for_one` 是个很容易理解的重启策略。它指的是，如果监督者监督了很多工作者，当其中的一个工作者失败时，只需重启这个工作者即可。当被监督的进程都是独立的、互不相关的，或者即便这些进程重启后丢失了自己的状态，也不会对其兄弟进程产生影响时，可以使用 `one_for_one` 策略。

2. `one_for_all`

`one_for_all` 和 musketeers[①]没有任何关系。当所有的工作者进程都受同一个监督者监督，且这些工作者进程必须互相依赖才能正常工作时，就使用这个策略。假如，我们想在第 15 章中实现的交易系统之上增加一个监督者。买卖双方，任何一方崩溃了，只重启失败的这一方是没有意义的，因为双方的交易状态会失去同步。同时重启交易的双方才是一个明智的选择，`one_for_all` 就是

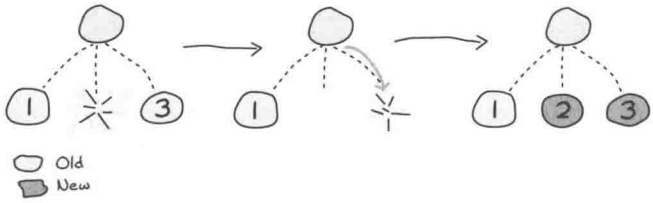
① 一款著名游戏，全称为 The Three Musketeers：One For All!。——译者注

专门解决此类问题的策略。



3. rest_for_one

rest_for_one 是一个更加特殊的策略。当需要启动一组进程，而这些进程互相依赖形成一条链（A 启动 B，B 启动 C，C 启动 D，以此类推）时，可以使用 rest_for_one。此外，如果有一些服务，它们之间存在类似的依赖关系（X 独立运行，Y 依赖于 X，Z 同时依赖于 X 和 Y），也可以使用这种策略。简单来讲，使用 rest_for_one 重启策略，如果一个进程死了，那么所有在这个进程之后启动的进程（依赖于该进程）都将被重启，反之不然。



4. simple_one_for_one

虽然叫这个名字，但是 simple_one_for_one 这个重启策略一点都不简单。这种类型的监督者只监督一种子进程，当希望以动态的方式向监督者中增加子进程，而不是静态启动子进程时，可以使用这种策略。

换种说法，simple_one_for_one 监督者就待在那里，它知道自己只能创建一种类型的子进程。当需要一个新的子进程时，向它发起请求，就能得到一个。理论上来讲，用标准的 one_for_one 监督者也可以达到这个目的，但是使用这个更简单的策略会有一些实际的好处，在本章后面讲到动态监督时，你会看到。

> **注意**　one_for_one 和 simple_one_for_one 两者之间有一个很大的不同。one_for_one 监督者会把它启动的所有子进程保存在一个列表中（包括那些曾经启动的，如果你手工操作后没有清除的话），按进程启动顺序排序，而 simple_one_for_one 只保

存一份针对所有子进程的定义，并用字典来保存子进程数据。一般来说，如果子进程的数量很多，那么当某一个子进程崩溃时，simple_one_for_one 监督者要快很多。

17.2.2 重启限制

RestartStrategy 元组中剩余的两个变量是 MaxRestart 和 MaxTime。它们的意思是，如果在 MaxTime（以秒为单位）指定的时间内，重启次数超过了 MaxRestart 指定的数字，那么监督者会放弃重启并终止所有子进程，然后自杀，永远停止运行。这两个变量是针对监督者管辖的所有子进程的重启来说的，不是单独针对某个子进程的。好在该监督者之上的监督者可能对其子进程还抱有希望，也许会重新启动它们。

17.2.3 子进程规格说明

现在来看看返回值中的 ChildSpec 部分。ChildSpec 表示子进程规格说明。前面的例子中有如下两个子进程的规格说明：

```
[{fake_id,
  {fake_mod, start_link, [SomeArg]},
  permanent,
  5000,
  worker,
  [fake_mod]},
 {other_id,
  {event_manager_mod, start_link, []},
  transient,
  infinity,
  worker,
  dynamic}]
```

子进程规格说明可以描述成如下更为抽象的形式：

```
{ChildId, StartFunc, Restart, Shutdown, Type, Modules}.
```

接下来我们看看每个部分都是什么意思。

1. ChildId

ChildId 只是监督者内部使用的一个名称。除了在调试或者想获取监督者的所有子进程列表时，这个名字会比较有用之外，基本上没有其他用途。可以把任意的数据项作为这个标识使用，不过，我建议最好选择一些易读的名字，以备调试使用。

2. StartFunc

StartFunc 是一个元组，用来指定子进程的启动方式。它采用了标准的{M,F,A}格式，这个格式前面我们已经多次使用过。注意，这里的启动函数是 OTP 兼容的，在执行时会和调用者进程链接在一起，这一点非常重要。(提示：始终记住，在自己的模块中调用 gen_*:start_link)。

3. Restart

Restart 指定了监督者在某个特定的子进程死后的处理方式，它可以取如下 3 个值：

- permanent;
- temporary;

- transient。

不管发生什么情况，一个永久（permanent）进程都要被重启。在之前的应用中，我们自己实现的监督者就只使用了这个策略。这个策略通常适用于那些至关重要的、需要在节点上长期存活的进程（或服务）。

另一方面，临时（temporary）进程指的是那种绝对不应该被重启的进程。此类进程都是些短暂的工作者进程，完成工作后就会死亡，基本上也没有什么代码依赖于它们。通常，仍然会希望使用监督者去监督它们，这样就知道它们在哪里，并通过监督者干净地清理它们。

暂态（transient）进程介于上述两种进程之间。这类进程会一直运行，如果被正常终止了，就不会被重启。不过，如果异常死亡（退出原因不是 normal、shutdown 和{shutdown, Reason}），就会被重启。如果要求工作者进程必须成功完成任务，但是完成后就不再需要它们了，则通常可以使用这个选项。

可以在单个监督者之下同时存在这 3 种类型的子进程。这可能会影响重启策略。临时进程的死亡不会触发 one_for_all 重启策略，但是如果一个永久进程先死了，那么同一个监督者管辖下的临时进程会被重启！

4. Shutdown

在本章的前面我提到过，可以通过监督者终止整个应用。它是这样做的：当要求最顶层的监督者终止时，它会对每个子进程调用 exit(ChildPid,shutdown)。如果这个子进程是一个工作者进程并且捕获了退出信号，那么就会调用自己的 terminate 函数；否则，进程死掉就行了。如果是一个监督者子进程收到了 shutdown 信号，它会用同样的方式将这个信号转发给它的子进程。

子进程规格说明中的 Shutdown 值用来指定终止的超时期限。对某些特定的工作者来说，你知道它们退出时需要做一些像正确地关闭文件、通知某个服务进程自己正在退出之类的事情。在这样的情况下，可以设置一个确定的终止超时，可以是多少毫秒，也可以是 infinity，如果有足够耐心的话。如果指定的时间过去了，进程还没有死，那么进程会被 exit(Pid,kill)强行杀死。如果对子进程并不在意，不设定超时等待时间时，子进程的死亡也没啥影响，那么可以将 Shutdown 设置成原子 brutal_kill。使用 brutal_kill 会调用 exit(Pid,kill)杀死子进程，此时，退出是即时的，子进程也无法捕获这个退出信号。

有些时候，选择一个恰当的 Shutdown 值是一件复杂或者难办的事情。假设有一条监督者链，它们的 Shutdown 值为 5 → 2 → 5 → 5，最后两个进程很可能会被强行杀死，因为链上的第二个监督者配置了较短的终止超时。选择一个合适的 Shutdown 值是和整个应用相关的，很少有通用的、放之四海皆准的建议。

> **注意**　在 Erlang R14B03 之前，使用 simple_one_for_one 重启策略的子进程不支持配置 Shutdown 时间。在 simple_one_for_one 的情况下，监督者进程只是自己退出了，每个工作者进程在它的监督者进程死掉后，要自己负责将本进程终止。

5. Type

Type 字段可以让监督者知道子进程是一个监督者（supervisor）（实现了 supervisor 或者 supervisor_bridge 行为）还是一个工作者（worker）（其他任何 OTP 进程）。在使用更多高阶 OTP 特性来做应用升级时，这个类型会比较重要，不过，目前你完全不用关心它——如实配置即可，一切都会正常工作的。要相信自己的监督者！

6. Modules

Modules 是一个列表，其中只有一个元素：子进程行为使用的回调模块名。有一个例外情况：事先无法知道回调模块的标识符（如事件管理器中的事件处理器模块）。此时，Modules 的值要设置成 dynamic，这样，在使用其他高级特性（如发布）时，整个 OTP 系统才能知道去找谁。

太好了，到现在为止，我们已经介绍了启动被监督进程所需要的基本知识。你现在可以休息一会，消化一下这些内容，也可以继续前进，学习一下监督者的实际运用。

17.3 乐队排练

是时候做些练习了。谈到练习，针对本章内容来说最完美的例子莫过于乐队排练了！（嗯，也不是那么完美，不过请稍安勿躁。）

假设，我们管理着一个名为*RSYNC 的乐队，由几个喜欢音乐的程序员组成：一个鼓手、一个主唱、一个贝斯手，还有一个键盘手（纪念辉煌但被遗忘的 20 世纪 80 年代）。尽管也有一些复古的打榜曲目，例如“线程安全性之舞”“周六之夜的编码者”等，但是乐队还是很难找到排练场地。含糖饮料再一次刺激我想出了用 Erlang 来模仿乐队的主意，我冲进你的办公室，把这个想法告诉你。你很疲惫，因为你和鼓手住在同一间公寓（这个鼓手是乐队中最差劲的，由于实在找不到其他鼓手，所以只能先凑合着），于是你接受了这个建议。

17.3.1 音乐人

首先要编写程序模拟每个乐队成员。在这个例子中，musicians（音乐人）模块要实现 gen_server 行为。每个音乐人都以一种乐器和一个技能等级作为参数（那个鼓手的技能等级很低，其他乐队成员都还不错）。一个音乐人进程一旦被创建，它就开始演奏起来。我们还给音乐人进程增加了一个停止选项，可以在需要时使用。基于此，我们定义了如下模块和接口函数。

```
-module(musicians).
-behavior(gen_server).

-export([start_link/2, stop/1]).
```

```
-export([init/1, handle_call/3, handle_cast/2,
         handle_info/2, code_change/3, terminate/2]).

-record(state, {name="", role, skill=good}).
-define(DELAY, 750).

start_link(Role, Skill) ->
    gen_server:start_link({local, Role}, ?MODULE, [Role, Skill], []).

stop(Role) -> gen_server:call(Role, stop).
```

我们定义了?DELAY 宏，用它作为每个音乐人两次弹奏之间的标准时间间隔。从记录定义可以看出，我们还给每个音乐人取了个名字，如下：

```
init([Role, Skill]) ->
    %% 这样，就可以知道父进程何时终止
    process_flag(trap_exit, true),
    %% 设定进程生存期内随机数生成使用的种子
    %% 使用当前时间。now()可以保证值唯一
    random:seed(now()),
    TimeToPlay = random:uniform(3000),
    Name = pick_name(),
    StrRole = atom_to_list(Role),
    io:format("Musician ~s, playing the ~s entered the room~n",
              [Name, StrRole]),
    {ok, #state{name=Name, role=StrRole, skill=Skill}, TimeToPlay}.
```

在 init/1 函数中，我们做了两件事情。首先，我们捕获了退出信号。你可以回想一下第 14 章中关于通用服务器的 terminate/2 函数的介绍，如果希望在服务器的父进程终止它时，它的 terminate/2 函数能得到调用，那么在服务器进程中就必须捕获退出信号。其次，init/1 函数还设置了一个随机种子（这样每个进程就能获得不同的随机数），并且还为自己创建了一个随机的名字。下面是创建名字的函数：

```
%% 没错，名字都来自《神奇校车》中的角色
%% 共 10 个名字！
pick_name() ->
    %% 使用随机函数时，必须设置种子。
    %% 请在调用了 init/1 函数的进程中使用
    lists:nth(random:uniform(10), firstnames())
    ++ " " ++
    lists:nth(random:uniform(10), lastnames()).

firstnames() ->
    ["Valerie", "Arnold", "Carlos", "Dorothy", "Keesha",
     "Phoebe", "Ralphie", "Tim", "Wanda", "Janet"].

lastnames() ->
    ["Frizzle", "Perlstein", "Ramon", "Ann", "Franklin",
     "Terese", "Tennelli", "Jamal", "Li", "Perlstein"].
```

现在可以来看看具体实现了。这个服务器的 handle_call 和 handle_cast 的实现非常

简单。

```
handle_call(stop, _From, S=#state{}) ->
    {stop, normal, ok, S};
handle_call(_Message, _From, S) ->
    {noreply, S, ?DELAY}.

handle_cast(_Message, S) ->
    {noreply, S, ?DELAY}
```

这里唯一的调用就是停止音乐人服务器进程，收到请求后，立即就结束了进程。如果收到了不期望的消息，我们没有回应，此时，调用者会崩溃。这可不是我们的问题。为什么在返回的{noreply, S, ?DELAY}元组中设置了超时，理由很简单，马上会看到。

```
handle_info(timeout, S = #state{name=N, skill=good}) ->
    io:format("~s produced sound!~n",[N]),
    {noreply, S, ?DELAY};
handle_info(timeout, S = #state{name=N, skill=bad}) ->
    case random:uniform(5) of
        1 ->
            io:format("~s played a false note. Uh oh~n",[N]),
            {stop, bad_note, S};
        _ ->
            io:format("~s produced sound!~n",[N]),
            {noreply, S, ?DELAY}
    end;
handle_info(_Message, S) ->
    {noreply, S, ?DELAY}.
```

每当服务器设置的超时到了，音乐人就会弹奏一个音符。如果这个音乐人的技能不错，那么一切完美。如果这个音乐人的技能水平糟糕，那它可能有五分之一的机会弹错一个音符，导致自己崩溃。在每个非终止调用的末尾也都设置了?DELAY 时间的超时。

接下来，我们增加一个空的 code_change/3 回调函数，这是 gen_server 行为要求的。

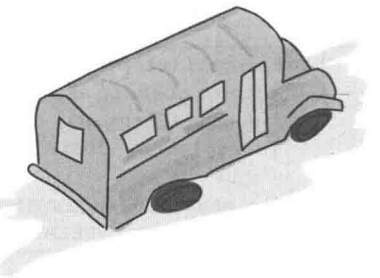
```
code_change(_OldVsn, State, _Extra) ->
    {ok, State}.
```

现在可以实现 terminate 函数了，如下：

```
terminate(normal, S) ->
    io:format("~s left the room (~s)~n",[S#state.name, S#state.role]);
terminate(bad_note, S) ->
    io:format("~s sucks! kicked that member out of the band! (~s)~n",
    [S#state.name, S#state.role]);
terminate(shutdown, S) ->
    io:format("The manager is mad and fired the whole band! "
              "~s just got back to playing in the subway~n",
               [S#state.name]);
terminate(_Reason, S) ->
    io:format("~s has been kicked out (~s)~n", [S#state.name, S#state.role]).
```

在 terminate 函数中处理了多条不同消息。如果终止原因是 normal，就说明调用了

stop/1 函数，因此就打印一条消息表明音乐人是自主离开的。如果原因是 bad_note，音乐人进程会崩溃，我们会打印一条消息说它是被管理者（也就是稍后我们会实现的监督者）开除乐队的。



接下来还有一个 shutdown 信息，这个 shutdown 来自于监督者。当出现这个原因时，意味着监督者决定杀死所有的子进程，对于我们的例子来说，就是开除所有的乐队成员。最后，我们增加了一个处理其他情况的通用错误处理函数。

我们简单使用一下 musicians 模块，如下：

```
1> c(musicians).
{ok,musicians}
2> musicians:start_link(bass, bad).
Ralphie Franklin, playing the bass entered the room
{ok,<0.615.0>}
Ralphie Franklin produced sound!
Ralphie Franklin produced sound!
Ralphie Franklin played a false note. Uh oh
Ralphie Franklin sucks! kicked that member out of the band! (bass)
3>
=ERROR REPORT==== 6-June-2013::03:22:14 ===
** Generic server bass terminating
** Last message in was timeout
** When Server state == {state,"Ralphie Franklin","bass",bad}
** Reason for termination ==
** bad_note
** exception error: bad_note
```

这里，音乐人 Ralphie 开始弹奏，在弹错了一个音符后崩溃了。如果用 good 参数启动音乐人进程，那么只能调用函数 musicians:stop(Instrument)结束演奏。

17.3.2　乐队监督者

现在，我们来实现乐队监督者。要实现的监督者有 3 种类型：宽容型（lenient）、爱发火型（angry）以及暴脾气型（jerk）。宽容型监督者的脾气其实也不好，每次只开除一个乐队成员（one_for_one）——就是那个出错的成员——直到它受够了，就会开除所有成员。另一方面，爱发火型监督者在每次出错时会开除多个乐队成员（rest_for_one），过一小段时间后，才会开除所有成员，放弃整个乐队。而暴脾气型监督者呢，即使乐队成员总体犯错并不多，但是只要有一个成员犯了一次错误，也会开除所有成员，放弃乐队。

```
-module(band_supervisou).
-behavior(supervisor).

-export([start_link/1]).
-export([init/1]).

start_link(Type) ->
    supervisor:start_link({local,?MODULE}, ?MODULE, Type).
```

```
%% 基于乐队监督者的情绪类型设定在解散乐队前所允许的犯错次数
%% 宽容型监督者对错误的容忍度要高于爱发火型的
%% 爱发火型监督者对于错误的容忍度要高于暴脾气型的监督者
init(lenient) ->
    init({one_for_one, 3, 60});
init(angry) ->
    init({rest_for_one, 2, 60});
init(jerk) ->
    init({one_for_all, 1, 60});
```

这里并没有列出 init 函数的完整定义，不过，这部分定义已经设定了每一种监督者的基调。宽容型监督者只会重启一个音乐人进程，并在 60 s 内出现第 4 次错误时终止。爱发火型监督者只能接受 2 次错误，而暴脾气型监督者的标准非常严格！

现在，我们来完成这个函数，实现乐队启动逻辑。

```
init({RestartStrategy, MaxRestart, MaxTime}) ->
    {ok, {{RestartStrategy, MaxRestart, MaxTime},
          [{singer,
           {musicians, start_link, [singer, good]},
           permanent, 1000, worker, [musicians]},
          {bass,
           {musicians, start_link, [bass, good]},
           temporary, 1000, worker, [musicians]},
          {drum,
           {musicians, start_link, [drum, bad]},
           transient, 1000, worker, [musicians]},
          {keytar,
           {musicians, start_link, [keytar, good]},
           transient, 1000, worker, [musicians]}
        ]}}.
```

乐队由 3 个技能不错的成员组成：主唱、贝斯手和键盘手。那个鼓手太糟糕了（让你抓狂）。乐队成员的 Restart 值不同(permanent、transient 以及 temporary)。主唱是 permanent，因此如果没有主唱，乐队就没法表演，即使当前这个主唱是自愿离开的。贝斯手是 temporary 的，因为即便没有贝斯手，乐队也能表演得不错（坦白讲，谁会在乎贝斯手呀？），其他乐队成员都是 transient 的，他们可以自愿离开，但是在出现错误时，仍需要有人取代他们。

有了上述代码，band_supervior 模块就完全可用了，现在来试一试：

```
3> c(band_supervisor).
{ok,band_supervisor}
4> band_supervisor:start_link(lenient).
Musician Carlos Terese, playing the singer entered the room
Musician Janet Terese, playing the bass entered the room
Musician Keesha Ramon, playing the drum entered the room
Musician Janet Ramon, playing the keytar entered the room
{ok,<0.623.0>}
Carlos Terese produced sound!
Janet Terese produced sound!
Keesha Ramon produced sound!
```

```
Janet Ramon produced sound!
Carlos Terese produced sound!
Keesha Ramon played a false note. Uh oh
Keesha Ramon sucks! kicked that member out of the band! (drum)
... <snip> ...
Musician Arnold Tennelli, playing the drum entered the room
Arnold Tennelli produced sound!
Carlos Terese produced sound!
Janet Terese produced sound!
Janet Ramon produced sound!
Arnold Tennelli played a false note. Uh oh
Arnold Tennelli sucks! kicked that member out of the band! (drum)
... <snip> ...
Musician Carlos Frizzle, playing the drum entered the room
... <snip for a few more firings> ...
Janet Jamal played a false note. Uh oh
Janet Jamal sucks! kicked that member out of the band! (drum)
The manager is mad and fired the whole band!
  Janet Ramon just got back to playing in the subway
The manager is mad and fired the whole band!
  Janet Terese just got back to playing in the subway
The manager is mad and fired the whole band!
  Carlos Terese just got back to playing in the subway
** exception error: shutdown
```

太神奇了！可以看到，一开始只有鼓手被开除了，过了一会儿，乐队中的每个成员都被开除了。它们都去地铁了！

你还可以尝试下其他类型的监督者，最终结局都是一样的。唯一不同之处在于重启策略。下面看一下爱发火型监督者是怎么做的：

```
5> band_supervisor:start_link(angry).
Musician Dorothy Frizzle, playing the singer entered the room
Musician Arnold Li, playing the bass entered the room
Musician Ralphie Perlstein, playing the drum entered the room
Musician Carlos Perlstein, playing the keytar entered the room
... <snip> ...
Ralphie Perlstein sucks! kicked that member out of the band! (drum)
... <snip> ...
The manager is mad and fired the whole band!
   Carlos Perlstein just got back to playing in the subway
```

当鼓手犯错时，爱发火型监督者会把鼓手和键盘手一起开除。不过和暴脾气型监督者的行为相比，这就不值一提了：

```
6> band_supervisor:start_link(jerk).
Musician Dorothy Franklin, playing the singer entered the room
Musician Wanda Tennelli, playing the bass entered the room
Musician Tim Perlstein, playing the drum entered the room
Musician Dorothy Frizzle, playing the keytar entered the room
... <snip> ...
Tim Perlstein played a false note. Uh oh
Tim Perlstein sucks! kicked that member out of the band! (drum)
```

```
The manager is mad and fired the whole band! Dorothy Franklin just got back to
playing in the subway
The manager is mad and fired the whole band! Wanda Tennelli just got back to
playing in the subway
The manager is mad and fired the whole band! Dorothy Frizzle just got back to
playing in the subway
```

关于静态重启策略的内容，就讲到这里。

17.4 动态监督

目前为止所介绍的监督类型都是静态的。我们先在源代码中明确列出要监督的所有子进程，之后才让它们运行起来。在真实应用中，绝大多数监督者都如此工作，一般用这种方式监督架构级组件。

另一方面，也会有一些监督者，它们监督的是一些不确定的工作者进程。这些工作者进程通常是按需创建的。可以考虑一下 Web 服务器的工作方式，每收到一个连接请求，就创建一个进程去处理它。在这种情况下，我们希望使用动态监督者去监督动态创建出来的所有进程。

17.4.1 动态使用标准监督者

每当使用 one_for_one、rest_for_one 或者 one_for_all 策略把工作者进程加入监督者中时，除了该进程的 pid 和其他一些信息外，还会向监督者持有的一个列表中增加子进程规格说明。在以后重启子进程或者执行其他任务时，会使用这份子进程规格说明。基于这种工作方式，相应的接口定义如下：

start_child(SupervisorNameOrPid, ChildSpec)

向列表中增加一个子进程规格说明，并且用该规格说明启动一个子进程。

terminate_child(SupervisorNameOrPid, ChildId)

终止或者强行杀死（brutal_kills）指定的子进程。子进程的规格说明仍然保留在监督者中。

restart_child(SupervisorNameOrPid, ChildId)

使用子进程规格说明重启子进程。

delete_child(SupervisorNameOrPid, ChildId)

删除指定 ChildId 所对应的子进程规格说明。

check_childspecs([ChildSpec])

检查一个子进程规格说明是否有效。在调用 start_child/2 函数前，可以用这个函数去测试一下规格说明的有效性。

count_children(SupervisorNameOrPid)

分类列举出该监督者下的所有子进程，包括活动进程个数、子进程规格说明个数、监督者类型的个数和工作者类型的个数。

which_children(SupervisorNameOrPid)

返回一个指定监督者下所有子进程信息的列表。

我们用乐队的例子演示一下这些函数，下面的显示中去掉了打印输出（动作要快，否则鼓手进程就死了！）。

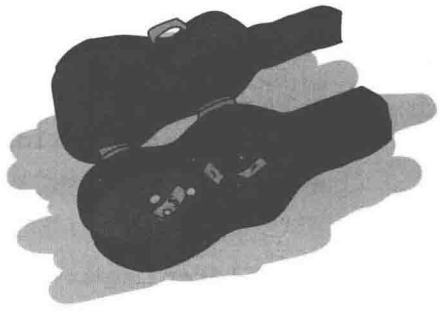
```
1> band_supervisor:start_link(lenient).
{ok,0.709.0>}
2> supervisor:which_children(band_supervisor).
[{keytar,<0.713.0>,worker,[musicians]},
{drum,<0.715.0>,worker,[musicians]},
{bass,<0.711.0>,worker,[musicians]},
{singer,<0.710.0>,worker,[musicians]}]
3> supervisor:terminate_child(band_supervisor, drum).
ok
4> supervisor:terminate_child(band_supervisor, singer).
ok
5> supervisor:restart_child(band_supervisor, singer).
{ok,<0.730.0>}
6> supervisor:count_children(band_supervisor).
[{specs,4},{active,3},{supervisors,0},{workers,4}]
7> supervisor:delete_child(band_supervisor, drum).
ok
8> supervisor:restart_child(band_supervisor, drum).
{error,not_found}
9> supervisor:count_children(band_supervisor).
[{specs,3},{active,3},{supervisors,0},{workers,3}]
```

通过演示可以看出，当子进程不多时，这些函数很适用于各种动态性要求（启动、终止等）。不过，由于内部使用的是列表，因此当需要快速访问大量子进程时，并不是很适用。

此时所需要的是 simple_one_for_one。

17.4.2　使用 simple_one_for_one 监督者

使用 simple_one_for_one 策略的监督者把所有子进程信息存放在一个字典中，这样可以快速查找，并且对监督者的所有子进程来说，只有一份子进程规格说明。这样做既节省内存，又节省时间——因为你不用亲自删除和保存任何子进程规格说明。



编写 simple_one_for_one 策略监督者的方式基本上和其他策略的监督者类似，只有一点不同：{M,F,A} 元组中的参数列表 A 并不是全部参数，完整的参数是把 supervisor: start_child(Sup, Args) 调用中的 Args 追加到 A 之后的新列表。没错，这里 supervisor: start_child/2 的含义改变了。因此，与原来的 supervisor:start_child(Sup, Spec) 调用 erlang: apply(M, F, A) 不同，现在的 supervisor:start_child(Sup, Args) 调用的是 erlang:apply(M, F, A++Args)。

只需在 band_supervisor 模块中增加如下的子句，就可以使用这个策略：

```
init(jamband) ->
    {ok, {{simple_one_for_one, 3, 60},
          [{jam_musician,
           {musicians, start_link, []},
           temporary, 1000, worker, [musicians]}
          ]}};
```

这里，我们让所有音乐人进程都是 temporary 的，并且监督者也是非常宽容的：

```
1> supervisor:start_child(band_supervisor, [djembe, good]).
Musician Janet Tennelli, playing the djembe entered the room
{ok,<0.690.0>}
2> supervisor:start_child(band_supervisor, [djembe, good]).
{error,{already_started,<0.690.0>}}
```

哎呀！之所以出错，是因为在调用 gen_server 时，我们给这个进程注册了名字 djembe。如果不去命名子进程或者为每个子进程取不同的名字，就不会有问题了。下面是使用另外一个名字 drum 的情况：

```
3> supervisor:start_child(band_supervisor, [drum, good]).
Musician Arnold Ramon, playing the drum entered the room
{ok,<0.696.0>}
3> supervisor:start_child(band_supervisor, [guitar, good]).
Musician Wanda Perlstein, playing the guitar entered the room
{ok,<0.698.0>}
4> supervisor:terminate_child(band_supervisou, djembe).
ok
```

看上去没问题了。

保持冷静

在 Erlang R14B03 以前的版本中，不能使用函数 supervisor:terminate_child(SupRef, Pid)终止子进程。调用这个函数会失败，并返回{error,simple_one_for_one}。要终止使用 simple_one_for_one 策略的监督者的子进程，最好的方法是：

```
5> musicians:stop(drum).
Arnold Ramon left the room (drum)
ok
```

如果希望代码能够向后兼容，就要把这点考虑进去。

这里给出一个一般性（有时可能不合适）的建议，仅当明确知道要监督的子进程数量不多并且（或者）不需要频繁地操控子进程，或者对性能要求不高的情况下，可以动态地使用标准监督者。对于其他需要动态监督的情况，尽可能使用 simple_one_for_one。

监督策略和子进程规格说明的内容就这些了。现在，你可能有些疑惑和想法。“到底如何使用这些内容构建出一个可以工作的应用？”如果真是这样，那么第 18 章的内容一定会让你满意。在第 18 章中，我们会用一棵矮小的监督树构建一个简单应用，演示一下真实世界中的实现方法。

第 18 章

构建应用

到目前为止，我们已经知道如何使用通用服务器、FSM、事件处理器以及监督者。然而，我们还没有学习如何把它们组装到一起，构建出完整的应用和工具。

Erlang 应用就是一组相关的代码和进程。OTP 应用特指用 OTP 行为实现进程，再用一种特定结构对它们进行打包。通过这个特定结构，虚拟机就知道如何进行初始化以及清理工作。在本章中，我们将使用 OTP 组件构建一个应用，由于不准备对它进行“打包”，因此这个应用还算不上一个完整的 OTP 应用。构建完整 OTP 应用所涉及的细节有些复杂，足以独立成章（第 19 章）。本章将关注于如何用 OTP 组件去实现一个应用，作为例子，我们会实现一个进程池。进程池背后的思想就是用一种通用的方式去管理和限制系统中运行的资源使用。

18.1 进程池

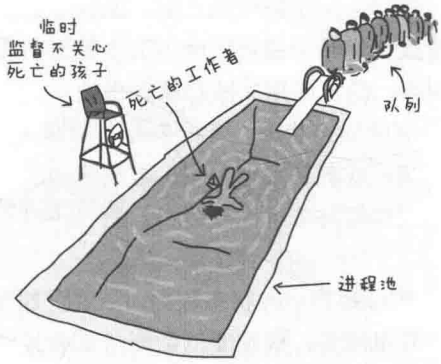
通过进程池，我们可以限制同时运行的进程数量。当运行的工作者进程数量达到上限时，进程池还可以把任务放到队列中。只要有进程资源被释放，排队的任务就能获得运行，否则任务只能阻塞，用户什么也做不了。

为什么要使用进程池呢？有如下几种可能。

- 限制一个服务器最多允许 N 个并发链接。
- 限制一个应用最多可以打开的文件个数。
- 通过给某些子系统分配多一些资源，而给另外的子系统少一些资源，从而让发布中不同的子系统具有不同的优先级。例如，可能希望处理客户请求的进程数多一些，而负责生成管理报表的进程数少一些。
- 通过对任务排队，在偶然突发的高负载情况下，可以让应用保持稳定。

本章中所构建的进程池应用，需要实现一些函数来满足如下需求。

- 启动和停止应用。
- 启动和停止一个特定的进程池（所有的进程池都属于这个进程池应用）。



- 在进程池中运行一个任务，如果池中的进程已满，要告诉调用者任务无法运行。
- 在进程池中运行一个任务，当进程池不满时立即运行；否则，让调用者进程保持等待，并将任务入队。一旦任务得以运行，就释放调用者进程。
- 以异步方式在进程池中运行一个任务，会尽量立即运行。如果池已满，让任务入队，并在以后运行。

我们会用这些需求来驱动程序设计。同时不要忘了，我们现在可以使用监督者，并且确实也很想用。尽管在系统的健壮性方面，监督者赋予了我们新的力量，但同时也限制了某些方面的灵活性。我们会在后面讨论一下利弊权衡。

18.1.1 洋葱理论

想清楚要监督什么以及该如何去监督，有助于在设计应用时用好监督者。第 17 章已经介绍过，可以有多种不同的监督策略，每种策略可以有不同的配置，分别针对可能产生不同错误的不同类型的代码。有太多种出错可能了！

如何应对状态丢失，无论对 Erlang 的初学者，还是具备 Erlang 编程经验的人来说，都是个棘手的问题。监督者杀死了进程，状态也丢失了，太倒霉了。为了解决这个问题，我们要识别出不同类型的状态。

- 静态状态：可以容易地从配置文件、另外一个进程或者重启应用的监督者中获取。
- 由可以重新计算出来的数据组成的动态状态：包括那些需要从初始形式经过转换才能得到的数据。
- 不能通过重新计算得到的动态状态：包括用户输入、现场数据、外部事件序列等。

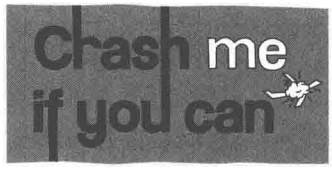
静态数据比较容易处理，绝大多数情况下，可以直接从监督者进程获取。这同样适用于动态可重新计算的数据。对于这种动态数据，可以先获取初始数据，然后在 `init/1` 函数中（或代码中其他任何地方）计算出来。最棘手的是那些不能重新计算出来的动态数据，只能祈祷它们不要丢失。有些情况下，可以把它们保存到数据库中，尽管这并不总是一个好的选择。

洋葱型系统的设计思想是，通过让不同类型的代码相互隔离，从而达成对不同类型的状态进行恰当保护的目的。换句话说，就是进程隔离。静态状态可以由监督者管理，因为，通常在系统

启动时，这类数据就确定了。每当某个子进程死亡时，监督者都会重启它，并把静态数据（这类数据一直是可用的）以某种形式传递给子进程。因为大多数监督者天然就是静态的，所以增加的每一层监督者都充当了一个屏障，防止应用失败和状态丢失。

对于可以通过重新计算得到的动态状态，有很多可选的解决方案。例如，可以基于监督者传递过来的静态数据把它构建出来，或者从其他进程、数据库、文本文件、当前环境等地方重新得到它。每次重启时，取回这些数据相对来说比较容易。所以，有了监督者来负责重启工作，就足以保证这类状态的可用性。

对于那些不能通过重新计算得到的动态状态数据，需要更加深思熟虑的解决方案。洋葱型设计方法的核心就体现在这里。其思想为，最重要的数据（或者那些丢失后麻烦最大的数据）必须是保护等级最高的。应用中根本不允许崩溃的部分称为应用的*错误内核*（error kernel）。



与代码中其他地方相比，你更想用 `try ... catch` 表达式加以保护的地方就是错误内核，因为这里的异常情况处理至关重要。这也是你不希望出错的地方。一定要对错误内核进行仔细的测试，特别是那种一旦出错就没有办法恢复的场景。你肯定不希望在处理到一半时丢失了客户的订单数据，对吧？

有些操作的安全性要比其他操作高。正因为如此，我们希望把关键数据尽可能地放置在最安全的内核中，把一切危险的东西都拒之门外。具体来说，这意味着所有相关的操作都要放在相同的监督树下，而不相关的操作应该放到其他树中。在同一棵监督树中，操作越容易出错，放置的位置就越要靠近树的叶子，那些不能崩溃的进程要放得离树根近一些。

基于这些原则构建出来的系统，软件中所有关联的部分会被划分到相同的树中，风险最高的操作位于树的最底层，这样就能降低核心进程死亡的风险，直到系统无法恰当地处理错误。我们后面要设计的进程池监督树，就是一个这样的例子。

18.1.2　进程池监督树

我们该如何组织这些进程池呢？关于此，有两种观点。一种观点是自底向上设计（先编写单个组件，然后按照要求把它们组装起来）。另外一个观点是自顶向下设计（先假想所有的组件已经就绪，做完高层设计后再去实际构建这些组件）。两种方法是等效的，具体使用哪种方法，取决于应用场景和个人风格。为了让每项工作都好理解，本例将采用自顶向下的方法。

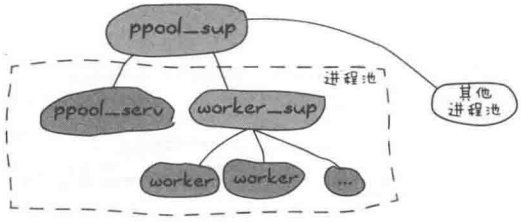
我们的监督树该是什么样子呢？嗯，有一个需求是能够把进程池应用作为一个整体启动，其中包括多个进程池，每个进程池包括多个可以处理排队任务的工作者进程。这个需求已经暗示了一些可能的设计约束。

每个进程池都需要一个 `gen_server`。这个服务器的任务是维护池中当前工作者进程的计数。方便起见，它还要保存任务队列。不过，谁来负责监督这些工作者进程呢？服务器自己吗？

让服务器来做这个事情还是比较有吸引力的。毕竟，服务器为了对工作者进程计数，需要跟踪这些进程，而自己去监督这些进程也是一个不错的跟踪方法。此外，服务器进程崩溃时，工作者进程必须要崩溃（否则，在服务器重启后，就不能跟踪这些任务了）。但是，这种做法也存在几个缺点：服务器职责太多、会变得更加脆弱，并且重复实现了现有的、经过良好测试的

模块功能。

确保所有工作者进程都得到恰当监督的一个好方法是用一个监督者来专门监督它们。

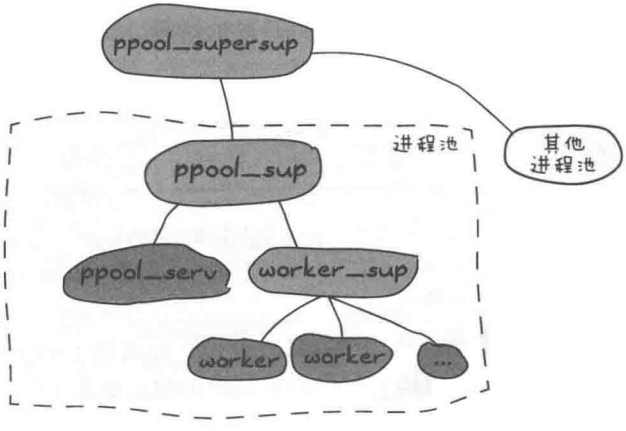


在本例中，有个单独的监督者来监督所有的进程池。进程池由一个池服务器和一个工作者进程监督者组成。池服务器知道工作者进程监督者的存在，会请求它增加新的工作者进程。由于增加子进程是一项非常动态的工作，并且子进程限制个数目前也未知，因此我们使用 `simple_one_for_one` 类型的监督者。

> **注意** 之所以选择 `ppool` 这个名字，是因为 Erlang 标准库中已经有了一个名为 `pool` 的模块。另外，`ppool` 也是个极好的同音词。

这种方法的好处在于，因为 `worker_sup` 监督者只需要监督一种类型的 OTP 工作者进程，所以能保证每个进程池里都只有一种定义明确的工作者进程，管理和重启策略也简单，很容易定义。这就是一个定义良好的错误内核范例。如果在 Web 连接管理中使用了一个套接字池，在日志文件处理中使用了另外一个服务器池，我们就可以保证日志文件处理中不正确代码或者权限错误所导致的问题，不会破坏应用中负责套接字的进程。不过，如果日志文件处理进程池崩溃的次数太多，它们会被终止，并且它们的监督者也会停止运行。哦，等等——它们的监督者也停止运行，这可是一个严重的问题！

因为所有进程池都在同一个监督者之下，所以如果某个特定的进程池或者服务器在一小段时间内重启次数过多，就会导致所有其他进程池都被终止。这个问题的解决方案是再增加一层监督者层。在同时存在多个进程池时，这种方案处理起来也更简单些。现在，我们的应用架构如下图所示：



这就更合理了。基于洋葱理论，所有进程池都是独立的，工作者进程之间互不依赖，并且 ppool_serv 服务器也和所有工作者进程隔离开了。这样的架构已经足够好了。一切似乎都已就绪。现在，可以开始实现了——当然，还是采用自顶向下的方法。

18.2　实现监督者

我们先来实现顶层监督者：ppool_supersup。它的工作是，根据需要启动一个进程池的监督者。它由几个函数构成：start_link/0，用来启动整个应用；stop/0，用来停止应用；start_pool/3，创建一个具体的进程池，以及 stop_pool/1，删除一个进程池。当然不能忘记 init/1，它是监督者行为唯一要求的回调函数。

```
-module(ppool_supersup).
-behavior(supervisor).
-export([start_link/0, stop/0, start_pool/3, stop_pool/1]).
-export([init/1]).

start_link() ->
    supervisor:start_link({local, ppool}, ?MODULE, []).
```

这里，我们把进程池的顶层监督者命名为 ppool（这解释了{local, Name}的用法，一种 OTP 在节点上命名 gen_*进程的惯用方法，还有另外一种用于分布式注册的形式）。这是因为我们知道每个 Erlang 节点上只会有一个 ppool，可以为它命名而不担心会发生冲突。很幸运，还可以用这个名字停止所有的进程池，像这样：

```
%% 从技术上讲，不太容易杀死一个监督者
%% 我们采取暴力手段！
stop() ->
    case whereis(ppool) of
        P when is_pid(P) ->
            exit(P, kill);
        _ -> ok
    end.
```

正如代码注释所描述的那样，无法优雅地终止一个监督者进程。OTP 框架为所有的监督者提供了一套定义良好的终止程序，但是不适用于我们目前的场景。我们会在第 19 章中介绍具体的做法，现在，最好的方法就是强行杀死这个监督者。

顶层监督者具体做什么呢？嗯，它只需在内存中保存所有的进程池，并监督它们。在本例中，它是一个没有孩子的监督者。

```
init([]) ->
    MaxRestart = 6,
    MaxTime = 3600,
    {ok, {{one_for_one, MaxRestart, MaxTime}, []}}.
```

现在，我们来考虑如何启动单个进程池的监督者并把它们加到 ppool 中。根据初始需求，可以确定需要两个参数：进程池可以容纳的工作者进程的个数，以及工作者进程监督者启动工作

者进程所需要的{M,F,A}元组。完整起见，还增加了一个名字参数。在启动单个进程池的监督者时，会把下面这个 ChildSpec 传给进程池的顶层监督者。

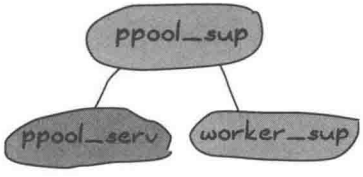
```
start_pool(Name, Limit, MFA) ->
    ChildSpec = {Name,
                 {ppool_sup, start_link, [Name, Limit, MFA]},
                 permanent, 10500, supervisor, [ppool_sup]},
    supervisor:start_child(ppool, ChildSpec).
```

从上面的代码可以看出，每个进程池的监督者都是永久（permanent）类型的，并且具有启动需要的参数（请注意一下此处是如何把程序员提供的数据放置到静态数据中的）。进程池的名称既作为参数传递给了监督者，同时也用作了子进程规格说明的标识符。最大终止时间被设置为 10 500。这个值的选取没有什么简单方法——只要保证这个值足够大，让所有子进程都有时间停止即可。可以先根据需要大致设置一个值，然后进行测试、调整，使之和当前应用匹配。如果实在不知道该用什么值，可以试试 infinity 选项。

要停止一个进程池，可以让 ppool 超级监督者（supersup!）杀死和进程池名字匹配的子进程。

```
stop_pool(Name) ->
    supervisor:terminate_child(ppool, Name),
    supervisor:delete_child(ppool, Name).
```

之所以可以这样做，是因为子进程规格说明的标识就是进程池的名字。太棒了！现在可以集中精力来编写进程池的直接监督者了！ppool_sup 管理一个进程池服务器和一个工作者进程监督者：



从上图中，你看出什么奇怪的事了么？ppool_serv 进程应该能够联系到 worker_sup 进程。如果它们是由同一个监督者同时启动的，除非通过 supervisor:which_children/1 玩一些小技巧（这种做法有时序问题，存在风险），或者给 ppool_serv 进程（这样用户就能通过名字来调用它）和工作者监督者进程都起个名字，否则根本没有办法让 ppool_serv 找到 worker_sup。然而，基于以下原因，我们不想给监督者起名字：

- 用户无需直接调用它们；
- 需要动态生成原子，这让我有些紧张；
- 有更好的办法。

解决方案就是让进程池服务器把工作者进程的监督者动态地增加到 ppool_sup 中。如果这听起来有点含糊，别担心——马上就会明白的。现在，我们只启动进程池服务器。

```
-module(ppool_sup).
```

```
-export([start_link/3, init/1]).
-behavior(supervisor).

start_link(Name, Limit, MFA) ->
    supervisor:start_link(?MODULE, {Name, Limit, MFA}).

init({Name, Limit, MFA}) ->
    MaxRestart = 1,
    MaxTime = 3600,
    {ok, {{one_for_all, MaxRestart, MaxTime},
          [{serv,
            {ppool_serv, start_link, [Name, Limit, self(), MFA]},
            permanent,
            5000, % 关闭时间
            worker,
            [ppool_serv]}]}}.
```

代码如上。注意，Name 和 self()，也就是监督者的 pid，被一起传给了服务器。这样，服务器就可以创建工作者进程监督者了，在创建时会传递 MFA 变量，从而让 simple_one_for_one 监督者知道要运行工作者进程所需的启动函数。

我们会在下一节中介绍服务器的具体工作内容。现在，先来编写应用中最后一个监督者 ppool_worker_sup，它负责管理监督所有的工作者进程。

```
-module(ppool_worker_sup).
-export([start_link/1, init/1]).
-behavior(supervisor).

start_link(MFA = {_,_,_}) ->
    supervisor:start_link(?MODULE, MFA).

init({M,F,A}) ->
    MaxRestart = 5,
    MaxTime = 3600,
    {ok, {{simple_one_for_one, MaxRestart, MaxTime},
          [{ppool_worker,
            {M,F,A},
            temporary, 5000, worker, [M]}]}}.
```

代码很简单。之所以选择 simple_one_for_one 类型，是因为工作者进程的数量会很大，创建的速度也会很快，并且我们也想限制工作者进程的类型。所有的工作者进程都是临时（temporary）的，因为使用了{M,F,A}元组来启动工作者进程，所以可以使用任意的 OTP 行为。

让工作者进程是临时的，有两方面考虑。首先，我们无法确定这些工作者进程在失败时是否需要重启，以及它们需要哪种重启策略。其次，在有些使用场景中，仅当工作者进程的创建者能够获取到工作者的 pid 时，进程池才有用。为了能够安全、简单地实现这个目标，我们不能只根据自己的要求去重启工作者进程，而是要跟踪工作进程的创建者并在重启时给它发送一条通知消息。这种做法非常复杂，而目的只是为了得到一个 pid。当然，你可以自由实现自己的 ppool_worker_sup，它不返回工作者进程的 pid，但会重启它们。这种设计本身并没有问题。

18.3 进程池服务器

进程池服务器是应用中最复杂的部分，这里汇集了所有精巧的业务逻辑。作为备忘，下面列出服务器必须支持的操作。

- 在进程池中运行一个任务，如果池中的进程个数已满，就要给出无法运行的指示。
- 在进程池中运行一个任务，当进程池尚有空间时立即运行；否则，让调用者进程保持等待，并将任务入队，直到任务可以运行。
- 以异步方式在进程池中运行一个任务，尽量立即运行；如果池中没有空间，就让任务入队，以后运行。

第一个操作由 run/2 函数实现，第二个由 sync_queue/2 函数实现，最后一个由 async_queue/2 函数实现。

```
-module(ppool_serv).
-behavior(gen_server).
-export([start/4, start_link/4, run/2, sync_queue/2, async_queue/2, stop/1]).
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         code_change/3, terminate/2]).

start(Name, Limit, Sup, MFA) when is_atom(Name), is_integer(Limit) ->
   gen_server:start({local, Name}, ?MODULE, {Limit, MFA, Sup}, []).

start_link(Name, Limit, Sup, MFA) when is_atom(Name), is_integer(Limit) ->
   gen_server:start_link({local, Name}, ?MODULE, {Limit, MFA, Sup}, []).

run(Name, Args) ->
   gen_server:call(Name, {run, Args}).

sync_queue(Name, Args) ->
```

```
    gen_server:call(Name, {sync, Args}, infinity).

async_queue(Name, Args) ->
    gen_server:cast(Name, {async, Args}).

stop(Name) ->
    gen_server:call(Name, stop).
```

对于 run/2、sync_queue/2 和 async_queue/2 函数，Args 会作为额外参数追加到传递给监督者的元组{M,F,A}中的列表 A 之后。注意，对于同步队列，我们把等待时间设置为 infinity。

之前说过，必须在服务器内部启动监督者进程。如果你是边看书边写代码的，那么为了可以继续跟着编写下去，你可能需要一个空的 gen_server 模板（或者使用线上的完整版本），因为我们将会以特性为基础来编写程序，而不是自顶向下。

我们先来编写创建监督者的代码。在 17.4 节中曾经介绍过，对那些只会增加少量子进程的情况，无需使用 simple_one_for_one 策略，使用 supervisor:start_ child/2 就可以了。首先，需要定义工作进程监督者的子进程规格说明。

```
%% 我们的监督者朋友是动态启动的!
-define(SPEC(MFA),
        {worker_sup,
         {ppool_worker_sup, start_link, [MFA]},
         permanent,
         10000,
         supervisor,
         [ppool_worker_sup]}).
```

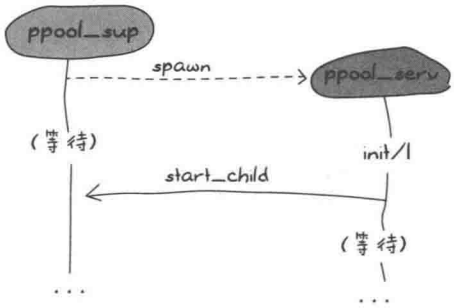
接下来，要定义服务器的内部状态。有几部分数据需要跟踪：可以运行的进程个数，监督者的 pid，以及存放所有任务的队列。为了知道工作者何时结束运行，从而可以从队列中再取出一项任务去运行，还需要在服务器中追踪每个工作者进程。完成这项工作最明智的方法是使用监控器，所以还要在状态记录中添加一个 refs 字段来保存所有的监控引用。

```
-record(state, {limit=0,
                sup,
                refs,
                queue=queue:new()}).
```

以上都就绪后，我们可以来实现 init 函数了。很自然就能想到如下的代码：

```
init({Limit, MFA, Sup}) ->
    {ok, Pid} = supervisor:start_child(Sup, ?SPEC(MFA)),
    {ok, #state{limit=Limit, refs=gb_sets:empty()}}.
```

然而，这段代码是错误的。在 gen_*行为中，启动该行为的进程会一直等到 init/1 函数返回才会恢复运行。这意味着，这里调用 supervisor:start_child/2 后，会产生如下死锁：



这两个进程会一直互相等待直到出现崩溃。解决这个问题最彻底的方法是使用一条特殊消息，服务器把它发送给自己，一旦 init 函数返回（进程池的监督者进程不再等待了），就立即在随后的 handle_info/2 函数中处理这条消息。

```
init({Limit, MFA, Sup}) ->
    %% 我们要在此处得到工作者进程监督者的 pid，
    %% 但是，唉，这需要调用监督者，而它正在等待我们的回应
    self() ! {start_worker_supervisor, Sup, MFA},
    {ok, #state{limit=Limit, refs=gb_sets:empty()}}.
```

代码干净多了。接下来，可以实现 handle_info/2 函数了，增加如下子句：

```
handle_info({start_worker_supervisor, Sup, MFA}, S = #state{}) ->
    {ok, Pid} = supervisor:start_child(Sup, ?SPEC(MFA)),
    {noreply, S#state{sup=Pid}};
handle_info(Msg, State) ->
    io:format("Unknown msg: ~p~n", [Msg]),
    {noreply, State}.
```

其中，第一条子句值得注意。我们匹配到给自己发送的消息（这应该也是收到的第一条消息），把工作者进程监督者增加到进程池监督者中，并在状态中记录下它的 pid，就是这样。现在，监督树已经完全初始化好了。你可以尝试编译所有代码，看看有没有什么错误。不过，目前还不能测试这个应用，因为还有相当一部分工作没有做呢。

> **注意** 如果你喜欢先构建整个应用，然后再运行的方式，也别担心。之所以采用这种方式，是因为想更清晰地展示整个应用的推导过程。虽然我大脑中有一个整体设计（就是我之前描述的），但是，还是会采用小步测试驱动的方式编写这个进程池应用，不时编写一些测试，并做一些重构来保持代码处于可工作状态。很少有 Erlang 程序员（和其他大多数语言的程序员一样）能够一次就编出产品质量的代码，当然，我本人也没有上面例子所展示的那么聪明。

解决了这个小问题，接下来看一下 run/2 函数。run/2 是一个同步调用，发送了消息 {run,Args}，消息处理如下：

```
handle_call({run, Args}, _From, S = #state{limit=N, sup=Sup, refs=R}) when N > 0 ->
    {ok, Pid} = supervisor:start_child(Sup, Args),
    Ref = erlang:monitor(process, Pid),
    {reply, {ok,Pid}, S#state{limit=N-1, refs=gb_sets:add(Ref,R)}};
```

```
handle_call({run, _Args}, _From, S=#state{limit=N}) when N =< 0 ->
    {reply, noalloc, S};
```

这个函数头很长，里面蕴含了很多管理逻辑。当进程池中尚有空间时（初始限制 N 是由程序员创建这个进程池时指定的），就接受并启动一个工作者进程。接下来，会监控这个进程，这样就能知道它什么时候完成任务，并把这些信息都保存到状态中，然后递减 N，处理完成。如果进程池没有空间了，简单回应 noalloc 即可。

sync_queue/2 调用的处理也非常类似：

```
handle_call({sync, Args}, _From, S = #state{limit=N, sup=Sup, refs=R}) when N > 0 ->
    {ok, Pid} = supervisor:start_child(Sup, Args),
    Ref = erlang:monitor(process, Pid),
    {reply, {ok,Pid}, S#state{limit=N-1, refs=gb_sets:add(Ref,R)}};
handle_call({sync, Args}, From, S = #state{queue=Q}) ->
    {noreply, S#state{queue=queue:in({From, Args}, Q)}};
```

第一个子句处理的是进程池尚有空间，可以容纳更多工作者进程的情况，和刚才 run/2 的消息处理逻辑一样。对于池中不能容纳更多工作者进程的情况，它们的处理是有差别的。并没有像 run/2 那样返回 noalloc，它没有立刻响应调用者，而是保存 From 信息，把它放入队列中，等到以后有空间容纳工作者进程时再运行它。稍后，我们会看到如何从队列中取出这些信息并处理它们，但是现在，我们先把 handle_call/3 函数的所有子句写完：

```
handle_call(stop, _From, State) ->
    {stop, normal, ok, State};
handle_call(_Msg, _From, State) ->
    {noreply, State}.
```

上面的子句处理了未知消息和 stop/1 调用。现在我们来实现 async_queue/2。因为 async_queue/2 根本不关心工作者进程何时运行，而且也无需回应，所以，我们用 cast 而不是 call 来实现它。它的处理逻辑和前面两个非常类似。

```
handle_cast({async, Args}, S=#state{limit=N, sup=Sup, refs=R}) when N > 0 ->
    {ok, Pid} = supervisor:start_child(Sup, Args),
    Ref = erlang:monitor(process, Pid),
    {noreply, S#state{limit=N-1, refs=gb_sets:add(Ref,R)}};
handle_cast({async, Args}, S=#state{limit=N, queue=Q}) when N =< 0 ->
    {noreply, S#state{queue=queue:in(Args,Q)}};
%% 下面这个无需解释！
handle_cast(_Msg, State) ->
    {noreply, State}.
```

同样地，除了不回应调用者，它和 run/2 消息处理最大的区别是：在进程池中没有多余空间时，任务信息被放入队列中。不过，这次并没有保存 From 信息，只保存了任务信息。这种情况下，状态信息中的 N 无需改变。

什么时候需要从队列中取出保存的任务来运行呢？嗯，我们对所有运行的工作者进程都设了监控器，并把它们的引用标识存放在 gb_sets 中。当一个工作者进程死亡时，我们会收到通知。就从这里入手吧。

```
    handle_info({'DOWN', Ref, process, _Pid, _}, S = #state{limit=L, sup=Sup, refs=Refs})
->
        io:format("received down msg~n"),
        case gb_sets:is_element(Ref, Refs) of
            true ->
                handle_down_worker(Ref, S);
            false -> %% 不处理这种情况
                {noreply, S}
        end;
    handle_info({start_worker_supervisor, Sup, MFA}, S = #state{}) ->
        ...
    handle_info(Msg, State) ->
        ...
```

在上面的代码片段中，我们要确认'DOWN'消息来自工作者进程。如果不是（这也挺奇怪的），不处理就行了。如果确实是由工作者进程发来的，就调用 handle_down_worker/2 函数：

```
    handle_down_worker(Ref, S = #state{limit=L, sup=Sup, refs=Refs}) ->
        case queue:out(S#state.queue) of
            {{value, {From, Args}}, Q} ->
                {ok, Pid} = supervisor:start_child(Sup, Args),
                NewRef = erlang:monitor(process, Pid),
                NewRefs = gb_sets:insert(NewRef, gb_sets:delete(Ref,Refs)),
                gen_server:reply(From, {ok, Pid}),
                {noreply, S#state{refs=NewRefs, queue=Q}};
            {{value, Args}, Q} ->
                {ok, Pid} = supervisor:start_child(Sup, Args),
                NewRef = erlang:monitor(process, Pid),
                NewRefs = gb_sets:insert(NewRef, gb_sets:delete(Ref,Refs)),
                {noreply, S#state{refs=NewRefs, queue=Q}};
            {empty, _} ->
                {noreply, S#state{limit=L+1, refs=gb_sets:delete(Ref,Refs)}}
        end.
```

这个函数非常复杂。因为工作者进程运行结束了，我们可以从队列中寻找下一个任务来运行。具体的做法是，从队列中取出一个元素，并对取出的结果进行检查。如果队列中至少还有一个元素，那结果肯定是{{value, Item}, NewQueue}这种形式。如果队列是空的，返回值就是{empty,SameQueue}。另外，如果所得到的值是{From, Args}形式，则意味着它来自 sync_queue/2，否则，就来自 async_queue/2。

队列中尚有任务的两种场景处理逻辑基本一致：向工作进程监控者中增加一个新工作者，老进程的监控引用被移除，并被新的替换。唯一的区别在于，对于同步调用的情况，会手工给调用者发送一个回应，对于异步的情况，则不发送回应。如果队列为空，就什么都不用做，只是将工作者进程上限增加 1。

最后，把标准的 OTP 回调函数补齐就行了：

```
    code_change(_OldVsn, State, _Extra) ->
        {ok, State}.
```

```
terminate(_Reason, _State) ->
    ok.
```

完成了——我们的进程池可以投入使用了！不过，它还不够友好。需要调用的函数散落在代码的各处。有些在 ppool_supersup 中，有些在 ppool_serv 中。另外，模块的名字也莫名其妙得很长。为了让进程池更好用，请在应用目录中创建如下 API 模块（就是将调用提取出来）：

```
%%% 进程池 API 模块
-module(ppool).
-export([start_link/0, stop/0, start_pool/3,
         run/2, sync_queue/2, async_queue/2, stop_pool/1]).

start_link() ->
    ppool_supersup:start_link().

stop() ->
    ppool_supersup:stop().

start_pool(Name, Limit, {M,F,A}) ->
    ppool_supersup:start_pool(Name, Limit, {M,F,A}).

stop_pool(Name) ->
    ppool_supersup:stop_pool(Name).

run(Name, Args) ->
    ppool_serv:run(Name, Args).

async_queue(Name, Args) ->
    ppool_serv:async_queue(Name, Args).

sync_queue(Name, Args) ->
    ppool_serv:sync_queue(Name, Args).
```

至此，进程池应用就真正写完了。

进程池的限制

你一定注意到了，我们的进程池应用中没有限制队列中可以存放的任务个数。在有些情况下，一个真实的服务器应用为了防止内存消耗过多而崩溃，需要设定可以入队的任务个数上限。尽管当调用者个数固定，且只调用 run/2 和 sync_queue/2 时，可以规避这个问题（如果所有的内容生产者都阻塞着等待进程池释放空间，那么在源头上就不会产生过多的内容）。

给队列长度增加上限的工作就留作练习吧，不过别怕，实现它还是比较简单的。要给服务器提供的所有函数都增加一个新的长度限制参数，服务器会在入队前检查这个限制。

另外，为了控制系统的负载，有时还会把调用改成同步的，从而在离负载源更近的地方施加限制。在系统因为生产者快于消费者而疲于应付时，同步调用可以阻塞住那些输入的请求。和毫无限制的做法相比，这种方法通常会让系统具有更好的响应性。

18.4 实现工作者

读到现在，你会发现我总是在撒谎！事实上，这个进程池还不能用。目前还没有实现工作者。是我忘记了。说起来真是惭愧，大家都知道，在编写并发应用的那一章中（第 13 章），我们还实现过一个漂亮的任务备忘录呢。显然，那个备忘录对我来说还是不够用，所以这个例子中，我们将实现一个更唠叨（nagger）的版本。

大致说来，在这个版本中，会为每个任务创建一个工作者，这个工作者会向我们发送重复消息，唠叨个不停，直到达到给定的最后期限。

这个唠叨版本需要考虑以下一些事情。

- 多长时间唠叨一次。
- 消息要发送的地址（pid）。
- 发送到进程邮箱中的唠叨消息，其中包括唠叨者进程自己的 pid，这样收到消息的进程能够用这个 pid 调用它。
- 一个停止函数，表明任务结束，唠叨者可以停止唠叨了。

开始实现：

```
%% 示例模块，唠叨者版本的任务备忘录，
%% 因为之前那个版本不够好
-module(ppool_nagger).
-behavior(gen_server).
-export([start_link/4, stop/1]).
-export([init/1, handle_call/3, handle_cast/2,
         handle_info/2, code_change/3, terminate/2]).

start_link(Task, Delay, Max, SendTo) ->
   gen_server:start_link(?MODULE, {Task, Delay, Max, SendTo} , []).

stop(Pid) ->
   gen_server:call(Pid, stop).
```

是的，我们再次使用了 gen_server 行为。你会发现大家总是在使用这个行为——甚至在不适用时！不要忘了，我们的进程池能够接受任何 OTP 兼容的进程，不只是 gen_server。

```
init({Task, Delay, Max, SendTo}) ->
   {ok, {Task, Delay, Max, SendTo}, Delay}.
```

初始化函数接收了一些基本参数，并把它们作为状态直接返回。

- Task 是发送的提醒消息。
- Delay 是两次消息发送之间的时间间隔。
- Max 是最大消息发送次数。
- SendTo 是消息接收方的 pid 或者名字。

注意，Delay 是返回值元组中的第三个元素，这表明 Delay 毫秒后会有一个超时消息发到 handle_info/2 函数中。

有了前面定义好的 API，剩下的代码就比较容易了。

```
%%% OTP 回调函数
handle_call(stop, _From, State) ->
    {stop, normal, ok, State};
handle_call(_Msg, _From, State) ->
    {noreply, State}.

handle_cast(_Msg, State) ->
    {noreply, State}.

handle_info(timeout, {Task, Delay, Max, SendTo}) ->
    SendTo ! {self(), Task},
    if  Max =:= infinity ->
        {noreply, {Task, Delay, Max, SendTo}, Delay};
       Max =< 1 ->
        {stop, normal, {Task, Delay, 0, SendTo}};
       Max > 1 ->
        {noreply, {Task, Delay, Max-1, SendTo}, Delay}
    end.
%% 不要使用下面的 handle_info 子句：
%% 如果这个子句执行了，计时器会被取消，进程也就基本成为了僵尸
%% 此时，崩溃是最好的选择。
%% handle_info(_Msg, State) ->
%%  {noreply, State}.

code_change(_OldVsn, State, _Extra) ->
    {ok, State}.

terminate(_Reason, _State) -> ok.
```

代码中唯一复杂的部分是 handle_info/2 函数。在第 14 章介绍 gen_server 时提到过，每当超时一到（在本例中，就是 Delay 毫秒之后），就会给进程发送一条 timeout 消息。基于此，就知道收到了多少条唠叨消息，是否需要继续发送。

工作者代码编写完毕了，现在真的可以运行进程池了！

18.5　运行进程池

我们来运行进程池应用。编译所有文件，并启动进程池的顶层监督者：

```
$ erlc *.erl
$ erl
... <snip> ...
1> ppool:start_link().
{ok,<0.33.0>}
```

此后，我们可以启动一个唠叨者版本的进程池，并试验一下它的各种功能：

```
2> ppool:start_pool(nagger, 2, {ppool_nagger, start_link, []}).
{ok,<0.35.0>}
3> ppool:run(nagger, ["finish the chapter!", 10000, 10, self()]).
{ok,<0.39.0>}
```

```
4> ppool:run(nagger, ["Watch a good movie", 10000, 10, self()]).
{ok,<0.41.0>}
5> flush().
Shell got {<0.39.0>,"finish the chapter!"}
Shell got {<0.39.0>,"finish the chapter!"}
ok
6> ppool:run(nagger, ["clean up a bit", 10000, 10, self()]).
noalloc
7> flush().
Shell got {<0.41.0>,"Watch a good movie"}
Shell got {<0.39.0>,"finish the chapter!"}
Shell got {<0.41.0>,"Watch a good movie"}
Shell got {<0.39.0>,"finish the chapter!"}
Shell got {<0.41.0>,"Watch a good movie"}
... <snip> ...
```

对于同步不入队的任务运行，一切正常。启动了进程池，增加了任务，消息也被发送到了正确的目的地。当我们试图运行的任务数超过允许上限时，被拒绝了。抱歉！无法完成清理任务。不过，其他工作者依然运行正常。

> **注意** ppool 是用 start_link/0 函数启动的。只要在 shell 中出了错，就会导致整个进程池应用被终止，只能重启。我们会在第 19 章解决这个问题。

现在来试验一下任务队列的功能（异步调用）：

```
8> ppool:async_queue(nagger, ["Pay the bills", 30000, 1, self()]).
ok
9> ppool:async_queue(nagger, ["Take a shower", 30000, 1, self()]).
ok
10> ppool:async_queue(nagger, ["Plant a tree", 30000, 1, self()]).
ok
❶ ... <wait a bit> ...
received down msg
received down msg
11> flush().
Shell got {<0.70.0>,"Pay the bills"}
Shell got {<0.72.0>,"Take a shower"}
❷ ... <wait some more> ...
received down msg
12> flush().
Shell got {<0.74.0>,"Plant a tree"}
ok
```

太棒了！入队功能工作正常。这里的打印并没有清楚地展示出所有的运行情况（为了达到最佳展示效果，要在❶和❷的地方等一段时间）。前两个“唠叨者”任务是立即运行的。接下来，因为达到了工作者进程的上限，第三个任务（“Plant a tree”任务）就需要放到队列中。当“Pay the bills”任务结束后，“Plant a tree”任务才得到调度，并在稍后发送消息。

同步队列的行为完全不同：

```
13> ppool:sync_queue(nagger, ["Pet a dog", 20000, 1, self()]).
{ok,<0.108.0>}
```

```
14> ppool:sync_queue(nagger, ["Make some noise", 20000, 1, self()]).
{ok,<0.110.0>}
15> ppool:sync_queue(nagger, ["Chase a tornado", 20000, 1, self()]).
received down msg
{ok,<0.112.0>}
received down msg
16> flush().
Shell got {<0.108.0>,"Pet a dog"}
Shell got {<0.110.0>,"Make some noise"}
ok
received down msg
17> flush().
Shell got {<0.112.0>,"Chase a tornado"}
ok
```

同样地，日志所展示的运行情况肯定不如你亲自去运行（极力推荐）体会得更清楚。基本的事件序列是：先把两个工作者增加到进程池中。在尝试去加第三个时，由于前两个仍在运行，因此，shell 被挂住了，直到 ppool_serv 进程（进程名是 nagger）收到一条工作者进程的退出消息（received down msg）。之后，第三个 sync_queue/2 调用才返回一个全新的工作者进程 pid。

现在，可以终止整个进程池应用：

```
18> ppool:stop_pool(nagger).
ok
19> ppool:stop().
** exception exit: killed
```

如果调用 ppool:stop()，那么所有进程池都会被终止，不过，你会收到很多出错消息。这是因为，我们强行杀死了 ppool_supersup 进程，只有以正确的方式终止时，它才会有序地关闭所有的进程池。在第 19 章中，将会介绍如何干净地终止进程池应用。

18.6　小结

本章中，我们完成了一个进程池应用，它的资源分配方式很简单。所有处理都并行、受限并且可以供其他进程调用。在监督者的帮助下，即便应用中的某个部分崩溃了，也可以在不破坏整体的前提下进行透明替换。有了这个进程池应用后，只用非常少的代码，就重写了之前备忘录应用的绝大部分代码。

我们考虑了单机上的故障隔离，也实现了并发处理。即便现在还不知道如何在 shell 中以更好的方式运行，它们依然是一些架构级构建块，足以用来编写坚实的服务器软件。

在第 19 章中，我们会展示如何把 ppool 应用打包成一个真正的 OTP 应用，可以方便地进行发布，并被其他产品使用。虽然我们并没有介绍 OTP 的所有高级特性，但是你现在的水平应该可以理解 OTP 和 Erlang 中大部分中阶和某些高阶内容了（至少那些非分布式的部分），这已经相当不错了！

第 19 章

构建 OTP 应用

现在，只需要一个简单的函数调用，就可以立即启动整个应用的监督树了，你可能会想，为什么还要想方设法再去包装它呢？还有什么能比一个函数调用还简单呢？

监督树背后的概念已经有点复杂了，对于系统的初次启动，我完全可以用一个脚本手动启动所有的树和子树。做完这些，就可以一整个下午都待在户外，寻找那些看上去像动物的云朵。

是的，完全可以这样做。这是一种可以接受的方法（尤其是关于云的那部分，因为最近全都是关于云计算的消息）。不过，与程序员和工程师所设计的大多数抽象类似，OTP 应用也是对众多专用系统进行一般化和规范化后的结果。

如果你自己编写了一组脚本和命令来启动自己的监督树，而你的开发者同事也编写了自己的版本，那么马上就会遇到大量的问题。然后，就会有人问，“每个人所用的启动方法都是一样的不就好了吗？他们的应用结构也都是一样的不就更好了？”

OTP 应用试图通过下面的手段来解决这类问题。

- 提供一种目录结构。
- 提供一种处理配置的方式。
- 提供一种定义环境变量和配置的方式。
- 提供多种依据依赖关系启动和停止应用的方式。
- 提供多种安全控制方法，用于冲突检测以及不停止应用的热升级。

所以除非不想要这些东西（以及其他一些辅助手段，如一致的结构和工具），否则你应该会对本章的内容感兴趣，因为在本章中，我们会介绍所有理解 OTP 应用所需的必要概念。

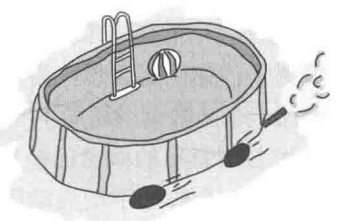
19.1 我还有辆车是一个游泳池

仅仅使用 OTP 组件不能保证创建出一个 OTP 应用，就像把人体的组织器官放在一起不能保

证可以构造出一个人一样，还可能构造出一个 Frankenstein 中的怪物。我们打算重用在第 18 章中编写的 ppool 应用，把它变成一个真正的 OTP 应用。

达成这个目标的第一步是把所有与 ppool 相关的文件复制到一个整洁的目录结构中：

```
ebin/
include/
priv/
src/
 - ppool.erl
 - ppool_sup.erl
 - ppool_supersup.erl
 - ppool_worker_sup.erl
 - ppool_serv.erl
 - ppool_nagger.erl
test/
 - ppool_tests.erl
```



现在，大部分的目录都是空的。在第 13 章中讲过，ebin/目录中存放编译过的文件，include/目录中存放公共的 Erlang 头文件（.hrl），priv/目录中存放可执行文件、其他程序以及应用运行需要的各种特殊文件，src/目录中存放 Erlang 源代码（以及私有.hrl 文件）。

我们新增加了一个 test/目录，这个目录中存放着我在第 18 章中编写的测试文件 ppool_tests.erl（如果你下载了相关的代码）。虽然测试很常见，但是也不必将它们作为应用的一部分进行发布。只有在开发代码或者向管理者证明自己时（测试通过了。我不知道应用为什么会杀人）才需要测试。根据情况，还会增加其他需要的目录。例如，doc/目录，当向应用中增加 EDoc 文档（一种标注 Erlang 代码的方法，可以用来生成文档）时，会创建这个目录。关于 EDoc 的更多信息，请参见 http://www.erlang.org/doc/apps/edoc/chapter.html。

有 4 个基本目录：ebin/、include/、priv/以及 src/。虽然在部署真实的 OTP 系统时只有 ebin/和 priv/会被输出，但是基本上每个 OTP 应用都会包含这 4 个目录。

19.2　应用资源文件

下一步做什么呢？嗯，接下来首先要增加一个应用文件。这个文件会告诉 Erlang VM 应用是什么、入口在哪里以及在哪里退出。这个文件位于 ebin/目录中，和所有编译过的模块在一起。

通常把应用文件命名为 yourapp.app（在本例中叫 ppool.app），其中包含一组 Erlang 的数据项，这些数据项以一种 VM 可以理解的方式定义了应用。（VM 非常不擅长猜测东西！）

> **注意**　有些人不喜欢把应用文件存放在 ebin/目录中，而是把一个名为 myapp.app.src 的文件放到 src/目录中。他们使用的构建系统会把这个文件复制到 ebin/目录中，甚至为了保持干净会生成一个应用文件。

应用文件的基本结构如下：

```
{application, ApplicationName, Properties}.
```

ApplicationName 是一个原子，Properties 是一个{Key, Value}元组列表，用来描述应用。OTP 会使用这些内容来弄清楚应用要做什么。这些元组都是可选的，不过，如果想对运行中的系统进行调试或者要保证不同应用之间的有序交互，它们非常有用。对于某些工具，这些元组也是必需的。现在，我们先来了解一下其中的一个子集，需要时再对其他的选项进行介绍。

{description, "Some description of your application"}

这个元组提供了一个简短的应用描述。默认值是一个空字符串。虽然这个元组是可选的，但是我建议始终提供一个应用描述，因为它可以让应用更好理解。

{vsn, "1.2.3"}

这个元组定义了应用的版本号。版本字符串可以采用任何格式。通常，遵守 Major.Minor.Patch 或者某种类似的格式是个不错的主意。当用工具来进行升级或者降级时，可以用这个字符串来标识应用的版本。

{modules, ModuleList}

这个元组包含着应用导入系统中的所有模块。一个模块最多属于一个应用，不能同时出现在两个应用的 app 文件中。系统和工具可以通过这个列表检查应用对模块的依赖关系，确保所有模块都就绪到位，并且和系统中其他已加载应用间没有冲突。如果使用了标准的 OTP 结构，并且使用了像 rebar 之类的 Erlang 构建工具，就会自动处理这些事情。

> **注意** Rebar 是 Erlang 社区常用的一种构建工具。它理解 OTP 应用背后的原则，可以像 Emakefile 那样编译文件。同时还可以根据需要从 git 和 mercurial 代码库中提取所依赖的应用。

{registered, AtomList}

这个元组包含所有被应用中的进程注册的名字。在试图把一组应用捆绑在一起时，OTP 可以通过这个信息知道是否存在名字冲突，不过，这完全基于相信开发人员提供了正确的信息。我们都知道，实际情况并非总是如此，因此不要盲目相信，一些测试工作是必不可少的。

{env, [{Key, Val}]}

这个元组是一个键/值列表，用来作为应用的配置信息。在运行时，可以通过调用 application:get_env(Key)或者 application:get_env(AppName, Key)获取这些信息。前者会从发起当前调用的进程所属应用的应用文件中进行查找。后者则从指定的应用中进行查找。这些值可以根据需要被改写（可以在启动时，或者通过调用 application:set_env(Application, Key,Value)）。由于这些值可能会被改写，因此应用资源文件

