开始

基本类型

在本章中，我们将更多地了解 Elixir 的基本类型：整数，浮点数，布尔值，原子，字符串，列表和元组。一些基本类型是：

```
iex> 1          # integer
iex> 0x1F       # integer
iex> 1.0        # float
iex> true       # boolean
iex> :atom      # atom / symbol
iex> "elixir"   # string
iex> [1, 2, 3]  # list
iex> {1, 2, 3}  # tuple
```

基本算术

打开并键入以下表达式：`iex`

```
iex> 1 + 2
3
```

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

```
iex> 5 * 5
25
iex> 10 / 2
5.0
```

请注意，返回了一个浮点数而不是一个整数。这是意料之中的。在 Elixir 中，运算符总是返回一个浮点数。如果要执行整数除法或获取除法余数，可以调用 and 函数：`10 / 2` `5.0` `5` `/` `div` `rem`

```
iex> div(10, 2)
5
iex> div 10, 2
5
iex> rem 10, 3
1
```

请注意，Elixir 允许你在调用至少一个参数的命名函数时去掉括号。此功能在编写声明和控制流构造时提供了更简洁的语法。然而，Elixir 开发人员通常更喜欢使用括号。

Elixir 还支持输入二进制，八进制和十六进制数字的快捷方式符号：

```
iex> 0b1010
10
iex> 0o777
511
iex> 0x1F
31
```

浮点数需要一个点，后跟至少一位数字，并且还支持科学记数法：`e`

```
iex> 1.0
1.0
iex> 1.0e-10
1.0e-10
```

Elixir 中的浮点数是 64 位双精度。

可以调用函数来获取最接近给定浮点数的整数，也可以调用函数来获取浮点数的整数部分。`round` `trunc`

```
iex> round(3.58)
4
iex> trunc(3.58)
3
```

识别功能和文档

Elixir 中的功能由其名称和 Arity 标识。函数的 arity 描述了函数采用的参数数量。从这一点开始，我们将在整个文档中使用函数名称及其 arity 来描述函数。 标识命名并接受参数的函数，而标识具有相同名称但 arity 为 `.trunc/1` `trunc` `1` `trunc/2` `2`

我们还可以使用此语法来访问文档。Elixir shell 定义了函数，你可以用它来访问任何函数的文档。例如，键入将打印函数的文档： `h` `h` `trunc/1` `trunc/1`

```
iex> h trunc/1
                                def trunc()

Returns the integer part of number.
```

`h trunc/1` 有效，因为它是在模块中定义的。模块中的所有函数都会自动导入到我们的命名空间中。大多数情况下，在查找给定函数的文档时，您还将包含模块名称： `Kernel` `Kernel`

```
iex> h Kernel.trunc/1
                                def trunc()

Returns the integer part of number.
```

您可以使用模块 + 函数来查找任何内容，包括运算符（try）。不带参数的调用将显示 的文档，其中定义了 和其他功能。 `h` `Kernel.+/2` `h` `IEx.Helpers` `h`

布尔 值

长生不老药支持和布尔值： `true` `false`

```
iex> true
true
iex> true == false
false
```

Elixir 提供了一堆谓词函数来检查值类型。例如，该函数可用于检查值是否为布尔值：`is_boolean/1`

```
iex> is_boolean(true)
true
iex> is_boolean(1)
false
```

还可以使用 或分别检查参数是整数、浮点数还是两者之一。
`is_integer/1` `is_float/1` `is_number/1`

原子

原子是一个常量，其值是它自己的名称。其他一些语言称这些符号。它们通常可用于枚举不同的值，例如：

```
iex> :apple
:apple
iex> :orange
:orange
iex> :watermelon
:watermelon
```

如果原子的名称相等，它们就是相等的。

```
iex> :apple == :apple
true
iex> :apple == :orange
false
```

通常，它们用于表示操作的状态，方法是使用 和 等值。`:ok` `:error`

布尔值和原子也是原子：`true` `false`

```
iex> true == :true
true
iex> is_atom(false)
true
iex> is_boolean(:false)
true
```

Elixir 允许你跳过原子的前导，和。`:` `false` `true` `nil`

最后，Elixir 有一个称为别名的结构，我们将在后面探讨。别名以大写开头，也是原子：

```
iex> is_atom(Hello)
true
```

字符串

Elixir 中的字符串由双引号分隔，并以 UTF-8 编码：

```
iex> "hellö"
"hellö"
```

> *注意：如果您在 Windows 上运行，则默认情况下您的终端可能不使用 UTF-8。您可以通过在进入 IEx 之前运行来更改当前会话的编码。* `chcp 65001`

Elixir 还支持字符串插值：

```
iex> string = :world
iex> "hellö #{string}"
"hellö world"
```

字符串中可以有换行符。您可以使用转义序列引入它们：

```
iex> "hello
...> world"
"hello\nworld"
```

```
iex> "hello\nworld"
"hello\nworld"
```

