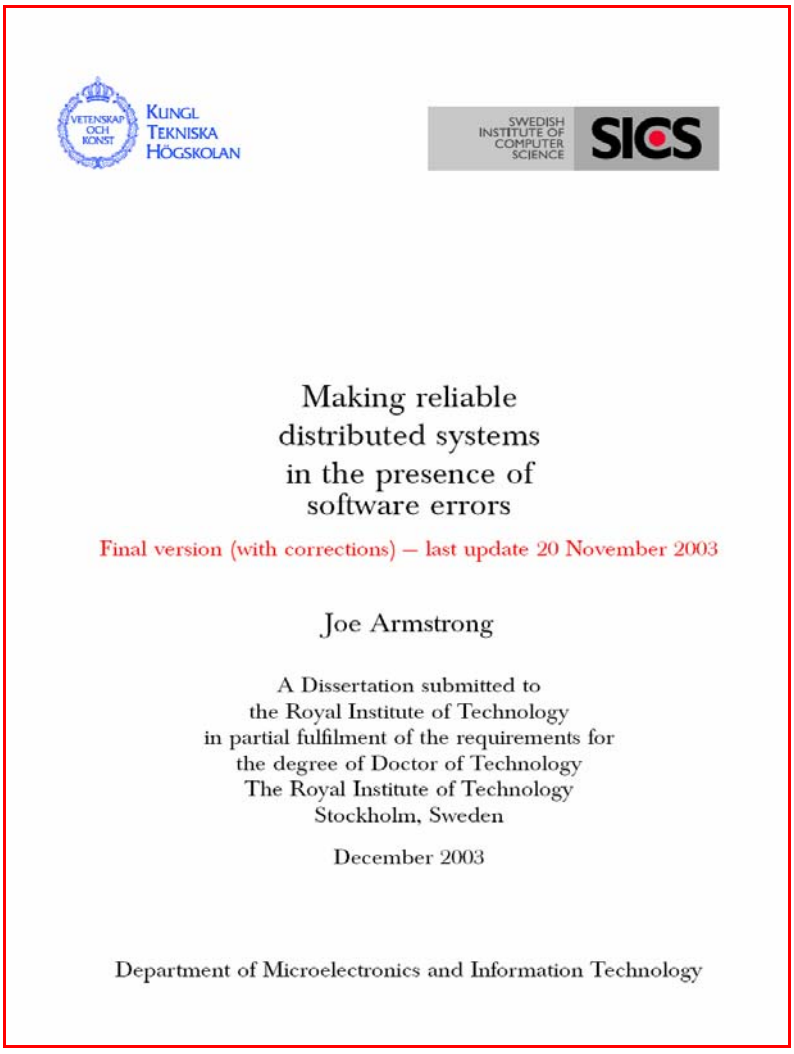
面对软件错误构建可靠的分布式系统

（瑞典）Joe Armstrong 撰

段先德 译
邓　辉 审



KUNGL TEKNISKA HÖGSKOLAN

SWEDISH INSTITUTE OF COMPUTER SCIENCE SICS

Making reliable
distributed systems
in the presence of
software errors

Final version (with corrections) – last update 20 November 2003

Joe Armstrong

A Dissertation submitted to
the Royal Institute of Technology
in partial fulfilment of the requirements for
the degree of Doctor of Technology
The Royal Institute of Technology
Stockholm, Sweden

December 2003

Department of Microelectronics and Information Technology

原文链接：http://www.erlang.org/download/armstrong_thesis_2003.pdf

译 感

近年来，“多核”、“分布式计算”、“集群计算”、“大并发量处理”等名词逐步从实验室中的概念走向了大众社会的应用，并给许多公司带来了商业上的巨大成功。一个新的信息处理时代已经悄然来临，召唤着新的适应这个时代的软件开发方法和工具。其实早在 1986 年，在 Ericsson 就有一些专家开始了如何编写出可靠的分布式系统方面的探索和研究。这些研究成果，成就了可靠性达到 99.9999999%的目前世界上最复杂的 ATM 交换机，并给我们带来了 Erlang/OTP 这套开源的开发工具和平台。Joe Armstrong 先生就是 Erlang 的主要发明人，而本文就是 Joe Armstrong 的博士论文。

之初学习 Erlang 编程的时候，为了深入了解一下 Erlang 的设计哲学，我决定把这篇论文精读一遍。我想无一遗漏地精读的方式，就是把它翻译一回。邓辉先生（《敏捷软件开发——原则、模式与实践》中文版译者）鼓励我说，翻译吧，也算是给国内的 Erlang 这个小圈子做点贡献。于是似乎有了一些责任。既然是要公开出来的，任何细节就更不能毛糙了，也就迫使我查阅了许多背景资料，而这些论文以外的知识对于我平时的 Erlang 编程颇有助益，算是意外之得了。

跟其他的理工类博士论文不一样，本论文并没有堆砌大量的公式推导，而是以平铺直叙的方式，在解答“如何在存在软件错误的情况下编写出具有合理行为的软件？”这一核心问题的过程中，详细地阐述了 Erlang/OTP 的设计哲学。论文分析了构建可靠的分布式系统的一些系统需求、语言需求、库需求，介绍了完全针对这些需求而打造的 JAM 系统、Erlang 语言和 OTP 开发库。本论文的重点不在于 JAM 的设计，不在于 Erlang 语言的描述，也不在于 OTP 库的介绍，虽然这些方面都有很多值得讨论的课题。本论文把重点放在了可容错的架构上，如何构建软件运行的模型？如何进行错误检测和恢复？如何实际编写出可容错的系统？本文的重点在于探讨这一系列的设计思想的问题。这是一门软件开发哲学，也是一个软件世界观。

不得不提的是“一切皆进程”的 Erlang 世界观。作者把软件所有要处理的事务划分成一系列层次化的任务，每个任务有一个“强隔离的进程”来执行，进程之间没有任何共享状态，只能通过“消息传递”来通信。这种“强隔离的进程”

不仅可以更真实地描述现实世界的信息处理过程，还成为软件错误发生时保护系统的可靠性的最有力模型。

也要说说本论文关于故障处理的论述。由于业务处理都在一个个“强隔离的进程”中——作者把它们称为“工作者”，就防止了一个进程出错会传播到其他的进程。业务处理进程的运行状况由另外专门的进程来看护——作者把它们称为“监视者”。“工作者”和“监视者”组成一个层次化的监督模型，使得一个进程发生故障时，整个系统可以作出相应的调整，保障系统最大限度地提供服务。

还要说说 behaviour。本论文无意中介绍了 OTP 库集中的 behaviour 库的设计思想，即将程序的并发处理和顺序化处理分开。本文还以 gen_server 为例，展示了如何把并发部分抽象出来，如何让程序从最初的“脆弱”演化到“可容错”，体味一下，会受到启发。

罢了罢了，关于本论文的内容我不能说太多，言多必失，恐怕淡漠了任何一方面，也许每个人看到的重点都不一样。总之，非常感谢 Joe Armstrong 给我们奉献了这篇论文。

在本译稿完成的时候，Joe Armstrong 关于 Erlang 编程的的新书《Programming Erlang》也已经出版面世了，引起了软件开发主流们不小的震动，Ralph Johnson 甚至预言 Erlang 会是下一个 Java。不管这种预言会不会成真，想深入学习 Erlang 编程的朋友一定不要错过这部伟大的著作。

非常感谢邓辉先生，他详细地审阅了本译文的初稿，提出了许多宝贵的意见，使得本译文更贴合论文原意。

无论我怎么努力，终究水平有限，肯定还存在许多对原文理解错误或中文表达欠妥之处，恳请读者指正。译者联系方式：sanderisme@hotmail.com。

段先德

2007 年 9 月 上海

面对软件错误构建可靠的

分布式系统

（Making reliable distributed systems in the presence of software errors）

2003 年 11 月 20 日最终修订版

Joe Armstrong

本论文作为工学博士

学位之要件提呈皇家技术研究院

瑞典・斯德哥尔摩

皇家技术研究院

2003 年 12 月

微电子与信息技术学部

献给 Helen、Thomas 和 Claire

摘要

本论文所描述的成果源于 1981 年开始的一个研究项目，该项目的目的是探索更好的编写电信应用软件的方法。电信类应用都是一些大型的程序，虽然经过了严密的测试，但是投入运行后还是难免会有许多错误。我们假设这些程序不可避免地会含有错误，进而寻求在软件包含错误的情况下构建可靠系统的方法。

该项研究的成果是开发出了一种新的编程语言（叫做 Erlang），一门设计方法学，和一个用以构建健壮系统的程序库（叫做 OTP）。就在本论文写作的时候，本文所描述的技术已经在 Ericsson 和 Nortel 的多款主流产品中被使用。也许还有许多小型公司正在为了发掘这一技术而成立。

本文主要关注的是在程序自身包含有错误的情况下如何构建出可靠的系统这一问题。构建这样的系统对所采用的任何一种编程语言都有一些特殊的要求。这里会讨论这些对语言的具体的特殊要求，并将展示 Erlang 是如何满足这些要求的。

这些要求可以在编程语言中解决，也可以在语言所附带的标准库中解决。我将论证在构建可容错系统时，哪些要求应该在语言中解决，而哪些要求可以在标准库中解决。这些合起来构成了构建可容错软件系统的基础。

实践见真章，没有得到实践证明的理论是不完整的。为了证明这些思想在实践中确实有用，我也列举了许多已经在大型上列产品中成功应用该技术的案例。到目前为止，使用该技术的最大的项目是 Ericsson 的一款重要产品，它包含超过 100 万行的 Erlang 代码。这款产品（AXD301）也是公认的 Ericsson 最可靠的产品之一。

最后，我印证了一下我们是否达到了寻找一种编写电信应用软件的更好的方法的目的，我还指出了我们的系统可以在哪些方面进行进一步的提高。

目录

1 绪论

我们如何去编写在有软件错误的条件下有合理行为的软件呢？这是我这篇论文想要回答的核心问题。大型的系统往往在交付的时候还存在着许多软件错误，然而我们却奢望它们能够运行正常。

系统有不完善的环节，而我们又希望它可靠，这就对系统提出了一定的要求。这些要求是能够被满足的，要么在所采用的编程语言中，要么在应用程序所调用的标准库中。

在本文中，我会列举出我认为的可容错系统所必须具备的**本质特性**，我还将展示这些特征在我们的系统（Erlang/OTP）中如何被满足。

某些本质特性是在我们的编程语言（Erlang）中被满足的，而另外一些则是在用 Erlang 编写的库模块中被满足的。语言和库合起来构成了构建可靠软件系统的基础，使得即使存在编程错误，系统仍然能够按照合理的方式运行。

说了本论文是关于什么的，我也该说说本文是不关于什么的。本论文并没有详述作为构造可容错系统的构建块的诸多算法的细节，因为算法本身并不是本文所关注的重点，本文关注的是用以表达这些算法的编程语言。本文也不关注构建可容错系统在硬件方面和软件工程方面的要求。

本文真正关注的是软件的容错性在语言、库和操作系统方面的要求。Erlang 属于一种**纯消息传递语言**——即一种基于独立性很强的并行进程的语言，我们的编程模型广泛使用了**速错**（fail-fast）进程。这项技术在构建可容错系统的硬件平台中被普遍使用，但是在软件设计方面却用得不多。这主要是因为传统的编程语言并不允许不同的软件模块以彼此互不干扰的方式存在。当前普遍使用的是多线程的编程模型，该模型中资源是共享的，这样就造成了线程很难真正隔离起来，就可能导致一个线程中的错误会传播到另一个线程中，这就破坏了系统内部的坚固性。

1.1 背景介绍

本文介绍的工作始于 1981 年的 Ericsson 计算机科学实验室（CSLab）。我是 1985 年加入 CSLab 的。这里介绍的是我在 1981 年至 2003 年间的工作，在此期间，我和我的同事们开发了 Erlang 编程语言和 OTP，也用 Elrang 写过一些其它应用程序。

Erlang/OTP 系统取得今天的成果，归功于许多人的集体努力。没有他们的出色才干和来自用户们的反馈，Erlang 也不可能有今天的成就。由于系统包含如此之多的部分，所以很难准确地说到底是谁在什么时候做了什么，是谁最初提出了某一项革新思想。在致谢中我将尽我所能地对大家的每一份贡献致以谢意。

Erlang/OTP 系统纪年表如下：

- 1981 年——Ericsson CSLab 成立。该实验室的一个目标就是“为未来的软件系统提出一些新的架构（architecture）、概念（concepts）和结构（structure）”[29]。
- 1986 年——我开始了后来称之为 Erlang 的语言方面的工作，虽然当时这种语言还没有名字。该语言产生于一项将并发进程加入到 Prolog 的试验（关于这项工作，在参考文献[10]中有详述），在那个阶段，我并没有想要设计一种新的编程语言，我只是关注如何编写 POTS（Plain Old Telephony Service）的程序，在那时看来编写 POTS 程序的最好的方法就是用扩展后的增加了对并行进程支持的 Prolog。
- 1987 年——Erlang 被最初提起。到 1987 年，Erlang 这个术语才被提炼出来（大概是由 CSLab 的领导 Bjarne Däcker 提出的）。这一年的年末，推出了 Erlang 的一个 Prolog 实现版本。这个版本的 Erlang 嵌入在 Prolog 中，运用了 Prolog 的中缀算子（infix operator），并且没有自己的语法。

  快到 1987 年年末的时候，Erlang 的第一个正式试验——一个原型工程——开始了，由 Kerstin Ödling 领导的几位 Ericsson 的工程师在 Bollnora 进行。他们用 Erlang 为一个叫“ACS/Dunder”的系统做原型。

ACS 是为了实现 Ericsson MD110 专用自动分组交换机（private automatic branch exchange， PABX）而设计的一个软件架构。

该项工程准备用Erlang和ACS架构实现PABX的许多典型特征，并且与PLEX[1]的编程能力（即做相同事情估计所需的时间）进行了比较。

目前 Erlang/OTP 系统的许多思想就可以追溯到当时的这个项目。

- 1988 年——就在 1988 年，Erlang 适宜于做电信系统编程的认识变得清晰了。在 Bollmora 的小组用 Erlang 写那个应用的同时，CSlab 团队进行了扩充，Robert Virding 和 Mike Williams 也加入进来一起改进 Erlang 系统。

  改进Erlang效率的一个尝试就是将其编译到一种并行逻辑编程语言Strand。从Erlang到Strand的编译过程在参考文献[37]第 13 章有描述。从Erlang到Strand的编译将Erlang的性能提高了 6 倍，但是这个项目仍然被认为是一个令人沮丧的失败[2]。

- 1989 年——ACS/Dunder 项目开始有了一些结果。Bollmora 的小组声称他们使用了 ACS/Dunder 和 Erlang 后，使得开发的效率比使用 PLEX 提高了 9 至 22 倍。这个结论仅仅是建立在对 Ericsson MD110 的 10%的原型化试验的基础上的，而这些数字引起了激烈的争论（取决于你是否信任 Erlang）。

  Bollmora 的小组估计要想将 ACS/Dunder 的原型机变成商用产品，他们就必须在性能上再提高 70 倍（而这一点我们马上就承诺我们可以做到）。

  为了提高 Erlang 的性能，我设计了 JAM 机（Joe’s abstract machine）。JAM 机的设计大致上是基于 Warren 虚拟机[68]的。因为 Erlang 最初是从 Prolog 扩展而来，所以提高 Prolog 的效率的一些技术应该也可以应用于 Erlang。这种直觉后来被证明是正确的。JAM 机采用了与 WAM 相似的方法，譬如加入并行进程、消息传递、错误检测机制，

[1] PLEX是用来编写MD110 的一种编程语言。
[2]关于Strand的书中并没有记录事实的真相。

省略了回溯机制。模式匹配（pattern matching）的编译本质跟 WAM 也是一样的。JAM 的原始指令集和编译的细节过程在参考文献[9]中有详述。

JAM 的设计完成于 1989 年。第一个版本的实现是用 Prolog 写的用来模拟 JAM 机的一个指令集模拟器。第一个版本是非常低效的，每秒钟大约只能做 4 次归约（reduction）运算，但是这对评估和测试这个虚拟机是足够的，并且还使得 Erlang 可以用自己编写 Erlang 的编译器。

完成了 JAM 的设计后，我马上开始了虚拟机的 C 语言版本的开发，可是当 Mike Williams 读了我的 C 源代码后，我很快就放弃了。从那以后我就一直致力于编译器方面，而由 Mike Williams 负责写虚拟机，由 Robert Virding 负责做 Erlang 库。

- 1990 年——1990 年后 JAM 机的运行趋于稳定，并且早已超过最初设定的比 Prolog 解释器效率提高 70 倍的目标。Erlang 从此也有了自己的语法（而此前它一直被看作是 Prolog 的一种方言），享有了一种语言的所有权利，不再是 Prolog 的一种方言。

- 1991 年——Claes Wikström 也加盟了 Erlang 团队。JAM 机已经相当稳定，并彻底取代了 Erlang 的 Prolog 实现。

- 1992 年——Ericsson Business Systems（EBC）决定基于 ACS/Dunder 开发一款产品。这款产品叫做移动服务器（Mobility Server）——Ericsson 曾经专门将它作为 Erlang 开发产品在日本横滨的“第 14 届国际交换机研讨会”上展示过。

- 1993 年——Ericsson 成立了一个叫做 Erlang Systems AB 的全资子公司。该公司的目的就是拓展 Erlang 的外部用户，并且向内部和外部用户提供培训和咨询服务。Erlang 本身的支持则由 Ericsson 计算机科学实验室（Ericsson Computer Science Laboratory）来做。这一年发布了 Erlang 的第一个商用版本。

- 1995 年——Ericsson 的 AXE-N 项目宣告失败。AXE-N 项目是要开发一款

“下一代交换机”以取代 Ericsson 的 AXE-10。这个庞大项目的开发从 1987 年持续到了 1995 年。AXE-N 项目失败后，Ericsson 决定用 Erlang “重做”这个项目。该项目也终于促成了 AXD301 交换机的开发成功。

该项目比以前用 Erlang 开发的所有项目都要大得多。因此一个为了支持 AXD 项目的新项目组也成立起来。Erlang 库被重新命名为 OTP（The Open Telecom Platform，开放电信平台），一个相应的项目组也应运而生。

- 1996 年——为了给 Erlang 用户提供一个稳定的软件库，一个叫 OTP 的项目启动了。OTP 最初被用于最新启动的 AXD 项目；所有已经存在的项目也被移植到 OTP 上。OTP 汇聚了此前 Erlang 项目经验中——特别是先前为 Mobility Server 开发的库中——的许多思想。

- 1997 年——OTP 项目组升级为 OTP 产品线，成为 Erlang 的正式负责单位。在这之前，一直由 CSLab 为 Erlang 正式负责。我也作为首席技术顾问（chief technical coordinator）从 CSLab 来到了 OTP 团队。1996 至 1997 年间，一个三人小组（我自己、 Magnus Fröberg 和 Martin Björklund）重新设计并实现了 OTP 的核心库。

- 1998 年——Ericsson 首次发布了 AXD301。AXD301 项目是我将在第 8 章作为案例研究的一个重点。目前（2003 年）AXD301 已经含有超过 170 万行 Erlang 代码，这很可能是历史上用函数式编程所编写过的最庞大的系统了。

1998 年 2 月 Ericsson 禁止在新开发项目中使用 Erlang——这条禁令的主要原因是 Ericsson 想成为一个软件技术的消费者而不是一个软件技术的生产者。

1998 年 12 月，Erlang和OTP库获得开源许可。从此以后可以从 http://www.erlang.org免费下载。

是年，我和 15 个原 Erlang 团队成员一起离开 Ericsson 成立了一家新公司——Bluetail AB。Bluetail 的宗旨就是要用 Erlang 技术编写使

互联网服务更可靠的软件产品。

- 1999 年——Bluetail 推出了两款用 Erlang 编写的产品：Mail Robustifier[11]和 Web Prioritizer。同期 Ericsson 也推出了许多 Erlang 产品（包括 AXD301 系统和 GPRS 系统）。
- 2000 年——Bluetail 被 Alteon Web Systems[3]收购，接着 Alteon 被 Nortel Networks 收购。
- 2000 年以后——Erlang/OTP 技术获得了广泛的认可。到今天，已经没有人知道到底有多少项目采用了 Erlang。Nortel 开发的 Erlang 产品，“每年销售额都有数亿克朗（kronor，瑞典货币单位）”[51]， AXD301 是 Ericsson 最成功的新产品之一，并且有十几家小公司正在用 Erlang 做产品开发。

1.2 论文概览

本论文由如下章节构成：

- 第 1 章大致介绍了论文关注的主要问题领域，给出了课题的背景和文中涉及的工作的进展纪年表以及本论文的详细章节编排。
- 第 2 章介绍了作为后续各章基础的一个架构模型。我分析了一个软件架构的内涵，并阐述了哪些部分必须包含在架构中。我详细说明了该架构模型所针对的问题领域，阐述了架构背后蕴藏的哲学观点，并介绍了“面向并发编程”（Concurrency Oriented Programming， COP）这一概念。

  该章接着详细地剖析了 COP 的思想，分析了一门编程语言和系统若要支持面向并发编程，应该具备哪些特性。

  该章中还回顾了此前的一些相关工作，并对以前的工作和论文中给出的材料之间的相似之处和不同之处进行了比较。
- 第 3 章描述了 Erlang 编程语言。其中，我对 Erlang 编程语言进行了近乎全面的介绍，并说明了促使我在 Erlang 设计过程中作出某些决策的缘由。

- 第 4 章我给出了一些 Erlang 编程方法的例子。展示了如何把一个设计“分解”到它的函数式和非函数式部件（component）中去。我还逐步引出了并发和容错的概念，展示了如何编写一个普通的 client-server 模型。该章还描述了一种技术，能够让“任何事物都是 Erlang 的一个进程”这一假说成为现实，并出示了如何编写错误处理的代码的例子。

- 第 5 章涉及到本论文的核心问题。本章着力于阐述了在存在软件错误的条件下，如何编写出有合理行为的软件系统。“容错”思想的重点在于“软件错误”的概念——我描述了“软件错误”的含义和我所指的“容错系统”的含义。该章还描述了一种基于“监督树”设想的策略，并说明了该策略对编写可容错的软件的意义。

- 第 6 章把前面几章中得出的编写可容错系统的一些一般原则和一些为了编写可容错系统而产生的特定编程模式联系起来，这些编程模式对于理解 OTP 系统是很关键的。该章还指明了如何用 Erlang 构建可容错软件。

  该章给出了一个完整的实例，其中用到了一个 client-server 模型、一个事件处理器（event-handler）和一个有限状态机，这三个部件被添加到监督树中，而监督树会监视当中的进程并在发生错误的时候重新启动它。这整个程序，被打包为一个 OTP 的“应用”。

- 第 7 章描述了 OTP 系统，OTP 即 Open Telecom Platform，它是随 Erlang 编程语言一起发布的一个用来编写可容错软件的应用操作系统（Application Operating System, AOS）。OTP 包含一个用来实现可容错系统的很大的库，以及为了便于用户理解该系统的文档资料和指南。

  该章中我概要描述了一下 OTP 的体系架构，并对系统的几个主要部件进行了细致的介绍。

- 第 8 章是我们的技术的一个“耐酸性”测试。我们的设想在实践中能行之有效吗？该章中我分析了几个使用 OTP 实现大型商用产品的成功实例。该章的目的也在于看看在存在软件错误的条件下，我们的程序是否真正达到了可靠运行的目标。

该章研究的一个工程实例就是 Ericsson 的 AXD301 项目，那是一款高性能、高可靠性的 ATM 交换机。该项目本身就非常引人注目，因为它是应用函数式编程所写过的最大的程序之一。

- 第 9 章主要关于 API 和协议。我解答了如何定义模块之间的接口和各交互部件之间的接口。
- 第 10 章我问了一些更广泛的问题。我们的设想是否确实有效？它们对我们的软件开发是促进的还是妨碍的？哪些方面还可以改进？我们未来的目标是什么且我们如何达到我们的目标？

2 架构模型

对于软件架构这个术语来说，没有一个标准的、被普遍接受的定义，因为它还是一门年幼的学科，……虽然没有标准的定义，却也不乏定义……

卡内基·梅隆大学软件工程学院

本章提出了一个用于构建容错系统的软件架构。虽然每个人对于*架构*一词都有一个模糊的概念，但是这个词却几乎没有一个广为接受的定义，这就导致了很多误解。我认为如下定义对软件架构进行了比较全面的总结：

> *架构是一组有关软件系统组织方式的重要决策；是对系统构成元素、元素接口以及这些元素间协作行为方式的选择；是一种把这些结构和行为元素逐步组合为更大子系统的合成方式；也是一种构建风格，在其指导下把这些元素、元素接口、元素间的协作和合成组织起来。*
>
> Booch，Rumbaugh 和 Jaclbson[19]

2.1 架构的定义

从最高的抽象层次上看，架构就是“一种思考世界的方式”。然而，从实用性的层次上看，我们就必需得把我们看待世界的方式转化为一本实用的手册和一组规程，它们可以告诉我们如何使用我们看待世界的特定方式来构造一个特定的系统。

我们的软件架构通过如下一些方面来刻画：

1. 问题领域——我们的架构是为解决什么类型的问题而设计的？软件架构一定不是通用的，而是为解决某一类特定问题而设计的。缺少了关于用来解决哪类问题的描述的架构是不完整的。
2. 哲学—— 软件构造方法背后的原理是什么？架构的核心思想是什么？

3．软件构造指南——我们如何来规划一个系统？我们需要一个明确的软件构造指南集。我们的系统将由一个程序员团队来编写和维护——所以对所有的程序员和系统设计者来说，理解系统的架构和它的潜在哲学是很重要的。从实用性的角度来讲，这些知识以软件构造指南的方式表现出来更便于维持。一个完整的软件构造指南集包括编程规则集、例子程序和培训资料等等。

4．预先定义好的部件——以“从一组预先定义好的部件中选择”的方式进行设计远比“从头设计”的方式要来得容易。Erlang 的 OTP 库包含了一个完整的现成部件集（称之 behaviour 库），一些常用的系统都可以使用这些部件构建起来。例如 gen_server 这种 behaviour 就可以用来构建 client-server 系统，gen_event 这种 behaviour 可以用来构建基于事件（event-based）的程序。关于预定义部件的更完整的讨论见 6.1 节。6.2.2 节将给出一个关于如何使用 gen_server 这种 behaviour 来编写一个服务器软件的简单例子。

5．描述方式——我们如何描述某一部件的接口？我们如何描述系统中两个部件之间的通信协议？我们如何来描述系统中的静态和动态结构？为了回答这些问题，我们将介绍一些专门的符号。其中一些用来描述程序的 API，而其他的则用来描述协议和系统结构。

6．配置方式——我们如何来启动、停止和配置我们的系统？我们可以在系统工作过程中进行重配置吗？

2.2 问题领域

我们的系统最初是为开发电信交换系统而设计的。电信交换系统对可靠性和容错性有着苛刻的需求。电信系统需要“永久地”运行，必须有软实时的响应能力，当发生软件和硬件故障的时候要有合理的反应。Däcker[30]给出了电信系统需要具有的十条属性要求。

1．系统必须能够应对超大量的并发活动。

2．必须在规定的时刻或规定的时间内完成任务。

3．系统应该可以跨计算机分布运行。

4．系统要能够控制硬件。

5．软件系统往往很庞大。

6．系统要具有复杂的功能，例如：特性冲突。

7．系统应该能不间断运行许多年。

8．软件维护（例如重配置等）应该能在不停止系统的情况下进行。

9．满足苛刻的质量和可靠性需求。

10．必须提供容错功能，包括硬件失灵和软件错误。

我们可以对上述需求作出如下分析：

- 并发（concurrency）——交换系统天生就应该是并发的，因为对于交换机来说，经常同时有数以万计的用户在与交换机进行交互。这就意味着交换系统必须能够有效地处理成千上万的并发活动。
- 软实时（soft real-time）——在电信系统中，很多操作必须要在规定的时间内完成。其中有些操作是严格要求实时的，也就是说如果给定的操作在给定的时段里没有执行完，整个操作就被取消。而有些操作只是受到某种形式的定时器的监视，如果定时器超时而操作尚未完成，则重新执行一遍。

  编写这样的系统，就需要有效地管理起数以万计的定时器。
- 分布式（distributed）——交换系统也是天生分布式的，我们的系统应该以一种便于从单节点系统（single-node system）向多节点分布式系统(multi-node distributed system)转变的方式来创建。
- 硬件交互（hardware interaction）——交换系统有大量的外围硬件需要控制和监控。这就意味着要能够写出高效的设备驱动程序，并且不同的设备驱动之间进行上下文切换也要高效。
- 大型软件系统（large software systems）——交换系统都很庞大，例

如 Ericsson 的 AXE10 交换机和 AT&T 的 5ESS 交换机，源代码都是几百万行的程序[71]。这就意味着交换软件系统必须在源代码达到数百万行的时候也能工作。

- 复杂的功能（complex functionality）——交换系统都有着复杂的功能。市场的压力迫使系统的开发和部署要具有许多复杂的特性。通常，在这些特性之间的相互影响还没有被很好的理解的情况下，就必须得部署系统。在系统的运行期间，这些特性集很可能需要以多种方式进行修改和扩展。功能和软件的升级必须“就地进行”，也就是说，不能够让系统停下来。

- 持续运行（continuous operation）——电信系统要设计成可以持续运行许多年。这就意味着在系统不停下来的情况下进行软件和硬件的维护。

- 高质量要求（quality requirements）——即使在发生错误时，交换系统也应该提供可接受的服务。特别是电话交换设备，可靠性要求极高[1]。

- 容错性（fault tolerance）——交换系统应该是“容错”的。即从开始我们就知道会发生故障，但是我们必须设计出一些可以处理这些错误的软件和硬件基础设施，并在发生故障的时候仍然能够提供可接受的服务。

虽然这些需求最初是来自电信界，但决不仅仅适用于该特定问题领域。许多现代互联网服务（例如 web 服务器）就有着非常相似的需求列表。

2.3 哲学

我们怎么才能够构建出在软件存在错误的时候具有合理行为的可容错的软件系统呢？这是本论文余下部分要回答的问题。我先给出一个简洁的答案，在本文的剩余部分会对其进行细化。

为了构建出在软件存在错误的时候仍具有合理行为的可容错软件系统，我们

[1]通常要求在 40 年里停机时间不超过 2 小时[48]。

做了如下这些事情：

- 我们将软件组织成一个系统要完成的任务的层次结构，每一个任务对应于一组目标，具有给定任务的软件必须尝试去完成和该任务相关的目标。

  所有任务按照复杂性排序。最顶层的任务最复杂。如果最顶层任务完的目标都被完成，那么整个系统就运转正常。较低层次的任务应当能够保持系统以一种可接受的方式运转，即使系统所提供的服务有所折扣。

  系统中低层任务较高层任务更容易完成其目标。

- 我们将尽力完成顶层的任务。
- 当在完成某一目标的过程中检测到了一个错误，我们将尝试纠正这个错误。当我们不能够纠正该错误的时候，我们将立即取消当前的任务而启动一个更简单一些的任务。

编写这样一个任务层次需要一套强有力的封装方法。我们需要强有力的封装方法来隔离错误。我们不想再去编写那种系统中的一个部分发生的错误会对其他部分产生不利影响的系统。

我们需要以一种能够检测到在试图完成目标时所发生的所有错误的方式，来隔离为了完成某一目标而编写的所有代码。并且，当我们在试图同时完成多个目标时，我们不希望系统中某个部分所发发生的错误，会传播到系统的另外一个部分中。

因此，在构建可容错软件系统的过程中要解决的本质问题就是故障隔离。不同的程序员会编写不同的模块，有的模块正确，有的存在错误。我们不希望有错误的模块对没有错误的模块产生任何不利的影响。

为了提供这种故障隔离机制，我们采用了传统操作系统中进程的概念。进程提供了保护区域，一个进程出错，不会影响到其他进程的运行。不同程序员编写的不同应用程序分别跑在不同的进程中；一个应用程序的错误不会对系统中运行

的其他应用程序产生副作用。

这种选择当然满足了初步的要求。然而因为所有进程使用同一片 CPU、同一块物理内存，所以当不同进程争抢 CPU 资源或者使用大量内存的时候，还是可能对系统中的其他进程产生负面影响。进程间的相互冲突程度取决于操作系统的设计特性。

在我们的系统中，进程和并发编程是语言的一部分，而不是由宿主操作系统提供的。这样做比直接采用操作系统进程拥有很多优势：

- 并发程序可以一致地运行在不同的操作系统上——不同的特定操作系中是如何实现进程的不会对我们造成限制。我们的程序运行在不同的操作系统和处理器上唯一可见的差异就是 CPU 的处理速度和内存的大小。所有的同步问题和进程间通信都应当跟宿主的操作系统的特性没有一点关系。

- 我们这种基于语言的进程比传统的操作系统进程要轻量得多。在我们的语言里，创建一个进程是非常高效的，要比大多数操作系统中进程的创建快几个数量级[12,14]，甚至比大多数语言中线程的创建都快几个数量级。

- 我们的系统对操作系统的要求非常少。我们只用了操作系统很小的一部分服务，所以把我们的系统移植到譬如嵌入式系统等特定环境下是相当简单的。

我们的应用程序是通过大量互相通信的并行进程构建起来的。我们采用这种方式是因为：

- 它提供了一个架构基础设施——我们可以用一组相互通信的进程组织起我们的系统。通过枚举出系统中的所有进程，并定义出进程间消息传递的通道，我们就可以很方便地把系统划分成定义良好的子部件，并可以对这些子部件进行单独实现和测试。这种方法学也是 SDL[45]系统设计方法学的最高境界。

- 巨大的潜在效率——设计成以许多独立的并行进程来实现的系统，可以

很方便地实现在多处理器上，或者运行在分布式的处理器网络上。注意，这种效率的提升只是潜在的，只有当应用程序可以被分解成许多真正独立的任务时，才能产生实效。如果任务之间有很强的数据依赖，这种提升往往是不可能的。

- 故障隔离——没有共享数据的并发进程提供了一种强大的故障隔离方法。一个并发进程的软件错误不会影响到系统中其他进程的运行。

在并发的这三种用法中，前两条并不是其本质特性，可以由某种内置的调度程序通过在进程间提供不同的伪并行（pseudo-parallel）时分方式来获得。

第三个特性对于编写可容错系统的软件来说，则是本质性的。每一项独立的活动都在一个完全独立的进程中来执行。这些进程没有共享数据，进程之间只通过消息传递的方式进行通信，这就限制了软件错误造成的影响。

一旦进程之间共享有任何公共资源，譬如内存，或指向内存的指针，或互斥体等等，一个进程中的一个软件错误破坏共享资源的可能性就存在。因为消除大型软件系统中的这类软件错误仍然是一个未解的难题，所以我认为构建大型的可靠系统的唯一现实的方法就是把系统分解成许多独立的并行进程，并为监控和重启这些进程提供一些机制。

2.4 面向并发编程

在我们的系统中，并发扮演着核心角色，它是如此核心以至于我塑造了*面向并发编程*（Concurrency Oriented Programming）这个术语，以把这种编程风格和其他编程风格区分开来[2]。

在面向并发编程中，程序的并发结构应该遵循应用本身的并发结构。这种编程风格特别适用于编写那些对现实世界建模或与现实世界进行交互的应用程序。

面向并发编程同样也具有面向对象编程的两个主要优点。即*多态*（polymorphism）以及使用预先定义的协议使得不同进程类型的实例之间可以具有相同的消息传递接口。

[2] 譬如面向对象编程风格用对象来对现实世界建模，函数式编程（Functional Programming）风格用函数（function），逻辑化编程（Logic Programming）则用关系（relation）。

当我们把一个问题分解成许多并发进程的时候，我们可以让所有的进程响应同一种消息（即多态），并且可以让所有的进程都遵循相同的消息传递接口。

并发一词是指同时发生的活动集合。现实世界就是并发的，是由无数同时发生的活动组成的。在微观上看，我们自己的身体就是由同时运动着的原子、分子组成的。从宏观上看，整个宇宙也是由同时运动着的星系组成的。

我们做一件简单的事情的时候，譬如在高速公路上开车时，我们能觉察到身边行驶着飞速的车流，但是我们一样能够完成开车这一复杂的任务，并且可以不假思索就避开潜在的危险。

在现实世界中，顺序化的（sequential）活动非常罕见。当我们走在大街上的时候，如果只看到一件事情发生的话我们一定会感到不可思议，我们期望碰到许多同时进行的活动。

如果我们不能对同时发生的众多事件所造成的结果进行分析和预测的话，那么我们将会面临巨大的危险，像开车这类的任务我们就不可能完成了。事实上我们是可以做那些需要处理大量并发信息的事情的，这也表明我们本来就是具有很多感知机制的，正是这些机制让我们能够本能地理解并发，而无需有意识地思考。

然而对于计算机编程来说，情况却突然变得相反。把活动安排成一个顺序发生的事件链被视为是一种规范，并认为在某种意义上讲这样更简单，而把程序安排成一组并发活动则是要尽可能避免的，并常常认为会困难一些。

我相信这是由于几乎所有传统的编程语言对真正的并发缺乏有力支持造成的。绝大多数的编程语言本质上都是顺序化的；在这些编程语言中所有的并发性都仅仅由底层操作系统来提供，而不是由编程语言来提供。

在本论文中，我展现了这样的一个世界，其中并发是由编程语言来提供的，而不是由底层操作系统来提供。我把对并发提供良好支持的语言称为*面向并发的语言*（Concurency Oriented Language），简称 COPL。

2.4.1 基于现实世界编程

我们常常想编写一些对现实世界进行建模或者和其交互的程序。用 COPL

编写这样的程序相当容易。首先，我们来进行一个分析，它有三个步骤：

1. 从真实世界中的活动中识别出真正的并发活动；

2. 识别出并发活动之间的所有消息通道；

3. 写下能够在不同的消息通道中流通的所有消息；

然后我们来编写程序。程序的结构要严格保持与问题的结构一致，即每一个真实世界里的活动都严格映射到我们编程语言中的一个并发进程上。如果从问题到程序的映射比例为 1:1，我们就说程序与问题是同构（isomorphic）的。

映射比例为 1:1 这一点非常重要。因为这样可以使得问题和解之间的概念隔阂最小化。如果比例不为 1:1，程序就会很快退化而变得难以理解。在使用非面向并发的编程语言来解决并发问题时，这种退化是非常常见的。在非面向并发的编程语言中，为了解决一个问题，通常要由同一个语言级的线程或进程来强制控制多个独立的活动，这就必然导致清晰性的损失，并且会使程序滋生复杂的、难以复现的错误。

在分析问题时，我们还必须为我们的模型选择一个合适的粒度。比如，我们在编写一个即时通信系统（instant messaging system）时，我们使用每个用户一个进程的方式，而不是将用户身上的每一个原子对应到一个进程。

2.4.2 COPL 的特征

COPL 可以由如下 6 个特性来刻画：

1. COPL 应当支持进程。每一个进程应该可以看作是一个自包含的虚拟机器（self-contained virtual machine）。

2. 运行在同一机器上的各个进程应该被高度隔离。一个进程中的故障不能对其他进程产生副作用，除非这种交互在程序中被明确化。

3. 每个进程必须用一个唯一的、不可仿造的标识符来标识。我们称之为进程的 Pid。

4. 进程之间没有共享状态。进程只通过消息传递来进行交互。只要知道进

程的 Pid，就可以向它发消息。

5. 消息传递被认为是不可靠的，无传输保障的。

6. 一个进程应当可以检测另一个进程中的故障，并可以知道发生故障的原因。

值得注意的是，COPL 提供的并发性一定是真正的并发性，因此以进程的形式存在的对象都是真正并发的，进程间的消息传递也是真正的异步消息，而不像许多面向对象语言中一样是通过远程过程调用（remote procedure call）来冒充。

还应当注意，故障的原因并不总是正确的。例如，在一个分布式系统中，我们可能收到进程已经死亡的通知消息，然而事实上是发生了一个网络错误。

2.4.3 进程隔离

对理解 COP 和创建可容错软件来说，一个核心的概念就是*进程隔离*（isolation），同一台计算机上运行的两个进程，应当如同分别独立运行在物理上分离的两台计算机上一样。

理想的架构当然是面向并发的程序的每一个进程都给分配一个专用的处理器。但是在理想成为现实之前，我们不得不面对的事实是多个进程要运行在同一台计算机上。然而我们仍然应当认为所有的进程都运行在物理上独立的计算机上。

进程隔离有着许多好处：

1. 进程具有“不共享任何资源”的语意。这一点很明显，因为进程被认为是运行在物理上独立的计算机上的。

2. 消息传递是进程之间传递数据的唯一方式。因为进程之间没有任何共享资源，进程间交互数据只能采用这种方式。

3. 进程隔离意味着消息传递必须是异步的。如果进程通信采用同步方式，那么当消息的接收者偶然发生一个软件错误时，就会永久阻塞住消息的发送者，破坏了隔离的特性。

4. 没有共享资源，所以进行分布式计算所需的任何数据都必须通过拷贝。因为没有共享资源，进程间的交互只能通过消息传递，所以我们也不会知道消息什么时候到达接收者（记住我们说过消息传递是天生不可靠的）。知道消息是否被正确送达的唯一方法就是发送一个确认消息回来。

乍一看，要编写一个满足上述规定的多进程系统是很困难的——毕竟在针对大多数顺序化编程语言所做的并发扩展中，几乎提供了完全相反的功能，诸如锁、信号量、共享数据保护以及可靠消息传递。幸运的是，我们这种相反的做法被证明是正确的——编写一个这样的系统简单得出奇，并且所编写的程序不费吹灰之力就可以变得可伸缩，变得可容错。

因为所有的进程都要求完全独立，所以增加新的进程不会对原系统产生影响。因为整个软件就是一组独立的进程的集合，因此无需对应用软件作大的更改就容纳更多的处理器。

因为没有对消息传递的可靠性加以任何假设，所以我们写的应用程序在消息传递并不可靠的时候必须一样可以工作，在消息传递发生错误的时候也一样要能够工作。我们这样做了以后，当我们需要向上伸缩我们的系统的时候，就会得到回报。

2.4.4 进程的名字

我们要求所有进程的名字都是不可仿造的。这就意味着不可能猜测一个进程的名字，从而与之交互。我们假设所有的进程都知道它们自己的名字，以及由它们所创建的其他进程的名字。也就是说，父进程知道其子进程的名字。

要想使用 COPL 进行编程，我们就需要一种机制来找到相关进程的名字。记住，一旦我们知道了一个进程的名字，我们就可以给它发消息。

系统的安全性与进程名的获取方法是密切相连的。如果别人不知道进程的名字，就没有任何方法可以与之交互，这个系统也就是安全的了。一旦进程的名字广为外界所知，这个系统的安全性就削弱了。我们把以受控的方式向其他进程透露名字的过程称为*名字散布问题*（name distribution problem）——系统安全性的关键就在于名字散布问题。当我们把一个 Pid 透露给另外一个进程，我们就说我

们公布了该进程的名字。如果一个进程的名字从未被公布过，就不会存在安全性问题了。

因此，获取进程的名字是安全性的关键因素。因为进程名是不可仿造的，所以只要我们能够将关于进程名字的知识限制在可信进程的范围内，我们的系统就肯定是安全的。

在许多古老的宗教信仰中，人们都相信人类可以通过灵魂的真名来支配灵魂，以获得超越灵魂的力量。一旦获知了灵魂的真名，就可以获得超越它的力量，并且可以用这个真名来驱使灵魂去做很多事。COPL 采用的是相同的思想。