中的 env 部分通常作为应用的默认配置。这些默认值可以让用户只需最小化的配置即可使用应用。

总之，这个元组非常适合用来存储配置数据，这样就不用把配置数据以某种随意的格式放在一组配置文件中，也不用关心把它们存放到哪里。不过，人们总是倾向于开发自己的配置管理系统，因为并不是所有人都喜欢用 Erlang 语法编写配置文件。

{maxT, Milliseconds}

这个元组指应用的最长运行时间，之后应用会被关闭。这个字段很少使用。Milliseconds 的默认值为 infinity，因此通常无需关心这个字段。

{applications, AtomList}

这个元组是应用所依赖的其他应用的列表。在自己的应用被加载/启动前，Erlang 的应用管理系统会确保这些被依赖的应用先被加载或者启动。所有的应用都至少要依赖于 kernel 和 stdlib，不过如果自己的应用要依赖于 ppool 先被启动，那么就要把 ppool 加到这个列表中。把所依赖的应用增加到这个列表中非常重要，因为 OTP 会根据这个列表判断一个应用是否能被加载或者启动。如果不这样做，会给自己制造麻烦。

> **注意**　标准库和 VM 的 kernel 本身也都是应用，这意味着，虽然 Erlang 是构建 OTP 所使用的语言，但是它的运行环境在工作时也要依赖于 OTP。这是一种循环依赖。这也是为什么这门语言被官方命名为 Erlang/OTP 的一个原因。

{mod, {CallbackMod, Args}}

这个元组定义了应用的回调模块，这个模块会实现应用行为（详见 19.4 节）。它会通知 OTP，在启动应用时，应该调用 CallbackMod:start(normal, Args)。同时，在停止应用时，要调用 CallbackMod:stop(Args)。通常用应用名字来命名 CallbackMod 模块。

上面的内容基本上完全可以满足我们当前（以及以后绝大多数应用编写）的需要。

19.3　把 ppool 转换成 OTP 应用

现在，我们来实践一下学到的知识！我们把第 18 章中的那些 ppool 进程转换成一个基本的 OTP 应用。转换的第一步是把文件重新分布到正确的目录中：

```
ebin/
include/
priv/
src/
    - ppool.erl
    - ppool_serv.erl
    - ppool_sup.erl
    - ppool_supersup.erl
    - ppool_worker_sup.erl
test/
```

```
    - ppool_tests.erl
    - ppool_nagger.erl
```

我们把 ppool_nagger.erl 移到了 test 目录中。这样做有一个原因：这个文件只是一个演示示例，和应用本身没什么关系，但是又是测试需要的。当应用打包之后，我们可以运行它来确保一切仍旧正常，但是现在这个文件没什么用处。

需要增加一个 Emakefile 文件（文件名为 Emakefile，存放在应用的根目录中），用于以后源代码的编译和运行。

```
{"src/*", [debug_info, {i,"include/"}, {outdir, "ebin/"}]}.
{"test/*", [debug_info, {i,"include/"}, {outdir, "ebin/"}]}.
```

这个文件告诉编译器，要对 src/和 test/目录下的所有文件增加 debug_info 选项，到 include/目录中寻找包含文件（如果需要的话），然后把编译后的结果放到 ebin/目录中。

既然说到了 ebin/目录，我们就先把应用文件放到这个目录中。

```
{application, ppool,
 [{vsn, "1.0.0"},
  {modules, [ppool, ppool_serv, ppool_sup, ppool_supersup, ppool_worker_sup]},
  {registered, [ppool]},
  {mod, {ppool, []}}
 ]}.
```

这个文件中只包含了必要的字段，env、maxT 和 applications 等字段都没有使用。

现在，我们要更改回调模块（ppool）的工作方式。到底该如何做呢？

我们先来学习一下 application 行为的相关知识。

> **注意** 虽然所有应用都要依赖于 kernel 和 stdlib 应用，但是我并没有在文件中包含它们。ppool 仍然可以正常工作，这是因为 Erlang VM 在启动时会自动启动这两个应用。你可能觉得把它们加进来意图更明确一些，不过现在不用这么做。

19.4 application 行为

记住，行为（behavior）都是用来分隔通用代码和专用代码的。它们的思路是，专用的代码要放弃自己的执行流，把自己作为一组回调函数插入执行流中，供通用代码使用。简单来讲，行为负责处理那些比较无趣的部分，你来完成系统全局的关联。对于应用这个行为来说，这个通用部分非常复杂，远超过其他类型的行为。

VM 一开始工作，就会启动一个称为应用控制器的进程（进程的注册名是 application_controller）。它会启动所有其他应用，并位于其中大部分应用之上。事实上，可以把应用控制器看作是针对所有应用的监督者。我们会在下一节介绍可用的监督策略。

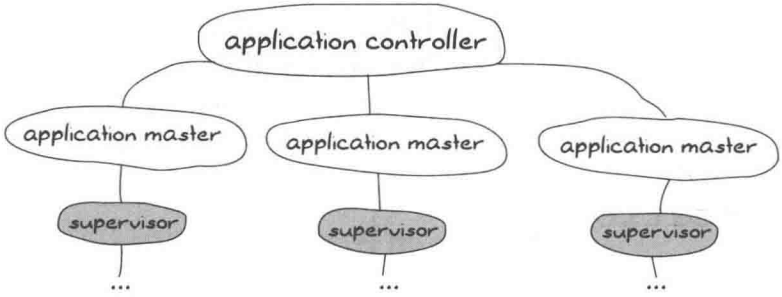
证实了规则的例外

从技术层面上讲，应用控制器并没有位于所有应用之上。kernel 应用是一个例外，它会启动一个名为 user 的进程。user 进程充当应用控制器的组领导者（group leader）角色，因此，kernel 应用需要一些特殊对待。我们无需关心这些内容，不过，准确起见，我觉得应该说明一下。

在 Erlang 中，I/O 系统依赖于一个名为组领导者的概念。组领导者代表着标准输入和输出，并被所有进程继承。有一个隐藏的 I/O 协议（http://erlang.org/doc/apps/stdlib/io_protocol.html）供进程在调用 I/O 函数时和组领导者通信使用。组领导者收到消息后负责把这些消息转发给可用的 I/O 通道，在本书中我们不用关心其中的细节。

当应用启动时，应用控制器（在 OTP 中，通常用 AC 指代）会启动一个应用主管理器（application master）。应用主管理器由两个进程组成，负责单个应用。这两个进程会对应用进行初始化，并充当应用的顶层监督者和应用控制器之间的中间人。OTP 是一个官僚机构，有很多中间管理层！我不会深入介绍其中的细节，因为大多数 Erlang 开发者根本不需要关心这些内容，并且相关的文档也非常少（代码就是文档）。只要知道应用主管理器有点像应用的保姆就行了（嗯，一个精神错乱的保姆）。它会照看子进程和孙子进程，当出现问题时，它就暴跳如雷，把整个家族树都终止掉。粗暴地杀死孩子是 Erlang 程序员之间的一个常见话题。

含有一组应用的 Erlang VM 如图所示：



到现在为止，我们介绍了行为的通用部分，那么专用部分呢？毕竟，这才是我们真正需要编写的部分。嗯，应用的回调模块只需要非常少的函数就可以工作：start/2 和 stop/1。

函数 start/2 的调用格式为 YourMod:start(Type, Args)。目前，Type 的值会被一直设定为 normal（另一个可能的值和分布式应用有关，会在第 27 章中介绍）。Args 的值来自于应用文件（就是{mod, {YourMod, Args}}元组中的 Args）。这个函数会完成应用所需的一切初始化工作，并返回应用顶层监督者的 pid，有两种形式：{ok, Pid}或者{ok, Pid, SomeState}。如果没有返回 SomeState，就把它默认设置为[]。

stop/1 函数会把 start/2 所返回的状态作为参数。当应用完成运行后会调用这个函数，它只用来做一些必要的清理工作。

介绍完了——一个庞大的通用部分加上一个微小的专用部分。幸亏有了这个行为，否则还得自己去频繁编写这个通用的部分。(如果想的话，可以看看源代码)，如果想对应用做更多控制，还有另外几个函数可供使用，不过我们目前不需要它们。也就是说，现在可以编写我们的 ppool 应用了！

19.5 从混乱到应用

我们已经有了应用文件，并且也大概知道了应用的工作原理。需要编写两个简单的回调函数。打开 ppool.erl 文件，定位到下面的代码行：

```
-export([start_link/0, stop/0, start_pool/3,
         run/2, sync_queue/2, async_queue/2, stop_pool/1]).

start_link() ->
    ppool_supersup:start_link().

stop() ->
    ppool_supersup:stop().
```

把上面这段代码更改成如下样子：

```
-behavior(application).
-export([start/2, stop/1, start_pool/3,
         run/2, sync_queue/2, async_queue/2, stop_pool/1]).

start(normal, _Args) ->
    ppool_supersup:start_link().

stop(_State) ->
    ok.
```

接下来，可以确认一下测试是否仍然正常。打开原来的 ppool_tests.erl 文件，把调用 ppool:start_link/0 替换成 application:start(ppool)，如下所示：

```
find_unique_name() ->
    application:start(ppool),
    Name = list_to_atom(lists:flatten(io_lib:format("~p",[now()]))),
    ?assertEqual(undefined, whereis(Name)),
    Name.
```

还得花点时间把 stop/0 从 ppool_supersup 中去除（也要从 export 属性中去除），因为 OTP 应用工具会自动停止应用。

现在，我们可以重新编译代码，并运行所有测试来确信一切正常（我们会在第 24 章中介绍 eunit 的用法）。

```
$ erl -make
Recompile: src/ppool_worker_sup
Recompile: src/ppool_supersup
... <snip> ...
$ erl -pa ebin/
```

```
... <snip> ...
1> make:all([load]).
Recompile: src/ppool_worker_sup
Recompile: src/ppool_supersup
Recompile: src/ppool_sup
Recompile: src/ppool_serv
Recompile: src/ppool
Recompile: test/ppool_tests
Recompile: test/ppool_nagger
up_to_date
2> eunit:test(ppool_tests).
  All 14 tests passed.
ok
```

由于测试代码中有几个地方使用了 `timer:sleep(X)` 进行同步，因此运行会花一点时间，不过测试应该像上面显示的那样都能通过。好消息：我们的应用很正常！

现在，就用这个了不起的新回调函数来展示一下 OTP 应用的神奇吧：

```
3> application:start(ppool).
ok
4> ppool:start_pool(nag, 2, {ppool_nagger, start_link, []}).
{ok,<0.142.0>}
5> ppool:run(nag, [make_ref(), 500, 10, self()]).
{ok,<0.146.0>}
6> ppool:run(nag, [make_ref(), 500, 10, self()]).
{ok,<0.148.0>}
7> ppool:run(nag, [make_ref(), 500, 10, self()]).
noalloc
9> flush().
Shell got {<0.146.0>,#Ref<0.0.0.625>}
Shell got {<0.148.0>,#Ref<0.0.0.632>}
... <snip> ...
received down msg
received down msg
```

上述示例中，神奇的命令是 `application:start(ppool)`。这个调用告诉应用控制器启动 ppool 应用。它会启动 ppool_supersup 监督者，此后，一切都和原来一样。调用 `application:which_applications()` 可以看到当前运行的所有应用：

```
10> application:which_applications().
[{ppool,[],"1.0.0"},
 {stdlib,"ERTS CXC 138 10","1.17.4"},
 {kernel,"ERTS CXC 138 10","2.14.4"}]
```

太惊讶了——ppool 竟然运行了（[]表明没有在.app 文件中提供应用描述）。正如前面提到的那样，所有的应用都依赖于 kernel 和 stdlib，这两个都处于运行状态。可以使用如下方式关闭 ppool 应用：

```
11> application:stop(ppool).
```

```
=INFO REPORT==== DD-MM-YYYY::23:14:50 ===
    application: ppool
    exited: stopped
    type: temporary
ok
```

完成了。你应该注意到，现在关闭应用的过程干净多了，并且还有一小段有用的信息报告，不像第 18 章中那样出现一堆混乱的** exception exit: killed。

> **注意** 有时，你会看到有人使用 MyApp:start(...)而不是 application:start(MyApp)来启动应用。虽然在测试时可以这样调用，但是这种做法会损失一个真正的应用本该具有的许多优势。应用不再是 VM 监督树的一部分，也不能访问自己的环境变量，在启动时不会对依赖进行检查等。若有可能，尽量使用 application:start/1。

等等！应用关闭时显示信息中的 temporary（临时）是什么意思？我们用 Erlang/OTP 编写程序的目的就是为了能让它们永远运行，不是只运行一会儿！VM 怎么敢这么说？秘密在于，我们可以给 application:start/1 提供不同的参数。根据参数的不同，在终止某个应用时，VM 会做出不同的反应。有时，VM 非常有爱，它会为了自己的孩子而死。有时，VM 会非常冷血、无情、现实，为了种族的延续，可以容忍大量的孩子死亡。

通过 application:start(AppName, temporary)启动的应用

如果应用正常结束，不需要什么特殊的处理，只有这个应用停止运行。

如果应用异常结束，会报告出错消息，这个应用会被终止，不再重启。

通过 application:start(AppName, transient)启动的应用

如果应用正常结束，不需要什么特殊的处理，只有这个应用停止运行。

如果应用异常结束，会报告出错消息，其他所有应用会被停止，VM 也会被关闭。

通过 application:start(AppName, permanent)启动的应用

如果应用正常结束，其他所有应用会被停止，VM 也会被关闭。

如果应用异常结束，结果一样：其他所有应用会被停止，VM 也会被关闭。

应用中的监督策略和以往相比，有些不同了。VM 不再会尝试挽救你了。此时，应该是发生了非常非常严重的错误，以至于这个错误传播到了 VM 的某个关键应用的监督树的最顶端——足以造成这个应用崩溃。当发生这种情况时，VM 对你的程序已经彻底失望。重复做同样的事情，还期望会有不同的结果出现显然是不明智的做法，因此 VM 明智地选择死亡，果断放弃。当然，真实的原因是有些东西被破坏了，需要修正，不过明白我的意思就行了。注意，所有的应用都可以通过 application:stop(AppName)被终止，此时不会影响其他的应用，就像出现了一次崩溃一样。

19.6 库应用

如果只想把普通模块包装到应用中，不想启动进程，也不需要应用回调模块时，该怎么做呢？

在抓狂了几分钟之后，发现只要把元组{mod, {Module, Args}}从应用文件中去掉即可。就这么简单。这种应用称为*库应用*（library application）。Erlang 的 stdlib（标准库）应用就是这类应用的一个例子。

如果你手边有 Erlang 的源代码包，可以找到 otp_src_<release>/lib/stdlib/src/stdlib.app.src，这个文件的内容如下：

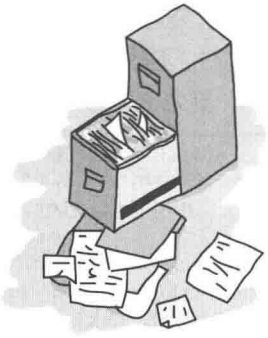
```
{application, stdlib,
 [{description, "ERTS CXC 138 10"},
  {vsn, "%VSN%"},
  {modules, [array,
             ...
             gen_event,
             gen_fsm,
             gen_server,
             io,
             ...
             lists,
             ...
             zip]},
{registered,[timer_server,rsh_starter,take_over_monitor,pool_master,
             dets]},
{applications, [kernel]},
{env, []}]}.
```

可以看出，这是一个非常规范的应用文件，不过没有包含回调模块的信息。再说一次，它是一个库应用。

接下来，我们再去学习一些更深入的应用知识，做些更复杂的应用，怎么样？

第 20 章

深入 OTP 应用

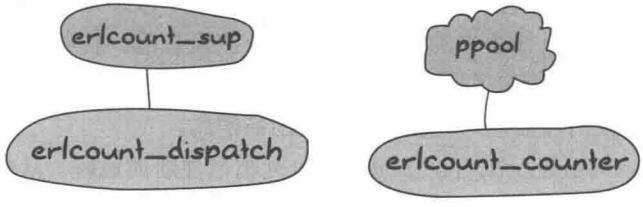


我们的 ppool 应用已经是一个有效的 OTP 应用了，现在，我们也理解了应用的含义。不过，如果能基于我们的进程池构建一个实用的应用不是更好吗？为了进一步加深对于应用的理解，我们会再编写一个应用。这个应用依赖于 ppool，不过，和第 19 章中的“唠叨者”相比，我们会在这个应用中让更多的工作自动化。

本章中要实现的应用的名字是 erlcount，它有一个简单的目标：递归地搜索某个目录，找到所有的 Erlang 文件（以.erl 结尾），然后用一个正则表达式对结果文件进行匹配，并对匹配上的字符串计数。然后，把计数结果累加起来形成最终结果，打印在屏幕上。

20.1 从 OTP 应用到真实的应用

erlcount 应用比较简单，在很大程度上依赖于我们的进程池去完成工作。其结构图如下：



在这幅图中，虽然 ppool 作为一个完整应用呈现，但是这只是为了表示 erlcount_worker 是这个进程池的工作者。它负责打开文件、运行正则表达式以及返回计数。erlcount_sup 进程/模块是应用的监督者。erlcount_dispatch 是一个服务器进程，负责浏览目录、让 ppool 去调度工作者进程并合并结果。我们还会增加一个 erlcount_lib 模块，这个模块中包含所有读取目录、合并数据之类的函数，这样，其他模块只需负责协调这些调用即可。最后一个是 erlcount

模块，它只用来充当应用的回调模块。

首先要创建出符合要求的目录结构。如果愿意的话，也可以在其中增加几个占位文件。

```
ebin/
- erlcount.app
include/
priv/
src/
 - erlcount.erl
 - erlcount_counter.erl
 - erlcount_dispatch.erl
 - erlcount_lib.erl
 - erlcount_sup.erl
test/
Emakefile
```

这和第 19 章中使用的结构没什么区别，甚至可以直接把旧的 Emakefile 文件复制过来使用。

我们可以很快编写完成应用的大部分内容。.app 文件、计数器、库以及监督者都比较简单。不过，要想应用可用的话，调度模块就需要完成一些复杂的任务。

20.1.1　应用文件

首先来编写应用文件，内容如下：

```
{application, erlcount,
 [{vsn, "1.0.0"},
 {modules, [erlcount, erlcount_sup, erlcount_lib,
            erlcount_dispatch, erlcount_counter]},
 {applications, [ppool]},
 {registered, [erlcount]},
 {mod, {erlcount, []}},
 {env,
  [{directory, "."},
   {regex, ["if\\s.+->", "case\\s.+\\sof"]},
   {max_files, 10}]}
]}.
```

这个应用文件比 ppool 要复杂一些。不过，其中有些字段还是一样的：这个应用的版本是 1.0.0，也列出了包含的模块。接下来的内容是 ppool 中没有的：应用依赖。正如之前介绍过的那样，applications 元组指定了所有需要在 erlcount 之前启动的应用列表。如果在启动该应用时，列表中的应用没有被启动，就会得到一条出错消息。

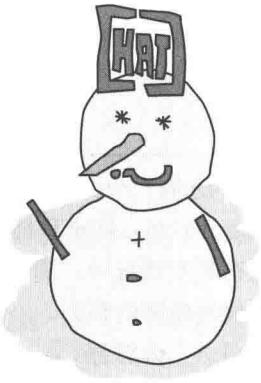
接下来，通过{registered, [erlcount]}列出已注册的进程。从技术上讲，erlcount 应用中没有任何进程需要一个名字。所有的进程都可以是匿名的。不过，我们知道 ppool 会用我们指定的名字注册 ppool_serv 进程，并且我们也知道会使用进程池，因此我们准备称之为 erlcount，并在这里做个备注。如果所有使用 ppool 的应用都这样做了，应该能够在以后检测到冲突。mod 元组和我们之前用过的类似。在 mod 元组中，定义了应用行为的回调模块。

最后一个新出现的是 env 元组。在这个元组中，可以以键/值的方式存储应用特定的配置变量。为了方便起见，这些变量保存在内存中，可以被应用的所有运行进程访问。一般来讲，可以

用它来替代应用的配置文件。

在本例中，我们定义了 3 个变量：

- directory 变量告诉应用从哪里寻找.erl 文件。如果我们在以 learn-you-some-erlang 为根目录（如果你下载了本书的代码包）启动的 Erlang 虚拟机中运行这个应用，那么"."指的就是这个目录。
- max_files 变量指定可同时打开的文件描述符个数。我们不想同时打开 10 000 个文件（如果真有那么多文件的话），因此这个变量的值和 ppool 中的最大工作者进程个数相等。
- regex 变量是最复杂的一个变量，它是一个列表，其中包含着对每个文件进行匹配的正则表达式，应用所统计的就是它们的匹配结果。



在此，我们不会深入学习 Perl 兼容的正则表达式（PCRE）的语法（如果有兴趣，可以查看 re 模块，其中包含了完整的细节），只会对本应用中使用的部分进行介绍。应用中使用的第一个正则表达式的意思是，“查找含有 if、if 后面是单空格符（\s，两个反斜杠是为了转义）并且以->结尾的字符串。此外，在 if 和->之间可以是任何内容（.+）”。第二个正则表达式的意思是，“查找含有 case、case 后面是单空格符（\s），并且以 of 结尾、of 前面是单空格符的字符串。在 case 和 of 之间可以是任何内容（.+）”。简单来讲，就是我们想知道在库代码中用了多少个 case ... of，以及多少个 if ... end。

保持冷静

用正则表达式来分析 Erlang 代码并不是最优选择。问题在于，很多情况会造成分析的结果不准确，包括文本中的字符串和与你正在寻找的技术模式（非代码模式）相匹配的注释。

要得到更准确的分析结果，需要直接使用 Erlang 去分析已被解析和展开的模块版本。虽然复杂一些，但是这种方法可以确保正确处理宏，将注释排除在外以及类似的事情。如果这是你希望的，可以去了解一下 erl_syntax 和 xref。

20.1.2 应用回调模块和监督者

有了应用文件后，就可以开始编写应用回调模块了。

```
-module(erlcount).
-behavior(application).
-export([start/2, stop/1]).

start(normal, _Args) ->
    erlcount_sup:start_link().

stop(_State) ->
    ok.
```

这个模块不复杂——其实，它只是启动了监督者进程。下面来配置监督者：

```
-module(erlcount_sup).
-behavior(supervisor).
-export([start_link/0, init/1]).

start_link() ->
    supervisor:start_link(?MODULE, []).

init([]) ->
    MaxRestart = 5,
    MaxTime = 100,
    {ok, {{one_for_one, MaxRestart, MaxTime},
          [{dispatch,
            {erlcount_dispatch, start_link, []},
            transient,
            60000,
            worker,
            [erlcount_dispatch]}]}}.
```

这是一个标准的监督者，从上面简短的配置说明中可以看出，它只负责监督 erlcount_dispatch。和进程终止相关的 MaxRestart、MaxTime 以及 60 s 这些值都是随机选择的，不过在真实情况中，需要根据需要慎重选择。这只是一个演示应用，这些值就不是那么重要了。作者保留偷懒的权力。

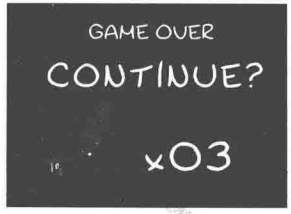
20.1.3　分派器

下一个进程和模块是分派器。分派器为完成工作需要满足如下几个复杂的需求。

- 当在目录中搜索以.erl 结尾的文件时，所有的目录应该只遍历一次，即使需要运行多个正则表达式。
- 一发现有满足条件的文件，就应该立即开始进行结果计数的调度。无需等获取全部文件列表后再进行统计。
- 对于每个正则表达式，要有一个独立的计数器，这样可以在最后比较结果。
- 无需等找到所有.erl 文件后才开始收集 erlcount_worker 工作者进程的结果。
- 可以同时运行多个 erlcount_worker。
- 在完成目录中文件搜索后，还应该可以继续收集工作进程返回的结果（尤其是文件很多且正则表达式复杂时）。

有两个关键点，现在就要考虑清楚：如何在递归遍历一个目录的同时能够从中返回搜索到的文件，从而开始调度，以及在遍历尚在运行时能够接收返回的匹配结果，且不导致混乱。

1. 使用 CPS 返回结果



乍看之下，能够在递归中间返回结果的最容易的方法是把它放到另外一个进程中。不过，这需要更改之前的结构，而更改的目的只是为了能够在监督树中加入另外一个进程，并让它们能够合作完成任务，这有点讨厌。事实上，有一个更简单的方法可以满足我们的要求：使用一种称为*延续传递风格*（continuation-passing style，CPS）的编程风格。

CPS 的基本思想是，把一个深度递归函数变成一步步的执行。每个执行步骤都会返回一个值（通常是累加器）和一个函数，然后通过调用这个函数可以继续向下执行。在本例中，函数有两个可能的返回值：

```
{continue, Name, NextFun}
done
```

当收到第一个返回值时，可以把 FileName 调度给 ppool，然后调用 NextFun 来继续搜索更多的文件。这个函数的实现在 erlcount_lib 中，如下：

```
-module(erlcount_lib).
-export([find_erl/1]).
-include_lib("kernel/include/file.hrl").

%% 查找所有以.erl 结尾的文件。
find_erl(Directory) ->
    find_erl(Directory, queue:new()).
```

啊，这里有点新东西!太让人惊喜了，我的心跳加速，血压升高。上面代码中的包含文件是由 file 模块提供的。它包含有一个记录（#file_info{}），其中的字段详细地说明了一个文件的细节信息，包括文件类型、大小、权限等。

我们的设计中使用了一个队列。为什么呢？嗯，目录中的文件数目超过一个是完全可能的。因此，当遍历到某个目录，其中包含了 15 个文件时，我们希望先处理第一个文件（如果它是一个目录，就打开它，在其中进行搜索，以此类推），然后再处理剩余的 14 个文件。为了做到这一点，只需把它们的名字保存在内存中，直到有时间处理它们为止。我们使用队列来完成这项任务，不过，如果确实不在乎文件处理顺序，那么使用栈或者任何其他的数据结构也是可以的。重点在于，在我们的算法中，这个队列充当了类似待完成工作列表这样的角色。

我们先来读取第一次调用传过来的第一个文件：

```
%%% 私有函数
%% 基于文件类型进行分派
find_erl(Name, Queue) ->
    {ok, F = #file_info{}} = file:read_file_info(Name),
    case F#file_info.type of
        directory -> handle_directory(Name, Queue);
        regular -> handle_regular_file(Name, Queue);
        _Other -> dequeue_and_run(Queue)
    end.
```

从这个函数中，可以看出几件事情。首先，我们只想处理常规文件和目录。对于每种情况，我们都写了一个函数专门来处理这些问题（handle_directory/2 和 handle_regular_file/2）。如果是其他类型的文件，会使用函数 dequeue_and_run/2 从队列里面取出之前放置的内容。现在，先来处理目录的情况，代码如下：

```
%% 打开目录，把其中的文件放入队列
handle_directory(Dir, Queue) ->
```

```
    case file:list_dir(Dir) of
        {ok, []} ->
            dequeue_and_run(Queue);
        {ok, Files} ->
            dequeue_and_run(enqueue_many(Dir, Files, Queue))
    end.
```

如果目录中没有任何文件，继续使用 dequeue_and_run/1 进行搜索。如果目录中有多个文件，就把它们放入队列，然后再继续搜索。函数 dequeue_and_run 会从文件名队列中取出一个元素。所取出的文件名会被用来调用 find_erl(Name, Queue)，然后就像刚开始一样继续运行。

```
%% 从队列中取出一个元素，以它为起点运行
dequeue_and_run(Queue) ->
    case queue:out(Queue) of
        {empty, _} -> done;
        {{value, File}, NewQueue} -> find_erl(File, NewQueue)
    end.
```

注意，如果队列为空（{empty, _}），这个函数会认为自己结束了（返回 done，CPS 函数选择的关键字），否则，会继续调用 find_erl。

另外一个需要考虑的函数是 enqueue_many/3。这个函数把在给定目录中找到的所有文件都放到队列中。实现如下：

```
%% 把一批文件加入队列
enqueue_many(Path, Files, Queue) ->
    F = fun(File, Q) -> queue:in(filename:join(Path,File), Q) end,
    lists:foldl(F, Queue, Files).
```

简单来讲，我们使用函数 filename:join/2 把目录路径和文件名合并在一起（这样就可以得到一个完整的路径）。然后，把这个完整的文件路径加入队列中。我们使用 fold 函数对给定目录中的所有文件重复这个相同的操作。所返回的新队列会被再次用来调用函数 find_erl/2，不过，这次队列中包含了所有新找到的待处理文件。

有点离题了。我们在哪里？噢，是的，我们在处理目录，现在已经完成了。接下来需要处理常规文件，并检查它们是否以.erl 结尾。

```
%% 检查文件是否以.erl 结尾
handle_regular_file(Name, Queue) ->
    case filename:extension(Name) of
        ".erl" ->
            {continue, Name, fun() -> dequeue_and_run(Queue) end};
        _NonErl ->
            dequeue_and_run(Queue)
    end.
```

可以看出，如果名字匹配（根据 filename:extension/1 返回结果），就返回延续。延续中包含了供调用者使用的 Name，并且把操作 dequeue_and_run/1 和需要后续处理的文件队列

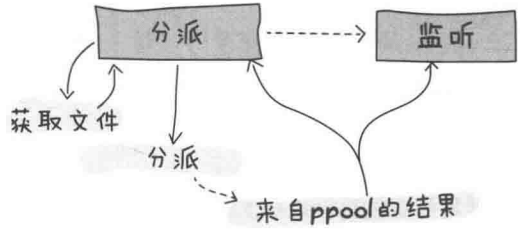
包装在一个匿名函数中。这样，调用者就可以调用这个匿名函数，继续执行，就像仍在递归调用中一样，并且仍能从递归的中间获取结果。如果文件名不以.erl 结尾，调用者对此类文件毫不关心，就继续处理队列中的其他文件。处理逻辑就是这样。

好哇，CPS 的内容讲完了。现在可以关注其他问题了。

2. 分派和接收

如何设计分派器，才能让它既可以进行分配，同时又可以接收消息呢？我们的建议是使用 FSM，毫无疑问，你会接受这个建议，因为我是本书的作者。

FSM 有两个状态。第一个是“分派”（dispatching）状态。当等待 find_erl 延续函数返回 done 时，会处于这个状态。在这个状态中，不用考虑计数结束的情况。这种情况只会出现在第二个状态，也是最后一个状态“监听”（listening）中。但是，我们始终都可以接收来自 ppool 的通知消息。



因此，必须要处理下面几种情况。

- 在分派状态中，收到有新的文件需要分配的异步事件。
- 在分派状态中，收到文件搜索完毕的异步事件。
- 在监听状态中，收到文件搜索完毕的异步事件。
- 当 ppool 工作者进程运行完正则表达式时，发送过来的全局事件。

下面，我们来逐步构建这个模块（gen_fsm）：

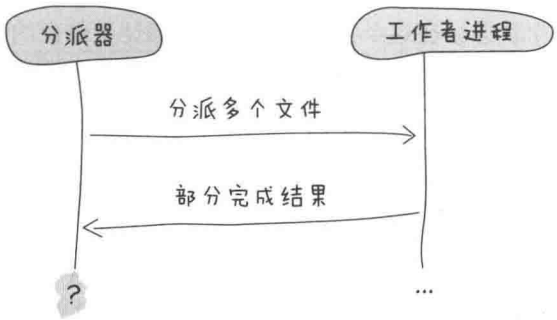
```
-module(erlcount_dispatch).
-behavior(gen_fsm).
-export([start_link/0, complete/4]).
-export([init/1, dispatching/2, listening/2, handle_event/3,
         handle_sync_event/4, handle_info/3, terminate/3, code_change/4]).

-define(POOL, erlcount).
```

这个模块的 API 由两个函数组成：一个供监督者使用（start_link/0），另一个供 ppool 的工作者进程使用（complete/4，很快会介绍）。其他的函数都是标准的 gen_fsm 回调函数，包括 listening/2 和 dispatching/2 这两个异步状态处理函数。其中还定义了一个?POOL 宏，用来指定 ppool 服务器的名字：erlcount。

那么，gen_fsm 中保存的状态数据应该是什么呢？因为处理是异步的，并且我们只会调用 ppool:run_async/2，而不是其他任何函数，所以根本无法知道所分派的文件是否处理完了。调用的时间线大体如下所示：

可以使用超时来解决这个问题，但是这个方法非常讨厌。超时太长或者太短？万一进程崩溃了呢？这么多的不确定性就像用柠檬制造的牙刷一样，只能用来娱乐。我们不会使用这种方法，而是使用另外一种方法：给每个工作者进程分配一个标识，我们可以跟踪这个标识，并把它和一个回应关联起来，有点像进入“完成任务的工作者”私人俱乐部时需要的安全密码。使用这种方法，我们可以对收到的消息进行一对一的匹配，并且可以知道是否做完了所有的工作。现在，我们知道状态数据中要包含的内容了：

```
-record(data, {regex=[], refs=[]}).
```



记录中第一个列表的内容是形如{RegularExpression,NumberOfOccurrences}的元组，第二个列表中包含了某种对消息的标识。可以使用任何东西充当这种标识，只要唯一就行。这两个 API 函数的实现如下：

```
%%% 公共 API
start_link() ->
    gen_fsm:start_link(?MODULE, [], []).

complete(Pid, Regex, Ref, Count) ->
    gen_fsm:send_all_state_event(Pid, {complete, Regex, Ref, Count}).
```

上面代码中定义了 complete/4 函数。很明显，工作者进程只需要发送回来 3 块数据：它们所运行的正则表达式、正则表达式关联的计数值以及前面提到的引用标识。很好——现在，可以去实现真正有趣的部分了！

```
init([]) ->
    {ok, Re} = application:get_env(regex),
    {ok, Dir} = application:get_env(directory),
    {ok, MaxFiles} = application:get_env(max_files),
    ppool:start_pool(?POOL, MaxFiles, {erlcount_counter, start_link, []}),
    case lists:all(fun valid_regex/1, Re) of
        true ->
            self() ! {start, Dir},
            {ok, dispatching, #data{regex=[{R,0} || R <- Re]}};
        false ->
            {stop, invalid_regex}
    end.
```

函数 init 首先从应用文件中加载所有需要的信息。加载完成后，以 erlcount_counter 为回调模块启动进程池。在开始实际调度工作者之前所做的最后一项工作是，确保所有的正则表达式都是有效的。这样做的原因很简单：如果不在此时检查，就需要在其他地方增加错误处理代码。这个地方很可能在 erlcount_counter 工作者中。如果加在那里，就必须要定义如何去处理当正则表达式无效时工作者进程崩溃的情况。相比而言，在应用启动时尽早崩溃要更容易处理一些。下面是 valid_regex/1 函数的定义：

```
valid_regex(Re) ->
    try re:run("", Re) of
        _ -> true
    catch
        error:badarg -> false
    end.
```

这个函数就是在空字符串上运行正则表达式。这样可以测试 re 模块的运行，并且运行时间也很快。如果正则表达式是有效的，就给自己发送一条消息{start, Directory}来让应用启动，并把状态设定为[{R,0} || R <- Re]。这个表达式会把列表从形式[a,b,c]变成形式[{a,0},{b,0},{c,0}]，也就是把每个正则表达式的计数值设定为 0。

我们要处理的第一条消息是 handle_info/2 函数中的{start, Dir}。记住，因为 Erlang 的行为基本上都是基于消息的，所以如果想触发一个函数调用，以自己的方式做些工作，就需要给自己发送消息。这虽然麻烦，但是可管理。

```
handle_info({start, Dir}, State, Data) ->
    gen_fsm:send_event(self(), erlcount_lib:find_erl(Dir)),
    {next_state, State, Data}.
```

这个函数把 erlcount_lib:find_erl(Dir) 的结果发送给自己。这个结果会在 dispatching 回调函数中收到，因为 FSM 的 init 函数把 State 的值设定为 dispatching。这段代码解决了我们的问题，同时也阐明了在整个 FSM 的实现中我们将使用的一般模式。因为 find_erl/1 函数是按照 CPS 风格编写的，所以我们可以给自己发送一个异步事件，并在每个正确的回调状态中处理这个事件。第一个延续的结果很可能是{continue, File, Fun}。由于在 init 函数中把分派状态设定为初始状态，因此就在这个状态中处理异步事件：

```
dispatching({continue, File, Continuation}, Data = #data{regex=Re, refs=Refs}) ->
    F = fun({Regex, _Count}, NewRefs) ->
        Ref = make_ref(),
        ppool:async_queue(?POOL, [self(), Ref, File, Regex]),
        [Ref|NewRefs]
    end,
    NewRefs = lists:foldl(F, Refs, Re),
    gen_fsm:send_event(self(), Continuation()),
    {next_state, dispatching, Data#data{refs = NewRefs}};
```

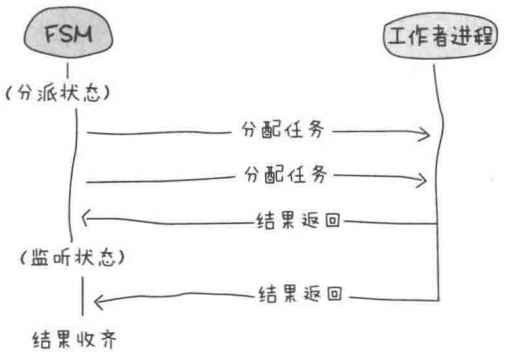
这段代码有点丑陋。对于每个正则表达式，都创建了一个唯一的引用标识，并把这个引用标识分派给一个 ppool 工作者进程，然后保存这个引用标识（为了知道工作者是否做完了）。这项

工作是通过 foldl 完成的，这样可以比较容易地收集所有新创建的引用标识。分派工作做完后，会再次调用延续以获取更多的结果，接着把新的引用标识存到状态中，并等待下一条消息。

接下来会收到哪一条消息呢？此时，有两个选择：要么没有任何工作者进程给我们返回结果（尽管我们尚未实现工作者代码），要么收到 done 消息，因为所有的文件都被找到了。先考虑第二种情况，把函数 dispatching/2 实现完整：

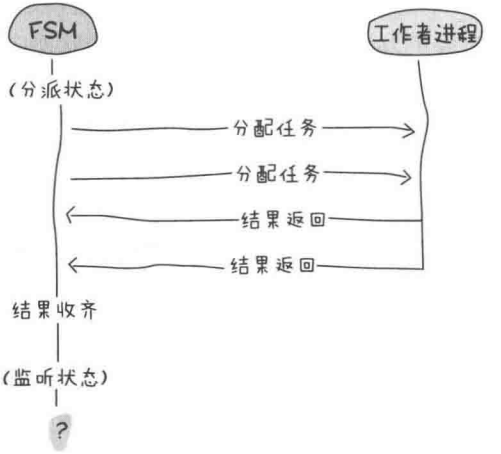
```
dispatching(done, Data) ->
    %% 这是一个特殊情况。在收到 done 时，我们不能假设消息还没有收全。
    %% 因此，我们没有等待外部事件，直接进入 listening/2
    listening(done, Data).
```

代码中的注释已经说得非常明白了。在任务调度期间，任务的返回结果既可以在 dispatching/2 中收到，也可以在 listening/2 中收到。可以用下图进行说明：



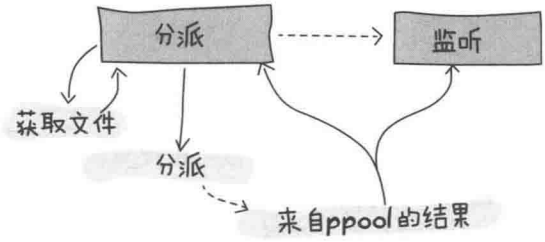
如果是上图中的情况，那么监听状态只需等待结果，然后宣布收到了所有结果。但是不要忘了，这里是 Erlang 的领地（Erland），我们工作在并行和异步的环境中！还可能出现下面的情况：

啊呀。我们的应用会永远停在那里，等待消息。这就是为何要手工调用 listening/2。通过这个调用强迫进行某种结果检测来确认已经收到了所有结果，这样就可以避免此时会发生的永久等待问题。代码实现如下：



```
listening(done, #data{regex=Re, refs=[]}) -> % 所有结果都收到了
    [io:format("Regex ~s has ~p results~n", [R,C]) || {R, C} <- Re],
    {stop, normal, done};
listening(done, Data) -> % 结果没有收全
    {next_state, listening, Data}.
```

如果 refs 为空，那么说明收到了所有的结果，可以把结果打印出来；否则，继续在监听状态下接收消息。再看一下 complete/4 函数和事件图：



结果消息是全局的，因为既可以在分派状态下收到它，也可以在监听状态下收到它。这个消息的处理实现如下：

```
handle_event({complete, Regex, Ref, Count}, State, Data = #data{regex=Re, refs=Refs})
->
    {Regex, OldCount} = lists:keyfind(Regex, 1, Re),
    NewRe = lists:keyreplace(Regex, 1, Re, {Regex, OldCount+Count}),
    NewData = Data#data{regex=NewRe, refs=Refs--[Ref]},
    case State of
        dispatching ->
            {next_state, dispatching, NewData};
        listening ->
            listening(done, NewData)
    end.
```

这个函数首先从 Re 列表中找出刚刚完成的正则表达式，列表中也包含着所有正则表达式的计数值。我们获取到这个计数值（OldCout），并使用 lists:keyreplace/4 把其替换成新值（OldCount+Count）。然后，用新的计数值更新 Data 记录，并删除工作者进程的 Ref，接着进入下一个状态。

在正常的 FSM 中，只要返回{next_state, State, NewData}就行了，但是在这个函数中，由于无法知道何时收完消息，因此必须再次手工调用 listening/2。这有些痛苦，但是，唉，这是一个必要的步骤。

分派器模块的代码就这些了。把缺少的行为回调函数补齐即可，具体如下：

```
handle_sync_event(Event, _From, State, Data) ->
    io:format("Unexpected event: ~p~n", [Event]),
    {next_state, State, Data}.

terminate(_Reason, _State, _Data) ->
    ok.
```

```
code_change(_OldVsn, State, Data, _Extra) ->
    {ok, State, Data}.
```

接下来，我们来实现 erlcount_counter 模块。你也许想先休息一会。强健的读者可以做几个俯卧撑放松一下，然后回来学习更多的内容。

20.1.4　计数模块

计数模块要比分派器模块简单一些。这个模块也需要实现一个行为来实现一些内容（本例中使用 gen_server），不过，要实现的内容非常少。这个模块只需做以下 3 件事情。

- 打开一个文件。
- 用一个正则表达式匹配文件内容，并对匹配结果计数。
- 返回结果。

对于第一项任务，file 模块中提供了大量函数可供使用。对于第三项任务，可以用我们定义过的 erlcount_dispatch:complete/4 来完成。对于第二项任务，可以使用 re 模块中的 run/2-3 函数，不过这个函数不能完全满足需要，原因如下：

```
1> re:run(<<"brutally kill your children (in Erlang)">>, "a").
{match,[{4,1}]}
2> re:run(<<"brutally kill your children (in Erlang)">>, "a",
2>        [global]).
{match,[[{4,1}],[{35,1}]]}
3> re:run(<<"brutally kill your children (in Erlang)">>, "a",
3>        [global, {capture, all, list}]).
{match,[["a"],["a"]]}
4> re:run(<<"brutally kill your children (in Erlang)">>, "child",
4>        [global, {capture, all, list}]).
{match,[["child"]]}
```

这个函数接收的参数确实是我们想要的（re:run(String, Pattern, Options)），不过它并没有返回正确的计数。我们需要在 erlcount_lib 中增加如下函数，然后就可以实现计数模块了：

```
regex_count(Re, Str) ->
    case re:run(Str, Re, [global]) of
        nomatch -> 0;
        {match, List} -> length(List)
    end.
```

这个函数只是计算出结果的数量，并把这个数量值返回。别忘了把它加到模块的 export 属性中。

现在，我们来实现工作者进程，如下：

```
-module(erlcount_counter).
-behavior(gen_server).
-export([start_link/4]).
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/3]).

-record(state, {dispatcher, ref, file, re}).
```

```
start_link(DispatcherPid, Ref, FileName, Regex) ->
    gen_server:start_link(?MODULE, [DispatcherPid, Ref, FileName, Regex], []).

init([DispatcherPid, Ref, FileName, Regex]) ->
    self() ! start,
    {ok, #state{dispatcher=DispatcherPid,
                ref = Ref,
                file = FileName,
                re = Regex}}.

handle_call(_Msg, _From, State) ->
    {noreply, State}.

handle_cast(_Msg, State) ->
    {noreply, State}.

handle_info(start, S = #state{re=Re, ref=Ref}) ->
    {ok, Bin} = file:read_file(S#state.file),
    Count = erlcount_lib:regex_count(Re, Bin),
    erlcount_dispatch:complete(S#state.dispatcher, Re, Ref, Count),
    {stop, normal, S}.

terminate(_Reason, _State) ->
    ok.

code_change(_OldVsn, State, _Extra) ->
    {ok, State}.
```

这段代码中有两个有趣的地方：一个是 `init/1` 回调函数，我们在其中给自己发了一条启动命令消息；另一个是 `handle_info/2` 子句，其中打开文件，并返回文件的二进制内容，这个内容会被传递给函数 `regex_count/2`，然后把结果通过 `complete/4` 发送回去。最后，工作者进程自行停止。其余的函数都是一些标准的 OTP 回调函数。

现在可以编译、运行整个应用了！

```
$ erl -make
Recompile: src/erlcount_sup
Recompile: src/erlcount_lib
Recompile: src/erlcount_dispatch
Recompile: src/erlcount_counter
Recompile: src/erlcount
Recompile: test/erlcount_tests
```

啊，很好。没有任何问题，打开香槟酒庆祝一下吧！

20.2 运行应用

应用有多种运行方法。确保你所在的目录中包含如下两个目录：

```
erlcount-1.0
ppool-1.0
```

现在，按照如下方式启动 Erlang：

```
$ erl -env ERL_LIBS "."
```

ERL_LIBS 是一个特殊的环境变量，用来指定 Erlang 从哪里寻找 OTP 应用。接下来，VM 就可以自动地在指定目录下寻找/ebin 目录。erl 可执行文件也可以使用-env NameOfVar Value 参数来快速地改写这个变量的值，上面的命令就是这种使用方式。ERL_LIBS 变量很有用，尤其是在安装库时，所以一定要记住它！

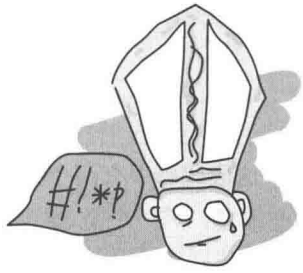
VM 启动后，可以测试一下是否所有模块都到位：

```
1> application:load(ppool).
ok
```

这个函数会尝试把所有能够找到的应用模块都加载到内存中。如果不调用这个函数，在启动应用时，也会自动进行加载，不过这个函数可以用来测试路径设置是否正确，很方便。下面启动应用：

```
2> application:start(ppool), application:start(erlcount).
ok
Regex if\s.+-> has 20 results
Regex case\s.+\sof has 26 results
```

根据目录中包含内容的不同，运行的结果也会不同。注意，如果目录中文件比较多，可能会耗费些时间。



如果想在应用中设置更多的变量时该怎么办呢？需要经常去更改应用文件吗？不！不需要！Erlang 对此提供了支持。例如，我们想看看 Erlang 开发者在源代码中表达愤怒的次数。

erl 可执行文件支持一组特殊形式的参数：-AppName Key1 Val1 Key2 Val2 ... KeyN ValN。对于上面的例子，我们可以对 R15B01 发布版中的 Erlang 源代码运行如下正则表达式：

```
$ erl -env ERL_LIBS "." -erlcount directory '"/home/ferd/otp_src_R15B01/lib/"' regex
'["shit","damn"]'
... <snip> ...
1> application:start(ppool), application:start(erlcount).
ok
Regex shit has 3 results
Regex damn has 1 results
2> q().
ok
```

注意，在这个例子中，所有的参数表达式都被包裹在单引号（'）之中。这是因为，我想让它们按照字面意义被 UNIX shell 接收。不同的 shell 可能有不同的规则。

也可以用更通用的表达式进行搜索，首字母可以大写，并把最大文件描述符设置得多一些：

```
$ erl -env ERL_LIBS "." -erlcount directory '"/home/ferd/otp_src_R15B01/lib/"'
regex '["[Ss]hit","[Dd]amn"]' max_files 50
... <snip> ...
1> application:start(ppool), application:start(erlcount).
```

```
ok
Regex [Ss]hit has 13 results
Regex [Dd]amn has 6 results
2> q().
ok
```

啊，OTP 程序员，是谁让你们如此愤怒（“因为使用 Erlang”这个答案是不能被接受的）？

这次运行会花费更长的时间，因为需要在数百个文件上做更复杂的检查。虽然现在一切正常，但是仍然有几个讨厌的问题存在。为什么总是要手工启动这两个应用？有没有更好的方法？

20.3 包含应用

包含应用（included application）是解决这个问题的一种方法。其基本思想是，把某个应用（本例中的 ppool）定义为另一个应用（本例中的 erlcount）的一部分。为了做到这一点，需要对这两个应用做不少的更改。

这种方法的大致做法是，先对应用做点修改，然后在其中增加一个称为*启动阶段*（start phases）的东西，这些工作需要遵循一个指定的协议，在 Eralng 文档中有详细描述。

现在，越来越不推荐使用包含应用，原因很简单：它们严重限制了代码重用。我们在 ppool 的架构上花了大量时间，目的就是为了让任何人都可以使用它、得到自己的进程池，并且自由地做他们想做的任何事。如果我们把它变成包含应用，那么它就不能再被包含到 VM 上的其他应用中了。同时，如果 erlcount 死了，ppool 也会被一起终止，使得想使用 ppool 的任何第三方应用都无法工作。

基于上述原因，虽然有些 Erlang 程序员仍旧喜欢包含应用，但是大部分 Erlang 程序员通常都不把包含应用放到自己的工具箱中。在后面几章中会看到，发布（release）基本上可以帮助我们完成同样的（甚至更多）功能，并且更加通用。

不过，在学习这些内容之前，还有另外一个关于应用的主题需要介绍。

20.4 复杂的终止

在有些情况下，在应用终止之前，还需要完成一些额外的步骤。此时，应用回调模块中的 stop/1 函数就不够用了，而且 stop/1 函数还是在应用已经被终止后调用的。想要在应用实际终止前做些清理工作的话，该怎么办呢？

方法很简单：在应用回调模块中增加一个 prep_stop(State) 函数即可。State 是 start/2 函数所返回的状态，prep_stop/1 的返回值会被传递给 stop/1。从技术上来讲，函数 prep_stop/1 位于 start/2 和 stop/1 之间，并在应用仍然活着但马上就要被关闭前执行。对于你自己的代码来说，当需要这个回调函数时，自然就知道了。我们的应用现在并不需要它。

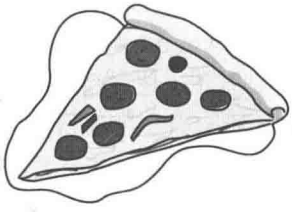
既然已经有了一些可以工作的基本应用，那么就可以开始考虑把这些应用打包发布了。

第 21 章

发布

进展到哪一步了？我们费了这么大劲，学习了这么多概念，但是到现在为止，连一次 Erlang 的可执行应用包都还没有发布过！也许，你和我一样，都认为要做不少工作才能让 Erlang 系统运转起来，尤其是和许多只需要运行一下编译器就搞定一切的语言相比。

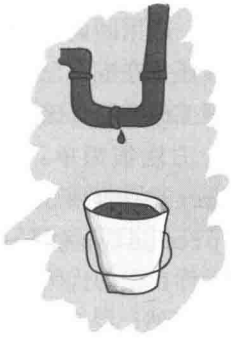
事实确实如此。我们可以编译文件、运行应用、检查依赖、处理崩溃等，但是如果不能轻易地把所编写的代码部署或者发布成一个可用的 Erlang 系统，那么这一切能有多大用处呢？如果披萨很棒，但是只有凉了之后才能交到你手上，又有什么用呢？（抱歉，此处不包括喜欢凉披萨的朋友。）



在这方面，OTP 团队也没有置之不理。OTP 中包含着一个应用打包系统，能以资源和依赖最小化的方式打包应用，OTP 发布（release）是其组成部分。在本章中，我们将介绍两种主要的发布处理方法：`systools` 和 Reltool。

21.1 修理漏水的管道

我们的第一个发布会重用前几章中的 `ppool` 和 `erlcount` 应用。不过，在这之前，需要做些零零散散的修改。



如果你一直读到现在，并且自己编写了代码，那么请把这两个应用复制到一个新目录 release/中，在本章后面的内容中，我会假设你已经做了这项工作。

21.1.1 终止 VM

`erlcount` 中最令人讨厌的是，它运行完毕后，VM 还保持运行，无事可做。我们希望大部分应用都能永远保持运行，但是本例并非如此。让 VM 一直运行，在开发阶段是有意义的，因为我们想在 shell 中做些实验，需要手动启动应用，但是现在不再需要这么做了。

因此，我们需要增加一条可以有序终止 Erlang 虚拟机的命令。增加这条命令的最佳位置是 erlcount_dispatch.erl 中的 terminate 函数，因为它会在我们得到结果后被调用。结束一切的最佳函数是 init:stop/0。这个函数非常复杂，它可以有序地终止我们的应用。它会清理掉文件描述符、套接字等。更改后的 terminate 函数如下所示：

```
terminate(_Reason, _State, _Data) ->
    init:stop().
```

代码的更改工作就这些。不过，还要做些其他工作。

21.1.2 更新应用文件

我们在前几章编写 app 文件时，只放入了应用运行所需的最少信息。为了让 Erlang 可以正确处理发布，需要增加另外几个字段。

首先，Erlang 发布构建工具要求我们对应用的描述稍微精确些。你看，虽然这些工具不能理解文档，但是对于代码，它们还是会本能的害怕，担心开发人员太过粗鲁，连一句关于应用是干什么的说明都不留下。因此，必须在文件 ppool.app 和 erlcount.app 中增加 description 元组。

在 ppool 中，增加下面的内容：

```
{description, "Run and enqueue different concurrent tasks"}
```

在 erlcount 中，增加下面的内容：

```
{description, "Run regular expressions on Erlang source files"}
```

现在，在查看系统时，就可以更清楚地知道应用到底是做什么的了。

细心的读者应该还记得，我在前面曾经说过，所用应用都依赖于 stdlib 和 kernel。不过，在这两个应用文件中并没有提到它们。我们把这两个应用增加到应用文件中。在 ppool 文件中增加如下元组：

```
{applications, [stdlib, kernel]}
```

同样，把这两个应用也增加到 erlcount 应用文件中，如下：

```
{applications, [stdlib, kernel, ppool]}.
```

保持冷静

在手工启动发布时，把 stdlib 和 kernel 应用增加到应用文件列表中好像没有实质的作用（甚至使用 systools 生成发布时，我们稍后会介绍），但是这样做是至关重要的。

在使用 Reltool 生成发布时（在本章中介绍的另外一个工具），为了能够让发布正常工作，并且能够恰当地关闭 VM，必须要包含这两个应用。我不是在开玩笑——真的是必需的。在开始写本章时我忘了这样做，结果浪费了一个晚上去查找问题，最后发现就是因为一开始没有写对。

你可能认为，因为绝大多数应用（除了极少数特别应用）都要依赖于它们，所以如果 Erlang 的发布系统可以隐式地加入这两个应用就更理想了。但是，唉，它们没有这样做。

21.1.3　编译应用

我们增加了终止命令，并且更新了应用文件。在构建发布之前，最后一个步骤是编译所有的应用文件。

逐个运行每个目录中的 Emakefile 文件（使用 `erl -make`）。否则，Erlang 工具并不会帮你调用，最后生成出来的发布中将不包含任何可执行代码。

21.2　使用 systools 构建发布

`systools` 应用是最简单的 Erlang 发布构建工具。它是 Erlang 发布的简易烤箱。要从 `systools` 烤箱中得到美味的发布，首先需要一本基础的食谱和一组原料。要成功制作一份 `erlcount` 应用（`erlcount 1.0.0`）的最小化发布，所需的原料如下：

- 所选的 Erlang 运行时系统（ERTS）；
- 标准库；
- `kernel` 库；
- `ppool` 应用，要永久存在；
- `erlcount` 应用。

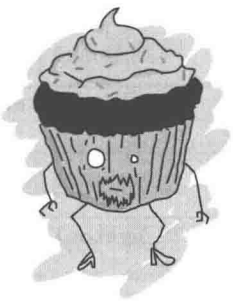
我说过我是个糟糕的厨师吗？我甚至连烙饼都做不好，不过至少我知道如何构建一个 OTP 发布。`systools` 制作 OTP 发布使用的原料列表在文件 erlcount-1.0.rel 中，直接存放在目录 release/下：

```
{release,
 {"erlcount", "1.0.0"},
 {erts, "5.9.1"},
 [{kernel, "2.15.1"},
  {stdlib, "1.18.1"},
  {ppool, "1.0.0", permanent},
  {erlcount, "1.0.0", transient}]}.
```

其中的内容和前面以文本形式描述的原料内容一样，不过，我们可以在这个文件中指定应用的启动方式（`temporary`、`transient` 或者 `permanent`）。还可以指定版本信息，这样就可以根据需要混合使用不同 Erlang 版本中的不同库。要看到发布中的所有版本信息，只需执行如下命令序列：

```
$ erl
Erlang R15B01 (erts-5.9.1) [source] [64-bit] [smp:4:4] [async-threads:0]
[hipe] [kernel-poll:false]

Eshell V5.9.1 (abort with ^G)
1> application:which_applications().
[{stdlib,"ERTS CXC 138 10","1.18.1"},
{kernel,"ERTS CXC 138 10","2.15.1"}]
```

从这个例子中可以看出，我运行的是 R15B01。紧挨着发布编号后面的是 ERTS 版本号（这里的版本是 5.9.1）。接着，在 shell 上调用 `application:which_applications()`，可以从



应用 kernel (2.15.1) 和 stdlib (1.18.1) 中看到两个需要的版本号。这里的版本号会随着 Erlang 版本的不同而不同。明确写出所需要的版本号是有好处的，这意味着，即使系统中安装了多个不同的 Erlang 版本，仍然可以使用应用运行所需要的老版本 stdlib。

你可能注意到了，我把发布命名为 erlcount，把其版本号设置为 1.0.0。这些信息与应用 ppool 和 erlcount 无关，这两个应用的版本号也都是 1.0.0，定义在其应用文件中。

现在，所有应用都编译过了，原料以及简易烤箱（绝妙的比喻）都准备好了。所需要的就是那个食谱了。

食谱会告诉我们几件事情：增加原料的顺序、如何混合原料、如何烹饪原料等。在应用文件的依赖列表中已经说明了原料增加顺序。systools 应用非常聪明，它可以从应用文件中找出运行的顺序。不过，我们需要接着处理其他指令。

21.2.1 创建启动文件

Erlang VM 在启动时，会从一个*启动文件*（boot file）中获取基本的配置信息。事实上，当在 shell 上启动 erl 可执行文件时，它隐式地使用一个默认启动文件调用 ERTS。这个启动文件会提供一些基本的指令，如“加载标准库”“加载 kernel 应用”“运行一个指定函数”等。这个启动文件是一个二进制文件，由*启动脚本*（http://www.erlang.org/doc/man/script. html）创建，启动脚本中包含了表示这些指令的元组。现在，我们就来写一个这样的启动脚本。

首先，编写如下内容：

```
{script, {Name, Vsn},
 [
  {progress, loading},
  {preLoaded, [Mod1, Mod2, ...]},
  {path, [Dir1,"$ROOT/Dir",...]}.
  {primLoad, [Mod1, Mod2, ...]},
  ...
```

我在开玩笑。没人会花时间做这项工作，我们也不会。从.rel 文件很容易就能生成启动脚本。从 release/目录启动 Erlang VM，并调用如下命令：

```
$ erl -env ERL_LIBS .
... <snip> ...
1> systools:make_script("erlcount-1.0", [local]).
ok
```

现在看看你的目录，会发现生成了几个新文件，其中包括 erlcount-1.0.script 和 erlcount-1.0.boot。命令中 local 选项的意思是，我们希望发布可以在任何环境运行，不仅仅是当前的安装环境。systools 应用还有许多其他选项（参见 http://www.erlang.org/doc/man/systools. html），不过由于 systools 没有 Reltool（我们会在下一节介绍）功能强大，因此我们不在此对其进行深入介绍。

现在，我们有了启动脚本，但是对于代码发布来说，还不够。

21.2.2　发布打包

回到 Erlang shell 中，运行如下命令：

```
2> systools:make_tar("erlcount-1.0", [{erts, "/usr/local/lib/erlang/"}]).
ok
```

如果使用的是 Windows 7 操作系统，就运行下面的命令：

```
2> systools:make_tar("erlcount-1.0",
2>                   [{erts, "C:/Program Files (x86)/erl5.9.1"}]).
ok
```

调用这个命令，systools 会寻找你的发布文件以及 ERTS（因为使用了 erts 选项）。如果不使用 erts 选项，那么生成的发布是不能自运行的，需要依赖于系统上已经安装的 Erlang 版本。

这个命令会创建一个名为 erlcount-1.0.tar.gz 的压缩文件。解开这个压缩文件，可以看到如下目录：

```
erts-5.9.1/
lib/
releases/
```

erts-5.9.1/目录中包含着 ERTS。lib/目录中包含着所有需要的应用，releases/目录中包含着启动文件以及与发布相关的其他文件。

进入这些文件的解压目录。在那里，我们构建一条调用 erl 的命令行。首先，我们指定了 erl 可执行文件以及启动文件（不带.boot 扩展名）的路径。在 Linux 中，命令行像下面这样：

```
$ ./erts-5.9.1/bin/erl -boot releases/1.0.0/start
```

在 Windows 7 中，命令完全一样（使用 Windows PowerShell）。

如果希望可以在计算机上的任意目录下运行这条命令，可以使用绝对路径。不过，现在先不要运行。运行起来也没啥用，因为当前目录中没有任何可供分析的源文件存在。如果你使用的是绝对路径，可以进入你想要分析的目录，并在那里执行这个调用。

保持冷静

不能够保证发布可以在任何系统上运行。如果你使用的是纯 Erlang 代码，没有使用 HiPE（Erlang 代码的本地编译器，会生成快一些的代码，尤其是对 CPU 密集型的应用）进行本地编译，那么你的代码是可移植的。问题是，你所发布的 ERTS 可能无法运行。因此，要么在大规模发布时针对不同平台创建多份二进制包，要么只发布 BEAM 文件，不携带 ERTS，让用户使用自己计算机上安装的 Erlang 系统运行这些代码。

erlcount 应用的实现中会把当前目录作为默认搜索起始点。不过，可以通过改写应用的 env 变量来配置搜索目录。我们在命令行上增加了-erlcount directory "'<path to the directory>'"选项。由于我们希望不要感觉到是在运行 Erlang，因此又增加了-noshell 参数。在我的计算机上，看起来是这样：

```
$ ./erts-5.9.1/bin/erl -boot releases/1.0.0/start -erlcount directory
'"/home/ferd/code/otp_src_R14B03/"' -noshell
Regex if\s.+-> has 3846 results
Regex case\s.+\sof has 55894 results
```

我运行 erlcount 所用的 Erlang 和 OTP 发布版本比较老。你可以在最近的版本上进行实验。使用绝对文件路径，命令行长了许多：

```
$ /home/ferd/code/learn-you-some-erlang/release/rel/erts-5.9.1/bin/erl -boot
/home/ferd/code/learn-you-some-erlang/release/rel/releases/1.0.0/start -noshell
```

无论从哪个路径运行命令，都会扫描这个路径。把这条命令封装在一个 shell 脚本或者批处理文件中，就一切就绪了。

21.3 使用 Reltool 构建发布

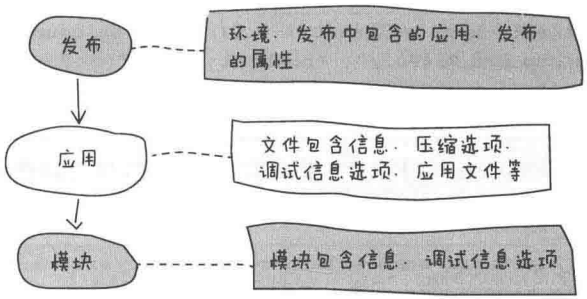
systools 有几个地方令人讨厌：在构建发布的过程中，我们基本上不能进行任何控制。手工指定启动文件路径也很痛苦。此外，生成的文件也有点大。整个发布文件大小超过 20 MB，如果打包的应用更多，就要再大一些。在这些方面，Reltool 做得要更好一些，功能也更加强大，不过作为交换，它要比 systools 复杂一些。

Reltool 工作时，使用的配置文件格式如下：

```
{sys, [
    {lib_dirs, ["/home/ferd/code/learn-you-some-erlang/release/"]},
    {rel, "erlcount", "1.0.0",
     [kernel,
      stdlib,
      {ppool, permanent},
      {erlcount, transient}
      % {LibraryApp, load} 该选项用于无需启动的应用
     ]},
    {boot_rel, "erlcount"},
    {relocatable, true},
    {profile, standalone},
    {app, ppool, [{vsn, "1.0.0"},
                  {app_file, all},
                  {debug_info, keep}]},
    {app, erlcount, [{vsn, "1.0.0"},
                     {incl_cond, include},
                     {app_file, strip},
                     {debug_info, strip}]}
]}.
```

瞧，这就是 Erlang 所谓的用户友好性！坦白说，没有办法以简单的方式介绍 Reltool。要让 Reltool 工作起来，这些选项都是必需的。这听起来有些令人难以理解，但却是合理的。

首先，Reltool 中的信息是分成不同层次的。第一层包含着和发布有关的信息。第二层是应用特定信息。第三层中包含着模块特定信息，在这一层，我们可以进行细粒度的控制。



每一层都有自己的选项。我们不会像百科全书一样介绍所有可能的选项，而是挑选一些关键的选项进行说明，然后会给出几个可能的配置示例。

我们首先介绍 `lib_dirs` 选项，有了它，就不必再做一些烦人的工作，比如必须要进入某个指定的目录中或者必须为 VM 设置正确的-env 参数。该选项的值是一个应用位置目录列表。使用这个选项，就不用设置-env ERL_LIBS *list:of:directories* 参数了，只要在配置文件中放入{lib_dirs, [*ListOfDirectories*]}，效果是一样的。

在 Reltool 配置文件中，另外一个重要选项是 `rel`。它是一个元组，和 `systools` 中使用的.rel 文件格式非常相似。在上面的演示文件中，它的内容是：

```
{rel, "erlcount", "1.0.0",
 [kernel,
  stdlib,
  {ppool, permanent},
  {erlcount, transient}
]},
```

其中定义了哪些应用必须被正确地启动。在这个元组之后，要增加一个类似下面形式的元组：

```
{boot_rel, "erlcount"}
```

这个元组告诉 Reltool，当某人运行了包含在这个发布中的 `erl` 可执行文件时，我们希望启动 `erlcount` 发布中包含的应用。有了 `lib_dirs`、`rel` 和 `boot_rel` 这 3 个选项，就可以得到一个有效的发布。

为了构建发布，我们把这 3 个元组放到一个 Reltool 可以解析的格式中，如下所示：

```
{sys, [
   {lib_dirs, ["/home/ferd/code/learn-you-some-erlang/release/"]},
   {rel, "erlcount", "1.0.0",
    [kernel,
     stdlib,
     {ppool, permanent},
     {erlcount, transient}
    ]},
  {boot_rel, "erlcount"}
]}.
```

我们只是把它们包装在一个{sys, [Options]}元组中，保存成文件 erlcount-1.0.config，

并放在 release/目录下。你可以把它放在任何地方（除了/dev/null，虽然写入速度极快！）。

接下来，打开一个 Erlang shell：

```
1> {ok, Conf} = file:consult("erlcount-1.0.config").
{ok,[{sys,[{lib_dirs,["/home/ferd/code/learn-you-some-erlang/release/"]},
    {rel,"erlcount","1.0.0",
        [kernel,stdlib,{ppool,permanent},{erlcount,transient}]},
    {boot_rel,"erlcount"}]}]}
2> {ok, Spec} = reltool:get_target_spec(Conf).
{ok,[{create_dir,"releases",
... <snip> ...
3> reltool:eval_target_spec(Spec, code:root_dir(), "rel").
ok
```

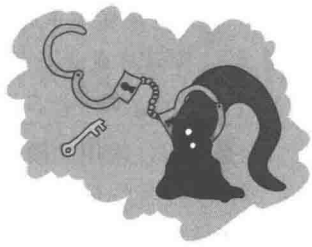
上面操作中的第一步是读取配置文件，并把读取的内容绑定到变量 `Conf`。接着调用 `reltool:get_target_spec(Conf)`。这个函数的执行要花些时间，它会返回大量信息，让人眼花缭乱。我们不用关心这些信息，只要把其保存在变量 `Spec` 中即可。

第三条命令以 `Spec` 为参数，命令 Reltool："我想让你根据我的发布规格说明，使用当前 Erlang 的安装路径，把它打包到 rel 目录"。这样就行了。查看 rel 目录，应该会看到其中有一组子目录。

现在不用关心这些子目录，直接调用如下命令：

```
$ ./bin/erl -noshell
Regex if\s.+-> has 0 results
Regex case\s.+\sof has 0 results
```

啊，这个命令简单了不少。只要保持同样的文件目录树，就可以把这些文件放在任何地方，并且可以在任何目录中去运行它们。



你发现有什么不同之处吗？我希望你发现了。我不再需要指定任何版本号了。在这个方面，Reltool 要比 `systools` 聪明一些。如果没有指定版本，它会在你提供的路径（`code:root_dir()` 返回的目录或者放在 `lib_dirs` 元组中的目录）中自动寻找最新的版本。

不过，如果我是一个怀旧派，不喜欢那些炫酷流行的最新应用该怎么办呢？我仍旧穿着迪斯科舞裤，并想使用老版本的 ERTS 和应用库（1977 年是我最具活动的一年！）。

很幸运，Reltool 可以生成使用 Erlang 老版本的发布。尊敬老人是 Erlang 工具的一个重要观念。

如果你的机器中装有 Erlang 的老版本，可以在配置文件中增加一个条目 `{erts, [{vsn, Version}]}`：

```
{sys, [
    {lib_dirs, ["/home/ferd/code/learn-you-some-erlang/release/"]},
    {erts, [{vsn, "5.8.3"}]},
    {rel, "erlcount", "1.0.0",
     [kernel,
```

```
        stdlib,
        {ppool, permanent},
        {erlcount, transient}
       ]},
      {boot_rel, "erlcount"}
]}.
```

现在，可以清空 rel/目录，把原先生成的使用新版本的发布删除掉。接着，再次执行这个丑陋的调用序列：

```
4> f(),
4> {ok, Conf} = file:consult("erlcount-1.0.config"),
4> {ok, Spec} = reltool:get_target_spec(Conf),
4> reltool:eval_target_spec(Spec, code:root_dir(), "rel").
ok
```

提醒一下：使用 f() 命令可以解除 shell 中变量的绑定。

现在，如果进入 rel 目录，并执行$ **./bin/erl** 命令，可以得到如下输出：

```
Erlang R14B02 (erts-5.8.3) [source] ...

Eshell V5.8.3 (abort with ^G)
1> Regex if\s.+-> has 0 results
Regex case\s.+\sof has 0 results
```

太好了！虽然机器上安装了新版本，但是这次运行的是版本 5.8（啊哈哈哈，还活着……）。当然这一切的前提是，你必须要先安装好 Erlang R14B02。

> **注意**　如果你看一下 rel/目录，会发现其中的内容和 systools 生成的有点相似。lib/目录中的内容有些不同，其中包含一组目录和.ez 文件。lib/下的目录中都有一个 include 目录，其中包含着使用该发布中的库进行开发时所需要的头文件，还可能会有一个 priv/目录，其中包含着像 C 驱动库或者应用运行所需的特殊文件之类的东西。.ez 文件是用 zip 压缩过的 BEAM 文件。Erlang VM 会在运行时解压它们，这样做只是为了少占些空间。

但是等等，应用的模块是怎么处理的呢？

嗯，现在我们从发布层面的配置转向应用层面的配置。发布层面的相关选项还有很多，但是我们现在只能顺着往下讲。在下一节中，我们再回过来介绍它们。

对于应用，我们可以增加更多的元组来指定其版本：

```
{app, AppName, [{vsn, Version}]}
```

对每个需要指定版本的应用，增加一个对应的元组。

Reltool 选项

现在，我们来介绍更多的选项，使用这些选项基本上可以做任何事情。我们可以指定发布中是否包含调试信息、是否对应用构成文件进行压缩、是否包含某些东西、在添加应用所依赖的其他应用或者模块时的严格程度等。此外，这些选项通常可以同时定义在发布层面和应用层面，因

此你可以先指定它们的默认值，之后可以用其他值替换默认值。

在此，我们会快速浏览一遍 Reltool 的选项。如果你觉得这部分内容比较复杂，可以直接跳到下一节，在下一节中会介绍几个 Reltool 的使用技巧。

1. 仅适用于发布层面的选项

下面这些 Reltool 选项只适用于发布层面。

{lib_dirs, [ListOfDirs]}

这个选项用来指定库的搜索目录。

{excl_lib, otp_root}

这个选项在 R15B02 中加入，使用这个选项，所生成的最终发布中将不包含标准 Erlang/OTP 安装路径中的任何应用。这样，发布中就只包含必要的库，只能基于系统中已安装的虚拟机启动。如果使用了这个选项，虚拟机的启动方式必须为`$ erl -boot_var RELTOOL_EXT_LIB path/to/releasedirectory/lib -boot path/to/boot/file`。这种方式可以让发布基于当前已安装的 Erlang/OTP 运行发布中包含的应用库。

{app, AppName, [AppOptions]}

这个选项用来指定应用层面的选项，应用层面的选项通常比发布层面的更具体一些。

{boot_rel, ReleaseName}

这个选项用来指定 `erl` 执行时启动的默认发布，也就是说，在调用 `erl` 时，就可以不用指定启动文件了。

{rel, Name, Vsn, [Apps]}

这个选项用来指定发布中要包含的应用。

{relocatable, true | false}

这个选项用来控制发布是可以在系统的任何目录中运行，还是只能在某个硬编码的目录中运行。默认情况下，这个选项被设置为 true，我倾向于保留这个默认值，除非有更好的理由。在需要更改时，自然就会有恰当的理由。

{profile, development | embedded | standalone}

这个选项提供了一种方法，可以基于发布的类型指定默认的`*_filters`（在下一节中介绍）。选项的默认值是 `development`。这个值会盲目地把大量应用文件和 ERTS 文件增加到发布中。值 `standalone` 会保守很多，`embedded` 则更保守一些，会丢弃更多的默认 ERTS 应用和二进制数据。

2. 发布层面和应用层面都适用的选项

下面这些是发布层面和应用层面都适用的选项，注意，对于如下所有选项，如果应用层面设定了选项的值，那么会覆盖其在发布层面设定的值。

{incl_sys_filters, [RegularExpressions]}和{excl_sys_filters,[Regular Expressions]}

这两个选项用来指定文件包含和排除过滤器，只有那些满足包含过滤器且不满足排除过

滤器的文件才会被包含进来，可以用这种方法包含或者排除特定文件。

{incl_app_filters, [RegularExpressions]}和{excl_app_filters, [Regular Expressions]}

这两个选项与 incl_sys_filters 和 excl_sys_filters 类似，不过它们针对的是应用文件。

{incl_archive_filters, [RegularExpressions]}和{excl_archive_filters, [RegularExpressions]}

这两个选项指定在.ez 压缩文件中哪些顶级目录必须被包含或者排除（下一节会详细介绍）。压缩文件中不包含的文件仍然可以包含在发布中，只是不被压缩。

{incl_cond, include | exclude | derived}

有些应用没有必要在 rel 元组中指定，这个选项决定这类应用的包含方式。值 include 的意思是，Reltool 会包含它能找到的几乎所有东西。值 derived 的意思是，Reltool 会去检测 rel 元组中的应用所使用的应用，并只包含这些应用。derived 是默认值。值 exclude 的意思是，默认不包含任何应用。当希望包含的东西最少时，通常可以在发布层面设置这个值，然后在应用层面根据需要改写这个值。

