类型规格和行为

类型和规格

Elixir是一种动态类型语言，因此Elixir中的所有类型都在运行时进行检查。尽管如此，Elixir带有字体规范，这是一种用于以下的符号：

1. 声明类型化函数签名（也称为规范）；
2. 声明自定义类型。

Typespecs 对于代码清晰度和静态代码分析很有用（例如，Erlang 的 Dialyzer 工具）。

功能规格

Elixir提供了许多内置类型，例如或，可用于记录函数签名。例如，将数字舍入到最接近的整数的函数。正如您在其文档中看到的，键入的签名写为：`integer` `pid` `round/1` `round/1`

```
round(number()) :: integer()
```

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

语法是将函数及其输入放在 的左侧，将返回值的类型放在右侧。请注意，类型*可以*省略括号。`::`

在代码中，函数规范是使用属性编写的，通常放在函数定义之前。规范可以描述公共和私有功能。属性中使用的函数名称和参数数必须与它描述的函数匹配。`@spec` `@spec`

Elixir 也支持化合物类型。例如，整数列表具有类型 ，或定义键和类型的映射（请参阅下面的示例）。`[integer]`

您可以在 typespecs 文档中查看 Elixir 提供的所有内置类型。

定义自定义类型

定义自定义类型有助于传达代码的意图并提高其可读性。自定义类型可以通过属性在模块中定义。`@type`

自定义类型实现的一个简单示例是提供现有类型的更具描述性的别名。例如，定义为类型会使函数规范比简单地使用：`year` `integer`

```
defmodule Person do
   @typedoc """
   A 4 digit year, e.g. 1984
   """
   @type year :: integer

   @spec current_age(year) :: integer
   def current_age(year_of_birth), do: # implementation
end
```

该属性与 and 属性类似，用于记录自定义类型。
`@typedoc` `@doc` `@moduledoc`

您可以定义复合自定义类型，例如映射：

```
@type error_map :: %{
   message: String.t,
   line_number: integer
}
```

结构提供类似的功能。

让我们看另一个示例来了解如何定义更复杂的类型。假设我们有一个模块，它执行通常的算术运算（总和、乘积等），但它不是返回数字，而是返回元组，运算结果作为第一个元素，随机备注作为第二个元素。`LousyCalculator`

```
defmodule LousyCalculator do
  @spec add(number, number) :: {number, String.t}
  def add(x, y), do: {x + y, "You need a calculator to do that?!"}

  @spec multiply(number, number) :: {number, String.t}
  def multiply(x, y), do: {x * y, "Jeez, come on!"}
end
```

元组是一种复合类型，每个元组由其内部的类型（在本例中为数字和字符串）标识。要理解为什么不写为 ，请再看一下 typespecs 文档。`String.t` `string`

以这种方式定义函数规范是有效的，但我们最终会一遍又一遍地重复该类型。我们可以使用该属性来声明我们自己的自定义类型并减少重复。`{number, String.t}` `@type`

```
defmodule LousyCalculator do
  @typedoc """
  Just a number followed by a string.
  """
  @type number_with_remark :: {number, String.t}

  @spec add(number, number) :: number_with_remark
  def add(x, y), do: {x + y, "You need a calculator to do that?"}

  @spec multiply(number, number) :: number_with_remark
  def multiply(x, y), do: {x * y, "It is like addition on steroids."}
end
```

通过 定义的自定义类型将导出，并在定义它们的模块外部可用：`@type`

```
defmodule QuietCalculator do
  @spec add(number, number) :: number
  def add(x, y), do: make_quiet(LousyCalculator.add(x, y))

  @spec make_quiet(LousyCalculator.number_with_remark) ::
number
  defp make_quiet({num, _remark}), do: num
end
```

如果要使自定义类型保持私有，可以使用该属性而不是 。可见性还会影响文档是否由Elixir的文档生成器ExDoc等工具生成。`@typep` `@type`

静态代码分析

类型规范不仅作为附加文档对开发人员有用。例如，Erlang工具Dialyzer使用类型规范来执行代码的静态分析。这就是为什么在示例中，我们为函数编写了一个规范，即使它被定义为私有函数。
`QuietCalculator` `make_quiet/1`

