过程

在 Elixir 中，所有代码都在进程内运行。进程彼此隔离，彼此并发运行，并通过消息传递进行通信。进程不仅是 Elixir 中并发的基础，而且还提供了构建分布式和容错程序的方法。

Elixir 的进程不应该与操作系统进程混淆。Elixir 中的进程在内存和 CPU 方面非常轻量级（甚至与许多其他编程语言中使用的线程相比）。因此，同时运行数万甚至数十万个进程的情况并不少见。

在本章中，我们将学习生成新进程的基本构造，以及在进程之间发送和接收消息。

spawn

生成新进程的基本机制是自动导入函数： `spawn/1`

```
iex> spawn(fn -> 1 + 2 end)
#PID<0.43.0>
```

`spawn/1` 获取一个将在另一个进程中执行的函数。

通知返回 PID（进程标识符）。此时，您生成的进程很可能已死。生成的进程将执行给定的函数，并在函数完成后退出： `spawn/1`

新闻： Elixir v1.15 发布

搜索。。。

接口文档

开始

1. 介绍
2. 基本类型
3. 基本运算符
4. 模式匹配
5. 案例、cond 和 if
6. 二进制文件、字符串和字符列表
7. 关键字列表和地图
8. 模块和功能
9. 递归
10. 枚举项和流
11. 过程
12. IO 和文件系统
13. 别名、要求和导入
14. 模块属性
15. 结构体
16. 协议
17. 理解
18. 印记
19. 尝试、捕捉和救援
20. 可选语法表
21. Erlang 库

```
iex> pid = spawn(fn -> 1 + 2 end)
#PID<0.44.0>
iex> Process.alive?(pid)
false
```

> *注意：您可能会获得与本指南中不同的进程标识符。*

我们可以通过调用来检索当前进程的 PID：`self/0`

```
iex> self()
#PID<0.41.0>
iex> Process.alive?(self())
true
```

当我们能够发送和接收消息时，流程变得更加有趣。

发送和接收

我们可以通过以下方式向进程发送消息并通过以下方式接收消息：`send/2` `receive/1`

```
iex> send(self(), {:hello, "world"})
{:hello, "world"}
iex> receive do
...>   {:hello, msg} -> msg
...>   {:world, _msg} -> "won't match"
...> end
"world"
```

将邮件发送到进程时，该邮件将存储在进程邮箱中。该块通过当前进程邮箱搜索与任何给定模式匹配的邮件。 支持守卫和许多子句，例如 .`receive/1` `receive/1` `case/2`

发送邮件的进程不会阻止，它会将邮件放入收件人的邮箱中并继续。特别是，进程可以向自身发送消息。`send/2`

如果邮箱中没有与任何模式匹配的邮件，则当前进程将等待，直到匹配的邮件到达。还可以指定超时：

```
iex> receive do
...>   {:hello, msg}  -> msg
...> after
...>   1_000 -> "nothing after 1s"
...> end
"nothing after 1s"
```

当您已经期望邮件在邮箱中时，可以给出 0 的超时。

让我们把它们放在一起，在进程之间发送消息：

```
iex> parent = self()
#PID<0.41.0>
iex> spawn(fn -> send(parent, {:hello, self()}) end)
#PID<0.48.0>
iex> receive do
...>   {:hello, pid} -> "Got hello from #{inspect pid}"
...> end
"Got hello from #PID<0.48.0>"
```

该函数用于将数据结构的内部表示形式转换为字符串，通常用于打印。请注意，当块被执行时，我们生成的发送方进程可能已经死了，因为它唯一的指令是发送消息。`inspect/1` `receive`

在 shell 中，您可能会发现帮助程序非常有用。它刷新并打印邮箱中的所有邮件。`flush/0`

```
iex> send(self(), :hello)
:hello
iex> flush()
:hello
:ok
```

链接

大多数时候，我们在 Elixir 中生成进程，我们将它们作为链接进程生成。在我们展示一个示例之前，让我们看看当一个进程以失败启动时会发生什么：`spawn_link/1` `spawn/1`

```
iex> spawn(fn -> raise "oops" end)
#PID<0.58.0>

[error] Process #PID<0.58.00> raised an exception
** (RuntimeError) oops
    (stdlib) erl_eval.erl:668: :erl_eval.do_apply/6
```

它只是记录了一个错误，但父进程仍在运行。这是因为进程是隔离的。如果我们希望一个进程中的失败传播到另一个进程，我们应该将它们链接起来。这可以通过以下方式完成：`spawn_link/1`

```
iex> self()
#PID<0.41.0>
iex> spawn_link(fn -> raise "oops" end)

** (EXIT from #PID<0.41.0>) evaluator process exited with
reason: an exception was raised:
    ** (RuntimeError) oops
        (stdlib) erl_eval.erl:668: :erl_eval.do_apply/6

[error] Process #PID<0.289.0> raised an exception
** (RuntimeError) oops
    (stdlib) erl_eval.erl:668: :erl_eval.do_apply/6
```

由于进程是链接的，我们现在看到一条消息，指出父进程（即 shell 进程）已从另一个进程接收到 EXIT 信号，导致 shell 终止。IEx 检测到这种情况并启动新的外壳会话。

