第 1 章 Erlang/OTP平台 1

本章概要

理解并发和Erlang的进程模型

Erlang的容错与分布式支持

Erlang运行时系统的重要属性

什么是函数式编程，如何用Erlang进行函数式编程

既然你正读着这本书，想必知道Erlang是一门编程语言——而且还是一门很有意思的语言，但正如书名所示，我们所关注的是如何用Erlang创建真实而鲜活的系统。为了实现这一目标，我们就需要OTP框架。Erlang的任何发布版本都带有这套框架，它与Erlang紧密集成，已令人难以将之与普通Erlang标准库明确地区分开来。因此，我们常用Erlang/OTP来同时指代二者或其中之一。尽管二者间的关系如此密切，但真正明了OTP的用途与运用之道的Erlang程序员却为数不多，即便真相往往只有一步之遥。就让本书来为你带路吧。

OTP 是什么意思

OTP 最初是开放电信平台（Open Telecom Platform）的缩写，Erlang 开源前这个名字多少还有点品牌效应。如今可没人稀罕它了；现在 OTP 就是 OTP。无论 Erlang 还是 OTP 都早已不

再局限于电信应用：更贴切的名字应该是“并发系统平台”。

作为编程语言，Erlang可以简化高度并行分布式容错系统的构建，并以此闻名。在跳到OTP框架相关内容之前，我们会先在第2章对该语言作一个全面的综述。不过话说回来，为什么非学OTP不可呢？或许你更乐于埋头实现自己的解决方案？且让我们来看看OTP的优点：

生产效率——运用OTP可在短时间内交付产品级的系统；

稳定性——基于OTP的代码可以更集中于逻辑，并避免重新实现那些容易出错而每个实际系统又都必备的基础功能，如进程管理、服务器、状态机等；

监督——这是由框架提供的一套简便的监视和控制运行时系统的机制，既有自动化方式，也有图形用户界面方式；

可升级——框架为处理代码升级提供了一套系统化的模式；

可靠的代码库——OTP的代码坚如磐石并全部经过严格的实战检验。

尽管有诸多优点，但恐怕对大部分Erlang程序员来说，OTP仍然神秘莫测，必须在艰涩的文档中摸爬滚打千锤百炼方能习得。而这正是我们所要改变的状况。据我们所知，本书是第一本专注于OTP学习的书，而我们想要告诉你的是这一过程远比你想象的要轻松。我们保证你学了肯定不会后悔。

在本书结束时，你将完整地掌握OTP框架中的概念、库与编程模式，学会如何运用OTP的组件和理念开发单个Erlang程序及整套基于Erlang的系统，并令其兼具容错、分布、并发、高效和易于监控的特点。你可能还会学到一些此前未曾留意过的有关Erlang语言、运行时系统、库和工

具的细节。

本章我们所要讨论的是Erlang/OTP平台中的那些用于构建OTP本身的核心概念和特性，包括：

并发编程；

容错；

分布式编程；

Erlang虚拟机和运行时系统；

Erlang的核心函数式语言。

我们不会一上来就劈头盖脸地扔出一大把概念，此处的重点是让你了解各种具体内容背后的思想，以便更好地理解后续第2、3章中论及的具体内容。Erlang很特别，本书中的诸多内容都需要花费时间去适应。在深入技术细节之前，我们希望这一章能让你明白各种机制背后的动机。

1.1 基于进程的并发编程

Erlang为了解决*并发*问题——也就是让多个任务同时运行——采用了全新的设计。并发是设计该语言时的核心关注点。借助进程的概念，Erlang内置的并发支持可以彻底隔离任务，令你设计出容错的架构，并充分发挥当今多核硬件的能力。不过在继续深入之前，我们有必要把并发和进程这两个术语的确切含义解释清楚。

1.1.1 理解并发

并发就是并行吗？不完全是，至少在讨论计算机和编程时二者并不等同。

有个常用的半正式定义是这么说的：“并发，用于形容那些无须以特定顺序执行的事物。”比如分别对两副牌排序，你可以排完一副再排另一副；三头六臂的话也可以两副并行一起来。这两个任务在执行顺序上不受约束，因此，它们是并发任务。它们的完成顺序也无所谓，你可以在两个任务间交替切换直至二者全部完成；倘若有多余的手脚（或是多个帮手），也可按真正的并行方式同时进行。[①]

这听起来好像有点怪：只有同时发生的任务才能算是并发任务吧？嗯，这个定义的关键在于它们可以同时发生，而我们则可以按自己的意愿随意调度它们。有些必须同时进行的任务相互之间根本无法独立；还有些任务相互之间虽然独立却不并发，必须按特定的顺序进行，比如做蛋卷必须先打蛋。除了这些情况其余都算并发。

Erlang的一大优势就是它帮你隐藏了任务实际执行的细节。如图1-1所示，如果有额外的CPU（或核，或超线程），Erlang会利用它们并行执行更多并发任务。如果没有，Erlang会利用现有的CPU处理能力一点一点地交替执行任务。你不必操心这些细节，Erlang程序能够自动适配不同的硬件——CPU越多它们跑得越快，前提是任务的组织方式允许它们被并发执行。

① 关于“并行”与“并发”的区别，另一个常见的说法是，“并行”形容两个或多个任务在同一时间同时发生，而“并发”形容两个或多个任务在一个时间段内交替进行，同一时间内只有一个任务在执行。而本书中的定义则将“并行”列为“并发”的一个概念子集。——译者注

