

Working with Unix Processes

理解 Unix进程

【加】 Jesse Storimer 著 门佳 译

- 唯一本针对Web开发人员的Unix编程书籍
- 无需借助C语言即可玩转Unix进程
- 自主编写并调试高效服务器



人民邮电出版社
POSTS & TELECOM PRESS

Working with
Unix Processes
——理解——
Unix进程

【加】 Jesse Storimer 著 门佳 译

人民邮电出版社
北京

图书在版编目(CIP)数据

理解Unix进程 / (加) 斯托里默 (Storimer, J.) 著;
门佳译. -- 北京 : 人民邮电出版社, 2013.6
(图灵程序设计丛书)

书名原文: Working with Unix Processes
ISBN 978-7-115-31689-9

I. ①理… II. ①斯… ②门… III. ①UNIX操作系统
IV. ①TP316.81

中国版本图书馆CIP数据核字(2013)第079983号

内 容 提 要

本书从 Unix 编程的基础概念着手，采用循序渐进的方法，详细介绍了 Unix 进程的内部工作原理。本书提供的许多简单而强大的技术，能够帮助 Web 开发人员深入了解 Unix 系统的并发性、守护进程、生成进程（spawning process）与信号等。同时，读者也可以使用这些技术和方法编写并调试自己的服务器。此外，本书附录部分也涉及了一些流行的 Ruby 项目，让读者进一步了解如何巧妙运用 Unix 进程。

本书适合 Unix 程序员、Web 开发人员阅读。

图灵程序设计丛书

理解Unix进程

◆ 著 [加] Jesse Storimer

译 门 佳

责任编辑 丁晓昀

执行编辑 陈婷婷

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号

邮编 100061 电子邮件 315@ptpress.com.cn

网址 <http://www.ptpress.com.cn>

北京铭成印刷有限公司印刷

开本：880×1230 1/32

◆ 印张：4

字数：97千字 2013年6月第1版

印数：1—3 500册 2013年6月北京第1次印刷

著作权合同登记号 图字：01-2012-7762号

ISBN 978-7-115-31689-9

定价：29.00元

读者服务热线：(010)51095186转604 印装质量热线：(010)67129223

反盗版热线：(010)67171154

版 权 声 明

Copyright © 2012 Jesse Storimer. Original English language edition, entitled *Working with Unix Processes*.

Simplified Chinese-language edition copyright © 2013 by Posts & Telecom Press. All rights reserved.

本书中文简体字版由 The Pragmatic Programmers, LLC 授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。

致 谢

我要向几位了不起的人致以深深的谢意，他们阅读了本书初稿，让我知道如何宣传好此书，在需要时给我动力，在各方面都给予我莫大的帮助。他们是 Sam Storry, Jesse Kaunisviita, 以及 Marc-André Cournoyer。

我也要向妻子和女儿表达无限的感激，你们不仅帮助我度过写书这段飘忽不定的日程安排，而且时刻陪伴着我，为我提供各种参考意见。没有你们的爱和支持，我无法做到这一切。是你们让这一切更有意义！

目 录

第 1 章 引言	1
第 2 章 基础知识	3
2.1 干嘛要在意?	3
2.2 驾驭神力!	4
2.3 概述	4
2.4 系统调用	5
2.5 命名法, wtf(2)	6
2.6 进程: Unix 之本	7
第 3 章 进程皆有标识	9
3.1 交叉参考	9
3.2 实践领域	10
3.3 系统调用	10
第 4 章 进程皆有父	12
4.1 交叉参考	12
4.2 实践领域	13
4.3 系统调用	13

2 | 目 录

第 5 章 进程皆有文件描述符	14
5.1 万物皆为文件	14
5.2 描述符代表资源	14
5.3 标准流	17
5.4 实践领域	18
5.5 系统调用	18
第 6 章 进程皆有资源限制	19
6.1 找出限制	19
6.2 软限制与硬限制	20
6.3 提高软限制	20
6.4 超出限制	21
6.5 其他资源	22
6.6 实践领域	22
6.7 系统调用	23
第 7 章 进程皆有环境	24
7.1 这是个散列吗?	25
7.2 实践领域	25
7.3 系统调用	26
第 8 章 进程皆有参数	27
8.1 这是个数组!	27
8.2 实践领域	28
第 9 章 进程皆有名	29
9.1 进程命名	29
9.2 实践领域	30
第 10 章 进程皆有退出码	31

目 录 | 3

第 11 章 进程皆可衍生	34
11.1 Luke, 使用 fork(2)	34
11.2 多核编程?	37
11.3 使用 block	38
11.4 实践领域	38
11.5 系统调用	38
第 12 章 孤儿进程	39
12.1 失控	39
12.2 弃子	40
12.3 管理孤儿	40
第 13 章 友好的进程	41
13.1 对 CoW 好点	41
13.2 MRI/RBX 用户	43
第 14 章 进程可待	44
14.1 看顾 (Babysitting)	45
14.2 Process.wait 一家子	46
14.3 使用 Process.wait2 进行通信	46
14.4 等待特定的子进程	48
14.5 竞争条件	49
14.6 实践领域	50
14.7 系统调用	51
第 15 章 僵尸进程	52
15.1 等待终有果	52
15.2 僵尸长什么样子?	53
15.3 实践领域	54
15.4 系统调用	54

4 | 目 录

第 16 章 进程皆可获得信号	55
16.1 捕获 SIGCHLD	55
16.2 SIGCHLD 与并发	56
16.3 信号入门	59
16.4 信号来自何方?	59
16.5 信号一览	61
16.6 重定义信号	62
16.7 忽略信号	63
16.8 信号处理程序是全局性的	64
16.9 恰当地重定义信号处理程序	64
16.10 何时接收不到信号?	66
16.11 实践领域	66
16.12 系统调用	67
第 17 章 进程皆可互通	68
17.1 我们的第一个管道	68
17.2 管道是单向的	70
17.3 共享管道	70
17.4 流与消息	72
17.5 远程 IPC?	74
17.6 实践领域	74
17.7 系统调用	74
第 18 章 守护进程	75
18.1 首个进程	75
18.2 创建第一个守护进程	76
18.3 深入 Rack	76
18.4 逐步将进程变成守护进程	77
18.5 进程组和会话组	78
18.6 实践领域	82

目 录 | 5

18.7 系统调用	83
第 19 章 生成终端进程	84
19.1 fork + exec	84
19.2 exec 的参数	86
19.3 实践领域	90
19.4 系统调用	91
第 20 章 尾声	92
20.1 抽象	92
20.2 通信	93
20.3 再会，而非永别	93
附录 A Resque 如何管理进程	95
附录 B Unicorn 如何收割工作进程	100
附录 C preforking 服务器	106
附录 D Spyglass	113

第 1 章

引言

从孩提时起，只要一有机会我就会坐在计算机前。倒不是为了写程序，而是为这台神奇的机器所能做的事而着迷。于是我成为了一名使用 ICQ、Winamp 和 Napster 的计算机用户。

长大后，我将更多的时间耗在了计算机的电子游戏上。起初是第一人称射击游戏，后来大部分时间都在玩即时战略游戏。再后来，我发现这些游戏竟然可以在线玩了！我年轻的时候就是个“computer guy”（计算机小子）：知道如何使用计算机，但对于计算机的工作原理却一无所知。

之所以告诉你我的经历，是因为想让你知道我并不是什么神童。我没有在七岁的时候自学 Basic 语言编程，而当我开始学编程时，也没能反客为主去指点老师并纠正他的错误。

直到大二那年，我才真正爱上了编程这种活动。也许有人会说我这是大器晚成，可是我觉得自己其实比你想象的还要普通。

尽管出于编程本身而热爱编程，但我对于计算机工作原理的理解仍然不够深入。如果那时候你告诉我说我所有的代码都在一个进程中运行，我肯定会将信将疑地对你另眼相看了。

2 | 第1章 引言

幸运的是，我在附近一家互联网初创公司谋到一份不错的差事。这让我有机会在真正的生产系统上从事一些编程工作，给我带来了全新的变化，让我有理由去学习事物究竟是如何运作的。

在这个高流量的生产系统上工作时，我碰到的问题越来越复杂。随着流量和资源需求的增长，我们不得不梳理软件的方方面面来调试和修复未解决的问题。单靠浏览应用程序的代码并不能让我们洞悉程序运作的全景。

我们的应用程序面对的东西可是不少：防火墙、负载均衡器、反向代理，还有 http 缓存。除了应用程序之外也还有很多层：作业队列、数据库服务器，以及统计收集器。每一个应用的组成部分都不尽相同，这本书也不会去教你所有的这些细节。

本书将讲解 Unix 进程方面所有你需要知道的知识，保证会增进你对应用程序任何一部分组件的理解。

托程序调试的福，我不得不深入研究了一些采用 Unix 编程概念的 Ruby 项目，例如 Resque 和 Unicorn。正是这两个项目引导我开始用 Ruby 进行 Unix 编程。

对工作原理有了深入的了解之后，我不仅可以更快地理解并诊断出现的问题，还能够对那些单靠查看代码依然让人摸不着头绪的难题进行排查。

凭借在这些项目中学到的技术，我甚至想到了一些更快、更有效的新方法来解决手头上的问题。好了，关于我的事儿已经说得够多了，让我们开始漫游 Unix 仙境吧。

第 2 章

基础知识

本章介绍了全书涉及的重要概念的相关知识。在深入主要章节之前，建议你先阅读该部分内容。

2.1 干嘛要在意？

自 1970 年起，Unix 编程模型就已经以某种形式存在了。当时，Unix 在贝尔实验室闪亮问世，随之一起诞生的还有 C 程序设计语言，或曰 C 程序设计框架。在随后的数十载中，作为一款可靠、安全和稳定的操作系统，Unix 经受住了时间的考验。

Unix 编程理念及技术并非一时之风，亦不是新近流行的编程语言。它们已超越了编程语言。无论你是使用 C、C++、Ruby、Python、JavaScript、Haskell，还是自己钟意的其他语言，这些技术都有用武之地。

Unix 编程模型已经存在了数十载之久，且大部分都没有改变。过去 40 年里，聪慧的程序员们一直在 Unix 编程模型中借助多种编程语言来解决各类难题，在接下来的 40 年里，他们仍将一如既往。

2.2 驾驭神力!

现在我要提醒你，本书所讲述的概念和技术将赐予你强大的力量。有了它，你能够编写出新的软件，理解现有的复杂软件，甚至能将你的职业生涯提升到新的高度。

请记住：能力越大，责任也越大。在阅读的过程中，我会详细告诉你如何获取这种能力，并避开各种陷阱。

2.3 概述

本书并非参考手册，它更像是一份攻略。因为每一章内容都是基于之前的章节，所以为了最有效地利用本书，你应该按章节顺序逐一读下去。读完全书后，你可以利用章节名来查找有关信息，温故而知新。

书中包含了大量的代码示例。我强烈建议你在 Ruby 解释器中逐一运行这些代码。自己动手调试有助于更深入地理解概念。

一旦读完全书并把玩过那些示例，你肯定想要接触一些更具深度的实际项目。那时，你可以看看书中提到的 Spyglass 项目。

Spyglass 是一个专门为本书编写的 Web 服务器，旨在传授 Unix 编程概念。它采用了你在这里学习到的各种概念，并展示了如何将其应用于真实的项目之中。有关详细介绍请参阅本书最后一章。

2.4 系统调用

要理解系统调用，首先需要了解 Unix 系统的组成，具体来说就是用户空间（userland）和内核。

Unix 系统内核位于计算机硬件之上，它是与硬件交互的中介。这些交互包括通过文件系统进行读/写、在网络上发送数据、分配内存，以及通过扬声器播放音频。鉴于它这些强大的能力，程序不可以直接访问内核，所有的通信都是通过系统调用来完成的。

系统调用为内核和用户空间搭建了桥梁。它规定了程序与计算机硬件之间所允许发生的一切交互。

所有的程序都运行在用户空间。就算是不借助系统调用，你的用户空间程序仍旧能做不少事情：数学运算、字符串操作，以及使用逻辑语句控制程序流程。不过我敢说，如果你打算让程序做些有意思的事情，那就还是得通过系统调用来使用内核。

如果你是一名 C 程序员，那么这些内容你可能已经驾轻就熟了。系统调用可谓是 C 编程的核心。

我估计你像我一样毫无 C 编程经验。你学习的是高级语言编程。在你学会将数据写入文件系统的时候，对于使用了哪些系统调用却并不知情。

总地来说，系统调用允许你的用户空间程序通过内核间接地与计算机硬件进行交互。在本书接下来的章节中，我们会考查一些常见的系统调用。

6 | 第2章 基础知识

2.5 命名法，wtf(2)

学习 Unix 编程的障碍之一就是不知道从哪里查找合适的文档。想不想听点出人意料的回答？其实所有东西都包含在 Unix 手册页 (manpages) 中了。如果你在用 Unix 系统，那么这些东西都已经在你的计算机里了。

要是你以前压根就没有用过手册页，那么你可以在终端中输入命令 `man man` 进行查看。

不错吧？嗯，还算凑合吧。系统调用 API 的手册页在以下两种情况中是很不错的资源。

- (1) 你是一名想知道如何使用特定系统调用的 C 程序员。
- (2) 你想搞明白某个系统调用的用途。

这里假设我们都不是 C 程序员，所以第一种情况就没什么用了，但是第二种情况却是相当受用。

你在整本书中都会看到类似于 `select(2)` 这样的引用。它告诉你可以从哪里找到特定系统调用的手册页。你可能有所不知，Unix 手册页包含了很多节 (section)。

以下是 FreeBSD 和 Linux 系统手册页中最常用的节：

- 节 1：一般命令
- 节 2：系统调用
- 节 3：C 库函数

□ 节 4：特殊文件

因此，节 1 专门用于一般命令，也就是 shell 命令。如果我希望你参考 `find` 命令的手册页，我就会这样写：`find(1)`。这就是说在手册页的节 1 你可以找到 `find` 的使用说明。

如果我希望你参考 `getpid` 系统调用的手册页，我就会这样写：`getpid(2)`。这就是说在手册页的节 2 你可以找到 `getpid` 的使用说明。

为什么手册页还要分这么多节？因为一个命令可能在不止一节中出现，也就是说它既可以作为 shell 命令，也可以作为系统调用。

例如 `stat(1)` 和 `stat(2)`。

要查看手册页的其他节，你可以像这样在命令行上输入：

```
$ man 2 getpid  
$ man 3 malloc  
$ man find # 等同于 man 1 find
```

这种记法并非本书所独创，而是大家参照手册页时通用的一套惯例^①。所以你最好还是从现在开始学习并适应它。

2.6 进程：Unix 之本

进程乃 Unix 系统的基石。为什么要这么说？因为所有的代码都是在进程中执行的。

^① http://en.wikipedia.org/wiki/Man_page#Usage

8 | 第2章 基础知识

例如，当你从命令行运行 `ruby` 时，你的代码就生成了一个新的进程。代码执行完毕，进程也随之退出。

```
$ ruby -e "p Time.now"
```

在你系统中运行的所有代码皆是如此。你知不知道一直运行的 MySQL 服务器？它运行在自己的进程中。你正在使用的电子阅读器软件呢？它也是运行在自己的进程中。那些声嘶力竭通知你收到了新消息的电子邮件客户端呢？别搭理它，接着看书。不过它同样也是运行在自己的进程中。

当你认识到一个进程可以生成并管理其他多个进程的时候，事情就变得有意思了。我们会在后面的部分讨论这些内容。

第 3 章

进程皆有标识

在系统中运行的所有进程都有一个唯一的进程标示符，称为 pid。

pid 并不传达关于进程本身任何信息，它仅仅是一个顺序数字标识。你的进程在内核眼中只是个数字而已。

以下是在 ruby 程序中查看当前 pid 的方法。启动 irb 并输入：

```
# 该行代码会打印出当前 ruby 进程的 pid。这可能是一个 irb 进程，一个 rake  
# 进程，一个 rails 服务器，或是一个普通的 ruby 脚本。  
puts Process.pid
```

pid 是对进程的一种简单通用的描述。因为它与进程内容无关，所以能够被所有的编程语言理解并通过简单的工具来获知。接下来我们会看到如何使用不同的工具来借助 pid 跟踪进程细节。

