英文原名：**Stuff Goes Bad: Erlang In Anger**
英文作者：**FRED HEBERT**

硝烟中的 Erlang

——Erlang 生产系统问题诊断、调试、解决指南

邓辉、孙鸣　译

译者序

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

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

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

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

强大的并发支持也是Erlang的特色之一，在这一点上常常被其他语言争相模仿。不过，Erlang和模仿者之间有个根本的不同点：公平调度。为了做到公平调度，Erlang可谓“不择手段”，并做到了“令人发指”的地步[2]。为什么要费劲做这些工作呢？对于一个高并发系统来讲，软实时、低延时、可响应性往往是渴求的目标，同时也是一项困难的工作。尤其是，在系统过载时，多么希望能具有一致的、可预测的服务降级能力。而公平调度则是达成这些目标的最佳手段，Erlang也是目前唯一在并发上做到公平调度的语言。

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

从某种意义上讲，Erlang不仅是一门编程语言，更是一个系统平台。它不仅提供了开发阶段需要的支持，更提供了其他语言所没有的运行阶段的强大支持。其实，在静态检查和测试阶段发现的问题往往都是些“不那么有趣”的问题，那些逃逸出来的bug才是真正难对付的。特别是对于涉及并发和分布式的bug，往往难以通过静态检查和测试发现，并且传统的调试

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

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

手段也无法奏效[3]。而Erlang则提供了强大的运行时问题诊断、调试、解决手段。使用Erlang的remote shell、tracing、自省机制以及强大的并发和容错支持，我们可以在系统工作时，深入到系统内部，进行问题诊断、跟踪和修正。甚至在需要时在线对其进行“高侵入性”的外科手术。一旦你用这种方法解决过一个困难的问题，你就再也离不开它了。如果要在静态类型和这项能力间进行选择，我会毫不犹豫地选择后者[4]。

对于Erlang存在的问题[5]，提得最多的有两个：一个是缺乏静态类型支持，另一个是性能问题。Erlang是动态类型语言，往往会被认为不适合构架大型的系统。我自己也非常喜欢静态类型。一个强大的静态类型系统不但能够大大提升代码的可读性，而且还为我们提供了一个在逻辑层面进行思考、设计的强大框架，另外还可以让编译器、IDE等获取更深入的代码结构和语义信息从而提供更高级的静态分析支持。

不过，在构建大型系统方面，我有些不同看法[6]。如果说互联网是目前最庞大的系统，相信没有人会反对。那么这个如此庞大的系统能构建起来的原因是什么呢？显然不是因为静态类型，根本原因在于系统的组织和交互方式。互联网中的每个部件都是彼此间隔离的实体，通过定义良好的协议相互通信，一个部件的失效不会导致其他部件出现问题。这种方式和Erlang的设计哲学是同构的。每个Erlang系统都是一个小型的互联网系统，每个进程对应一台主机，进程间的消息对应协议，一个进程的崩溃不会影响其他进程……。Erlang所推崇的设计哲学：crash-oriented以及protocol-oriented，是架构大型系统的最佳方式[7]。

当然现在，你鱼和熊掌可以兼得，Erlang已经支持丰富的静态类型定义和标注功能[8]，并且可以通过Dialyzer工具进行类型推导和静态检查。

再来说说性能问题。在计算密集型领域，Erlang确实性能不高[9]。因此，如果你要编写的是需要大量计算的工具程序，那么Erlang是不适合的。不过，如果说涉及计算的部分只是系统中的一个局部模块，而亟需解决的是一些更困难的系统层面的设计问题：并发、分布式、伸缩、容错、短响应时间、在线升级以及调试运维等等，那么Erlang则是最佳选择。此时，可以用Erlang作为工具来解决这些系统层面的难题，局部的计算热点可以用其他语言（比如C语言）甚至硬件来完成。Erlang提供了多种和其他语言以及硬件集成的方法，非常方便，可以根据自己的需要（安全性、性能）进行选择。

我前段时间曾经开发过一款webRTC实时媒体网关[10]，就是采用了Erlang + C的方案。其中涉及媒体处理的部分全部用C编写，通过NIF和Erlang交互，系统层面的难题则都交给Erlang完成。系统上线几个月，用户量就达到数百万。期间，系统运行稳定，扩容方便，处理性能也不错（尤其是高负载时的服务降级情况令人满意）。不是说使用其他语言无法做到，不过要付出的努力何止 10 倍[11]！

---

[3] 比如一个和竞争有关的 bug，一旦加上断点，竞争可能就不会出现了。
[4] 其实可以兼得，Erlang 现在已经支持类型定义、标注和推导。
[5] 那些主观性太强的我们不讨论，比如，有人觉得 Erlang 语法怪异。
[6] 这些看法只针对 Erlang。对于其他的动态类型语言，在程序规模变大后，确实有难以理解和维护的问题。在 Erlang 中，由于其 let it crash 哲学，很多动态类型语言的问题可以在很大程度上被避免。
[7] 目前火热的 micro-service 架构某种程度上和 Erlang 的哲学类似。
[8] 无论如何，给程序加上类型标注都是一项好的实践。
[9] 这方面的性能大概是 C 语言的 1/7 左右。
[10] 主要作用是完成浏览器的 webRTC 媒体流和 IMS 网络媒体流的之间的互通，需要大量转码和控制
[11] 这个是我自己的对比数字，我曾经用 C++开发过类似的系统。

目前市面上关于Erlang的书籍也有不少，这些书几乎都把重点放在基本的Erlang语法和设计方法上。基本上没有书籍专门讲解如何解决Erlang生产系统中出现的疑难问题（比如，过载、内存泄漏和碎片、CPU过度占用等）[12]。本书则可以说是填补了这方面的空白。

作者Fred Hebert在构建和运维大型Erlang系统方面经验丰富，而且擅长写作[13]。在本书中，他不仅把自己多年实战中积累的设计、问题诊断、调试以及解决的经验分享给大家，而且也对Erlang的内存和调度器工作原理及其强大的运行时监控和自省能力做了深入的分析和介绍。更难能可贵的是，他还把自己解决问题的方法工具化，便于大家使用。

在阅读本书之后，读者可以对 Erlang 的设计哲学和虚拟机工作机制有更深层次的理解。掌握了书中的知识和工具，虽然不能保证在运维 Erlang 生产系统时高枕无忧，但是至少在面对困难时知道从何处下手、如何收集数据、如何定位问题，为最后真正解决难题打下坚实的基础。

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

邓辉
2014.11 于上海

[12] 一方面是因为大部分系统其实没有那么庞大复杂，只要按照基本的 Erlang 原则进行设计开发，一般就不会出现什么问题；另一方面，有经验、有时间并且能写好的人确实也不多。
[13] Fred Hebert 也是 Learn you some Erlang for great good 一书的作者。

目录

介绍

运行中的软件错误

和大多数编程语言相比，Erlang 在如何处理错误方面显得非常独特。我们一般会认为，语言、编程环境以及方法论都应该尽其所能来防止错误出现。所有会导致运行时错误的东西都要被避免，如果无法避免，那么这些东西就会被置于解决方案之外，不予考虑。

程序编写完成，会被部署到情况多变的生产环境中。如果在生产环境中出现错误，就会发布新的程序版本。

Erlang则采取了与此不同的方法，Erlang认为，错误一定会发生，这些错误可能是开发人员，运维人员引起的，也可能是硬件相关的。要想根除掉程序或者系统中的所有错误，是不实际的，甚至是不可能的[14]。如果不用千方百计地阻止错误发生，而是能够在错误发生时去处理它们，那么我们就可以用这种方式应对几乎所有的程序未定义行为。

这就是“Let it Crash”[15]概念的来源：因为可以处理错误，并且（在投入到生产环境之前）清除掉系统中所有复杂的bug需要付出高昂的成本，那么程序员只需处理那些他们知道如何处理的错误，其他的错误都交给另外一个进程（supervisor）或者虚拟机来处理。

因为大多数bug都是暂态的[16]，因此在碰到错误时，简单地把进程重启到一个已知的稳态，是一个非常不错的策略。

Erlang 编程环境采取了和人体免疫系统一样的方法，在这个类比下，其他编程语言则只关心“卫生”情况，以防止细菌侵入人体。对我来讲，这两种方式都非常重要。几乎所有的编程环境都提供了卫生保证机制（程度可能不同）。基本上没有其他任何编程环境提供了免疫系统：可以在运行时处理错误，并视此为一种生存能力。

当刚刚出现问题时，系统并不会立即垮掉，因此 Erlang/OTP 也可以使我们成为一名医生。我们可以在生产环境中直接进入到系统内部，在系统运行时仔细地观察内部的所有情况，甚至以交互式的方式修复系统。继续沿用前面的类比，在 Erlang 中，我们可以施行大量的测试来进行诊断，可以实施各种程度的外科手术（甚至是高侵入性的），而此时，病人却无需坐在我们边上，也无需中断自己的日常生活。

这本书意在成为一本 Erlang 战地医生的轻量级指南。首先，它是一本用来理解错误来源的提示和技巧集，其次，它也是一本字典，其中包含了一些代码片段和实践，以帮助开发人员调

---

[14] 生命关键的系统不在此范畴内。

[15] Erlanger 目前更喜欢“let it fail”的说法，因为听起来温和一点。

[16] 根据 Jim Gray 的论文 Why Do Computer Stop and What Can Be Done About It？132 个 bug 中，131 个都是暂态的（这些 bug 是非确定性的，在想定位时，找不到他们，重启一下就都好了）。

试用 Erlang 构建的生产系统。

目标读者

本书不适用于初学者。从大多数教程、书籍以及培训课程上所学到的知识，还不能用来运维、诊断以及调试生产环境中的 Erlang 系统。在程序员学习新的语言和环境时，都需要一个摸索阶段，也就是学会在社团的帮助下，脱离指南，解决实际问题。

本书假设读者精通基本的Erlang和OTP框架。在本书中，会对一些难以理解Erlang/OTP特性进行解释，另外，希望不熟悉Erlang/OTP基本知识的读者自行学习必要的内容[17]。

本书不要求读者知道如何调试Erlang软件，如何理解已有的代码库，如何诊断问题，以及知道一些如何在生产环境中部署Erlang的最佳实践[18]。

本书的阅读方法

本书分为两个部分。

Part 1 关注于如何编写 Application。内容包括：如何理解代码库（第 1 章），编写开源 Erlang 软件的一般性建议（第 2 章），以及在系统设计中如何为过载做计划（第 3 章）。

Part 2 关注于如何成为一名Erlang医生，重点关心已有的、正在使用的系统。其中会介绍连接到一个正在运行的node（第 4 章），以及如何获得基本的可用运行时度量数据（第 5 章）。同时，还会解释如何通过crash dump文件对系统的死因进行剖析（第 6 章），如何识别并修复内存泄漏（第 7 章），以及如何发现失踪的CPU使用（第 8 章）。最后一章会介绍如何在生产环境中使用recon[19]跟踪Erlang函数调用情况，这样就可以在系统崩溃前，了解到问题所在（第 9 章）。

每章后面都会有一些可选的练习，或是问题，或是动手实验（如果觉得自己对讲述的内容都理解了，或者想更进一步时，可以做做）。

[17] 如果需要免费资源，我强烈推荐 Learn You Some Erlang 或者官方的 Erlang 文档。
[18] 在 screen 或者 tmux 会话中运行 Erlang 并不是一种部署策略。
[19] http://ferd.github.io/recon/，这是一个库，便于使用，包含的都是些产品安全的函数。

PART 1 编写Application

第 1 章：了解已有的代码库

“读源代码”不是一件令人愉快的事情，但是在和 Erlang 程序员打交道时，却又不得不经常做这件事情，因为关于代码的文档要么不完全，要么已经过时了，或者干脆就没有。还有一点就是，Erlang 程序员和 Lisp 程序员有点像，他们编写库的目的只是为了解决自己的问题，并没有在其他场景中进行过测试或者实验，如果在新场景中需要扩展或者出现问题，那么扩展和修正这事只能是使用者自己来做。

因此，不管你是想继承一个代码库，还是需要去修正代码库的 bug，还是在工作中需要去理解一个代码库，都必须得深入到这个一无所知的源码库中。其实，对于几乎所有的编程语言来说，只要项目不是自己亲自设计的，都会面临这个问题。

在实际中，会碰到 3 大类的 Erlang 代码库：原始的 Erlang 代码库，OTP Application，以及 OTP release。在本章中，我们会对这 3 种类型逐一介绍，并提供一些有用的阅读建议。

1.1原始Erlang

如果碰到原始的 Erlang 代码库，那就只能靠自己了。这些库基本上不会遵循任何规范标准，只能用笨办法去理解代码了。

这意味着，希望有一个 README.md 文件或者类似的东西能给出应用的入口点，这样可以以此为理解的起点，或者希望能够得到库作者的联系信息，方便去问些问题。

幸运的是，在实际中很少会碰到这类原始的Erlang代码库，这些库通常也是些新手项目，或者虽然是个很不错的项目，是由新手曾经编写的，但是现在需要彻底地重写。不过，由于像 rebar[20]这样的工具的出现，大家几乎都使用OTP application。

1.2OTP Application

理解 OTP Application 通常就容易多了。它们都会具有同样的目录结构，如下：

[20] https://github.com/rebar/rebar/，一个构建工具，第 2 章会对它进行简单介绍。

```
doc/
ebin/
src/
test/
LICENSE.txt
README.md
rebar.config
```

虽然不同 Application 间会有略微的差异，但是大体的结构是相同的。

OTP Application需要包含一个app文件，要么是ebin/<AppName>.app，要么是更常见的src/<AppName>.app.src[21]。app文件有两种不同的形式：

```
{application, useragent, [
  {description, "Identify browsers & OSes from useragent strings"},
  {vsn, "0.1.2"},
  {registered, []},
  {applications, [kernel, stdlib]},
  {modules, [useragent]}
]}.
```

和：

```
{application, dispcount, [
  {description, "A dispatching library for resources and task "
                "limiting based on shared counters"},
  {vsn, "1.0.0"},
  {applications, [kernel, stdlib]},
  {registered, []},
  {mod, {dispcount, []}},
  {modules, [dispcount, dispcount_serv, dispcount_sup,
             dispcount_supersup, dispcount_watcher, watchers_sup]}
]}.
```

我们称第一种形式为 library application，称第二种形式为标准（regular）application。

1.2.1 Library Application

Library Application 通常由一些名为 *appname*_something 的模块和一个名为 appname 的模块组成。这个名为 *appname* 的模块一般是该 library 的接口模块，处于核心位置，通过它可以方便地使用 library 所提供的绝大部分功能。

通过查看该模块的源代码，可以毫不费力地知道它的工作原理：如果模块依附于某个特定的 behaviour（gen_server，gen_fsm，等），那么很可能需要在自己的 supervisor 下启动一个进程，并按照这种方式来调用它。如果没有任何 behaviour，那么这个库很可能是一个纯函数的，无状态的库。此时，通过查看这个模块的导出函数，就可以快速知道它的意图。

[21] 构建系统会在 ebin 目录中产生最终的文件。请注意，在这种情况下，src/<AppName>.app.src 文件中一般不会指明模块名，构建系统会自动填上。

1.2.2 标准Application

在标准 Application 中，有两个可以作为入口点的模块：

1、 *appname*

2、 *appname*_app

第一个文件的用途和在 library application 中的类似（作为入口点），第二个文件会实现 application behaviour，并作为整个 application 进程层级结构的根。在有些情况下，第一个文件会同时充当这两种角色。

如果只是想把某个 application 作为依赖加入到自己的应用中，那么可以从 *appname* 中寻找详细的信息。如果需要维护或者是修复某个 application，那么就需要查看 *appname*_app 文件。

Application会启动一个顶层的supervisor，并返回它的pid。这个顶层的supervisor中包含有所有子进程的规格说明，这些子进程都由该supervisor启动[22]。

进程在层级树中的位置越高，对于 application 的生存来说就可能越重要。还可以从进程的启动顺序上对其重要性进行评估（supervisor 树中的子进程都按照顺序、以深度优先的方式启动）。Supervisor 树上启动较晚的进程很可能会依赖于启动较早的进程。

此外，application 中相互依赖的 worker 进程（比如，socket 通信中缓冲并中继数据的进程和负责理解协议的有限状态机进程）很可能会被组织在同一个 supervisor 下，当出现问题时会一起失败。这是一个有意的选择，因为重启两个进程，使 application 从一个干净的状态开始，要比试图去修复某个状态丢失或者遭到破坏的进程容易得多。

Supervisor 的重启策略反应了该 supervisor 中进程间的关系：

- one_for_one和simple_one_for_one策略适用于互相没有直接依赖关系的进程，不过它们的集体失败还是会导致整个application的shutdown[23]。
- rest_for_one 策略适用于具有线性依赖关系的进程。
- one_for_all 策略适用于完全相互依赖的进程。

通过这样的结构，可以非常容易地以自顶向下探索 supervisor 子树的方式浏览 OTP application。

对于每个被监控的 worker 进程，它所实现的 behaviour 提供了一个有关其用途的很好的线索：

- gen_server 会掌握一些资源，并且遵循 client-server 模式（再一般化一点，就是 request/response 模式）
- gen_fsm 会处理事件或者输入序列，并据此作出反应，就像一个有限状态机。一般

[22] 有些时候，supervisor 中并不会指明任何子进程：这些子进程要么是由某些 API 函数或者在 application 的启动过程中动态创建出来的，要么该 supervisor 的存在只是为了加载一些 OTP 环境变量（存在 app 文件的 tuple 中）。

[23] 在 rest_for_one 更合适的情况下，还是有些开发者会使用 one_for_one supervisor。他们需要严格的顺序来保证原始启动的正确性，但是在重启或者高层进程死亡时却没能保证这个顺序。

会用来实现协议

- gen_event 会作为回调的事件聚合器，或者用来处理某种类型的通知

所有这些模块都包含同样的结构：输出面向用户的接口函数，输出作为回调模块的函数，以及私有函数，一般也都是这个顺序。

基于监控关系和 behaviour 的典型角色，通过查看被其他模块使用的接口以及所实现的 behaviour，能够揭示出想要了解的代码的许多信息。

1.2.3 依赖

所有的application都有依赖[24]，被依赖的application也有自己的依赖。OTP application之间往往没有任何共享状态，因此只要开发者遵循正确的方式，仅仅看看app文件就可以知道代码之间的依赖关系。图 1.1 展示了一副基于app文件的图，有助于我们理解OTP application的结构。

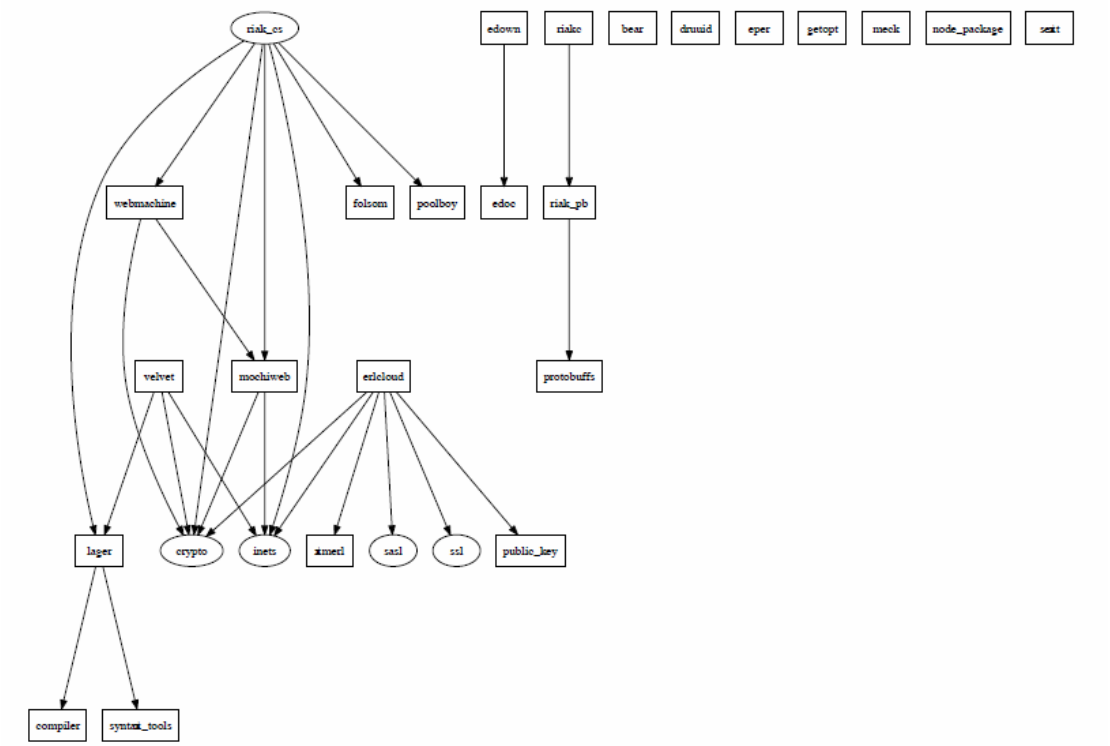


图 1.1： Basho 的开源云库 riak_cs 的依赖图。图中忽略了对于 kernel 和 stdlib 等公共应用的依赖。椭圆表示标准 application，矩形表示 library application。

