开始

IO 和文件系统

本章介绍输入/输出机制、文件系统相关任务以及相关模块（如 IO、File 和 Path）。IO系统提供了一个很好的机会来阐明Elixir和VM的一些哲学和好奇心。

IO 模块

IO模块是Elixir中用于读取和写入标准输入/输出（），标准错误（），文件和其他 IO 设备的主要机制。该模块的使用非常简单： :stdio :stderr

```
iex> IO.puts("hello world")
hello world
:ok
iex> IO.gets("yes or no? ")
yes or no? yes
"yes\n"
```

默认情况下，模块中的函数从标准输入读取并写入标准输出。例如，我们可以通过传递作为参数来更改它（以便写入标准错误设备）：
IO :stderr

```
iex> IO.puts(:stderr, "hello world")
hello world
```

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

```
:ok
```

文件模块

文件模块包含允许我们将文件作为IO设备打开的功能。默认情况下，文件以二进制模式打开，这需要开发人员使用模块中的特定 and 函数：`IO.binread/2` `IO.binwrite/2` `IO`

```
iex> {:ok, file} = File.open("path/to/file/hello",
[:write])
{:ok, #PID<0.47.0>}
iex> IO.binwrite(file, "world")
:ok
iex> File.close(file)
:ok
iex> File.read("path/to/file/hello")
{:ok, "world"}
```

文件也可以使用编码打开，编码告诉模块将从文件中读取的字节解释为 UTF-8 编码的字节。`:utf8` `File`

除了打开、读取和写入文件的功能外，该模块还具有许多与文件系统配合使用的功能。这些函数以其 UNIX 等效项命名。例如，可用于删除文件、创建目录、创建目录及其所有父链。甚至还有递归地分别复制和删除文件和目录（即，也复制和删除目录的内容）。
`File` `File.rm/1` `File.mkdir/1` `File.mkdir_p/1` `File.cp_r/2` `File.rm_rf/1`

您还会注意到模块中的函数有两个变体：一个是 "常规" 变体，另一个是带有尾随爆炸 （） 的变体。例如，当我们读取上面示例中的文件时，我们使用。或者，我们可以使用：
`File` `!` `"hello"` `File.read/1` `File.read!/1`

```
iex> File.read("path/to/file/hello")
{:ok, "world"}
iex> File.read!("path/to/file/hello")
"world"
iex> File.read("path/to/file/unknown")
{:error, :enoent}
iex> File.read!("path/to/file/unknown")
```

```
** (File.Error) could not read file
"path/to/file/unknown": no such file or directory
```

请注意，带有 的版本返回文件的内容而不是元组，如果出现任何问题，函数将引发错误。 `!`

当您想要使用模式匹配处理不同的结果时，首选不带的版本： `!`

```
case File.read("path/to/file/hello") do
  {:ok, body} -> # do something with the `body`
  {:error, reason} -> # handle the error caused by
`reason`
end
```

但是，如果您希望文件在那里，则 bang 变体更有用，因为它会引发有意义的错误消息。避免写：

```
{:ok, body} = File.read("path/to/file/unknown")
```

因为，如果出现错误，将返回并且模式匹配将失败。您仍然会得到所需的结果（引发的错误），但消息将是关于不匹配的模式（因此对于错误的实际内容是神秘的）。 `File.read/1` `{:error, reason}`

因此，如果不想处理错误结果，则首选使用以感叹号结尾的函数，例如 。 `File.read!/1`

路径 模块

模块中的大多数函数都需要路径作为参数。最常见的是，这些路径将是常规二进制文件。路径模块提供了处理此类 路径 的工具： `File`

```
iex> Path.join("foo", "bar")
"foo/bar"
iex> Path.expand("~/hello")
"/Users/jose/hello"
```

最好使用模块中的函数而不是直接操作字符串，因为模块透明地处理不同的操作系统。最后，请记住，Elixir 在执行文件操作时会自动在 Windows 上将斜杠（）转换为反斜杠（）。 `Path` `Path` `/` `\`

有了这个，我们已经介绍了Elixir提供的用于处理IO和与文件系统交互的主要模块。在下一节中，我们将深入了解一下，并了解如何在 VM 中实现 IO 系统。

过程

您可能已经注意到，它返回了一个元组，如下所示：`File.open/2` `{:ok, pid}`

```
iex> {:ok, file} = File.open("hello", [:write])
{:ok, #PID<0.47.0>}
```

发生这种情况是因为模块实际上与进程一起工作（参见第 11 章）。给定一个文件是一个进程，当你写入一个已经关闭的文件时，你实际上是在向一个已经终止的进程发送一条消息：`IO`

```
iex> File.close(file)
:ok
iex> IO.write(file, "is anybody out there")
** (ErlangError) Erlang error: :terminated:

  * 1st argument: the device has terminated

    (stdlib 5.0) io.erl:94: :io.put_chars(#PID<0.114.0>,
"is anybody out there")
    iex:4: (file)
```

让我们更详细地看看当您请求时会发生什么。该模块向具有所需操作标识的进程发送消息。一个小的临时过程可以帮助我们看到它：`IO.write(pid, binary)` `IO` `pid`

```
iex> pid = spawn(fn ->
...>  receive do: (msg -> IO.inspect msg)
...> end)
#PID<0.57.0>
iex> IO.write(pid, "hello")
{:io_request, #PID<0.41.0>, #Reference<0.0.8.91>,
 {:put_chars, :unicode, "hello"}}
** (ErlangError) erlang error: :terminated
```

