开始

结构体

在第 7 章中，我们了解了地图：

```
iex> map = %{a: 1, b: 2}
%{a: 1, b: 2}
iex> map[:a]
1
iex> %{map | a: 3}
%{a: 3, b: 2}
```

结构是在提供编译时检查和默认值的映射之上构建的扩展。

定义结构

要定义结构，请使用构造：`defstruct`

```
iex> defmodule User do
...>   defstruct name: "John", age: 27
...> end
```

用于 的关键字列表定义了结构将具有的字段及其默认值。`defstruct`

结构采用定义它们的模块的名称。在上面的示例中，我们定义了一个名为 .`User`

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

现在，我们可以使用类似于创建映射的语法来创建结构：`User`

```
iex> %User{}
%User{age: 27, name: "John"}
iex> %User{name: "Jane"}
%User{age: 27, name: "Jane"}
```

> *注：如果在单独的文件中定义了结构，则可以在 IEx 中编译该文件，然后再继续运行 。请注意，如果您直接在文件中尝试上述示例，您可能会收到一条错误消息，因为定义何时解析。`c "file.exs"` `the struct was not yet defined`*

结构提供*编译时*保证，仅允许结构中存在通过定义的字段：`defstruct`

```
iex> %User{oops: :field}
** (KeyError) key :oops not found expanding struct:
User.__struct__/1
```

访问和更新结构

当我们讨论地图时，我们展示了如何访问和更新地图的字段。相同的技术（和相同的语法）也适用于结构：

```
iex> john = %User{}
%User{age: 27, name: "John"}
iex> john.name
"John"
iex> jane = %{john | name: "Jane"}
%User{age: 27, name: "Jane"}
iex> %{jane | oops: :field}
** (KeyError) key :oops not found in: %User{age: 27, name:
"Jane"}
```

使用更新语法 （） 时，VM 知道不会向结构中添加任何新键，从而允许下面的映射在内存中共享其结构。在上面的示例中，两者在内存中共享相同的键结构。`|` `john` `jane`

结构还可用于模式匹配，既用于匹配特定键的值，也用于确保匹配值是与匹配值类型相同的结构。

```
iex> %User{name: name} = john
%User{age: 27, name: "John"}
iex> name
"John"
iex> %User{} = %{}
** (MatchError) no match of right hand side value: %{}
```

结构是下面的裸地图

在上面的示例中，模式匹配有效，因为结构下方是具有一组固定字段的裸映射。作为映射，结构存储一个名为“特殊”字段，该字段保存结构的名称：`__struct__`

```
iex> is_map(john)
true
iex> john.__struct__
User
```

请注意，我们将结构称为裸映射，因为为映射实现的协议都不适用于结构。例如，您既不能枚举也不能访问结构：

```
iex> john = %User{}
%User{age: 27, name: "John"}
iex> john[:name]
** (UndefinedFunctionError) function User.fetch/2 is
undefined (User does not implement the Access behaviour)
             User.fetch(%User{age: 27, name: "John"},
:name)
iex> Enum.each(john, fn {field, value} -> IO.puts(value)
end)
** (Protocol.UndefinedError) protocol Enumerable not
implemented for %User{age: 27, name: "John"} of type User
(a struct)
```

但是，由于结构只是映射，因此它们与模块中的函数一起使用：`Map`

```
iex> jane = Map.put(%User{}, :name, "Jane")
%User{age: 27, name: "Jane"}
iex> Map.merge(jane, %User{name: "John"})
%User{age: 27, name: "John"}
```

```
iex> Map.keys(jane)
[:__struct__, :age, :name]
```

结构和协议为 Elixir 开发人员提供了最重要的功能之一：数据多态性。这就是我们将在下一章中探讨的内容。

默认值和必需的键

如果在定义结构时未指定默认键值，将假定：`nil`

```
iex> defmodule Product do
...>   defstruct [:name]
...> end
iex> %Product{}
%Product{name: nil}
```

您可以定义一个结构，将这两个字段与显式默认值和隐式值组合在一起。在这种情况下，您必须首先指定隐式默认为 nil 的字段：`nil`

```
iex> defmodule User do
...>   defstruct [:email, name: "John", age: 27]
...> end
iex> %User{}
%User{age: 27, email: nil, name: "John"}
```

以相反的顺序执行此操作将引发语法错误：

```
iex> defmodule User do
...>   defstruct [name: "John", age: 27, :email]
...> end
** (SyntaxError) iex:107: unexpected expression after
keyword list. Keyword lists must always come last in lists
and maps.
```

您还可以强制在通过 module 属性创建结构时必须指定某些键：`@enforce_keys`

```
iex> defmodule Car do
...>   @enforce_keys [:make]
...>   defstruct [:model, :make]
```

```
...> end
iex> %Car{}
** (ArgumentError) the following keys must also be given
when building struct Car: [:make]
    expanding struct: Car.__struct__/1
```

强制实施密钥提供了简单的编译时保证，以帮助开发人员构建结构。它不会在更新时强制执行，并且不提供任何类型的值验证。