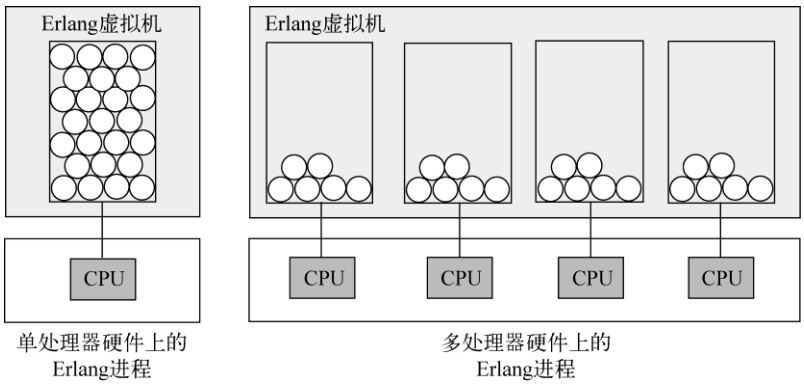

图1-1 分别运行于单处理器硬件和多处理器硬件上的Erlang进程。运行时系统会自动将负载分配到可用的CPU资源上

但若任务就是无法并发呢？若你的程序就必须先执行X，再轮到Y，最后是Z呢？这时你就得好好考虑一下待解决的问题中所隐含的实际的依赖关系了。也许X和Y无所谓谁先谁后，只要它们在Z之前完成就行。又或许X和Y各自完成了一部分的时候就可以部分启动Z。在这个问题上没有捷径可循，但往往稍微动下脑筋便收效甚佳，越有经验越容易。

重新考察问题，削减任务间不必要的依赖，可以令代码在现代硬件上运行得更为高效。但这通常不该是你的首要关注点。将程序中内聚性低的部分隔离成独立的任务，最重要的收益是更清晰可读的代码，你的精力也得以集中到实际问题上；相反，试图一蹴而就一次性完成多个任务只会令你事倍功半。这种隔离意味着生产效率的提高和缺陷数量的降低。但首先，我们需要一种更具体的表征独立任务的手段。

1.1.2 Erlang 的进程模型

在Erlang中，并发的基本单位是进程。每个进程代表一个持续的活动，它是某段程序代码的

执行代理，与其他按各自的节奏执行自身代码的进程一起并发运行。进程和人有些相似：个人占有的东西不能与他人共用。这不是慷慨与否的问题，以食物为例，你多吃一口，别人必定少吃一口；更关键的是，你吃了不干净的东西，也只有你一个人会生病。每个人都凭借自己的头脑和脏器独立于他人去思考和生存。这也正是进程的行为模式，相互隔离，并确保自身内部状态的改变不对其他进程造成影响。

进程拥有自己的工作内存空间和自己的信箱，其中信箱用于存放外来消息；而许多其他语言和操作系统中的线程却是共享相同内存空间的并发活动（随之而来的是层出不穷的互踩脚趾的机会）。因此与线程相比，Erlang进程更加安全，不会有人在周围指手划脚，也不必担心有人在下一微秒出其不意地篡改自己的数据。因此我们说进程封装了状态。

进程的一个实例

以 Web 服务器为例：服务器接收网页请求，查找该网页的数据，再将数据传回请求发起方（有时会先分块，再一块一块地传输），如果请求失败则需要返回一条错误信息。显然，请求与请求之间的关联度不高[①]；然而如果服务器一次仅接受一个请求，不处理完毕就不受理下一个的话，热门站点上请求队列的长度很快就会过千。

反之，如果在请求抵达后服务器立即分配独立的进程来处理请求的话，就不再需要队列，大部分请求整体的处理时延也将趋于相等。这时单个进程所封装的状态就是请求中的URL、响应的接收方以及当前请求的处理进度。请求处理完毕后，进程退出，清理掉这个请求并回收内存。一旦某个 bug 导致某个请求失败，只有对应的那个进程会崩溃，其他进程仍完好无损。

由于进程不能直接改变其他进程的内部状态，容错便相对容易。无论一个进程执行的代码有

① 不过，在Web 2.0的大潮下，情况可能有所不同。——译者注

多烂，其他进程的内部状态都不会受损。即便是在程序内较细粒度的层面上，你也同样可以设置这种隔离，就好像电脑桌面上的浏览器和文字处理器之间的关系一样。事实证明这种隔离非常有效，后续我们讲到进程监督时你就会有所体会了。

由于进程之间互不共享内部状态，它们只能进行复制式通信。一个进程要跟其他进程交换信息，就会发送一条消息。这条消息是发送方所持有数据的一个只读副本。消息传递的基本语义使分布式与Erlang自然地融为一体。现实生活中，你是无法共享线路上的数据的——你只能复制它。Erlang的进程间通信机制总会让接收方获取一份私有的消息副本，即便消息收发双方同处在一台机器上。初听起来可能很奇怪，但这意味着网络编程和单机编程完全一样！

这种分布透明性支持令Erlang程序员可以将网络视作一组资源的集合——我们不用关心进程X和进程Y是否运行在不同的机器上，因为无论它们运行在何处，通信方法都一样。我们将在下一小节对各种语言和系统中的进程通信方式做一个综述，以让你明白其中的利弊权衡。

1.1.3 4 种进程通信范式

所有并发系统的核心问题，都是信息共享，这也是所有实现者都必须解决的问题。将一个问题切分为若干不同的任务后，任务间如何通信？这个问题看起来简单，实则不然，不少大师级人物都在这个问题上绞尽了脑汁，并经年累月地尝试了很多手段，其中一些被收编为编程语言的特性，另一些则形成了独立的库。

