开始

尝试、捕捉和救援

Elixir 有三种错误机制：错误，抛出和退出。在本章中，我们将探讨它们中的每一个，并包括有关何时应该使用它们的注释。

错误

当代码中发生异常情况时，将使用错误（或*异常*）。可以通过尝试向原子添加数字来检索示例错误：

```
iex> :foo + 1
** (ArithmeticError) bad argument in arithmetic expression
    :erlang.+(:foo, 1)
```

可以使用以下命令随时引发运行时错误： `raise/1`

```
iex> raise "oops"
** (RuntimeError) oops
```

传递错误名称和关键字参数列表可能会引发其他错误： `raise/2`

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> raise ArgumentError, message: "invalid argument foo"
** (ArgumentError) invalid argument foo
```

您还可以通过创建模块并使用其中的构造来定义自己的错误。这样，你将创建一个与在其中定义的模块同名的错误。最常见的情况是使用消息字段定义自定义异常：`defexception`

```
iex> defmodule MyError do
iex>   defexception message: "default message"
iex> end
iex> raise MyError
** (MyError) default message
iex> raise MyError, message: "custom message"
** (MyError) custom message
```

可以使用以下构造来拯救错误：`try/rescue`

```
iex> try do
...>   raise "oops"
...> rescue
...>   e in RuntimeError -> e
...> end
%RuntimeError{message: "oops"}
```

上面的示例拯救了运行时错误并返回异常本身，然后在会话中打印该异常。`iex`

如果对异常没有任何用处，则不必将变量传递给：`rescue`

```
iex> try do
...>   raise "oops"
...> rescue
...>   RuntimeError -> "Error!"
...> end
"Error!"
```

在实践中，Elixir 开发人员很少使用该结构。例如，当文件无法成功打开时，许多语言会强制您挽救错误。相反，Elixir 提供了一个函数，该函数

返回一个元组，其中包含有关文件是否已成功打开的信息：`try/rescue` `File.read/1`

```
iex> File.read("hello")
{:error, :enoent}
iex> File.write("hello", "world")
:ok
iex> File.read("hello")
{:ok, "world"}
```

这里没有。如果要处理打开文件的多个结果，可以使用以下构造使用模式匹配：`try/rescue` `case`

```
iex> case File.read("hello") do
...>   {:ok, body} -> IO.puts("Success: #{body}")
...>   {:error, reason} -> IO.puts("Error: #{reason}")
...> end
```

对于您希望文件存在的情况（并且缺少该文件确实是一个*错误*），您可以使用：`File.read!/1`

```
iex> File.read!("unknown")
** (File.Error) could not read file "unknown": no such
file or directory
    (elixir) lib/file.ex:272: File.read!/1
```

归根结底，由您的应用程序决定打开文件时的错误是否异常。这就是为什么 Elixir 不会对许多其他功能施加例外。相反，它让开发人员选择最佳方式进行。`File.read/1`

标准库中的许多函数都遵循这样的模式：让对应函数引发异常，而不是返回要匹配的元组。约定是创建一个返回或元组的函数 （） 和另一个函数（，名称相同但带有尾随），该函数采用与参数相同的参数，但如果出现错误，则会引发异常。 如果一切正常，应该返回结果（不包装在元组中）。`File` 模块是此约定的一个很好的例子。`foo` `{:ok, result}` `{:error, reason}` `foo!` `!` `foo` `foo!`

快速失败 / 让它崩溃

在 Erlang 社区和 Elixir 社区中常见的一种说法是“快速失败”/“让它崩溃”。让它崩溃背后的想法是，万一发生*意外*，最好让异常发生，而不是挽救它。

强调*意外*这个词很重要。例如，假设您正在构建一个脚本来处理文件。脚本接收文件名作为输入。预计用户可能会犯错误并提供未知的文件名。在这种情况下，虽然您可以使用 读取文件并在文件名无效的情况下让它崩溃，但使用并向脚本用户提供关于问题所在问题的清晰而精确的反馈可能更有意义。`File.read!/1` `File.read/1`

其他时候，您可能完全期望某个文件存在，如果它不存在，则意味着其他地方发生了严重的错误。在这种情况下，就是您所需要的。
`File.read!/1`

第二种方法也有效，因为正如进程一章所讨论的，所有Elixir代码都在隔离的进程内运行，默认情况下不共享任何内容。因此，一个进程中未处理的异常永远不会崩溃或损坏另一个进程的状态。这使我们能够定义主管流程，这些流程旨在观察流程何时意外终止，并在其位置启动一个新流程。

归根结底，“快速失败”/“让它崩溃”是一种说法，当*发生意外*情况时，最好在一个新的流程中从头开始，由主管重新开始，而不是盲目地试图挽救所有可能的错误案例，而没有完整的上下文何时以及如何发生它们。

重新筹集

虽然我们通常避免在 Elixir 中使用，但我们可能想要使用这种结构的一种情况是用于可观察性 / 监控。想象一下，你想记录出了什么问题，你可以做：`try/rescue`