2.4.5 消息传递

消息传递须遵循如下规则：

1. 消息传递当是原子化的（atomic），意思是一个消息要么整个儿被传递，要么根本就不传递。
2. 一对进程之间的消息传递是有序的，意思是当在任何一对进程之间进行消息序列的收发时，消息被接收的顺序与对方发送的顺序相同。
3. 消息不能包含指向进程中的数据结构的指针——它们只能够包含常量和（或）Pid。

注意，第 2 点只是一个设计决策，并没有对用来传送消息的网络的基础语意作任何反映。下层的传输网络可能将消息重新排序，但是对于任一对进程来讲，这些消息在被交付前会被进行缓存和重组，以使它们形成正确的顺序。比起硬要允许消息按任意顺序传递来，这种假设可以使得编写消息传递的应用程序要容易得多。

我们说这种消息传递具有*发送并祈祷*（send and pray）之义。我们发送一条消息以后，就祈祷它能够到达对方。一旦收到对方发送回来的确认消息（有时候也叫做往返确认），就可以确认消息已经送达对方。有趣的是，很多程序员只相信往返确认，哪怕传输层提供的是可靠数据传输，哪怕这种确认是完全没有意义的。

消息传递还可以用于*同步*（synchronisation）。假设我们希望同步两个进程 A 和 B。如果 A 向 B 发送了一条消息，那么 B 只能在 A 发送了这个消息之后的某个时间点才收到该消息。这一点就是分布式系统理论里的因果次序（casual ordering）。在 COPL 中，所有的进程间同步都是基于这一简单的思想。

2.4.6 协议

部件之间隔离，采用消息传递的交互方式，这在架构上对于保护系统免受错误影响来说是足够了。但是对于说明系统的行为来说，是不够的，对于在发生了某种错误时判断到底是哪个部件出了错也是不够的。

到目前为止，我们都只是假设了单个部件出错，单个部件要么就正常运行，要么死了就死了。然而，实际会发生的情况是可能没有观察到有部件死掉，而系统却已没有如期地工作。

为了完善我们的模型，我们添加了一些新的东西。我们不仅需要部件的完全独立性，部件之间只通过消息传递进程交互，我们也需要制定部件之间相互通信所采用的协议。

通过制定出通信协议，如果遵循该协议进行通信的两个部件中一旦有谁违犯了协议，我们就可以很容易地识别出来。我们可以通过对程序的静态分析——如果可能的话，还可以把运行时检查编译到生成码中，以便当静态分析失效时也报告错误——来保证协议被贯彻了。

2.4.7 COP 与程序员团队

构建大的软件系统需要许多程序员的共同努力，有时候甚至达到好几百人。为了把这么多人的工作都协调起来，通常是把程序员组织成小一些的开发小组或团队，每个小组负责系统中的一个或多个逻辑部件。日复一日，各个小组之间通过消息传递（如 email 或电话）来进行交流，而不必频繁地见面。在某些情况下，开发小组分布在不同的国家，从来都不见面。有趣的是，你会发现不仅仅是软件系统因各种原因需要被组织成独立的部件，各部件以纯消息传递的方式进行通信，而且这也是大型软件开发群体的组织方式。

2.5 系统需求

为了支持面向并发的编程风格，为了构建满足电信系统需求的软件，我们对系统的根本特性提出了一组需求。这些需求对于系统来说，是一个整体——我并不关心这些需求是由编程语言来满足，还是由语言所附带的库或创建方法来满足。

我们对下层的操作系统和编程语言有 6 条根本需求。

R1. 并发性——我们的系统必须支持并发性。创建或销毁一个并发进程的计算开销一定要非常小，即使创建大量的并发进程，也不应当带来灾难。

R2. 错误封装——一个进程中发生的错误一定不能破坏系统中其他的进程。

R3. 故障检测——一定要可以检测到本地异常（本地进程中发生的异常）和远程异常（非本地进程中发生的异常）。

R4. 故障识别——我们要能够识别出异常产生的原因。

R5. 代码升级——要有某种机制来替换执行中的代码，而不必停下系统。

R6. 持久存储——我们需要把数据按某种策略存储下来，以便恢复一个已经崩溃的系统。

还有一点非常重要，即为了满足上述需求所采用的实现方式一定要高效——如果不能够可靠地创建几十万个进程，那么并发性就没什么大用；如果故障报告中没有包含足够的信息使得随后可以纠正故障，那么故障识别也就没有什么大用。

上述需求的实现方式可以是多种多样。譬如并发性，既能够由语言原语来提供（例如 Erlang 语言），也能够由操作系统来提供（例如 Unix）。像 C 和 Java 之类的语言本身并不是面向并发的，但是可以利用操作系统的那些让人觉得可以达到并发性的原语来获得并发性。确实，并发程序可以由本身并不具备并发性的语言来编写。

2.6 语言需求

用来编写并行系统的编程语言必须包括：

- 封装原语——语言必须有多种手段来限制错误的蔓延。应当可以把一个进程隔离起来，免得它会破坏其他进程。
- 并发性——语言必须提供一种轻量化的机制来创建并行进程，以及在进程间发送消息。进程的上下文切换、消息传递必须非常高效。并行进程还必须以一种合理的方式来分享 CPU 时间，以便当前使用 CPU 的进程不至于垄断 CPU，而其他的进程处于“准备好”状态而得不到处理。
- 错误检测原语——语言应当允许一个进程监控另一个进程，从而检测被监控进程是否因任何原因而终止。
- 位置透明——如果我们知道了一个进程的 Pid，我们就应该可以向它发送消息，无论它是本地还是远程的。
- 动态代码升级——应该可以动态替换运行时系统中的代码。注意，因为许多进程可能同时按照同一份代码在运行，所以我们需要一种机制，来允许现有的进程按照“老”的代码运行，而同时“新”进程按照修改后的代码运行。

上述对于编程语言的需求不仅要被满足，而且要以一种合理有效的方式被满足。当我们编程的时候，不希望我们的表达自由受到诸如进程数目之类的限制，我们也不希望担心当一个进程试图垄断 CPU 时会发生什么事情。

系统中进程个数的上限应该足够大，以便我们编程时不用把进程的个数作为一个限制因素来考虑。例如，为了构建一个处理 1 万个并行用户会话的交换系统，我们可能需要创建多达 10 万个进程[3]。

上述 6 条特性对于简化应用程序的编写是必要的。如果我们能够将问题的并发结构以 1:1 的方式映射到解决该问题的应用程序的进程结构上的话，我们把语义上一组分布式的交互部件映射到 Erlang 程序的过程就会极大地简化。

[3] 假设每一个会话需要 10 个进程。

2.7 库需求

语言并不是无所不能的——许多东西是由附带的系统库提供的。基本程序库必须提供：

- 持久存储——由它存储用于故障恢复的信息。
- 设备驱动程序——这些程序提供了一种与外界交互的机制。
- 代码升级——它允许我们升级运行系统中的代码。
- 运行基础——它解决系统的启动、停止和错误报告问题。

观察一下我们的程序库，不难看出它们虽然是用 Erlang 编写的，但是它们提供的服务都是本来可以由操作系统很方便地提供的服务。

因为 Erlang 的进程是彼此隔离的，只以消息传递的方式彼此通信，所以它们的行为就非常像操作系统的进程，后者是通过管道（pipe）和套接字（socket）进行通信。

本来可以很方便就由操作系统提供的许多特性被移到了编程语言中，于是操作系统就只需要提供设备驱动的一组原语就够了。

2.8 应用程序库

持久化存储等特性并不是作为 Erlng 的语言原语来提供的，而是由基本 Erlang 库来提供。这个基本库是构建一个复杂的应用软件的前提条件。更复杂的应用需要比持久化存储等层次更高的抽象。为了构建这样的应用程序，我们需要一些现成的软件实体来辅助我们编写诸如客户－服务器式（client-server）的程序。

OTP 库就给我们提供了用来构建可容错系统的一个完整的设计模式（我们称之为 behaviour）库。本论文中我会介绍 behaviour 库的一个最小集，可以用它们来构建可容错的应用软件，它们是：

- supervisor——一个监督模型 behaviour。
- gen_server——一种用于实现客户－服务器式应用程序的 behaviour。

- gen_event——一种用于实现事件处理式应用程序的 behaviour。
- gen_fsm——一种用于实现有限状态机的 behaviour。

这些库程序当中，用于编写可容错应用软件的核心部件就是那个监督模型。

2.9 软件构造指导方针（guidelines）

作为对编写可容错软件的一般哲学的阐释的一种补充，我们还提供了更多的指导方针，这些指导方针在编程语言中已经被采用了，我们也希望在编写应用程序时采用它们。我们还提供了例子程序，还有一些如何使用程序库的例子。

在开源的 Erlang 版本中，这些指导方针已经作为构建大型 Erlang 系统的基础被包含其中。附录 B 重新诠释了这些编程方针，也可以在开源版的 Erlang 中找到它们。本论文包含了一些额外的指导方针，按如下结构组织：

- 本章介绍了我们的模型的一个全面的哲学思想。
- 软件错误（error）的概念在多处都有讨论。第 5.3 节和第 4.3 节描述了软件错误的含义；第 4.4 节对于在编写错误处理程序的时候该采用哪种软件错误类型给出了一些建议。
- 关于如何编写简单的部件的例子在第 4 章可以找到，关于如何使用 OTP 的 behaviour 的例子在第 6 章可以找到。

2.10 相关的工作

各个软件部件不能很好地彼此隔离，是许多流行的编程语言不能够用来构建健壮的软件的主要原因。

> *安全性的本质，在于要能够将互不信任的程序隔离起来，在于要保护基本平台不受这些程序的破坏。隔离在面向对象系统中是相当困难的，因为对象很容易被别名化（aliased）*[4]*。*——Bryce[21]

Bryce 继续说，对象的别名化是很难缠的，而且在实际编程中不可能被检测

[4] 一个别名化对象即至少由两个其他的对象拥有该对象的引用的对象。

到，建议使用*保护域*（protection domains）（类似于操作系统的进程）来解决这一问题。

Sun Microsystems 的 Java Czajkowski 和 Daynès 在一篇文章中曾写道：

> *在同一台计算机上执行多个用 Java 编写的应用程序的唯一安全的方式，是给每一个应用程序开一个 JVM，并且每一个 JVM 运行在一个单独的 OS 进程中。这样又会造成资源利用的效率方面的大大下降，会引起性能、伸缩性、程序启动时间等方面的恶化。这样一来，Java 语言所提供的好处就只剩下可移植性和提升程序员的生产力了。这些固然重要，但是语言提供的所有潜在安全性并没有完全被实现。事实是，在“语言安全性”与“真实安全性”之间存在着离奇的差异。*

在这篇文章中，他们介绍了一个 MVM（JVM 的一种扩展），其目标是：

> *……把 JVM 变成一个类似于 OS 的执行环境。尤其是现代 OS 所提供的进程抽象，也就是基于特性的角色模型；计算之间的相互隔离；资源的审计和控制以及资源的终止和回收。*

为了达到这一点，他们认为：

> *……任务不得直接共享对象，任务之间通信的唯一方式是使用标准的、拷贝式的通信机制，……*

这些结论并不新奇。早在 20 多年前 Jim Gray 就得出了非常类似的结论，他曾经在《*Why do computers stop and what can be done about it*》这篇非常通俗的文章中描述过。他说：

> *与硬件系统一样，软件的容错性关键在于把大的系统逐级分解成模块，每一个模块既是提供服务的最小单位，也是发生故障的最小单位，一个模块的故障不会传播到模块之外。*
>
> *……*
>
> *进程要想达到容错性，就不能与其他进程有共享状态；它与其他进程的唯一联系就是由内核消息系统传递的消息。*——[38]

支持这种编程风格（并行的进程，没有共享数据，纯消息传递）就是Andrews和Schneider[4]所提到的“面向消息的语言”。一种“美其名曰”PLITS[5]（1978）[35]的语言很可能就是这种编程语言的第一个实例了：

> *在 RIG6 的实现过程中的基本设计决策就是采用了一种没有共享数据结构的严格的消息规范。用户与服务器之间所有的通信消息都是通过 Aleph 内核来路由的。这种消息规范被证明是非常灵活、可靠的。*——[35]

暂时撇开语言不管，让我们想想一个单独的进程应该具备哪些性质呢？

Schneider[60,59]回答了这个问题，他给出了他认为一个硬件系统如果要适合在其上缔造可容错系统应该具备 3 条性质。Schneider 把这些性质称作：

1. *故障即停*（Halt on failure）——当一个处理器出错时，应当立即停止下来，而不是继续执行可能不正确的操作。
2. *故障曝光性质*（Failure status property）——当一个处理器发生故障时，系统中的其他处理器应该得到通知，故障的原因必须交代清楚。
3. *持久存储性质*（Stable storage property）——处理器的存储器应当分为持久存储器（stable storage，处理器崩掉时依然存在）和临时存储器（volatile storage，处理器崩掉就没了）。

Schneider把具备这些性质的处理器称为*错即停处理器*（fail-stop processor）。其思想就是一旦错误[7]发生，就没有必要继续运行了。出错的处理应该停下来，以免继续执行会引起更大的破坏。在一个错即停处理器中，状态存储在临时或持久处理器里。当处理器崩溃时，临时存储器中的所有数据将丢失掉，而持久存储器中的所有数据在崩溃后仍然可以使用。

在参考文献[38]中，当 Gray 谈到“速错”（fail-fast）进程时也谈到了一个非常相似的想法。

> *用进程的方法达到故障隔离的思想提倡每个进程都是速错的，要么它就正确地运行着，要么它就应该检测到错误，报告错误并停止运行。*

[5] Programming Language In The Sky.
[6] RIG是用PLITS编写的一个小的系统。
[7] 这里Schneider所说的错误是指哪些不能校正的错误。

进程以防护性编程（defensive programming）的方式达到“速错”。它们对其所有的输入参数、中间结果和数据结构进行例行检查。一旦检测到错误，就立即报告该错误并停止运行。用Cristian 的话来讲，即速错软件具有很短的检测潜伏期（detection latency）。——[38]

Schneider 和 Gray 两人的思想本质是一样的；只不过一个说的是硬件，一个说的是软件，但是其根本原则如出一辙。

Renzel 也认为当进程发生不可校正的错误时应该尽快停下来这一点非常重要：

一个软件系统中的一个错误可能会引起一个或更多其他错误。从故障发生到其被检测到的间隔时间——即潜伏时间——越长，代价就会越大，因为这样会增加对故障进行回退分析的复杂性……

为了有效地处理错误，我们应该尽早地检测到错误并停下来。——[58]

综合以上这些意见和我们的原始需求，我建议一个系统应该具备如下一些性质：

1. 以进程作为错误封装的单位——即一个进程中发生的错误不会影响到系统中其他的进程。我们称这一性质为*强隔离*（strong isolation）。
2. 进程要么就规规矩矩地运行，要么就痛痛快快地死掉。
3. 故障和故障原因应该可以被其他进程检测到。
4. 进程之间没有共享状态，唯以消息传递的方式通信。

要想一个编程语言或平台具有如上这些性质，并可以用来构建可容错的软件系统，还需要具备一些必要的前提条件。我们将看到这些性质是如何在 Erlang 及其编程库中被满足的。

本论文的许多思想并不新奇——关于构建可容错系统的基本原则在参考文献[38]Gray 的文章中都有描述。

Gray 的 Tandem 计算机的许多特性与 OTP 的设计原理和 COP 的基本原则都

极其相似，而前者早就被 Gray 详细地讨论过。

这里我们引用 Gray 的文章中的两段文字，首先是关于设计原则的，见参考文献[38]第 15 页：

达到软件可容错性的关键在于：

• 按照进程、消息来划分软件模块。

• 错误限制在速错（fail-fast）的软件模块内。

• 由成对进程（process-pairs）来容纳硬件故障和瞬时的软件故障。

• 由“事务机制”（transaction mechanism）来提供数据和消息的完整性。

• 成对进程与“事务机制”相结合来简化异常处理，并容纳软件故障。

按照进程和消息来划分软件模块。就像硬件一样，软件容错性的关键在于把大的软件系统层次化地划分成模块，每个模块作为服务的单位，也作为故障的单位。一个模块的故障不会传播到模块以外。

关于如何将软件模块化是有颇多争议的。从开始 Burroughs 的 Espol 语言到后来的 Mesa、Ada 语言，编译器编写者们总是把硬件系统想得很完美，并主张由他们通过静态的编译时类型检查来提供良好的隔离性。与编译器编写者们相反，操作系统设计者们则主张运行时检查，并主张将进程作为保护单位和故障单位。

尽管编译器检查和由编程语言提供的异常处理确实有用，但是从历史上看，人们似乎更偏向于用运行时检查加进程的方式来达到故障封闭的目标。因为这种方式具有简单性这一优势——一旦一个进程或它的处理器出错，只管停下它！这种方式中进程就充当了一种干净的模块单位、服务单位、容错单位、出错单位的角色。

故障被限制在速错的软件模块之内。

进程因为与其他进程没有任何共享状态所以具有容错性；进程与其他

进程联系的唯一方式就是通过内核消息系统发送的消息。——[38]

如果我们将这些观点与我们现在的Erlang系统比较我们会发现许多惊人的相似。当然也有些不同之处——在Erlang中并不建议使用“防护性编程”的风格，因为编译器增加了一些必要的检测，使得这种编程风格并无必要。Gray所指的“事务机制”由mnesia数据库来提供[8]。而Gray所指的错误限制和处理则由OTP库中的“监督树”behaviour来完成。

“‘速错’模块”的思想对应于我们的编程指导方针，在我们的编程指导方针里，我们说进程应该严格按照我们期望的方式运行，否则就应该停掉。我们的系统中的监督层次结构对应于 Gray 所指的模块的层次结构。这种思想在 Candea 和 Fox[22]的作品关于“脆崩软件”（crash-only software）的论述中也可找到——他们认为，应当允许软件部件崩掉然后重启它，这样会简化故障模型，并有利于保证代码的可靠性。

现代面向对象系统方面的工作也越来越认识到使软件部件彼此隔离的重要性。在参考文献[21]中，Bryce 和 Razafimahefa 认为使程序各部件之间、各程序（操作系统上运行的程序）之间相互隔离这一点是根本性的。他们认为这一点是任何一个对象系统都应该具备的根本特性。然而如他们的文章中所指出的，这一点在面向对象的背景下是很难真正达到的。

[8] mnesia是用Erlang编写的。

3 Erlang

本章就来介绍一下 Erlang 编程语言。这里对编程语言的介绍并不追求完整性，若要寻求一个较为完整的介绍资料，请参考参考文献[5]。随着 Erlang 的发展，参考文献[5]已被纳入到了 OTP 文档[34]中。一个更正式的关于 Erlang 的介绍可以在《Erlang Specification》 [17]和《Core Erlang》[23]中找到。

Erlang 属于*面向消息的语言*（message-oriented language）一类——面向消息的语言都是通过并行进程的方式提供并发性的。在面向消息的语言中，没有任何共享的对象，取而代之的是进程之间以收发消息来达到交互。

在本章中，我将介绍 Erlang 语言的一部分，这些部分对于理解本文中所有的例子来说是足够的。

3.1 概览

Erlang 的世界观可以归纳为如下的一些观念：

- 一切皆进程。
- 进程强隔离。
- 进程的生成和销毁都是轻量的操作。
- 消息传递是进程交互的唯一方式。
- 每个进程有其独有的名字。
- 你若知道进程的名字，你就可以给它发消息。
- 进程之间不共享资源。
- 错误处理非本地化。
- 进程要么好好跑，要么死翘翘。

把进程作为抽象的基本单位，缘于期望设计出一种适合编写大型的可容错的软件系统的语言。编写这类软件要解决的一个基本问题就是要限制错误的传播——进程的抽象正好提供了一种阻止错误传播的抽象边界。

例如，Java 就对于限制错误传播无能为力的，所以 Java 不适合用来编写“安全的”（原文如此）应用程序（见 2.10 节对 Java Czajkowski 和 Daynès 的文章的引用）。

如果进程真正是隔离的（必须做到对错误传播的限制），那么进程的其他性质——例如只能以消息传递的方式进行交互——就顺理成章地成为这种隔离性的结果。

关于错误处理的观点似乎并不明显。当我们构建一个可容错系统时，我们需要至少两台物理上独立的计算机。只用一台计算机是不成的，一旦它崩溃了，就什么东西都没有了。我们能够想象的最简单的可容错系统也由两台计算机组成，如果一台崩溃了，另外一台就可以接过第一台的所有工作。在这种最简单的情形下，也要求故障恢复软件做到非本地化；故障发生在第一台计算机上，而由运行在第二台计算机上的软件来纠正该错误。

Erlang 的世界观就是“万物皆进程”，当我们把真实的计算机也模拟成进程时，我们就得到了错误处理应该非本地化这一思想。其实，这是一个修正后的事实，远程错误处理只有在本地尝试修复错误失败的情况下才会发生。如果有异常发生，一个本地进程应该可以检测到它并纠正它所造成的故障，在这种情况下对于系统中所有其他进程来说，根本感觉不到异常的发生。

如果把 Erlang 看作是一种并发语言，它是非常简单的。因为没有共享数据结构，没有监视（monitor）或同步方法，所以需要学习的东西很少。语言的主体部分，或许也是最平淡无奇的部分，就是这种语言的顺序化（sequential）子集。这个顺序化子集可以用一种动态类型、严格的函数式编程来刻画，而函数式编程是完全没有副作用的。在这个顺序化子集里，有少数操作是有一些副作用的，但是事实上这些操作不是必需的。

本章的后续部分首先介绍了 Erlang 的这个顺序化子集，接着的章节介绍了

并行编程和分布式编程以及错误处理，最后介绍了用以指明 Erlang 数据和函数类型的一种类型符号。

作为过渡，我以 Erlang 顺序化编程的一段示例代码开始下文的描述。

3.2 例子

图 3.1 给出了一个简单的 Erlang 程序。该程序包含如下一些结构：

1. 这个程序以一个*模块*（module）的定义（第 1 行）开始，接着是*导出*（export）和*导入*（import）声明，接着是一些函数（function）。
2. 导出声明（第 2 行）是说函数 areas/1 从本模块导出，符号 areas/1 意思是一个叫 areas 的函数，有 1 个参数。只有包含在导出列表中的函数才可以在模块以外被调用。
3. 第 3 行的导入声明是说函数 map/2 可以在模块 lists 中被找到。

```
1  -module(math).
2  -export([areas/1]).
3  -import(lists, [map/2]).
4
5  areas(L) ->
6      lists:sum(
7                map(
8                    fun(I) -> area(I) end,
9                    L)).
10
11 area({square, X}) ->
12     X*X;
13 area({rectangle,X,Y}) ->
14     X*Y.
```

图 3.1：一个 Erlang 模块

5. 第 5 行到第 14 行是 2 个函数的定义。
6. 第 6 行是对 lists 模块中的 sum 函数的调用。

7. 第 7 行到第 9 行是调用lists模块的map/2 函数。注意这里对map和sum两个函数的调用的不同：两个函数都在同一个模块里，一个是用了全格修饰名（fully qualified name）（即lists:sum）调用而另一个用了简短调用序列（即以map(…)取代lists:map(…)）。这两种调用方式的不同有由第 3 行的导入声明来解决，该声明意味着map/2 函数将可以在模块lists中找到。

8. 第 8 行创建了一个*匿名函数*（fun），作为 map 的第 1 个参数。

9. 第 11 至 14 行都是函数 area/1。这个函数有两个*子句*（clause）。第 1 个子句是第 11 至 12 行，第 2 个子句是第 13 至 14 行，两个子句之间以分号隔开。

10. 每个子句有一个*头*（head）和一个*体*（body）。头和体以一个“—>”符号分隔。

11. 一个函数头的的每一个参数位置是一个*模式*（pattern），或者还有一个*保护式*（guard）（加第 3.3.4 节）。在第 13 行，就是一个{rectangle，X，Y}的模式。在这个模式中，花括符表示一个*元组*（tuple），元组的第 1 个参数是一个*原子*（atom）（即“rectangle”），第 2、第 3 个参数是*变量*（variable）。变量以大写字母打头，原子以小写字母打头。

为了让这个程序运行起来，我们得启动一个 Erlang 的 shell（译注：解释器的输入输出程序），编译一下这个程序并输入一些函数求值（译注：函数式编程里，将函数的调用也称为求取该函数的值，下同。）的命令，如图 3.2 所示。该图中所有用户输入都标以下划线。Erlang 的 shell 的提示符是字符“>”，意思是系统正在等待输入。

- 图 3.2 的第 1 行是启动一个 Erlang 的 shell。
- 第 5 行编译 math 模块。

```
1  $ erl
2  Erlang (BEAM) emulator version 5.1 [source]
3
4  Eshell V5.1  (abort with ^G)
5  1> c(math).
6  ok,math
7  2> math:areas([{rectangle,12,4},{square,6}]).
8  84
9  3> math:area({square,10}).
10 ** exited: {undef,[{math,area,[{square,10}]},
11                   {erl_eval,expr,3},
12                   {erl_eval,exprs,4},
13                   {shell,eval_loop,2}]} **
```

图 3.2：在 shell 里编译并运行一个程序

- 第 7 行是一个函数求值命令，解释器接受了命令，求取了函数的值，并把结果在第 8 行打印出来。
- 第 9 行试图求取一个没有从 math 模块导出的函数的值。产生了一个异常，并打印了出来（第 10 至 13 行）。

3.3 Erlang 顺序化编程

3.3.1 数据结构

Erlang有 8 种原始数据类型[1]：

- **整数**（integer）——整数被记作一串十进制数字，例如，12,12375 和 -23427 都是整数。整数的算术运算是准确的，没有精度限制[2]。
- **原子**(atom)——原子在程序中用来表示特异值(distinguished value)。原子记为一串连续的字母数字字符，打头的字符要是小写的。如果原子用一对单引号括起来的话，那么它就可以包含任意字符，包括转义字符。

[1] 也称为常量。
[2] 整数的精度只受可用内存的限制。

- **浮点数**（float）——浮点数被表示为满足 IEEE754[43]规则的 64 位浮点数。所有±10E308 范围内的实数都可以用 Erlang 浮点数表示。
- **引用**（reference）——引用是全局唯一的符号，只用来比较两个引用是否相等。引用可以通过调用 Erlang 原语 make_ref()来创造。
- **二进制数**（binary）——一个二进制数是一个字节序列。二进制数为二进制数据的存贮提供了一种高空间效率的方法。Erlang 提供了组合和分解二进制数的原语，也提供了二进制数的高效的输入/输出原语。对二进制数的完整介绍见参考文献[34]。
- **Pid**——Pid 是 Process Identifier(进程标识符)的缩写,Pid 由 Erlang 的 spawn（…）原语创建，Pid 是 Erlang 进程的引用。
- **端口**（port）——端口用于与外界通信，由内置函数（BIF[3]）open_port 来创建。消息可以通过端口进行收发，但是这些消息必须遵守所谓“端口协议”（port protocol）的规则。
- **匿名函数**（fun）——匿名函数是函数闭包[4]，由表达式“fun(…) -> … end.”来创建。

还有两种复合数据类型：

- **元组**(tuple)——元组是一种包含固定个数的 Erlang 数据的容器。{ D1, D2, …, Dn }表示一个元组，他的参数是 D1, D2, …, Dn。这些参数可以是原始数据类型，也可以是复合数据类型。对元组的元素的访问时间是恒定的。
- **列表**（list）——列表是包含可变个数的 Erlang 数据的容器。[Dh｜Dt]表示一个列表它的第 1 个元素是 Dh，余下的元素是一个列表 Dt。[]表示空列表。

[D1, D2, .., Dn]是[D1 | [D2 | .. | [Dn | [ ]]]]的简写形式。列表的第 1 个元素的访问时间是恒定的。列表的第 1 个元素称为列表的*头*（head），除第 1 个元

[3] BIF即Built In Function的缩写。
[4] 在其他语言里被称为lambda表达式。

素以外的剩余部分称为列表的*尾*（tail）。

Erlang 还提供了两种形式的“语法糖衣”（syntactic sugar）：

- **字符串**（string）——字符串记作用双引号引起来的字符系列。这种写法只不过是字符串里的字符的 ASCII 码组成的整数列表的“语法糖衣”。例如，字符串“cat”只是是列表[97, 99, 116]的速记法。
- **记录**（record）——记录提供了对每个元素都带有标记的元组的一种便利的访问方式。使得我们可以通过名字而不是通过位置来访问元组的元素。一个预编译器会获取到记录的定义，并用正确的元组引用来替换掉记录。

3.3.2 变量

Erlang 的变量是以大写字母打头的一个字符序列，第 1 个字符后面可以跟字母序列或任意字符或下划线“_”。

Erlang 里的变量要么为*未绑定*（unbound）的——即还没有给它绑定值，要么是*已绑定*(bound)的——即已经绑定了一个值。变量一旦绑定了一个值，就不能再改变。这种变量称为*单赋值变量*（single assignment variable）。因为变量的值不能被改变，所以程序员在进行一个“有破坏性”（destructve）的赋值时，就不得不创建一个新的变量。

譬如在 C 语言里的如下表达式：

```
x = 5;
x = x + 10;
```

在 Erlang 里要实现等价的功能就要写作：

```
X = 5.
X1 = X + 10.
```

在这里我们不能改变 X 的值，所以我们创造了一个新变量 X1。

3.3.3 项式（term）与模式（pattern）

一个*基项*（ground term）的递归定义为：基项是一个原始数据类型（primitive data type），或者由基项构成的元组，或者由基项构成的列表。

一个*模式*的递归定义为：模式是一个原始数据类型，或者一个变量，或者由模式构成的元组，或者由模式构成的列表。

一个原始*模式*（primitive pattern）是其所有变量都不同的模式。

模式匹配（pattern matching）就是一个模式与一个基项进行比较的行为。如果一个模式是一个原始模式而基项也与之同形，或者模式中出现的所有常量都在基项中的同样的位置出现，我们就说该模式与该基项匹配成功，否则就说它们匹配失败。一旦匹配成功，就会把模式中出现的所有变量都绑定上项式（term）中对应位置的对应数据元素。我们称这个过程为一致化（unification）。

更正式地，如果 P 代表一个原始模式而 T 代表一个项式，如果 P 和 T 满足如下条件我们就说 P 与 T 匹配：

- 如果 P 是由头 Ph 和尾 Pt 构成的列表，而 T 是由头 Th 和尾 Tt 构成的列表，那么必须 Ph 与 Th 匹配，且 Pt 与 Tt 匹配。
- 如果 P 是元组{P1, P2, …, Pn}，而 T 是元组{T1, T2, …, Tn}，那么必须 P1 与 T1 匹配，P2 与 T2 匹配，依此类推。
- 如果 P 是常量，那么 T 必须是相同的常量。
- 如果 P 是一个自由变量，那么 T 必须是一个已绑定的变量。

下面是一些例子：

模式{P, abcd}与项式{123, abcd}匹配，并在匹配时发生绑定：P|->123。

模式[H | T]与项式“cat”匹配，并发生绑定：H|->99, T|->[79, 116]。

模式{abc, 123}与项式{abc, 124}不匹配。

3.3.4 保护式（guard）

保护式是只使用谓词的表达式。保护式紧随原始模式之后，由关键字 when 引出。例如我们称如下程序片断：

```
{P,abc,123} when P == G
```

为一个保护模式（guard pattern）。

保护式记作一系列以逗号格开的保护测试（guard test），所有的保护测试都是如下形式：

T1 二元运算符 T2

这里 T1 和 T2 都是保护项（guard term）。

所有可用的二元运算符如下表所列：

运算符	含义
X > Y	X 大于 Y
X < Y	X 小于 Y
X =< Y	X 小于或等于 Y
X >= Y	X 大于或等于 Y
X == Y	X 等于 Y
X /= Y	X 不等于 Y
X =:= Y	X 严格等于 Y
X =/= Y	X 严格不等于 Y

当保护式被作为表达式使用时，通常取值为原子 true 或 false。true 代表*成功*（succeeded），false 代表*失败*（failed）。

3.3.5 扩展模式匹配

在原始模式里，所有的变量必须不同。扩展模式与原始模式有相同的语法，只不过它不要求所有的变量都不同。

在进行模式匹配的时候，我们首先把扩展模式转换成原始模式和保护式，然后当作原始模式进行匹配。

如果变量X在模式中出现了N次，把第 2 个及后面的X都替换为一个新鲜变量[5]，如F1、F2 等等。对每一个新鲜变量，都在保护式里加上一个谓词Fi == X。

使用这一规则，就使得

```
{X,a,X,[B|X]}
```

被转换成

```
{X,a,F1,[B|F2]} when F1==X, F2==X
```

最后要说，在模式中，变量“_”被用来表示“匿名变量”，匿名变量可与与任何项式匹配，但是不会发生变量绑定。

3.3.6 函数

函数遵循如下规则：

1. 一个函数有一个或多个由分号分隔的*子句*（clause）组成。
2. 一个子句含有一个*头*（head），紧接着是分隔符->,再接着是*体*（body）。
3. 函数头是由一个原子，以及后边紧跟着的一组用括号括起来的模式，以及后面可能紧跟着的一个保护式组成。如果有保护式，则保护式用关键字 when 引出。
4. 函数体由一序列用逗号分隔的*表达式*（expression）组成。

即如下形式：

[5] 新鲜变量是指在当前上下文中没有出现过的变量。

```
FunctionName(P11,...,P1N) when G11,...,G1N ->
    Body1;
FunctionName(P21,...,P2N) when G11,...,G1N ->
    Body2;
    ...
FunctionName(PK1, PK2, ..., PKN) ->
    BodyK.
```

这里 P11，…, PKN 即前一节所描述的扩展模式。

下面是两个例子：

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


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

函数按下述方法执行：

在求取函数 Fun(Arg1, Arg2, …, ArgN)的值时，我们首先寻找该函数的定义。第一个满足其子句头里的模式与参数 Arg1…ArgN 匹配的函数当选为被调用函数。如果模式匹配成功且所有的保护测试都成功的话，该子句体就被求值。子句头里的所有自由变量都通过模式匹配获得了调用时提供的实际参数。下面用一个求取表达式 member(dog, [cat, man, dog, ape])的值的例子来说明每一步的行为。

我们假设 member 函数的定义如下：

```
member(H, [H1|_]) when H == H1 -> true;
member(H, [_|T] -> member(H, T);
member(H, []) -> false.
```

1．求值 member(dog, [cat, man, dog, ape])。

2．第 1 个子句模式匹配成功，且有绑定{H|->dog, H1|->cat}，但是保护测

试失败。

3. 第 2 个子句模式匹配成功，且有绑定{H|->dog, T|->[man, dog ape]}，本子句没有保护测试,所以系统就用当前 H 和 T 的绑定值求取 member(H, T)。

4. 求值 member(dog, [man, dog, ape])。

5. 如先前一样，这次第 2 个子句匹配成功，且有绑定{ H|->dog, T|->[dog ape]}.

6. 求值 member(dog, [dog, ape])。

7. 第 1 个子句模式匹配成功，且有绑定{ H|->dog, H1|->dog}。这时保护测试也成功。

8. 求得值 true，最终结果即是 true。

注意每次进入到一个函数子句中时，使用的是一组新的变量绑定值，所以上面第 3 步中变量 H 和 T 的值与第 5 步中是不同的。

3.3.7 函数体

函数体是一序列的表达式。序列中的各个表达式依次求值，而函数体的值是序列中最后一个表达式的求值结果。

例如，假设我们定义了一个函数来操作一个银行帐户：

```
deposit(Who, Money) ->
    Old = lookup(Who),
    New = Old + Money,
    insert(Who, New),
    New.
```

这个函数的函数体由 4 个语句的序列组成。如果我们要求值表达式 deposit(joe, 25)，那么就会带着绑定集{ Who|->dog, Money|->25}进入上面的函数体。接着 lookup(Who)被调用，假设返回值是 W，这个返回值（即 W）就被匹

配到自由变量 Old，匹配成功。接着绑定集为{ Who|->dog, Money|->25, Old|->W}，继续……

3.3.8 尾递归

进行函数调用时，如果函数体中的最后一个语句都是调用系统里的其他函数，这个函数调用就是*尾递归的*（tail-recursive）。

例如，看看下面的函数：

```
p() ->
    ...
    q(),
    ...

q() ->
    r(),
    s().
```

在执行函数 p 的某个时候，函数 q 被调用。q 中最后去调用 s，当 s 返回的时候，将值返回给 q，但是 q 原封不动地将该值返回给 p。

这里函数 q 中最后对 s 的调用称为*尾调用*（tail-call），在传统的栈机制计算机（stack machine）中，尾调用可以编译成仅仅跳转到 s 的代码处。不需要把返回地址压入栈中，因为程序执行到这里的时候，栈中的返回地址本来就是对的，故 s 的调用结束不需要返回到 q 中，而是直接返回到 p 函数体中对 q 调用的地方就可以了。

如果函数的所有可能执行路径都以尾调用结束，就说明该函数是尾递归的。

这里值得注意的是尾递归函数可以在循环结构（loop）中被调用而不消耗栈空间，这一点很重要。这种函数通常称为“迭代函数”（iterative function）。

许多函数既可以用迭代风格来编写，也可以用非迭代（递归）风格来编写。为了阐明这一点，我们来看用这两种风格编写的求阶乘的函数。首先是非尾递归

的方式：

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

为了用尾递归的方式编写，就需要用到一个额外的函数：

```
factorial(N) -> factorial_1(N, 1).


factorial_1(0, X) -> X;
factorial_1(N, X) -> factorial_1(N-1, N*X).
```

许多非尾递归的函数可以通过引入一个辅助函数，增加一个额外参数[6]的方法改写成尾递归的。

Erlang 中的许多函数是被设计成运行在无限循环结构（infinite loop）中的，特别是在客户－服务器模式中，就假设了服务器会无限循环地运行。这种循环结构就必须写成尾递归的方式。一个典型的无限循环结构可能如下所示：

```
loop(Dict) ->
    receive
        {store, Key, Value} ->
            loop(dict:store(Key, Value, Dict));
        {From, {get, Key}} ->
            From ! dict:fetch(Key, Dict),
            loop(Dict)
    end.
```

这就是尾递归。

3.3.9 特殊形式

Erlang 中有两种特殊形式，用于表达式顺序化地条件求值。它们是 case 语句和 if 语句。

[6] 称之为聚集器。

3.3.10 case 语句

case 语句的语法形式如下：

```
case Expression of
    Pattern1 -> Expr_seq1;
    Pattern2 -> Expr_seq2;
    ...
end
```

case 语句这样被求值：首先对 Expression 求值，假设求得的值为 Value。然后 Value 依次与 Pattern1、Pattern2…等等匹配，直到有一个匹配成功。一找到匹配成功的模式 Pattern[I]，该模式对应的表达式序列 Expr_seq[I]就被求值，该 Expr_seq[I]的求值结果就是这个 case 语句的值。

3.3.11 if 语句

另一种条件求值的原语是 if，其语法形式如：

```
if
    Guard1 ->
        Expr_seq1;
    Guard2 ->
        Expr_seq2;
    ...
end
```

if 语句这样被求值：首先求 Guard1 的值，如果其取值为 true，则 if 语句的值就是对表达式序列 Expr_seq1 求值的结果。如果 Guard1 取值不为 true，则 Guard2（以及其后的保护式 Guard[i]）被依次求值，直到有一个保护式匹配 true 成功。if 语句必须至少有一个保护式取值为 true，否则将产生一个异常。通常 if 语句的最后一个保护式为原子 true，以确保其他保护式都失败时有最后一条能够成为 if 语句的返回值。

3.3.12 高阶函数（higher order function）

高阶函数就是将函数作为输入参数或返回值的函数。lists 模块中的 map 函数就是前一种高阶函数的例子。它是如下定义的：

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

map（F,L）将 F 应用于列表 L 的每一个元素产生一个新的队列，如下例：

```
> lists:map(fun(I) -> 2 *I end, [1,2,3,4]).
[2,4,6,8]
```

高阶函数可以用来为语言中原本不存在的语法结构创建*控制抽象*（control abstraction）。

例如，C 语言中提供了 for 的循环结构，可以这样用：

```
sum = 0;
for(i = 0; i < max; i++){
   sum += f(i)
}
```

Erlang 中没有 for 循环，但是我们可以很轻松地创造一个：

```
for(I, Max, F, Sum) when I < Max ->
   for(I+1, Max, F, Sum + F(I));
for(I, Max, F, Sum) ->
   Sum.
```

它可以这样用：

```
Sum0 = 0,
Sum = for(0, Max, F, Sum0).
```

我们也可以定义返回新函数的函数。下面这个在 Erlang 的 shell 里运行的例子就说明了这一点：

```
1> Adder = fun(X) -> fun(Y) -> X + Y end end.
```

```
#Fun<erl_eval.5.123085357>
2> Adder10 = Adder(10).
#Fun<erl_eval.5.123085357>
3> Adder10(5).
15
```

这里变量 Adder 函数包含一个 X 变量；求值 Adder(10)将 X 绑定为 10，并返回一个函数 fun(Y) –> 10 + Y end。

如果异常聪明的话，还可以定义递归函数，比如阶乘：

```
6> Fact = fun(X) ->
            G = fun(0,F) -> 1;
                   (N, F) -> N*F(N-1,F)
                end,
            G(X, G)
          end.
#Fun<erl_eval.5.123085357>
7> Fact(4).
24
```

函数还可以用“fun 名字/参数个数”的语法形式来引用。例如下面的表达式：

```
X = fun foo/2
```

是下面的写法的缩写形式：

```
X = fun(I, J) -> foo(I, J) end
```

其中 I 和 J 是在定义 X 的函数中没有出现过的自由变量。

3.3.13 表理解（list comprehension）

表理解是产生值列表的表达式。表理解的语法形式如下：

```
[X || Qualifier1, Qualifier2, ...]
```

这里 X 是一个任意表达式，而每一个限定词（qualifier）是一个生成器（generator）或一个过滤器(filter)。

- 生成器写成 Pattern<-ListExpr 的形式，这里 ListExpr 必须是一个取值为项式列表（list of terms）的表达式。
- 过滤器可以是谓词表达式或布尔表达式。

举个例子，著名的快速排序算法可以用两个表理解表达式来表示：

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

这里“++”是中缀添加运算符（infix append operator）。

如果你对填字游戏（crossword puzzle）感兴趣的话，你可以用下面的 perms 函数计算出一个串的所有排列：

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

这里的中缀运算符“X--Y”的作用是把在列表 X 中而不在列表 Y 中的所有元素作为一个列表拷贝出来。

如下例所示：

```
> perms("123").
["123","132","213","231","312","321"]
```

3.3.14 二进制数（binary）

二进制数是为存储非类型化数据（untyped data）而设计的，最初用于存储大量非结构化数据和高效的 I/O 操作。二进制数比列表或元组具有更高的空间效

率。例如，一个字符串作为列表存储时，每个字符需要 8 个字节的空间，而作为一个二进制数来存时，每个字符只需要 1 个字节的空间，再加上一个很小的固定开销。

内置函数 list_to_binary 可以把一个 io 表（io-list）转换成一个二进制数，binary_to_list 则可以进行相反的操作；term_to_binary 把一个任意项式转换成一个二进制数，而 binary_to_term 则反之。