基于这样的层级图，再辅以关于每个application的简要说明，就能够绘制出一幅关于所有想要找的东西位置的大致的、一般性的地图。recon库中有个script目录，调用escript script/app_deps.erl[25]，可以产生出一幅类似的图。使用observer[26]应用也可以产生这样的图，

[24] 至少需要依赖 kernel 和 stdlib application。
[25] 这个脚本依赖于 graphviz。
[26] http://www.erlang.org/doc/apps/observer/observer_ug.html

不过只能针对单一的supervisor树。基于所有这些信息，可以容易地了解到代码库中的职责分配情况。

1.3OTP release

OTP release 并不比在实际中碰到的 OTP application 难理解多少。一个 release 就是把一组可以产品化的 OTP application 打包在一起，这样在启动和关闭时就不需要手动调用任何 app 的 application:start/2 函数。当然，对 release 来说，还有一些其他的内容，不过，一般来讲，理解单个 OTP application 所用的方法，也同样适用于 release。

通常，release中会有一个类似于配置文件的文件供systools和reltool使用，其中会声明所有属于该release的application，还会有一些[27]关于如何打包的选项。要想理解它们，我建议去阅读相关的文档。如果你在研究的工程使用了relx[28]，那可真够幸运的，因为这是个更容易理解的工具（正式发布于 2014 年初）。

1.4练习

复习题

1、 如何确定代码库是一个 application，还是一个 release？
2、 标准 application 和 library application 之间有什么区别？
3、 one_for_all 监控策略下的进程有什么特点？
4、 在什么情况下会选择使用 gen_fsm 而不是 gen_server？

动手题

从https://github.com/ferd/recon_demo把代码下载下来。本书中所有练习都需要使用这份代码。目前，你应该还不熟悉这份代码，正好可以使用本章介绍的提示和技巧去理解它。

1、 这个 application 是个库呢，还是个独立系统？
2、 它有哪些功能？
3、 它有依赖吗？都有哪些依赖？
4、 App 的 README 中说它是非确定性的。你能证明这一点吗？如何证明？
5、 你能把其中 application 之间的依赖链表达出来吗？请产生一幅图示。
6、 除了 README 中描述的那些进程，你能向主 application 中添加更多的进程吗？

[27] 很多
[28] https://github.com/erlware/relx/wiki

第 2 章：构建开源的Erlang软件

大部分 Erlang 书籍都会介绍如何构建 Erlang/OTP application，不过很少会深度介绍如何和 Erlang 的开源社团进行集成。有些书干脆故意避谈这方面的内容。在本章中，我们将对此进行一个快速的介绍。

绝大多数开源代码都是 OTP application。事实上，大家在需要构建一个 OTP release 时，都会把它做成一个包罗万象的 OTP application。

如果你要编写的是一块独立的代码，供其他人构建产品时使用，那么就可以把它做成OTP application。如果你要构建的是一款独立的产品，并且希望用户按照原样进行部署（或者进行少许配置），那么应该把它做成OTP release[29]。

rebar 和 erlang.mk 是主要的构建工具。前者是一个可移植的 Erlang 脚本，其中包装了大量标准的功能，还有一些自己特定的功能，后者是一个非常复杂的 makefile，功能稍微弱一点，但是在编译时速度更快。在本章中，我会重点介绍使用 rebar 来进行构建，因为它是事实上的标准，非常成熟，并且 rebar 也支持用 erlang.mk 构建的 application 作为依赖。

2.1 项目结构

OTP application 和 OTP release 的结构是不同的。OTP application 可以只有一个顶层 supervisor（如果有的话），其下可能有一串依赖者。OTP release 则通常由许多 OTP application 组成，它们之间可以有，也可以没有依赖。这样就导致两大类 application 布局方法。

2.1.1 OTP application

对于 OTP application，其结构和我们在 1.2 节中介绍的基本一样：

```
doc/
deps/
ebin/
src/
test/
LICENSE.txt
README.md
rebar.config
```

其中多了一个deps/目录，这个目录非常有用，rebar[30]会在需要时自动生成该目录。由于Erlang

[29] 如何构建 OTP application 或者 release 的细节，可以查阅手边的 Erlang 介绍性书籍。

[30] 许多人会把 rebar 也打包到他们的应用中。这种做法最初是为了帮助那些从来没有使用过 rebar 的人以一种自举的方式使用库和工程。是把 rebar 全局地安装到系统中，还是保持一个局部拷贝，从而可以用特定版本来构建系统，都是可以的。

并没有一个正式的包管理方法。因此，大家都使用rebar，rebar会以单个工程为单位，局部地提取依赖。这种做法很好，可以避免很多冲突，不过这意味着每个工程都得去下载自己的依赖包。

可以在 rebar.config 文件中增加一些配置行来实现依赖下载：

```
{deps,
 [{application_name, "1.0.*",
   {git, "git://github.com/user/myapp.git", {branch,"master"}}},
  {application_name, "2.0.1",
   {git, "git://github.com/user/hisapp.git", {tag,"2.0.1"}}},
  {application_name, "",
   {git, "https://bitbucket.org/user/herapp.git",  "7cd0aef4cd65"}},
  {application_name, "my regex",
   {hg, "https://bitbucket.org/user/theirapp.hg" {branch, "stable"}}}]}.
```

这些application会以递归的方式从git（或者hg，svn）中直接获取。接着会对它们进行编译，可以在配置文件中用{erl_opts，List}选项来指定编译选项[31]。

日常的 OTP application 开发工作可以基于这些目录进行。想编译时，可以调用 rebar get-deps compile，这个命令会下载所有需要的依赖，然后把它们和你的 app 一起进行构建。

在发布自己 application 时，不要带上依赖。其他开发者的 application 很可能会依赖于相同的 application，因此没必要重复带上它们。构建系统（比如：rebar）应该会找到重复的项，去掉重复后再去获取所有需要的东西。

2.1.2 OTP release

release的目录结构应该有些不同[32]。release是application的集合，它的结构会反映出这一点。

在 release 中并没有一个顶层的 app，application 嵌套在下一层，并被分为两个类别：apps 和 deps。apps 目录中包含了自己 application 的源代码（内部业务代码），deps 目录中包含了独立管理的依赖 application。

```
apps/
doc/
deps/
LICENSE.txt
README.md
rebar.config
```

这种结构非常适合于产生release。之前我们介绍过像Systool和RelTool之类的工具[33]，它们给

[31] 调用 rebar help compile，可以获取更多详细信息。

[32] 说*应该*是因为不少 Erlang 开发者把他们的最终系统分成唯一的一个顶层 application（在 src 目录），以及一些依赖者（在 deps 目录），这不是最好的系统分发方法，并且和 OTP 所假定的目录结构有冲突。这样做往往是为了从产品服务器获取源代码，然后运行一些定制化的命令启动 application。

用户提供了很强的功能。relx[34]是一个比较新的更易用的工具。

针对上面目录结构的 relx 配置文件看起来像这样：

```
{paths, ["apps", "deps"]}.
{include_erts, false}. % will use currently installed Erlang
{default_release, demo, "1.0.0"}.

{release, {demo, "1.0.0"},
    [members,
     feedstore,
     ...
     recon]}.
```

调用./relx（如果可执行文件在当前目录的话）会构建一个 release，并把结果放在_rel/目录中。如果喜欢 rebar 的话，可以在 rebar.config 中使用一个 hook，这样就可以在项目编译中构建出 release:

```
{post_hooks,[{compile, "./relx"}]}.
```

这样，每次调用 rebar compile 时，都会生成 release。

2.2 supervisor和 start_link的语义

在复杂的产品系统中，几乎所有的故障和错误都是暂态的，对某个操作进行重试是一种不错地解决问题方法——Jim Gray的论文[35]中指出，使用这种方法处理暂态故障，系统的平均故障间隔时间（MTBF）提升了 4 倍。不过，supervisor所做的不仅仅是重启这么简单。

Erlang的supervisor以及其supervisor树有一个非常重要的特性：它们的启动阶段是同步的。每个OTP进程都可能会妨碍其兄弟以及堂兄弟进程启动。如果该进程死亡了，会不断重启它，直到它正常工作或者重启次数太多为止。

在这方面，大家会犯一个非常常见的错误。在supervisor重启一个崩溃的子进程前，并没有一个避让或者冷却周期。当某个基于网络的application在初始化阶段去建立一个连接，而此时远程服务不可用时，该application会在多次无果的重启后失败。接着系统会停止。于是，许多Erlang开发者都希望supervisor能有一个冷却周期。我坚决反对这种看法，原因很简单：履行承诺。

2.2.1 履行承诺

重启进程是要把它带回到稳定、已知的状态。此后，重试操作才有意义。如果初始化不是稳

[33] http://learnyousomeerlang.com/release-is-the-word
[34] https://github.com/erlware/relx/wiki
[35] http://mononcqc.tumblr.com/post/35165909365/why-do-computers-stop

定的，那么 supervisor 基本上就没啥价值。在任何情况下，初始化过的进程都应该是稳定的。只有这样，随后启动的兄弟、堂兄弟进程才能够确信，在它们之前启动的部分是健康的。

如果不能保证这种稳定状态，或者以异步的方式启动整个系统，那么这个结构能带来的好处也就只是和在循环中的放入 try…catch 差不多。

被监控的进程在其初始化阶段履行承诺，而不是尽力就行了。这意味着，在为数据库或者某个服务编写客户程序时，不能把连接建立作为初始化过程的一部分，除非能够保证一定成功。

比如，如果数据库和客户程序在一台机器上，并且一定先于Erlang系统启动，那么可以在初始化中要求建立连接。这样，重启操作也就没啥问题。如果有某些无法理解或者不期望的东西会破坏这些承诺，那么节点就会最终崩溃，这种情况正是所希望的：系统启动的前置条件没得到满足。这是一种系统范围内的断言失败。

不过，如果数据库在一个远程主机上，就应该能预见到连接会失败。这是分布式系统中无法避免的故障情况[36]。此时，客户进程唯一能够做出的承诺就是，它可以处理请求，但是不会和数据库通信。比如，它可以在网络分裂（net split）时返回{error, not_connected}。

接下来，可以在不影响系统稳定性的情况下，采用合适的冷却或者避让策略去重连数据库。作为优化，可以放在初始化阶段尝试一下，不过，在发生连接断开时，进程应该能在初始化后再进行重连。

如果事前知道在调用外部服务时会出现问题，那么就不要把它当成系统的承诺。我们面对的是现实世界的问题，外部服务的故障总是会出现的。

2.2.2 副作用

如果调用客户程序的库或者进程必须在有数据库的情况下才能工作，那么它们就会失败。这完全是另外一个无关问题，这个问题和业务规则、客户程序的约束有关，不过这个问题是可以解决的。比如，假设客户程序要访问某个存储有运维度量数据的服务——调用该客户程序的代码可以完全忽略掉调用错误，不会对整个系统造成任何负面影响。

初始化和监控方法的不同之处在于，对错误的容忍程度的决策是由客户程序的调用者做出的，而不是客户程序自己。在设计容错系统时，这个区分非常重要。是的，supervisor 只是在做重启，但是它们要确保重启到一个稳定的已知状态。

2.2.3 例子：不承诺连接建立的初始化过程

下面的代码试图把建立好的连接作为进程状态的一部分：

[36] 或者是因为延时太大，和故障之间无法辨别。

```
init(Args) ->
    Opts = parse_args(Args),
    {ok, Port} = connect(Opts),
    {ok, #state{sock=Port, opts=Opts}}.

[...]

handle_info(reconnect, S = #state{sock=undefined, opts=Opts}) ->
    %% try reconnecting in a loop
    case connect(Opts) of
        {ok, New} -> {noreply, S#state{sock=New}};
         _ -> self() ! reconnect, {noreply, S}
    end;
```

下面的代码则与此相反：

```
init(Args) ->
    Opts = parse_args(Args),
    %% you could try connecting here anyway, for a best
    %% effort thing, but be ready to not have a connection.
    self() ! reconnect,
    {ok, #state{sock=undefined, opts=Opts}}.

[...]

handle_info(reconnect, S = #state{sock=undefined, opts=Opts}) ->
    %% try reconnecting in a loop
    case connect(Opts) of
        {ok, New} -> {noreply, S#state{sock=New}};
        _ -> self() ! reconnect, {noreply, S}
    end;
```

现在，初始化过程中的承诺就弱了不少：连接一定能够建立的承诺没有了，变成了承诺连接管理器一定是可用的。

2.2.4 小结

在我参与过的产品系统中，会混合使用这两种方式。

类似配置文件，访问文件系统（为了记日志），可以信赖的本地资源（打开日志用的 UDP 端口），从磁盘或者网络恢复到一个稳定的状态之类的事情，我会把它们当成是 supervisor 工作所必需的东西，不管花费多长时间，都会同步执行（有些应用在极端情况下启动会耗费 10 分钟左右时间，不过，这是可以接受的，因为为了能够正常工作，确实需要同步上 G 的数据以建立一个基础状态）。

另一方面，如果是依赖于非本地数据库，或者是外部服务，我就会在supervisor树中做一个快速的部分启动，因为如果一个错误会经常性地出现在日常的操作中，那么是现在出现还是以后出现是没啥区别的。必须得以同样的方法处理它，对于系统中的这部分代码，不做出严格的承诺往往是更好的解决方案。

2.2.5 application策略

一连串的错误并不一定会导致节点的退出。一旦系统被分成不同的 OTP application，我们就能够从中挑选出那些对节点来说至关重要的 application。每个 OTP application 都可以以 3 种方式启动：temporary、transient、permanent，启动方式可以通过在 application:start(Name, Type) 中手动指定，也可以在 release 的配置文件中指定：

- permanent：如果 app 终止了，那么整个系统都会停止工作，通过调用 application:stop/1 手动终止 app 的情况除外。
- transient：如果 app 以 normal 的原因终止，没有影响。任何其他终止原因都会导致整个系统关闭。
- temporary：application 可以以任何原因终止。只会产生报告，没有其他任何影响。

应用也可以作为included application启动，它会启动在你的OTP supervisor之下，服从supervisor的重启策略。

2.3 练习

复习题

1、 Erlang supervisor 树是深度优先启动的，还是广度优先启动的？是同步的，还是异步的？
2、 Application 有哪三种启动策略？分别是怎么做的？
3、 app 和 release 目录结构的主要区别是什么？
4、 何时需要 release？
5、 对于可以放在进程初始化函数中的那些状态，和不应该放在进程初始化函数中的那些状态，分别列举两个例子。

动手题

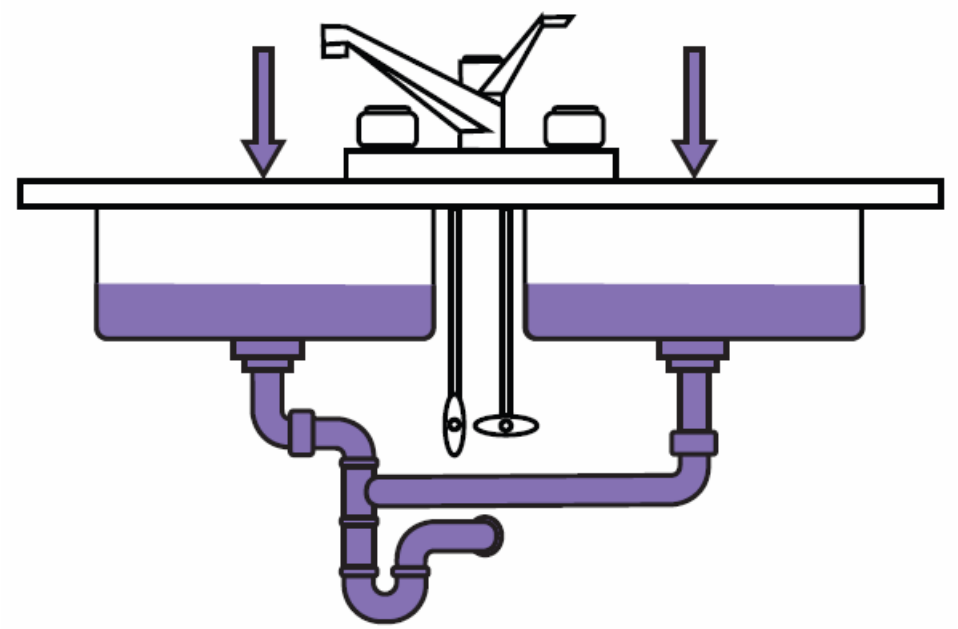
使用https://github.com/ferd/recon_demo中的代码：

1、 从 release 中把主 application 提取出来，把它变成独立的，从而可以包含在其他项目中。
2、 把这个 application 放在某个地方（Github，Bitbucket，本地服务器），构建一个 release，把该 application 作为依赖。
3、 Application 的主工作者进程（council_member）会启动一个服务器，并在其 init/1 函数中去连接它。你能把这个连接逻辑放到 init 函数外面吗？针对这个例子，这样做有什么好处吗？

第 3 章：为过载做计划

到目前为止，我在实际工作中所碰到最常见的错误，基本上都是节点内存耗尽。而且通常都和过长的消息队列有关[37]。解决这类问题的方法有很多，不过只有在深入、全面的理解系统后，才能做出正确的选择。

基本上，我从事的所有项目都可以简化类比成一个非常大的浴室水槽。用户请求和数据从龙头流入。Erlang 系统则是水槽和管道，可以把水流出的地方（数据库，外部 API 或者服务，等等）看作是下水道系统。



当 Erlang 节点由于队列溢出而死亡时，找到原因所在是至关重要的。是流入槽中的水太多了吗？是下水道堵塞了吗？还是把管道设计得太细了？

要找到膨大的队列并不是件难事，这个信息可以从 crash dump 中获得。知道队列膨大的原因则要困难得多。根据进程的角色，或者做些运行时的检查，就可以找出一些可能的原因：消息泛滥，进程阻塞无法快速处理消息，等等。

决定如何修复是最困难的部分。由于对水的滥用导致水槽积水时，我们可以换一个大一些的水槽（程序中崩溃的部分，处于边缘）。接着，发现水槽出水口太小，优化它。再接着发现管道太细，优化它。负载一直被推向系统的下游，直到下水道无法承受。此时，我们可能会试着增加水槽或者增加浴室来缓解这个全局性的输入问题[38]。

不过，事情总会发展到无法在浴室层面解决的地步。发出的日志过多，要保持一致性的数据库成为了瓶颈，或者只是因为公司没有解决问题所需的知识或者人力。

---

[37] 第 6 章会介绍如何定位出有问题的消息队列（6.2 小节）

[38] 译者注：这里的增加水槽或者增加浴室只是增加缓冲区的大小，解决暂时过载的问题，不能根本解决持续过载的问题。

只有到了此时，我们才找到了系统的真正瓶颈所在，前面所做的优化虽然都不错（并且成本也不低），不过可能没啥用。

我们需要更聪明一些，退后一步，在上游解决问题。可以对进入系统的信息做些处理，让它更轻量一些（可以通过压缩，用更好的算法和数据表示，缓存等等）。

即使这样，还是会出现系统负载过大的情况，此时我们不得不在限制系统输入，丢弃输入，或者接受系统在崩溃前会持续降低服务质量之间做出艰难的选择。这些机制隶属于两种大的策略：反压（back-pressure）和减载（load-shedding）。

