开始

关键字列表和地图

现在让我们谈谈关联数据结构。关联数据结构能够将键与某个值相关联。不同的语言称呼这些不同的名称，如字典、哈希、关联数组等。

在 Elixir 中，我们有两种主要的关联数据结构：关键字列表和地图。是时候更多地了解它们了！

关键字列表

关键字列表是用于将选项传递给函数的数据结构。假设您要拆分一串数字。我们可以使用： `String.split/2`

```
iex> String.split("1 2 3", " ")
["1", "2", "3"]
```

但是，如果数字之间有额外的空格会发生什么：

```
iex> String.split("1  2  3", " ")
["1", "", "2", "", "3"]
```

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

如您所见，我们的结果中现在有空字符串。幸运的是，该函数允许将选项设置为 true：`String.split/3` `trim`

```
iex> String.split("1  2  3", " ", [trim: true])
["1", "2", "3"]
```

`[trim: true]` 是一个关键字列表。此外，当关键字列表是函数的最后一个参数时，我们可以跳过括号并编写：

```
iex> String.split("1  2  3", " ", trim: true)
["1", "2", "3"]
```

顾名思义，关键字列表只是列表。特别是，它们是由 2 项元组组成的列表，其中第一个元素（键）是原子，第二个元素可以是任何值。两种表示形式相同：

```
iex> [{:trim, true}] == [trim: true]
true
```

由于关键字列表是列表，因此我们可以使用列表可用的所有操作。例如，我们可以用来向关键字列表中添加新值：`++`

```
iex> list = [a: 1, b: 2]
[a: 1, b: 2]
iex> list ++ [c: 3]
[a: 1, b: 2, c: 3]
iex> [a: 0] ++ list
[a: 0, a: 1, b: 2]
```

您可以使用括号语法读取关键字列表的值：

```
iex> list[:a]
1
iex> list[:b]
2
```

如果键重复，添加到前面的值是获取的值：

```
iex> new_list = [a: 0] ++ list
[a: 0, a: 1, b: 2]
iex> new_list[:a]
0
```

关键字列表很重要，因为它们具有三个特殊特征：

- 密钥必须是原子。
- 密钥按照开发人员的指定进行排序。
- 钥匙可以多次提供。

例如，Ecto 库利用这些功能为编写数据库查询提供优雅的 DSL：

```
query =
  from w in Weather,
    where: w.prcp > 0,
    where: w.temp < 20,
    select: w
```

虽然我们可以在关键字列表上进行模式匹配，但在实践中很少这样做，因为列表上的模式匹配需要项目的数量及其顺序来匹配：

```
iex> [a: a] = [a: 1]
[a: 1]
iex> a
1
iex> [a: a] = [a: 1, b: 2]
** (MatchError) no match of right hand side value: [a: 1,
b: 2]
iex> [b: b, a: a] = [a: 1, b: 2]
** (MatchError) no match of right hand side value: [a: 1,
b: 2]
```

为了操纵关键字列表，Elixir提供了 `关键字` 模块。但请记住，关键字列表只是列表，因此它们提供与它们相同的线性性能特征：列表越长，查找键、计算项目数等所需的时间就越长。出于这个原因，关键字列表在Elixir中主要用于传递可选值。如果需要存储多个项目或保证最多一个值的一键关联，则应改用地图。

待办事项块和关键字

正如我们所看到的，关键字在语言中主要用于传递可选值。实际上，我们之前在本指南中使用过关键字。例如，我们已经看到：

```
iex> if true do
...>   "This will be seen"
...> else
...>   "This won't"
...> end
"This will be seen"
```

碰巧块只不过是关键字之上的语法便利。我们可以将上述内容重写为：`do`

```
iex> if true, do: "This will be seen", else: "This won't"
"This will be seen"
```

请密切注意这两种语法。在关键字列表格式中，我们用逗号分隔每个键值对，每个键后跟。在 -blocks 中，我们去掉了冒号、逗号，并用换行符分隔每个关键字。它们之所以有用，正是因为它们消除了编写代码块时的冗长。大多数情况下，您将使用块语法，但很高兴知道它们是等效的。`:` `do`

请注意，只有少数关键字列表可以转换为块：、、、 和 。这些都是 Elixir 控制流结构使用的关键字。我们已经学习了其中的一些，我们将来将学习其他一些。`do` `else` `catch` `rescue` `after`

有了这个，让我们看看如何使用嵌套数据结构。

映射为键值对

每当你需要存储键值对时，地图就是 Elixir 中的“去”数据结构。使用以下语法创建地图：`%{}`

```
iex> map = %{:a => 1, 2 => :b}
%{2 => :b, :a => 1}
iex> map[:a]
1
```

```
iex> map[2]
:b
iex> map[:c]
nil
```