之后，我们可以看到模块发送的请求打印出来（一个四元素元组）。不久之后，我们看到它失败了，因为模块期望某种结果，而我们没有提供。`IO.write/2` `IO` `IO`

通过使用进程对 IO 设备进行建模，Erlang VM 允许在运行分布式 Erlang 的不同节点之间路由 IO 消息，甚至交换文件以跨节点执行读/写操作。整洁！

IODATA 和 Chardata

在上面的所有示例中，我们在写入文件时使用二进制文件。然而，Elixir 中的大多数 IO 函数也接受 “iodata” 或 “chardata”。

使用 “iodata” 和 “chardata” 的主要原因之一是为了性能。例如 想象一下，您需要在应用程序中问候某人：

```
name = "Mary"
IO.puts("Hello " <> name <> "!")
```

给定 Elixir 中的字符串是不可变的，就像大多数数据结构一样，上面的示例会将字符串 “Mary” 复制到新的 “Hello Mary！” 字符串中。虽然这对于上述短字符串不太可能重要，但对于大字符串来说，复制可能非常昂贵！出于这个原因，Elixir 中的 IO 函数允许你传递一个字符串列表：

```
name = "Mary"
IO.puts(["Hello ", name, "!"])
```

在上面的示例中，没有复制。相反，我们创建一个包含原始名称的列表。我们将此类列表称为 “iodata” 或 “chardata”，我们将很快了解它们之间的确切区别。

这些列表非常有用，因为它实际上可以在多种情况下简化处理字符串。例如，假设您有一个值列表，例如要写入磁盘的值，以逗号分隔。你怎么能做到这一点？`["apple", "banana", "lemon"]`

一种选择是使用值并将其转换为字符串：`Enum.join/2`

```
iex> Enum.join(["apple", "banana", "lemon"], ",")
"apple,banana,lemon"
```

上面通过将每个值复制到新字符串中来返回一个新字符串。但是，根据本节中的知识，我们知道我们可以将字符串列表传递给 IO/File 函数。因此，我们可以执行以下操作：

```
iex> Enum.intersperse(["apple", "banana", "lemon"], ",")
["apple", ",", "banana", ",", "lemon"]
```

“iodata” 和 “chardata” 不仅包含字符串，而且还可能包含任意嵌套的字符串列表：

```
iex> IO.puts(["apple", [",", "banana", [",", "lemon"]]])
```

“iodata” 和 “chardata” 也可以包含整数。例如，我们可以使用 as 分隔符打印逗号分隔的值列表，它是表示逗号 （） 的整数：`?,` `44`

```
iex> IO.puts(["apple", ?,, "banana", ?,, "lemon"])
```

“iodata” 和 “chardata” 之间的区别正是所述整数所代表的。对于 iodata，整数表示字节。对于 chardata，整数表示 Unicode 代码点。对于 ASCII 字符，字节表示形式与代码点表示形式相同，因此它符合这两种分类。但是，默认的 IO 设备适用于 chardata，这意味着我们可以执行以下操作：

```
iex> IO.puts([?O, ?l, ?á, ?\s, "Mary", ?!])
```

总体而言，列表中的整数可能表示一堆字节或一堆字符，使用哪一个取决于 IO 设备的编码。如果打开文件而不进行编码，则文件应处于原始模式，并且必须使用模块中以 开头的函数。这些函数需要 a 作为参数，其中列表中的整数表示字节。`IO` `bin*` `iodata`

另一方面，默认的 IO 设备 （） 和用编码打开的文件与模块中的其余函数一起使用。这些函数需要 a 作为参数，其中整数表示代码点。`:stdio` `:utf8` `IO` `chardata`

尽管这是一个细微的区别，但如果您打算将包含整数的列表传递给这些函数，则只需担心这些细节。如果您传递二进制文件或二进制文件列表，则没有歧义。

最后，还有最后一个结构称为 charlist，它是 chardata 的一个特例，我们有一个列表，其中它的所有值都是代表 Unicode 代码点的整数。它们可以使用符号创建：`~c`

```
iex> ~c"hello"
~c"hello"
```

> *注意：以上内容在 Elixir v1.14 及更早版本中打印为 “hello”，这是字符列表的已弃用语法。*

它们主要在与 Erlang 接口时出现，因为一些 Erlang API 使用 charlist 作为字符串的表示形式。因此，任何包含可打印 ASCII 代码点的列表都将打印为字符列表：

```
iex> [?a, ?b, ?c]
~c"abc"
```

我们在这个小节中打包了很多内容，所以让我们分解一下：

- IODATA 和 Chardata 是二进制文件和整数的列表。这些二进制文件和整数可以任意嵌套在列表中。他们的目标是在处理 IO 设备和文件时提供灵活性和性能

- IODATA 和 Chardata 之间的选择取决于 IO 设备的编码。如果打开文件而不进行编码，则文件需要 iodata，并且必须使用模块中以 开头的函数。默认 IO 设备 （） 和用编码打开的文件需要 chardata，并使用模块中的其余函数`IO` `bin*` `:stdio` `:utf8` `IO`

- charlists 是 chardata 的一个特例，它只使用整数 Unicode 代码点的列表。它们可以用符号创建。如果列表中的所有整数都表示可打印的 ASCII 代码点，则使用符号自动打印整数列表。`~c` `~c`

我们对 IO 设备和 IO 相关功能的了解到此结束。我们已经了解了三个 Elixir模块 - `IO`、`文件` 和 `路径` - 以及VM如何使用底层IO机制的进程以及如何使用和用于IO操作。`chardata` `iodata`

← 上一页　返回页首　下一→