我们会在本章中介绍这两种策略，也顺便介绍一些会“撑大”Erlang 系统的常见事件。

3.1 常见的过载源

在导致 Erlang 系统队列膨大、过载的原因中，有一些比较常见，在系统的开发过程中迟早会遇到它们。出现这些问题时，通常意味着系统变大了，需要在伸缩性方面做些工作；或者是存在某种没有预见到的故障，其产生的影响让问题复杂化。

3.1.1 error_logger爆炸

具有讽刺意味的是，负责错误日志的进程竟然是最为脆弱的之一。在Erlang的缺省安装中，error_logger[39]负责记录日志（写入磁盘或者发送到网络上），它的速度要比错误产生的速度慢得多。

尤其是在记录用户产生的日志消息（不是错误）或者大量进程崩溃时，更是如此。对于前者，是因为 error_logger 本来就不适用于记录高度连续的消息。它只适用于真正的异常情况，并不期望过多的消息量。对于后者，是因为崩溃进程的完整状态（包括其消息邮箱）都会拷贝出来进行日志记录。这种消息无需太多，就会耗费大量内存，即使不能立即造成节点内存耗尽（OOM），也足以减慢 logger 的处理速度，使得后续消息不断累积，导致最终的内存耗尽。

到目前为止，最好的解决方案是使用另外一个日志库：lager。

虽然 lager 不能解决所有的问题，但是它会删节掉巨大的日志消息，会在超过门限时选择性的丢弃掉 OTP 产生的错误消息，并且会在用户提交消息时自动地在同步、异步模式间切换，以达到自我调控的目的。

对于有些极端情况，lager 也无能为力，比如：用户提交的消息非常巨大，并且都来自一次性进程。不过，这种情况非常少见，而且此时程序员往往具有更多的控制权。

[39] 参见：http://www.erlang.org/doc/man/error_logger.html

3.1.1 锁和阻塞操作

如果某个进程需要持续地接收新任务，那么其在执行耗时过长的锁或者阻塞操作时，就会出现问题。

最为常见的例子之一就是：某个进程使用了 TCP socket，阻塞在了接收新的连接或者等待消息上面。在执行此类阻塞操作时，消息会不受限制地堆积在消息队列中。

一个更为糟糕的例子是我曾经为lhttpc库的某个分支写的http连接池管理器。在大多数测试用例下，它都工作正常，我们甚至把连接的超时时间设置为 10ms，以确保不会耗费太多的时间[40]。正常工作了几周后，该HTTP连接池导致了一次服务中断，原因是有个远程服务器down机了。

这次恶化背后的原因是：当远程服务器 down 机后，突然之间，所有的连接操作都至少要耗费 10ms 的时间，也就是连接超时时间。每秒大约有 9,000 个消息发给中心进程，通常每个消息的处理时间在 5ms 之内，这个超时的影响等同于每秒 18,000 个消息，情况变得失控。

我们的解决方案是，把进行 HTTP 连接的任务放到调用进程中，并加以限制，感觉仍像是管理器自己在做这件事一样。现在，阻塞操作分布到库的所有使用者之中，管理器要做的工作更少了，可以接受更多的请求。

当程序中有任何一个点，最终成为接收消息的中心点时，耗时的任务要尽可能从中移出。对于可以预见的过载[41]，增加更多的进程（它们或者处理阻塞的操作，或者在主进程阻塞时充当缓冲）往往是个不错的方法。

如果某些活动并没有内在的并发要求，那么这种方法会增加进程管理方面的复杂性，所以在这样做之前要确定自己确实有这个需要。

另外一种做法是，把阻塞任务变成异步的。如果场景适用，服务器可以启动一个进程来执行耗时（等待资源）的任务，赋予该进程一个唯一的令牌，把令牌和原始请求者保存起来。当资源可用时，这个进程会把令牌附带在消息中发送给服务器。服务器最终会得到这个消息，通过令牌匹配原始请求者，并发送回应，期间不会被其他请求所阻塞[42]。

这种做法要比使用多个进程的方法更晦涩一些，也很容易造成回调地狱（callback hell），但是使用的资源要更少些。

---

[40] 10ms 非常短，不过对于放置在一起的用于实时竞标的服务器来说是合适的。

[41] 你确定某些事在生产环境中一定会过载。

[42] redo application 中的 redo_block 模块是这种做法的一个例子。该模块（文档正在编写中）把一个管道式连接变成阻塞的，不过在转换时一直保持管道指向调用者——这样，当超时时，调用者就能知道只有一个调用失败了，而非所用中间调用，期间服务器不会停止接受请求。

3.1.3 不期望的消息

如果使用 OTP application，很少会出现未知消息的情况。因为 OTP 的 behaviour 会要求你在 handle_info/2 中处理一些东西，因此不期望的消息不会大量堆积。

不过，所有OTP兼容的系统中都会有些没有实现behaviour的进程存在，这些非behaviour进程中可能会存在不当的消息处理。如果足够幸运，使用监控工具[43]就可以发现内存的持续增长，通过对过大消息队列[44]进行检查，可以帮助我们发现有问题的进程。接下来，只要让进程按照要求处理消息，就可以解决这个问题。

3.2 限制输入

限制输入是 Erlang 系统中控制消息队列增长最简单的方法。说它最为简单是因为，它就是让使用者变慢（应用反压方法），不需要进一步的优化就可以立即解决问题。不过，它会让使用者的体验变得很差。

最常见的限制数据输入的方法是，如果进程的消息队列增长会失控，那么就以同步的方式去调用它。只有在收到回应后，才能进行下一个请求，这样，就可以基本保证问题的源头能够慢下来。

这种方法的难点在于，导致队列增长的瓶颈往往不在系统的边缘位置，而是在系统内部，只有在把前面的部分都优化过了才能找到它。这种瓶颈通常都是数据库操作，磁盘操作或者是一些网络服务。

这就意味着，如果在系统内部引入同步行为，那么就需要逐层处理反压问题，直到系统的边缘才能通知到使用者，让它们“慢下来”。理解这种模式的开发者通常会在系统的入口处放置可以对单个使用者进行限制的API [45]。这种做法是有效的，尤其是它可以保证系统的基本服务质量（QOS），可以按需更公平（或者不公平）地分配资源。

3.2.1 超时应该设置多长

在以同步调用的方式通过反压处理过载时，有个难点在于必须得确定出典型的操作要花费的时间，或者更准确一点，系统的超时要设置多长。

这个问题的本质在于，第一个定时器需要在系统的边缘启动，但是关键的操作却在系统内部发生。这意味着，位于系统边缘的定时器的等待时间要比系统内部的长一些，除非允许即使内部的操作成功了，位于边缘的操作也可以报告超时的情况出现。

---

[43] 见 5.1 小节

[44] 见 5.2.1 子小节

[45] 在把所有请求都同等变慢和限制速率之间需要做个权衡，这两种方法都是有效的。针对单个使用者限制速率意味者当有更多新使用者冲击系统时，还是必须得增加处理能力或者降低所有使用者的上限，而不做区分的同步系统可以更容易地适应任意的负载情况，不过会缺失公平性。

对这个问题有个很容易想到的解决方法，就是使用无限时定时器。Pat Helland[46]对此有个有趣的回应：

> 有些应用开发者坚持不设置超时，并声称永久等待没啥问题。我一般会建议他们把超时设置成 30 年。结果，有回应说我应该明智一点，不要做傻事。为啥等 30 年就是愚蠢，而无限等待就是明智的？我还真就看到过这么一个消息应用，它真的在无限等待……

归根到底，这得具体问题具体分析。在许多情况下，采用一种不同的流控机制也许更为实际一点[47]。

3.2.2 请求允许

一种次简单点的反压方法是，找出那些我们必须阻塞在其上的资源，也就是那些无法变得更快，而又对业务和使用者来说至关重要的东西。把这些资源保护在一个模块或者过程背后，调用者必须得到许可才能发起请求并使用它们。可供选用的条件变量有很多：内存、CPU、总负载、调用的限制次数、并发、响应时间、或者前面几个变量的组合，等等。

SafetyValve[48] application是一个系统级的框架，在觉得反压方法适用时，可以使用它。

对于和业务或者系统故障有关的那些更加特定的情况，有许多断路器application可用。比如：breaky[49]，fuse[50]，或者Klarna的circuit_breaker[51]。

除此之外，还可以使用进程，ETS 或者其他可用的工具编写一些特定的解决方案。重点在于，系统（子系统）的边缘部分会阻塞请求并获取许可，但是代码中的关键瓶颈部分则决定是否赋予许可。

这种做法的优点在于，定时器以及每一层的同步抽象这些难搞的工作都可以被避免掉。只需在瓶颈处以及某个边缘或者控制点处设防，中间的代码都可以以最易读的方式进行表达。

3.2.3 情况报告

反压策略的难点在于如何报告。在通过同步调用隐式地实施反压时，知道它在过载时起作用的唯一方法就是发现系统的处理速度和可用性都在降低。遗憾的是，这也可能是因为硬件不好、网络不好、无关的过载或者可能是使用者本身就慢造成的。

试图通过度量响应性来判断系统是否采用了反压的策略，就如同通过观察发热来诊断病人得了什么病一样。它只能告诉你有问题出现了，但是却说不出是什么问题。

---

[46] 幂等性不是一种健康状况（Idempotence is Not a Medical Condition），acm queue,2012.4.14。

[47] 在 Erlang 中，使用值 inifity 会避免创建定时器，节省了一些资源。如果你这样做了，那么请记得至少要在调用序列的某个地方放置一个合理的超时。

[48] https://github.com/jlouis/safetyvalve

[49] https://github.com/mmzeeman/breaky

[50] https://github.com/jlouis/fuse

[51] https://github.com/klarna/circuit_breaker

请求允许，作为一种机制，通常允许定义出可以显式汇报系统情况的接口：系统整体上已经过载，或者你所执行的操作已经达到了速率限制的上限，请做相应的调整。

在设计系统时，要做出一个选择。限制是针对每个账号的，还是针对整个系统的？

针对整个系统或者节点的限制一般比较容易实现，但是具有不公平的缺点。一个请求量占90%的使用者可能会造成整个平台对其他大多数使用者不可用。

针对每个账号进行限制则可以做到非常公平，可以用来实现非常复杂的控制模式，比如可以让付费用户不受通常的限制。这非常好，不过缺点是，系统的使用者越多，系统的全局有效限制就越要往上移。比如，开始有 100 个用户，每个用户每分钟可以做 100 个请求，此时系统在整体上每分钟可以处理 10000 个请求。如果保持这个处理速率，当再增加 20 个新用户时，系统就会突然变得经常崩溃。

在设计系统时所建立的安全误差范围（safe margin of error）会随着使用者的增多而逐步变小。因此，从这个视角出发去考虑业务所能做出的让步是非常重要的，因为比起所允许的请求量不断下降，用户往往可能会更愿意接受整个系统时不时的服务中断。

3.3 丢弃数据

如果在 Erlang 系统之外，没啥东西可以慢下来，并且系统也不能向上扩展（scale up），那么就只能要么选择丢弃数据，要么选择崩溃（大多数情况下，崩溃都是粗暴地丢弃途中数据）。

没有人真愿意去丢弃数据。程序员，软件工程师以及计算机科学家所受的训练是剪除掉无用的数据，所有有用的数据都不能丢失。要通过优化来达到成功，而不是丢弃。

然而，即使 Erlang 系统本身有足够快的处理能力，还是会碰到数据的进入速度超过离开速度的情况。有时，阻塞点是下游部件。

如果无法控制数据的接收速度，那么为了避免崩溃，就只能丢弃数据。

3.3.1 随机丢弃

随机丢弃是最容易的一种丢弃方法，因为其简单，也是实现起来也最为可靠的。

要点在于定义一个位于 0.0 和 1.0 之间的阈值，并随机在该范围内取值：

```
-module(drop).
-export([random/1]).

random(Rate) ->
    maybe_seed(),
    random:uniform() =< Rate.

maybe_seed() ->
    case get(random_seed) of
        undefined -> random:seed(erlang:now());
        {X,X,X} -> random:seed(erlang:now());
        _ -> ok
    end.
```

如果想让 95%的消息得以发送，可以这样达成：调用 case drop:random(0.95) of ture -> send(); false -> drop() end，如果在丢弃消息时不需要什么特殊处理的话，调用可以更简短一些，drop:random(0.95) andalso send()。

maybe_seed 函数会检查进程字典中是否已有有效的种子，如果有就使用这个更好的种子，仅在种子以前没有被定义过的情况下，才会调用 now()（这是一个需要全局锁的单调函数），这样可以避免过频地调用 now()。

这种方法中有一个“坑”需要注意：丢弃操作必须要在生产者侧进行，而不能在队列（接收者）侧进行。避免队列过载最好的方法就是不要向它发送数据。Erlang 中不会限制邮箱的大小，因此在接收进程中丢弃消息，只能使该进程处于瞎忙的状态，忙着丢弃消息并阻碍调度器处理实际有用的工作。

在生产者侧丢弃消息则可以确保负载均匀分布在所有进程之中。

我们还可以做一种有趣的优化：工作进程或者某个监控进程[52] 使用 ETS 表或者 application:set_env/3 动态地增加或者减少这个和随机数一起使用的阈值。这样，就可以做到根据负载决定丢弃消息的量，并且任何进程都可以使用application:get_env/2 来更高效地获取配置数据。

采用同样的方法，我们还可以实现针对不同的消息优先级设定不同的丢弃率，而不是都按照消费者的标准设定。

3.3.2 队列缓冲

如果想对要扔掉的消息有更多的控制，而不是随机地把它们丢弃，尤其是如果能预见到过载是突发的，而非持续时，那么队列缓冲是一种不错的选择。

即使通常的进程邮箱已经具有了队列的功能，一般来讲，还是希望能够尽可能地把所有消息从中拿出来。安全起见，队列缓冲需要两个进程：

[52] 任何一个负责用启发式的方法（比如，使用 process_info(Pid, message_queue_len)）来检查特定进程负载的进程都可以成为监控者。

- 完成工作的常规进程（比如：gen_server）
- 另外一个只用来负责缓冲的进程。来自外部的消息都要发送给它。

工作原理为，缓冲进程只需要把所用消息从其邮箱中取出，放到一个自己管理的队列数据结构[53]中。当服务进程可以处理更多的工作时，就会向缓冲进程发出请求，让缓冲进程给其发送一定数量的消息用于处理。缓冲进程从自己的队列中取出一定数量的消息，把它们转发给服务进程，然后接着去累积消息。

当队列超过一定大小[54]，如果此时再收到一个新的消息，就把最老的那个消息弹出，把这个新的消息放入，在这个过程中，最老的那些消息会被丢弃[55]。

这会把接收到的消息的数量保持在一个稳定的长度，可以很好地解决过载问题，有点类似于环形缓冲的函数式版本。

PO Box[56]库就实现了这么一个队列缓冲。

3.3.3 堆栈缓冲

如果既想要队列缓冲的控制，又有低延时的要求，那么堆栈缓冲就非常适用了。

把堆栈作为缓冲，也需要两个进程，不过不能使用队列数据结构，而要使用列表数据结构[57]。

堆栈缓冲非常适用于低延时需求的原因，和缓冲膨胀[58]中类似的问题有关。如果队列中已经缓冲了一些消息，那么队列中所有消息都会慢下来，并多了几个毫秒的等待时间。最终，这些消息都会变得很老，整个缓冲都得被丢弃掉。

如果采用堆栈，那么只会有有限数量的消息保持等待，而新一点的消息会即时、持续地发给服务器处理。

当堆栈大小超过某个特定值，或者某个元素太老而无法满足 QoS 要求时，只需把以此为界的部分从堆栈中丢弃掉，然后接着处理就行了。PO Box 中也提供了这种缓冲实现。

堆栈缓冲有个主要的缺点，消息不能按照提交的顺序处理——这对于独立的任务来说没啥问题，不过当事件必须按顺序处理时就会出现问题。

---

[53] Erlang 中的 queue 模块是一个纯函数式的队列数据结构，非常适合于实现这种缓冲。

[54]在计算队列长度时，最好是采用一个计数器，随着消息的收、发而增、减，而不是每次都去遍历队列。这会稍微多花费点内存，但是计数的计算工作分布更均匀一些，可预测性会更好一些，可以避免邮箱中消息的突然堆积。

[55] 如果认为老的消息更重要的话，可以丢弃最新的消息，缓冲最老的消息。

[56] 该库位于：https://github.com/ferd/pobox，该库已经在Heroku的大型产品代码中应用了很长时间，可以认为是成熟的。

[57] Erlang 的 list 就是堆栈。它提供的 push 和 pop 操作具有 O(1)的复杂性，非常快，在各个方面都满足要求。

[58] http://queue.acm.org/detail.cfm?id=2071893

3.3.4 时间敏感缓冲

如果想在老事件变得太老之前，对它们做些处理，那么事情就会变得更加复杂，因为每次都得深入到堆栈内部去寻找，并且从堆栈底部连续丢弃的方式也很低效。此时，桶（bucket）是一种有趣的解决方法，桶方法使用多个堆栈，每个堆栈桶包含有自己的时间片段。当请求变得太老不能满足 QoS 约束时，这个桶会被全部丢弃掉，缓冲的其他部分不受影响。

牺牲一些请求来使多数受益的做法似乎有点违反直觉——中位数很好，但是 99 百分位数糟糕——不过这种做法有个前提，那就是无论如何都得丢弃消息，尤其是当需要低延时时，这种做法就更可取了。

3.3.5 应对持续的过载

如果面对的是持续的过载，那么就需要一种新的解决方案。队列和缓冲都非常适用于那些偶发的过载（即使持续时间会长一点），当能够预见到输入速度最终会降下来，处理随后也能跟得上时，这两种方法都非常可靠。

当试图向一个进程发送太多的消息，以至于无法不过载时，就会出现问题。针对这种情形，有两种常用的解决方法：

- 让更多的进程来充当缓冲，并通过它们进行负载均衡（横向伸展）
- 使用 ETS 表实现锁和计数器（减少输入）

一般来讲，ETS 表每秒能处理的请求量要远高于进程，不过它们只能支持简单一些的操作。读一个值，原子地增加、减少一个计数器也就是最复杂的操作了。

这两种方法都需要使用 ETS 表。

一般来讲，第一种方法也可以使用常规的进程注册机制：采用 N 个进程来分担负载，给它们指定一个名字，并从中选择一个来发送消息。因为基本上知道一定会过载，所以在服从均匀分布的进程中随机挑选一个的做法非常可靠：不需要任何状态通信，工作会以大致平均的方式分摊，而且对错误非常不敏感。

不过，在实际中，我们想避免动态生成原子（atom），因此更倾向于把进程注册到 ETS 表中，并把 read_concurrency 选项设置为 true。这要多做点工作，不过更灵活一点，可以在以后更改工作者的数量。

在前面提到的lhttpc库[59]中，实现了一个类似的方法，基于域（domain）来分割、均衡负载。

对于第二种方法（使用锁和计数器），基本的结构和第一种一样（从多个选择中挑选一个，向它发送消息），不过在实际发送消息之前，必须得原子地更新一个ETS计数器[60]。所有使用

[59] 这个库中的 lhttpc_lb 模块实现了这种方法

[60] 使用 ets:update_counter/3 方法。

者共享一个已知的限制（通过它们的supervisor，或者其他配置，或者ETS值），在向进程发送每一个请求前，都需要先通过这个限制。

dispcount[61]中采用了这种方法来避免使用消息队列，保证可以低延时地回应任何没有被处理的消息，这样不必等待就可以知道请求被拒绝。接下来就由库的使用者来决定是立即放弃，还是去尝试其他的工作者。

3.3.6 丢弃的方式

这里介绍的大部分解决方案都是基于消息数量的，不过也可以尝试基于消息的大小，或者其他期望的指标（如果能预测的话）。如果使用的是队列或者堆栈缓冲，可以把统计消息数量的方式改成统计大小，或者为它们指定一个负载限制。

