模块和功能

在 Elixir 中，我们将几个功能分组到模块中。在前面的章节中，我们已经使用了许多不同的模块，例如 `字符串` 模块：

```
iex> String.length("hello")
5
```

为了在 Elixir 中创建我们自己的模块，我们使用宏。模块的第一个字母必须为大写。我们使用宏来定义该模块中的函数。每个函数的第一个字母必须为小写（或下划线）： `defmodule` `def`

```
iex> defmodule Math do
...>   def sum(a, b) do
...>     a + b
...>   end
...> end

iex> Math.sum(1, 2)
3
```

在以下部分中，我们的示例的大小将变得更长，并且在 shell 中键入它们可能会很棘手。现在是我们学习如何编译 Elixir 代码以及如何运行 Elixir 脚本的时候了。

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

汇编

大多数情况下，将模块写入文件很方便，以便可以编译和重用它们。假设我们有一个以以下内容命名的文件：`math.ex`

```
defmodule Math do
  def sum(a, b) do
    a + b
  end
end
```

此文件可以使用以下方法编译：`elixirc`

```
$ elixirc math.ex
```

这将生成一个名为的文件，其中包含已定义模块的字节码。如果我们重新开始，我们的模块定义将可用（前提是在字节码文件所在的同一目录中启动）：`Elixir.Math.beam` `iex` `iex`

```
iex> Math.sum(1, 2)
3
```

Elixir 项目通常分为三个目录：

- `_build` - 包含编译工件
- `lib` - 包含长生不老药代码（通常是文件）`.ex`
- `test` - 包含测试（通常是文件）`.exs`

在处理实际项目时，调用的构建工具将负责为您编译和设置正确的路径。出于学习和方便的目的，Elixir 还支持脚本模式，该模式更加灵活，不会生成任何编译的工件。`mix`

脚本化模式

除了 Elixir 的文件扩展名，Elixir 还支持用于脚本的文件。Elixir 以完全相同的方式处理这两个文件，唯一的区别在于意图。 文件旨在编译，而文件用于脚本。遵循此约定的项目，例如 `.ex` `.exs` `.ex` `.exs` `mix`

例如，我们可以创建一个名为： `math.exs`

```
defmodule Math do
  def sum(a, b) do
    a + b
  end
end

IO.puts Math.sum(1, 2)
```

并按以下方式执行：

```
$ elixir math.exs
```

因为我们使用 代替 ，模块被编译并加载到内存中，但没有文件写入磁盘。在以下示例中，建议将代码写入脚本文件并执行它们，如上所示。 `elixir` `elixirc` `.beam`

命名函数

在模块内部，我们可以定义函数和私有函数。定义的函数可以从其他模块调用，而私有函数只能在本地调用。 `def/2` `defp/2` `def/2`

```
defmodule Math do
  def sum(a, b) do
    do_sum(a, b)
  end

  defp do_sum(a, b) do
    a + b
  end
end

IO.puts Math.sum(1, 2)    #=> 3
IO.puts Math.do_sum(1, 2) #=> ** (UndefinedFunctionError)
```

函数声明还支持保护和多个子句。如果一个函数有几个子句，Elixir 将尝试每个子句，直到找到一个匹配的子句。下面是一个函数的实现，用于检查给定的数字是否为零：

```
defmodule Math do
  def zero?(0) do
    true
  end

  def zero?(x) when is_integer(x) do
    false
  end
end

IO.puts Math.zero?(0)         #=> true
IO.puts Math.zero?(1)         #=> false
IO.puts Math.zero?([1, 2, 3]) #=> ** (FunctionClauseError)
IO.puts Math.zero?(0.0)       #=> ** (FunctionClauseError)
```

后面的问号表示此函数返回布尔值。要了解有关Elixir中模块，函数名称，变量等的命名约定的更多信息，请参阅命名约定。`zero?`

给出与任何子句都不匹配的参数会引发错误。

类似于 这样的构造，命名函数支持两者和 -block 语法，正如我们在上一章中学到的。例如，我们可以编辑成这样：`if` `do:` `do` `math.exs`

```
defmodule Math do
  def zero?(0), do: true
  def zero?(x) when is_integer(x), do: false
end
```

它将提供相同的行为。您可以用于单行，但始终将 -blocks 用于跨多行的函数。如果您希望保持一致，则可以在整个代码库中使用 -blocks。
`do:` `do` `do`