注意：io 表是指其所有元素都是小整数（小整数是指在 0～255 范围内的整数）或二进制数或 io 表的列表。内置函数 list_to_binary(A)是将 io 表 A 扁平化，并且据其产生一个二进制数。binary_to_list/1 返回的是一个扁平的小整数列表。只有当 A 是一个扁平的小整数列表时，binary_to_list 才与 list_to_binary(A)严格相反。

可以用 concatenate_binaries 函数将二进制数的列表连接起来，并且可以用 split_binary 函数将单个二进制数分割成两个二进制数。

看下面的 shell 中的例子，就说明了一些对二进制数的操作：

```
1 > B1=list_to_binary([1,2,3]).
<<1,2,3>>
2> B2=list_to_binary([4,5,[6,7],[],[8,[9]],245]).
<<4,5,6,7,8,9,245>>
3 > B3=concat_binary([B1,B2]).
<<1,2,3,4,5,6,7,8,9,245>>
4> split_binary(B3,6).
{<<1,2,3,4,5,6>>, <<7,8,9,245>>}
```

表达式 1 将列表[1, 2, 3]转换成二进制数 B1。这里<<I1, I2, …>>表示由字节 I1,I2…等等组成的一个二进制数。

表达式 2 将一个 io 表转换成一个二进制数。

表达式 3 将两个二进制数 B1 和 B2 连接成一个二进制数 B3，表达式 4 将 B4 分割成两个二进制数。

```
5> B = term_to_binary({hello,"joe"}).
<<131,104,2,100,0,5,104,101,108,108,111,107,
0,3,106,111,101>>
6> binary_to_term(B).
{hello,"joe"}
```

内置函数 term_to_binary 将其参数转换成一个二进制数。相反的函数是 binary_to_term，它从二进制数中造出一个项式。由 term_to_binary 产生的二进制数是以所谓“外部项式格式”（external term format）的方式存储的。通过 term_to_binary 将项式转换成二进制数以后，可以存入文件，作为消息通过网络等发送，而以后又可以将该项式还原。这对于把复杂的数据结构存入文件或把复杂数据结构发送给远程的计算机来说，是非常非常重要的。

3.3.15 位语法

位语法为构造二进制数和对二进制数的内容进行模式匹配提供了一种符号。为了理解二进制数是如何构造的，下面给出一些在 shell 中的例子：

```
1> X=1,Y1=1,Y2=255,Y3=256，Z=1.
2> <<X,Y1,Z>>.
<<1,1,1>>
3> <<X,Y2,Z>>.
<<1,255,1>>
4> <<X,Y3,Z>>.
<<1,0,1>>
5> <<X,Y3:16,Z>>.
<<1,1,0,1>>
6> <<X,Y3:32,Z>>.
<<1,0,0,1,0,1>>
```

第 1 行定义了一些变量 X,Y1...Y3 和 Z。第 2 行用 X，Y1 和 Z 构造了一个二进制数，结果是<<1，1，1>>。

第 3 行 Y2 是 255，其值被原封不动地拷贝到二进制数的第 2 个字节中。当

我们试图用 Y3 来构造一个二进制数时，它的值被切断了，因为 256 在一个字节里放不下。而第 5 行的量词“:16”修正了这个错误。

如果我们不指定一个整数的空间大小，它就被默认设定成占 8 比特（bit）。第 6 行显示了一个 32 比特量词的效果。

我们不仅可以指定一个整数的存储空间大小，也可以指定其字节序，例如：

```
7> <<256:32>>.
<<0,0,1,0>>
8> <<256:32/big>>.
<<0,0,1,0>>
9> <<256:32/little>>.
<<0,1,0,0>>
```

第 7 行显示用整数 256 创建了一个二进制数，并打包（pack）成 32 比特。第 8 行告诉系统系统按“大头”字节序创建一个 32 比特的整数，第 9 行用“小头”字节序。

位域（bit field）也可以按下面的方式打包：

```
10> <<1:1,2:7>>.
<<130>>
```

上例用一个 1 比特域和一个 7 比特域创建了一个单个字节的二进制数。

与打包成二进制数相反，也可以将二进制数拆包：

```
11> <<X:1,Y:7>> = <<130>>.
<<130>>
12> X.
1
13> Y.
2
```

第 11 行是第 10 行的反过程。

对二进制数的模式匹配操作最先是设计用来处理包数据的。模式匹配在二进

制数的 0 个或多个“段”（segment）上依次进行。每一段按照如下形式来记述：

Value:Size/TypeSpecifierList

这里 TypeSpecifierList 是一个 End-Sign-Type-Unit 格式的以连字符分隔的条目表，各个条目意义如下：

- End——指明机器的字节序，可以为 big(译注：大头序，以下类似)，little(小头序)或 native(本地序)。
- Sign——可以为 signed(有符号)或 unsigned(无符号)。
- Type——可以为 integer（整型），float（浮点型）或 binary（二进制）。
- Unit——按照“unit:Int”的格式，这里 Int 是一个 1～256 范围内的字面整数（literal integer）。这样这个段的大小就等于 Size×Int 比特，这个总大小必须是 8 比特的倍数。

上面的每个条目都可以省略，且可以以任意顺序出现。

在《Erlang Open Source Distribution》一书[34]的名为“4.4.版以后 Erlang 的扩展”的章节中有一个使用二进制数的非常漂亮的例子。该例子展示了在如何通过一个模式匹配操作解析 IPv4 的报文，如下图：

```
1  -define(IP_VERSION, 4).
2  -define(IP_MIN_HDR_LEN, 5).
3
4  ...
5  DgramSize = size(Dgram),
6  case Dgram of
7    <<?IP_VERSION:4, HLen:4, SrvcType:8, TotLen:16,
8      ID:16, Flgs:3, FragOff:13,
9      TTL:8, Proto:8, HdrChkSum:16,
10     SrcIP:32,
11     DestIP:32, RestDgram/binary>> when HLen >= 5, 4*HLen =< DgramSize ->
12         OptsLen = 4*(HLen - ?IP_MIN_HDR_LEN),
13         <<Opts:OptsLen/binary,Data/binary>> = RestDgram,
14         ...
```

上例第 7～11 行在单单一个模式匹配表达式中匹配 IP 报文。这个模式是比较复杂的，书写就占了 5 行之多，不过该模式也明释了不是以字节为分界的数据是如何被轻松提取的（例如 Flgs 域和 FragOff 域就分别是 3 比特和 13 比特长）。

只要 IP 报文匹配成功，其报头和数据部分就被分离开了（如第 12～13 行）。

3.3.16 记录（record）

记录提供了把一个名字和元组的一个具体元素关联起来的一种方法。元组的问题在于当它包含的元素的个数变得很大时，就很难记得哪个元素究竟是什么意思。

对于一个小的元组来说，这不是什么问题，所以我们经常可以看到程序中对元素个数很少的元组直接进行操作。

随着元组的元素个数越来越大，要跟踪元组中每个元素的含义就变得越来越困难。当元组的元素个数很多，或我们出于其他目的[7]希望元组里的每个元素都有一个名字的时候，我们就可以用记录来代替元组。

记录的定义按照如下语法形式来书写：

```
-record(Name, {
                Key1 = Default1,
                Key2 = Default2,
                ...
              }).
```

这里 Name 是记录的名字。Key1，Key2...是记录中字段的名字。记录中的每个字段可以有一个默认值，供记录创建而没有为该字段指定值时使用。

例如，我们如下所示定义一个 person 的记录：

```
-record(person, {
                 firstName="",
                 lastName = "",
                 age}).
```

定义了一个记录以后，就可以创建该记录的实例，例如：

```
Person = #person{firstName="Rip",
```

[7] 例如，为了文档化。

```
lastname="Van Winkle",
age=793
}
```

这就创建了一个“Rip Van Winkel”[8]的人。

我们可以写程序对记录的字段进行模式匹配，并且创建新的记录。因此如果知道 Van Winkel 先生的生辰，我们就可以调用函数：

```
birthday(X=#person{age=N}) ->
    X#person{age=N+1}.
```

一旦上面的子句模式匹配成功，就会使得 X 绑定到整个记录，而 N 绑定到记录的 age 字段。X#person{age=K}则创建了 X 的一个拷贝，不过新的记录中的 age 字段以 K 替代。

3.3.17 Erlang 预处理程序（epp）

在 Erlang 模块编译之前，是由 Erlang 预处理程序 epp 来处理的。Erlang 预处理程序进行宏展开并插入所需的包含文件。

预处理程序的处理结果可以通过 compile:file(M, [‘P’]) 命令保存到文件。该命令编译 M.erl 文件中的所有代码，生成一个清单放入 M.P 文件，文件中所有的宏都已被展开，所有必需的包含文件都已经被包含进来了。

3.3.18 宏

Erlang 的宏写作：

```
-define(Constant, Replacement).
-define(Func(Var1, Var2,..., Var), Replacement).
```

当遇到形如“?MacroName”的表达式的时候，宏就被Erlang预处理程序展开。宏定义中所有的变量都用宏调用处的对位参数原封不动地替换掉。

```
-define(macro1(X, Y), {a, X, Y}).
```

[8] 连“Google大人”都不知道Van Winkel先生多大年纪，所以 793 岁纯属臆测。

```
foo(A) ->
    ?macro1(A+10, b)
```

被扩展为：

```
foo(A) ->
    {a,A+10,b}.
```

宏调用的参数和和返回值必须是完整、协调的表达式。因此不能像下面这样使用宏：

```
-define(start, {).
-define(stop, }).

foo(A) ->
    ?start,a,?stop.
```

补充一点，系统还有一些关于当前模块的信息的预定义宏。它们是：

- ?FILE：扩展为当前文件名。
- ?MODULE：扩展为当前模块名。
- ?LINE：扩展为当前行号。

3.3.19 包含文件

Erlang 中文件这样被包含：

```
-include(Filename).
```

通常包含文件都具有“.hrl”的扩展文件名。FileName 变量应该包含一个被包含文件的绝对路径或相对路径，以便预处理程序找到正确的文件。

库文件可以这样被包含：

```
-include_lib(Name).
```

例如：

```
-include_lib("kernel/include/file.hrl").
```

像这种情况下 Erlang 编译器就能找到正确的包含文件。

3.4 并发（concurrent）编程

在 Erlang 中，可以通过调用 spawn 原语来创建并行进程，表达式如：

```
Pid = spawn(F)
```

这里 F 是一个参数个数为 0 的函数，该表达式创建了一个对 F 求值的并行进程。spawn 返回一个进程标识符（Pid），通过 Pid 我们可以访问该进程。

语句“Pid ! Msg”表示将一个消息 Msg 发送给进程 Pid。消息可以用 receive 原语来接收，语法形式如下：

```
receive
    Msg1 [when Guard1] ->
        Expr_seq1;
    Msg2 [when Guard2] ->
        Expr_seq2;
    ...
    MsgN [when GuardN] ->
        Expr_seqN;
    ...
    [; after TimeOutTime ->
        Timeout_Expr_seq]
end
```

Msg1…MsgN 都是模式，模式也可能带有保护式。当向一个进程发送一个消息时，该消息就被放进一个属于该进程的邮箱（mailbox）中。下次进程对 receive 语句进行求值的时候，系统就会查看一下邮箱，并且试图拿邮箱中的第 1 条消息与当前 receive 语句中的所有模式进行匹配。如果邮箱中收到的消息没有与任何模式匹配成功，则该消息就被转移到一个临时的“保管”队列中，进程被挂起，等待下一条消息。如果消息匹配成功，且与之匹配的模式所带的保护式也取真的

话，该模式后面的语句系列就会依次被求值。同时，所有被临时保管的消息也被放回到进程的邮箱中。

receive 语句可以有一个可选择的超时值。如果在超时期限内没有收到可匹配的消息的话，超时条件下面的表达式就会被求值。

3.4.1 注册进程名

当我们想向一个进程发送消息的时候，我们需要知道该进程的名字。这是很安全的，但是当我们要向一个给定的进程发送消息时必须设法获取该进程的名字，这一点某种程度上会带来一些不便。

如下表达式：

```
register(Name, Pid)
```

会创建一个全局进程，并把原子 Name 与继承标识符 Pid 关联起来。这样就可以通过调用“Name ! Msg”来给进程 Pid 发送消息。

3.5 错误处理

在 Erlang 里求取一个函数的值一定只有两种结果：要么函数就返回一个值，要么它就产生一个异常。

异常可以隐式地产生（即由 Erlang 运行时系统产生），也可以通过调用 exit（X）原语来显式地产生。隐式地产生异常将在下一节讲述。

下面是一个隐式地产生异常的一个例子，假设我们写一个函数如：

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

求值 factorial(10)将返回一个值 3628800，但是如果求取 factorial(abc)的值，则将产生一个异常{‘EXIT’, {badarith,…}。异常会引起程序停下正在执行的操作转而去做其他的事情——这就是它们被称作异常的原因。如果我们写：

```
J = factorial(I)
```

如果 I 是整数的话，我们期望 J 被赋上 factorial(I)的值。如果用一个非整数的参数调用 factorial，则该语句将没有意义。下面的程序片断：

```
I = "monday",
J = factorial(I),
```

就是没有意义的，因为我们无法计算 factorial(“monday”)。因此 J 并没有赋值而且赋值也没有任何意义。

许多编程语言对于有效值和异常之间的区别视而不见，即使程序已经变得毫无意义也依然盲目地执行下去。

3.5.1 异常

异常是为 Erlang 运行时系统所检测到的一种非正常状态。Erlang 程序是被编译成虚拟机指令并且由一个虚拟机仿真器来执行的。而虚拟机仿真器是 Erlang 运行时系统的一部分。

一旦仿真器检测到某种不知所措的状态，它就会产生一个异常。一共有 6 种类型的异常：

1. 值错误（value error）——就是诸如“被 0 除”之类的错误。这种情况下传给函数的参数的类型是正确的，但是值错了。
2. 类型错误（type error）——这类错误是指调用 Erlang 的内置函数的时候所填的参数类型不正确。例如，有一个内置函数为 atom_to_list(A)，是将原子 A 转换成其 ASCII 码的一个整数列表。如果变量 A 并不是一个原子，运行时系统就会产生一个异常。
3. 模式匹配错误（pattern-matching error）——这类错误是指试图将一个数据结构与一些模式进行匹配，却找不到匹配成功的模式的错误。这种错误会在函数头匹配时产生，或者在诸如 case，receive 或 if 语句中进行匹配时产生。
4. 显式调用 exit（explicit exits）——这类错误是在显式调用表达式 exit(Why)时产生的，该调用会产生一个 Why 异常。

5. 错误传播（error propagation）——如果一个进程收到一个 exit 信号，它可以选择停掉自己并把该 exit 信号传播给所有它连接着的进程（见 3.5.6 节）。

6. 系统异常(system exception)——运行时系统也许会因为内存耗尽或检测到一个内部表不一致时终结掉一个进程。这类错误不在程序员的控制范围之内。

3.5.2 catch 原语

可以通过调用 catch 原语将异常转换成有效值。我们可以试试在 Erlang 的 shell 里求值一个会导致异常产生的表达式，就可以印证这一点。我们来试试将值 1/0 绑定到自由变量 X 上，看看发生了什么：

```
1> X = 1/0.

=ERROR REPORT==== 23-Apr-2003::15:20:43 ===
Error in process <0.23.0> with exit value:
{badarith,[{erl_eval,eval_op,3},{erl_eval,expr,3},
{erl_eval,exprs,4},{shell,eval_loop,2}]}
** exited: {badarith,[{erl_eval,eval_op,3},
                      {erl_eval,expr,3},
                      {erl_eval,exprs,4},
                      {shell,eval_loop,2}]} **
```

这里在 Erlang 的 shell 里输入的表达式 X = 1/0 会引起一个异常，并且有一条错误消息被打印到了标准输出上。如果我们试图打印变量 X 的值，我们将看到：

```
2> X.
** exited: {{unbound,' X'},[{erl_eval,expr,3}]} **
```

显然，因为 X 并没有值，所以产生了另一个异常，所以打印出了另一条错误消息。

为了将异常转换成有效值，我们可以在一个 catch 语句中求值它，如下：

```
3> Y = (catch 1/0).
{' EXIT' ,{badarith,[{erl_eval,eval_op,3},
                     {erl_eval,expr,3},
                     {erl_eval,exprs,4},
                     {shell,eval_loop,2}]}}
```

现在 Y 就有一个值，即一个包含 2 个元素的元组，第 1 个元素是原子'EXIT'，第二个元素是项式{badarith, …}。Y 是一个标准的 Erlang 项式，能够像其他任何 Erlang 数据结构一样自由地被检测和使用。如下表达式：

```
Val = (catch Expr)
```

在一定的上下文中求取 Expr 的值。如果求值正常结束，那么 catch 就返回该表达式的值；如果求值过程中发生了异常，则求值过程立即终止并产生一个异常。异常是一个用来描述错误的 Erlang 对象，在这种情况下，catch 的值就是所产生的异常的值。

如果求值（catch Expr）返回了一个形如{'EXIT', W}的项式，那么我们就认为表达式 Expr 因 W 而终止了。

如果一个 catch 表达式的内部任何地方产生了一个异常，那么 catch 的值就是该异常的值。如果在 catch 的作用域之外产生了一个异常，那么产生该异常所在进程就要死掉，并且该异常将传播给当前连接到该进程的所有进程。进程连接可以通过调用内置函数 link（Pid）来创建。

3.5.3 exit 原语

可以通过调用 exit/1 原语来显式地生成一个异常。下面是一个例子：

```
sqrt(X) when X < 0 ->
    exit({sqrt,X});
sqrt(X) ->
    ...
```

上例中，如果用一个负数作为参数 X 的值调用 sqrt 的话，就会产生异常{sqrt, X}。

3.5.4 throw 原语

throw 原语用于改变异常的语法形式。

- 如果在某个函数 F 的作用域内调用 exit(P)产生了一个异常，那么求值(catch F)的结果将是一个{‘EXIT’, P}形式的项式。
- 如果在某个函数 F 的作用域内调用 throw(Q)产生了一个异常，那么求值(catch F)的结果就将是项式 Q。

throw 可以用来区别用户产生的异常和运行时系统产生的异常。

3.5.5 已修正错误与未修正错误

假设我们写了如下程序片断：

```
g(X) ->
    case (catch h(X)) of
        {'EXIT', _} ->
            10;
        Val ->
            Val
    end.

h(cat) -> exit(dog);
h(N) -> 10*N.
```

求值 h(cat)将产生一个异常，而求值 h(20)将返回值 200。求值 g(cat)或 g(dog)将返回值 10 而求值 g(10)将返回值 100。

当我们求值 g(cat)时，将发生如下一系列事情：

1．求值 h(cat)被求值。

2．h 产生一个异常。

3．该异常被 g 捕获。

4．g 返回一个值。

当求取 g(dog)之值时，将引发下列事情：

1．h(dog)被求值。

2．在 h 的函数体中，变量 N 被绑定为值 dog。

3．求取 N=dog 时 N*10 的值。

4．在函数‘*’（译注：进行乘法运算的函数）中产生了一个异常。

5．该异常被传播给函数 h。

6．该异常被函数 g 捕获。

7．g 返回一个值。

如果我们仔细观察上面的过程我们就可以发现：在求值 h(dog)中产生了一个异常，而该异常在 g 中被捕获并被纠正了。

在此我们就可以说一个错误确实发生了，但是它被修正了。

如果我们是直接求值 h(dog)，那么将会产生异常，但是并没有被捕获和修正。

3.5.6 进程连接与监视者

一旦一个进程死掉，我们希望其他的进程得到通知。回想一下在 2.5 节我们说过我们需要这一点来编写一个可容错系统。有两种方式可以做到这一点，我们可以用进程连接或进程监视者。

进程连接是将一组进程聚合在一起的一种方式，在进程连接中，任意一个进程中发生了错误，其他所有的进程都将连带被停掉。

进程监视者是用一个单独的进程来监视系统中的所有其他的进程。

进程连接

catch 原语用于截获一个进程中发生的错误。那我们现在来问一问，如果程序的顶层 catch 都不设法修正一个它所检测到的错误的话，会发生什么事情呢？答案是该进程将终止。

出错的原因只是异常的一个参数。当一个进程出错时，出错的原因将被广播给它所归属的一个所谓“连接集”（link set）的所有其他进程。进程 A 可以通过调用内置函数 link(B)将 B 加入到它的连接集中。进程之间的连接是对称的，也就是说，如果 A 连接到了 B，那么 B 也连接到了 A。

连接也可以在进程被创建的时候创建。如果 A 通过下面的调用方式来创建进程 B：

```
B = spawn_link(fun() -> ... end),
```

那么进程B在创建的时候就连接到了进程A。这种调用方法在语义上等价于先调用spawn紧接着调用link，只不过这两个表达式是一起执行的，不是分步的。spawn_link原语的引入，是为了规避进程在创建的过程中还没有来得及执行link语句就死掉这种罕见的编程错误[9]。

如果进程 P 死掉的时候产生了一个{‘EXIT’, Why}的未捕获异常，那么退出信号{‘EXIT’, P, Why}就会被发送给进程 P 的连接集中的所有进程。

我刚刚提到“信号”。信号是进程终止的时候在进程之间传递的一种东西。信号是一个{‘EXIT’, P, Why}形式的元组，这里 P 是终止的进程的 Pid，而 Why 是一个描述终止原因的项式。

任何收到 Why 不为 normal（正常）的退出信号的进程都将死掉。对于这一规则有一个例外：如果接收进程是一个系统进程，那么该进程不会死掉，而是将退出信号转换成一个正常的进程间消息，并被添加到该进程的邮箱中。可以调用内置函数 process_flag(trap_exit, true)来将一个一般进程变成一个系统进程。

系统进程处理其他进程的故障的典型代码片断如下：

```
start() -> spawn(fun go/0).
```

[9] 例如，在一个进程试图创建另一个进程的过程中如果使用了一个根本不存在的模块中的代码，就会发生这种错误。

```
go() ->
    process_flag(trap_exit, true),
    loop().

loop() ->
    receive
        {'EXIT',P,Why} ->
            ... handle the error ...
    end
```

另外一个原语exit/2 将完成这个拼图。exit(Pid, Why)将给进程Pid发送一个原因为Why的退出信号。调用exit/2 的进程本身不会终止，因此这种消息能够用来“伪装”一个进程的死亡[10]。

不过对于“系统进程将会把所有信号都转换成消息”这一点来说，也存在一个例外：如果调用 exit(P, kill)，将向 P 发送一个*不可阻挡的退出信号*（unstoppable exit），收到该信号后进程 P 将不顾一切后果地终结掉。exit/2 的这种用法在客客气气地请求一个进程自觉终结而遭到拒绝的时候就有用。

进程连接对于建立进程群组（group）是有用的，进程群组中的一个进程出错，所有进程都将死掉。通常我们把属于一个应用的进程连接起来，并且让其中的一个进程充当“监视者”的角色。监视者被设定来捕获退出信号。如果进程群组中有任何一个进程出错了，群组中除了监视者以外的其它所有进程都将死掉，而由监视者来接收群组中的进程的出错消息，这些出错消息描述了故障原因。

进程监视者

进程连接对于整个进程群组来说是有用的，但是对于非对称的进程对的监视来说没什么用。在典型的客户－服务器模型中，客户与服务器的关系在考虑到错误处理的时候就是非对称的。假设一个服务器处理着大量不同客户的大量长时间会话（long-lived session），那么当服务器崩掉的时候我们可能会杀死所有的客户，

[10] 这是一个特性，而不是bug（缺陷）。

但是当某一个客户崩掉的时候我们并不希望杀掉服务器。

erlang:monitor/2 原语就是用来设置一个监视者的。如果进程 A 有求值：

```
Ref = erlang:monitor(process, B)
```

那么当 B 因为原因 Why 死掉的时候，就会向 A 发送一条如下格式的消息：

```
{'DOWN', Ref, process, B, Why}
```

监视消息的发送者 A 和接收者 B 都不必是系统进程。

3.6 分布式（distributed）编程

Erlang 程序能够很轻易地从一个单处理器平台移植到多处理器平台。每一个完整的自包含的（self-contained）Erlang 系统被称为一个 *节点*（node）。一个宿主操作系统上面可以跑一个或多个 Erlang 节点。多个 Erlang 节点可以运行在同一个操作系统上这一点简化了分布式应用的测试。可以通过让所有的节点运行在同一个处理器上，来进行一个分布式应用程序的开发和测试。当应用投入使用时，可以将在同一处理器上工作的不同节点变成分布式网络处理器上的不同节点。除了定时操作（timing）以外，所有操作的工作方式都应该与在同一个节点严格相同。

分布式处理需要如下两个原语：

- spawn(Node, Fun)——在一个远端节点 Node 上产生一个处理函数是 Fun 的进程。
- monitor(Node)——用来监视整个节点的行为。

这里的 monitor 类似于 link，不同之处在于被控制的对象是一整个节点而不是某个进程的行为。

3.7 端口（ports）

端口给 Erlang 程序与外界的通信提供了一种机制。端口可以通过调用内置函数 open_port/2 来创建。每个端口都有一个与之相关联的“控制进程”

（controlling-process）。我们称控制进程*拥有*（own）该端口。从该端口收到的所有的消息都被发送给其控制进程，且只有其控制进程才可以向该端口发送消息。端口的控制进程被初始化为创建该端口的进程，但是这个进程可以被改变。

如果 P 是一个端口，而 Con 是其控制进程的 pid，那么可用调用如下的表达式来让端口做某些事情：

```
P ! {Con, Command}
```

这里的 Command 变量可以取如下三种可能的值：

- {command, Data}——把数据 Data 通过端口发送给外部对象。Data 必须要是一个 io 表（参见 3.3.14 节关于“io 表”的定义）。io 表是扁平化的，表中所有的数据元素都被发送给外部的应用程序。
- close——关闭一个端口。被关闭的端口必须向控制进程回复一个{P, closed}的消息。
- {connect, Pid1}——将端口的控制进程变为 Pid1。该端口必须要给原来的控制进程回应一个{Port, connected}的消息，此后该端口收到的所有新消息都经发送给新的控制进程。

通过端口收到的所有外部应用程序的数据都将以{Port, {data, D}}的消息格式发送给其控制进程。

消息的确切格式以及该消息是如何组帧的，则取决于端口是如何被创建的。更多细节请参见参考文献[34]。

3.8 动态代码替换

Erlang 支持一种简单的动态代码替换机制。在一个运行时的 Erlang 节点上，所有的进程都共享同一份代码。因此我们必须要考虑如果我们替换了一个运行时系统的代码，会发生什么事情？

在顺序化编程语言里只有一个控制线（thread of control），所以如果我们期望动态替换代码，我们只需要考虑对该唯一控制线的影响。在一个顺序化系统里，

如果我们期望改变代码，我们实际上通常的做法是停止该系统，替换代码，然后重新启动程序。

然而在一个实时控制系统中，我们通常并不希望停下该系统来替换代码。在某些特定的实时控制系统中，我们也决不允许关掉系统来替换代码，所以这些系统需要被设计成不停止系统而支持代码替换。这种系统的一个例子就是 NASA 设计的 X2000 卫星控制系统[2]。

Erlang 系统的每个模块的代码允许存在两个版本。如果一个模块的代码被加载进来，那么调用该模块代码的所有新启动的进程就会动态地连接到该模块的最新版本上。如果一个模块后来被替换了，那么原来执行该模块代码的进程就既可以选择继续执行老的代码，也可以选择执行新加载的代码。这种选择决定于该代码是如何被调用的。

如果代码是通过全修饰名被调用的，即以“ModuleName:FuncName”的方式调用的，那么就总是调用该模块的最新版本，否则就调用该模块的当前版本（译注：老版本）。举个例子，假设我们写了下面的一个服务循环：

```
-module(m).
...
loop(Data, F) ->
    receive
        {From, Q} ->
            {Reply, Data1} = F(Q, Data),
            m:loop(data1, F)
    end.
```

在模块 m 第一次被调用到的时候，该模块就被加载了进来，譬如从外部调用 m:loop 函数的时候。因为这时候 m 模块只有一个版本，所以调用的是当前模块的 loop 函数。

假设我们现在修改了模块 m 的代码，重新编译并加载了该模块。那么当我们在最后的 receive 语句中调用 m:loop 函数时，新版本的 m 模块中的代码就会被调用。注意，所调用的新代码与老代码的兼容性由程序员来保障。强烈建议把所

有的代码替换调用都做成尾调用（参见 3.3.8 节）的，这样的话一个尾调用就不必返回到老代码中，因此在一个尾调用以后，一个模块的所有的老代码就可以被安全地删除了。

如果我们希望继续执行当前模块（老版本）的代码，而不切换到新模块的代码中，那么我们就可以用非全修饰名调用的方式写该 loop 循环，即：

```
-module(m).
...
loop(Data, F) ->
    receive
        {From, Q} ->
            {Reply, Data1} = F(Q, Data),
            loop(data1, F)
    end.
```

在这种情况下，模块的新版本的代码就不会被调用。

灵活地运用这种机制使得进程可以同时执行不同模块的新、老代码版本。

需要注意，代码存在两个版本有一个局限性。如果第三次试着重新载入一个模块，则正在执行第一个模块的所有进程将被全部杀掉。

除了以上调用约定以外，还有许多内置函数用来达到代码替换的目的。这些函数在参考文献[5]中有详尽的描述。

3.9 一种类型符号（type notation）

我们在构建一个软件模块的时候，是怎么描述该模块的用法的呢？通常，我们都会说通过对一组 API（Application Programming Interface）的调用来使用它。这组 API 就是模块提供可供外部调用的一组函数，以及这些函数的输入值的类型的要求和返回值的类型的描述。

下面的例子说明了如何用 Erlang 的类型符号来指定一些函数的类型：

```
+type file:open(fileName(), read | write) ->
```

```
        {ok, fileHandle()}| {error, string()}.

    +type file:read_line(fileHandle()) ->
        {ok, string()}| eof.

    +type file:close(fileHandle()) ->
        true.

    +deftype fileName() = [int()]
    +deftype string() = [int()].
    +deftype fileHandle() = pid().
```

每一种 Erlang 的原始数据类型都有它的类型。这些原始类型是：

- int()——是整数类型。
- atom()——是原子类型。
- pid()——是 Pid 类型。
- ref()——是引用类型。
- float()——是 Erlang 的浮点数类型。
- port()——是端口类型。
- bin()——是二进制类型。

列表类型、元组类型以及选择（alternation）类型是如下递归式地定义的：

- 如果 T1,T2,…,Tn 都是类型的话，那么{T1,T2,…,Tn}就是*元组类型*（tuple type)。此时如果{X1,X2,…,Xn}中的 X1 是 T1 类型，X2 是 T2 类型，……Xn 是 Tn 类型，我们就说{X1,X2,…,Xn}是{T1,T2,…,Tn}类型。
- 如果 T 是一个类型，那么[T]就是一个*列表类型*（list type)。如果[X1,X2,…,Xn]中的所有 Xi 都是 T 类型的话，那么我们就说

[X1,X2,…,Xn]是[T]类型。注意，空表[]的类型也是[T]，其中 T 是任意类型。

- 如果 T1 和 T2 都是类型，则 T1|T2 就是选择类型（alternation type）。如果 X 的类型可能是 T1 或者 T2，我们就说 X 的类型是 T1|T2。

可以通过如下的符号来引入新的类型：

```
+deftype name1() = name2() = ... = Type.
```

这里 name1、name2……等名字应遵循 Erlang 的原子（atom）的语法。Type 是类型变量，需按照 Erlang 的变量的语法来书写。例如我们可以定义：

```
+deftype bool() = true | false.
+deftype weekday() = monday|tuesday|wednesday|
                     thursday|friday.
+deftype weekend() = saturday() | sunday().
+deftype day() = weekday() | weekend().
```

函数类型按如下书写：

```
+type functionName(T1, T2, ..., Tn) -> T.
```

这里所有的 Ti 都是类型。如果在一个类型的定义中某个类型变量出现了不止一次，那么该类型的实例中与其定义对应的位置的所有变量都必须具有相同的类型。

下面是一些例子：

```
+deftype string() = [int()].
+deftype day() = number() = int().
+deftype town() = street() = string().

+type factorial(int()) -> int().
+type day2int(day()) -> int().
+type address(person()) -> {town(), street(), number()}.
```

最后，还有匿名函数的类型如下书写：

```
+type fun(T1, T2, ..., Tn) -> T end
```

因此，map/2 的类型就应该如下书写：

```
+type map(fun(X) -> Y end, [X]) -> [Y].
```

这里的类型符号是 Wadler&Marlow[49]所开发的类型符号的一种极其简化的版本。

3.10 讨论

本章介绍了 Erlang 很重要的一个子集，至少足以用来理解本论文中的所有例子。但是我还没有回答“Erlang 是用于编写可容错系统的恰当的语言吗？”我确信答案是“正是。”我在前文中曾经说过，用于编写可容错系统的语言一定要满足某些特征（参见 2.6 节 R1～R6）。我现在就来印证一下，Erlang 确实是满足了这些特征的，理由如下：

- 进程是 Erlang 的基础，所以 R1 满足。
- 因为 Erlang 中的进程就是错误封装单元，所以 R2 满足。如果一个进程因为软件原因终止的话，同一个 Erlang 节点中的其它进程将不会受到影响（当然，除非有进程被连接到了将会终止的进程上，这种情况下进程间的影响是有意的）。
- 如果进程中的函数用了错误的参数来调用，或者系统的 BIF 用了错误的参数来调用，那么该进程就立即终止。即刻终止符合 Gray 的*速错进程*（fail-fast process）的概念（参见 2.10 节），也符合 Schneider 的*错即停处理器*（fail-stop processor）的概念（参见 2.10 节），还符合 Renzel 关于我们必须检测错误，并尽量早地停下来的观点（参见 2.10 节）。
- 当一个进程出错时，出错的原因会被广播给该进程的当前连接集，因此满足 R3 和 R4。

- R5 由第 3.8 节描述的一种代码升级机制来满足。
- R6 在 Erlang 语言里没有被满足，但是在 Erlang 的库里得到了满足。持久存储可以用 dets 或 mnesia 来实现。dets 是一个单机的基于磁盘的存储系统。如果一个进程或者一个节点崩溃了，存储在 dets 中的数据却得以幸存。为了达到更好地保护数据的目的，数据应该被存储在物理上独立的两个节点上，这时候可以用 mnesia 数据库，它是 OTP 的一个应用程序。

我还要指出，Schneider 的“出错即停止”（halt on failure）、“错误状态属性”（Failure status property）、“稳定存储属性”（Stable storage property）（参见 2.10 节）等观点，也由 Erlang 语言自己或 Erlang 的库直接或间接地满足了。

4 编程技术

前面的章节讲述了 Erlang 语言，但是没有讲如何用 Erlang 编程。本章就是关于 Erlang 编程技术的。编程技术涉及如下诸方面：

- *抽象出并发*——某种意义上讲，并发程序比顺序化程序要难得多。为了避免在同一个模块里既有并发的代码又有顺序化的代码，我展示了如何将代码组织到两个模块里，其中一个全部是并发代码，另一个则只有纯的顺序化代码。
- *抱持 Erlang 的世界观*——在 Erlang 的世界里，万事万物都是进程。为了帮助我们*抱持*这种观点，我介绍了一种协议转换器（protocol converter）的思想，它有助于程序员建立任何事物都是 Erlang 进程这一观念。
- *Erlang 的错误观*——Erlang 的错误处理方式与其他语言有本质的区别。我将展示在 Erlang 中该如何编写出错情况下的程序。
- *显意编程*——这是一种程序员能够轻易就从源代码中看出编程者的意图的编程风格，而不是通过对代码进行表面的分析来猜测编程者的意图。

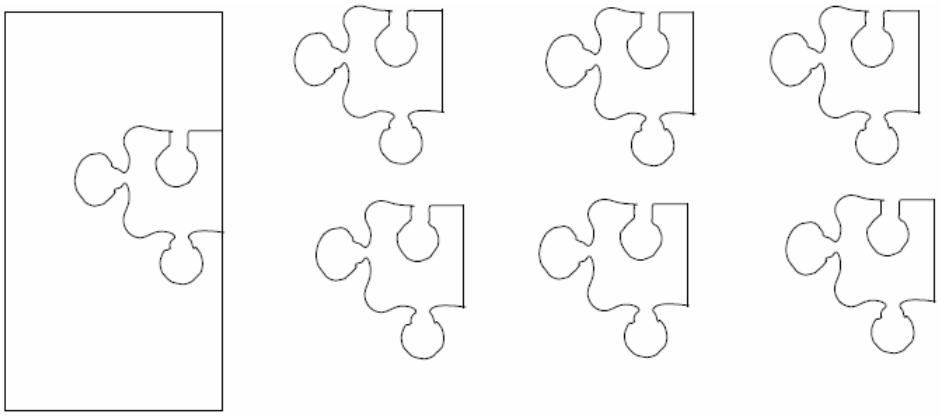


图 4.1: 一个通用部件及其插件。所有的并发和错误处理都在

通用部件中进行，插件则是用纯顺序化代码编写。

4.1 抽象出并发

我们编程的时候，经常期望将代码划分为“困难的”和“容易的”模块。困难的模块要少，由专家程序员来编写；而容易的模块多，由不那么有经验的程序员来编写。图 4.1 就显示了一个通用部件（困难的部分），和许多用来对通用部件进行参数化的“插件”（容易的部分）。

通用部件应该对插件隐藏起并发和容错机制的细节，而那些插件则应该只用有着良好类型定义的顺序化代码来编写。

接下来，我将要描述如何将一个客户-服务器（client-server）划分成一个通用部件和许多插件。

将一个系统划分成通用部件和插件是一种常用的编程技术——在我们的方式中的非常之处在于通用部件能够提供一个丰富的环境来执行插件。插件的代码中可以包含错误，插件的代码可以动态替换，整个插件可以在网络上自由搬移，所有的这些都不需要对插件的代码做任何额外的设计。

抽象出并发是用来划分一个大型软件系统的最有力的手段之一。运用该方法不仅可以轻轻松松地用 Erlang 来编写并发程序，更是可以把显式地对并发进行处理的代码约束在尽量少的模块中。

这样做的原因是，并发处理的代码一般都很难以无副作用（side-effect free）

的方式来编写，就使得并发程序比纯顺序化的、无副作用的代码更难以理解和分析。在一个包含大量进程的系统中，消息传递顺序化问题和潜在的死锁（dead-lock）或活锁（live-lock）问题会使得并发系统非常难以理解和编写。

用 Erlang 编写应用程序，用的最普遍的一种抽象就是客户-服务器抽象。事实上，在用 Erlang 编写的所有应用程序中，对客户-服务器抽象的使用远远多于对其它抽象的使用。例如，在 8.3.1 小节我们将看到，在 AXD301 系统使用的所有 behaviour 中，gen_server 这种 behaviour 就占到了 63%，该 behaviour 提供了一种客户-服务器抽象。

我以一个简单的通用客户-服务器 server1 开始，然后展示如何将它参数化构成一个名字服务程序。

我还将用两种方式来扩展这个简单的服务器，首先我将修改基本服务器来构造一个*可容错的*服务器 server2，然后把它扩展到提供*动态代码升级*功能的版本（server3）。服务器代码从 server1 到 server2 再到 server3 的一步步地演进，最终会演变成 gen_server，就是 OTP 库中的一种标准 behaviour。gen_server 的代码比这里展示的简单服务器的代码所要完成的任务要多许多。但是，gen_server 的原理与这里展示的简单服务器的原理是相同的。即，把客户-服务器分为一个通用部分——该部分负责并发处理，和一些插入模块——它们仅仅是将通用服务器用某种特殊的方式进行参数化，以创建一个具体的服务器的实例。

这两种扩展方式都被故意地简化了。为了简单地阐释其中包含的原理，我忽略了许多实现方面的问题。

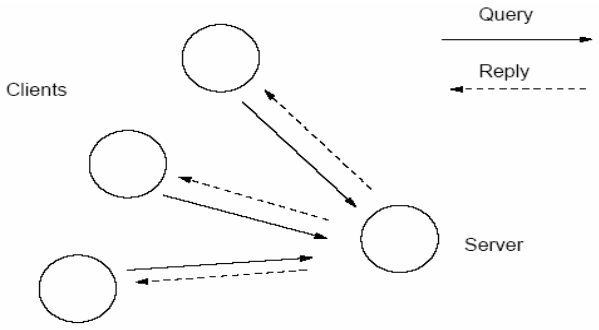


图 4.2: 客户-服务器模型

图 4.2 展示了客户-服务器模型。客户-服务器模型的特色是它有一个中心服

务器和任意个数的客户。客户-服务器模型通常用于资源管理业务。我们假设有许多不同的客户要共享某一公共的资源，而一个服务器负责管理该资源。

如果我们忽略该服务器的启动、停止和所有出错情况，那么我们就可以用一个单独的函数 F 来描述该服务器。

我们来假设服务器处于个 State 的状态，它收到来自某个客户的 Query 消息。服务器应当给该客户的查询返回一条 Reply 的消息，并自身状态变为 State1。

```
-module(server1).

-export([start/3, stop/1, rpc/2]).

start(Name, F, State) ->
    register(Name,
             spawn(fun() ->
                           loop(Name, F, State)
                   end)).

stop(Name) ->  Name ! stop.

rpc(Name, Query) ->
    Name ! {self(), Query},
    receive
        {Name, Reply} -> Reply
    end.

loop(Name, F, State) ->
    receive
        stop ->
            void;
        {Pid, Query} ->
            {Reply, State1} = F(Query, State),
            Pid ! {Name, Reply},
            loop(Name, F, State1)
    end.
```

图 4.3: 一个简单的服务器程序

这些值完全由服务器函数 F 来决定，并且通过如下的 Erlang 表达式计算：

```
{State1, Reply} = F(Query, State)
```

这个求值过程是在 server 内进行的。

最初的通用服务器的代码 server1.erl 如图 4.3 所示。客户桩子程序(client stub routine）rpc（13－17 行）向服务器发送了一条消息（14 行），并且等待一个应答（15－17 行)。服务器收到客户桩发来的消息（23 行），计算出一个应答和一个新状态（24 行)，将应答发送回客户（25 行)，然后递归地调用它自己（26 行)。注意对 loop/3 的递归调用（26 行）是一个尾调用（参见 3.3.8 小节），因为是尾调用，变量 State 不能再被任何代码访问，所以原来那些 State 占用的而现在无法通过 State1 访问到的所有存储空间都将最终由垃圾回收器（garbage collector）回收。因此也可以说 loop/3 是运行在本地存储器上（变量 State 即存放在一个本地存储器上）的一个固定大小的空间中的。服务器需要这个空间来存储它自己。server1.erl 导出了三个函数：

- start(Name, Fun, State)——启动一个叫 Name 的服务器。服务器的初始状态是 State，Fun 是一个完全刻画了服务器的行为特征的函数。
- stop(Name)——停止服务器 Name。
- rpc(Name, Q)——执行名为 Name 的服务器上的一个远程过程调用。

我们可以用这个服务器来实现一个非常简单的“家庭地址登记”（Home Location Register[1]）程序，我们也叫它为VSHLR（Very Simple HLR,非常简单的HLR)，我们的VSHLR有如下的接口：

- start()——启动 HLR。
- stop()——停止 HLR。
- i_am_at(Person, Loc)——告诉 HLR 有一个人 Person 在地址 Loc。
- find(Person)-> {ok, Loc}|error——试图在 HLR 中找到 Person 的地址。如果 Loc 是该 Person 最后一次报告的地址，则 HLR 返回{ok, Loc}；如果 HLR 找不到该 Person 的地址，则返回 error。