{mod_cond, all | app | ebin | derived | none}

这个选项控制模块的包含策略。值 none 的意思是，不会包含任何模块（这个值用处不大）。值 derived 的意思是，Reltool 会尝试找出被已经包含的模块使用的模块，并把它们包含进来。值 app 的意思是，Reltool 把所有应用文件中指定的以及通过 derived 方式得到的模块都包含进来。值 ebin 的意思是，把 ebin/目录中以及通过 derived 方式得到的模块都包含进来。all 是默认值，它的意思是等同于值 ebin 和 app 的结合使用。

{app_file, keep | strip | all}

这个选项决定了在包含一个应用时如何管理应用文件。值 keep 会保证发布中使用的应用文件和你自己编写的应用文件完全一样。它是默认值。如果选择值 strip，那么 Reltool 会尝试生成一个新的应用文件，其中会去掉不想要的模块（也就是那些被过滤器或者其他选项排除在外的模块）。值 all 会保留原始文件，但是会在其中增加所有满足包含条件的模块。使用选项值 all 有一个好处，如果没有任何可用的应用文件，它可以帮你生成出来。

3. 模块层面专用的选项

下面是 Reltool 中模块层面专用的选项：

{incl_cond, include | exclude | derived}

可以用这个选项覆盖掉在发布和应用层面设定的 mod_cond 选项的值。

4. 适用于所有层面的选项

下面这个选项适用于所有的层面。层次越低，优先级越高。

{debug_info, keep | strip}

如果在编译文件时，使用了 debug_info（推荐使用），那么这个选项可以用来决定是否保留调试信息。debug_info 对文件的反编译或者调试有益，不过会多占用一些空间。

5. 小结

是的，我们已经介绍了不少 Reltool 选项。尽管我并没有介绍所有的选项，但是这些内容足以作为一份不错的参考文档了。如果你想了解完整的内容，可以查阅官方文档：http://www.erlang.org/doc/man/reltool.html。

21.4 Reltool 技巧集

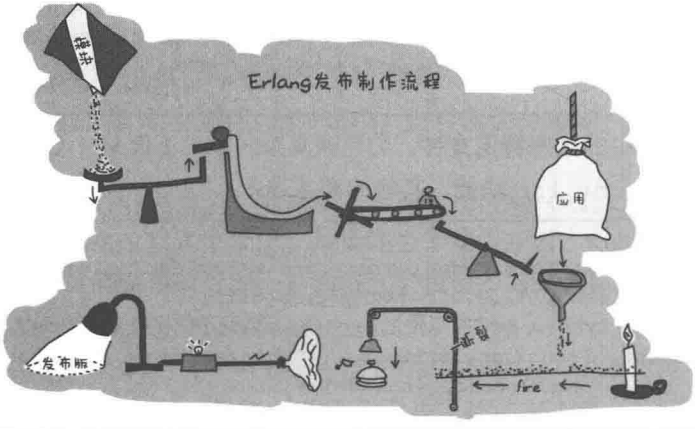
现在，我们来介绍一些编写.rel 文件的通用窍门和方法，可以达成某些特定的效果，例如，生成一个小一点的发布，或者在生产的发布中要包含足够的库，以满足开发所需。

1. 开发版本

在发布中放入特定项目开发所需的库并不困难，通常，使用默认配置就已经很好了。只要使用我们讲过的那些基本的选项，就应该可以得到不错的结果。

```
{sys, [
    {lib_dirs, ["/home/ferd/code/learn-you-some-erlang/release/"]},
    {rel, "erlcount", "1.0.0", [kernel, stdlib, ppool, erlcount]},
    {boot_rel, "erlcount"}
]}.
```

为了保证新构建的发布可以正常工作，Reltool 会尽量导入足够多的库。在某些情况下，你可能想把 OTP 团队创建的标准 VM 中的所有库和你自己的某些库都放到发布中。此时，可以通过如下配置完成这项工作：




```
{sys, [
    {lib_dirs, ["/home/ferd/code/learn-you-some-erlang/release/"]},
    {rel, "start_clean", "1.0.0", [kernel, stdlib]},
    {incl_cond, include},
    {debug_info, keep}
]}.
```

通过把选项 incl_cond 的值设置为 include，当前 ERTS 安装目录以及 lib_dirs 中找到的所有应用都会被包含在发布中。

> **注意**　当没有指定 boot_rel 选项时，为了让 Reltool 正常工作，需要有一个名为 start_clean 的发布。在启动关联的 erl 执行文件时，会默认选择这个发布。

如果想排除某个特定的应用，如 megaco 应用，因为从来没有关心过这个应用——就可以使用如下配置：

```
{sys, [
    {lib_dirs, ["/home/ferd/code/learn-you-some-erlang/release/"]},
    {rel, "start_clean", "1.0.0", [kernel, stdlib]},
    {incl_cond, include},
    {debug_info, keep},
    {app, megaco, [{incl_cond, exclude}]}
]}.
```

在此，我们可以指定一个或者多个应用（每个应用具有独立的 app 元组），app 元组中的值会改写掉发布层面 incl_cond 选项的值。在上面的例子中，发布中会包括除 megaco 之外的所有应用。

2. 导入或者导出库的部分文件

在我们的发布中，有个地方令人讨厌：像 ppool 之类的应用，在发布中还保留着它们的测试文件，即使已经不再需要它们了。进入 rel/lib/目录，解压 ppool-1.0.0.ez（需要先更改扩展名），就会看到这些测试文件。

移除这些文件最容易的方法是使用排除过滤器，如下所示：

```
{sys, [
    {lib_dirs, ["/home/ferd/code/learn-you-some-erlang/release/"]},
    {rel, "start_clean", "1.0.0", [kernel, stdlib, ppool, erlcount]},
    {excl_app_filters, ["_tests.beam$"]}
]}.
```

如果只想导入应用中的某些特定文件，会稍微复杂一些。下面是一个例子，其中我们只想导入 erlcount 的 erlcount_lib 功能，其他的都不导入：

```
{sys, [
    {lib_dirs, ["/home/ferd/code/learn-you-some-erlang/release/"]},
    {rel, "start_clean", "1.0.0", [kernel, stdlib]},
    {incl_cond, derived}, % 也可以使用 exclude，但是不能使用 include。
    {app, erlcount, [{incl_app_filters, ["^ebin/erlcount_lib.beam$"]},
                     {incl_cond, include}]}
]}.
```

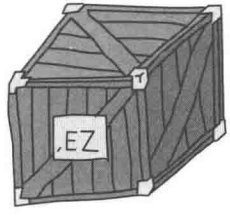
对于这种要求，我们使用了比{incl_cond, include}限制性更强的 incl_cond 选项值。这是因为，如果开始时使用了宽松的限制把所有的东西都包含进来，那么之后要想只包含单个库，唯一的方法就是使用 excl_app_filters 排除掉其他所有库。但是，当选择的限制性更强时（此例中，我们使用了 derived，不会包含 erlcount，因为它不在 rel 元组中），我们可以专门设

定 erlcount 应用的包含方式，只把那些和正则表达式（选择 erlcount_lib 模块）匹配的文件包含到发布中。这会引发另外一个问题，如何构建出一个最小的发布，对吧？

3. 构建更小的发布

如果想让发布更小一些，Reltool 会变得非常复杂，配置文件相当冗长：

```
{sys, [
    {lib_dirs, ["/home/ferd/code/learn-you-some-erlang/release/"]},
    {erts, [{mod_cond, derived},
            {app_file, strip}]},
    {rel, "erlcount", "1.0.0", [kernel, stdlib, ppool, erlcount]},
    {boot_rel, "erlcount"},
    {relocatable, true},
    {profile, embedded},
    {app_file, strip},
    {debug_info, strip},
    {incl_cond, exclude},
    {excl_app_filters, ["_tests.beam$"]},
    {app, stdlib, [{mod_cond, derived}, {incl_cond, include}]},
    {app, kernel, [{incl_cond, include}]},
    {app, ppool, [{vsn, "1.0.0"}, {incl_cond, include}]},
    {app, erlcount, [{vsn, "1.0.0"}, {incl_cond, include}]}
]}.
```



配置文件中的内容很多。可以看到，在 erts 配置方面，我们让 Reltool 只保留必要的东西。把 mod_cond 设置成 derived，app_file 设置成 strip，这两个设置会让 Reltool 进行检查，只保留有用的东西。这也是为什么在发布层面也使用了{app_file, strip}的原因。

profile 选项的值被设置成 embedded。如果你看了前面例子中的.ez 压缩文件，会发现其中包含有源文件、测试目录等内容。当使用 embedded 时，则只会把头文件、二进制文件和 priv/目录包含进来。我们也去除所有文件的 debug_info，即使它们在编译时使用了这个选项。这样做会丧失掉一些调试能力，不过会缩小文件的大小。

我们还去除了测试文件，并且只包含那些显式指定的应用（{incl_cond, exclude}）。然后，在想要包含的应用中改写这个设定。如果缺少了某些东西，Reltool 会给出警告，因此，可以尽量少包含东西，试着做各种设置，直到得到了想要的结果。你可以像我们对 stdlib 所做的设定那样，把某些应用的选项设置为{mod_cond, derived}，这样就只会保留最少的应用文件。

结果会有什么不同呢？一般来说，一个发布的大小要超过 35 MB，上面的例子发布可以减小到 20 MB。虽然还是挺大，但是确实削减掉了相当一部分。主要是因为 ERTS，ERTS 本身就占用了 18.5 MB。如果愿意，就可以再做些深入的工作，了解一些 ERTS 的构建细节，得到更小的发布。可以选择删除一些 ERTS 中你不需要的二进制文件：脚本执行文件、Erlang 远程运行文件、测试框架的二进制文件以及不同模式的运行命令（如是否使用 SMP）。

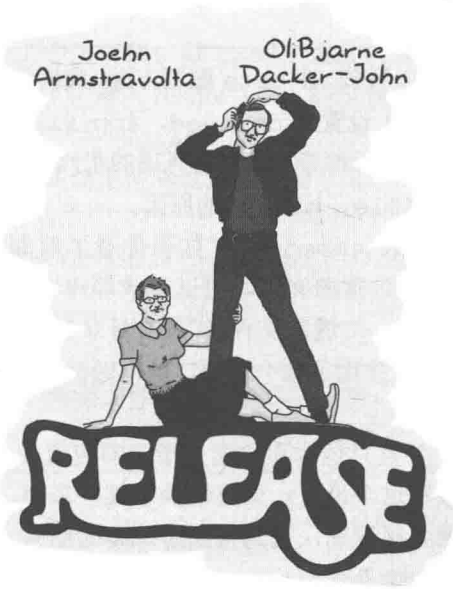
如果能够假设其他用户已经安装了 Erlang，就可以得到最小的发布。使用这种方式时，必须把 rel/目录中的内容增加到 ERL_LIBS 环境变量中，并自己去调用启动文件（和 systools 有点像），不过，这确实是可行的。程序员可以用脚本来实现这些工作。

> **注意**　最近，Erlang 程序员非常喜欢使用 rebar 工具完成发布工作，rebar 可以看成是对 Emakefile 和 Reltool 的包装。花时间理解 Reltool 的工作原理是值得的。rebar 使用的配置文件和 Reltool 几乎完全一样，这两个工具之间的区别不大。

21.5　基于 release 文件发布

好了，两种主要的发布构建方法介绍完了。这是一个复杂的主题，但是也是标准的软件发布方法。对很多读者来说，了解了应用的内容就足够了，即便长期专注于应用方面的内容也不是什么问题。不过，如果想让公司的运维人员更喜欢你一些，偶尔去了解一些发布的内容还是有用的，因为你知道（或者至少有一个概念）在必要时如何去部署 Eralng 应用。

当然，还有什么能比压根就不会停止服务更能让运维人员高兴的呢？下一个挑战是，在发布版本运行期间，对软件进行升级。



第 22 章

升级 Process Quest

在 Erlang 中，对代码进行热加载非常简单。重编译、执行一个全称函数调用，然后就可以大功告成了。但是，要正确地（以及安全地）做这项工作则要困难得多。

在实际操作中，可能出错的地方太多了。

在本章中，我们将了解代码热加载带来的问题，以及解决它们的一些原则，这些原则被证明是有效的。接着，我们会进行一次详细的实战演练，我们将选取一个已有的 OTP 发布，在这个发布运行时，通过 OTP 的 appup 和 relup（应用和发布升级）机制对它进行升级，重新加载它的新版本。

> **注意** 本章没有包含所有的示例代码。在开始学习本章前，请从 http://learnyousomeerlang.com/static/erlang/processquest.zip 下载需要的代码。如果已经从 *Learn You Some Erlang* 网站下载了完整的代码包，里面就已经包含了所需的代码。

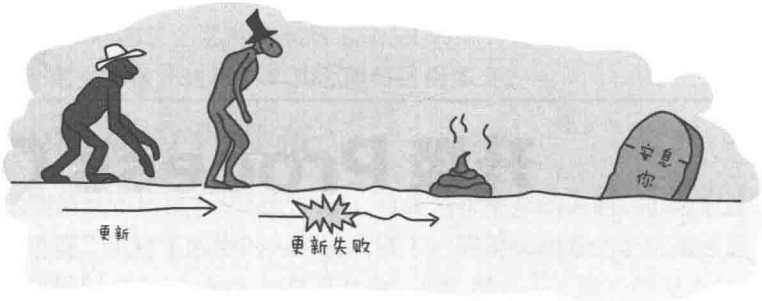
22.1 升级面临的问题

一个非常简单的挑战就会让代码重新加载出现问题。为了更好地理解这个问题，我们把大脑切换到美妙的 Erlang 编程模式，假想有这么一个 `gen_server` 进程。这个进程有一个 `handle_cast` 函数，接收某种类型参数。你改变了这个函数接收的参数类型，然后编译它，并提交到产品代码中。一切都不错，不过因为你的应用已经在运行了，你不想停止它，所以决定在生产环境的 VM 中加载这个模块，使之运行。

接下来，大量错误报告开始涌现。

事实表明，新旧两个 `handle_cast` 函数并不兼容。因此，在第二次调用时，没有找到匹配的函数子句。用户和老板都非常生气。负责运维的家伙当然也非常恼火，因为他得赶过去回退代码、消除问题。你可能会有幸成为那个负责运维的家伙。你一直待到很晚，看门工人美妙的夜晚时光也被你破坏了（他喜欢随着音乐哼歌，并且在工作时跳跳舞，但是你在场时，他不好意思这

么做)。你很晚才回家，你的妻子、朋友、魔兽世界游戏作战队友、孩子都对你非常恼怒。他们大喊、尖叫、摔门，而你只能孤单地待着。你曾经许诺过不会出现任何问题——不会有停止服务时间。毕竟，你使用的是 Erlang，对吧？但是现实却并非如此。你孤零零的一个人，像球一样蜷缩在厨房的角落里，吃着冰冷的微波快餐。



当然，事情不会总是这么糟糕，不过我想表达的观点依然成立。如果更改了模块的外部接口，改变了内部数据结构、改变了函数名字、修改了记录（记住，它们就是元组！）等，那么在生产系统中对其进行在线代码升级是非常危险的。这些更改都可能会造成崩溃。

当我们在第 13 章第一次进行代码重新加载时，使用了一个进程，它在处理某个隐藏消息时会进行一次完全限定调用。回想一下，进程看起来像这样：

```
loop(N) ->
    receive
        some_standard_message -> N+1;
        other_message -> N-1;
        {get_count, Pid} ->
            Pid ! N,
            loop(N);
        update -> ?MODULE:loop(N);
    end.
```

不过，如果想改变 `loop/1` 的参数，那么这种方法无法满足要求。我们需要这样扩展一下：

```
loop(N) ->
    receive
        some_standard_message -> N+1;
        other_message -> N-1;
        {get_count, Pid} ->
            Pid ! N,
            loop(N);
        update -> ?MODULE:code_change(N);
    end.
```

在 `code_change/1` 函数中，会调用新版本的 `loop` 函数。不过这个技巧不适用于通用的循环。考虑如下例子：

```
loop(Mod, State) ->
    receive
```

```
        {call, From, Msg} ->
            {reply, Reply, NewState} = Mod:handle_call(Msg, State),
            From ! Reply,
            loop(Mod, NewState);
        update ->
            {ok, NewState} = Mod:code_change(State),
            loop(Mod, NewState)
    end.
```

看出问题了吗？在这个实现中，不能安全地更改 Mod 并加载它的新版本。Mod:handle_call(Msg, State)这个调用本身就是一个完全限定调用，并且可能会在重新加载了模块和处理 update 消息之间收到{call, From, Msg}消息。在这种情况下，模块的更新是失控的。然后，程序就崩溃了。

解决这个问题的秘密隐藏在 OTP 内部。我们必须要冻结时间！要达到这个目的，需要额外的秘密消息：暂停进程消息、更改代码消息以及恢复执行消息。OTP 行为的内部实现中，隐藏着一个专门处理这些事务的协议。具体的处理是通过 sys 模块和 release_handler 模块进行的，这两个模块属于系统架构支持库（SASL）应用。它们负责完成所有工作。

诀窍在于，可以调用 sys:suspend(*PidOrName*)来暂停 OTP 进程（所有进程都可以通过在监督树中查找监督者子进程的方法找到）。接着调用 sys:change_code(*PidOrName*, Mod, OldVsn, Extra)强迫进程更新自己的代码。最后，调用 sys:resume(*PidOrName*)恢复进程的执行。

总是编写一些临时脚本手工调用这些函数，显然不是一种切实可行的做法。不过，我们倒是可以学习一下 relup 是怎么做的。

22.2 Erlang 学习的第 9 级

对于正在运行的发布，为它制作出一个新版本，然后在它运行期间更新版本的行为是非常危险的。一项看起来很简单的 appup（更新单个应用的指令文件）和 relup（更新整个发布的指令文件）的组装工作，突然之间就会变成一场与一些 API 和暗藏假设的斗争。

我们现在要学习的是 OTP 中最复杂的一部分内容，即使在其上花费了大量时间，仍然难以理解和正确运用。事实上，如果在进行简单的常规升级时，允许通过重启 VM 来启动新的应用，从而避免整个发布升级流程（此后，我们称这个流程为 relup），那我推荐你这么做。Relup 应该是一种“破釜沉舟”型工具——在没有其他选择时，可以考虑使用它。

在进行发布升级时，会执行一组步骤，下面是简化后的步骤描述。

（1）编写 OTP 应用。

（2）把一组应用打包成一个发布。

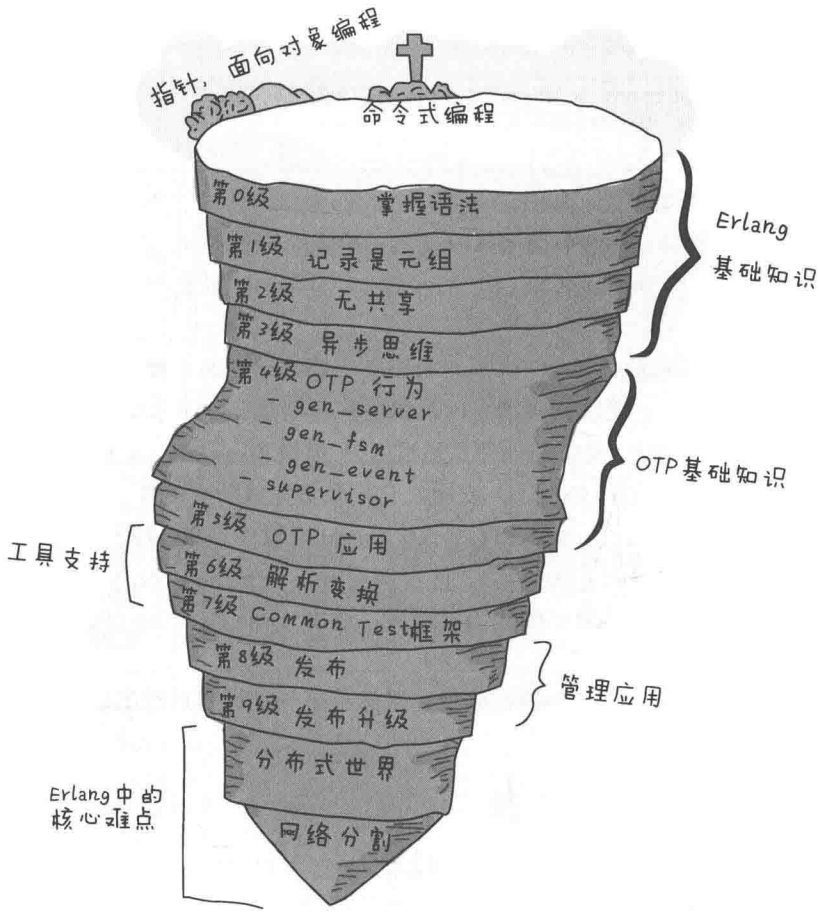
（3）创建一个或多个 OTP 应用的新版本。

（4）创建一个 appup 文件，其中指出了要成功进行新老版本转换需要做的改变。

（5）用新应用版本创建一个新的发布。

（6）基于这两个发布生成一个 relup 文件。

（7）打开一个 Erang shell，安装新的应用。

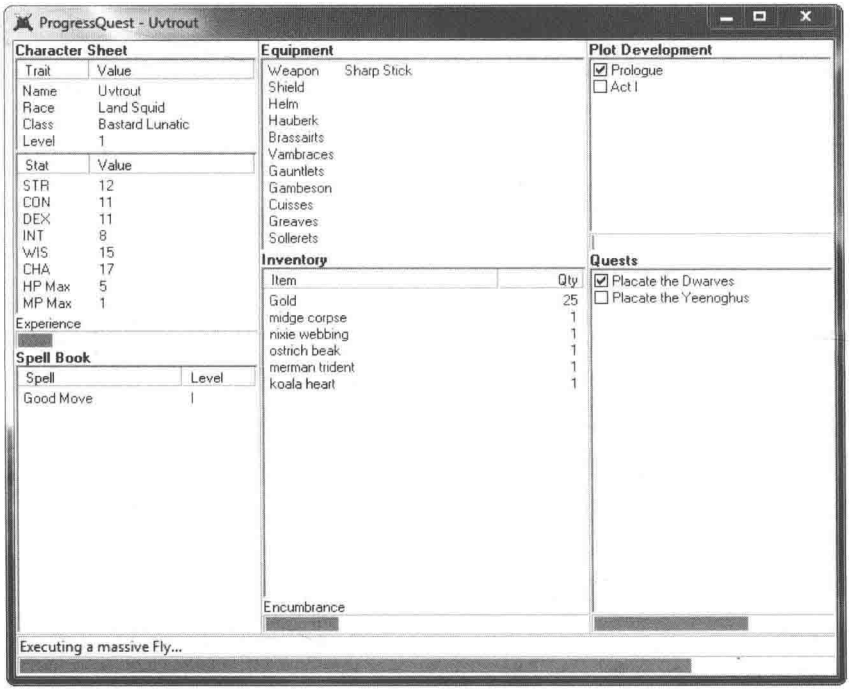


到目前为止，我们只介绍了前面 3 个步骤。前面介绍的应用，其运行时间可能还没有升级花费的时间长，为了演示一下如何对更适合的应用进行升级（嗯，谁会在乎正则表达式运行期间是否重启呢？），我们会以一款优异的视频游戏应用为例。

22.3 Process Quest

Progress Quest（http://progressquest.com/）是一款革命性的角色扮演类游戏（RPG）。事实上，我会称其为 RPG 中的 OTP。如果你之前曾经玩过某款 RPG 游戏，就会注意到很多步骤都非常类似：到处走动、杀死敌人、获得经验值、获得金钱、升级、获得技能，完成使命——反复循环。某些高手玩家会使用一些快捷功能，如宏，甚至会编写一些机器人程序在自己周围四处走动，并用它们来抬高自己的身价。

Progress Quest 游戏使用所有这些通用步骤，并把它们变成一个流线型游戏，你可以在一边坐着休息，欣赏自己的角色完成所有工作：



在得到了这款优秀游戏的作者 Eric Fredricksen 的许可后，我制作了这款游戏的简单 Erlang 克隆版本，我称之为 Process Quest。Process Quest 大体上和 Progress Quest 类似，但是它不是一个单用户应用，而是一个可以容纳大量原始套接字连接（可以通过 *telnet* 使用）的服务器应用，玩家可以随时使用终端接入，玩会儿这个游戏。

这个游戏由 `regis`、`processquest` 和 `sockserv` 这 3 个应用组成。

22.3.1 regis-1.0.0 应用

`regis` 应用是一个进程注册库。它的接口和 Erlang 内置的进程注册库有点相似，不过它可以接受任何类型的数据项，并适用于动态场景。它的性能会稍低一点，因为所有的调用在进入服务器之后都是串行执行的，不过它要比内置的进程注册库好用一些，因为内置的进程注册库不适用于动态的情况。如果本书能够自动用外部库来更新自身（需要大量的工作），那么我会使用 `gproc`。

该应用包含 3 个模块：`regis.erl`、`regis_server.erl` 和 `regis_sup.erl`。第一个模块是对后面两个模块的封装（它也是一个应用回调模块）。`regis_server` 是具有注册名字的主 `gen_server`，`regis_sup` 是应用的监督者。

22.3.2 processquest-1.0.0 应用

`processquest` 应用是整个发布的核心。它包含了所有的游戏逻辑——敌人、集市、杀戮场以及统计。玩家是一个 `gen_fsm`，会给自己发送消息以保持游戏一直进行。它包含如下模块。

`pq_enemy.erl`

这个模块随机地选择一个敌人作战，格式是`{<<"Name">>,[{drop, {<<"DropName">>,`

Value}}, {experience, ExpPoints}]}。这会让玩家去攻击这个敌人。

pq_market.erl

这个模块实现了一个集市，玩家可以从中找到一些具有给定价值或者给定威力的物品。所有物品的格式都是{<<"Name">>, Modifier, Strength, Value}。该模块提供了获取武器、装备、防御、头盔等物品的函数。

pq_stats.erl

这个小模块是一个角色属性生成器。

pq_events.erl

这个模块是对 gen_event 事件管理器的封装。它充当一个通用的集线器，订阅者可以使用自己的事件处理器连接进来，接收来自每个玩家的消息。它还会负责为玩家动作增加一定的延迟，避免游戏速度过快。

pq_player.erl

这是一个最重要的模块。它是一个 gen_fsm，运行在一个状态循环中：先到杀戮场，接着去集市，然后又到杀戮场等。它会用到上述所有模块。

pq_sup.erl

这个模块是一个监督者，位于 pq_events 和 pq_player 进程之上。这两个进程需要联合工作，否则，要么玩家进程会被孤立，没有任何动作，要么事件管理器收不到任何事件。

pq_supersup.erl

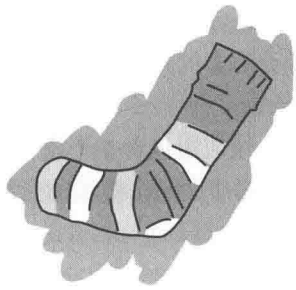
这个模块是应用的顶层监督者。它位于一组 pq_sup 进程之上。可以用它来创建任意数量的玩家。

processquest.erl

这是一个包装和应用回调模块。它提供了基本的玩家接口。你可以启动一个玩家，然后开始订阅事件。

22.3.3 sockserv-1.0.0 应用

sockserv 应用是一个定制化的原生套接字服务器，仅用于 processquest 应用。它会创建多个 gen_server，每个 gen_server 都管理着一个 TCP 套接字，会把收到的字符串推送给客户。可以使用 telnel 和它相连。（从技术上讲，telnet 并不是用于原生套接字连接的，它具有自己的协议，不过几乎所有的现代客户端都可以适应这种情况）。sockserv 应用的模块如下。



sockserv_trans.erl

这个模块把来自玩家事件管理器的消息翻译成可打印的字符串。

sockserv_pq_events.erl

这个模块是一个简单的事件处理器，它把来自玩家的所有消息都发送给套接字 gen_server。

sockserv_serv.erl

这个模块是一个 gen_server，它负责接受连接、与客户交互，并向客户转发信息。

sockserv_sup.erl

这是一个监督者，会监督一组套接字服务器。

sockserv.erl

这个模块是整个应用的回调模块。

22.3.4 发布

我在一个名为 processquest 的目录中准备好了一切，结构如下：

```
apps/
 - processquest-1.0.0
   - ebin/
   - src/
   - ...
 - regis-1.0.0
   - ...
 - sockserv-1.0.0
   - ...
rel/
（保存发布的地方）
processquest-1.0.0.config
```

基于这里的内容，我们可以构建出一个发布。

发布的其他配置

如果你查看一下 processquest-1.0.0.config 文件，就会发现其中包含 Crypto 和 SASL 之类的应用。要想更好地初始化伪随机数生成器，Crypto 应用是必需的，而 SASL 则是对系统进行 appup 升级所不可缺少的。如果忘记了在发布中包含 SASL，是绝不可能进行系统升级的。

配置文件中使用了一个新过滤器：{excl_archive_filters, [".*"]}。这个过滤器保证不会生成任何.ez 文件，只会创建常规的文件和目录。这是必须的，因为我们所使用的工具不能从.ez 文件中找出所需要的东西。

你可能还注意到了，配置文件中没有要求去掉 debug_info 的指令。如果没有了 debug_info，在进行 appup 时会由于某些原因失败。始终包含 debug_info 通常都是有益的。

按照第 21 章中构建发布的指令，我们首先对所有应用调用 erl -make。做完这一步后，从 processquest 目录启动一个 Erlang shell，并输入如下内容：

```
1> {ok, Conf} = file:consult("processquest-1.0.0.config"),
1> {ok, Spec} = reltool:get_target_spec(Conf),
1> reltool:eval_target_spec(Spec, code:root_dir(), "rel").
ok
```

命令执行后，应该生成了一个可用的发布。我们来试一试。输入./rel/bin/erl -sockserv port 8888 启动任意版本的 VM（或者使用其他端口，默认是 8082）。这条命令在执行时，会显示出很多进程启动的日志信息（这是 SASL 的功能之一），之后进入一个常规的 Erlang shell。在你的本地主机上，使用任意 telnet 客户端启动一个 telnet 会话：

```
$ telnet localhost 8888
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
What's your character's name?
hakvroot

Stats for your character:
  Charisma: 7
  Constitution: 12
  Dexterity: 9
  Intelligence: 8
  Strength: 5
  Wisdom: 16

Do you agree to these? y/n
```

对我来说，这个角色的智慧和魅力值太高了。因此我输入 n 并按回车：

```
n
Stats for your character:
  Charisma: 6
  Constitution: 12
  Dexterity: 12
  Intelligence: 4
  Strength: 6
  Wisdom: 10

Do you agree to these? y/n
```

好，这个角色又丑、又笨、又弱——正好是我苦苦寻找的英雄角色。

```
y
Executing a Wildcat...
Obtained Pelt.
Executing a Pig...
Obtained Bacon.
Executing a Wildcat...
Obtained Pelt.
Executing a Robot...
Obtained Chunks of Metal.
...
Executing a Ant...
Obtained Ant Egg.
Heading to the marketplace to sell loot...
Selling Ant Egg
```

```
Got 1 bucks.
Selling Chunks of Metal
Got 1 bucks.
...
Negotiating purchase of better equipment...
Bought a plastic knife
Heading to the killing fields...
Executing a Pig...
Obtained Bacon.
Executing a Ant...
```

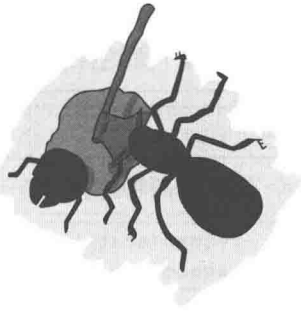
游戏玩够了。我输入 quit，接着按回车键关闭连接：

```
quit
Connection closed by foreign host.
```

如果愿意，就可以一直保持连接，观看自己不断升级、不断累积统计数据等。这个游戏基本上可以使用了，你可以用多个客户端来试玩它。它应该一直正常工作，不会出现问题。

太好了，对吧？嗯，我们可以做得更好。

22.4 改进 Process Quest



Process Quest 应用目前存在几个问题。首先，可对抗的敌人种类太少。其次，显示的文本看起来有些奇怪（“Executig a Ant …”是什么意思？）。最后，游戏太简单，要增加一个闯关模式！另外还有一个问题是，Progress Quest 游戏中，你在集市上出售物品的价值直接和你在游戏中的级别绑定，而 Process Quest 游戏中完全没有这项功能。最后——仅当你阅读了源代码，并且试着主动关闭客户端时才会发现这个问题——客户端关闭了连接后，玩家进程仍然在服务器上存活着。哎哟，内存泄漏！

我们需要修复这些问题！

> **注意** 我先把那两个需要修正的应用复制了一份，然后把复制目录命名为 processquest-1.1.0 和 sockserv-1.0.1。（我使用的版本号模式是“主版本.增强.错误修正”）。在复制的代码中，我完成了所有需要的更改。在此，我不会介绍所有的更改，因为我们的主要任务是升级应用，而不是学习这个特定应用的所有细节和难点。如果你想了解所有的小细节，我在代码中做了注释，你可以找到理解它们所需要的信息。

22.4.1 更改 code_change 函数

在 processquest-1.1.0 中，我更改了模块 pq_enemy.erl、pq_events.erl 和 q_player.erl。我还增加了一个文件 pq_quest.erl，它实现了玩家需要杀死多少敌人才能完成某项任务的功能。在这些文件中，只有对 pq_player.erl 的更改是不兼容的，需要在升级时暂停进程。我们把记录。

```
-record(state, {name, stats, exp=0, lvlexp=1000, lvl=1,
                equip=[], money=0, loot=[], bought=[], time=0}).
```

更改为

```
-record(state, {name, stats, exp=0, lvlexp=1000, lvl=1,
                equip=[], money=0, loot=[], bought=[],
                time=0, quest}).
```

其中字段 quest 会保存函数 pq_quest:fetch/0 的返回值。

由于这个改变，我们需要修改版本 1.1.0 中的 code_change/4 函数。事实上，要基于两种情况修改这个函数：一种是针对升级（从 1.0.0 升级到 1.1.0），还有一种是针对降级（1.1.0 到 1.0.0）。还好，对这两种情况，OTP 会传入不同的参数。在升级时，OTP 会传入模块的老版本号。此时，我们对这个版本号并不关心，直接忽略它。在降级时，OTP 会传入{down, Version}。这样，就可以比较容易地对这两种操作进行匹配了：

```
code_change({down, _}, StateName, State, _Extra) ->
    ...;
code_change(_OldVsn, StateName, State, _Extra) ->
    ...
```

等等！现在，我们不能像往常那样随便获取状态数据。我们需要对状态进行升级。问题是，我们不能像下面这样做：

```
code_change(_OldVsn, StateName, S = #state{}, _Extra) ->
    ...
```

我们有两种选择。第一种选择是，定义一个新的状态记录，它有新的格式和名字。实现如下：

```
-record(state, {...}).
-record(new_state, {...}).
```

接下来，我们需要更改模块中每个函数子句使用的记录。这非常讨厌，和导致的风险相比，也不值得。另外一种方法要简单一些，把记录展开成其底层的元组形式（在第 9 章介绍记录时讲过），具体如下：

```
code_change({down, _},
            StateName,
            #state{name=N, stats=S, exp=E, lvlexp=LE, lvl=L, equip=Eq,
                   money=M, loot=Lo, bought=B, time=T},
            _Extra) ->
    Old = {state, N, S, E, LE, L, Eq, M, Lo, B, T},
    {ok, StateName, Old};
code_change(_OldVsn,
            StateName,
            {state, Name, Stats, Exp, LvlExp, Lvl, Equip, Money, Loot,
             Bought, Time},
            _Extra) ->
    State = #state{
        name=Name, stats=Stats, exp=Exp, lvlexp=LvlExp,
        lvl=Lvl, equip=Equip, money=Money, loot=Loot,
```

```
        bought=Bought, time=Time, quest=pq_quest:fetch()
    },
    {ok, StateName, State}.
```

这就是我们的 code_change/4 函数实现！它所做的就是在两种元组形式之间进行转换。在新元组中，我们小心地增加了一个新的 quest 字段——如果增加了 quest 字段，但是现有的所有玩家都无法使用它，那将是非常令人讨厌的。

你会注意到，我们仍然忽略了_Extra 参数。这个参数来自 appup 文件（后面会介绍），你会得到在那里定义的值。现在，我们不关心这个值，因为我们只会升级或者降级到一个发布版本。对于某些更复杂的情况，可能需要用这个变量来传递发布特定信息。

对于 sockserv-1.0.1 应用，只需更改 *sockserv_serv.erl*。还好，不需要重启它们，只需要增加一条新的消息匹配即可。

两个应用的新版本都做好了。不过，现在还不是庆祝的时候。我们需要一种方法，能够让 OTP 知道每种更改所要求的操作指令类型。

22.4.2 增加 appup 文件

appup 文件是一组 Erlang 命令列表，在升级某个指定应用时需要执行这些命令。文件的内容由元组和原子列表组成，描述了在什么情况下要做什么事情。appup 文件常见的格式如下所示：

```
{NewVersion,
 [{VersionUpgradingFrom, [Instructions]}]
 [{VersionDownGradingTo, [Instructions]}]}.
```

文件中要求的是一个版本列表，这是因为可能升级或者降级到多个不同版本。对于我们的 processquest-1.1.0 例子来说，appup 文件的内容如下：

```
{"1.1.0",
 [{"1.0.0", [Instructions]}],
 [{"1.0.0", [Instructions]}]}.
```

Instructions 中既包含有高层命令，也包含有低层命令。不过，通常只需关心高层命令。

{add_module, Mod}

模块 Mod 是首次加载。

{load_module, Mod}

模块 Mod 已经加载到 VM 中，被更改了。

{delete_module, Mod}

从 VM 中删除模块 Mod。

{update, Mod, {advanced, Extra}}

暂停所有运行 Mod 模块的进程，把 Extra 作为最后一个参数调用模块的 code_change 的函数，然后恢复执行所有运行 Mod 模块的进程。如果在升级时需要其他信息，可以放到 Extra 中传递给 code_change 函数。

{update, Mod, supervisor}

这个命令用来重新定义监督者的 init 函数，可以影响其重启策略（one_for_one、rest_for_one 等）或者改变子进程规格说明（不会影响已经创建的进程）。

{apply, {M, F, A}}

调用 apply(M,F,A)。

模块依赖

如果想确保先处理其他一些模块，之后再执行一条命令，可以使用{load_module, Mod,[ModDependencies]}或者{update, Mod,{advanced, Extra}, [ModDeps]}。这条命令尤其适用于 Mod 及其所依赖的模块不属于同一个应用的情况。遗憾的是，delete_module 指令没有类似的依赖处理指令。

注意，在生成 relup 时，不需要任何特殊指令来删除或者增加应用。生成 relup 文件（发布升级文件）的函数会自动进行检测，完成需要的工作。

可以用这些指令为我们的应用编写两个 appup 文件。文件必须命名为 NameOfYourApp.appup，并且放在应用的 ebin/目录下。

processquest-1.1.0 的 appup 文件如下：

```
{"1.1.0",
 [{"1.0.0", [{add_module, pq_quest},
             {load_module, pq_enemy},
             {load_module, pq_events},
             {update, pq_player, {advanced, []}, [pq_quest, pq_events]}]}],
 [{"1.0.0", [{update, pq_player, {advanced, []}},
             {delete_module, pq_quest},
             {load_module, pq_enemy},
             {load_module, pq_events}]}]}.
```

可以看出，我们增加了一个新模块，加载了两个无需暂停进程的模块，然后安全地更新了 pq_player 模块。在降级代码时，我们以相反的方式做了同样的事情。有趣的是，在升级时，{load_module, Mod}会加载一个新的版本，而在降级时，它会加载老的版本。这一切都完全依赖于此时的上下文是升级还是降级。

由于 sockserv-1.0.1 中只有一个模块需要改变，并且不用暂停进程，因此它的 appup 文件很简单，如下：

```
{"1.0.1",
 [{"1.0.0", [{load_module, sockserv_serv}]}],
 [{"1.0.0", [{load_module, sockserv_serv}]}]}.
```

接下来要做的是使用新模块构建一个新发布。processquest-1.1.0.config 文件内容如下所示：

```
{sys, [
   {lib_dirs, ["/home/ferd/code/learn-you-some-erlang/processquest/apps"]},
   {erts, [{mod_cond, derived},
           {app_file, strip}]},
```

```
    {rel, "processquest", "1.1.0",
     [kernel, stdlib, sasl, crypto, regis, processquest, sockserv]},
    {boot_rel, "processquest"},
    {relocatable, true},
    {profile, embedded},
    {app_file, strip},
    {incl_cond, exclude},
    {excl_app_filters, ["_tests.beam"]},
    {excl_archive_filters, [".*"]},
    {app, stdlib, [{mod_cond, derived}, {incl_cond, include}]},
    {app, kernel, [{incl_cond, include}]},
    {app, sasl, [{incl_cond, include}]},
    {app, crypto, [{incl_cond, include}]},
    {app, regis, [{vsn, "1.0.0"}, {incl_cond, include}]},
    {app, sockserv, [{vsn, "1.0.1"}, {incl_cond, include}]},
    {app, processquest, [{vsn, "1.1.0"}, {incl_cond, include}]}
]}.
```

这个文件只是把老文件中的内容复制/粘贴过来，然后更改了几个版本号。首先，使用 `erl -make` 编译这两个新的应用（新版本在前面提到的 zip 文件中）。然后就可以生成新的发布了。编译完这两个新应用后，输入如下命令：

```
$ erl -env ERL_LIBS apps/
1> {ok, Conf} = file:consult("processquest-1.1.0.config"),
1> {ok, Spec} = reltool:get_target_spec(Conf),
1> reltool:eval_target_spec(Spec, code:root_dir(), "rel").
ok
```

等等，还有不少手工工作要做！

（1）把 rel/releases/1.1.0/processquest.rel 复制成 rel/releases/1.1.0/processquest-1.1. 0.rel。

（2）把 rel/releases/1.1.0/processquest.boot 复制成 rel/releases/1.1.0/processquest-1. 1.0.boot。

（3）把 rel/releases/1.1.0/processquest.boot 复制成 rel/releases/1.1.0/start.boot。

（4）把 rel/releases/1.0.0/processquest.rel 复制成 rel/releases/1.0.0/processquest-1.0. 0.rel。

（5）把 rel/releases/1.0.0/processquest.boot 复制成 rel/releases/1.0.0/processquest-1.0. 0.boot。

（6）把 rel/releases/1.0.0/processquest.boot 复制成 rel/releases/1.0.0/start.boot。

保持冷静

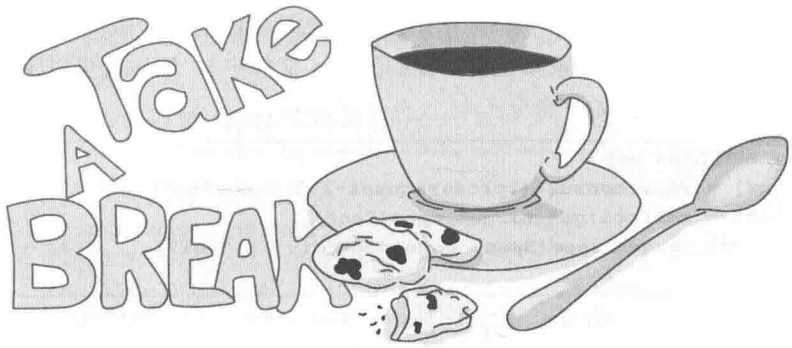
为什么不使用 systools 来构建我们的发布呢？嗯，systools 有其自身的问题。首先，它生成的 appup 文件有时会包含一些奇怪的版本号，无法正常工作。另外，它会要求一种目录结构，而这种结构的形式几乎没有文档说明，只知道和 Reltool 的目录结构有点接近。不过，最大的问题是，它把默认的 Erlang 安装目录当作根目录，在解压时会导致各种权限问题。

无论是使用哪种工具，发布构建都不是一项容易的工作，它需要很多的手工操作。因此，我们把操作两个模块的命令以一种非常复杂的方式组成一条链，这样可以少做些工作。

现在，我们可以生成 relup 文件了。要完成这项工作，请启动一个 Erlang shell，输入如下命令：

```
    erl -env ERL_LIBS apps/ -pa apps/processquest-1.0.0/ebin/ -pa apps/sockserv-1.0.0/
bin/
    1> systools:make_relup("./rel/releases/1.1.0/processquest-1.1.0",
    1>                     ["rel/releases/1.0.0/processquest-1.0.0"],
    1>                     ["rel/releases/1.0.0/processquest-1.0.0"]).
    ok
```

因为 ERL_LIBS 环境变量只会寻找应用的最新版本，所以需要增加-pa *Path_to_older_applications* 选项，这样 systools 的 relup 生成器就可以找到所有的应用。命令执行后，把 relup 文件移到 rel/releases/1.1.0/目录下。在升级代码时，会从这个目录中寻找需要的东西。现在，还有一个问题，发布处理器模块要依赖于一组文件，它会认为这组文件已经存在，但是这组文件有可能不存在。



22.4.3　发布升级

很好——我们有了 relup 文件。不过在使用它之前，还得做些工作。下一步是为这个全新的发布版本生成一个 tar 文件：

```
    2> systools:make_tar("rel/releases/1.1.0/processquest-1.1.0").
    ok
```

文件会被生成在 rel/releases/1.1.0/目录。现在，需要手工把它移到 rel/releases 目录。调用上面的命令时，需要在文件名后面手动加上版本号。又是一个硬编码的工作！我们的做法如下：

```
    $mv rel/releases/1.1.0/processquest-1.1.0.tar.gz rel/releases/
```

接下来的这一步，你可以在*启动真正的产品应用之前的任何时间点*去做。之所以要放在启动应用之前做，是因为它可以让你在完成发布升级后回退到最初的版本。如果不做这一步，那么只能把产品应用降级到第一个版本之后的版本，但是无法回退到第一个版本！

打开一个 shell，运行如下命令：

```
    1> release_handler:create_RELEASES(
```

```
1>     "rel",
1>     "rel/releases",
1>     "rel/releases/1.0.0/processquest-1.0.0.rel",
1>     [{kernel,"2.14.4", "rel/lib"}, {stdlib,"1.17.4","rel/lib"},
1>      {crypto,"2.0.3","rel/lib"},{regis,"1.0.0", "rel/lib"},
1>      {processquest,"1.0.0","rel/lib"},{sockserv,"1.0.0", "rel/lib"},
1>      {sasl,"2.1.9.4", "rel/lib"}]
1> ).
```

这个函数的一般形式如下：

```
release_handler:create_RELEASES(RootDir, ReleasesDir, Relfile, [{AppName, Vsn,
LibDir}])
```

这条命令会在目录 rel/release（或者其他任何 ReleasesDir）中创建一个名为 RELEASES 的文件，其中包含着关于发布的基本信息，relup 在寻找要加载的文件和模块时，需要使用这些信息。

现在，我们可以把老版本的代码运行起来。如果运行 rel/bin/erl，会默认启动 1.1.0 发布。这是因为新发布版本构建于 VM 启动之前。为了能够演示，需要使用下面的命令启动发布：

```
$ ./rel/bin/erl -boot rel/releases/1.0.0/processquest
```

应该可以看到一切启动良好。打开一个 telnet 客户端，连接到套接字服务器，这样就可以看到热升级的进展情况了。

如果你觉得升级准备工作已经就绪，可以切换到当前运行 Process Quest 游戏的 Erlang shell，并调用如下函数：

```
1> release_handler:unpack_release("processquest-1.1.0").
{ok,"1.1.0"}
2> release_handler:which_releases().
[{"processquest","1.1.0",
  ["kernel-2.14.4","stdlib-1.17.4","crypto-2.0.3",
   "regis-1.0.0","processquest-1.1.0","sockserv-1.0.1",
   "sasl-2.1.9.4"],
  unpacked},
{"processquest","1.0.0",
 ["kernel-2.14.4","stdlib-1.17.4","crypto-2.0.3",
  "regis-1.0.0","processquest-1.0.0","sockserv-1.0.0",
  "sasl-2.1.9.4"],
 permanent}]
```

上面第二条命令显示，已经做好发布的升级准备，但是发布还没有安装或者永久化。要安装发布，可以输入如下命令：

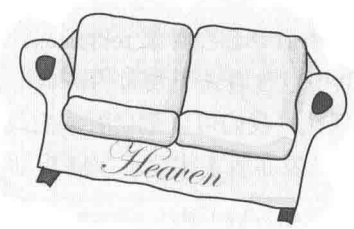
```
3> release_handler:install_release("1.1.0").
{ok,"1.0.0",[]}
4> release_handler:which_releases().
[{"processquest","1.1.0",
  ["kernel-2.14.4","stdlib-1.17.4","crypto-2.0.3",
   "regis-1.0.0","processquest-1.1.0","sockserv-1.0.1",
   "sasl-2.1.9.4"],
  current},
```

```
{"processquest","1.0.0",
 ["kernel-2.14.4","stdlib-1.17.4","crypto-2.0.3",
  "regis-1.0.0","processquest-1.0.0","sockserv-1.0.0",
  "sasl-2.1.9.4"],
 permanent}]
```

现在，发布 1.1.0 应该运行起来了，不过它还没有被永久化。当然，你可以让应用就以这种方式运行。如果想让发布永久化，可以调用如下函数：

```
5> release_handler:make_permanent("1.1.0").
ok.
```

啊，糟糕——有一批进程死了（上面示例中去掉了错误输出）。但是，如果你看看 telnet 客户端，好像确实升级成功了。问题在于，由于接受 TCP 连接是一个阻塞操作，因此 sockserv 中所有等待连接请求的 gen_server 进程都无法监听消息了。因此，在新版本代码被加载后，服务器不能进行升级，从而被 VM 杀死了。我们可以通过下面的操作进行证实：




```
6> supervisor:which_children(sockserv_sup).
[{undefined,<0.51.0>,worker,[sockserv_serv]}]
7> [sockserv_sup:start_socket() || _ <- lists:seq(1,20)].
[{ok,<0.99.0>},
 {ok,<0.100.0>},
 ... <snip> ...
 {ok,<0.117.0>},
 {ok,<0.118.0>}]
8> supervisor:which_children(sockserv_sup).
[{undefined,<0.112.0>,worker,[sockserv_serv]},
 {undefined,<0.113.0>,worker,[sockserv_serv]},
 ... <snip> ...
 {undefined,<0.109.0>,worker,[sockserv_serv]},
 {undefined,<0.110.0>,worker,[sockserv_serv]},
 {undefined,<0.111.0>,worker,[sockserv_serv]}]
```

第一条命令显示，所有等待连接请求的子进程都已经死亡。剩下的都是已经建立活动会话的子进程。这个结果说明了保持代码可响应的重要性。如果进程能始终接收并处理消息，就不会出现这种问题。

为了修正这个问题，我们在后面两条命令启动了更多的工作者进程。虽然这种做法能解决问题，但是它需要在升级时人工介入。无论如何，这都算不上理想的解决方案。

这个问题有一个更好的解决方法：改变应用的工作方式，用一个监控进程监视 sockserv_sup 的子进程数量。当子进程数目低于某个阈值时，监控进程就启动更多的子进程。

还有一种方法是，更改接受连接的代码，使其一次阻塞几秒，阻塞结束时可以尝试去接收消息，处理完消息再去阻塞等待。这样 gen_server 就有时间去完成升级，不过，你得在安装了发布后，在使其永久化之前等待合适的时间。这两个方案的实现作为练习留给读者（因为我

有点懒)。

出现这种崩溃再次表明，在产品系统升级之前，需要做些测试。如果想对计划中的发布升级进行真正的测试，那么必须要测试升级和降级这两种情况，并且在失败时能够重启节点，这样做只是为了以防万一。

总之，现在我们已经解决了问题。我们来看看升级过程是如何进行的：

```
9> release_handler:which_releases().
[{"processquest","1.1.0",
  ["kernel-2.14.4","stdlib-1.17.4","crypto-2.0.3",
   "regis-1.0.0","processquest-1.1.0","sockserv-1.0.1",
   "sasl-2.1.9.4"],
  permanent},
 {"processquest","1.0.0",
  ["kernel-2.14.4","stdlib-1.17.4","crypto-2.0.3",
   "regis-1.0.0","processquest-1.0.0","sockserv-1.0.0",
   "sasl-2.1.9.4"],
  old}]
```

值得庆祝一下。你可以调用 `release_handler:install(OldVersion)`.来尝试一下版本降级。应该不会有什么问题，不过这个操作可能会杀死那些无法更新自己的进程。

保持冷静

如果使用本章介绍的技术回退到发布的第一个版本时一直失败，可能是忘了创建 RELEASES 文件。如果在调用 `release_handler:which_releases()`时，在`{YourRelease, Version, [], Status}`中看到一个空列表，就可以确定是这种情况。这是一个应用列表，会从这个列表中寻找要加载和重加载的模块。创建这个列表的时机有两个：一个是在 VM 启动时基于 RELEASES 文件中的内容创建，另一个是在解压一个新发布时创建。

22.5 Relup 回顾

我们来总结一下，以下操作列表是成功地发布升级所必需的。

(1) 为软件的第一个迭代编写 OTP 应用。

(2) 编译应用。

(3) 使用 Reltool 构建一个发布（1.0.0)。必须包含调试信息，并且不能生成.ez 文件。

(4) 确保在启动产品应用之前创建了 RELEASES 文件。可以调用 `release_handler:create_RELEASES(RootDir, ReleasesDir, Relfile, [{AppName, Vsn, LibDir}])` 完成该文件的创建。

(5) 运行发布！

(6) 发现发布中的 bug。

(7) 在应用的新版本中修正 bug。

(8) 针对每个应用编写 appup 文件。

（9）编译新应用。

（10）构建新发布（在我们的例子中是 1.1.0）。必须包含调试信息，并且不能生成.ez 文件。

（11）把 rel/releases/NewVsn/RelName.rel 复制成 rel/releases/NewVsn/RelName-NewVsn.rel。

（12）把 rel/releases/NewVsn/RelName.boot 复制成 rel/releases/NewVsn/RelName-NewVsn.boot。

（13）把 rel/releases/NewVsn/RelName.boot 复制成 rel/releases/NewVsn/start.boot。

（14）把 rel/releases/OldVsn/RelName.rel 复制成 rel/releases/OldVsn/RelName-OldVsn.rel。

（15）把 rel/releases/OldVsn/RelName.boot 复制成 rel/releases/OldVsn/RelName-OldVsn.boot。

（16）把 rel/releases/OldVsn/RelName.boot 复制成 rel/releases/OldVsn/start.boot。

（17）调用 `systools:make_relup("rel/releases/Vsn/RelName-Vsn",["rel/releases/OldVsn/RelName- OldVsn"],["rel/releases/DownVsn/RelName- DownVsn"])` 生成 relup 文件。

（18）把 relup 文件移到目录 rel/releases/Vsn 中。

（19）调用 `systools:make_tar("rel/releases/Vsn/RelName-Vsn")` 生成新发布的 tar 文件。

（20）把 tar 文件移到目录 rel/releases/中。

（21）进入运行第一个发布版本的 Erlang shell。

（22）调用 `release_handler:unpack_release("NameOfRel-Vsn")`。

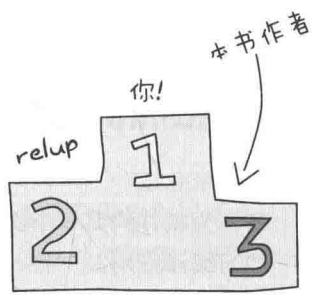
（23）调用 `release_handler:install_release(Vsn)`。

（24）调用 `release_handler:make_permanent(Vsn)`。

（25）确信一切正常。如果有问题，通过安装老版本进行回退。

你可以写些脚本来自动化这些步骤。

再说一次，发布升级是 OTP 中非常混乱的一部分内容——也非常难以理解。你可能会遇到许多从未遇到过的错误，这些错误可能比上面介绍的都更难以理解。这个过程中暗含了一些对你情况的假设，创建发布时所选择的工具也会影响做事的方式。你甚至可能受到诱惑，想直接使用 sys 模块中的函数编写自定义的升级代码！也可能会想使用类似 rebar 这样的工具来自动化那些比较痛苦的步骤。不管怎样，本章所介绍的内容和示例已经是作者（一个有时喜欢用第三人称来称呼自己的人）所能教授的最好的知识了。



如果不是必须使用 relup，而是可以通过其他方法来升级应用，那么我推荐这样做。据说，爱立信公司的一些部门使用了 relup，他们花费在 relup 测试上的时间和应用测试上的一样多。Relup 这种工具主要适用于那些绝对不能停止服务的产品。在需要使用它们时，你自然就会知道，因为你已经做好了承受它所带来的麻烦的准备（要学着喜欢循环论证！）。当真有那样的需求时，relup 还是非常有用的。

接下来，我们去学习一些 Erlang 中更加友好的特性，如何？在第 23 章中，我会给大家介绍用 Erlang 进行套接字编程。

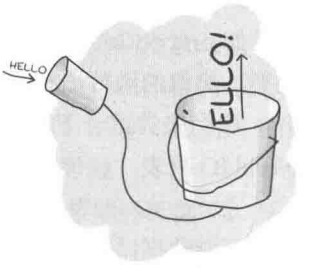
第 23 章

套接字编程

目前为止，我们都是在 Erlang 内部编写有趣的程序，基本上没有和外部世界通信。即使有过通信，也只是读取某个临时的文本文件。虽说自娱自乐也挺有趣的，不过是时候走出我们的藏身之处和外边的世界说说话了。套接字（socket）就是实现这个目标的一种方法。

本章将介绍套接字编程中的 3 部分内容：IO 列表、UDP 套接字以及 TCP 套接字。

23.1 IO 列表



你已经知道，可以使用字符串（整数列表）或者二进制串（存有数据的二进制数据结构）来表示文本数据。在网络上传递像“Hello World”之类的信息，可以使用字符串（"Hello World"）或者二进制串（<<"Hello World">>）——它们长得很像，效果也相仿。

它们之间的区别在于数据组装的方式。

字符串有点像一个整数链表。对于每个字符，除了自身占用存储空间外，还需要保存一个指向剩余列表的链接。此外，如果想向列表中增加元素——不管是中间还是末尾——都必须要遍历整个列表直到需要修改的那个位置，然后才能增加元素。不过，向列表中前置增加（prepend）元素时，有所不同：

```
A = [a]
B = [b|A] = [b,a]
C = [c|B] = [c,b,a]
```

进行列表前置操作时，A、B 或者 C 中的内容都不需要改写。可以把 C 的表示形式看作[c,b,a]、[c|B]或者[c|[b|[a]]]（还有其他形式）。从最后一种形式中可以看出，列表末尾处的 A 的形状和其定义时的形状完全一样，B 也是如此。下面是追加（append）操作时的形状变化：

```
A = [a]
```

```
B = A ++ [b] = [a] ++ [b] = [a|[b]]
C = B ++ [c] = [a|[b]] ++ [c] = [a|[b|[c]]]
```

你看到那些改写操作了吗？当创建 B 时，需要改写 A。在生成 C 时，必须要改写 B（包括它所含有的[a|...]部分）。如果用同样的方式再定义一个 D，那么就需要改写 C。如果字符串很长，那么这种方式就非常低效，并且会产生大量的垃圾，需要 Erlang VM 进行清理。

如果使用二进制串，情况就会好很多：

```
A = <<"a">>
B = <<A/binary, "b">> = <<"ab">>
C = <<B/binary, "c">> = <<"abc">>
```

在这个例子中，二进制串知道自己的长度，数据可以在常量时间内结合在一起。看起来不错——比列表要好很多，也更加紧凑。因此，以后在使用文本时，我们会尽量使用二进制串的形式。

不过，二进制串也有一些缺点。我们只能以特定的方式处理数据，并且二进制串的修改、分割等操作也有开销。此外，我们有时会写一些代码，其中交替使用了字符串、二进制串和单字符列表。频繁地在这些类型之间进行转换非常麻烦。

对于这些问题，IO 列表是我们需要的解决方案。IO 列表是一种奇特的数据结构。它们是包含字节（从 0 到 255 的整数）、二进制串或者其他 IO 列表的列表。这意味着，以 IO 列表为参数的函数可以接受像这样的值：[$H, $e, [$l, <<"lo">>, " "]、[[["W","o"], <<"rl">>]] | [<<"d">>]]。此时，Erlang VM 会根据需要把列表展平，得到字符序列 *Hello World*。

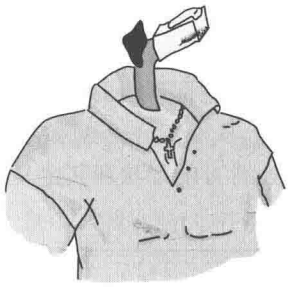
以 IO 列表为参数的函数有哪些呢？绝大多数向外输出数据的函数基本上都接受 IO 列表类型，io 模块、file 模块以及 TCP 和 UDP 套接字中的所有函数都可以处理它们。还有一些库函数，如 unicode 模块中的一些函数，以及 re（正则表达式）模块中的所有函数也都能处理这种类型，类似的函数还有许多。

我们来尝试一下，在 shell 中用 io:format("~s~n", [IoList])打印一下前面的 Hello World IO 列表，应该可以正常工作。

总之，IO 列表是一种非常巧妙的字符串构建方法，在输出的数据需要动态构建时，它能规避不可变数据结构所导致的问题。

23.2　UDP 和 TCP：伙伴协议

Erlang 中提供的第一种套接字类型是基于用户数据报协议（UDP）的。UDP 协议构建在 IP 层之上，它提供了一些有用的抽象，如端口号。



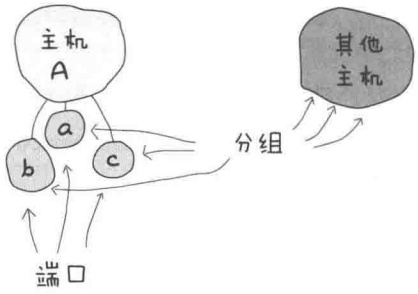
UDP 是一种*无连接*协议。发往 UDP 端口的数据会被分成一些小的、无标记的、不带会话标识的片段（数据报），并且也不保证所收到的数据片段的顺序和它们的发送顺序一致。事实上，也不保证你一定能够收到他人发送给你的包。因此，人们只会在如下一些场合中选择使用 UDP：

- 包不大；

- 偶尔的包丢失，后果不严重；
- 没有太多复杂的包交换；
- 低延迟是必需的。

UDP 和像传输控制协议（TCP）这样的面向连接的协议正好相反，面向连接的协议会完成丢包重发、重新排序、保证多个发送者和接收者之间的会话隔离等工作。TCP 可以达成可靠的信息交换，不过在会话建立阶段要慢一些、重量一些。UDP 要快一些，但是可靠性要差一些。请根据自己的需要谨慎选择。

在 Erlang 中使用 UDP 比较简单。只要用给定的端口初始化套接字，然后就可以用它收发数据了：



这有点像你住所外边的一组邮箱（每个邮箱是一个端口），每个邮箱都可以接收一些写有简短消息的小纸片。这些纸片上的内容五花八门，从“你穿这样的裤子很好看”到“这个纸片就来自这所房子”！当某些消息很大，一张纸片上写不下时，会写在多张纸片上一起投递到邮箱中。你的工作是把这些纸片按照合理的顺序排列起来，然后驱车赶到某处住所，把回应放入住所边上的邮箱中。如果消息完全是通知性的（“嗨，你的门没锁”）或者无关紧要的（“你穿的是什么衣服？——Ron”），那就没啥问题，你可以用一个邮箱来接收所有这些询问。不过，如果消息很复杂，你可能会让每个会话专用一个端口，对吧？啊，不！可以使用 TCP！

如果使用 TCP，协议就是有状态、面向连接的了。在发送消息之前，必须进行握手。握手的意思是说，某个要投递消息的人先向某个邮箱（和刚刚对 UDP 的类比相似）发送一条类似“嗨，伙计，这是一个来自 IP 地址 94.25.12.37 的呼叫。想聊聊吗？”这样的消息。然后你回应道“当然可以。请用一个数字 N 来标记你的消息，此后每次发送消息时，都递增这个数字”。在这之后，当你和 IP 地址 94.25.12.37 之间想互相通信时，就能够以明确的方式来排序多张纸片、请求重发丢失的纸片、回应收到的纸片了。基于这种方式，你只使用一个邮箱（或者端口）就可以保持所有的通信正常进行。这是 TCP 非常漂亮的一个特性。这会增加一些开销，不过它可以保证所有的消息都是有序的，并被真正地投递到目的地。

如果你不喜欢这种类比，不要失望——我们现在直奔主题，看看如何在 Erlang 中使用 TCP 和 UDP 套接字。

23.2.1 UDP 套接字

UDP 只有很少的几个基本操作：创建套接字、发送消息、接收消息以及关闭连接。这些操作

间的使用关系如图：

无论如何，都要先打开一个套接字。这项工作可以通过调用 gen_udp:open/1-2 完成。最简单的调用形式是{ok, Socket} = gen_udp:open(PortNumber)。

端口号是 1 到 65 535 之间的任意整数。

位于 0 到 1 023 之间的端口号，是*知名端口号*。大多数情况下，操作系统不允许去监听一个知名端口，除非具有管理员权限。

从 1 024 到 49 151 的端口号是已登记端口号。虽然有些已登记端口被一些著名的服务（参见 http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xml）使用，但是，此类端口通常无需允许就可以自由使用。

剩余的端口号是*动态*或者*私有*端口。它们常常被用作*暂态*（ephemeral）端口，当连接发起者连接到一个特定服务时，会为这个会话随机分配一个暂态端口。

在我们的实验中，会使用安全些的端口号，如 8 789，这类端口不太可能被其他应用占用。

在实验之前，先来学习一下 gen_udp:open/2 函数。这个函数的第二个参数是一个选项列表，指定想要接收的数据类型（list 或者 binary）以及接收的方式：以消息方式（{active, true}），还是以函数调用返回结果的方式（{active, false}）。还有另外一些选项，如套接字是否被设置成 IPv4(inet4)或者 IPv6(inet6)、UDP 套接字是否用于广播信息({broadcast, true|false})、缓冲区的大小等。现在，我们只使用简单的选项，随后你可以自行查阅其他的套接字选项。

现在，我们来打开一个 UDP 套接字。首先启动一个 Erlang shell：

```
1> {ok, Socket} = gen_udp:open(8789, [binary, {active,true}]).
{ok,#Port<0.676>}
2> gen_udp:open(8789, [binary, {active,true}]).
{error,eaddrinuse}
```

在第一个函数调用中，我们打开了一个套接字，要求它返回二进制数据，并告诉它我们希望它进入主动(active)模式。可以看到，这个调用返回了一个新数据结构：#Port<0.676>。它代表了我们刚刚打开的套接字。套接字的使用方式和进程很像。甚至可以与之建立链接，这样，当发生崩溃时，失败会传播至套接字，导致其关闭！

第二个函数调用试图再次打开同一个套接字，这是不允许的。因此，返回了{error, eaddrinuse}。还好，第一个套接字依旧处于打开状态。

接下来，我们再启动一个 Erlang shell。在这个 shell 中，用一个不同的端口号打开第二个 UDP 套接字：

```
1> {ok, Socket} = gen_udp:open(8790).
{ok,#Port<0.587>}
2> gen_udp:send(Socket, {127,0,0,1}, 8789, "hey there!").
ok
```

啊，一个新函数！在第二个调用中，我们使用 gen_udp:send/4 发送消息（这个描述性名字真是妙不可言）。其参数依次为 gen_udp:send(OwnSocket, RemoteAddress, RemotePort, Message)。参数 RemoteAddress 可以是一个含有域名（"example.org"）的字符串或者原子、一个描述 IPv4 地址的 4 元素元组或者一个描述 IPv6 地址的 8 元素元组。接下来，我们指定了接收方的端口号（也就是我们将要投递纸片的邮箱），然后是消息，消息可以是字符串、二进制数据或者 IO 列表。

消息到底有没有被发送出去？回到第一个 shell，刷新一下数据：

```
3> flush().
Shell got {udp,#Port<0.676>,{127,0,0,1},8790,<<"hey there!">>}
ok
```

好极了。打开第一个套接字的进程，会收到形如{udp, Socket, FromIp, FromPort, Message}的消息。有了这些字段，我们就能够知道消息来自哪里、经过了哪个套接字以及内容是什么。

我们已经介绍了如何打开套接字、如何发送数据，以及如何以主动模式接收数据。那么被动（passive）模式是什么样呢？为此，我们要关闭第一个 shell 中的套接字，并重新打开一个：

```
4> gen_udp:close(Socket).
ok
5> f(Socket).
ok
6> {ok, Socket} = gen_udp:open(8789, [binary, {active,false}]).
{ok,#Port<0.683>}
```

在这个 shell 中，我们关闭了套接字，解除了 Socket 变量的绑定，然后把它重新绑定到我们新打开的套接字上，这次使用的是被动模式。在发送消息前，先试试如下命令：

```
7> gen_udp:recv(Socket, 0).
```

shell 卡住了。这里调用的函数是 recv/2。这个函数会从被动模式的套接字中轮询（poll）消息。这里的值 0 是所希望的消息的长度。有趣的是，这个长度值在 gen_udp 中完全被忽略了。（gen_tcp 中有个类似的函数，它会使用这个值。）总之，如果不向这个套接字发送消息，recv/2 就永远不会返回。回到第二个 shell，发送一条新消息：

```
3> gen_udp:send(Socket, {127,0,0,1}, 8789, "hey there!").
ok
```

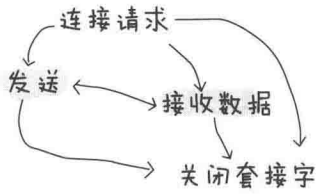
第一个 shell 中会显示出函数 recv/2 的返回值{ok,{{127,0,0,1},8790,<<"hey there!">>}}。如果不想一直等待该怎么办呢？增加一个超时值即可：

```
8> gen_udp:recv(Socket, 0, 2000).
{error,timeout}
```

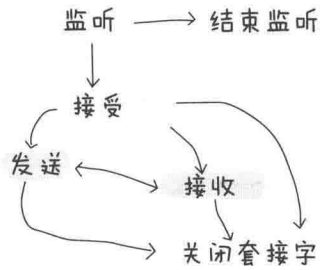
关于 UDP 的内容基本上讲完了。没骗你，是真的！

23.2.2　TCP 套接字

虽然 TCP 套接字和 UDP 套接字的接口函数有很大一部分是相同的，但是这些接口函数的工作方式却有重大不同。其中最大的区别是，TCP 的客户端和服务器是两种完全不同的东西。客户端的行为方式如下图所示：



服务器侧的工作方式则遵循如下模式：



看起来很奇怪，对吧？客户端的行为和 gen_udp 有点像。连接一个端口，收发数据，然后关闭。但是在服务器端，增加了一个新的工作状态：监听（listening）。这是因为 TCP 在建立会话时需要这样做。

首先，打开一个新 shell，调用 gen_tcp:listen(Port, Options) 启动一个监听套接字：

```
1> {ok, ListenSocket} = gen_tcp:listen(8091, [{active,true}, binary]).
{ok,#Port<0.661>}
```

监听套接字只负责等待连接请求。可以看到，我们使用的选项和 gen_udp 中使用的类似。这是因为，对于所有的 IP 套接字来说，绝大部分的选项都类似。也有几个专用于 TCP 套接字的选项，如连接的后备队列（backlog）（{backlog, N}）、是否保活（{keepalive, true|false}）以及包的成帧格式（{packet, N}），其中 N 是包头的长度，以字节为单位，包头会被自动解析和去除，等等。

打开监听套接字之后，任何进程（可以多于一个）都可以使用这个监听套接字，并进入“接受连接”状态，然后一直锁定在这个状态直到有客户端发起连接请求：

```
2> {ok, AcceptSocket} = gen_tcp:accept(ListenSocket, 2000).
** exception error: no match of right hand side value {error,timeout}
3> {ok, AcceptSocket} = gen_tcp:accept(ListenSocket).
** exception error: no match of right hand side value {error,closed}
```

糟糕。超时，然后崩溃了。当和监听套接字关联的 shell 进程死亡时，它会被关闭。再来一次，这次不设置 2 s（2000 ms）的超时：

```
4> f().
ok
5> {ok, ListenSocket} = gen_tcp:listen(8091, [{active, true}, binary]).
{ok,#Port<0.728>}
6> {ok, AcceptSocket} = gen_tcp:accept(ListenSocket).
```

此时，进程卡住了。很好！打开一个新的 shell：

```
1> {ok, Socket} = gen_tcp:connect({127,0,0,1}, 8091, [binary, {active,true}]).
{ok,#Port<0.596>}
```

这个调用使用了和往常一样的选项，如果不想一直等待，可以在最后增加一个 `Timeout` 参数。第一个 shell 中应该会返回`{ok, SocketNumber}`。此后，被接受的套接字和客户套接字之间就可以进行一对一的通信了，和使用 `gen_udp` 时类似。进入第二个 shell，给第一个 shell 发送消息：

```
3> gen_tcp:send(Socket, "Hey there first shell!").
ok
```

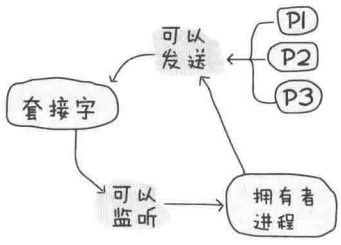
在第一个 shell 中，输入如下内容：

```
7> flush().
Shell got {tcp,#Port<0.729>,<<"Hey there first shell!">>}
ok
```

两个套接字可以用同样的方式发送消息，并且都可以用 `gen_tcp:close(Socket)` 关闭。注意，关闭一个被接受的套接字只会导致它自己被关闭，而关闭一个监听套接字则会关闭所有和其关联的被接受套接字。

Erlang 中关于 TCP 套接字的内容基本上讲完了！是真的吗？

啊，当然还有些其他内容。如果你曾经有过一点套接字编程经验，也许注意过套接字拥有权的这个概念。拥有权的意思是，无论是 UDP 套接字、TCP 客户端套接字还是被接受的 TCP 套接字，任何已有进程都可以通过它们发送消息，但是只有创建这个套接字的进程才能够从它们那里读取收到的消息：



这不是非常实用，对吧？这意味着，为了中继消息，套接字拥有者进程必须要一直活着，即使我们在业务层面并不需要它。如果能够像下面一样行事不是会更好一些吗？

```
1.进程 A 启动一个套接字
2.进程 A 发送一个请求
3.进程 A 创建进程 B，并把套接字传递给 B
4a.进程 A 把套接字所有权移交给进程 B          4b.进程 B 处理响应消息
5a.进程 A 发送一个请求                        5b.进程 B 继续处理响应消息
6a.进程 A 创建进程 C，并把套接字传递给 C      6b. ...
...
```

这里，进程 A 负责发起一组请求，但是每个新创建的进程负责等待回应、处理回应。这样，进程 A 就非常聪明地把任务委托给一个新进程去执行。这里的难点在于套接字拥有权的移交。

诀窍如下：gen_tcp 和 gen_udp 模块都有一个名为 controlling_process(Socket, Pid) 的函数。这个函数只能被套接字的当前拥有者调用。调用这个函数就等同于拥有者进程告诉 Erlang，“你知道吗？马上让这个 Pid 接管我的套接字。我放弃”。此后，作为函数参数的 pid 就成为那个可以从套接字中读取或者接收消息的进程。就这样。

23.3　使用 Inet 进行更多的控制

到目前为止，我们介绍了如何打开套接字、如何通过它们发送消息、如果更改拥有权等。我们也学习了如何用被动模式和主动模式接收消息。在 UDP 的例子中，当我们想从主动模式切换到被动模式时，需要重新创建套接字、更新变量才能完成切换。这非常不实用，尤其是在使用 TCP 的过程中，当我们需要去中断一个活动会话时，会很想进行这种切换。

还好，在 Erlang 中有一个名为 inet 的模块，它负责处理 gen_tcp 和 gen_udp 套接字所有的公共操作。对于我们当前的问题——在主动模式和被动模式之间切换——可以使用 inet:setopts(Socket, Options) 函数进行解决。Options 参数是一个列表，其中包含创建套接字时使用的选项。

> **警告**　注意！有一个叫 inet 的模块，还有一个叫 inets 的模块。我们这里需要的是 inet 模块。inets 是一个 OTP 应用，其中包含一组实现好的服务和服务器（包括 FTP、TFTP、HTTP 等）。分辨它们有一个简单的方法：inets 是构建于 inet 之上的服务，或者如果你更喜欢的话，可以把 inets 看作 inet+s(ervices)。

启动一个 shell，让它成为一个 TCP 服务器：

```
1> {ok, Listen} = gen_tcp:listen(8088, [{active,false}]).
{ok,#Port<0.597>}
2> {ok, Accept} = gen_tcp:accept(Listen).
```

再启动一个 shell，输入如下内容：

```
1> {ok, Socket} = gen_tcp:connect({127,0,0,1}, 8088, []).
{ok,#Port<0.596>}
2> gen_tcp:send(Socket, "hey there").
ok
```

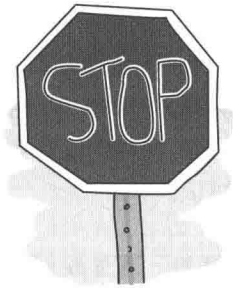
然后，回到第一个 shell，套接字应该接受成功。更新一下 shell 消息邮箱，看看是否收到

消息：

```
3> flush().
ok
```

当然不会收到消息，因为使用的是被动模式。我们来修正一下：

```
4> inet:setopts(Accept, [{active, true}]).
ok
5> flush().
Shell got {tcp,#Port<0.598>,"hey there"}
ok
```



收到了！有了对套接字主动模式和被动模式的完全掌控，我们更强大了。但是，如何在主动模式和被动模式之间选择呢？

一般来讲，如果期望立即得到消息，那么被动模式要更快一些。Erlang 不需要把消息放到进程邮箱中，进程也不需要去做扫描这个邮箱、获取消息之类的事情。使用 `recv` 会更高效一些，尤其是在要接收的数据大小未知的情况下。不过，`recv` 会把进程的处理从事件驱动方式变为主动轮询方式。如果进程需要在套接字和其他 Erlang 代码间扮演中间人的角色，那么这种改变会让阻塞和消息处理的逻辑变得有些复杂。

对于这种情况，切换到主动模式是一个好办法。如果包是以消息方式发送的，只需要执行一个 `receive` 等待（或者 `gen_server` 中的 `handle_info` 函数），然后处理它们就行了，就和处理其他任何消息一样。除了速度之外，主动模式还有一个和速度限制有关的缺点。

如果 Erlang 盲目地接受所有来自外部世界的数据，并把它们转换成消息，那么 VM 外面的人就很容易用大量的数据淹没并杀死它。被动模式有一个优点，它可以限制消息何时、以何种方式被放入 Erlang VM 中，把消息阻塞、排队以及丢弃的任务委托给低层实现。

那么，如果既想要主动模式的语义，又想要被动模式的安全性，该怎么办呢？我们可以使用 `inet:setopts/2` 在被动模式和主动模式之间进行快速切换，不过，这样做很可能会引发竞争条件。还有一种模式可供使用，它叫*单次主动*（active once），对应的选项是 `{active, once}`。我们来试一下，看看它是如何工作的。

保持前面运行服务器的 shell 打开，输入如下内容：

```
6> inet:setopts(Accept, [{active, once}]).
ok
```

现在进入客户 shell，再调用两次 `send/2`：

```
3> gen_tcp:send(Socket, "one").
ok
4> gen_tcp:send(Socket, "two").
ok
```

然后回到服务器 shell，输入如下命令：

```
7> flush().
```

```
Shell got {tcp,#Port<0.598>,"one"}
ok
8> flush().
ok
9> inet:setopts(Accept, [{active, once}]).
ok
10> flush().
Shell got {tcp,#Port<0.598>,"two"}
ok
```

看到了吗？直到再次设置了{active, once}，才把包含"two"的包转变成一个消息，这意味着，套接字又回到了被动模式。因此，单次主动模式能让我们以安全的方式在主动模式和被动模式之间来回切换。它提供了非常好的语义和安全性。

inet 模块还有其他一些有用的函数：读取统计值、获取当前主机信息、审查套接字等。inet 模块的文档中包含着更多其所提供功能的细节信息。

嗯，关于套接字，需要了解的基本上就这么多了。接下来，我们要在实践中应用一下这些知识。

> **注意**　在互联网这个广阔的天地中，有许多开源的库，可以处理各种各样的协议：HTTP、ZeroMQ、UNIX 原生套接字等。不过，标准 Erlang 发布中提供的主要协议只有两种：TCP 和 UDP 套接字。其中也包含了一些 HTTP 服务器和协议解析代码，不过它们的实现不是最高效的。

23.4　重新审视 sockserv

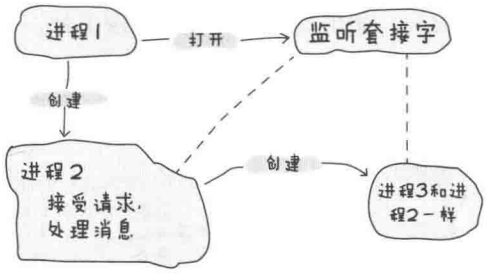
在这个例子中，我不会引入过多的新代码。我们会回顾一下第 22 章中介绍的 Process Quest 中的 sockserv 服务器，这是一个完全可用的服务器。我们会学习一下如何在 OTP 监督树内用 gen_server 完成对连接请求的处理。

下面的代码是 TCP 服务器一种简单的实现：

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

start_server(Port) ->
    Pid = spawn_link(fun() ->
        {ok, Listen} = gen_tcp:listen(Port, [binary, {active, false}]),
```

```
        spawn(fun() -> acceptor(Listen) end),
        timer:sleep(infinity)
    end),
    {ok, Pid}.

acceptor(ListenSocket) ->
    {ok, Socket} = gen_tcp:accept(ListenSocket),
    spawn(fun() -> acceptor(ListenSocket) end),
    handle(Socket).