3.1 交叉参考

为一览全貌，我们可以使用 ps(1) 将 pid 与内核所看到的情形进行对照。保持 irb 会话运行的同时在终端执行下面的命令：

10 第3章 进程皆有标识

```
$ ps -p <pid-of-irb-process>
```

该命令应该会显示出一个名为 irb 的进程，其 pid 与 irb 会话中所显示的一模一样。

3.2 实践领域

仅仅知道 pid 并没什么用处。那应该用在何处呢？

在实际操作中，我们常常会在日志文件中发现 pid。当有多个进程向一个文件中写入日志的时候，知道哪一行日志是由哪一个进程写入的至关重要。在每一行中加入 pid 就能够解决这个问题。

加入 pid 还可以让你通过类似 top(1)或是 lsof(1)的命令，与操作系统之间进行信息的交叉参考。下面列出的是一些来自 Spyglass 服务器启动时的输出信息，每行第一个中括号内注明了该日志的 pid 来源。

```
[58550] [Spyglass::Server] Listening on port 4545
[58550] [Spyglass::Lookout] Received incoming connection
[58557] [Spyglass::Master] Loaded the app
[58557] [Spyglass::Master] Spawns 4 workers. Babysitting now...
[58558] [Spyglass::Worker] Received connection
```

3.3 系统调用

Ruby 的 Process.pid 对应于 getpid(2)。

还有一个全局变量也保存着当前的pid值。你可以访问\$\$来获取。

Ruby从它的前辈们那里继承了这种特性（Perl和bash都支持\$\$），不过我都尽可能地避免使用它。相比\$\$变量，完整地输入Process.pid能够更好地表达你的意图，而且也能够减少那些没见过\$\$的人们的困惑。

第 4 章

进程皆有父

系统中运行的每一个进程都有对应的父进程。每个进程都知道其父进程的标识符（称为 ppid）。

在多数情况下，特定进程的父进程就是调用它的那个进程。假设你是一名 OS X 用户，启动了终端并进入 bash 提示符。因为万物皆为进程，所以你刚才的举动便创建了一个新的终端进程，而新的终端进程又创建了一个 bash 进程。

新创建的 bash 进程的父进程就是终端进程。如果你再从 bash 提示符中调用 ls(1)，那么 bash 便是 ls 进程的父进程。明白了吧！

既然内核只和 pid 打交道，那么就有一种方法可以获得当前父进程的 pid。在 Ruby 中可以这样做：

```
# 注意，与获得当前进程的 pid 相比，只有一个字母之差。  
puts Process.ppid
```

4.1 交叉参考

保持 irb 会话处于运行状态，在终端输入以下命令：

```
$ ps -p <ppid-of-irb-process>
```

上述命令会显示一个带有 pid 的名为 bash (或是 zsh 之类的) 的进程，这个 pid 的值和 irb 会话中得出的值一模一样。

4.2 实践领域

ppid 在实际操作中并没有太多用处。但它在检测守护进程时却发挥着重要的作用，后面的章节会涉及这方面的知识。

4.3 系统调用

Ruby 的 `Process.ppid` 对应着 `getppid(2)`。

第 5 章

进程皆有文件描述符

如同 pid 代表运行的进程，文件描述符代表打开的文件。

5.1 万物皆为文件

Unix 哲学指出，在 Unix 世界中，万物皆为文件。这意味着可以将设备视为文件，将套接字和管道视为文件，将文件也视为文件。

因为所有一切都是文件，所以当讨论一般意义上的文件（包括设备、管道、套接字等）时，我将使用“资源”这个词；当表示传统定义（文件系统中的文件）的时候，将使用“文件”这个词。

5.2 描述符代表资源

无论何时在进程中打开一个资源，你都会获得一个文件描述符编号（file descriptor number）。文件描述符并不会在无关进程之间共享，它只存在于其所属的进程之中。当进程结束后，会和其他由进程所打开的资源一同被关闭。当衍生出一个进程时，文件描述符共享会有一些特殊的含义，更多相关内容会随后详述。

在Ruby中，IO类描述了打开的资源。任意一个IO对象都有一个相关的文件描述符编号。可以使用IO#fileno进行访问。

```
passwd = File.open('/etc/passwd')
puts passwd.fileno
```

输出：

3

进程打开的所有资源都会获得一个用于标识的唯一数字。这便是内核跟踪进程所用资源的方法。

当打开多个资源的时候会怎么样呢？

```
passwd = File.open('/etc/passwd')
puts passwd.fileno

hosts= File.open( '/etc/hosts')
puts hosts.fileno

# 关闭打开的 passwd 文件，释放其文件描述符编号以供后续打开的资源使用。
passwd.close

null = File.open( '/dev/null' )
puts null.fileno
```

输出：

3
4
3

16 | 第5章 进程皆有文件描述符

这个例子有两个关键之处。

- (1) 所分配的文件描述符编号是尚未使用的最小的数值。我们打开的第一个文件 `passwd`, 获得的文件描述符是 3, 接下来打开的文件获得的文件描述符是 4, 这是因为 3 号描述符已经被使用了。
- (2) 资源一旦关闭, 对应的文件描述符编号就又能够使用了。一旦关闭了文件 `passwd`, 它的文件描述符编号就可以再次使用。所以打开文件 `dev/null` 时, 就获得了未使用过的最小值, 也就是 3。

需要注意, 文件描述符只是用来跟踪打开的资源, 已经关闭的资源是没有文件描述符的。

从内核的角度来看, 此举意义重大。一旦资源被关闭, 它就不再需要同硬件层打交道了, 因此内核也就无需再对其进行跟踪。

鉴于以上的论述, 文件描述符有时候也被称作“打开文件描述符”(`open file descriptors`)。这样的叫法有点用词不当, 因为并没有所谓“关闭文件描述符”(`closed file descriptor`)这样的东西。事实上, 试图获取已关闭资源的文件描述符会产生一个异常。

```
passwd = File.open('/etc/passwd')  
puts passwd.fileno  
passwd.close  
puts passwd.fileno
```

输出:

```
3  
-e:4:in `fileno': closed stream (IOError)
```

你可能已经注意到了，当我们打开一个文件，然后查询它的文件描述符编号时，得到的那个最小值是 3。那 0、1 和 2 到哪儿去了？

5.3 标准流

每个 Unix 进程都有三个打开的资源，它们是标准输入（STDIN）、标准输出（STDOUT）和标准错误（STDERR）。

这些标准资源是因为一个很重要的原因而存在，而如今我们已把这个原因视为理所当然。STDIN 提供了一种从键盘或管道中读取输入的通用方法，STDOUT 和 STDERR 提供了一种向显示器、文件、打印机等输出写入内容的通用方法。这是 Unix 的一个创新。

在 STDIN 出现之前，为了能够支持键盘，你得在程序中列入一个键盘驱动程序！而且如果想在屏幕上显示一些信息，你还得知道如何控制所需的屏幕像素。所以我们要感谢标准流！

```
puts STDIN.fileno  
puts STDOUT.fileno  
puts STDERR.fileno
```

输出：

```
0  
1  
2
```

原来前 3 个文件描述符编号在这里。

5.4 实践领域

文件描述符是使用套接字、管道等进行网络编程的核心所在，也是文件系统操作的核心。

因此，每个运行中的进程都在使用文件描述符，它们也是你通过计算机来实现的大多数趣事之关键所在。在之后的章节或是 Spyglass 项目中，你会看到更多例子揭示如何使用文件描述符。

5.5 系统调用

Ruby 的 IO 类中的不少方法都对应着同名的系统调用。其中包括 open(2)、close(2)、read(2)、write(2)、pipe(2)、fsync(2)和 stat(2)。

第 6 章

进程皆有资源限制

在上一章我们了解了这样一个事实：文件描述符代表已打开的资源。你可能注意到当资源没有被关闭时，文件描述符编号一直处于递增状态。这就产生一个问题：一个进程可以拥有多少个文件描述符？

答案取决于你的系统配置，不过重要的一点是：内核为进程施加了某些资源限制。

6.1 找出限制

让我们继续文件描述符的话题。我们可以使用 Ruby 来直接查询所允许的最大的文件描述符编号：

```
p Process.getrlimit(:NOFILE)
```

在我的机器上输出了以下片段：

```
[2560, 9223372036854775807]
```

我们使用名为 `Process.getrlimit` 的方法以及符号 `:NOFILE` 来查询

20 | 第6章 进程皆有资源限制

可以打开的最大文件数。它返回了一个包含两个元素的数组。

数组的第一个元素是文件描述符数量的软限制 (soft limit)，第二个元素是文件描述符数量的硬限制 (hard limit)。

6.2 软限制与硬限制

两者有什么不同吗？问得好。软限制其实算不上一种限制。也就是说如果超出了软限制（在这里指一次打开了超过 2560 个资源），将会产生异常。不过只要你愿意，就可以修改这个限制。

注意，我系统上对于文件描述符数量的硬限制是一个大得不可思议的整数。有可能一次打开这么多文件吗？不大可能。我肯定你在一次性打开那么多的资源之前硬件就已经撑不住了。

系统上的这个数字实际上表示的是无限制。这个值也出现在常量 `Process::RLIMIT_INFINITY` 中。对比一下这两个值你就知道了。所以，在我的系统中，如果我为了自己的需求修改了软限制，那么想打开多少资源都没问题。

所以说任何进程都可以修改自身的软限制，但是对于硬限制呢？通常情况下，只有超级用户能修改硬限制。不过如果进程有足够的权限，那么也可以修改硬限制。如果你对更改系统级别的限制感兴趣，可以查看 `sysctl(8)`。

6.3 提高软限制

让我们试着提高当前进程的软限制：

```
Process.setrlimit(:NOFILE, 4096)
p Process.getrlimit(:NOFILE)
```

输出：

```
[4096, 4096]
```

可以看到我们为可打开的文件数量设置了一个新限制，再次查询可以发现硬限制和软限制都被设成了新值 4096。

还可以选择给 `Process.setrlimit` 传递第 3 个参数来指定新的硬限制，假设我们有权这么做。注意，像上面的代码片段中那样降低硬限制是不可逆转的——一旦调低，就没法再调回去。

接下来的例子是一个将系统资源的软限制提高至硬限制水平，即所允许的最大值的常用方法。

```
Process.setrlimit(:NOFILE, Process.getrlimit(:NOFILE)[1])
```

6.4 超出限制

要注意的是如果超出了软限制，则会抛出 `Errno::EMFILE` 异常。

```
# 将可以打开的最大文件数设置为 3。我们知道这会超出最大限制,
# 因为标准流已经占用了前 3 个文件描述符。
Process.setrlimit(:NOFILE, 3)
```

```
File.open('/dev/null')
```

输出：

22 | 第 6 章 进程皆有资源限制

```
Errno::EMFILE: Too many open files - /dev/null
```

6.5 其他资源

你可以使用同样的方法来检查及修改其他的资源限制。一些常见的例子如下：

```
# 当前用户所允许的最大并发进程数。  
Process.getrlimit(:NPROC )
```

```
# 可以创建的最大的文件。  
Process.getrlimit(:FSIZE )
```

```
# 用于进程栈的最大段的大小。  
Process.getrlimit(:STACK )
```

请查看 `Process.getrlimit` 的文档^①以获取完整的可用选项列表。

6.6 实践领域

多数程序通常并不需要修改系统资源限制。但对于一些特殊的工具，这可是至关重要的。

其中一种情况就是某个进程需要处理成千上万的并发网络连接。例如 http 性能测试工具 `httpperf(1)`。像 `httpperf --hog --server www --num-conn 5000` 这样的命令会使得 `httpperf(1)` 创建 5000 个并发连接。由于默认的软限制，我的系统显然会出现问题。因此在进行正确的测

^① <http://www.ruby-doc.org/core-1.9.3/Process.html#method-c-setrlimit>

试之前，httpperf(1)需要调高对应的软限制。

另一个需要进行系统资源限制的实际用例是，在执行第三方代码并需要对其施加一些运行限制时。你可以对代码所属的进程设置限制，并取消修改这些限制的权限，这样就能确保其无法使用超出许可范围的资源数量。

6.7 系统调用

Ruby 的 `Process.getrlimit` 和 `Process.setrlimit` 分别对应于 `getrlimit(2)` 及 `setrlimit(2)`。

第 7 章

进程皆有环境

这里所说的环境，指的是“环境变量”。环境变量是包含进程数据的键-值对（key-value pairs）。

所有进程都从其父进程处继承环境变量。它们由父进程设置并被子进程所继承。每一个进程都有环境变量，环境变量对于特定进程而言是全局性的。

下面是一个简单的例子，演示了在 `bashshell` 中设置一个环境变量，启动一个 Ruby 进程并读取该环境变量。

```
$ MESSAGE='wing it'ruby -e "puts ENV['MESSAGE']"
```

`VAR=value` 这种语法是 `bash` 设置环境变量的方法，也可以通过在 Ruby 中使用 `ENV` 常量来实现。

```
# 用更少的输入完成同一件事情!
ENV['MESSAGE'] = 'wing it'
system "echo $MESSAGE"
```

这两个例子都会打印出：

wing it

在 `bash` 中，环境变量通过使用`$VAR`来访问。从例子中你可以看到，环境变量能够用于在执行不同语言（在这里是 `bash` 和 `ruby`）的进程之间共享状态。

7.1 这是个散列吗？

尽管 `ENV` 使用了散列式的存取器 API，它其实并非 Hash。例如，它实现了 `Enumerable` 和部分 Hash API，但并非全部。像 `merge` 这样的重要方法就没有实现。因此你可以进行 `ENV.has_key?`这样的操作，但是别指望所有的散列操作都能奏效。

```
puts ENV['EDITOR']
puts ENV.has_key?('PATH')
puts ENV.is_a?(Hash)
```

输出：

```
vim
true
false
```

7.2 实践领域

在实践操作中，环境变量有很多用法。下面是 Ruby 社区中一些常见的用法：

26 | 第 7 章 进程皆有环境

```
$ RAILS_ENV=production rails server  
$ EDITOR=mate bundle open actionpack  
$ QUEUE=default rake resque:work
```

环境变量经常作为一种将输入传递到命令行程序中的通用方法。所有的终端（Unix 或 Windows）均已支持环境变量，而且大多数程序员对此颇为熟悉。比起解析命令行选项，使用环境变量的开销通常更小一些。

7.3 系统调用

系统调用不能直接操作环境变量，不过 C 库函数 `setenv(3)` 和 `getenv(3)` 却可以完成这样的工作。相关内容请参考 `environ(7)`。

第 8 章

进程皆有参数

所有进程都可以访问名为 `ARGV` 的特殊数组。其他编程语言可能在实现方式上略微不同，但是都会有 `argv`。

`argv` 是 `argument vector` 的缩写，换句话说就是一个参数向量或数组。它保存了在命令行中传递给当前进程的参数。下面是一个检查 `ARGV` 的例子，并传递了一些简单的选项。

```
$ cat argv.rb
p ARGV
$ ruby argv.rb foo bar -va
["foo", "bar", "-va"]
```

8.1 这是个数组！

在上一章中我们得知 `ENV` 并不是 `Hash`，而这里的 `ARGV` 只是一个 `Array`。你可以随意地从中添加、删除和更改元素。但如果它仅仅代表的是从命令行传入的参数，那就没有必要修改它。

有些库会读取 `ARGV` 来解析命令行选项。你可以在这些库读取 `ARGV`

之前，通过编程的方式对其进行修改，以在运行时更改命令选项。

8.2 实践领域

ARGV 最常见的用例大概就是将文件名传入程序。写一个在命令行中接收一个或多个文件名作为参数并进行处理的程序，这种做法很常见。

其他的一些常见用例是解析命令行输入。有很多 Ruby 库可以用来处理命令行输入。在标准库中就有一个叫做 **optparse** 的库。

既然现在你了解了 **ARGV** 的工作原理，对于一些简单的命令行选项，就可以跳过额外的开销来手动进行解析。如果你想支持若干标志，可以直接使用数组操作来实现。

```
# 用户请求帮助了吗?  
ARGV.include?('--help')  
# 获取-c 选项的值  
ARGV.include?('-c') && ARGV[ARGV.index('-c') + 1]
```

