开始

协议

协议是一种在 Elixir 中实现多态性的机制，当你希望行为根据数据类型而变化时。我们已经熟悉解决此类问题的一种方法：通过模式匹配和保护子句。考虑一个简单的实用程序模块，它将告诉我们输入变量的类型：

```
defmodule Utility do
  def type(value) when is_binary(value), do: "string"
  def type(value) when is_integer(value), do: "integer"
  # ... other implementations ...
end
```

如果此模块的使用仅限于您自己的项目，您将能够继续为每个新数据类型定义新函数。但是，如果此代码由多个应用作为依赖项共享，则可能会出现问题，因为没有简单的方法来扩展其功能。`type/1`

这就是协议可以帮助我们的地方：协议允许我们根据需要扩展任意数量的数据类型的原始行为。这是因为协议上的调度可用于已实现协议的任何数据类型，并且任何人都可以随时实现协议。

以下是我们如何编写与协议相同的功能：`Utility.type/1`

新闻：Elixir v1.15 发布

搜索。。。

```
defprotocol Utility do
  @spec type(t) :: String.t()
  def type(value)
end

defimpl Utility, for: BitString do
  def type(_value), do: "string"
end

defimpl Utility, for: Integer do
  def type(_value), do: "integer"
end
```

我们使用 - 它的函数和规范可能类似于其他语言中的接口或抽象基类来定义协议。我们可以添加任意数量的实现，只要我们喜欢使用。输出与我们拥有具有多个功能的单个模块完全相同： `defprotocol` `defimpl`

```
iex> Utility.type("foo")
"string"
iex> Utility.type(123)
"integer"
```

然而，使用协议，我们不再需要不断修改相同的模块来支持越来越多的数据类型。例如，我们可以获取上面的调用并将它们分布在多个文件中，Elixir将根据数据类型将执行调度到适当的实现中。协议中定义的函数可能有多个输入，但调度将始终基于第一个输入的数据类型。 `defimpl`

您可能遇到的最常见的协议之一是 `String.Chars` 协议：为自定义结构实现其功能将告诉Elixir内核如何将它们表示为字符串。稍后我们将探讨所有内置协议。现在，让我们实现我们自己的。 `to_string/1`

例

现在您已经看到了协议帮助解决的问题类型以及它们如何解决这些问题的示例，让我们看一个更深入的示例。

在 Elixir 中，我们有两个习惯用法来检查数据结构中有多少项：和。 表示必须计算信息。例如，需要遍历整个列表以计算其长度。另一方面，不要依赖于元组和二进制大小，因为大小信息是在数据结构中预先计算的。 `length` `size` `length` `length(list)` `tuple_size(tuple)` `byte_size(binary)`

即使我们有特定于类型的函数来将大小内置到 Elixir 中（例如），我们也可以实现一个通用协议，所有预先计算大小的数据结构都将实现该协议。`tuple_size/1` `Size`

协议定义如下所示：

```
defprotocol Size do
  @doc "Calculates the size (and not the length!) of a
data structure"
  def size(data)
end
```

该协议期望实现一个名为接收一个参数（我们想知道其大小的数据结构）的函数。现在，我们可以为具有兼容实现的数据结构实现此协议：`Size` `size`

```
defimpl Size, for: BitString do
  def size(string), do: byte_size(string)
end

defimpl Size, for: Map do
  def size(map), do: map_size(map)
end

defimpl Size, for: Tuple do
  def size(tuple), do: tuple_size(tuple)
end
```

我们没有为列表实现协议，因为没有为列表预先计算“大小”信息，并且必须计算列表的长度（使用 ）。 `Size` `length/1`

现在定义了协议并有了实现，我们可以开始使用它了：

```
iex> Size.size("foo")
3
iex> Size.size({:ok, "hello"})
2
iex> Size.size(%{label: "some label"})
1
```

传递未实现协议的数据类型会引发错误：

```
iex> Size.size([1, 2, 3])
** (Protocol.UndefinedError) protocol Size not implemented
for [1, 2, 3] of type List
```

可以为所有 Elixir 数据类型实现协议：

- `Atom`
- `BitString`
- `Float`
- `Function`
- `Integer`
- `List`
- `Map`
- `PID`
- `Port`
- `Reference`
- `Tuple`

