开始

理解

在 Elixir 中，循环遍历枚举是很常见的，通常会过滤掉一些结果并将值映射到另一个列表中。理解是这种结构的句法糖：它们将这些常见任务分组为特殊形式。`for`

例如，我们可以将整数列表映射到它们的平方值：

```
iex> for n <- [1, 2, 3, 4], do: n * n
[1, 4, 9, 16]
```

理解由三部分组成：生成器、过滤器和收藏品。

发电机和过滤器

在上面的表达式中，是生成器。从字面上看，它是生成用于理解的值。任何可枚举项都可以在生成器表达式的右侧传递：`n <- [1, 2, 3, 4]`

```
iex> for n <- 1..4, do: n * n
[1, 4, 9, 16]
```

生成器表达式还支持左侧的模式匹配；*忽略所有*不匹配的模式。想象一下，我们有一个关键字列表，而不是一个范围，其中键是原子，或者我们只想计算值的平方：`:good` `:bad` `:good`

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> values = [good: 1, good: 2, bad: 3, good: 4]
iex> for {:good, n} <- values, do: n * n
[1, 4, 16]
```

除了模式匹配之外，过滤器可用于选择某些特定元素。例如，我们可以选择 3 的倍数并丢弃所有其他倍数：

```
iex> for n <- 0..5, rem(n, 3) == 0, do: n * n
[0, 9]
```

推导式会丢弃过滤器表达式返回的所有元素或；将选择所有其他值。`false` `nil`

理解通常提供比使用 and 模块中的等效函数更简洁的表示形式。此外，推导还允许给出多个生成器和滤波器。下面是一个示例，它接收目录列表并获取这些目录中每个文件的大小：`Enum` `Stream`

```
dirs = ["/home/mikey", "/home/james"]

for dir <- dirs,
    file <- File.ls!(dir),
    path = Path.join(dir, file),
    File.regular?(path) do
  File.stat!(path).size
end
```

多个生成器也可用于计算两个列表的笛卡尔积：

```
iex> for i <- [:a, :b, :c], j <- [1, 2], do:  {i, j}
[a: 1, a: 2, b: 1, b: 2, c: 1, c: 2]
```

最后，请记住，推导内的变量赋值，无论是在生成器、过滤器还是块内，都不会反映在推导之外。

位串生成器

还支持位串生成器，当您需要理解位串流时非常有用。下面的示例从二进制文件中接收像素列表，其中包含各自的红色、绿色和蓝色值，并将它们

转换为每个元组包含三个元素：

```
iex> pixels = <<213, 45, 132, 64, 76, 32, 76, 0, 0, 234,
32, 15>>
iex> for <<r::8, g::8, b::8 <- pixels>>, do: {r, g, b}
[{213, 45, 132}, {64, 76, 32}, {76, 0, 0}, {234, 32, 15}]
```

位串生成器可以与“常规”可枚举生成器混合使用，并且还支持筛选器。

“输入”选项

在上面的示例中，所有推导都返回列表作为其结果。但是，通过将选项传递给理解，可以将理解的结果插入到不同的数据结构中。`:into`

例如，位串生成器可以与该选项一起使用，以便轻松删除字符串中的所有空格：`:into`

```
iex> for <<c <- " hello world ">>, c != ?\s, into: "", do:
<<c>>
"helloworld"
```

集合、地图和其他词典也可以提供给该选项。通常，接受实现 可收集协议 的任何结构。`:into` `:into`

一个常见的用例是转换映射中的值：`:into`

```
iex> for {key, val} <- %{"a" => 1, "b" => 2}, into: %{},
do: {key, val * val}
%{"a" => 1, "b" => 4}
```

让我们使用流再举一个例子。由于该模块提供流（s 和 s），因此可以使用推导实现回显式回显任何类型向上大小写版本的回显终端：
`IO` `Enumerable` `Collectable`

```
iex> stream = IO.stream(:stdio, :line)
iex> for line <- stream, into: stream do
```

```
...>   String.upcase(line) <> "\n"
...> end
```

现在在终端中键入任何字符串，您将看到相同的值将以大写形式打印。不幸的是，这个例子也让你的 IEx shell 卡在理解中，所以你需要点击两次才能摆脱它。:) `Ctrl+C`

其他选项

理解支持其他选项，例如 和 。以下是了解有关理解的更多信息的其他资源： `:reduce` `:uniq`

- `用于` Elixir文档中的官方参考
- 米切尔·汉伯格（Mitchell Hanberg）对 Elixir 理解的综合指南

← 上一页　返回页首　下一→