第 9 章

进程皆有名

Unix 进程几乎没有什么固有的方法来获悉彼此的状态。

程序员们对此已经找到了解决之道，并发明了像日志文件这样的方法。日志文件使得进程可以通过向文件系统写入信息的方式来了解彼此的状态信息，不过这种操作属于文件系统层面，而非进程本身所固有的。

与此类似，进程可以借助网络来打开套接字同其他进程进行通信。但是这也并非运作在进程层面，因为它是依靠网络来实现的。

有两种运作在进程自身层面上的机制可以用来互通信息。一个是进程名称，另一个是退出码。

9.1 进程命名

系统中每一个进程都有名称。举例来说，当你启动一个 `irb` 会话，对应的进程就获得了“`irb`”的名称。进程名的妙处在于它可以在运行期间被修改并作为一种通信手段。

在 Ruby 中，你可以在变量 `$PROGRAM_NAME` 中获得当前进程的名称。同样地，你也可以给这个全局变量赋值来修改当前进程的名称。

```
puts $PROGRAM_NAME

10.downto(1) do |num|
  $PROGRAM_NAME = "Process: #{num}"
  puts $PROGRAM_NAME
end
```

输出：

```
irb
Process: 10
Process: 9
Process: 8
Process: 7
Process: 6
Process: 5
Process: 4
Process: 3
Process: 2
Process: 1
```

你可以启动一个 `irb` 会话，打印 `pid` 并改变进程名称，然后使用 `ps(1)` 查看修改后的系统效果。就算是一个趣味练习吧！

可惜，这个全局变量（及其对应的`$0`）是Ruby提供的唯一一个实现该特性的机制。再没有其他更直观的方法来修改当前的进程名称了。

9.2 实践领域

请阅读附录 1 “Resque 如何管理进程” 来了解如何在真实项目中应用本章所学的内容。

第 10 章

进程皆有退出码

当进程即将结束时，它还有最后一线机会留下自身的信息：退出码。所有进程在退出的时候都带有数字退出码（0-255），用于指明进程是否顺利结束。

按惯例，退出码为 0 的进程被认为是顺利结束；其他的退出码则表明出现了错误，不同的退出码代表不同的错误。

尽管退出码通常用来表明不同的错误，它们其实是一种通信途径。你只需以适合自己程序的方式来处理各种进程退出码，便打破了传统。

坚持“退出码 0 代表顺利结束”的传统通常是一个不错的主意，这样你的程序就能同其他的 Unix 工具顺畅合作。

如何退出进程

在 Ruby 中有多种方法可以退出进程，每一种都有不同的作用。

`exit`

退出进程最简单的方法是使用 `Kernel#exit`。虽然你的脚本没有明

32 | 第 10 章 进程皆有退出码

确指出使用 `exit` 语句结束，但实际上它在幕后用的就是这种方式。

```
#这将使你的程序携带顺利状态码(0)退出。  
exit  
  
#你可以给这个方法传递一个定制的退出码。  
exit 22  
  
#当 Kernel#exit 被调用时，在退出之前，Ruby 会调用由 Kernel#at_exit  
#所定义的全部语句块。  
at_exit { puts 'Last!' }  
exit
```

输出

```
Last!
```

```
exit!
```

`Kernel#exit!`几乎和 `Kernel#exit`一模一样，但有两处关键不同。第一处是它将错误状态码设置为默认的 1，另一处是它不会调用任何由 `Kernel#at_exit` 所定义的代码块。

```
# 这使得程序携带状态码 1 退出。  
exit!  
  
# 你仍然可以传递一个退出码。  
exit! 33  
  
# 这个语句块永远不会被调用。  
at_exit { puts 'Silence!' }  
exit!
```

`abort`

`Kernel#abort` 提供了一种从错误进程中退出的通用方法。

Kernel#abort 会将当前进程的退出码设置为 1。

```
# 携带退出码 1 退出。  
abort  
  
#你可以传递一条消息给 Kernel#abort。在进程退出之前，该消息会被打印到  
#STDERR。  
abort "Something went horribly wrong."  
  
# 当使用 Kernel#abort 时，会调用 Kernel#at_exit 语句块。  
at_exit { puts 'Last!' }  
abort "Something went horribly wrong."
```

输出：

```
Something went horribly wrong.  
Last!
```

raise

另一种结束进程的方法是使用一个未处理的异常。在生产环境中你绝对不想出现这种情况，不过在开发和测试环境中，这几乎是不可避免的。

和之前那些方法不同的是，Kernel#raise 不会立刻结束进程。它只是抛出一个异常，该异常会沿着调用栈向上传递并可能会得到处理。如果没有代码对其进行处理，那么这个未处理的异常将会终结该进程。

以此种方式结束的进程仍然会调用 at_exit 处理程序，并向 STDERR 打印出异常消息和回溯。

```
# 类似于 abort，一个未处理的异常会将退出码设置为 1。  
raise 'hell'
```

第 11 章

进程皆可衍生

11.1 Luke，使用 fork(2)

衍生（forking^①）是 Unix 编程中最强大的概念之一。fork(2)系统调用允许运行中的进程以编程的形式创建新的进程。这个新进程和原始进程一模一样。

到目前为止，我们都是通过在终端中运行的方式来创建新的进程。我们也提到了低层操作系统调用 fork(2)的工作原理。

进行衍生时，调用 fork(2)的进程被称为“父进程”，新创建的进程被称为“子进程”。

子进程从父进程处继承了其所占用内存中的所有内容，以及所有属于父进程的已打开的文件描述符。让我们来简单回顾一下从前三章中学到的有关子进程的知识。

因为子进程是一个全新的进程，所以它拥有自己唯一的 pid。

^① fork 一词在 Unix/Linux 编程中出现的频率非常高。但是目前并没有统一的译法。在本书中，将此词译为“衍生”。——译者注

子进程的上层进程(parent)显然就是它的父进程。因此子进程的 ppid 就是调用 fork(2)的进程的 pid。

在 fork(2)调用时，子进程从父进程处继承了所有的文件描述符，也获得了父进程所有的文件描述符的编号。这样，两个进程就可以共享打开的文件、套接字，等等。

子进程继承了父进程内存中的所有内容。借助这种方式，一个进程，比如说 Rails，可以将一个 500MB 的代码库（ codebase ）装入内存，然后该进程衍生出两个子进程，这些子进程实际上各自享有一份已载入内存代码库的副本。

fork 调用几乎瞬间就可以返回，这样我们就得到了 3 个进程，每个进程都可以使用 500MB 的内存空间。这对于想要在内存中载入多个应用程序实例而言简直就是完美的解决方案。因为只需要一个进程来载入应用程序，而且进程衍生的速度很快，所以这种方法比分别载入 3 个应用程序实例要快得多。

子进程可以随意更改其内存内容的副本，而不会对父进程造成任何影响。在下一章，我们将讨论“写时复制”(copy-on-write) 技术，以及该技术在进程衍生时如何影响内存内容。

下面通过一个令人费解的例子开始我们在 Ruby 中的进程衍生之旅。

```
if fork
  puts "entered the if block"
else
  puts "entered the else block"
end
```

输出：

36 | 第 11 章 进程皆可衍生

```
entered the if block  
entered the else block
```

到底怎么回事？对 `fork` 方法的调用采用了我们都熟悉的 `if` 结构，但是结果却是南辕北辙。这段代码不知道怎么回事，竟然同时执行了 `if` 语句的 `if` 和 `else` 代码块。

这一切其实不难理解。对于 `fork` 方法的一次调用实际上返回了两次。记住，`fork` 创造了一个新进程。所以它在调用进程（父进程）中返回一次，在新创建的进程（子进程）中又返回一次。

如果我们把 `pid` 打印出来的话，上面的例子就更显而易见了。

```
puts "parent process pid is #{Process.pid}"  
  
if fork  
  puts "entered the if block from #{Process.pid}"  
else  
  puts "entered the else block from #{Process.pid}"  
end
```

输出：

```
parent process is 21268  
entered the if block from 21268  
entered the else block from 21282
```

现在一切都豁然开朗了：`if` 语句块中的代码是由父进程执行的，而 `else` 语句块中的代码是子进程执行的。子进程执行完 `else` 语句块之后退出，父进程则继续运行。

这种执行顺序可不是乱来的，它和 `fork` 方法的返回值有关。在子进程中，`fork` 返回 `nil`。因为 `nil` 为假，所以子进程便执行了 `else` 语句块中的代码。

在父进程中，`fork` 返回新创建的子进程的 `pid`。因为此整数值为真，所以父进程执行的是 `if` 语句块中的代码。

只需打印 `fork` 调用的返回值，这个概念便能清晰地展现出来。

```
puts fork
```

输出

```
21423  
nil
```

这里我们获得了两个不同的返回值。第一个返回值是新创建的子进程的 `pid`，这个值来自于父进程；第二个返回值是来自于子进程的 `nil`。

11.2 多核编程?

间接说来，就是这样。通过生成新的进程，你的代码可以（不能完全保证）被分配到多个 CPU 核心中。

在配备了 4 个 CPU 的系统中，如果衍生出 4 个新进程，那么这些进程会分别由不同的 CPU 来处理，从而实现多核并发（multicore concurrency）。

然而，并不保证它们会并行操作。在繁忙的系统中，有可能所有 4 个进程都由同一个 CPU 来处理。

38 | 第 11 章 进程皆可衍生

`fork(2)` 创建了一个和旧进程一模一样的新进程。所以如果一个使用了 500MB 内存的进程进行了衍生，那么就有 1GB 的内存被占用了。

重复同样的操作十次，你很快就会耗尽所有的内存。这通常被称为“`fork` 炸弹”（`fork bomb`）。使用并发执行前，请务必确保你知道这样做的后果。

11.3 使用 block

在上面的例子中，我们使用 `if/else` 结构演示了 `fork` 的用法。其实也可以通过 `block` 来使用 `fork`，这种方法在 Ruby 中更常见。

如果你将一个 `block` 传递给 `fork` 方法，那么这个 `block` 将在新的子进程中执行，而父进程则会跳过 `block` 中的内容。子进程执行完 `block` 之后就会退出，它并不会像父进程那样执行随后的代码。

```
fork do
  # 此处的代码仅在子进程中执行
end

# 此处的代码仅在父进程中执行
```

11.4 实践领域

请参看附录或 Spyglass 项目来学习 `fork(2)` 的实践用法。

11.5 系统调用

Ruby 的 `Kernel#fork` 对应于 `fork(2)`。

第 12 章

孤儿进程

12.1 失控

在运行上一章中的例程时你可能就已经注意到了，如果涉及子进程，就不能再像我们已经习惯的那样从终端来控制一切。

当通过终端启动单个进程时，通常只有这个进程向 STDOUT 写入，从键盘获取输入或是侦听 Ctrl-C 以待退出。

一旦进程衍生出了子进程，这一切就变得不那么简单了。如果你按下 Ctrl-C，哪一个进程应该退出？是全部退出还是只有父进程退出？

了解这些问题还是有好处的，因为实际上很容易创建孤儿进程。

```
fork do
  5.times do
    sleep 1
    puts "I'm an orphan!"
  end
end

abort "Parent process died..."
```

如果从终端运行这个程序，你会注意到父进程结束后，立刻返回到终端命令提示符下，此时终端被子进程输出到 STDOUT 的内容所重写！在进行进程衍生的时候，就会发生这些莫名其妙的事。

12.2 弃子

当父进程结束之后，子进程会怎样？

简短的回答就是“安然无恙”。也就是说，操作系统并不会对子进程区别对待。因此父进程结束后，子进程照常继续运行。父进程可不会带着子进程同归于尽。

12.3 管理孤儿

那么还能够对孤儿进程进行管理吗？

对于这个问题我们接触得有点早了，不过这涉及两个有意思的概念。

第一个是守护进程。守护进程是一种长期运行的进程，为了能够一直保持运行，它们有意作为孤儿进程存在。在后面我们会对其进行详述。

另一个是与脱离终端会话的进程进行通信。你可以使用 Unix 信号来做到这一点。同样我们会在随后的章节中详述。

接下来我们很快就会讲述如何正确管理和控制子进程。

第 13 章

友好的进程

我们暂时把代码搁在一边，来讨论一个更高层次的概念，以及在不同的 Rudy 实现中如何处理这个概念。

13.1 对 CoW 好点

正如我们在讲解进程衍生那章所提到的那样，`fork(2)` 创建了一个和父进程一模一样的子进程。它包含了父进程在内存中的一切内容。

实实在在地复制所有的数据所产生的系统开销不容小觑，因此现代的 Unix 系统采用写时复制（copy-on-write，CoW）的方法来克服这个问题。

顾名思义，CoW 将实际的内存复制操作推迟到了真正需要写入的时候。

所以说父进程和子进程实际上是在共享内存中的数据，直到它们其中的某一个需要对数据进行修改，届时才会进行内存复制，使得两个进程保持适当的隔离。

42 | 第 13 章 友好的进程

```
arr = [1,2,3]

fork do
  # 此时子进程已经完成初始化。
  # 借助 CoW，子进程并不需要复制变量 arr，因为它并没有修改任何共享变量，
  # 因此可以继续从和父进程同样的内存位置中进行读取。
  p arr
end

arr = [1,2,3]
fork do
  # 此时子进程已经完成初始化。
  # 由于 CoW，变量 arr 并不会被复制。
  arr << 4
  # 上面的代码修改了数组，因此在进行修改之前需要为子进程创建一个该数组的
  # 副本。父进程中的这个数组并不会受到影响。
end
```

在使用 fork(2)时，这可是个不得了的优势，因为它节省了资源。由于不需要对父进程的内存内容进行任何的复制，这意味着 fork(2)的执行速度很快。也同样意味着子进程只获得了它所需要的那部分数据的副本，其余的部分依然可以共享。

CoW 固然不错，但可惜 MRI 或 Rubinius 对其并不支持。

要想 CoW 正常运作，程序需要以一种 CoW 友好的方式来编写。也就是说，它们得用一种能够使 CoW 成为可能的方式来管理内存。MRI 和 Rubinius 并不是采用这种方式编写的。

为何不可？

MRI 的垃圾收集器使用了一种“标记-清除”(mark-and-sweep)的算法。简单地说，这意味着当垃圾收集器被调用时，它必须对每

个已知的对象进行迭代并写入信息，指出该对象是否应该被回收。关键在于垃圾收集器每运行一次，内存中的所有对象都会被写入信息。

因此，在进行衍生之后，首次进行垃圾收集的时候，写时复制所带来的好处会被撤销。

这就是创建 Ruby 企业版(Ruby Enterprise Edition)^①的主要原因之一。REE 是 CoW 友好的。对于此问题的深入讨论以及 Ruby 企业版是如何解决的，请参阅 Google Tech Talk: Building a More Efficient Ruby Interpreter^②。

13.2 MRI/RBX 用户

对于其他的 Ruby 虚拟机用户来说，这又意味着什么？

很简单，在衍生进程的时候，你享受不到 CoW 带来的好处。子进程需要获取调用进程所占有的内存内容的完整副本。

如果你编写了一个大量依赖于 fork(2)的程序，那么你要认真考虑一下 Ruby 企业版了。展望未来，MRI 主干现在已经拥有了使得垃圾收集与 CoW 和睦相处的补丁^③。所以 Ruby2.0 会提供 CoW 友好的垃圾收集器。

① <http://www.rubyenterpriseedition.com/>

② <http://www.youtube.com/watch?v=ghLCtCwAKqQ>

③ <http://bugs.ruby-lang.org/issues/5839>

第 14 章

进程可待

在迄今为止的 fork(2)的例子中，我们都是让父进程与子进程一道运行。有时候这会导致一些奇怪的结果，比如当父进程先于子进程退出时。

这种情境只适用于一种用例——即发即弃 (fire and forget)。如果你希望让子进程异步地处理其他事务，而父进程按原计划运行，那么这种方式便能派得上用场。

```
message = 'Good Morning'  
recipient = 'tree@mybackyard.com'  
  
fork do  
  # 在这个假想的例子中，父进程衍生出一个子进程来负责将数据发送给统计收集器。  
  # 父进程同时继续进行自己实际的数据发送工作。  
  
  # 父进程不希望自身被这项任务所拖缓，即便任务出于某种原因失败，  
  # 父进程也不会受到影响。  
  StatsCollector.record message, recipient  
end  
  
#发送信息给接收方
```