我们将简要讨论一下近年来受到广泛认同的四种进程通信手段。我们不打算在此耗费过多的时间，但应该足以让你大致了解当今各种语言和系统中的进程通信手段，同时我们会着重介绍它

们和Erlang的区别。这4种手段分别是持锁共享内存、软件事务性内存、future和消息传递。让我们从最古老但仍旧最流行的方法开始。

1. 持锁共享内存

共享内存差不多算得上是我们这个时代的GOTO：身为当今主流的进程通信技术，它不仅历史悠久，而且跟GOTO语句一样，为你提供了一大把搬起石头砸自己的脚的办法。因为它，世代工程师都对并发产生了深深的恐惧（未曾对此心怀畏惧的人只是尚未尝试过罢了）。然而我们必须承认，正如GOTO一样，在一些底层场合中共享内存是无法取代的。

在这种范式下，两个或多个进程可以同时读写一块或多块常规内存区域。有时进程需要在这些内存区域上执行一些具备原子性的操作序列，其他进程在操作完成前不得访问这些区域，这就需要一种令该进程阻止其他进程访问这些区域的方法。解决之道就是锁：一种一次仅允许一个进程访问某种资源的构件。

锁的实现需要内存系统的支持，一般由硬件以特殊指令的形式提供支持。使用锁的时候进程之间必须通力合作：所有进程必须先获取锁才能访问共享内存区域，访问结束后还要将锁释放给其他进程使用。使用锁必须万分小心，差之毫厘谬以千里，因此，操作系统或编程语言分别以系统调用或语言构件的形式提供了信号量、监视器和互斥量等以基本锁为基础的高级构件，用以确保锁的请求和释放的正确性。尽管借助这些可以绕开最棘手的问题，但仍然难以克服锁的诸多缺点。随便列举几条：

即便冲突几率很低，锁的开销仍难以忽略；

它们是内存系统中的竞争热点；

出错的进程可能将正处于加锁状态的锁弃之不顾；

当锁出现问题时极难调试。

还有就是，用锁同步两三个进程还没什么问题，但随着进程数的增长，形势便会越发失控。最终很可能（在很多情况下，几乎肯定）会引发即便是经验最老到的开发者也无法预见的复杂死锁。

我们认为最好只在底层编程场合，比如在操作系统内核中，处理此类同步问题。但当今流行的大部分编程语言和脚本语言中都能看到锁的身影。它的广泛存在可能是因为锁本身的实现并不复杂，同时也不会对这些语言的编程模型造成影响。遗憾的是，虽然多处理器系统早在好几年前就已经普及，锁的广泛应用还是阻碍了我们对并发问题的思考和对大规模并发的应用。

2. 软件事务性内存（STM）

我们所要考察的第一种非传统方法就是STM（Software Transactional Memory，软件事务性内存）。目前可以在Haskell编程语言的GHC实现和基于JVM的Clojure语言中看到这种机制。STM将内存当作传统数据库，用事务来决定何时写入什么内容。通常，这种实现以一种乐观方式来规避锁：将一组读写访问视为单个操作，若两个进程同时试图访问共享区域，则各自启动一个事务，最终只有一个事务会成功。另一个进程会得知事务失败，并应该在检查共享区域的新内容后重试。该模型直截了当，谁都不需要等待其他进程释放锁。

STM的主要缺点在于你必须重试失败的事务（当然，它们可能再三失败）。事务系统本身也

会有比较显著的开销，另外在确定哪个进程成功之前，还需要额外的内存来存放你试图写入的数据。理想情况下，系统应该像支持虚拟内存那样对事务性内存提供硬件支持。

对程序员而言，STM的可控发性看起来比锁要好，只要竞争不会频繁导致事务重启，并发的优势就能充分得到发挥。我们认为该方法本质上是持锁共享内存的变体，它在操作系统层面的作用要更甚于应用编程层面。不过针对该课题的研究还很活跃，局面还可能会出现改观。

3. Future、Promise及同类机制

另一个更现代的手段是采用所谓的future或promise。这个概念还有另处一些形式；在E[①]和MultiLisp等语言以及Java的一个库中可以找到它的身影。类似的还有Id和Glasgow Haskell中的I-var和M-var、Concurrent Prolog中的并发逻辑变量，以及Oz中的数据流变量。

其基本思路是，每个future代表一个被外包到其他进程的计算结果，该进程可能跑在别的CPU甚至是别的计算机上。Future可以像其他对象一样被四处传递，但无法在计算完成之前读取结果，必须等待计算完成。这种方法虽然概念简单、简化了并发系统中的数据传递，但也令程序在远端进程故障和网络故障面前变得脆弱：计算结果尚未就绪而连接又不幸断开时，试图访问promise的值的代码便会无所适从。[②]

4. 消息传递

正如1.1.2节所说，Erlang进程靠消息传递来通信。这意味着接收进程实际上获取了一份独立的数据副本，发送方感知不到接收方对副本所做的任何操作。向发送方回传信息的唯一途径就是

① 这里的E语言指的是一种基于JVM的并发语言，和汉语编程的“易语言”不是一个东西。——译者注

② 本段中future和promise所指的是同一个概念，因此作者在此混用。——译者注

反向发送另一条消息。由此而得出的一个重要结论是，无论收发双方是身处同一台机器上还是被网络所隔离，它们都能以相同的方式进行通信。

消息传递一般可分为两类：同步方式和异步方式。在同步方式下，消息抵达接收端之前发送方什么事也做不了；在异步方式下，消息一经投递发送方便可立即着手于其他事务。（在现实世界中，机器间的同步通信要求接收方给发送方回复一个确认，以告知一切OK，不过这些细节对程序员而言可以是透明的。[①]）