在实际中，我发现，在丢弃时不关心消息的特定属性的方式工作得非常好，不过每个应用都有自己独特的一些折中，不能一概而论[62]。

还有些情形，其中数据是以“发送然后忘记”的方式发出的——整个系统是某个异步管道中的一部分——那么，在请求被丢弃或者遗失时，就难以给最终用户提供这方面的反馈。此时，如果能够保留一种特定的消息类型来收集丢弃的回应消息，并告诉使用者“N 个消息被丢弃了，原因是 X”，这种折中对于使用者来说要可接受得多。Heroku 的 logplex 日志路由系统中就采用了这样做法，它会发送 L10 错误，报警给使用者系统中的某个部分无法应对目前的消息量。

最后，处理过载的方式是否可被接受，往往取决于使用系统的人。通常，稍微改变点需求要比开发一项新技术容易得多，不过有时候也是无法避免的。

3.4 练习

复习题

1、 说出 Erlang 系统中的常见过载源
2、 处理过载的两类主要策略是什么？
3、 如何让耗时的操作变得安全？
4、 在同步操作中，如何设定超时？
5、 说出超时的一种替代方法
6、 在什么情况下，会优先选择队列缓冲？

---

[61] https://github.com/ferd/dispcount

[62] 一些老的论文，比如 Butler.W.Lampson 的 Hints for Computer System Designs，会推荐丢弃消息的做法：“通过减载来控制需要，不能允许系统过载。”这篇论文还提到“如果对资源的需求超过了系统能力的 2/3，那么就不要期望系统能正常工作，除非非常了解负载的特征。”并补充道“只有对于已经被理解的非常清楚的负载，系统中的聪明做法才会生效。”

开放问题

1、 什么是真正的瓶颈？如何找到它？
2、 有个应用调用了第三方 API，响应时间严重依赖于其他服务器的健康情况。如何进行系统设计才能防止某些偶然的慢请求会阻塞针对同一个服务的其他并发请求？
3、 有个过载的延迟敏感服务，其堆栈缓冲中已经有数据堆积，那么此时来了新的请求会如何处理？老的请求呢？
4、 说明一下如何使一个采用减载机制应对过载的系统同时也支持反压机制。
5、 说明一下如何把反压机制变成减载机制。
6、 丢弃或者阻塞请求的行为对用户有何风险？如何防止消息重复和丢失？
7、 如果一开始忘记了应对过载，现在突然需要增加反压或者减载机制，那么你会希望在 API 设计时做哪些工作呢？

Part 2 诊断Application

第 4 章：连接远程节点

要和运行中的服务程序进行交互，习惯上会使用两种方式。一种是使用后台运行的 screen 或者 tmux 会话保持一个交互的 shell，使用者可以连接到它。另外一种是通过程序管理功能或者能动态加载的面面俱到的配置文件。

交互会话方式对于严格按照读取—求值—打印—循环（REPL）方式运作的软件来说通常是适用的。通过管理功能或者配置文件的方法则要求对需要执行的任务进行仔细的计划，并指望着能做对。很多系统都可以采用这种方法，不过我会略过这个方法，因为我对计划失败，出现了问题但是没有对应的处理措施的情况更感兴趣。

Erlang 采用的方式比 REPL 更像一个“交互者”（interactor）。事实上，一个常规的 Eralng 虚拟机可以不需要 REPL，一直运行字节码就可以了，也可以不需要任何 shell。不过，由于它的并发和多处理（multiprocessing）工作机制以及对于分布式的良好支持，可以让 REPL 在任意的 Erlang 进程运行。

这意味着，与一个screen会话一个shell的情况不同，同时可以有任意多个Erlang shell连接到同一个虚拟机，并和其进行交互[63]。

一般情况下，想要连接起来的两个节点上要有同样的cookie[64]，不过也有些可以不使用cookie的方法。大部分情况下，要求节点必须具有名字，并且在连接到节点前需要做些事先度量，以确保安全可行。

4.1 作业（Job）控制模式

在 Erlang shell 中按下^G 键，就可以看到作业控制模式（JCL mode）的菜单。在菜单中，有个选项能让我们连接到一个远程 shell。

[63] 更详细的介绍，请参见：http://ferd.ca/repl-a-bit-more-and-less-than-that.html
[64] 更多细节，参见：http://learnyousomeerlang.com/distribunomicon#cookies 或者 http://www.erlang.org/doc/reference_manual/distributed.html#id83619

```
(somenode@ferdmbp.local)1>
User switch command
 --> h
  c [nn]            - connect to job
  i [nn]            - interrupt job
  k [nn]            - kill job
  j                 - list all jobs
  s [shell]         - start local shell
  r [node [shell]]  - start remote shell
  q                 - quit erlang
  ? | h             - this message
 --> r 'server@ferdmbp.local'
 --> c
Eshell Vx.x.x  (abort with ^G)
(server@ferdmbp.local)1>
```

连接到远程 shell 后，所有的行编辑和作业管理都由本地 shell 完成，不过求值的工作是在远程完成的。远程求值的结果输出全部被转发给本地 shell。

要退出 shell，按^G 回到 JCL 模式。作业管理是在本地进行的，因此通过^G q 的方式退出 shell 是安全的：

```
(server@ferdmbp.local)1>
User switch command
 --> q
```

可以选择采用隐藏模式启动初始 shell（使用-hidden 参数），这样可以避免自动连接上整个集群。

4.2 Remsh

还有一个和 JCL 模式完全类似，但是调用方式不同的机制。使用这种机制，JCL 模式的所有操作步骤都可以被绕过，只需像下面这样启动 shell，对于长名字：

```
erl -name local@domain.name -remsh remote@domain.name
```

对于短名字：

```
erl -sname local@domain -remsh remote@domain
```

所有其他的 Erlang 参数（比如，-hidden 和-setcookie $COOKIE）也是有效的。底层的运作机制和使用 JCL 模式时完全一样，不过初始 shell 是远程而非本地启动的（JCL 还是本地的）。^G 仍然是最安全的退出远程 shell 的方法。

4.3 SSH Daemon

Erlang/OTP 的发布版本中自带了 SSH 实现，既可以当服务器，也可以当客户端。其中包含了一个 demo application，可以在 Erlang 中提供远程 shell。

要使用该功能，通常需要先准备好具有远程访问 SSH 权限的 key，不过如果只是为了快速的测试一下，可以这样做：

```
$ mkdir /tmp/ssh
$ ssh-keygen -t rsa -f /tmp/ssh/ssh_host_rsa_key
$ ssh-keygen -t rsa1 -f /tmp/ssh/ssh_host_key
$ ssh-keygen -t dsa -f /tmp/ssh/ssh_host_dsa_key
$ erl
1> application:ensure_all_started(ssh).
{ok,[crypto,asn1,public_key,ssh]}
2> ssh:daemon(8989, [{system_dir, "/tmp/ssh"},
2>                   {user_dir, "/home/ferd/.ssh"}]).
{ok,<0.52.0>}
```

我仅仅设置了些简单的选项，system_dir和user_dir，system_dir指定了宿主（host）文件所在的目录，user_dir中包含了一些SSH配置文件。还有很多其他选项可供使用，包括：允许特殊密码，公钥的定制化处理等等[65]。

任何 SSH 客户端都可以通过如下方式连接该 SSH Daemon：

```
$ ssh -p 8989 ferd@127.0.0.1
Eshell Vx.x.x  (abort with ^G)
1>
```

采用这种方式，无需在当前机器上安装Erlang，就可以和一个Erlang系统进行对话。想要离开时，只需断掉SSH连接即可（关掉终端）。千万不要运行q() 和init:stop()函数，它们会终止掉远程主机[66]。

如果在连接时碰到问题，可以在 ssh 命令中增加这个选项：-oLogLevel=DEBUG，就可以看到详细的调试信息了。

4.4 命名管道

在使用命名管道（named pipeline）连接到一个Erlang节点时，不需要该节点以显式的分布式方式启动，这种方法鲜为人知。具体做法为：用run_erl启动Erlang，把Erlang包装在一个命名管道中[67]：

```
$ run_erl /tmp/erl_pipe /tmp/log_dir "erl"
```

第一个参数是充当命名管道的文件的名称。第二个参数指定了日志保存文件[68]。

然后使用 to_erl 程序来连接节点：

---

[65] 选项完整设置说明，请参考：http://www.erlang.org/doc/man/ssh.html#daemon-3
[66] 无论采用哪种方法和远程 Erlang 节点交互，都不要运行它们。
[67] “erl”是要执行的命令。其他参数可以附在其后。比如，”erl +K true”，会打开 kernel polling。
[68] 这种做法会导致每次输出时都调用 fsync，如果标准输出 IO 动作过多时，会有很大的性能损失。

```
$ to_erl /tmp/erl_pipe
Attaching to /tmp/erl_pipe (^D to exit)

1>
```

这样，Shell 就连接上了。直接关闭 stdio（按^D）会断开和 shell 的连接，Erlang 系统本身还保持运行。

4.5 练习

复习题

1、 连接远程节点有哪 4 种方法？
2、 能连接一个没有名字的节点吗？
3、 进入到作业控制模式（JCL）的命令是什么？
4、 如果系统会向标准输出打印很多的数据，那么哪种连接远程 shell 的方式要避免使用？
5、 哪种远程连接不能用^G 断开？
6、 断开一个会话时，哪些命令绝对不能使用？
7、 本章提到的所有方法都支持多个用户同时连接到相同的 Erlang 节点，并且不会出现问题吗？

第 5 章：运行时度量

在用于产品开发时，Erlang VM 的最佳卖点之一是，其所提供的无比透明的、包罗万象的运行时自省（introspection）、调试、信息收集和分析功能。

能够以编程的方式访问这些运行时度量数据的优势在于：编写依赖于这些数据的工具非常容易，并且让某些任务或者看门狗功能自动化也同样简单[69]。此外，在需要时，还可以略过这些工具，直接深入到VM内部获取信息。

在系统成长过程中，要保持其在生产环境中的健康性，确保系统在宏观、微观每个角度都可观测是一种有效的方法。并没有什么普适的方法可以事先知道系统运行正常与否。你必须不断收集大量的数据，并经常地观察它们，这样才能形成系统正常运行时是个什么样子的概念。如果哪天情况出现了异常，你就可以根据所学，从各个角度进行查看，从而相对容易地找到问题所在。

在本章中（以及几乎后面所有章节）所展示的全部概念和特性，基本上都只依赖于标准库（OTP 常规发布的一部分）中的功能。

不过，这些特性并没有集中在一个地方，并且在生产系统中很容易用错，导致问题。它们只是些构建块，还不足以成为有用的工具。

因此，为了能够更加简便、好用。recon[70]库对一些常用的操作进行了重新组织，可以在生产环境中安全地用。

5.1 全局视图

要想从大的方面了解 VM，可以跟踪那些通用的 VM 统计和度量数据，不用管其上所运行的代码是啥。此外，所采用的方法要能满足针对每个度量的长时视图——因为有些问题只有在运行很长时间（几周）后才会出现，在小的时间窗口内不能被检测到。

内存、进程泄漏问题都是需要通过长时视图才能暴露出来的典型例子，还有就是识别像天或者周这样时间长度中某些活动的规则或者不规则峰值时，往往也需要数月的数据才能进行确认。

对于这些情况，可以采用已有的 Erlang 度量应用。一些常见的选择如下：

- folsom[71]，它把全局的以及应用相关的度量数据都存在VM的内存中。
- vmstat[72]和statsderl[73]，通过statsd[74]把度量数据发送给graphite。

[69] 要保证自动化的过程不会失效且在进行矫正行为时不极端，是一项更复杂的工作。
[70] http://ferd.github.io/recon/
[71] https://github.com/boundary/folsom
[72] https://github.com/ferd/vmstats
[73] https://github.com/lpgauth/statsderl

- exometer[75]，这是一个高级的度量系统，可以和folsom（其中之一）以及众多后端（graphite、collectd、statsd、Riak、SNMP等）集成。它是这类应用中的新成员。
- ehmon[76]，它直接打印到标准输出上，随后可以通过一些特殊代理或者splunk之类的抓取
- 自制解决方案，通常使用ETS表和进程来周期性地转储数据[77]。
- 或者当没啥可用，并且遇到麻烦时，可以在shell中循环调用一个函数把东西打印出来[78]。

最好去了解一下这些应用，从中挑选出一个，并把度量数据持久化起来，这样可以随时对它们进行查看。

5.1.1 内存

在大多数工具中所报告的 Erlang VM 内存基本上都是基于 erlang:memory()实现的：

```
1> erlang:memory().
[{total,13772400},
 {processes,4390232},
 {processes_used,4390112},
 {system,9382168},
 {atom,194289},
 {atom_used,173419},
 {binary,979264},
 {code,4026603},
 {ets,305920}]
```

对这些内容需要做些解释。

首先，返回值的单位都是字节，它们表示已经分配了的内存（Erlang VM 实际使用的内存，不是操作系统分配给 Erlang VM 使用的内存）。它会慢慢变得比操作系统报告的数字小很多。

total字段包含了由进程和系统使用的内存之和（这个数字是不全面的，除非使用了instrumented VM[79]）。Processes字段指示的是Erlang进程、进程堆栈和堆使用的内存。system字段包含了其余的部分：ETS表、VM中原子、refc binary[80]、以及一些我没有提及的隐藏数据。

如果想得到虚拟机占用的所有内存，也就是会触发系统限制（ulimit）的内存量，就很难从VM内部获取了。如果不想通过调用top或者htop得到这个数据，就只能深入到VM的内存分配器中去寻找[81]。

幸运的是，可以使用 recon 中的函数 recon_alloc:memory/1 来得到这个值，参数如下：

---

[74] https://github.com/etsy/statsd/
[75] https://github.com/Feuerlabs/exometer
[76] https://github.com/heroku/ehmon
[77] 有些常见的模式可以使用 extr application，见：https://github.com/heroku/ectr
[78] 紧急时，可使用 recon application 中的 recon:node_stats_print/2 函数
[79] 译者注：采用-instr 标志启动 erl
[80] 见 7.2 节
[81] 见 7.3.2 节

- used 参数用来报告已分配的 Erlang 数据实际占用的内存大小
- allocated 参数用来报告 VM 保留的内存大小。包括：已使用的内存，以及虽未使用但是 OS 已经分配出来的内存。如果需要应对 ulimit 或者想知道 OS 报告的值，这个数值正是所需的。
- unused 参数用来报告由 VM 保留但是尚未分配出去的内存大小。其值等于 allocated - used。
- usage 参数会返回已使用的和已分配的内存百分比（0.0..1.0）.

还有其他一些选项可用，不过只有在解决第 7 章中介绍的内存泄漏问题时，才需要用上它们。

5.1.2 CPU

对于 Erlang 开发者来说，有个坏消息，CPU 非常难以度量。原因有如下几个：

- 在调度时，VM 做了很多和进程无关的工作——是调度工作忙还是 Erlang 进程工作忙非常难以区分。
- VM 内部使用了一种基于 reduction 的模型，reduction 代表了某些数量的工作执行。每个函数调用，包括 BIF，都会增加进程的 reduction 计数。进程在执行一定数量的 reduction 后，就会让出执行权。
- 当负载变低时，Erlang 调度器所在的线程会进入忙等。这可以保证以低的延时处理突发的负载。通过使用 VM 标志 +sbwt none | very_short | short | medium | long | vey_long 来改变这个值。

这些因素综合起来，导致非常难以测量出 CPU 在运行实际 Erlang 代码上所花费的时间。在生产环境中，经常会看到有些 Erlang 节点做的实际工作并不多，却占用了大量的 CPU，不过，在负载增加时，这些保留的处理能力会用来处理许多工作。

关于这方面最为精确的数据是调度器钟表时间（wall clock）。这是个可选的测量数据，需要手工打开，并定期轮询。它反映了调度器花在运行进程、常规的 Erlang 代码、NIF、BIF、垃圾回收等上的时间与花在空转以及调度进程上的时间的百分比。

这个值表示的是调度器利用率，不是 CPU 利用率。比值越高，负载就越大。

Erlang/OTP参考手册[82]上给出了基本的调用方法，不过调用recon可以直接得到这个值：

```
1> recon:scheduler_usage(1000).
[{1,0.9919596133421669},
 {2,0.9369579039389054},
 {3,1.9294092120138725e-5},
 {4,1.2087551402238991e-5}]
```

函数 recon:scheduler_usage(N)的查询间隔为 N 毫秒（例子中是 1 秒），然后输出每个调度器的结果。这示例中，VM 有两个高负载的调度器（分别为 99.2%和 93.7%），还有两个基本没有使用的调度器（使用率远低于 1%）。像 htop 之类的工具会（针对每个核）报告出类似的结果：

[82] http://www.erlang.org/doc/man/erlang.html#statistics_scheduler_wall_time

```
[|||||||||||||||||||||||||      70.4%]
[|||||||                        20.6%]
[|||||||||||||||||||||||||||||||100.0%]
[|||||||||||||||||              40.2%]
```

这两个结果说明，在所报告的 CPU 利用率中，有相当一部分是可以在需要时用于执行实际的 Erlang 工作的（假定调度器大部分时间是在忙等，小部分时间在挑选要执行的任务），但是却被 OS 报告为忙。

还可能会出现另外一个有趣的现象，recon 显示的调度器使用率会比 OS 所报告的具有更高的值（1.0）。那些在等待操作系统资源的调度器，由于不能处理其他工作，会被认为是占用了。那么，如果 OS 自己挂在一些不使用 CPU 的任务上，就可能会出现虽然 Erlang 调度器没做什么工作，但却报告出 100%的使用率的情况。

在进行容量规划时，对这些情况的考虑尤为重要，比起去查看 CPU 利用率或者负载，它们能给出更好的余量指示。

5.1.3 进程

如果想评估一下 VM 中正在执行的任务量时，可以看看进程的总体信息情况。在 Erlang 中，通常都会使用进程来表示真正并发的行为——在 web 服务器上，通常一个请求或者连接就是一个进程，如果系统是有状态的，那么每个用户就是一个进程——因此，节点上的进程数量可以作为一种负载度量。

5.1 小节中提到的大多数工具都能以某种方式来追踪进程，不过，如果想用手工的方式获得进程数目的话，下面的做法就足够了：

```
1> length(processes()).
56535
```

对这个值进行跟踪，并结合其他度量数据，非常有助于评估、刻画负载，或者检测进程泄漏。

5.1.4 端口

应该以对待进程同样的方式来对待端口。端口是一种数据类型，涵盖了各种和外部世界通信的连接和 socket：TCP socket、UDP socket、SCTP socket、文件描述符等等。

有个常用的函数（和进程类似）可以计算端口的数量：length(erlang:ports())。不过，这个函数会把所有类型的端口合并成一项。而 recon 则可以把它们按类型分别显示：

```
1> recon:port_types().
[{"tcp_inet",21480},
 {"efile",2},
 {"udp_inet",2},
 {"0/1",1},
 {"2/2",1},
 {"inet_gethost 4 ",1}]
```

这个列表中包含了端口的类型以及每种类型的端口数目。类型的名字是一个字符串，由 Erlang VM 自己定义。

*_inet 类型的端口基本上都是 socket，前缀指示出所使用的协议（TCP、UDP、SCTP）。efile 类型表示文件，”0/1”、”2/2”则分别表示标准 I/O 通道（stdin 和 stdout）以及标准错误通道（stderr）的文件描述符。

其他类型基本上都会被赋予与其通信的驱动同样的名字，基本上都是port程序[83]或者port驱动[84]。

和进程一样，跟踪这个值有助于评估系统的负载或者使用率、检测泄漏，等等。

5.2 深入细节

当某个“大的”视图（也许是日志）为我们指出了针对某个问题的潜在原因时，接下来就可以做一件非常有趣的工作：进行针对性的实验。进程处于一种奇怪的状态吗？也许需要对它进行跟踪[85]！如果想观察某个特定函数的调用、输入以及输出情况，使用跟踪就再合适不过了，不过在求助于跟踪方法之前，还需要了解很多其他细节信息。