14.1 看顾 (Babysitting)

对于其他多数涉及 fork(2)的用例来说，你会希望有一些能够监视子进程动向的方法。在 Ruby 中，`Process.wait` 就提供了这么一种技术。我们将上一章那个会产生孤儿进程的例子进行改写并执行，结果并不出人意料。

```
fork do
  5.times do
    sleep 1
    puts "I am an orphan!"
  end
end

Process.wait
abort "Parent process died..."
```

这一次输出如下：

```
I am an orphan!
Parent process died...
```

不仅如此，直到所有输出都被打印出来之后，控制才会返回到终端。

那么 `Process.wait` 究竟做了什么？`Process.wait` 是一个阻塞调用，该调用使得父进程一直等到它的某个子进程退出之后才继续执行。

14.2 Process.wait 一家子

我在上文中提到了一处关键的地方：`Process.wait` 会一直保持阻塞，直到其任意一个子进程退出为止。如果父进程拥有不止一个子进程，并且使用了 `Process.wait`，那么你就需要知道究竟是哪个子进程退出了。这时可以使用返回值来解决这个问题。

`Process.wait` 返回退出的那个子进程的 pid。来看一下。

```
# 创建 3 个子进程。
3.times do
  fork do
    # 每个子进程随机休眠一段时间（不超过 5 秒）。
    sleep rand(5)
  end
end

3.times do
  # 等待每个子进程退出并打印其返回的 pid。
  puts Process.wait
end
```

14.3 使用 Process.wait2 进行通信

等等！`Process.wait` 还有个表亲，它叫做 `Process.wait2`！

干嘛起一个这么容易混淆的名字？不过你若知道 `Process.wait` 返回一个值（pid），`Process.wait2` 返回两个值（pid, status）的话，那一切就都说得通了。

14.3 使用 Process.wait2 进行通信 | 47

通过退出码，这些状态可以用作进程间的通信。在讲解退出码的那章我们曾提到可以使用退出码来编码信息，以供其他进程使用。Process.wait2 则可以让你直接访问这些信息。

从 Process.wait2 返回的 status 是 Process::Status 的一个实例。它包含大量有用的信息，可让我们获知某个进程是如何退出的。

```
# 创建 5 个子进程。
5.times do
  fork do
    # 每个子进程生成一个随机数。如果是偶数，就以退出码 111 退出，否则以
    # 退出码 112 退出。
    if rand(5).even?
      exit 111
    else
      exit 112
    end
  end
end

5.times do
  # 等待子进程逐个退出。
  pid, status = Process.wait2

  # 如果子进程的退出码是 111，那么我们就知道它生成的是偶数。
  if status.exitstatus == 111
    puts "#{pid} encountered an even number!"
  else
    puts "#{pid} encountered an odd number!"
  end
end
```

进程间的通信既不需要文件系统，也不需要网络！

14.4 等待特定的子进程

再等等！`Process.wait` 的表亲也有两个表亲：`Process.waitpid` 和 `Process.waitpid2`。

你可能已经猜到它们两个是做什么的了。它们的功能和 `Process.wait` 以及 `Process.wait2` 一样，不过不是等待任意的子进程退出，而是只等待由 pid 指定的子进程退出。

```
favourite = fork do
  exit 77
end

middle_child = fork do
  abort "I want to be waited on!"
end

pid, status = Process.waitpid2 favourite
puts status.exitstatus
```

尽管`Process.wait`和`Process.waitpid`的行为看似不同，但千万别上当！它们实际上就是同一事物的两个名字。两者接受同样的参数，行为也一样。

你可以传递一个pid给`Process.wait`，让它等待某个特定的子进程。你也可以将-1作为pid传递给`Process.waitpid`，让它等待任意的子进程。

对于`Process.wait2`和`Process.waitpid2`来说，情况也是一样。

就如同`Process.pid`和`$$`那样，我认为程序员在可能的情况下使

用现有的工具来展现自己的意图才是重要的。尽管这些方法相同，但是在等待任意子进程的时候，你应该使用 `Process.wait`；当你等待特定进程的时候，就使用 `Process.waitpid`。

14.5 竞争条件

在查看这些简单的代码示例的时候，你也许对竞争条件（race conditions）感到疑惑。

当另一个子进程退出时，处理某个退出进程的代码还在运行，这时候会怎么样？如果我还没来得及从 `Process.wait` 返回，另一个进程也退出了，这又会怎样？让我们来看看：

```
# 创建两个子进程。
2.times do
  fork do
    # 两个子进程立即退出。
    abort "Finished!"
  end
end

# 父进程等待第一个子进程，然后休眠 5 秒钟。
# 其间第二个子进程也退出，不再运行。
puts Process.wait
sleep 5

# 父进程会再等待一次，让人惊叹的是，第二个子进程的退出信息会被加入队列并
# 在此返回。
puts Process.wait
```

如你所见，这项技术能够避免竞争条件。内核将退出的进程信息加入

50 | 第 14 章 进程可待

队列，这样一来父进程就总是能够依照子进程退出的顺序接收到信息。

因此，即便父进程处理每个退出子进程的速度缓慢，当它准备妥当的时候，也总能获取到每个子进程的退出信息。

注意，如果不存在子进程，那么调用**Process.wait**的任一变体都会抛出**Errno::ECHILD**异常。所以最好记录一下到底创建了多少个子进程，以免出现这种异常。

14.6 实践领域

关注子进程的理念是一般的 Unix 编程模式的核心。这种模式有时被称为看顾进程，*master/worker* 或者 *preforking*。

此模式的核心是这样一种概念：你有一个衍生出多个并发子进程的进程，这个进程看管着这些子进程，确保它们能够保持响应，并对子进程的退出做出回应，等等。

例如 Web 服务器 Unicorn^①就采用了这种模式。你可以告诉服务器自己希望启动多少个工作进程，比方说 5 个吧。

那么 *unicorn* 进程就会启动并衍生出 5 个子进程来处理 Web 请求。父进程（或者说是主进程）同每个子进程维持联系，并保证所有的子进程能够保持响应。

① <http://unicorn.bogomips.org>

这种模式兼顾了并发性和可靠性。阅读本书末的附录，以获悉更多有关 Unicorn 的信息。

欲了解这项技术的其他用法，请查看 Spyglass 项目的 Lookout 类。

14.7 系统调用

Ruby 的 `Process.wait` 及其表亲都对应于 `waitpid(2)`。

第 15 章

僵尸进程

在上一章之初，我们考查了一个例子，该例子使用子进程以即发即弃的方式来异步地处理任务。我们需要重新审视这个例子，确保我们正确地清除了子进程，以免它变成僵尸！

15.1 等待终有果^①

在上一章中，我展示了内核会将已退出的子进程的状态信息加入队列。所以即便你在子进程退出很久之后才调用 `Process.wait`，依然可以获取它的状态信息。我肯定你现在已经发现问题了……

内核会一直保留已退出的子进程的状态信息，直到父进程使用 `Process.wait` 请求这些信息。如果父进程一直不发出请求，那么状态信息就会被内核一直保留着。因此创建即发即弃的子进程，却不去读取状态信息，便是在浪费内核资源。

如果你不打算使用 `Process.wait`（或下一章将要介绍的技术）来等

^① 原文是 Good Things Come to Those Who wait(2)。作者巧妙地模仿了英文中的一句习语 Good Things Come to Those Who Wait。——译者注

待某个子进程退出，那么你就需要分离这个子进程。下面是上一章那个即发即弃的例子，代码已经修改，使之能够正确分离子进程：

```
message = 'Good Morning'  
recipient = 'tree@mybackyard.com'  
  
pid = fork do  
  # 在这个人人为设计的例子中，父进程衍生出一个子进程来负责将数据发送给统计  
  # 收集器。同时，父进程继续进行自己实际的数据发送工作。  
  
  # 父进程不希望自身被这项任务所拖缓，即便任务出于某种原因失败，  
  # 父进程也不会受到影响。  
  StatsCollector.record message, recipient  
end  
  
# 这一行代码确保进行统计收集的进程不会变成僵尸。  
Process.detach(pid)
```

`Process.detach` 做了些什么？它不过是生成了一个新线程，这个线程的唯一工作就是等待由 `pid` 所指定的那个子进程退出。这确保了内核不会一直保留那些我们不需要的状态信息。

15.2 僵尸长什么样子？

```
# 创建一个子进程，1 秒钟之后退出。  
pid = fork { sleep 1 }  
# 打印出该子进程的 pid。  
puts pid  
# 让父进程长眠，以便于我们检查子进程的进程状态信息。  
sleep
```

利用上面的代码片段所打印出的 `pid`，在终端执行下列命令，便会输

54 | 第 15 章 僵尸进程

出僵尸进程的状态信息。状态为“z”或“Z+”就表示这是一个僵尸进程。

```
ps -ho pid,state -p [pid of zombie process]
```

15.3 实践领域

请注意，任何已经结束的进程，如果它的状态信息一直未能被读取，那么它就是一个僵尸进程。所以任何子进程如果在结束之时其父进程仍在运行，那么这个子进程很快就会成为僵尸。一旦父进程读取了僵尸进程的状态信息，那么它就不复存在，也就不再消耗内核资源。

采用即发即弃的方式衍生出子进程，却对其状态信息不理不问，这种情形极其少见。如果需要在后台执行工作，更为常见的做法是采用一个专门的后台排队系统。

有一个叫做 `spawn`^① 的 Rubygem 就提供了这样的功能。除了具备通用的进程/线程 API 之外，它还能确保正确地分离即发即弃的进程。

15.4 系统调用

`Process.detach` 并没有对应的系统调用，因为在 Ruby 中它仅仅通过线程和 `Process.wait` 来实现。在 Rubinius^② 中的实现则是简洁十足。

① <https://github.com/tra/spawn>

② <https://github.com/rubinius/rubinius/blob/c6e8e33b37601d4a082ddcbbd60a568767074771/kernel/common/process.rb#L377-395>

第 16 章

进程皆可获得信号

在上一章我们学习了 `Process.wait`。它为父进程提供了一种很好的方式来监管子进程。但它是一个阻塞调用：直到子进程结束，调用才会返回。

一个繁忙的父进程会怎么做？可不是每个父进程都有闲暇一直等着自己的子进程结束。对此倒是有一个解决方案，这就是我们要介绍的 Unix 信号。

16.1 捕获 SIGCHLD

我们沿用上一章那个简单的例子，并针对一个繁忙的父进程对其进行改写。

```
child_processes = 3
dead_processes = 0
# 衍生出 3 个子进程。
child_processes.times do
  fork do
    # 各自休眠 3 秒钟。
```

56 | 第 16 章 进程皆可获得信号

```
sleep 3
end
end

# 父进程忙于执行一些密集的数学运算，但是仍想知道子进程何时退出。

# 通过捕获:CHLD 信号，内核会提醒父进程它的子进程何时退出。
trap(:CHLD) do
  # 由于 Process.wait 将它获得的数据都加入了队列，因此可以在此进行查询，
  # 因为我们知道其中一个子进程已经退出了。

  puts Process.wait
  dead_processes += 1
  # 一旦所有的子进程统计完毕，就直接退出。
  exit if dead_processes == child_processes
end

# 父进程需要执行的密集数学运算。
loop do
  (Math.sqrt(rand(44)) ** 8).floor
  sleep 1
end
```

16.2 SIGCHLD 与并发

在继续往下进行之前，我有一个忠告：信号投递是不可靠的。我的意思是说如果你的代码正在处理 CHLD 信号，这时候另一个子进程结束了，那么你未必能收到第二个 CHLD 信号。

这会导致上面的代码片段产生不一致的后果。有时候时机正好，就会一切顺利；而有时候，你都不知道某个子进程已经结束了。

如果同一个信号在极短的间隔内被多次接收到，就会出现这种情况，

不过你至少总能接收到一次信号。对于在 Ruby 中要处理的其他信号，这个忠告同样成立。请继续阅读来获知更多这方面的内容。

要正确地处理 CHLD，你必须在一个循环中调用 `Process.wait`，查找所有已经结束的子进程，这是因为在进入信号处理程序之后，你可能会收到多个 CHILD 信号。但是，`Process.wait` 不是一个阻塞调用吗？如果只有一个已结束的子进程，而我却调用了两次 `Process.wait`，又该如何避免阻塞整个进程呢？

现在我们得派上 `Process.wait` 的第二个参数了。在上一章我们将一个 pid 传给 `Process.wait` 作为首个参数，不过它也可以将标志作为第二个参数。这样的标志可以告诉内核，如果没有子进程退出，那么就不需要进行阻塞。这恰恰就是我们需要的东西！

常量 `Process::WNOHANG` 描述了这个标志的值，它可以像这样来使用：

```
Process.wait(-1, Process::WNOHANG)
```

够简单吧！

下面将本章开头的那个代码片段进行重写，重写后的代码不会“错过”任何子进程的死亡：

```
child_processes = 3
dead_processes = 0
# 衍生出 3 个子进程。
child_processes.times do
  fork do
    # 各自休眠 3 秒钟。
    sleep 3
```

58 | 第 16 章 进程皆可获得信号

```

    end
end

# 设置$stdout 的 sync，使得在 CHLD 信号处理程序中不会对#puts 调用进行缓冲。
# 如果信号处理程序在调用#puts 之后被中断，则会引发一个 ThreadError。
# 如果你的信号处理程序需要执行 IO 操作，那么这不失为一个好方法。
$stdout.sync = true

# 父进程忙于执行一些密集的数学运算，但是仍想知道子进程何时退出。

# 通过捕获:CHLD 信号，内核会提醒父进程它的子进程何时退出。
trap(:CHLD) do
  # 由于 Process.wait 将它获得的数据都加入了队列，因此可以在此进行查询
  # 因为我们知道其中一个子进程已经退出了。

  # 我们需要执行一个非阻塞的 Process.wait 以确保统计每一个结束的子进程。
begin
  while pid = Process.wait(-1, Process::WNOHANG)
    puts pid
    dead_processes += 1
    # 一旦所有的子进程统计完毕就退出。
    exit if dead_processes == child_processes
  end
rescue Errno::ECHILD
end
end

# 父进程需要执行的密集数学运算。
loop do
  (Math.sqrt(rand(44)) ** 8).floor
  sleep 1
end

```

还有一点需要注意：如果没有子进程存在，Process.wait 乃至其变量，将会抛出 Errno::ECHILD 异常。因为信号可以在任何时间到达，很可能最后一个 CHLD 信号到达的时候，之前的 CHLD 处理程序已

经先后两次调用 `Process.wait`, 并得到了最后的可用状态信息。这种异步的情形会让人困惑不已。任何一行代码都能够被信号中断。这一点我早就说过了!

你必须在自己的 CHLD 信号处理程序中处理 `Errno::ECHILD` 异常。同样地, 如果你不知道需要等待多少个子进程, 那么你就应该捕获这个异常并正确处理它。

16.3 信号入门

这是我们第一次和 Unix 信号碰面。信号是一种异步通信。当进程从内核那里接收到一个信号时, 它可以执行下列某一操作:

- (1) 忽略该信号
- (2) 执行特定的操作
- (3) 执行默认的操作

16.4 信号来自何方?