同步很容易用异步实现，令接收方总是向发送方回传一个显式的回复即可。因此，Erlang中的消息传递原语是异步的。不过，发送方常常并不关心消息是否抵达——消息抵达与否其实没那么重要，因为你无法预知接收方接下来会怎样：说不定它旋即就挂掉了。这种异步的“即发即祷告”式的通信方法也意味着发送方在消息投递过程中无须挂起（特别是在慢速通信链路上发送消息时）。

当然，收发双方在这一层面的隔离并不是免费的。复制大型数据结构时成本很高，如果发送方还要保留数据副本，势必造成较高的内存消耗。在实践中，这意味着你必须在发送消息时小心掌控消息的大小和复杂度。不过一般来说，地道的Erlang程序用到的大部分消息都比较小，复制

① 如果考虑到收发进程的故障以及数据链路的故障，那么这些细节就无法对程序员透明。消息必然会有丢失或重复的概率；一旦透明化，应用层必然要面临因消息丢失而造成的无限等待以及消息重复的风险。事实上两点之间能保证消息既不丢失也不重复的容错消息传递协议是不存在的，最多是通过时间戳、超时、重传来实现以消息重复为代价避免消息丢失的协议。在这点上即便是TCP也不例外。实践中我们能够做到的就是尽量减小丢失和重复的概率，同时减小这些错误发生时对应用造成的代价。详情可参见D. Belsnes发表于1976年的Single Message Communication以及Henry Robinson的博文：Consensus with lossy links: Establishing a TCP connection

开销通常可以忽略。

我们希望以上论述能有助于你理解Erlang在当今并发编程领域中所处的位置。消息传递可能并不是其中最炫的技术，但就Erlang的发展历史来看，从系统工程角度出发，只有它最务实、最灵活。

1.1.4 用 Erlang 进程编程

开发Erlang程序时，你得问问自己：“在我要解决的问题中哪些活动是并发的——或者说哪些活动可以彼此相互独立地进行？”简要回答这个问题后，你便可以开始搭建系统了，你找出来的那些并发活动的每个实例都应该是系统中的一个独立的进程。

与大多数语言相反，Erlang中的并发很廉价。派生一个进程跟你在普通面向对象语言中分配一个对象的开销差不多。你可能得先好好适应一下，这个理念可真是闻所未闻！但等你适应之后，魔术便开始上演。描绘一组复杂运算，将之切分为若干并发部件，再全部建模为独立的进程。启动运算、派生进程、处理数据，在输出结果后的那一瞬间，所有进程神奇地烟消云散，它们的内部状态、它们持有的数据库句柄、它们打开的套接字，以及一切你不乐意手工清理的东西，都一并消失得无影无踪。

在这一节余下的内容中，我们将简要地看看进程是多么易于创建、多么轻量，它们之间的通信又是多么简单。

1. 创建进程：派生

（http://the-paper-trail.org/blog/?p=110）。——译者注

Erlang进程不是操作系统线程。它们由Erlang运行时系统实现，比线程要轻量得多，运行在商用硬件上的单个Erlang系统可以轻易派生出成百上千个进程。运行时系统中所有进程之间相互隔离；单个进程的内存不与其他进程共享，也不会被其他濒死或跑疯的进程破坏。

在现代操作系统中，典型的线程会在地址空间中为自己预留数兆的栈空间（也就是说32位的机器上并发线程数最多也就几千个），栈空间溢出便会导致崩溃。另一方面，Erlang进程在启动时栈空间只需要几百字节，并且会自动按需伸缩。

Erlang创建进程的语法很直接，如下所示。让我们来派生一个执行`io:format("erlang!")`后立即退出的进程：

```
spawn(io, format, ["erlang!"])
```

这就行了。（`spawn`函数有若干个变体，这是其中最简单一个。）这段代码启动一个独立进程，在终端上打印文本“ erlang!”后退出。

我们将在第2章给出Erlang语言及其语法的综述，但现在，在进一步阐述之前我们希望你能自行抓住示例代码的要点。Erlang的一大优点便是即便从来没见过这种语言，也能相对容易地读懂代码[①]。不妨试试看吧。

2. 进程之间怎么打交道

进程被派生并运行起来后还有别的事情要做——它们要进行信息交换。Erlang让通信变得简单。用于消息发送的基本运算符是！，读作“ bang”，用法是“ 目的地！消息”。这就是最简单的消息传递，就像寄明信片一样。OTP框架将进程间通信提升到了另一个层面，我们将在第3章对此

① 这个……真的是不敢苟同哇。——译者注

做深入探讨。现在，让我们来看看两个独立的并发进程间的通信，Erlang简单的通信机制一定会令你惊叹不已，详情参见代码清单1-1：

代码清单1-1 Erlang进程间通信

```
run() ->
    Pid = spawn(fun ping/0),
    Pid ! self(),
    receive
        pong -> ok
    end.

ping() ->
    receive
        From -> From ! pong
    end.
```

❶ From中包含发送方ID

花上一两分钟看看这段代码，相信即便从未接触过Erlang你也能看懂。稍微需要注意的地方：调用了spawn的一个变体，参数是一个函数引用，该函数“名为ping、参数数目为零”；再就是函数self()，它返回当前进程的标识符，用于告知新进程该把消息回复给谁[1]。

这便是Erlang进程通信的概况。每调用spawn一次都会得到一个新的进程标识符，用于唯一标识新创建的子进程。这个标识符后续可用于向子进程发送消息。每个进程都有一个信箱，无论进程繁忙与否，都会先把外来的消息存放在这儿，直到进程下次检查信箱前所有消息都寄存于此。随后进程会在自己乐意的时候用receive表达式从信箱中分检和读取消息，如同示例所示（此处是取走第一条就绪的消息）。