[1] 家庭地址登记程序在电信行业被广泛地使用，用来记录移动电话用户的当前地址。

vshlr1 可以通过将 server1 进行参数化来实现，如图 4.4 所示。

```
-module(vshlr1).

-export([start/0, stop/0, handle_event/2,
         i_am_at/2, find/1]).

-import(server1, [start/3, stop/1, rpc/2]).
-import(dict,    [new/0, store/3, find/2]).

start() -> start(vshlr, fun handle_event/2, new()).

stop() -> stop(vshlr).

i_am_at(Who, Where) ->
    rpc(vshlr, {i_am_at, Who, Where}).

find(Who) ->
    rpc(vshlr, {find, Who}).

handle_event({i_am_at, Who, Where}, Dict) ->
    {ok, store(Who, Where, Dict)};
handle_event({find, Who}, Dict) ->
    {find(Who, Dict), Dict}.
```

图 4.4: 一个非常简单的家庭地址登记程序

下面是对该服务器的一个简单的使用情景：

```
1> vshlr1:start().
true
2> vshlr1:find("joe").
error
3> vshlr1:i_am_at("joe", "sics").
ack
4> vshlr1:find("joe").
{ok,"sics"}
```

虽然我们的 VSHLR 程序是相当简单的，但是它却阐明了对许多设计问题的一些简单解决办法。读者应当注意到：

- 把功能*完全地*划分到了两个不同的模块。负责创建进程、接收和发送消息等等工作的所有代码*都*包含在 server1.erl 模块中，负责*实现* VSHLR 的具体工作的所有的代码都包含在 vshlr1.erl 模块中。
- vshlr1.erl 模块中的代码并没有用到 Erlang 的并发处理的任何原语。编写这部分代码的程序员不需要知道并发和错误处理的任何事情。

第 2 点特别重要。这就是分解出并发（factoring out concurrency）的一个例子。因为编写并发程序一般都认为是比较困难的，大多数程序员更擅长于编写顺序化代码，所以能够分解出并发是一个明显的优势。

我们不仅可以分解出并发，还能够屏蔽用来参数化服务器的函数代码中可能存在的错误。这一点在下一节中可以看到。

4.1.1 一个可容错的客户-服务器模型

我现在来扩展我们的服务器程序，增加错误恢复的代码，如图 4.5 所示。一旦函数 F/2 发生错误，原先的服务器程序就会崩溃掉。“容错”一词通常是说硬件的，但是在这里我们的意思是包容用以参数化服务器的函数 F/2 中的错误。

函数 F/2 在一个 catch 语句内进行求值，如果一个 RPC 请求会导致服务器崩溃，就将发起该 RPC 的客户杀死掉。

比较一下新的服务器代码，我们发现跟老的代码相比有两点小小的变化：rpc 代码被改成了：

```
rpc(Name, Query) ->
    Name ! {self(), Query},
    receive
        {Name, crash} -> exit(rpc);
        {Name, ok, Reply} -> Reply
    end.
```

```
-module(server2).
-export([start/3, stop/1, rpc/2]).

start(Name, F, State) ->
    register(Name,  spawn(fun() -> loop(Name,F,State) end)).

stop(Name) -> Name ! stop.

rpc(Name, Query) ->
    Name ! {self(), Query},
    receive
        {Name, crash} -> exit(rpc);
        {Name, ok, Reply} -> Reply
    end.

loop(Name, F, State) ->
    receive
        stop -> void;
        {From, Query} ->
            case (catch F(Query, State)) of
                {'EXIT', Why} ->
                    log_error(Name, Query, Why),
                    From ! {Name, crash},
                    loop(Name, F, State);
                {Reply, State1} ->
                    From ! {Name, ok, Reply},
                    loop(Name, F, State1)
            end
    end.

log_error(Name, Query, Why) ->
    io:format("Server ~p query ~p caused exception ~p~n",
              [Name, Query, Why]).
```

图 4.5: 具有错误恢复功能的一个简单服务器程序

并且 loop/3 内 receive 语句的一段改成了：

```
case (catch F(Query, State)) of
    {'EXIT', Why} ->
        log_error(Name, Query, Why),
        From ! {Name, crash},
        loop(Name, F, State);
    {Reply, State1} ->
        From ! {Name, ok, Reply},
```

```
            loop(Name, F, State1)
    end
```

让我们再仔细看看这些变化的细节，我们会发现，如果在服务器的 loop 函数里对 F/2 的求值发生了异常，则会发生三件事：

1. 会报告该异常——在我们的程序中，我们只是将该异常打印了出来，但要是在更成熟的系统中，我们会将该异常记录到一个稳定存储器中。
2. 向客户发送一个 crash 消息——当客户收到该 crash 消息时，会在客户代码中产生一个异常。因为这时候客户程序再运行下去很可能已经没有意义了，所以这正是期望的结果。
3. 服务器继续对*老的*状态变量进行操作。因此说 RPC 遵循了“事务语义”（transaction semantics），也就是说，它要么操作完全成功，服务器的状态被更新，要么操作失败，服务器的状态保持原样不动。

注意 server2.erl 只能保护发生在将服务器参数化的特征函数中的错误。如果服务器本身死掉了（这是可能的，例如被系统中的其他进程故意杀死），那么客户的 RPC 桩就被无限挂起了，一直等待着一个永远不会到来的回应消息。如果我们还想保护这种可能性，那么我们可以像这样来写 RPC 函数：

```
rpc(Name, Query) ->
    Name ! {self(), Query},
    receive
        {Name, crash} -> exit(rpc);
        {Name, ok, Reply} -> Reply
    after 10000 ->
        exit(timeout)
    end.
```

这种解决方法解决了一个问题，但是却带来了另一个问题：我们应该把超时设置多长时间？一个更好的解决办法是用到监督树，这个我在这里不展开讲。服

务器发生了故障，不应该由客户软件来检测它，而应该由专门负责纠正服务器故障的特殊的监督者进程来检测它。

```
-module(vshlr2).

-export([start/0, stop/0, i_am_at/2, find/1]).

-import(server2, [start/3, stop/1, rpc/2]).
-import(dict,    [new/0, store/3, find/2]).

start() -> start(vshlr, fun handle_event/2, new()).

stop() -> stop(vshlr).

i_am_at(Who, Where) ->
    rpc(vshlr, {i_am_at, Who, Where}).

find(Who) ->
    rpc(vshlr, {find, Who}).

handle_event({i_am_at, Who, Where}, Dict) ->
    {ok, store(Who, Where, Dict)};
handle_event({find, "robert"}, Dict) ->
    1/0;
handle_event({find, Who}, Dict) ->
    {find(Who, Dict), Dict}.
```

图 4.6: 故意产生错误的家庭地址登记程序

现在，我们可以来运行这个用含有故意错误的 VSHLR 版本（vshlr2）作为参数的服务器程序，如图 4.6 所示。

该程序的一个执行片断如下所示：

```
> vshlr2:start().
true
2> vshlr2:find("joe").
error
3> vshlr2:i_am_at("joe", "sics").
ok
```

```
4> vshlr2:find("joe").
{ok,"sics"}
5> vshlr2:find("robert").
Server vshlr query {find,"robert"}
caused exception {badarith,[{vshlr2,handle_event,2}]}
** exited: rpc **
6> vshlr2:find("joe").
{ok,"sics"}
```

异常中的信息足以用来帮助我们调试程序（满足第 2.10 节中的 R3 的要求）。

我们最后对图 4.5 中的服务器程序所做的改进就是让我们可以 “在系统运行中”（on-the-fly）对服务器的程序进行修改。改进后的程序如图 4.7 所示。

我可以对图 4.7 的程序用 vshlr3 进行参数化，vshlr3 没有在这里贴出来，它跟 vshlr2 基本上是一样的，只有一点不同：第 3 行的 server2 改成 server3。

下面的执行片断展示了如何在“在系统运行中”修改服务器程序的代码。第 1－3 行显示服务器程序工作正常，server3 可以处理 1 除以 0 这种错误而不会崩溃，例如第 5 行显示运行正常。第 6 行，我们发送一条命令，将服务器程序的代码改回到 vshlr1 中的版本。这条命令执行完后，服务器程序会如第 7 行所示地正常工作。

```
1> vshlr3:start().
true
2> vshlr3:i_am_at("joe", "sics").
ok
3> vshlr3:i_am_at("robert", "FMV").
ok
4> vshlr3:find("robert").
Server vshlr query {find,"robert"}
caused exception {badarith,[{vshlr3,handle_event,2}]}
** exited: rpc **
```

```
5> vshlr3:find("joe").
{ok,"sics"}
6> server3:swap_code(vshlr,
fun(I,J) -> vshlr1:handle_event(I, J) end).
ok
7> vshlr3:find("robert").
{ok,"FMV"}
```

```
-module(server3).
-export([start/3, stop/1, rpc/2, swap_code/2]).

start(Name, F, State) ->
    register(Name, spawn(fun() -> loop(Name,F,State) end)).

stop(Name) -> Name ! stop.

swap_code(Name, F) -> rpc(Name, {swap_code, F}).

rpc(Name, Query) ->
    Name ! {self(), Query},
    receive
        {Name, crash} -> exit(rpc);
        {Name, ok, Reply} -> Reply
    end.

loop(Name, F, State) ->
    receive
        stop -> void;
        {From, {swap_code, F1}} ->
            From ! {Name, ok, ack},
            loop(Name, F1, State);
        {From, Query} ->
            case (catch F(Query, State)) of
                {'EXIT', Why} ->
                    log_error(Name, Query, Why),
                    From ! {Name, crash},
                    loop(Name, F, State);
                {Reply, State1} ->
                    From ! {Name, ok, Reply},
                    loop(Name, F, State1)
            end
    end.

log_error(Name, Query, Why) ->
    io:format("Server ~p query ~p caused exception ~p~n",
              [Name, Query, Why]).
```

图 4.7 一个具有错误恢复和动态代码替换功能的简单服务器程序

编写 vshlr3 的程序员完全不必知道 server3 的任何实现细节，也不必知道服务器代码可以在服务不停止的情况下被动态修改。

在不停止服务器的情况下替换服务器程序代码的能力部分满足了第 2.2 节中的“需求 8”——即不停止系统而升级系统的软件。

如果我们回顾一下图 4.5（服务器程序）中的 server2 的代码和图 4.6（应用程序）中的 vshlr2 的代码，我们会发现：

1. 服务器程序中的代码可以重复用来构建许多不同的客户-服务器模型的应用程序。
2. 应用程序的代码比服务器程序的代码要简单很多。
3. 要理解服务器程序的代码，程序员就必须要理解 Erlang 并发模型的所有细节。这就涉及到名字注册、进程产生、向进程发送不可捕获 exit 异常、发送和接收消息。对上报异常来说，程序员还必须理解异常的概念，对 Erlang 的异常处理机制相当熟悉。
4. 要编写应用程序的代码，程序员就只需要理解一份简单的顺序化程序——他们不需要了解关于并发和错误处理的任何事情。
5. 我们可以想象，同一份应用程序的代码可以与越来越成熟的一系列服务器程序配合运行。我已经展示过三个版本的服务器程序，并且我们还可以往服务器程序中添加越来越多的功能，而保持服务器程序/应用程序（server/application）的接口不变。
6. 不同的服务器程序（server1、server2 等等）渗透给应用程序的是不同的非功能特性（non-functional characteristics）。而所有服务器程序的功能特性（functional characteristics）都是一样的（即，输入正确的参数程序最终产生的都是同一个结果）；但是非功能特性却不一样。
7. 实现系统的非功能性需求（我们所说的非功能性需求是指在系统出现故障的时候系统的行为，函数求值需要多长时间等等）的部分代码被限制在服务器程序之内，对编写应用程序的程序员来说是不可见的。

8. 远程过程调用（remote procedure call）如何实现的细节被隐藏在服务器程序模块内。这就意味着对于要使今后服务器程序的修改而不会影响到客户程序，这一点是必须的。例如，我们可以修改图 4.5 中 rpc/2 的实现细节，而不必修改调用 server2 中的函数的客户程序。

将整个服务器的功能划分成一个非功能性部件（none-functional part）和一个功能性部件（functional part）是一种好的编程实践，可以给系统带来许多可观的好处，就如：

1. 并发编程通常认为是比较难的。在一个大型的编程团队中，程序员的技能层次往往不同，那么专家程序员应该编写通用服务器部分代码，而经验尚浅的程序员应该去编写应用部分代码。

2. 形式化方法（formal method）可以应用于（简单一些的）应用部分代码之上。在对 Erlang 代码进行形式化证明（formal verification）的时候，或设计类型系统进行类型推断的时候，一遇到并发编程往往就会有问题。如果假设通用服务器程序已经正确无误这一假设成立，那么证明系统的性质的问题就简化为证明顺序化程序的性质的问题。

3. 在一个充满大量客户-服务器的系统中，所有的服务器程序就可以利用同一份通用服务器程序来编写。这就使得程序员理解和维护起许多服务器程序来更简单。在 8.3.1 小节我们在对一个有许多服务器程序的大型系统进行分析时会印证这个说法。

4. 通用服务器程序和应用部分程序可以分别独立进行测试。如果长时期内接口保持恒定不变，那么这两者可以独立进行改进。

5. 应用部分代码能够“插入到”许多不同的通用服务器程序中，不同的通用服务器具有不同的非功能特性。在具有相同的接口的情况下，有的服务器可以提供加强的调试环境，而有的服务器可以提供集群化、热切换等特性。这一点已经在很多项目中实行过，例如 Eddie[31]服务器程序提供了集群化能力，Blue-tail 邮件加固器[11]提供了一个具有热切换功能的服务器。

4.2 抱持 Erlang 的世界观

Erlang 的世界观就是*一切皆进程*，进程之间只能通过交换消息来进行交互。

当我们的 Erlang 程序需要跟外界软件交互时，一般都是写一个接口程序来完成交互，这个接口程序体现出“一切皆进程”的精神，而且很便利。

举个例子：我们考虑一下如何实现一个 web 服务器？web 服务器是通过 RFC2616[36]建议中定义的 HTTP 协议与客户通信的。

从一个 Erlang 程序员的角度来看，web 服务器内部的循环会给每一个连接产生一个进程，接受来自客户的请求，并作出适当的响应。程序代码可能如下：

```
serve(Client) ->
    receive
        {Client, Request} ->
            Response = generate_response(Request)
            Client ! {self(), Response}
    end.
```

这里 Request 和 Response 是 Erlang 的项式（term），表示 HTTP 协议的请求和 HTTP 协议的响应。

上面的服务器程序非常简单，它期望来一个单独的请求，作出一个单独的响应，然后就终止了连接。

一个更成熟的服务器程序，还要支持 HTTP/1.1 规定的持久连接，支持这种持久连接的代码也是非常简单的：

```
serve(Client) ->
    receive
        {Client, close} ->
            true;
        {Client, Request} ->
            Response = generate_response(Request)
```

```
            Client ! {self(), Response},
            server(Client);
        after 10000 ->
            Client ! {self(), close}
    end.
```

这个 11 行的函数就从本质上完成了一个简单的支持持续连接的 web 服务器的功能。

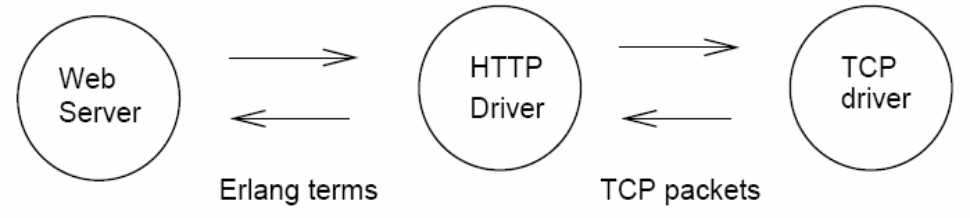


图 4.8: 一个 Web 服务器

web 服务器并不直接跟产生 HTTP 请求的客户交互，因为那样的话一些无关的细节将会严重地干扰 web-server 的实现，并且使得程序结构难以理解。

这里我们用了一个“中间人”进程（见图 4.8）。“中间人”进程（一个 HTTP 驱动器）完成 HTTP 请求、应答和表示这些请求、应答的对应 Erlang 项式之间的互换。

HTTP 驱动器程序的全部代码如下：

```
relay(Socket, Server, State) ->
    receive
        {tcp, Socket, Data} ->
            case parse_request(State, Data) of
                {completed, Request, State1} ->
                    Server ! {self(), {request, Request }},
                    relay(Socket, Server, State1);
                {more, State1} ->
                    relay(Socket, Server, State1)
```

```
            end;
        {tcp_closed, Socket} ->
            Server ! {self(), close};
        {Server, close} ->
            gen_tcp:close(Socket);
        {Server, Response} ->
            Data = format_response(Response),
            gen_tcp:send(Socket, Data),
            relay(Socket, Server, State);
        {'EXIT', Server, _} ->
            gen_tcp:close(Socket)
    end.
```

如果通过一个 TCP socket 从客户收到一个包，这个包就通过调用 parse_request/2 进行解析。当响应已经完成，一个表示该请求的 Erlang 项式就被发送给服务器。如果收到一个服务器的响应，则该响应被转换格式（reformat）并被发送给客户。如果有任何一端终止连接，或者服务器发生一个错误，这个连接就会被关掉。如果该进程因任何原因终止，则所有的连接也会被自动关掉。

变量 State 是一个状态变量，用来表示可重入解析器的状态，该解析器解析收到的 HTTP 请求的。

web-server 的完整代码并没有在这里展示出来，但是可以通过参考连接[15]下载。

4.3 错误处理哲学

Erlang 的错误处理与其他大多数编程语言中的错误处理有着根本的不同。Erlang 关于错误处理的哲学可以用如下几条标语来表达：

- 让其它进程来修复错误。
- 工作者不成功，便成仁。（if you can't do what you want to do, die.）
- 任它崩溃。

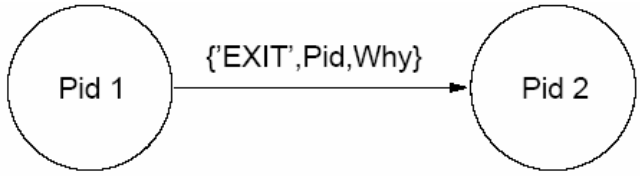
- 杜绝防御式编程。

4.3.1 让其它进程来修复错误

在分布式系统中，我们如何来处理错误呢？为了处理硬件错误，我们需要备份；而为了处理整台计算机的错误，我们需要两台计算机。

如果计算机 1 发生故障，那么计算机 2 会发现故障并改正错误

如果第 1 台计算机崩溃了，第 2 台计算机会检测到该故障，并试图修复该故障引起的错误。在 Erlang 中，我们就是使用的这种办法，只不过我们把计算机换成了进程。



如果进程 1 发生故障，那么进程 2 会发现故障并改正错误

如果 Pid1 出错且 Pid1 和 Pid2 是连接在一起的，且 Pid2 被设置为捕获（trap）错误，那么当 Pid1 出错时，一条{'EXIT', Pid1, Why}格式的消息就被发送给 Pid2。Why 描述了出错的原因。

注意，如果运行 Pid1 的计算机死掉了，也会有一条退出消息{'EXIT', Pid1, machine_died}发送给 Pid2。该消息*貌似*是来自 Pid1，但是实际上来自运行着 Pid2 的节点的实时系统。

非要使一个硬件错误看起来像一个软件错误的原因是，我们不想用两种方法来处理错误，一种处理软件错误而另一种处理硬件错误。为了概念上的完整性，我们期望用统一的机制。再综合考虑硬件错误的极端情形——即整个处理器发生故障，就产生了我们的错误处理思想：即不在出错的地方，而在系统的其他地方

来进行处理。

因此在任何情况下，包括硬件发生故障，都是由 Pid2 来纠正错误。这就是为什么我说“*让其他进程修复错误*”。

这种哲学与顺序化编程语言是完全不同的，在顺序化编程语言中，除了试图在发生错误的控制线程中处理所有的错误，没有其他选择。在提供有异常处理的顺序化编程语言中，程序员将任何可能发生故障的代码用一个异常处理结构包含起来，试图包住该结构中所有可能发生的错误。

远程错误处理有许多好处：

1. 错误处理代码和出错代码运行在不同的控制线程中。
2. 解决问题的代码不会被处理异常的代码扰乱。
3. 该方法可以用于分布式系统，所以一个单节点系统的代码移植到一个分布式系统中只需对错误处理代码做很少的修改。
4. 系统可以在单一节点系统上构建与测试，然后无需进行大的修改就可以部署到多节点系统上。

4.3.2 工作者与监督者

为了将执行正常工作的进程与处理错误的进程更清楚地区别开来，我们经常会谈到*工作者*（worker）和*监督者*（supervisor）。

一个进程，即工作者进程，负责执行正常的工作。另一个进程，即监督者进程，来检测工作者。如果工作者中发生了一个错误，监督者会采取措施来纠正该错误。这种方式的妙处在于：

1. 职责划分很清晰。负责做事的进程（工作者）不用担心错误处理。
2. 我们可以用特别的进程专门来负责错误处理。
3. 我们可以在物理上独立的计算机上运行工作者进程和监督者进程。
4. 往往会发现错误纠正代码是*有通用性的*（generic），即对许多应用程序都

普遍适用，而工作者进程则更多因应用而异。

第三点是至关重要的——使得 Erlang 满足了 R3 和 R4（见第 2.10 节），从而可以把工作者和监督者运行在物理上独立的计算机上，因此可以构建可以包容引起所有进程出错的硬件错误的系统。

4.4 任它崩溃

我们的错误处理哲学如何适用于我们的编程实际呢？当程序员发现一个错误的时候，他该编写什么代码？我们的哲学是*让其它进程来修复错误*，但是这对编码者来说，意味着什么？答案是*任它崩溃*。我的意思是，当发生一个错误的时候，就让程序崩溃好了。什么算是错误？就编程而言，我所说的错误即：

- 那些运行时系统也不知道该如何处理的*异常*。
- 那些程序员也不知道如何处理的*错误*。

如果一个异常是由运行时系统产生的，但是程序员之前就预见到了该异常，并知道如何纠正引起异常的条件，那么这就不是一个错误。例如，打开一个不存在的文件会产生一个异常，但是程序员可以不把它当作错误。程序员可以写代码报告这个异常，并进行必要的纠正。

有些错误发生时连程序员也不知道该如何处理。程序员应该遵照规格说明书来编程，但是往往规格说明书也没有说该怎么办，所以程序员也就不知道该怎么办。这里有一个例子：

假设我们现在写一个程序来为一个微处理器生成操作码，规格说明书说：一个 load 操作应该返回操作码 1，一个 store 操作应该返回操作码 2。程序员就把该规格说明写成如下代码了：

```
asm(load) -> 1;
asm(store) -> 2.
```

现在假设系统试图求值 asm(jump)——该怎么处理呢？假设你是该程序员，并且你已经习惯于编写防御式（defensive）代码，那么你可能会写：

```
asm(load) -> 1;
```

```
asm(store) -> 2;
asm(X) -> ??????
```

但是??????该是什么呢？你会在该处写什么样的代码？你现在碰到的情形就如同运行时系统遇到被 0 除一样的情形，你写不出有意义的代码，你所能做的只有终止程序，于是你写道：

```
asm(load) -> 1;
asm(store) -> 2;
asm(X) -> exit({oops,i,did,it,again,in,asm,X}).
```

干嘛这么费劲？在 Erlang 编译器编译

```
asm(load) -> 1;
asm(store) -> 2.
```

的时候，就如同你已经写了：

```
asm(load) -> 1;
asm(store) -> 2;
asm(X) -> exit({bad_arg,asm,X}).
```

防御式编码会破坏了代码的纯净性，使代码的阅读者容易产生混淆。而且防御式编码的诊断信息也未必比编译器自动提供的诊断信息好。

4.5 显意（intentional）编程

“显意编程”（intentional programming）是我给一种编程风格所起的名字，这种编程风格使得程序的阅读者能够很轻易地看到程序员编写一段代码的意图。代码的意图应该从它所调用的函数的名字上显而易见，而不应该需要通过对代码的结构分析来推断。下面的例子很好地说明了这一点：

在早期的 Erlang 的库模块 dict 中，导出了一个 lookup/2 的函数，接口如下：

```
lookup(Key, Dict) -> {ok, Value} | notfound
```

在这种定义下 lookup 被用在了三种不同的上下文中：

1．用于*数据获取*（data retrieval）——程序员可能会写：

```
lookup(Key, Dict) -> {ok, Value} | notfound
```

这里 lookup 用一个已知的键（key）从字典（dictionary）中提取一个条目。Key 应该在字典中，否则就是一个编程错误，所以如果键没有找到将会产生一个异常。

2. 用于*搜索*（searching）——如下代码段：

```
case lookup(Key, Dict) of
    {ok, Val} ->
        ... do something with Val ...
    not_found ->
        ... do something else ...
end.
```

是搜索一个字典，我们并不知道 Key 是否存在——如果键不在字典中，将不会是一个编程错误。

3. 用于*测试一个键的存在性*——代码段：

```
case lookup(Key, Dict) of
    {ok, _} ->
        ... do something ...
    not_found ->
        ... do something else ...
end.
```

是测试一个指定的键 Key 是否在字典中。

在读过数千行这种代码后，我们开始担心代码的意图了——我们问我们自己一个问题“程序员编写这一行代码的意图到底是什么？”——在分析了上述三种用法后，我们的答案即*数据获取*，*搜索*和*测试*。

在很多不同的上下文中，我们都需要在一个字典里查找键。在某种情况下，程序员知道一个指定的键应该存在于字典中，如果该键不在该字典中，则应该是一个编程错误，程序应该终止。另一种情况下，程序员不知道该键对应的条目是

否在字典中，他们的程序应该能处理键在字典中和不在字典中两种情况。

抛弃对程序员的意图的猜测，分析一下代码，一组更好的库函数是：

```
dict:fetch(Key, Dict) = Val | ‘EXIT’
dict:search(Key, Dict) = {found, Val} | not_found.
dict:is_key(Key, Dict) = Boolean
```

这就简洁地表达了程序员的意图——不需要对程序进行分析和猜测，我们清晰地看到了程序的意图。

显然大家都知道 fetch 可以用 search 来实现，search 也可以用 fetch 来实现。但是如果 fetch 是原子性的，则我们也可以写：

```
search(Key, Dict) ->
    case (catch fetch(Key, Dict)) of
        {’EXIT’, _} ->
            not_found;
        Value ->
            {found, Value}
    end.
```

不过这并不是什么好代码，因为我们先是产生了一个异常（该异常本应说明程序已错），后来却修改了错误。

更好的用法应当如：

```
find(Key, Dict) ->
    case search(Key, Dict) of
        {ok, Value} ->
            Value;
        not_found ->
            exit({find, Key})
    end.
```

这样正好产生一个异常代表发生了一个错误。

4.6 讨论

编程是一项严格的活动。编写结构清晰、意图明显的代码是困难的。困难一部分源于要选择正确的抽象。为了对付复杂的情况，我们使用了“分而治之”（divide and conquer）的方法，我们把复杂的问题分解成简单一些的子问题，然后解决这些子问题。

本章阐述了如何把许多复杂的问题分解成更简单的子问题。在谈到错误处理的时候，我阐释了如何“抽象出”错误，并表明了程序应该将“纯净”的代码与“修复错误”的代码划分开的观点。

在编写一个服务器程序的时候，我展示了如何抽象出服务器程序的两个非功能特性。我展示了如何编写一个当特征函数（特征函数定义了服务器的行为）中发生一个错误时不会导致服务器崩溃的服务器程序，我还展示了在不停下服务器的情况下如何修改服务器的行为。

错误恢复、运行时修改系统的代码是许多真实系统需要的两项典型的非功能特性。通常的编程语言和系统对编写已经定义好的功能行为的代码提供了强力的支持，但是对程序的非功能性部分的支持却很贫乏。

在大多数的编程语言中，编写纯的函数（其值确定地依赖于函数的输入）是容易的[2]，但是要做到修改运行时系统的代码，或以一种通用的方式处理错误，或保护我们的代码不受系统部分发生的故障的影响这一类事情，却要困难得多，有时甚至是不可能的。因此，程序员运用了操作系统提供的服务——操作系统通常以进程的面貌提供了保护区域、并发机制等等。

从某种意义上讲，操作系统提供了“被编程语言设计者遗忘了的东西”。但是在 Erlang 这样的编程语言中，操作系统是几乎不需要的。OS 真正提供给 Erlang 的只是一些设备驱动程序，而 OS 提供的诸如进程、消息传递、调度、内存管理等等机制都不需要。

用 OS 的机制来弥补编程语言的不足所带来的问题是，操作系统的低层机制不能够轻易地被改变。例如操作系统中关于什么是进程的概念以及进程间调度的

[2] 这也应当是容易的。

策略都不能修改。

通过给程序员提供轻量级的进程和关于错误检测和处理的基本机制，应用程序的编写者就很容易地设计和实现他们自己的应用操作系统，这种应用操作系统是专为他们的特定的问题的特征而特别设计的。OTP 系统——用 Erlang 编写的一个应用程序——便是此中一例。

5 编写可容错系统

电话交换机的设计者们在软件设计中花了一半的精力在错误的检测和纠正上[48]。

Richard Kuhn，国家标准与技术协会

什么是可容错系统？如何编写可容错系统？这个问题是本论文的重点所在，也是我们理解如何构建可容错系统的关键。在本章中，我们定义了我们所说的“容错”的含义，并提出了用来编写可容错系统的一种特殊方法。我们以两条引述来开始本章：

> *如果一个系统的程序在出现逻辑错误的时候仍然能够正确地执行，我们就说该系统是可容错的。*——[16]
>
> ……
>
> *要想设计并构造一个可容错系统，你必须要明白系统在什么情况下应该正常工作，在什么情况下该失效，可能会发生什么类型的错误。错误检测是容错系统的一个基本部件。也就是说，如果你知道发生了一个错误，你可能用替换掉出错部件的方法、采用另一种计算方式的方法或上报一个异常的方法来达到包容该错误的目的。然而，你希望避免为了达到可容错性而给系统增添不必要的复杂性，因为这些复杂性可能会导致系统可靠性的降低。*——Voas 对 Dugan 的引用[67]。

我在这里的表述延续了 Dugan 的建议，我将说明当检测到一个反常情况发生时会发生什么事情，以及来建造一个软件机制来检测和纠正错误。

本章的余下部分讲述了：

- *一种可容错编程的策略*——该策略简而言之就是当你不能纠正一个错误的时候，马上放弃，只去做你可以做到的简单一些的事情。
- *监督层级*（supervision hierarchies）——就是对任务的层次化组织。

- *乖函数*（well-behaved function）——就是那些应该正确地工作的函数。乖函数产生的异常我们把它解释成故障。

5.1 可容错编程

为了使得系统可容错，我们把软件组织成一系列层次化的待执行的任务（task）。最高层的任务按照某个规格说明来执行着应用逻辑。如果这个任务不能够执行，系统将会试图执行某个更简单的任务。如果这个更简单的任务仍然无法执行，则系统将会尝试执行一个更更简单的任务，如此类推。如果系统中最底层的任务都无法执行，那么就当系统发生了故障。

这个方法直观上感觉是很有吸引力的。它的意思是，*如果我们不能做到我们想做的，那就做一些更容易做到的*。我们还试图组织一下我们的软件，使得更简单的任务由更简单的软件来执行，这样的话当任务变得更简单时，成功的可能性就越高。

当任务变得更简单时，操作的侧重点也发生了变化——相比提供完全的服务，我们变得更关注于保护系统免受摧毁。虽然随着任务层次的降低我们变得更保守，但是在所有的层次，我们的目标都是要提供一个可接受级的服务。

当故障发生的时候，我们更关注于保护系统，并且报告故障的确切原因——以便我们接下来可以对故障做点什么。这就意味着我们需要某种可以不受系统崩溃影响的持久化错误日志。在异常环境下，我们的系统会发生故障，但是当发生故障的时候我们绝不应当丢失有关系统为什么会发生故障的信息。

为了实现我们的任务层级，我们需要对“故障”（failure）这个词有一个准确的认识。在 Erlang 中，对一个函数进行求值可能会导致异常（exception）。但是异常不等于错误（error），而且不是所有的错误都将造成故障（failure）。所以我们需要讨论一下异常、错误和故障之间的区别。

异常、错误和故障之间最大的区别在于是在系统的哪个部分检测到的非正常事件，该事件被如何处理，被如何解释。我们来跟踪一下当我们的系统中发生了一次异常情况时会发生什么事情——这里的描述是“自底向上”的，即从最初检测到错误发生的点开始。

- 在系统的最底层，Erlang 虚拟机检测到了一个内部错误——它检测到了一个被 0 除的情况，或者一个模式匹配错误或其他的情况。重要的是在检测到这些情况时，进程对发生错误的地方后续的求值已经变得没有意义了。所以虚拟机仿真器无法继续，它做了唯一能做的事情，就是抛出一个异常。

- 在它的相邻层，该异常可能会也可能不会被捕获。捕获异常的程序段可能能够也可能不能够纠正异常所引起的错误。如果错误能够成功被纠正，那么就不会造成什么伤害，进程会恢复到正常。如果该错误被捕获了，但是纠正不了，那么可能会产生另一个异常，产生该异常的进程可以捕获也可以不捕获该异常。

- 如果一个异常产生了但是没有“捕捉处理者”（catch handler）[1]，那么该进程将会发生故障。故障的原因将会被传播给当前与之相连的所有进程。

- 收到这种故障信号的所有进程像对待正常的进程间消息一样，可能会也可能不会截取并处理这些信号。

现在我们看到了当虚拟机仿真器中发生的一个非正常情况，是如何在系统中从下往上传播的。在错误向上传播的过程中，在每个点上都将尝试着去纠正它。这种尝试可能成功或失败，所以我们就可以自如地决定在哪里、如何处理该错误。

一个“被纠正了的”错误不会再被看作是一个故障，但是这要求该错误情形要能够事先被预见到，并且针对该错误的纠正代码要成功地执行。

至此，我么已经看到了一个非正常情形是如何产生的，如何导致异常，该异常是如何被捕获的，未被被捕获的异常如何导致进程故障，进程故障如何被系统的其他进程检测到。这些正是我们赖以实现我们的“任务层级”的一些可用的机制。

[1] 也就是说，当前函数并没有在catch语句的作用范围内求值。

5.2 监督层级

回想一下，我们在本章的开始说到过“任务层级”的思想，其基本思想是：

1．尽力执行一个任务。

2．如果你不能够执行一个任务，则去执行一个简单一些的任务。

我们将每个任务关联上一个*监督者进程*（supervisor process）——监督者将会被赋予一个*工作者*(worker)来试图达到该任务规定的目标。如果该工作者进程失败并发出一个非正常的退出信号，则监督者就会假定该任务已经失败，并发起某种错误恢复程序。错误恢复程序可能会重启工作者，或者如果重启失败则转而去做一些更简单的事情。

监督者和工作者被按照如下规则安排成层次化的树型关系：

1．*监督树*是*监督者*组成的树。

2．*监督者*监视*工作者*和*监督者*。

3．*工作者*是 *behaviour* 的实例。

4．*behaviour* 用*乖函数*（well-behaved function）来参数化。

5．*乖函数*在发生错误时会产生异常。

在这里：

- *监督树*是监督者形成的层次化树。树中的每一个节点负责监视它的子节点中发生的错误。
- *监督者*是系统中监视其他进程的进程。被监督的对象是监督者或工作者。监督者必须能够检测到被监视对象所产生的异常，能够启动、停止或重启被监督对象。
- *工作者*是执行任务的进程。

如果一个工作者进程以一个非正常退出信号（参见 3.5.6 节）而终结，那么监督者就会认为已经发生了一个错误，就会采取措施来修复该错误。

在我们的模型中，工作者并不是任意的进程，而是为数不多的通用进程的实例（称之为 *behaviour*）。

- *behaviour* 是其操作被一些回调函数完全特征化的通用进程。这些回调函数一定要是*乖函数*。

一个关于 behaviour 的例子是 gen_server，该 behaviour 用于编写分布式的、可容错的客户-服务器程序。behaviour 需要用一些 WBF 来进行参数化。

所有程序员都应该理解如何编写 WBF，才能编写出可容错的分布式客户-服务器程序。gen_server 这种 behaviour 为并发和分布式特性提供了一个可容错的框架。程序员只需要关心的是编写 WBF 来参数化该 behaviour。

为了简便起见，我们这里考虑两种监督层次结构，分别为*线性层次体系*（linear hierarchies）和 *AND/OR 层次树*（AND/OR hierarchy trees）。在接下来的章节里，我将对它们进行图形化描述。

5.2.1 图形表示法

监督者和工作者可以用图 5.1 所示的符号来简便地表示。

监督者记作方角矩形。在矩形的右上角用一个符号 T 来标明监督者的类型。T 的值要么为“O”，代表“或（or）”型监督，要么为“A”，代表“与（and）”型监督。关于监督的类型稍后再详述。

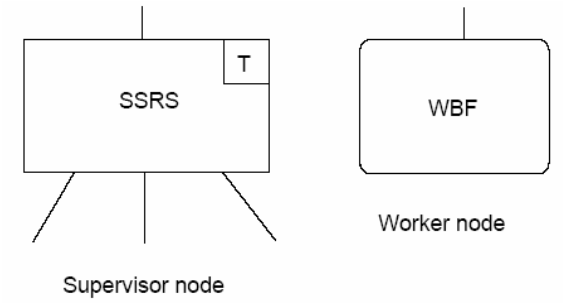


图 5.1：监督者和工作者的记号

监督者能够监督任意个数的工作者或监督者。对每一个被监督的实体，监督

者都要知道如何来启动、停止和重启该实体。这种信息被保存在 SSRS 中，SSRS 即“Start Stop and Restart Specification”（启动停止重启说明）。在 6.5.2 小节中将会有一个 SSRS 的简单的例子，来说明三个不同的监督者是如何被监督的。

每个监督者（监督层次体系中的顶层监督者除外）都有且仅有一个监督者直接在它的上方，我们称直接上层监督者为直接下层的监督者的*父亲*（parent）。相反地，在监督层次体系中某个监督者直接下方的监督者为该监督者的*孩子*（children）。图 5.1 中的监督者节点有一个父亲和三个孩子。

工作者被记作圆角矩形（见图 5.1）。工作者由乖函数（即图中的 WBF）来参数化。

5.2.2 线性监督

我先说线性层次结构。图 5.2 显示了一个由三个监督者组成的线性层次结构。每个监督者针对其每一个孩子都有一个 SSRS，遵守下面的规则：

- 如果一个监督者被其父亲停止，那么该监督者将停止其所有的孩子。
- 如果一个监督者的任何一个孩子崩溃，那么该监督者将重启该孩子。

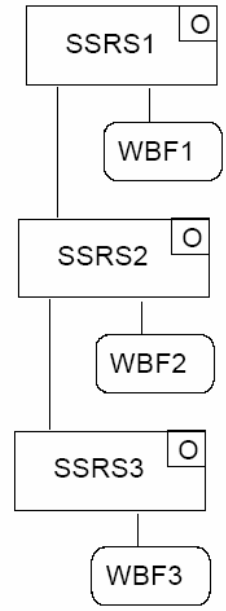


图 5.2 一个线性监督层次体系

系统通过最顶层的监督者启动而启动。最顶层的监督者第一次启动时，需要

用到 SSRS1。顶层监督者有两个孩子，即一个工作者和一个监督者。顶层监督者启动一个工作者（为一个通过用乖函数 WBF1 进行参数化的 behaviour），同时启动一个子监督者。层次体系中的下层监督者也是按照类似的方式启动起来，整个系统就跑起来了。

5.2.3 与/或监督层级

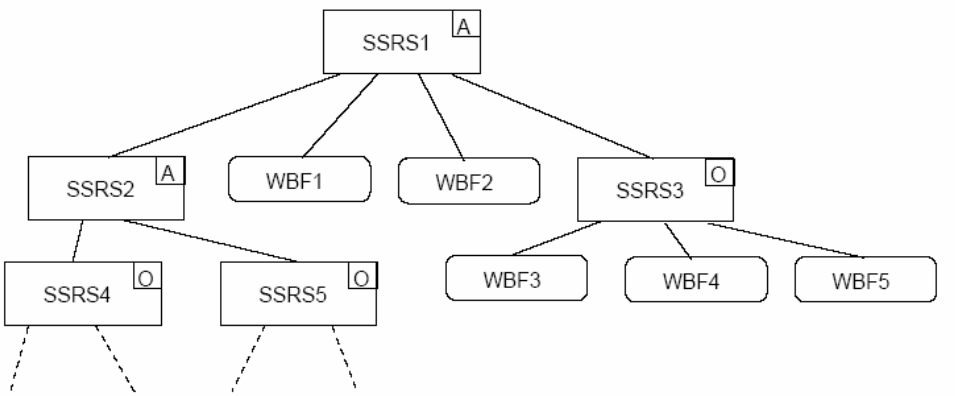


图 5.3 一个与/或监督层次体系

我们可以把我们的简单的监督层次体系扩展成一个含有与节点或或节点的树型结构。图 5.3 就给我们展示了这样一棵树。带记号“A”表示一个“与”监督者，带记号“0”表示一个“或”监督者。在一个与/或树中的监督者应遵循如下规则：

- 如果一个监督者被其父亲停止，那么该监督者将停止其所有的孩子。
- 如果一个监督者的一个孩子崩溃了，而自己是一个“与”监督者，那么该监督者将停止所有的孩子，然后重启所有的孩子。
- 如果一个监督者的一个孩子崩溃了，而自己是一个“或”监督者，那么该监督者将重启该孩子。

“与”型监督用于*依赖性*(dependent)或*关联性*（co-ordinate）的进程。在“与”型树中，系统运行的成功依赖于所有孩子的成功——因此，当有任何一个孩子崩溃时，就应该停止所有孩子并重启它们。

“或”型监督可以用来协调独立进程（independent process）的行为。在

“或”型树中，孩子们的行为被认为是彼此独立的，所以一个孩子不会影响到其它孩子，因此一个孩子出错只需将该孩子进程重启。

落实到具体，我们的“任务层次体系”就是用一个“监督层次体系”来表示的。

在我们的系统中，我们把所有的任务等价于一系列目标，这些目标都具有一个*不变量*（invariant）[2]——如果与目标相关联的不变量为非假，我们就说达到了该目标。在大多数程序中，对不变量的取值的判断通常对应于一个特别指定的函数的求值语句是否产生了异常。

Candea 和 Fox[22]之前已经做过相似的工作，他们曾做过一个基于“可递归重启的（recursively-restartable）Java 组件”的系统。

请注意，我们将错误区分成了两类：*可纠正的*（correctable）错误和*无法纠正的*（uncorrectable）错误。可纠正的错误是指那些在部件中可以被检测到和纠正的错误。无法纠正的错误是指那些能够被检测到，但是没有指定其纠正程序的错误。

上面的讨论都是相当模糊的，因为我们还没有说到底什么算是错误，也没有说在实践中我们如何区分可纠正的错误与无法纠正的错误。

再加上实际情况是绝大多数规格说明书只说明了当系统中的每个部件都根据计划运转时该怎么做，而很少说明当某个特定的错误发生时该怎么做——这就使得情况更加复杂了。的确，如果一个规格说明书严格说明了当一个特定错误发生时该做什么，那或许就有很多人会说这种情况根本就不是错误，而是系统的一个预期特性。这就使得“错误”一词的含意更加模糊。