从技术上来说, 信号由内核发送, 如同短信由手机用户发出一样。但是短信有原始发送端, 信号也是如此。信号是由一个进程发送到另一个进程, 只不过是借用内核作为中介。

信号最初的目的是用来指定终结进程的不同方式。就让我们从这里说起吧。

启动两个 ruby 程序, 然后用其中一个干掉另外一个。

60 | 第 16 章 进程皆可获得信号

对于这些例子，我们不使用 `irb`，因为它定义了自己的信号处理程序，这有碍于我们的演示。因此，我们将使用 `ruby` 程序本身。

试试这个：不加任何参数执行 `ruby` 程序。输入一些代码，然后点击 `Ctrl-D`。

这样会执行你刚才输入的代码并退出。

使用上面介绍的方法启动两个 `ruby` 进程，然后使用信号来结束其中一个进程。

(1) 在第一个 `ruby` 会话中执行下列代码：

```
puts Process.pid  
sleep # 以便于有时间发送信号
```

(2) 在第二个 `ruby` 会话中发出下列命令来使用信号终结第一个会话：

```
Process.kill(:INT, <pid of first session>)
```

因此第二个进程会向第一个进程发送一个 INT 信号，使其退出。INT 是 INTERRUPT（中断）的缩写。

当一个进程接收到这个信号时，系统默认该进程应当中断当前操作并立即退出。

16.5 信号一览

下面的表格列出了 Unix 系统通常支持的信号。每一个 Unix 进程都能够响应这些信号，这些信号也都可以被发送到任意的进程。

命名信号的时候，名字中的 SIG 部分是可选的。表中的“动作”一列描述了每个信号的默认操作。

Term

表示进程会立即结束

Core

表示进程会立即结束并进行核心转储（栈跟踪）

Ign

表示进程会忽略该信号

Stop

表示进程会停止运行（暂停）

Cont

表示进程会恢复运行（继续）

信号	值	动作	注释
<hr/>			
SIGHUP	1	Term	由控制终端或控制进程终止时发出
SIGINT	2	Term	来自键盘的中断信号（通常是 Ctrl-C）
SIGQUIT	3	Core	来自键盘的退出信号（通常是 Ctrl-/）

62 | 第 16 章 进程皆可获得信号

SIGILL	4	Core	非法指令
SIGABRT	6	Core	来自 <code>abort(3)</code> 的终止信号
SIGFPE	8	Core	浮点数异常
SIGKILL	9	Term	Kill 信号
SIGSEGV	11	Core	非法内存地址引用
SIGPIPE	13	Term	管道损坏 (Broken pipe)：向没有读取进程的管道写入信息
SIGALRM	14	Term	来自 <code>alarm(2)</code> 的计时器到时信号
SIGTERM	15	Term	终止信号
SIGUSR1	30,10,16	Term	用户自定义信号 1
SIGUSR2	31,12,17	Term	用户自定义信号 2
SIGCHLD	20,17,18	Ign	子进程停止或终止
SIGCONT	19,18,25	Cont	如果停止，则继续执行
SIGSTOP	17,19,23	Stop	停止进程执行 (来自非终端)
SIGTSTP	18,20,24	Stop	来自终端的停止信号
SIGTTIN	21,21,26	Stop	后台进程的终端输入
SIGTTOU	22,22,27	Stop	后台进程的终端输出

`SIGKILL` 和 `SIGSTOP` 信号不能被捕获、阻塞或忽略。

这张表看起来有点奇怪，但能让你大致了解给进程发送某个信号时会发生什么。可以看到大部分信号的默认操作都是终止进程。

注意两个有意思的信号 `SIGUSR1` 和 `SIGUSR2`。这两个信号对应的操作是由你的进程来定义的。很快我们就可以随意地重新定义需要的信号操作，不过这两个信号是专门供你使用的。

16.6 重定义信号

让我们返回到那两个 ruby 会话来找点乐子。

(1) 在第一个 ruby 会话中使用下面的代码重新定义 INT 信号的行为：

```
puts Process.pid  
trap(:INT) { print "Na na na, you can't get me" }  
sleep # 以便于有时间发送信号
```

现在就算接收到 INT 信号，我们的进程也不会退出了。

- (2) 在第二个 ruby 会话中使用下面的命令，你会发现第一个进程正在笑话我们呢！

```
Process.kill(:INT, <pid of first session>)
```

- (3) 试着用 Ctrl-C 来终结第一个会话，但结果还是一样！

- (4) 表格中显示有一些信号是不能被重定义的。SIGKILL 会告诉那些小子们谁才是老大。

```
Process.kill(:KILL, <pid of first session>)
```

16.7 忽略信号

- (1) 在第一个 ruby 会话中使用下面的代码：

```
puts Process.pid  
trap(:INT, "IGNORE")  
sleep # 以便于有时间发送信号
```

- (2) 在第二个 ruby 会话中使用下面的命令，注意第一个进程并没有受到影响。

```
Process.kill(:INT, <pid of first session>)
```

64 | 第 16 章 进程皆可获得信号

第一个 ruby 会话安然无恙。

16.8 信号处理程序是全局性的

信号是一个了不起的工具，非常适用于某些情况。不过最好还是记住：捕获一个信号有点像使用一个全局变量，你有可能把其他代码所依赖的东西给修改了。和全局变量不同的是，信号处理程序并没有命名空间。

因此，在把信号处理程序添加到你的开源代码库之前，务必要了解下面的内容。

16.9 恰当地重定义信号处理程序

有一个方法可以保留其他 Ruby 代码定义的处理程序，这样你的信号处理程序就不会破坏其他已经定义好的处理程序了。这个方法如下：

```
trap(:INT) { puts 'This is the first signal handler' }

old_handler = trap(:INT) {
  old_handler.call
  puts 'This is the second handler'
  exit
}
sleep # 以便于有时间发送信号
```

给它发送一个 Ctrl-C 看看效果吧。两个信号处理程序都被调用了。

现在来看看我们是否可以保留系统的默认行为。运行下面的代码并使用 Ctrl-C。

```
system_handler = trap(:INT) {  
    puts 'about to exit!'  
    system_handler.call  
}  
sleep # 以便于有时间发送信号
```

这次搞砸了！所以我们没法使用这种方法保留系统的默认行为，不过可以保留其他 Ruby 代码所定义的处理器。

从最佳实践的角度来说，你的代码不应该定义任何信号处理器，除非它是服务器。正如一个从命令行启动的长期运行的进程，库代码极少会捕获信号。

```
# 一种“友好的”捕获信号的方法  
  
old_handler = trap(:QUIT) {  
    # 进行清理  
    puts 'All done!'  
  
    old_handler.call if old_handler.respond_to?(:call)  
}
```

这个 QUIT 信号的处理器会保留所有之前已定义好的 QUIT 处理程序。尽管这看起来很友好，但通常不是个好主意。考虑这样一个场景：某个 Ruby 服务器告知用户可以发送 QUIT 信号，然后服务器便可平稳关闭。你告诉你的代码库用户可以发送 QUIT 信号，然后代码库会绘制出一幅 ASCII 彩虹（ASCII rainbow）。现在如果一个用户发送了 QUIT 信号，那么两个信号处理器都会被调用，这可是两个库都不想遇到的问题。

是否保留之前定义的信号处理器取决于你自己，只要你知道自己为

66 | 第 16 章 进程皆可获得信号

为什么要这么做。如果你只是想加入一些操作，以在退出之前能够清理资源，那就可以使用一个 `at_exit` 钩子，在退出码那章我们已经提到了这一点。

16.10 何时接收不到信号？

进程可以在任何时候接收到信号。这就是信号的美之所在！它们是异步的。

你的进程可以从繁忙的 `for` 循环中解脱出来，转而使用信号处理程序，甚至连长时间的 `sleep` 都可以不需要。如果进程在接收到一个信号的同时还在处理其他的信号，那么它可以从一个信号处理程序转到另一个信号处理程序中。不过，不出所料的话，它总会执行完所有被调用的信号处理程序中的代码。

16.11 实践领域

有了信号，一旦知道了对方的 `pid`，系统中的进程便可以彼此通信。这使得信号成为了一种极其强大的通信工具。常见的用法是在 `shell` 中使用 `kill(1)` 发送信号。

在实践中，信号多是由长期运行的进程使用，例如服务器和守护进程。多数情况下，发送信号的都是人类用户而非自动化程序。

例如，Web 服务器 Unicorn^①通过终止其所有进程并立即关闭来响应

① <http://unicorn.bogomips.org>

INT 信号。通过重新执行来响应 USR2 信号，从而实现零关闭时间重启(zero-downtime restart)。通过增加运行的工作进程数量来响应 TTIN 信号。

请查看 Unicorn 自带的 SIGNALS 文件^①，以了解它所支持的全部信号以及响应方式。

memprof 项目是友好处理信号的好例子^②。

16.12 系统调用

Ruby 的 Process.kill 对应于 kill(2)，Kernel#trap 基本对应于 sigaction(2)。signal(7)同样有所帮助。

① <http://unicorn.bogomips.org/SIGNALS.html>

② <https://github.com/ice799/memprof/blob/d4bc228aca323b58fea92dbde20c1f8ec36e5386/lib/memprof/signal.rb#L8-16>

第 17 章

进程皆可互通

到目前为止，我们已经见过相关的进程之间共享内存以及打开的资源。但是多个进程之间的通信信息又是怎样的呢？

这属于进程间通信（简称 IPC）研究领域的一部分。有很多种方法可以实现 IPC，不过我打算讲解两个常见的实用方法：管道和套接字对（socket pairs）。

17.1 我们的第一个管道

管道是一个单向数据流。换句话说，你可以打开一个管道，一个进程拥有管道的一端，另一个进程拥有另一端。然后数据就沿着管道单向传递。因此如果某个进程将自己作为 reader（读者），而非 writer（写者），那么它就无法向管道中写入数据。反之亦然。

在涉及多进程之前，让我们先来看看如何创建一个通道以及可以从中获得什么：

```
reader, writer = IO.pipe #=> [#<IO:fd 5>, #<IO:fd 6>]
```

`IO.pipe` 返回一个包含两个元素的数组，这两个元素皆为 `IO` 对象。Ruby 神奇的 `IO` 类^①是 `File`、`TCPSocket`、`UDPSocket` 等的超类。所有这些资源都有一个通用的接口。

从 `IO.pipe` 返回的 `IO` 对象可以看作是类似于匿名文件的东西。基本上你可以将其视为 `File` 来对待。你可以调用 `#read`、`#write` 和 `#close`，等等。不过 `#path` 对 `IO` 对象无效，`IO` 对象在文件系统中并没有对应的位置。

先暂不考虑多进程，我们来演示一下如何使用管道进行通信：

```
reader, writer = IO.pipe
writer.write("Into the pipe I go...")
writer.close
puts reader.read
```

输出：

```
Into the pipe I go..
```

相当简单吧？有没有注意到我向管道写完信息后就关闭了 `writer`？这是因为当 `reader` 调用 `IO#read` 时，它会不停地试图从管道中读取数据，直到读到一个 EOF（文件结束标志^②）。这个标志告诉 `reader` 已经没有数据可读了。

只要 `writer` 仍旧保持打开，那么 `reader` 就可能读取到更多的数据，因此它就会一直等待。在读取之前关闭 `writer`，将一个 EOF 放入管道中，

① <http://librelist.com/browser//usp.ruby/2011/9/17/the-ruby-io-class/>

② [Uhttp://en.wikipedia.org/wiki/End-of-file](http://en.wikipedia.org/wiki/End-of-file)

这样一来，`reader` 获得原始数据之后就会停止读取。要是你跳过关闭 `writer` 这一步，那么 `reader` 就会被阻塞并不停地试图继续读取数据。

17.2 管道是单向的

```
reader, writer = IO.pipe  
reader.write("Trying to get the reader to write something")
```

输出：

```
>> reader.write("Trying to get the reader to write something")  
IOError: not opened for writing  
    from (irb):2:in 'write'  
    from (irb):2
```

由 `IO.pipe` 返回的 `IO` 对象只能用于单向通信。因此 `reader` 只能读取，`writer` 只能写入。

现在让我们来引入多个进程。

17.3 共享管道

在讲解进程衍生那一章的时候，我讲过当某个进程衍生出一个子进程时，如何共享或复制打开的资源。管道也被认为是一种资源，它有自己的文件描述符以及其他的一切，因此也可以与子进程共享。

下面是一个使用管道在父进程与子进程之间进行通信的简单例子。子进程通过向管道写入信息来告诉父进程它已经完成了自己的一轮工作：

```
reader, writer = IO.pipe

fork do
  reader.close

  10.times do
    # 写入数据
    writer.puts "Another one bites the dust"
  end
end

writer.close
while message = reader.gets
  $stdout.puts message
end
```

输出十次 `Another one bites the dust`。

请注意，和上面一样，我们关闭了管道未使用的一端，以免干扰正在发送的 EOF。如今涉及到两个进程，在考虑 EOF 时就需要再多考虑一层。因为文件描述符会被复制^①，所以现在就出现了 4 个文件描述符。其中只有两个会被用于通信，其他两个必须关闭。因此多余的文件描述符就被关闭了。

因为管道的两端都是 `IO` 对象，我们可以在它们之上调用任意的 `IO` 方法，不仅仅局限于`#read` 和`#write`。在这个例子中，我使用`#puts` 和`#gets` 来对一个由行终止符（`newline`）分隔的 `String` 进行读取。实际上，我在这里使用它们简化了管道的一个方面：管道中流淌的是数据流。

① 在衍生出子进程时。——译者注

17.4 流与消息

在我说“流”的时候，我的意思是在管道中读写数据时，并没有开始和结束的概念。当使用诸如管道或 TCP 套接字这样的 IO 流时，你将数据写入流中，之后跟着一些特定协议的分隔符（delimiter）。比如 HTTP 使用一连串行终止符来分隔头部和主体。

随后从 IO 流中读取数据时，一次读取一块（chunk），遇到分隔符就停止读取。这就是为什么在上一个例子中我要使用#puts 和#gets：它们使用行终止符作为分隔符。

你大概已经猜到，也可以使用消息来代替流进行通信。我们没法在管道中使用消息，不过在 Unix 套接字中就可以。简单地说，Unix 套接字是一种只能用于在同一台物理主机中进行通信的套接字。它比 TCP 套接字快得多，非常适合用于 IPC。

在下面的例子中，我们创建了一对可以通过消息来通信的 Unix 套接字：

```
require 'socket'
Socket.pair(:UNIX, :DGRAM, 0) #=> [<#<Socket:fd 15>, #<Socket:
fd 16>]
```

这段代码创建了一对已经相互连接好的 UNIX 套接字。这些套接字并不使用流，而是使用数据报（datagram）通信。在这种方式中，你向其中一个套接字写入整个消息，然后从另一个套接字中读取整个消息，不需要分隔符。

下面是个有点复杂的管道例子，其中子进程等待父进程告诉它要做什么，并在工作完成后向父进程报告：

```
require 'socket'

child_socket, parent_socket = Socket.pair(:UNIX, :DGRAM, 0)
maxlen = 1000

fork do
  parent_socket.close

  4.times do
    instruction = child_socket.recv(maxlen)
    child_socket.send("#{instruction} accomplished!", 0)
  end
end
child_socket.close

2.times do
  parent_socket.send("Heavy lifting", 0)
end
2.times do
  parent_socket.send("Feather lifting", 0)
end

4.times do
  $stdout.puts parent_socket.recv(maxlen)
end
```

输出：

```
Heavy lifting accomplished!
Heavy lifting accomplished!
Feather lifting accomplished!
Feather lifting accomplished!
```

所以说管道提供的是单向通信，套接字对提供的是双向通信。父套接字可以读写子套接字，反之亦然。

17.5 远程 IPC?