3. 进程的终止

进程完工后，便会消失。它的工作内存、信箱和其他资源都会被回收。该进程若是用作其他进程的数据源，那么它必须在终止前显式地将数据以消息的形式投递出去。

崩溃（异常）造成进程意外提前终止，一旦发生崩溃，其他进程会得到通知。之前我们曾说过进程之间相互独立，单个进程的崩溃不会破坏其他进程，因为它们互不共享内部状态。这构成了Erlang另一主要特性的一个支柱，该特性就是：容错。我们将在下一节做详细论述。

1.2　Erlang 的容错架构

在现实世界中容错就是真金白银。程序员并不完美，需求往往也不完善。正如航空工程师处理有缺陷的钢材和铝材一样，为了有效处理有缺陷的代码和数据，我们需要能够容错的系统，以防系统在遭遇突发状况时土崩瓦解。

和许多其他编程语言一样，Erlang也具备异常处理机制来捕获特定代码段的错误，不过它还有一套独一无二的可以有效处理进程故障的进程链接系统，我们即将在此进行讨论。

1.2.1　进程链接如何工作

Erlang进程意外退出时，会产生一个退出信号。所有与濒死进程链接的进程都会收到这个信号。默认情况下，接收方会一并退出并将信号传播给与它链接的其他进程，直到所有直接或间接链接在一起的进程统统退出为止（参见图1-2）。这种级联行为可以使一组进程像单个应用一样退出，因此系统整体重启时你不必担心是否还有残存下来未能完全关闭的进程。

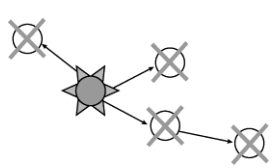

图1-2　崩溃进程发出的退出信号被传播到所有与之链接的进程，一般情况下它们会共同退出，以便完成对整个进程组的清理工作

前面我们曾提到过利用进程来清理复杂状态。其基本原理是：每个进程完整封装自己的全部状态，因此进程退出时系统的其余部分不会受损。如同单个进程一样，这一点对相互链接的进程组也同样适用。一个进程崩溃，与之协作的其他进程也一并退出，如此便可干净利落地抹掉之前建立的所有复杂状态，既节省了程序员的时间也减少了错误。

鼓励崩溃

当你还在绝望地纠结于如何挽回那些你可能根本无能为力的局面时，Erlang的哲学却是“鼓励崩溃”——精确记录下事发位置和经过后，把一切彻底抛下重新再来。这不太常见，但的确是一条强大的容错秘诀，而且按这个思路建立起来的系统无论多复杂度都可调试。

1.2.2 监督与退出信号捕捉

OTP实现容错的主要途径之一就是改写退出信号默认的传播行为。通过设置`trap_exit`进程标记，你可以令进程不再服从外来的退出信号，而是将之捕捉。这种情况下，进程接收到信号后，会先将其转为一条格式为`{'EXIT', Pid, Reason}`的消息，该消息描述了哪个进程出于什么原因而发生故障，然后这条消息会像普通消息一样被丢入信箱，捕捉到信号的进程就能分检并处理这类消息了。

这类会捕捉信号的进程有时被称为系统进程，它们执行的代码往往有别于普通的工作进程（即通常不捕捉信号的进程）。身为防范退出信号进一步传播的壁垒，系统进程阻断了与之链接的其他进程和外界之间的联系，因而可用于汇报故障乃至重启故障的子系统，正如图1-3所示。我们将这类进程称为监督者。

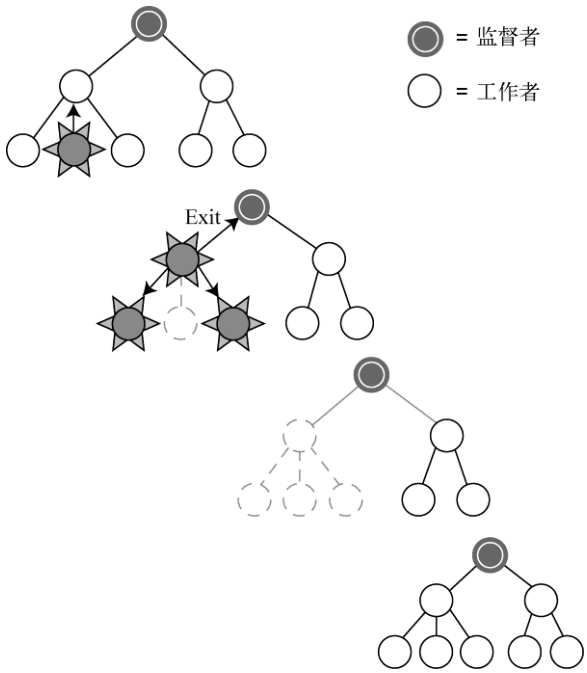

图1-3 监督者、工作者和信号：某工作进程的崩溃被级联传播至所有与之链接的其他进程，信号传播至监督者后，监督者将进程组重启。同一监督者辖区内的其他进程组则不受影响

停止并重启整个子系统的目的在于将系统恢复到一个已知的可正常工作的状态。这有点类似于重启电脑：通过重启你可以快刀斩乱麻地将电脑迅速恢复到可工作状态。但重启整台电脑的问题在于粒度太大。理想状况下，应该可以只重启系统的一部分，粒度越小越好。Erlang的进程链接与监督者共同提供了一种细粒度的“ 重启” 机制。