5.3 什么是错误？

当我们的程序运行的时候，运行时系统根本不知道该把什么当作是错误——它只管按照代码来执行。判断运行出现错误的唯一迹象就是产生的异常。

当运行时系统不能够决定该怎么做时，就会自动产生一个异常。例如，当执

[2] 一个不变量是指一个恒为真值的量。（译注：原文为“An invariant is something that is always true”）

行一个除法操作时，运行时系统就可能会去检测一种“被 0 除”的情况，当出现该情况时，就产生一个异常，因为运行时系统不知道该如何处理。一个会产生异常的所有情况的完整清单可以在 3.5.1 小节查到。

一个异常并不总是对应于错误。例如，如果一个程序员已经编写了代码来正确地应对“被 0 除”这一异常，那么出现这个异常就不必再当成是错误了。

一个异常是否对应于一个错误完全有程序员来决定——在我们的系统中，程序员必须明确地说明系统中那些函数绝对不能产生异常。

Schneider 在他 1990 年的 ACM 指南文件中曾给出了许多关于可容错的定义。在他的文章中，他曾说：

> *一旦一个组件的行为与它的规格说明不再一致，就说该组件发生了故障。*——[61]

为了我们特定的目的，我们将一个错误定义为*观察到的系统行为与期望的系统行为之间的背离*。这里期望的行为是指“规格说明中说明的系统应该具有的行为”。

程序员必须确保一旦系统的行为方式与规格说明发生背离，就能够启动某种错误恢复程序，并且这种情况的记录能够被某种持久的错误日志记录下来，以便日后改正。

在构建真实的系统时，情况会因为我们并没有一个完整的规格说明而变得复杂。在这种情况下，程序员应当对什么应该当成错误，什么不应当作错误有一些通用的概念。在缺少显式的规格说明的情况下，我们需要一个隐式的机制，来符合我们的直觉的想法，即一个错误是“导致程序崩溃的事件”。

在 OTP 系统中，指望程序员来编写乖函数（Well-behaved function, WBF）——乖函数是用来参数化 OTP 的 behaviour 的。这些函数由 OTP 的 behaviour 中的代码来调用。如果对参数化函数的调用产生了一个异常，那么这就*被定义为*一个错误，一条错误诊断就会被添加到错误日志中。

在此有必要回顾一下 4.1.1 小节图 4.5 中通用服务器程序的内部循环部分的

代码，如下：

```
loop(Name, F, State) ->
    receive
        ...
        {From, Query} ->
            case (catch F(Query, State)) of
                {'EXIT', Why} ->
                    log_error(Name, Query, Why),
                    From ! {Name, crash},
                    loop(Name, F, State);
                {Reply, State1} ->
                    From ! {Name, ok, Reply},
                    loop(Name, F, State1)
            end
    end.
```

回调函数 F 是在一个 catch 语句中被调用的。如果产生了一个异常 Why，则该异常被当作是一个错误，一条错误消息就被添加到错误日志中。

这只是一个非常简单的例子，但是已经阐明了 OTP 的 behaviour 中错误处理的基本原理。例如，在 OTP 的 gen_server 这种 behaviour 中，程序员必须编写一个用来参数化服务器的回调模块 M。这个模块 M 除了其他事情之外，还必须导出回调函数 handl_call/2（在 6.2.2 小节的图 6.1 第 23－29 行展示了一个这样的例子）——该回调函数必须要是一个乖函数。

5.3.1 乖函数（Well-behaved functions）

乖函数（WBF）是指正常情况下不应该发生异常的函数。如果一个 WBF 发生了一个异常，那么这个异常将被解释成一个错误。

如果在对一个 WBF 进行求值的时候产生了一个异常，那么该 WBF 应该尽力扭转产生异常的环境。如果 WBF 中产生了一个不能纠正的异常，那么程序员

应该用一个显式的退出（exit）语句来结束该函数。

乖函数的编写应该遵循如下规则：

规则1——程序应该与规格说明同构（isomorphic）。

程序应该忠实地遵循规格说明。规格说明书让做什么，程序就应该做什么，哪怕是愚蠢的事情。程序必须忠实地再生规格说明书中的错误。

规则2——如果规格说明没有说明该做什么，就产生一个异常。

这一点是非常重要的。规格说明通常会说明当发生某种情况时该做什么，而忽略了如果其他情况时该做什么。那么答案就是“产生一个异常”。不幸的是许多程序员都在这时候充分发挥了他们的创造性的猜想力（guess-work），试图猜测设计者当时应该会是怎么样的意图。

如果按照这样来编写系统，那么观测到的异常就会反映出规格说明中的错误。

规则3——如果发生的异常没有包含足够的信息使得可以将该错误隔离，那么就在异常中加一些额外的有用信息。

在程序员编写代码的时候，他们应该自问一下，在一个错误发生时，应该往错误日志中写入些什么信息呢？如果出错信息对调试来说不充分，那么他们就应该往异常中添加足够的信息，以使得程序在下一步能够被调试。

规则4——把非功能性需求变成可在运行时进行检查的断言（assertion）（不变量）。如果断言失败，就产生一个异常。

这种情况的一个例子就是关于循环的终止——一个编程错误可能会致使一个函数进入一个无限循环，从而导致函数不能退出。像这样的错误应该通过请求某个函数在一个规定的时间内终结来检测到。通过时间检测，如果一个函数在规定的时间内没有终止，就产生一个异常，从而结束该函数。

6 构建应用

前面各章分别介绍了编写可容错系统的一个一般模型，介绍了用以监视系统的行为的“监督树”的思想。本章将从一般的理论方面转移到监督者在 OTP 系统中的特定实现。

为了阐明监督原理，我构建了一个简单的 OTP 应用（application）。该应用包含有一个监督者进程，来管理三个工作者进程，这三个工作者进程是 gen_server，gen_event 和 gen_fsm 这三种 behaviour 的实例。

6.1 behaviour 库

使用了 OTP 平台软件的应用都是由许多的“behaviour”构建的。behaviour 是对一些公共编程模式的抽象，在用 Erlang 语言来实现一个系统时可以作为构建块（building blocks）来使用。本章的余下部分将要讨论的 behaviour 如下所列：

- gen_server——这种 behaviour 用来构建在客户-服务器模型中使用的服务器程序。
- gen_event——这种 behaviour 用来构建事件处理器程序。事件处理器程序是指像错误日志记录器一样之类的程序。一个事件处理器是响应一个事件流的程序，它不必对向事件处理器发送事件的进程作出应答。
- gen_fsm——这种 behaviour 用来实现有限状态机。
- supervisor——这种 behaviour 用来实现监督树。
- application——这种 behaviour 用作打包整个应用程序的容器。

对于每种 behaviour，我会介绍其一般原理，还会介绍它的编程 API 的一些特殊细节，并且会给出一个如何创建该 behaviour 的实例的一个完整的例子。

使用 OTP 平台构建的系统遵循如下层次化的方式：

- 发布（releases）——发布处于层级的顶端。一个发布包含有构建和运行一个系统的所有必要信息。一个发布由一个软件档案（archive）（以

某种形式打包）和一组安装该发布的规程组成。由于发布升级必须在不停止目标系统的情况下安装，因此安装一个发布的过程非常复杂。一个OTP发布将这种复杂性打包到一个单独的抽象单元中。在一个发布内部，包含零个或多个应用。

- 应用（applications）——应用比发布要简单，它包含所有的代码和运行一个单独应用所需要的所有操作规程，但并不是整个系统。当一个发布包含多个应用时，系统就应该按照这种方式来组织：要么确保每个不同的应用之间充分独立，要么不同的应用都有着严格的层次化依赖关系。
- 监督者——OTP的应用一般都是由一些监督者的实例构成。
- 工作者——OTP的监督者监督工作者节点。工作者节点通常是gen_server、gen_event或gen_fsm等behaviour的实例。

我们要特别解释一下应用。应用是从工作者节点开始自底向上（bottom-up）构建的。我会创建三个工作者节点（gen_server、gen_event和gen_fsm的实例各一个）。工作者节点由一个简单的监督树来管理，监督树被打包成一个应用。我就从工作者节点说起。

6.1.1 behaviour库是怎么写成的

OTP的behaviour都是用类似第4.1节的例子中的编程风格来编写的。只有一个主要的不同，我们不是通过任意的函数来参数化behaviour，而是通过模块的名字来参数化一个behaviour。该模块必须导出一些指定的预定义的（pre-defined）函数。具体哪些函数需要被导出，依赖于behaviour的定义方式。每个behaviour的完整的API在其使用手册中有详细的文档。

举个例子，假设xyz是gen_server这种behaviour的一个实例，那么xyz.erl就必须包含如下代码：

```
-module(xyz).
-behaviour(gen_server).
```

```
-export([init/1, handle_call/3, handle_cast/2,
         handle_info/2, terminate/2, change_code/3]).
...
```

xyz.erl 必须导出如上所示的 init/1…等六个函数。要创建一个 gen_server 的实例，我们就要调用：

```
gen_server:start(ServerName, Mod, Args, Options)
```

这里 ServerName 给服务器命名，Mod 填写原子 xyz，Args 是传递给 xyz:init/1 的参数，Options 是用来控制服务器自身的行为的参数。Options 不会作为参数传递给模块 xyz。

第 4 章中给出的例子中对 behaviour 的参数化的方法在某种程度上比 OTP 所采用的方法要更通用。造成这种差异主要是由于历史原因，最初的 behaviour 是在 fun 这种语法增加到 Erlang 中之前编写的。

6.2 通用服务器（Generic Server）的原理

第 4 章中我们介绍了通用服务器的思想。通用服务器提供了一个“空的”服务器，即一个可以被实例化为服务器的框架。第 4 章中的例子故意写得比较简短，但是清晰地阐明了制作一个通用服务器的相关原理。

在 OTP 系统中，Erlang 模块 gen_server 用来构造客户-服务器的服务器模块。gen_server 可以通过许多不同的途径被参数化成许多不同类型的服务器。

6.2.1 通用服务器的 API

为了便于理解 gen_server 的 API，我们来看看服务器程序与应用之间的控制流。我会描述一下 gen_server 的 API 中在本章的例子中将会用到的一个子集。

```
gen_server:start(Name1,Mod,Arg,Options) -> Result
```

在此：

Name1 = 服务器的名字（见注解 1）。

Mod = 回调模块的名字（见注解 3）。

Arg = 传递给 Mod:init/1 的参数（见注解 4）。

Options = 控制服务器工作方式的一组选项。

Result = 通过求值 Mod:init/1 而获得的值（见注解 4）。

```
gen_server:call(Name2,Term) -> Result
```

在此：

Name2= 服务器的名字（见注解 2）。

Term = 传递给 Mod:handle_call/3 的参数（见注解 4）。

Result = 通过求值 Mod:handle_call/1 而获得的值（见注解 4）。

```
gen_server:cast(Name2,Term) -> ok
```

在此：

Name2= 服务器的名字（见注解 2）。

Term = 传递给 Mod:handle_cast/3 的参数（见注解 4）。

注解：

1. Name1 应为如{local, Name2}或{global, Name2}般的项式。启动一个本地服务器会在一个单节点上创建一个服务器。启动一个全局服务器会在一个可为其它分布式 Erlang 节点透明地访问的节点上创建一个服务器。
2. Name2 是一个原子。
3. Mod 应当导出如下一些或全部函数：init/1，handle_call/3，handle_cast/3，terminate/2。这些函数将会被 gen_server 调用。
4. gen_server 的某些函数的参数会原封不动地作为参数传递给 Mod 的某些函数。类似的， Mod 的函数的返回值中包含的某些项式也会出现在 gen_server 的某些函数的返回值中。

Mod 所提供的回调函数应遵循如下规格：

Mod:init(Arg) -> {ok,State}|{stop,Reason}

此函数试图启动服务器：

Arg 是提供给 gen_server:start/4 的第 3 个参数。

{ok, State} 意思是服务器成功启动了。服务器的内部状态变成了状态 State，说明此时对 gen_server:start 的原始调用返回了{ok, Pid}，这里 Pid 是服务器的标识符。

{stop, Reason}意思是服务器启动失败了，这种情况下对 gen_server:start 的调用会返回{error, Reason}。

Mod:handle_call(Term,From,State) -> {reply,R,S1}

此函数在用户调用 gen_server:call(Name, Term)的时候被调用：

Term 是任意的一个项式（译注：该项式为用户自定义，用于标识具体的调用请求）。

From 标识客户。

State 是服务器当前的状态。

{reply, R, S1} 使 gen_server:call/2 的返回值为 R，而服务器的新状态变为 S1。

Mod:handle_cast(Term,State) -> {noreply, S1}|{stop,R,S1}

此函数在用户调用 gen_server:cast(Name, Term)的时候被调用：

Term 是任意的一个项式。

State 是服务器当前的状态。

{noreply, S1} 使服务器的状态变为 S1。

{stop, R,S1} 使服务器停止。服务器停止时要调用 Mod:terminate(R,S1)。

Mod:terminate(R,S) -> void

此函数在服务器停止的时候被调用，返回值被忽略：

R 是服务器终止的原因。

State 是服务器当前的状态。

6.2.2 通用服务器的例子

这里举一个用gen_server实现简单的*键-值*（Key-Value）服务器的例子。本键-值服务器用图 6.1 所示的一个叫kv[1]的回调模块来实现的。

kv 的第 2 行告诉编译器本模块时 gen_server 这种 behaviour 的回调模块。那么如果本模块没有导出 gen_server 所需要的正确的回调函数集，编译器就会产生告警。

kv.erl 导出了一些客户函数（见第 4 行）和一些回调函数（见第 6、7 行）。客户函数可以在系统内部任何地方调用。回调函数只会在 gen_server 模块内部被调用。

kv:start() 通过调用 gen_server:start_link/4 来启动服务器。传给 gen_server:start_link/4 的第 1 个参数为服务器的位置。在我们的例子中，位置为{local, kv}，意思是服务器是一个本地注册的进程，名字为 kv。关于位置的参数，还可以填写许多其他的值。包括{global, Name}，这种值标明用一个全局名字（而不是本地名字）来注册服务器。用一个全局名字将允许服务器可以被一个分布式 Erlang 系统中的其他任何节点访问。

gen_server:start_link/4的其余参数为：回调模块名字（kv）、初始化参数（arg1）、和一组控制和调试选项参数（[]）。如果把控制和调试选项参数设置成[{debug, [trace, log]}]那么将会开启调试器，并把调试信息写入到一个日志记录（log）文件。

当调用 gen_server:start_link/4 时，gen_server 会调用 kv:init(Arg)来对其内部数据结构进行初始化，这里 Arg 为提供给 gen_server:start_link/4 的第 3 个参数。一般来说，init/1 应该返回一个{ok, State}式的元组。

kv 的第 18－21 行导出的客户函数：store/2 和 lookup/1 通过调用

[1] kv.erl的第 26 行有一个故意的错误，现在请忽略它。

gen_server:call/2 来实现。

```
1  -module(kv).
2  -behaviour(gen_server).
3
4  -export([start/0, stop/0, lookup/1, store/2]).
5
6  -export([init/1, handle_call/3, handle_cast/2,
7           terminate/2]).
8
9  start() ->
10     gen_server:start_link({local,kv},kv,arg1,[]).
11
12 stop()  -> gen_server:cast(kv, stop).
13
14 init(arg1) ->
15     io:format("Key-Value server starting~n"),
16     {ok, dict:new()}.
17
18 store(Key, Val) ->
19     gen_server:call(kv, {store, Key, Val}).
20
21 lookup(Key) -> gen_server:call(kv, {lookup, Key}).
22
23 handle_call({store, Key, Val}, From, Dict) ->
24     Dict1 = dict:store(Key, Val, Dict),
25     {reply, ack, Dict1};
26 handle_call({lookup, crash}, From, Dict) ->
27     1/0;  %% <- deliberate error :-)
28 handle_call({lookup, Key}, From, Dict) ->
29     {reply, dict:find(Key, Dict), Dict}.
30
31 handle_cast(stop, Dict) -> {stop, normal, Dict}.
32
33 terminate(Reason, Dict) ->
34     io:format("K-V server terminating~n").
```

图 6.1：一个简单的服务器

在内部，远程过程调用（remote procedure call）的实现是通过调用回调函数 handle_call/2 来实现的。第 23－29 行实现了服务器侧的远程过程调用所需要的回调函数。handle_call 的第 1 个参数是一个模式，必须要与调用 gen_server:call/2

时使用的第 2 个参数匹配。第 3 个（译注：原文说是第 2 个）参数为服务器的状态。在一般情况下，handle_call 应该返回一个{reply, R, State1}，这里 R 是远程过程调用的返回值（该值也会成为 gen_server:call/2 的返回值，最终返回给客户），State1 将变成服务器的新的状态值。

在第 12 行 stop/0 中调用的 gen_server:cast(kv, stop)用来停止服务器。gen_server:cast(kv, stop)的第 2 个参数 stop 作为了 31 行中 handle_cast/2 的第 1 个参数，handle_cast/2 的第 1 个参数为服务器的状态。handle_cast 返回的{stop, Reason, State}将迫使通用服务器去调用 kv:terminate(Reason, State)。这种处理给了服务器一个机会去执行任何希望在退出之前执行的临终操作。当 termintate/2 返回时，通用服务器会停止下来，其所有已注册的名字也被移除。

在本例中，我们只是展示了一个使用通用服务器的简单的例子。gen_server 的手册将给出传给 gen_server 的回调函数和控制函数的参数能够接受的值的所有选择。通用服务器可以用许多种不同的方式来参数化，以便简化作为本地服务器或分布式 Erlang 节点网络上的全局服务器的运行。

通用服务器还有许多内置的调试帮助手段，可以方便程序员使用。用 gen_server 构建的服务器的内部发生一个错误时，关于哪里发生了错误的一个完整的调用轨迹会被自动添加到系统的错误日志中。该信息对于服务器的死因调查通常是很有意义的。

6.3 通用事件管理器（Event Manager）的原理

事件管理器 behaviourgen_event 提供了构建特定于应用的事件处理函数的一种通用框架。事件管理器可以完成如下任务：

- 错误处理。
- 告警管理。
- 调试。
- 设备管理。

事件管理器可以提供命名对象，事件可以发送给这些命名对象。在 1 个事件

管理器中，可以安装 0 个或多个事件处理器（event handler）。

当一个事件达到一个事件管理器时，它将会被该事件管理器内部安装的所有事件处理器进行处理。事件管理器可以在运行时被操纵，特别是我们可以在运行时安装一个事件处理器，去掉一个事件处理器或用另一个处理器来代替一个处理器。

我们先来看一些定义：

- 事件（Event）——发生的某件事情。
- 事件管理器（Event Manager）——一个对某一类事件的处理进行协调的程序。事件管理器提供一个命名对象，事件可以发送给它。
- 通知（Notification）——向一个事件管理器发送一个事件的动作。
- 事件处理器（Event Handler）——一个可以处理事件的函数。事件处理器必须是类型如下的函数：

```
State x Event -> State'
```

事件管理器维护一个{M, S}形式的“模块×状态”二元组的列表。我们称这样的列表为*模块-状态*（MS）列表。

假设事件管理器的内部状态可以用如下 MS 列表来表示：

[{M1,S1},{M2,S2},…]

当事件管理器接收到一个事件 E 的时候，如上的列表将变为：

[{M1,S1New},{M2,S2New},…]。

这里应该有{ok, SiNew} = Mi:handle_event(E, Si)。

事件管理器可以被当作是一个一般的常规有限状态机，只不过不是维护一个状态，我们维护的是一“组”状态和一组状态迁移函数。

如同我们可能预期的那样，gen_event 的 API 中也有许多接口函数，是用来操纵服务器中的{Module, State}对的。gen_event 比我们在这里的一点简单介绍要

强大得多。可以通过阅读 OTP 文档中关于事件处理方面的手册来了解的所有的细节。

6.3.1 通用事件管理器的 API

事件管理器（gen_event）导出了下列函数：

```
gen_event:start(Name1) -> {ok, Pid}|{error, Why}
```

创建一个事件管理器。

Name1 是事件管理器的名字（见注解 1）。

{ok, Pid}意味着事件管理器开启成功。Pid 就是事件管理器的进程 PID。

{error, Why}是在事件管理器开启失败时的返回值。

```
gen_event:add_handler(Name2,Mod,Args) -> ok | Error
```

添加一个新的处理器到事件管理器中。如果事件管理器的原有状态是 L，那么当此操作成功时，事件管理器的状态将变成[{Mod, S} | L]，这里 S 是调用 Mod:init(Args)获得的值。

Name2 是事件管理器的名字（见注解 1）。

Mod 是回调模块的名字（见注解 2）。

Arg 是传递给 Mod:init/1 的参数。

```
gen_event:notify(Name2,E) -> ok
```

发送一个事件 E 给事件管理器。如果事件管理器的状态是一个{Mi, Si}的集合集且收到一个事件 E，那么事件管理器的状态将编程{Mi, SiNew}的集合，而{ok, SiNew}=Mi:handle_event(E, Si)。

```
gen_event:call(Name2,Mod,Args) -> Reply
```

执行事件管理器中的某个事件处理器上的某个操作。如果事件管理器的状态列表包含一个元组{Mod, S}，那么将会调用 Mod:handle_call(Args, S)。Reply 就是源自该调用的返回值。

gen_event:stop(Name2) -> ok

停止事件管理器。

注解：

1. 事件管理器遵循与通用服务器相同的命名约定。
2. 一个事件处理器必须导出下列中的一些或全部函数：init/1，handle_event/2，handle_call/3，terminate/2。

一个事件处理器模块应该具有下列 API：

Mod:init(Args) -> {ok,State}

这里：

Args 来自 gen_event:add_handler/3 的第 3 个参数。

State 是本事件处理器的初始状态值。

Mod:handle_event(E,S) -> {ok,S1}

这里：

E 来自 gen_event:notify/2 的第 2 个参数。

S 是本事件处理器的原有状态值。

S1 本事件处理器的新的状态值。

Mod:handle_call(Args, State) -> {ok,Reply,State1}

这里：

Args 来自 gen_event:call/2 的第 2 个参数。

State 是本事件处理器的原有状态值。

Reply 将成为 gen_event:call/2 的返回值。

State1 是本事件处理器的新的状态值。

```
-module(simple_logger).
-behaviour(gen_event).

-export([start/0, stop/0, log/1, report/0]).

-export([init/1, terminate/2,
         handle_event/2, handle_call/2]).

-define(NAME, my_simple_event_logger).

start() ->
    case gen_event:start_link({local, ?NAME}) of
      Ret = {ok, Pid} ->
        gen_event:add_handler(?NAME,?MODULE,arg1),
        Ret;
      Other ->
        Other
    end.

stop() -> gen_event:stop(?NAME).

log(E) -> gen_event:notify(?NAME, {log, E}).

report() ->
    gen_event:call(?NAME, ?MODULE, report).

init(arg1) ->
    io:format("Logger starting~n"),
    {ok, []}.

handle_event({log, E}, S) -> {ok, trim([E|S])}.

handle_call(report, S)    -> {ok, S, S}.

terminate(stop, _)        -> true.

trim([X1,X2,X3,X4,X5|_])  -> [X1,X2,X3,X4,X5];
trim(L)                   -> L.
```

图 6.2：一个简单的错误记录器

Mod:terminate(Reason, State) -> void

这里：

Reason 标明事件管理器为什么被停止。

State 是本事件处理器的当前状态值。

6.3.2 通用事件管理器的例子

图 6.2 展示了如何用 gen_event 来构建一个简单的错误记录器。该错误记录器会跟踪最近的 5 个错误消息，还可以在收到 report 事件时显示最近的 5 个错误消息。

注意，simple_logger.erl 中的代码是纯顺序化的。在此，细心的读者应该会注意到传递给 gen_server 的参数的形式与传递给 gen_event 的参数的形式的相似之处。一般而言，传递给不同 behaviour 模块中诸如 start，stop，handle_call 等等函数的参数，我们会设计得尽量的相似。

6.4 通用有限状态机（Finite State Machine）的原理

许多应用（例如协议栈）可以用有限状态机（FSM）来建模。FSM 可以用有限状态机 behaviour，即 gen_fsm 来编写。

一个 FSM 可以用如下形式的一组规则来描述：

```
State(S) x Event(E) -> Actions (A) x State(S')
...
```

这个规则的意思是：

如果我们处于状态 S，发生了一个事件 E，那么我们应该执行操作 A，并把状态迁移到 S'。

如果我们选择用 gen_fsm 这种 behaviour 来编写一个 FSM，那么上面的状态迁移规则就应该被写作一些遵循如下约定的 Erlang 函数：

```
StateName(Event, StateData) ->
    .. code for actions here ...
    {next_state, StateName', StateData'}
```

6.4.1 通用有限状态机的 API

有限状态机 behaviour（gen_fsm）导出了下列函数：

```
gen_fsm:start(Name1,Mod,Arg,Options) -> Result
```

该函数的功能跟先前讨论过的 gen_server:start/4 一样。

```
gen_fsm:send_event(Name1, Event) -> ok
```

发送一个事件给标识符为 Name1 的 FSM。

回调模块 Mod 必须导出下列函数：

```
Mod:init(Arg) -> {ok,StateName,StateData}
```

当一个 FSM 启动的时候，它会调用 init/1，Mod:init/1 应该返回一个初始状态 StateName，和一些该状态的相关数据 StateData。接下来调用 gen_fsm:send_event(…, Event)时，FSM 会调用 Mod:StateName(Event, StateData)。

```
Mod:StateName(Event,SData) -> {nextstate,SName1,SData1}
```

在 FSM 运转时，StateName、Event 和 SData 表示 FSM 的当前状态。而 FSM 的下一个状态应为 SName1，下一个状态相关的数据应该为 SData1。

6.4.2 通用有限状态机的例子

为了描述一个典型 FSM 的应用，我利用 gen_fsm 写了一个简单的包聚合器（packet assembler）的程序。该包聚合器有 2 个状态：waiting 和 collecting。当它处于 waiting 状态时，它期望收到包含有包长度的信息，此时它会进入 collecting 状态。当它处于 collecting 状态时，它期望收到许多小的数据包，这些小的数据包将会被聚合。当所有小数据包的长度等于总的包长度时，FSM 会打印出聚合包，并重新进入 waiting 状态。

图 6.3 即用 gen_fsm 写的一个简单的包聚合器。在第 11 行我们调用了 gen_fsm:start_link/4 来创建一个 FSMbehaviour 的本地实例——注意与图 6.1 中第 10 行 gen_server:start_link/4 的相似之处。该调用的第 3 个参数作为传递给第 17 行 init/1 的参数。

waiting 状态用函数 waiting/2（第 21 行）来刻画，collecting 状态由 collecting/2（第 23—34 行）来刻画。与每个状态相关的数据存储在这些函数的第 2 个参数

```
1  -module(packet_assembler).
2  -behaviour(gen_fsm).
3
4  -export([start/0, send_header/1, send_data/1]).
5
6  -export([init/1,terminate/3,waiting/2,collecting/2]).
7
8  -define(NAME, my_simple_packet_assembler).
9
10 start() ->
11   gen_fsm:start_link({local, ?NAME},?MODULE,arg1,[]).
12
13 send_header(Len) -> gen_fsm:send_event(?NAME, Len).
14
15 send_data(Str)   -> gen_fsm:send_event(?NAME, Str).
16
17 init(arg1) ->
18   io:format("Packet assembler starting~n"),
19   {ok, waiting, nil}.
20
21 waiting(N, nil) ->
22   {next_state, collecting, {N,0,[]}}.
23
24 collecting(Buff0, {Need, Len, Buff1}) ->
25   L = length(Buff0),
26   if
27     L + Len < Need ->
28       {next_state, collecting,
29         {Need, Len+L, Buff1++Buff0}};
30     L + Len == Need ->
31       Buff = Buff1 ++ Buff0,
32       io:format("Got data:~s~n", [Buff]),
33       {next_state, waiting, nil}
34   end.
35
36 terminate(Reason, State, Data) ->
37   io:format("packet assembler terminated:"
38             "~p ~n", [Reason]),
39   true.
```

图 6.3：一个简单的包聚合器

中。这两个函数的第 1 个参数都为由调用 gen_fsm:send_event/2 时传入的第 2 个参数。例如，在 send_data/1 调用 gen_fsm:send_event/2 时第 2 个参数时 Len，这个参数就成为了第 21 行 waiting/2 的第 1 个参数。

本 FSM 的数据用一个 3 元组{Need, Len, Buff}来表示。当收集数据的时候，Need 为需要收集的数据的总长度，Len 为收集到的数据的实际长度，Buff 为包含收集来的数据的缓冲区。这个 3 元组是第 24 行 collecting/2 的第 2 个参数。

我们可以在 Erlang 的 shell 中下达一段命令来看看这个包聚合器的用法：

```
> packet_assembler:start().
{ok,<0.44.0>}
> packet_assembler:send_header(9).
ok
> packet_assembler:send_data("Hello").
ok
> packet_assembler:send_data(" ").
ok
> packet_assembler:send_data("Joe").
Got data:Hello Joe
ok
```

再次强调，gen_fsm 比这里所描述的要有用得多。

6.5 通用监督者（Supervisor）的原理

到目前为止，我们所着重讲到的都是为了解决典型应用问题的一些基本 behaviour，而编写应用中大部分问题也都可以用基本的客户-服务器、事件处理、和 FSM 等 behaviour 来解决。这里要讲的 gen_sup 这种 behaviour 是第一个*元行为*（meta-behaviour），即用来将基本 behaviour 粘合成一个监督体系的 behaviour。

6.5.1 通用监督者的 API

通用监督者的 API 是极其简单的：

```
supervisor:start_link(Name1,Mod,Arg) -> Result
```

本函数开启一个监督者，其间调用 Mod:init(Arg)函数。

回调模块 Mod 必须导出 init/1 函数，规格如：

```
Mod:init(Arg) -> SupStrategy
```

SupStrategy 是描述监督树的项式。

SupStrategy 是一个描述监督树中的工作者们如何被启动、停止和重启的项式。我不在这里详细描述，接下来的一个简单的监督树的例子会有比较详尽的描述。关于通用监督者的完整细节可以参见用户手册的相关部分。

6.5.2 通用监督者的例子

图 6.4 中的例子是一个监督这监督了前面各节所介绍的三个工作者。回想一下kv.erl中故意包含了一个错误（见图 6.1 第 26 行），并且simple_logger.erl也包含了一个错误[2]（我故意没有提）。我们现在来看看当运行时发生这些错误时，会发生什么事情。

simple_sup.erl 模块（图 6.4）定义了该监督者的行为。开始在第 7 行调用了superivsor:start_link/3——这与系统中其它 behaviour 的调用习惯是一致的。?MODULE 是一个宏，被展开为当前模块的名字 simple_sup。最后一个参数被设置为 nil。监督者开启的时候会用 start_link/3 的第 3 个参数作为参数去调用指定的回调模块中的 init/1 函数。

init/1 返回一个定义了监督树的形状和所采用的策略的数据结构。项式{one_for_one, 5, 1000}（第 11 行）告诉监督者构建一个“或”型监督树（参见 5.2.3 小节）——这是因为它所监督的三个工作者是彼此没有关系的。数字 5 和 1000 指定了一个重启频率（restart frequency）——如果监督者在 1000 秒钟内重启了被监督者超过 5 次，则监督者本身将会出错。

[2] 我希望你已经发现了该错误。

```
1  -module(simple_sup).
2  -behaviour(supervisor).
3
4  -export([start/0, init/1]).
5
6  start() ->
7    supervisor:start_link({local, simple_supervisor},
8                          ?MODULE, nil).
9  init(_) ->
10   {ok,
11    {{one_for_one, 5, 1000},
12     [
13      {packet,
14        {packet_assembler, start, []},
15        permanent, 500, worker, [packet_assembler]},
16      {server,
17        {kv, start, []},
18        permanent, 500, worker, [kv]},
19      {logger,
20        {simple_logger, start, []},
21        permanent, 500, worker, [simple_logger]}]}}.
```

图 6.4：一个简单的监督者

这里我们的监督树中有三个被监督对象，但是我只描述包聚合器是如何添加到监督树中的。另外两个工作者的添加方法依次类推。

第 13－15 行指定了包聚合器这个工作者。

第 13 行开始，元组中的第一个元素描述了包聚合器如何被监督。原子 packet 是一个任意的名字（在是在本监督者实例的内部要保证是唯一的），可以用来指示监督树中的节点。

因为被监督者本身也是 OTP 的 behaviour 的实例，所以把他们添加到监督树中会很容易。下一个参数（第 14 行）是一个 3 元组{M, F, A}，被监督者用来启动指定的进程。如果监督者要启动一个被监督的进程，它会去调用 apply(M,F,A)。

第 15 行的第一个参数 permanent 是说被监督的进程是一个所谓的“永恒”进程。一个永恒进程在它出错时将会被其监督者自动重启。

一个被监督进程不单要指明如何被启动，还需要按照一定的方式来编写。例如，它必须能够在监督者要求它终止时井然有序地终止。为了做到这一点，被监督进程必须遵守所谓的“停止协议”（shutdown protocol）。

监督者通过调用 shutdown(P, How)来终止一个工作者进程，这里 P 是工作者的 Pid，而 How 决定了工作者如何被停止。shutdown 定义如下：

```
shutdown(Pid, brutal_kill) ->
    exit(Pid, kill);
shutdown(Pid, infinity) ->
    exit(Pid, shutdown),
    receive
        {'EXIT', Pid, shutdown} -> true
    end;
shutdown(Pid, Time) ->
    exit(Pid, shutdown),
    receive
        {'EXIT', Pid, shutdown} ->
            true
    after Time ->
        exit(Pid, kill)
    end.
```

如果 How 是 brutal_kill，那么工作进程会被杀死（参见第 3.5.6 小节）。

如果 How 是 infinity，那么一个 shutdown 的信号会被发送给工作者进程，而工作者进程应当回以一条{‘EXIT’, Pid, shutdown}消息。

如果 How 是一个整数 T，那么工作者进程需要在给定的 T 毫秒事件内终止，如果在 T 毫秒之内没有收到{‘EXIT’, Pid, shutdown}的消息，那么该进程会被无条件杀死。

图 6.4 的第 15 行的整数 500 是关停协议所需要的一个“关停时间”。着说明如果监督者想要停止一个被监督进程时，它被允许有最多 500 毫秒的时间来停止

目前正在处理的事情。

参数 worker 表示被监督进程是一个工作者进程（回想一下，在 5.2 节中我们说过一个被监督者进程可以为一个工作者或监督者进程），[packet_assembler]是本监督者使用的所有模块的列表（这个参数在同步代码变更操作时要用到）。

一旦所有的事情都定义好了，我们就可以编译运行该监督者了。在接下来的的演示脚本中，我启动了一个监督者，并触发了被监督者中的几个错误。被监督者会死掉并被监督者自动重启。

第一个例子是展示一下当包聚合器中发生一个错误时，会发生什么。我们启动监督者，并检查一下包聚合器的 Pid。

```
1> simple_sup:start().
Packet assembler starting
Key-Value server starting
Logger starting
{ok,<0.30.0>}
2> whereis(my_simple_packet_assembler).
<0.31.0>
```

打印输出显示，所有的服务器都起来了。

现在我们来发送一个指定聚合长度为 3 字节的命令，而接下来发送一条 4 字节长的数据：[3]

```
3> packet_assembler:send_header(3).
ok
4> packet_assembler:send_data("oops").
packet assembler terminated:
    {if_clause,
        [{packet_assembler,collecting,2},
        {gen_fsm,handle_msg,7},
```

[3] 这就是我故意留下的第二个错误，我相信你一定已经发现了！

```
        {proc_lib,init_p,5}]}
ok
Packet assembler starting
=ERROR REPORT==== 3-Jun-2003::12:38:07 ===
** State machine my_simple_packet_assembler terminating
** Last event in was "oops"
** When State == collecting
**     Data   == {3,0,[]}
** Reason for termination =
** {if_clause,[{packet_assembler,collecting,2},
              {gen_fsm,handle_msg,7},
              {proc_lib,init_p,5}]}
```

这个错误引起的打印相当多。首先是包聚合器崩溃了，看第一条错误输出就知道。紧接着，监督者检测到了包聚合器崩溃的情况并重启了它——该进程重启的时候会打印“Packet assembler starting”消息。最后，有一条长长的、含有所期望的有用信息的出错消息。

该出错消息包含了 FSM 在崩溃的时刻的状态信息。它告诉我们，FSM 当时所处的状态是 collecting，该状态关联的数据为一个 3 元组{3,0,[]}，并且引起 FSM 崩溃的事件是“oops”。这些信息对于 FSM 的调试是相当有用的。

在这里，错误日志被直接定向到了标准输出。但是在一个产品系统中，错误日志可以被配置为定向到持久存储设备。对错误日志的分析对于系统死因的诊断应该是很有意义的。

我们可以确认一下，监督者已经正确地重启了包聚合器，求值一下 whereis(my_simple_packet_assembler)就会返回新起来的包聚合器的 Pid。

```
6> whereis(my_simple_packet_assembler).
<0.40.0>
7> packet_assembler:send_header(6).
ok
8> packet_assembler:send_header("Ok now").
```

```
Got data:Ok now
ok
```

用类似的方法，我们可以触发在 Key-Value 服务器中故意留下的那个错误：

```
12> kv:store(a,1).
ack
13> kv:lookup(a).
{ok,1}
14> spawn(fun() -> kv:lookup(crash) end).
<0.49.0>
K-V server terminating
Key-Value server starting
15>
=ERROR REPORT==== 3-Jun-2003::12:54:10 ===
** Generic server kv terminating
** Last message in was {lookup,crash}
** When Server state == {dict,1,
                                16,
                                16,
... many lines removed ...
** Reason for termination ==
** {badarith,[{kv,handle_call,3},{proc_lib,init_p,5}]}
15> kv:lookup(a).
error
```

请注意，kv:lookup(crash)必须通过一个没有连接到shell进程（query shell）的临时进程来调用。这是因为监督者是通过调用supervisor:start_link/4 的方式来启动的，所以监督者被连接到了shell进程。在shell里直接调用kv:lookup(crash)会使监督者进程也崩溃掉，这很可能不是我们所期望的。[4]

[4] 我最初这样的尝试失败了，是Erlang邮件清单(mailing list)中的Chandrashekhar Mullaparthi非常友好地指出了为什么我的做法会失败。

还请注意通用监督者和预先定义的（pre-defined）behaviour是如何*一起*（together）工作的。通用监督者与基本behaviour不是设计成各自孤立的，而是设计成相互补充的。

还有，默认的做法是在错误日志中提供尽可能多的有用信息，并努力使系统处于一种安全的状态。

6.6 通用应用（Application）的原理

我们迄今已经构建了三种基本 behaviour，并把他们放进了一棵监督树中；剩下的事情就是把所有的东西都塞到一个应用（application）里。

一个应用就是一个包含交付一个应用程序时需要的一切事物容器。

应用的编写方式跟先前讨论的 behaviour 的编写方式不一样。之前的 behaviour 都要用到回调模块，回调模块导出一些预定义函数。

应用不使用回调函数，而是表现为文件系统中的文件、目录、子目录的一种特殊的组织形式。一个应用的最重要的部分包含在应用描述子文件（application descriptor file）（一个扩展名为.app 的文件）中，该文件描述了一个应用所需要的所有资源。

6.6.1 通用应用的 API

应用是用一个应用描述子文件来描述的。一个应用描述子文件的扩展名是.app。在 MAN(4)用户手册中，对于一个应用的.app 文件的结构作了如下定义：

```
{application, Application,
   [{description,          Description},
    {vsn,                  Vsn},
    {id,                   Id},
    {modules,              [Module1, .., ModuleN]},
    {maxT,                 MaxT},
    {registered,           [Name1, .., NameN]},
    {applications,         [Appl1, .., ApplN]},
```

```
{included_applications, [Appl1, .., ApplN]},
{env,                   [{Par1, Val1}, .., {ParN, ValN}]},
{mod,                   {Module, StartArgs}},
{start_phases,
        [{Phase1, PhaseArgs1}, ..,
          {PhaseN, PhaseArgsN}]}]}.
```

应用联合清单（application association list）中的所有键（key）都是可选的，如果被忽略，就会采用一个合理的默认值。

6.6.2 通用应用的例子

为了将我们的包含三个基本 behaviour 和一个监督者的这么一个应用打包，我们使用了一个图 6.5 所示的应用文件 simple.app。

```
{application, 'simple',
 [{description, "A simple application"},
  {vsn,         "1.0"},
  {modules,      [simple,kv,packet_assembler,
                  simple_sup,simple_logger]},
  {maxT,         infinity},
  {registered,   [kv, my_simple_event_logger,
                  my_simple_packet_assembler]},
  {applications, []},
  {included_applications, []},
  {env,          []},
  {mod,          {simple, go}}]}.

```

图 6.5：simple.app ——一个简单的应用

在我们的例子中，.app 文件的结构是相当直白的。

应用文件的主要目的是为了命名和描述应用，列举出应用中用到的所有的模块和注册进程的名字。

除了 simple.app 以外，我们还需要一个主程序，用来“发动”（launch）应用程序；我们可以用图 6.6 所示的 simple.erl 作为主程序。simlpe.erl 包含两个对应

用进行开启和停止的函数。

```
-module(simple).
-behaviour(application).

-export([start/2).

start(_, _) -> simple_sup:start().