除内存泄漏之外（需要特殊的技术，会在第 7 章介绍），要了解的细节信息基本上都和进程和端口（文件描述符和 socket）有关。

5.2.1 进程

对一个运行中的 Erlang 系统来说，进程绝对是重要的组成部分。正因为进程是所有运行实体的基础，因此会想去了解它们的更多信息。幸运的是，VM 提供了大量的可用信息，其中有些可以安全使用，有些在生产环境中使用是不安全的（因为会返回非常大的数据集合，拷贝到 shell 以及打印所需的内存量会造成节点崩溃）。

进程的所有信息都可以通过调用process_info（Pid, Key）或者process_info(Pid, [keys])[86]得到。下面是一些常用的键值[87]：

---

[83] http://www.erlang.org/doc/tutorial/c_port.html
[84] http://www.erlang.org/doc/tutorial/c_portdriver.html
[85] 参见第 9 章
[86] 如果进程包含有敏感信息，那么可以调用 process_flag(sensitive, true)来要求数据保持私密。
[87] 查看所有选项信息，请参见：http://www.erlang.org/doc/man/erlang.html#process_info-2

元信息

dictionary 返回进程字典中的所有数据项[88]。通常可安全使用，因为一般不应该把G字节的数据存放于此。

group_leader 进程的group_leader决定IO（文件，io:format的输出）的去向[89]。

registered_name 如果进程有名字的话，可以通过这个键值获取。

status 进程在调度器内部的分类。有如下几种值：

exiting 进程已经结束，但是还未被完全清除

waiting 进程在 receive … end 中等待

running 运行中

runnable 准备就绪，但是尚未被调度，因为另一个进程正在运行中

garbage_collecting 垃圾回收中

suspended 不管是因为 BIF 调用，还是因为 socket 或端口缓冲满，作为一种反压机制的结果而被挂起，都是此值。仅当端口不忙时，进程才会再次变成 runnable 的。

信号

links 会报告出进程所链接的所有其他进程以及端口（socket、文件描述符）列表。一般来说可安全调用，不过对于一些大的 supervisor 要小心使用，因为返回的列表中会有成千上万的项。

monitored_by 会返回所有监控当前进程（通过 erlang:monitor/2）的进程列表。

monitors 和 monitored_by 相反，返回所有被当前进程监控的进程列表。

trap_exit 如果该进程捕获 exit，就返回 true，否则返回 false。

位置

current_function 以 tuple 的形式（{Mod, Fun, Arity}）显示进程的当前运行函数。

current_location 显示进程在模块中的位置，显示格式为：{Mod, Fun, Arity, [{file, FileName},{line, Num}]}

current_stacktrace 上一个选项（current_location）的更详细版本，会以“current_location”列表的形式显示当前的堆栈跟踪信息。

initial_call 显示进程 spawn 时运行的函数，形式为：{Mod, Fun, Arity}。该信息可以用来识别进程创建时执行的函数，不能识别当前运行的函数。

内存使用

binary 显示所有refc binary[90]的引用及其大小。如果此类binary分配过多，那么调用起来就不安全。

garbage_collection 报告进程中有关垃圾回收的信息。这部分信息在官方文档中被声明为“subject to change”，应当以此为准。这部分内容往往会包括进程已经执行的垃圾回收的次数，完整清理（full-sweep）垃圾回收的选项，堆大小等等。

heap_size 典型的 Erlang 进程包含有一个“老”堆和一个“新”堆，并会经历分代型

---

[88] 参见http://www.erlang.org/course/advanced.html#dict 和 http://ferd.ca/on-the-use-of-the-processdictionary-in-erlang.html。

[89] 更多细节，请参见：http://learnyousomeerlang.com/building-otp-applications#the-application-behaviour 和 http://erlang.org/doc/apps/stdlib/io_protocol.html。

[90] 参见 7.2 小节

（generational）的垃圾回收。这项内容会显示进程最新代的堆大小，通常也包括堆栈大小。返回值的单位是 word。

memory 返回进程占用内存的大小，单位为字节，包括：调用栈、堆以及作为进程的一部分由 VM 使用的内部结构。

message_queue_len 显示进程邮箱中等待的消息个数。

messages 返回进程邮箱中的所有消息。在生产环境中，这个调用是极度危险的，比如，如果正在调试某个进程而导致它被锁住，那么其邮箱中会堆积上百万条消息。一定要先调用 message_queue_len 以确保安全。

total_heap_size 和 heap_size 类似，不过也会包含所有其他部分的堆，包括老堆。返回值的单位是 word

工作量

reductions Erlang VM 基于 reduction 来进行调度，reduction 是一种规定的工作量单位，用来保证调度实现的高度可移植性（基于时间的调度很难做到在所有 Erlang 运行的 OS 上都高效）。进程的 reduction 值越高，表示其（在 CPU 和函数调用意义上）所做的工作就越多。

幸运的是，对于所有常用并且安全的项，都可以使用 recon 函数 recon:info/1 来获取到：

```
1> recon:info("<0.12.0>").
[{meta,[{registered_name,rex},
        {dictionary,[{'$ancestors',[kernel_sup,<0.10.0>]},
                     {'$initial_call',{rpc,init,1}}]},
        {group_leader,<0.9.0>},
        {status,waiting}]},
 {signals,[{links,[<0.11.0>]},
           {monitors,[]},
           {monitored_by,[]},
           {trap_exit,true}]},
 {location,[{initial_call,{proc_lib,init_p,5}},
            {current_stacktrace,[{gen_server,loop,6,
                                  [{file,"gen_server.erl"},{line,358}]},
                                 {proc_lib,init_p_do_apply,3,
                                  [{file,"proc_lib.erl"},{line,239}]}]}]},
 {memory_used,[{memory,2808},
               {message_queue_len,0},
               {heap_size,233},
               {total_heap_size,233},
               {garbage_collection,[{min_bin_vheap_size,46422},
                                    {min_heap_size,233},
                                    {fullsweep_after,65535},
                                    {minor_gcs,0}]}]},
 {work,[{reductions,35}]}]
```

为了方便起见，recon:info/1 的第一个参数可以是任意类型的 pid 表示：原始 pid 值、字符串（"<0.12.0>"）、进程注册原子、全局名称（{global, Atom}）、第三方注册库（比如，gproc：{via, gproc, Name}）中的注册名或者 tuple（{0,12,0}）。进程只需是调试节点的本地进程即可。

如果只想要某一类信息，可以直接使用类属名作为参数：

```
2> recon:info(self(), work).
{work,[{reductions,11035}]}
```

也可以按照和 process_info/2 完全一样的方式调用：

```
3> recon:info(self(), [memory, status]).
[{memory,10600},{status,running}]
```

后一种形式可以用来获取不安全数据[91]。

有了这些数据，我们就具备了调试系统所需要的基础。接下来的挑战往往在于基于进程的个体以及全局信息，找出那些有问题的目标进程。

比如，如果寻找内存占用高的进程，可以把节点中的所有进程都罗列出来，并找到占用最高的前 N 个。使用前面列出来的属性以及 recon:proc_count(Attribute, N)函数，可以得到需要的信息：

```
4> recon:proc_count(memory, 3).
[{<0.26.0>,831448,
  [{current_function,{group,server_loop,3}},
   {initial_call,{group,server,3}}]},
 {<0.25.0>,372440,
  [user,
   {current_function,{group,server_loop,3}},
   {initial_call,{group,server,3}}]},
 {<0.20.0>,372312,
  [code_server,
   {current_function,{code_server,loop,1}},
   {initial_call,{erlang,apply,2}}]}]
```

前面提到的所有属性都可以作为参数使用，如果节点中有长期运行的、可能会引发问题的进程，这个函数非常有用。

不过，如果大部分进程都是短期的，尤其是生存期短到无法通过工具观察，或者需要一个移动的观察窗口时（比如，*现在*有哪些进程在消耗内存或者运行代码），就会出现问题。

对于这种情况，Recon 提供了 recon:proc_window(Attribute, Num, Milliseconds)函数。

有一点很重要，那就是要把这个函数看作是一段滑动窗口的快照。一段程序的采样时间线可能如下所示：

```
--w---- [Sample1] ---x----------y----- [Sample2] ---z--->
```

这个函数会间隔参数 Milliseconds 指定的时间，采样两次。

[91] 译者注：不安全数据是指进程邮箱之类的数据，在 recon:info/1 中是被排除掉的。

有些进程的生命期在 w 和 x 之间，有些在 y 和 z 之间，有些在 x 和 y 之间。这些样本是不完整的，因此仅供参考。

如果大多数进程的生存期都在x到y这个时间区间（绝对意义上），那就要确保采样时间要比这个区间小，这样对于许多进程来说，其生存期就能够等价于在w和z之间。否则，会导致结果扭曲：由于长生存期进程会花 10 倍的时间去累加数据（比如reduction数据），从而会被误认为是这方面资源的头号消耗者[92]。

这个函数运行产生的结果如下：

```
5> recon:proc_window(reductions, 3, 500).
[{<0.46.0>,51728,
  [{current_function,{queue,in,2}},
   {initial_call,{erlang,apply,2}}]},
 {<0.49.0>,5728,
  [{current_function,{dict,new,0}},
   {initial_call,{erlang,apply,2}}]},
 {<0.43.0>,650,
  [{current_function,{timer,sleep,1}},
   {initial_call,{erlang,apply,2}}]}]
```

有了这两个函数，就可以定位出引发问题或者行为错乱的某个具体进程了。

5.2.2 OTP进程

如果有问题的进程是 OTP 进程（产品系统中的大部分进程都绝对应该是 OTP 进程），那么可使用的检查工具就会立即多出不少。

一般来讲，这些工具都包含在sys模块[93]中。请阅读它的文档，然后就会明白它为何如此有用。对于任何OTP进程，它都可以完成如下的功能：

- 可以把所有消息和状态迁移打印到 shell、文件、甚至是某个可查询的内部缓冲中。
- 信息统计（reduction、消息数量、时间等等）
- 获取进程状况（status）（包括状态数据在内的元信息）
- 获取进程状态（state）（#state{}记录中的内容）
- 替换进程状态
- 定制调试回调函数

它还提供了挂起或者恢复进程运行的功能。

我不打算介绍关于这些函数的过多细节，不过得知道可以从文档中查到它们。

[92] 警告：这个函数依赖于在两个快照中所采集到的数据，然后会构建一个字典来取差值。当进程数目很多时（上万），会耗费大量的内存和不少时间。

[93] 参见：http://www.erlang.org/doc/man/sys.html

5.2.3 端口

和进程类似，对 Erlang 端口也可以做很多自省（introspection）操作。基本信息可以通过调用 erlang:port_info(Port, Key)获取，使用 inet 模块则可以获取到端口的更多其他信息。大部分信息都被 recon:port_info/1-2 函数进行了重新组织，这两个函数的使用接口和进程方面的对等函数类似。

元信息

id 端口的内部索引。除了用来区分端口外，没啥特殊用途。

name 端口的类型——比如像"tcp_inet"、"udp_inet"或者"efile"之类的名字

os_pid 如果端口不是 inet socket，而是代表一个外部的进程或者程序，那么这个值会包含和外部程序对应的 OS 进程的 pid

信号

connected 每个端口都会有一个控制进程来对其负责，connected 指的就是这个进程的 pid。

links 端口可以和进程链接起来，方式和进程间的类似。links 包含的就是所链接进程的列表。如果端口曾经的 owner 并不多，或者没有被手工和许多进程链接在一起，这个调用应该是安全的。

monitors 端口（代表外部程序）可以让外部程序监控 Erlang 进程。这个调用会显示这些进程。

IO

input 端口读入的字节数

output 写入端口的字节数

内存使用

memory 运行时系统为该端口分配的内存（单位为字节）。这个值会偏小，并没有包含端口自己分配的空间

queue_size 端口程序有一个特殊的队列，称为驱动程序队列[94]。这个调用会返回队列大小，单位为字节。

特定类型

Inet Ports 返回inet特定的数据，包括统计数据[95]、socket（sockename）的本地地址和端口号以及使用的inet选项[96]。

其他 目前除了 inet 端口外，recon 不支持其他类型的端口，会返回空列表。

如下调用可以得到这个列表：

---

[94] 驱动程序队列用来对由仿真器发送给驱动的数据排队（从驱动发送给仿真器的数据会由仿真器采用常规 Erlang 消息队列进行排队）。当驱动需要等待慢速设备，想把控制权交回给仿真器时，这个队列非常有用。

[95] http://www.erlang.org/doc/man/inet.html#getstat-1

[96] http://www.erlang.org/doc/man/inet.html#setopts-2

```
1> recon:port_info("#Port<0.818>").
[{meta,[{id,6544},{name,"tcp_inet"},{os_pid,undefined}]},
 {signals,[{connected,<0.56.0>},
           {links,[<0.56.0>]},
           {monitors,[]}]},
 {io,[{input,0},{output,0}]},
 {memory_used,[{memory,40},{queue_size,0}]},
 {type,[{statistics,[{recv_oct,0},
                     {recv_cnt,0},
                     {recv_max,0},
                     {recv_avg,0},
                     {recv_dvi,...},
                     {...}|...]},
        {peername,{{50,19,218,110},80}},
        {sockname,{{97,107,140,172},39337}},
        {options,[{active,true},
                  {broadcast,false},
                  {buffer,1460},
                  {delay_send,...},
                  {...}|...]}]}]
```

除此之外，recon 中同样有用来发现具有特定问题的端口的函数，用法和处理进程的类似。要注意的是，到目前为止 recon 仅支持针对 inet 端口的调用，能使用的属性也受限：发送的字节数、接收的字节数、以及收发的字节数（分别对应 send_oct、recv_oct、oct）；还有就是发送的包数、接收的包数以及收发的包数（分别对应 send_cnt、recv_cnt、cnt）。

使用总的累积结果，可以帮助我们发现是谁在慢慢吃掉带宽：

```
2> recon:inet_count(oct, 3).
[{#Port<0.6821166>,15828716661,
  [{recv_oct,15828716661},{send_oct,0}]},
 {#Port<0.6757848>,15762095249,
  [{recv_oct,15762095249},{send_oct,0}]},
 {#Port<0.6718690>,15630954707,
  [{recv_oct,15630954707},{send_oct,0}]}]
```

从中可以看出，有些端口只是在接收，并且量很大。接下来，可以调用函数 recon:port_info(“#Port<0.682166>”)获取更详细的信息，找到该 socket 的拥有者以及下一步工作需要的信息。

或者，在其他情况下，我们想看看在给定的时间窗口内[97]谁发送了最多的数据，此时可以使用函数recon:inet_window(Attribute, Count, Milliseconds)：

```
3> recon:inet_window(send_oct, 3, 5000).
[{#Port<0.11976746>,2986216,[{send_oct,4421857688}]},
 {#Port<0.11704865>,1881957,[{send_oct,1476456967}]},
 {#Port<0.12518151>,1214051,[{send_oct,600070031}]}]
```

在这个输出中，元组中间的那个值是在所选择的特定时间区间中（例子中是 5 秒）累加的

[97] 参见前面小节中对 recon:proc_window/3 的介绍。

send_oct 的数目（也可以在调用中选择其他属性）。

要从行为异常的端口正确地找出相关的进程(然后可能要找到某个特定的使用者或者客户)，仍需要一些手工工作，不过做这些工作所需的工具都已经具备了。

5.3 练习

复习题

1、 关于 Erlang 的内存，可以报告出哪些类型的信息？
2、 如果想了解进程的总体情况，可以做的有价值的度量是什么?
3、 什么是端口？如何了解其总的信息？
4、 对于 Erlang 系统，为什么不能用 top 或者 htop 来获取 CPU 的使用率？可用的备选方法是什么？
5、 请说出两种可获取到的和进程信号有关的信息？
6、 如何找到进程当前正在运行的代码？
7、 对于一个特定进程来说，可以获取到不同种类的内存信息分别是什么？
8、 如何知道进程工作量的大小？
9、 请列举出在生产系统中检查进程时，获取起来不安全的信息。
10、sys 模块中针对 OTP 进程提供了哪些功能？
11、在检查 inet 端口时，可以得到哪些信息？
12、如何得到端口的类型（文件、TCP、UDP）？

开放问题

1、 在进行全局性度量时，为何需要长的时间窗口？
2、 针对如下的每一项，分别说说在需要定位和其相关的问题时，recon:proc_count/2 和 recon:proc_window/3 哪一个更合适？
   a) reduction
   b) 内存
   c) 消息队列长度
3、 如何找到某个给定进程的 supervisor？
4、 什么情况下使用 recon:inet_count/2？什么情况下使用 recon:inet_window/3？
5、 操作系统报告的内存情况和 Erlang 的内存函数报告的情况有所不同的原因是什么？
6、 为何 Erlang 会在实际不忙时却看起来很忙？
7、 你能找到节点中那些已经准备就绪但是又无法立即得到调度的进程吗？

动手题

运行https://github.com/ferd/recon_demo中的代码，回答如下问题
1、 系统内存有多少？

2、 节点耗费的 CPU 资源多吗？
3、 可有进程的邮箱溢出了？
4、 哪个聊天进程（council_member）占用的内存最多？
5、 哪个聊天进程最耗 CPU？
6、 哪个聊天进程最耗带宽？
7、 哪个聊天进程发送的 TCP 消息最多？哪个最少？
8、 你能找到节点中哪个进程同时持有多个连接或者文件描述符吗？
9、 你能找到节点中当前哪个函数被最多的进程同时调用吗？

第 6 章 理解崩溃文件

Erlang节点在崩溃时，都会产生一个崩溃文件（crash dump）[98]。

Erlang的官方文档中对文件的格式有详细的说明[99]，通过阅读文档，可以知道每个数据项的含义，从而有助于深度理解崩溃文件的内容。其中有些数据项不是那么容易理解，因为需要同时理解它们所涉及的VM部分内容，不过这部分内容过于复杂，不在本文中进行介绍。

崩溃文件会被命名为erl_crash.dump，默认会放在Erlang进程的运行目录。通过设置ERL_CRASH_DUMP环境变量[100]可以改变这个行为（以及文件名称）。

6.1 全局信息

阅读崩溃文件有助于在事后找出节点的死因。一种快速浏览的方法是使用recon的脚本erl_crashdump_analyzer.sh[101]，用它来分析崩溃文件：

```
$ ./recon/script/erl_crashdump_analyzer.sh erl_crash.dump
analyzing erl_crash.dump, generated on:  Thu Apr 17 18:34:53 2014

Slogan: eheap_alloc: Cannot allocate 2733560184 bytes of memory
(of type "old_heap").

Memory:
===
  processes: 2912 Mb
  processes_used: 2912 Mb
  system: 8167 Mb
  atom: 0 Mb
  atom_used: 0 Mb
  binary: 3243 Mb
  code: 11 Mb
  ets: 4755 Mb
  ---
  total: 11079 Mb
```

[98] Segment fault 或者在转储时由于违反了 ulimit 而被 OS 杀死的情况除外。
[99] http://www.erlang.org/doc/apps/erts/crash_dump.html
[100] Heroku 的路由和远程监测团队使用 heroku_crashdumps 应用来设置崩溃文件的路径和名称。可以把这个应用加到一个工程中，它会在启动时命名 dump 文件，并把它们放置到一个预先配置好的位置。
[101] https://github.com/ferd/recon/blob/master/script/erl_crashdump_analyzer.sh

```
Different message queue lengths (5 largest different):
===
      1 5010932
      2 159
      5 158
     49 157
      4 156

Error logger queue length:
===
0

File descriptors open:
===
  UDP:  0
  TCP:  19951
  Files:  2
  ---
  Total:  19953

Number of processes:
===
36496

Processes Heap+Stack memory sizes (words) used in the VM (5 largest
different):
===

      1 284745853
      1 5157867
      1 4298223
      2 196650
     12 121536

Processes OldHeap memory sizes (words) used in the VM (5 largest
different):
===
      3 318187
      9 196650
     14 121536
     64 75113
     15 46422

Process States when crashing (sum):
===
      1 Garbing
     74 Scheduled
  36421 Waiting
```