不过，如果就到此为止，你还是得自己从头实现监督机制，这需要缜密的思考和丰富的经验，bug的清除和各种边界情况的处理也要花费大量的时间。幸运的是，OTP框架提供了你所需要的一切：既有运用监督机制来构建应用程序的一套方法，也有稳定的、经过实战考验的基础库。

OTP允许监督者按预设的方式和次序来启动进程。我们还可以告知监督者如何在单个进程故障时重启其他进程、一段时间内尝试重启多少次后放弃重启等。你所要做的就是提供一些参数和回调。

然而系统不应该只允许一层监督者工作者结构。在任何复杂系统中，你都可以用多层的监督树在多个层级重启子系统来解决各种意外问题。

1.2.3 进程的分层容错

通过分层可以将相关的子系统归于同一个监督者的辖区之内。更重要的是，这样做可以定义多个层级的基准工作状态，随时供你重置。在图1-4中，你可以看到两个分别受独立监督进程监督的工作进程组A和B。这两个组和它们的监督者共同形成了一个更大的进程组C，并由树中更高层的一个监督者负责。

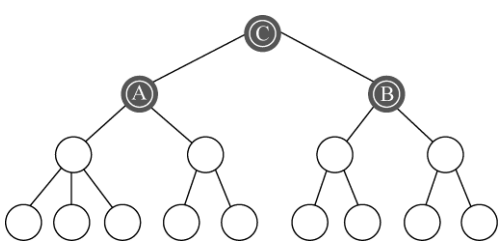

图1-4 一个分层的监督者工作者系统。如果出于某种原因监督者A崩溃或退出，它辖区内所有尚还存活的进程都会被强制关闭，同时C会收到通知，于是进程树的左半边会被重启。监督者B则不受影响，除非C决定关闭整个系统

我们假设A组进程的任务是输出供B组使用的数据流。无须B组，A组也可正常工作。更具体一点，比方说A组在处理和编码多媒体数据，B组则予以展现。我们再假设A组处理的数据中有一小部分受损，且数据损坏的模式无法在开发应用时预测。

这种畸形数据会导致A组的进程工作异常。按照鼓励崩溃的哲学，进程不会尝试去解决问题而是直接崩溃；由于进程相互隔离，其他进程并不会受到错误输入的影响。监督者检测到进程崩溃后，会将A组重启以回退到预设的基准状态，从而使整个系统恢复到一个已知的基准点。美妙的是身为展现系统的B组完全不知晓也不关心这个过程。只要A组能为B组持续提供足够的优质数据，使后者能为用户展现质量过关的内容，你的系统就是成功的。

通过隔离系统中不相关的部分并将它们组织成监督树，你可以划分出多个子系统，每个都可独立地在几分之一秒内完成重启，这样一来，即便你的系统碰上不可预期的错误也可以稳健地运行。若A组无法正常重启，它的监督者最终会放弃重启并将问题上报至C组的监督者。在这种情况下，C组的监督者会一并关闭B组然后停工。想象一下若系统中同时运行着几百个C这样的子系统，这就相当于因数据错误而丢弃了一个多媒体连接，其他连接仍然照常工作。

然而，既然大家都跑在同一台机器上，就不得不共用一些东西：内存、硬盘驱动器、网络连接，乃至处理器和所有相关电路，还有一样最重要的，就是从同一个插座上接出来的那根电源线。如果这些东西中有一样发生故障或断开，不管怎么分层怎么做进程隔离都无法避免宕机。这就把我们带入了下一个主题，也就是分布式——能助你实现最高级别的容错并令你的解决方案伸缩自如的，正是Erlang的这个特性。

1.3 分布式 Erlang

借助于语言属性和基于复制的进程通信，Erlang程序天然就可以分布到多台计算机上。要问

为什么，且让我们来看两个用Java或C++这类语言写成的进程，它们运作良好并以共享内存为通信手段。假设你已经搞定了锁的问题，一切精准而高效，但就在你试图将其中一个线程挪到另一台机器上时，问题出现了。或许是为了利用更高效的计算能力和内存，或许是为了预防两个线程在硬件故障造成的宕机中同时挂掉，无论如何，这一刻降临时，程序员往往被迫重新设计代码结构，以便配合新的分布式环境中迥异的通信机制。显然，这将耗费大量的开发成本，而且很可能会引入数年才能彻底清除的bug。

Erlang程序却不受这些问题的影响。正如我们在1.1.2节所解释的那样，Erlang规避了数据共享并通过复制进行通信，这使得Erlang代码可以直接分布到多台机器上。在命令式语言里用线程编程时，各部分代码往往会因数据共享引入复杂的依赖关系；这类问题在Erlang中则很少见。今天能跑在你的笔记本上，明天就能跑在集群上。

Erlang应用通常可以直接分布到多个网络节点上，这同时意味着伸缩性问题也简化为一个数量级。你仍然需要考虑好各类进程的职能，每类进程需要运行多少个实例，在哪些机器上运行，怎样均衡负载以及怎样管理数据；但至少以下这类问题不用再劳你费心了：“我到底该怎么切分现有的程序才能搭建出冗余的分布式系统？”，“它们之间该怎么通信？”，还有“我该怎样得体地处理故障？”。

实际案例

在一个雇主那里，我们在网络上跑着多个各式各样的 Erlang 应用。独立 OTP 应用的类型大概有至少 15 种，它们协同完成某个共同的任务。做集成测试时，我们固然可以在 15 个虚拟机上测试这个跑着 15 个不同应用的集群，但这绝非最便捷的做法。实际上我们一行代码也没