```

图 6.6：simple.erl ——一个简单的应用

现在，我们已经准备好运行该应用了。假设所有的 Erlang 文件都已经被编译过，而且与.app 文件在同一目录下，那么我们可以通过如下方式开启该应用，并测试其中的一个服务器：

```
1> application:start(simple, temporary).
Packet assembler starting
Key-Value server starting
Logger starting
ok
2> packet_assembler:send_header(2).
ok
3> packet_assembler:send_data("hi").
ok
Got data:hi
```

现在我们可以停止该应用：

```
4> application:stop(simple).
=INFO REPORT==== 3-Jun-2003::14:33:26 ===
     application: simple
     exited: stopped
     type: temporary
ok
```

在停止了应用以后，应用中运行着的所有进程都将依次关掉。

6.7 系统与发布（release）

本章的铺陈是“自底向上”的。我以简单的东西开始，将它们组合成更大的更复杂的单元。我是以几个基本 behaviour 如 gen_server、gen_event 和 gen_fsm 开始的，然后把这些基本通用模式组织到了一个监督层次体系中，然后把这个监督层次体系构建到了一个应用包中。

最后一步（这里没有展示出来）是将应用包构建到一个发布中。一个发布可以将多个不同的应用打包成一个单一概念单元。结果就是可以移植到目标环境的少数几个文件。

构建一个完整的发布是一个复杂的过程——一个发布不仅要描述系统的当前状态，而且还要知道系统的之前的版本。

发布不但要包含软件当前版本的信息，而且还要包含软件的之前的发布的信息。特别地，发布应该包含将系统从早先版本的软件升级到当前版本的软件的规程。这种升级通常需要在不停下系统的情况下进行。一个发布还必须能够处理新软件因某些原因出现安装失败的情况。如果一个新发布出错，系统还应该能够回退到之前的某个稳定状态。所有的这些都由 OTP 系统的发布管理组件来处理。

等到我们考察第 8 章的 AXD301 项目的时候，我们会发现有 gen_server 的 122 个实例、gen_event 的 36 个实例和 gen_fsm 的 10 个实例，有 20 个监督者和 6 个应用，所有的这些都被打包到一个发布中。

我认为，这些 behaviour 中最简单的要数 gen_server，它也恰好是单独作为设计模式使用得最多的。在 gen_server 的回调模块中发生的错误应该产生有信息含量的、能够帮助进行系统死因诊断的出错消息。

使用上面的 behaviour 也许要在设计和编程效率方面有所折衷。使用设计模式来编写一个系统要快一些，但是最终代码很可能会比纯手写的解决同样问题的代码要低效一些。

6.8 讨论

- OTP 系统中实现 behaviour 的通用模块是有专家 Erlang 编程人员编写的。这些模块都是建立在许多年的经验的基础上的，代表了编写代码来解决某些特殊问题的“最佳实践”。

- 使用 OTP 的 behaviour 来构建的系统拥有非常有规则的结构，例如，所有的客户-服务器和监督树都有着同样的结构。使用 behaviour，就会迫使解决某一问题时采用公共的结构。应用程序员只需要提供定义他们的特殊问题的语义的代码，而所有的基础设施都由 behaviour 自动提供。

- 对于加入已经存在的团队的一个新程序员来说，基于 behaviour 的解决问题的方式更容易理解。只要他们熟悉了 behaviour，他们就能够很轻易地识别出哪种情况下应该用哪种 behaviour。

- 系统编程中大部分的“复杂问题”（tricky）都被隐蔽在了 behaviour 的实现中（这些复杂的问题实际上比我们这里描述的还要复杂得多）。如果你回头看看客户-服务器和事件处理器 behaviour，你会发现所有处理并发、消息传递等等事务的代码都被隔离在了 behaviour 的“通用”部分，而“问题相关”的代码都是一些有着良好的类型定义的纯顺序化函数。

  这正是编程中人高度期望的一种境界——“困难”的并发程序被隔离成了系统中的一些定义良好的小的部分。系统中绝大部分代码能够用有着良好类型定义的顺序化的程序来编写。

在我们的系统中，behaviour 解决的都是*正交的问题*（orthogonal problems）——例如，客户-服务器与工作者-监督者没有任何关系。在构建真实系统的时候，我们会挑选并混合使用 behaviour，并把他们用不同的方式组合起来解决问题。

为一个软件设计者提供一个小的、混合的 behaviour 集有如下诸多好处：

- 它关注于一小组久经考验的技术。我们事先都都知道单个技术可以在

实现中工作很好。如果对设计完全不加限制并且活动绝对自由，那么设计者就可能会受到诱惑制造出一些有着不必要的复杂性的东西，或者制造一些根本不能实现的东西。

- 它允许设计者以一种精确的方式来构造和讨论设计。它提供了一个谈论时的共同词汇。
- 它完成了设计与实现之间的反馈环。这里所讲的所有 behaviour 都有实用。例如，他们都曾在 Ericsson 的 AXD301 产品中使用。

7 OTP 介绍

开放电信平台（Open Telecom Platform, OTP）是为了构建和运行电信系统而设计的一个开发系统。图 7.1 给出了该系统的一个方块图，该图来自参考文献[66]。如图中所示，OTP 系统是设计来运行在通常的操作系统之上的一个所谓的“中间件平台”。

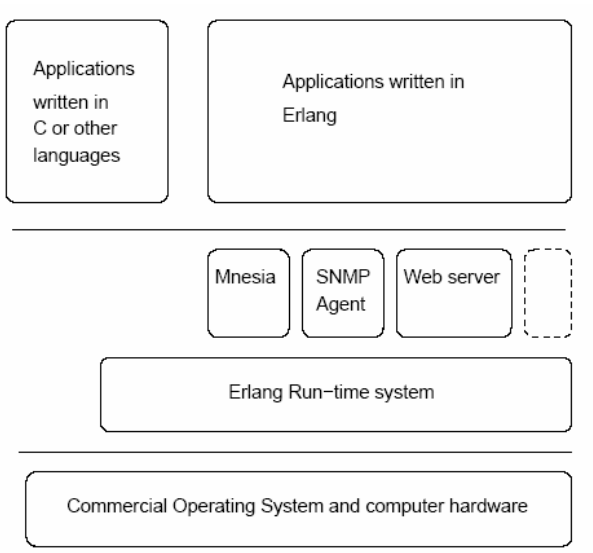


图 7.1：OTP 系统架构

OTP其实是在Ericsson内部开发的，但是其中大部分软件已经按照Erlang公开许可（Erlang public license）[1]公布给了公共领域。

OTP 的发布包含有如下一些部件：

1．Erlang 的编译器和开发工具。

2．适应多种不同目标环境的 Erlang 运行时系统。

3．覆盖广泛的公共应用的一些库。

4．实现公共行为模式的一组设计模式。

5．用来学习如何使用该系统的一些教学资料。

6．大量的文档。

[1] 非常类似于一个开源许可。

OTP 已经被移植到了许多不同的操作系统上，包括所有的 Unix 类的系统（Linux、 FreeBSD、Solaris、OS-X…），大多数的 Windows 操作系统（Windows 95、98、NT…）和一些 VxWorks 之类的嵌入式操作系统。

Erlang 运行时系统是一个用来运行由 Erlang 的 BEAM 编译器产生的中间码（intermediate code）的虚拟机。它同时也为 Erlang 编译器产生的本地码（native code）提供运行时支撑服务。

Erlang 的 BEAM 编译器自 1998 年后就取代了原始的 JAM 编译器。BEAM 编译器[41,42]将 Erlang 源代码编译成为 32bit 字宽的线索化解释器（threaded interpreter）使用的指令序列。而原始的 JAM 机是一个非线索化（non-threaded）字节码解释器（byte code interpreter）。

另外，为了提高效率，Erlang 程序还可以用 Uppsala 大学开发的 HIPE 编译器[47]编译成本地码（native code）。编译成被解释执行的 BEAM 中间码的模块和编译成本地码的模块在模块级是可以自由混合的，即整个模块既可以被编译成 BEAM 码，也可以被编译成 HIPE 码，但是在模块内部，两种码不能混合。

BEAM 机和 HIPE 机共同使用了 Erlang 运行时系统中关于内存管理、输入/输入、进程管理、垃圾收集等部件的代码。

Erlang 运行时系统提供了许多传统上由操作系统提供的服务，所以，Erlang 运行时系统远不仅仅提供纯序列化语言的运行时支撑，而比这要复杂得多。所有得 Erlang 进程都由 Erlang 运行时进程来管理——即使在一个 Erlang 运行时系统控制着数以万计的 Erlang 进程的时候，宿主操作系统也只会感到只有一个进程在运行，那就是 Erlang 运行时系统本身。

另一方面，与其他语言相比，Erlang 的编译器又是相当简单的。编译通常只是一个从 Erlang 代码到一条合适的虚拟机原语的一个简单翻译。所以，例如 Erlang 中的 spawn 原语被翻译成虚拟机中的一条单独的操作码（opcode）（即 spawn 原语的实现），然后付出很大的努力使得操作码的实现尽量的高效。

7.1 库

OTP 的发布包包含由一个很大的库集，为了发布的目的，其中所有的库都作为 OTP 应用的实例。例如发布包 R9B（译注：Erlang/OTP 的一个发布版本号）就包含如下这些应用：

- appmon ——一个监控和操纵监督树的一个图形化工具。
- asn1 ——一个按照 ASN.1 定义的一个编译器和运行时编/解码支持包。
- compiler ——Erlang 的编译器。
- crypto ——一个用于加密/解密数据和计算消息摘要（message digests）的函数集。
- debugger ——一个 Erlang 源代码调试器。
- erl_interface——一个用于与分布式 Erlang 节点通信的库文件集。
- erts——Erlang 运行时系统。
- et——一个事件跟踪器和一些记录事件数据并进行图形化表示的工具。
- eva——负责“事件与告警”处理的应用。
- gs——一个图形系统，一组用于构建 GUI 的图形函数。
- ic——Erlang 的 IDL 编译器
- inets——一个 HTTP 服务器和一个 FTP 客户。
- jinterface——一个编写 Java 与 Erlang 的接口的工具。
- kernel——系统得以运行所需要的两个基本库之一（另一个是 stdlib）。本库包含文件服务器、代码服务器的实现。
- megaco——支持 Megaco2/H248 协议的库集。
- mnemosyne——一种用在 mnesia 上的数据库查询语言。

[2] Media Gateway Control，即媒体网关控制器

- mnesia——一个具有 Erlang 的软实时特性的 DBMS（译注：数据库管理系统）。
- observer——一个用于跟踪和观测分布式系统的行为的工具集。
- odbc——一个用于 Erlang 访问 SQL 数据库的 ODBC 接口。
- orber——一个 CORBA 对象请求代理的 Erlang 实现。注意：还有其它一些单独的应用，来提供对不同 CORBA 服务（如事件、通知、文件传输等）的访问。
- os_mon——一个监控外部操作系统的资源使用情况的工具。
- parsetool——解析 Erlang 的工具。包括 yecc，即 LALR(1)解析器生成器（parser generator）。
- pman——一个查看系统状态的图形化工具。pman 可以用来查看本地或远端的 Erlang 节点。
- runtime_tools——运行时系统所需要的各种小函数。
- sasl——“System Architecture Support Libraries”（系统结构支持库）的缩写。本应用包含对告警处理(alarm handling)和发布管理（managing releases）的支持。
- snmp——简单网络管理协议（Simple Network Management Protocol）[24]的 Erlang 实现。本应用包含一个 MIB 编译器和一些 MIB 编写的工具。
- ssl——一个 Erlang 的安全套接字层（secure sockets layer）接口。
- stdlib——系统得以运行的“必备的”Erlang 库集。另一个必备的库集是 kernel。
- toolbar——一个可以从中开启应用的图形化工具条。
- tools——一个由各种用于分析和监测 Erlang 程序的独立应用组成的包。这些应用即一些性能评估（profiling）、覆盖率分析（coverage analysis）、交叉引用分析（cross reference analysis）的工具。

- tv——一个“表浏览器”（table viewer）。本表浏览器是一个可以对 mnesia 数据库种的表进行图形化浏览的图形化应用。
- webtool——一个用于管理基于网页的工具（如 inets）的系统。

OTP 库集提供了一个高度成熟的工具集，是编写任何商用软件的一个很好的起点，然而，OTP 库集是相当庞杂的。

回想一下，我们的第 6 章只是对五种 behaviour（gen_server、gen_event、gen_fsm、supervisor、application）进行了一个简明的解释，而对其中任何一种 behaviour 的完整的解释都超出了本文的范围。本文只是在第 4.1 节对其中一个 behaviour（gen_server）背后的原理进行了较完整的交待。

发布包 R9B 中的 stdlib 应用一共包含 71 个模块——我们已经在这里描述了其中的 4 个。

8 案例研究

论文的本部分将展示几个系统的研究案例，这些系统均是用Erlang/OTP平台开发的。第一个系统是Ericsson的AXD301 系统——AXD301 系统是一个大容量的ATM[1]交换机。在这里我们所研究的AXD301 的版本有超过 110 万行Erlang代码，是用函数式编程编写过的最庞大的系统之一。AXD301 广泛应用了OTP库集，因此它为OTP库集的功能性提供了一个很好的证明。

在该案例之后，我还将陈述对 Bluetail 公司或 Alteon Web Systems/Nortel Networks 公司的几个小产品的研究。为了避免混淆，我需要补充说明一下：Bluetail AB 公司是由几个来自 Ericsson CSLab 的“Erlang 人”（包括我自己）成立的一个公司，后来 Bluetail 被 Alteon Web Systems 公司收购，再后来，Alteon 又被 Nortel Networks 收购。然后，那些产品是由同一个核心团队的人开发的。

这些产品包括Bluetail Mail Robustifier（BMR）和Alteon Web Systems公司开发而后来Nortel Networks公司销售的“SSL[2] accelerator”。其中SSL accelerator这款产品是在一段相当短的时间（9 个月）内开发完成的，而迅速成为“嵌入式安全套接字层设备”这一不大市场中的“领头羊”。SSL accelerator也广泛地使用了Erlang/OTP平台和库集。

这些项目代表了两个极限。AXD301 由一个大型的程序员团队开发，有超过 40 位程序员为该系统的编写贡献了超过 4 年的时间。众所周知，大型软件项目的管理是相当困难的，所编写的代码大多也非常难以理解。在此我正要关心的一件事情是：OTP 的设计方法学对于大型系统的构建的支持是好还是不好呢？

第二组产品的开发是由一个小得多的团队编写（5－10 个程序员，取决与产品的不同）的，但是却在一个短得多得时间（6 个月）内完成的。所有的开发者都是经验丰富的 Erlang 程序员。其中的两位，Magnus Fröberg 和 Martin Björklund 是 OTP 的 behaviour 的最初设计和编写者；其中另一位，Clares Wikström 是 Erlang 书第 2 版的联合作者，Wikström 还是分布式 Erlang 和 mnesia 数据库的主要实现者。

[1] Asynchronous Transfer Mode，异步传输模式。
[2] Secure Socket Layer，安全套接字层。

8.1 方法学

在案例研究中，我所关注的是如下方面：

- 问题领域——问题领域是什么？这个问题属于 Erlang/OTP 的设计所要解决的问题范围吗？
- 代码的量化特性——写了多少行代码？一共有多少个模块？这些代码是如何组织的？程序员都遵循了设计规范吗？定下设计规范有用处吗？哪些是好的？哪些是不好的？
- 可容错性的证据——系统可容错吗？Erlang 的初衷（raison d'être）就是为了构建可容错的系统。有证据证明确实发生了运行时错误并被成功地纠正了吗？一个编程错误发生的时候产生的信息对于后续的程序纠正是不是足够充分？
- 系统的可理解性——系统的可理解性如何？便于维护吗？

我不是问一些关于系统属性的笼统的问题，而是去寻找了一些明确的证据，来证明系统确实在按照我们期望的方式运行。特别是：

1. 是否有证据证明系统确实曾因为编程错误而崩溃过，并且该错误被纠正了，并且系统能够从该错误中恢复过来，并且在错误被纠正后能以一种让人满意的方式运行？
2. 是否有证据证明系统已经运行了很长时间，并且期间已经发生过软件错误而系统依然稳固？
3. 是否有证据证明系统的代码曾经“在运行中”（on the fly）升级。
4. 是否有证据证明像垃圾回收这些机制起了作用（也就是我们已经长时间运行了垃圾回收系统而没有发生垃圾回收的错误）？
5. 是否有证据证明错误日志中的信息对于出错后的错误定位很有意义？
6. 是否有证据证明系统的所有代码都以某种方式组织了起来，从而大多数的程序员不必关心系统中使用的并发模式的细节？

7．是否有证据证明监督树如期所望地工作着？

8．代码是否按照“净/脏”风格来组织？

上面的第 1、2、5 条的存在是因为我们希望检测我们关于编写可容错系统的思想在实践中起到了作用。

第 4 条检验的是对于必须长时间运行的系统来说，垃圾回收确实起了作用。

第 6 条是对 OTP 的 behaviour 的抽象能力的一个衡量。有太多的原因使我们希望能够“抽象出”很多情形下普遍存在的并发处理的细节。OTP 的 behaviour 集就是尝试这么做。对于一个初涉我们程序的程序员来说，他在多大程度上忽略了并发处理是度量 OTP 的 behaviour 是否适合开发系统软件的一个重要的指标。我们可以通过观察程序员在他们的代码中多么频繁地使用显式消息传递和进程操作原语来评估并发处理可以被忽略的程度。

第 7 条检验了监督者策略是否如期望般地起效。

第 8 条检验了我们是否可以按照附录 B 中给出的编程规范来编程。特别是它的指导方针强调了按照“净/脏”方式来组织系统的重要性。这里我们说的“净”代码是指没有副作用的代码，这样的代码比“脏”代码更容易理解，而“脏”代码是指有副作用的代码。

我们的整个系统跟硬件操作是息息相关的，而这种硬件操作就会带来副作用。因此，我们关心的不是可否避免副作用，而是我们能够在多大程度上把副作用限制在尽量少的模块中。与其让有副作用的代码均匀地散布在整个系统中，不如希望能够把大量的副作用限制在少数“脏”模块中，而大多数模块都以无副作用的方式编写，与“脏”模块组合起来成为整个系统。对代码进行一下分析就可以揭示这种组织方式是否可行。

当然“反例”也很重要。我们想知道我们的范型不适用的所有情形，以及这种不适用是否是一个大问题。

8.2 AXD301

AXD301[18]系统是 Ericsson 生产的一种高性能异步传输模式（ATM）交换机。整个系统由许多可伸缩的模块组成——每个模块提供 10GBit/s 的交换容量，那么 16 个模块加起来联在一起就能形成一台 160GBit/s 的交换机。

AXD301 是为支持“运营商级”不停机运转[70]而设计的。该系统由重复的硬件来提供硬件冗余，并且硬件可以在不打断业务的情况下添加到系统中或从系统中移除。软件必须要能够应付硬件和软件故障。因为系统是为不停机运转而设计的，所以它必须能够在不干扰系统流量的前提下进行软件修改。

8.3 软件的量化特性

下面展示了一个对 AXD301 软件的简单统计的分析结果。这份简单统计显示了该系统在 2001 年 12 月 5 日当时的状态。

这份分析报告只关注于系统的 Erlang 代码的一些量化特性。系统的总体量化特性如下：

总的 Erlang 模块的数量	2248
“净”模块数	1472
“脏”模块数	776
代码行数	1136150
总的 Erlang 函数个数	57412
“净”函数的个数	53322
“脏”函数的个数	4090
“脏”函数个数 / 代码行数之比率	0.359%

上表中只是粗浅地对每个模块或函数是“净”是“脏”进行区分而做的一个简单的分析。如果一个模块中有任何一个函数是“脏”的，我们就认为它是“脏”的，否则就认为它是“净”的。为了简化处理，如果一个函数进行了接收或发送数据，或者调用了如下一些 Erlang 的 BIF，我们就说它是脏的：

```
apply,  cancel_timer,  check_process_code,  delete_module,
```

demonitor, disconnect_node, erase, group_leader, halt, link, load_module, monitor_node, open_port, port_close, port_command, port_control, process_flag, processes, purge_module, put, register, registered, resume_process, send_nosuspend, spawn, spawn_link, spawn_opt, suspend_process, system_flag, trace, trace_info, trace_pattern, unlink, unregister, yield.

之所以这样区分，是因为调用了这些 BIF 的代码段都会有潜在的危险性。

请注意我在这里给“脏模块”下了一个特别简化的定义。直觉上似乎应该给“脏模块”一个递归地定义，即如果一个模块中有任何函数调用了“危险的”BIF 或另一个模块中的“脏函数”，则该模块为“脏模块”。不幸的是，如果按照这个定义来判定，那么系统中几乎所有的模块都将被判为“脏模块”。

原因在于，如果你统计一下对某个模块导出的所有函数的调用的传递闭包（transitive closure），你就会发现这个传递闭包实际上囊括了系统的几乎所有模块。这个传递闭包之所以这么大，应归咎于 Erlang 库中许多模块都发生了“泄漏”（leakage）。

我们简单地认为，所有的模块都写得很好且经过了测试，并且如果一个模块确实包含有副作用，那么在编写该模块时也会注意这种副作用不会泄漏出该模块，从而对调用该模块的代码造成影响。

按照这种定义，65％的模块就都是“净模块”。由于只要模块中含有一个“脏函数”该模块就被视为“脏模块”，那么看看“净函数”/“脏函数”的比率或许更有意思。只要函数中发生了一次对不清白的 BIF 的调用，该函数就被视为“脏函数”。在函数级，我们可以看到 92％的函数都是以无副作用的方式编写的。

还应注意，在 113 万行代码中，总共包含有 4090 个“脏函数”，也就是每 1000 行代码包含的脏函数个数不超过 4 个。

脏函数的分布如图 8.1 所示。从结果看，脏函数的分布情况既有值得褒奖的地方，也有亟待改进的地方。好消息是 95％的脏函数都出现在极少数的 1％的模块中，坏消息是有大量的模块中都包含有极少量的脏函数。例如，有 200 个模块

中只有 1 个脏函数，有 156 个模块中包含 2 个脏函数，等等。

这些数据中有一点很有意思，就是我们并没有系统性地为达到代码的“干净程度”而付出努力。因此这种编程的“原始风貌”正好迎合了一种编程风格，即少数模块包含大量的副作用，而大量模块包含极少的副作用。

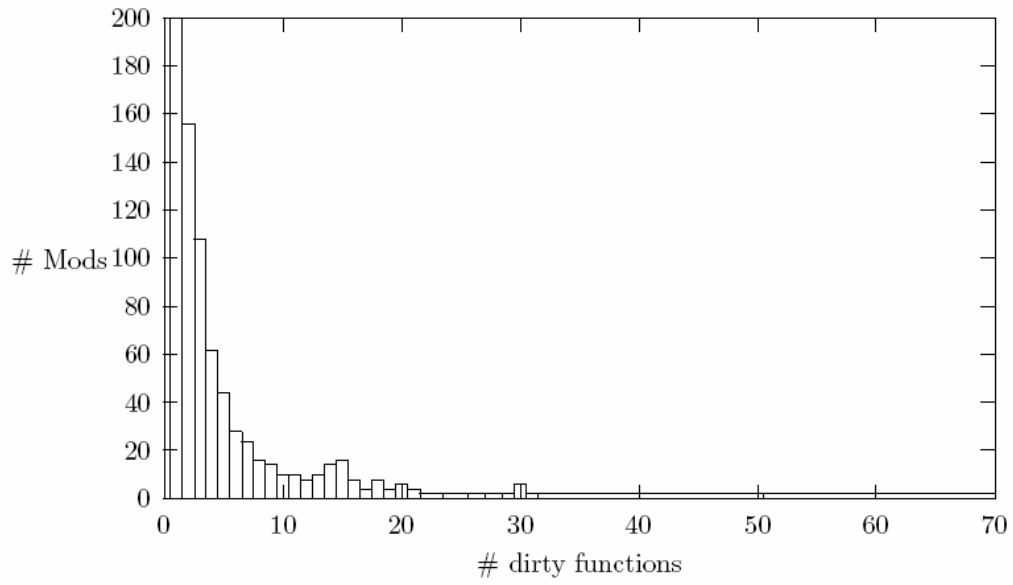


图 8.1：脏函数的分布

Erlang 编程规范也积极支持这种编程风格，意图就是要让更有经验的程序员编写和测试那些包含副作用的代码。基于对 AXD301 的代码的观察，我有一个主意，那就是明确地定义哪些模块是允许包含副作用的，作为一种质量控制的强制要求。

如果我们再深入一点看看对会引入副作用的原语的使用的次数，我们会发现如下一个顺序：

put (1904), apply (1638), send (1496), receive (760), erase (336), process_flag (292), spawn (258), unlink (200), register (154), spawn_link (126), link (106), unregister (38), open_port (20), demonitor (14), processes (14), yield (12), halt (10), registered (6), spawn_opt (4), port_command (4), trace (4), cancel_timer (2), monitor_node (2).

用得最广泛的原语就是 put，一共被使用了 1904 次。

从这个使用统计数据我们可以看出，我们的 Erlang 原语“黑名单”上的许多原语从来就没有被用到过。使用得最广泛的会带来有副作用的原语就是 put——而是否真的会产生副作用则取决于对它的使用方法。其中一种广泛的用法就是用 put 来断言一个用于调试目的的进程的一个全局特性，而这种用法基本上是安全的，虽然没有自动化的分析程序能够证明这一事实。

真正有危险的副作用是那些改变应用程序的并发结构的原语，因此那些使用了 link、unlink、spawn、spawn_link 等原语的模块要仔细地检查一下。

更危险的代码是调用了 halt 或 processes 原语的代码——我确信这样的代码一定被非常小心地检查过了。

8.3.1 系统结构

AXD301 的代码是用 OTP 监督树来组织的，因此可以根据这些树的形状推断出 AXD301 的整个代码组织结构为一棵大的监督树。这棵监督树的内部节点本身是一些监督者节点，而叶子节点则都是 OTP 的 behaviour 的实例或专门的应用相关的进程。

AXD301 系统的监督树有 141 个节点，使用了 OTP 的 behaviour 的 191 个实例。每种 behaviour 的实例的个数如下：

```
gen_server (122), gen_event (36), supervisor (20), gen_fsm (10),
application (6).
```

可见 gen_server 的使用最多，一共有 122 个通用服务器的实例，而 gen_event 的实例个数居于其次。有意思的一点是，实际上所需要的共同行为是相当少的。客户-服务器抽象（即 gen_server）是如此有用，以至于整个系统中的一般对象有 63％是客户-服务器行为的实例。

在 OTP 库集中，一个监督者通过调用它所监督的进程的所谓 child_spec（子进程说明）信息中的一个函数来开启一个应用。“子进程说明”（child specification）中包含有许多信息（译注：参见 6.5.2 小节中的例子），其中{Mod, Func, Args}元

组就使用来标明如何开启一个被监督的进程的。

开启一个被监督进程的方法是完全通用的，因为监督者可以根据任意一个函数开启一个被监督者。在 AXD301 的案例中，这种方法的通用性并没有完全体现出来，而是在所有的监督层次结构中只用了三种开启被监督进程的方法中的一种。在这三种方法中，有一种完全占据统治地位，应用到了除 3 棵监督树中的其他所有监督树中。

AXD301 的架构师定义了一个主监督者，它可以用许多标准化的方式进行参数化。AXD301 的所有监督者也被打包成为一个常见的 OTP 应用，它们的行为在一个所谓的.app 文件[34]中进行描述。分析一下 AXD301 的所有 app 文件，可以给我们一个关于 AXD301 的软件静态结构的一个很好的总体认识。

AXD301 的软件一共有 141 个.app 文件。这 141 个文件展示了 11 棵独立的监督树。这些监督树中的大多数是非常平的，并没有很复杂的结构。

展示这种结构的一个简单的方法就是用简单的 ASCII 码显示来画出该结构的树型图。例如，这里就是上面所说的 11 棵顶层树中的处理“Standby”业务的一棵树的树型图。

```
|-- chStandby Standby top application
| |-- stbAts Standby parts of ATS Subsystem
| | |-- aini_sb Protocol termination of AINI, ...
| | |-- cc_sb Call Control.Standby role
| | |-- iisp_sb Protocol termination of IISP ...
| | |-- mdisp_sb Message Dispatcher, standby role
| | |-- pch_sb Permanent Connection Handling ...
| | |-- pnni_sb Protocol termination of PNNI ...
| | |-- reh_sb Standby application for REH
| | |-- saal_sb SAAL, standby application.
| | |-- sbm_sb Standby Manager, start standby role
| | |-- spvc_sb Soft Permanent Connection ...
| | |-- uni_sb Protocol termination of UNI, ...
| |-- stbSws Standby applications - SWS
| | |-- cecpSb Circuit Emulation for AXD301, ...
| | |-- cnh_sb Connection Handling on standby node
| | |-- tecSb Tone & Echo for AXD301, SWS, ...
```

正如我们看到的一样，这棵树的结构是相当简单的，平而浅。树中只有 2

个一级子节点，而它们下面的监督者的结构完全是扁平的。

需要注意的是，这样显示的数据只是显示了监督者节点的组织。作为树中叶子节点的子节点的实际的进程并没有显示出来，监督类型（即“与”型监督还是“或”型监督）也没有显示出来。

为什么把树组织得扁平而不是多层？其原因反映了从 AXD 软件中获取的一条经验，简单来说，就是“*多层的（cascading）重启经常失效*”。

AXD301 的首席软件架构师 Wiger[69]发现，用同样的参数去重启一个出错的进程经常是可以的，但是当这个简单的重启过程失败后，多层的重启（即重启它的上一层）却往往不奏效。

很有意思，早在 1985 年的时候，Gray 就观察到了大多数的硬件故障都是瞬时的，可以通过重新初始化硬件的状态然后重试操作来纠正。他于是猜测这一点对于软件来说也是一样：

> *我猜想软件中也存在类似的现象——很多产品故障都是很微妙的。如果程序状态被重新初始化一下，然后重试先前失败的操作，这个操作在第二次就往往不再失败了。*——[38]

本论文中提出的出错处理模型的一般化方法——即在发生故障时采用一个尽量简单的策略——只是被部分采用了。之所以是部分采用，与其说是有意设计，不如说是偶然凑成——OTP 库集本身提供的与文件系统的接口和以及与 socket 这样系统级服务的接口在编写之初就十分注意当发生故障的时候要保护系统的完整性。所以，如果一个文件或 socket 的控制进程因任何原因而终止时，这个文件或 socket 就会被自动关闭掉。

由 OTP 库服务提供的保护级别将自动提供“更简单的服务级别”（simpler level of service），这也正是我们的容错处理模型的用意。

8.3.2 故障被修复的证据

在接下来的几节中，我将举出错误恢复机制确实如我们设计的那样发生作用的证据。这些证据基于对 Ericsson 的故障报告数据库中包含的一些条目的分析。

我将不假修饰地引用故障报告数据库中的数据条目，只是删除了一些无关紧要的细节。

8.3.3 故障报告 HD90439

2003 年 5 月 14 日的故障报告 HD90439 包含如下一些信息：

```
1. Description

Heading: CRASH REPORT - Performance measurements on ET2 issue
Priority: C:   3 M, Minor fault or opinion: no traffic disturbance
Status: FI:   Finish
Hot TR: NO

...

2. Observation   Top of page

EFFECT:
CRASH REPORT - Performance measurements on ET2 issue

DESCRIPTION:
Node: AXD305 R7D PP6
Customer: ***************

=CRASH REPORT==== 14-May-2003::14:05:00 ===
  crasher:
    pid: <0.5605.0>
    registered_name: []
    error_info: {function_clause,[{etcpPrm,get_hwm_base,[ne_cv_l_ga,et2]},
                                  {prmMibExt,create_mep,3},
                                  {prmMibExt,get_counter_values,4},
                                  {perfCollect,collect_group,2},
                                  {proc_lib,init_p,5}]}
    initial_call: {perfCollect,collect_generic,
                               [{observed_object_group,
                                    {groupCb,prmMibExt},
                                    1504,
                                    [127],
                                    undefined},
                                63220104300]}
```

```
35      ancestors: [perfServer,perfSuper,omsSuper,omAxd301Super,<0.4396.0>]
36      messages: []
37      links: [<0.4523.0>]
38      dictionary: [{{eqm_mi_apply,{em,70},if_type_to_sublayer,1},
39                    {intfIfDbase,if_type_to_sublayer_int}}]
40      trap_exit: false
41      status: running
42      heap_size: 121393
43      stack_size: 23
44      reductions: 1332403
45    neighbours:
46
47  MEASURES:
48  Performance measurements were turned off and the crash report stopped
49  occurring
50
51  ...
52
53  4. Answer
54
55  ...
56  P R O B L E M & C O N S E Q U E N C E
57  ========================================================
58
59  A combination of the wildcard implementation together with a DS1
60  measurement can cause the described crash. The effect is that the
61  measurement fails.
62
63
64  S O L U T I O N
65  ========================================================
66
67  The fault has been detected, but it will not be released in R7D unless
68  it is important for a customer.
69
70  The wildcard implementation is greatly improved in R8B where this problem
71  does not exist. In R7D, we recommend to specify the PDH interfaces to be
72  included
73  in the DS1 measurement.
```

编号为 90439 的崩溃报告是非常典型的，它说明了硬件发生的位置情况、修复情况和系统恢复正常使用的情况。崩溃报告的第 23－27 行为错误日志，它包含有如下信息：

```
{function_clause,
    [{etcpPrm,get_hwm_base,[ne_cv_l_ga,et2]},
    {prmMibExt,create_mep,3},
    {prmMibExt,get_counter_values,4},
    {perfCollect,collect_group,2},
    {proc_lib,init_p,5}]}
```

当调用 etcpPrm:get_hw_base(ne_cv_l_ga, et2)时，函数调用因模式匹配失败而错误。有意思的是，这个错误发生在 2003 年 5 月 14 日，而我所评估的系统版本是 2001 年 12 月 5 日的，所以我推断这个错误可能在我评估的版本的代码中就已经存在了。为了满足我自己的好奇心，我检查了一下 etcpPrm 模块的代码，我看

到的代码是这样的：

```
get_hwm_base(locd_cnt, et155) ->
    ?et155dpPmFm_MEPID_Frh_LOCDEvt;
... 386 lines omitted ...
get_hwm_base(fe_pdh2_uat_tr, et2x155ce) ->
    ?et2x155cedpPmFm_MEPID_Frh_FE_E1_UAT_Tr.
```

确实没有与调用参数相匹配的模式，所以如果用错误日志中的参数来调用该函数当然会引起错误。

我可以清晰地定位出这个错误，并且理解为什么程序会发生故障，虽然我对于这个错误意味着什么一无所知。

必须要提出表扬的是写这部分代码的程序员并没有用防御式编程的方法来写这段程序，即没有像这样写：

```
get_hwm_base(locd_cnt, et155) ->
    ?et155dpPmFm_MEPID_Frh_LOCDEvt;
... 386 lines omitted ...
get_hwm_base(fe_pdh2_uat_tr, et2x155ce) ->
    ?et2x155cedpPmFm_MEPID_Frh_FE_E1_UAT_Tr;
get_hw_base(X, Y) ->
    exit(....).
```

他们确实严格按照了我在第 4.4 节中推荐的风格来编写代码。回想一下，按照这种方式编写代码的动机在于防御式编程是不必要的，因为 Erlang 编译器会自动加上一些为了便于程序调试的附加信息。

8.3.4 故障报告 HD29758

编号为 HD29758 的崩溃报告更有意思。看看日志和研究过这个问题的工程师的后续的注释，我们可以看出运行时发生了一个以前曾经发生过的错误。虽然错误发生了，但是它并没有影响系统的流量，所以工程师认为不需要修改。

如下是从故障报告中摘取的一段：

```
2   T R O U B L E  D E S C R I P T I O N

R7D NIT test case 3.4.1.2, takeover of OM process by CH CP.
Traffic continues to run successfully,
but we get several ERROR REPORTS in the
CP being blocked.

 Here's an example, see enclosure for erlang log:

  =ERROR REPORT==== 5-Apr-2002::09:03:55 ===
      error_info: {{case_clause,[]},
                   [{mdispGenServ,from_plc,4},
                    {mdispGenServ,handle_info,2},
                    {gen_server,handle_msg,6},
                    {proc_lib,init_p,5}]}
      msg_info_Tag: from_plc
      msg_info_MFA: {mdispGenServ,from_plc,
                                  [old_hc,
                                   {hcid,
                                       {ncs,mlgCmHcc,{97575,0}},
                                       msgQueue,
                                       undefined}]}
      msg_info_PlcResult: true
      node: 'axd301@cp1-1'
      proc_info: [{pid,{<0.20804.4>,"MDISP Server"}},
                  {message_queue_len,0},
                  {dictionary,[{mdispPerfPid,<0.20805.4>},
                               {'$ancestors',
... many lines omitted ...

4.2 Answer Text

The fault has been solved in version R8B of block
MDISP. No solution is planned in earlier versions
for mainly two reasons:

1) The fault has not yet given any obvious negative
effects on traffic handling.

...
```

第 10－28 行显示确实发生了一个错误。第 4 行和第 37－38 行显示尽管发生了错误，系统还是能够“*对流量的处理没有明显的负面影响*”地运行。

这个错误是相当细微的，它显示了由于某个特定的执行路径，错误才会出现。系统检测到了错误，并且从错误中恢复过来了，因此，仅就这个特定的错误来说，

系统在面对错误的时候是以一种合理的方式在运转的。

8.3.5 OTP 结构的不足

一个有意思的地方是 Erlang 编程模型并不太适合呼叫发起(call setup)和呼叫终止(call termination)的处理，而这一处理是 AXD301 的软件中需要用到的。

在关于 COP 的设计哲学的章节，我曾说过要将问题的结构 1:1 地映射到软件的架构上。但是在 AXD301 的软件的一个重要部分中，这种映射是做不到的，这一部分就是对呼叫的发起和终止的处理。

为了理解这一点，我必须深入 AXD301 交换机提供的最重要的业务的一些细节。AXD301 是一款交换系统，因此它的职责是维护巨大数量的连接（connection）。

在任何时间，系统中的任何一个模块都在处理这巨大数量的虚拟通道（virtual channel）。每个通道代表了一路连接。典型情况下，一个节点可能会处理多达 50000 个连接。每个连接都可能处于如下三种状态之一：

1. *发起*——在这种状态下，一个新的连接正在被建立。这时候有大量的信令交互。
2. *已连接*——在这种状态下，连接已经被建立起来了。这时候信令很少，只是一些监控连接存在的信令。
3. *终止*——在这种状态下，一个连接正在被终结。这时候连接的两个端点之间有一些信令。

连接的发起和终止阶段是非常快的，一般只需要几个毫秒的时间。连接的终止比发起又要简单一些。在已连接阶段只有一些监督信令被处理，已连接阶段持续的时间要比连接发起和连接终止阶段长过许多数量级。已连接状态持续的时间可能从几秒种到几个小时，甚至几年。

在任何时间，每个节点上都有可能有多达 50000 个连接，其中绝大多数都处于已连接状态。系统的设计处理能力是每秒种处理 120 路呼叫（120 calls/second）——这个数字是指同时处于呼叫建立和呼叫终止阶段的连接数，不是同时处于呼

叫已建立阶段的连接数。

如果要为这个问题建立一个自然的并发模型，大约每路呼叫发起/连接需要 6 个并发进程。如果每路连接的发起都有 6 个进程，并且每路已连接的呼叫都需要 2 个进程，那么就需要几十万个 Erlang 进程。而这些进程中的大多数没有做什么事情，被长时间地挂起着（就是那些监督已连接呼叫的进程）——有许多原因使得我们不能这么做。

首先，进程会消耗空间，即使它什么也不做。其次，已连接阶段的所有相关状态都需要跨物理硬件边界进行备份。这是为了在出现硬件故障的时候能够提供连续的服务。

AXD301 使用了下面的一些策略来使得已连接的呼叫的处理进程的数量达到最小化：

1. 在呼叫发起阶段，创建 6 个进程来处理每一路呼叫。
2. 如果呼叫发起已成功，该呼叫的所有相关信息都被简化为一个“呼叫记录”（call record），用以描述这个呼叫。在呼叫发起阶段使用过的 6 个进程被摧毁，并且描述该呼叫的呼叫记录被本地保存起来。最后，一条包含有呼叫记录的异步消息被发往被节点的备份节点。
3. 在呼叫终止阶段，终止本次呼叫所需要的所有进程结构都通过呼叫记录中的数据来创建和初始化。进行完呼叫终止处理后，所有的相关进程都被摧毁。

因为呼叫发起进程很容易理解，呼叫发起阶段每个进程有内存需求也就很容易理解。每个呼叫发起进程所需的栈和堆空间的最大值是经过多方测量定制好了的。使用这个事先计算好的数据，就可以用足够大的原始栈和堆空间来初始化这 6 个呼叫发起进程，从而避免了呼叫发起阶段进行垃圾回收。

在呼叫发起阶段，并不尝试将呼叫状态进行跨物理边界进程备份，因此如果此时系统发生崩溃，呼叫就会被丢失。这时候用户会意识到呼叫失败了，只需要重新再试一遍。当发生硬件故障的时候，用户的重试会被直接定向到一个新的硬件模块，这时应该会成功的。因为呼叫发起是如此之快，所以这几乎不是问题。

在呼叫发起阶段完成后，所有的信息都被简化为一条“呼叫记录”（大概是每路呼叫 1K 字节），呼叫发起阶段所用到的 6 个进程都将被摧毁。一条包含有呼叫记录的异步消息被发送到备用处理器。硬件单元通常是成对配备的。对于一对设备（A,B）来说，机器 A 被视为机器 B 的备份机，而机器 B 也被视为机器 A 的备份机。

来自具体用户的信令被一个硬件分发单元作出裁决，硬件分发单元将来自某个具体用户的所有信令都发往该呼叫的“主”单元。如果主单元发生故障而该呼叫正处于已连接状态，那么裁决器将把信令定向到它的备份单元。

呼叫的终止，或对呼叫的修改，都与呼叫发起的过程相反。当系统检测到对某个呼叫的操作时，呼叫记录被取出，所有操作该呼叫所需要的进程结构都将根据呼叫记录中的数据被创建和初始化。之后呼叫处理的过程与呼叫发起是一样的。

这种模型将进程的个数最小化为执行某个特定操作所需要的进程的动态集合。当应用到达某一时刻，此时对数据的活动处理非常少，那么相应的进程将会被销毁，重新创建这些进程结构所需的所有相关数据都被保存到一个数据库中，并且被异步备份到一个备份机。

这种方式的优点是有一个灵活的（flexible）进程结构，这个进程结构就是呼叫发起和终止的复杂阶段所需要的——但是当进程变到不活动状态时，就需要删除进程，并为在持久存储设备中存储他们的状态数据而申请存储空间。

状态数据被异步备份这种做法在不危害系统完整性的条件下提升了系统的吞吐量。

这种解决方案完美地适应了 Erlang 的进程模型。因为进程都是轻量的，我们可以“根据需要”（on-demand）创建和销毁进程。重量级进程的系统也可能使用相似的解决方案，但是进程执行完后就不是销毁这么简单了，而是需要回收，系统需要维护一个进程池。

在 AXD301 的软件开发的时候，对于允许同时运行的 Erlang 进程的总数有一个上限的限制。这个数目尽管很大，但是也不足以大到允许系统中几十万个进

程同时存在。系统的后来的版本允许存在几十万个 Erlang 进程，虽然也不允许达到上百万。

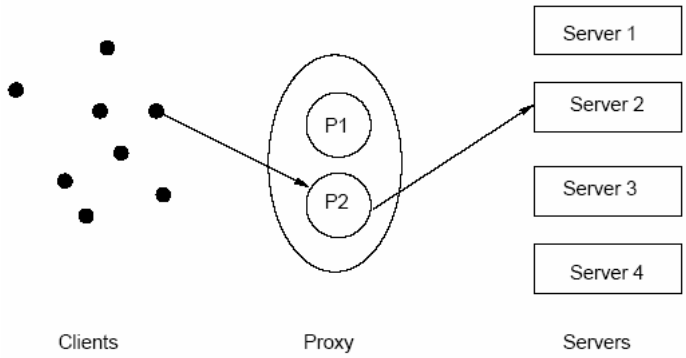
虽然系统的设计者最初并不必担心系统中进程的总数，但是他们应该能够粗略地估计一下系统，并且对于一个应用的一次操作过程需要用到多少个进程要有一个大致的了解。

对于很长时间存在的进程，在进程不再活动的时候把进程状态存储到一个数据库种是一个有吸引力的选择。这种方法有一个额外的优点，就是将状态数据备份到持久存储设备中可以使得软件变得更能容错。

8.4 用 Erlang 开发的小产品

第二个案例我们将来研究一下 Bluetail AB 公司做的 2 款产品。第一个就是“Bluetail Mail Robustifier”（Bluetail 邮件加固器，BMR），关于这款产品在参考文献[11]中有所描述。

8.4.1 Bluetail Mail Robustifier （邮件加固器）



Bluetail Mail Robustifier 是一款设计用来提高已有的 e-mail 服务的可靠性的软件。BMR 被设计成一个置于客户和许多 e-mail 服务器之间的一个代理服务器（proxy），客户要能够享受到 e-mail 服务。