行为

许多模块共享相同的公共 API。看看 Plug，正如其描述所述，它是 Web 应用程序中可组合模块的 **规范**。每个*插头*都是一个模块，**必须**实现至少两个公共功能：和。`init/1` `call/2`

行为提供了一种方法：

- 定义一组必须由模块实现的功能；
- 确保模块实现了该集中的所有函数。

如果有必要，你可以考虑像 Java 等面向对象语言中的接口这样的行为：模块必须实现的一组函数签名。与协议不同，行为与类型 / 数据无关。

定义行为

假设我们要实现一堆解析器，每个解析器解析结构化数据：例如，JSON 解析器和 MessagePack 解析器。这两个解析器中的每一个都将以相同的方式*运行*：两者都将提供一个函数和一个函数。该函数将返回结构化数据的Elixir表示形式，而该函数将返回可用于每种类型数据（例如，JSON文

件）的文件扩展名列表。
`parse/1` `extensions/0` `parse/1` `extensions/0` `.json`

我们可以创建一种行为：`Parser`

```
defmodule Parser do
  @doc """
  Parses a string.
  """
  @callback parse(String.t) :: {:ok, term} | {:error,
atom}

  @doc """
  Lists all supported file extensions.
  """
  @callback extensions() :: [String.t]
end
```

采用该行为的模块必须实现使用该属性定义的所有函数。如您所见，需要一个函数名称，但也需要一个函数规范，就像我们上面看到的属性一起使用的那些一样。另请注意，该类型用于表示解析的值。在 Elixir 中，类型是表示任何类型的快捷方式。
`Parser` `@callback` `@callback` `@spec` `term` `term`

实施行为

实现行为很简单：

```
defmodule JSONParser do
  @behaviour Parser

  @impl Parser
  def parse(str), do: {:ok, "some json " <> str} # ...
parse JSON

  @impl Parser
  def extensions, do: [".json"]
end
```

```
defmodule CSVParser do
  @behaviour Parser
```

```
  @impl Parser
  def parse(str), do: {:ok, "some csv " <> str} # ...
parse CSV

  @impl Parser
  def extensions, do: [".csv"]
end
```

如果采用给定行为的模块未实现该行为所需的回调之一，则会生成编译时警告。

此外，您还可以确保以明确的方式从给定行为实现正确的回调。例如，以下解析器同时实现 和 。但是，由于拼写错误，正在实现而不是.`@impl` `parse` `extensions` `BADParser` `parse/0` `parse/1`

```
defmodule BADParser do
  @behaviour Parser

  @impl Parser
  def parse, do: {:ok, "something bad"}

  @impl Parser
  def extensions, do: ["bad"]
end
```

此代码会生成一条警告，告知您错误地实现了 而不是 。 您可以在模块文档中阅读更多信息。`parse/0` `parse/1` `@impl`

使用行为

行为很有用，因为您可以将模块作为参数传递，然后可以*回调*行为中指定的任何函数。例如，我们可以有一个函数来接收文件名、多个解析器，并根据其扩展名解析文件：

```
@spec parse_path(Path.t(), [module()]) :: {:ok, term} |
{:error, atom}
def parse_path(filename, parsers) do
  with {:ok, ext} <- parse_extension(filename),
       {:ok, parser} <- find_parser(ext, parsers),
       {:ok, contents} <- File.read(filename) do
    parser.parse(contents)
  end
```

```
  end

  defp parse_extension(filename) do
    if ext = Path.extname(filename) do
      {:ok, ext}
    else
      {:error, :no_extension}
    end
  end

  defp find_parser(ext, parsers) do
    if parser = Enum.find(parsers, fn parser -> ext in
parser.extensions() end) do
      {:ok, parser}
    else
      {:error, :no_matching_parser}
    end
  end
```

当然，您也可以直接调用任何解析器：。`CSVParser.parse(...)`

请注意，您无需定义行为即可在模块上动态调度，但这些功能通常是齐头并进的。