改，就在单个 Erlang 实例上启动了所有应用并完成了测试。在那个节点上，它们使用相同的通信方式、相同的语法，就跟分布在网络中的多个节点上一模一样。

这个案例所示范的概念称作位置透明性。其基本含义是当你用进程的唯一 ID 作为目标地址向进程发送消息时，你不用了解甚至不用关心那个进程所处的位置——只要接收方还“活得好好的”，Erlang 运行时系统便会替你把消息投递到它的信箱里[①]。

现在你已经大致了解Erlang能做些什么了，下面我们来讲讲位于核心的引擎，好让你明白在Erlang程序运行时究竟发生了些什么。

1.4 Erlang 运行时系统和虚拟机

那么上述这一切背后的驱动力是什么呢？标准Erlang实现的核心是一个称作Erlang运行时系统（ERTS）的应用：这是一大块用C语言写成的代码，负责Erlang中所有底层的玩意儿。通过它你才能跟文件系统和终端打交道，它还处理内存，实现Erlang进程的也是它。ERTS知道如何将这些进程分布到现有的CPU资源上才能充分发挥计算机硬件的能力。同时，哪怕你只有一个单核CPU它也能实现Erlang进程的并发执行。ERTS还负责处理进程间的消息传递，并使处在不同机器上运行在各自的ERTS中的进程能够像身处同一台机器上一样进行通信。Erlang中所有需要底层支持的东西都由ERTS处理，所以ERTS移植到哪个平台Erlang就能在哪个平台上跑。

ERTS中特别重要的一个部分就是Erlang的虚拟机模拟器：这是执行Erlang程序经编译后产出的字节码的地方。这个虚拟机也就是Bogdan Erlang抽象机（BEAM）[②]，它非常高效：虽然我们

① 作者在此处显然不够严谨，如果收发双方身处不同机器，要投递成功至少还需要双方之间网络畅通。——译者注

② Bogdan指的是BEAM的发明人Bogumil Hausman。BEAM原本叫作Turbo Erlang，后来由于复杂的法律问题更名为BEAM。详情参见4.7节。——译者注

也可以将Erlang程序编译为本地机器码，但一般没有那个必要，因为BEAM模拟器已经够快的了。注意虚拟机和ERTS之间并没有明确的界线；通常人们（包括我们自己）口中的Erlang VM指的就是模拟器加上运行时系统。

运行时系统中有许多有趣的特性，若不在文档中挖地三尺或是长期浸淫于Erlang邮件列表，你是不会知道的。它们正是Erlang能同时处理那么多进程的精要之所在，也是Erlang如此特别的原因之一。Erlang语言的基本哲学加上实现者所采取的务实方案，共同为我们带来了异常高效、面向生产的稳定系统。在这一节，我们将讨论促成了Erlang的强大和高效的3个重要方面：

调度器——处理运行中的Erlang进程，令所有就绪的进程共享可用的CPU资源，并在新消息到达或发生超时的时候唤醒相应的睡眠中的进程；

I/O模型——防止系统在进程与外部设备通信时阻塞，令系统平稳运行；

垃圾回收器——回收不再使用的内存。

我们从调度器开讲。

1.4.1 调度器

经过多年的演进，ERTS的进程调度器提供了其他平台无法比拟的灵活性。它最初的设计目标是在单CPU上并发运行轻量级Erlang进程，而不用关心底层用的是什么操作系统。ERTS运行的时候通常就是单个操作系统进程（在操作系统的进程列表中一般名为beam或werl）。这个进程中，就跑着管理所有Erlang进程的调度器。

随着线程在大多数操作系统中的普及，ERTS也有所变化，开始将I/O系统这类东西从运行

Erlang进程的线程中拿出来，放到独立的线程中去，但完成主体工作的线程仍然只有一个。如果你用的是多核系统，就必须在同一台机器上运行多个ERTS实例。不过从2006年5月起，Erlang/OTP第11版中增加了对称多处理器（SMP）支持。这是一项重大突破，令Erlang运行时系统可以在内部使用不止一个进程调度器，每个占用一个独立的操作系统线程。其效果参见图1-1。

这意味着现在Erlang进程可以以*n*：*m*的方式映射到操作系统线程。每个调度器处理一个进程池。可并行运行的Erlang进程最多能有*m*个（每个调度器线程执行一个），但同一池内的进程仍像之前所有进程共用一个调度器那样分时运行。在此基础之上，进程可以在进程池之间迁移以便维持可用调度器上的负载均衡。在最新的Erlang/OTP发布版中，甚至可以根据机器上CPU的拓扑情况将进程绑定到特定的调度器上，从而更好地利用硬件的缓存架构。这意味着，大多数时候，作为一名Erlang程序员你不用担心手头有多少CPU或有多少个核：你只要中规中矩地写程序，并尽量将程序切分为尺寸适中的并行任务就好，负载均衡之类的事情就让Erlang运行时系统去操心吧。不管是单核还是128核——都一样，只会更快。

Erlang新手常犯的一个错误就是过分依赖时序，这往往导致那些在他们的笔记本或工作站上运行良好的程序一迁移到多核服务器上就出错，因为在多核服务器上时序的不确定性要大得多。要发现这类问题，只能借助测试。好在现在的笔记本基本上都至少是双核的了，这类错误也可以被尽早发现。

Erlang的调度器还涉及运行时系统的另一个重要特性：I/O子系统。这正是我们的下一个主题。

1.4.2 I/O 与调度

很多并发语言都有的一个毛病就是它们没怎么拿I/O当回事儿。单个进程进行I/O时，它们几乎都存在整个系统或大半系统阻塞的问题。这真是既恼人又没有必要，尤其是Erlang早在二十年前就已经解决了这个问题。在前一节，我们曾讨论过Erlang的进程调度器。除了处理进程调度，调度器还替系统优雅地处理了I/O问题。在系统的最底层，Erlang以事件驱动的方式处理所有I/O，当数据进出系统时，程序可以以非阻塞方式完成数据处理。这降低了连接建立和断开的频次，还避免了OS层面上的加锁开销和上下文切换。