这些转储数据并没有直接给出问题所在，不过提供了很好的线索。例如，其中的节点用光了内存，总共有 15GB 内存（我之所以知道，是因为这是我所使用的实例的最大容量），报告使用了 11079MB。这可能是如下问题的症状表现：

- 内存碎片
- C 代码或者驱动中有内存泄漏
- 在产生该崩溃文件前，大量内存正在被垃圾回收[102]

[102] 尤其是基于引用计数的 binary 内存，这些内存位于全局堆中，在产生崩溃文件前正在被垃圾回收。因此，binary 内存就会被少报。详细细节请参见第 7 章。

更常用的方法是，看看有哪些内存方面的异常。把它和进程的数量以及邮箱的大小关联起来。往往可以找到原因。

在示例转储文件中，有个进程的邮箱中有 500 万条消息。这就明显告诉我们。要么这个进程在接收消息时存在不匹配的情况，要么它过载了。还有数十个进程的消息队列中堆积了上百条消息——这可能表明有过载或者竞争发生。虽然很难给出对所有转储文件都适用的一般性建议，不过还是有一些建议有助于我们找出问题所在。

6.2 邮箱爆满

对于过载的邮箱，最好的方法是看看消息比较多的那些。如果只有一个消息量大的邮箱，那么就去调查一下崩溃文件中这个进程的其他详细信息。看看是不是因为存在有某些无法匹配的消息，或者是过载了。如果有一个类似的节点正在运行，可以登陆到其上，做些检查。如果发现很多邮箱都过载了，可以使用 recon 的 queue_fun.awk 来找出在崩溃时它们正在执行的函数：

```
$ awk -v threshold=10000 -f queue_fun.awk /path/to/erl_crash.dump
MESSAGE QUEUE LENGTH: CURRENT FUNCTION
=====================================
10641: io:wait_io_mon_reply/2
12646: io:wait_io_mon_reply/2
32991: io:wait_io_mon_reply/2
2183837: io:wait_io_mon_reply/2
730790: io:wait_io_mon_reply/2
80194: io:wait_io_mon_reply/2
...
```

这个脚本会遍历崩溃文件，把邮箱中消息个数大于 1000 的进程正在运行的函数打印出来。在上面示例中，可以看出，整个节点在调用 io:format/2 时被锁在了等待 IO 上。

6.3 进程太多（或者太少）

如果知道节点正常工作时的平均进程数量[103]，那么在定位问题时，进程的数目信息就非常有用了。

进程数量比通常高时，可能表示存在某种泄漏或者过载情况（和具体应用有关）。

如果进程数量远低于平常，那么可以看看节点在终止时是不是有这样的 slogan：

```
Kernel pid terminated (application_controller)
  ({application_terminated, <AppName>, shutdown})
```

[103] 详细信息请参见 5.1.3 小节

在上面示例中，问题的原因是某个 application（<AppName>）达到了其 supervisor 所允许的最大启动次数，从而导致节点终止。对于那些由于错误日志引发的级联故障要认真梳理，以找出问题根源所在。

6.4 端口太多

和进程一样，如果知道系统正常情况下的端口数量，那么通过端口数量信息来定位问题是非常简单、有效的。

数量比正常高说明系统可能过载、受到拒绝服务攻击或者出现常见的资源泄漏。弄清楚泄漏端口的类型（TCP、UDP 以及文件）可以帮助我们定位出问题所在：某些特定资源存在竞争情况，或者是使用这些资源的代码编写错了。

6.5 无法分配内存

此类崩溃是最为常见的一种崩溃类型。关于它，要讲的内容太多，因此我们会在第 7 章专门对此进行介绍，并会讲解如何在正在运行的系统上进行必要的调试。

无论如何，崩溃文件在事后帮助我们找出问题方面总能给予帮助。进程的邮箱及其内存堆方面的信息往往都是不错的解决问题线索。如果进程的邮箱都不是特别大，但是还是耗尽了内存，那么可以使用 recon 脚本看看进程的堆和栈的大小。

如果在内存使用量的分布上有几个大的异类（outlier）存在，那么可以肯定节点的内存是被有限的几个进程所耗尽。如果基本上分布均匀，那么就看看所报告的总的内存使用量是不是很大。

如果这几个方面看起来都没啥问题，可以去找到崩溃文件中的“Memroy”小节，看看是不是有某个内存占用相当大的类型项（比如，ETS 或者 binary）。如果有的话，可能意味着有不期望的资源泄露发生。

6.6 练习

复习题

1、 如何指定崩溃文件存放的位置？
2、 在节点内存耗尽时，常用的定位问题的方法是什么？
3、 如果进程的数量远低于正常值，该如何定位问题？
4、 如果节点因为有个进程使用过多的内存而崩溃，如何找到那个进程？

动手题

基于 6.1 小节中的崩溃文件分析，回答：

1、 导致问题的具体异类（outlier）是什么？

2、 文件中报告出的错误是问题所在吗？如果不是，那可能是什么？

第 7 章 内存泄漏

Erlang 中有很多方式会导致内存流失。有些方式非常易于发现，有些则异常难以定位（幸运的是，这种情况很少发生），不过，你也可能根本就不会碰到这些问题。

发现内存泄漏有两种方式：

1、 通过崩溃文件（见第 6 章）。
2、 通过监控数据中不健康的趋势。

本章主要关注第 2 种情况，因为这种情况更易于调查和实时观测一些。本章会重点介绍：如何找出节点中哪些东西在不断增长以及一些常用的补救方法，如何处理 binary 泄漏（内存泄漏的一种特殊情况），如何检测内存碎片。

7.1 常见的泄漏源

碰到有人因节点崩溃而求助时，首先要做的就是寻求数据。一些有趣的问题和要考虑的数据如下：

- 有崩溃文件吗？其中有特别针对内存的抱怨吗？如果没有，可能和内存无关。否则，就深入进去，可以得到大量的数据。
- 崩溃是周期性的吗？可预测吗？同时还有其他情况发生吗？它们之间有相关性吗？
- 崩溃是只在系统负载峰值时发生呢？还是任何时间都会发生？仅在峰值发生的崩溃通常是由于糟糕的过载管理（参见第 3 章）造成的。任何时间（即使是高峰过后的负载低谷）都会发生的崩溃则更可能是真正的内存问题。

如果所有这些都指向了内存泄漏，那么请安装recon以及第 5 章中提到的度量库中的一个，做好深入研究的准备[104]。

不管是其中哪种情况，首先要看的都是趋势。使用 erlang:memory()，或者某个库或度量系统中的类似工具查看总的内存情况。检查时，遵循如下要点：

- 是不是有某种类型内存的增长速度超过其他种类？
- 是不是有某种类型内存的使用量占了可用空间的绝大部分？
- 是不是有某种类型内存的占用量从不下降，总是在增长（atom 除外）？

有不少解决办法可供选择，它们和增长内存的类型相关。

7.1.1 原子（Atom）

*不要动态生成原子！*原子都存在一个全局表中，并且会一直缓存在那里。看看你在哪些地方

[104] 如果想知道如何连接一个运行节点，请参见第 4 章

调用了 erlang:binary_to_term/1 和 erlang:list_to_atom/1，考虑把它们换成更安全的调用（erlang:binary_to_term(Bin, [safe])以及 erlang:list_to_existing_atom/1）。

如果使用了Erlang发布中自带的xmerl库，可以考虑换成其他开源版本[105]，或者想办法使用自己安全版本的SAX解析器[106]。

如果没有这些问题，那么可以检查一下和节点交互时采用的方式。我曾经在生产环境中遭遇过这样一个问题，我们常用的一些工具会使用随机的名字连接远程节点，或者会产生一些具有随机名字的节点，并通过中心服务器把它们互联起来[107]。Erlang的节点名字会被转换为原子，因此这种做法就足以慢慢地耗尽原子表的空间。要确保从一个固定的集合中产生节点名，或者保证产生速度足够缓慢，在长期运行中不会产生问题。

7.1.2 Binary

请参见 7.2 小节

7.1.3 代码

Erlang 节点上的代码会被加载到一个专属的内存区域，并会一直呆在那里直到被垃圾回收掉。一个模块同时可以共存两份拷贝，因此找到那些非常大的模块不是一件难事。

如果没有大小异常的模块，那么可以看看使用HiPE编译的代码[108]。和常规的BEAM代码不同，HiPE代码是本地代码，因此即使加载了新的版本，也不能从VM中被垃圾回收掉。如果采用本地编译的模块数量比较多或者比较大，并且会在运行时加载，那么内存占用通常会缓慢的累积。

你也可以经常看看节点中是否含有不是自己加载的奇怪模块存在，有的话就该担心有人侵入你的系统了。

7.1.4 ETS

ETS 表是不会被垃圾回收的，只要记录没有被从表中删除，就会一直占用内存。只有手动删除记录（或者删除表）才会释放内存。

ETS 数据很少会泄漏，如果出现了这种情况，可以在 shell 中调用非公开的 ets:i()函数。它会打印出 ETS 表中关于条目数（size）以及内存占用（mem）的信息。可以通过这些信息定位出问题所在。

---

[105] 我喜欢 exml 和 erlsom

[106] 可参考 Ulf Wiger 的意见 http://erlang.org/pipermail/erlang-questions/2013-July/074901.html

[107] 这是一种节点互联的常用方法：有一、两个具有固定名字的中心节点，让其他节点都和其连接。接下来，连接就会自动传播。

[108] http://www.erlang.org/doc/man/HiPE_app.html

很有可能，表中的数据完全都是合法的，此时你将面临一个难题：需要将数据集进行分片（shard），并将它们分布在多个节点上。这部分内容超出了本书的范围，只能祝你好运了。不过，如果时间紧急的话，可以考虑一下对表进行压缩[109]。

7.1.5 进程

导致进程内存增长的情况有很多种。其中，最有趣的都是些常见情况：进程泄漏（你泄漏了进程）、有些进程泄漏了内存等等。有时，泄漏的原因可能不止一个，此时需要进行多种形式的检查。关于进程数量的知识前面已经介绍过了，在此略掉。

链接和监控

从进程的总数上能看出泄漏的迹象吗？如果能，那么就需要检查一下那些没有进行链接的进程，或者窥探一下 supervisor 的子进程列表，看看有没有什么异常情况。

使用一些基本的命令，很容易就能找到那些没有被链接或者监控的进程：

```
1> [P || P <- processes(),
         [{_,Ls},{_,Ms}] <- [process_info(P, [links,monitors])],
         []==Ls, []==Ms].
```

这段程序会返回既没被链接也没被监控的进程列表。对于 supervisor，可以直接调用 supervisor:count_children(SupervisorPidOrName)，看看数量是不是正常，往往可以得到不错的线索。

使用的内存

我们会在 7.3.2 小节中简单介绍进程的内存模型，不过一般来讲，看看 memory 属性即可找出占用内存最多的那个进程。可以从绝对值或者滑动窗口两个维度获取内存使用信息。

对于内存泄漏，除非问题表现为可预测的快速增加，否则绝对值通常都是值得首先检查的：

[109] 请参见 ets:new/2 的 compressed 选项

```
1> recon:proc_count(memory, 3).
[{<0.175.0>,325276504,
  [myapp_stats,
   {current_function,{gen_server,loop,6}},
   {initial_call,{proc_lib,init_p,5}}]},
 {<0.169.0>,73521608,
  [myapp_giant_sup,
   {current_function,{gen_server,loop,6}},
   {initial_call,{proc_lib,init_p,5}}]},
 {<0.72.0>,4193496,
  [gproc,
   {current_function,{gen_server,loop,6}},
   {initial_call,{proc_lib,init_p,5}}]}]
```

除了 memory 外，5.2.1 小节中列出的其他属性也值得去检查一下，包括：message_queue_len，不过 memory 基本上涵盖了所有其他类型。

垃圾回收

进程很可能只是短期使用了大量内存。对于操作开销比较大的长期运行节点，虽然通常不会造成问题，但是在内存资源变得紧张时，还是要避免这种突发的、大量使用内存的情况。

在shell中实时监控所有垃圾回收的成本很高。不过，我们可以设置Erlang的系统监控器[110]，这是最好的方法。

通过 Erlang 的系统监控器，我们可以对垃圾回收周期过长或者进程堆过大之类的情况进行跟踪。可以采用如下方式设置一个临时的监控器：

```
1> erlang:system_monitor().
undefined
2> erlang:system_monitor(self(), [{long_gc, 500}]).
undefined
3> flush().
Shell got {monitor,<4683.31798.0>,long_gc,
                   [{timeout,515},
                    {old_heap_block_size,0},
                    {heap_block_size,75113},
                    {mbuf_size,0},
                    {stack_size,19},
                    {old_heap_size,0},
                    {heap_size,33878}]}
5> erlang:system_monitor(undefined).
{<0.26706.4961>,[{long_gc,500}]}
6> erlang:system_monitor().
undefined
```

第一个命令检查发现系统还没有设置任何监控器——你也不希望改变已有应用或者同事做过的设置。

第二个命令是说，当垃圾回收的时间超过 500 毫秒时就会得到通知消息。第 3 条命令显示得

[110] http://www.erlang.org/doc/man/erlang.html#system_monitor-2

到的消息结果。如果也想对堆的大小进行监控，可以去设置检查{large_heap，NumWords}。一开始不确定时，最好使用大一点的值。过小的值（比如，1 个 word 左右）会导致进程的邮箱被通知消息撑满。

第 5 条命令会取消系统监控器（监控进程退出或者被杀死也会取消设置），第 6 条命令确认取消成功。

设置好监控器，接下来就可以检查一下监控通知消息是否和导致内存泄漏（或者滥用）的内存增长情况同时发生，试着在事情变得太糟糕之前找到问题的根源。看到监控消息后快速反应并迅速检查进程信息（可以使用 recon:info/1）有助于找到应用中的问题所在。

7.1.6 一切正常

如果针对前面介绍的内容，一切看起来都正常，那么问题的根源就可能是 binary 泄漏（见 7.2 小节）或者内存碎片（见 7.3 小节）。如果也不是这两个问题，则可能是 C driver、NIF、甚至 VM 自身出现了泄漏。当然，也不排除内存使用和节点的负载是相称的，其实没啥泄漏，也不需要修改。仅仅需要为系统增加资源或者节点。

7.2 Binaries

Erlang的binary分为两大类：ProcBin和Refc binary[111]。64 字节及以下的binary都直接在进程自己的堆上分配，其整个生命期都在堆中。大于 64 字节的binary会分配在一个binary专用的全局堆上，每个使用它的进程都会在自己的局部堆中持有一个对其的本地引用。这类binary是引用计数的，仅当所有进程中所持有的引用都被垃圾回收了，才会释放binary的内存。

在 99%的情况下，这种机制都可以完美工作。不过，在有些情况下，进程会：

1、 要么做的工作过少，分配和垃圾回收的行为都不会发生
2、 要么逐渐在堆和栈中存放了大量的各式数据结构，对它们进行收集，然后又使用了大量的 refc binary。使用 binary 再次填充满堆（即使是使用虚拟堆来记录 refc binary 的实际大小）会花费很长时间，导致垃圾回收之间有很长的延时。

7.2.1 检测泄漏

检测引用计数型的 binary 泄漏非常容易：先测量一下每个进程中的 binary 引用列表（使用 binary 属性），强制做一次垃圾回收，然后再测量一次，最后计算差值。

可以直接调用 recon:bin_leak(Max)来做这项工作，然后观察一下节点在调用前后的总内存情况：

[111] http://www.erlang.org/doc/efficiency_guide/binaryhandling.html#id65798

```
1> recon:bin_leak(5).
[{<0.4612.0>,-5580,
  [{current_function,{gen_fsm,loop,7}},
   {initial_call,{proc_lib,init_p,5}}]},
 {<0.17479.0>,-3724,
  [{current_function,{gen_fsm,loop,7}},
   {initial_call,{proc_lib,init_p,5}}]},
 {<0.31798.0>,-3648,
  [{current_function,{gen_fsm,loop,7}},
   {initial_call,{proc_lib,init_p,5}}]},
 {<0.31797.0>,-3266,
  [{current_function,{gen,do_call,4}},
   {initial_call,{proc_lib,init_p,5}}]},
 {<0.22711.1>,-2532,
  [{current_function,{gen_fsm,loop,7}},
   {initial_call,{proc_lib,init_p,5}}]}]
```

这个调用会显示出每个进程所持有的以及随后释放的 binary 数量的差值。值-5580 的意思是调用之后的 refc binary 比调用前少 5580。

一般来讲，在每个时间点上都会有一定数量的 refc binary 存在，并不是所有的数字都意味着存在问题。如果发现执行这个调用后，VM 使用的内存急剧降低，则说明 VM 持有了大量没用的 refc binary。

同样的，如果发现有些进程持有了数量巨大的refc binary[112]，则说明可能有问题存在。

接下来，可以使用 recon 中的特殊属性 binary_memory 来找出位于总 binary 内存占用前列的进程：

```
1> recon:proc_count(binary_memory, 3).
[{<0.169.0>,77301349,
  [app_sup,
   {current_function,{gen_server,loop,6}},
   {initial_call,{proc_lib,init_p,5}}]},
 {<0.21928.1>,9733935,
  [{current_function,{erlang,hibernate,3}},
   {initial_call,{proc_lib,init_p,5}}]},
 {<0.12386.1172>,7208179,
  [{current_function,{erlang,hibernate,3}},
   {initial_call,{proc_lib,init_p,5}}]}]
```

这个调用会返回按照 refc binary 所引用的内存总量大小排序的前 N 个进程，可以帮助我们找出持有一些大 binary（不是指引用本身大小）的具体进程。这个函数最好在 recon:bin_leak/1 之前调用，因为后者会对整个节点进行垃圾收集。

7.2.2 修正泄漏

一旦通过 recon:bin_leak(Max)确定了存在 binary 内存泄漏问题，接下来，应该很容易就能找

[112] 我们在 Heroku 调查泄漏时，就曾经发现有一些进程持有上万个 refc binary！

出泄漏最多的前几个进程，看看它们都做了哪些工作。

一般来讲，refc binary 内存泄漏问题有几种不同的解决方法（和泄漏源有关）：

- 每过一定时间间隔就手工进行垃圾回收（不爽，但是高效）
- 不使用 binary（不可取）
- 如果只是持有大binary中的一小段（小于 64 字节）的话[113]，使用binary:copy/1-2[114]。
- 把涉及大 binary 的工作移到临时的一次性进程中，做完工作就死亡（一种弱形式的手动 GC）。
- 或者在合适的时候调用 hibernate（对非活动进程来说，是最干净的方案）

坦白讲，前两种方法不是很合适，除非其他方法都不可行才予考虑。后面三种方法都是不错的方案。

路由 binary

针对有些 Erlang 用户所报告的一种特定用例，有一种针对性的解决方案。这种情况通常是有一个中间人进程把 binary 从一个进程路由给另一个进程。因此，这个中间人进程会获取每一个经过它的 binary 的引用，从而常常会成为 refc binary 主泄漏源。

这类问题的解决方案是，让路由进程返回目标进程的 pid，由原始调用者来进行 binary 的移动。这样，只有需要接触到 binary 的进程才会接触 binary。

这个修正可以透明地在路由 API 函数中实现，调用者无需任何修改。

7.3 内存碎片

内存碎片问题和 Erlang 的内存模型（会在 7.3.2 小节介绍）密切相关。它是目前为止长期运行的 Erlang 节点（通常连续运行几个月）中最难解决的问题之一，相对来说也极难出现。

内存碎片问题的一般症状是，系统在负载峰值期间分配了大量内存，但是事后内存并没有被释放掉。确切的证据则是，节点内部所报告出的内存使用量（通过 erlang:memory()）远低于操作系统报告的使用量。

7.3.1 找出碎片

我们开发了 recon_alloc 模块，专门用来检测和帮助解决这类问题。

迄今为止，这类问题在社团中鲜有报道（或许虽然出现了，但开发人员不知道是啥问题），因此，这里只给出一些大尺度的检测方法。这些方法不会给出确定性的结果，需要使用者自己判断。

---

[113] 拷贝 refc binary 的一大部分也可能是值得的。比如，从 2G 的 binary 中拷贝 10M 就值得付出短期的开销，因为这可以让 2G 的 binary 被垃圾回收，只需要保持 10M 的即可。

