开始

枚举项和流

1 枚举项

2 渴望与懒惰

3 管道操作员

4 流

枚举项

Elixir 提供了枚举的概念和 `Enum` 模块来处理它们。我们已经学习了两个枚举：列表和映射。

```
iex> Enum.map([1, 2, 3], fn x -> x * 2 end)
[2, 4, 6]
iex> Enum.map(%{1 => 2, 3 => 4}, fn {k, v} -> k * v end)
[2, 12]
```

该模块提供了大量的函数来转换、排序、分组、筛选和检索可枚举项。它是开发人员在 Elixir 代码中经常使用的模块之一。`Enum`

Elixir 还提供范围：

```
iex> Enum.map(1..3, fn x -> x * 2 end)
[2, 4, 6]
iex> Enum.reduce(1..3, 0, &+/2)
6
```

顾名思义，Enum 模块中的函数仅限于枚举数据结构中的值。对于特定操作，例如插入和更新特定元素，您可能需要访问特定于数据类型的模块。

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

例如，如果要在列表中的给定位置插入元素，则应使用 `List` 模块中的函数，因为将值插入到例如区域中是没有意义的。`List.insert_at/3`

我们说模块中的函数是多态的，因为它们可以处理不同的数据类型。特别是，模块中的函数可以与实现 `Enumerable` 协议的任何数据类型一起使用。我们将在后面的章节中讨论协议；现在，我们将继续讨论一种称为流的特定类型的可枚举。`Enum` `Enum`

渴望与懒惰

模块中的所有功能都渴望。许多函数需要可枚举的函数并返回一个列表：`Enum`

```
iex> odd? = &(rem(&1, 2) != 0)
#Function<6.80484245/1 in :erl_eval.expr/5>
iex> Enum.filter(1..3, odd?)
[1, 3]
```

这意味着当执行多个操作时，每个操作都将生成一个中间列表，直到我们得到结果：`Enum`

```
iex> 1..100_000 |> Enum.map(&(&1 * 3)) |>
Enum.filter(odd?) |> Enum.sum()
7500000000
```

上面的示例具有操作管道。我们从一个范围开始，然后将范围中的每个元素乘以 3。第一个操作现在将创建并返回包含项的列表。然后我们保留列表中的所有奇数元素，生成一个新列表，现在包含项目，然后我们汇总所有条目。`100_000` `50_000`

管道操作员

上面代码片段中使用的符号是管道运算符：它从左侧的表达式获取输出，并将其作为第一个参数传递给右侧的函数调用。它类似于 Unix 运算符。其目的是突出显示由一系列函数转换的数据。要了解它如何使代码更简洁，请查看上面不使用运算符重写的示例：`|>` `|` `|>`

22. 调试
23. 类型规格和行为
24. 下一步去哪里

混合和一次性密码

1. 混音简介
2. 代理
3. GenServer
4. 主管和申请
5. 动态主管
6. 电子交易体系
7. 依赖项和伞形项目
8. 任务和 gen_tcp
9. 文档测试，模式和
10. 分布式任务和标签
11. 配置和发布

ELIXIR 中的元编程

1. 报价和取消报价
2. 宏
3. 域特定语言

```
iex> Enum.sum(Enum.filter(Enum.map(1..100_000, &(&1 * 3)),
odd?))
7500000000
```

通过阅读其文档了解有关管道操作员的更多信息。

流

作为替代方法，Elixir 提供了支持惰性操作的 `Stream` 模块： `Enum`

```
iex> 1..100_000 |> Stream.map(&(&1 * 3)) |>
Stream.filter(odd?) |> Enum.sum
7500000000
```

流是惰性的、可组合的可枚举项。

在上面的示例中， 返回一个数据类型，一个实际流，表示范围内的计算： `1..100_000 |> Stream.map(&(&1 * 3))` `map` `1..100_000`

```
iex> 1..100_000 |> Stream.map(&(&1 * 3))
#Stream<[enum: 1..100000, funs: [#Function<34.16982430/1
in Stream.map/2>]]>
```

此外，它们是可组合的，因为我们可以管道许多流操作：

```
iex> 1..100_000 |> Stream.map(&(&1 * 3)) |>
Stream.filter(odd?)
#Stream<[enum: 1..100000, funs: [...]]>
```

流不是生成中间列表，而是构建一系列计算，这些计算仅在我们将底层流传递给模块时才被调用。流在处理大型（*可能是无限*的）集合时很有用。 `Enum`

模块中的许多函数接受任何可枚举的参数，并返回流作为结果。它还提供用于创建流的功能。例如，可用于创建无限循环给定可枚举的流。注意不要像在这样的流上那样调用函数，因为它们会永远循环： `Stream` `Stream.cycle/1` `Enum.map/2`

```
iex> stream = Stream.cycle([1, 2, 3])
#Function<15.16982430/2 in Stream.unfold/2>
iex> Enum.take(stream, 10)
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1]
```

另一方面，可用于从给定的初始值生成值：`Stream.unfold/2`

```
iex> stream = Stream.unfold("hełło",
&String.next_codepoint/1)
#Function<39.75994740/2 in Stream.unfold/2>
iex> Enum.take(stream, 3)
["h", "e", "ł"]
```

另一个有趣的函数是，它可以用来环绕资源，保证它们在枚举之前打开并在枚举之后关闭，即使在失败的情况下也是如此。例如，在流式传输文件之上构建：`Stream.resource/3` `File.stream!/1` `Stream.resource/3`

```
iex> stream = File.stream!("path/to/file")
%File.Stream{
  line_or_bytes: :line,
  modes: [:raw, :read_ahead, :binary],
  path: "path/to/file",
  raw: true
}
iex> Enum.take(stream, 10)
```

上面的示例将获取所选文件的前 10 行。这意味着流对于处理大文件甚至慢速资源（如网络资源）非常有用。

`Enum` 和 `Stream` 模块中的功能数量起初可能令人生畏，但您将逐案熟悉它们。特别是，首先关注模块，并且只针对需要惰性的特定场景，以处理缓慢的资源或大型（可能是无限的）集合。`Enum` `Stream`

接下来，我们将看看 Elixir 的核心功能，即 Processes，它允许我们以一种简单易懂的方式编写并发，并行和分布式程序。

← 上一页　返回页首　下一→