您可以使用模块中的函数打印字符串：`IO.puts/1` `IO`

```
iex> IO.puts("hello\nworld")
hello
world
:ok
```

请注意，该函数在打印后返回原子。`IO.puts/1` `:ok`

Elixir 中的字符串在内部由称为二进制文件的连续字节序列表示：

```
iex> is_binary("hellö")
true
```

我们还可以获取字符串中的字节数：

```
iex> byte_size("hellö")
6
```

请注意，该字符串中的字节数为 6，即使它有 5 个字素。这是因为字素“ö”需要 2 个字节才能用 UTF-8 表示。我们可以使用以下函数根据字素的数量获取字符串的实际长度：`String.length/1`

```
iex> String.length("hellö")
5
```

字符串模块包含一堆函数，这些函数对 Unicode 标准中定义的字符串进行操作：

```
iex> String.upcase("hellö")
"HELLÖ"
```

匿名函数

Elixir 还提供匿名功能。匿名函数允许我们存储和传递可执行代码，就好像它是整数或字符串一样。它们由关键字和： `fn` `end`

```
iex> add = fn a, b -> a + b end
#Function<12.71889879/2 in :erl_eval.expr/5>
iex> add.(1, 2)
3
iex> is_function(add)
true
```

在上面的示例中，我们定义了一个匿名函数，该函数接收两个参数和 ，并返回。参数始终位于左侧，要执行的代码始终位于右侧。匿名函数存储在变量 中。 `a` `b` `a + b` `->` `add`

我们可以通过向匿名函数传递参数来调用它。请注意，变量和括号之间需要一个点 （） 才能调用匿名函数。点确保调用与变量匹配的匿名函数和命名函数之间没有歧义。在处理模块和函数时，我们将编写自己的命名函数。现在，请记住，Elixir明确区分了匿名函数和命名函数。 `.` `add` `add/2`

Elixir 中的匿名函数也由它们收到的参数数量来识别。我们可以通过以下方式检查函数是否具有任何给定的 arity： `is_function/2`

```
# check if add is a function that expects exactly 2
arguments
iex> is_function(add, 2)
true
# check if add is a function that expects exactly 1
argument
iex> is_function(add, 1)
false
```

最后，匿名函数还可以访问定义函数时作用域中的变量。这通常称为闭包，因为它们在其范围内关闭。让我们定义一个新的匿名函数，它使用我们之前定义的匿名函数： `add`

```
iex> double = fn a -> add.(a, a) end
#Function<6.71889879/1 in :erl_eval.expr/5>
```

```
iex> double.(2)
4
```

在函数内部赋值的变量不会影响其周围环境：

```
iex> x = 42
42
iex> (fn -> x = 0 end).()
0
iex> x
42
```

（链接）列表

Elixir 使用方括号来指定值列表。值可以是任何类型：

```
iex> [1, 2, true, 3]
[1, 2, true, 3]
iex> length [1, 2, 3]
3
```

可以分别使用 and 运算符连接或减去两个列表：`++/2` `--/2`

```
iex> [1, 2, 3] ++ [4, 5, 6]
[1, 2, 3, 4, 5, 6]
iex> [1, true, 2, false, 3, true] -- [true, false]
[1, 2, 3, true]
```

列表运算符从不修改现有列表。连接到列表或从列表中删除元素将返回一个新列表。我们说Elixir数据结构是不*可*变的。不可变性的一个优点是它会导致更清晰的代码。您可以自由传递数据，并保证没有人会在内存中改变它 - 只转换它。

在整个教程中，我们将讨论很多关于列表的头部和尾部的内容。头部是列表的第一个元素，尾部是列表的其余部分。可以使用函数和。让我们为变量分配一个列表并检索其头部和尾部：`hd/1` `tl/1`

```
iex> list = [1, 2, 3]
iex> hd(list)
```

```
1
iex> tl(list)
[2, 3]
```

获取空列表的头部或尾部会引发错误：

```
iex> hd([])
** (ArgumentError) argument error
```

有时，您将创建一个列表，它将返回前面带有 的引号值。例如：`~c`

```
iex> [11, 12, 13]
~c"\v\f\r"
iex> [104, 101, 108, 108, 111]
~c"hello"
```

在 v1.15 之前的 Elixir 版本中，这可能会显示为单引号：

```
iex> [104, 101, 108, 108, 111]
'hello'
```

当 Elixir 看到可打印的 ASCII 数字列表时，Elixir 会将其打印为字符列表（字面意思是字符列表）。字符列表在与现有 Erlang 代码交互时非常常见。每当您在 IEx 中看到一个值并且不太确定它是什么时，您可以使用来检索有关它的信息：`i/1`

```
iex> i ~c"hello"
Term
  i ~c"hello"
Data type
  List
Description
  ...
Raw representation
  [104, 101, 108, 108, 111]
Reference modules
  List
Implemented protocols
  ...
```

