开始

别名、要求和导入

为了便于软件重用，Elixir 提供了三个指令（和）以及一个宏，总结如下：`alias` `require` `import` `use`

```
# Alias the module so it can be called as Bar instead of
Foo.Bar
alias Foo.Bar, as: Bar

# Require the module in order to use its macros
require Foo

# Import functions from Foo so they can be called without
the `Foo.` prefix
import Foo

# Invokes the custom code defined in Foo as an extension
point
use Foo
```

我们现在将详细探讨它们。请记住，前三个称为指令，因为它们具有词法范围，而是一个通用的扩展点，允许使用的模块注入代码。`use`

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

别名

`alias` 允许您为任何给定的模块名称设置别名。

假设一个模块使用在 中实现的专用列表。该指令允许引用，就像在模块定义中一样： `Math.List` `alias` `Math.List` `List`

```
defmodule Stats do
  alias Math.List, as: List
  # In the remaining module definition List expands to
Math.List.
end
```

原始名称仍可通过完全限定名称 访问。 `List` `Stats` `Elixir.List`

> *注意：Elixir 中定义的所有模块都在主命名空间中定义，例如。但是，为了方便起见，您可以在引用它们时省略“Elixir.”。* `Elixir` `Elixir.String`

别名经常用于定义快捷方式。实际上，在没有选项的情况下调用会自动将别名设置为模块名称的最后一部分，例如： `alias` `:as`

```
alias Math.List
```

与以下相同：

```
alias Math.List, as: List
```

请注意，这是**按词法作用域的**，它允许您在特定函数中设置别名： `alias`

```
defmodule Math do
  def plus(a, b) do
    alias Math.List
    # ...
  end

  def minus(a, b) do
    # ...
```

```
  end
end
```

在上面的例子中，由于我们在函数内部调用，别名将仅在函数内部有效。完全不会受到影响。`alias` `plus/2` `plus/2` `minus/2`

需要

Elixir 提供宏作为元编程（编写生成代码的代码）的机制。宏在编译时展开。

模块中的公共函数是全局可用的，但为了使用宏，您需要通过要求在其中定义它们的模块来选择加入。

```
iex> Integer.is_odd(3)
** (UndefinedFunctionError) function Integer.is_odd/1 is
undefined or private. However, there is a macro with the
same name and arity. Be sure to require Integer if you
intend to invoke this macro
    (elixir) Integer.is_odd(3)
iex> require Integer
Integer
iex> Integer.is_odd(3)
true
```

在 Elixir 中，被定义为宏，以便它可以用作守卫。这意味着，为了调用，我们需要首先需要模块。
`Integer.is_odd/1` `Integer.is_odd/1` `Integer`

请注意，与指令一样，也是词法范围的。我们将在后面的章节中详细讨论宏。`alias` `require`

进口

每当我们想从其他模块访问函数或宏而不使用完全限定名称时，我们都会使用。请注意，我们只能导入公共函数，因为私有函数永远无法从外部访问。`import`

例如，如果我们想多次使用模块中的函数，我们可以导入它：
`duplicate/2` `List`

```
iex> import List, only: [duplicate: 2]
List
iex> duplicate(:ok, 3)
[:ok, :ok, :ok]
```

我们仅从 .虽然是可选的，但建议使用它，以避免在当前范围内导入给定模块的所有函数。 也可以作为一个选项提供，以便导入模块中除函数列表*之外*的所有内容。 `duplicate` `List` `:only` `:except`

请注意，这也是**词法范围**。这意味着我们可以在函数定义中导入特定的宏或函数： `import`

```
defmodule Math do
  def some_function do
    import List, only: [duplicate: 2]
    duplicate(:ok, 10)
  end
end
```

在上面的示例中，导入的内容仅在该特定函数中可见。 在该模块（或任何其他模块）的任何其他函数中不可用。 `List.duplicate/2` `duplicate/2`

请注意，在语言中通常不鼓励使用 s。在处理自己的代码时，首选 。 `import` `alias` `import`

用

宏经常用作扩展点。这意味着，当您一个模块时，您允许该模块在当前模块中注入*任何*代码，例如导入自身或其他模块、定义新功能、设置模块状态等。 `use` `use` `FooBar`