这是一种高效的I/O处理方法。可惜，程序员往往难以分析和理解这种技术，这也是为什么只有在明确要求高可靠性和低延迟的系统中才能见到这种技术。早在2001年，Dan Kegel就在他的论文The C10K Problem中描述过这个问题，虽然现在已经略显过时，但这篇文章仍然很值得一读。它针对这个问题及可能的解决方案给出了良好的综述。这些方案实现起来全都既复杂又痛苦，这正是Erlang运行时系统替你包办这些问题的原因。Erlang在进程调度器中整合了基于事件的I/O系统。事实上，你一点儿都不用操心就能享受一切便利。这让用Erlang/OTP构建高可靠性系统变得轻松了很多。

我们要讲解的最后一个ERTS特性就是内存管理。它对进程所起的作用超出你的想象。

1.4.3 进程隔离与垃圾回收器

如你所想，Erlang跟Java等大部分现代语言一样，会自动管理内存。这儿没有显式的释放操作。相反，垃圾回收器会定期搜寻和回收不再使用的内存。垃圾回收（GC）算法是一片广袤而复杂的研究领域，我们无法在这儿给出详尽的阐述；不过针对那些对此有一定了解又心怀好奇的

读者，可以告诉你Erlang当前使用的是一个简单明了的分代复制式垃圾回收器。

虽然实现相对简单，Erlang程序却不太会像其他语言开发的系统那样在GC时遭受停顿。这主要因为Erlang进程之间的隔离：每个进程所使用的内存都是自己的，随进程的创建和结束而分配和释放。听起来好像没什么要紧，实则不然。首先，这意味着垃圾回收器可以在不影响其他进程运行的前提下单独暂停目标进程。其次，单个进程占用的内存通常较小，遍历可以快速完成。（也有内存占用量大的进程，但这些进程一般不用做出快速响应。）再次，调度器知道每个进程最后一次运行的时间，如果某个进程自上次垃圾回收后什么也没干，调度器会跳过它。正是这些因素让Erlang既可以轻松使用垃圾回收器，又可以保证较短的停顿时间。除此以外，有时候进程自派生到完工，再到退出，根本就没有触发过垃圾回收。这种情况下，进程的作用相当于一块昙花一现的内存，除自动分配和释放外，没有任何额外的开销。

本节所描述的运行时系统的特性使Erlang程序能够充分利用可用的CPU来运行大量进程、执行I/O操作，并自动回收内存，与此同时还能维持软实时响应能力。了解了平台这些方面的知识，便可更好地理解自己的系统自启动后的各种行为。

最后，在这一章结束前，我们还要说一说Erlang中的函数式编程。不会太罗嗦，因为我们还会在下一章详细探讨Erlang语言。

1.5 函数式编程：Erlang 的处世之道

对本书的许多读者而言，函数式编程可能还是个新概念。对另一些人来说则不是。函数式编程绝对算不上Erlang的决定性特征——并发才是——但它仍是该语言的一个重要方面。近年来越

来越多的人意识到，函数式编程及其背后的观念天然适用于并发与分布式编程问题。（单提一下Google MapReduce就够了吧？）

要对函数式编程做一个总结的话，其主要思想就是将函数看作和整数、字符串一样的数据；运用函数调用而非`while`或`for`这样的循环结构来表达算法；以及不修改变量和值（参见附录B中对引用透明性和列表的讨论）。这些看似人为设置的约束，从工程角度来看却具有非凡的意义，Erlang程序本身也因此极为自然而可读。

Erlang并不是一门“纯粹”的函数式语言——它仍仰仗副作用。但仅限于一个操作：复制式消息传递[①]。每条消息都会对外界产生影响，同时外界也通过向进程发消息来影响它们。但每个进程本身运行的基本上都是纯函数式的程序。遵循这个模型的程序比C++和Java这类传统语言写成的程序更易于分析，同时又不至于像Haskell那样迫使你在程序中使用monad。

在下一章，我们会介绍Erlang编程语言中的关键部分。估计不少读者会觉得这个语法很别扭——它主要借鉴自Prolog而非C。虽然异于常规，它却并不复杂。忍上一段时间，它就会成为你的第二天性。待熟悉之后，你便能够看明白大部分Erlang内核模块的代码了，这才是真刀真枪的语法测试：看看你最后能否看得懂？

1.6 小结

本章我们介绍了Erlang/OTP平台中身为OTP基石的最关键的概念和特性：用进程和消息传递进行并发编程、通过链接来容错、分布式编程、Erlang运行时系统和虚拟机，还有Erlang核心的

函数式语言。这一切共同构成了一个稳固、高效、灵活的平台，让你得以构筑可靠、低延迟、高度可用的系统。

如果你曾有过Erlang相关的经验，这些内容大部分都不新鲜，但希望我们的讲解还算有意思，至少能让你看到一些先前未曾留意的方面。目前为止，我们还没怎么讨论OTP框架。我们把这部分内容放在第3章，不过届时进度会比较快，所以趁现在好好回味一下这些背景材料吧。接下来，第2章会针对Erlang编程语言做一个完整的综述。

① 这点也是不准确的，Erlang中的进程派生、针对进程字典的操作以及各种I/O操作都依赖于副作用。——译者注