[114] http://www.erlang.org/doc/man/binary.html#copy-1

检查已分配内存

调用 recon_alloc:memory/1 会报告出各种各样的内存度量结果，要比 erlang:memory/0 灵活得多。下面是一些可能的参数：

1、recon_alloc:memory(usage)。该调用会返回位于 0 和 1 之间的值，表示当前 Erlang 数据项所真正使用的内存与 Erlang VM 根据工作需要从 OS 那里所申请来的内存的百分比。如果返回值接近 100%，就表明基本上不会有内存碎片问题。确实需要使用这么多内存。

2、recon_alloc:memory(allocated)。可以检查该调用的返回值是否和OS报告的匹配[115]。如果真是碎片问题或者Erlang数据项的内存泄漏，那么这个匹配结果会相当接近[116]。

这些检查结果应该可以确认是否有碎片存在。

找出有问题的分配器

调用 recon_alloc:memory(allocated_type)，看看哪种类型的工具分配器（见 7.3.2 小节）分配的内存最多。然后可以和 erlang:memory()的结果进行比较，看看是否能找出有问题的那一个。

然后试试recon_alloc:fragmentation(current)。它会打印出节点中不同分配器的使用率[117]。

如果有使用率很低的情况存在，可以看看和 recon_alloc:fragmentation(max)打印结果的差别，这个调用会显示出在最大内存负载下分配器的使用情况。

如果两者差别很大，那么很可能某些特定的分配器在峰值使用过后存在内存碎片的问题。

7.3.2 Erlang内存模型

全局级

要想知道内存去哪儿了，首先必须要理解所使用的众多分配器。对于整个虚拟机来说，Erlang 的内存模型是分层级的。如图 7.1 所示，其中有两个主分配器，和一系列子分配器（编号为 1-9）。

[115] 可以调用 recon_alloc:set_unit(Type)把 recon_alloc 所报告的返回值的单位设置成字节，K 字节，M 字节以及 G 字节。

[116] 译者注：如果差距很大，则说明可能有人在 C driver 或者 NIF 中分配了内存。

[117] 更多信息，请参见：http://ferd.github.io/recon/recon_alloc.html

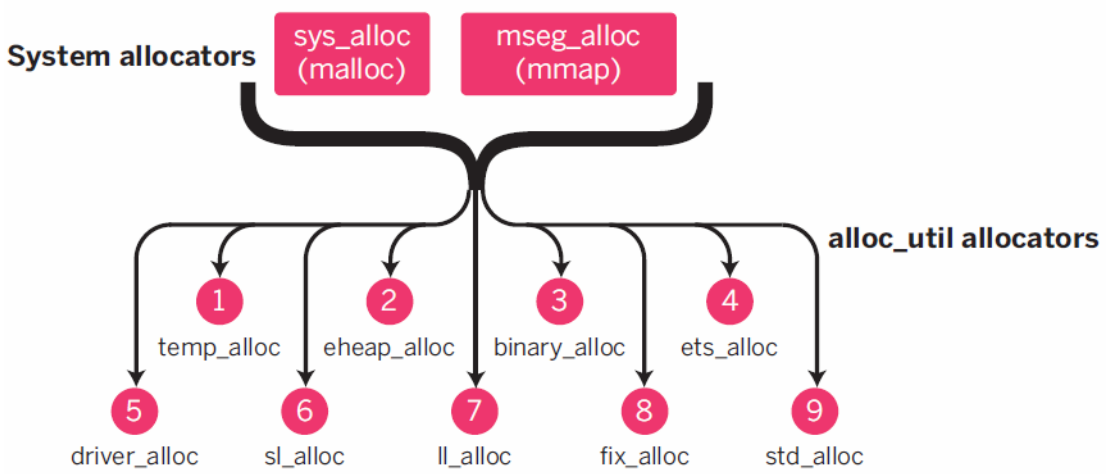


图 7.1：Erlang 的内存分配器和其层级关系。图中没有显示那个特殊的超级内存载体（super carrier），自 R16B03 后可以选择用它来为 Erlang VM 预先分配（并限制）所有可用的内存。

这些子分配器都是些专属分配器，Erlang代码和VM会直接使用它们来为大部分数据类型分配内存[118]：

1、temp_alloc：用于短时使用的临时分配（比如，生存期为一个 C 函数调用的数据）。
2、eheap_alloc：堆数据，用于像 Erlang 进程堆之类东西的分配。
3、binary_alloc：用于引用计数型 binary 的分配器（也就是 binary 全局堆）。存在 ETS 表中的引用计数型 binary 也属于此分配器。
4、ets_alloc：ETS 表中的数据会存在一个隔离的内存空间，并且不参与垃圾回收，会随着数据项的存入和删除而分配和释放。
5、driver_alloc：用于一些特殊驱动数据的存储，驱动也可以使用其他分配器创建 Erlang 数据项。所分配的驱动数据包括：lock/mutex、选项数据、Erlang 端口等等。
6、sl_alloc：用于保存短时内存块，包括像某些 VM 调度信息、处理某些数据类型的小缓冲区之类的东西。
7、ll_alloc：用于保存长时内存块。比如，Erlang 代码自己所占用的内存和原子表等。
8、fix_alloc：用于经常使用的固定大小的内存块的分配。比如，内部进程的 C 结构体，供 VM 内部使用。
9、std_alloc：统一分配器，不适用于前面介绍的情况，都会由它处理。比如，用于命名进程的进程注册表就由它分配。

缺省情况下，在每个调度器中（每个核应该只有一个调度器），每种分配器会有一个实例存在，使用异步线程的 linked-in 驱动会需要一个额外的实例。结果和图 7.1 中显示的结构类似，不过在每个叶子处要变成 N 份。

所有这些子分配器都需要从 mseg_alloc 和 sys_alloc 处请求内存，根据情况不同，主要有两种可能的方式。第一种方式是充当一个多块载体（mbcs），它会获取多个大的内存块，同时被许多 Erlang 数据项使用。针对每个 mbc，VM 都会保留一定数量的内存（在我们的环境中是 8M，可以调整 VM 的选项进行配置），在进行数据项分配时，可以自由在多个 mbcs 中查看，以找出最佳的驻留地。

[118] 每种数据类型所处位置的完整列表，请参见：erts/emulator/beam/erl_alloc.types

当要分配的数据项大小超过了单块载体门限（sbct）[119]时，分配器就会在单块载体（sbcs）中进行这次分配。对于前mmsbc[120]个单块载体，其内存会直接从mseg_alloc分配，后面的从sys_alloc分配，并把数据项存在此处直到释放。

让我们来看看 binary 分配器，它的内存分配情况可能会和图 7.2 中显示的类似：

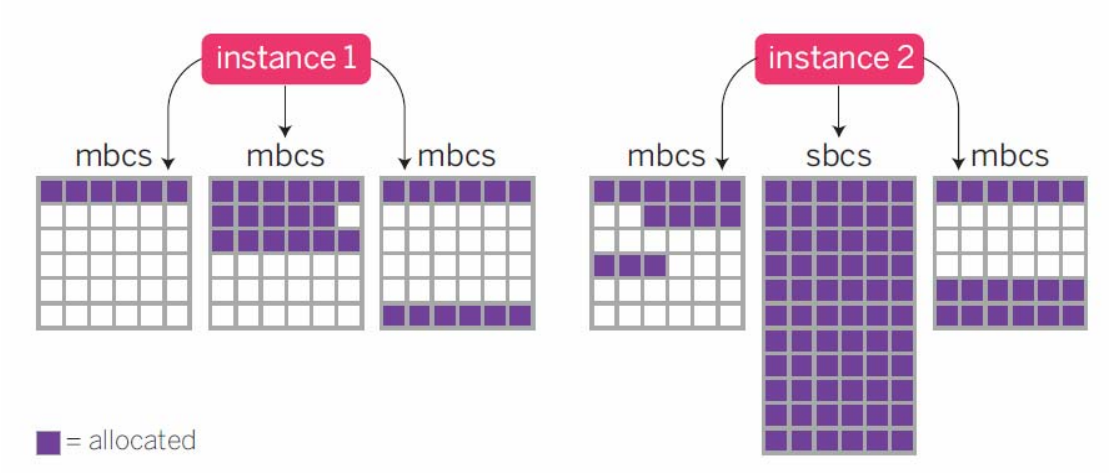


图 7.2：某个特定子分配器中的内存分配情况示例

每当多块载体（或者前 mmsbc 个单块载体）可以被释放时，mseg_alloc 都会把它在内存中保持一段时间，这样在 VM 遭遇下一次内存峰值时，就可以使用预先分配好的内存，而不用每次都向操作系统请求。

接下来，我们需要了解一下 Erlang 虚拟机中不同的内存分配策略：

1、 最佳匹配（bf）
2、 地址序最佳匹配（aobf）
3、 地址序优先匹配（aoff）
4、 地址序优先载体最佳匹配（aoffcbf）
5、 地址序优先载体地址序最佳匹配（aoffcaobf）
6、 良好匹配（gf）
7、 基本匹配（af）

每一种alloc_util分配器都可以独立配置这些分配策略[121]。

最佳匹配（bf）是指，VM 会基于所有可用块的大小构建一棵平衡二叉树，会尽量寻找能够容纳数据项的最小的那个，并把数据项分配在此。在图 7.3 中，有项数据需要 3 个块，可能会把它分配到区域 3 中。

[119] http://erlang.org/doc/man/erts_alloc.html#M_sbct
[120] http://erlang.org/doc/man/erts_alloc.html#M_mmsbc
[121] http://erlang.org/doc/man/erts_alloc.html#M_as

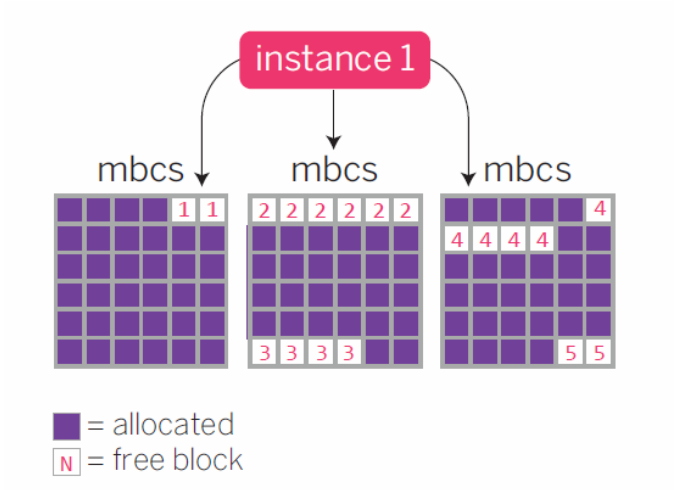


图 7.3：某个特定子分配器中的内存分配情况示例

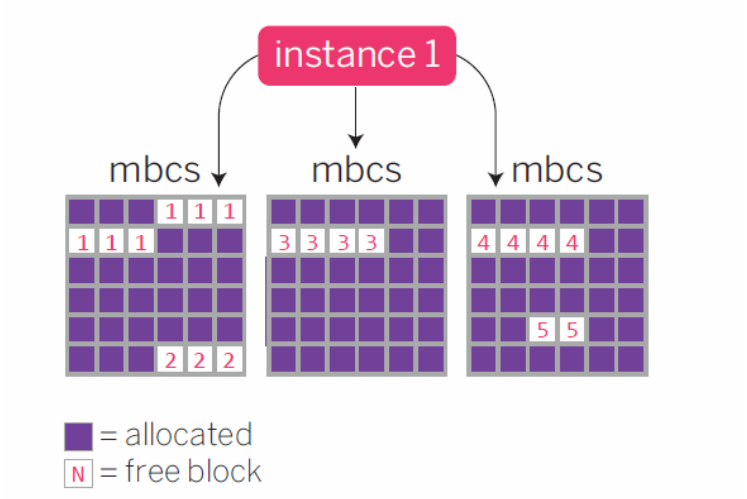


图 7.4：某个特定子分配器中的内存分配情况示例

地址序最佳匹配（aobf）的工作原理类似，不过树是基于块的地址的。因此，虽然 VM 也会去寻找能够容纳数据项的最小的块，不过当有多个大小一样的块同时满足要求时，会优先选择地址低的那一个。如果有一项数据需要 3 个块，那么仍然会选择区域 3，不过如果只需要 2 个块的话，这个策略会优先选择图 7.3 中第一个 mbcs 中的区域 1（而不是区域 5）。这会使 VM 倾向于在多次分配中使用同样的载体。

地址序优先匹配（aoff）会以地址序为优先进行搜索，只要有个块能够容纳，aoff 就会使用它。在图 7.3 中，在需要分配 4 个块时，aobf 和 bf 都会选择区域 3，而 aoff 则会优先选择区域 2，因为它的地址最小。在图 7.4 中，如果要分配 4 个块，会优选区域 1 而不是 3，因为它的地址比较小，而 bf 则会选择 3 或者 4，aobf 则会选择 3。

地址序优先载体最佳匹配（aoffcbf）策略会先选择能够容纳所需大小的载体，接着在其中选择一个最佳匹配。因此，如果要在图 7.4 中分配 2 个块，bf 和 aobf 都会选择区域 5，aoff 会选择区域 1，而 aoffcbf 则会选择区域 2，因为第一个 mbcs 可以能够容纳 2 个块，而区域 2 要比区域 1 有更好的匹配度。

地址序优先载体地址序最佳匹配（aoffcaobf）策略和 aoffcbf 类似，不过如果某个载体中有多个区域具有相同的大小，它会优先选择其中地址最小的那个，而不是随意选择一个。

良好匹配（gf）是一种不同类型的分配策略；它工作起来很像最佳匹配（bf），但是具有限定的搜索时间。如果在限定时间内没有找到完美的匹配，就会从已经搜索过的区域中选择一个最好的。这个值可以通过VM参数mbsd[122]进行配置。

最后一个是基本匹配（af），它是一种用于临时数据的分配策略，它会寻找单个已有的内存块，如果数据能够放进去，af 就使用它。否则，af 会分配一个新的。

这些策略中的每一个都可以单独用到每种分配器中，因此堆分配器和 binary 分配器可以使用不同分配策略。

最后，从Erlang 17.0 版本开始，每个调度器上的每个alloc_util分配器都有一个mbcs池。mbcs池用来防止VM中出现内存碎片的。当分配器的多块载体中的某一个快变空时[123]，该载体会被整个弃用。

后面所有的分配都不会使用这个被弃用的载体。在需要新的多块载体时，会先从mbcs池中获取。不同调度器间，只要alloc_util分配器的类型一样，可以共用池中载体。这种做法可以把基本上变空的载体缓存起来，而不用强制释放其内存[124]。它也使得在载体中的数据过少时，可以根据需要在调度器之间进行载体迁移。

进程级

在小一些的尺度上，对每个 Erlang 进程来说，布局情况有些不同。大体上，可以把进程内存想象成一个盒子，如下：

```
1 [                  ]
```

一端是堆，另外一端是栈：

```
1 [heap |      | stack]
```

实际上，还会有一些其他数据（用于分代型 GC 的老堆和新堆，以及特定的子分配器用来记录引用计数型 binary 大小的虚拟 binary 堆（进程不会使用它）——binary_alloc vs. eheap_alloc）：

```
1 [heap   ||    stack]
```

所分配的空间会越来越多，直到栈或者堆无法容纳为止。此时会触发一次小型的 GC。这个

[122] http://www.erlang.org/doc/man/erts_alloc.html#M_mbsd
[123] 门限值可以配置，参见：http://www.erlang.org/doc/man/erts_alloc.html#M_acul
[124] 如果它消耗的内存过多，可以通过选项+MBacul 0 来禁止这项功能

小型 GC 会把需要的数据移到老堆中。然后，回收其余的数据，最后也可能会重新分配一些空间。

经过一定数量的小型 GC 和重新分配，会执行一次全面的 GC，这次 GC 会详细检查老堆和新堆，释放更多的空间。当某个进程死亡了，栈和堆都会被立刻回收，binary 引用计数会减少，如果计数变成 0，这些 binay 也会被释放。

在 80%的情况下，进程死亡时只会发生一件事情，那就是其所用的内存会在子分配器中被标记为可用，会被新的进程或者其他变大了的进程使用。仅当这块内存没被使用——并且多块载体也没有被使用的情况下——才会被归还给 mseg_alloc 或者 sys_alloc，它们也许会再把内存多保存一段时间，也许会立即释放。

7.3.3 通过改变分配策略来修复泄漏

调整 VM 的内存分配选项也可能会解决我们的问题。

当然，需要对自己的内存负载和使用类型有一个良好的理解，并且要做好实施很多深入测试的准备。recon_alloc模块中包含了很多工具函数，可以用来提供指导，阅读一下模块的说明文档[125]也是很有必要的。

需要找出数据的平均大小是多少，分配和释放的频率是多少，数据是否能够放入 mbcs 或者 sbcs，然后需要实验一下所提到的 recon_alloc 的一系列选项，尝试一下不同的策略，部署它们，再看看是不是有所改善，或是变得更糟。

这是一个长期的过程，其中没有捷径，如果一个节点要几个月才出现一次问题，那么就要做好长期战斗的准备。

7.4 练习

复习题

1、 说出 Erlang 程序中的几个常见泄漏源
2、 Erlang 中的两种主要 binary 类型是什么？
3、 如果哪种数据类型都不像是泄漏源，那么可能是什么问题？
4、 如果某个节点崩溃了，其中有个进程占用了大量内存，如何找出这个进程？
5、 代码本身如何导致泄漏？
6、 如何确定垃圾回收是否耗费了很长时间？

[125] http://ferd.github.io/recon/recon_alloc.html

开放问题

1、 如何验证泄漏是由于忘记了杀死进程造成的，还是因为进程本身使用了太多的内存造成的？
2、 一个进程以 binary 模式打开了一个 150M 的文件，从中获取一点信息，然后把该信息保存在 ETS 表中。如果此时发现了 binary 内存泄露，那么如何做才能使节点上 binary 内存使用的最少？
3、 如何才能发现 ETS 表增长的太快？
4、 通过哪些步骤才能知道节点可能存在内存碎片？如果有人说是因为 NIF 或者驱动泄露内存造成的，如何证明这是错误的？
5、 如何知道某个具有大的邮箱（通过 message_queue_len）的进程，是内存泄漏了，还是根本没有处理新的消息？
6、 某个进程使用了大量内存，但是很少进行垃圾回收。如何解释这一点？
7、 何时需要更改节点上的内存分配策略？是优先选择调整配置呢？还是调整自己代码的编写方式？

动手题

1、对于任何一个你所知道或者正在维护的 Erlang 系统（包括玩具系统），你能知道其中是否有 binary 内存泄露吗？

第 8 章 CPU和调度器占用

虽然内存泄漏一定会杀死系统，CPU处理能力的枯竭则往往会造成瓶颈，限制了节点能完成的最大工作量。当出现此类的问题时，Erlang开发者往往会选择水平扩展的方法。对于比较基本的代码块，通常是比较容易扩展的。往往只有那些位于中心的全局状态（进程注册表、ETS表等等）需要修改[126]。当然，如果在选择水平扩展前，想先进行一下局部优化，那么就需要找到那些过多占用CPU和调度器的东西。

一般来讲，很难对 Erlang 节点中的 CPU 使用情况进行恰当的分析，并找出具体有问题的代码片段。一切都是并发的，并且是在虚拟机环境中，因此到底是某个进程、驱动、自己写的 Erlang 代码、自己安装的 NIF，还是某个第三方库耗尽了处理能力，是无法保证能够知道的。

如果是自己编写的代码的问题，那么现有的方法通常仅限于 profiling 和统计 reduction 数，如果问题出在其他地方（也是自己的代码），则也只能去监测调度器的工作情况。

8.1 Profiling和Reduction计数

为了能够定位出导致问题的具体Erlang代码片段，有两种主要的方法。一种方法就是采用传统的标准profiling例程，可以使用如下几个application[127]：