也可以通过调用 手动完成链接。我们建议您查看流程模块，了解 流程 提供的其他功能。`Process.link/1`

在构建容错系统时，进程和链接起着重要作用。Elixir 进程是隔离的，默认情况下不共享任何内容。因此，一个进程中的失败永远不会崩溃或损坏另一个进程的状态。但是，链接允许流程在发生故障时建立关系。我们经常将我们的流程与主管联系起来，主管将检测流程何时死亡并在其位置启动新流程。

虽然其他语言会要求我们捕获 / 处理异常，但在 Elixir 中，我们实际上可以让进程失败，因为我们希望主管正确地重新启动我们的系统。“快速失败”（有时称为“让它崩溃”）是编写 Elixir 软件时的常见哲学！

`spawn/1` 并且是在 Elixir 中创建过程的基本原语。尽管到目前为止我们只使用它们，但大多数时候我们将使用基于它们构建的抽象。让我们看看最常见的一种，称为任务。 `spawn_link/1`

任务

任务建立在生成函数之上，以提供更好的错误报告和内省：

```
iex> Task.start(fn -> raise "oops" end)
{:ok, #PID<0.55.0>}

15:22:33.046 [error] Task #PID<0.55.0> started from
#PID<0.53.0> terminating
** (RuntimeError) oops
    (stdlib) erl_eval.erl:668: :erl_eval.do_apply/6
    (elixir) lib/task/supervised.ex:85:
Task.Supervised.do_apply/2
    (stdlib) proc_lib.erl:247: :proc_lib.init_p_do_apply/3
Function: #Function<20.99386804/0 in :erl_eval.expr/5>
    Args: []
```

代替 和 ，我们使用 和 返回而不仅仅是 PID。这就是使任务能够在监督树中使用的原因。此外，还提供方便的功能，如和，以及便于分发的功能。 `spawn/1` `spawn_link/1` `Task.start/1` `Task.start_link/1` `{:ok, pid}` `Task` `Task.async/1` `Task.await/1`

我们将在 *Mix 和 OTP 指南*中探讨这些功能，现在记住使用这些功能即可获得更好的错误报告。 `Task`

州

到目前为止，我们还没有在本指南中讨论状态。如果要构建需要状态的应用程序（例如，保留应用程序配置），或者需要分析文件并将其保存在内存中，则将其存储在哪里？

流程是这个问题最常见的答案。我们可以编写无限循环、维护状态以及发送和接收消息的进程。例如，让我们编写一个模块，该模块启动新进程，这些进程在名为 的文件中用作键值存储： `kv.exs`

```
defmodule KV do
  def start_link do
    Task.start_link(fn -> loop(%{}) end)
  end

  defp loop(map) do
    receive do
      {:get, key, caller} ->
        send caller, Map.get(map, key)
        loop(map)
      {:put, key, value} ->
        loop(Map.put(map, key, value))
    end
  end
end
```

请注意，该函数启动运行该函数的新进程，从空映射开始。然后，（私有）函数等待消息并对每条消息执行适当的操作。我们通过使用而不是。如果是消息，它会将消息发送回调用方并再次调用，以等待新消息。虽然消息实际上是使用新版本的地图调用的，但给定和存储。
`start_link` `loop/1` `loop/1` `loop/1` `defp` `def` `:get` `loop/1` `:put` `loop/1` `key` `value`

让我们通过运行来尝试一下：`iex kv.exs`

```
iex> {:ok, pid} = KV.start_link()
{:ok, #PID<0.62.0>}
iex> send(pid, {:get, :hello, self()})
{:get, :hello, #PID<0.41.0>}
iex> flush()
nil
:ok
```

起初，流程图没有键，因此发送消息然后刷新当前流程收件箱会返回 。让我们发送一条消息并重试：`:get` `nil` `:put`

```
iex> send(pid, {:put, :hello, :world})
{:put, :hello, :world}
iex> send(pid, {:get, :hello, self()})
{:get, :hello, #PID<0.41.0>}
iex> flush()
```

```
:world
:ok
```

请注意进程如何保持状态，我们可以通过发送进程消息来获取和更新此状态。事实上，任何知道上述情况的进程都能够向其发送消息并操纵状态。`pid`

也可以注册，给它一个名字，并允许知道这个名字的每个人都向它发送消息：`pid`

```
iex> Process.register(pid, :kv)
true
iex> send(:kv, {:get, :hello, self()})
{:get, :hello, #PID<0.41.0>}
iex> flush()
:world
:ok
```

使用进程来维护状态和名称注册是Elixir应用程序中非常常见的模式。但是，大多数时候，我们不会像上面那样手动实现这些模式，而是通过使用Elixir附带的众多抽象之一来实现。例如，Elixir提供了代理，它们是围绕状态的简单抽象：

```
iex> {:ok, pid} = Agent.start_link(fn -> %{} end)
{:ok, #PID<0.72.0>}
iex> Agent.update(pid, fn map -> Map.put(map, :hello,
:world) end)
:ok
iex> Agent.get(pid, fn map -> Map.get(map, :hello) end)
:world
```

也可以提供一个选项，它将自动注册。除了代理之外，Elixir还提供了一个API，用于构建通用服务器（称为），任务等，所有这些都由下面的进程提供支持。这些以及监督树将在*混合和OTP*指南中更详细地探讨，该指南将从头到尾构建一个完整的Elixir应用程序。`:name` `Agent.start_link/2` `GenServer`

现在，让我们继续探索 Elixir 中的 I / O 世界。

有什么不对吗？ 在 GitHub 上编辑此页面。