请记住，单引号和双引号表示在 Elixir 中并不等同，因为它们由不同的类型表示：

```
iex> 'hello' == "hello"
false
iex> 'hello' == ~c"hello"
true
```

单引号是字符，双引号是字符串。我们将在“二进制文件、字符串和字符列表”一章中详细讨论它们。

元组

Elixir 使用大括号来定义元组。与列表一样，元组可以保存任何值：

```
iex> {:ok, "hello"}
{:ok, "hello"}
iex> tuple_size {:ok, "hello"}
2
```

元组在内存中连续存储元素。这意味着通过索引访问元组元素或获取元组大小是一项快速操作。索引从零开始：

```
iex> tuple = {:ok, "hello"}
{:ok, "hello"}
iex> elem(tuple, 1)
"hello"
iex> tuple_size(tuple)
2
```

也可以使用以下命令将元素放在元组的特定索引处：`put_elem/3`

```
iex> tuple = {:ok, "hello"}
{:ok, "hello"}
iex> put_elem(tuple, 1, "world")
{:ok, "world"}
iex> tuple
{:ok, "hello"}
```

请注意，返回了一个新元组。未修改变量中存储的原始元组。与列表一样，元组也是不可变的。元组上的每个操作都会返回一个新元组，它永远不会更改给定的元组。`put_elem/3` `tuple`

列表还是元组？

列表和元组有什么区别？

列表作为链表存储在内存中，这意味着列表中的每个元素都保存其值并指向下一个元素，直到到达列表的末尾。这意味着访问列表的长度是一个线性操作：我们需要遍历整个列表才能弄清楚它的大小。

同样，列表串联的性能取决于左侧列表的长度：

```
iex> list = [1, 2, 3]
[1, 2, 3]

# This is fast as we only need to traverse `[0]` to
prepend to `list`
iex> [0] ++ list
[0, 1, 2, 3]

# This is slow as we need to traverse `list` to append 4
iex> list ++ [4]
[1, 2, 3, 4]
```

另一方面，元组连续存储在内存中。这意味着获取元组大小或按索引访问元素的速度很快。但是，向元组更新或添加元素的成本很高，因为它需要在内存中创建新的元组：

```
iex> tuple = {:a, :b, :c, :d}
{:a, :b, :c, :d}
iex> put_elem(tuple, 2, :e)
{:a, :b, :e, :d}
```

请注意，这仅适用于元组本身，不适用于其内容。例如，更新元组时，除已替换的条目外，旧元组和新元组之间共享所有条目。换句话说，Elixir 中的元组和列表能够共享其内容。这减少了语言需要执行的内存分配量，并且由于语言的不可变语义才有可能。

这些性能特征决定了这些数据结构的使用。元组的一个非常常见的用例是使用它们从函数返回额外信息。例如， 是可用于读取文件内容的函数。它返回一个元组： `File.read/1`

```
iex> File.read("path/to/existing/file")
{:ok, "... contents ..."}
iex> File.read("path/to/unknown/file")
{:error, :enoent}
```

如果给定的路径存在，则返回一个元组，其中原子作为第一个元素，文件内容作为第二个元素。否则，它将返回一个包含和错误描述的元组。
`File.read/1` `:ok` `:error`

大多数时候，Elixir 会指导你做正确的事情。例如，有一个访问元组项的函数，但没有内置的列表等效项： `elem/2`

```
iex> tuple = {:ok, "hello"}
{:ok, "hello"}
iex> elem(tuple, 1)
"hello"
```

在计算数据结构中的元素时，Elixir 还遵守一个简单的规则：如果操作处于恒定时间（即值是预先计算的）或操作是线性的（即计算长度随着输入的增长而变慢），则命名函数。作为助记符，“长度” 和 “线性” 都以 “l” 开头。 `size` `length`

例如，到目前为止，我们已经使用了 4 个计数函数：（用于字符串中的字节数）、（用于元组大小）、（用于列表长度）和（用于字符串中的字素数）。我们用来获取字符串中的字节数 —— 一个便宜的操作。另一方面，检索 Unicode 字素的数量会使用 ，并且可能很昂贵，因为它依赖于整个字符串的遍历。
`byte_size/1` `tuple_size/1` `length/1` `String.length/1` `byte_size` `String.length`

Elixir 还提供，和作为数据类型（通常用于进程通信），我们将在讨论进程时快速浏览它们。现在，让我们看一下基本类型附带的一些基本运算符。 `Port` `Reference` `PID`

← 上一页　返回页首　下一→

有什么不对吗？ 在 GitHub 上编辑此页面。

© 2012–2023 长生不老药团队。

Elixir和Elixir标志是The Elixir Team 的注册商标。