协议和结构

Elixir 可扩展性的力量来自于协议和结构一起使用。

在上一章中，我们已经了解到，虽然结构是映射，但它们不与映射共享协议实现。例如， `MapSet` s（基于映射的集合）作为结构实现。让我们尝试将协议与： `Size` `MapSet`

```
iex> Size.size(%{})
0
iex> set = %MapSet{} = MapSet.new
MapSet.new([])
iex> Size.size(set)
** (Protocol.UndefinedError) protocol Size not implemented
for MapSet.new([]) of type MapSet (a struct)
```

结构不需要与映射共享协议实现，而是需要自己的协议实现。由于 a 已预先计算其大小并可通过 访问，我们可以为其定义一个实现：`MapSet` `MapSet.size/1` `Size`

```
defimpl Size, for: MapSet do
  def size(set), do: MapSet.size(set)
end
```

如果需要，您可以为结构的大小提出自己的语义。不仅如此，您还可以使用结构来构建更健壮的数据类型（如队列），并为此数据类型实现所有相关协议，例如和可能。`Enumerable` `Size`

```
defmodule User do
  defstruct [:name, :age]
end

defimpl Size, for: User do
  def size(_user), do: 2
end
```

实现任何

手动实现所有类型的协议很快就会变得重复和乏味。在这种情况下，Elixir 提供了两种选择：我们可以显式地为我们的类型导出协议实现，或者自动实现所有类型的协议。在这两种情况下，我们都需要实现 的协议。`Any`

推导

Elixir 允许我们基于实现推导出协议实现。让我们首先实现如下：`Any` `Any`

```
defimpl Size, for: Any do
  def size(_), do: 0
end
```

上面的实现可以说不是一个合理的实现。例如，说 a 或 a 的大小是没有意义的。`PID` `Integer` `0`

但是，我们应该对 的实现感到满意，为了使用这样的实现，我们需要告诉我们的结构显式派生协议： `Any` `Size`

```
defmodule OtherUser do
  @derive [Size]
  defstruct [:name, :age]
end
```

在派生时，Elixir 将基于提供的实现实现协议。 `Size` `OtherUser` `Any`

回退到 `任何`

另一种替代方法是显式告诉协议在找不到实现时回退。这可以通过在协议定义中设置为 来实现： `@derive` `Any` `@fallback_to_any` `true`

```
defprotocol Size do
  @fallback_to_any true
  def size(data)
end
```

正如我们在上一节中所说，for 的实现不能应用于任何数据类型。这就是选择加入行为的原因之一。对于大多数协议，在未实现协议时引发错误是正确行为。也就是说，假设我们已经实现了上一节：
`Size` `Any` `@fallback_to_any` `Any`

```
defimpl Size, for: Any do
  def size(_), do: 0
end
```

现在，所有尚未实现该协议的数据类型（包括结构）都将被视为大小为 .`Size` `0`

在派生和回退之间哪种技术最好取决于用例，但是，鉴于 Elixir 开发人员更喜欢显式而不是隐式，您可能会看到许多库正在推动这种方法。
`Any` `@derive`

内置协议

Elixir 附带了一些内置协议。在前面的章节中，我们已经讨论了该模块，该模块提供了许多函数，这些函数适用于实现该协议的任何数据结构：`Enum` `Enumerable`

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

另一个有用的示例是协议，它指定如何将数据结构转换为字符串形式的人类表示形式。它通过以下函数公开：`String.Chars` `to_string`

```
iex> to_string :hello
"hello"
```

请注意，Elixir 中的字符串插值调用了该函数：`to_string`

```
iex> "age: #{25}"
"age: 25"
```

上面的代码片段之所以有效，是因为数字实现了协议。例如，传递元组将导致错误：`String.Chars`

```
iex> tuple = {1, 2, 3}
{1, 2, 3}
iex> "tuple: #{tuple}"
** (Protocol.UndefinedError) protocol String.Chars not
implemented for {1, 2, 3} of type Tuple
```

当需要“打印”更复杂的数据结构时，可以使用基于协议的功能：`inspect` `Inspect`

```
iex> "tuple: #{inspect tuple}"
"tuple: {1, 2, 3}"
```

该协议是用于将任何数据结构转换为可读文本表示的协议。这就是像 IEx 这样的工具用来打印结果的：`Inspect`

```
iex> {1, 2, 3}
{1, 2, 3}
iex> %User{}
%User{name: "john", age: 27}
```

请记住，按照惯例，每当检查的值以 开头时，它都以无效的 Elixir 语法表示数据结构。这意味着检查协议是不可逆的，因为信息可能会在此过程中丢失：`#`

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

Elixir中还有其他协议，但这涵盖了最常见的协议。您可以在`协议`模块中了解有关协议和实现的更多信息。

← 上一页　返回页首　下一→