客户所要关心的就是所有的服务器都有相同的 IP 地址（即代理服务器的地址）——在代理服务器内部至少有两台物理机器（为了容错嘛），代理服务器截取用户消息并把它们转发到后端（back-end）服务器。后端服务器自身运行着一些标准化的邮件服务程序。BMR 支持 3 种邮件协议：SMTP[57]、POP3[52]和

IMAP4[27]。

BMR 是作为一套提供完整解决方案的系统交付给客户的。BMR 的需求[11]如下：

1. *十年中当机时间不能超过几分钟。*
2. *如果有邮件服务器发生故障，一定要有其他的邮件服务器在最短时间内顶上，用户不应该注意到服务器的故障。*
3. *应该可以远程管理起该系统。我们希望能够添加或移除邮件服务器，或在不影响服务质量的前提下让某个服务器停止。*
4. *应该可以在不停止 BMR 系统运行的前提下升级 BMR 系统的软件。*
5. *一旦 BMR 的软件发生错误，要能够在不停止系统运行的前提下回退到之前的软件版本。*
6. *系统要能够与现有的邮件系统一起工作，并能够提高现有系统的性能。*
7. *系统应该能够在现有的硬件系统和现有的操作系统上运行。*
8. *系统应该至少与常规的命令式语言实现的系统效率相当。*
9. *系统应该有一个常规的 Unix 式的命令行界面，和常规的 Unix 式的帮助文档。*
10. *系统应该有一个 GUI（图形用户界面）。*

第 1、3、4 点是典型的软实时系统的需求，特别是第 4 点，是这类系统的一个典型需求，很少会在那些只需连续运行较短时间的系统中发现这一需求。第 5 点与第 4 点是相关的，我们将尽最大的可能让软件能够完全自动运行，人为干预最少化。事实上，BMR 能够远程管理，并且能够自动远程地进行软件的升、降级这一点对于产品的销售是非常重要的。确实，购买 BMR 这款软件的主要目的也就是要消除手动监控和维护 e-mail 服务器的负担。

BMR 系统的编写用了 108 个 Erlang 模块，一共有 35285 行 Erlang 代码。该项目开始后，从一无所有到交付给客户第 1 个版本，只用了 6 个月的时间。BMR

自 1999 年起一直伴随 Swedish Telenordia ISP 公司连续运行至今，每年都要处理几百万份电子邮件。

BMR 做了一套自己的发布管理系统，是对 OTP 系统中的通用发布包的一个扩展。BMR 系统是一个天生的分布式系统。我们希望分布式系统的软件升级有一个“事务级”（transaction）的语义，即要么系统的所有节点整体进行软件升级，要么升级失败任何节点上的软件都没有改变。

在 BMR 中，整个系统的软件同时有两个版本共存，一个老版本和一个新版本。增加一个新版本软件的时候，当前版本就关成了老版本，而新添加的版本变成了新版本。

为了做到这一点，所有的 BMR 软件升级包都是以一种可逆（reversible）的方式编写的。即不但可以将老版本软件动态地升级到新版本，而且可以从一个新版本变回到老版本。

升级所有节点的软件是按四个步骤完成的。

1. 在第一阶段，新版本的软件被分发到每个节点——这通常会成功。
2. 在第二阶段，所有节点上的软件都从老版本变到新版本。如果有任何节点上的转换失败，则所有运行新版本软件的节点都退回到运行老版本的软件。
3. 在第三阶段，系统中所有节点都运行着新版本的软件，但是如果发生任何错误，则所有节点都回退去运行老版本的软件。这时候系统尚未确认只运行新版本软件。
4. 在新系统已经成功运行了一段很长时间以后，操作人员可以“确认”（commit）软件的改变。系统确认（即第四阶段）将改变系统的行为。在确认以后如果再发生错误，那么系统将用新版本软件重新启动，而不是回退到老版本。

有趣的是，几乎同样的机制被用到了 NASA 开发的 X2000 系统[63]上，该系统是作为深太空应用的，他们的软件也需要在不停止系统运行的前提下实现升

级。

需要补充一下的是，BMR 升级系统必须要考虑到在软件正在进行升级的时候分布式系统中有节点处于“掉线”（out-of-service）状态的情况。在这种情况下，当该节点重新加入到系统中时，它将学习一下它离线期间系统的变化，之后进行任何必要的软件升级。

8.4.2 Alteon SSL accelerator（SSL 加速器）

Alteon的SSL[3] accelerator是Bluetail AB被Alteon Web Systems公司收购之后开发的第 1 款产品。SSL accelerator是一个硬件设备，包含有专门用于对加密通信进行加速的硬件。SSL accelerator由Nortel Networks进行市场销售。SSL accelerator的控制系统是用Erlang编写的。

> *据 Infonetics Research 报道，Alteon SSL Accelerator 是 SSL 加速器设备市场的领导者。NortelNetworks 的 SSL accelerator 比其他供应商的 SSL 加速器卖得都要多，正是凭借了其创新性的新应用和诸如后端加密（back-end encryption）、综合负载均衡(integrated load balancing)、会话持续(session persistence)、应用地址转化(application address translation)、第 7 层过滤(Layer 7 filtering)、可靠的全局服务器负载均衡（secure global server load balancing, GSLB）等等新特性。在赢得了所有分类评估之后，Network Computing 杂志称 Nortel Networks 公司的 Alteon SSL Accelerator 为“一代霸主”（King of the Hill），因为他们在最新的 SSL 加速器市场的激烈竞争（bake-off）中，用他们的业界领先的性能（performance）、特色（features）、易管理性(manageability)等突出特征赢得了胜利。*——[53]

SSL accelerator 是作为硬件产品推出的。有趣的是它的推出周期是非常短的（不到 1 年），却很快成为了市场的领头羊，还赢得了 Network Computing 杂志授予的所有分类测试的“测试头名”（best in test）奖。

SSL accelerator 的软件架构是从曾在 Bluetail Mail Robustifier 中使用过的通用架构演变而来的。

[3] Secure Socket Layer，安全套接字层。

8.4.3 代码的量化特性

遗憾的是，Nortel Networks 公司不允许我在任何细节方面分析他们的源代码，所以我只能报告一下他们的代码的一个很粗略的特性。下面的数据来自所有的 Bluetail/Alteon 公司的 Erlang 产品，并不区分具体属于哪一款。

总的 Erlang 模块的数量	253
“净”模块数	122
“脏”模块数	131
代码行数	74440
总的 Erlang 函数个数	6876
“净”函数的个数	6266
“脏”函数的个数	610
“脏”函数个数 / 代码行数之比率	0.82%

这些产品都广泛地使用了 OTP 的 behaviour，一共由 103 个 behaviour 的实例。gen_server 再次首屈一指。各个 behaviour 被使用的次数如下：

```
gen_server (56), supervisor (19), application (15), gen_event (9),
rpc_server (2), gen_fsm (1), supervisor_bridge (1).
```

对于这些数据不必再多说了，除了有一点，就是乍看上去 AXD 项目中的函数要长一些，而使用的 behaviour 比 Bluetail/Alteon 的项目中用到的要少一些。

我采访过编写 SSL accelerator 的程序员们，他们告诉我在实际情况下，他们的软件架构工作得很好，还没有发生过由软件引起的不可恢复的错误，但是他们没有保存记录来证实他们的观察。

他们还说，他们的产品升级机制已经比 OTP 中开发的机制向前发展了很多。每个产品版本升级都被设计成完全可回退的。也就是说，当他们打算从版本 N 升级到版本 N+1 时，他们不但要编写处理从版本 N 升级到版本 N+1 的处理代码，还要编写从版本 N+1 回退到版本 N 的处理代码。

他们严格遵守这一原则，因此他们可以把系统的当前版本“回退”（roll-back）到很久以前的早期版本。出于商业方面的原因，Nortel 不希望公开这一处理过程

的细节的任何详细信息。

8.4 讨论

其实在我开展这些案例研究之前，就对用哪些参数来衡量我们的技术有一个非常清楚的想法。而对一项技术的最根本的检验就是问一下这项技术的使用者们下面的一个问题：

“它确有实效吗？”

如果你拿这个问题去问 Ericsson 或 Nortel 的人，他们将会十分惊奇你为什么会这么问，还会告诉你：

“当然有效！”

我原本希望他们给我提供一些书面证据证明系统确实像我在本文中描述的那样工作起来了。我本来期待从系统操作记录文件中找到一些物证据明代码升级确实可行，系统确实崩溃过而且恢复了，而且系统已经不中断地运行了几千小时。

我采访的人们很高兴地告诉我，事实确实是那样的，代码升级之类的功能确实像期望一样地工作得很好。但是他们不能给我提供任何正面的证据来证明这一点（也就是系统操作记录文件中没有记录代码升级确曾发生过）。

之所以没有书面证据的原因是系统运行的时候并没有要求把“代码升级已经发生”之类的事情写入系统操作记录文件中，因此没有留下持久的物证。不过有意思的是也没有反证证明代码升级过程失败过。我想如果升级失败发生了，那么一定会有许多故障报告之类的材料记录下这个事实。

同时我也没有正式收集找到关于系统已经长时间稳定运行的证据。对于 Ericsson 的 AXD301 系统来说，关于系统长期稳定性的唯一的信息来自一个演讲稿，该演讲稿出示了一些数字，声称一个大客户已经在 11 个节点上运行了 AXD301，可靠性是 99.9999999%，然而也没有说明这个数字是怎么获得的。

在 BMR 和 SSL accelerator 的案例中，也没有很强的证据证明系统已经运行了很长很长的时间。这是因为这样的消息没有被记录到系统操作记录文件中。

不过 BMR 当然是一个可容错的分布式系统，有许多互连的节点。在这样的系统中，一个单一节点的故障不是什么“大事件”，因为系统从设计之初就是设计得够应付单节点崩溃这样的故障的。

如果单个节点确实崩溃了（这样得事情确实发生过，有一些非正式证据可以说明有这么回事），那么系统的整体性能受到的影响应该是很轻微的，但是这不会带来影响整个系统运转的问题。

我没有听说过整个系统发生故障的事（即所有节点都发生故障）——即使发生这样的事，我也会怀疑这是由于影响到系统的所有机器的重大硬件故障引起的。这类故障机器少见，即使发生了，故障的原因也应该跟软件没有任何关系，而完全是硬件问题。

在分布式系统中，甚至在我们的谈论中，关于故障的理解是需要修正的。我们再谈论整个系统的故障已经没有什么意义了（因为这种事情极少发生了）——我们应该谈论的是对于服务质量降低的衡量。

我们的案例研究中的软件系统是如此可靠，以至于操作这些系统的人倾向于认为这些系统没有错误。但事实不是这样的，软件错误在运行时确实发生过，但是这些错误很快就被纠正过来了，所以没有人曾经注意到错误的发生。为了得到关于长时间稳定性的准确的统计数据，我们就必须记录下系统启动和停止的次数，以此来作为衡量系统的“健康度”的参数。如果在系统级没有收集到任何这样的统计数据，就说明系统的表现是完全“健康”。

至于 AXD301 的代码库的分析，我本来希望能够找到一些编程规范被正确执行了的证据的——例如，我希望看到一个关于“净”代码与“脏”代码的清晰的分界线。我承认，系统的有些部分经常要以一种怀疑的方式来编写（因为效率方面的原因，或者其他方面的原因），但是我非常希望这样的代码应该从代码的主体中清晰地分离出来，并且在这样的代码上进行严格的测试流程。

但是事实不是这样的。绝大部分代码都是干净的，但是脏代码的分布不是一个真正的“阶梯”函数（step function）（也就是没有清晰的分界线来区分“这部分代码不好，要小心对待”以及“这部分代码是好的”）而是一个发散的分布，

即有少数的模块有许多副作用（我不担心这一部分），而更让人担忧的则是那些数量众多的仅含有一两个带副作用原语的模块。

由于对于代码没有一个更深入的理解，我在这里也无法断定这就是问题的根源，也不能说这些具有潜在副作用的调用给系统带来了问题，或者这些调用是有害的。

无论如何，我想表达的东西是清楚的。光靠编程规范并不足以让我们的系统都以一种特别的方式来编写。如果有人希望把代码强制性地分为干净的代码和肮脏的代码，那么就一定要辅以工具的支持，还要有一些强制执行的政策的支持。是否真的要这么做是有争议的——也有可能最好的方式是允许程序员们违反编程规范，只是希望他们在违反编程规范的时候知道正在做什么。

对一项技术最终极的检验当然是进行“它是否确有实效”的测试。在 AXD301 和 Nortel SSL accelerator 的案例中，答案毫无疑问是“Yes”。这两款产品都是竞争激烈的产品，但是它们都在各自的市场做了领头羊。

9 API 与协议

在我们编写一个软件模块的时候，我们需要描述如何使用它。有一种做法就是为模块所有的导出函数定义一套编程语言 API。为了做到这一点，我们可以用 3.9 节提到过的类型系统。

定义 API 的方法其实是很普遍的。不同的语言之间，类型符号的细节有所不同，不同的系统之间，底层的语言实现对于类型系统的要求的强制性程度也不一样。如果对类型系统有严格的强制要求，那么这种语言就被称为是“强类型的”（strongly typed），否则它就被称为“弱类型的”（untyped）——这一点经常会引起混淆，因为许多要求进行类型声明的语言它的类型系统是很容易被违反的。Erlang 语言不要求类型声明，但是是“类型安全”（type safe）的，意思是不能以一种会破坏系统的方式违反底层类型系统。

即使我们的语言不是强类型的，但是类型声明可以作为一种有价值的文档，而且可以作为一个动态类型检查器的输入，动态类型检查器能够用来进行运行时类型检查。

不幸的是，只按照惯常的方式写出 API 的对于理解程序的行为是不够的。例如，看下面的代码片断：

```
silly() ->
    {ok, H} = file:open("foo.dat", read),
    file:close(H),
    file:read_line(H).
```

按照类型系统的要求和 3.9 节的例子中给出的 API，这段程序是完全合法的。但是它明显是完全没有意义的，因为我们不可能期望从一个已经关闭了的文件中读取东西。

为了改正上面的问题，我们可以添加一个额外的状态参数。辅以一种相当明了的符号，关于文件操作的 API 可以这样写：

```
+type start x file:open(fileName(), read | write) ->
```

```
        {ok, fileHandle()} x ready
        | {error, string()} x stop.
+type ready x file:read_line(fileHandle()) ->
        {ok, string()} x ready
        | eof x atEof.
+type atEof | ready x file:close(fileHandle()) ->
        true x stop.
+type atEof | ready x file:rewind(fileHandle()) ->
        true x ready
```

这种 API 模型用了四种状态变量：start, ready, atEof 和 stop。状态 start 表示文件还没有被打开。状态 ready 表示文件已经准备好被读取，atEof 表示到了文件的结尾。文件操作总是以 start 状态开始，而以 stop 状态终止。

现在 API 就可以这么解释了，例如，当文件处于状态 ready 是，进行 file:read_line 的函数操作是合法的。它要么返回一个字符串，这时候它仍然处于 ready 状态；或者它返回 eof，此时它处于 atEof 状态。

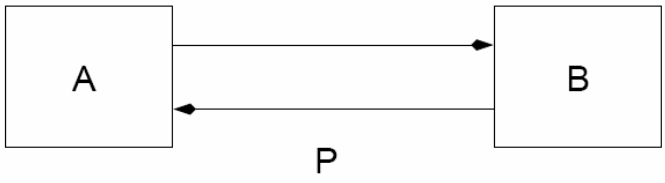
在 atEof 状态的时候，我们可以关闭文件或回倒（rewind）文件，所有其他的操作都是非法的。如果我们选择回倒文件，那么文件将重新回到 ready 状态，这时候 read_line 操作就又变得合法了。

为 API 增加了状态信息，就为我们提供了一种判定一系列操作是否与模块的的设计相吻合的方法。

9.1 协议

可见我们可以标定一套 API 的使用顺序，其实同样的思想也可以应用到协议的定义上。

假设有两个部件使用纯消息传递的方式进行通信，我们要能够在某一个抽象层次说明一下这两个部件之间流动的消息的协议。



两个部件 A 和 B 之间的协议 P 可以用一个非确定的有限状态机（non-deterministic finite state machine）来描述。

假设进程 B 是一个文件服务器，而 A 是一个要使用这个文件服务器的客户程序，进一步假设会话是面向连接的。那么文件服务器应当遵循的协议可以按如下方式来说明：

```
+state start x {open, fileName(), read | write} ->
        {ok, fileHandle()} x ready
        | {error, string()} x stop.
+state ready x {read_line, fileHandle()} ->
        {ok, string()} x ready
        | eof x atEof.
+state ready | atEof x {close, fileHandle()}->
        true x stop.
+state ready | atEof x {rewind, fileHandle()) ->
        true x ready
```

这个协议描述的意思是，如果文件服务器处于 start 状态，那么它就可以接收{open, filename(), read|write}这种类型的消息，文件服务器的响应要么是返回一个{ok, fileHandle()}类型的消息，并迁移到 ready 状态，要么是返回一个{error, string()}的消息，并迁移到 stop 状态。

如果一个协议用类似上面的方式来描述，那么就可能写一个简单的“协议检查”程序，置于进行协议通信的两个进程中间。图 9.1 就展示了在进程 X 和 Y 之间放一个协议检查器 C 的情形。

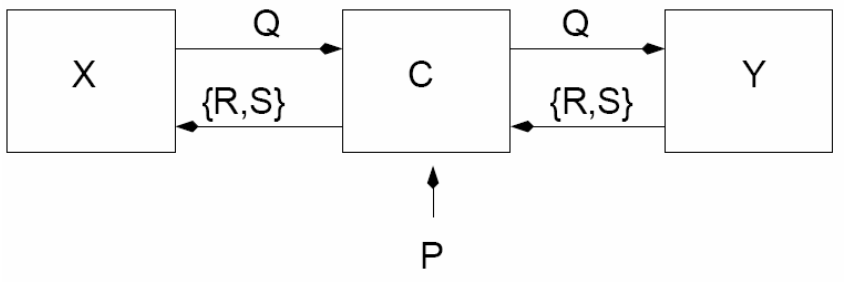


图 9.1：两个进程和一个协议检查器

当 X 向 Y 发送一个消息 Q（Q 是一个询问）时，Y 会以一个响应 R 和一个新状态 S 作为回应。值对{R, S}就可以用协议描述中的规则进行类型检查了。协议检查器 C 位于 X 和 Y 之间，根据协议描述对 X 和 Y 之间来往的所有消息进行检查。

为了检查协议规则，检查器就需要访问服务器的状态，这是因为协议描述可能还有如下的条目：

```
+state Sn x T1 -> T2 x S2 | T2 x S3
```

在这种情况下，只观察返回消息 T2 的类型并不足以区分服务器的下一个状态是 S2 还是 S3。

如果我们回忆一下图 4.3 的简单的通用服务器的例子，我们程序的主循环就可以是这样的：

```
loop(State, Fun) ->
    receive
        {ReplyTo, ReplyAs, Q} ->
            {Reply, State1} = Fun(State, Q),
            Reply ! {ReplyAs, Reply},
            loop(State1, Fun)
    end.
```

这个主循环又可以很容易地改成：

```
loop(State, S, Fun) ->
    receive
```

```
        {ReplyTo, ReplyAs, Q} ->
            {Reply, State1, S1} = Fun(State, S, Q),
            Reply ! {ReplyAs, S1, Reply},
            loop(State1, S1, Fun)
    end.
```

这里 S 和 S1 代表协议描述中的状态变量。注意接口（即协议描述中使用的状态变量的值）的状态与服务器的状态 State 是不同的。

进行了上面的修改后，通用服务器就彻底变成了一种允许安装在客户和服务器之间的动态协议检查器了。

9.2 API 还是协议？

前面我们展示了如何用两种本质上相同的方式来做同样的事情。我们可以在我们的编程语言上强加一个类型系统，或者我们可以在用消息传递方式通信的两个部件之间强加一个契约检查机制。这两个方法中，我更喜欢契约检查器这种方法。

第一个方面的原因跟我们系统的组织方式有关系。在我们的编程模型中，我们采用了独立部件和纯消息传递的方式。每个部件被当作是“黑盒子”，从黑盒子的外面，完全看不到*里面*的计算是*怎么*进行的。*唯*一重要的事情就是黑盒子的行为是否遵循他的协议描述。

在黑盒子的内部，可能因为效率或其他编程方面的原因，我们需要使用一些晦涩的编程方法，甚至违背所有的常识规则和好的编程实践。但是只要系统的外部行为遵守了协议描述，就没有丝毫关系。

只要简单的扩展，协议说明就可以扩展成系统的非功能属性。例如，我们可以向我们的协议描述语言中添加一个时间的概念，那么我们就可以这样来表达：

```
    +type Si x {operation1, Arg1} ->
            value1() x Sj within T1
            | value2() x Sk after T2
```

意思是 operation1 应该在 T1 时间内返回一个 value1()类型的数据结构，或在

T2 时间后返回一个 value2()类型的数据结构。

第二个方面的原因跟我们所做的工作在系统中的位置有关。在一个部件的*外面*放置一个契约检查器决不干涉到该部件本身的构造，并且还给我们的系统增加或删除各种自我测试手段提供了一种灵活的途径。使得我们的系统可以进行运行时检查，并以能以更多的方式进行配置。

9.3 交互部件系统

Erlang 系统如何与外界通信呢？——当我们想要构建一个分布式系统，而 Erlang 只是许多交互部件中的一个时，这个问题就变得很有意思了。在参考文献[35]中我们看到：

> *任何 PLITS 系统都是建立在模块（module）和消息（message）这两种基本构件之上的。模块是一种自包含（self-contained）的实体，就如同 Simula 或 Smalltalk 中的类、SAIL 进程、CLU 模块一样。模块本身用什么编程语言来编码并不重要，我们希望做到不同的机器上的不同模块完全可以用不同的语言来编写。*——[35]

为了做一个交互部件系统，我们必须使得不同部件在许多方面达成一致，我们需要：

- 一种传输格式，和一种从语言实体到传输格式的映射方法。
- 一套类型系统，它建立在传输格式的实体之上。
- 一种基于类型系统的议描述语言。

关于这种传输格式，有一种叫做 UBF（Universal Binary Format 的缩写）的方法，这种方法方法是为参考文献[13]中所讲的快速解析而设计的，该论文的一个稍微修订后的版本见附录 C

9.4 讨论

我还是想回归本论文的核心问题——我们如何在出现错误的时候构建一个可靠的系统？跳出这个系统，把它看作是一组相互通信的黑盒子，是一种有益的

思路。

如果我们可以形式化地描述两个黑盒子之间通信通道所遵循的协议，那么我们就可以利用这一点，作为一种检测和识别错误的手段。我们还可以准确地说出那个部件出了错。

这种架构恰好满足 2.5 节的需求 R1－R4，因此按照我的推理，这种架构是可以用在编写可容错系统上的。

这一点与第 5.1 节的“试图做较容易做到的事”的说法也是相符的。如果一个黑盒子函数的实现发生故障，那么我们可以切换到黑盒子函数的另一个较简单的实现。那么协议检查机制就可以用在元层（meta-level）来决定该使用哪一个实现，一旦发生错误，就选择一个较简单的实现。当部件处于物理上独立的不同机器上时，做到强隔离也就是自然的要求了。

10 总结

从某种意义上来说，一切还远没有完成，但是，你却不得不结束……在这篇论文中，我罗列了许多东西——一种新的编程语言和一种编写可容错系统的方法。对于编写可靠系统来说，Erlang 确实是一个很有用的工具。令人惊奇的是，Erlang 已经被用在了许多应用上，这些应用已经远远超出了 Erlang 当初设计时所针对的问题领域。

10.1 我们已经做到了哪些？

本文中所描述的工作，以及其他人做的相关工作，已经证明了许多不同寻常的东西，即：

- Erlang 及其相关技术是有效的。这是很有意思的一个结果。很多人认为，像 Erlang 这样的语言是不可能全面胜任工业级软件的开发的。然而在 AXD301 和 Nortel 的诸多产品上成功地展示了 Erlang 对于这类产品的开发是一种很合适的语言。不但这些产品成功了，而且在它们各自的激烈竞争的市场中成为领头羊，这一事实非常有意义。
- 用轻量化进程和无共享内存的方式编程在实践中是确实可行的，而且可以用来设计复杂的大容量工业级软件。
- 构建在出现软件错误的时候作出合理的行为的系统是可行的。

10.2 关于将来工作的想法

Erlang 的故事如果没有关于未来的章节将是遗憾的，我对于 Erlang 语言的发展有许多期望的方向。

- *概念完整性*——我们可以发展 Erlang，以进一步强化*一切皆进程*的世界观吗？我们可以使得用 Erlang 编写的系统更整齐（regular）、更容易理解吗？
- *提高内核质量*——第 3.10 节的结论其实是有所文饰的。其实在现有的

大家所知的所有的 Erlang 实现中，还没有一个真正做到了强隔离。系统中一个进程可能会因为向另一个进程发送大量的消息而分配大量的内存，从而影响到另外一个进程；一个恶意的进程可能会因为创建了大量的原子（atom）而使得原子表（atom table）溢出，从而破坏整个系统。当前的 Erlang 实现都没有特别的设计来防止系统受到恶意攻击，提高内核的实现以做到防止类似的攻击是可能的。

- *我们怎么编写部件*——系统由一组相互通信的进程组成的思想自然地引出了用不同语言编写不同部件的思想。但是我们怎么做到这一点呢？

10.2.1 概念完整性

怎么使得我们的系统更容易理解？Erlang 的编程模型是“*一切皆进程*”，我们可以只需稍稍修改一下语言，就能更突出这一点。有了这些改变，大部分（但不是全部）BIF 就都不必要了。只要我们拥有如下假设：

- 一切皆进程。
- 每个进程都有一个名字。
- 消息传递是显式的。

我还要介绍一种新的中缀式远程过程调用运算符（infix remote procedure call operator），叫作“bangbang”（译注：该操作符是将两个“!”连写在一起，读做“bang…bang…”，于是作者形象地把这种操作符叫做 bangbang），写作：

```
A !! B
```

如果 A 是一个 Pid，那么这个写法就是如下这段代码的简写：

```
A ! {self(), B},
receive
    {A, Reply} ->
        Reply
end
```

事实上 A 可以是一个 Pid、原子、字符串，或有 Pid、原子、字符串组成的列表。

求值[A1, A2, A3,…] !! X 将会返回[V1, V2, V3, …]，这里 V1 = A1 !! X，V2 = A2 !! X……，所有的 RPC 都是并行执行的。

10.2.2 文件与 bangbang

我们说一切皆进程，并且所有的进程都有名字，那么文件也是进程。

如果 F 是一个文件，那么 F !! read 就是读取该文件，而 F !!{write, Bin}就是写入该文件。

下面的这段代码是读取一个文件，并把它拷贝到一个新的地方：

```
{ok, Bin} = "/home/joe/abc" !! read,
"/home/joe/def" !! {write, Bin}
```

而下面的例子是并行地读取三个文件：

```
[A, B, C] =
          ["/home/joe/foo",
          "http://www.sics.se/~joe/another_file.html",
          "ftp://www.some.where/pub/joe/a_file"] !! read
```

颇有一些“语法糖衣”的意思。

现在假设我在家里上班，将我的工作文件保留一份拷贝在一个远端计算机上。下面的代码就是将我的文件的本地拷贝与远端机上的拷贝进行比较，如果他们不相同，就更新远端机上的拷贝。

```
L = ["/home/joe/foo",
    Remote= "ftp://www.sics.se/~joe/backup/foo"],
case L !! read of
    {X, X} -> true;
    {{ok,Bin},_} -> Remote !! {write, Bin};
    _ -> error
```

```
end
```

10.2.3 分布式与 bangbang

如果一个进程的名字是一个“erl://Node/Name”形式的字符串，那么操作将会执行在一个远端节点上，而不是本地节点上，即：

```
"erl://Node/Name" !! X
```

其意思是在节点 Node 上执行 Name !! X 操作，将结果返回给用户。那么：

```
{ok,Bin} = "erl://joe@enfield.sics.se/home/joe/abc" !! read,
"/home/joe/def" !! {write, Bin}
```

就是读取一个远端文件，并把它保存在本地计算机上。

10.2.4 产生进程与 bangbang

在 Erlang 系统启动的时候，就已经存在一些“神秘的”进程，它们的名字形如“/dev/…”——这些进程是预先定义好的，做一些神秘的事情。

特别地，“/dev/spawn”是一个产生进程的装置（device），也是一个进程。如果你给“/dev/spawn”发送一个 Erlang 函数 fun，它将返回给你一个 Erlang Pid，因此：

```
Pid1 = "/dev/spawn" !! fun() -> looper() end,
Pid2 = "erl://joe@bingbang.here.org/dev/spawn"
       !! fun() -> ... end
```

将创建两个 Erlang 进程，一个在 bingbang.here.org 上，另一个在本地节点上。

现在我们可以扔掉 Erlang 的 spawn 原语了。它的作用可以由 spawn 装置来替代。

10.2.5 进程命名

既然说到“/dev/spawn”，这里还有一些意义相似的装置：

/dev/spawn	进程产生器
/dev/sdtout	标准输出
/dev/log/spool/errors	错误日志器
/dev/code	代码服务器
/proc/Name	某进程
/Path/To/File	某文件
/dev/dets	dets 表
/dev/ets	est 表
/dev/mnesia	mnesia 表
http://Host/File	某只读文件
erl://Node@Host/..	某远端进程

我们还需要扩展一下进程注册原语，来允许我们将一个字符串注册为一个进程的名字。

10.2.6 利用 bangbang 来编程

我们是以一个简单的例子开始的。图 10.1 是我们的一个老朋友了，一个家庭地址登记程序，让我们再次回顾一下。这次 vshlr4 是用纯 Erlang 函数来写的，

```
-module(vshlr4).

-export([start/0, stop/0, handle_rpc/2, handle_cast/2]).
-import(server1, [start/3, stop/1, rpc/2]).
-import(dict,    [new/0, store/3, find/2]).

start() -> start(vshlr4, fun handle_event/2, new()).

stop() -> stop(vshlr4).

handle_cast({i_am_at, Who, Where}, Dict) ->
    store(Who, Where, Dict).

handle_rpc({find, Who}, Dict) ->
    {find(Who, Dict), Dict}.
```

图 10.1：一个简单的家庭地址登记程序（回顾）

没有尝试隐藏对内部桩函数（stub function）（译注：指 handle_rpc/2, handler_cast/2 这两个回调函数）的访问，相反，我们在接口中暴露了这些函数。

有了 bangbang 的概念之后，第 4.1 节的 shell 交互就会变成下面的样子：

```
1> vshlr4:start().
true
2> vshlr4 !! {find, "joe"}.
error
3> vshlr4 ! {i_am_at, "joe", "sics"}.
ack
4> vshlr4 !! {find, "joe"}.
{ok,"sics"}
```

我们得到了与之前相同的结果，但是这次我们不必将远程过程调用封装到一个桩函数中了，而是把它们暴露给了程序员。

10.3 暴露接口——讨论

我们是否应该暴露接口？我们来想想做事情的两种方式。假设我们有一个远端文件服务器，可以通过一个变量 F1 来访问，还假设文件服务器与客户的通信协议的代码被隐藏在一个 file_server.erl 的模块中，在这个模块中，我们有这样的代码：

```
-module(file_server).
-export([get_file/2]).
...
get_file(Fs, FileName) ->
    Fs ! {self(), {get_file, Name}},
    receive
        {Fs, Reply} ->
            Reply
```

```
end
```

客户代码通过调用 get_file/2 函数从文件服务器获取一个文件。语句

```
File = file_server:get_file(F1, "/home/joe/foo")
```

就是获取一个文件。而实现这一功能的代码被很好地隐藏在 file_server 模块中，用户不知道也不需要知道下面的协议。在《Concurrent Programming Erlang》一书中，我说过隐藏接口的细节是一个好的编程实践：

> *接口函数的目的就是创建一个抽象，来隐藏客户和服务器之间使用的协议的具体细节。一个服务的用户不需要知道用来实现该服务的协议的细节，以及服务器内部所用的数据结构和算法。这样，就使得服务的实现者可以在维护这个用户接口的时候随时自由地修改这些内部细节。*——[5]（第81－82 页）

这种封装的一个不良后果，就是我们不能很容易地将对不同的文件服务器的同时访问进行并行化处理，也不能将对某个文件服务器的请求进行串行化处理。

如果我们希望并行地执行多个 RPC，我们就应该分发所有的请求，然后收集起所有的响应，但是只有接口被暴露出来才能做到这些。

10.4 编写交互部件系统

有意思的是，构建 Internet 应用的一种主要方式也涉及到下面的这些方面：

- 隔离的部件
- 纯消息传递
- 多样化的协议

在Internet应用中，部件是真正物理隔离的，例如，一个瑞典的客户可能会使用一个位于澳大利亚的服务器上的服务，通信方式是没有共享数据的纯消息传递。大概是因为这是系统构建者能够理解和构建一个分布式应用——而这些应用中使用了许多只是在RFC[1]中存在非正式描述的协议——的唯一方式吧。

[1] Request For Comments（征求意见稿）——描述Internet的各个方面的备忘录的集合，包括所有的应用协议。

使用一种标准的语法（例如 XML、lisp 的 S-expression、UBF 项式等）来描述 RFC 将会是一个巨大的进步，并将会使得一种解析器可以用于所有的应用。附录 C 中提议的一种契约检查器也会使编写 Internet 应用变得相当简单，并且还有望使得所编写的 Internet 应用变得更可靠。

有趣的是，Internet 模型即使没有关于故障原因的报告也能工作得很好。如果我们对我们对话的部件抱有一种严重的不信任态度，我们就会觉察到这一点，但是对于一般应用，故障原因报告会使得应用的实现个调试简单许多。

使用契约检查器还可以精确地定位到错误发生的位置，这一点在添加到系统中的交互部件越多时，就显得越重要。

虽然基于相互通信部件的模型广泛地用在分布式应用上，但对于构建单节点的应用，这种方式使用得还很少。为了效率方面的好处，设计者们拒绝使用进程作为他们的软件的保护域，反而对更青睐共享对象模型。

我相信单节点应用的编程模型应该跟分布式应用的编程模型完全一样：不同的部件彼此之间应该用进程来进行保护；要做到部件之间绝对不会相互破坏；部件之间应该使用已定义的协议进行通信，而这些协议由一个契约来说明并强制用契约检查器来检查。

这样，我们才能构建可靠的应用。

附录 A 致谢

Erlang 系统、OTP 库集、还有 Erlang 内置的所有应用都是许多人进行了大量的工作后的集体成果。

致谢部分总是很难写的，因为我不想弄丢任何对系统有贡献的人，也不想出任何差错——我将力图做到准确，但是如果还有任何疏忽或差错，我先在这里表示歉意。

首先，我要感谢当初的“管理者”们，他们使得一切成为可能。我在 Ericsson 时候的领导 Bjarne Däcker 大力支持我们的工作，为了维护我们而积极地斗争——谢谢你，Bjarne。Torbjörn Jonsson（Bjarne 的领导）也为了维护 Bjarne 而积极地斗争，谢谢你，Torbjörn，我从你身上学到了许多东西。Jane Walerud（即 Open Source Erlang 的幕后策划者），Bluetail 的常务懂事，是她教会我如何经营一家小公司。

现在轮到Erlang的开发者的一个大名单了。Erlang团队的最初成员有我、Robert Virding和Mike Williams。当初我编写了Erlang的编译器，Robert编写了库，Mike编写了JAM仿真器。Robert和我写过多个版本的Erlang仿真器，大部分是用Prolog写的，后来Mike用C重写了一遍。两年后，Claes Wikström（我们叫他Klacke）加入了进来，为Erlang增加了分布式的能力，Bogumil Hausman这时候发明了改进版的Erlang虚拟机，即BEAM[1]。

还有 CSLab 的许多成员先后加入到我们的项目中，用 Erlang 编写了许多程序，后来又去做其它的事情去了。Carl Wilhelm Wellin 写了 yecc，Klacke 和 Hans Nilsson 写了 mnesia 和 mnemosyne。Tony Rogvall 和 Klacke 给 Erlang 语言增加了二进制语法，使得 Erlang 可以如此漂亮地支持网络编程。Per Hedeland 的耐心相当好，回答了我所有 Unix 方面的愚蠢的问题，确保了我们的系统能够一直工作的很好，并且他主动解决了 Erlang 仿真器中的一些顽疾。Magnus Fröberg 编写了调试器。Torbjörn Törnkvist 编写了接口生成器，使得 Erlang 可以与 C 交互。

当 Erlang 从 CSLab 搬出来， OTP 诞生的时候，我们的团队扩大了，并且进

[1] Bogdans Erlang Abstract Machine，即Bogdans Erlang虚拟机。

行了改组。Magnus Fröberg，Martin Björklund 和我设计了 OTP 的库结构和 behaviour 的结构，OTP 的 behaviour 库是在我们实验室多年的思想积累之上编写的。Klacke 之前就编写过一个类似于 gen_server 的“通用服务器”，Peter Högfelt 之前也写过另一个通用服务器，是监督树模型的早期的版本。关于进程监督的许多思想来源于 Peter 在移动服务器项目中的工作。

我离开 Ericsson 之后，系统的日常维护和开发交给了新一带的程序员们。Björn Gustavsson 维护着仿真器模块，Lars Thorsén、Kenneth Lundin、Kent Boortz、Raimo Niskanen、Patrik Nyblom、Hans Bolinder、Richard Green、Håkan Mattsson 和 Dan Gudmundsson 维护着 OTP 库集。

而现在呈现在我们的用户面前的 Erlang/OTP 系统，则是从与我们的忠实的用户们的互动中，已经得到了非常显著的提高。

我们的第一个用户群，即用 Eralng 构建第一个正式产品的团队是：Mats Persson、Kerstin Ödling、Peter Högfeld、Åke Rosberg、Håkan Karlsson 和 Håkan Larsson。

在 AXD301 项目中，Ulf Wiger 和 Staffan Blau 在首次用 Erlang 实现运营商级的应用做了许多意义非凡的开创性工作。

无论是 Ericsson 内部的用户，还是 Ericsson 之外的用户，都做得非常棒。英国的“one-2-one”公司（现在是 T-mobile 公司）的 Sean Hinde 简直是“一个人的 Erlang 工厂”。

最后要说，Erlang 的 mailing list 也是我们的灵感和勇气的源泉。今天无论是任何人想要了解 Erlang 的任何事，他们只需要“问一下 Erlang 的 mailing list”，就可以很快得到一个准确而全面的回答。感谢 mailing list 中的所有人，特别是那些素未谋面，确有过很长很长的有意思的 email 来往的人。

最后，感谢我在 SICS 的朋友和同事们，感谢 Seif Haridi 教授，是他审阅了这篇论文。感谢 Per Brand，是他鼓励我写这篇论文的。感谢分布式系统实验室的所有成员们，与他们的讨论给我留下了深刻的印象。

谢谢大家！

附录 B 编程规范和约定

用 Erlang 进行程序开发的编程规范和约定[1]

K Eriksson, M Williams, J Armstrong

1996 年 3 月 13 日

摘要

本文描述了用 Erlang 编写系统的一些编程规范和建议。

注释

本文档只是一个初步文档，并不完整。

对 EBC 的“Base System”的使用在这里并没有做要求，如果要使用“Base System”，那么就应该在设计的很早的阶段就遵守它。这些要求已经写入了 1/10268-AND 10406 Uen“MAP－Start and Error Recovery”一文。

[1] 这里发表的是对Ericsson内部文档（EPK/NP 95:035）重新整理后的版本——原文已经作为Open Source Erlang的配套资料的一部分公开发布。

目录

1 目的

本文罗列了在使用 Erlang 说明和编写软件系统时应该考虑的一些方面。本文并不尝试对与 Erlang 的使用没有太大关系的一般的需求和设计活动给出一个完整的描述。

2 结构和 Erlang 术语

Erlang 系统被划分成**模块**（module）。模块由**函数**（function）和**属性**（attribute）组成。函数要么只能在模块内部可访问，要么**被导出**（exported），即能够被其它模块中的函数调用。属性由“-”开头，被放在模块的开头处。

用 Erlang 编写的系统，所有的*工作*（work）都是由**进程**（process）来完成的。一个进程是一个可以使用其他模块中函数的作业（job）。进程之间通过**发送消息**（sending message）来进行通信，一个进程可以决定准备接收什么消息，而其它的消息就排队等待，直到进程准备好接收它们。

一个进程可以通过与其它进程建立**连接**（link）来监控其它进程的存在。当一个进程终结时，会自动向与之相连的进程发送一个**退出信号**（exit signal），收到一个退出信号以后，一个进程的默认行为就是终结掉自己，并向与之相连的所有进程传播该退出信号。一个进程可以通过**捕获退出信号**（trap exits）来改变这种默认行为，这样可以把发送给一个进程的所有退出信号转换成正常消息。

纯函数（pure function）是一个不管处于什么样的上下文中，只要使用的参数相同，就会返回相同结果的函数。这也是我们平常期望的一个数学函数的行为。一个非纯函数的函数我们说它具有副作用（side effects）。

如果一个函数做了下面的这些事情，一般就会产生副作用：a）发送一个消息；b）接收一个消息；c）调用 exit；d）调用任何会改变一个进程的环境或操作模式的 BIF（例如 get/1、put/2、erase/1、process_flag/1 等等）。

警告：这篇文档中就包含有坏代码的例子。

3 软件工程原则

3.1 从一个模块导出的函数越少越好

模块是 Erlang 的基本代码结构实体。一个模块可以包含有大量的函数，但是只有包含在导出列表中的函数才能够在模块外部被调用。从一个模块的外部看，模块的复杂性决定与它所导出的函数的个数。一个只导出一、两个函数的模块肯定比导出几十个函数的模块更容易理解。

用户希望一个模块的导出函数/非导出函数的比率越低越好，这样他只需要理解导出函数的功能就足够了。

还有，只要模块的外部接口保持不变，模块代码的编写者和维护者可以任意改变模块的内部结构。

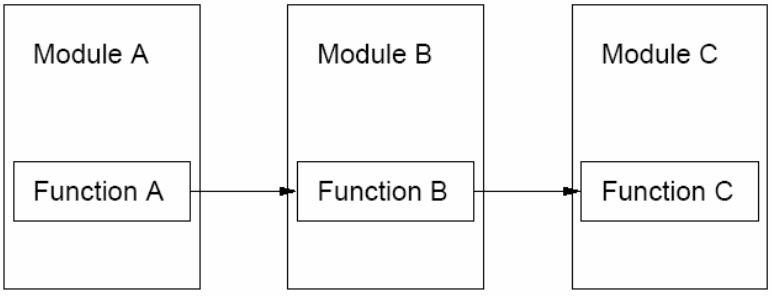
3.2 尽量降低模块间的依赖

一个调用了许多不同模块中的函数的模块要比只调用了很少的几个模块中的函数的模块要难以理解得多。

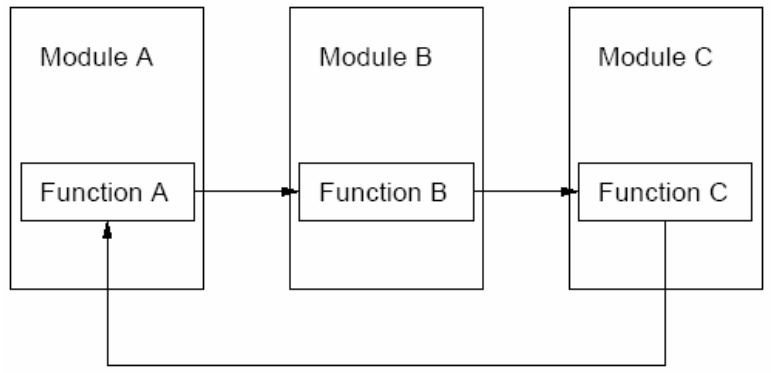
这是因为每次我们修改一个模块的接口时，我们必须要检查所有调用了该模块的地方。降低这种模块间的相互依赖将会简化这些模块的维护工作。