函数捕获

在本教程中，我们一直在使用符号来引用函数。碰巧此表示法实际上可用于检索命名函数作为函数类型。启动 ，运行上面定义的文件：
`name/arity` `iex` `math.exs`

```
$ iex math.exs
```

```
iex> Math.zero?(0)
true
iex> fun = &Math.zero?/1
&Math.zero?/1
iex> is_function(fun)
true
iex> fun.(0)
true
```

请记住，Elixir 区分了匿名函数和命名函数，其中前者必须在变量名和括号之间用点（）调用。捕获运算符 （） 通过允许将命名函数分配给变量并作为参数传递来弥合这一差距，就像我们分配、调用和传递匿名函数一样。 `.` `&`

本地或导入的函数，如 ，可以在没有模块的情况下捕获：
`is_function/1`

```
iex> &is_function/1
&:erlang.is_function/1
iex> (&is_function/1).(fun)
true
```

您还可以捕获运算符：

```
iex> add = &+/2
&:erlang.+/2
iex> add.(1, 2)
3
```

请注意，捕获语法也可以用作创建函数的快捷方式：

```
iex> fun = &(&1 + 1)
#Function<6.71889879/1 in :erl_eval.expr/5>
iex> fun.(1)
2

iex> fun2 = &"Good #{&1}"
#Function<6.127694169/1 in :erl_eval.expr/5>
iex> fun2.("morning")
"Good morning"
```

表示传递到函数中的第一个参数。以上与。上面的语法对于简短的函数定义很有用。`&1` `&(&1 + 1)` `fn x -> x + 1 end`

您可以在 `Kernel.SpecialForms` 文档中阅读有关捕获运算符的更多信息。`&`

默认参数

Elixir 中的命名函数也支持默认参数：

```
defmodule Concat do
  def join(a, b, sep \\ " ") do
    a <> sep <> b
  end
end

IO.puts Concat.join("Hello", "world")      #=> Hello world
IO.puts Concat.join("Hello", "world", "_") #=> Hello_world
```

允许任何表达式用作默认值，但在函数定义期间不会对其进行计算。每次调用函数并且必须使用其任何默认值时，都将计算该默认值的表达式：

```
defmodule DefaultTest do
  def dowork(x \\ "hello") do
    x
  end
end
```

```
iex> DefaultTest.dowork
"hello"
iex> DefaultTest.dowork 123
123
iex> DefaultTest.dowork
"hello"
```

如果具有默认值的函数有多个子句，则需要创建一个函数头（没有主体的函数定义）来声明默认值：

```
defmodule Concat do
  # A function head declaring defaults
  def join(a, b \\ nil, sep \\ " ")

  def join(a, b, _sep) when is_nil(b) do
    a
  end

  def join(a, b, sep) do
    a <> sep <> b
  end
end

IO.puts Concat.join("Hello", "world")      #=> Hello world
IO.puts Concat.join("Hello", "world", "_") #=> Hello_world
IO.puts Concat.join("Hello")               #=> Hello
```

当函数或子句不使用变量时，我们在其名称中添加前导下划线 （） 以表示此意图。我们的命名约定文档中也介绍了此规则。 `_`

使用默认值时，必须小心避免重叠的函数定义。请考虑以下示例：

```
defmodule Concat do
  def join(a, b) do
    IO.puts "***First join"
    a <> b
  end

  def join(a, b, sep \\ " ") do
    IO.puts "***Second join"
    a <> sep <> b
  end
end
```

Elixir 将发出以下警告：

```
concat.ex:7: warning: this clause cannot match because a
previous clause at line 2 always matches
```

编译器告诉我们，使用两个参数调用函数将始终选择第一个定义，而第二个定义仅在传递三个参数时调用： `join` `join`

```
$ iex concat.ex
```

```
iex> Concat.join "Hello", "world"
***First join
"Helloworld"
```

```
iex> Concat.join "Hello", "world", "_"
***Second join
"Hello_world"
```

在这种情况下，删除默认参数将修复警告。

我们对模块的简短介绍到此结束。在接下来的章节中，我们将学习如何使用命名函数进行递归，探索可用于从其他模块导入函数的 Elixir 词法指令，并讨论模块属性。

← 上一页　返回页首　下一→