例如，为了使用 ExUnit 框架编写测试，开发人员应使用以下模块： `ExUnit.Case`

```
defmodule AssertionTest do
  use ExUnit.Case, async: true

  test "always pass" do
    assert true
```

```
  end
end
```

在后台，需要给定的模块，然后调用其回调，允许模块将一些代码注入当前上下文。某些模块（例如，上面的 ，但也和 ）使用此机制用一些基本行为填充模块，您的模块旨在覆盖或完成这些行为。
`use` `__using__/1` `ExUnit.Case` `Supervisor` `GenServer`

一般来说，以下模块：

```
defmodule Example do
  use Feature, option: :value
end
```

被编译成

```
defmodule Example do
  require Feature
  Feature.__using__(option: :value)
end
```

由于允许任何代码运行，因此如果不阅读其文档，我们就无法真正知道使用模块的副作用。因此，请谨慎使用此功能，并且仅在严格要求的情况下使用。不要在 or 会做的事情上使用。 `use` `use` `import` `alias`

了解别名

在这一点上，您可能想知道：Elixir 别名到底是什么，它是如何表示的？

Elixir 中的别名是一个大写的标识符（如，等），在编译过程中被转换为原子。例如，别名默认转换为原子：
`String` `Keyword` `String` `:"Elixir.String"`

```
iex> is_atom(String)
true
iex> to_string(String)
"Elixir.String"
iex> :"Elixir.String" == String
true
```

通过使用该指令，我们正在更改别名扩展到的原子。`alias/2`

别名扩展到原子，因为在Erlang VM（以及随后的Elixir）模块中总是由原子表示：

```
iex> List.flatten([1, [2], 3])
[1, 2, 3]
iex> :"Elixir.List".flatten([1, [2], 3])
[1, 2, 3]
```

这就是我们用来调用 Erlang 模块的机制：

```
iex> :lists.flatten([1, [2], 3])
[1, 2, 3]
```

模块嵌套

现在我们已经讨论了别名，我们可以谈谈嵌套以及它在 Elixir 中的工作原理。请考虑以下示例：

```
defmodule Foo do
  defmodule Bar do
  end
end
```

上面的示例将定义两个模块：和 。第二个可以在内部访问，只要它们在相同的词汇范围内。`Foo` `Foo.Bar` `Bar` `Foo`

如果稍后将模块移到模块定义之外，则必须使用其全名 （） 引用它，或者必须使用上面讨论的指令设置别名。`Bar` `Foo` `Foo.Bar` `alias`

注意：在Elixir中，在能够定义模块之前，您不必定义模块，因为它们实际上是独立的。以上也可以写成：`Foo` `Foo.Bar`

```
defmodule Foo.Bar do
end

defmodule Foo do
```

```
  alias Foo.Bar
  # Can still access it as `Bar`
end
```

为嵌套模块设置别名不会将父模块纳入作用域。请考虑以下示例：

```
defmodule Foo do
  defmodule Bar do
    defmodule Baz do
    end
  end
end

alias Foo.Bar.Baz
# The module `Foo.Bar.Baz` is now available as `Baz`
# However, the module `Foo.Bar` is *not* available as
`Bar`
```

正如我们将在后面的章节中看到的，别名在宏中也起着至关重要的作用，以确保它们是卫生的。

多别名 / 导入 / 要求 / 使用

可以同时使用、、或多个模块。一旦我们开始嵌套模块，这特别有用，这在构建 Elixir 应用程序时很常见。例如，假设您有一个应用程序，其中所有模块都嵌套在 下，您可以为模块设置别名，并立即如下所示：
`alias` `import` `require` `use` `MyApp` `MyApp.Foo` `MyApp.Bar` `MyApp.Baz`

```
alias MyApp.{Foo, Bar, Baz}
```

至此，我们完成了对长生不老药模块的游览。下一个要介绍的主题是模块属性。

← 上一页　返回页首　下一→