%% 把收到的内容发送回去。
handle(Socket) ->
    inet:setopts(Socket, [{active, once}]),
    receive
        {tcp, Socket, <<"quit", _/binary>>} ->
            gen_tcp:close(Socket);
        {tcp, Socket, Msg} ->
            gen_tcp:send(Socket, Msg),
            handle(Socket)
    end.
```

为了理解上述代码的工作原理，我们可以用图示的方式来表达：



函数 start_server 打开一个监听套接字、创建一个接受器进程，然后就永远闲着。空闲等待是必要的，因为监听套接字和打开它的进程绑定在一起，只要我们想接受连接请求，就必须要让这个进程一直活着。接受器进程会等待接受连接请求。一旦收到了一个连接请求，接受器进程就会启动一个新的、同样的进程，并将监听套接字分享给这个新进程。然后，接受器进程可以继续执行，处理请求，而新进程则会等待接受连接请求。每个处理进程都会把收到的消息回送回去，直到收到了以"quit"作为起始的消息，此时会关闭连接。

> **注意** 模式<<"quit", _/binary>>的意思是，先匹配包含字符 q、u、i 和 t 的二进制字符串，剩余的二进制数据（_）我们不关心。

在 Erlang shell 上调用 naive_tcp:start_server(8091)启动这个服务器。然后，打开一个 telnet 客户端（记住，从技术角度上讲，telnet 客户端不用于原生 TCP，不过，如果只是测试我们的服务器，用它们就行了，无需自己去编写一个），连接到 localhost，会看到如下

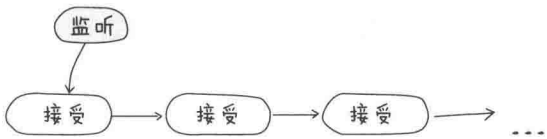
情况：

```
$ telnet localhost 8091
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
hey there
hey there
that's what I asked
that's what I asked
stop repeating >:(
stop repeating >:(
quit doing that!
Connection closed by foreign host.
```

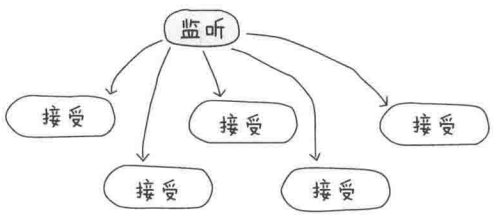
太好了。现在我们可以创建一个新公司，名字叫作 People Inc.，并用我们的服务器构建几个社交网络。不过，正如模块名字所暗示的，这是一个幼稚的实现。代码很简单，并且也没有考虑并行处理。如果请求都是逐个到来的，那么这个简单的服务器是可以应对的。但是，如果现在有 15 个人，他们想同时连接到服务器，会发生什么呢？

此时，每次只能回复一个连接请求，其工作流程为：进程首先等待连接，对它做些初始化，然后创建一个新的接受器进程。所发起的第 15 个请求必须要等到前 14 个连接都被初始化后才有机会得到请求与服务器通信的权利。如果是生产环境中的服务器，每秒的请求数大约在 500 到 1 000 个左右！这种做法基本上不能用。

我们需要做的是把串行的工作流



变成并行程度更高的工作流



通过让多个接受器进程同时就绪待命，可以大大减少请求回复的延迟。

现在，我们不会去编写另外一个演示示例，而是直接使用第 22 章中的 sockserv-1.0.1。能基于真正的 OTP 组件和真实世界中的实践进行一些探索会比较好。事实上，sockserv 的总体模式结构和 cowboy 服务器（毋庸置疑，cowboy 要比 sockserv 可靠得多）以及 etorrent 种子客户端和服务器的完全一样。

我们将采用自顶向下的方式来构建 Process Quest 的 sockserv。需要的模式结构是：一个监督者，下面有多个工作者。根据上面给出的并行处理图示，监督者会持有监听套接字，并将这个监听套接字分享给所有工作者进程，工作者进程负责接受连接。

如何实现一个能和所有工作者进程共享东西的监督者呢？用常规的方式是无法实现的。无论使用 one_for_one、one_for_all 还是 rest_for_all 重启策略，所有的子进程都是完全独立的。你可能会自然想到用某些全局状态——让一个注册进程持有监听套接字，并把它转交给工作者进程。你必须要抑制住这种本能想法和小聪明。使用原力（回顾一下前面第 17 章的内容，在第 17 章中介绍了监督者）。给你 2 min 时间想出一个解决方案（是否 2 min 你自己说了算，我完全信任你）。

关键就在于使用 simple_one_for_one 监督者。因为 simple_one_for_one 监督者会和所有子进程共享子进程规格说明，我们只要把监听套接字放到其中，让所有的子进程访问就行了！

下面就是这个极其漂亮的监督者的实现代码：

```
%%% 管理所有接受器进程的监督者
-module(sockserv_sup).
-behavior(supervisor).

-export([start_link/0, start_socket/0]).
-export([init/1]).

start_link() ->
    supervisor:start_link({local, ?MODULE}, ?MODULE, []).

init([]) ->
    {ok, Port} = application:get_env(port),
    %% 把套接字设置为{active_once}模式
    %% 更多细节，请查看 sockserv_serv 中的注释
    {ok, ListenSocket} = gen_tcp:listen(Port, [{active,once}, {packet,line}]),
    spawn_link(fun empty_listeners/0),
    {ok, {{simple_one_for_one, 60, 3600},
          [{socket,
           {sockserv_serv, start_link, [ListenSocket]}, % 传递监听套接字!
           temporary, 1000, worker, [sockserv_serv]}
          ]}}.

start_socket() ->
    supervisor:start_child(?MODULE, []).

%% 创建了 20 个接收器进程，这样可以同时接受多个连接请求，避免串行化
%% 理想情况下，要一直对活动接收器进程进行计数，确保在进程被杀死太多时，
%% 不会出现糟糕的情况
empty_listeners() ->
    [start_socket() || _ <- lists:seq(1,20)],
    ok.
```

这段代码做了什么呢？其中有标准的 start_link/0 和 init/1 函数。可以看到，sockserv 使用了 simple_one_for_one 重启策略，并且子进程规格说明中传递了 ListenSocket。通过

start_socket/0 启动的每个子进程都会自动从参数中得到这个套接字。太神奇了！

不过，仅仅得到这个套接字还不够。我们希望应用能够尽可能快地处理请求。因此，我们增加了调用 spawn_link(fun empty_listeners/0)。函数 empty_listeners/0 会启动 20 个工作者进程，阻塞着等待到来的连接请求。我们用它作为参数调用 spawn_link/1，原因很简单：监督者进程正处于 init/1 阶段，不能回复任何消息。如果在 init 函数中调用自己，进程就会进入死锁状态，永远无法继续运行。正因如此，才需要一个外部进程。

注意　在上面的代码片段中，我们把{packet, line}选项传递给了 gen_tcp。这个选项会把收到的所有包都分割成独立的行，并以行为单位进行排队（行的结尾符仍保存在所收到的字符串中）。针对我们例子中 telnet 客户端的使用方式，这种做法可以避免出现一些常见错误。

嗯，这就是最复杂的部分。现在，我们可以专注于工作者进程了。

回想一下第 22 章中的 Process Quest 会话，我们是按照如下步骤和游戏交互的。

（1）用户连接到服务器。

（2）服务器要求用户输入角色名。

（3）用户把角色名发送给服务器。

（4）服务器给出建议的角色类型。

（5a）用户拒绝，回到第 4 步。

（5b）用户接受，进入第 6 步。

（6）游戏一直给玩家发送事件，直到……

（7）用户发送 quit 命令给服务器或者套接字被强制关闭。

这意味着，服务器进程接收两种类型的输入：一种来自 Process Quest 应用，另外一种来自用户。来自用户的数据是通过套接字输入的，因此会在 gen_server 的 handle_info/2 函数中进行处理。来自 Process Quest 的数据能够以一种受控的方式进行发送，这种情况下，由 handle_cast 函数进行处理是合理的。

我们先来编写服务器模块：

```
-module(sockserv_serv).
-behavior(gen_server).

-record(state, {name,  % 玩家姓名
                next,  % 下个步骤，初始化时使用
                socket}). % 当前套接字

-export([start_link/1]).
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         code_change/3, terminate/2]).
```

这是一个非常标准的 gen_server 回调模块。唯一不同的是：我们定义了一个包含角色名字、套接字以及 next 字段的状态记录。next 字段中可以存放任何和服务器状态有关的临时信

息。这个模块也完全可以使用 gen_fsm 实现。

服务器的启动代码如下：

```
-define(TIME, 800).
-define(EXP, 50).

start_link(Socket) ->
    gen_server:start_link(?MODULE, Socket, []).

init(Socket) ->
    %% 为进程设置适当的随机种子
    <<A:32, B:32, C:32>> = crypto:rand_bytes(12),
    random:seed({A,B,C}),
    %% 因为接受连接是阻塞调用，所以不能在此调用，
    %% 把它转到服务器的循环中
    gen_server:cast(self(), accept),
    {ok, #state{socket=Socket}}.

%% 我们不需要 handle_call
handle_call(_E, _From, State) ->
    {noreply, State}.
```

这里定义的两个宏（?TIME 和?EXP）是两个特殊的参数，一个用来设定玩家动作之间的基线延时（800 ms），另一个用来设定升到第二级需要的经验值总数（50，此后每升一级此值翻倍）。

你应该注意到，start_link/1 函数接收了一个套接字。这就是从 sockserv_sup 传过来的监听套接字。

随机种子的那部分代码是为了保证每个进程都被设置了适当的种子，供以后生成角色统计值时使用。否则，多个进程会共享同一个默认种子值，这不是我们希望的。我们之所以在 init/1 函数中进行初始化，而不是在使用随机数的库中进行，是因为种子是存储到进程中的（讨厌——可变状态!），我们不希望每次调用库函数都去设置新的种子。

这段代码的重点在于，我们给自己投掷（cast）了一条消息。这是因为，gen_tcp:accept/1-2 函数是一个阻塞操作，并且所有的 OTP init 函数都是同步的。如果花 30 s 时间去等待接受一个连接，那么启动这个进程的监督者也要被挂起 30 s。因此，我们给自己投掷了一条消息，然后把监听套接字放入状态记录的套接字字段中。

保持冷静

如果你阅读其他人编写的代码，常常会看到程序员以 now() 的返回值作为参数调用 random:seed/1。now() 是一个不错的函数，因为它返回的是单调时间（一直增加，两次调用不会重复）。不过，对于 Erlang 中使用的随机算法，它是一个糟糕的种子值。因此，更好的方法是使用 crypto:rand_bytes(12) 来生成 12 个密码级安全的随机字节（想更安全的话，可以使用 crypto:strong_rand_bytes(12)）。通过<<A:32, B:32, C:32>>这种方法，我们把这 12 字节强制转换成 3 个整数传给 random:seed/1。

扯的够多了。现在我们来接受连接：

```
handle_cast(accept, S = #state{socket=ListenSocket}) ->
    {ok, AcceptSocket} = gen_tcp:accept(ListenSocket),
    %% 记住，  你本尘土，必归于尘土
    %% 在这个应用中，我们想一直保持给定数目的子进程
    sockserv_sup:start_socket(), % 一个新接受器进程诞生了，赞美主
    send(AcceptSocket, "What's your character's name?", []),
    {noreply, S#state{socket=AcceptSocket, next=name}};
```

这个函数接受了连接，接着启动一个替代的接受器进程（这样就可以一直有 20 个就绪的接受器进程处理新连接请求），然后用所接受的套接字替换掉之前存储的 ListenSocket，并在 next 字段中指定希望从套接字接收的下一条消息，此处是名字消息。

在函数返回之前，我们使用 send 函数向客户端发送了一个问题，send 函数的定义如下：

```
send(Socket, Str, Args) ->
    ok = gen_tcp:send(Socket, io_lib:format(Str++"~n", Args)),
    ok = inet:setopts(Socket, [{active, once}]),
    ok.
```

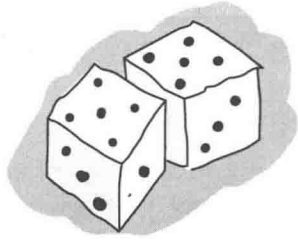
小把戏！因为我们希望对收到的每条消息都做出回应，因此把设置单次主动模式这个例行工作放在了 send 函数中，并且也增加了换行符。这种做法其实是把偷懒行为锁定到了一个函数中。这可能不是最优的设计，因为这种做法意味着，如果不想永远收不到消息，就必须回应收到的每条消息（不发送回应消息，会一直待在{active, false}模式），此外，目前{active, once}的限速效果本质上是由我们发送数据的速度决定的，而非处理数据的速度。正如我说的，这是锁定在一个函数中的偷懒行为。

我们已经完成了步骤 1 和 2，现在需要等待来自套接字的用户输入：

```
handle_info({tcp, _Socket, Str}, S = #state{next=name}) ->
    Name = line(Str),
    gen_server:cast(self(), roll_stats),
    {noreply, S#state{name=Name, next=stats}};
```

我们不知道 Str 字符串的内容，不过这不要紧，因为通过状态记录的 next 字段可以得知收到的是一个名字。对于这个演示应用，我们期望用户使用的是 telnet 客户端，因此所收到的文本中会包含行结束符。函数 line/1 会把行结束符去掉，其实现如下：

```
%% 去掉空格，忽略行后的任何内容。
%% 简化处理 telnet 消息。
line(Str) ->
    hd(string:tokens(Str, "\r\n ")).
```



收到名字后，我们把它保存起来，然后给自己投掷了一条生成玩家角色统计信息的消息（roll_stats），也就是接下来的一个步骤。

> **注意** 如果你查看了源文件，就会发现代码中不是用完整的消息格式去匹配，而是使用了一个简短的宏?SOCK(Var)。这个宏被定义为-define(SOCK(Msg), {tcp, _Port, Msg})，对于像我这样的懒人，这是一种匹配字符串的快捷方法，还可以少敲几次键盘。

roll_stats 消息会被下面的 handle_cast 子句处理：

```
handle_cast(roll_stats, S = #state{socket=Socket}) ->
    Roll = pq_stats:initial_roll(),
    send(Socket,
         "Stats for your character:~n"
         " Charisma: ~B~n"
         " Constitution: ~B~n"
         " Dexterity: ~B~n"
         " Intelligence: ~B~n"
         " Strength: ~B~n"
         " Wisdom: ~B~n~n"
         "Do you agree to these? y/n~n",
         [Points || {_Name, Points} <- lists:sort(Roll)]),
    {noreply, S#state{next={stats, Roll}}};
```

模块 pq_stats 中含有随机产生统计值的函数，整个 handle_cast 子句只是用来输出统计值。~B 格式化参数的意思是：希望打印出一个整数。放置在状态记录 next 字段中的内容有点多。因为我们询问了玩家是否接受这个角色，所以需要等待玩家回应，如果玩家拒绝，就丢弃这个结果，并生成新的统计值，如果玩家接受，就将其传递给随后启动的 Process Quest 角色进程。

我们来看看用户输入了什么，这次是在 handle_info 函数中处理的：

```
handle_info({tcp, Socket, Str}, S = #state{socket=Socket, next={stats, _}}) ->
    case line(Str) of
        "y" ->
            gen_server:cast(self(), stats_accepted);
        "n" ->
            gen_server:cast(self(), roll_stats);
        _ -> % 用户输入有误，再次询问
            send(Socket, "Answer with y (yes) or n (no)", [])
    end,
    {noreply, S};
```

我很想在这个函数子句中直接启动角色进程，不过我克制住了这个念头。handle_info 函数是用来处理用户输入的，handle_cast 函数才是处理和 Process Quest 有关的事情的。关注点分离！如果用户拒绝了，只要再次触发 roll_stats 就行了。这个逻辑前面已经实现过了。如果用户接受，就启动 Process Quest 角色进程，在那个进程中等待事件输入：

```
%% 玩家接受了角色统计值，开始游戏！
handle_cast(stats_accepted, S = #state{name=Name, next={stats, Stats}}) ->
    processquest:start_player(Name, [{stats,Stats},{time,?TIME},
                                     {lvlexp, ?EXP}]),
    processquest:subscribe(Name, sockserv_pq_events, self()),
```

```
    {noreply, S#state{next=playing}};
```

这些都是为游戏定义的常规调用。启动一个玩家进程，使用 sockserv_pq_events 事件处理器订阅事件。下一个状态是 playing，意思是，之后收到的所有消息基本上都来自于游戏本身：

```
%% 来自 process quest 的事件。
%% 我们知道事件的来源，因为所有事件元组都以玩家的名字作为第一个元素，
%% 这是我们自定义的标准协议
handle_cast(Event, S = #state{name=N, socket=Sock}) when element(1, Event) =:= N ->
    [case E of
       {wait, Time} -> timer:sleep(Time);
       IoList -> send(Sock, IoList, [])
     end || E <- sockserv_trans:to_str(Event)], % 转换成字符串。
    {noreply, S}.
```

代码中，sockserv_trans:to_str(Event)会把游戏事件转换成一个列表，列表的元素既可以是 IO 列表，也可以是元组{wait, Time}，{wait, Time}表示事件的不同组成部分之间需要等待的时延（在显示敌人丢弃了什么物品之前，会打印“executing a …”消息，并停顿一会）。

在要实现的步骤列表中，现在只剩下一个了：当用户通知服务器想退出时，要退出游戏。把下面的函数子句作为 handle_info 函数的第一个子句：

```
handle_info({tcp, _Socket, "quit"++_}, S) ->
    processquest:stop_player(S#state.name),
    gen_tcp:close(S#state.socket),
    {stop, normal, S};
```

终止角色进程，关闭套接字，然后停止进程自身。

还有一些其他因素会导致退出。例如，TCP 套接字被客户端关闭：

```
handle_info({tcp_closed, _Socket, _}, S) ->
    {stop, normal, S};
handle_info({tcp_error, _Socket, _}, S) ->
    {stop, normal, S};
handle_info(E, S) ->
    io:format("unexpected: ~p~n", [E]),
    {noreply, S}.
```

由于进程基本上都处于主动模式，因此虽然可能会晚一会，但是无论如何都会收到上面的出错消息，不过，你也可以在调用 gen_tcp:send/3 或者 inet:setopts/2 时检查类似的错误情况（发生这种情况时，gen_tcp:send/3 不会返回 ok）。

我们还增加了一个额外的子句来处理未知的消息。这样，如果用户输入了一些我们不期望的内容，不会导致服务器崩溃。

现在，只剩下 terminate/2 和 code_change/3 函数了：

```
code_change(_OldVsn, State, _Extra) ->
    {ok, State}.
```

```
terminate(normal, _State) ->
    ok;
terminate(_Reason, _State) ->
    io:format("terminate reason: ~p~n", [_Reason]).
```

如果你一直跟随着编写了整个程序，现在就可以编译这个文件，并用它来替换发布中相应的BEAM 文件，看看是否可以正常运行。如果复制了正确的文件，应该是正常的（我如果这么做，应该也是正常的）。

23.5 下一步的工作

如果你接受的话，我给你布置一项任务：给客户端增加几条新的命令。可以增加一条“暂停”命令，把所有动作都保存在队列里一段时间，当恢复服务器执行时再把它们一起输出。或者，如果你很调皮的话，可以把目前为止所有的级别和统计值都记录在 sockserv_serv 模块中，并增加可以从客户端获取这些信息的命令。我一直讨厌给读者留练习题，不过实在忍不住时也会留那么一两道，希望你能喜欢！

去阅读一些现有服务器实现的源代码，或者自己编写一个，也是一项不错的练习。实现一个基本的 Web 服务器不是一项容易的工作，很少有编程语言会把它当成一个练习留给初学者，不过，Erlang 就是这样的语言。多实践，它就会成为你的第二本能。用 Erlang 编写出实用的软件需要学习很多内容，和外部世界通信只是其中的一部分。在第 24 章中我们会介绍一个工具，可以使用这个工具编写单元测试，从而保证软件的可用性不会随着时间退化。

第 24 章

EUnit：单元测试框架

我们所编写的软件逐渐地变大，也更加复杂。此时，如果我们还采用以往的工作方式：启动一个 Erlang shell，在其中运行一些东西，看看结果，判断一下所做的更改是否导致了问题，就会非常乏味。随着开发工作的进展，如果能够运行一些事先准备好的测试，而不用总是手工逐项检查，就会轻松不少。也许，你本来就热衷于测试驱动开发方法，因此会认为测试非常有用。

在第 8 章开发 RPN 计算器时，我们手工编写了一些测试。这些测试就是一组形如 *Result = Expression* 的模式匹配，当出现问题时会崩溃，否则就表示成功。这种方法对于自己私下使用的简单代码来说没什么问题，不过，当需要做一些重要测试时，肯定会想使用更好的工具，如一个框架。

对于单元测试来说，我们倾向于使用 EUnit。对于集成测试，EUnit 和 Common Test 都是适用的。事实上，Common Test 适用于任何类型的测试，从单元测试到系统测试，甚至可以对非 Erlang 编写的外部软件进行测试。现在，我们来学习 EUnit，因为它非常简单，达成的效果也不错。我们会在第 28 章介绍 Common Test。

24.1 什么是 EUnit

EUnit，从根本上说，就是一种可以自动运行模块中名字以_test()结尾的函数的方法，EUnit 会把这些函数看作单元测试。如果查阅第 8 章中编写的 RPN 计算器程序，会看到如下代码：

```
rpn_test() ->
    5 = rpn("2 3 +"),
    87 = rpn("90 3 -"),
    -4 = rpn("10 4 3 + 2 * -"),
    -2.0 = rpn("10 4 3 + 2 * - 2 /"),
    ok = try
```

```
    rpn("90 34 12 33 55 66 + * - +")
catch
    error:{badmatch,[_|_]} -> ok
end,
4037 = rpn("90 34 12 33 55 66 + * - + -"),
8.0 = rpn("2 3 ^"),
true = math:sqrt(2) == rpn("2 0.5 ^"),
true = math:log(2.7) == rpn("2.7 ln"),
true = math:log10(2.7) == rpn("2.7 log10"),
50 = rpn("10 10 10 20 sum"),
10.0 = rpn("10 10 10 20 sum 5 /"),
1000.0 = rpn("10 10 20 0.5 prod"),
ok.
```

这就是我们编写的测试函数，用于保证计算器工作正常。找到原来的模块，试试如下操作：

```
1> c(calc).
{ok,calc}
2> eunit:test(calc).
  Test passed.
ok
```

只要调用一下 `eunit:test(Module)` 就行了！耶，我们现在已经学会了 EUnit！打开香槟酒庆祝一下，我们可以继续学习下一章了！

显然，只能做这点事情的测试框架是没啥用处的，用程序员的行话来说，就是“不尽人意”（not very good）。

EUnit 的功能远比自动导出和运行以 `_test()` 结尾的函数多得多。例如，你还可以把测试移到另外一个模块中，这样产品代码和测试代码就不会混在一起。这意味着，不能再对私有函数进行测试了，而且如果所有的测试都基于模块的接口（那些导出函数）编写，那么在重构代码时就无需重写测试了。

我们把测试和代码放到两个简单的模块中试试：

```
-module(ops).
-export([add/2]).

add(A,B) -> A + B.
```

```
-module(ops_tests).
-include_lib("eunit/include/eunit.hrl").

add_test() ->
    4 = ops:add(2,2).
```

我们有两个模块，`ops` 和 `ops_tests`，第二个模块中含有针对第一个模块的测试。此时，可以这样使用 EUnit：

```
3> c(ops).
{ok,ops}
4> c(ops_tests).
```

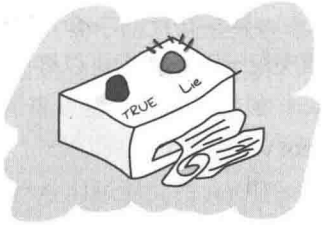
```
{ok,ops_tests}
5> eunit:test(ops).
  Test passed.
ok
```

eunit:test(Mod) 调用会自动寻找模块 Mod_tests，并运行其中的测试。

我们来更改一下测试代码（改成 3 = ops:add(2,2)），看看出错时会怎样：

```
6> c(ops_tests).
{ok,ops_tests}
7> eunit:test(ops).
ops_tests: add_test (module 'ops_tests')...*failed*
::error:{badmatch,4}
  in function ops_tests:add_test/0


=======================================================
  Failed: 1. Skipped: 0. Passed: 0.
error
```

我们可以看到是哪个测试失败了（ops_tests: add_test...），以及失败原因（::error:{badmatch,4}）。还得到了关于测试通过个数和失败个数的完整报告。



不过，这个显示结果非常糟糕——至少和常见的 Erlang 崩溃一样糟糕。例如，它并没有给出一个清晰的解释（4 到底和什么不匹配？）。如果测试框架只是运行测试，但是不能提供更详细的测试运行信息，那么这个测试框架不能给我们带来多少帮助。

基于这个原因，EUnit 提供了一些非常有用的宏。它们提供了更整洁的报告（包括行号）和更清晰的语义。有了这些宏，我们不仅能知道错误发生了，还能知道出错的原因：

?assert(Expression)和?assertNot(Expression)

这两个宏用来测试布尔值。如果传给?assert 的值不是 true，就会显示一条错误。?assertNot 的用法一样，不过针对假值。这两个宏与 true = X 和 false = Y 基本等价。

?assertEqual(A, B)

这个宏对表达式 A 和 B 进行严格的比较（和=:=等价）。如果它们不同，会发生错误。该宏大体上等价于 true = X =:= Y。还有一个与?assertEqual 相反的宏?assertNotEqual。

?assertMatch(Pattern, Expression)

这个宏可以用来进行像 *Pattern = Expression* 这种形式的匹配，它不会进行变量绑定。这意味着，你可以这样调用：?assertMatch({X,X}, some_function())，断言会得到一个二元元组，且两个元素相同。并且，后面还可以这样调用：?assertMatch(X,Y)，

X 不会被绑定。

事实上，和 *Pattern* = *Expression* 相比，这个宏的语义更像(fun (Pattern) -> true; (_) -> erlang:error(nomatch) end)(Expression)。函数头模式中的变量绝不会跨多个断言绑定。在 R14B04 中，EUnit 增加了另一个宏?assertNotMatch。

?assertError(Pattern, Expression)

这个宏告诉 EUnit，*Expression* 应该导致一个错误。例如，?assertError(badarith, 1/0)是一个成功的测试。

?assertThrow(Pattern, Expression)

这个宏和?assertError 一样，不过针对的是 throw(Pattern)而非 erlang:error(Pattern)。

?assertExit(Pattern, Expression)

这个宏和?assertError 一样，不过针对的是 exit(Pattern)（不是 exit/2）而非 erlang:error(Pattern)。

?assertException(Class, Pattern, Expression)

这个宏是上面 3 个宏的通用形式。例如，?assertException(error, Pattern, Expression)和?assertError(Pattern, Expression) 完全一样。从 R14B04 开始，提供了另外一个宏?assertNotException/3。

使用这些宏，可以编写出更好的测试，如下：

```
-module(ops_tests).
-include_lib("eunit/include/eunit.hrl").

add_test() ->
    4 = ops:add(2,2).

new_add_test() ->
    ?assertEqual(4, ops:add(2,2)),
    ?assertEqual(3, ops:add(1,2)),
    ?assert(is_number(ops:add(1,2))),
    ?assertEqual(3, ops:add(1,1)),
    ?assertError(badarith, 1/0).
```

运行一下试试：

```
8> c(ops_tests).
./ops_tests.erl:12: Warning: this expression will fail with a 'badarith' exception
{ok,ops_tests}
9> eunit:test(ops).
ops_tests: new_add_test...*failed*
::error:{assertEqual_failed,[{module,ops_tests},
                             {line,11},
                             {expression,"ops : add ( 1 , 1 )"},
                             {expected,3},
```

```
                                {value,2}]}
  in function ops_tests:'-new_add_test/0-fun-3-'/1
  in call from ops_tests:new_add_test/0


=======================================================
  Failed: 1. Skipped: 0. Passed: 1.
error
```

可以看出，错误报告改善了不少。我们现在知道，ops_tests 中第 11 行的?assertEqual 失败了。在调用 ops:add(1,1)时，我们认为会得到数值 3，结果得到的是 2。当然，必须按照 Erlang 数据项的格式理解这些信息，不过这已经相当不错了。

令人讨厌的是，即使我们的 5 个断言中只有 1 个失败了，整个测试还是被认为全部失败了。如果能够知道某个断言失败了，并且不用让其后的其他断言看起来也像失败了，就更好了。我们的测试就像是参加这样一种考试：你一犯错，立马就宣告你考试失败，并把你赶出学校。接着，你的狗死了，就这样过了极其糟糕的一天。

因为对这种灵活性的需求很普遍，所以 EUnit 提供了一种解决方案：测试生成器。

24.2　测试生成器

测试生成器是一种用函数来封装断言的简写方法，这些函数可在将来以一种巧妙的方式执行。测试生成器没有使用以_test()结尾的函数和?assertSomething 形式的宏，它使用的是以_test_()结尾的函数，宏的形式是?_assertSomething。变化虽小，却可以使测试更加强大。

下面两个测试是等价的：

```
function_test() -> ?assert(A == B).
function_test_() -> ?_assert(A == B).
```

此处，函数 function_test_()称为测试生成器函数，?_assert(A == B)称为测试生成器。之所以这样叫，是因为?_assert(A == B)在内部被实现为 fun() -> ?assert(A,B) end，也就是说，这是一个生成测试的函数。

和常规断言相比，测试生成器的优势在于，它们是 fun 类型的。这意味着，可以在不执行的情况下操纵它。事实上，我们可以创建如下形式的测试集合：

```
my_test_() ->
    [?_assert(A),
     [?_assert(B),
      ?_assert(C),
     [?_assert(D)]],
    [[?_assert(E)]]].
```

测试集合可以是深层嵌套的测试生成器列表。我们可以让函数返回测试！把下面的代码增加到 ops_tests 中：

```
add_test_() ->
    [test_them_types(),
     test_them_values(),
     ?_assertError(badarith, 1/0)].

test_them_types() ->
    ?_assert(is_number(ops:add(1,2))).

test_them_values() ->
    [?_assertEqual(4, ops:add(2,2)),
     ?_assertEqual(3, ops:add(1,2)),
     ?_assertEqual(3, ops:add(1,1))].
```

由于只有函数 add_test_()是以_test_()结尾的，因此函数 test_them_types()和 test_them_values()不会被当作测试。事实上，它们是被函数 add_test_()调用来生成测试的：

```
1> c(ops_tests).
./ops_tests.erl:12: Warning: this expression will fail with a 'badarith' exception
./ops_tests.erl:17: Warning: this expression will fail with a 'badarith' exception
{ok,ops_tests}
2> eunit:test(ops).
ops_tests:25: test_them_values...*failed*
[...]
ops_tests: new_add_test...*failed*
[...]


=======================================================
  Failed: 2. Skipped: 0. Passed: 5.
error
```

仍然得到了预期中的错误，不过现在可以看到，测试的总数量从 2 跳到了 7——测试生成器的魔法。

如果只想对测试套件中的一部分进行测试，如只测试 add_test_/0，该怎么办呢？EUnit 提供了一些锦囊妙计。

```
3> eunit:test({generator, fun ops_tests:add_test_/0}).
ops_tests:25: test_them_values...*failed*
::error:{assertEqual_failed,[{module,ops_tests},
                             {line,25},
                             {expression,"ops : add ( 1 , 1 )"},
                             {expected,3},
                             {value,2}]}
in function ops_tests:'-test_them_values/0-fun-4-'/1

=======================================================
  Failed: 1. Skipped: 0. Passed: 4.
error
```

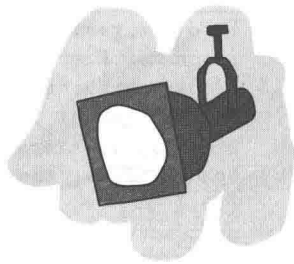
注意，这种方法只适用于测试生成器函数。这里使用的{generator, *Fun*}在 EUnit 的术语中称为*测试描述*（test representation）。

EUnit 提供了如下几种测试描述：

- {module, *Mod*}，运行 *Mod* 中的所有测试；
- {dir, *Path*}，运行在 *Path* 中找到的所有模块中的测试；
- {file, *Path*}，运行单个已编译模块中的所有测试；
- {generator, *Fun*}，运行单个测试生成器函数中的测试，前面的例子就是这种情况；
- {application, *AppName*}，运行针对 *AppName* 的.app 文件中提到的所有模块的测试。

使用这些测试描述，就可以比较容易地运行针对整个应用甚至整个发布的测试了。

24.3　测试夹具



仅仅使用断言和测试生成器是很难去测试整个应用的，这也是*测试夹具*（fixture）存在的原因。虽然测试夹具不是一个在应用层面管理和运行测试的完备方案，但是可以用它来构建一些测试脚手架（scaffolding）。

这里所说的脚手架是一种通用结构，用来为每个测试定义初始化准备和事后清理函数。使用这两个函数，可以搭建出测试工作需要的状态和环境。此外，还可以用脚手架指定测试的运行方式（是想本地运行、在另外的进程中运行，还是其他方式？）。

有几种类型的测试夹具可供选择，它们之间略有不同。第一种类型称为 setup 测试夹具。setup 测试夹具的形式有以下几种：

```
{setup, Setup, Instantiator}
{setup, Setup, Cleanup, Instantiator}
{setup, Where, Setup, Instantiator}
{setup, Where, Setup, Cleanup, Instantiator}
```

嗯！要理解测试夹具是如何工作的，我们得先学习一些 EUnit 的术语。

初始化函数

这是一个没有任何参数的函数。这个函数的返回值会传递给每一个测试，对于每个实例化器（instantiator），初始化函数只会调用一次。

清理函数

这是一个函数，它把初始化函数的返回值作为参数，执行所需的清理工作。如果说 OTP 的 terminate 函数做的事情和 init 函数正好相反的话，那么在 EUnit 中，清理函数和初始化函数的工作也正好相反。每调用一次初始化函数，都会有一次匹配的清理函数调用。

实例化器

这是一个函数，它以初始化函数的返回值作为参数，返回一个测试集合（测试集合可以是一个深度嵌套的?_Macro 形式的断言列表）。这个字段有时也可以是一个实例化器列表，对列表中的每个实例化器，都会调用初始化函数和清理函数。

运行方式

这是一个函数，它指定了测试的运行方式，可以是 local、spawn 和{spawn, node()}。

好了，那么实际中，如何使用它们呢？嗯，我们来考虑一个测试，通过这个测试可以确认某个假想的进程注册库是否正确处理了同一个进程两次注册不同名字的情况：

```
double_register_test_() ->
  {setup,
   fun start/0,                    % 初始化函数
   fun stop/1,                     % 清理函数
   fun two_names_one_pid/1}.       % 实例化器

start() ->
   {ok, Pid} = registry:start_link(),
   Pid.

stop(Pid) ->
   registry:stop(Pid).

two_names_one_pid(Pid) ->
   ok = registry:register(Pid, quite_a_unique_name, self()),
   Res = registry:register(Pid, my_other_name_is_more_creative, self()),
   [?_assertEqual({error, already_named}, Res)].
```

这个夹具首先在函数 start/0 中启动服务器 registry。接着调用实例化器函数 two_names_one_pid(ResultFromSetup)。在这个测试中，只做了一件事情，就是把当前进程注册两次。

这些工作是在实例化器函数中完成的。第二次注册的结果被保存在变量 Res 中。该函数返回一个包含单个测试(?_assertEqual({error, already_named}, Res))的测试集合。这个测试集合会被 EUnit 运行。接着会调用清理函数 stop/1。stop/1 函数使用初始化函数返回的 pid 去关闭之前启动的注册库。

更妙的是，我们还可以把整个测试夹具都放进一个测试集合中：

```
some_test_() ->
   [{setup, fun start/0, fun stop/1, fun some_instantiator1/1},
    {setup, fun start/0, fun stop/1, fun some_instantiator2/1},
    ...
    {setup, fun start/0, fun stop/1, fun some_instantiatorN/1}].
```

这样就行了！

这种写法有点令人讨厌，我们需要一直重复初始化和清理函数，尤其是当它们始终一样时。

第二种类型的测试夹具正是为解决这个问题而存在的，它称为 foreach 测试夹具。

foreach 测试夹具和 setup 测试夹具非常相似，不同之处在于 foreach 测试夹具接收的是实例化器列表。

```
{foreach, Where, Setup, Cleanup, [Instantiator]}
{foreach, Setup, Cleanup, [Instantiator]}
{foreach, Where, Setup, [Instantiator]}
{foreach, Setup, [Instantiator]}
```

下面是用 foreach 测试夹具编写的 some_test_/0 函数：

```
some_test2_() ->
    {foreach
     fun start/0,
     fun stop/1,
     [fun some_instantiator1/1,
      fun some_instantiator2/1,
      ...
      fun some_instantiatorN/1]}.
```

看起来好多了。foreach 测试夹具会为列表中的每个实例化器运行初始化和清理函数。

现在，我们已经介绍了如何使用单实例化器测试夹具和多实例化器测试夹具（对其中的每个实例化器，都会调用初始化和清理函数）。但是，如果我们希望对多个实例化器只调用一次初始化函数和一次清理函数，该怎么办呢？

换句话说，如果我们有多个实例化器，但是对于某些状态，我们只想初始化一次，该怎么办呢？对于这个问题，没有简单的方法，不过有一个技巧可以完成这项工作：

```
some_tricky_test_() ->
    {setup,
     fun start/0,
     fun stop/1,
     fun (SetupData) ->
         [some_instantiator1(SetupData),
          some_instantiator2(SetupData),
          ...
          some_instantiatorN(SetupData)]
     end}.
```

基于测试集合可以是深度嵌套的列表这个事实，我们可以把一组实例化器封装在一个匿名函数中，就像是一个生成它们的实例化器。

24.3.1　其他测试控制方法

在使用测试夹具时，还可以对测试的运行方式进行更细粒度的控制。有 4 个可用的选项：

{spawn, TestSet}

这个选项会让测试在另外一个独立进程中运行，而不在主

测试进程中。主测试进程会等待所有创建出来的进程运行结束。

{timeout, *Seconds*, TestSet}

这个选项会限制测试只能运行 *Seconds* 秒。如果运行时间超过 *Seconds* 秒，会终止测试。

{inorder, TestSet}

这个选项会让测试严格按照测试集合中的顺序运行。

{inparallel, Tests}

这个选项会让测试尽可能地并行运行。

例如，可以把 some_tricky_test_/0 测试生成器重写成如下样子：

```
some_tricky_test2_() ->
    {setup,
     fun start/0,
     fun stop/1,
     fun(SetupData) ->
         {inparallel,
          [some_instantiator1(SetupData),
           some_instantiator2(SetupData),
           ...
           some_instantiatorN(SetupData)]}
    end}.
```

24.3.2 测试文档

测试夹具的内容基本上就这些了，不过还有一个不错的方法值得介绍。你可以用一种非常整洁的方式描述测试。看看下面：

```
double_register_test_() ->
    {"Verifies that the registry doesn't allow a single process to "
     "be registered under two names. We assume that each pid has the "
     "exclusive right to only one name",
    {setup,
     fun start/0,
     fun stop/1,
     fun two_names_one_pid/1}}.
```

很不错，对吧？可以把测试夹具封装在{Comment, Fixture}中，这样测试就会非常易读。我们会在实践中采用这种方式。

24.4 测试 regis

虚构的测试很无趣，而假装去测试并不存在的软件则更糟糕，因此在本节中，我们来研究一下我为 regis-1.0.0 进程注册库编写的测试，Process Quest 中使用了这个进程注册库。

regis 是以测试驱动的方式完成的。希望你不讨厌测试驱动开发（TDD），不过即便如此，情况应该也不会太糟，因为我们将要看到的只是测试驱动生成的测试套件。而我在写这些测试时

所经历的那些试错和回退的过程都被删减了，只看这些测试套件会觉得我很能干，文本编辑简直太神奇了。

regis 应用由 3 部分组成：一个监督者、一个主服务器以及一个应用回调模块。由于监督者进程只会检查服务器，而应用回调模块只是作为另外两个模块的接口，所以编写一个只测试服务器的测试套件就足够了，不会涉及任何外部依赖。

作为一个 TDD 爱好者，我会首先罗列出所有期望测试覆盖到的特性。

- 具有和 Erlang 默认进程注册库类似的接口。
- 服务器要有一个注册名，这样就可以不用通过 pid 和它通信。
- 进程可以使用我们的服务注册它的进程名，然后其他进程可以通过这个名字和它通信。
- 可以得到所有已注册进程的列表。
- 对于没有被任何进程注册的名字，要返回原子 undefined（和 Eralng 的标准进程库很像），这样可以让使用它的调用崩溃。
- 同一个进程不能有两个名字。
- 两个进程不能重名。
- 已经注册的进程在解除注册后，可以再次注册。
- 解除注册时绝不会导致调用者进程崩溃。
- 已注册进程的退出会导致解除注册它的名字。

这个列表相当不错。通过对其中的内容逐条分析，添加用例，我把这个规格说明中的每一条都变成了一个测试。最终的文件是 regis_server_tests。我使用一个简单结构来编写这些测试，如下所示：

```
-module(regis_server_tests).
-include_lib("eunit/include/eunit.hrl").

%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% TESTS DESCRIPTIONS %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%
%%% SETUP FUNCTIONS %%%
%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%
%%% ACTUAL TESTS %%%
%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%
%%% HELPER FUNCTIONS %%%
%%%%%%%%%%%%%%%%%%%%%%%%%
```

在模块中没有内容时，这些注释看起来有些奇怪，不过随着逐渐向其中添加内容，这些注释的意义也会逐渐体现出来。

增加了第一个测试（服务器能够被启动，并能通过名字找到它）之后，文件格式如下：

```
-module(regis_server_tests).
-include_lib("eunit/include/eunit.hrl").

%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% TESTS DESCRIPTIONS %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
start_stop_test_() ->
    {"The server can be started, stopped and has a registered name",
     {setup,
      fun start/0,
      fun stop/1,
      fun is_registered/1}}.

%%%%%%%%%%%%%%%%%%%%%%%%
%%% SETUP FUNCTIONS %%%
%%%%%%%%%%%%%%%%%%%%%%%%
start() ->
    {ok, Pid} = regis_server:start_link(),
    Pid.

stop(_) ->
    regis_server:stop().

%%%%%%%%%%%%%%%%%%%%%
%%% ACTUAL TESTS %%%
%%%%%%%%%%%%%%%%%%%%%
is_registered(Pid) ->
    [?_assert(erlang:is_process_alive(Pid)),
     ?_assertEqual(Pid, whereis(regis_server))].

%%%%%%%%%%%%%%%%%%%%%%%%
%%% HELPER FUNCTIONS %%%
%%%%%%%%%%%%%%%%%%%%%%%%
```

现在能明白这种组织形式的含义了吧？应该比刚才好多了。文件的第一部分只包含测试夹具和特性的概括描述。第二部分包含需要的初始化和清理函数。最后一部分包含返回测试集合的实例化器。

在这个例子中，实例化器是 `is_registered(Pid)`，它会确认服务器能够被启动和停止。这个函数的细节，我们稍后再看。

如果你把这个测试的最终版本和本书的其他代码一起下载下来了，那么，可以看到文件前两部分的内容如下：

```
-module(regis_server_tests).
-include_lib("eunit/include/eunit.hrl").
```

```
-define(setup(F), {setup, fun start/0, fun stop/1, F}).

%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% TESTS DESCRIPTIONS %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%

start_stop_test_() ->
    {"The server can be started, stopped and has a registered name",
     ?setup(fun is_registered/1)}.

register_test_() ->
    [{"A process can be registered and contacted",
      ?setup(fun register_contact/1)},
     {"A list of registered processes can be obtained",
      ?setup(fun registered_list/1)},
     {"An undefined name should return 'undefined' to crash calls",
      ?setup(fun noregister/1)},
     {"A process cannot have two names",
      ?setup(fun two_names_one_pid/1)},
     {"Two processes cannot share the same name",
      ?setup(fun two_pids_one_name/1)}].

unregister_test_() ->
    [{"A process that was registered can be registered again iff it was "
      "unregistered between both calls",
      ?setup(fun re_un_register/1)},
     {"Unregistering never crashes",
      ?setup(fun unregister_nocrash/1)},
     {"A crash unregisters a process",
      ?setup(fun crash_unregisters/1)}].

%%%%%%%%%%%%%%%%%%%%%%%%%
%%% SETUP FUNCTIONS %%%
%%%%%%%%%%%%%%%%%%%%%%%%%
start() ->
    {ok, Pid} = regis_server:start_link(),
    Pid.

stop(_) ->
    regis_server:stop().

%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% HELPER FUNCTIONS %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 尚无内容
```

很好，不是吗？注意，在编写测试时，我发现除了 start/0 和 stop/1，我根本就不需要任何其他的初始化和清理函数。因此，我定义了?setup(Instantiator)宏，这样，代码看起来要比把测试夹具的内容完全展开好看一些。

现在，可以清楚地看出特性列表中的每一条和测试之间的对应关系。你会注意到，我把测试分成了服务器的启动和停止（start_stop_test_/0）、进程注册（register_test_/0）以及进程解除注册（unregister_test_/0）几个类别。

通过阅读测试生成器的定义，我们可以了解模块要做的工作。测试变成了文档（当然不能替代规范的文档）。

我们稍微来研究一下这些测试，看看它们为何能按照确定的方式工作。下面是列表中的第一个测试，start_stop_test_/0，它对应一个简单的需求——服务器可以注册一个名字：

```
start_stop_test_() ->
    {"The server can be started, stopped and has a registered name",
     ?setup(fun is_registered/1)}.
```

测试本身的实现放在 is_registered/1 函数中，没有什么变化：

```
%%%%%%%%%%%%%%%%%%%%
%%% ACTUAL TESTS %%%
%%%%%%%%%%%%%%%%%%%%
is_registered(Pid) ->
    [?_assert(erlang:is_process_alive(Pid)),
     ?_assertEqual(Pid, whereis(regis_server))].
```

这个实例化器和前面的完全一样。除了函数 erlang:is_process_alive(Pid) 可能有些陌生之外，这个测试并没有什么特别的地方。正如名字所述，这个函数检查进程当前是否在运行。包含这个测试的原因很简单：服务器可能刚一启动就崩溃了，或者根本就没有被启动。如果我们认为注册失败是由代码错误导致的，实际上是因为服务器根本就没能启动，这样的测试结果会令人困惑。

第二个测试是关于进程注册的：

```
{"A process can be registered and contacted",
 ?setup(fun register_contact/1)}
```

测试代码如下：

```
register_contact(_) ->
    Pid = spawn_link(fun() -> callback(regcontact) end),
    timer:sleep(15),
    Ref = make_ref(),
    WherePid = regis_server:whereis(regcontact),
    regis_server:whereis(regcontact) ! {self(), Ref, hi},
    Rec = receive
        {Ref, hi} -> true
        after 2000 -> false
    end,
    [?_assertEqual(Pid, WherePid),
     ?_assert(Rec)].
```

是的，这段代码的确不怎么优雅。计时器是最令人讨厌的地方，应该使用某种同步的进程初

始化机制（可以使用行为或者 proc_lib:start_link 及其关联的同步化函数，proc_lib 模块的文档中对此有介绍）取代它。测试中创建了一个进程，这个进程只是注册自己并回应我们发给它的消息。这些工作都在辅助函数 callback/1 中完成，定义如下：

```
%%%%%%%%%%%%%%%%%%%%%%%%
%%% HELPER FUNCTIONS %%%
%%%%%%%%%%%%%%%%%%%%%%%%
callback(Name) ->
    ok = regis_server:register(Name, self()),
    receive
        {From, Ref, Msg} -> From ! {Ref, Msg}
    end.
```

该函数注册自身进程，接收一条消息，并发送一条回应。进程一启动，实例化器 register_contact/1 会等待 15 ms（这个微小延迟是为了确保另一个进程完成注册），接着使用 regis_server 模块的 whereis 函数获取进程的 pid，并给该进程发送消息。如果 regis 服务器运行正确，就会有一条消息返回，并且函数末尾测试中针对两个 pid 的匹配也能成功。

保持冷静

在第二个测试中，我们使用了计时器。Erlang 程序的并发和时间敏感特点，在测试中常常需要加入像这样的小计时器，它们只是用来同步代码。

因此，问题就变成了如何才能定义一个好的计时器，让等待的时间足够长。当系统中运行的测试很多或者计算机的负载很高时，计时器等待的时长还够吗？

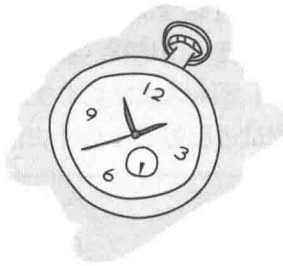
有时，为了尽可能减少测试中需要的同步操作，Erlang 程序员在编写测试时必须要非常聪明。在这方面没有简单的解决方案。

下一个测试如下：

```
{"A list of registered processes can be obtained",
 ?setup(fun registered_list/1)}
```

当注册了一批进程时，应该可以得到所有进程的注册名列表。这个功能和 Erlang 的 registered()函数调用类似：

```
registered_list(_) ->
    L1 = regis_server:get_names(),
    Pids = [spawn(fun() -> callback(N) end) || N <- lists:seq(1,15)],
    timer:sleep(200),
    L2 = regis_server:get_names(),
    [exit(Pid, kill) || Pid <- Pids],
    [?_assertEqual([], L1),
     ?_assertEqual(lists:sort(lists:seq(1,15)), lists:sort(L2))].
```



首先，我们要确保第一个注册进程列表是空的(?_assertEqual(L1, []))，这样就知道当没有任何进程注册过自己时，一切正常。接着创建了 15 个进程，都用数字（1 到 15）作为名进行了注册。我们让测试睡了一会，确保所有进程都完成了注册，然后调用 regis_server: get_names()。所返回的名字中应该包含 1 到 15 之间的所有整数。接下来，进行了一点清理工作，清除了所有的注册进程——我们不想让进程泄露。

注意，在测试中，我们总是把状态提前保存在变量（L1 和 L2）中，然后在测试集合中使用它们。原因是：所返回的测试集合要在测试实例化器（所有的有效代码）运行完毕后才被执行。如果把依赖于其他进程的函数调用以及时间敏感的事件放到?_assert*宏中，就会让一切失去同步，对你以及使用你软件的用户通常都会造成非常糟糕的结果。

下一个测试很简单：

```
{"An undefined name should return 'undefined' to crash calls",
?setup(fun noregister/1)}
...
noregister(_) ->
    [?_assertError(badarg, regis_server:whereis(make_ref()) ! hi),
     ?_assertEqual(undefined, regis_server:whereis(make_ref()))].
```

可以看出，这个测试测了两件事情：返回了 undefined 以及使用 undefined 确实让调用者崩溃了（规格说明中要求的）。这里，无需使用临时变量保存状态，这两个测试可以在 regis 服务器生存期内的任何时间点运行，因为我们不会改变它的状态。

接着往下看：

```
{"A process cannot have two names",
 ?setup(fun two_names_one_pid/1)},
...
two_names_one_pid(_) ->
    ok = regis_server:register(make_ref(), self()),
    Res = regis_server:register(make_ref(), self()),
    [?_assertEqual({error, already_named}, Res)].
```

这个测试和本章前面的示例测试完全一样。在这个测试中，我们只检查是否得到了正确的返回值，以及测试进程不能以不同的名字注册自身两次。

下一个测试正好和 two_names_one_pid 相反：

```
{"Two processes cannot share the same name",
 ?setup(fun two_pids_one_name/1)}].
...
two_pids_one_name(_) ->
    Pid = spawn(fun() -> callback(myname) end),
    timer:sleep(15),
    Res = regis_server:register(myname, self()),
    exit(Pid, kill),
    [?_assertEqual({error, name_taken}, Res)].
```

在这个测试中，需要有两个进程，但是只需要其中一个的结果，所以我们使用了一个技巧：只创建一个进程（我们不需要这个进程的结果），接着在当前测试进程中完成关键的工作。

可以看到，其中使用了计时器来保证创建出来的进程先注册名字（在 callback/1 函数中），测试进程等了一会才去注册，并期望返回结果是一个错误元组（{error, name_taken}）。

使用唯一值

你也许已经注意到，在前面的例子中有不少地方使用了函数 make_ref()。只要可能，尽量去使用像 make_ref()这样能生成唯一值的函数。在将来的某一天，如果有人想并行运行测试或者使用某个永不停止的 regis 服务器运行测试，那么测试无需更改就可以直接运行。

如果我们在测试中使用了硬编码的值，如 a、b 和 c，那么在同时运行多个测试套件时，迟早会发生名字冲突。regis_server_tests 套件中的测试并没有都遵循这个建议，因为它们主要用于演示。

和进程注册相关的测试介绍完毕。下面只剩下和进程解除注册相关的测试了：

```
unregister_test_() ->
[{"A process that was registered can be registered again iff it was "
  "unregistered between both calls",
  ?setup(fun re_un_register/1)},
 {"Unregistering never crashes",
  ?setup(fun unregister_nocrash/1)},
 {"A crash unregisters a process",
  ?setup(fun crash_unregisters/1)}].
```

我们来看看这些测试是如何实现的。第一个比较简单：

```
re_un_register(_) ->
    Ref = make_ref(),
    L = [regis_server:register(Ref, self()),
         regis_server:register(make_ref(), self()),
         regis_server:unregister(Ref),
         regis_server:register(make_ref(), self())],
    [?_assertEqual([ok, {error, already_named}, ok, ok], L)].
```

上面代码把所有调用顺序排列在一个列表中，这是一个实用的技巧，当需要测试多个调用的返回结果时，我很喜欢使用它。通过把调用放到一个列表中，就可以对调用结果序列和期望序列[ok, {error, already_named}, ok, ok]进行比较，检查代码的运行情况。注意，Erlang 语言并没有明确规定会按照顺序对列表求值，不过我使用这个技巧时基本上没有遇到过问题。

接下来是进程不能崩溃的测试，如下：

```
unregister_nocrash(_) ->
    ?_assertEqual(ok, regis_server:unregister(make_ref())).
```

哇，慢点，朋友！这样就行了吗？是的，这样就行了。re_un_register 函数已经测试了进程“解除注册”功能。在 unregister_nocrash 中，我们只想知道解除注册一个没有注册过

的进程时是否正常。

现在，只剩下最后一个测试了，也是对所有进程注册库来说最重要的一个：具名进程崩溃后，它的名字会被解除注册。这个特性的意义深远，因为如果不把名字去除，那么注册库服务器中的名字就会不断增加，可用的名字就会越来越少。

```
crash_unregisters(_) ->
    Ref = make_ref(),
    Pid = spawn(fun() -> callback(Ref) end),
    timer:sleep(150),
    Pid = regis_server:whereis(Ref),
    exit(Pid, kill),
    timer:sleep(95),
    regis_server:register(Ref, self()),
    S = regis_server:whereis(Ref),
    Self = self(),
    ?_assertEqual(Self, S).
```

这段代码表达了下面的动作序列。

（1）注册一个进程。

（2）确信进程已经注册。

（3）杀死这个进程。

（4）盗用这个进程的名字标识（间谍就是这么干的）。

（5）检查名字是否已经赋予自己。

老实讲，上面的测试本可以写得更简单：

```
crash_unregisters(_) ->
    Ref = make_ref(),
    Pid = spawn(fun() -> callback(Ref) end),
    timer:sleep(150),
    Pid = regis_server:whereis(Ref),
    exit(Pid, kill),
    ?_assertEqual(undefined, regis_server:whereis(Ref)).
```

那段盗用死亡进程名字标识的代码完全只是一个无聊的偷盗幻想。

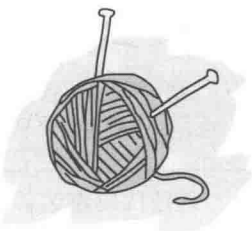
就这样了！如果没犯什么错误的话，代码应该能够编译通过，测试套件也应该能够成功运行：

```
$ erl -make
Recompile: src/regis_sup
... <snip> ...
$ erl -pa ebin/

1> eunit:test(regis_server).
  All 13 tests passed.
ok
2> eunit:test(regis_server, [verbose]).
======================== EUnit ========================
module 'regis_server'
  module 'regis_server_tests'
    The server can be started, stopped and has a registered name
```

```
    regis_server_tests:49: is_registered...ok
    regis_server_tests:50: is_registered...ok
    [done in 0.006 s]
... <snip> ...
  [done in 0.520 s]
=======================================================
  All 13 tests passed.
ok
```

噢耶，看到了吧，使用 verbose 选项可以把测试描述和运行时信息显示在报告中。非常整洁。

24.5　EUnit 小结



到此，我们已经介绍了 EUnit 的主要特性，以及如何运行用这些特性编写的测试套件。更为重要的是，你还学到了一些并发进程的测试技术，其中使用的模式也适用于实际开发工作。

我们再来介绍最后一个技巧。当你想测试 gen_server 和 gen_fsm 之类的进程时，可能想去查验它们的内部状态。sys 模块提供了一种好方法：

```
3> regis_server:start_link().
{ok,<0.160.0>}
4> regis_server:register(shell, self()).
ok
5> sys:get_status(whereis(regis_server)).
{status,<0.160.0>,
        {module,gen_server},
        [[{'$ancestors',[<0.31.0>]},
         {'$initial_call',{regis_server,init,1}}],
        running,<0.31.0>,[],
        [{header,"Status for generic server regis_server"},
         {data,[{"Status",running},
                {"Parent",<0.31.0>},
                {"Logged events",[]}]},
         {data,[{"State",
                 {state,{1,{<0.31.0>,{shell,#Ref<0.0.0.333>},nil,nil}},
                        {1,{shell,{<0.31.0>,#Ref<0.0.0.333>},nil,nil}}}}]}]]}
```

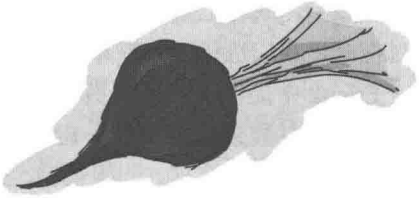
很清楚，对吧？服务器的所有内部细节都呈现在你面前，现在，你可以随时查看所有需要的信息了！

如果想了解关于服务器测试的更多信息，我推荐你去阅读 Process Quest 中 player 模块的测试代码，地址为 http://learnyousomeerlang.com/static/erlang/processquest/apps/processquest-1.1.0/test/pq_player_tests.erl。这个模块在测试 gen_server 时采用了另外一种技术，在测试中直接调用了 handle_call、handle_cast 以及 handle_info。该模块的开发仍然采用了测试驱动的方式，不过这种直接调用方式会导致写出不同的测试代码。

在第 25 章中，我们将使用 ETS——一个所有 Erlang 进程都可以访问的内存数据库——重写进程注册库，到那时，你会明白测试的真正价值所在。

第 25 章

ETS：免费的内存 NoSQL 数据库



有一件事情我们已经做过很多次了：用进程来实现某种存储装置。我们制造了存储东西的冰箱，实现了注册进程的 regis，还编写了一些键/值型存储库等。如果我们是使用面向对象设计的程序员，那么应该已经设计了一堆的单例对象、专用的存储类以及其他一些莫名其妙的东西了。事实上，把像字典和 GB 树这样的数据结构封装到进程中的做法和这有点类似。

本章将介绍 ETS。ETS 是一个内存数据库，它提供了另外一种数据存储方法。

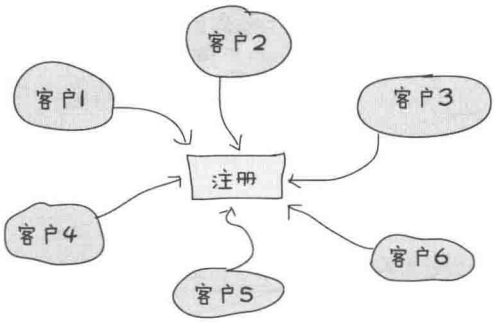
25.1 为什么需要 ETS

在很多情况下，用进程来保存数据结构确实是一个不错的选择，例如，当进程内执行某项任务需要这些数据时，可以把它们当作进程的内部状态。这也是我们的主要用法。不过，有一种情况，这种方法可能不是最好的选择：进程保存的数据结构主要是为了和其他进程共享。

我们曾经编写过的一个应用就存在这方面的问题。你能猜到是哪一个吗？你肯定猜到了。我在上一章结束时曾经提到过，`regis`（我们在第 22 章开发的 Process Quest 游戏的一部分）需要被重写。不是因为它功能不对或者不好用，而是因为它的作用就是充当一个网关，与数量众多的其他进程共享数据。

`regis` 是 Process Quest 中负责消息传递的中心应用（其他应用都要使用它），发往有名进程的所有消息都要由它进行转发。这意味着，虽然我们精心设计用独立的进程让我们的应用高度并发，并且确保我们的监督结构能够按比例扩展，但是我们所有的操作却都依赖于这个位于中心的、只能顺序回应消息的 `regis` 进程：

如果要转发的消息数量众多，那么 `regis` 就会越来越忙。当需要转发的消息多到一定程度时，整个系统就会变成顺序执行，越来越慢。非常糟糕。



注意　并没有直接证据表明 regis 是 Process Quest 的瓶颈所在。事实上，和真实世界中的许多应用相比，Process Quest 的消息量是很少的。如果把 regis 用在消息量和查询量都非常大的场合，问题就会非常明显。

要解决这个问题，可以把 regis 分割成多个子进程，从而加快查询速度（对数据分区（sharding）），或者把数据存储在某个允许并行、并发访问的数据库中。第一种方法更有趣一些，不过在此，我们选择较为容易的第二种方法。

Erlang 中提供了 Erlang 数据项存储（Erlang Term Storage, ETS）表，这是 Erlang VM 自带的一种高效内存数据库。ETS 是 Erlang VM 的一部分，支持破坏性更新，不使用 GC，并且它所使用的内存不和进程共享。一般来讲，ETS 表的速度非常快，对 Erlang 程序员来说，当代码中的某些部分性能不够时，使用 ETS 是一种非常简单的代码优化方法。

ETS 表在读写方面提供了有限的并发支持（比没提供任何支持的进程邮箱好多了），可以让我们优化掉不少问题，但是也可能会增加一些新问题。这是因为，如果使用了 ETS，那么原本用来保证 Erlang 并发安全性的大部分概念都要被丢弃掉。

保持冷静

虽然 ETS 是一种很好的应用优化方法，但是在使用时一定要小心。默认情况下，VM 把 ETS 表的个数限定为 1 400 个。这个值是可以修改的（通过 erl -env ERL_MAX_ETS_TABLES Number），不过，如此低的默认设定值，本身就表明要尽量避免一个进程一个表的情况。

在用 ETS 重写 regis 之前，我们先来了解一下 ETS 的一些设计原则。

25.2　ETS 的概念

ETS 表是作为 BIF 实现在 ets 模块中的。ETS 的设计目标是：在 Erlang 中提供一种具有常量访问时间的、大规模数据存储方法（函数式数据结构一般都是对数级访问时间），并且为了简单易用，符合习惯，这种存储方式看起来要和基于进程实现的一样。

注意　让 ETS 表看起来像进程，并不是说你可以像进程那样去创建它们或者建立和它们的链

接。而是说，它们要遵循无共享的语义，把调用封装在函数式接口后面，可以处理任何 Erlang 原生数据类型，并且能够给它们命名（在另外一个注册库中）。同时，虽然无法和表建立链接，但是 ETS 具有一个类似的机制，在本节结束时会介绍。

所有 ETS 表天生就能存储 Erlang 元组，并且只能存储元组。元组中可以包含任何想要的东西，元组中的某个元素会作为排序时使用的主键。例如，把一组形如`{Name, Age, PhoneNumber, Email}`的人员信息元组存储在 ETS 表中，看起来会像下面的样子：

```
{Name, Age, PhoneNumber, Email},
{Name, Age, PhoneNumber, Email},
{Name, Age, PhoneNumber, Email},
{Name, Age, PhoneNumber, Email},
{Name, Age, PhoneNumber, Email},
...
```

如果想把电子邮件地址作为表的索引，我们可以让 ETS 把主键的位置设置成 4（25.3.1 节会介绍这项功能）。主键确定之后，你可以选择不同的数据存储方式。

set

`set` 类型的表要求主键的值必须唯一。在上面数据库例子中，电子邮件地址不能重复。当需要一个具有常量访问时间的标准键/值型数据库时，`set` 型表是一个不错的选择。

ordered_set

`ordered_set` 类型也要求表中主键值唯一，但是它还增加了几个有趣的特性。首先，`ordered_set` 表中的元素是有序的（谁能想到呢？）。表中的第一个元素是最小的，最后一个元素是最大的。如果以迭代的方式遍历一个表（反复跳到下一个元素），值应该是递增的，set 表不会保证这一点。当需要频繁进行范围操作时（例如，“我想要 12 到 50 之间的记录”），`ordered_set` 表非常合适。不过，它们也有缺点，访问 `ordered_set` 表要慢一些（它的访问时间是 $O(\log N)$，N 为所存储的记录数量）。

bag

在 `bag` 表中允许存在多个主键相同的元组，只要这些元组本身不完全相同。这意味着，表中可以同时包含`{key, some, values}`和`{key, other, values}`而不会出现问题，这对于 `set` 表来说是不可能的（这两个元组的主键一样）。不过，表中不能包含两条`{key, some, values}`记录，因为它们是完全一样的。

duplicate_bag

这种类型的表和 `bag` 表类似，不过它允许同一个表中存在多条完全一样的元组。

注意 `ordered_set` 表的所有操作都会把值 1 和 1.0 看作是相等的。其他类型的表则认为它们是不同的。虽然能够同时使用整数和浮点数是一种不错的特性，但是很少有函数会将这两种类型都作为返回类型。一般来讲，在程序中把返回值限定为其中的一种类型可以避免很多问题。

ETS 中还有一个概念比较常见，那就是表的所有权。当一个进程调用了创建新 ETS 表的函数时，该进程就成为这个表的拥有者。

默认情况下，只有表的拥有者才可以向表中写入数据，其他进程只能从表中读取数据。这称为 protected 级别的访问权限。你可以把权限设置成 public，这样所有进程都可以对表进行读写操作，或者设置成 private，此时只有表的拥有者有权限对表进行读写。

表所有权的概念还有一些更深层次的含义。ETS 表和拥有者进程是紧密连接在一起的。如果进程死了，表就会消失（内容也都消失了）。不过，表的控制权是可以移交的，这一点和套接字及其控制进程的概念相像，可以给表指定一个继承者进程，如果拥有者进程死了，表会被自动移交给这个继承者进程。

25.3　ETS 的基本操作

现在，我们已经掌握了 ETS 的一些概念，可以接着学习表的基本操作函数了。ETS 函数可以创建、删除表，还可以进行数据的插入和查询。

25.3.1　表的创建和删除

要创建一个 ETS 表，可以调用函数 ets:new/2。这个函数的第一个参数是 Name（原子类型），接着是一个选项列表。该函数返回一个唯一的标识符，这个标识是使用表时必需的，和进程的 pid 类似。

可用的选项如下。

Type = set | ordered_set | bag | duplicate_bag

这个选项设置表的类型，之前对类型做过介绍。默认值是 set。

Access = private | protected | public

这个选项设置表的访问权限，之前做过介绍。默认值是 protected。

named_table

非常奇怪，如果调用 ets:new(some_name, [])，得到的将是一个具有 protected 权限、没有名字的 set 表。如果想通过名字来访问表（名字要唯一），就必须把 named_table 选项传递给这个函数。否则，表的名字只用于归档，会显示在 ets:i()（该函数会打印出系统中所有 ETS 表的信息）之类的函数输出中。

{keypos, Position}

你应该记得，ETS 表中存储的都是元组。Position 参数是一个 1 到 *N* 之间的整数值，指定元组中的哪个元素充当数据表的主键。默认的主键位置为 1。这意味着，在使用记录结构时，要非常小心，因为记录结构的第一个元素是记录名（还记得记录结构以元组方式呈现时的形式吧）。如果想把记录结构中的其他字段当作主键，可以使用 {keypos, #RecordName.FieldName}，它会返回 FieldName 在记录元组表示中的位置。

{heir, Pid, Data} | {heir, none}

ETS 表有一个父进程。如果该进程死了，表就会消失。如果你希望保留表中的数据，那么可以定义一个继承者进程。当表的父进程死亡时，继承者进程会收到一条{'ETS-TRANSFER', TableId, FromPid, Data}消息，消息中的 Data 就是该选项定义中的那个 Data 参数。表会自动被继承者接管。默认情况下，表是没有继承者的。可以在创建表后，通过调用 ets:setopts(Table, {heir, Pid, Data})或者 ets:setopts(Table, {heir, none})去定义或者更改继承者。如果只是想转交表的控制权，可以调用 ets:give_away(Tab, Pid, Data)。

{read_concurrency, true | false}

这个选项用来优化表的并发读操作。把这个选项设置为 true 意味着表的读取操作会变得非常廉价，但是在切换到写操作时会更加昂贵。简单来讲，当读取操作很多，而写操作很少，并且希望好一点的性能时，应该把这个选项设置为 true。如果读写操作均衡、交叠，那么使用这个选项反而会损伤性能。

{write_concurrency, true | false}

通常情况下，对表的写入操作会使用一个全局锁，在写操作结束前，任何其他进程都无法访问表，无论是读还是写。把这个选项设置为 true，可以让读写操作并发进行，而不影响 ETS 的 ACID 属性。不过，这样做会降低单个进程的顺序写入性能以及表的并发读性能。当读写操作的量都很巨大时，可以把这个选项和 read_concurrency 选项结合起来使用。

> **注意** ACID 代表原子性、一致性、隔离性和持久性。ACID 属性是可靠的数据库事务系统所需要的属性。更多的信息，可以参见 http://en.wikipedia.org/wiki/ACID。

compressed

使用这个选项可以对表中的数据进行压缩，压缩是针对除主键之外的其他大部分字段进行的。当需要查验表中的所有元素时，这个选项会导致一定的性能开销。

和表创建相反的操作是表的删除。删除表时只要调用 ets:delete(Table)即可，其中 Table 可以是表的 ID，也可以是具名表的名字。如果想删除表中的单条记录，可以调用一个非常相似的函数：ets:delete(Table, Key)。

25.3.2 数据的插入和查询

还有另外两个基本的表操作函数，即 ets:insert(Table, ObjectOrObjects)和 ets:lookup(Table, Key)。在 insert/2 中，参数 ObjectOrObjects 既可以是单个元组，也可以是一个元组列表。