IPC 意味着运行在同一台机器上的进程间的通信。如果你想从单机扩展到多台机器,且同时实现类似于 IPC 的功能,那么有几个方案可供考虑。第一个是使用 TCP 套接字来实现。对于重大系统而言,这会比其他方案需要更多的样板代码 (boilerplate code)^①。其他可能的方案包括 RPC^② (远程过程调用),类似于 ZeroMQ^③的消息系统,或分布式系统^④。

17.6 实践领域

管道和套接字都是对进程间通信的有益抽象。它们既快速又简单,多被用作通信通道,来代替更为原始的方法,如共享数据库或日志文件。

使用哪种方法取决于你的需要。当衡量抉择的时候,记得管道是单向的,套接字对是双向的。

更深入的例子请查看 Spyglass 项目中的 Spyglass Master 类。它比上面的例子更复杂,其中多个子进程要通过单个管道同它们的父进程通信。

17.7 系统调用

Ruby 的 `I0.pipe` 对应于 `pipe(2)`, `Socket.pair` 对应于 `socketpair(2)`。
`Socket.recv` 对应于 `recv(2)`, `Socket.send` 对应于 `send(2)`。

① <http://en.wikipedia.org/wiki/Boilerplatecode>。——译者注

② http://en.wikipedia.org/wiki/Remote_procedure_call

③ <http://www.zeromq.org/>

④ http://en.wikipedia.org/wiki/Distributed_computing

第 18 章

守护进程

守护进程是在后台运行的进程，不受终端用户控制。Web 服务器或数据库服务器都属于常见的守护进程，它们一直在后台运行响应请求。

守护进程也是操作系统的核心功能。有很多进程一直在后台运行以保证系统正常运作。比如 GUI 系统中的窗口服务器、打印服务或音频服务，这样你的音箱才能播放出那讨厌的“叮”提示音。

18.1 首个进程

有一个特殊的守护进程对于操作系统的意义重大。我们在前面说过每个进程都有一个父进程。这个结论对所有的进程都成立吗？系统最开始的那个进程又是怎样的呢？

这是一个典型的“鸡生蛋，蛋生鸡”的问题，答案很简单。当内核被引导时会产生一个叫做 `init` 的进程。这个进程的 `ppid` 是 0，作为所有进程的祖父。它是首个进程，没有祖先。它的 `pid` 是 1。

18.2 创建第一个守护进程

开始之前我们需要做点什么？无需太多。任何进程都可以变成守护进程。

在这里我们以 rack 项目^①为例进行讲解。rack 带有一个 `rackup` 命令，该命令使用不同的 rack 支持的 Web 服务器为应用程序提供服务。Web 服务器就是永不结束的进程中一个极好的例子。只要应用程序还在运行，你就需要有一个服务器来侦听连接。

`rackup` 命令包括一个选项，可以将服务器变成守护进程并置于后台运行。我们来看看这将如何实现。

18.3 深入 Rack

```
def daemonize_app
  if RUBY_VERSION < "1.9"
    exit if fork
    Process.setsid
    exit if fork
    Dir.chdir "/"
    STDIN.reopen "/dev/null"
    STDOUT.reopen "/dev/null", "a"
    STDERR.reopen "/dev/null", "a"
  else
    Process.daemon
  end
end
```

① <http://github.com/rack/rack>

这里涉及很多方面。先看 `else` 代码块。Ruby 1.9.x 中带有一个叫做 `Process.daemon` 的方法，可以将当前的进程变成守护进程！这太方便了！

难道你不想知道它的工作原理吗？肯定想吧！真相就是，如果你查看 `Process.daemon`^① 的 MRI 源代码，并审视 C 语言代码，会发现它最后所做的事与 Rack 在上面的 `if` 语句块中所做的是一样的。

我们继续以上面的代码为例并逐行分析。

18.4 逐步将进程变成守护进程

```
exit if fork
```

这一行代码灵活运用了 `fork` 方法的返回值。回忆一下在讲解进程衍生那章中曾经说过 `fork` 会返回两次，一次在父进程中，另一次在子进程中。在父进程中返回子进程的 `pid`，在子进程中返回 `nil`。

和往常一样，这个返回值在父进程中为真，在子进程中为假。这就意味着父进程将会退出，已成为孤儿的子进程继续照常运行。

如果一个进程是孤儿，那么执行 `Process.ppid` 时会怎样呢？

有关 `init` 进程的知识在这里就派上用场了。孤儿进程的 `ppid` 始终是 1。这是内核能够确保一直运行的唯一进程。

^① <https://github.com/ruby/ruby/blob/c852d76f46a68e28200f0c3f68c8c67879e79c86/process.c#L4817-4860>

创建守护进程时，这第一步是必须的。因为这使得调用此脚本的终端认为该命令已经执行完毕，于是将控制返回到终端，不再考虑那条命令了。

Process.setsid

调用 `Process.setsid` 完成了以下三件事：

- (1) 该进程变成一个新会话的会话领导
- (2) 该进程变成一个新进程组的组长
- (3) 该进程没有控制终端

要想正确理解这三件事有什么影响，我们得先跳出 Rack 的例子，学习一些更深入的知识。

18.5 进程组和会话组

进程组和会话组都和作业控制有关。我用“作业控制”来指引终端处理进程的方法。

我们先来看进程组。

每一个进程都属于某个组，每一个组都有唯一的整数 `id`。进程组是一个相关进程的集合，通常是父进程与其子进程。但是你也可以按照需要将进程分组，只要使用 `Process.setpgroup(new_group_id)` 来设置进程的组 `id` 即可。

看看下列代码的输出：

```
puts Process.getpgrp  
puts Process.pid
```

如果你在一个 `irb` 会话中运行这些代码，那么这两个值就是相等的。通常情况下，进程组 `id` 和进程组组长的 `pid` 相同。进程组组长是终端命令的“发起”进程。也就是说，如果你在终端启动一个 `irb` 进程，那么它就会成为一个新进程组的组长。它所创建的子进程就成为同一个进程组的组员。

试运行下面的例子来查看所继承的进程组。

```
puts Process.pid  
puts Process.getpgrp  
  
fork {  
  puts Process.pid  
  puts Process.getpgrp  
}  
...
```

你可以看到尽管子进程获得了唯一的 `pid`，但它的组 `id` 却是从父进程中继承而来。因此这两个进程都是同一个进程组的成员。

回忆一下我们之前讲过的孤儿进程。那时我说过子进程并不会得到内核的特殊对待，子进程在父进程退出后照常运行。这是父进程退出后的行为，但如果父进程由终端控制并被信号终止的话，孤儿进程的行为就不大一样了。

考虑一下：一个支持某个长期运行的 shell 命令的 Ruby 脚本，比方说长期备份脚本，如果使用 `Ctrl-C` 终止这个 Ruby 脚本会怎样？

如果你动手试一下，就会发现那个长期备份脚本并没有变成孤儿，当

80 | 第 18 章 守护进程

父进程被终止后，它不再继续运行。我们并没有安排任何代码将来自父进程的信号转发给子进程，这一切是怎么回事？

终端接收信号，并将其转发给前台进程组中的所有进程。在这种情况下，因为 Ruby 脚本和那个长期运行的 shell 命令均是同一个进程组的组员，因此会被同一个信号终止。

接下来是会话组。

会话组是更高一级的抽象，它是进程组的集合。看看下面的 shell 命令：

```
git log | grep shipped | less
```

在这个例子中，每个命令都有自己的进程组，这是因为每个命令都可能创建子进程，但这些子进程并不属于其他命令。尽管这些命令不属于同一个进程组，Ctrl-C 仍可以将其全部终止。

这些命令都是同一个会话组的成员，在 shell 中的每一次调用都会获得自己的会话组。一次调用可以是单个命令，也可以是由管道连接的一串命令。

就像上面例子中那样，一个会话组可以依附于一个终端，也可以不依附于任何终端，比如守护进程。

终端又用一种特殊的方法来处理会话组：发送给会话领导的信号被转发到该会话中的所有进程组内，然后再被转发到这些进程组中的所有进程。如同海龟背地球一般（Turtles all the way down）^①；

^① 参看 http://en.wikipedia.org/wiki/Turtles_all_the_way_down。——译者注

有一个系统调用 `getsid(2)` 可以用来检索当前的会话组 id，但是 Ruby 核心库并没有对应的接口。`Process.sesid` 会返回其新创建的会话组的 id，你可以保存起来以备不时之需。

现在返回到 Rack 的例子中，第一行代码衍生出一个子进程，然后父进程退出。启动该进程的终端觉察到进程退出后，将控制返回给用户，但是之前衍生出的子进程仍然拥有从父进程中继承而来的组 id 和会话 id。此时这个衍生进程既非会话领导，也非组长。

因此终端与衍生进程之间仍有牵连，如果它发出信号到衍生进程的会话组，这个信号仍会被接收到，但是我们想要的是完全脱离终端。

`Process.setsid` 会使衍生进程成为一个新进程组和新会话组的组长兼领导。注意，如果在某个已经是进程组组长的进程中调用 `Process.setsid`，则会失败，它只能从子进程中调用。

新的会话组并没有控制终端，不过从技术上来说，可以给它分配一个。

```
exit if fork
```

已成为进程组和会话组组长的衍生进程再次进行衍生，然后退出。

这个新衍生出的进程不再是进程组的组长，也不是会话领导。由于之前的会话领导没有控制终端，并且此进程也不是会话领导，因此这个进程绝不会有控制终端。终端只能够分配给会话领导。

如此以来就确保了我们的进程现在完全脱离了控制终端并且可以独自运行。

```
Dir.chdir "/"
```

82 | 第 18 章 守护进程

此行代码将当前工作目录更改为系统的根目录。并非一定要这么做，但是这额外的一步确保了守护进程的当前工作目录在执行过程中不会消失。

这就避免了守护进程的启动目录出于这样或那样的问题被删除或卸载。

```
STDIN.reopen "/dev/null"
STDOUT.reopen "/dev/null", "a"
STDERR.reopen "/dev/null", "a"
```

这将所有的标准流设置到`/dev/null`，也就是将其忽略。因为守护进程不再依附于某个终端会话，那么这些标准流也就没什么用了。不能简单地将其关闭，因为一些程序还指望着它们随时可用。重定向到`/dev/null`确保了它们对于一些程序依然能用，但实际上毫无效果。

18.6 实践领域

我们提到过，`rackup` 命令有一个命令行选项可以将进程变为守护进程。对于任何流行的 Ruby Web 服务器来说都是这样。

如果你想深入研究守护进程，那就应该看看 rubygem 中的 `daemons`^①。

如果你想创建一个守护进程，那就应该问自己一个基本的问题：这个进程需要一直保持响应吗？

^① <http://rubygems.org/gems/daemons>

如果答案为否，那么你也许可以考虑定时任务或后台作业系统。如果答案是肯定的，那么你可能已经有了好的候选方案。

18.7 系统调用

Ruby 的 `Process.setsid` 对应于 `setsid(2)`，`Process.getpgrp` 对应于 `getpgrp(2)`。本章涉及的其他系统调用在之前的章节中已经详细讲解过了。

第 19 章

生成终端进程

Ruby 程序中一个常见的交互是在程序中通过 shelling out 的方式在终端执行某个命令，这在编写 Ruby 脚本来将若干常用命令粘合在一起时尤为常见。在 Ruby 中有很多方法可以生成进程来执行终端命令。

在讨论各种 shelling out 方法之前，让我们先看看它们背后的工作原理。

19.1 fork + exec

下面描述的所有方法都是 fork(2) + exec(2)的变体。

之前我们已经学习过 fork(2)，对于 exec(2)却是首次提及。exec(2)非常简单，它允许你使用另一个进程来替换当前进程。

换句话说，exec(2)可以让你将当前进程转变成另外一个进程。你可以先使用一个 Ruby 进程，然后把它变成 Python 进程、ls(1)进程，或是另一个 Ruby 进程。

exec(2)的这种转变是有去无回的。一旦你将 Ruby 进程转变成别的什

么，那就再也变不回来了。

```
exec 'ls', '--help'
```

要生成新进程的时候，`fork + exec` 的组合是常见的一种用法。`exec(2)` 是将当前进程转变为另一个进程的一种强大而有效的方法，唯一的缺点就是当前进程再也无法恢复了。而 `fork(2)` 就没有这个问题。

你可以使用 `fork(2)` 创建一个新进程，然后用 `exec(2)` 把这个进程变成其他你想要的进程。你的当前进程仍像从前一样运行，也仍可以根据需要生成其他进程。

如果程序依赖于 `exec(2)` 调用的输出，你可以使用上一章学到的工具来处理。`Process.wait` 可以确保你的程序一直等到子进程完成它的工作，这样就能够取回结果。

记住，`exec(2)` 不会关闭任何打开的文件描述符（默认情况下）或是进行内存清理。有时候这是优点，有时候这又可能会造成资源使用方面的问题。

```
hosts = File.open('/etc/hosts')

exec 'python', '-c', "import os; print os.fdopen_#{hosts.
fileno}.read()"
```

在这个例子中，我们启动了一个 Ruby 程序并打开文件 `/etc/hosts`。然后 `exec(2)` 产生一个 `python` 进程，告诉它文件 `/etc/hosts` 打开时 Ruby 所获得的文件描述符编号。可以看到，`python` 识别出了此文件描述符（因为它通过 `exec(2)` 共享），并能从中进行读取，而无需再次打开文件。

19.2 exec 的参数

有没有注意到在上面所有的例子中，我传递了一个参数数组给 `exec`，而非一个参数字符串？这两种参数形式之间存在微妙的差异。

把字符串传递给 `exec`，它实际上会启动一个 shell 进程，然后再将这个字符串交由 shell 解释。传递一个数组的话，它会跳过 shell，直接将此数组作为新进程的 ARGV。

除非你真的需要，通常我们都会避免传递字符串，而是尽可能地传递数组。传递一个字符串，然后通过 shell 来执行代码，这会产生安全隐患。如果涉及用户输入，那么用户有可能直接将恶意命令插入到 shell 中，来获得当前进程所拥有的任何权限。如果你希望实现类似于 `exec('ls * | awk '{print($1)}')` 的效果，那就必须通过字符串来传递了。

Kernel#system

```
system('ls')
system('ls', '--help')
system('git log | tail -10')
```

Kernel#system 的返回值用最基本的方式反映了终端命令的退出码。如果终端命令的退出码是 0，它就返回 `true`，否则返回 `false`。

借助 `fork(2)` 的魔力，终端命令与当前进程共享标准流，因此来自终端命令的任何输出同样也会出现在当前进程中。

Kernel#`

```
`ls`  
`ls --help`  
%x[git log | tail -10]
```

Kernel#`略有不同，它的返回值是由终端程序的 STDOUT 汇集而成的一个字符串。

正如前面提到的那样，它使用 fork(2)来实现，对于 STDERR 并不做任何特殊处理，因此在第二个例子中你可以看到 STDERR 像在 Kernel#system 中那样被打印到屏幕上。

Kernel#`和%x□做的事完全一样。

Process.spawn

```
# 仅适用于 Ruby 1.9!  
  
# 此调用会启动 rails server 进程并将环境变量 RAILS_ENV 设置为 test  
Process.spawn({'RAILS_ENV' => 'test'}, 'rails server')  
  
# 该调用在执行 ls --help 阶段将 STDERR 与 STDOUT 进行合并  
Process.spawn('ls', '--help', STDERR => STDOUT)
```

Process.spawn 是非阻塞的，这是它与众不同的地方。

比较下面两个例子，你就可以看出 Kernel#system 会阻塞到命令执行完毕，而 Process.spawn 则会直接返回。

```
# 以阻塞方式执行  
system 'sleep 5'  
  
# 以非阻塞方式执行
```

```
Process.spawn 'sleep 5'

# 使用 Process.spawn 以阻塞方式执行
# 返回子进程的 pid
pid = Process.spawn 'sleep 5'
Process.waitpid(pid)
```

上一个例子是 Unix 编程灵活性的极好展示。在之前的章节中我们讨论过很多关于 `Process.wait` 的内容，但总是出现在“衍生出子进程，然后运行若干 Ruby 代码”的情况下。从这个例子中可以看到：内核不关心你在进程中做了什么，它始终按照同样的方式运行。

所以即便我们执行 `fork(2)`，然后运行 `sleep(1)` 程序（一个 C 程序），内核仍旧知道如何等待那个进程结束。不仅如此，它还能够正确地返回退出码，一如在 Ruby 程序中那样。

一切代码对于内核来说都是一个样。正因此才形成了一个如此灵活的系统。你可以在不同的编程语言之间进行交互，所有的编程语言均被一视同仁。

`Process.spawn` 有很多选项可以用来控制子进程的行为。在上面的例子中我已经展示了其中一些有用的选项。请参看官方的 `rdoc`^① 来获取完整的列表。

IO.popen

```
# 这个例子会返回一个文件描述符 (IO 对象)
# 对其进行读取会返回该 shell 命令打印到 STDOUT 中的内容
IO.popen('ls')
```