与关键字列表相比，我们已经可以看到两个区别：

- 映射允许任何值作为键。
- 地图的键不遵循任何排序。

与关键字列表相比，地图在模式匹配方面非常有用。当在模式中使用映射时，它将始终与给定值的子集匹配：

```
iex> %{} = %{:a => 1, 2 => :b}
%{2 => :b, :a => 1}
iex> %{:a => a} = %{:a => 1, 2 => :b}
%{2 => :b, :a => 1}
iex> a
1
iex> %{:c => c} = %{:a => 1, 2 => :b}
** (MatchError) no match of right hand side value: %{2 =>
:b, :a => 1}
```

如上所示，只要模式中的键存在于给定映射中，映射就会匹配。因此，空地图匹配所有地图。

`Map` 模块提供了一个与该模块非常相似的 API，具有添加、删除和更新映射键的便捷函数： `Keyword`

```
iex> Map.get(%{:a => 1, 2 => :b}, :a)
1
iex> Map.put(%{:a => 1, 2 => :b}, :c, 3)
%{2 => :b, :a => 1, :c => 3}
iex> Map.to_list(%{:a => 1, 2 => :b})
[{2, :b}, {:a, 1}]
```

固定键地图

在上一节中，我们使用 maps 作为键值数据结构，可以随时添加或删除键。但是，使用一组预定义的键创建映射也很常见。它们的值可能会更

新，但永远不会添加或删除新键。当我们知道我们正在处理的数据的形状时，这很有用，如果我们得到不同的键，这可能意味着错误 是在其他地方完成的。

我们使用与上一节相同的语法定义此类映射，除了 所有键都必须是原子：

```
iex> map = %{:name => "John", :age => 23}
%{name: "John", age: 23}
```

从上面的打印结果中可以看出，Elixir 还允许您 使用与关键字列表相同的语法编写原子键的映射。`key: value`

当键是原子时，我们也可以使用语法访问它们：`map.key`

```
iex> map = %{name: "John", age: 23}
%{name: "John", age: 23}

iex> map.name
"John"
iex> map.agee
** (KeyError) key :agee not found in: %{name: "John", age:
23}
```

此语法有一个很大的好处，因为如果映射中不存在键，它会引发异常。有时 Elixir 编译器甚至可能发出警告。这使得获得快速反馈并尽早发现错误和拼写错误非常有用。这也是 用于支持另一个称为“结构”的 Elixir 功能的语法。

Elixir开发人员在使用地图时通常更喜欢使用语法和模式匹配而不是模块中的函数，因为它们会导致自信的编程风格。José Valim 的这篇博客文章提供了有关如何通过在Elixir中编写自信代码来获得更简洁，更快速的软件的见解和示例。`map.key` `Map`

嵌套数据结构

通常我们会在地图中拥有地图，甚至在地图中拥有关键字列表，等等。Elixir 为通过，和其他宏操作嵌套数据结构提供了便利，提供了与命令式语言相同的便利，同时保持了语言的不可变属性。
`put_in/2` `update_in/2`

假设您有以下结构：

```
iex> users = [
  john: %{name: "John", age: 27, languages: ["Erlang",
"Ruby", "Elixir"]},
  mary: %{name: "Mary", age: 29, languages: ["Elixir",
"F#", "Clojure"]}
]
[
  john: %{age: 27, languages: ["Erlang", "Ruby",
"Elixir"], name: "John"},
  mary: %{age: 29, languages: ["Elixir", "F#", "Clojure"],
name: "Mary"}
]
```

我们有一个用户关键字列表，其中每个值都是一个包含名称，年龄和每个用户喜欢的编程语言列表的映射。如果我们想了解约翰的年龄，我们可以这样写：

```
iex> users[:john].age
27
```

碰巧我们也可以使用相同的语法来更新值：

```
iex> users = put_in users[:john].age, 31
[
  john: %{age: 31, languages: ["Erlang", "Ruby",
"Elixir"], name: "John"},
  mary: %{age: 29, languages: ["Elixir", "F#", "Clojure"],
name: "Mary"}
]
```

宏类似，但允许我们传递一个控制值如何变化的函数。例如，让我们从 Mary 的语言列表中删除 “Clojure”：`update_in/2`

```
iex> users = update_in users[:mary].languages, fn
languages -> List.delete(languages, "Clojure") end
[
  john: %{age: 31, languages: ["Erlang", "Ruby",
"Elixir"], name: "John"},
  mary: %{age: 29, languages: ["Elixir", "F#"], name:
```

```
  "Mary"}
]
```

还有更多需要学习的内容，包括允许我们立即提取值并更新数据结构的。还有 ，并且允许动态访问数据结构。有关详细信息，请查看 `内核` 模块中各自的文档。在访问之间 模块和模式匹配，Elixir开发人员拥有丰富的工具 用于操作嵌套和复杂的数据结构。

`put_in/2` `update_in/2` `get_and_update_in/2` `put_in/3` `update_in/3` `get_and_update_in/3`

总结

我们对 Elixir 中关联数据结构的介绍到此结束。 作为总结，您应该：

- 使用关键字列表将可选值传递给函数
- 将映射用于常规键值数据结构以及处理已知数据（使用固定键）时

现在我们可以继续讨论模块和功能。

← 上一页　返回页首　下一→