```
1> ets:new(ingredients, [set, named_table]).
ingredients
2> ets:insert(ingredients, {bacon, great}).
true
3> ets:lookup(ingredients, bacon).
```

```
[{bacon,great}]
4> ets:insert(ingredients, [{bacon, awesome}, {cabbage, alright}]).
true
5> ets:lookup(ingredients, bacon).
[{bacon,awesome}]
6> ets:lookup(ingredients, cabbage).
[{cabbage,alright}]
7> ets:delete(ingredients, cabbage).
true
8> ets:lookup(ingredients, cabbage).
[]
```

你会注意到，lookup 函数返回的是一个列表。对于所有类型的表，它都返回列表，即使这个表是至多返回一条记录的 set 类型的表。这就意味着，即便使用了 bag 或者 duplicate_bag 表（这些类型的表对单个键值会返回多个结果），也能够以通用的方式处理 lookup 函数的返回结果。

上述代码中，还有一件事情值得注意，对同一个键插入两次会覆盖第一次的数据。对于 set 和 ordered_set 类型的表，始终是这样，但是对于 bag 或者 duplicate_bag 类型的表则不然。如果想避免这种情况，可以使用 ets:insert_new/2 函数，这个函数仅在表中不包含相同主键的记录时才会插入数据。

> **注意**　在 ETS 表中，元组中元素的个数不必都一样，虽然让元组具有相同大小是一种最佳实践。不过，元组中元素的数量不能小于主键所在的那个位置。

如果只想获取元组中的部分内容，可以使用另外一个查询函数 ets:lookup_element(TableID, Key, PositionToReturn)。该函数会返回匹配上的元素（如果是 bag 或者 duplicate_bag 类型的表，就会返回一个匹配元素列表）。如果没找到任何匹配，就会出错，原因为 badarg。

我们用 bag 类型的表来实验一下前面的例子：

```
9> TabId = ets:new(ingredients, [bag]).
16401
10> ets:insert(TabId, {bacon, delicious}).
true
11> ets:insert(TabId, {bacon, fat}).
true
12> ets:insert(TabId, {bacon, fat}).
true
13> ets:lookup(TabId, bacon).
[{bacon,delicious},{bacon,fat}]
```

因为是 bag 类型的表，所以即使我们插入了两次{bacon, fat}，表中也只含有一个{bacon, fat}，但是可以看到，表中包含了两个以 bacon 为主键的条目。从这个例子中还可以看出，在没有使用 named_table 选项的情况下，必须用 TabId 操作表。

> **注意** 如果你在运行这些例子期间，shell 进程崩溃了，这些表也会消失，因为它们的父进程（shell）死亡了。

最后一组可用的基本操作是逐个遍历表中的元素，ordered_set 类型的表最适合这项任务：

```
14> ets:new(ingredients, [ordered_set, named_table]).
ingredients
15> ets:insert(ingredients,
15>             [{ketchup, "not much"}, {mustard, "a lot"},
15>              {cheese, "yes", "goat"}, {patty, "moose"},
15>              {onions, "a lot", "caramelized"}]).
true
16> Res1 = ets:first(ingredients).
cheese
17> Res2 = ets:next(ingredients, Res1).
ketchup
18> Res3 = ets:next(ingredients, Res2).
mustard
19> ets:last(ingredients).
patty
20> ets:prev(ingredients, ets:last(ingredients)).
onions
```

可以看出，元素是按照顺序排列的，可以依次访问它们，既可向前，亦可向后。

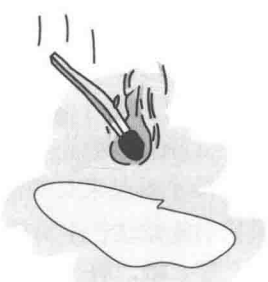
噢耶，我们来看看到表的边界时会出现什么情况：

```
21> ets:next(ingredients, ets:last(ingredients)).
'$end_of_table'
22> ets:prev(ingredients, ets:first(ingredients)).
'$end_of_table'
```

当看到以$开始的原子时，就应该知道它们是些特殊的值（OTP 团队按照惯例选择的），用来指示发生了某种情况。当试图迭代到表的边界外时，会返回$end_of_table 原子。一般情况下，为了避免混乱或者与 OTP 内部定义的原子冲突，请不要使用以$开始的原子。

现在，我们已经知道了如何把 ETS 当成一个基本的键/值存储使用。接下来，我们学习一些更高阶的内容。

25.4 匹配操作



如果在查询时，不仅仅只想基于主键进行匹配该怎么办呢？当你思考这个问题时，能想到的最好的数据选择方法肯定是模式匹配，对吧？如果能以某种方式把匹配模式保存在一个变量中（或者作为一个数据结构），然后把它传给某个 ETS 函数，并让那个函数完成实际的工作，该有多好呀。

这种理想的方式称为*高阶模式匹配*，遗憾的是，Erlang 对此并不支

持。事实上，支持它的语言几乎没有。不过，Erlang 提供了一种子语言，Erlang 程序员们一致使用这种语言来描述要匹配的模式，模式会被描述成一串常规的数据结构。

这种语言的表示法是基于元组的，非常适用于 ETS。使用它，可以简单地定义一些变量（常规变量和“可忽略”变量），这些变量可以放在元组中进行模式匹配。常规变量的书写形式为'$0'、'$1'、'$2'等（数值编号不重要，只用来标识获取的结果）。“可忽略”变量可被书写成'_'。可以把这些原子（'_'、'$0'、'$1'等）放到一个元组中，用这个元组来表示一种表记录模式，例如：

```
{items, '$3', '$1', '_', '$3'}
```

这个模式和常规模式匹配{items, C, A, _, C}的含义大致等价。据此，可以推测出，元组的第 1 个元素必须是原子 items，第 2 个元素和第 5 个元素必须相等。

为了让这种表示法派上用场，ETS 提供了两个函数：match/2 和 match_object/2。（还有另外两个函数 match/3 和 match_object/3，不过它们的用法超出了本书的范围——希望读者自行学习相关的文档。）

函数 match/2 返回模式中的变量，函数 match_object/2 则返回和模式匹配的完整记录：

```
1> ets:new(table, [named_table, bag]).
table
2> ets:insert(table, [{items, a, b, c, d}, {items, a, b, c, a},
2>                    {cat, brown, soft, loveable, selfish},
2>                    {friends, [jenn,jeff,etc]}, {items, 1, 2, 3, 1}]).
true
3> ets:match(table, {items, '$1', '$2', '_', '$1'}).
[[a,b],[1,2]]
4> ets:match(table, {items, '$114', '$212', '_', '$6'}).
[[d,a,b],[a,a,b],[1,1,2]]
5> ets:match_object(table, {items, '$1', '$2', '_', '$1'}).
[{items,a,b,c,a},{items,1,2,3,1}]
6> ets:delete(table).
true
```

函数 match/2 有一点很好，它只返回那些严格需要的东西。这种特性很有用，因为 ETS 表遵循无共享原则（之前介绍过）。如果记录非常大，那么只复制必要的字段是非常值得做的一件事情。

你可能已经注意到，虽然变量的编号没有什么明确的意义，但是编号的大小顺序是重要的。在最终的返回值列表中，模式中$114 所绑定的值总是会出现在$6 所绑定的值之后。如果没有任何匹配，就会返回空列表。

也可以基于模式匹配删除表中的记录。此时，可以使用 ets:match_delete(Table, Pattern)函数。

除了看起来有些奇怪，这种方法还是不错的，它能够对任何字面值进行模式匹配。当然，如果我们能做这些事情就更好了：进行比较和设定范围、清楚明白的输出格式（也许我们不想要列表）等。哦，等一下，你可以办到！

25.5 选择操作

Erlang 提供了一种方法，使用这种方法，我们可以获得某种和函数头的模式匹配几乎等价的能力，包括非常简单的卫语句（guard）。如果你曾使用过 SQL 数据库，应该见过很多种包含元素比较的查询方法，如大于、等于和小于。我们也想在 ETS 中使用这种好的特性。

为此，Erlang 的开发者们对匹配的语法进行了近乎疯狂的增强，直到它足够强大为止。糟糕的是，它的可读性也大大降低了。它看起来像这样：

```
[{{'$1','$2',<<1>>,'$3','$4'},
  [{'andalso',{'>','$4',150},{'<','$4',500}},
   {'orelse',{'==','$2',meat},{'==','$2',dairy}}],
  ['$1']},
 {{'$1','$2',<<1>>,'$3','$4'},
  [{'<','$3',4.0},{is_float,'$3'}],
  ['$1']}]
```

这太丑了——你可不希望数据结构设计成这个样子。信不信由你，我们将要学习如何编写这种称为匹配规格说明的东西，不过，不会写成上面这种形式——千万别，写成这种形式也太难了。但是，我们必须要学会读懂它们！

从一个较高的视角来看，匹配规格说明的形式如下：

```
[{InitialPattern1, Guards1, ReturnedValue1},
 {InitialPattern2, Guards2, ReturnedValue2}].
```

视角再高一点，我们会看到这种形式：

```
[Clause1,
 Clause2]
```

这些东西大致上等同于函数头中的模式、卫语句以及函数体。`InitialPattern` 中仍然只能使用`'$N'`变量，和匹配函数一样。卫语句是新增的内容，可以用它来做些和常规卫语句类似的工作。例如，卫语句`[{'<','$3',4.0},{is_float,'$3'}]`和`... when Var < 4.0, is_float(Var) -> ...`非常相似。

再看一个更复杂的卫语句，如下：

```
[{'andalso',{'>','$4',150},{'<','$4',500}},
 {'orelse',{'==','$2',meat},{'==','$2',dairy}}]
```

它的含义和`... when Var4 > 150 andalso Var4 < 500, Var2 == meat orelse Var2 == dairy -> ...`一样。明白了吗？

所有运算符或者卫函数都使用前缀语法，也就是说，使用这样的顺序：`{FunctionOrOperator, Arg1, ..., ArgN}`。因此，`is_list(X)`变成了`{is_list, '$1'}`，`X andalso Y`变成了`{'andalso', X, Y}`，等等。必须把像 `andalso` 和 `orelse` 之类的保留关键字以及像`==`之类的运算符转换成原子，否则 Erlang 解析器会报错。

模式的最后一部分是想要返回的内容。只要把变量放在那里就行了。如果想返回整个匹配对象，可以使用变量'$_'。匹配规格说明的完整介绍可以参见 Erlang 文档：http://www.erlang.org/doc/apps/erts/match_spec.html。

我之前曾经说过，我们不会以这种方式来编写模式，因为还有一种更好的写法。ETS 具有解析转换（parse transform）功能。解析转换是一种未公开（在使用过程中无论遇到什么问题，OTP 团队都不会提供正式的支持）的技术，可以在编译过程中访问 Erlang 的解析树。勇敢的 Erlang 程序员可以用这种技术把模块代码转换成另外一种新形式。只要不改变语言的语法和记号（token），解析转换几乎无所不能，可以把现有的 Erlang 代码改变成其他任何东西。

如果模块想使用 ETS 自带的解析转换功能，就必须要手工使能。做法如下：

```
-module(SomeModule).
-include_lib("stdlib/include/ms_transform.hrl").
...
some_function() ->
    ets:fun2ms(fun(X) when X > 4 -> X end).
```

`-include_lib("stdlib/include/ms_transform.hrl").`这行引入了一些非同寻常的代码，这些代码会改写模块中使用的 `ets:fun2ms(SomeLiteralFun)` 的含义。和高阶函数不同，解析转换会对 fun 中的内容（模式、卫语句以及返回值）进行分析，移除对函数 `ets:fun2ms/1` 的调用，并用一个真正的匹配规格说明取代它。很不可思议，对吗？最棒的还在于，这一切都发生在编译期，采用这种方法不会导致任何运行时开销。

我们可以在 shell 上进行尝试，此时无需这个头文件：

```
1> ets:fun2ms(fun(X) -> X end).
[{'$1',[],['$1']}]
2> ets:fun2ms(fun({X,Y}) -> X+Y end).
[{{'$1','$2'},[],[{'+','$1','$2'}]}]
3> ets:fun2ms(fun({X,Y}) when X < Y -> X+Y end).
[{{'$1','$2'},[{'<','$1','$2'}],[{'+','$1','$2'}]}]
4> ets:fun2ms(fun({X,Y}) when X < Y, X rem 2 == 0 -> X+Y end).
[{{'$1','$2'},
  [{'<','$1','$2'},{'==',{'rem','$1',2},0}],
  [{'+','$1','$2'}]}]
5> ets:fun2ms(fun({X,Y}) when X < Y, X rem 2 == 0; Y == 0 -> X end).
[{{'$1','$2'},
  [{'<','$1','$2'},{'==',{'rem','$1',2},0}],
  ['$1']},
 {{'$1','$2'},[{'==','$2',0}],['$1']}]
```

现在，一切写起来都太容易了！当然，fun 理解起来也更简单一些。

本节开始处的那个复杂例子该如何表达呢？可以用 fun 来进行表达，如下：

```
6> ets:fun2ms(fun({Food, Type, <<1>>, Price, Calories})
6>   when Calories > 150 andalso Calories < 500,
6>        Type == meat orelse Type == dairy;
6>        Price < 4.00, is_float(Price) ->
```

```
6> Food end).
[{{'$1','$2',<<1>>,'$3','$4'},
  [{'andalso',{'>','$4',150},{'<','$4',500}},
   {'orelse',{'==','$2',meat},{'==','$2',dairy}}],
  ['$1']},
 {{'$1','$2',<<1>>,'$3','$4'},
  [{'<','$3',4.0},{is_float,'$3'}],
  ['$1']}]
```

尽管这个匹配规格说明乍看之下并非完全合理，但至少说明，如果用名字而非编号来表示变量，在含义理解方面会变得简单一些。

有一件事情需要注意，并不是所有的 fun 都是有效的匹配规格说明：

```
7> ets:fun2ms(fun(X) -> my_own_function(X) end).
Error: fun containing the local function call 'my_own_function/1' (called in body)
 cannot be translated into match_spec
{error,transform_error}
8> ets:fun2ms(fun(X,Y) -> ok end).
Error: ets:fun2ms requires fun with single variable or tuple parameter
{error,transform_error}
9> ets:fun2ms(fun([X,Y]) -> ok end).
Error: ets:fun2ms requires fun with single variable or tuple parameter
{error,transform_error}
10> ets:fun2ms(fun({<<X/binary>>}) -> ok end).
Error: fun head contains bit syntax matching of variable 'X', which cannot be
translated into match_spec
{error,transform_error}
```

函数头中只能匹配单个变量或者单个元组，返回值中不能调用非卫语句函数，并且不允许在二进制表示内部赋值。请在 shell 中做做试验，看看可以做哪些操作。

要让匹配规格说明有用，当然得去使用它们了。下面这些函数使用了规格匹配说明。

- ets:select/2 提取匹配结果。
- ets:select_reverse/2 以反序获取 ordered_set 表的结果（如果是其他类型的表，则和 select/2 一样）。
- ets:select_count/2 返回与规格说明匹配的结果的数量。
- ets:select_delete(Table, MatchSpec) 删除与规格说明匹配的记录。

保持冷静

像 ets:fun2ms 之类的函数看起来很酷，对吧？不过在使用时，要非常小心！ets:fun2ms 函数有一个问题，当在 shell 中时，它可以处理动态的 fun（你可以把 fun 传给它，它会接受），不过，如果 ets:fun2ms 处在编译过的模块中，就不能处理 shell 中定义的动态 fun 了。这是因为，Erlang 有两种 fun：shell fun 和模块 fun。

模块 fun 会被编译成 VM 可以理解的紧凑形式。它们是不透明的，无法查验到它的内部细节。而 shell fun 则是还没有被求值的抽象数据项。它们被定义成某种形式，可以让 shell 调用求

值器对它们进行求值。因此，函数 fun2ms 有两个版本：一个用于编译过的代码，一个用于 shell。

这没啥问题，只是不同类型的 fun 之间是不能互换的。这意味着，不能在 shell 中用编译过的 fun 去调用 ets:fun2ms 函数，也不能把一个动态的 fun 传递给一段编译过的代码，在那里以它为参数调用 fun2ms。太糟糕了！

我们来试一下。首先为表定义一个记录，然后向表中写入内容：

```
11> rd(food, {name, calories, price, group}).
food
12> ets:new(food, [ordered_set, {keypos,#food.name}, named_table]).
food
13> ets:insert(food, [#food{name=salmon, calories=88, price=4.00, group=meat},
13>  #food{name=cereals, calories=178, price=2.79, group=bread},
13>  #food{name=milk, calories=150, price=3.23, group=dairy},
13>  #food{name=cake, calories=650, price=7.21, group=delicious},
13>  #food{name=bacon, calories=800, price=6.32, group=meat},
13>  #food{name=sandwich, calories=550, price=5.78, group=whatever}]).
true
```

接下来，就可以基于不同的卡路里选择食物了：

```
14> ets:select(food, ets:fun2ms(fun(N = #food{calories=C}) when C < 600 -> N end)).
[#food{name = cereals,calories = 178,price = 2.79,group = bread},
 #food{name = milk,calories = 150,price = 3.23,group = dairy},
 #food{name = salmon,calories = 88,price = 4.0,group = meat},
 #food{name = sandwich,calories = 550,price = 5.78,group = whatever}]
15> ets:select_reverse(food, ets:fun2ms(fun(N = #food{calories=C}) when C < 600 ->
N end)).
[#food{name = sandwich,calories = 550,price = 5.78,group = whatever},
 #food{name = salmon,calories = 88,price = 4.0,group = meat},
 #food{name = milk,calories = 150,price = 3.23,group = dairy},
 #food{name = cereals,calories = 178,price = 2.79,group = bread}]
```

或者，我们想要些美味的食物：

```
16> ets:select(food, ets:fun2ms(fun(N = #food{group=delicious}) -> N end)).
[#food{name = cake,calories = 650,price = 7.21,group = delicious}]
```

删除操作稍微有点古怪。你得在模式中返回 true，而非其他值：

```
17> ets:select_delete(food, ets:fun2ms(fun(#food{price=P}) when P > 5 -> true end)).
3
18> ets:select_reverse(food, ets:fun2ms(fun(N = #food{calories=C}) when C < 600 ->
N end)).
[#food{name = salmon,calories = 88,price = 4.0,group = meat},
 #food{name = milk,calories = 150,price = 3.23,group = dairy},
 #food{name = cereals,calories = 178,price = 2.79,group = bread}]
```

从最后一个选择操作的结果可以看出，价格超过 5.00 美元的记录已经被从表中删除。

ETS 模块中还有许多其他函数，例如，把 ETS 表转换成列表或者文件（ets:tab2list/1, ets:tab2file/1, ets:file2tab/1），获取所有 ETS 表的信息（ets:i/0,

ets:info(Table)）等。请去查看官方文档以了解更多关于这些函数的内容。

还有一个名为 tv（表查看器）的模块，可以用它来可视化地管理 Erlang VM 上的 ETS 表。只要调用 tv:start()，就会打开一个窗口，其中会显示所有的表。

25.6 DETS

DETS 是 ETS 的磁盘实现版本，它们之间的主要不同在于：

- DETS 中没有 ordered_set 类型的表；
- DETS 磁盘文件的大小不能超过 2 GB；
- 像 prev/1 和 next/1 之类的操作不再安全和快速；
- 表的创建和关闭有些变化。创建新的数据库表要调用 dets:open_file/2，关闭这个表可以调用 dets:close/1。此后，可以调用 dets:open_file/1 重新打开该表。

其他的 API 和 ETS 基本一样，DETS 为我们提供了一种读写文件中数据的简单方法。

保持冷静

DETS 很慢，因为它的数据只保存在磁盘中。你可能会想把 ETS 和 DETS 表结合起来，从而得到一个在 RAM 和磁盘中都保存数据的高性能数据库。

如果真想这么做，那么最好去了解一下 Mnesia（在第 29 章中会介绍）数据库系统，它就是这么做的，同时还提供了对分区（sharding）、事务和分布式的支持。

25.7 少说一点，多做一点

在这个冗长的节标题（以及前面的冗长介绍）之后，我们转向一个实际的问题，就是本章开始时，促使我们学习 ETS 的那个问题：在 regis 中使用 ETS，消除潜在的性能瓶颈。

在开始之前，我们要考虑一下操作处理的方式，以及哪些操作是安全的，哪些是不安全的。不做任何修改的单次查询操作（不是一段时间内的多次查询）应该是安全的。任何进程可以在任何时间点执行这些操作。其他所有会涉及写入、更新、删除以及有一致性要求的多次读取等操作都是不安全的。

因为 ETS 不提供任何事务支持，所以所有不安全的操作都要放到表的拥有者进程中执行。那些安全的操作应该是公共的——可以在拥有者进程之外执行。在修改 regis 时，我们要牢记这一点。

首先，我们要制作一份 regis-1.0.0 的副本：regis-1.1.0（从 http://learnyousomeerlang.com/static/erlang/regis-1.1.0.zip 可以下载到 regis-1.1.0 的代码）。我加大了版本编号中的第二个数字，没有加大第三个数字，因为我们要做的更改不会破坏现有的接口，从技术层面讲，也不属于 bug 修正，所以这个版本只是一个特性升级。

25.7.1　接口

在新目录中，首先需要修改的只有 regis_server.erl 文件。我们不会对接口进行变动，因此其他结构方面的内容也不需要大的更改。

```
%%% 应用的核心：负责跟踪进程的服务器。
-module(regis_server).
-behavior(gen_server).
-include_lib("stdlib/include/ms_transform.hrl").

-export([start_link/0, stop/0, register/2, unregister/1, whereis/1,
         get_names/0]).
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         code_change/3, terminate/2]).

%%%%%%%%%%%%%%%%%%
%%% INTERFACE %%%
%%%%%%%%%%%%%%%%%%
start_link() ->
    gen_server:start_link({local, ?MODULE}, ?MODULE, [], []).

stop() ->
    gen_server:call(?MODULE, stop).

%% 注册进程名字
register(Name, Pid) when is_pid(Pid) ->
    gen_server:call(?MODULE, {register, Name, Pid}).

%% 解除注册进程的名字
unregister(Name) ->
    gen_server:call(?MODULE, {unregister, Name}).

%% 根据名字获取进程标识
whereis(Name) -> ok.

%% 获取所有已注册的名字
get_names() -> ok.
```

在公共接口函数中，只需要对 whereis/1 和 get_names/0 进行更改和重写。这是因为，正如前面提到的，它们是单次读取的安全操作。其余的函数都需要放到表的拥有者进程中串行化执行。

API 就介绍到这里。下面来看看模块的内部实现。

25.7.2　实现细节

我们要使用 ETS 表来完成存储功能，因此在 init 函数中创建表是合理的。此外，由于函数 whereis/1 和 get_names/0 会对表进行公共的访问（性能方面的原因），因此有必要给表起个用于外部访问的名字。给表命名和给进程命名取得的效果类似，我们可以把表名硬编码到函数中，这样就无需到处传递表的 ID 了。

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% GEN_SERVER CALLBACKS %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
init([]) ->
    ?MODULE = ets:new(regis, [set, named_table, protected]),
    {ok, ?MODULE}.
```

接下来的函数是handle_call/3，该函数会处理register/2中定义的消息{register, Name, Pid}。

```
handle_call({register, Name, Pid}, _From, Tid) ->
    %% 已有的表记录中不能包含 Name 和 Pid，因此我们用如下匹配规格说明对表进行遍历，匹配这两个变量。
    MatchSpec = ets:fun2ms(fun({N,P,_Ref}) when N==Name; P==Pid -> {N,P} end),
    case ets:select(Tid, MatchSpec) of
        [] -> % 可以插入
          Ref = erlang:monitor(process, Pid),
          ets:insert(Tid, {Name, Pid, Ref}),
          {reply, ok, Tid};
        [{Name,_}|_] -> % 可能有多个结果，名字匹配上了。
          {reply, {error, name_taken}, Tid};
        [{_,Pid}|_] -> % 可能有多个结果，Pid 匹配上了。
          {reply, {error, already_named}, Tid}
    end;
```

这是到目前为止模块中最复杂的一个函数。有3个基本规则需要遵循。

- 进程不能被注册两次。
- 名字不能被使用两次。
- 如果不违背上面两条规则，进程就可以被注册。

这正是上面代码所做的工作。我们使用从fun({N,P,_Ref}) when N==Name; P==Pid -> {N,P} end导出的匹配规格说明遍历整个表，寻找那些和要注册的名字或者pid相匹配的记录。如果存在匹配，会将名字和pid都返回。这种做法可能会让你觉得有些奇怪，但是当我们看到后面case ... of中使用的匹配模式时，就会明白将这两个变量都返回是有意义的。

第一个模式意味着没有匹配存在，因此可以执行插入操作。我们会监控已注册的进程（以便于出故障时能够解除注册这个进程），接着把记录增加到表中。如果要注册的名字已经存在于表中，会匹配上模式[{Name,_}|_]。如果pid已经存在，则会匹配上模式[{_,Pid}|_]。这就是为什么要返回这两个值：不在匹配规格说明中区分到底是Pid还是Name匹配上了，在后面针对整个元组的匹配中进行判断会简单一些。

为什么匹配模式采用[Tuple|_]的形式，而不是[Tuple]呢？原因很简单：在遍历表寻找相匹配的pid或者名字时，所返回的列表可能是[{NameYouWant, SomePid},{SomeName, PidYouWant}]。如果返回这种结果，[Tuple]这样的匹配模式会导致表的拥有者进程崩溃，太糟糕了。

哦，对了，不要忘了把-include_lib("stdlib/include/ms_transform.hrl").增加到模块中，否则fun2ms就会导致进程死亡，原因很奇怪：

```
** {badarg,{ets,fun2ms,
            [function,called,with,real,'fun',should,be,transformed,with,
             parse_transform,'or',called,with,a,'fun',generated,in,the,
             shell]}}
```

这就是忘了引入这个头文件所导致的错误。想想自己曾经受到的警告。过马路前要左右看，不要去穿越河流，不要忘了必须引入的头文件。

接下来的代码要处理的是解除注册进程的请求消息：

```
handle_call({unregister, Name}, _From, Tid) ->
    case ets:lookup(Tid, Name) of
        [{Name,_Pid,Ref}] ->
            erlang:demonitor(Ref, [flush]),
            ets:delete(Tid, Name),
            {reply, ok, Tid};
        [] ->
            {reply, ok, Tid}
    end;
```

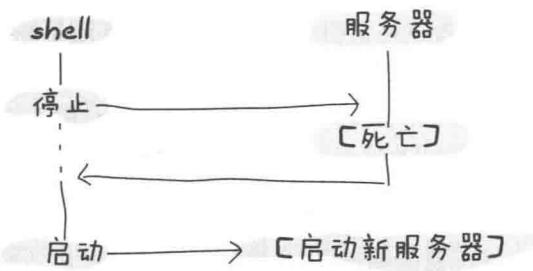
这段代码和老版本的比较相似。思路很简单：找到监控器引用（通过名字进行查询），取消监控，接着删除记录。如果没有找到记录，也会假装删除成功了，这样，每个人都很愉快。哦，我们太不诚实了。

下面的代码处理服务器停止消息：

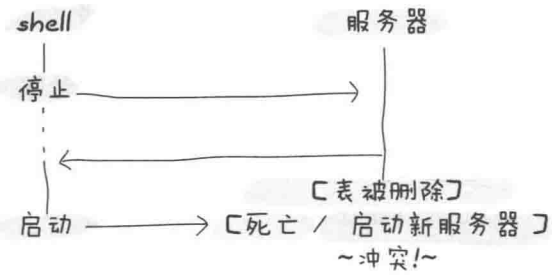
```
handle_call(stop, _From, Tid) ->
    %% 考虑到这个调用是同步的，并且因为清空 ETS 表可能比删除内存中的数据结构耗费的时间长一些，
    %% 所以在此删除表会更安全一些，可以避免一些微妙的竞争条件，尤其是在那些会大量执行服务器
    %% start/stop 操作的测试中。在常规代码中，无需这样做
    ets:delete(Tid),
    {stop, normal, ok, Tid};
handle_call(_Event, _From, State) ->
    {noreply, State}.
```

正如代码注释中所说，不去管 ETS 表，让它被垃圾回收掉也是可以的。不过，我们在第 24 章中编写的测试套件会不停地启动和停止服务器，此时它们之间的延迟就有点危险了。

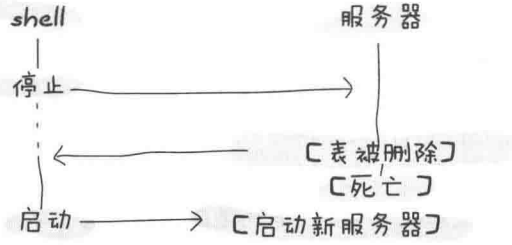
下面是老版本代码中进程的时间线视图：



如果新版本中不删除 ETS 表，就可能会发生如下的情况：



如果删除了 ETS 表，我们就把更多的工作放在代码的同步处理中，从很大程度上避免了错误的发生：



如果你不会频繁地运行测试套件，就可以完全不用关心这些事情。我讲这些是为了在出现问题时，你不会觉得莫名其妙。在非测试系统中，这种边界情况基本上不会出现。

下面是其他的 OTP 回调函数：

```
handle_cast(_Event, State) ->
    {noreply, State}.

handle_info({'DOWN', Ref, process, _Pid, _Reason}, Tid) ->
    ets:match_delete(Tid, {'_', '_', Ref}),
    {noreply, Tid};
handle_info(_Event, State) ->
    {noreply, State}.

code_change(_OldVsn, State, _Extra) ->
    {ok, State}.

terminate(_Reason, _State) ->
    ok.
```

在这些函数中，我们只需关心处理 DOWN 消息的那个函数，在收到 DOWN 消息时，意味着我们监控的某个进程死亡了。此时，会根据消息中的引用删除掉相关的记录，然后继续运行。

你可能注意到，我们其实可以在 code_change/3 函数中完成新老 regis_server 之间的转换。具体实现作为练习留给读者完成。我痛恨那些只留练习而不提供答案的图书，因此，为了有别于那些不提供答案的图书作者，我在这里给出一点提示：你需要使用 gb_trees:map/2 或者 gb_trees 迭代器把老版本中两个 GB 树中的内容填写到一个新的 ETS 表中。降级函数则反向

编写。

现在只剩下两个未实现的公共函数了。当然，我们可以写一个%% TODO 注释，结束一天的工作，然后去喝一杯，醉到忘记自己是个程序员，不过那就太不负责任了。我们来完成这项工作：

```
%% 根据名字获取进程标识。
whereis(Name) ->
    case ets:lookup(?MODULE, Name) of
        [{Name, Pid, _Ref}] -> Pid;
        [] -> undefined
    end.
```

这个函数会查询名字，根据记录找到与否返回关联的 pid 或者 undefined。注意，我们在此使用 regis（?MODULE）作为表名——这就是为什么把这个 ETS 表设置成受保护且具名的原因。

下面是第二个函数：

```
%% 获取所有已注册的名字。
get_names() ->
    MatchSpec = ets:fun2ms(fun({Name, _, _}) -> Name end),
    ets:select(?MODULE, MatchSpec).
```

我们再次使用 fun2ms 去匹配名字，在返回结果中只保留了名字信息。选择操作会返回一个列表，这正好是我们需要的。

完工了！你可以运行 test/目录中的测试套件，让系统运行起来：

```
$ erl -make
... <snip> ...
Recompile: src/regis_server
$ erl -pa ebin
... <snip> ...
1> eunit:test(regis_server).
  All 13 tests passed.
ok
```

嗯，很好——现在我觉得我们可以说自己对 ETS 已经相当精通了。

接下来学些什么好呢？我们会去学习 Erlang 分布式编程方面的知识。或许，在结束 Erlang 的学习之前，还会有些让我们脑洞大开的内容。让我们拭目以待。

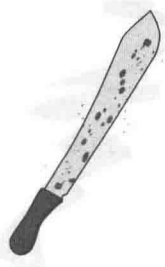
第 26 章

分布式编程

你好，请坐。欢迎来到这里。当你第一次听说 Erlang 时，可能会被它的几个特点所吸引：Erlang 是一门函数式语言，它具有极好的并发语义，并且支持分布式编程。前两方面的内容我们已经介绍过了，还介绍了许多其他方面的内容。现在，我们来介绍最后一项重要内容：分布式编程。

之所以等了这么长时间才介绍这项内容，是因为如果程序不能在单台机器上正常工作，那么把它变成分布式的也没有什么用处。经过千辛万苦，我们终于有资格学习本章的内容了。

和 Erlang 的其他特性类似，在 Erlang 语言中增加分布式支持首先是为了容错。在单台机器上运行的软件始终要面对机器故障、应用下线的风险。如果软件运行在多台机器上，就可以比较容易地处理硬件故障问题，当然前提是，应用的实现必须正确。如果应用运行在多台机器上，但是却不能处理某台机器失效的情况，则是无法容错的。



分布式编程就像你独自一人待在黑暗之中，周围都是怪兽。你非常害怕，不知道要做什么，也不知道有什么东西会扑过来。坏消息是：即使使用了分布式 Erlang，你仍然还是独自一人待在黑暗中和吓人的怪兽作战。困难的工作还是得你自己来做。好消息是：你并非只是有些零钱和差劲的方向感（因此更难杀死怪兽），Erlang 为你提供了一把手电筒、一把砍刀以及一副漂亮的海扁王胡须，让你更有安全感（也适用于女性读者）。

这种结果并不是 Erlang 的实现方式造成的，而是和分布式软件的本质有关。Erlang 提供了一些用于分布式编程的基础构件块：各式各样的节点间通信方式、通信中数据的序列化和反序列化、把多进程的概念扩展到多节点、各种监控网络故障的方法等。但是，它没有提供特定于软件问题的解决方案，如“当某个东西崩溃时该怎么办”。

这是 OTP 中的标准做法，“提供工具，而非解决方案”，前面已经做过介绍。在 Erlang 中，你几乎不能直接得到全功能的软件和应用，但是可以得到许多系统构建组件。Erlang 提供了在系统的某个部分发生故障或者故障恢复时发出通知的工具，以及其他网络相关的工具，但是不存在

自动帮你解决问题的银弹。

我们来看看这些工具的强大威力。

26.1　这是我的火枪

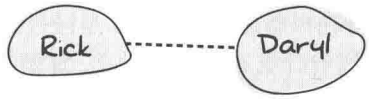
为了对付黑暗中的怪兽，我们被授予了一件非常有用的武器：完全的网络透明性。

一个运行着的、可以和其他 VM 相连的 Erlang VM 实例就是节点（node）。尽管有些编程语言或者社区会把一台服务器当成一个节点，但是在 Erlang 中，每个 VM 都是一个节点。你既可以在单台机器上运行 50 个节点，也可以在 50 台机器上运行 50 个节点，这并没有什么不同。

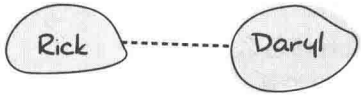
当启动一个节点时，可以为它指定一个名字，该节点会和一个称为 Erlang 端口映射器守护进程（Erlang Port Mapper Daemon，EPMD）的应用进行连接，这个应用运行在 Erlang 集群中的每台计算机上。EPMD 充当着名字服务器，节点会向它进行注册，这样就可以使用名字而非端口号去访问其他节点，在出现名字冲突时，它会给出警告。

此后，这个节点就可以建立和其他节点之间的连接了。连接建立后，它们之间会自动地相互监控，并且在连接断掉或者节点消失时也能够知晓。更重要的是，一个新节点只要和已经连接在一起的某个节点群组中的一个节点建立连接，那么该新节点就会自动和群里的所有节点建立连接关系。

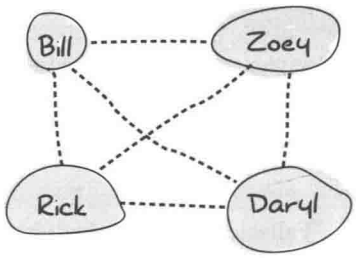
为了解释一下 Erlang 节点间是如何建立连接的，我们可以假想这样一个情景：在僵尸爆发期间，有一组幸存者。这些幸存者是 Zoey、Bill、Rick 以及 Daryl。Zoey 和 Bill 相互认识，通过步话机在同一频段上通信。Rick 和 Daryl 处于孤立状态，如下图所示：



假设 Rick 和 Daryl 在去幸存者营地的路上相遇了。他们共享了步话机的频段，这样就可以一直保持联系了，如下图所示：



后来，Rick 遇到了 Bill。两人对于这次相遇非常高兴，因此也决定共享频段。此时，连接关系扩散开了，最终的连接图如下所示：



这意味着，每两个幸存者之间都可以直接联系。这种连接形式是很有用的，因为任何一个幸存者死亡了，都不会造成某个人被孤立。Erlang 节点之间就是这样的连接关系：所有节点之间都相互连接在一起。

保持冷静

以这样的方式连接节点，虽然对于某些情况有很好的容错效果，但对于伸缩性来说是一个不小的缺陷。很难以这种方式搭建具有数百个节点的 Erlang 集群，因为需要太多的连接关系，并且保持这些连接关系也需要大量的节点间通信。事实上，每连接一个节点，都需要一个暂态端口（ephemeral port）。虽然有些大型项目设法解决了这个问题，不过简单地把节点组分割成小一些的集群是一种更容易的方法。

如果你打算用 Erlang 来构建如此大规模的集群，就请仔细阅读本章，你会知道 Erlang 这样做的原因，还将学习如何解决这个问题。

节点之间建立连接之后，它们保持完全的独立。每个节点都具有自己的进程注册库和 ETS 表（具有自己的表名），节点所加载的模块也是相互独立的。集群中一个连接节点的崩溃不会导致其他节点出问题。

连接着的节点之间可以互相交换消息。Erlang 分布式模型的设计目标就是让本地进程可以和远程进程联系，向其发送常规的消息。如果节点间没有任何共享，所有进程注册库都是独立的话，这怎么可能发生呢？在 26.5.2 节中，我们将深入分布式模型的具体细节，介绍一种访问某个特定节点上注册进程的方法。一旦联系到了进程，就可以发送消息了。

Erlang 的消息会自动进行序列化和反序列化，这项工作对你来说是透明的。所有的数据结构，包括 pid，在本地和远程中的使用方式完全一样。这意味着，我们可以通过网络发送 pid，然后在接收端就可以向这些 pid 发送消息了。更好的是，链接和监控器也支持跨网络建立，只要你能得到 pid！

那么，既然 Eralng 为了让一切看起来透明做了这么多的工作，那为什么我还说它只是为我们提供了一把砍刀、一个手电筒以及一副胡须呢？

26.2 分布式计算中的谬误

正如一把砍刀只能杀死一种类型的怪兽一样，Erlang 所提供的工具只能处理某些种类的分布式计算问题。为了理解 Erlang 所提供的工具，我们首先得了解一下分布式世界中会存在哪些情况，

以及 Erlang 为了容错所做出的假设。

在过去的数十年间，有些非常聪明的家伙花费了不少时间对分布式计算中会出现的问题进行了分类。他们总结出了人们做出的 8 条主要假设（其中有些是 Erlang 设计者综合考虑多方面原因做出的），都是些事后让人自食其果的假设。

> **注意**　分布式计算中的谬误来自“Fallacies of Distributed Computing Explained”，作者是 Arnon Rotem-Gal-Oz，下载地址为 http://www.rgoarchitects.com/Files/fallacies.pdf。

26.2.1　网络是可靠的

分布式计算的第一个谬误是：认为应用可以跨网络分布。这听起来有些怪诞，不过网络确实会经常因为令人讨厌的原因停止工作：停电、硬件故障、某人碰掉了电缆、通向其他空间的涡旋吞噬了关键的组件、猎头蟹侵入、铜缆被盗等。

你会犯的一个最大错误是：认为肯定可以联系到远程节点，并和它们通信。通过增加更多硬件和冗余，也许能够达成这个目标，这样，当某个硬件故障时，应用可以联系其他的硬件。还有另外一件事情要考虑，那就是要做好消息和请求丢失的准备，同时也要做好收不到回应的准备。尤其是当你的软件系统依赖于某些已经不存在的第三方服务时，更要如此（即使此时你自己的软件系统还保持良好运行）。

Erlang 没有提供任何度量网络可靠性的方法，因为这些决策通常都是应用相关的。毕竟，除你之外，还有谁知道某个具体组件的重要性呢？当然，你也不是完全没有工具可用，在分布式 Erlang 中，当失去和其他节点的连接（或者不响应）时，是能够被检测出来的。还有些专门的用于监控节点的函数可供使用，在节点之间的连接中断时，链接和监控器机制也会被触发。

针对这种情况，Erlang 提供的最佳解决方案是其异步通信模式。通过异步地发送消息，并迫使开发者在一切正常时发送一条回应，Erlang 让所有的消息传递活动都能直观地处理错误。如果某个进程正在和你通信，它所在的节点由于网络故障失去了连接，那么你可以很自然地像处理任何本地崩溃一样处理这种情况。这也是 Erlang 之所以具有良好伸缩性的原因之一（性能方面和设计方面都能够伸缩）。

> **保持冷静**
>
> 跨节点的链接和监控具有危险性。在发生网络故障时，所有的远程链接和监控机制会被同时触发。这会产生成千上万条发往不同进程的信号和消息，会给系统带来严重的、非期望的负载。
>
> 为不可靠的网络做准备也意味着为突发的失败做准备，还要保证不能因为某个部件的突发失效导致整个系统受到损害。

26.2.2　网络没有延迟

某些看起来不错的分布式系统其实是一把双刃剑，因为它们隐藏了这样一个事实：发起的函

数调用是远程的。你期望某些函数调用要非常得快，不过一旦调用是跨网络进行的，就不能满足要求了。本地调用和远程调用之间的区别类似于在披萨店中购买披萨和把披萨从另外一个城市送到你家中之间的区别。虽然都需要一个基本的等待时间，但是如果运送花费的过程过长，到你手中时，披萨可能已经凉了。

如果快速返回是个必需的要求，那么忽视了网络通信会使返回变慢（即使消息很小）将会导致严重的后果。Erlang 模型会为我们带来帮助。在 Erlang 中，本地应用都是基于隔离的进程、异步消息以及超时构建的，并且我们总是会把进程失败的可能性考虑进来，因此可以很快把应用变成分布式的。超时、链接、监控器以及异步模式等的语义完全一样，且同样可靠。Erlang 不会偷偷地隐藏延迟的存在，事实上，它总是期望能通过设计显式地进行处理。

不过，你可以在设计中做这个假设，并认为回应的速度比实际速度快一些。只要小心些就好。

26.2.3 带宽是无限的

虽然网络的传输速度变得越来越快，并且每字节的传输成本也越来越低，但是如果认为发送海量数据非常得简单、容易，那还是要承担风险的。

由于 Erlang 构建本地应用所采用的方式，这个问题通常不会在 Erlang 中发生。这里有一个技巧需要牢记：只发送那些描述发生了什么的消息，而不是发送一个全新的状态（玩家 X 发现了物品 Y，而不是把玩家 X 的所有物品不断地发来发去）。

如果由于某些原因确实需要发送很大的消息，一定要非常小心。分布式 Erlang 节点之间的通信对于大的消息非常敏感。如果两个节点连接在一起，那么它们之间所有的通信都基于单条 TCP 连接。因为我们通常会希望保持两个进程间消息的顺序（即使跨网络），所以在这个连接上，消息都是顺序发送的。这就意味着，如果有一个非常大的消息存在，就会阻塞这个通道上的其他所有消息。

更糟糕的是，Erlang 是通过发送心跳消息（heartbeats）来维持节点活性的。心跳消息都很小，定期地在两个节点之间发送，内容大体为“我还活着。继续保持联系！”这种方式跟我们的僵尸幸存者之间定期通过消息互相询问类似。“Bill，你在吗？”如果 Bill 从不回应，那么你可以认为他已经死了（或者没电了），随后你就不再跟他通信了。心跳消息和常规消息在同一条通道上发送。

问题是，大消息会阻塞心跳消息。如果大消息的数量过多，就会长时间地阻塞心跳消息，最终导致某个节点认为对方不再响应并关闭连接。这太糟糕了。

为了防止这种问题发生，有一个非常有用的 Erlang 设计经验，请铭记在心：保持消息短小。这样的话，一切都会很好。

26.2.4 网络是安全的

在分布式系统中，相信一切都很安全是非常危险的。例如，完全相信收到的消息。轻则有人会向你发送伪造的消息、对包进行拦截和修改（或者偷看敏感数据），重则会完全控制你的应用或者系统。

很遗憾，分布式 Erlang 就是这么假设的。Erlang 的安全模型如下所示：

你没看错。这是因为分布式 Erlang 最初只是用于容错和组件冗余的。在 Erlang 语言刚刚出现时，它只用在电话交换机和其他电信应用中，Erlang 系统基本都会被部署在运行环境怪异的硬件上——地处偏远，条件恶劣（有时，由于地面潮湿，工程师得把服务器挂在墙壁上，或者要在森林中安装定制的加热系统以保证硬件能工作在最优的温度环境中）。在这些情况下，用于故障切换的硬件和主硬件位于同样的物理位置。这也是分布式 Erlang 通常的运行环境，这就解释了 Erlang 的设计者们为何会假设网络环境是安全的了。

遗憾的是，这意味着现代的 Erlang 应用无法自动地形成跨多个数据中心的集群。事实上，也不推荐这么做。大多数情况下，基于多个小规模的、互相隔离的 Erlang 节点集群构建系统就足够了，它们通常都位于一处。对于更复杂的情况，需要开发者使用如下方法自行实现。

- 转向使用 SSL。
- 实现自己的高层通信层。
- 基于安全通道建立隧道。
- 重新实现节点间的通信协议。

在安全套接字层用户指南中，对 SSL 的使用方法进行了介绍（第 3 章，在分布式 Erlang 中使用 SSL，参见 http://www.erlang.org/doc/apps/ssl/ssl_distribution.html）。ERTS 用户指南中给出了如何实现自定义分布式 Erlang 承载协议的介绍（第 3 章，如何实现分布式 Erlang 的替代承载协议，参见 http://www.erlang.org/doc/apps/ssl/ssl_distribution.html），其中也包含了关于分布式协议的详细介绍（第 9 章，参见 http://www.erlang.org/doc/apps/erts/erl_dist_protocol.html）。

即使在这种场景中，也必须要小心，因为如果有人获得了分布式节点中一个节点的访问权，那么他就具有了所有节点的访问权，并且可以在这些节点上运行任何命令。

26.2.5　网络拓扑不会变化

在初次设计运行在多台服务器上的分布式应用时，你的脑海中可能会想着固定数目的服务器，或许还有一组主机名。或许，你思考设计时还会使用具体的 IP 地址。这样的做法是不对的。硬件会发生故障、运维人员会搬移服务器、会增加新的机器，还会移除一些机器。网络的拓扑会不断变化。如果在设计时，把具体的网络拓扑硬编码到应用中，那么在网络发生变化时就很难应对。

在 Erlang 中，对此并没有做明显的假设。然而，这种情况却很容易蔓延到你的应用中。每个 Erlang 节点都有一个名字和一个主机名，它们会被不断地更改。对于 Erlang 进程，除了需要考虑如何对它命名外，还得考虑它在集群中的位置。如果你把名字和主机名都进行了硬编码，那么在

以后出现故障时，就可能陷入困境。不过，不必过分担心。在后面 26.10 节中，我们会介绍一些有趣的库，使用它们，在定位具体的进程时，可以完全不用关心节点名和网络拓扑。

26.2.6 只有一个管理员

任何编程语言或者库所提供的分布式功能，都不能保证这一点。这个谬误的意思是，对于你的软件和服务器来说，不会只有一个主管理员，尽管在设计时可能是这样认为的。如果你把多个节点运行在一台计算机上，就不用考虑这个谬论了。不过，如果跨多个不同地点运行应用，或者有某个第三方系统依赖于你的代码，那么就要小心了。

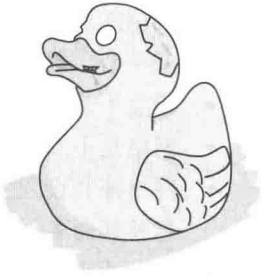
在为其他人提供工具来诊断你系统的问题时，要多加小心。如果可以手动操作 VM，Erlang 系统调试起来就是非常容易的——在需要时，甚至可以在线加载代码。不过，对于坐在节点前无法访问终端的人，则需要为其提供不同的操作工具。

这个谬论的另外一方面内容是，像重启服务器、在数据中心之间转移实例、部分升级软件协议栈之类的工作不一定只由一个人或者一个团队控制。对于非常大的软件项目，很可能有多个团队，甚至多个不同的软件公司协作管理系统的不同部分。

如果你正在为软件栈编写协议，就必须要考虑协议的多个版本，用于应对用户或者合作伙伴在升级其代码时的快慢差异。根据自己的需要，可以选择一开始就在协议中包含版本相关的信息，或者做到能够在操作过程中改变协议。我相信你能够想出更多种可能的出错情况。

26.2.7 传输成本是零

传输成本为零的谬误涉及两个假设。一个针对数据传输所花费的时间，另一个针对数据传送所花费的金钱。



第一个假设认为，像序列化数据这样的操作基本上无需花费时间，非常快，没什么大的影响。在实际中，对大的数据结构进行序列化所花费的时间要比小数据长很多，并且还得在接收方进行反序列化。在网络上进行传输时，无论数据大小都需要花费时间，尽管小消息所花费的时间不那么明显。

第二个假设认为，在金钱方面，数据传送是零成本的。在现代服务器硬件栈中，内存（RAM 或者磁盘）和带宽比起来通常是便宜的，你得不停地支付带宽费用，除非整个网络都是你自己的。所以，在很多情况下，对系统进行优化，使之具有更少的请求和更小消息，是值得的。

在 Erlang 中，由于其最初的使用场景，并没有对此做特别的考虑。例如，对跨节点的消息进行压缩（尽管这类函数已经存在）。相反，原设计者选择让开发者在需要时去实现自己的通信层。于是，这个职责就转移到程序员身上，由他们来保证小的消息，并进行度量去最小化数据传输的成本。

26.2.8 网络是同质的

最后一个假设是说，网络应用的所有部件在互操作时都使用同一种语言，或者同样的格式。

对于我们的僵尸幸存者来说，就是不能认为这些幸存者在做计划时他们都说英语（或者规范的英语），或者大家对某个词的含义理解一致。

在编程方面，通常指的是不要依赖于封闭的标准，而要使用开放的标准，或者随时都可以容

易地从一种协议切换到另外一种。对于 Erlang 来说，虽然分布式协议是完全公开的，但是所有的 Erlang 节点都认为与其通信的其他节点都使用同样的协议。其他类型的系统如果想和 Erlang 集群集成到一起，要么必须得理解、实现 Erlang 的协议，要么必须得为 Erlang 应用提供某种对 XML、JSON 等的翻译层。

实现 Erlang 的协议相对来说比较简单。只要遵从了协议，就可以装成一个 Erlang 节点，即使不是用 Erlang 语言实现的。如果能像鸭子一样嘎嘎叫，像鸭子一样摇摆着走路，那么就一定是只鸭子。这就是 C 节点（C node）背后的原理。C 节点（或者其他非 C 语言实现的节点）是一些实现了 Erlang 协议，可以在集群中装成 Erlang 节点的程序，可以方便地将工作分配在这些节点上。更多细节可以参见 http://www.erlang.org/doc/tutorial/cnode.html。

还有一种数据交换解决方案：使用 BERT 或者 BERT-RPC（文档可参见 http://bert-rpc.org/）。这是一种类似 XML 或者 JSON 的交换格式，其定义和 Erlang 的外部数据项格式有些相似（文档可参见 http://www.erlang.org/doc/apps/erts/erl_ext_dist.html）。

26.2.9　谬误小结

前面介绍了一组和分布式计算谬误有关的假设。简言之，你要始终保持谨慎，并仔细考虑如下几点。

- 不能认为网络是可靠的。在 Erlang 中，除了会检测是否出现故障外（这已经不错了），并没有提供其他任何专门的措施。
- 网络的速度会时不时地变慢。Erlang 提供了异步机制，并考虑了这种情况，不过你还是要小心，确保自己的应用中也考虑了这种可能性。
- 带宽不是无限的。描述性的短小消息有助于减少带宽使用。
- 网络是不安全的，对此，Erlang 没有提供任何的默认帮助。
- 网络拓扑会发生变化。Erlang 对此没有做显式的假设，不过你可能需要对进程的位置信息及其命名方式做些假设。
- 你（或者你的组织）基本不能通盘掌控系统的结构。系统的某些部分可能过时了或者使用了不同的版本，又或者意外地重启或退出了。
- 数据传输是有成本的。同样地，短小的消息有助于降低成本。
- 网络是异质的。不是所有的东西都是一样的，数据交换应该基于具有良好文档说明的格式。

26.3　死亡还是失去联系

上面对分布式计算中谬误的介绍，部分地解释了为什么说我们是在黑暗中和怪兽作战。虽然我们有了一些有用的工具，但是仍有不少问题和困难需要我们自己去解决。在做和谬误相关的设计决策（小消息、减少通信等）时，我们要非常小心。最困难的问题往往和节点死亡或者网络不可靠有关。由于通常很难知道某个东西是死了还是活着（只是不能和它取得联系），会使得这个问题更加严重。

我们还是回到 Bill、Zoey、Rick 和 Daryl 这 4 个幸存者那里。他们在一个安全的房子里面相

遇了，在那里待了几天，吃光了所有能够找到的罐装食物。不久，他们需要离开这个房子，分头到城里寻找更多的资源。他们商定了一个集合地点，就是城市边上的一个小营帐。在寻找资源期间，他们通过步话机保持联系。他们会通告所发现的东西，如其他幸存者。

现在，假设在安全的房子和集合地点之间的某个地方，Rich 想和其他伙伴进行联系。他联系上了 Bill 和 Zoey，但是没法联系上 Daryl。Bill 和 Zoey 也联系不上他。此时的问题是，根本没法知道 Daryl 是被僵尸吃掉了，还是他的电池用光了，还是睡着了，或者只是钻到了地下。团队必须要决定是一直等他呢，还是再呼叫一会儿呢，或者认为他已经死了，进行下一步行动。

同样的两难问题也存在于分布式系统的节点之间。当某个节点不响应时，是因为硬件故障？还是应用崩溃？还是网络拥塞？或者是网络故障？有时，应用已经不再运行了，你可以直接忽略那个节点，继续你的工作。不过，有时应用仍在那个被孤立的节点上运行着，从这个节点的视角来看，所有其他节点都死了。

在默认情况下，Erlang 会把无法联系的节点看作是死节点，把可以联系的节点看作是活节点。这是一种悲观的方法，在需要快速应对灾难性的故障时是一种合理的选择。它认为，在系统中，网络要比硬件和软件更可靠些，考虑到 Erlang 最初的应用场景，这种假设是合理的。还有一种乐观些的方法（认为节点仍然活着），这种方法会延后做出崩溃的判断，因为它认为硬件和软件比网络更可靠一些，所以会让集群等待更长的时间，让断开的节点重新连接进来。

这就引发了一个问题：在一个悲观的系统中，如果我们认为已经死亡的节点根本就没有死，突然又回来了，该怎么办呢？这个活着的死节点会让我们措手不及，这个节点有自己的运行状态，和集群中的其他节点在各个方面都格格不入——数据、连接等。这会引发一些非常令人讨厌的问题。

假设现在你有一个两节点系统，它们位于不同的数据中心。在该系统中，保存着用户的账户和现金数额信息，这些数据会在两个节点中完整保存。每次操作都会在节点间进行数据同步。当两个节点都正常时，用户可以不断取钱，直到账户中的数额为空。

软件一直工作良好，突然有个节点失去了和另外一个节点的连接。无法知道另外一个节点是活着还是死了。问题的关键是，这两个节点仍然能够从外部接受请求，但是却不能相互通信。

对于这种情况，有两种常见的策略：停止所有的操作以及不停止操作。选择第一种策略的风险在于，你的产品变得不可用了，会损失金钱。选择第二种策略的风险在于，某个账户中只有 1 000 美元的用户现在在两台服务器上可以各取出 1 000 美元来，也就是共 2 000 美元！无论选择哪种策略，如果不能正确处理的话，我们都要承担损失金钱的风险。

是否存在一种方法，能够让应用在网络分裂时仍保持可用且不会丢失服务器间的数据，从而完全避免上述问题呢？

26.4 CAP 定理

对于前面的问题，一个简短的回答是：不存在。很遗憾，在出现网络分裂时，不可能同时保证应用既可用且正确。

这正是著名的 CAP 定理所阐述的。在 CAP 定理中，首先声

明了所有分布式系统的 3 个核心属性：一致性（consistency）、可用性（availability）和分区容忍（partition tolerance）。

26.4.1　一致性

在前面的例子中，一致性指的是，无论系统由 2 个还是 1 000 个能够提供服务的节点组成，在某个给定的时刻，所有节点上看到的都是同样数量的现金余额。这通常会使用事务（在事务中，节点在更改数据库前，首先要协商一致，并且更改是原子的）或者其他等效的机制实现。

根据定义，一致性意味着，所有的操作看上去像是被作为单个、不可分割的原子块一起完成的，即使跨越了多个节点。一致性和时间无关，而是指不会出现针对同一块数据的两个不同操作，它们的操作方式会导致系统产生出多种结果。它能够保证你在修改一块数据时，不用担心同时发生且也在更改这块数据的其他操作影响你的操作结果。

26.4.2　可用性

可用性的含义是，如果你向系统请求一些数据，你能够得到回应。如果你得不到回答，那么系统对你来说就是不可用的。注意，像“对不起，我无法给你结果，因为我死了”这样的回答不算真正的回应，这只是一个糟糕的借口。这和完全不回答相比，没有提供任何更多有用的信息（学术界对此有点分歧）。

> **注意**　在 CAP 定理中，有一个重要考虑：只对那些没有死的节点，才会有可用性的问题。一个死亡节点无法发送回应，是因为它首先不能接收查询请求。这和那种由于所依赖的东西失效了因此节点不能发送回应的情况是不同的。如果节点不能接收请求、更改数据或者返回错误结果，从技术上讲，在正确性方面它不会对系统的平衡造成威胁。集群的其他部分只需处理更多的负载，直到死亡的节点重新恢复正常并同步好数据。

26.4.3　分区容忍

分区容忍是 CAP 定理中最复杂的部分。一般情况下，它意味着，即使系统的组成部件之间不能通信了，它还可以继续工作（且持有有用的信息）。分区容忍的要点在于，在系统组件之间可能会丢消息的情况下，系统仍然可以工作。这个定义有些抽象和开放，我们后面会知道原因。

简单来讲，CAP 定理规定：在任何分布式系统中，你只能同时得到两个 CAP 属性：一致性+可用性（CA）、一致性+分区容忍（CP）或者可用性+分区容忍（AP）。不能同时拥有它们。这既是一个好消息，也是一个坏消息。坏消息是，无法在网络分区的情况下同时保证一致性和可用性。好消息是，这是一个定理。如果客户要求你同时提供这三者，你就可以非常淡定地告诉客户，这根本不可能，而不用多费口舌（当然得和他们解释什么是 CAP 定理）。

在这 3 种选择中，通常可以被排除掉的是 CA。仅当你敢说网络绝对不会出问题，或者虽然会出问题，但是问题是原子性的（如果一个点出问题了，整个系统会同时出问题）时，才可以考虑这个选择。

除非有人发明了一种网络和硬件，它们根本不会出问题，或者如果某个出问题了，整个系统就会同时失效，否则网络分区就是必需的选择。这样 CAP 定理中就只剩下两种组合了：AP 或者 CP。

如果系统被网络分区分隔为多个部分，那么它要么保持可用，要么保持一致，但是不能同时满足。

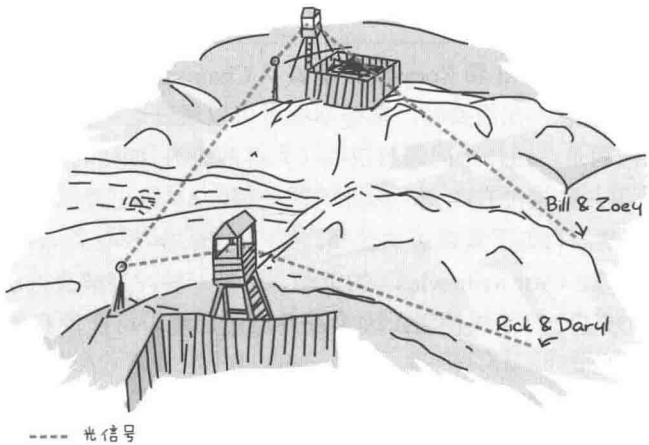
> **注意** 有些系统既不选择 A 也不选择 C。对于某些需要高性能的场合，像吞吐量（能够回答多少查询）或者延迟（能够多快回答问题）之类的要求会影响设计决策，CAP 定理并非一定要选择两个属性（CA、CP 或者 AP），还可以选择更少的属性。还要注意，有些系统可以在没有网络分区时具有完全的一致性，而在网络分区时放松对一致性的要求。

26.4.4 僵尸幸存者和 CAP

再次回到我们的幸存者这里，他们已经很久没有遇到僵尸了。那些肢体破损的僵尸们已经被抛在了身后。Bill、Zoey、Rick 和 Daryl 的电池最终都耗尽了，他们再也无法通信了。很幸运，他们找到了两个幸存者聚集区，其中都是些对僵尸存活特别感兴趣的计算机科学家和工程师。这些幸存者非常了解分布式编程，他们自制了一些协议，使用光线和镜子进行通信。

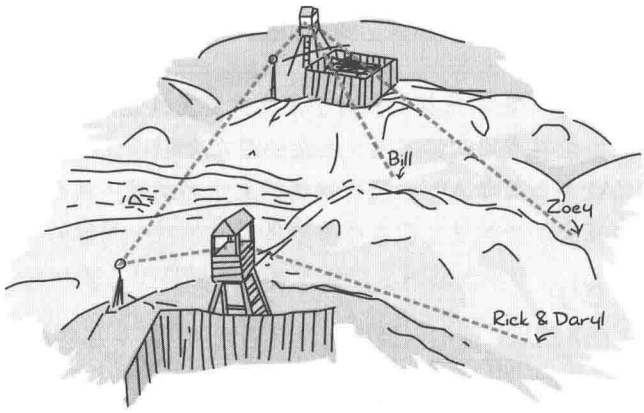
Bill 和 Zoey 找到了 Chainsaw 聚集区，Rick 和 Daryl 则找到了 Crossbow 营地。在营地中的其他人看来，我们的 4 个幸存者都是新来的，因此总是派他们到野外去寻找食物、杀死那些靠近营地边界的僵尸，而聚集区中的其他人则在争论着 Vim 和 Emacs 哪个更好（僵尸灾难以来唯一没有消失的争论）。

在我们 4 个幸存者来到聚集区的第 100 天，他们被派出去到位于两个营地中间的某个地点进行物品交换。出发前，Chainsaw 和 Crossbow 这两个聚集区商定了一个见面地点。如果在他们出发后想要更改之前约定的目的地或者时间，Rick 和 Daryl 可以给 Crossbow 发送一条消息，而 Zoey 和 Bill 则可以给 Chainsaw 发送一条消息。收到消息的聚集区都会把该信息传递给另外一个聚集区，这个聚集区会把信息再转发给本方的幸存者，如下图所示：



某个周日的一大早，我们的 4 个幸存者就出发了，他们需要步行走很远的路，约定的碰面时间是在周五早上破晓之前。一切进展顺利（除了沿途偶然会和一些小规模僵尸战斗）。

糟糕的事情发生了，周三，由于磅礴的大雨，并遭遇到大规模的僵尸，Bill 和 Zoey 走散了，他们迷路了，行程被耽误了。新的情况如下图所示：



更糟糕的是，大雨过后，通常会是晴天，这次却大雾弥漫，Chainsaw 的计算机科学家无法和 Crossbow 的人通信。Bill 和 Zoey 把他们遭遇到的问题发送给了他们的聚集区，并请求重新设定一个见面时间。如果没有雾，一切都好办，但是现在他们陷入了和网络分区同样的困境。

如果两个营地都选择 CP，那么 Zoey 和 Bill 就无法设置新的见面时间。通常，选择 CP 就意味着放弃对数据进行修改，从而让数据保持一致，此时，幸存者们还是可以向各自的营地请求数据。只是在更改时会被拒绝。这就保证了幸存者们不可能搞乱已经计划好的见面时间。无法联系上的其他幸存者则仍会按照约定时间见面。

如果两个营地都选择 AP，而不选择 CP，就允许幸存者更改见面时间。被分开的每一方都会有自己的见面时间数据。因此，如果 Bill 要求在周五晚上见面，那么大概的数据状态如下：

```
Chainsaw: Friday night
Crossbow: Friday before dawn
```

只要分区情况一直持续，Bill 和 Zoey 就只能够从 Chainsaw 处获取信息，而 Rick 和 Daryl 则只能从 Crossbow 处获取信息。在需要时，某些幸存者可以通过这种方式重新组队。

这里比较有趣的问题是：当分区问题解决后（大雾散去），如何对不同版本的数据进行处理。CP 对此的处理非常简单：数据没有改变，因此无需处理。AP 则更灵活一些，但是同时要解决的问题也会更多一些。通常，有如下策略可供选择。

- 使用最后写入者胜（last write wins）的策略，这是一种冲突解决方法，它会保留最后做出的更新。这种方法有时无法使用，因为在分布式环境中可能没有使用时间戳，或者有些更改是同时发生的。
- 随机选择一个胜利者。
- 使用更复杂的、基于时间的方法来辅助减少冲突的发生，如在最后写入者胜策略中使用逻辑时钟。逻辑时钟中不使用绝对时间值，而是使用单调递增的值（例如，每当文件被修改时会增加这个值）。如果想了解更多的细节，可以去学习关于 Lamport 时钟或者向量时钟（vector clock）的内容。
- 可以把冲突解决的职责推给应用层（在本例中，就是幸存者）。由接收到结果的那一端从

冲突的数据中选择出正确的即可。这和 Subversion、Mercurial 或者 Git 之类的源代码控制工具合并冲突的情况有点类似。

哪种策略更好呢？我讲解这些内容的方式可能会让你觉得：要么选择完全的 AP，要么选择完全的 CP，就像一个通断开关。其实，在现实世界中，我们可以有更多的选择，就像是一个群体仲裁（quorum）系统，其中不再是对/错类型的问题，而是有一个刻度盘，可以让我们选择希望的一致性程度。

群体仲裁系统基于一些简单的规则。假设系统中有 N 个节点，只有当其中的 M 个都同意时，才能去修改数据。如果系统的一致性要求比较低，那么可以只要求 15%的节点同意。这意味着，在发生网络分区时，分区中比较小的那一部分依然可以修改数据。如果要求更高的一致性，可以把比例设置为 75%，这意味着，要更改数据，系统中大部分的节点必须存在。在这种情况下，如果少数节点被孤立了，它们是没有权力更改数据的。不过，系统中仍然连接在一起的大部分节点还可以正常工作。

如果把需要的节点数量 M 的值设置成等于 N（节点总数），那么所得到的就是完全一致的系统。如果把 M 设置成 1，就是一个完全 AP 的系统，没有任何一致性保证。

此外，你可以针对单个请求来设置这些值。那些无关紧要的查询（谁刚刚登录了！）可以具有低一些的一致性要求，而和库存以及金钱相关的查询则需要更高的一致性要求。针对每种情况，把它和不同的冲突解决方法结合起来，可以构建出异常灵活的系统。

上述策略和各种不同的冲突解决方案结合在一起，就形成了分布式系统中的众多设计选项，不过它们实现起来相当复杂。虽然在使用时我们不用关心其中的细节，但是我认为，知道有哪些东西可用、还有哪些其他选择还是很重要的。

现在我们已经知道了使用 Erlang 进行分布式计算的一些问题，下面我们来介绍如何搭建一个基本的分布式系统。

26.5 搭建 Erlang 集群

除了分布式计算谬误中的那些问题，分布式 Erlang 中最困难的部分是正确地把系统搭建起来。跨不同的主机连接 Erlang 节点有点麻烦。为了避免这项工作，在进行实验时，我们通常会让多个节点跑在单台计算机上，这样会容易一点。

26.5.1 节点命名

我们在前面介绍过，为了能够定位和联系节点，Erlang 为每个节点都取了名字。名字的格式为 `Name@Host`，其中主机部分基于可用的 DNS 条目，这些条目或者在网络服务器中，或者在本机的 host 文件中（在 Mac OS X、Linux 以及其他类型的 Unix 系统中，该文件是/etc/hosts，对于大多数 Windows 系统，该文件是 C:\Windows\system32\drivers\etc\hosts）。为了避免冲突，名字必须唯一。如果启动节点时，使用的名字和主机名与某个已经启动的节点相同，就会得到一条非常恐怖的崩溃消息。

在启动节点前，需要再介绍一点关于节点名字的内容。名字有以下两种类型。

- 长名字（long name）：基于全称域名（aaa.bbb.ccc）。如果某个域名中含有句点（.），那么

许多 DNS 解析器会把它当作全称域名。

- 短名字（short name）：基于不带句点的主机名。使用本地 host 文件或者任何可能的 DNS 条目对其进行解析。

一般来讲，在单台计算机上使用短名字搭建一批 Erlang 节点要比长名字容易一些。

还请注意，短名字节点不能和长名字节点连通，反之亦然。

要使用短名字，可以这样启动 Erlang VM：`erl -sname short_name@domain`。长名字的启动方式为 `erl -name long_name@some.domain`。也可以只使用名字来启动节点：`erl-sname short_name` 或者 `erl -name long_name`。Erlang 会根据操作系统的配置自动增加主机名。还可以通过直接指定 IP 地址的方式启动节点，如 `erl -name name@127.0.0.1`。

> **注意**　Windows 用户最好使用 `werl` 而不是 `erl`。不过，为了启动一个分布式节点，并为其提供一个名字，要在命令行上启动节点，而不是单击某个快捷方式或者可执行文件。

26.5.2　连接节点

首先，启动两个节点：

```
$ erl -sname ketchup
... <snip> ...
(ketchup@ferdmbp)1>
```

```
$ erl -sname fries
... <snip> ...
(fries@ferdmbp)1>
```

要把节点 `fries` 和 `ketchup` 连接起来，请到第一个 shell 中，输入如下函数：

```
(ketchup@ferdmbp)1> net_kernel:connect(fries@ferdmbp).
true
```

调用函数 `net_kernel:connect(NodeName)` 会和另外一个 Erlang 节点建立连接。（有些教程中会使用 `net_adm:ping(Node)`，不过我觉得 `net_kernel:connect/1` 听起来更正式，让我更有信心！）如果看到这个函数调用返回 `true`，祝贺你，你现在已经进入了分布式 Erlang 模式。如果看到的是 `false`，你得花点功夫去解决你的网络问题。有一个简单的方法，编辑 host 文件，加入你想使用的任何主机。然后，再试一下。

调用 BIF 函数 `node()`，可以看到自己的节点名字，调用 BIF 函数 `nodes()`，可以看到所连接的节点：

```
(ketchup@ferdmbp)2> node().
ketchup@ferdmbp
(ketchup@ferdmbp)3> nodes().
[fries@ferdmbp]
```

为了演示节点之间的通信，我们要使用一个简单的技巧。把每个节点的 shell 进程都在本地注册为 shell：

```
(ketchup@ferdmbp)4> register(shell, self()).
true
```

```
(fries@ferdmbp)1> register(shell, self()).
true
```

现在，就能够通过名字{Name, Node}来给进程发送消息了。在两个 shell 上都试一下：

```
(ketchup@ferdmbp)5> {shell, fries@ferdmbp} ! {hello, from, self()}.
{hello,from,<0.52.0>}
```

```
(fries@ferdmbp)2> receive {hello, from, OtherShell} -> OtherShell ! <<"hey there!">>
end.
<<"hey there!">>
```

显然，消息收到了，我们可以向另外一个 shell 发送一条不同的消息：

```
(ketchup@ferdmbp)6> flush().
Shell got <<"hey there!">>
ok
```

可以看到，我们透明地发送元组、原子、pid 以及二进制数据，毫无问题。任何其他的 Erlang 数据结构也都可以通过消息发送。

就这么多。你已经知道如何使用分布式 Eralng 了！

26.5.3 更多工具

对于分布式 Erlang，还有一些其他有用的 BIF 函数。例如，erlang:monitor_node(NodeName, Bool)函数，如果某个进程调用它时使用 true 作为 Bool 参数的值，那么在节点死亡时，该进程会收到一条格式为{nodedown, NodeName}的消息。

> **注意** 除非在编写某个特定库时需要依赖于其他节点的死活，否则基本上不太会使用 erlang:monitor_node/2。这是因为，像 link/1 和 monitor/2 之类的函数也是可以跨节点工作的。不过，当跨节点的监控器和链接太多时，使用 erlang:monitor_node/2 函数还是有价值的。如果在这种情况中使用函数 link/1 和 monitor/2，就意味着，在某个节点死亡时，数以千计的监控器会被同时触发，而不是只有一条节点监控消息，之后可以本地转发这个信息。

假设我们在 fries 节点上进行了如下设置：

```
(fries@ferdmbp)3> process_flag(trap_exit, true).
false
(fries@ferdmbp)4> link(OtherShell).
true
(fries@ferdmbp)5> erlang:monitor(process, OtherShell).
#Ref<0.0.0.132>
```

接着，我们杀死 ketchup 节点。此时，节点 fries 的 shell 进程会收到一条'EXIT'消息和

一条监控消息：

```
(fries@ferdmbp)6> flush().
Shell got {'DOWN',#Ref<0.0.0.132>,process,<6349.52.0>,noconnection}
Shell got {'EXIT',<6349.52.0>,noconnection}
ok
```

以上就是消息内容。

> **注意**　除了通过杀死节点来中断和它的连接外，还可以使用 BIF 函数 erlang:disconnect_node(Node)来中断和节点的连接，无需关闭它。

嘿，等等！为什么 pid 是那个样子呢？我们看错了吗？

```
(fries@ferdmbp)7> OtherShell.
<6349.52.0>
```

什么？这个值不应该是<0.52.0>吗？不。pid 的这种显示方式只是真正的进程标识符的一种可视化表示形式。第一个数值表示节点（0 意味着进程来自当前节点），第二个数值是个计数器，第三个数值是另外一个计数器，用于进程数量太多，第一个计数器不够用的情况。pid 真实的底层表示更像下面这样：

```
(fries@ferdmbp)8> term_to_binary(OtherShell).
<<131,103,100,0,15,107,101,116,99,104,117,112,64,102,101,
  114,100,109,98,112,0,0,0,52,0,0,0,0,3>>
```

二进制序列<<107,101,116,99,104,117,112,64,102,101,114,100,109,98,112>>实际上是<<"ketchup@ferdmbp">>的 Latin-1（或者 ASCII）表示，也就是进程所在的节点的名字。接下来是两个计数器：<<0,0,0,52>> 和 <<0,0,0,0>>。最后一个值（3）是一个标记值，用来区分该 pid 来自何种节点，一个老节点、一个死亡的节点等。这正是 pid 可以透明地跨节点使用的原因。

> **注意**　如果不知道某个 pid 所在的节点，无需先把它转换成二进制形式，再从中读取节点名。调用函数 node(Pid)就行了，进程所在节点的名字会以原子类型返回。

还有其他一些有趣的 BIF 函数可供使用：spawn/2，spawn/4，spawn_link/2 以及 spawn_link/4。这些函数的用法和其他 spawn BIF 类似，只是它们可以在远程节点上创建进程。在 ketchup 节点上试试如下调用：

```
(ketchup@ferdmbp)6> spawn(fries@ferdmbp,
(ketchup@ferdmbp)6>       fun() -> io:format("I'm on ~p~n", [node()]) end).
I'm on fries@ferdmbp
<6448.50.0>
```