我们可以通过减少一个特定模块调用的不同模块的数量来简化我们的系统结构。

还要注意，模块间的调用依赖关系最好形成的是树型结构而不是环状结构。例如：



而不要是：



3.3 将公用的代码放入库中

公用的代码应该放入库中。这个库应该是相关的函数的集合，应该努力使得库中包含的是同类型的函数。因此，像一个叫 lists 的库只包含对列表操作的函数，这是一个好的决策；而一个叫 lists_and_maths 的库既包含对列表的操作的函数，也包含有数学运算的函数，就不是一个好的决策。

最好的库函数是没有副作用的。含有带副作用的函数的函数的库限制了其可复用性。

3.4 将“复杂的”或“脏的”代码隔离到单独的模块中

一个问题的解决通常需要综合用到净代码和脏代码，那么就把净代码和脏代码放入彼此隔离的模块中。

脏代码是指做了一些脏事情的代码。如：

- 使用进程字典。
- 目的不明地使用了 erlang:process_info/1。
- 做了你很不想做，但是又不得不做的事情。

要致力于将净代码的数量最大化，而将脏代码的数量最小化。要将脏代码隔离出来并清楚地加以注释，否则就应该把这部分代码的所有副作用和要解决的问题文档化清楚。

3.5 不要对调用者如何使用函数调用的结果作出任何假设

不要假设一个函数为什么会被调用，不要假设调用者将用函数的结果做些什么。

例如，假设我们用一组可能非法的参数调用了一个函数。该函数的实现者就不应该假设调用该函数的人期望当参数非法时应该发生什么事。

因此我们不应该这样写代码：

```
do_something(Args) ->
    case check_args(Args) of
        ok ->
            {ok, do_it(Args)};
        {error, What} ->
            String = format_the_error(What),
            %% Don't do this
            io:format("* error:~s¥n", [String]),
            error
    end.
```

而应该这样写代码：

```
do_something(Args) ->
    case check_args(Args) of
        ok ->
            {ok, do_it(Args)};
        {error, What} ->
            {error, What}
    end.

error_report({error, What}) ->
    format_the_error(What).
```

在前一种写法中，错误字符串总是在标准输出上打印出来，而在后一种写法中，错误描述信息被返回给应用程序。后一种写法中应用程序就可以自行决定如何处理错误描述信息。

通过调用 error_report/1，应用程序就可以将错误描述信息转换成可打印的字符串并把它打印出来，如果它要这样做的话。但是这也可能并不是用户期望的行为——任何情况下，对函数的结果作何处理都应该留给调用者来决定。

3.6 抽象出代码的共用样式或行为

无论何时，只要同一样式的代码在代码中两个或更多的地方出现了，就应该尽量把这段代码隔离到一个共用的函数中，然后调用该函数，而不是在两个以上的地方拥有这段代码。要知道，重复的代码维护起来更困难。

如果你在代码中的两个或更多的地方看到相似的代码样式（即几乎相同），那么就值得花点时间看看是否可以稍微改变一下问题，使得不同情况下的问题变得一样，然后写少许额外代码来描述两种情况的不同之处。

请避免“‘拷贝’加‘粘贴’”式的编程，请使用函数！

3.7 自顶向下

用自顶向下的方式来编写你的程序，而不是自底向上（开始就关注细节）。自顶向下是逐步接近细节的实现的一种很好的方法，最后会定义出一些原语级的函数。这样代码就会与细节的表示（representation）独立，因为在进行代码的高层次设计的时候还并不知道细节的表示。

3.8 不要优化代码

起初阶段不要优化你的代码。首先要让它对，然后（如果有必要的话）再让它快（这时候要保证它对）。

3.9 牢记“最小惊诧”原则

系统的响应应该给它的用户带来的是“最小的惊诧”，也就是说，当用户进

行某个操作时，应该能够预测到系统的响应，而不是被系统运行的结果吓一跳。

这一点应该被一致执行，一个不同的模块用一致的方式做事情的一致的系统要比那些每个模块做事情的方式都不一样的系统要容易理解得多。

如果你被一个函数的行为感到惊讶，那么要么是因为你的函数被用来解决错误的问题，要么是因为它的名字不对。

3.10 尽力消灭副作用

Erlang 的许多原语具有副作用。使用了这些原语的函数*不能轻易地被复用*，因为它们给它们的环境造成了永久的变化，你再调用这些函数之前，一定要知道进程的确切状态。

尽可能地编写无副作用的代码。

使纯函数的数量最大化。

将有副作用的函数集中到一起，并且清楚地注明它们的所有的副作用。

只要稍加用心，大多数的代码都是可以写成没有副作用的——这将使得系统容易维护、测试和理解得多。

3.11 不要让私有数据结构从模块中“泄漏”出来

这一点最好用一个简单的例子来说明。我们定义了一个叫 queue 的简单的模块——用来实现队列。

```
-module(queue).
-export([add/2, fetch/1]).

add(Item, Q) ->
    lists:append(Q, [Item]).

fetch([H|T]) ->
    {ok, H, T};
```

```
fetch([]) ->
    empty.
```

以上代码用列表（list）来实现队列，但是不幸的是用户使用这个队列的时候必须要知道队列是用列表来表示的。一个使用这个队列的典型的程序片断如下：

```
NewQ = [], % Don’t do this
Queue1 = queue:add(joe, NewQ),
Queue2 = queue:add(mike, Queue1), ....
```

这就糟糕了——因为 a）用户需要知道队列是用列表来实现的；b）实现者不能改变队列的内部实现（而他可能为了提供一个更好的实现而需要这样做）。

更好的实现方法是：

```
-module(queue).
-export([new/0, add/2, fetch/1]).

new() ->
    [].

add(Item, Q) ->
    lists:append(Q, [Item]).

fetch([H|T]) ->
    {ok, H, T};
fetch([]) ->
    empty.
```

现在我们就可以这样写：

```
NewQ = queue:new(),
Queue1 = queue:add(joe, NewQ),
Queue2 = queue:add(mike, Queue1), ...
```

这种做法就好得多，纠正了这个问题。现在假设用户需要知道队列的长度，他们可能这样写：

```
Len = length(Queue) % 你可不要这么做
```

因为他们知道队列是用列表来实现的。这是一种会导致代码难以理解和维护的很不好的编程实践。如果他们需要知道队列的长度，应该在队 queue 模块中加上求长度的函数，一如：

```
-module(queue).
-export([new/0, add/2, fetch/1, len/1]).

new() -> [].

add(Item, Q) ->
    lists:append(Q, [Item]).

fetch([H|T]) ->
    {ok, H, T};
fetch([]) ->
    empty.

len(Q) ->
    length(Q).
```

现在用户就可以调用 queue:len(Queue)来获取队列的长度了。

这里我们说我们“抽象出了”一个队列的所有细节（这就是为什么队列被称为一种“抽象数据类型”）。

我们为什么要这么麻烦？抽象出实现的内部细节的实践允许我们在改变我们的实现时，不需要改变使用了我们所修改的模块中的函数的模块的代码。比如，我们有如下更好的队列实现方法：

```
-module(queue).
```

```
-export([new/0, add/2, fetch/1, len/1]).

new() ->
    {[],[]}.

add(Item, {X,Y}) -> % Faster addition of elements
    {[Item|X], Y}.

fetch({X, [H|T]}) ->
    {ok, H, {X,T}};
fetch({[], []}) ->
    empty;
fetch({X, []}) ->
    % Perform this heavy computation only sometimes.
    fetch({[],lists:reverse(X)}).

len({X,Y}) ->
    length(X) + length(Y).
```

3.12 使代码达到最大确定性（deterministic）

一个确定性的程序，无论运行多少次，都会按照同样的方式运行。一个非确定性（non-deterministic）的程序每次运行都会产生不同的结果。对于程序调试的目的来讲，使程序尽量达到确定性是一个好主意，这有助于让错误复现。

例如，假设一个进程必须启动五个并行进程，并检查它们都已经正常启动起来了，进一步假设这五个进程启动的顺序是不重要的。

那么我们就可以选择先启动五个并行的进程，然后检查它们是否都已正常启动，但是更好的做法是一次启动一个进程，并在启动下一个进程之前就检查前一个进程是否已经正常启动。

3.13 不要“防御式”编程

一个防御式程序是这样的程序，编写者在编写该程序的时候，并不“信任”他们编写的系统部分的输入数据。一般来讲，编程者不应该测试函数的输入数据的正确性。系统的大部分代码在编写的时候应该假设当前函数的输入数据是正确的。只有小部分的代码应该真正进行数据检查，并且是在数据第一次“进入”系统的时候进行的。因此一旦数据在进入系统的时候被检查过了，就应该从此被认为是正确的。

例如：

```
%% Args: Option is all | normal
get_server_usage_info(Option, AsciiPid)
    Pid = list_to_pid(AsciiPid),
    case Option of
        all -> get_all_info(Pid);
        normal -> get_normal_info(Pid)
    end.
```

当 Option 既不是 normal 也不是 all 的时候，这个函数就会崩溃，这是没有错的。但是这个函数的调用者更有义务提供正确的输入。

3.14 用设备驱动隔离硬件接口

硬件应该通过设备驱动隔离在系统之外。设备驱动应该实现硬件接口，使得硬件看起来像 Erlang 进程一样。硬件应该被做成看起来、运行起来都像正常的 Erlang 进程一样。硬件应该表现得更普通进程一样接收和发送正常的 Erlang 消息，并且当发生错误时，以常规的方式作出响应。

3.15 do 与 undo 都在一个函数里做

假设我们有一个程序，它打开一个文件，对其做了某些事情，然后关闭该文件。那么程序就应该这样编写：

```
do_something_with(File) ->
    case file:open(File, read) of,
        {ok, Stream} ->
            doit(Stream),
            file:close(Stream) % The correct solution
        Error -> Error
    end.
```

注意我们是怎么在同一个函数中打开一个文件（file:open）并关闭它（file:close）的。下面的解决方案就要费解得多，并且究竟哪个文件被关闭了很不明显。不要像这样编写程序：

```
do_something_with(File) ->
    case file:open(File, read) of,
        {ok, Stream} ->
            doit(Stream)
        Error -> Error
    end.

doit(Stream) ->
    ....,
    func234(...,Stream,...).
    ...

func234(..., Stream, ...) ->
    ...,
    file:close(Stream) %% Don’t do this
```

4 错误处理

4.1 将错误处理代码和正常情况代码分离

不要把处理“正常情况”的代码和处理异常的代码搅和在一起。你应该尽量编写正常情况的代码。如果正常处理的代码出错了，你的进程应该尽快报告该错误并自己退出。不要试图纠正错误并继续处理。错误应该在另一个不同的进程中处理。（参看 5.5 条“每个进程只有一个角色”。）

将错误恢复代码和正常情况代码干净地分离，可以极大地简化整个系统的设计。

发生软件或硬件错误时产生的错误日志应该对后续的错误诊断和错误纠正有所帮助。一个进程所产生的对错误的后续处理有帮助的所有信息都应该保存一份永久化的记录。

4.2 标明错误内核

进行系统设计的时候，一项基本的工作就是标识出系统的哪些部分如果出错是应该被纠正的，哪些部分如果出错是不必被纠正的。

在一般的操作系统设计中，都认为系统的内核一旦出错是应该也必须被纠正的，而用户程序出错而错误不会影响到系统的完整性的时候，就不需要纠正了。

前者在进行系统设计的时候，就必须标明这部分如果出错就必须被纠正，我们成这部分为错误内核（error kernel）。通常，错误内核都有某种实时内存驻留数据库，用来保存硬件的状态。

5 进程、服务器和消息

5.1 将一个进程的实现放在一个模块中

一个进程的实现代码应该放在一个模块中。一个进程可以调用函数库中的任何函数，但是进程的“主循环”（top loop）应该只包含在一个模块中。一个进程

的主循环不要被分开到多个模块中——这样的控制流会造成理解的极大困难。这不是意味着我们不能用通用服务器库，通用服务器库是用来帮助组织进程的控制流的。

反过来讲，一个单独的模块中只应该包含一种进程的代码的实现。包含了多个不同进程的实现的模块也可能会很难以理解，最好把每个单独进程的实现代码分别放到单独的模块中。

5.2 利用进程来组织系统

进程是系统的基本构成元素。但是如果只需要进行函数调用，就不要用进程和消息传递。

5.3 注册进程

注册进程（registered process）应该给注册一个跟模块名相同的名字。这使得查找进程的实现代码容易一些。

长时间的运行的进程都应该进行注册。

5.4 给系统中的每个真正的并发活动注册唯一一个并行进程

在决定是用顺序化方法还是并行进程来实现一个处理时，应该保持与问题的内在结构保持一致。主要的原则是：

“用一个并行进程来模拟真实世界中的一个真实存在的并发活动。”

当并行进程的个数和真实世界中并行活动的真实个数是 1 对 1 的映射时，程序就容易理解得多。

5.5 每个进程应该只有一个“角色”

进程可以扮演系统中的不同角色，例如在客户-服务器模型中就可以扮演客户和服务器。

应该尽可能地让一个进程只扮演一个角色，即，一个进程可以是客户，也可

以是服务器，但是不要把他们合起来。

进程通常可以扮演的其它角色有：

监督者 监控其它进程，如果它们出错就重启它们。

工作者 一个正常的工作者进程（可能会有错误）。

可靠的工作者 不允许含有错误的工作者进程。

5.6 在服务器和协议处理器中尽可能地使用通用函数

在许多情况下，使用在标准库中实现了的通用服务器（如 gen_server）是一个很好的主意。程序中一致使用几个集中实现的通用服务器可以很大程度上简化整个系统的结构。

对于大多数的协议处理软件来说，这样的做法也同样可行。

5.7 给消息打上标签

所有的消息都应该打上标签。这会使接收语句中的接收顺序变得更随便，而且易于实现新消息的添加。

不要像这样写：

```
loop(State) ->
    receive
        ...
        {Mod, Funcs, Args} -> % Don't do this
            apply(Mod, Funcs, Args},
            loop(State);
        ...
    end.
```

这种写法使得在增加一种新消息{get_status_info, From, Option}时，会引起参与上面的{Mod, Func, Args}消息的混淆。

如果消息是同步的，则返回消息应该打一个新的标签，描述所返回的消息。例如：如果进来的消息是用 get_status_info 打标的，则返回消息就可以用 status_info 来打标。选择不同的标签的一个原因是便于调试。

如下是一个好的做法：

```
loop(State) ->
    receive
        ...
        % Use a tagged message.
        {execute, Mod, Funcs, Args} ->
            apply(Mod, Funcs, Args},
            loop(State);
        {get_status_info, From, Option} ->
            From ! {status_info,
            get_status_info(Option, State)},
            loop(State);
        ...
    end.
```

5.8 清掉未知消息

每个进程处理函数至少在一个 receive 语句中具有 Other 选项。这是为了避免消息队列被挤爆。例如：

```
main_loop() ->
    receive
        {msg1, Msg1} ->
            ...,
            main_loop();
        {msg2, Msg2} ->
            ...,
```

```
            main_loop();
        Other -> % Flushes the message queue.
            error_logger:error_msg(
                "Error: Process ~w got unknown msg ~w~n.",
                [self(), Other]),
            main_loop()
    end.
```

5.9 编写尾递归的进程处理函数

所有的进程处理函数都*必须*是尾递归的，否则这个进程将会把系统的内存消耗殆尽。

不要像这样写程序：

```
loop() ->
    receive
        {msg1, Msg1} ->
            ...,
            loop();
        stop ->
            true;
        Other ->
            error_logger:log({error, {process_got_other,
            self(), Other}}),
            loop()
    end,
    % 不要这样写
    % 这是非尾递归的
    io:format("Server going down").
```

正确的写法是：

```
loop() ->
    receive
        {msg1, Msg1} ->
            ...,
            loop();
        stop ->
            io:format("Server going down");
        Other ->
            error_logger:log({error, {process_got_other,
            self(), Other}}),
            loop()
    end. % 这是尾递归的
```

如果你用了某种 behaviour 库中的服务器，譬如 gen_server，你就可以自动避免犯这种错误。

5.10 接口函数

尽可能使用函数作为接口，避免直接向进程发送消息。把消息传递封装到接口函数中。有许多情况下，你都不能直接给进程发消息。

消息协议是内部信息，对其它模块来说应该是看不见的。

接口函数的例子如：

```
-module(fileserver).
-export([start/0, stop/0, open_file/1, ...]).

open_file(FileName) ->
    fileserver ! {open_file_request, FileName},
    receive
        {open_file_response, Result} -> Result
    end.
```

```
...<code>...
```

5.11 超时

在 receive 语句中使用 after 时，一定要小心。要确保消息在超时以后到达的情况得到处理（见 5.8“冲掉未知消息”）。

5.12 捕获退出信号（trapping exits）

让尽可能少的进程去捕获退出信号。进程要么应该捕获退出信号，要么不应该捕获。一般来说，让一个进程在捕获和不捕获退出信号之间切换是很坏的实践。

6 一些 Erlang 的特别约定

6.1 使用 record 作为标准数据结构

使用 record 作为标准数据结构。record 是具有标签的 tuple，自从在 Eralng4.3 版中被引入以后就一直沿用（见 EPK/NP 95:034）。record 类似于 C 中的 struct 和 Pascal 中的 record。

如果一个 record 要在多个模块中被使用，那么它的定义应该放在一个头文件（文件名以.hrl 为后缀）中，然后在模块中包含这个头文件。如果一个 record 只在一个模块中被使用，那么这个 record 的定义就应该放在该模块的定义文件的开始部分。

Erlang 的 record 的特性可以用来确保跨模块数据结构的一致性，因此当不同模块之间需要传递数据结构时，可以用在接口函数中。

6.2 使用选择器（selector）和构造器（constructor）

使用 record 提供的选择器和构造器的特性来管理 record 的实例。不要显式地把 record 作为一个 tuple 进行匹配。例如：

```
demo() ->
    P = #person{name = "Joe", age = 29},
```

```
    #person{name = Name1} = P,% 像这样使用匹配, 或者...
    Name2 = P#person.name.      % 像这样.
```

不要这样写程序：

```
demo() ->
    P = #person{name = "Joe", age = 29},
    % 不要这样做
    {person, Name, _Age, _Phone, _Misc} = P.
```

6.3 给返回值打标签

应该给返回值打上标签。

不要这样写程序：

```
keysearch(Key, [{Key, Value}|_Tail]) ->
    Value; %% 不要返回未打标签的值!
keysearch(Key, [{_WrongKey,_WrongValue}|Tail]) ->
    keysearch(Key, Tail);
keysearch(Key, []) ->
    false.
```

因为这里的 Value 的值有可能是 false。下面才是正确的解决方法：

```
keysearch(Key, [{Key, Value}|_Tail]) ->
    {value, Value}; %% 对，返回打了标签的值.
keysearch(Key, [{_WrongKey, _WrongValue}|Tail]) ->
    keysearch(Key, Tail);
keysearch(Key, []) ->
    false.
```

6.4 使用 catch 和 throw 时要极其小心

只有在你非常明白自己在做什么的时候，才可以使用 catch 和 throw！应该尽量杜绝使用 catch 和 throw。

catch 和 throw 在处理特别复杂和不可靠的输入（来自外部世界的输入，而不是来自你的内部可靠的程序的输入）的时候可能很有用，因为这些输入可能导致代码中很深的地方的许多错误。一个例子就是编译器程序。

6.5 使用进程字典（process dictionary）时要极其小心

只有在你非常明白自己在做什么的时候，才可以使用 get 和 put 等操作！应该尽量杜绝使用 get 和 put 等操作。

一个使用了进程字典的函数可以用引入一个新参数的方法进行重写。

例如：

不要这样写：

```
tokenize([H|T]) ->
    ...;
tokenize([]) ->
    % 不要使用 get/1 (像这样)
    case get_characters_from_device(get(device)) of
        eof -> [];
        {value, Chars} ->
            tokenize(Chars)
    end.
```

正确的写法是

```
tokenize(_Device, [H|T]) ->
    ...;
tokenize(Device, []) ->
    % 这种方法更好
    case get_characters_from_device(Device) of
        eof -> [];
        {value, Chars} ->
            tokenize(Device, Chars)
```

```
end.
```

使用 get 和 put 可能会引起在用同一个输入参数进程调用一个函数时，函数的行为会有所不同。这种不确定性会造成代码难以阅读。调试这种函数也会更复杂，因为这种使用了 get 和 put 的函数的行为不仅与其输入参数相关，而且还跟进程字典相关。Erlang 中的许多运行时错误（例如 bad_match）可以指示函数的输入参数出错，但是没有哪种错误指示进程字典出错。

6.6 不要使用 import

不要使用-import。用了这种方法以后，就会造成代码更难懂，因为不能直接看出一个函数是在哪个模块中定义的。可以使用 exref（Cross Reference Tool）来找出模块之间的依赖关系。

6.7 导出（export）函数

标明一个函数被导出的原因。一个函数可能会因为下面这些原因被导出（打个比方）：

- 它是模块的用户接口。
- 它是其它模块要使用的一个接口函数。
- 它要被本模块内部调用的 apply、spawn 函数调用。

这时候应该使用不同的-export 进行分组，并分别注释一下。例如：

```
%% user interface
-export([help/0, start/0, stop/0, info/1]).

%% intermodule exports
-export([make_pid/1, make_pid/3]).
-export([process_abbrevs/0, print_info/5]).

%% exports for use within module only
```

```
-export([init/1, info_log_impl/1]).
```

7 特别的词汇和风格约定

7.1 不要编写深度嵌套的代码

嵌套的代码是指在 case/if/receive 语句中包含有 case/if/receive 语句的代码。写出深度嵌套的代码是一种糟糕的编程风格——代码会无限制地向右扩展，很快就会变得很不可读。应该限制你的代码最多有两级缩进（indentation）。这一点可以通过把代码分成更小的函数来达到。

7.2 不要编写很大的模块

一个模块应该包含不超过 400 行源代码。多个较小的模块比一个大模块要好。

7.3 不要编写很长的函数

不要编写超过 15～20 行代码的函数。把大的函数分割成许多小的函数。也不要为了减少函数行数而写出很长很长的行。

7.4 不要编写很长的代码行

不要编写特别长的代码行。一行应该不超过 80 个字符（例如应该在 A4 的纸上写得下）。

在 Erlang4.3 和之后的版本中，字符串常量会被自动连接。例如：

```
io:format("Name: ~s, Age: ~w, Phone: ~w ~n"
          "Dictionary: ~w.~n", [Name, Age, Phone, Dict])
```

7.5 变量名

选择一个有意义的变量名——这是很难的。

如果一个变量名有多个词组成，就用下划线“_”或首字母大写来分隔它们。例如：My_variable 或 MyVariable。

避免使用“_”作为不关心的变量，而应该使用一个“_”打头的变量。例如：_Name。如果以后你需要这个变量的值，就只需要删掉前面的下划线就行了。这样做你就很容易找到要替换哪些下划线，并且代码会更易读。

7.6 函数名

函数名必须与函数做了什么保持一致。函数的返回值应该与函数名所暗示的一致。不要给读者带来惊讶。给一些惯用的函数一些惯用的名字（start, stop, init, main_loop）。

不同模块中解决相同问题的函数应该具有相同的名字。例如：Module:module_info()。

函数的名字取得不好，是最普遍的编程错误之一——给函数挑选一个合适的名字是很困难的！

一些命名习惯在编写的函数量特别多时是非常有用的。例如，前缀“is_”能够用来表明这个函数将会返回原子 true 或 false。

```
is_...() -> true | false
check_...() -> {ok, ...} | {error, ...}
```

7.7 模块名

Erlang 的模块结构是扁平的（即模块中不包含模块）。然而，我们经常希望可以模拟出模块层次化结构的效果。通过用相同的前缀来命名一组相关的模块，可以实现这一点

例如，如果一个 ISDN 处理器用了五个不同的相关模块来实现。这些模块就应该这样命名：

```
isdn_init
isdn_partb
```

```
isdn_...
```

7.8 程序格式保持一致风格

一种一致的编程风格会有助于你和其他人理解你的代码。不同的人有不同的缩进、留白风格等等。

例如你可能写元组的时候各个元素之间只有一个逗号：

```
{12,23,45}
```

而别人可能用了一个逗号跟一个空格：

```
{12, 23, 45}
```

一旦采用了一种风格，就应该贯彻它。

在一个大的项目中，所有的部门应该采用相同的风格。

8 代码文档化

8.1 注明代码归属

你一定要在所有代码的模块头中正确地注明代码的归属。要说明代码中蕴涵的所有思想的来源——如果你的代码是从其它代码的基础上衍生的，就要说明你从哪里获得该代码，原来是谁写的。

不要盗窃代码——盗窃代码是指从其它模块中拷贝来代码，进行了修改，但是*忘记了*说谁编写了原始的代码。

一个有用的归属说明的例子如下：

```
-revision(' Revision: 1.14 ').
-created(' Date: 1995/01/01 11:21:11 ').
-created_by(' eklas@erlang').
-modified(' Date: 1995/01/05 13:04:07 ').
-modified_by(' mbj@erlang').
```

8.2 提供代码中对规格说明的引用

代码中应该提供对任何与代码相关的文档的引用。例如，如果一个代码实现了某种通信协议或者硬件接口，就应该给出一个该文档准确的引用，并注明编写代码时引用的该文档的页码。

8.3 将所有错误文档化

所有的错误都应该用英文描述一下它的意义，集中为一个清单，并放在一个单独的文档中（见 10.4 节“错误消息”）。

这里所说的错误是指已经被系统检测到的错误。

在你的程序中的某个点，你检测到了一个逻辑错误，则称之为错误日志：

```
error_logger:error_msg(Format,
                       {Descriptor, Arg1, ....})
```

要确保{Descriptor, Arg1, ...}这一行被添加到了错误消息文档中。

8.4 将消息中的所有的标准数据结构文档化

当在系统的各个部分之间发送消息时，使用打了标签的元组作为标准数据结构。

Erlang 的 record 的特性（自 Erlang 的 4.3 版引入之后）可以用来确保跨模块数据结构的一致性。

对所有这些数据结构的一个英文描述也应该文档化（见 10.4 节“消息描述”）。

8.5 注释

注释应当清晰简洁，避免不必要的辞藻。要确保注释与代码同步。要检查注释确实有助于代码的理解。注释应该用英文来书写。

关于模块的注释不应该缩格，应该以 3 个百分号（%%%）打头，（参见 8.10 “文件头，描述”一节）。

关于函数的注释也不应该缩格，应该以 2 个百分号（%%）打头，（参见 8.6 “注释每个函数”一节）。

在 Erlang 代码中的注释应该以 1 个百分号（%）打头。如果一行只包含一个注释，就应该与 Erlang 代码同进缩。这种注释应该放在它所说明的语句的紧上面。如果注释能够放在代码语句的同一行，就最好了。

例如：

```
%% Comment about function
some_useful_functions(UsefulArgugument) ->
  another_functions(UsefulArgugument),    % Comment at end of line
  % Comment about complicated_stmnt at the same level of indentation
  complicated_stmnt,
......
```

8.6 注释每个函数

需要注释的重要的事项有：

- 本函数的目的。
- 本函数的合法输入域。即函数参数的数据结构和它们的意义。
- 本函数的输出域。即函数可能返回的所有可能的数据结构和它们的意义。
- 如果本函数实现了一个复杂的算法，就要描述一下。
- 可能引起错误和退出信号的原因，包括由 exit/1、throw/1 或任何不明显的运行时错误产生的退出信号。注意故障和返回一个错误的区别。
- 本函数的任何副作用。

例如：

```
%%--------------------------------------------------------------------
%% Function: get_server_statistics/2
%% Purpose: Get various information from a process.
%% Args:   Option is normal|all.
%% Returns: A list of {Key, Value}
%%     or {error, Reason} (if the process is dead)
%%--------------------------------------------------------------------
get_server_statistics(Option, Pid) when pid(Pid) ->
  ......
```

8.7 数据结构

一个 record 的定义应该伴随着一个简单的文本描述。例如：

```
%% File: my_data_structures.h

%%----------------------------------------------------------------------
%% Data Type: person
%% where:
%%    name: A string (default is undefined).
%%    age: An integer (default is undefined).
%%    phone: A list of integers (default is []).
%%    dict:     A dictionary containing various information about the person.
%%       A {Key, Value} list (default is the empty list).
%%----------------------------------------------------------------------
-record(person, {name, age, phone = [], dict = []}).
```

8.8 文件头，版权

每个源代码文件都应该以一个版权信息开头，例如：

```
%%%----------------------------------------------------------------------
%%% Copyright Ericsson Telecom AB 1996
%%%
%%% All rights reserved. No part of this computer programs(s) may be
%%% used, reproduced,stored in any retrieval system, or transmitted,
%%% in any form or by any means, electronic, mechanical, photocopying,
%%% recording, or otherwise without prior written permission of
%%% Ericsson Telecom AB.
%%%----------------------------------------------------------------------
```

8.9 文件头，修订历史

每个源代码文件都必须具备其修订历史的文档，这个文档显式了有谁维护过这个文件，他们做了什么。

```
%%%----------------------------------------------------------------------
%%% Revision History
%%%----------------------------------------------------------------------
%%% Rev PA1 Date 960230 Author Fred Bloggs (ETXXXXX)
%%% Initial pre release. Functions for adding and deleting foobars
%%% are incomplete
%%%----------------------------------------------------------------------
%%% Rev A Date 960230 Author Johanna Johansson (ETXYYY)
%%% Added functions for adding and deleting foobars and changed
%%% data structures of foobars to allow for the needs of the Baz
%%% signalling system
%%%----------------------------------------------------------------------
```

8.10 文件头，描述

每个文件都应该有一个关于其中包含的模块的简短的描述，和所有导出的函

数的一个简单的描述。

```
%%%----------------------------------------------------------------------
%%% Description module foobar_data_manipulation
%%%----------------------------------------------------------------------
%%% Foobars are the basic elements in the Baz signalling system. The
%%% functions below are for manipulating that data of foobars and for
%%% etc etc etc
%%%----------------------------------------------------------------------
%%% Exports
%%%----------------------------------------------------------------------
%%% create_foobar(Parent, Type)
%%%   returns a new foobar object
%%%   etc etc etc
%%%----------------------------------------------------------------------
```

如果你知道代码的任何缺点、bug、难以测试的特性，一定要用一个特别的注释来标明它们，不要试图隐藏它们。如果模块的任何部分不完整，就应该增加一个特别的注释。为对模块将来的维护者们有帮助的任何事情加以注释。即使你编写的产品相当成功，它也可能在今后十年里被你可能并不认识的人修改或改进。

8.11 不要注释过时代码——删除它

删除后，在修改历史中添加一条注释。记住，源代码控制系统也会帮助你的！

8.12 使用一个源代码控制系统

所有有价值的项目都应该使用诸如 RCS、CVS 或 Clearcase 之类的源代码控制系统（source code control system）来跟踪所有的模块。

9 最普遍的错误

- 写出跨越多页的函数（见 7.3“不要写很长的函数”）。
- 写出具有深度嵌套的 if/receive/case 等（见 7.1“不要写深度嵌套的代码”）。
- 写出糟糕的类型的函数（见 6.3“给返回值打标签”）。
- 变量名没有意义（见 7.5“变量名”）。
- 在不需要进程的时候使用了进程（见 5.3“给系统中的每个真正的并发

活动注册唯一一个并行进程”)。

- 数据结构选得不好（糟糕的表示)。
- 糟糕的注释或压根儿没有注释（经常只注释一下参数和返回值)。
- 编码无进缩。
- 用了 put/get（见 6.5“使用进程字典时要机器小心”)。
- 没有消息队列的控制（见 5.8“清掉未知消息”和 5.11“超时”)。

10 必备文档

本节描述了一些在设计和维护用 Erlang 编写的系统时的一些必要的系统级文档。

10.1 模块描述

每个模块一个章节。包括每个模块的描述，和按下面的方式对所有的导出函数的描述：

- 函数的参数的数据结构和意义。
- 返回值的数据结构和意义。
- 函数的目的。
- 所有可能引起故障和退出信号的原因，包括显式调用 exit/1 产生的退出信号。

后续定义的文档的格式：

10.2 模块描述

除了模块内部定义的以外，所有进程见消息的格式。

后续定义的文档的格式：

10.3 进程

关于系统中所有注册的进程，和它们的接口及目的的描述。

关于动态进程和它们间的接口的描述。

后续定义的文档的格式：

10.4 错误消息

关于错误消息的描述。

后续定义的文档的格式：

附录 C

（略）

【本附录为一篇论文，题目：Getting Erlang to talk to the outside world，作者：Joe Armstrong.】

附录 D

（略）

【本附录为论文原版的排版说明。】

参考文献

[1] Ingemar Ahlberg, John-Olof Bauner, and Anders Danne. Prototyping cordless using declarative programming. XIV International Switching Symposium, October 1992.

[2] Leon Alkalai and Ann T. Tai. Long-life deep-space applications. IEEE Computer, 31:37–38, April 1998.

[3] Marie Alpman. Svenskt internetbolag köps för 1,4 miljarder. Ny Teknik, August 2000.

[4] Gregory R. Andrews and Fred B. Schneider. Concepts and notations for concurrent programming. ACM Computing Surveys (CSUR), 15(1):3–43, 1983.

[5] J. Armstrong, M. Williams, C. Wikström, and R. Virding. Concurrent Programming in Erlang. Prentice-Hall, Englewood Clics, N.J., 1996.

[6] J. L. Armstrong. Ubf - universal binary format, http://www.sics.se/ joe/ubf. 2002.

[7] J. L. Armstrong. Erlguten. 2003. http://www.sics.se/ joe/erlguten.html.

[8] J. L. Armstrong and T. Arts. A practical type system for erlang. Erlang User Conference, 2002.

[9] J. L. Armstrong, B. O. Däcker, S. R. Virding, and M. C. Williams. Implementing a functional language for highly parallel real-time applications. In Sodware Engineering for Telecommunication Switching Systems, April 92.

[10] J. L. Armstrong, S. R. Virding, and M. C. Williams. Use of Prolog for Developing a New Programming Language. In C. Moss and K. Bowen, editors, Proc. 1st Conf. on The Practical Application of Prolog, London, England, 1992. Association for Logic Programming.

[11] Joe Armstrong. Increasing the reliability of email services. In Proceedings of the 2000 ACM symposium on Applied Computing, pages 627–632. ACM Press,

2000.

[12] Joe Armstrong. Concurrency oriented programming. Lightweight Languages Workshop (LL2), November 2002.

[13] Joe Armstrong. Getting erlang to talk to the outside world. In Proceedings of the 2002 ACM SIGPLAN workshop on Erlang, pages 64–72. ACM Press, 2002.

[14] Joe Armstrong. Concurrency oriented programming in erlang. GUUG 2003, March 2003.

[15] Joe Armstrong. A webserver daemon. 2003. This is available at http://www.sics.se/~joe/tutorials/web_server/web_server.html.

[16] A. Avienis. Design of fault-tolerant computers. In Proceedings of the 1967 Fall Joint Computer Conference. AFIPS Conf. Proc., Vol. 31, Thompson Books, Washington, D.C., 1967, pp. 733-743, pages 733–743, 1967.

[17] Jonas Barklund. Erlang 5.0 specification. 2000. available from http//www.bluetail.com/~rv.

[18] Stacan Blau and Jan Rooth. Axd 301 – a new generation of atm switching. Ericsson Review, (1), 1998.

[19] Grady Booch, James Rumbaugh, and Ivar Jacobson. The Unified Modeling Language user guide. Addison Wesley Longman Publishing Co., Inc., 1999.

[20] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler (Eds). Extensible markup language (xml) 1.0 (second edition). October 2000, http://www.w3.org/tr/2000/rec-xml-20001006. 2000.

[21] Ciaràn Bryce and Chrislain Razafimahefa. An approach to safe object sharing. In Proceedings of the conference on Object-oriented programming, systems, languages, and applications, pages 367–381. ACM Press, 2000.

[22] George Candea and Armando Fox. Crash only sodware. In Proceedings of the 9th workshop on Hot Topics in Operating Systems (TotOS-IX), May 2003.

[23] Richard Carlsson, Thomas Lindgren, Björn Gustavsson, Sven-Olof Nyström, Robert Virding, Erik Johansson, and Mikael Pettersson. Core erlang 1.0. November 2001.

[24] J. D. Case, M. S. Fedor, M. L. Schocstall, and C. Davin. Simple network management protocol (SNMP). RFC 1157, Internet Engineering Task Force, May 1990.

[25] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (wsdl) 1.1, march 2001, http://www.w3.org/tr/2001/note-wsdl-20010315/. 2001.

[26] Dan Connolly, Bert Bos, Yuichi Koike, and Mary Holstege. http://www.w3.org/2000/04/schema_hack/. 2000.

[27] M. R. Crispin. Internet message access protocol - version 4. RFC 1730, Internet Engineering Task Force, December 1994.

[28] Grzegorz Czajkowski and Laurent Daynès. Multitasking without comprimise: a virtual machine evolution. In Proceedings of the OOPSLA ’01 conference on Object Oriented Programming Systems Languages and Applications, pages 125–138. ACM Press, 2001.

[29] Bjarne Däcker. Datalogilaboratoriet - de första 10 °aren. March 1994.

[30] Bjarne Däcker. Concurrent functional programming for telecommunications: A case study of technology introduction. November 2000. Licentiate Thesis.

[31] A. Dahlin, M. Froberg, J. Grebeno, J.Walerud, and P. Winroth. Eddie: A robust and scalable internet server. May 1998.

[32] D. C. Fallside (Ed). Xml schema part 0: Primer. may 2002. http://www.w3.org/tr/2001/rec-xmlschema-0-20010502/. 2002.

[33] Dick Eriksson, Mats Persson, and Kerstin Ödling. A switching sodware architecture prototype using real time declarative language. XIV International

Switching Symposium, October 1992.

[34] Open source erlang distribution. 1999.

[35] J. A. Feldman, J. R. Low, and P. D. Rovner. Programming distributed systems. In Proceedings of the 1978 ACM Computer Science Conference, pages 310–316, 1978.

[36] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. RFC 2616, The Internet Society, June 1999. Seehttp://www.ietf.org/rfc/rfc2616.txt.

[37] Ian Foster and Stephen Taylor. Strand: new concepts in parallel programming. Prentice-Hall, Inc., 1990.

[38] Jim Gray. Why do computers stop and what can be done about it? Technical Report 85.7, Tandem Computers, 1985.

[39] M. Gudgin, M. Hadley, J-J. Moreau, and H. F. Nielsen. Soap version 1.2 part 1: Messaging framework, december 2001, http://www.w3.org/tr/2001/wd-soap12-part1-20011217. 2001.

[40] M. Gudgin, M. Hadley, J-J. Moreau, and H. F. Nielsen. Soap version 1.2 part 2: Adjuncts, december 2001, http://www.w3.org/tr/2001/wdsoap12-part2-20011217. 2001.

[41] Bogumil Hausman. Turbo erlang. International Logic Programming Symposium, October 1993. [42] Bogumil Hausman. Turbo erlang: Approaching the speed of c. In Evan Tick and Giancarlo Succi, editors, Implementations of Logic Programming Systems, pages 119–135. Kluwer Academic Publishers, 1994.

[43] American National Standards Institute, Institute of Electrical, and Electronic Engineers. IEEE standard for binary floating-point arithmetic. ANSI/IEEE Standard, Std 754-1985, New York, 1985.

[44] ISO/IEC. Osi networking and system aspects - abstract syntax notation one

(asn.1). ITU-T Rec. X.680 — ISO/IEC 8824-11, ISO/IEC, 1997.

[45] ITU. Recommendation Z.100 – specification and description language (sdl). ITU-T Z.100, International Telecommunication Union, 1994.

[46] D. Reed J. Oikarinen. RFC 1459: Internet relay chat protocol. May 1993.

[47] Erik Johansson, Sven-Olof Nyström, Mikael Pettersson, and Konstantinos Sagonas. Hipe: High performance erlang.

[48] D. Richard Kuhn. Sources of failure in the public switched telephone network. IEEE Computer, 30(4):31–36, 1997.

[49] Simon Marlow and Philip Wadler. A practical subtyping system for Erlang. In International Conference on Functional Programming, pages 136–149. ACM, June 1997.

[50] B. Martin and B. Jano (Eds). Wap binary xml content format, june 1999, http://www.w3.org/tr/wbxml. 1999.

[51] H°akan Millroth. Private communication. 2003.

[52] J. Myers and M. P. Rose. Post oece protocol - version 3. RFC 1939, Internet Engineering Task Force, May 1996.

[53] Nortel Networks. Alteon ssl accelerator product brief. September 2002.

[54] (Ed) Nilo Mitra. Soap version 1.2 part 0: Primer. december 2001, http://www.w3.org/tr/2001/wd-soap12-part0-20011217. 2001.

[55] Hans Olsson. Ericsson lägger ner utveckling. Dagens Nyheter, December 1995.

[56] OMG. Common Object Request Broker Architecture (CORBA)—v2.6.1 Manual. The Object Management Group, Needham, U.S.A, 2002.

[57] J. B. Postel. Simple mail transfer protocol. RFC 821, Internet Engineering Task Force, August 1982.

[58] K. Renzel. Error handling for business information systems. 2003.

[59] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. Computer Systems, 1(3):222–238, 1983.

[60] Fred B. Schneider. Byzantine generals in action: implementing fail-stop processors. ACM Transactions on Computer Systems (TOCS), 2(2):145–154, 1984.

[61] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. ACM Computing Surveys (CSUR), 22(4):299–319, 1990.

[62] R. Srinivasan. RFC 1832: XDR: External data representation standard. August 1995.

[63] Ann T. Tai, Kam S. Tso, Leon Alkalai, Savio N. Chau, andWilliam H. Sanders. On the ecectiveness of a message-driven confidence-driven protocol for guarded sodware upgrading. Performance Evaluation, 44(1-4):211–236, 2001.

[64] Hän Thˆe Thänh. Micro-typographic extensions to the tex typesetting system. Masaryk University Brno, 2000.

[65] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn (Eds). Xml schema part 1: Structures. w3c recommendation, may 2001. http://www.w3.org/tr/2001/rec-xmlschema-1-20010502/. 2001.

[66] Seved Torstendahl. Open telecom platform. Ericsson Review, (1), 1997.

[67] Jecrey Voas. Fault tolerance. IEEE Sodware, pages 54–57, July–August 2001.

[68] David H.D. Warren. An abstract Prolog instruction set. Tecnical Note 309, SRI International, Menlo Park, California, October 1983.

[69] Ulf Wiger. Private communication.

[70] Ulf Wiger, Gösta Ask, and Kent Boortz. World-class product certification using erlang. In Proceedings of the 2002 ACM SIGPLAN workshop on Erlang, pages 24–33. ACM Press, 2002.

[71] Weider D. Yu. A sodware fault prevention approach in coding and root cause analysis. Bell Labs Technical Journal, 3(2), 1998.