- eprof[128]，最古老的Erlang profiler。它会给出大体上的百分比值，报告中基本上只有花费的时间。
- fprof[129]，eprof的替代者，功能更强。它对并发有完全的支持，并会产生深度的报告。事实上，由于报告过于细节，以至于常常被认为晦涩、难懂。
- eflame[130]，最新的profiler工具。它会产生出一幅火焰图（flame graph），来展示给定代码中的深度调用序列和使用热点。通过它，一眼就能看出问题所在。

请读者自行去阅读每个 application 的文档说明。另外一种方法是调用 recon:proc_window/3，我在 5.2.1 小节中做过介绍：

[126] 通常可以采用分片的方式或者找到某种合适的状态复制模式即可。不过这不是一件容易的工作，不过和要解决程序的语义不适合于分布式系统的问题比起来，就不算什么了，Erlang 通常会强迫你在第一时间就解决这个问题。

[127] 所有这些 profiler 工具都无限制的使用了 Erlang 的跟踪功能。会对应用的运行性能产生影响，因此不要用在生产系统中。

[128] http://www.erlang.org/doc/man/eprof.html

[129] http://www.erlang.org/doc/man/fprof.html

[130] https://github.com/proger/eflame

```
1> recon:proc_window(reductions, 3, 500).
[{<0.46.0>,51728,
  [{current_function,{queue,in,2}},
   {initial_call,{erlang,apply,2}}]},
 {<0.49.0>,5728,
  [{current_function,{dict,new,0}},
   {initial_call,{erlang,apply,2}}]},
 {<0.43.0>,650,
  [{current_function,{timer,sleep,1}},
   {initial_call,{erlang,apply,2}}]}]
```

在 Erlang 中，reduction 数目和函数调用有直接的联系，高的数目通常意味着高的 CPU 占用。

关于这个函数有个有趣的用法，就是在系统忙时以短的时间间隔去调用它[131]。重复多次，你可能会发现有个模式浮现出来：相同的进程（或者同种进程）会一直处在前面。

通过代码定位[132]并找到当前正在运行的函数，应该能够定位出哪些代码在消耗着调度器。

8.2 系统监控器

如果通过 profiling 和检查 reduction 计数都没有发现什么异常情况，那么问题就可能出在 NIF、垃圾回收等中。这类工作不是总能正确地增加其 reduction 计数，因此前面的方法不会发现它们，只能基于“较长耗时”这个特征来找到它们。

要发现这些问题，最好的方法是使用erlang:system_monitor/2 去监控long_gc和long_schedule。前者会报告出工作量过大的垃圾回收（是要花时间的！），后者则会捕捉到由于使用了NIF或者其他使得进程难以释放调度器的做法，从而导致进程忙碌的问题[133]。

我们在 7.1.5 小节（垃圾回收）中已经介绍过如何设置这类系统监控器，不过在此会展示一种不同的用法[134]，我以前曾经用它来捕捉具有“较长耗时”特征的目标。

---

[131] 参见 5.1.2 小节

[132] 调用 recon:info(PidTerm, location)或者 process_info(Pid, current_stacktrace)来获取这个信息。

[133] 长的垃圾回收是被包括在调度时间之内的。根据具体的系统情况，大量的长时调度很可能会被绑定在垃圾回收上。

[134] 如果是 17.0 或者更新的版本，通过使用有名形式，可以更容易地在 shell 中定义递归函数，不过为了尽可能地和老 Erlang 版本兼容，就采用这种形式。

```
1> F = fun(F) ->
    receive
        {monitor, Pid, long_schedule, Info} ->
            io:format("monitor=long_schedule pid=~p info=~p~n", [Pid, Info]);
        {monitor, Pid, long_gc, Info} ->
            io:format("monitor=long_gc pid=~p info=~p~n", [Pid, Info])
    end,
    F(F)
end.
2> Setup = fun(Delay) -> fun() ->
     register(temp_sys_monitor, self()),
     erlang:system_monitor(self(), [{long_schedule, Delay}, {long_gc, Delay}]),
     F(F)
end end.
3> spawn_link(Setup(1000)).
<0.1293.0>
monitor=long_schedule pid=<0.54.0> info=[{timeout,1102},
                                         {in,{some_module,some_function,3}},
                                         {out,{some_module,some_function,3}}]
```

要确保把 long_schedule 和 long_gc 设置成大一些、比较合理的值。在本例中，它们被设置为 1000 毫秒。要想杀死监控器，可以调用 exit(whereis(temp_sys_monitor), kill)（会依次把 shell 杀死，因为它们链接在一起），或者断开和节点的连接即可（会杀死该进程，因为它和 shell 链接在一起）。

可以把这类代码和监控功能放到独立的模块中，把监控报告信息存为长期的日志，这些日志可以在性能降级或者过载检测时为我们提供有价值的情报。

8.2.1 挂起的端口

系统监控器中有个有趣的部分，是关于端口的，这个监控只和调度有点关系。当某个进程向一个端口发送了太多的消息，以至于撑满了端口的内部队列，Erlang 调度器会强迫发送者释放调度器直到空间被释放。这可能会让有些不期望 VM 偷偷地实施反压（back-pressure）的读者感到惊讶。

要监控这类事件，可以在系统监控器中使用 busy_port 参数。特别是对于集群节点，如果想知道何时一个本地进程在联系一个远程节点中的进程时，由于负责节点间通信的端口忙而被迫放弃调度权时，可以使用 busy_dist_port 参数。

如果发现存在此类问题，可以尝试把关键路径上的发送函数替换为使用 erlang:port_command(Port, Data, [nosuspend])（针对端口）或者 erlang:send(Pid, Msg, [nosuspend])（针对分布式进程间消息）。这样，当消息无法发送时，原本会因此而导致释放调度权，但是这两种方法则会通过返回值的方式通知调用者端口忙。

8.3 练习

复习题

1、 定位 CPU 占用方面问题的两种主要方法是什么？
2、 说出几种可用的 profiling 工具。哪种方法比较适合生产环境使用？为啥？
3、 为啥长运行时间监控能用来发现 CPU 或者调度器的过度消耗？

开放问题

1、 如果某个进程从 reduction 数来看做的工作很少，但是占用的调度时间很长，你觉得是啥问题？它运行的代码可能是啥？
2、 你能设置一个系统监控器，然后使用常规的 Erlang 代码触发它吗？你能用它来找出进程调度的平均时间吗？可能需要你在 shell 上手工随机启动一些工作更主动的进程（比起系统中提供的那些进程）。

第 9 章 跟踪

在 Erlang 和 BEAM 虚拟机中，比较少为人知以及绝对未被充分利用的功能之一就是其强大的跟踪功能。

忘记调试器吧，它们的能力太受限了[135]。在Erlang系统生命周期中的每个阶段，跟踪都是非常有用的，不管是处于开发阶段，还是需要对一个运行中的生产系统进行诊断。

跟踪 Erlang 程序，有几种选择：

- sys[136]，标准OTP自带，可以设置定制的跟踪函数，记录所有种类的事件等等。对于开发阶段来说，它是完备和适用的。不过，在生产环境中使用会有点问题，因为它没有把IO重定向到远程shell，也不具有对跟踪消息的限速能力。其说明文档还是值得推荐阅读一下。
- dbg[137]，也是标准Erlang/OTP自带。它的接口有些难用，不过提供的功能完全够用。它的问题在于，你必须非常清楚自己所做的事情的后果，因为它会把节点上的几乎所有信息都进行日志记录，因此可以瞬间杀死节点。
- erlang 模块中的跟踪 BIF 函数。上面提到的应用的功能都是基于这些更基本的函数实现的，它们的抽象层次太低，比较难以使用。
- redebug[138]可以安全地在生产环境中使用，它是eper[139]套件的一部分。它具有内置的限速机制，接口也易于使用。要使用它，必须得把eper的所有依赖都增加进来。这个工具套件非常全面，使用起来也令人愉快。
- recon_trace[140]是recon中处理跟踪的模块。它的目标是既具有和redebug同样级别的产品安全性，又不需要过多的依赖。接口和redebug的不同，限速选项也不完全一样。它只能跟踪函数调用，不能跟踪消息[141]。

本章主要讲解使用 recon_trace 进行跟踪，不过其中使用的术语和概念完全适用于其他的 Erlang 跟踪工具。

9.1 跟踪原则

Erlang的跟踪BIF可以跟踪任何Erlang代码[142]。它们的使用有两部分组成：pid规格说明，以及

---

[135] 那些可以设置断点，单步运行程序的调试器都一个共同的问题，它们和大部分 Erlang 程序不兼容：在一个进程中设置断点，而周围其他的进程还在运行。在实际中，这使得调试器的作用非常受限，因为当某个进程需要和一个正在调试的进程交互时，它的调用就超时或者崩溃，这可能会进而导致整个节点失效。而跟踪则不会干扰程序的执行，并且仍然能够提供需要的所有数据。

[136] http://www.erlang.org/doc/man/sys.html

[137] http://www.erlang.org/doc/man/dbg.html

[138] https://github.com/massemanet/eper/blob/master/doc/redbug.txt

[139] https://github.com/massemanet/eper

[140] http://ferd.github.io/recon/recon_trace.html

[141] 未来会考虑在库中增加对消息跟踪的支持。在实际中，使用 OTP 时很少会有消息跟踪的需要，因为 behaviour 以及对具体参数匹配的支持基本上可以完成同样的工作。

[142] 如果进程包含有敏感性息，可以调用 process_flag(sensitive, true)来强制数据私有。

跟踪模式。

pid 规格说明用来指定要跟踪的进程。可以是某些特定的 pid、所有 pid、现有的 pid，或者新创建的 pid（在调用时，还没有创建的进程）。

跟踪模式则用来指定函数。函数的指定分为两个部分：一部分是指定模块、函数以及参数个数，另外一部分是使用Erlang的匹配规格说明[143]来添加对参数的约束。

一个特定的函数调用是否会被跟踪，取决于这两者的交集。见图 9.1。

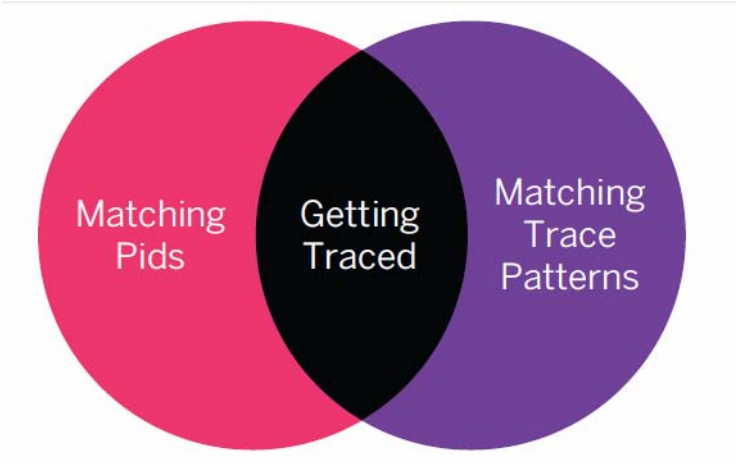


图 9.1：只有当 pid 和跟踪模式都匹配时，函数才会被跟踪。

如果进程不符合 pid 的规格说明，或者调用不符合跟踪模式，那么不会接收到任何跟踪消息。

在使用像 dbg（以及跟踪 BIF）之类工具时，必须时刻牢记这幅 Venn 图。Pid 的匹配规则集合和调用的跟踪规则集合是独立设置的，只有落入两者交集中的才会被跟踪。

而像 redebug 和 recon_trace 之类的工具，则对此进行了抽象。

9.2 使用Recon进行跟踪

在缺省情况下，Recon 会匹配所有的进程。在许多调试工作中，这种做法都不会有啥问题。在多大数情况下，真正需要设置的部分是跟踪模式的规格说明。Recon 支持几种声明它们的方法。

最基本的形式是{Mod, Fun, Arity}，其中 Mod 是字面的模块名，Fun 是函数名，Arity 是要跟踪的函数的参数个数。它们中的任一个可以用通配符（'_'）来替代。Recon 会禁止范围过宽的匹配（比如：{'_', '_', '_'}），因为在生产环境中使用是有危险的。

一种复杂点的形式是把arity替换成一个函数来匹配一组参数列表。这类函数必须和ETS[144]的

[143] http://www.erlang.org/doc/apps/erts/match_spec.html
[144] http://www.erlang.org/doc/man/ets.html#fun2ms-1

匹配规格说明中要求的类似。列表中可以放置多个模式以拓宽匹配的范围。

在限速方面，也有两种方式：静态计数，或者设定每个时间间隔内匹配的数量。

我们不再深入更多的细节，接下来看看一些例子和演示：

————————————————————————————————————

```
%% All calls from the queue module, with 10 calls printed at most:
recon_trace:calls({queue, '_', '_'}, 10)

%% All calls to lists:seq(A,B), with 100 calls printed at most:
recon_trace:calls({lists, seq, 2}, 100)

%% All calls to lists:seq(A,B), with 100 calls per second at most:
recon_trace:calls({lists, seq, 2}, {100, 1000})

%% All calls to lists:seq(A,B,2) (all sequences increasing by two) with 100 calls
%% at most:
recon_trace:calls({lists, seq, fun([_,_,2]) -> ok end}, 100)

%% All calls to iolist_to_binary/1 made with a binary as an argument already
%% (a kind of tracking for useless conversions):
recon_trace:calls({erlang, iolist_to_binary,
fun([X]) when is_binary(X) -> ok end},
10)

%% Calls to the queue module only in a given process Pid, at a rate of 50 per
%% second at most:
recon_trace:calls({queue, '_', '_'}, {50,1000}, [{pid, Pid}])

%% Print the traces with the function arity instead of literal arguments:
recon_trace:calls(TSpec, Max, [{args, arity}])

%% Matching the filter/2 functions of both dict and lists modules, across new
%% processes only:
recon_trace:calls([{dict,filter,2},{lists,filter,2}], 10, [{pid, new]})

%% Tracing the handle_call/3 functions of a given module for all new processes,
%% and those of an existing one registered with gproc:
recon_trace:calls({Mod,handle_call,3}, {1,100}, [{pid, [{via, gproc, Name}, new]}

%% Show the result of a given function call, the important bit being the
%% return_trace() call or the {return_trace} match spec value.
recon_trace:calls({Mod,Fun,fun(_) -> return_trace() end}, Max, Opts)
recon_trace:calls({Mod,Fun,[{'_', [], [{return_trace}]}]}, Max, Opts)
```

每个新的调用都会覆盖掉前一个调用，可以使用 recon_trace:clear/0 来取消所有的调用。

还有其他一些选项可供组合使用：

{pid, PidSepc}

指定要跟踪的进程。有效的值为 all、new、existing 或者进程描述符（{A,B,C}, “<A.B.C>”、表示进程名字的原子、{global, Name}、{via, Registrar, Name}或者 pid）。也支持指定多个，可以把它们放到列表中。

{timestamp, format | trace}

缺省情况下，formatter 进程会在收到的消息中增加时间戳。如果需要精确的时间戳，可以通过选项{timestamp, trace}强制直接在跟踪消息中使用时间戳。

{args, arity | args}

是打印出函数调用的参数个数，还是打印出其字面表示（缺省）。

{scope, global | local|}

缺省情况下，只有’global’（全称函数调用）的才会被跟踪，不会跟踪模块的内部调用。要想强制跟踪内部调用，可以传入{scope, local}选项。当想跟踪进程中以 Fun(Args) 而不是 Module:Fun(Args)的方式调用的代码时，这个选项很有用。

使用这些选项，可以针对特定函数的特定调用，甚至是不太好描述的情况，进行多种方式的模式匹配，可以更快地诊断开发和生产系统中的问题。在你想说“嗯，也许需要在这里多加点日志信息，看看是什么导致了这个奇怪的行为”时，跟踪常常是一种获取所需数据的快捷方法，无需部署任何代码，也不会对代码的可读性提出警告。

9.3 实例演示

首先，我们来跟踪被任一个进程调用的 queue:new 函数：

```
1> recon_trace:calls({queue, new, '_'}, 1).
1
13:14:34.086078 <0.44.0> queue:new()
Recon tracer rate limit tripped.
```

我们把限制设置为 1，也就是最多跟踪一次，在达到这个限制时，recon 会通知我们。

接下来，我们来跟踪所有的 queue:in/2 调用，看看我们都在队列中插入了什么东西：

```
2> recon_trace:calls({queue, in, 2}, 1).
1
13:14:55.365157 <0.44.0> queue:in(a, {[],[]})
Recon tracer rate limit tripped.
```

为了能够看到想要的内容，需要把跟踪模式更改成一个 fun，用它来匹配整个参数列表（_），并返回 return_trace()。return_trace()会产生针对每个调用的再次跟踪，其中会包括返回值：

```
3> recon_trace:calls({queue, in, fun(_) -> return_trace() end}, 3).
1

13:15:27.655132 <0.44.0> queue:in(a, {[],[]})

13:15:27.655467 <0.44.0> queue:in/2 --> {[a],[]}

13:15:27.757921 <0.44.0> queue:in(a, {[],[]})
Recon tracer rate limit tripped.
```

可以使用更复杂的方式来匹配参数列表：

```
4> recon_trace:calls(
4>   {queue, '_',
4>    fun([A,_]) when is_list(A); is_integer(A) andalso A > 1 ->
4>        return_trace()
4>    end},
4>   {10,100}
4> ).
32

13:24:21.324309 <0.38.0> queue:in(3, {[],[]})

13:24:21.371473 <0.38.0> queue:in/2 --> {[3],[]}

13:25:14.694865 <0.53.0> queue:split(4, {[10,9,8,7],[1,2,3,4,5,6]})

13:25:14.695194 <0.53.0> queue:split/2 --> {{[4,3,2],[1]},{[10,9,8,7],[5,6]}}

5> recon_trace:clear().
ok
```

请注意，在上面的模式中，并没有指定要匹配的函数（'_'）。而是使用 fun 来限制函数必须有两个参数，第一个参数要么是个 list，要么是个大于 1 的整数。

要小心那些没有严格限速（或者使用了绝对意义上很高的值）且匹配过于宽泛的模式，它们会对节点的稳定性造成影响，recon_trace 对此也无能为力。同样地，如果跟踪的函数调用数量过于庞大（比如，所有函数调用，或者所有 io 调用），产生跟踪消息的速度超出了节点上任何进程的处理速度的话，也是非常危险的，当然故意为之的情况除外。

如有疑问，先选择最具限制性的跟踪方式，使用低的门限，然后逐步增加范围。

9.4 练习

复习题

1、 为什么调试器的作用在 Erlang 中有限？
2、 跟踪 OTP 进程有哪些选择？
3、 进程和函数被跟踪的决定因素是什么？

4、 在使用 recon_trace 时，如何停止跟踪？其他工具呢？
5、 如何跟踪没有输出（non-exported）的函数调用？

开放问题

1、 何时会需要把跟踪的时间戳移到 VM 的跟踪机制中？这样做有什么缺点？
2、 假设节点在向外发送信息时使用了一个多租户（multi-tenant）系统的 SSL 通道。然而，由于想对发出的数据进行检验（有客户抱怨问题），需要能看到明文信息。你能想出一个方法使得既可以窥探到通过 SSL socket 发送给该客户的数据，又不会泄漏发送给其他客户的数据吗？

动手题

使用https://github.com/ferd/recon_demo处的代码（需要对代码有良好的理解）：

1、 聊天进程（conucil_member）能给自己发消息吗？（提示：如果是使用注册名能行吗？需要去检查最爱聊天的进程，看看它是否给自己发送了消息吗？）
2、 估算一下消息发送的整体上大致频率。
3、 你能用任意一个跟踪工具让一个节点崩溃吗？（提示：dbg 灵活性更高，因此更容易做到这点）

结论

软件的维护和调试工作永远不会停止。新的 bug 和难以理解的行为会随时随地跳到你的面前。即使对于最干净、整洁的系统来说，所出现的问题也会装满几十本像本书这样的册子。

我希望在阅读本书后，下次碰到问题时，情况不会太糟糕。不过，我们还是需要经常去调试生产系统。即使最坚固的桥梁也需要经常重新粉刷油漆，以避免因腐蚀而导致坍塌。

祝你好运。