① <http://www.ruby-doc.org/core-1.9.3/Process.html#method-c-spawn>

IO.popen 最常见的用法是用纯 Ruby 来实现 Unix 管道 (pipes)，这即是 popen 中 “p”的由来。在底层 IO.popen 仍旧做的是 fork+exec 的事，但它还设置了一个通道用于同生成进程进行通信。这个管道作为块参数 (block argument) 被传递到 IO.popen 的代码块中。

```
# 一个 IO 对象被传递到代码块中。在本例中我们打开 stream 进行写入,  
# 因此将 stream 设置为生成进程的 STDIN。  
#  
# 如果打开 stream 进行读取（默认操作），那么将 stream 设置为生成进程的 STDOUT。  
IO.popen('less', 'w') { |stream|  
  stream.puts "some\ndata"  
}
```

使用 IO.popen 时必须选择访问哪个流。你无法一次访问所有的流。

Open3

Open3 允许同时访问一个生成进程的 STDIN、STDOUT 和 STDERR。

```
# Open3 是标准库的一员  
require 'open3'  
  
Open3.popen3('grep', 'data') { |stdin, stdout, stderr|  
  stdin.puts "some\ndata"  
  stdin.close  
  puts stdout.read  
}  
  
# 在可行的情况下，Open3 会使用 Process.spawn  
# 可以像这样将选项传递给 Process.spawn:  
Open3.popen3('ls', '-uhh', :err => :out) { |stdin, stdout, stderr|  
  puts stdout.read  
}
```

从用法上来说，`Open3` 像是一个更灵活的 `IO.popen`。

19.3 实践领域

所有这些方法在实践中都很常见。因为它们的行为各异，你必须根据需要选择其中某个方法。

这些方法的一个缺点是都依赖 `fork(2)`。这有什么问题呢？想象这样一个场景：你有一个需要数百兆内存的大型 Ruby 应用，现在需要进行 `shell out`，采用上面任何一种方法都会引发由进程衍生所带来的成本。

即便利用 `shell out` 来执行一个简单的 `ls(1)` 调用，内核仍需保证新的 `ls(1)` 进程可以使用 Ruby 进程的全部内存。为什么？因为 `fork(2)` 的缘故。当你使用 `fork(2)` 衍生出进程时，内核并不知道你打算通过 `exec(2)` 将其替换掉。你使用进程衍生也许是为了运行 Ruby 代码，但这种情况下需要所有的内存都可用。

`fork(2)` 是有成本的，记住这一点有益无害，有时候它会成为性能瓶颈。如果你需要频繁进行 `shell out`，又不想引发 `fork(2)` 所带来的成本，那该如何实现？

有一些用于生成进程的原生 Unix 系统调用并不会带来 `fork(2)` 所引发的那些开销。然而，Ruby 语言核心库并不支持它们。不过有一个 Rubygem 提供了这些系统调用的 Ruby 接口。`posix-spawn` 项目^① 提供了对 `posix_spawn(2)` 的访问，该调用在大多数 Unix 系统中都可用。

^① <http://github.com/rtomayko posix-spawn>

`posix-spawn` 模拟了 `Process.spawn` API。实际上你传递给 `Process.spawn` 的大多数选项同样可以传递给 `POSIX::Spawn.spawn`。因此你可以使用同样的 API，并享受到速度更快、资源利用效率更高的进程生成所带来的好处。

在基本层面上，`posix_spawn(2)` 是 `fork(2)` 的一个子集。回想一下由 `fork(2)` 所获得的新子进程的两个独特属性：1) 获得了一份父进程在内存中所有内容的副本；2) 获得了父进程已打开的所有文件描述符的副本。

`posix_spawn(2)` 只保留了第 2 条，没有保留第 1 条，这是两者最大的不同。因此可以想见，新生成的进程可以访问由父进程打开的所有文件描述符，却无法与父进程共享内存。这就是为什么 `posix_spawn(2)` 比 `fork(2)` 更快、更有效率的原因。但是要记住，这也会使得它缺乏灵活性。

19.4 系统调用

Ruby 的 `Kernel#system` 对应于 `system(3)`，`Kernel#exec` 对应于 `execve(2)`，`I0.popen` 对应于 `popen(3)`，`posix-spawn` 使用 `posix_spawn(2)`。

第 20 章

尾 声

与 Unix 进程打交道事关两件事：抽象和通信。

20.1 抽象

内核对于进程的观点极其抽象和简单。程序员习惯于通过查看源代码来区分两个程序。

我们擅长多种编程语言，根据需要选择不同的语言。我们不可能使用配备了垃圾收集器的语言编写出高效利用内存的代码，只有用 C 才能做到。如果需要对象，那就用 C++，以此类推。

但所有的一切在内核眼中没什么两样。到头来，所有的代码都会被编译成内核能够理解的简单形式。在那个层面工作的时候，所有的进程都被同等对待，所有的一切都会获得数字标识符，都能够平等地访问内核资源。

说了这么多到底想要阐明什么呢？借助 Unix 编程可以让你深入解相关原理，让你完成那些在编程语言层面上无法完成的事情。

Unix 编程与具体的编程语言无关^①。它也可以让你使用 C 程序作为 Ruby 脚本的接口，反之亦然。它也能让你在各种编程语言中重用其概念。你在 Ruby 中学到的 Unix 编程技巧同样可以用于 Python、node.js 或 C。这些都是关于编程的通用技巧。

20.2 通信

除了创建新进程这种基本操作之外，几乎我们讨论的每一件事都关乎通信。遵循上面讲过的抽象原则，内核为进程间通信提供了非常抽象的方式。

系统中任意的两个进程都可以使用信号来通信。通过为进程命名，你可以同任何在命令行中查看你的程序的用户进行通信。你可以使用退出码给所有期待运行结果的进程发送成功/失败的消息。

20.3 再会，而非永别

结束了！恭喜你终于到达了这里！信不信由你，现在你比大多数程序员都更了解 Unix 进程的内部工作原理。

既然你已经掌握了这些基础原理，便可以将新学到的知识应用于你手头的工作。所有的事情开始变得更有意义。你运用的新知识越多，事情就越发清晰。已经没有什么能够阻挡你了。

① 相关方面的书籍推荐可以参看 <http://www.hanselman.com/blog/SixEssentialLanguageAgnosticProgrammingBooks.aspx>。——译者注

但是还没讨论联网呢，我们将在下一版中谈及这一话题。

阅读书后面的附录，来了解一些流行的 Ruby 项目，以及它们如何巧妙运用 Unix 进程。

关于本书，如果你有任何反馈、勘误，或是运用学到的新知识做出了一些不错的玩意儿，我将洗耳恭听。请发送邮件到 jesse@working-withunixprocesses.com。编码快乐！

附录A

Resque 如何管理进程

这一章，我们来看看流行的 Ruby 作业队列 Resque^①是如何有效地管理进程的。尤其是它利用 fork(2)来管理内存并非出于并发或运行速度上的考量。

A.1 架构

要理解 Resque 的工作方式为何如此，我们需要对系统的工作原理有一个基本的了解。

来自 README：

Resque是一个基于Redis的库，用于创建后台作业，将这些作业放入多个队列并随后处理。

其中我们感兴趣的部分是 Resque worker。Resque worker 负责“随后处理”这一环节。它的工作是启动并载入应用程序环境，然后连接 Redis

① <http://github.com/defunkt/resque#readme>

96 | 附录 A Resque 如何管理进程

并设法保留任何尚未处理的后台作业。如果能够保留此类作业，它就将该作业从队列中清除，然后返回到第一步。够简单吧！

对于大型应用来说，一个 Resque worker 是不够的。因此很常见的做法是利用多个 Resque worker 并行清除作业。

A.2 利用进程衍生进行内存管理

Resque worker 使用 fork(2)进行内存管理。我们先来看看相关代码，然后逐行分析。

```
if @child = fork
  srand # Reseeding
  procline "Forked #{@child} at #{Time.now.to_i}"
  Process.wait(@child)
else
  procline "Processing #{job.queue} since #{Time.now.to_i}"
  perform(job, &block)
  exit! unless @cant_fork
end
```

Resque 每清除一个作业都会执行这些代码。

如果你读过第 11 章，那么可能已经很熟悉这里的 if/else 结构了。不然的话就马上读一下吧！

我们从查看父进程中的代码（也就是 if 语句块内）开始。

```
srand # Reseeding
```

这一行的存在纯粹是因为 MRI Ruby 1.8.7 的某个修正版中的 bug^①。

^① <http://redmine.ruby-lang.org/issues/4338>

```
procline "Forked #{@child} at #{Time.now.to_i}"
```

`procline` 是 Resque 更新当前进程名的内部方式。我们说过，可以使用`$0` 来修改当前进程的名称，但是无法通过可编程的方式来实现。

这就是 Resque 的解决方案，`procline` 可以设置当前进程的名称。

```
Process.wait(@child)
```

如果你读过 `Process.wait` 那章，那你也应该很熟悉这行代码了。

变量`@child` 是指 `fork` 调用的值。因此在父进程中会是子进程的 `pid`。这行代码告诉父进程一直阻塞到子进程结束。

现在我们来看子进程中发生了什么。

```
procline "Processing #{job.queue} since #{Time.now.to_i}"
```

注意 `if` 和 `else` 语句块中都调用了 `procline`。尽管这两行是同一个逻辑结构的不同部分，但它们是在两个不同的进程中执行的。因为进程名针对特定的进程，所以这两个调用会分别为父进程和子进程设置一个名称。

```
perform(job, &block)
```

在这个子进程中，作业由 Resque 来执行。

```
exit! unless @cant_fork
```

然后子进程退出。

A.3 何必自找麻烦?

在本章第一段中我们提到过, Resque 这么做并不是为了实现并发或提速。实际上, 它为每个作业处理多添加了一个步骤, 这使得整个处理过程更慢了。那干嘛要自找麻烦? 为什么不逐一处理作业?

Resque 使用 `fork(2)` 来确保其工作进程使用的内存不会膨胀。让我们来看一看当 Resque worker 衍生出进程后发生了什么, 这对 Ruby 虚拟机又会产生怎样的影响。

回忆一下, `fork(2)` 创建了一个和原始进程一模一样的新进程。在这种情况下, 原始进程只是预先载入了应用程序环境而已。因此我们知道在 `fork` 之后, 会获得一个载入了应用程序环境的新进程。

然后子进程执行清除作业的任务, 这就是内存使用出现问题的地方。后台作业可能需要将图像文件载入主存进行处理, 从数据库中读入大量的 ActiveRecord 对象, 或者执行其他消耗大量内存的操作。

一旦子进程处理完作业并退出, 将会释放其所占用的全部内存并交由操作系统进行清理。然后原始进程就可以恢复运行, 同样只载入应用程序环境。

就内存占用而言, 每当 Resque 处理完一个作业, 都会返回到洁净状态。也就是说, 处理作业时, 内存使用量或会激增, 但终会回落到一个适宜的基线水平。

A.4 难道垃圾收集器就不帮我们清理一下?

它会帮忙, 不过干得并不出色, 只能说还可以。真相是: 在不需要释

放内存的时候，MRI 的垃圾收集器作用不大。

当 Ruby 虚拟机启动时，内核会给它分配一块主存。如果不够用，那么它需要向内核再申请一块。

由于 Ruby 垃圾收集器的诸多问题（如方法不当或磁盘碎片），虚拟机很少能够将内存块释放并归还给内核。因此 Ruby 进程的内存使用率有可能逐渐攀升，而非减少。这时，Resque 的方法便有意义了。

如果 Resque worker 只是在作业可用时将其清除，那它没法将内存占用维持在适宜的基线水平。只要它处理了需要大量主存的作业，这些内存就会一直被工作进程占用直到它退出。

即便随后的作业需要的内存要少得多，Ruby 也很难将内存退还给内核。因此工作进程不可避免地变得越来越大，再也降不下来。

多亏了 fork(2)的强大威力，Resque worker 是靠得住的，在处理一定数量的作业后也无需重启。

附录B

Unicorn 如何收割工作进程

在 Ruby 语言中研究 Unix 编程,如果不详细讨论 Web 服务器 Unicorn^①,无疑是不负责任的。的确,本书也已多次提到过这个项目。

这个项目为什么如此重要?作为 Web 服务器,Unicorn 试图将尽可能多的责任推给内核。它大量地采用了 Unix 编程,代码库中遍布着 Unix 编程技术。

不仅如此,它还拥有出色的性能和可靠性。很多大型的 Ruby 网站都使用了 Unicorn,比如 Github 和 Shopify。

关键在于,如果本书激发了你的兴趣,促使你想要学习更多有关 Ruby 中的 Unix 编程知识,那么你应该深入研究一下 Unicorn。这也许会让你陷入诸多谜团,困惑不已,但最后你会有更深刻的理解和新的想法。

B.1 收割什么?

在深入代码之前,我要提供一点有关 Unicorn 工作原理的背景。从宏

① <http://unicorn.bogomips.org>

观的层面上来看，Unicorn 是一个 pre-forking Web 服务器。

也就是说，你将要启动它并告诉它需要多少个工作进程。Unicorn 通过初始化自己的网络套接字开始运行，然后载入你的应用程序。随后，它采用我们在讲解进程衍生那章所提到的 master-worker 模式，使用 fork(2)创建工作进程。

Unicorn 主进程与其每个工作进程保持联系，确保它们不会花费太多的时间来处理请求。使用下面的代码可以退出 Unicorn 主进程。正如我们在进程衍生那章说过的那样，如果父进程在退出之前没有终结其子进程，那么这些子进程将继续运行下去。

因此 Unicorn 应该在退出之前先对自身进行清理。下面的代码被调用作为 Unicorn 退出过程的一部分。在调用这段代码之前，它会发送 QUIT 信号给所有的工作进程，让它们平稳退出。

Unicorn 使用这些代码来清理工作进程并确保它们全部正常退出。

我们来分析一下这段代码。

```
# 收割所有还未被收割的工作进程
def reap_all_workers
  begin
    wpid, status = Process.waitpid2(-1, Process::WNOHANG)
    wpid or return
    if reexec_pid == wpid
      logger.error "reaped #{status.inspect} exec()-ed"
      self.reexec_pid = 0
      self.pid = pid.chomp('.oldbin') if pid
      proc_name 'master'
    else
      worker = WORKERS.delete(wpid) and worker.close rescue nil
      m = "reaped #{status.inspect} worker=#{worker.nr} rescue"
    end
  end
end
```

102 | 附录 B Unicorn 如何收割工人进程

```
'unknown'}"  
status.success? ? logger.info(m) : logger.error(m)  
end  
rescue Errno::ECHILD  
break  
end while true  
end
```

下面我们逐行进行分析。

```
begin  
...  
end while true
```

首先，你应该注意，该方法第一行的 `begin` 语句块实际上是一个无限循环。在 Ruby 中用其他的方法也可以写出无限循环，但应该牢记的是，我们身处一个无限循环中，需要明确的 `return` 或 `break` 来跳出循环。

```
wpid, status = Process.waitpid2(-1, Process::WNOHANG)
```

这一行各位应该比较熟悉。在 `Process.wait` 那章我们讲过 `Process.waitpid2`，不过我们总是将 `pid` 作为第一个参数来传递。

将一个合法的 `pid` 作为第一个参数传递会使得 `Process.wait` 只等待这个 `pid` 所对应的进程。如果将 `-1` 传给 `Process.waitpid` 又会怎样？我们知道不会有进程的 `pid` 小于 `1`，因此……

使用 `-1` 会使得 `Process.wait` 等待任意的子进程退出。这是该方法的默认选项，如果不指定 `pid`，它就默认使用 `-1`。在这里，因为程序的作者需要为第二个参数赋值，所以第一个参数就不能留空，因此就

将其设置为默认值。

如果你想等待任意子进程，那干嘛不用 `Process.wait2` 呢？我怀疑作者之所以如此，是因为在指定 `pid` 值时，用 `waitpid` 的变体最易读。这点我也赞同。尽管上面的代码只是指定了默认值，但如果要指定 `pid` 值，还是使用 `waitpid` 最一目了然。

第二个参数用于特定的标志。传递 `Process::WNOHANG` 会使得这个通常情况下的阻塞调用变成非阻塞调用。使用该标志的时候，如果没有进程退出，那么它就不会阻塞，而只是返回 `nil`。

wpid or return

这一行看起来有点奇怪，但其实是一个条件 `return` 语句。如果 `wpid` 是 `nil`，那么该行会返回 `nil`。这意味着没有已经退出的子进程为我们返回它们的状态。