```
try do
  ... some code ...
rescue
  e ->
    Logger.error(Exception.format(:error, e,
__STACKTRACE__))
    reraise e, __STACKTRACE__
end
```

在上面的示例中，我们拯救了异常，记录了它，然后重新引发它。我们在格式化异常和重新引发时都使用该结构。这可确保我们按原样重新引发异常，而不会更改值或其来源。`__STACKTRACE__`

一般来说，我们从字面上理解Elixir中的错误：它们是为意外和/或特殊情况保留的，而不是为了控制我们的代码流。如果您确实需要流控制构造，则应使用*抛出*。这就是我们接下来要看到的。

抛出

在 Elixir 中，可以抛出一个值，然后被捕获。 并且保留用于无法检索值的情况，除非使用 和 。`throw` `catch` `throw` `catch`

这些情况在实践中非常罕见，除非与不提供适当 API 的库接口。例如，假设模块没有提供任何用于查找值的 API，并且我们需要在数字列表中找到 13 的第一个倍数：`Enum`

```
iex> try do
...>   Enum.each(-50..50, fn x ->
...>     if rem(x, 13) == 0, do: throw(x)
...>   end)
...>   "Got nothing"
...> catch
...>   x -> "Got #{x}"
...> end
"Got -39"
```

由于*确实*提供了适当的API，因此在实践中是要走的路：`Enum` `Enum.find/2`

```
iex> Enum.find(-50..50, &(rem(&1, 13) == 0))
-39
```

出口

所有 Elixir 代码都在相互通信的进程内运行。当一个进程因“自然原因”（例如，未处理的异常）而死亡时，它会发送一个信号。进程也可以通过显式发送信号而死亡：`exit` `exit`

```
iex> spawn_link(fn -> exit(1) end)
** (EXIT from #PID<0.56.0>) shell process exited with
reason: 1
```

在上面的例子中，链接的进程通过发送值为 1 的信号而死亡。Elixir shell 会自动处理这些消息并将它们打印到终端。`exit`

`exit` 也可以使用以下方法“捕获”：`try/catch`

```
iex> try do
...>   exit("I am exiting")
...> catch
...>   :exit, _ -> "not really"
...> end
"not really"
```

使用已经不常见，用它来捕捉出口的情况就更是少见了。`try/catch`

`exit` 信号是 Erlang VM 提供的容错系统的重要组成部分。进程通常在监督树下运行，监督树本身就是侦听来自受监督进程的信号的进程。一旦收到信号，监督策略就会启动，监督过程就会重新启动。`exit` `exit`

正是这种监督系统使结构在 Elixir 中如此罕见。与其挽救错误，我们宁愿“快速失败”，因为监督树将保证我们的应用程序在错误后返回到已知的初始状态。`try/catch` `try/rescue`

后

有时，有必要确保在执行某些可能引发错误的操作后清理资源。该构造允许您执行此操作。例如，我们可以打开一个文件并使用子句来关闭它 —— 即使出现问题：`try/after` `after`

```
iex> {:ok, file} = File.open("sample", [:utf8, :write])
iex> try do
...>   IO.write(file, "olá")
...>   raise "oops, something went wrong"
...> after
...>   File.close(file)
...> end
** (RuntimeError) oops, something went wrong
```

无论尝试的块是否成功，都将执行该子句。但请注意，如果链接进程退出，此过程将退出，并且子句将不会运行。因此仅提供软保证。幸运的是，Elixir 中的文件也链接到当前进程，因此如果当前进程崩溃，它们将

始终被关闭，与子句无关。您会发现其他资源（如 ETS 表、套接字、端口等）也是如此。`after` `after` `after` `after`

有时您可能希望将函数的整个主体包装在构造中，通常是为了保证之后将执行某些代码。在这种情况下，Elixir 允许您省略以下行：`try` `try`

```
iex> defmodule RunAfter do
...>   def without_even_trying do
...>     raise "oops"
...>   after
...>     IO.puts "cleaning up!"
...>   end
...> end
iex> RunAfter.without_even_trying
cleaning up!
** (RuntimeError) oops
```

Elixir 会自动将函数体包装在指定或或之一时。
`try` `after` `rescue` `catch`

还

如果存在一个块，只要块在没有抛出或错误的情况下完成，它就会与块的结果匹配。`else` `try` `try`

```
iex> x = 2
2
iex> try do
...>   1 / x
...> rescue
...>   ArithmeticError ->
...>     :infinity
...> else
...>   y when y < 1 and y > -1 ->
...>     :small
...>   _ ->
...>     :large
...> end
:small
```

不会捕获块中的异常。如果块内没有模式匹配，将引发异常；当前块不会捕获此异常。 `else` `else` `try/catch/rescue/after`

变量范围

与 Elixir 中的 、 和其他构造类似，块内定义的变量不会泄漏到外部上下文。换句话说，此代码无效： `case` `cond` `if` `try/catch/rescue/after`

```
iex> try do
...>   raise "fail"
...>   what_happened = :did_not_raise
...> rescue
...>   _ -> what_happened = :rescued
...> end
iex> what_happened
** (CompileError) undefined variable "what_happened"
```

相反，您应该返回表达式的值： `try`

```
iex> what_happened =
...>   try do
...>     raise "fail"
...>     :did_not_raise
...>   rescue
...>     _ -> :rescued
...>   end
iex> what_happened
:rescued
```

此外，在 do-block 中定义的变量在内部也不可用。这是因为块可能随时失败，因此变量可能从未被绑定过。所以这也是无效的： `try` `rescue/after/else` `try`

```
iex> try do
...>   raise "fail"
...>   another_what_happened = :did_not_raise
...> rescue
...>   _ -> another_what_happened
...> end
```

```
** (CompileError) undefined variable
"another_what_happened"
```

我们对 、 和 的介绍到此结束。你会发现它们在 Elixir 中的使用频率低于其他语言。 `try` `catch` `rescue`

← 上一页　返回页首　下一→