这在本质上是一个远程过程调用。无需更复杂的操作，我们就可以在其他节点上运行任何代码！有趣的是，函数在另外的节点上运行，但是输出却在本地。没错——即使是打印输出也可以被透明地跨网络重定向。这一切要归功于进程组领导者（group leader）机制。进程组领导者的继

承方式和进程位置无关，它们会不停地把 IO 操作转发给父进程，直到遇到正确的输出驱动，也就是最初调用它们的那个 shell。

这些就是使用 Erlang 编写分布式代码需要的全部工具了。现在，你已经拿到了砍刀、手电筒和胡须。而且,你所处的位置层次，对于那些不提供这种分布式层支持的语言来说，那可是需要花费九牛二虎之力才能企及的。所以，是时候去杀死怪兽了。或许，我们还是要先讨论一下 cookie 怪兽。

26.6 cookie

在本章的开始，我解释过 Erlang 节点之间是如何建立网状连接的。如果某个节点连上了一个节点，那么它会和其他所有节点建立连接。不过，你也许希望在同样的硬件上运行不同的 Erlang 集群。此时，你肯定不希望一不小心，两个节点就连接在一起。为了解决这个问题，Erlang 的设计者们在连接建立过程中增加了一个称为 cookie 的小令牌值。

在很多参考资料中，如 Erlang 的官方文档，都把 cookie 放到安全相关的主题下。不过，这只是一个玩笑，因为任何人都不会真的认为 cookie 是安全的。

cookie 是一个小且唯一的值，只有 cookie 值相同的节点才能相互连接在一起。cookie 的概念更像是用户名而非口令，我可以肯定地说，没人会把拥有用户名（再没其他东西）当成是一个安全特性。cookie 主要作为一种节点集群的分隔机制，而非一种认证机制。

要给节点设定一个 cookie，只需在启动它的命令行上增加一个`-setcookie cookie`参数。我们启动两个新节点试试：

```
$ erl -sname salad -setcookie 'myvoiceismypassword'
... <snip> ...
(salad@ferdmbp)1>
```

```
$ erl -sname mustard -setcookie 'opensesame'
... <snip> ...
(mustard@ferdmbp)1>
```

现在，这两个节点具有不同的 cookie，因此它们之间不能通信：

```
(salad@ferdmbp)1> net_kernel:connect(mustard@ferdmbp).
false
```

连接请求被拒绝了，但是并没有给出拒绝的原因。我们来看看 mustard 节点：

```
=ERROR REPORT==== 10-Dec-2013::13:39:27 ===
** Connection attempt from disallowed node salad@ferdmbp **
```

很好。现在，如果想让 salad 节点和 mustard 节点能够连接起来该怎么办呢？有一个名为 `erlang:set_cookie/2` 的 BIF 函数正好能够满足我们的需要。如果调用了函数 `erlang:set_cookie (OtherNode, Cookie)`，那么 7 仅当和那个节点连接时才会使用那个 cookie。如果 `erlang:set_cookie(node(), Cookie)` 这样调用，那么就等于更改了节点当前的

cookie 值，以后和其他节点建立连接时都使用这个新 cookie 值。要看到更改后的值，可以使用函数 `erlang:get_cookie()`：

```
(salad@ferdmbp)2> erlang:get_cookie().
myvoiceismypassword
(salad@ferdmbp)3> erlang:set_cookie(mustard@ferdmbp, opensesame).
true
(salad@ferdmbp)4> erlang:get_cookie().
myvoiceismypassword
(salad@ferdmbp)5> net_kernel:connect(mustard@ferdmbp).
true
(salad@ferdmbp)6> erlang:set_cookie(node(), now_it_changes).
true
(salad@ferdmbp)7> erlang:get_cookie().
now_it_changes
```

我们再介绍最后一种设置 cookie 的方法。如果你运行了本章前面的例子，那么去看看你的 home 目录。你应该看到该目录中有一个名为.erlang.cookie 的文件。如果打开这个文件，你会看到一串类似 PMIYERCHJZNZGSRJPVRK 的随机字符串。在启动分布式节点时，如果没有指定 cookie，那么 Erlang 就会创建一个 cookie，并放置在.erlang.cookie 文件中。于是，以后每次启动节点，但不指定 cookie 时，VM 就会从 home 目录中查找这个文件，并以其中的字符串作为 cookie。

26.7　远程 shell

我们在学习 Erlang 时，一开始就给大家介绍了如何通过 `ctrl-G`（`^G`）来中断正在运行的代码。其中，我们会看到一个关于分布式 shell 的菜单项：

```
(salad@ferdmbp)1>
User switch command
--> h
c [nn]                  - connect to job
i [nn]                  - interrupt job
k [nn]                  - kill job
j                       - list all jobs
s [shell]               - start local shell
r [node [shell]]        - start remote shell
q        - quit erlang
? | h                   - this message
```

其中，`r[node [shell]]`就是我们要寻找的用于远程 shell 的选项。我们可以使用如下方式在 `mustard` 节点启动一个作业：

```
--> r mustard@ferdmbp
--> j
  1 {shell,start,[init]}
  2* {mustard@ferdmbp,shell,start,[]}
--> c
Eshell V5.9.1 (abort with ^G)
(mustard@ferdmbp)1> node().
mustard@ferdmbp
```

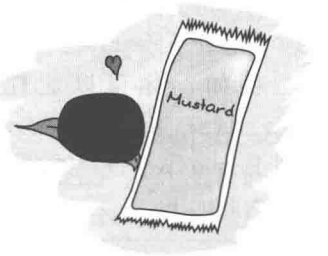
启动完成了。现在，你可以和使用本地 shell 完全一样的方式使用远程 shell 了。在老一些的 Erlang 版本中，情况稍有不同，例如没有自动补齐功能。另外，当你想去更改某个以-noshell 选项启动的节点上的东西时，远程 shell 非常有用。如果以-noshell 启动的那个节点具有名字，那么就可以连接到它，做一些 DevOps 之类的工作。例如，重新加载模块，调试一些代码，等等。

再次按下 ctrl-G，就可以返回到原来的节点。不过，在停止会话时要非常小心。如果调用了 q()或者 init:stop()，那么将会关闭远程节点！

26.8 隐藏节点

可以通过调用函数 net_kernel:connect/1 连接 Erlang 节点，但是，需要注意的是，任何形式的节点间交互，几乎都会导致它们之间建立连接关系。调用 spawn/2 函数或者向某个外部的 pid 发送消息都会自动建立节点间的连接。

假设现在有一个正常工作的集群，你只想连上其中的一个节点去更改些东西，但这个连接造成的后果实在有些讨厌。你不希望你的管理节点突然就集成到了整个集群中，从而让其他节点认为集群中新增了一个可以承担任务的工作节点。如果只想连接到一个远程节点，而不希望自动连接到和该远程节点相连的所有其他节点，可以使用一个不常用的函数：erlang:send(Dest, Message, [noconnect])，该函数只发送消息，不建立连接，不过这很容易导致错误。



因此，你可能更希望使用-hidden 选项来启动节点。

假设你有两个正在运行的节点：mustard 和 salad。我们启动第三个节点：olives，并且只想让它连接到 mustard（cookie 要一致！）：

```
$ erl -sname olives -hidden
... <snip> ...
(olives@ferdmbp)1> net_kernel:connect(mustard@ferdmbp).
true
(olives@ferdmbp)2> nodes().
[]
(olives@ferdmbp)3> nodes(hidden).
[mustard@ferdmbp]
```

啊哈！该节点没有和 salad 建立连接，咋一看，它好像也没有和 mustard 建立连接。不过，调用 nodes(hidden)则显示，它和 mustard 之间建立了连接！我们来看看 mustard 节点上的情况：

```
(mustard@ferdmbp)1> nodes().
[salad@ferdmbp]
(mustard@ferdmbp)2> nodes(hidden).
[olives@ferdmbp]
(mustard@ferdmbp)3> nodes(connected).
[salad@ferdmbp,olives@ferdmbp]
```

mustard 节点上也给出了类似的结果，不过我们多了一次 nodes(connected) BIF 调用，

这个调用显示出了所有类型的连接。节点 salad 不会和 olives 建立连接，除非专门去建立这个连接。

nodes/1 还有另外一个有趣的用法：nodes(known)，这个调用会显示出当前节点曾经连接过的所有节点。

有了远程 shell、cookie 和隐藏节点这些工具，管理一个分布式 Erlang 系统会变得简单一些。

26.9　防火墙问题

在使用分布式 Erlang 时，如果需要穿越防火墙（不想使用隧道），就得开放一些供 Erlang 通信使用的端口。端口 4369 是需要被开放出来的，它是 EPMD 的默认端口（Erlang 端口映射器守护应用，本章前面做过介绍）。最好使用这个端口号，因为爱立信公司对此做过正式的注册。这意味着，任何遵从标准的操作系统都会保留该端口，供 EPMD 使用。

接下来，需要开放一组端口供节点之间的连接使用。问题是，对于节点之间的连接，Erlang 采用的是随机分配端口的策略。不过，可以使用两个隐藏的应用变量指定可分配端口的范围。这两个变量位于 kernel 应用中，是 inet_dist_listen_min 和 inet_dist_listen_max。

例如，你可以这样启动 Erlang：erl -name left_4_distribudead -kernel inet_dist_listen_min 9100 -kernel inet_dist_listen_max 9115，它设置了 15 个供 Erlang 节点使用的端口。你还可以使用一个名为 ports.config 的配置文件来指定这些端口，文件的内容如下：

```
[{kernel,[
  {inet_dist_listen_min, 9100},
  {inet_dist_listen_max, 9115}
]}].
```

接着，可以这样启动 Erlang 节点：erl -name the_army_of_darknodes -config ports。变量会以同样的方式设置。注意，这些端口都是监听端口，因此，每台机器上的每个节点都要占用一个。如果你在一台服务器或者计算机上运行两个 VM，那么需要两个监听端口。

26.10　高级调用

除了到目前为止我们介绍的所有 BIF 和选项之外，在开发分布式系统时，还有一些模块能为开发者带来帮助。

26.10.1　net_kernel 模块

之前，我们在建立和中断节点之间的连接时使用过这个模块。

该模块还包含其他复杂的功能，例如把一个非分布式节点变成一个分布式节点：

```
$ erl
... <snip> ...
1> net_kernel:start([romero, shortnames]).
{ok,<0.43.0>}
(romero@ferdmbp)2>
```

你可以使用 shortnames 或者 longnames，它们分别对应着-sname 和-name。此外，如果你知道某个节点将要发送一些大消息，因此可能需要长一些的节点间心跳时间，可以向列表中再传递一个参数：net_kernel:start([Name, Type, HeartbeatInMilliseconds])。默认情况下，心跳延时（又称为 tick 时间）被设置成 15 s，或者 15 000 ms。

模块中还有两个函数值得介绍，一个是 net_kernel:set_net_ticktime(S)，该函数用来改变节点的 tick 时间，以避免连接中断。另外一个函数是 net_kernel:stop()，这个函数把节点从分布式模式切换回普通模式：

```
(romero@ferdmbp)2> net_kernel:set_net_ticktime(5).
change_initiated
(romero@ferdmbp)3> net_kernel:stop().
ok
4>
```

26.10.2 global 模块

另外一个可以用于分布式开发的模块是 global。global 模块提供了另外一种进程注册库。它会自动把数据传播到所有相连的节点、自动复制数据并处理节点故障，并且支持多种节点恢复时的冲突解决策略。

你可以调用 global:register_name(Name, Pid) 注册一个名字，调用 global:unregister_name(Name) 解除名字注册。如果想做一个原子性的名字转换，可以调用 global:re_register_name(Name, Pid)。可以调用 global:whereis_name(Name) 得到 pid，调用 global:send(Name, Message) 向进程发送消息。该模块包括了所有你需要的东西。更令人愉快的是，在注册时，可以把任意数据项当作进程名字使用。

如果有两个节点，它们上面都运行着一个具有相同名字的进程，那么当这两个节点突然建立连接关系时，就会出现名字冲突。在这种情况下，global 默认会随机杀死其中的一个。有多种取代这个默认行为的方法。当注册或者重新注册一个名字时，可以多传入一个参数，如下：

```
5> Resolve = fun(_Name,Pid1,Pid2) ->
5>    case process_info(Pid1, message_queue_len) > process_info(Pid2, message_queue_
len) of
5>       true -> Pid1;
5>       false -> Pid2
5>    end
5> end.
#Fun<erl_eval.18.59269574>
6> global:register_name({zombie, 12}, self(), Resolve).
yes
```

Resolve 函数会选择保留邮箱中消息数量更多的进程（该函数所返回的就是这个进程的 pid）。你也可以实现其他一些策略，例如，保留订阅者最多的进程，或者保留最先回应的进程等。如果 Resolve 函数崩溃了，或者返回了一些不是 pid 的东西，就会解除该进程名的注册。为了便于使用，global 模块预定义了以下 3 个函数。

（1）fun global:random_exit_name/3 会随机杀死一个进程。这个是默认选项。

（2）fun global:random_notify_name/3 会随机地从两个 pid 中挑选一个保留，并给落选的进程发送一条{global_name_conflict, Name}消息。

（3）fun global:notify_all_name/3 会同时解除两个进程的注册，并给这两个进程都发送一条{global_name_conflict, Name, OtherPid}消息。把解决问题的责任交给这两个进程，之后它们可以再次注册。

global 模块有一个缺点：节点故障和名字冲突的检测通常需要花费较长时间。所以，对于那些注册规模不大，并且不太会经常变化的情况，global 模块比较适用。除此之外，global 还是一个很有用的模块，甚至有些 OTP 行为中也提供了对它的支持。只需把所有 gen_Something:start_link(...)调用中使用的本地名字（{local, Name}）更改为{global, Name}，然后在所有 call 和 cast（以及和它们等价的函数）调用中都使用{global, Name}。做完这些以后，就变成分布式的了。

> **注意**　在 R15B01 以后的 Erlang 版本中，可以使用除 local 和 global 之外的其他注册方式。可以利用{via, RegistryModule, Name}这种形式的名字使用任何自定义的进程注册库，只要接口兼容就行。

26.10.3　rpc 模块

要介绍的下一个模块是 rpc（远程过程调用）。它含有可以在远程节点执行命令的函数，以及一些支持并行操作的函数。为了测试一下，我们启动两个不同的节点，并把它们连接起来，本章前面介绍过如何完成这项工作。这两个节点的名字分别为 cthulu 和 lovecraft。

rpc 模块中最基本的操作是 rpc:call/4-5。通过它可以在远程节点上运行一个给定的操作，并把结果返回到本地：

```
(cthulu@ferdmbp)1> rpc:call(lovecraft@ferdmbp, lists, sort, [[a,e,f,t,h,s,a]]).
[a,a,e,f,h,s,t]
(cthulu@ferdmbp)2> rpc:call(lovecraft@ferdmbp, timer, sleep, [10000], 500).
{badrpc,timeout}
```

从 cthulu 节点上发起的这个调用中可以看出，该函数有 4 个参数，形式为 rpc:call(Node, Module, Function, Args)。还可以增加一个超时参数。rpc 调用会返回它所运行函数的返回值，出错时会返回{badrpc, Reason}。

如果你之前曾经学习过分布式或者并行计算方面的知识，可能听说过 promise 或者 future。promise 和 future 与远程过程调用有点像，不过它们是异步的。使用 rpc 模块，我们可以这么做：

```
(cthulu@ferdmbp)3> Key = rpc:async_call(lovecraft@ferdmbp, erlang, node, []).
<0.45.0>
(cthulu@ferdmbp)4> rpc:yield(Key).
lovecraft@ferdmbp
```

把函数 rpc:async_call/4 和函数 rpc:yield(Res)联合起来使用，我们可以进行异步的分布式调用，并在以后提取结果。如果你知道所发起的远程过程调用要执行一段时间才能返回，那么这种方法就非常适用。在这种场景中，你发起一个调用，在等待结果期间可以去处理其他任务（发起其他调用、从数据库提出记录、喝杯茶等），等到实在无事可做时，再去等待结果。当然，如果需要，也可以在自己的节点上执行这种调用：

```
(cthulu@ferdmbp)5> MinTime = rpc:async_call(node(), timer, sleep, [30000]).
<0.48.0>
(cthulu@ferdmbp)6> lists:sort([a,c,b]).
[a,b,c]
(cthulu@ferdmbp)7> rpc:yield(MinTime).
... <long wait> ...
ok
```

如果在使用 yield/1 函数时，想使用超时功能，就可以调用 rpc:nb_yield(Key, Timeout)。如果只想轮询一下结果，就可以使用 rpc:nb_yield(Key)（和 rpc:nb_yield(Key,0)等价）：

```
(cthulu@ferdmbp)8> Key2 = rpc:async_call(node(), timer, sleep, [30000]).
<0.52.0>
(cthulu@ferdmbp)9> rpc:nb_yield(Key2).
timeout
(cthulu@ferdmbp)10> rpc:nb_yield(Key2).
timeout
(cthulu@ferdmbp)11> rpc:nb_yield(Key2).
timeout
(cthulu@ferdmbp)12> rpc:nb_yield(Key2, 1000).
timeout
(cthulu@ferdmbp)13> rpc:nb_yield(Key2, 100000).
... <long wait> ...
{value,ok}
```

如果对结果不感兴趣，可以使用 rpc:cast(Node, Mod, Fun, Args)向另外一个节点发送命令，也不用等待结果。

但是，如果想同时向多个节点发起调用该怎么办呢？向我们的小集群中再增加 3 个节点：minion1, minion2 和 minion3。这 3 个节点是节点 cthulu 的从节点。当我们想向这 3 个节点发起请求并期待它们回答时，需要进行 3 次不同的 call 调用，而要向它们下发命令时，则要发起 3 次 cast 调用。真是糟糕透了，难以扩展到大量从节点的情况。

解决方法是使用另外两个用于 call 和 cast 的 rpc 函数，分别为 rpc:multicall(Nodes, Mod, Fun, Args)（还有一个可选的 Timeout 参数）和 rpc:eval_everywhere(Nodes, Mod, Fun, Args)：

```
(cthulu@ferdmbp)14> nodes().
[lovecraft@ferdmbp, minion1@ferdmbp, minion2@ferdmbp, minion3@ferdmbp]
(cthulu@ferdmbp)15> rpc:multicall(nodes(), erlang, is_alive, []).
{[true,true,true,true],[]}
```

这个调用结果显示，4 个节点全都活着（返回值中没有不可用的节点）。返回值元组的第一个元素表示活着节点的返回值，第二个元素是不可用的节点。不过，由于 `erlang:is_alive()` 的返回值标识了运行该调用的节点是否活着，因此看起来有些奇怪。再强调一下，在分布式系统中，活着只意味着“可以联系上”，并非“运行良好”。再看一个例子，假设 `cthulu` 对其从节点并不满意，决定杀死它们，更准确地说，让它们自杀。这是一个命令，因此是一个 `cast` 调用。所以，我们使用 `eval_everywhere/4` 在从节点上调用 `init:stop()`：

```
(cthulu@ferdmbp)16> rpc:eval_everywhere([minion1@ferdmbp, minion2@ferdmbp,
minion3@ferdmbp], init, stop, []).
abcast
(cthulu@ferdmbp)17> rpc:multicall([lovecraft@ferdmbp, minion1@ferdmbp,
minion2@ferdmbp, minion3@ferdmbp], erlang, is_alive, []).
{[true],[minion1@ferdmbp, minion2@ferdmbp, minion3@ferdmbp]}
```

当我们再次询问哪些节点还活着时，只剩下一个节点：`lovecraft` 节点。从节点们都非常听从命令。

`rpc` 模块中还有其他一些有趣的函数，不过其核心功能在本章中都被覆盖到了。如果想了解更多的内容，我建议你去仔细阅读该模块的文档。

26.11　小结

我们基本上介绍了分布式 Eralng 的所有基础知识。有太多的东西需要我们思考和牢记。当你开发一个分布式应用时，问一问自己可能会面临哪些分布式计算中的谬误（如果有的话）。如果有客户要求你构建一个在网络分区时能够保证一致性和可用性的系统，你应该知道，要么平静地给他解释 CAP 定理，要么赶快逃跑（想取得夸张点的效果，就跳窗而逃）。

一般来讲，如果应用由 1 000 个孤立的节点组成，这些节点之间没有依赖，无需通信就能完成自己的工作，那么这种应用的伸缩性就是最好的。无论使用什么样的分布式支持层，节点之间的依赖越多，系统就变得越难伸缩。这和僵尸很像。（真的!）僵尸之所以恐怖，是因为它们数量众多，且很难从整体上把其杀死。虽然单个僵尸非常缓慢，也不会造成大的威胁，但是成群的僵尸所带来的破坏却是巨大的，即使损失了大量僵尸成员，也无甚大碍。人类幸存者团队则需要通过共同谋划、相互沟通来完成一些伟大的任务，而且每个个体的损失对团队以及生存能力造成的伤害也要更大一些。

尽管如此，你还是掌握了前行所需要的工具。在第 27 章中，我们会介绍分布式 OTP 应用。这种应用提供了应对硬件故障的接管和故障转移机制，它并不适用于一般的分布式场景。它更像是让已经死亡的僵尸复活。

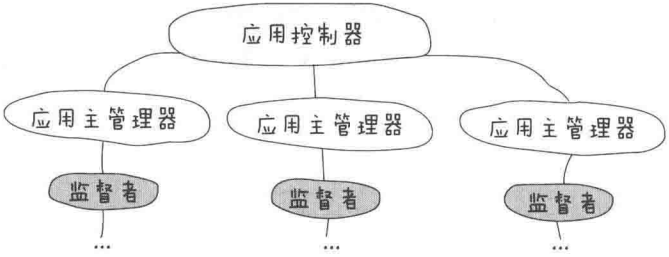
第 27 章

分布式 OTP 应用

尽管在用 Erlang 构建分布式系统时，有不少工作要做，不过 Erlang 还是为我们提供了一些解决方案。其中之一就是分布式 OTP 应用。分布式 OTP 应用，或者在 OTP 上下文中，简称为分布式应用，为我们提供了接管（takeover）和故障切换（failover）机制。在本章中，我们将介绍这两种机制的定义以及工作原理，并会编写一个小型应用来演示这两个概念。

27.1 更多 OTP 内容

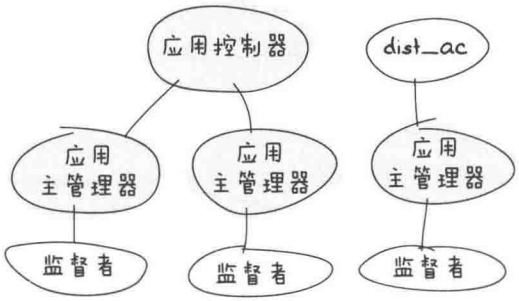
在第 19 章中，我们简单介绍了应用的结构：有一个中心应用控制器，它管理着应用主管理器，每个应用主管理器都负责监控着应用的顶级监督者，如下图所示。



在标准 OTP 应用中，应用可以被加载、启动、停止和卸载。对于分布式应用，工作方式有些不同。分布式应用控制器，一个和应用控制器层级相同的进程（通常称为 `dist_ac`），分担了应用控制器的工作，如下图所示。

根据应用文件的配置，应用的所有权会发生变化。每个节点上都会启动一个 `dist_ac` 进程，所有 `dist_ac` 进程之间会互相通信。它们之间的通信内容和分布式应用不太相关，有一点除外。标准的应用有 4 种状态：已加载、已启动、已停止和已卸载。分布式应用把已启动状态再分为两个状态：*已启动*和*正在运行*。有了这种区分，就可以把应用定义成集群内全局的。这种全局应用同时只能运行在一个节点上，而常规的 OTP 应用则不关心其他节点上的事情。因此，分布式应用

会在集群中的所有节点上启动，但是只会在其中一个节点上运行。



对于那些应用已经在其上启动，但是没有运行的节点来说，这意味着什么呢？它们唯一要做的事情是：等待运行应用的节点死亡。也就是说，当运行应用的节点死亡时，另外的节点取代它继续运行应用。这种方法可以让应用在不同子系统间搬迁，从而避免业务中断。

27.2　接管和故障切换

在分布式应用中，有两个重要的概念：故障切换和接管。

故障切换的意思是，如果应用在某台机器上停止运行了，就在其他机器上重新启动应用。在有冗余硬件的情况下，这种策略非常有效。你可以在一台主计算机或者服务器上运行某些应用，如果它出现了故障，就把它搬移到备用计算机上。在较大规模的部署中，你也许会用 50 台服务器来运行你的软件（所有机器的负载大概都在 60%～70%），并期望正常工作的服务器可以承担出现故障的服务器上的任务。故障切换的概念对于前一种情况非常有用，但是不太适合于后一种情况。

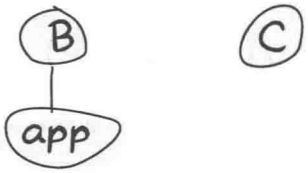
分布式 OTP 应用的第二个重要概念是接管。接管的意思是，某个死亡的节点重新正常工作，由于它比备份节点更好一些（如有更好的硬件），于是决定在其上重新运行应用。通常的做法是，优雅地停掉备用机上运行的应用，在主用机上重新启动它。

> **注意**　根据分布式计算的谬误中的论述，分布式 OTP 应用认为，当出现故障时，更可能是硬件方面的故障，而不是网络分区。如果你认为网络分区比硬件故障更可能发生，那么必须要知道，当网络问题修复后，可能会出现应用在主节点和备份节点上面同时运行的奇怪情形。如果会出现这种情况，那么分布式 OTP 应用也许不是合适的选择。

假设有一个系统，由 3 个节点组成，其中只有一个节点运行着某个给定应用。

节点 B 和 C 被定义为节点 A 死亡时的备份节点，现在，假设节点 A 死亡了。

有一个短暂的时段，什么也没有运行。过了一段时间后，节点 B 意识到了这个问题，决定接管应用。



这就是故障切换。接着，B 死亡了，应用在节点 C 上重新启动。

又发生了一次故障切换，一切安好。

现在，假设节点 A 又活了过来。节点 C 正愉快地运行着应用，但是节点 A 才是我们定义的主节点。此时就会发生接管行为。应用会在节点 C 上正常地停止，并在节点 A 上重新启动。

之后再有故障发生时，照此进行。

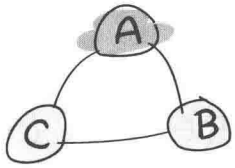
这个过程中有一个明显的问题，像这样不停地终止应用很可能会导致重要状态的丢失。遗憾的是，这是你要解决的问题。你得负责考虑所有重要状态的存储和共享，避免问题的发生。对于分布式应用，OTP 并没有提供针对这种情况的特殊处理。

了解了这些概念，下面我们来看一个实际例子。

27.3 神奇 8 号球

神奇 8 号球是一个简单的玩具，不管什么问题，只要摇一摇，就可以从神那里得到有用的答案。你可以问这样的问题“我最喜欢的运动队今晚会赢得比赛吗？”你摇一摇球，它就会给出这样的回答“毫无疑问”。接下来，你就可以放心地下注了。你还可以问其他问题，如“我应该为将来做些谨慎的投资吗？”它会回答“不必”或者“不确定。”在过去的几十年间，神奇 8 号球在西方国家的政治决策中起着至关重要的作用，因此，我们很自然地选择它作为容错的例子。

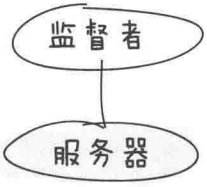
在实现中，我们不会使用真实世界的切换机制来自动找到服务器，如循环式 DNS 服务器或者负载均衡器。我们只使用纯 Erlang，构建一个具有 3 个节点（分别用 A、B 和 C 表示）的分布式 OTP 应用。节点 A 是运行神奇 8 号球服务的主节点，节点 B 和 C 为备份节点：



当节点 A 失效时，会在节点 B 或者 C 上重新启动神奇 8 号球应用，并且在这两个节点上仍然可以透明地使用它。

27.3.1　构建应用

在初始化分布式 OTP 应用之前，我们先来构建应用本身。应用的设计简单得令人难以置信：



该应用一共有 3 个模块：监督者、服务器和启动需要的应用回调模块。

1. 监督者模块

监督者模块非常简单。我们称之为 m8ball_sup（表示神奇 8 号球监督者），把它放到标准 OTP 应用的 src/目录中：

```
-module(m8ball_sup).
-behaviour(supervisor).
-export([start_link/0, init/1]).

start_link() ->
    supervisor:start_link({global,?MODULE}, ?MODULE, []).

init([]) ->
    {ok, {{one_for_one, 1, 10},
          [{m8ball,
           {m8ball_server, start_link, []},
           permanent,
           5000,
           worker,
           [m8ball_server]
          }]}}.
```

这个监督者会启动服务器（m8ball_server），一个永久的工作进程，每 10 s 只允许失败一次。

2. 服务器模块

神奇 8 号球服务器稍微有些复杂。它是一个具有如下接口的 gen_server：

```
-module(m8ball_server).
-behaviour(gen_server).
-export([start_link/0, stop/0, ask/1]).
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         code_change/3, terminate/2]).

%%%%%%%%%%%%%%%%%%
%%% INTERFACE %%%
%%%%%%%%%%%%%%%%%%
start_link() ->
    gen_server:start_link({global, ?MODULE}, ?MODULE, [], []).

stop() ->
    gen_server:call({global, ?MODULE}, stop).

ask(_Question) -> % 问题本身是什么不重要!
    gen_server:call({global, ?MODULE}, question).
```

注意，在启动服务器时，我们使用的名字是{global, ?MODULE}，并且在每个调用中，都通过同样的名字元组访问服务器。这种名字形式可以让 OTP 行为使用 global 模块，我们在第 26 章中见过这个模块。

接下来是回调函数，也是真正的实现逻辑。神奇 8 号球会随机地从众多可选答案中挑选一个，这些答案来自某个配置文件。之所以使用配置文件，是因为可以方便地根据需要增删答案。

首先，如果我们想使用随机功能，就需要在 init 函数中设置随机种子：

```
%%%%%%%%%%%%%%%%%%
%%% CALLBACKS %%%
%%%%%%%%%%%%%%%%%%
init([]) ->
    <<A:32, B:32, C:32>> = crypto:rand_bytes(12),
    random:seed(A,B,C),
    {ok, []}.
```

在第 23 章，我们也使用过这种方式。在此，我们使用 12 个随机字节初始化随机种子，这个种子会被后面的 random:uniform/1 函数使用。

下一步是从配置文件中读取答案，并从中挑选一个。我们在第 19 章讲过，最简单的配置方案是使用 app 文件（在 env 元组中）。下面是我们的做法：

```
handle_call(question, _From, State) ->
    {ok, Answers} = application:get_env(m8ball, answers),
    Answer = element(random:uniform(tuple_size(Answers)), Answers),
    {reply, Answer, State};
handle_call(stop, _From, State) ->
    {stop, normal, ok, State};
handle_call(_Call, _From, State) ->
    {noreply, State}.
```

第一个子句展示了我们想做的事情。我们期望env元组中有一个名为answers的值，它是一个包含了所有可能答案的元组。为什么选择元组呢？只是因为访问元组中元素的时间复杂度是常量，而对列表的访问则是线性的（因此列表越长，花费时间也越长）。这个函数子句最后返回了答案。

> **注意**　每次提问时，服务器都会调用application:get_env(m8ball, answers)读取答案。如果你使用调用application:set_env(m8ball, answers, {"yes","no","maybe"})来设置新的答案，那么这3个新答案会立即生效。长期来看，在启动时只读取一次答案的做法要更高效一些，但是这样的话，只能通过重启应用或者增加一个专门函数的方法来更新答案。

你应该已经注意到，我们并不关心所问的问题是什么，甚至根本就没有把问题传递给服务器。因为我们是随机返回答案，所以在进程之间复制问题完全没有意义。我们干脆忽略掉它以节省工作量。但我们仍然保留了这个参数，只是为了让最终接口看起来更自然些。如果愿意的话，我们也可以做些修改，让神奇8号球对于同样的问题一直返回同样的答案，但在本例中，我们不打算做这件事。

模块的其余部分和一般的gen_server模块中的一样，没做什么实际工作：

```
handle_cast(_Cast, State) ->
    {noreply, State}.

handle_info(_Info, State) ->
    {noreply, State}.

code_change(_OldVsn, State, _Extra) ->
    {ok, State}.

terminate(_Reason, _State) ->
    ok.
```

现在，我们可以去做更重要的工作了，也就是应用文件和回调模块。先来编写回调模块m8ball.erl：

```
-module(m8ball).
-behaviour(application).
-export([start/2, stop/1]).
-export([ask/1]).

%%%%%%%%%%%%%%%%%
%%% CALLBACKS %%%
%%%%%%%%%%%%%%%%%
start(normal, []) ->
    m8ball_sup:start_link().

stop(_State) ->
```

```
    ok.

%%%%%%%%%%%%%%%%%
%%% INTERFACE %%%
%%%%%%%%%%%%%%%%%
ask(Question) ->
    m8ball_server:ask(Question).
```

很简单。下面是 app 文件 m8ball.app：

```
{application, m8ball,
[{vsn, "1.0.0"},
 {description, "Answer vital questions"},
 {modules, [m8ball, m8ball_sup, m8ball_server]},
 {applications, [stdlib, kernel, crypto]},
 {registered, [m8ball, m8ball_sup, m8ball_server]},
  {mod, {m8ball, []}},
  {env, [
    {answers, {<<"Yes">>, <<"No">>, <<"Doubtful">>,
               <<"I don't like your tone">>, <<"Of course">>,
               <<"Of course not">>, <<"*backs away slowly and runs away*">>}}
 ]}
]}.
```

和所有 OTP 应用一样，这个应用也依赖于 stdlib 和 kernel 应用，由于服务器使用了随机种子，因此它还要依赖于 crypto 应用。注意，所有答案都放在一个元组中，和服务器的要求相匹配。在本例中，答案都是二进制类型，不过字符串的格式并不重要——使用列表也完全可以。

27.3.2 变身分布式应用

到目前为止，所有的工作看起来都是在构建一个非常完美的普通 OTP 应用。只需对文件做很少的修改，就可以把这个普通的 OTP 应用变成一个分布式 OTP 应用。事实上，只要增加一个函数子句，回到 m8ball.erl 模块：

```
%%%%%%%%%%%%%%%%%
%%% CALLBACKS %%%
%%%%%%%%%%%%%%%%%

start(normal, []) ->
    m8ball_sup:start_link();
start({takeover, _OtherNode}, []) ->
    m8ball_sup:start_link().
```

当一个更重要的节点接管一个备份节点时，就会把{takeover, OtherNode}作为参数传递给 start/2 函数。在神奇 8 号球应用中，无需更改任何东西，因此直接以同样的方式启动监督者就行了。

重新编译代码，马上就要完成了。

不过，等等，如何指定哪些节点是主节点，哪些是备份节点呢？答案是：在配置文件中指定。因为我们的系统有 3 个节点（a@yourhost、b@yourhost 和 c@yourhost），所以需要 3 个配

置文件（名字分别为 a.config、b.config 和 c.config，把它们都放到应用目录中的 config/ 目录下）：

```
[{kernel,
  [{distributed, [{m8ball,
                   5000,
                   [a@ferdmbp, {b@ferdmbp, c@ferdmbp}]}]},
   {sync_nodes_mandatory, [b@ferdmbp, c@ferdmbp]},
   {sync_nodes_timeout, 30000}
  ]}].
```

```
[{kernel,
  [{distributed, [{m8ball,
                   5000,
                   [a@ferdmbp, {b@ferdmbp, c@ferdmbp}]}]},
   {sync_nodes_mandatory, [a@ferdmbp, c@ferdmbp]},
   {sync_nodes_timeout, 30000}
  ]}].
```

```
[{kernel,
  [{distributed, [{m8ball,
                   5000,
                   [a@ferdmbp, {b@ferdmbp, c@ferdmbp}]}]},
   {sync_nodes_mandatory, [a@ferdmbp, b@ferdmbp]},
   {sync_nodes_timeout, 30000}
  ]}].
```

这 3 个文件的总体结构完全一样：

```
[{kernel,
  [{distributed, [{AppName,
                   TimeOutBeforeRestart,
                   NodeList}]},
   {sync_nodes_mandatory, NecessaryNodes},
   {sync_nodes_optional, OptionalNodes},
   {sync_nodes_timeout, MaxTime}
  ]}].
```

NodeList 值的通常形式为[A, B, C, D]，其中 A 为主节点，B 为第一个备份节点，C 为下一个备份节点，D 为最后一个备份节点。还有另外一种语法形式：[A, {B, C}, D]，其中 A 仍为主节点，B 和 C 地位一样，都是位居第二的备份节点，以此类推。

sync_nodes_mandatory 元组要结合 sync_nodes_timeout 一起使用。当用这个配置启动分布式 VM 时，VM 会一直处于锁定状态，直到所有强制性节点都启动并被锁定。接着，它们之间会进行同步，然后再继续执行。如果启动所有节点的时间超过了 MaxTime，那么它们都将崩溃。

还有许多其他选项可供使用，如果你想了解更多的信息，我推荐你去看看 kernel 应用的文档（http://www.erlang.org/doc/

man/kernel_app.html)。

我们来运行一下 m8ball 应用。如果你不确定是否能够在 30 s 内启动完成这 3 个 VM，可以根据需要增加 sync_nodes_timeout 的值。然后，启动 3 个 VM：

```
$ erl -sname a -config config/a -pa ebin/
```

```
$ erl -sname b -config config/b -pa ebin/
```

```
$ erl -sname c -config config/c -pa ebin/
```

在启动第三个 VM 后，其他两个 VM 应该被同时解锁。在每个 VM 上，使用 application:start(AppName) 命令将 crypto 和 m8ball 分别启动起来。

现在，你可以在任何一个相连接的节点上使用神奇 8 号球的服务了：

```
(a@ferdmbp)3> m8ball:ask("If I crash, will I have a second life?").
<<"I don't like your tone">>
(a@ferdmbp)4> m8ball:ask("If I crash, will I have a second life, please?").
<<"Of Course">>
```

```
(c@ferdmbp)3> m8ball:ask("Am I ever gonna be good at Erlang?").
<<"Doubtful">>
```

太激动了。在每个节点上都调用 application:which_applications()，可以查看到应用的运行情况。只有节点 a 上运行了应用：

```
(b@ferdmbp)3> application:which_applications().
[{crypto,"CRYPTO version 2","2.1"},
 {stdlib,"ERTS CXC 138 10","1.18.1"},
 {kernel,"ERTS CXC 138 10","2.15.1"}]
```

```
(a@ferdmbp)5> application:which_applications().
[{m8ball,"Answer vital questions","1.0.0"},
 {crypto,"CRYPTO version 2","2.1"},
 {stdlib,"ERTS CXC 138 10","1.18.1"},
 {kernel,"ERTS CXC 138 10","2.15.1"}]
```

节点 c 上应该显示出和节点 b 一样的内容。此时，如果你杀死节点 a（粗暴地关掉运行 Erlang shell 的窗口就可以了），应用就不会再在其上运行。我们来看看它跑到哪里去了：

```
(c@ferdmbp)4> application:which_applications().
[{crypto,"CRYPTO version 2","2.1"},
 {stdlib,"ERTS CXC 138 10","1.18.1"},
 {kernel,"ERTS CXC 138 10","2.15.1"}]
(c@ferdmbp)5> m8ball:ask("where are you?!").
<<"I don't like your tone">>
```

这正是所期望的，因为节点 b 的优先级要高一点。5 s 后（我们把超时设置为 5 000 ms），节点 b 上应该显示出该应用正在运行：

```
(b@ferdmbp)4> application:which_applications().
[{m8ball,"Answer vital questions","1.0.0"},
 {crypto,"CRYPTO version 2","2.1"},
```

```
 {stdlib,"ERTS CXC 138 10","1.18.1"},
 {kernel,"ERTS CXC 138 10","2.15.1"}]
```

应用运行良好。现在，以同样粗暴的方法杀死节点 b，5 s 后，应用应该运行在节点 c 上：

```
(c@ferdmbp)6> application:which_applications().
[{m8ball,"Answer vital questions","1.0.0"},
 {crypto,"CRYPTO version 2","2.1"},
 {stdlib,"ERTS CXC 138 10","1.18.1"},
 {kernel,"ERTS CXC 138 10","2.15.1"}]
```

此时，如果按照前面的方式重新启动节点 a，那么它会被挂起。配置文件中规定，要让节点 a 能启动，节点 b 就必须要启动。如果你不指望所有的节点一起启动，就必须把 b 或者 c 指定成可选的。现在，如果我们同时启动节点 a 和 b，那么应用应该能够自动回到节点 a 上执行，对吧？

```
(a@ferdmbp)4> application:which_applications().
[{crypto,"CRYPTO version 2","2.1"},
 {stdlib,"ERTS CXC 138 10","1.18.1"},
 {kernel,"ERTS CXC 138 10","2.15.1"}]
(a@ferdmbp)5> m8ball:ask("is the app gonna move here?").
<<"Of course not">>
```

啊，糟糕。问题是，要使这种机制能够工作，应用的启动必须作为节点启动过程的一部分。为了解决这个问题，可以使用如下方式启动节点 a：

```
$ erl -sname a -config config/a -pa ebin -eval 'application:start(crypto), application:
start(m8ball)'
... <snip> ...
(a@ferdmbp)1> application:which_applications().
[{m8ball,"Answer vital questions","1.0.0"},
 {crypto,"CRYPTO version 2","2.1"},
 {stdlib,"ERTS CXC 138 10","1.18.1"},
 {kernel,"ERTS CXC 138 10","2.15.1"}]
```

此时，节点 c 上会显示出如下内容：

```
=INFO REPORT==== 8-Jan-2013::19:24:27 ===
    application: m8ball
    exited: stopped
    type: temporary
```

在 VM 的启动过程中，会对-eval 选项进行求值。显然，一种更好的做法是：使用 OTP 发布来完成这些初始化工作，不过如果我们把前面学到的知识都包含进来，那么对于这个例子来说就过于重量了。

记住，一般来讲，使用 OTP 发布来保证系统中一切相关部件都就绪是分布式 OTP 应用的最佳选择。

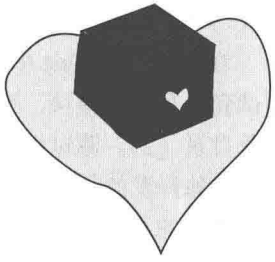
我曾经在前面说过，对许多应用来说（包含神奇 8 号球应用），有时让多个实例同时运行，并在它们之间同步数据，要比强迫应用只在一个节点上运行更简单一些。并且，这种设计也更易于系统的伸缩。不过，如果你需要某种故障切换/接管机制，那么分布式 OTP 应用可能就是你所需要的。

第 28 章

不寻常的 Common Test

在第 24 章，我们介绍了如何使用 EUnit 进行单元和模块级测试，也涉及了部分并发测试的内容，此时，我们遇到了 EUnit 的局限。复杂的初始化工作和相互纠结在一起的冗长测试带来了许多问题。另外，EUnit 也没能利用 Erlang 的分布式能力提供任何对于分布式测试的帮助。还好，我们有另外一个测试框架，它非常适合于更加复杂的测试工作。

28.1 什么是 Common Test



程序员都喜欢把自己的程序看作黑盒。能够将我们所编写的程序看作是一个黑盒是公认的良好抽象的核心原则。你既可以向盒子中放入一些东西，也可以从盒子中取出一些东西。你不用关心盒子内部的工作原理，只要能够得到想要的东西即可。

在测试领域，我们也有类似的观点。在使用 EUnit 时，我们把模块看作黑盒，只测试模块导出的函数，而不关心任何内部没有导出的函数。我也给出了一些以白盒的方式进行测试的例子，如 Process Quest 玩家模块的测试。为了简化测试，我们会去了解该模块的内部状态。这样做是必要的，因为盒子的组成部件是动态的，它们之间的交互使得从外部进行测试非常复杂。

前面的测试是针对模块和函数的。如果我们把镜头拉远一些，关注更大范围的视图会怎么样呢？库、应用该如何测试呢？范围还可以再大一些，一个完整的系统该如何测试呢？我们需要一个更擅长系统测试的工具。

EUnit 非常适用于模块级别的白盒测试，也是一个相当不错的用以测试库和 OTP 应用的工具。我们也可以用它来进行系统测试和黑盒测试，但是这不是最佳选择。

Common Test 是系统测试的最佳工具。用它来测试库和 OTP 应用也相当不错，也可以用它来测试单个模块，不过不是最佳选择。因此，测试的规模越小，EUnit 就越合适（也更灵活和有趣），测试的规模越大，Common Test 就越合适（更灵活，也多点乐趣）。

你也许听说过 Common Test，也试图通过 Erlang/OTP 自带的文档去了解过它。不过，可能很快就放弃了。不用担心。因为 Common Test 非常强大，所以用户指南也非常长。在本书写作期间，Common Test 的大部分文档内容都还是它在爱立信公司内部使用时的内部文档。事实上，它的文档更像是供已经熟悉它的人使用的参考手册，而不是教程。

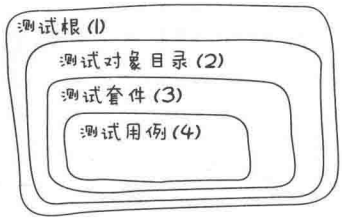
为了更好地学习 Common Test，我们会从最简单的部分开始，逐渐延伸到系统测试。不过在开始之前，我们先来看看 Common Test 的组织结构。

28.2　Common Test 的组织结构

因为 Common Test 是针对系统测试的，所以它会假设两件事情：

- 需要用数据去实例化一些对象；
- 需要有个地方来保存具有“副作用”性质的操作产生的结果。

基于这两个假设，Common Test 通常会被组织成如下方式：

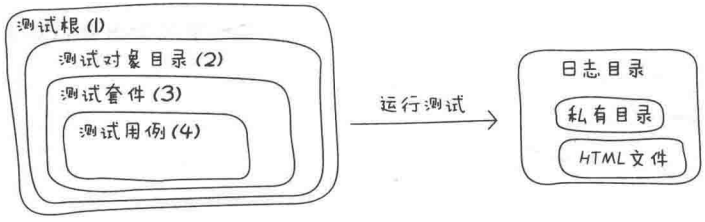


测试用例的部分最为简单，就是一些要么成功、要么失败的代码。如果发生了崩溃，测试就不成功（真令人惊讶）；否则，就认为测试成功。

在 Common Test 中，每个测试用例都是一个函数。所有这些函数都位于测试套件（3）中，测试套件是一个模块，把相关的测试用例组织在一起。每个测试套件都位于一个目录中——测试对象目录（2）。测试根（1）是一个目录，其中包含多个测试对象目录，不过由于 OTP 应用通常都是单独开发的，因此 Erlang 程序员往往会忽略掉这一层。

现在来看看前面的两个假设：要实例化对象，接着进行有副作用的操作。每个测试套件都是一个名字以_SUITE 结尾的模块。假设我们要对第 27 章中的神奇 8 号球应用进行测试，那么就会把测试套件命名为 `m8ball_SUITE`。每个测试套件可以有一个数据目录，通常以 *<Module>_SUITE_data/*命名。以神奇 8 号球应用为例，数据目录的名字为 *m8ball_SUITE_data/*。该数据目录可以包含任何对测试有用的东西。

那么副作用呢？嗯，由于测试会被运行多次，因此 Common Test 需要更多一点的结构，如下：



每次运行测试时，Common Test 都会在某个地方记录日志（通常是在当前目录，不过我们会在 28.7 节介绍如何进行相关的配置）。为此，Common Test 会创建一个专门的目录来存储你的数据。这个目录（结构图中的私有目录）和数据目录会一起作为初始状态的一部分传递给测试。接下来，测试可以在私有目录中写入任何想写的东西，并在以后对其进行查验，这样就不会覆写掉某些重要的东西或者前面测试的运行结果。

组织结构方面的东西说得够多了，下面我们开始编写第一个测试套件。

28.3 创建一个简单的测试套件

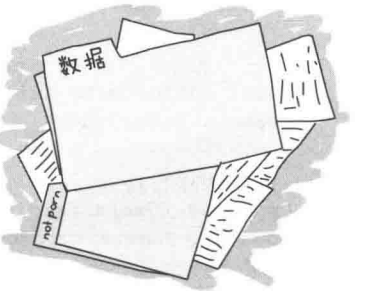
我们来创建一个包含两个测试用例的简单测试套件。先创建一个名为 ct/（或者任何你喜欢的名字——毕竟这是一个自由的国度）的目录。这个目录是测试根目录。在该目录下创建一个名为 demo/的目录，存放简单的示例测试。这个目录是测试对象目录。

在测试对象目录中，创建一个名为 basic_SUITE.erl 的模块，我们用它来检验 Common Test 的基本功能。可以不创建 basic_SUITE_data/目录——此时还不需要，该目录不是 Common Test 必需的。

该模块的代码如下：

```
-module(basic_SUITE).
-include_lib("common_test/include/ct.hrl").
-export([all/0]).
-export([test1/1, test2/1]).

all() -> [test1, test2].

test1(_Config) ->
    1 = 1.

test2(_Config) ->
    A = 0,
    1/A.
```

我们逐条进行说明。首先，需要引入"common_test/include/ct.hrl"文件。该文件中有一些有用的宏，虽然 basic_SUITE 现在并没有使用这些宏，不过，一般来讲，引入这个文件是一个好习惯。



接下来，看看函数 all/0。该函数返回一个测试用例的列表。简单来讲，这个函数告诉 Common Test：“嗨，我想运行这些测试用例！”EUnit 的做法是使用*_test() 或者*_test_()这样的函数名字。Common Test 则是使用显式的函数调用。

_Config 变量是干什么用的？目前还未使用，不过出于学习的目的，我可以告诉你，它包含着测试用例需要的初始状态。这个状态是一个属性列表，初始包含两个值：data_dir 和 priv_dir——也就是存放静态数据和进行副作用操作的两个目录。

运行测试

可以通过命令行或者 Erlang shell 运行测试。在命令行中，调用 `ct_run -suite Name_SUITE` 即可。

> **注意** 对于 R15（2011 年 12 月左右发布）之前的 Erlang/OTP 版本，默认的命令是 `run_test` 而不是 `ct_run`（有些系统两个都支持）。更改为一个不太通用的名字是为了减少和其他应用名字冲突的风险。

运行这个命令，可以得到如下结果：

```
$ ct_run -suite basic_SUITE
... <snip> ...
Common Test: Running make in test directories...
Recompile: basic_SUITE
... <snip> ...
Testing ct.demo.basic_SUITE: Starting test, 2 test cases

- - - - - - - - - - - - - - - - - - - - - - - - - - - -
basic_SUITE:test2 failed on line 13
Reason: badarith
- - - - - - - - - - - - - - - - - - - - - - - - - - - -

Testing ct.demo.basic_SUITE: *** FAILED *** test case 2 of 2
Testing ct.demo.basic_SUITE: TEST COMPLETE, 1 ok, 1 failed of 2 test cases

Updating /Users/ferd/code/self/learn-you-some-erlang/ct/demo/index.html... done
Updating /Users/ferd/code/self/learn-you-some-erlang/ct/demo/all_runs.html... done
```

可以看出，有一个测试用例失败了。另外，可以看到产生了一组 HTML 文件。在了解这些文件的内容之前，先来看看如何在 Erlang shell 中运行测试：

```
$ erl
... <snip> ...
1> ct:run_test([{suite, basic_SUITE}]).
... <snip> ...
Testing ct.demo.basic_SUITE: Starting test, 2 test cases

- - - - - - - - - - - - - - - - - - - - - - - - - - - -
basic_SUITE:test2 failed on line 13
Reason: badarith
- - - - - - - - - - - - - - - - - - - - - - - - - - - -
... <snip> ...
Updating /Users/ferd/code/self/learn-you-some-erlang/ct/demo/index.html... done
Updating /Users/ferd/code/self/learn-you-some-erlang/ct/demo/all_runs.html... done
ok
```

我对输出结果做了些裁剪，不过仍可以看出 shell 给出的结果和命令行的完全一样。

现在来看看这些 HTML 文件：

```
$ ls
all_runs.html
basic_SUITE.beam
basic_SUITE.erl
ct_default.css
ct_run.NodeName.YYYY-MM-DD_20.01.25/
ct_run.NodeName.YYYY-MM-DD_20.05.17/
index.html
variables-NodeName
```

哦，天呀，Common Test 对我们整洁的目录究竟都做了些什么呀？这样的行为简直太可耻了。里面多了两个目录。如果胆大的话，可以随意到里面去看看，不过像我这样比较胆小的人，更倾向于先看看 all_runs.html 或者 index.html。前者包含了所有已运行测试的索引链接，后者则只包含最新一次运行的链接。选择一个文件打开，然后在浏览器中用鼠标寻找（如果不信任鼠标的话，可以使用键盘），直到找到那个包含两个测试用例的测试套件：

Num	Module	Case	Log	Time	Result	Comment
1	basic_SUITE	test1	< >	0.000s	Ok	
2	basic_SUITE	test2	< >	0.000s	FAILED	{basic_SUITE,test2,13} badarith
	TOTAL			0.139s	**FAILED**	1 Ok, 1 Failed of 2

可以看到，test2 失败了。单击带下划线的行号，可以看到模块的原始副本。单击 test2 链接，可以看到详细的运行情况日志：

```
=== source code for basic_SUITE:test2/1
=== Test case started with:
basic_SUITE:test2(ConfigOpts)
=== Current directory is "Somewhere on my computer"
=== Started at 2013-01-20 20:05:17
[Test Related Output]
=== Ended at 2013-01-20 20:05:17
=== location [{basic_SUITE,test2,13},
              {test_server,ts_tc,1635},
              {test_server,run_test_case_eval1,1182},
              {test_server,run_test_case_eval,1123}]
=== reason = bad argument in an arithmetic expression
   in function basic_SUITE:test2/1 (basic_SUITE.erl, line 13)
   in call from test_server:ts_tc/3 (test_server.erl, line 1635)
   in call from test_server:run_test_case_eval1/6 (test_server.erl, line 1182)
   in call from test_server:run_test_case_eval/9 (test_server.erl, line 1123)
```

日志中准确地记录了详细的失败信息，其详细程度要超过 Erlang shell 给出的信息。因此，如果你是一个 shell 用户，就会觉得 Common Test 使用起来异常痛苦。但如果你是一个倾向于使

用 GUI 的人，那一定会觉得乐趣无穷。

HTML 方面的内容了解这么多就够了，下面我们来看看如何进行有状态的测试。

> **注意**　如果你想不借助时间机器体验一下回到过去的感觉，可以去下载一个 R15B 之前的 Erlang 版本，并使用其中的 Common Test。你会惊讶地看到你的网页和日志风格把你带回了 20 世纪 90 年代末期。

28.4　带状态的测试

我们在第 24 章介绍过，EUnit 中有个概念叫测试夹具（fixture），可以为测试用例提供一些特定的实例化（初始化）和清理代码，这些代码会在测试用例执行的前后调用。

Common Test 中也有测试夹具的概念。不过，和 EUnit 的风格不同，它依赖于两个函数：

- 初始化函数 `init_per_testcase/2`；
- 清理函数 `end_per_testcase/2`。

我们创建一个新的测试套件 `state_SUITE`（仍然在 demo/目录下）来演示一下这两个函数的使用方法，在模块中增加如下代码：

```
-module(state_SUITE).
-include_lib("common_test/include/ct.hrl").

-export([all/0, init_per_testcase/2, end_per_testcase/2]).
-export([ets_tests/1]).

all() -> [ets_tests].

init_per_testcase(ets_tests, Config) ->
    TabId = ets:new(account, [ordered_set, public]),
    ets:insert(TabId, {andy, 2131}),
    ets:insert(TabId, {david, 12}),
    ets:insert(TabId, {steve, 12943752}),
    [{table,TabId} | Config].

end_per_testcase(ets_tests, Config) ->
    ets:delete(?config(table, Config)).

ets_tests(Config) ->
    TabId = ?config(table, Config),
    [{david, 12}] = ets:lookup(TabId, david),
    steve = ets:last(TabId),
    true = ets:insert(TabId, {zachary, 99}),
    zachary = ets:last(TabId).
```

这是一个普通的 ETS 测试，检查 `ordered_set` 类型的一些特点。我们感兴趣的是其中的两个新函数：`init_per_testcase/2` 和 `end_per_testcase/2`。这两个函数要得到调用，就需要被导出。如果导出了这两个函数，模块中的所有测试用例都会调用这两个函数。可以按照

参数来分隔这两个函数的子句。第一个参数是测试用例的名字(原子类型),第二个参数是 Config 属性列表，这个参数是可修改的。

> **注意** 从 Config 中读取数据时，并没有使用 proplists:get_value/2 函数。Common Test 的包含文件中有一个?config(Key, List)宏，会返回和给定键匹配的值。该宏其实是对 proplists:get_value/2 的封装，文档中也是这样说明的，因此你可以把 Config 当成属性列表进行处理，不用担心会出现问题。

例如，如果有 a、b、c 这 3 个测试用例，并且只想为前两个测试用例提供初始化函数和清理函数，那么可以这样编写 init 函数：

```
init_per_testcase(a, Config) ->
    [{some_key, 124} | Config];
init_per_testcase(b, Config) ->
    [{other_key, duck} | Config];
init_per_testcase(_, Config) ->
   %% 忽略其他所有情况。
   Config.
```

end_per_testcase/2 函数的编写方式与此类似。

在 state_SUITE 测试套件中，有一个测试用例，不过真正有趣的地方是实例化 ETS 表的代码。我们没有指定 ETS 的继承者进程，而测试却可以在 init 函数执行完成后正常运行。

正如第 25 章所讲，ETS 表属于创建它的进程。在这个例子中，我们并没有对表做什么。如果你运行测试，就会发现测试套件会成功执行。

从中，我们可以推断出，init_per_testcase 函数和 end_per_testcase 函数与测试用例运行在同一个进程之中。因此，可以安全地在这两个函数中做诸如设置进程链接或者创建表之类的事情，而不用担心这些函数运行在不同的进程中可能会导致的问题。如果测试用例中出现了错误怎么办？还好，即使测试用例中出现了崩溃，Common Test 还是会调用 end_per_testcase 函数执行清理工作，kill 退出信号的情况是一个例外。

到目前为止，Common Test 在功能和灵活性方面看起来和 EUnit 完全一样。虽然 Common Test 不具备 EUnit 中那么多种漂亮的断言宏，但是它具有精美的报告、类似的测试夹具以及可以随意写入的私有目录。还不满足吗？

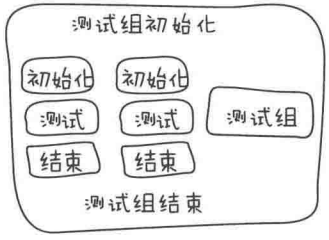
> **注意** 如果想在测试中打印一些信息，以帮助调试或者指示测试的进展，就会发现 io:format/1-2 函数只会把信息打印到 HTML 日志中，不会输出到 Erlang shell 上。如果希望在两者中都有输出（还能自动带上时间戳信息），可以使用函数 ct:pal/1-2。它的工作方式和 io:format/1-2 类似，但是会同时输出到 shell 和日志中。

28.5 测试组

到目前为止，我们所看到的测试套件的结构如下图所示：

如果有多个测试用例，其 init 函数有部分相同、部分不同时该如何处理呢？嗯，最容易的办法当然是复制/粘贴、修改，但是这会造成维护问题。此外，如果不想逐个串行运行，而是想并行或者以随机次序运行多个测试该怎么办呢？基于我们现在学习到的知识，这不是一个容易解决的问题。这也正是 EUnit 的局限所在。

为了解决这些问题，需要引入测试组的概念。Common Test 的测试组可以把测试重组成层次化的形式，甚至可以把某些组放到另外一些组中，如下图所示：



在这个层次结构中，测试组具有自己的初始化和结束函数，可以包含多个测试用例或者其他组。这样，我们就可以定义一些公共环境，把它用于一大批相关的测试，甚至是其他的测试组。例如，如果有一半的测试用例需要用某种配置连接到数据库，另一半需要用另外一种配置连接到数据库，那么以不同测试组的方式进行这种配置设置是最为高效的做法。

28.5.1　定义测试组

要实现上述设想，就需要定义测试组。首先，增加一个 groups() 函数，该函数返回所有的测试组：

```
groups() -> ListOfGroups.
```

ListOfGroups 的内容如下：

```
[{GroupName, GroupProperties, GroupMembers}]
```

下面是一个具体的例子：

```
[{test_case_street_gang,
  [],
  [simple_case, more_complex_case]}].
```

这是一个非常简单的 test_case_street_gang 测试组。下面是一个更复杂的例子：

```
[{test_case_street_gang,
  [shuffle, sequence],
  [simple_case, more_complex_case,
   emotionally_complex_case,
   {group, name_of_another_test_group}]}].
```

这个例子中指定了两个属性：shuffle 和 sequence。稍后会对它们的含义进行介绍。

该例子也展示了如何在一个组中包含另外一个组。此时，就要求把 groups() 函数定义成如下样子：

```
groups() ->
    [{test_case_street_gang,
      [shuffle, sequence],
      [simple_case, more_complex_case, emotionally_complex_case,
       {group, name_of_another_test_group}]},
     {name_of_another_test_group,
      [],
      [case1, case2, case3]}].
```

也可以在组中直接定义另外一个组，如下所示：

```
[{test_case_street_gang,
  [shuffle, sequence],
  [simple_case, more_complex_case,
   emotionally_complex_case,
   {name_of_another_test_group,
    [],
    [case1, case2, case3]}
  ]}].
```

现在代码看起来有点复杂了，是吧？仔细理解例子，慢慢就会觉得简单了。记住，不是必须以嵌套的方式定义组，如果觉得难以理解，可以不使用它。

等等，如何在测试中使用这些组呢？只要把它们放入 all/0 函数中即可：

```
all() -> [some_case, {group, test_case_street_gang}, other_case].
```

这样，Common Test 就知道运行的是单个测试用例还是测试组了。

28.5.2 测试组属性

在上面的例子中，我们用到了一些测试组属性，包括 shuffle、sequence 以及空列表。可用的测试组属性如下所示。

空列表/无选项

测试组中的测试用例依次运行。如果一个测试失败了，其后的测试会接着运行。

shuffle

随机运行测试用例。使用的随机种子（初始值）会以{A,B,C}的形式打印到 HTML 日志中。如果某个特定的测试序列失败了，可以使用 HTML 日志中的种子值，把 shuffle 选项更改为{shuffle,{A,B,C}}来进行重现。在需要时，可以采用这种方式以准确的顺序重现随机运行。

parallel

测试会在不同的进程中运行。需要小心的是，如果忘记导出函数 init_per_group 和

end_per_group，Common Test 会忽略掉这个选项，不会给出任何提醒。

sequence

这个选项并不意味着测试一定会按照顺序运行，而是说如果组中的一个测试失败了，那么随后的所有测试都会被忽略掉。这个选项可以和 shuffle 选项结合在一起使用，这样的话，如果随机测试中任何一个测试失败，就都不会再运行随后的测试。

{repeat,Times}

这个选项会让组运行 Times 次。使用选项[parallel, {repeat, 9}]可以让这个测试序列并行运行 9 次。也可以把 Times 设置成 forever，当然这里的“永远”有夸大的成分，因为毕竟会出现硬件故障，或者宇宙热寂。

{repeat_until_any_fail, N}

这个选项会让所有的测试都运行 N 次或者某个测试失败时停止。N 可以是 forever。

{repeat_until_all_fail, N}

这个选项和上一个的工作方式一样，不同之处在于，只有所有测试用例都失败了，运行才会停止。

{repeat_until_any_succeed, N}

这个选项和上两个的工作方式一样，不同之处在于，只要有一个测试用例成功了，运行就会结束。

{repeat_until_all_succeed, N}

你应该已经猜到了这个选项的含义，保险起见，还是说明一下：这个选项和上面几个的工作方式一样，不同之处在于，只有所有测试用例都成功了，运行才会结束。

坦白地讲，关于测试组的内容说得够多了，是时候给出一个例子了。

28.5.3　会议室

为了使用测试组，我们创建一个会议室预定模块：

```
-module(meeting).
-export([rent_projector/1, use_chairs/1, book_room/1,
         get_all_bookings/0, start/0, stop/0]).
-record(bookings, {projector, room, chairs}).

start() ->
    Pid = spawn(fun() -> loop(#bookings{}) end),
    register(?MODULE, Pid).

stop() -> ?MODULE ! stop.

rent_projector(Group) -> ?MODULE ! {projector, Group}.

book_room(Group) -> ?MODULE ! {room, Group}.
```

```
use_chairs(Group) -> ?MODULE ! {chairs, Group}.
```

这些简单的函数会调用一个中心登记管理进程。使用这些函数，可以预定会议室、租用投影仪以及使用座椅。出于练习的目的，假设我们处在一个庞大的组织中，这个组织有复杂的企业结构。因此，有 3 个人分别负责投影仪、会议室和座椅，但是只有一个中心登记处。基于这种结构，无法同时预定这 3 项东西，只能发送 3 条不同的消息。

为了知道详细的预定信息，可以给登记进程发送一条消息获取所有的预定记录：

```
get_all_bookings() ->
    Ref = make_ref(),
    ?MODULE ! {self(), Ref, get_bookings},
    receive
        {Ref, Reply} ->
            Reply
    end.
```

登记进程的代码如下：

```
loop(B = #bookings{}) ->
    receive
        stop -> ok;
        {From, Ref, get_bookings} ->
            From ! {Ref, [{room, B#bookings.room},
                          {chairs, B#bookings.chairs},
                          {projector, B#bookings.projector}]},
            loop(B);
        {room, Group} -> loop(B#bookings{room=Group});
        {chairs, Group} -> loop(B#bookings{chairs=Group});
        {projector, Group} -> loop(B#bookings{projector=Group})
    end.
```

代码就这些。

要进行一次成功的会议预定，必须要连续地调用这些函数：

```
1> c(meeting).
{ok,meeting}
2> meeting:start().
true
3> meeting:book_room(erlang_group).
{room,erlang_group}
4> meeting:rent_projector(erlang_group).
{projector,erlang_group}
5> meeting:use_chairs(erlang_group).
{chairs,erlang_group}
6> meeting:get_all_bookings().
[{room,erlang_group},
 {chairs,erlang_group},
 {projector,erlang_group}]
```

不过，这看起来好像不太对。你可能总觉得哪个地方会出问题。绝大多数情况下，如果能足够快地发起这 3 个调用，应该可以得到想要的东西而不会出现问题。但是，如果两个人同时发起调用，并且在调用之间有短暂的停顿，那么就很可能出现两个（或者多个）组同时去租用同一个设备的情况。

哦，不！突然，糟糕的情况出现了，程序员们拿到了投影仪，董事长预定到了会议室，而人力资源部门租到了所有的座椅。所有的资源都分配了，但是没人能开得了会。

在此，我们不会去修正这个问题。我们会使用 Common Test 的测试套件来揭示这个问题。

所创建的测试套件名为 meeting_SUITE.erl，它基于一个简单的思路：通过引发一个竞争条件来搞乱登记进程的数据。我们会创建 3 个测试用例，每个用例代表了一个团体。Carla 代表女性，Mark 代表男性，Dog 代表动物（它们想用人类的工具召开一次会议），代码如下：

```
-module(meeting_SUITE).
-include_lib("common_test/include/ct.hrl").
...
carla(_Config) ->
    meeting:book_room(women),
    timer:sleep(10),
    meeting:rent_projector(women),
    timer:sleep(10),
    meeting:use_chairs(women).

mark(_Config) ->
    meeting:rent_projector(men),
    timer:sleep(10),
    meeting:use_chairs(men),
    timer:sleep(10),
    meeting:book_room(men).

dog(_Config) ->
    meeting:rent_projector(animals),
    timer:sleep(10),
    meeting:use_chairs(animals),
    timer:sleep(10),
    meeting:book_room(animals).
```

我们不关心这些测试中实际的测试内容。它们的存在只是为了使用 meeting 模块（后面会把这个模块放置到正确的地方）并试图产生出错误的预定结果。

要想知道这些测试间是否出现了竞争条件，我们还需要第 4 个，也是最后一个测试用例，其中使用了 meeting:get_all_bookings() 函数：

```
all_same_owner(_Config) ->
    [{_, Owner}, {_, Owner}, {_, Owner}] =
    meeting:get_all_bookings().
```

这个测试对所有可预定对象的拥有者进行模式匹配，看看它们是不是被同一个拥有者所预定。这正是成功召开一次会议所期望的。

有了这 4 个测试用例，接下来如何做才能引发出竞争条件呢？我们需要聪明地使用测试组。首先，要引发竞争条件，需要并行地运行一批测试用例。其次，为了能够看到竞争条件所导致的问题，要么在测试用例运行期间多次调用 all_same_owner，要么在整个测试用例运行完成后调用一次。

我选择了后者：

```
all() -> [{group, clients}, all_same_owner].

groups() -> [{clients, [parallel, {repeat, 10}], [carla, mark, dog]}].
```

这段代码创建了一个 clients 测试组，其中包含 3 个单独的测试：carla，mark 以及 dog。这 3 个测试用例会并行执行，每个执行 10 次。

可以看到，all/0 函数中包含了这个测试组，随后跟着的是 all_same_owner。这是因为，默认情况下 Common Test 会按照 all/0 中声明的顺序运行测试用例和测试组。

等等！我们忘记启动和停止 meeting 进程了。对所有的测试，不管是不是属于 clients 测试组，meeting 进程都要保持活着。解决方法是再增加一个嵌套层次，将它们都放到另外一个测试组中：

```
all() -> [{group, session}].

groups() -> [{session, [], [{group, clients}, all_same_owner]},
             {clients, [parallel, {repeat, 10}], [carla, mark, dog]}].

init_per_group(session, Config) ->
   meeting:start(),
   Config;
init_per_group(_, Config) ->
   Config.

end_per_group(session, _Config) ->
   meeting:stop();
end_per_group(_, _Config) ->
   ok.
```

我们使用函数 init_per_group 和 end_per_group 让 session 测试组（包含{group, clients}和 all_same_owner）使用一个活动的 meeting 进程。别忘了导出初始化函数和清理函数，否则，并行运行的开关将不起作用。

好了，运行一下测试，看看结果如何：

```
1> ct_run:run_test([{suite, meeting_SUITE}]).
... <snip> ...
Common Test: Running make in test directories...
... <snip> ...
TEST INFO: 1 test(s), 1 suite(s)
```

```
Testing ct.meeting.meeting_SUITE: Starting test (with repeated test cases)
- - - - - - - - - - - - - - - - - - - - - - - - - - - -
meeting_SUITE:all_same_owner failed on line 50
Reason: {badmatch,[{room,men},{chairs,women},{projector,women}]}
- - - - - - - - - - - - - - - - - - - - - - - - - - - -
Testing ct.meeting.meeting_SUITE: *** FAILED *** test case 31
Testing ct.meeting.meeting_SUITE: TEST COMPLETE, 30 ok, 1 failed of 31 test cases
... <snip> ...
ok
```

很好！出现了一个带有 3 个元组的 badmatch 错误，其中预定对象的拥有者并不相同。此外，从输出中可以看出，是 all_same_owner 测试失败了。这是个很好的征兆，预示着 all_same_owner 的崩溃符合预期。查看一下 HTML 日志，就可以看到所有的运行结果，其中含有失败的那个测试以及失败的原因。单击该测试用例的名字，可以看到详细的信息。

注意　关于测试组，还有一件重要的事情需要知道，测试用例的 init 函数和测试用例运行在同一个进程中，但是测试组的 init 函数和测试用例运行在不同的进程中。这意味着，如果在初始化函数中创建了进程，并建立了链接，那么要确保首先取消它们之间的链接。对于 ETS 表来说，需要定义一个继承者进程，确保 ETS 表不会消失。这个原则适用于所有附着于进程的对象，如 socket 和文件描述符。

28.6　再谈测试套件

在使用测试套件时，是否存在一种比把测试组嵌套起来、以层次结构的方式来控制测试运行更好的方法呢？真的不存在，但是即便如此，我们还是可以再增加一个针对测试套件本身的层次：

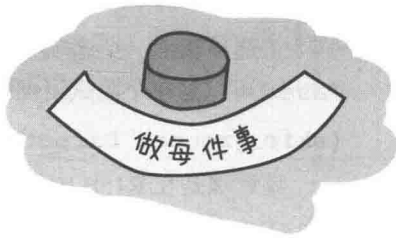
其中，又增加了两个函数：init_per_suite(Config) 和 end_per_suite(Config)。这两个函数和其他的 init 和 end 函数一样，旨在提供更多的数据和进程初始化方面的控制。

init_per_suite/1 和 end_per_suite/1 只会在所有测试组和测试用例运行前后运行一次。可以用它们来处理所有测试都需要的公共状态和依赖关系。例如，手工启动所依赖的其他应用。

28.7　测试规格说明

测试运行之后，测试目录中的内容非常混乱：大量日志相关的内容散落在目录中——CSS 文件、

HTML 日志、目录、测试运行历史等。如果能够把这些文件存放到一个目录中，就会整洁很多。



还有一个问题，到目前为止都是运行一个测试套件中的所有测试用例，还不知道如何同时运行多个测试套件，以及如何只运行一个测试套件（或者多个测试套件）中的部分测试用例和测试组。

我既然提出了这些问题，当然是有解决方案的。有多种方法可以解决这些问题，有基于命令行的，也有基于 Erlang shell 的，可以从 `ct_run` 的帮助文档中（http://www.erlang.org/doc/man/ct_run.html）找到相关内容。不过，在这里，我们不介绍每次运行测试时手工配置的方法，而是使用*测试规格说明*。

测试规格说明是一些特殊文件，用于详细说明测试的运行方式。它同时适用于 Erlang shell 和命令行。测试规格说明可放置在具有任意扩展名的文件中（我个人比较喜欢.spec 扩展名）。

28.7.1 规格说明文件内容

规格说明文件由 Erlang 的元组构成，非常像 consult 文件（包含 Erlang 数据项的文件，可以使用 `file:consult/1` 进行解析）。下面是一些可以出现在规格说明文件中的条目：

`{include, IncludeDirectories}`

在 Common Test 编译测试套件时，可以使用这个选项指定头文件的查找目录。`IncludeDirectories` 的值必须是字符串（列表）或者是字符串列表（列表的列表）。

`{logdir, LoggingDirectory}`

在产生日志时，所有的日志都会被记录在 `LoggingDirectory`（一个字符串）。注意，在测试运行前，该目录必须要存在，否则 Common Test 会提示错误。

`{suites, Directory, Suites}`

该选项指示在目录 `Directory` 中寻找测试套件 `Suites`。`Suites` 可以是原子（`some_SUITE`）、原子列表或者是原子 `all`（运行目录中的所有测试套件）。

`{skip_suites, Directory, Suites, Comment}`

这个选项用于把测试套件 `Suites` 从先前定义的测试套件中去掉，并在运行时忽略它们。`Comment` 参数是一个字符串，用来解释忽略的原因。这个注释会被放到最终的 HTML 日志中。在表中会以黄色显示“SKIPPED: *Reason*”，*Reason* 就是 `Comment` 中的内容。

`{groups, Directory, Suite, Groups}`

该选项的意思是，从给定的 `Suite` 测试套件中只选出部分测试组运行。`Groups` 变量可以是一个原子（测试组名字）或者 `all`（所有组）。这个变量的值也可以更加复杂，用来覆盖掉测试用例中 `groups()` 函数返回的测试组定义，例如，你可以使用像`{GroupName, [parallel]}`这样的值把测试组 `GroupName` 的属性改写为 `parallel`，无需重新编译测试即可生效。