如果是这样的话，该方法将会返回，其作业也被处理完毕。

```
if reexec_pid == wpid
  logger.error "reaped #{status.inspect} exec()-ed"
  self.reexec_pid = 0
  self.pid = pid.chomp('.oldbin') if pid
  proc_name 'master'
```

我不想花费太多的时间讨论这段代码。`reexec` 与 `Unicorn` 的内部实现有关，尤其是处理零关闭时间重启。我可能会在以后讨论这个进程。

你需要注意的一点是 `proc_name` 调用。它类似于 `Resque` 那章中提到

104 | 附录 B Unicorn 如何收割工人进程

的 `procline` 方法。Unicorn 也有一个方法可以修改当前进程的名称，这是同你的软件用户进行通信的关键。

```
else
  worker = WORKERS.delete(wpid) and worker.close rescue nil
```

Unicorn 将当前活动的工作进程列表存储在常量 `WORKERS` 中。`WORKERS` 是一个散列，它的键是工作进程的 `pid`，值是 `Unicorn::Worker` 的一个实例。

因此这一行代码将某个工作进程从 Unicorn 内部的跟踪列表 (`WORKERS`) 中移除并对这个工作进程调用`#close`，关闭不再需要的心跳机制 (heartbeat mechanism)。

```
m = "reaped #{status.inspect} worker=#{worker.nr rescue
  'unknown'}"
```

这一行代码根据 `Process.waitpid2` 调用所返回的 `status` 构造一条日志消息。

通过首次对变量 `status` 进行检查来构造出一个字符串。这个字符串看起来是这样的：

```
#<Process::Status: pid=32227, exited(0)>
# 或者
#<Process::Status: pid=32308, signaled(SIGINT=2)>
```

它包括已结束进程的 `pid` 以及结束的方式。在第一行中，退出的进程包含了一个退出码 0。在第二行中，进程由信号 SIGINT 终止。所有类似这样的信息都会被添加到 Unicorn 的日志中。

日志行的第二部分 `worker.nr` 是工作进程号在 Unicorn 的内部描述。

```
status.success? ? logger.info(m) : logger.error(m)
```

这行代码使用已经构造好的日志消息，将其发送到日志记录器。它在 `status` 对象上使用 `success?` 方法将日志消息以 INFO 级别或 ERROR 级别进行登记。

`success?` 方法只有在进程携带为 0 的退出码时才返回 `true`。如果是其他的退出码，则返回 `false`。如果进程被信号终止，则返回 `nil`。

```
rescue Errno::ECHILD  
  break
```

这是该方法中处于顶层位置的 `begin` 语句的一部分。如果产生异常，那么无限循环会执行 `break` 并返回。

如果当前进程没有子进程，那么 `Process.waitpid2` 或是它的任何表亲会产生 `Errno::ECHILD` 异常。如果本例中出现了异常，那就意味着该方法的任务已经完成！所有的子进程都已经收割完毕，方法顺利返回。

B.2 总结

如果这些代码引起了你的兴趣并促使你希望学习更多有关 Ruby 中的 Unix 编程知识，那么 Unicorn 是个不错的资源。登录官方网站 <http://unicorn.bogomips.org>，然后开始学习吧！

附录C

preforking 服务器

很高兴你已经读到了这里，因为这可能是全书中实践性最强的一章。preforking 服务器将书中提及的诸多概念结合起来，形成了解决特定问题的强大而高效的方法。

你很可能已经用过 Phusion Passenger^①或 Unicorn^②。这两个服务器以及本书中所讲的 Web 服务器 Spyglass，均是 preforking 服务器的实例。

这些项目的核心是 preforking 模型。preforking 在很多方面都很特别，以下是其中的三个方面。

- (1) 高效的内存利用。
- (2) 高效的负载均衡。
- (3) 高效的系统管理。

我们来逐条讨论。

① <http://www.modrails.com/>

② <http://unicorn.bogomips.org>

C.1 高效的内存利用

在进程衍生那章我们讨论过 `fork(2)` 如何创建一个和调用（父）进程一模一样的新进程，其中包括父进程当时在内存中所有的一切内容。

载入一个Rails应用

在我的 MacBook Pro 上，只载入 Rails 3.1（不包括库或应用代码）大约花费 3 秒钟。载入完毕后，进程占用了大概 70MB 的内存。

在这里，这个数字和你机器上的数字是否相同并不重要。我只是拿这个数字作为基线来讲解后面的例子。

preforking 要比生成多个无关的进程更能有效地利用内存。我们可以将运行带有 10 个工作进程的 Unicorn 与运行 10 个 Mongrel（非 preforking 服务器）做个比较。

让我们从进程的角度来看看，当启动每个服务器的 10 个实例时都发生了什么，先看 Mongrel，然后是 Unicorn。

C.1.1 多个Mongrel

并行启动 10 个 Mongrel 进程和串行启动 10 个 Mongrel 进程看起来基本无异。

并行启动时，10 个进程会从内核处竞争资源。每一个都会消耗资源来载入 Rails，也通常都会花费 3 秒钟来启动，合计下来就是 30 秒钟。除此之外，一旦 Rails 载入完毕，每个进程都要占用 70MB 的内存，那么 10 个进程一共就要占用 700MB 的内存。

108 | 附录 C preforking 服务器

一个 preforking 服务器可以做得更好。

C.1.2 多个Unicorn

启动 10 个 Unicorn 工作进程将会产生 11 个进程。其中一个是主进程，负责看护其他 10 个工作进程。

当启动 Unicorn 时，只有一个进程，也就是主进程，会载入 Rails。不会出现竞争内核资源的情况。

主进程花费 3 秒钟的时间来载入，然后几乎瞬间就衍生出 10 个进程。主进程消耗 70MB 的内存来载入 Rails，而由于写时复制技术，子进程应该不会消耗任何内存。

这一切的真相就是：Unicorn 得花费一定时间衍生出一个进程，不是瞬间就能完成的，每个子进程还会有一些内存开销。但相比启动多个 Mongrel 所用的开销，这些都可以忽略不计了。preforking 胜出。

记住：如果你使用MRI，就享受不到写时复制的诸多好处了。如果想获得这些好处，你需要使用REE。

C.2 高效的负载均衡

我已经强调过 fork(2)会创建一个和调用进程一模一样的副本，其中包括父进程已经打开的所有文件描述符。

套接字基础

高效的负载均衡与套接字的工作原理有很大关系。因为我们讨论的是Web服务器，所以套接字至关重要，它们是联网的核心所在。正如我之前提到的：套接字和联网都是复杂的课题，涉及内容之广根本无法在本书中讨论。不过为了理解下面内容，你需要理解最基本的工作流程。

使用一个套接字涉及多个步骤：1) 打开一个套接字并绑定到特定的端口；2) 在这个套接字上使用accept(2)接受一个连接；3) 可以在这个连接上读写数据，最后关闭连接。套接字一直处于打开状态，但是连接会被关闭。

通常这都是在同一个进程内发生的。一个套接字被打开，然后该进程在此套接字上等待、处理并关闭连接，继而开始下一个循环。

preforking 服务器使用了一个不同的流程来让内核在套接字上对繁重的负载进行均衡。让我们来看看这是如何实现的。

在像 Unicorn 和 Spyglass 这样的服务器中，主进程做的第一件事就是打开套接字，这一步甚至在载入 Rails 应用之前就进行了。这个套接字用于来自 Web 客户端的外部连接。但是主进程并不接受连接。由于 fork(2)的工作方式，当主进程衍生出工作进程时，每个工作进程都获得一个打开的套接字副本。

这就它的奇特之处。

每个工作进程拥有一个打开的套接字副本，并在此套接字上使用 accept(2)来接受连接。这时候由内核接管并在套接字的 10 个副本上均衡负载，它确保只有单个进程接受每个连接。即使在负载繁重的

110 附录 C preforking 服务器

情况下，内核也能够保证负载得到均衡，并且连接只由一个进程进行处理。

来比较一下 Mongrel 是如何实现负载均衡的。

考虑到 10 个无关进程无法共享一个套接字，因此每个进程都必须绑定到特定的端口。那么一些基础设施就必须置于所有的 Mongrel 进程之前。它必须知道每个 Mongrel 进程被绑至哪个端口，还必须确保每个 Mongrel 进程一次只处理一个连接，以及多个连接负载均衡。

无论从简单性还是资源利用有效性上来看，preforking 都是双赢。

C.3 高效的系统管理

这一点更多的是针对于人，而非技术。

preforking 服务器的管理人员一般只需要向主进程发出命令（通常是信号）。它会负载处理消息的跟踪并转发给工作进程。

当管理非 preforking 服务器的多个实例时，系统管理员必须跟踪每个实例，分别对其进行管理并确保命令得以执行。

preforking 服务器的基础样例

下面是 preforking 服务器一些非常基础的代码，可以使用多个进程并行响应请求并利用内核来进行负载均衡。更多有关 preforking 服务器的例子，我建议你查看 Spyglass 源代码（下一章）或 Unicorn 的源代码。

```
require 'socket'

# 打开一个套接字
socket = TCPServer.open('0.0.0.0', 8080)

# 预载入应用代码
# require 'config/environment'

# 将相关的信号转发给子进程
[:INT, :QUIT].each do |signal|
  Signal.trap(signal) {
    wpids.each { |wpid| Process.kill(signal, wpid) }
  }
end

# 跟踪子进程的 pid
wpids = []

5.times {
  wpids << fork do
    loop {
      connection = socket.accept
      connection.puts 'Hello Readers!'
      connection.close
    }
  end
}

Process.waitall
```

你可以结合 nc(1)或者 telnet(1)来查看实际效果。

```
$ nc localhost 8080
$ telnet localhost 8080
```

112 | 附录 C preforking 服务器

有没有注意到我偷偷添加了一些东西到代码里？我们之前并未看到过的`Process.waitall`出现在上面样例代码的最后一行。

`Process.waitall`只不过是`Process.wait`的一个便捷用法而已。它执行一个循环，等待所有的子进程退出，然后返回一个进程状态数组。如果你对进程状态信息不感兴趣的话，`Process.waitall`就派上用场了，它只是等待子进程退出。

附录D

Spyglass

如果你想了解更多有关 Unix 进程的知识，那么接下来就要学习一下 Spyglass 项目。为什么？因为它是专门用来演示 Unix 编程概念的。

如果购买了本书却没有获得随书代码，请给我发电子邮件，我会帮你解决：jesse@workingwithunixprocesses.com。

你读过的那些案例研究也是为了演示同样的东西。如果你是 Unix 编程的初学者，那么它们有时候会显得太过深入而难于理解。Spyglass 可以弥补这两者之间的差距。

D.1 Spyglass的体系

Spyglass 是一个 Web 服务器，它向外部世界打开一个套接字并处理 Web 请求。Spyglass 解析 HTTP，它兼容 Rack，表现相当出色。

下面简要归纳了如何启动 Spyglass 服务器，以及它接收到 HTTP 请求后的处理过程。

D.2 启动Spyglass

```
$ spyglass  
$ spyglass -p other_port  
$ spyglass -h # 帮助信息
```

D.3 请求抵达之前

启动后,控制权交予 `Spyglass::Lookout`。这个类并不会预载入 Rack 应用,对于 HTTP 也一无所知,它只是等待某个连接。这时的 `Spyglass` 相当轻便,仅仅是一个打开的套接字而已。

D.4 建立连接

当 `Spyglass::Lookout` 注意到某个连接已经建立,它便衍生出一个 `Spyglass::Master` 来处理这个连接。在衍生出主进程之后,`Spyglass::Lookout` 使用 `Process.wait` 保持空闲,直到主进程退出。

`Spyglass::Master` 负责预载入 Rack 应用并衍生/看护工作进程。主进程本身并不了解 HTTP 解析或请求处理。

实际的工作是在 `Spyglass::Worker` 中完成的。它使用 preforking 那章介绍的方法来接受连接,依靠内核进行负载均衡。一旦获得一个连接,它就解析 HTTP 请求,调用 Rack 应用,并为客户端生成响应。

D.5 万事皆毕

只要有稳定的流量进入，Spyglass 就会一直作为 preforking 服务器进行运作。如果在内部计时器超时之前没有接收到任何接入请求，那么主进程和它所有的工作进程都会退出。控制权返回到 `Spyglass::Lookout`，这个工作流程再次从头开始。

D.6 启程

Spyglass 并非可用于生产环境的服务器，所以千万别急着把它用在你的项目中。它是一个旨在供人阅读学习的代码库，其中包含了大量注释以及由 rocco^①生成的格式化文档。

此时最适合的事情就是从终端进入本书代码的 `code` 目录，找到 Spyglass 代码库，然后执行 `rake read`。这将会在浏览器中打开格式化文档，为你提供阅读的乐趣。

现在就去阅读代码吧！愿 fork(2)与你同在！

① <http://rtomayko.github.com/rocco>

索引

- ARGV, 27, 28, 86
- CoW, 41, 42, 43
- exec, 84, 85, 86, 89, 90
- IO.pipe, 69, 70, 74
- IO.popen, 88, 89, 90, 91
- Kernel#abort, 32, 33
- Kernel#at_exit, 32
- Kernel#exit, 31, 32
- Kernel#exit!, 32
- Kernel#system, 86, 87, 91
- Open3, 89, 90
- preforking, 50, 106, 107, 108, 109, 110, 114, 115
- Process.detach, 53, 54
- Process.getpgid, 83
- Process.getrlimit, 19, 23
- Process.setrlimit, 21, 23
- Process.setsid, 78, 81, 83
- Process.spawn, 87
- Process.wait, 45, 46, 55
- Process.wait2, 46, 47, 48
- Process.waitpid, 48
- Process.waitpid2, 48
- Resque, 2, 95
- Resque worker, 95
- shelling out, 84
- SIGCHLD, 55, 56
- Spyglass, 4, 106, 109, 113, 114, 115
- Unicorn, 2, 50, 100, 101, 107, 108, 110
- 父进程, 12, 34, 35, 36
- 孤儿进程, 39, 77
- 管道, 14, 68, 70, 73, 89
- 环境变量, 24
- 会话组, 78, 80, 81
- 僵尸进程, 52, 54
- 进程组, 78, 79
- 竞争条件, 49
- 内核, 5, 9, 12, 15, 16, 19
- 手册页, 6, 7
- 守护进程, 40, 75, 76, 77
- 退出码, 29, 31
- 文件描述符, 14, 16, 18, 19
- 衍生, 34
- 用户空间, 5
- 子进程, 34, 35

“这本书让我意识到Unix编程是多么简单而强大，而我对Unix编程的了解又是何其之少！在下一个版本的Thin的设计中，这本书无疑会对我有很大的启发。”

——Marc-André Cournoyer，创建了Thin Web服务器

“《理解Unix进程》这本书填补了当今许多开发人员的空白。掌握Unix的基本原理无需编写C代码。本书就说明了这一点。”

——David Bryant Copeland，著有*Build Awesome Command-Line Applications in Ruby*

《理解Unix进程》教你如何游刃有余地利用系统编程技术。许多Web开发人员或许对Unix系统的基本原理知之甚少，本书可弥补你这方面的缺憾，帮你从底层了解Unix系统进程的机制。本书的重点内容如下：

- ◆ 文件描述符及其运作机制
- ◆ 何时才需要守护进程
- ◆ 如何用fork(2)创建新进程
- ◆ 退出进程的4种不同的方式
- ◆ 对于生成shell命令的实际考量以及如何避免这种情况
- ◆ 从高级层面上讨论了创建进程所带来的开销及陷阱
- ◆ Resque和Unicorn的内部工作原理

书中代码都是用Ruby写的，但是其原理适用于任何编程语言，无论你用的是C、Python，还是新锐的Go语言。



图灵社区：www.ituring.com.cn

新浪微博：[@图灵教育](#) [@图灵社区](#)

反馈/投稿/推荐信箱：contact@turing.com.cn

热线：(010)51095186转604

分类建议 计算机/操作系统/Unix

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-31689-9



ISBN 978-7-115-31689-9

定价：29.00元