{groups, Directory, Suite, Groups, {cases,Cases}}

该选项和上一个选项类似，不过它可以通过 Cases 让你指定加入测试组中的测试用例，Cases 可以是单个测试用例的名字（原子）、名字列表或者原子 all。

{skip_groups, Directory, Suite, Groups, Comment}

该选项是在 R15B 中增加的，在 R15B01 中提供了文档说明。用它可以忽略掉测试组，和 skip_suites 选项非常相似。

{skip_groups, Directory, Suite, Groups, {cases,Cases}, Comment}

这个选项和上一个类似，不过通过它还可以忽略掉一些测试用例。R15B 之后的版本才提供该选项。

{cases, Directory, Suite, Cases}

这个选项指定运行 Suite 套件中的 Cases 用例。Cases 可以是一个原子、原子列表或者 all。

{skip_cases, Directory, Suite, Cases, Comment}

这个和 skip_suites 类似，不同之处在于这个选项忽略的是测试用例，而不是测试套件。

{alias, Alias, Directory}

输入这么多目录名（尤其是全路径名字）非常烦人，因此 Common Test 允许使用别名（原子）来代替目录名。该选项可以大大简化书写的内容。

28.7.2　创建规格说明文件

我们来实验一个小例子。首先，创建一个和目录 ct/demo/同级的目录 ct/logs/。显然，它是用来存放所有 Common Test 日志的。

下面是到目前为止编写的所有测试的规格说明文件，我们给它起一个非常有想象力的名字 spec.spec：

```
{alias, demo, "./demo/"}.
{alias, meeting, "./meeting/"}.
{logdir, "./logs/"}.

{suites, meeting, all}.
{suites, demo, all}.
{skip_cases, demo, basic_SUITE, test2, "This test fails on purpose"}.
```

这个规格说明文件中定义了两个别名，即 demo 和 meeting，分别指向我们创建的两个目录。我们把日志放到新创建的目录 ct/logs 中。接下来，我们要求运行 meeting 目录中的所有测试套件，也就是 meeting_SUITE 套件。

接下来是 demo 目录中的两个测试套件。我们要求忽略掉 basic_SUITE 套件中的 test2 用例，因为该用例中有一个除零操作，肯定会失败。

28.7.3 通过规格说明文件运行测试

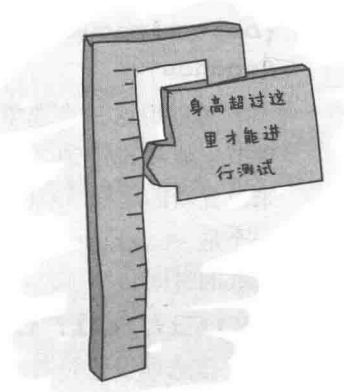
在命令行中，可以使用 `ct_run -spec spec.spec`（如果是 R15 之前的 Eralng 版本，可以使用 `run_test`）运行测试。在 Erlang Shell 中，可以调用函数 `ct:run_test([{spec, "spec.spec"}])`.运行测试：

```
Common Test: Running make in test directories...
... <snip> ...
TEST INFO: 2 test(s), 3 suite(s)

Testing ct.meeting: Starting test (with repeated test cases)

- - - - - - - - - - - - - - - - - - - - - - - - - -
meeting_SUITE:all_same_owner failed on line 51
Reason: {badmatch,[{room,men},{chairs,women},{projector,women}]}
- - - - - - - - - - - - - - - - - - - - - - - - - -

Testing ct.meeting: *** FAILED *** test case 31
Testing ct.meeting: TEST COMPLETE, 30 ok, 1 failed of 31 test cases

Testing ct.demo: Starting test, 3 test cases
Testing ct.demo: TEST COMPLETE, 2 ok, 0 failed, 1 skipped of 3 test cases

Updating /Users/ferd/code/self/learn-you-some-erlang/ct/logs/index.html... done
Updating /Users/ferd/code/self/learn-you-some-erlang/ct/logs/all_runs.html... done
```

花点时间到日志目录中看看，就会发现其中有两个目录分别对应两个不同的测试运行。其中一个包含有一个失败——正是期望的会议室预订失败。另外一个中有一个成功的用例和一个被忽略的用例，格式为 1 (1/0)。忽略表示的一般格式为“*忽略的总数（有意忽略的个数/因为错误忽略的个数）*”。在这个例子中，忽略是由规格说明文件指定的，因此属于*有意忽略*。由于某个 `init` 函数失败造成的忽略则属于*错误忽略*。

现在 Common Test 看起来已经是一个非常不错的测试框架了，当然，如果它能够利用 Erlang 的分布式编程能力就更好了。

28.8 大规模测试



Common Test 支持分布式测试。在我们跃跃欲试去编写代码前，先来看看它都提供了哪些支持。嗯，提供的支持其实也不是很多。关键之处在于，Common Test 可以在多个不同节点上启动测试，并且可以动态地启动这些节点，并使它们相互监控。

因此，如果有大量测试套件需要并行地在多个节点上运行，那么 Common Test 的分布式特性是非常有用的。一方面这样做可以节省时间；另一方面，由于代码在生产环境中就是运行在多个计算机上的，因此能够体现出这一点的自动化测试也是所希望的。

当进行分布式测试时，Common Test 要求有一个*中心节点*（CT master）管理其他所有节点。一切指令都由该节点下发，包含启动节点、安排测试运行的顺序、

收集日志等。

要把测试变成分布式的，首先需要扩展测试规格说明的内容。我们将增加两个新的元组：{node, NodeAlias, NodeName}和{init, NodeAlias, Options}。

{node, NodeAlias, NodeName}和{alias, AliasAtom, Directory}非常相似，区别在于{alias, AliasAtom, Directory}是针对测试套件、测试组以及测试用例的，而{node, NodeAlias, NodeName}则是针对节点名称的。NodeAlias 和 NodeName 都必须是原子类型。当 NodeName 是长节点名时，这个选项非常有用，可以避免冗长的节点名在文件中多次重复出现所造成的混乱。

{init, NodeAlias, Options} 选项要复杂一些，用来进行节点的启动。NodeAlias 既可以是单个节点的别名，也可以是多个别名的列表。Options 供 ct_slave 模块使用。

下面是一些可用的 Options 的值。

{username, UserName}和{password, Password}

Common Test 会使用 UserName 和 Password，尝试对 NodeAlias 给出的节点名中的主机进行 SSH（端口 22）连接，后续的工作都基于此连接的建立。

{startup_functions, [{M,F,A}]}

这个选项定义了一个函数列表，当其他节点启动时会立即调用这些函数。

{erl_flags, String}

该选项用来设置在启动 erl 时传递给它的标准参数。例如，如果想用 erl -env ERL_LIBS ../ -config conf_file 来启动一个节点，该选项就可以设置为{erl_flags, "-env ERL_LIBS ../ -config config_file"}。

{monitor_master, true | false}

如果该选项设置为 true，那么当 CT 主节点停止运行时，从节点也会停止运行。如果创建了远程节点，我推荐设置这个选项为 true。否则，当主节点停止后，从节点还会在后台保持运行。而且，当再次运行测试时，Common Test 会连接到这些节点，这些节点中可能保存有某些不期望的状态。

{boot_timeout, Seconds}、{init_timeout, Seconds}和{startup_timeout, Seconds}

可以用这 3 个选项设置远程节点启动的不同阶段的等待时间。boot_timeout 用来设置节点能够 ping 通之前的等待时间，默认值是 3 s。init_timeout 是一个内部计时器，用来设置新的远程节点回调 CT 主节点，报告它已经启动完成的等待时间，默认值是 1 s。最后一个是 startup_timeout，用来设置等待 startup_functions 选项中定义的函数执行完成的时间。

{kill_if_fail, true | false}

该选项设置对上一个选项中 3 个超时的处理方式。当有任何一个超时发生时，Common Test 都会中止连接，忽略测试，但是不一定会杀死节点，除非该选项的值为 true。还好，

true 就是默认值。

> **注意** 所有这些选项都在 ct_slave 模块中提供。可以定义自己的模块去启动从节点，只要遵从正确的接口即可。

好了，我们已经有不少设置远程节点选项了，这些选项赋予了 Common Test 分布式测试的能力。现在，我们能够启动节点了，并且具备了和 shell 中手工启动节点同样多的控制能力。对于分布式测试来说，还有其他一些和启动节点无关的选项可供使用：

```
{include, Nodes, IncludeDirs}
{logdir, Nodes, LogDir}
{suites, Nodes, DirectoryOrAlias, Suites}
{groups, Nodes, DirectoryOrAlias, Suite, Groups}
{groups, Nodes, DirectoryOrAlias, Suite, GroupSpec, {cases,Cases}}
{cases, Nodes, DirectoryOrAlias, Suite, Cases}
{skip_suites, Nodes, DirectoryOrAlias, Suites, Comment}
{skip_cases, Nodes, DirectoryOrAlias, Suite, Cases, Comment}
```

这些选项和前面介绍过的类似，区别之处在于它们多了一个 Nodes 参数。使用这些选项，可以把测试套件分配到不同的节点上运行。在进行系统测试时，如果需要多个节点，每个节点又运行着不同的环境或者子系统（如数据库或者外部应用），那么这些选项是非常有用的。

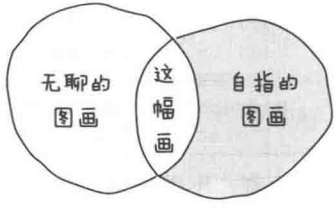
28.8.1 创建分布式测试规格说明文件

我们把前面的 spec.spec 文件更改成分布式的，简单展示一下分布式测试的工作方式。把该文件复制成 dist.spec，然后把它修改成如下的样子：

```
{node, a, 'a@ferdmbp.local'}.
{node, b, 'b@ferdmbp.local'}.

{init, [a,b], [{node_start, [{monitor_master, true}]}]}.

{alias, demo, "./demo/"}.
{alias, meeting, "./meeting/"}.

{logdir, [all_nodes, master], "./logs/"}.

{suites, [b], meeting, all}.
{suites, [a], demo, all}.
{skip_cases, [a], demo, basic_SUITE, test2, "This test fails on purpose"}.
```

在这个文件中，我们定义了两个从节点 a 和 b，在测试时需要被启动起来。对这两个从节点没有什么特殊的要求，只是当主节点死亡时，它们也要死亡。目录的别名和先前一样，没有什么变化。

logdir 的值有些意思。我们并没有把某个节点的别名定义为 all_nodes 或者 master，但是却使用了它们。在 Common Test 中，all_nodes 别名表示所有非主节点，master 表示主节点本身。要包含所有节点，就需要使用 [all_nodes, master]。（为啥选取这两个名字并没有明确的解释。）



logdir 的值之所以这样设置，是因为 Common Test 会为每

个从节点都创建日志文件（和目录），并且也会在主节点产生一组关联到从节点的日志。我们希望所有这些日志都存在 logs/目录中。注意，从节点的日志分别存储在每个从节点上。因此，除非所有节点共享文件系统，否则主节点日志中的 HTML 链接无法跳转到从节点上的内容，只能访问每个从节点获取各自的日志内容。

最后是 suites 和 skip_cases 选项。它们和前面的定义基本一样，只是增加了节点信息。这样就可以只在某些给定节点上忽略某些测试，这些节点可能不具备需要的库或者依赖物，也可能硬件配置比较低，难以承担繁重的测试任务。

28.8.2　运行分布式测试

要运行分布式测试，必须使用-name 启动一个分布式节点，并用 ct_master 来运行测试套件（不能用 ct_run 运行分布式测试）：

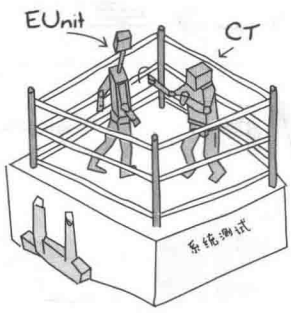
```
$ erl -name ct
Erlang R15B (erts-5.9) [source] [64-bit] [smp:4:4] [async-threads:0] [hipe]
[kernel-poll:false]

Eshell V5.9 (abort with ^G)
(ct@ferdmbp.local)1> ct_master:run("dist.spec").
=== Master Logdir ===
/Users/ferd/code/self/learn-you-some-erlang/ct/logs
=== Master Logger process started ===
<0.46.0>
Node 'a@ferdmbp.local' started successfully with callback ct_slave
Node 'b@ferdmbp.local' started successfully with callback ct_slave
=== Cookie ===
'PMIYERCHJZNZGSRJPVRK'
=== Starting Tests ===
Tests starting on: ['b@ferdmbp.local','a@ferdmbp.local']
=== Test Info ===
Starting test(s) on 'b@ferdmbp.local'...
=== Test Info ===
Starting test(s) on 'a@ferdmbp.local'...
=== Test Info ===
Test(s) on node 'a@ferdmbp.local' finished.
=== Test Info ===
Test(s) on node 'b@ferdmbp.local' finished.
=== TEST RESULTS ===
a@ferdmbp.local_____________________finished_ok
b@ferdmbp.local_____________________finished_ok

=== Info ===
Updating log files
Updating /Users/ferd/code/self/learn-you-some-erlang/ct/logs/index.html... done
Updating /Users/ferd/code/self/learn-you-some-erlang/ct/logs/all_runs.html... done
Logs in /Users/ferd/code/self/learn-you-some-erlang/ct/logs refreshed!
=== Info ===
Refreshing logs in "/Users/ferd/code/self/learn-you-some-erlang/ct/logs"... ok
[{"dist.spec",ok}]
```

注意，不管测试是否真正执行成功，Common Test 都会显示 ok。这是因为 ct_master 的显

示结果只是表明是否联系上了所有从节点。测试用例的实际运行结果会存储在每个从节点上。

同样请注意，Common Test 还会显示出它用什么 cookie 值启动了从节点。如果在没有停止主节点的情况下再次运行测试，就会显示如下的警告信息：

```
WARNING: Node 'a@ferdmbp.local' is alive but has node_start option
WARNING: Node 'b@ferdmbp.local' is alive but has node_start option
```



这个警告无关紧要。它只是表明 Common Test 能够连接到远程节点，没有必要再去执行测试规格说明文件中的 `init` 元组选项了，因为节点已经处于活动状态。对于 Common Test 来说，它无需亲自启动任何用来运行测试的远程节点，不过一般来讲，这样做还是很有用的。

以上就是分布式规格说明文件的全部要点。当然，你可以弄出一些更加复杂的场景，在其中搭建出更加复杂的集群，并编写一些了不起的分布式测试。但是，测试变得越复杂，你对于它们能够成功验证软件属性的信心就会越低，原因很简单，当测试变得难以理解时，测试本身就可能会包含更多的错误。

28.9 集成 Common Test 和 EUint

因为 EUnit 和 Common Test 各有所长，所以你可能很想把它们集成在一起。

把 Common Test 的测试套件包含到 EUnit 中非常困难，但是反过来却要容易得多。诀窍在于，在调用 `eunit:test(SomeModule)` 时，如果成功就返回 `ok`，如果失败就返回 `error`。

这就意味着，要把 EUnit 的测试集成到 Common Test 中，所要做的只是定义一个如下函数：

```
run_eunit(_Config) ->
    ok = eunit:test(TestsToRun).
```

`TestsToRun` 所描述的 EUnit 测试都会运行。如果出现了失败，结果就会出现在 Common Test 的日志中，可以查看输出的日志确定错误所在。非常简单。

28.10 还有其他内容吗

你肯定认为还有更多的内容。Common Test 确实非常复杂。Common Test 还提供了很多功能：增加针对某些变量的配置文件，增加会在测试执行期间的不同时刻运行的钩子，在测试套件的运行过程中使用它的事件回调函数做些事情，使用模块 `ct_ssh`、`ct_telnet`、`ct_snmp` 和 `ct_ftp` 进行基于协议 SSH、telnet、SNMP 和 FTP 的测试等。

本章仅仅触及了 Common Test 的皮毛，不过已经足以让你进行更深入的探索。Common Test 的用户指南包含了更加完备的内容，它包含在 Erlang/OTP 中（http://www.erlang.org/doc/apps/common_test/users_guide.html）。在本章开头我曾经说过，指南本身比较难以理解，但是理解了本章的内容肯定可以让学习指南变得容易。

第 29 章

Mnesia——记忆的艺术

有个人有很多朋友，你是他最好的朋友。你们有一些共同的朋友，都已经认识很长时间了。这些朋友来自世界各地，从西西里到纽约。他们对你们都很尊敬、关心，你们对他们也一样。

在遇到一些麻烦事时，他们会向你求助，因为你有能力并且讲义气。他们是你的好朋友，所以你得帮忙。但是，帮忙也是有代价的。每帮一次忙，都要及时记录下来，也许在将来的某一天，你会要求他们进行回报。

你始终信守承诺——你们为人可靠。因此，他们称呼你的朋友为*老板*（boss），称呼你为*军师*（consigliere）。你辅佐管理着一个最受尊敬的黑手党家族。

但是，记住所有的朋友关系是一件痛苦的事情，随着你们的影响力蔓延到全世界，要记录哪个朋友欠你的以及你欠了哪个朋友的就变得越来越困难。

你是一个乐于助人的顾问，因此决定对传统的系统进行升级，原先是把信息记录在笔记本中，并藏匿于不同的地方，现在想使用 Erlang 系统完成信息的记录。

一开始，你觉得 ETS 和 DETS 是首选。不过，当你出差海外，不在老板身边时，就很难保持信息的同步。

你可以在 ETS 和 DETS 表之上再编写一个复杂的层，以保持所有信息的一致。这样做没有什么问题，不过作为一个人，你知道自己会犯错，会编写出有 bug 的软件。而对于那些重要的朋友关系，是不允许出现这样的错误的，因此你上网搜索如何确保系统正确工作的方法。

于是，你读到了这一章。在本章中，我们将讲解 Mnesia——一个专门用来解决此类问题的 Erlang 分布式数据库。

29.1 Mnesia 是什么

Mnesia 是位于 ETS 和 DETS（我们在第 25 章做过介绍）之上的一个软件层，为这两个数据存储系统增加了许多新功能。它包含的大多数特性都是需要密集使用 ETS 和 DETS 软件系统所必

需的。Mnesia 可以自动管理 ETS 和 DETS 的写入，它既具有 DETS 的持久化能力，又具有 ETS 的性能，并且还能够自动地在多个 Erlang 节点间进行数据库的复制。

Mnesia 还提供了另外一个有用的特性：事务（transaction）。简单来讲，事务可以让进程在一个或者多个表上执行多个操作，结果就像是只有这个进程在访问这些表一样——ETS 是做不到这一点的。当希望把混合着读写的并发操作当成一个原子执行单元时，这个特性是至关重要的。例如，我们想先读取数据库看看某个用户名是否被占用，如果没有被占用，就创建该用户名。如果没有事务，那么在数据库中查看用户名和注册用户会被认为是两个不同的操作，中间会被打断引起混乱。此时，就可能出现这样一种操作场景，其中不止一个进程会认为自己可以去创建这个唯一的用户名，从而导致大量混乱。事务让多个操作成为一个原子的执行单元，从而解决了这个问题。

Mnesia 是唯一支持原生 Erlang 数据项的存储和查询，且现成可用的全功能数据库（编写本书时）。其缺点是，它沿袭了 DETS 表在某些模式下的一些局限性，例如，磁盘单表不能存储超过 2 GB 的数据（这个缺点可以通过使用分片（fragmentation）的特性予以克服，详见 http://www.erlang.org/doc/apps/mnesia/Mnesia_chap5.html#id75194）。

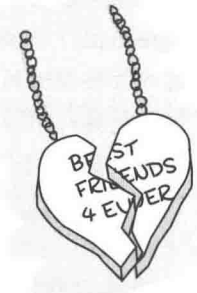
从 CAP 定理（在第 26 章做过介绍）的角度来说，Mnesia 是 CP 的，不是 AP。这意味着，它不支持最终一致性，针对某些网络分区的情况，也应对得不好，但是，如果认为网络是可靠的（有时不能这样认为），那么 Mnesia 就可以提供强一致性保证。

Mnesia 并不是用来取代标准的 SQL 数据库的，也不像 NoSQL 领域的明星产品常常宣称的那样，可以跨多个数据中心处理太字节级别的数据。Mnesia 是用来处理分布在有限几个节点上的小规模数据的。虽然可以在大量节点上使用 Mnesia，不过大部分实践结果表明，节点的上限在 10 个左右。判断 Mnesia 是否适用，可以从这几个方面进行：是否明确知道节点个数，是否明确知道数据规模，数据的访问方式是否与 ETS 和 DETS 在单节点上的方式类似。

Mnesia 和 Erlang 到底有多亲密呢？Mnesia 的核心思想是：用记录来定义表的结构。每个表可以存储一批相似的记录，任何能够放到记录中的东西都可以存储到 Mnesia 表中，包括原子、进程标识、引用等。

29.2 应该存储什么呢

使用 Mnesia 时，首先要弄清楚需要的信息以及存储这些信息的表结构。



29.2.1 需要存储的数据

对于我们的黑手党朋友跟踪应用（我决定给它命名为 `mafiapp`）来说，需要存储的信息包括以下内容。

- 朋友的名字，在请求或者提供帮助时，我们需要知道在和谁谈话。
- 朋友的联系方式，知道如何能够找到朋友。联系方式可以有多种，如电子邮件、手机号码，甚至可以是朋友经常出入的地点。

- 附加信息，如出生年月、职业、习惯、特点等。
- 独特的技能——也就是朋友的专长。把它独立出来是因为我们希望明确地知道这项信息。如果某人擅长烹饪，我们又急于承办一场宴会，那么就知道该找谁了。如果我们身陷麻烦，需要躲避一段时间，那么就可以寻找会开飞机的朋友、易容高手或者优秀的魔术师。

接下来，需要考虑一下要交换的服务。我们想知道哪些信息呢？下面列出一些条目。

- 谁能提供服务。也许是你自己，也就是军师。也许是教父（padrino）。也许是你朋友的朋友。也许是可以成为你朋友的某个人。我们需要知道这个信息。
- 谁接受服务。
- 提供服务的时间。这个信息一般用来作为提醒，尤其是在需要回报时。
- 服务的细节。记录下所提供服务的每一个细节和日期会更好一些（也更令人生畏）。

29.2.2　表结构

我在上一节曾经提到，Mnesia 是基于记录和表（ETS 和 DETS）的。更确切地说，你可以定义一条 Erlang 记录，然后让 Mnesia 把这条定义变成一张表。

例如，如果我们要开发一款食谱应用，并决定把记录定义成如下形式：

```
-record(recipe, {name, ingredients=[], instructions=[]}).
```

接下来，就可以让 Mnesia 创建一个食谱（recipe）表，其中可以存储任意数目的`#recipe{}`记录作为该表的内容。例如，关于披萨饼（pizza）的食谱记录如下：

```
#recipe{name=pizza,
        ingredients=[sauce,tomatoes,meat,dough],
        instructions=["order by phone"]}
```

关于汤羹（soup）的食谱如下：

```
#recipe{name=soup,
        ingredients=["who knows"],
        instructions=["open unlabeled can, hope for the best"]}
```

可以把这两条记录直接插入 recipe 表中。然后，可以从表中原样取出这些记录，并像使用其他任何记录一样使用它们。

主键——具有唯一值的字段，通过它能够以最快的速度对表进行查询——是食谱的名字。这是因为 name 是`#recipe{}`记录定义中的第一个字段。你可能也注意到了，在披萨饼食谱中，配料（ingredients）的类型是原子，在汤羹食谱中，配料的类型则是字符串。和 SQL 表不同，Mnesia 表没有内置类型限制，只要满足表本身的结构即可（Mnesia 表中的所有条目都必须是同类型的记录，也就是具有相同大小、相同头元素的元组）。

那么，该如何表达 mafia 应用中的朋友和服务信息呢？可以把它们放到同一张表中吗？

```
-record(friends, {name,
                  contact=[],
                  info=[],
                  expertise,
                  service=[]}). % 服务数据的格式是{To, From, Date, Description}吗?
```

不过，这不是最佳选择。把服务数据和朋友相关的数据嵌套在一起意味着，在增加和更改服务相关信息时，必须同时更改朋友信息。这很讨厌，尤其是服务一般都关联着至少两个人时，往往会导致更糟糕的情况。对每条服务数据，都需要从数据库取出两条朋友记录数据，并更改它们，即使本来没什么朋友相关的信息需要修改。

还可以采用一种更为灵活的模型，每种数据类型都使用一张表进行存储：

```
-record(mafiapp_friends, {name,
                          contact=[],
                          info=[],
                          expertise}).
-record(mafiapp_services, {from,
                           to,
                           date,
                           description}).
```

两张表的方案可以更灵活地对表进行搜索和修改，开销也不大。

保持冷静

你会注意到，我为 friends 和 services 记录名都增加了 mafiapp_前缀。虽然这两条记录是在模块内部定义的，但是 Mnesia 的表是整个集群中所有节点全局可见的。这意味着，一不小心就会造成名字冲突。因此，手动为表增加命名空间是一个好主意。

29.3 从记录到表

既然已经知道了要存储的数据，接下来就要决定如何存储这些数据。记住，Mnesia 是使用 ETS 和 DETS 表来存储数据的。这样我们就有两种存储选择：存在磁盘中或者存在内存中。我们需要选择一种策略！

可用的存储类型选项如下。

ram_copies

该选项会把数据都存储在 ETS 中，因此全部在内存中。理论上，如果 VM 是以 32 位进行编译的，那么最大内存空间理论上为 4 GB（实际上是 3 GB 多一点），不过在 64 位（再加上 half-word 半字模式）机器上要远远超过这个限制，当然实际可用内存也必须要超过 4 GB。

disc_only_copies

该选项会把数据都存储在 DETS 中，也就是只存储在磁盘上，可存储的规模受限于 DETS 的 2GB 约束。

disc_copies

该选项会把数据同时保存在 ETS 中和 DETS 中，因此既在内存中，又在磁盘上。disc_copies 表不受 DETS 限制的约束，因为 Mnesia 使用了一种复杂的事务日志和检查点（checkpoint）系统，可以对内存表进行磁盘备份。

对于当前的应用，我们使用 disc_copies 选项。朋友关系会持续很长一段时间，因此需要持久保存。如果一觉醒来，发现费尽千辛万苦才输入的朋友关系数据因为一次停电都不见了，就太令人生气了。你也许会问："为何不使用 disc_only_copies 呢？"嗯，把数据也在内存中存一份，通常更适合于进行复杂的查询和搜索操作，因为无需访问硬盘，而硬盘往往是计算机中最慢的部件。

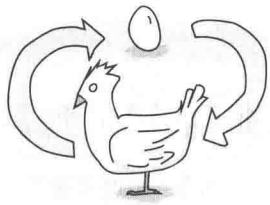
要把我们宝贵的数据存储到数据库中还有件事情要解决。基于 ETS 和 DETS 的工作原理，我们需要定义表的类型。可用的表类型与 ETS 和 DETS 中的完全一样：set，bag 以及 ordered_set。其中，ordered_set 类型不支持 disc_only_copies 表。（所有存储类型的表都不支持 duplicate_bag 类型。）如果忘记了这些类型的含义，可以参考第 25 章的内容。

好消息是，我们基本上决定了如何存储数据。坏消息是，在实际去存储之前，还有一些 Mnesia 的内容需要了解一下。

29.4　Mnesia 数据模式和表

Mnesia 既可以在孤立的节点上运行，也能够很好地支持分布式和多节点复制。为了能够正常工作，Mnesia 需要知道表在磁盘上的存储方式、表的加载方式以及要进行数据同步的节点等信息，这些信息都保存在数据模式（schema）中。

在启动时，Mnesia 会默认在内存中创建一个数据模式。对于只需保存在 RAM 中的表来说，这就足够了。不过，如果数据模式需要经受住 Mnesia 集群中多个节点上 VM 的重启，事情就变得有些复杂了。



Mnesia 依赖于数据模式，而数据模式是由 Mnesia 创建的。这就形成了一个怪圈，要在不运行 Mnesia 的情况下，让 Mnesia 去创建出数据模式！还好，在实际中，这个问题很好解决。只需在启动 Mnesia 之前，调用函数 mnesia:create_schema(ListOfNodes) 即可。这个函数会在每个节点上创建一组文件，其中保存着所有需要的表信息。在调用该函数时，无需先去进行节点连接，但是节点需要运行起来。该函数会去建立连接，并搞定一切。

默认情况下，数据模式文件会保存在当前工作目录中，也就是 Erlang 节点的运行目录。要改变这一点，可以设置 Mnesia 应用的 dir 变量，该变量指定了数据模式的存储位置。你可以使用 erl -name SomeName -mnesia dir where/to/store/the/db 启动节点，或者使用 application:set_env(mnesia, dir, "where/to/store/the/db") 动态配置 dir 变量。

> **注意**　在创建数据模式时可能会失败，有如下几种原因：数据模式已经存在；在要创建数据模式的节点上，Mnesia 正在运行；Mnesia 无法向目录中写入东西，或者发生了其他常见的文件访问问题。

一旦数据模式创建成功，就可以启动 Mnesia，创建数据表了。可以使用函数 mnesia:

create_table/2 创建表。它有两个参数：表名和一个选项列表。一些可用的选项如下：

{attributes, List}

该选项是一个列表，其中包含了表的所有字段。它的默认值为[*key*, *value*]，意味着需要定义形为-*record*(*TableName*, {*key*,*val*})的记录。几乎所有人都会采用一个技巧，用一个特殊的结构（编译器支持的宏）来从记录中提取出字段的名称。该结构看起来像是一个函数调用。如果把该结构应用到朋友记录上，就是{attributes, record_info(fields, mafiapp_friends)}。

{disc_copies, NodeList}、{disc_only_copies, NodeList}和{ram_copies, NodeList}

这些选项用于指定表的存储方式，在 29.3 节中曾经做过介绍。注意，这些选项可以同时使用。例如，你可以定义一个表，通过同时使用这 3 个选项，使其在主节点上既存储在磁盘中，又存储在内存中，在所有的从节点上都只存储在内存中，在一个专门的备份节点上则只存储在磁盘中。

{index, ListOfIntegers}

Mnesia 表在 ETS 和 DETS 功能之上提供了*多级索引*功能。如果打算基于非主键对记录字段进行搜索，这个选项就很有用。例如，我们的朋友表中就需要对 expertise 字段进行索引。我们可以这样来定义该索引：{index, [#mafiapp_friends.expertise]}。一般来讲，索引最好建立在数据值不太重复的记录字段上（这几乎对所有数据库来说都成立）。如果一个表有数十万条记录，而索引却只能把表分类成两个组，那么索引所耗费的大量资源（内存和处理器时间）就非常不划算了。如果一个索引能将同样的表分成 *N* 个组，且每组元素个数少于 10，那么相对于其耗费的资源来讲，效率就很高了。注意，无需对记录中的第一个字段建立索引（元组中的第二个元素），因为已经默认对其建立了索引。

{record_name, Atom}

如果想让表的名称和记录名称不一样，可以使用这个选项。不过，这样做会强迫你使用一些不同的表操作函数，而无法使用那些常用的表操作函数。因此，不推荐使用这个选项，除非真需要它。

{type, Type}

Type 的值可以是 set、ordered_set 或者 bag，它们和 ETS 中的类型完全一样，请参见第 25 章中的介绍。

{local_content, true | false}

在默认情况下，Mnesia 表会把该选项设置为 false。如果想让表和表中的数据在隶属于同一数据模式的所有节点（以及那些在 disc_copies、disc_only_copies 或者 ram_copies 选项中指定的节点）之间复制，那么就不要更改这个默认值。把这个选项设置为 true 仍然会在所有节点上创建出所有的表，但是表的内容存储在本地，不会在节点间共

享。此时，Mnesia 变成了一个在多个节点上初始化同样空表的引擎。

创建 Mnesia 数据模式和表时，需要注意下面一些事项。

- 首次启动 Mnesia 会在内存中创建一个数据模式，这只适用于 ram_copies 类型的表，不适用于其他类型的表。
- 如果在启动 Mnesia 前（或者在停止它之后），手动创建一个数据模式，就可以创建基于磁盘的表了。
- 要先启动 Mnesia，接着才能创建表。不能在 Mnesia 没有运行的情况下创建表。

注意　还有另外一种表操作方法。如果某个 Mnesia 节点正在运行，而你想把已经创建的表存储到磁盘上，可以调用函数 mnesia:change_table_copy _type(Table, Node, NewType) 完成这项功能。特别地，如果忘了把数据模式创建到磁盘上，那么调用 mnesia:change_table_copy_type(schema, node(), disc_copies) 可以把 RAM 数据模式变成基于磁盘的数据模式。

现在可以开始我们的应用，并实际操作一下 Mnesia 了。

29.5　创建表

我们将使用 Common Test，以弱测试驱动开发（TDD）的风格创建应用和表。也许你不喜欢测试驱动开发，不过别担心，我们会以宽松的方式进行，只是用它来引导我们的设计，不会出现任何“要确保测试首先失败”之类的事情（当然，你可以这样做）。所产生的测试只是一个良好的副产品，并不是我们的目标。我们主要关注于 mafiapp 应用接口的行为和外观定义，我们不想只通过 Erlang shell 来完成这项工作。虽然测试中没有涉及任何分布式，但是这个例子仍然是一个很好的机会，可以在学习 Mnesia 的同时，实践一下 Common Test。

针对这个例子，创建一个名为 mafiapp-1.0.0 的目录，目录结构遵循标准的 OTP 风格：

```
ebin/
logs/
src/
test/
```

29.5.1　安装数据库

我们先来考虑一下如何安装数据库。因为在使用数据库之前，要先创建数据模式和表，所以在所有测试的初始化函数中会调用一个安装函数。在正常情况下，这个安装函数会在 Common Test 的 priv_dir 目录中创建数据模式和表。我们先来写一个简单的测试套件 mafiapp_SUITE，把它存放在 test/目录中：

```
-module(mafiapp_SUITE).
-include_lib("common_test/include/ct.hrl").
-export([init_per_suite/1, end_per_suite/1,
         all/0]).
all() -> [].
```

```
init_per_suite(Config) ->
    Priv = ?config(priv_dir, Config),
    application:set_env(mnesia, dir, Priv),
    mafiapp:install([node()]),
    application:start(mnesia),
    application:start(mafiapp),
    Config.

end_per_suite(_Config) ->
    application:stop(mnesia),
    ok.
```

这个测试套件中目前还没有任何测试，不过它给出了该如何做事的初步规格说明。首先，通过把 dir 变量的值设置为 priv_dir，指定了存放 Mnesia 的数据模式和数据库文件的位置。这样，每个数据库和模式实例都会被放置到由 Common Test 产生的私有目录中，确保多次测试运行之间不会产生问题和冲突。

我们把安装函数命名为 install，它的参数是需要进行安装的节点列表。采用列表类型参数的方法比把节点名硬编码到 install 函数中要好一些，因为更灵活一些。调用完 install 之后，要启动 Mnesia 和 mafiapp 应用。

现在，我们可以编写文件 src/mafiapp.erl 了，看看如何实现 install 函数。首先要把前面定义的记录放到文件中：

```
-module(mafiapp).
-export([install/1]).

-record(mafiapp_friends, {name,
                          contact=[],
                          info=[],
                          expertise}).
-record(mafiapp_services, {from,
                           to,
                           date,
                           description}).
```

看起来不错。install 函数的实现如下：

```
install(Nodes) ->
    ok = mnesia:create_schema(Nodes),
    application:start(mnesia),
    mnesia:create_table(mafiapp_friends,
                        [{attributes, record_info(fields, mafiapp_friends)},
                         {index, [#mafiapp_friends.expertise]},
                         {disc_copies, Nodes}]),
    mnesia:create_table(mafiapp_services,
                        [{attributes, record_info(fields, mafiapp_services)},
                         {index, [#mafiapp_services.to]},
                         {disc_copies, Nodes},
                         {type, bag}]),
    application:stop(mnesia).
```

首先，我们在 Nodes 列表中指定的节点上创建数据模式。然后，启动 Mnesia，这是创建表所必需的。我们创建了两个表，分别以记录 #mafiapp_friends{}和 #mafiapp_services{}命名。为了能够用专长来搜索朋友，我们在 expertise 字段上创建了一个索引。

你还会注意到，服务表的类型是 bag。这是因为可能存在具有相同提供者和接受者的多个服务。如果表使用 set 类型，那么就只能有唯一一个提供者，而 bag 类型的表则可以处理这种情况。我们还在表的 to 字段上建立了索引。这是因为我们希望既可以基于服务提供者，也可以基于服务接受者进行服务查询，所以建立两个索引可以让基于任何一个字段的查询都更高效。

最后要注意的是，在表创建完成后，停止了 Mnesia。这只是为了满足我在测试中指定的行为要求。测试代码给出了我所期望的代码使用方式，因此最好让代码满足这个方式。在安装完成后，继续保持 Mnesia 运行是完全没有问题的。

现在，如果我们的 Common Test 测试套件中有测试用例，就会调用这个 install 函数顺利完成初始化过程。不过，当有多个节点时，运行测试会在 Erlang shell 中产生失败的消息。你知道这是为什么吗？安装的流程如下所示：

```
Node A                          Node B
------                          ------
create_schema ------------>  create_schema
start Mnesia
creating table ----------> ???
creating table ----------> ???
stop Mnesia
```

Mnesia 必须在所有节点上都运行时才能在所有节点上成功创建表。但是在创建数据模式时，却要求 Mnesia 不能在任何节点上运行。理想情况下，我们希望可以远程启动和停止 Mneisa。好消息是，可以做到这点。还记得第 26 章介绍的 rpc 模块吗？我们可以调用函数 rpc:multicall(Nodes, Module, Function, Args)来完成这项任务。我们把 install/1 函数的实现改成下面这样：

```
install(Nodes) ->
    ok = mnesia:create_schema(Nodes),
    rpc:multicall(Nodes, application, start, [mnesia]),
    mnesia:create_table(mafiapp_friends,
                        [{attributes, record_info(fields, mafiapp_friends)},
                         {index, [#mafiapp_friends.expertise]},
                         {disc_copies, Nodes}]),
    mnesia:create_table(mafiapp_services,
                        [{attributes, record_info(fields, mafiapp_services)},
                         {index, [#mafiapp_services.to]},
                         {disc_copies, Nodes},
                         {type, bag}]),
     rpc:multicall(Nodes, application, stop, [mnesia]).
```

使用 rpc，我们可以控制所有节点上 Mnesia 的运行和停止。现在的运行情况如下所示：

```
Node A                          Node B
------                          ------
create_schema ------------> create_schema
start Mnesia -------------> start Mnesia
creating table -----------> replicating table
creating table -----------> replicating table
stop Mnesia --------------> stop Mnesia
```

不错，非常不错。

29.5.2 启动应用

接下来，我们要关注一下 `init_per_suite/1` 函数中启动 `mafiapp` 应用的部分。实际上，我们不需要这样做，因为我们的整个应用完全基于 Mnesia。启动了 Mnesia 就等于启动了我们的应用。不过，在 Mnesia 启动完成和从磁盘上加载完所有表之间有一个明显的延迟，尤其是当表比较大时，这个延迟也更长。对于这种情况，定义一个 `mafiapp:start/2` 函数，即使我们不需要像通常的实现那样创建进程，把这个等待放入其中也是一个非常不错的选择。

mafiapp.erl 实现了应用行为(`-behavior(application).`)，它实现如下两个回调函数(别忘了把它们导出)：

```
start(normal, []) ->
    mnesia:wait_for_tables([mafiapp_friends,
                            mafiapp_services], 5000),
    mafiapp_sup:start_link().

stop(_) -> ok.
```

关键在于 `mnesia:wait_for_tables(TableList, TimeOut)` 函数。该函数调用会至多等待 5 s（这个值可以任意设定，可以把它更改为你觉得合适的值），如果提前加载好，就立即返回。

下面这段代码并没有显示出监督者的行为方式，因为 `mafiapp_sup` 根本没做什么工作：

```
-module(mafiapp_sup).
-behaviour(supervisor).
-export([start_link/0]).
-export([init/1]).

start_link() ->①
    supervisor:start_link(?MODULE, []).

%% 该函数没做任何工作，只是为了完成表加载等待
init([]) ->
    {ok, {{one_for_one, 1, 1}, []}}.
```

监督者什么也没做，不过因为 OTP 应用的启动是同步的，所以这里确实是放置这种同步点的最佳选择。

① 原书中 `start_link` 有一个参数，而上面应用 `start` 回调函数中没有参数，所以修改使之一致。——译者注

最后，把下面的 mafiapp.app 文件放到 ebin/目录中，以确保应用可以被启动起来：

```
{application, mafiapp,
 [{description, "Help the boss keep track of his friends"},
  {vsn, "1.0.0"},
  {modules, [mafiapp, mafiapp_sup]},
  {applications, [stdlib, kernel, mnesia]}]}.
```

现在，万事俱备，我们可以去编写真正的测试并实现我们的应用了。真的吗？

29.6　数据表存取上下文

在动手实现应用之前，我们先来看看 Mnesia 是如何操作表的。

对 Mnesia 表的所有更改，包括读取操作，都必须在一个*活动访问上下文*（activity access context）中进行。这些上下文代表着不同类型的事务，或者不同的查询运行方法。可供选择的上下文类型如下。

transaction

Mnesia 事务可以把一组数据库操作当成单个功能操作块。整个功能操作块要么在所有节点上运行，要么在所有节点上都不运行，要么全部成功，要么全部失败。当事务返回时，可以保证表处于一致的状态，不同的事务，即使它们试图操作同样的数据，之间也不会相互干扰。

这种类型的活动上下文是部分异步的。对于本地节点的操作它是同步的，但是对于远程节点，它只会等待远程节点即将提交事务的确认消息，而不是等到它们真正提交完事务。对于 Mnesia 来说，如果事务在本地执行成功，并且其他节点也都同意提交，那么就认为它在所有节点上都会成功。如果实际上并没有全部成功，可能是因为网络或者硬件的故障，那么在稍后的某个时间点会回滚这个事务。之所以定义出容忍这种情况的协议是出于效率方面的考虑，不过协议在某个成功的事务以后回滚时会给你一个确认。

sync_transaction

该活动上下文和 transaction 上下文基本完全一样，不过它是完全同步的。如果 transaction 所提供的保证不能完全满足要求，你不喜欢某个事务由于一些奇怪原因失败了却声称自己是成功的，尤其是你希望在事务成功时，去做一些具有副作用（例如，通知外部服务、创建进程等）的操作，此时 sync_transaction 正是你所需要的。同步事务会一直等待收到所有其他节点的最终确认时才返回，确保事情 100%完成。

还有一个有趣的使用场景，如果你运行了非常多的事务——多到足以使得其他节点过载——那么把事务切换到同步模式可以迫使事务发起的速度变慢，减少事务的累积，把过载的问题推升到应用的更高一级处理。

async_dirty

async_dirty 活动上下文基本上完全绕开了所有的事务协议和锁操作（虽然在继续处理前也会等待所有活动事务完成）。不过，所有的日志、复制等工作都会照常进行。

async_dirty 活动上下文会尝试在本地进行所有的操作，然后返回，向其他节点的复制工作都异步进行。

sync_dirty

该活动上下文和 async_dirty 的区别与 sync_transaction 和 transaction 的区别一样。它会等到远程节点工作完成的确认，不过仍然绕开了所有的锁操作和事务上下文。一般来讲，脏（dirty）上下文要比事务上下文快不少，不过设计风险也更高一些。使用它们时要小心。

ets

ets 活动上下文完全绕开了 Mnesia 所做的一切，直接针对底层的 ETS 表进行操作。它不会进行任何复制操作。通常情况下，无需使用 ets 活动上下文。它同样适用于这个原则："需要使用它时，自然会知道，如果有疑惑，不要使用它。"

上面列出的就是可以在其中运行常用 Mnesia 操作的所有上下文。可以把操作封装到 fun 中，并调用 mnesia:activity(Context, Fun) 执行。fun 中可以包含任何 Erlang 函数调用，但是要注意，在出现失败或者被其他事务打断的情况下，事务是可能被执行多次的。

这就意味着，如果一个事务从表中读取一个值，并在向表中写回某些值之前发送了一条消息，那么完全有可能出现该消息被发送多次的情况。因此，*不要在事务中包含任何具有这种副作用的操作。*

29.7 读、写以及其他操作

在前面，我们曾经多次提到过表的修改函数，现在来定义它们。毫无疑问，大部分表操作函数都与 ETS 和 DETS 中的相似。

write

调用 mnesia:write(Record) 函数，可以把记录 Record 插入和记录同名的表中，如果表的类型是 set 或者 ordered_set，且存在主键（从元组的视角来看，主键是记录的第二个字段，不是其名字）相同的元素，则该元素会被替换掉。对于 bag 类型的表来说，只有整条记录完全相同，才会被替换。

如果写操作执行成功，write/1 会返回 ok。否则，它会抛出一个异常，并中止事务。抛出异常的情况不会经常发生。大部分的异常情况是 Mneisa 没有运行、找不到表或者记录无效。

delete

该函数的调用方式为 mnesia:delete(TableName, Key)。键值相同的记录会从表中删除。该函数或者返回 ok，或者抛出异常，语义和 mnesia:write/1 相似。

read

该函数的调用方式为 mnesia:read({TableName, Key})，该函数会返回一个主键和 Key 匹配的记录列表。和 ets:lookup/2 类似，即使对于至多只有一个匹配结果的 set

类型的表，它也始终返回一个列表。如果没有任何记录匹配，会返回一个空列表。与 delete 和 write 操作一样，失败时会抛出异常。

match_object

该函数和 ETS 中的 match_object 函数类似。它使用类似第 25 章中介绍的模式，从数据库表中返回完整的记录。例如，根据某项专长查找朋友的一种快捷方法为 mnesia:match_object(#mafiapp_friends {_ = '_', expertise = given})。这个调用会返回表中所有匹配记录的列表。同样，失败时会抛出异常。

select

该函数和 ETS 中的 select 函数类似。它使用匹配规格说明或者 ets:fun2ms 进行查询。（如果忘记了相关的细节，请复习一下 25.5 节，其中介绍了匹配规格说明。）该函数的调用方式为 mnesia:select(TableName, MatchSpec)，这个调用会返回所有满足匹配规格说明的记录列表。同样，在失败时，该函数会抛出异常。

还有其他一些 Mnesia 表操作函数。不过，前面介绍的这些已经足够我们使用了。如果你对其他操作感兴趣，可以去学习 Mnesia 的参考手册。其中有像 first、last、next 以及 prev 等用于单次迭代的函数，也有像 foldl 和 foldr 等在整个表上进行折叠（fold）操作的函数，还有一些对表本身进行操作的函数，你可能会感兴趣，如 transform_table（在向记录或者表中增加或者删除字段时，该函数尤为有用）和 add_table_index。

对函数的介绍已经够多了。我们再来写点测试，看看如何在实际中使用这些函数。

29.8　实现第一个请求

现在，我们来编写几个向 mafia 应用中增加数据以及基于这些数据进行朋友和服务查询的测试用例。

29.8.1　测试增加服务

在实现请求之前，我们将先编写一个简单的测试，说明一下我们希望的应用行为。该测试是关于增加服务的，不过也隐含着一些对其他功能的测试。

我们以标准的初始化操作开始，这是大多数 Common Test 测试套件都需要的。

```
...
-export([init_per_suite/1, end_per_suite/1,
         init_per_testcase/2, end_per_testcase/2,
         all/0]).
-export([add_service/1]).

all() -> [add_service].
...

init_per_testcase(add_service, Config) ->
   Config.

end_per_testcase(_, _Config) ->
```

```
    ok.
```

下面是测试用例：

```
%% 服务是双向的：从朋友到boss或者从boss到朋友
%% 必须要有一个boss朋友
add_service(_Config) ->
    {error, unknown_friend} = mafiapp:add_service("from name",
                                                  "to name",
                                                  {1946,5,23},
                                                  "a fake service"),
    ok = mafiapp:add_friend("Don Corleone", [], [boss], boss),
    ok = mafiapp:add_friend("Alan Parsons",
                            [{twitter,"@ArtScienceSound"}],
                            [{born, {1948,12,20}},
                             musician, 'audio engineer',
                             producer, "has projects"],
                            mixing),
    ok = mafiapp:add_service("Alan Parsons", "Don Corleone",
                             {1973,3,1}, "Helped release a Pink Floyd album").
```

因为我们是在增加服务，所以应该把涉及服务交换的两个朋友都添加进来。我们使用函数mafiapp:add_friend(Name, Contact, Info, Expertise)完成朋友的添加。添加完朋友后，就可以增加服务了。

> **注意** 如果你看过其他 Mnesia 教程，就会发现有些人非常喜欢在函数中直接使用记录（如mafiapp:add_friend (#mafiapp_friend{name=...})）。我会避免这样的用法，因为最好把记录保持私有。实现层面的变化可能会破坏记录的底层表达方式。这本身不是问题，不过一旦更改了记录的定义，就需要重新编译，甚至要原子化地升级一个正在运行的应用中所有使用该记录的模块，否则它们将无法继续工作。简单地把记录封装在函数中，提供了一个更加干净的接口，这样，使用数据库或者应用的模块就不需要通过.hrl 文件去包含记录定义，坦白地讲，包含.hrl 文件是很令人讨厌的。

你应该注意到，在刚刚编写的测试中，我们并没有查找服务。这是因为我们将在查找用户时再去搜索服务。现在，我们使用 Mnesia 事务来实现测试要求的功能。在 mafiapp.erl 中编写的第一个函数是向数据库中增加用户：

```
add_friend(Name, Contact, Info, Expertise) ->
    F = fun() ->
        mnesia:write(#mafiapp_friends{name=Name,
                                      contact=Contact,
                                      info=Info,
                                      expertise=Expertise})
    end,
    mnesia:activity(transaction, F).
```

我们定义了一个向数据库中写入#mafiapp_friends{}记录的函数。这是一个简单的事务。

add_service/4 函数稍微复杂些：

```
add_service(From, To, Date, Description) ->
    F = fun() ->
        case mnesia:read({mafiapp_friends, From}) =:= [] orelse
             mnesia:read({mafiapp_friends, To}) =:= [] of
            true ->
              {error, unknown_friend};
            false ->
              mnesia:write(#mafiapp_services{from=From,
                                             to=To,
                                             date=Date,
                                             description=Description})
          end
      end,
      mnesia:activity(transaction,F).
```

在这个事务中，首先会进行一次或者两次读取操作，看看数据库中是否已经存在服务涉及的朋友。如果其中有任何一个朋友不存在，就返回测试要求的元组{error, unknown_friend}。如果两个朋友都存在，就把服务写入数据库中。

> **注意** 对输入数据的有效性验证留给你来完成。验证代码和其他Erlang程序的编写方式类似。如果可能的话，将尽量多的验证代码放到事务上下文的外面是一个不错的方法，因为事务中的代码可能会多次运行，并且会竞争数据库资源。

有了这些函数，应该可以运行我们的测试了。我们使用下面的测试规格说明文件 mafiapp.sepc（放置在工程的根目录中）：

```
{alias, root, "./test/"}.
{logdir, "./logs/"}.
{suites, root, all}.
```

我们还需要如下的 Emakefile 文件（也在根目录中）：

```
{["src/*", "test/*"],
 [{i,"include"}, {outdir, "ebin"}]}.
```

接下来，就可以运行测试了：

```
$ erl -make
Recompile: src/mafiapp_sup
Recompile: src/mafiapp
$ ct_run -pa ebin/ -spec mafiapp.spec
... <snip> ...
Common Test: Running make in test directories...
Recompile: mafiapp_SUITE
... <snip> ...
Testing learn-you-some-erlang.mafiapp: Starting test, 1 test cases
... <snip> ...
Testing learn-you-some-erlang.mafiapp: TEST COMPLETE, 1 ok, 0 failed of 1 test
```

```
cases
... <snip> ...
```

不错，测试通过了。很好。我们来编写下一组测试。

> **注意** 在运行 Common Test 测试套件时，可能会出现一些找不到目录的错误。解决方案是，使用 ct_run -pa ebin/或者 erl -name ct -pa 'pwd'/ebin（或者全路径）。虽然启动 Erlang shell 会使当前工作目录成为节点的当前工作目录，但是调用 ct:run_test/1 会把当前工作目录更改到一个新的位置。这会让./ebin/这样的相对路径失效。使用绝对路径可以解决这个问题。

29.8.2 测试查询

add_service/1 测试让我们测试了增加朋友和服务的操作。接下来的测试会关注于信息的查询。方便起见，我们会把 boss 增加到后续所有的测试用例中：

```
init_per_testcase(add_service, Config) ->
    Config;
init_per_testcase(_, Config) ->
    ok = mafiapp:add_friend("Don Corleone", [], [boss], boss),
    Config.
```

我们将关注的用例是，通过名字来查找朋友。虽然只查找服务也能满足要求，但是在实际中可能更希望通过名字而不是行为去查找人。boss 基本上不会这样问："谁曾经为谁弹奏过一曲吉他？"他更可能会这样问："谁曾经给我们的朋友 Pete Cityshend 弹奏过一曲吉他？"并通过这个朋友的名字去查询他的历史记录以找出服务的细节。因此，下一个测试是 friend_by_name/1：

```
-export([add_service/1, friend_by_name/1]).

all() -> [add_service, friend_by_name].
...
friend_by_name(_Config) ->
    ok = mafiapp:add_friend("Pete Cityshend",
                            [{phone, "418-542-3000"},
                             {email, "quadrophonia@example.org"},
                             {other, "yell real loud"}],
                            [{born, {1945,5,19}},
                             musician, popular],
                            music),
    {"Pete Cityshend",
     _Contact, _Info, music,
     _Services} = mafiapp:friend_by_name("Pete Cityshend"),
    undefined = mafiapp:friend_by_name(make_ref()).
```

这个测试验证了我们可以插入一个朋友，并对其进行查询，以及在查询不到时应该返回什么值。查询成功时，会返回一个包含所有细节信息的元组结构，其中也包含我们暂时无需关注的服务信息。我们主要想找人，不过，那些多余的信息会让测试更加严格一些。

我们可以用一个 Mnesia 读取操作来实现 mafiapp:friend_by_name/1 函数。#mafiapp_friends{}记录定义中把朋友名当成表的主键（记录中定义的第一个字段）。使用 mnesia:read({Table, Key})函数可以使工作变得简单，只需对其进行很少的封装就可以满足测试的要求：

```
friend_by_name(Name) ->
    F = fun() ->
        case mnesia:read({mafiapp_friends, Name}) of
            [#mafiapp_friends{contact=C, info=I, expertise=E}] ->
                {Name,C,I,E,find_services(Name)};
            [] ->
                undefined
        end
    end,
    mnesia:activity(transaction, F).
```

只要有了这一个函数，就足以让测试通过了。当然，别忘了导出它。我们现在不关心 find_services(Name)函数，因此先给它打个桩：

```
%%% 私有函数
find_services(_Name) -> undefined.
```

做完这些后，新的测试应该可以通过：

```
$ erl -make
... <snip> ...
$ ct_run -pa ebin/ -spec mafiapp.spec
... <snip> ...
Testing learn-you-some-erlang.wiptests: TEST COMPLETE, 2 ok, 0 failed of 2
test cases
... <snip> ...
```

最好能测试一下查询请求中返回更多的服务信息细节的情况。下面是相应的测试：

```
-export([add_service/1, friend_by_name/1, friend_with_services/1]).

all() -> [add_service, friend_by_name, friend_with_services].
...
friend_with_services(_Config) ->
    ok = mafiapp:add_friend("Someone", [{other, "at the fruit stand"}],
                            [weird, mysterious], shadiness),
    ok = mafiapp:add_service("Don Corleone", "Someone",
                             {1949,2,14}, "Increased business"),
    ok = mafiapp:add_service("Someone", "Don Corleone",
                             {1949,12,25}, "Gave a Christmas gift"),
    %% 我们在这个测试中不关心列表中的服务顺序，所以就按照函数的实际返回来写测试的匹配结果
    {"Someone",
     _Contact, _Info, shadiness,
     [{to, "Don Corleone", {1949,12,25}, "Gave a Christmas gift"},
      {from, "Don Corleone", {1949,2,14}, "Increased business"}]} =
     mafiapp:friend_by_name("Someone").
```

在这个测试中，Don Corleone 为某个身份不明的人提供了一个水果摊位以扩展其业务。而这个身份不明的人没有忘记这个恩惠，随后回赠了一份圣诞节礼物给 boss。

从这个测试中可以看出，我们仍然使用了函数 `friend_by_name/1` 进行查找。虽然该测试过于通用，也不太完整，但是仍然可以从中看出我们想要做的事情。还好，现在没有维护性方面的要求，可以接受这种不完整。

函数 `find_service/1` 的实现要稍微复杂一些。函数 `friend_by_name/1` 仅通过主键的查询就能够完成所需的功能，但是对于服务来说，仅当搜索 `from` 字段时，才可以把朋友名字当作主键使用。我们还需要基于 `to` 字段进行搜索。处理这种情况有多种方法，例如多次调用 `match_object` 函数，或者先读取整个表的内容，然后手工过滤。我选择使用匹配规格说明和 `ets:fun2ms` 解析转换：

```
-include_lib("stdlib/include/ms_transform.hrl").
...
find_services(Name) ->
    Match = ets:fun2ms(
            fun(#mafiapp_services{from=From, to=To, date=D, description=Desc})
                when From =:= Name ->
                  {to, To, D, Desc};
               (#mafiapp_services{from=From, to=To, date=D, description=Desc})
               when To =:= Name ->
                  {from, From, D, Desc}
            end
    ),
     mnesia:select(mafiapp_services, Match).
```

匹配规格说明中有两条子句：当 `From` 匹配 `Name` 时，返回`{to, ToName, Date, Description}`元组。当`Name`匹配`To`时，返回`{from, FromName, Date, Description}`元组，这样就可以用单个操作同时查询到提供和接受的服务。

注意，函数 `find_services/1` 没有运行在任何事务中。这是因为该函数只会在 `friend_by_name/1` 函数中调用，而这个函数已经运行在事务中了。Mnesia 可以支持嵌套事务，不过在这种情况下无需这样做。

> **注意** 使用 Mnesia 编写较大规模的应用时，可以考虑在操作 Mnesia 中存储数据的代码和实际运行这些操作的代码（使用 `mnesia:activity/2`）。这样，就能够任意组合这些操作，并让调用代码决定是否使用同步、异步事务或者其他的上下文。

再次运行测试，3 个测试用例全部成功通过。

最后一个用例是基于专长搜索朋友。下面的测试用例说明当我们需要一个攀登高手来完成某项任务时，怎样找到我们的朋友 Red Panda：

```
-export([add_service/1, friend_by_name/1, friend_with_services/1,
         friend_by_expertise/1]).

all() -> [add_service, friend_by_name, friend_with_services,
          friend_by_expertise].
...
friend_by_expertise(_Config) ->
    ok = mafiapp:add_friend("A Red Panda",
                            [{location, "in a zoo"}],
                             [animal,cute],
                             climbing),
    [{"A Red Panda",
      _Contact, _Info, climbing,
      _Services}] = mafiapp:friend_by_expertise(climbing),
    [] = mafiapp:friend_by_expertise(make_ref()).
```

要实现这个功能，需要基于非主键从数据表中读取记录。我们仍旧可以使用匹配规格说明，不过我们已经用过了，并且我们只需匹配一个字段。函数 mnesia:match_object/1 非常适合这种情况：

```
friend_by_expertise(Expertise) ->
    Pattern = #mafiapp_friends{_ = '_',
                               expertise = Expertise},
    F = fun() ->
        Res = mnesia:match_object(Pattern),
        [{Name,C,I,Expertise,find_services(Name)} ||
          #mafiapp_friends{name=Name,
                           contact=C,
                           info=I} <- Res]
    end,
    mnesia:activity(transaction, F).
```

在上面的代码中，我们先定义了匹配模式。我们需要使用_='_'，也就是用一个匹配一切的符号（'_'）去表示所有未给出的记录字段值。否则，match_object/1 函数只会去寻找那些除 expertise 字段外，其他字段值均为 undefined 的记录。

得到结果后，我们把记录格式化成一个满足测试要求的元组。再次编译和运行测试，可以看到该实现满足测试要求。万岁！我们实现了所有的需求！

29.8.3　账目和新的需求

任何软件项目都不会真正结束。使用软件系统的用户要么会提出新的需求，要么会以没有预见到的使用方式让软件崩溃。我们的老板，甚至在还没有使用这个全新的软件系统前就要求增加一个新的特性：可以快速地遍历所有的朋友，看看谁亏欠我们，以及我们亏欠谁。

下面是关于这项新特性的测试：

```
...
init_per_testcase(accounts, Config) ->
    ok = mafiapp:add_friend("Consigliere", [], [you], consigliere),
```

```
    Config;
...
accounts(_Config) ->
    ok = mafiapp:add_friend("Gill Bates", [{email, "ceo@macrohard.com"}],
                            [clever,rich], computers),
    ok = mafiapp:add_service("Consigliere", "Gill Bates",
                            {1985,11,20}, "Bought 15 copies of software"),
    ok = mafiapp:add_service("Gill Bates", "Consigliere",
                            {1986,8,17}, "Made computer faster"),
    ok = mafiapp:add_friend("Pierre Gauthier", [{other, "city arena"}],
                             [{job, "sports team GM"}], sports),
    ok = mafiapp:add_service("Pierre Gauthier", "Consigliere", {2009,6,30},
                            "Took on a huge, bad contract"),
    ok = mafiapp:add_friend("Wayne Gretzky", [{other, "Canada"}],
                            [{born, {1961,1,26}}, "hockey legend"],
                            hockey),
    ok = mafiapp:add_service("Consigliere", "Wayne Gretzky", {1964,1,26},
                            "Gave first pair of ice skates"),
%% Wayne Gretzky 亏欠我们，因此债务是负值。
%% Gill Bates 服务和债务情况正好抵消。
%% 我们欠 Gauthier 一个服务。
[{-1,"Wayne Gretzky"},
 {0,"Gill Bates"},
 {1,"Pierre Gauthier"}] = mafiapp:debts("Consigliere"),
[{1, "Consigliere"}] = mafiapp:debts("Wayne Gretzky").
```

我们增加了 3 个用于测试的朋友：Gill Bates、 Pierre Gauthier 以及冰球名人堂的 Wayne Gretzky。在这 3 个人和你（consigliere）之间，都有一个服务交换。（在这个测试中，我们没有使用 boss，因为它已经被用于其他测试了，如果用在这里，会导致结果混乱!）

函数 mafiapp:debts(Name) 以朋友名字为参数，统计所有涉及该名字的服务个数。当某人欠我们一个服务时，统计值为负数。当正好平衡时，值为 0。当我们欠其他人一个服务时，值为 1。因此，我们可以认为 debt/1 函数的返回值是欠其他人的服务个数。

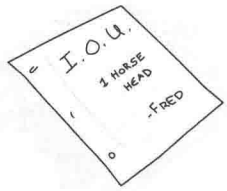
这个函数的实现更加复杂一些：

```
-export([install/1, add_friend/4, add_service/4, friend_by_name/1,
         friend_by_expertise/1, debts/1]).
...
debts(Name) ->
    Match = ets:fun2ms(
        fun(#mafiapp_services{from=From, to=To}) when From =:= Name ->
            {To,-1};
           (#mafiapp_services{from=From, to=To}) when To =:= Name ->
            {From,1}
        end),
    F = fun() -> mnesia:select(mafiapp_services, Match) end,
    Dict = lists:foldl(fun({Person,N}, Dict) ->
                          dict:update(Person, fun(X) -> X + N end, N, Dict)
                       end,
                       dict:new(),
```

```
                    mnesia:activity(transaction, F)),
    lists:sort([{V,K} || {K,V} <- dict:to_list(Dict)]).
```

Mnesia 的查询变得越复杂，就越可能使用匹配规格说明来解决问题。匹配规格说明具有运行简单 Erlang 函数的能力，在需要指定生成的结果时，这种能力会非常有用。在这个函数中，我们使用匹配规格说明来进行这样的操作：当服务的提供者的名字是 Name 时，返回-1（我们提供服务，其他人欠我们一个服务），当 Name 匹配 To 时，返回 1（我们接受一个服务，我们欠别人一个服务）。在这两种情况下，返回值都是一个元组，元组的另外一个元素是 Name。

把名字包含进来是接下来的计算所必需的，其中要统计为每个人提供的服务个数，并给出一个唯一的累加值。同样地，有多种计算这个值的方法。我选择了一种在事务中停留时间最短，并且和数据库代码隔离程度最大的方法，这样就可以运行更多的事务。这对于 mafiapp 应用也许没有什么意义，但是在需要高性能的情况下，这种做法可以在很大程度上减少资源的争用。



我所选择的解决方案是：拿到所有的值，把它们放到字典中，使用函数 dict:update(Key, Operation, Dict)，根据服务提供的方向进行计数的增加或者减少。把这个操作放到一个针对 Mnesia 返回值列表的折叠（fold）操作中，我们可以得到一个列表，其中包含着所有需要的结果。

最后一步是对值进行翻转（从{Key, Debt}到{Debt, Key}），并基于翻转的结果进行排序，这样就得到了所期望的结果。

29.9　满足老板

我们的软件至少应该在生产环境中试用一次。为了试用，我们将先设置好老板要使用的节点，然后再设置好军师使用的节点。

```
$ erl -name corleone -pa ebin/
```

```
$ erl -name genco -pa ebin/
```

这两个节点启动完成后，可以把它们连接起来，并安装应用：

```
(corleone@ferdmbp.local)1> net_kernel:connect('genco@ferdmbp.local').
true
(corleone@ferdmbp.local)2> mafiapp:install([node()|nodes()]).
{[ok,ok],[]}
(corleone@ferdmbp.local)3>
=INFO REPORT==== 8-Apr-2013::20:02:26 ===
    application: mnesia
    exited: stopped
    type: temporary
```

接下来，调用 application:start(mnesia)和 application:start(mafiapp)，在这两个节点上启动 Mnesia 和 mafiapp。启动完成后，调用函数 mnesia:system_info()来检查一下是否一切正常，该函数会显示关于整个初始化的状态信息：

```
(genco@ferdmbp.local)2> mnesia:system_info().
===> System info in version "4.7", debug level = none <===
opt_disc. Directory "/Users/ferd/.../Mnesia.genco@ferdmbp.local" is used.
use fallback at restart = false
running db nodes   = ['corleone@ferdmbp.local','genco@ferdmbp.local']
stopped db nodes   = []
master node tables = []
remote             = []
ram_copies         = []
disc_copies        = [mafiapp_friends,mafiapp_services,schema]
disc_only_copies   = []
[{'corleone@...',disc_copies},{'genco@...',disc_copies}] = [schema,
mafiapp_friends,
mafiapp_services]
5 transactions committed, 0 aborted, 0 restarted, 2 logged to disc
0 held locks, 0 in queue; 0 local transactions, 0 remote
0 transactions waits for other nodes: []
yes
```

可以看到，两个节点都在运行数据库节点列表（running database nodes）中，并且表和数据模式都同时写入了磁盘和 RAM（disc_copies）。现在可以进行数据库的读写操作了。当然，首先把 Don Corleone 写入数据库是一个不错的选择：

```
(corleone@ferdmbp.local)4> ok = mafiapp:add_friend("Don Corleone", [], [boss], boss).
ok
(corleone@ferdmbp.local)5> mafiapp:add_friend(
(corleone@ferdmbp.local)5>     "Albert Einstein",
(corleone@ferdmbp.local)5>     [{city, "Princeton, New Jersey, USA"}],
(corleone@ferdmbp.local)5>     [physicist, savant,
(corleone@ferdmbp.local)5>         [{awards, [{1921, "Nobel Prize"}]}]],
(corleone@ferdmbp.local)5>     physicist).
ok
```

很好，我们在 corleone 节点增加了朋友信息。现在，在 genco 节点增加一个服务试试：

```
(genco@ferdmbp.local)3> mafiapp:add_service("Don Corleone",
(genco@ferdmbp.local)3>                     "Albert Einstein",
(genco@ferdmbp.local)3>                     {1905, '?', '?'},
(genco@ferdmbp.local)3>                     "Added the square to E = MC").
ok
(genco@ferdmbp.local)4> mafiapp:debts("Albert Einstein").
[{1,"Don Corleone"}]
```

所有这些变化都可以在 corleone 节点上查询到：

```
(corleone@ferdmbp.local)6> mafiapp:friend_by_expertise(physicist).
[{"Albert Einstein",
  [{city,"Princeton, New Jersey, USA"}],
  [physicist,savant,[{awards,[{1921,"Nobel Prize"}]}]],
  physicist,
  [{from,"Don Corleone",
         {1905,'?','?'},
```

```
          "Added the square to E = MC"}]}]
```

现在，如果关闭一个节点，然后重新启动它，那么一切应该仍然正常：

```
(corleone@ferdmbp.local)7> init:stop().
Ok

$ erl -name corleone -pa ebin
... <snip> ...
(corleone@ferdmbp.local)1> net_kernel:connect('genco@ferdmbp.local').
true
(corleone@ferdmbp.local)2>
application:start(mnesia), application:start(mafiapp).
ok
(corleone@ferdmbp.local)3> mafiapp:friend_by_expertise(physicist).
[{"Albert Einstein",
... <snip> ...
          "Added the square to E = MC"}]}]
```

很不错吧？现在，我们已经成功使用了 Mnesia！

> **注意** 如果在系统运行过程中发现表的数据出现混乱，或者只是出于好奇，想看看所有的表记录，可以调用函数 `tv:start()`。它会启动一个图形化的表查看器，可以可视化地和表进行交互，无需编写代码。

29.10 删除操作示例

等等！我们是不是不打算讲任何关于如何从数据库中删除记录的内容了？啊，不！我们来增加一个表，通过它来学习一下如何删除记录。

我们会为你和老板加入一点新的特性，允许存储一些仅供私人查看的敌人信息：

```
-record(mafiapp_enemies, {name,
                          info=[]}).
```

因为这是私人信息，所以需要一些不同的表设置选项，在安装表时，会采用 `local_content` 选项。这个选项会让表成为每个节点私有的，这样每个人的私有敌人信息就不会被其他人无意中看到（不过，使用 `rpc` 可以轻易绕过这个限制）。

下面是新的 `install` 函数以及 `mafiapp:start/2` 函数，这两个函数针对这个新表做了相应的更改：

```
start(normal, []) ->
    mafiapp_sup:start_link([mafiapp_friends,
                            mafiapp_services,
                            mafiapp_enemies]).
...
install(Nodes) ->
    ok = mnesia:create_schema(Nodes),
    application:start(mnesia),
    mnesia:create_table(mafiapp_friends,
```

```
                        [{attributes, record_info(fields, mafiapp_friends)},
                         {index, [#mafiapp_friends.expertise]},
                         {disc_copies, Nodes}]),
    mnesia:create_table(mafiapp_services,
                        [{attributes, record_info(fields, mafiapp_services)},
                         {index, [#mafiapp_services.to]},
                         {disc_copies, Nodes},
                         {type, bag}]),
    mnesia:create_table(mafiapp_enemies,
                        [{attributes, record_info(fields, mafiapp_enemies)},
                         {disc_copies, Nodes},
                         {local_content, true}]),
application:stop(mnesia).
```

现在，start/2 函数把表 mafiapp_enemies 传递给监督者，并在其中完成表的加载等待。函数 install/1 在测试或者初次安装时很有用，不过，如果是在生产环境中，你就可以直接调用 mnesia:create_table/2 函数增加表。你可以根据系统的负载以及节点个数预先做一些实战演练。

现在，可以编写一个简单的测试，看看情况如何，该测试仍然在 mafiapp_SUITE 中：

```
...
-export([add_service/1, friend_by_name/1, friend_by_expertise/1,
         friend_with_services/1, accounts/1, enemies/1]).

all() -> [add_service, friend_by_name, friend_by_expertise,
          friend_with_services, accounts, enemies].
...
enemies(_Config) ->
    undefined = mafiapp:find_enemy("Edward"),
    ok = mafiapp:add_enemy("Edward", [{bio, "Vampire"},
                                      {comment, "He sucks (blood)"}]),
    {"Edward", [{bio, "Vampire"},
                {comment, "He sucks (blood)"}]} =
       mafiapp:find_enemy("Edward"),
    ok = mafiapp:enemy_killed("Edward"),
    undefined = mafiapp:find_enemy("Edward").
```

函数 add_enemy/2 和 find_enemy/1 的实现和前面的类似。对于 add_enemy/2 来说，就是一个基本的插入操作，对于 find_enemy/1 来说，就是一个基于主键的 mnesia:read/1 调用：

```
add_enemy(Name, Info) ->
    F = fun() -> mnesia:write(#mafiapp_enemies{name=Name, info=Info}) end,
    mnesia:activity(transaction, F).

find_enemy(Name) ->
    F = fun() -> mnesia:read({mafiapp_enemies, Name}) end,
    case mnesia:activity(transaction, F) of
        [] -> undefined;
```

```
        [#mafiapp_enemies{name=N, info=I}] -> {N,I}
    end.
```

函数 enemy_killed/1 稍微有些不同：

```
enemy_killed(Name) ->
    F = fun() -> mnesia:delete({mafiapp_enemies, Name}) end,
    mnesia:activity(transaction, F).
```

这对于简单的删除操作来说足够了。把这些函数导出，运行测试套件，所有测试都仍然应该通过。

当在两个节点上测试时（要先删除之前的数据模式，或者只调用 create_table 函数），可以看到表之间并没有共享数据：

```
$ erl -name corleone -pa ebin
```

```
$ erl -name genco -pa ebin
```

启动完节点后，重新安装数据库：

```
(corleone@ferdmbp.local)1> net_kernel:connect('genco@ferdmbp.local').
true
(corleone@ferdmbp.local)2> mafiapp:install([node()|nodes()]).
=INFO REPORT==== 8-Apr-2013::21:21:47 ===
... <snip> ...
{[ok,ok],[]}
```

启动应用，并进行操作：

```
(genco@ferdmbp.local)1> application:start(mnesia), application:start(mafiapp).
ok
```

```
(corleone@ferdmbp.local)3> application:start(mnesia), application:start(mafiapp).
ok
(corleone@ferdmbp.local)4>mafiapp:add_enemy("Some Guy", "Disrespected his family").
ok
(corleone@ferdmbp.local)5> mafiapp:find_enemy("Some Guy").
{"Some Guy","Disrespected his family"}
```

```
(genco@ferdmbp.local)2> mafiapp:find_enemy("Some Guy").
undefined
```

可以看到数据没有共享。删除记录也很简单：

```
(corleone@ferdmbp.local)6> mafiapp:enemy_killed("Some Guy").
ok
(corleone@ferdmbp.local)7> mafiapp:find_enemy("Some Guy").
undefined
```

结束！

29.11 列表推导式查询

如果你从头到尾阅读本章到了这一节（或者前面没看，直接跳到这里），并且心中暗想："真讨厌，我不喜欢 Mnesia 的工作方式。"那么你可能会喜欢这一节。如果你并不讨厌 Mnesia，那么你可能也会喜欢这一节。如果你喜欢列表推导（list comprehension），那么你肯定会喜欢这一节。

列表推导式查询（query list comprehension，QLC）在本质上是一个编译器技巧，使用解析转换（parse transform）把列表推导应用在任何可以搜索和迭代的数据结构之上。它在 Mnesia、DETS 和 ETS 中均有实现，也可以在像 `gb_trees` 之类的数据结构中实现。

一旦把`-include_lib("stdlib/include/qlc.hrl")`添加到模块中，就可以将*查询句柄*当成生成器（generator）用在列表推导中。查询句柄可以使任何可迭代的数据结构为 QLC 所用。在 Mnesia 中，你可以把 `mnesia:table(TableName)` 当成一个列表推导生成器，之后就可以把列表推导包装在调用 `qlc:q(...)` 中去查询任何数据库表了。

这个调用会返回一个修改过的查询句柄，该句柄包含了比表所返回的句柄更多的细节。之后，可以使用 `qlc:sort/1-2` 之类的函数继续修改这个新返回的句柄，也可以使用函数 `qlc:eval/1` 或者 `qlc:fold/1` 对它进行求值。

我们来进行一些实验。我们将重写几个 `mafiapp` 中的函数。请制作一份 mafiapp-1.0.0 的副本，并把它命名为 mafiapp-1.0.1（别忘了增加.app 文件中的版本号）。

我们先来改造 `friend_by_expertise`。这个函数目前是基于 `mnesia:match_object/1` 实现的。下面是使用 QLC 实现的版本：

```
friend_by_expertise(Expertise) ->
    F = fun() ->
        qlc:eval(qlc:q(
            [{Name,C,I,E,find_services(Name)} ||
              #mafiapp_friends{name=Name,
                               contact=C,
                               info=I,
                               expertise=E} <- mnesia:table(mafiapp_friends),
            E =:= Expertise]))
    end,
    mnesia:activity(transaction, F).
```

可以看出，除了调用 `qlc:eval/1` 和 `qlc:q/1` 的部分，它就是一个常规的列表推导。最终的表达式是 `{Name,C,I,E,find_services(Name)}`，生成器是 `#mafiapp{...} <- mnesia:table(...)`，最后还有个条件是 `E =:= Expertise`。现在，对表的搜索看起来更加自然，更加符合 Erlang 的风格。

下面我们来尝试一个更加复杂的例子，即改造 `debts/1` 函数。它原先的实现中使用了匹配规格说明，然后在一个字典上进行折叠操作。下面是使用 QLC 实现的代码：

```
debts(Name) ->
    F = fun() ->
        QH = qlc:q(
```

```
        [if Name =:= To -> {From,1};
            Name =:= From -> {To,-1}
    end || #mafiapp_services{from=From, to=To} <-
             mnesia:table(mafiapp_services),
           Name =:= To orelse Name =:= From]),
qlc:fold(fun({Person,N}, Dict) ->
          dict:update(Person, fun(X) -> X + N end, N, Dict)
         end,
         dict:new(),
         QH)
end,
lists:sort([{V,K} || {K,V} <- dict:to_list(mnesia:activity(transaction, F))]).
```

不再需要匹配规格说明了。列表推导（保存在 QH 查询句柄中）替代了匹配规格说明。折叠操作被移到事务中，用它来对查询句柄进行求值。返回的字典和前面使用 `lists:foldl/3` 返回的完全一样。最后，把 `mnesia:activity/1` 返回的字典转换成列表，然后对其排序，排序是在事务之外进行的。

很好。如果你是在 mafiapp-1.0.1 应用中编写这些函数的话，现在运行测试套件，所有 6 个测试都仍然应该通过。

29.12　记住 Mnesia

关于 Mnesia 的内容就讲到这里。它是一个相当复杂的数据库，本章只覆盖了全部 Mnesia 内容的一半左右。如果想继续深入学习，就得去学习 Erlang 的手册，并研读代码。真正具有大规模、可伸缩系统 Mnesia 使用经验的程序员少之又少。你可能会在邮件列表中遇到其中的几个，有时他们会回答一些问题，但是一般来讲，他们都非常繁忙。

不过，对于小规模的，或者规模大一点但节点数目已知的应用来说，Mnesia 永远都是一个非常好的可选工具。能够直接存储并复制 Erlang 数据项是一件非常好的事情——在其他语言中，需要花费多年时间去编写对象—关系映射层来解决这个问题。

有趣的是，如果愿意的话，你甚至可以为 SQL 数据库或者其他任何支持迭代的存储系统编写 QLC 选择器。

Mnesia 及其工具链对于你未来的某些应用可能是非常有用的。不过现在，我们要转向另外一个对开发 Erlang 系统有帮助的工具——Dialyzer。

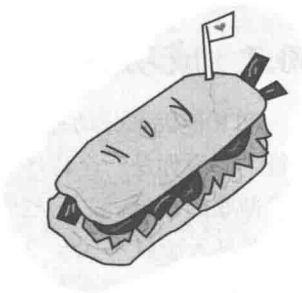
第 30 章

类型规格说明与 Dialyzer

本章将关注 Dialyzer，它是一个非常高效的 Erlang 代码分析工具。它可以发现各种缺陷，例如，从来未被执行的代码，不过，它主要还是用来检测 Erlang 代码中的类型错误。我们会介绍一下为什么会创建 Dialyzer 这样的工具，它背后的指导原则是什么以及它在发现类型错误方面的能力。当然，我们还会通过一些例子来展示如何使用 Dialyzer。

30.1 PLT 是最好的三明治

使用 Dialyzer 的第一步是创建 Dialyzer 的持久化查询表（persistent lookup table，PLT）。持久化查询表中汇集了关于标准 Erlang 发布中的应用和模块，以及 OTP 之外的代码的所有细节信息，Dialyzer 可以识别这些信息。



这个汇集所有信息的过程非常耗时，如果你使用的平台不支持 HiPE 本地编译（也就是说，Windows 系统），或者使用的 Erlang 版本比较老，那么这个过程会更加耗时。好在这个过程在逐步变快，最新发布的 Erlang（R15B02 之后）已经支持并行 PLT 构建，速度也变得更快。

在终端输入如下命令，等待它运行结束（在我的环境中，大概要 10 min）：

```
$ dialyzer --build_plt --apps erts kernel stdlib crypto mnesia sasl common_test eunit
Compiling some key modules to native code... done in 1m19.99s
Creating PLT /Users/ferd/.dialyzer_plt ...
eunit_test.erl:302: Call to missing or unexported function eunit_test:nonexisting_
function/0
Unknown functions:
compile:file/2
compile:forms/2
... <snip> ...
xref:stop/1
```

```
Unknown types:
compile:option/0
done in 6m39.15s
done (warnings were emitted)
```

这条命令会构建出包含指定 OTP 应用的 PLT。如果愿意，你可以无视 Dialyzer 给出的警告信息，因为那些未知的函数不影响 Dialyzer 寻找类型错误（这和它使用的类型推导算法有关，会在下一节介绍）。有些 Windows 用户会看到一条出错消息“需要设置 HOME 环境变量，这样 Dialyzer 就知道从哪里寻找默认的 PLT。”这是因为在 Windows 中不能保证一定设置了 HOME 环境变量，于是，Dialyzer 就不知道该把 PLT 放在哪里了。请把该变量设置成你希望 Dialyzer 存放 PLT 文件的目录位置。

如果愿意，可以把 `ssl` 和 `reltool` 之类的应用包含进来，方法是把它们加入`--apps` 后面的序列中，如果 PLT 已经构建过了，就可以使用如下方法：

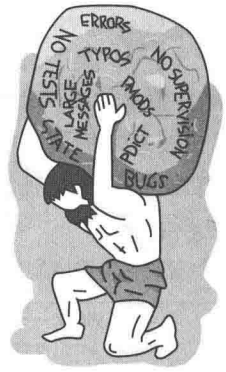
```
$ dialyzer --add_to_plt --apps ssl reltool
```

如果想把自己的应用或者模块加到 PLT 中，可以使用`-r Directories`，它会寻找指定目录中的所有.erl 和.beam 文件（只要它们都采用 `debug_info` 选项进行编译）。

此外，你可以用 Dialyzer 创建多个 PLT，而在使用时，则可以通过在命令中使用`--plt Name` 选项，从这些 PLT 中选择一个使用。此外，如果你创建了多个不相交的 PLT 文件，也就是说，PLT 之间没有任何共享的模块，则可以通过`--plts Name1 Name2 ... NameN` 把它们“合并”到一起。当希望对于不同的项目或者 Erlang 版本使用不同的 PLT 时，这个功能尤为有用。

PLT 还在构建，趁这个时间，我们来了解一下 Dialyzer 的类型错误发现机制。

30.2　成功类型化



和其他大部分动态编程语言类似，Erlang 程序常常要承受类型错误的风险。假设你是一个程序员，在调用一个函数时传错了参数，可能又忘了进行彻底的测试。之后程序被部署到生产环境中，一切看起来都不错。然后，在某天凌晨 4 点，公司运维人员的手机响个不停，就是因为你的那段代码不停地崩溃——次数太多，以至于监督者都无法处理。

第二天早上，你来到办公室，发现自己的电脑已经被格式化，汽车也被用钥匙划坏了，代码提交权限也被撤销，而这些都是那个被你的错误打乱了生活的运维人员干的。

如果有一个带有静态类型分析器的编译器，就可以在程序运行前对其进行检验，那么整个灾难是完全可以避免的。

由于 Erlang 对运行时错误采用了反应式处理方式，因此 Erlang 并不像其他动态语言那样渴求一个类型系统，不过，如果能在早期通过发现类型相关的错误来提供额外的安全性，那就再好不过了。

通常，具有静态类型系统的语言就是这样设计的。语言的语义在很大程度上受到它们自身类型系统规定的影响。例如，考虑如下函数：

```
foo(X) when is_integer(X) -> X + 1.
foo(X) -> list_to_atom(X).
```

大部分的类型系统都不能正确地表示出这个函数的类型。类型系统知道该函数接收一个整数或者列表，返回一个整数或者原子，但是类型系统无法跟踪函数输入类型和输出类型之间的依赖关系（条件类型和交集类型可以做到，但是非常烦琐）。这意味着，像这种在 Erlang 中非常普通的函数，如果在代码中使用了它们，也会给类型分析器带来一些不确定性。

一般来讲，分析器希望能够确切证明在运行时不会出现任何类型错误，就像数学证明一样。这意味着，在有些情况下，类型检查器为了去除一些可能导致崩溃的不确定性会禁止某些实际上有效的操作。

实现这样的一个类型系统，在很大程度上意味着要强迫 Erlang 去更改它的语义。

问题是，当 Dialyzer 出现时，Erlang 已经在一些大型项目中使用多时了。Dialyzer 这样的工具要想被接受，就必须遵循 Erlang 的哲学。如果 Erlang 允许一些完全没有意义，只能在运行时确定的类型，那么就只能这样了。类型检查器没有权利抱怨。程序员们都不喜欢一个工具告诉他们已经在生产环境中运行了几个月的程序是不能运行的！

另外一个选择是，类型系统不去证明没有错误，而是去尽力检测它能检测的错误。这种检测可以做得很好，但是无法做到完美——这是一个权衡。

Dialyzer 类型系统的设计者所做的决定是，不去证明程序是没有类型错误的，而是在不与当前 Erlang 实现冲突的情况下去找出尽可能多的错误。正如支撑 Dialyzer 的论文“Practical Type Inference Based on Success Typings”（http://www.it.uu.se/research/group/hipe/papers/succ_types.pdf）中所述，针对 Erlang 这种语言的类型检查器，应该无需类型声明就可以工作（可以接受一些提示），应该简单、清晰，应该去适应语言（而非让语言适应它），并且仅在确信会发生崩溃时才给出类型错误的警告。

> 我们的主要目标是揭示出 Eralng 代码中的隐式类型信息，使之在程序中显式可用。鉴于典型 Erlang 应用的规模，类型推导应该是完全自动的，并忠于该语言的操作语义。此外，不能进行任何形式的代码改写。原因很简单。仅仅为了类型推导的需要，就对那些具有数百万行代码且安全关键的应用进行改写是不会取得成功的。但是，大型软件应用必须要维护，并且维护人员常常不是最初的开发者。因此，通过自动地揭示已经存在的类型信息，我们提供了一份和程序一起演化且不会腐化的自动化文档。我们同样认为，达成准确性和清晰性之间的平衡是非常重要的。最后但绝非最不重要的一点是，推导出来的类型绝对不能有错。

在开始每次分析时，Dialyzer 都是乐观的，它会认为所有的函数都是良好的。它会认为函数总会成功，可以接受任何东西，也能够返回任何东西，无论以何种方式使用一个未知的函数都是正确的。这就是在产生 PLT 时，为何可以无视未知函数警告的原因。事情总是好的。在类型推导方面，Dialyzer 是一个天生的乐观主义者。

随着分析的进行，Dialyzer 对函数的了解越来越深入。这样，它就可以对代码进行分析，并发现一些有趣的信息。

假设，有个函数中的两个参数之间有一个+操作符，并且返回了相加后的值。Dialyzer 就不再假设这个函数可以接收任何东西，并返回任何东西，而是会期望参数是数值类型的（整数或者浮点数），返回值也是数值类型的。Dialyzer 会赋予该函数一个基本的类型：接收两个数值类型，并返回一个数值类型。

现在，假设有个函数用一个原子和数值作为参数调用了上面描述的那个函数。Dialyzer 就会分析这段代码并有所发现，“等等，不能把原子和数值作为+操作符的参数！”接着它会大发脾气，因为一个原本会返回数值类型的函数，当以这种方式使用时，就什么也返回不了了。

不过，在某些更一般的情况中，你可能会发现，对于不少肯定存在问题的代码，Dialyzer 却保持沉默。例如，考虑如下代码片段：

```
main() ->
    X = case fetch() of
        1 -> some_atom;
        2 -> 3.14
    end,
    convert(X).

convert(X) when is_atom(X) -> {atom, X}.
```

这段代码调用了 `fetch/0` 函数，`fetch` 函数会返回 1 或者 2。根据返回值，`main` 函数要么返回一个原子，要么返回一个浮点数。

我们可以看出，在某些情况下，对 `covert/1` 的调用会失败。当 `fetch()` 返回 2 时，我们希望得到一个类型错误的提示，因为此时会给 `convert/1` 传入一个浮点值。Dialyzer 不这样认为。记住，Dialyzer 是乐观主义者。Dialyzer 完全信任你的代码，由于对 `convert/1` 函数的调用存在成功的可能性，因此 Dialyzer 保持沉默。对于这种情况，不会给出任何错误报告。

30.3　类型推导和错误

我们将用 Dialyzer 分析 discrep1.erl、discrep2.erl 和 discrep3.erl 这 3 个模块来实践一下 30.2 节中介绍的原则。

discrep1.erl 的代码如下：

```
-module(discrep1).
-export([run/0]).

run() -> some_op(5, you).

some_op(A, B) -> A + B.
```

这个例子中的错误很明显。不能把 5 和 you 原子相加。我们可以让 Dialyzer 来分析这段代码，假设 PLT 已经创建完成：

```
$ dialyzer discrep1.erl
  Checking whether the PLT /home/ferd/.dialyzer_plt is up-to-date... yes
  Proceeding with analysis...
discrep1.erl:4: Function run/0 has no local return
```

```
discrep1.erl:4: The call discrep1:some_op(5,'you') will never return
since it differs in the 2nd argument from the success typing arguments:
(number(),number())
discrep1.erl:6: Function some_op/2 has no local return
discrep1.erl:6: The call erlang:'+'(A::5,B::'you') will never return
since it differs in the 2nd argument from the success typing arguments:
(number(),number())
  done in 0m0.62s
done (warnings were emitted)
```

啊，太有趣了——Dialyzer 发现了一些东西。它到底发现了什么呢？

第一条出错消息是在 Dialyzer 使用过程中经常会看到的。“Function Name/Arity has no local return”是一条标准的 Dialyzer 警告信息，在确信函数不会返回时（抛出异常的情况除外）打印这条信息，原因是：这个函数调用了另外一个函数，被调用的函数正好不满足 Dialyzer 的类型错误检测器或者自己抛出了一个异常。当出现这种情况时，被调用函数可能的返回值类型集合为空，实际上就不会返回。这个错误会传播到它的调用者那里，就会打印出“no local return”的错误。

第二条错误更好理解一些。它的意思是，Dialyzer 检测到了函数 `some_op` 正常工作要求的类型：两个数值类型（`number()`和 `number()`），但是调用 `some_op(5, 'you')`不满足这个类型。这种类型表示方式确实有些另类，我们会在稍后详细介绍。

第三条错误同样是“no local return”。出现第一条的原因是 `some_op/2` 函数会失败，出现这条的原因则是因为+操作会失败。这也正是第四条，也是最后一条错误所描述的。加操作符（实际上是函数 `erlang:'+'/2`）不能把数值 5 和原子 `you` 相加。

discrep2.erl 是什么呢？其代码如下：

```
-module(discrep2).
-export([run/0]).

run() ->
    Tup = money(5, you),
    some_op(count(Tup), account(Tup)).

money(Num, Name) -> {give, Num, Name}.
count({give, Num, _}) -> Num.
account({give, _, X}) -> X.

some_op(A, B) -> A + B.
```

对该文件运行 Dialyzer，发现的错误和 discrep1.erl 类似：

```
$ dialyzer discrep2.erl
  Checking whether the PLT /home/ferd/.dialyzer_plt is up-to-date... yes
  Proceeding with analysis...
discrep2.erl:4: Function run/0 has no local return
discrep2.erl:6: The call discrep2:some_op(5,'you') will never return
since it differs in the 2nd argument from the success typing arguments:
(number(),number())
discrep2.erl:12: Function some_op/2 has no local return
discrep2.erl:12: The call erlang:'+'(A::5,B::'you') will never return
```

```
since it differs in the 2nd argument from the success typing arguments:
(number(),number())
 done in 0m0.69s
done (warnings were emitted)
```

在分析过程中，Dialyzer 可以通过函数 count/1 和 account/1 得到类型信息。它先推导出元组中每个元素的类型，接着弄清楚传给它们的值。然后，就能容易地发现错误。

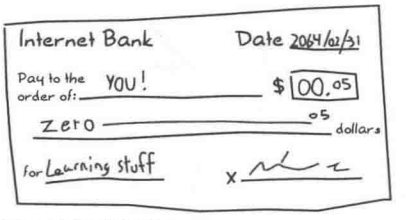
再增加点难度，我们来看看 discrep3.erl：

```
-module(discrep3).
-export([run/0]).

run() ->
    Tup = money(5, you),
    some_op(item(count, Tup), item(account, Tup)).

money(Num, Name) -> {give, Num, Name}.

item(count, {give, X, _}) -> X;
item(account, {give, _, X}) -> X.

some_op(A, B) -> A + B.
```

这个版本多引入了一个间接层次。其中并没有把 count 和 account 明确地定义成函数，而是使用原子值在不同的函数子句之间进行选择。如果使用 Dialyzer 对这个模块进行分析，可以得到如下信息：

```
$ dialyzer discrep3.erl
  Checking whether the PLT /home/ferd/.dialyzer_plt is up-to-date... yes
  Proceeding with analysis... done in 0m0.70s
done (passed successfully)
```

哎呀——看起来这次对文件的更改把事情弄得太复杂了，以至于 Dialyzer 都迷失在我们的类型定义中了。错误依然存在。在 30.5 节中，我们会再来看这个问题，并弄清楚 Dialyzer 为何没有发现文件中的错误，以及如何进行修正。现在，我们要先试试 Dialyzer 的另外几种使用方法。



如果想用 Dialyzer 分析我们的 Process Quest 发布，可以按如下方式进行：

```
$ cd processquest/apps
$ ls
processquest-1.0.0 processquest-1.1.0 regis-1.0.0
regis-1.1.0 sockserv-1.0.0 sockserv-1.0.1
```

目录中有一组库。Dialyzer 无法处理多个具有相同名字的模块，因此我们得手工指定目录：

```
$ dialyzer -r processquest-1.1.0/src regis-1.1.0/src sockserv-1.0.1/src
  Checking whether the PLT /home/ferd/.dialyzer_plt is up-to-date... yes
  Proceeding with analysis...
```

```
dialyzer: Analysis failed with error:
No .beam files to analyze (no --src specified?)
```

哦，对了——默认情况下，Dialyzer 只会寻找.beam 文件。我们得增加一个--src 标志让 Dialyzer 分析.erl 文件：

```
$ dialyzer -r processquest-1.1.0/src regis-1.1.0/src sockserv-1.0.1/src --src
  Checking whether the PLT /home/ferd/.dialyzer_plt is up-to-date... yes
  Proceeding with analysis... done in 0m2.32s
done (passed successfully)
```

注意，在所有命令中，我们都指定了 src 目录。我们本可以不这么做，但是如果不这么做，Dialyzer 就会给出一堆关于 EUnit 测试文件的出错消息，这些出错消息与代码分析器处理某些断言宏的方式有关——而我们并不想关心这些。另外，有时我们想针对失败的情况进行测试，会在测试中故意让软件崩溃，Dialyzer 会对此给出警告，这种情况也不是我们所希望的。

30.4 类型的种类

如 discrep3.erl 中所示，在有些情况下，Dialyzer 不能以我们希望的方式推断出所有类型。这是因为，Dialyzer 无法读懂我们的心思。为了帮助 Dialyzer（其实主要是帮助我们自己），我们可以定义一些类型，并对函数进行标注，这样既可以起到文档化的作用，又可以规范化我们放置在代码中关于类型的隐式期望。

30.4.1 单例类型

Erlang 的类型可以像数字 42 一样简单，类型书写形式就是 42（和平常的写法没有什么两样），或者是具体的原子，如 cat 或者 molecule。这些类型称为*单例类型*（singleton type），因为它们仅有一个和自身一样的值。表 30-1 中列出了一些单例类型。

表 30-1 Erlang 单例类型

类 型	描 述
'some atom'	任何原子都是值为自身的单例类型
42	给定的整数
[]	空列表
{}	空元组
<<>>	空二进制

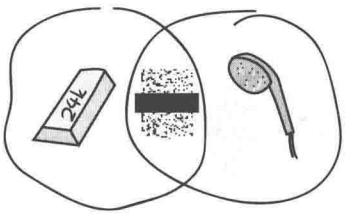
可以看出，在编写 Erlang 程序时，如果只能使用这些类型，是非常令人讨厌的。因为完全没有办法在程序中使用单例类型表达像年龄之类的概念，更不用说表达“所有整数”了。而且，即使我们有办法同时指定多个类型，但是如果不把它们全部手工写出来，是很难表达像“任何整数”这样的概念的，而这又是不可能的。因此，Erlang 提供了*联合类型*（union type）和一些*内置类型*。

30.4.2 联合类型和内置类型

联合类型可以用来描述一些更加复杂的概念，如具有两个原子值的类型。内置类型是一些预先定义好的类型，有些内置类型无法自己手工创建。

一般来讲，联合类型和内置类型的语法比较相似，书写形式为 *TypeName()*。例如，表示所有整数的类型可以书写成 integer()。使用括号的原因是为了进行区分，例如，atom()类型表示所有的原子，而 atom 则只表示 atom 这个具体的原子。此外，为了使代码更加清楚，许多 Erlang 程序员会选择在类型声明中把原子都用单引号括起来，会使用'atom'而不是 atom。这种做法显式地表达了'atom'指的是单例类型，而不是忘记增加括号的内置类型。

表 30-2 中列出了 Erlang 语言提供的内置类型。注意，它们并不像联合类型那样完全具有一样的语法形式。其中有些类型的语法形式比较特殊，如二进制和元组，这样是为了更加好用。



有了表中列出的内置类型，在 Erlang 程序中定义类型就变得容易一点。不过还是缺些东西。也许内置类型过于笼统或者不能满足我们的要求。还记得前面某个 discrep*N* 模块的错误中提到的 number()类型吗？这个类型既不是单例类型，也不是内置类型。它是一个联合类型，这意味着我们可以自己定义它。

表 30-2　Erlang 内置类型

类　型	描　述
any()	任意 Erlang 数据项
none()	一个特殊类型，表示无效的数据项和类型。通常，当 Dialyzer 认为函数的返回值是 none()时，就意味着该函数会崩溃。它和“这个东西会出问题”同义
pid()	进程标识
port()	端口是文件描述符（很少见到，除非深入 Erlang 库的实现代码中）、套接字以及一般意义上的 Erlang 用来和外部世界进行通信的东西（如 erlang:open_port/2 函数）的底层表示。在 Erlang shell 上，端口看起来像这样：#Port<0.638>
reference()	make_ref()或者 erlang:monitor/2 返回的唯一值
atom()	原子
binary()	一块二进制数据
<<_:Integer>>	长度已知的二进制数据，Integer 表示长度
<<_:_*Integer>>	单位大小已知的二进制数据，长度未知
<<_:Integer, _:_*OtherInteger>>	前面两种形式的混合，用于指定具有最小长度的二进制数据
integer()	整数
N..M	整数范围。例如，想表达一年中的月份时，可以定义一个范围 1..12。注意，Dialyzer 保留把范围变大的权利
non_neg_integer()	大于等于 0 的整数

续表

类　型	描　述
pos_integer()	大于 0 的整数
neg_integer()	小于等于-1 的整数
float()	任意浮点数
fun()	任意函数
fun((...) -> Type)	匿名函数，返回给定类型，具有任意参数个数。返回列表类型的函数可以书写成 fun((...) -> list())
fun(() -> Type)	匿名函数，返回给定类型，无参数
fun((Type1, Type2, ..., TypeN) -> Type)	参数类型已知且数目固定的匿名函数。例如，接收一个整型，一个浮点型值的函数可以声明成 fun((integer(), float()) -> any())
[Type()]	由给定类型组成的列表。整数列表可以定义为[integer()]，还可以写成 list(Type())。有时，列表不是良构的（如[1, 2 | a]）。因此，Dialyzer 中有针对非良构列表的类型声明：improper_list(TypeList, TypeEnd)。例如，非良构列表[1, 2 | a]的类型为 improper_list(integer(), atom())。更复杂的是，有时无法知道一个列表是否是良构的。此时可以使用类型 maybe_improper_list(TypeList, TypeEnd)。
[Type(), ...]	[Type()]的一种特殊形式，要求列表不能为空
tuple()	任意元组
{Type1, Type2, ..., TypeN}	元素个数、元素类型已知的元组。例如，一个二叉树节点可以定义为{'node', any(), any(), any(), any()}，表示{'node', LeftTree, RightTree, Key, Value}

表示联合类型的符号是管道符(|)。简单来讲，联合类型可以让我们用一组类型 *Type1* | *Type2* | ... | *TypeN* 的联合来表示某个给定类型 *TypeName*。因此，number()类型（包含整数和浮点数）可以表示为 integer() | float()，Boolean 类型可以被定义为'true' | 'false'。在定义类型时，也可以只使用一个其他类型。这看起来像是一个联合类型，实际上是类型别名。

事实上，Erlang 语言中预定义了许多联合类型和类型别名。表 30-3 中列出了其中的一部分。

表 30-3　预定义的联合类型和类型别名

类　型	定　义
term()	和 any()等价。之所以预定义它，是因为有些工具之前已经使用了 term()。另外，可以把变量_作为 term()和 any()的别名
boolean()	被定义为'true' | 'false'

续表

类　型	定　义
byte()	被定义为 0..255，表示所有合法的字节
char()	被定义为 0..16#10ffff，但是不确定这个类型是否和某个具体的字符集标准相关。这是一种非常常用的避免冲突的方法
number()	integer()|float()
maybe_improper_list()	类型 maybe_improper_list(any(), any())的简洁别名，表示一般意义上的非良构列表
maybe_improper_list(T)	类型 maybe_improper_list(T, any())的别名，其中 T 是任意类型
string()	被定义为[char()]，表示字符列表。还有一个类型 nonempty_string()，被定义为[char(), ...]。遗憾的是，目前为止还没有针对二进制字符串的类型，主要因为它们只是一些二进制数据块，可以把它们解释为任意类型
iolist()	被定义为 maybe_improper_list(char() | binary() | iolist(),binary() | [])。可以看出，iolist 是递归定义的。从 R13B04 开始，Dialyzer 开始支持递归类型。在那之前，像 iolist 这样的类型定义起来非常麻烦
module()	表示模块名的类型，目前是 atom()类型的别名
timeout()	被定义为 non_neg_integer() | 'infinity'，表示可以被 receive 表达式中的 after 部分接受的值
node()	Erlang 节点名，是一个原子
no_return()	类型 none()的别名，用于函数的返回类型，主要标注那些永远循环（希望如此）而不会返回的函数

30.4.3　定义类型

嗯，表中已经定义了一些类型。表 30-2 中提到了一个表示树的类型，书写为{'node', any(), any(), any(), any()}。现在，我们来看看如何在模块中定义这个类型。在模块中定义类型的语法如下：

```
-type TypeName() :: TypeDefinition.
```

于是，可以这样定义树类型：

```
-type tree() :: {'node', tree(), tree(), any(), any()}.
```

此外，还可以用一种特殊的语法对其进行定义，其中可以使用变量名作为类型注释，具体如下：

```
-type tree() :: {'node', Left::tree(), Right::tree(), Key::any(), Value::any()}.
```

不过，上面的定义有些问题，因为它不允许树是空的。通过递归思维，我们可以得到一个更好的定义，就像第 5 章中 tree.erl 模块中的做法。在那个模块中，空树被定义为{node, 'nil'}。每当递归函数处理到这样的结点时就停止。正常的树结点（非空）书写为{node, Key, Val, Left, Right}。把这种思路翻译到类型中，可以得到如下形式的树结点类型定义：

```
-type tree() :: {'node', 'nil'}
              | {'node', Key::any(), Val::any(), Left::tree(), Right::tree()}.
```

这样，我们就定义了一棵树，它要么是一个空结点，要么是具有内容的结点。注意，我们可以使用'nil'而不是{'node', 'nil'}，Dialyzer 也可以毫无问题地处理它。写成这样只是想和 tree 模块中的写法保持一致。

我们还想给另外一块 Erlang 代码赋予类型，不过其中会用到我们还没有介绍过的类型。

30.4.4 记录类型

如何定义记录类型呢？对于记录，可以用一个非常方便的语法声明其类型。作为示例，假设有一个#user{}记录。我们想存储用户的名字、一些具体的记录（使用 tree()类型）、用户的年龄、朋友列表和一个简短的自传。

```
-record(user, {name="" :: string(),
               notes :: tree(),
               age :: non_neg_integer(),
               friends=[] :: [#user{}],
               bio :: string() | binary()}).
```

记录类型声明的一般语法形式为 Field :: Type，如果某个字段存在默认值，那么形式就为 Field = Default :: Type。对于#user{}记录，可以看出名字字段是一个字符串，注释字段是树类型，年龄字段是任何非负整数（有谁知道人能活多大岁数吗？）。

比较有意思的是 friends 字段。类型[#user{}]表示 user 记录中可以包含 0 个或者多个其他 user 记录。它还表明，记录也可以作为类型使用，书写形式为#RecordName{}。记录的最后一部分表明，自传可以是字符串或者二进制数据。

另外，为了使类型声明和定义的形式更加一致，大家往往会使用-type Type() :: #Record{}.这样的类型别名。我们可以把 friends 的定义更改成使用 user()类型的形式，结果如下：

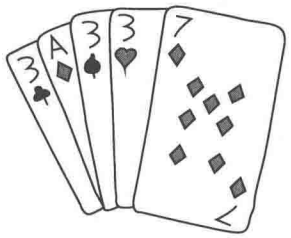
```
-record(user, {name = "" :: string(),
               notes :: tree(),
               age :: non_neg_integer(),
               friends=[] :: [user()],
               bio :: string() | binary()}).

-type user() :: #user{}.
```

这里，我们为记录中的所有字段都定义了类型，不过其中有些没有默认值。如果我们创建了

一个记录实例#user{age=5}，那么是不会出现任何类型错误的。因为，如果没有为记录字段提供默认值，就会自动把一个隐式的'undefined'类型和它原先定义的类型联合起来。而在早期的版本中，这种声明方式会导致类型错误。

30.5　为函数增加类型声明

虽然我们可以夜以继日地定义类型，不停地把它们写到文件中，接着把文件打印出来，装订成册，并感到极大的满足，但是 Dialyzer 的类型推导引擎并不会因此就自动使用它们。Dialzyer 无法根据你所声明的类型去决定哪些能执行，哪些不能执行。



那么，为什么还要声明这些类型呢？难道是为了文档化吗？有这方面的原因。要想让 Dialzyer 理解我们所声明的类型，还有另外一项工作要做。我们需要为所有希望增加类型的函数都加上类型签名声明，从而建立起类型声明和模块中函数之间的联系。

到目前为止，我们看到的都是语法介绍方面的内容，现在，我们来看一个实际的例子。我们要为一个纸牌游戏定义类型。纸牌共有 4 种花色：黑桃（spades）、梅花（clubs）、红桃（hearts）以及方块（diamonds）。牌的编号从 1 到 10（A 为 1），然后是 J、Q 和 K。

如果不使用 Dialyzer，我们就可以把牌定义为{Suit, CardValue}，这样，黑桃 A 就被表示为{spades, 1}。现在，无需使用这种不明确的方式，我们可以定义出表示它的类型：

```
-type suit() :: spades | clubs | hearts | diamonds.
-type value() :: 1..10 | j | q | k.
-type card() :: {suit(), value()}.
```

类型 suit()是 4 个原子的联合类型，表示花色。类型 value()表示纸牌的值，普通牌为 1 到 10(0..10)，脸谱牌为 j、q 和 k。类型 card()是一个元组类型，把上面两个类型合在一起。

现在，可以在常规的函数中使用这 3 个类型来表示纸牌，这些类型会为我们提供一些有价值的保证。以下面的 card.erl 为例：

```
-module(cards).
-export([kind/1, main/0]).

-type suit() :: spades | clubs | hearts | diamonds.
-type value() :: 1..10 | j | q | k.
-type card() :: {suit(), value()}.

kind({_, A}) when A >= 1, A =< 10 -> number;
kind(_) -> face.

main() ->
    number = kind({spades, 7}),
    face = kind({hearts, k}),
    number = kind({rubies, 4}),
    face = kind({clubs, q}).
```

函数 kind/1 会返回一张纸牌的类型：数字牌或者脸谱牌。你会注意到，这个函数没有检查花色。在 main/0 函数中，可以看到第三个函数调用中的花色参数是 rubies，显然，我们没有在类型中定义它，当然也不希望在 kind/1 函数中使用它：

```
$ erl
... <snip> ...
1> c(cards).
{ok,cards}
2> cards:main().
face
```

一切正常。这不是我们想要的。即使运行 Dialyzer 也没有发现任何问题。现在，我们来给 kind/1 函数增加如下类型签名：

```
-spec kind(card()) -> 'face' | 'number'.
kind({_, A}) when A >= 1, A =< 10 -> number;
kind(_) -> face.
```

接下来将会发生一些更有趣的事情。不过，在运行 Dialyzer 之前，先来介绍一下类型签名的工作原理。

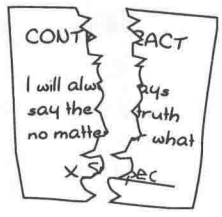
类型签名的格式为-spec *FunctionName*(*ArgumentTypes*) ->*ReturnTypes*.。上面例子中类型签名的意思是，kind/1 函数的参数是我们创建的 card()类型，它的返回值是原子 face 或者 number。

运行 Dialyzer 对这个模块进行分析，会产生如下信息：

```
$ dialyzer cards.erl
  Checking whether the PLT /home/ferd/.dialyzer_plt is up-to-date... yes
  Proceeding with analysis...
cards.erl:12: Function main/0 has no local return
cards.erl:15: The call cards:kind({'rubies',4}) breaks the contract (card()) -> 'face'
| 'number'
 done in 0m0.80s
done (warnings were emitted)
```

啊，太有趣了。根据函数类型签名，使用花色为 rubies 的“纸牌”调用 kind/1 函数是无效的。

在这个例子中，Dialyzer 遵循了我们给出的类型签名，当它分析 main/0 函数时，发现在第三行对 kind/1 函数的使用不当。于是，就给出了对第 15 行(number = kind({rubies, 4}),)的警告信息。此后，Dialyzer 认为既然类型签名是可靠的，并且如果代码必须要依照这个签名来使用的话，那么上面的代码在逻辑上就是不合理的。这个对于契约的违背会传播到 main/0 函数，不过在 main/0 函数这一层也无法给出关于失败原因的更多解释——于是，就显示了如上的出错消息。



> **注意**　只有在定义了类型签名的情况下，Dialyzer 才会给出错误提示。如果没有提供类型签名，Dialyzer 是不能假设你只打算用 card() 类型的参数使用 kind/1 函数的。有了类型签名，Dialyzer 就可以把它当作函数自己的类型定义使用了。

给下面这个函数定义类型会更有趣一些，它位于 convert.erl 中：

```
-module(convert).
-export([main/0]).

main() ->
    [_,_] = convert({a,b}),
    {_,_} = convert([a,b]),
    [_,_] = convert([a,b]),
    {_,_} = convert({a,b}).

convert(Tup) when is_tuple(Tup) -> tuple_to_list(Tup);
convert(L = [_|_]) -> list_to_tuple(L).
```

从代码中可以明显看出，最后两个对函数 convert/1 的调用会失败。该函数接受一个列表，返回一个元组，或者接受一个元组，返回一个列表。最后两个对该函数的调用与此不符，期望从元组得到元组，从列表得到列表。不过，如果用 Dialyzer 分析这段代码，就发现不了任何问题。

这是因为，Dialyzer 会推导出来类似下面这样的类型签名：

```
-spec convert(list() | tuple()) -> list() | tuple().
```

或者换句话说，这个函数接受列表和元组，返回列表和元组。遗憾的是，这没有什么错误——简直太对了。函数的实现要比类型签名的要求更加严格。此时，Dialyzer 无法百分之百确信没有问题，因此只能袖手旁观。

为了给 Dialyzer 提供一点帮助，我们可以给出一个稍微复杂点的类型声明：

```
-spec convert(tuple()) -> list();
             (list()) -> tuple().
convert(Tup) when is_tuple(Tup) -> tuple_to_list(Tup);
convert(L = [_|_]) -> list_to_tuple(L).
```

我们没有把类型 tuple() 和 list() 放在一起，形成一个联合类型，而是采用上面的语法用可选子句来声明类型签名。如果用元组作为参数调用函数 convert/1，我们期望返回一个列表，反之，则期望返回一个元组。

有了这些更具体的信息，Dialyzer 就能够给出一些更有价值的结果：

```
$ dialyzer convert.erl
  Checking whether the PLT /home/ferd/.dialyzer_plt is up-to-date... yes
  Proceeding with analysis...
convert.erl:4: Function main/0 has no local return
convert.erl:7: The pattern [_, _] can never match the type tuple()
```

```
 done in 0m0.90s
done (warnings were emitted)
```

啊，这次它发现错误了。成功！现在 Dialyzer 能够输出一些我们已经知道的信息了。当然，这样说听起来好像没有意义，但是，如果你正确地定义了函数的类型，并在随后犯了一点小错，又忘了进行检查，此时 Dialyzer 将成为你的后盾，这肯定要比会在半夜把你叫醒的错误日志系统好（也比那个做运维的家伙用钥匙划坏你的汽车好）。

注意 有些人更喜欢用下面的语法来定义多子句的类型签名：

```
-spec convert(tuple()) -> list()
      ;       (list()) -> tuple().
```

这和我们前面使用的语法完全一样，只是把分号放在了下一行，这样可读性会好一些。在写本书时，还不存在得到广泛认可的标准写法。

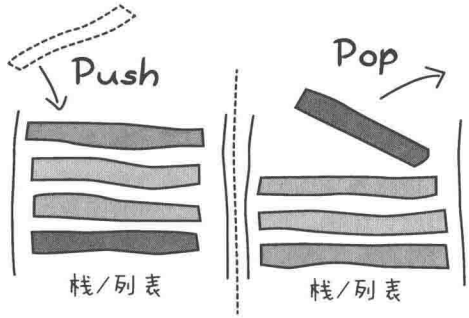
以这种方式使用类型定义和规格说明，就能够让 Dialyzer 发现前面 `discrep` 模块中的错误。我们来看看 discrep4.erl 是如何做的：

```
-module(discrep4).
-export([run/0]).
-type cents() :: integer().
-type account() :: atom().
-type transaction() :: {'give', cents(), account()}.

run() ->
    Tup = money(5, you),
    some_op(item(count,Tup), item(account,Tup)).

-spec money(cents(), account()) -> transaction().
money(Num, Name) -> {give, Num, Name}.

-spec item('count', transaction()) -> cents();
          ('account', transaction()) -> account().
item(count, {give, X, _}) -> X;
item(account, {give, _, X}) -> X.

some_op(A,B) -> A + B.
```

上面代码中最有用的类型定义是 `item/2` 的两个可选子句类型定义。它能帮助 Dialyzer 追踪返回值和输入值之间的关系，从而发现类型错误。

30.6 类型定义实践

现在，我们来实现一个支持 FIFO 操作的队列模块。你应该知道什么是队列，因为 Erlang 的消息邮箱就是队列。最先加到队列里的元素会被最先取出（除非使用了选择性接收）。队列模块的工作方式如下：



我们将使用两个列表，将它们当成栈来模拟队列。一个列表用于存储新元素，另外一个列表用来从队列中取出元素。元素永远被增加到第一个列表中，并从第二个列表中取出。当负责取元素的列表为空时，就把负责增加元素的列表反转，使之成为新的负责取出元素的列表。通常情况下，相比用一个列表来同时负责存取操作来说，这种方法可以保证比较好的平均性能。

下面是 FIFO 模块的实现，其中使用了一些类型签名，供 Dialyzer 检查使用：

```
-module(fifo_types).
-export([new/0, push/2, pop/1, empty/1]).
-export([test/0]).

-spec new() -> {fifo, [], []}.
new() -> {fifo, [], []}.

-spec push({fifo, In::list(), Out::list()}, term()) -> {fifo, list(), list()}.
push({fifo, In, Out}, X) -> {fifo, [X|In], Out}.

-spec pop({fifo, In::list(), Out::list()}) -> {term(), {fifo, list(), list()}}.
pop({fifo, [], []}) -> erlang:error('empty fifo');
pop({fifo, In, []}) -> pop({fifo, [], lists:reverse(In)});
pop({fifo, In, [H|T]}) -> {H, {fifo, In, T}}.

-spec empty({fifo, [], []}) -> true;
           ({fifo, list(), list()}) -> false.
empty({fifo, [], []}) -> true;
empty({fifo, _, _}) -> false.

test() ->
    N = new(),
    {2, N2} = pop(push(push(new(), 2), 5)),
    {5, N3} = pop(N2),
    N = N3,
    true = empty(N3),
    false = empty(N2),
    pop({fifo, [a|b], [e]}).
```

上面代码把队列定义成形如`{fifo, list(), list()}`的元组。你会注意到，我们并没有定义`fifo()`类型，主要是为了方便定义空队列和有元素队列的函数子句。`empty(...)`的类型规格说明反应出了这一点。

none()类型进一步说明

你会注意到，在函数 pop/1 的类型签名中，即使它的一个函数子句调用了 erlang:error/1，我们也没有使用 none()类型。

类型 none()的意思是，某个函数不会返回。如果函数可能失败，也可能返回一个值，那么把返回值类型和 none()同时放到其类型定义中是没有意义的。类型 none()是默认存在的，因此，联合类型 Type() | none()和单独的类型 Type()其实是一样的。

当你编写一个在被调用时会永远失败的函数时，可以使用 none()类型，例如，你想自己实现 erlang:error/1 函数。

看起来所有的类型规格说明都很合理。作为确认，我们运行一下 Dialyzer，看看结果：

```
$ dialyzer fifo_types.erl
  Checking whether the PLT /home/ferd/.dialyzer_plt is up-to-date... yes
  Proceeding with analysis...
fifo_types.erl:16: Overloaded contract has overlapping domains; such contracts are
currently
unsupported and are simply ignored
fifo_types.erl:21: Function test/0 has no local return
fifo_types.erl:28: The call
  fifo_types:pop({'fifo',nonempty_improper_list('a','b'),['e',...]})
   breaks the contract
   ({'fifo',In::[any()],Out::[any()]}) -> {term(),{'fifo',[any()],[any()]}}
 done in 0m0.96s
done (warnings were emitted)
```

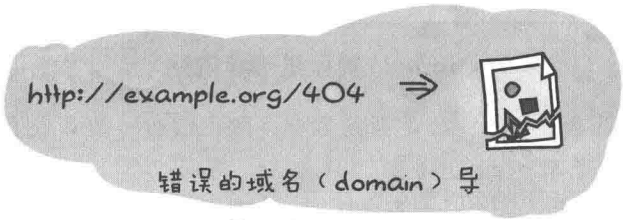
糟糕，出现了一堆错误，这些错误还不太容易理解。不过，至少我们知道如何处理第二条错误："Function test/0 has no local return."如果解决了后面一条错误，它就会消失。

现在，我们先来看看第一条错误——它和重叠域契约有关。在模块 fifo_types 的第 16 行，我们看到：

```
-spec empty({fifo, [], []}) -> true;
           ({fifo, list(), list()}) -> false.
empty({fifo, [], []}) -> true;
empty({fifo, _, _}) -> false.
```

重叠域是什么意思呢？我们需要参考一下两个数学概念：定义域和值域。简单来讲，定义域是函数的所有可能输入值的集合，值域是函数所有可能输出值的集合。"重叠域"指的是两个输入集合有部分重叠。

要找到问题的根源，需要了解一下 list()类型。list()类型和[any()]类型基本一样，两者都包含空列表。这就是造成重叠域的原因。当在类型中使用 list()时，就会和类型[]重叠。为了修正这个问题，我们需要把类型签名替换成如下形式：




```
-spec empty({fifo, [], []}) -> true;
           ({fifo, nonempty_list(), nonempty_list()}) -> false.
```

或者，也可以采用这种形式：

```
-spec empty({fifo, [], []}) -> true;
           ({fifo, [any(), ...], [any(), ...]}) -> false.
```

再次运行 Dialyzer，就不会出现这个警告了。

接下来看看下一条错误（我把它分成了多行显示）：

```
fifo_types.erl:28:
The call fifo_types:pop({'fifo',nonempty_improper_list('a','b'),['e',...]})
breaks the contract
({'fifo',In::[any()],Out::[any()]}) -> {term(),{'fifo',[any()],[any()]}}
```

这句话的意思是，在第 28 行，有一个对 pop/1 函数的调用，这个函数的推导类型和文件中指定的类型签名不相符：

```
pop({fifo, [a|b], [e]}).
```

上面就是这个调用。现在，出错消息表明，Dialyzer 发现了一个非良构列表（还不为空），完全正确——[a|b]就是一个非良构列表。出错消息中还提到这个调用违反了契约。我们需要解决下面出错消息中类型定义的违背问题：

```
{'fifo',nonempty_improper_list('a','b'),['e',...]}
{'fifo',In::[any()],Out::[any()]}
{term(),{'fifo',[any()],[any()]}}
```

对这个违背问题有 3 种解释。

- 类型签名是对的，调用也是对的，问题出在期望的返回值上。
- 类型签名是对的，调用是错误的，问题出在输入参数上。
- 调用是对的，类型签名是错误的。

肯定不会是第一种情况。我们根本没有使用返回值。因此，就只剩下第二、三种情况了。问题可以归结为我们是否允许使用非良构列表实现队列。这个决定应该由队列库的编写者做出，我可以肯定地说，我不想在代码中使用非良构列表。事实上，基本上没有程序员想使用非良构列表。那肯定就是第二种情况了：调用是错误的。解决这个问题有两个选择：删除掉这个调用，或者修正它：

```
test() ->
    N = new(),
    {2, N2} = pop(push(push(new(), 2), 5)),
    ...
    pop({fifo, [a, b], [e]}).
```

再次运行 Dialyzer：

```
$ dialyzer fifo_types.erl
  Checking whether the PLT /home/ferd/.dialyzer_plt is up-to-date... yes
  Proceeding with analysis... done in 0m0.90s
done (passed successfully)
```

完全正确。

30.7 类型导出

到目前为止，一切都还不错。我们能定义类型和签名，也拥有了额外的安全性和检验机制。那么，如果想在其他模块中使用队列该怎么做呢？dict、gb_trees 和 ETS 表等常用的模块又是怎么做的呢？如何用 Dialyzer 发现和它们相关的类型错误呢？

我们可以使用其他模块中定义的类型。通常，你得去查看文档才能找到它们。例如，ets 模块的文档（http://www.erlang.org/doc/man/ets.html）中，在“DATA TYPES”标题下含有如下的条目：

continuation()

函数 select/1 和 select/3 使用的延续，这个延续对调用者不透明。

match_spec() = [{match_pattern(), [term()], [term()]}]

匹配规格说明，参见上面的介绍。

match_pattern() = atom() | tuple()

tab() = atom() | tid()

tid()

表标识符，由函数 new/2 返回。

这些就是 ets 模块导出的数据类型。如果我们要定义一个类型规格说明，接受一个 ETS 表和一个键值，返回一条匹配的记录，就可以把它定义成下面这样：

```
-spec match(ets:tab(), Key::any()) -> Entry::any().
```

这样就完成了。

导出自定义类型的方式和导出函数基本上完全一样。要做的就是增加一个格式为 -export_type([*TypeName*/*Arity*]).的模块属性。例如，通过增加下面一行，就可以把 cards 模块中的 card()类型输出：

```
-module(cards).
-export([kind/1, main/0]).
```

```
-type suit() :: spades | clubs | hearts | diamonds.
-type value() :: 1..10 | j | q | k.
-type card() :: {suit(), value()}.

-export_type([card/0]).
...
```

此后，如果该模块对 Dialyzer 可见（把它加到 PLT 中，或者把它和使用它的模块一起进行分析），我们就可以在其他模块的类型规格说明中以 `cards:card()` 的形式对其进行引用。

不过，这种做法有一个缺点：以这种方式使用类型，不能阻止使用 `cards` 模块的人去窥探类型的实现细节。任何人都可以编写出这样的匹配代码：`{Suit, _} = ...`。这不是一种好的做法，因为这会妨碍我们将来更改 `cards` 模块的实现。我们更希望在 `dict` 和 `fifo_types`（如果导出类型的话）之类的数据结构模块中加以限制，从而杜绝这种做法。

Dialyzer 提供了一种类型导出方式，通过这种方式可以告诉使用者："你知道吗？你可以使用我的类型，但是绝不能窥探它们的内部实现！"只要把声明方式换一下即可，即把

```
-type fifo() :: {fifo, list(), list()}.
```

换成

```
-opaque fifo() :: {fifo, list(), list()}.
```

后面仍然用`-export_type([fifo/0])`导出类型。

把类型声明成`-opaque` 意味着只有定义该类型的模块才有权利查看类型的内部实现并对其进行修改。这就阻止了其他模块对它的值进行模式匹配，而不是完整地使用它，从而保证了（如果使用了 Dialyzer 的话）在改变类型实现时不会影响到这些使用模块。

保持冷静

在有些情况下，opaque 数据类型的实现不够严格，从而无法达到希望的目标，或者干脆存在实现错误（也就是说，有 bug）。

Dialyzer 仅在初次推导出某个函数的 success typing 时才会参考该函数的类型规格说明。

这意味着，如果不参考`-type` 信息，类型会过于通用，此时 Dialyzer 就会被某些 opaque 类型搞糊涂。例如，在分析 opaque 版本的 `card()` 数据类型时，Dialyzer 在推导后，可能会认为它是`{atom(), any()}`。有些模块正确地使用了 `card()` 类型，没有违反类型契约，但是 Dialyzer 还是会对它们发出违约警告。这是因为 `card()` 类型本身没有包含足够的信息，可以让 Dialyzer 对真实发生的情况做出正确的判断。

通常，如果在使用 opaque 数据类型时出现了错误，在元组中增加一个类型标签会有所帮助。把类型从-opaque card() :: {suit(), value()}.变成-opaque card() :: {card, suit(), value()}.，这样 Dialyzer 就应该能够正常使用 opaque 类型了。

Dialyzer 的作者目前正尝试着改善 opaque 数据类型的实现，并增强对它们的推导。在 Dialyzer 的分析过程中，他们也努力让用户提供的规格说明的地位更加重要，并更加信任这些规格说明，不过这项工作还在进行中。

30.8 OTP 行为类型

在第 14 章中，我们介绍了如何使用 behavior_info/1 函数来定义行为。导出了这个函数的模块的名字就是行为的名字，可以在另外一个模块中增加一条模块属性 -behavior(ModName).来实现行为回调函数。

例如，下面的代码展示了 gen_server 模块的行为定义：

```
behavior_info(callbacks) ->
    [{init, 1}, {handle_call, 3}, {handle_cast, 2}, {handle_info, 2},
     {terminate, 2}, {code_change, 3}];
behavior_info(_Other) ->
    undefined.
```

这种方式的问题是，Dialyzer 无法对它的类型定义进行检查。事实上，行为模块根本无法指定它期望回调模块实现的类型种类，因此 Dialyzer 无法对此提供任何帮助。

从 R15B 开始，对 Erlang/OTP 编译器进行了升级，使之可以处理一个新的模块属性：-callback。-callback 模块属性的语法和 spec 相似。如果用这个模块属性来指定函数的类型，编译器就会自动定义 behavior_info/1 函数，并且也会在模块的元数据中加入供 Dialyzer 使用的规格说明信息。例如，下面是 R15B 之后的 gen_server 类型定义：

```
-callback init(Args :: term()) ->
    {ok, State :: term()} | {ok, State :: term(), timeout() | hibernate} |
    {stop, Reason :: term()} | ignore.
-callback handle_call(Request :: term(), From :: {pid(), Tag :: term()},
                      State :: term()) ->
    {reply, Reply :: term(), NewState :: term()} |
    {reply, Reply :: term(), NewState :: term(), timeout() | hibernate} |
    {noreply, NewState :: term()} |
    {noreply, NewState :: term(), timeout() | hibernate} |
    {stop, Reason :: term(), Reply :: term(), NewState :: term()} |
    {stop, Reason :: term(), NewState :: term()}.
-callback handle_cast(Request :: term(), State :: term()) ->
    {noreply, NewState :: term()} |
    {noreply, NewState :: term(), timeout() | hibernate} |
    {stop, Reason :: term(), NewState :: term()}.
-callback handle_info(Info :: timeout() | term(), State :: term()) ->
    {noreply, NewState :: term()} |
    {noreply, NewState :: term(), timeout() | hibernate} |
```

```
    {stop, Reason :: term(), NewState :: term()}.
-callback terminate(Reason :: (normal | shutdown | {shutdown, term()} | term()),
                               State :: term()) ->
    term().
-callback code_change(OldVsn :: (term() | {down, term()}), State :: term(),
                                 Extra :: term()) ->
    {ok, NewState :: term()} | {error, Reason :: term()}.
```

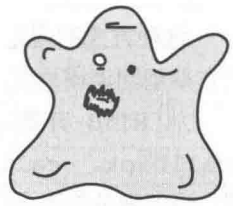
行为模块内部实现的变化不会影响到你的代码。不过，需要知道的是，一个模块不能同时使用-callback 模块属性和 behavior_info/1 函数——只能使用其中一个。这意味着，如果要创建自定义的行为，就要记得 R15 之前的 Erlang 版本和之后版本间的不同。

这种新写法的好处是，Dialyzer 可以对返回类型中的错误进行分析检查。

> **注意**　在 R15B 版本中，仅在回调模块包含了-behaviour 时而不是-behavior 时，Dialyzer 才会检查回调的类型，这是一个 bug，在后续的版本中已被修正。

30.9　多态类型

天哪，这是一个什么标题呀。如果你从来没有听说过多态类型（polymorphic type，也称为参数化类型），会感觉有些恐怖。还好，它实际上没有听起来那么复杂。



对于多态类型的需求来自于这样的事实：当我们在为不同的数据结构定义类型时，可能会希望它们能存储的类型更具体一些。例如，我们希望在本章前面实现的队列中，有时可以放入任何东西，有时只能放入纸牌，有时则只能存入整数。对于后两种情况，我们希望在把浮点数放到整数队列中，或者把占卜牌放到纸牌队列中时，Dialyzer 能够给出警告。

仅靠前面介绍的类型使用方式是无法满足这项要求的。

多态类型是这样一种类型，可以用其他类型对其进行“配置”。很幸运，我们已经知道了如何使用它的语法。之前，我曾经说过，我们可以把整数列表定义为[integer()]或者 list(integer())——这就是多态类型。多态类型接受类型作为参数。

要使队列只接受整数或者纸牌，可以把它的类型定义成下面的形式：

```
-type queue(Type) :: {fifo, list(Type), list(Type)}.
-export_type([queue/1]).
```

当另外一个模块想使用类型 queue /1 时，需要为它提供参数。这样，cards 模块中构建一副新纸牌的函数签名如下：

```
-spec new() -> fifo:queue(card()).
```

Dialyzer 会尝试对模块进行分析，确保该模块遵从这个要求，在队列中仅仅存放 card()类型的值。

30.9.1　我们买了一个动物园

为了展示一下多态类型的使用，假设我们决定购买一个动物园来庆祝一下即将完成这本书的

学习。在动物园中，有两个动物：一只小猫熊和一只乌贼（是的，这是个很小的动物园，不过门票可是不便宜哟）。

由于我们是程序员，而程序员出于懒惰的原因，喜欢自动化一切，因此我们决定自动为动物们喂食。在做了一番研究之后，发现小猫熊可以吃竹子、某些鸟类、蛋类以及浆果，也发现乌贼会捕猎抹香鲸，因此决定就用 zoo.erl 模块为它们提供这些食物：

```
-module(zoo).
-export([main/0]).

feeder(red_panda) ->
    fun() ->
        element(random:uniform(4), {bamboo, birds, eggs, berries})
    end;
feeder(squid) ->
    fun() -> sperm_whale end.

feed_red_panda(Generator) ->
    Food = Generator(),
    io:format("feeding ~p to the red panda~n", [Food]),
    Food.

feed_squid(Generator) ->
    Food = Generator(),
    io:format("throwing ~p in the squid's aquarium~n", [Food]),
    Food.

main() ->
    %% 随机种子
    <<A:32, B:32, C:32>> = crypto:rand_bytes(12),
    random:seed(A, B, C),
    %% 动物园为小猫熊和乌贼购买了喂食器
    FeederRP = feeder(red_panda),
    FeederSquid = feeder(squid),
    %% 进食时间到
    %% 下面应该出现错误
    feed_squid(FeederRP)
    feed_red_panda(FeederSquid).
```

这段代码中使用了函数 feeder/1，该函数接收一个动物的名字，返回一个喂食器（一个会返回食物的函数）。给小猫熊喂食时需要使用小猫熊喂食器，给乌贼喂食时需要使用乌贼喂食器。使用像 feed_red_panda/1 和 feed_squid/1 这样的函数定义，是无法在用错喂食器时发出警告的。即使在运行时进行检查，也不行。因为一旦开始喂食，一切就都晚了：

```
1> zoo:main().
throwing bamboo in the squid's aquarium
feeding sperm_whale to the red panda
sperm_whale
```

啊，不！我们的动物不能吃这样的食物！也许类型能提供帮助。我们可以设计如下的类型规

格说明，其中用到了多态类型：

```
-type red_panda() :: bamboo | birds | eggs | berries.
-type squid() :: sperm_whale.
-type food(A) :: fun(() -> A).

-spec feeder(red_panda) -> food(red_panda());
            (squid) -> food(squid()).
-spec feed_red_panda(food(red_panda())) -> red_panda().
-spec feed_squid(food(squid())) -> squid().
```

food(A)是我们感兴趣的类型。A 是一个自由类型，会在以后决定。通过声明 food(red_panda())和 food(squid())，我们在 feeder/1 的类型规格说明中对食物类型做了限定。此后，食物类型就变成了 fun(() -> red_panda())和 fun(() -> squid())，而不是一个返回未知类型的抽象函数。把这些规格说明加到文件中，运行 Dialyzer，会有如下输出：

```
$ dialyzer zoo.erl
  Checking whether the PLT /Users/ferd/.dialyzer_plt is up-to-date... yes
  Proceeding with analysis...
zoo.erl:18: Function feed_red_panda/1 will never be called
zoo.erl:23: The contract zoo:feed_squid(food(squid())) -> squid() cannot be right
because
 the inferred return for feed_squid(FeederRP::fun(() -> 'bamboo' | 'berries' | 'birds'
|'eggs'))
 on line 44 is 'bamboo' | 'berries' | 'birds' | 'eggs'
zoo.erl:29: Function main/0 has no local return
 done in 0m0.68s
done (warnings were emitted)
```

错误提示完全正确。多态类型万岁！

30.9.2　注意事项

虽然上面的例子很能说明问题，但是对代码做些微小的改动就会让 Dialyzer 失效。例如，把 main/0 函数的代码进行如下更改：

```
main() ->
    %% 随机种子
    <<A:32, B:32, C:32>> = crypto:rand_bytes(12),
    random:seed(A, B, C),
    %% 动物园为小猫熊和乌贼购买了喂食器
    FeederRP = feeder(red_panda),
    FeederSquid = feeder(squid),
    %% 进食时间到
    feed_squid(FeederSquid),
    feed_red_panda(FeederRP),
    %% 下面应该出现错误
    feed_squid(FeederRP),
    feed_red_panda(FeederSquid).
```

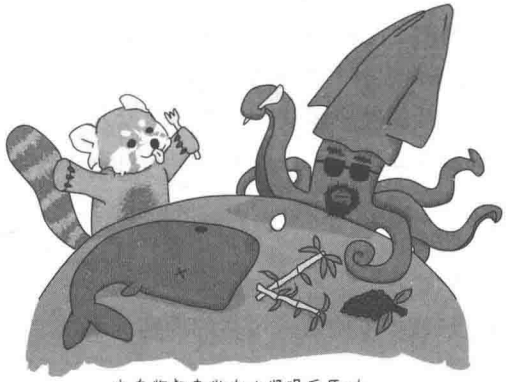
此时，结果就会不同。在使用错误的喂食器类型调用函数之前，先用正确的喂食器类型进行调用。从 R15B01 版本开始，Dialyzer 都不会发现这段代码中的错误。从观察到的现象来看，在

一个函数体中，只要对某个函数的调用成功了，Dialyzer 就会忽略掉同一段代码中对这个函数的后续错误使用。

对许多喜欢静态类型的人来说，这有些糟糕，不过我们已经被非常清楚地警告过了。下面的引用来自论文“Practical Type Inference Based on Success Typings”，在本章开头曾经提到过：

> ……success typing 是这样一种类型签名，它是对函数能够求值的类型集合的一种近似。类型签名的定义域包含了能够作为函数参数的所有可能值，它的值域包含了针对该定义域的所有可能返回值……
>
> ……对于静态类型爱好者来说，这可能太弱了，但是 success typing 具有这样的特点，它们刻画了如下事实：如果以 success typing 不允许的方式使用了某个函数（例如，在调用函数时，使用了参数 $\overline{p}$，而 $\overline{p} \not\in \overline{\alpha}$），那么这次调用一定会失败。这正是一个从不会喊“狼来了”的缺陷检测工具所需要的属性。同时，success typing 还可以对程序进行自动的文档化，因为它绝不会遗漏掉一些可能的——不管是多么的无意——函数使用方式。

再次提醒，Dialyzer 在分析代码时是非常乐观的，要想有效使用它，牢记这一点是至关重要的。



当有监督者做你的坚强后盾时，
谁还会在乎食物中毒？

30.10 Dialyzer，我的好朋友

在使用 Dialyzer 的过程中，它有时会唠叨个不停，这可能会让你不想使用它，但是事实证明，在用 Erlang 编程时，Dialyzer 是你真正的朋友。记住，Dialyzer 几乎不会犯错，而你则会时不时地犯错。你也许会认为有些错误根本不值一提，但是和许多其他类型系统不同，Dialyzer 只在确认自己是对的时候才发声，并且 Dialyzer 的代码质量很高，基本没有什么 bug。也许，Dialyzer 会让你感到沮丧，强迫你保持谦虚，但是它会把那些糟糕、不整洁的代码扼杀在摇篮里。

30.11 朋友们，就这么多

嗨，这本书的内容写完了。非常感谢你阅读本书。没有什么可多说的了，不过如果你想了解更多的主题，或者想从我这里得到一些概括性的总结，可以阅读本书的后记。

后记

你终于读到后记了。干得好。如果你决定把 Erlang 作为自己的开发语言，想了解更多的内容，我会在这里给出一些有价值的信息，不过在这之前，我想先花点时间聊聊在编写本书时的一些情况。那真是一段不堪回首的历程。整个过程历时 3 年，期间不仅要辛苦写作，还要全职地工作和学习，以及处理其他生活事宜（如果我有小孩的话，他们肯定会由于照顾不周而夭折）。

由于我把这本书放到了网上，再加上一些运气成分和其他努力，我得到了一份工作，内容涉及 Erlang 培训、课件编写和软件开发。这份工作让我可以游历全球并遇到了很多非常有趣的人。编写这本书耗费了我大量的精力、金钱和时间成本，但是它也用完全可以想象的方式给予了我 10 倍的回报。

然后，这本书要被出版，需要做更多的工作。虽然在本书的“致谢”中我对相关的人已经做了感谢，但是在此我还是要对所有 Erlang 社区的朋友再次表示感谢。他们让我学到了很多知识、无偿地对本书内容进行仔细的评审、修正拼写错误并在书面英语和写作方面给予了我很大的帮助。我也要感谢 No Starch 出版社的整个团队，他们甚至投入了更多的时间对这本书进行专业的编辑。最后，再次感谢 Jenn（我的女朋友），她对书中的所有插图都进行了修改，让它们更适于打印。

其他 Erlang 应用

由于各方面的限制，我无法在本书中放入过多的内容。不过，本书目前的内容已经不少了。这本书花费了几年时间才编写完成，我很疲惫，不过很高兴它终于写完了（现在有这么多的自由时间，干些什么好呢？），但是还有不少主题是我本想包含在本书中的。下面是一些应用的快捷列表，你可以在 Erlang 自带的文档中找到它们。

跟踪 BIF 和 DBG

Erlang VM 是可以被彻底跟踪的。在无法理解某个 bug 或者调用栈跟踪时该怎么办呢？只要打开几个跟踪开关，VM 就会向你敞开一切。DBG 是 Erlang 自带的一个应用，它构建在这些跟踪 BIF 之上。消息、函数调用、函数返回、垃圾回收、进程创建和死亡等一切都是可以跟踪和观测的。同时，对于一门像 Erlang 这样的并发语言来说，DBG 要优于任何其他的调试器。它最好的特性是什么呢？它可以直接在 Erlang 中进行跟踪，这样你就可以让 Erlang 程序跟踪它们自己！如果你查看了这些函数，觉得它们有些难以理解，那么只使用 `sys` 模块的跟踪函数也是可以的。这些函数只能用于实现了 OTP 行为的进程，但是通常来说已经够用了。

性能分析

Erlang 中自带了几个不同的性能分析工具可以对程序进行分析，找出各种瓶颈。fprof 和 eprof 可以用于运行时间分析，cprof 可以用于函数调用分析，lcnt 用于锁分析，percept 用于并发情况分析，cover 用于代码覆盖分析。有趣的是，这些工具基本上都使用了 Erlang 语言中的跟踪 BIF。

其他自省工具

Erlang 中也有一些和 Unix 操作系统中的 top 命令类似的工具，如 etop，它属于 observer 应用。你也可以使用 Erlang 的调试器，不过我推荐使用 DBG，不推荐使用调试器。你可以使用 observer 应用来浏览节点的完整监督树。

文档

EDoc 是一个可以把 Erlang 模块转换成 HTML 文档的工具。它支持标注，并且提供了一些定义特定页面的方法，可以用构建小型网站的方式文档化代码。它和 Java 用户使用的 Javadoc 工具类似。

图形界面

wx 应用是 Erlang 中多平台 GUI 支持的新标准。我 GUI 方面的知识比较糟糕，因此在本书中没有介绍这个应用，对所有读者来说，这可能是一件好事。

其他 Erlang 库

Erlang 中还自带了许多有用的库：加密工具、Web 服务器、Web 客户端工具、各种协议实现等。可以从 http://www.erlang.org/doc/applications.html 得到更全面的介绍。

社区贡献的库

Erlang 社区也贡献了大量的库。我在本书中没有介绍它们，一方面是因为它们时常会发生变化，另一方面我也不想顾此失彼。下面是这些应用的简介（在书中无法很好地使用链接，可以从 http://learnyousomeerlang.com/conclusion 处获得实际的链接）。

- Rebar 和 Sinan，用于构建系统。
- Redbug，提供了一种更友好的跟踪方法。
- Gporc，实现了一个更强大、灵活的进程注册库。
- Mochiweb、Cowboy 和 Yaws，用于 Web 服务器。
- riak_core，一个非常强大的 Eralng 分布式库。
- lhttpc，Web 客户端。
- PropEr、QuickCheck 和 Triq，强大的基于属性（property-based）的测试工具（你有必要去尝试一种）。
- Entop，类似 top 的工具。
- 众多的 JSON 库（mochijson2、jsx、ejson 等）。
- UX，用于 Unicode 处理，其中有些通用的与 Unicode 相关的算法会被加入语言中（计划

在 R16B 中加入）。

- Seresye 和 eXAT，人工智能工具。
- 数据库客户端库。
- Lager，健壮的日志系统，和 Erlang 的错误日志绑定在一起。
- Poolboy，一个通用的、基于消息的进程池。

还有许多库没有在这里列出来。社区库的内容太多，完全可以独立成书。

你的想法非常吸引我，我想订阅你的最新文章

我有一个博客，网址是 http://ferd.ca。我会在上面撰写各种类型的文章（至少我想这么做），不过最终会聚集到 Erlang 相关的内容，因为我一直在用 Erlang。

本书结束了吗

没有结束，还有一个附录！

附录

Erlang 语法

许多 Erlang 的初学者会尝试着理解 Erlang 的语法和程序，但是一直无法适应。我看到过，也听到过很多对 Erlang 语法的抱怨，像“蚂蚁粪球一样的符号”（对，、；和.的一种主观搞笑说法）太令人讨厌了等。

Erlang 的语法来自 Prolog。虽然这一点解释了 Erlang 当前语法形式的成因，但是并不会神奇地让人们喜欢 Erlang 的语法。我根本没指望有人在听到这个解释后会说，“哦，原来是 Prolog，我明白了。非常合理！”因此，在阅读 Erlang 代码时，我给出 3 种可能让代码更容易理解的方法。

A.1 模板

模板方法是我本人最喜欢的。要理解它，你首先必须去除代码行的概念，基于表达式进行思考。表达式是会返回某些东西的任意 Erlang 代码块。在 shell 中，句号（.）是表达式的结束标志。输入了 2 + 2 之后，必须要增加一个句号（然后按回车键），才能运行这个表达式，并返回一个值。

在模块中，句号是结构形式（form）的结束标志。模块的属性和函数定义都是结构形式。结构形式不是表达式，因为它们不会返回任何东西。这就是为什么它们的终结方式和其他所有东西都不同的原因。由于结构形式不是表达式，所以可以说，在 shell 中使用.来终结表达式的做法是不标准的。因此，我建议在使用这种方法阅读 Erlang 代码时不要考虑 shell 的情况。

第一条规则是，逗号（,）是表达式的分隔符：

```
C = A+B, D = A+C
```

这一条很简单。不过，要注意，`if ... end`、`case ... of ... end`、`begin ... end`、`fun() -> ... end` 和 `try ... of ... catch ... end` 都是表达式。下面的写法是合法的：

```
Var = if X > 0 -> valid;
         X =< 0 -> invalid
      end
```

你会从 `if ... nd` 中得到一个返回值。这就解释了为什么有时会看到这种语言结构后面跟着

一个逗号。它只是意味着还有另外一个表达式会在这个表达之后求值。

第二条规则是，分号（;）有两个作用。第一个作用是分隔不同的函数子句：

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

第二个作用是分隔像 if ... end、case ... of ... end 之类的表达式中的不同分支。

```
if  X < 0 -> negative;
    X > 0 -> positive;
    X == 0 -> zero
end
```

这个作用可能最令人困惑，因为表达式的最后一个分支后面不需要跟着分号。这是因为;是用来分隔分支的，并不是终结分支。基于表达式进行思考，不要基于代码行。有人觉得把上面的表达式用下面这种方式编写更能够说明其分隔符的作用，可读性也更好：

```
if  X < 0 -> negative
 ; X > 0 -> positive
 ; X == 0 -> zero
end
```

这种写法凸显了分隔符的作用。它处在分支和子句之间，并不是在它们后面。

由于分号是用来分隔表达式分支和函数子句的，因此当 case 语句后面跟着另外一个表达式时，它的后面可以跟着一个逗号，当 case 语句位于一个函数子句定义的最后时，它的后面可以跟着一个分号。

必须要把 C 和 Java 之类的语言中基于行进行终结的逻辑彻底忘掉，而是要把代码看作有待填充的通用模板（这就是为何叫模板方法的原因）：

```
head1(Args) [Guard] ->
    Expression1, Expression2, ..., ExpressionN;
head2(Args) [Guard] ->
    Expression1, Expression2, ..., ExpressionN;
headN(Args) [Guard] ->
    Expression1, Expression2, ..., ExpressionN;
```

这些规则非常合理，不过你得进入一种不同的阅读模式。这也正是困难所在——从代码行和代码块迁移到预先定义好的模板。如果你仔细想想，就会发现 for(int i = 0; i >= x; i++) {...}（或者甚至是 for(...);）之类的语法在与支持它们的语言中的其他大部分语法结构比较时显得非常奇怪。我们只是太习惯于看到这些语法结构，以至于都见怪不怪了。

A.2 英语句子

英语句子方法是把 Erlang 代码和英语进行比较。虽然我本人并不喜欢这种方式，但是我意识到人们在理解逻辑概念时确实会采用不同的方法，而这种方法是被交口称赞的。

假设你正在列物品清单。嗯，不。别想了，直接读吧。

```
I will need a few items on my trip:
```

```
if it's sunny, sunscreen, water, a hat;
if it's rainy, an umbrella, a raincoat;
if it's windy, a kite, a shirt.
```

转换成 Erlang 代码后有点儿相似：

```
trip_items(sunny) ->
    sunscreen, water, hat;
trip_items(rainy) ->
    umbrella, raincoat;
trip_items(windy) ->
    kite, shirt.
```

其中，只要把物品替换成表达式就行了。像 `if ... end` 之类的表达式可以看作嵌套列表。

A.3 与、或、完成

有人在#erlang 上给我建议了英语句子方法的另外一个变种，我觉得它是最为优雅的一个。用户只要简单地把“蚂蚁粪球一样的符号”读成如下含义即可：

- 把, 读成与（and）；
- 把; 读成或（or）；
- 把. 读成完成（done）。

这样，函数定义就可以被当成一组嵌套的逻辑语句和断言。

A.4 结论

有些人永远也不会喜欢“蚂蚁粪球一样的符号”，也不喜欢在交换代码行时需要更改行尾符号。尽管我觉得在风格和偏好方面很难做出改变，但是我还是希望这个附录会有些用处。毕竟，语法只是令人生畏，绝对谈不上困难。

Erlang趣学指南

Learn You Some Erlang for Great Good!

Erlang是编写健壮的并发应用的最佳语言，但是Erlang语言奇怪的语法和函数式设计让很多初学者望而却步。幸运的是，现在有了一件对抗Erlang恐惧症的新武器——《Erlang趣学指南》！

Erlang专家Fred Hébert从简单概念入手，让你轻松掌握Erlang的基础知识：你会学习Erlang的非传统语法、类型系统以及基本的函数式编程技术。一旦你理解了这些简单的概念，就可以学习Erlang语言中最核心的内容——并发、分布式计算、代码热加载以及那些能够让Erlang成为当今有见识的开发者之间热门话题的黑魔法了。

随着深入到Erlang语言解决实际问题的奇妙世界，你将学到：

- 使用EUnit和Common Test对应用进行测试；
- 使用OTP框架构建和发布应用；
- 传递消息、抛出错误异常以及在多节点上启动/终止进程；
- 使用Mnesia和ETS存储和读取数据；
- 使用TCP、UDP和inet模块进行网络编程；
- 编写分布式并发应用带来的简单快乐和其中的潜在陷阱。

本书行文轻松、幽默，示例新颖、实用，是你进入偶尔令人抓狂，但一直令人兴奋的Erlang世界的最佳指引！

作者介绍

Fred Hébert是一位自学成才的程序员，具有Web前端、Web服务开发经验以及全面的后端多语言编程经验。他撰写的Erlang在线教程*Learn You Some Erlang for Great Good!*被公认为最好的Erlang学习教程。在Erlang Solutions工作期间，他撰写过Erlang培训资料，并在西方国家到处教授Erlang编程。目前，他在用Erlang开发一款实时竞价平台（AdGear），他被提名为2012年度Erlang User。

译者介绍

邓辉 独立咨询顾问，捷中科技首席架构师，有10年Erlang编程经验。《敏捷软件开发：原则、模式与实践》一书的译者，Erlang领域两本重要著作《面对软件错误构建可靠的分布式系统》和《硝烟中的Erlang》中文译本的审校者和译者。对指示语义、类型理论、程序语言理论和分布式系统设计有浓厚的兴趣。目前在研究学习TLA+。

孙鸣 中兴通讯资深架构师，具有丰富的电信软件架构设计经验。《敏捷软件开发：原则、模式与实践》（C#版）、《硝烟中的Erlang》等书的译者。对函数式编程有浓厚的兴趣，平时喜欢设计各种各样的小语言。最喜欢的编程语言是Scheme和Erlang，并用Erlang语言重写了*Structure and Interpretation of Computer Programs*一书中的全部代码。

异步社区 www.epubit.com.cn
新浪微博 @人邮异步社区
投稿/反馈邮箱 contact@epubit.com.cn

no starch press

ISBN 978-7-115-43190-5

分类建议：计算机／程序设计／Erlang
人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-43190-5
定价:79.00 元