开始

递归

通过递归循环

由于不变性，Elixir 中的循环（与任何函数式编程语言一样）的编写方式与命令式语言不同。例如，在像 C 这样的命令式语言中，人们会这样写：

```
for(i = 0; i < sizeof(array); i++) {
  array[i] = array[i] * 2;
}
```

在上面的例子中，我们正在改变数组和变量。然而，Elixir 中的数据结构是不可变的。出于这个原因，函数式语言依赖于递归：递归调用函数，直到达到阻止递归操作继续的条件。在此过程中不会发生任何数据突变。考虑以下示例，该示例将字符串打印任意次数：`i`

```
defmodule Recursion do
  def print_multiple_times(msg, n) when n > 0 do
    IO.puts(msg)
    print_multiple_times(msg, n - 1)
  end

  def print_multiple_times(_msg, 0) do
    :ok
  end
end

Recursion.print_multiple_times("Hello!", 3)
# Hello!
# Hello!
```

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

```
# Hello!
:ok
```

与 类似，一个函数可能有许多子句。当传递给函数的参数与子句的参数模式匹配并且其守卫的计算结果为 .`case` `true`

在上面的示例中最初调用 时，参数等于 。`print_multiple_times/2` `n` `3`

第一句有一个保护，说“当且仅当大于” 时使用此定义。由于是这种情况，它打印 然后调用自己 pass （） 作为第二个参数。`n` `0` `msg` `n - 1` `2`

现在我们再次执行相同的函数，从第一个子句开始。给定第二个参数 ，仍然大于 0，我们打印消息并再次调用自己，现在第二个参数设置为 。然后我们最后一次打印消息并调用 ，再次从顶部开始。
`n` `1` `print_multiple_times("Hello!", 0)`

当第二个参数为零时，守卫的计算结果为 false，并且第一个函数子句不会执行。然后，Elixir 继续尝试下一个函数子句，该子句明确匹配了其中的情况。此子句也称为终止子句，通过将消息参数分配给变量来忽略 message 参数并返回 atom 。`n > 0` `n` `0` `_msg` `:ok`

最后，如果你传递的参数与任何子句都不匹配，Elixir 会引发一个：
`FunctionClauseError`

```
iex> Recursion.print_multiple_times "Hello!", -1
** (FunctionClauseError) no function clause matching in
Recursion.print_multiple_times/2

    The following arguments were given to
Recursion.print_multiple_times/2:

        # 1
        "Hello!"

        # 2
        -1

    iex:1: Recursion.print_multiple_times/2
```

简化和映射算法

现在让我们看看如何使用递归的力量对数字列表求和：

```
defmodule Math do
  def sum_list([head | tail], accumulator) do
    sum_list(tail, head + accumulator)
  end

  def sum_list([], accumulator) do
    accumulator
  end
end

IO.puts Math.sum_list([1, 2, 3], 0) #=> 6
```

我们使用列表和初始值作为参数进行调用。我们将尝试每个子句，直到找到一个根据模式匹配规则匹配的子句。在这种情况下，列表匹配绑定到和；设置为 。`sum_list` `[1, 2, 3]` `0` `[1, 2, 3]` `[head | tail]` `head` `1` `tail` `[2, 3]` `accumulator` `0`

然后，我们将列表的头部添加到累加器中，并以递归方式再次调用，将列表的尾部作为其第一个参数传递。尾部将再次匹配，直到列表为空，如下所示：`head + accumulator` `sum_list` `[head | tail]`

```
sum_list [1, 2, 3], 0
sum_list [2, 3], 1
sum_list [3], 3
sum_list [], 6
```

当列表为空时，它将匹配返回 的最终结果的最后一个子句。`6`

获取列表并将其*减少*到一个值的过程称为 *reduce 算法*，是函数式编程的核心。

如果我们想要将列表中的所有值加倍怎么办？

```
defmodule Math do
  def double_each([head | tail]) do
    [head * 2 | double_each(tail)]
  end

  def double_each([]) do
```

```
    []
  end
end
```

```
$ iex math.exs
```

```
iex> Math.double_each([1, 2, 3]) #=> [2, 4, 6]
```

在这里，我们使用递归遍历列表，将每个元素加倍并返回一个新列表。获取列表并*在其*上进行映射的过程称为*映射算法*。

递归和尾部调用优化是Elixir的重要组成部分，通常用于创建循环。但是，在Elixir中编程时，您很少会使用上述递归来操作列表。

我们将在下一章中看到的`枚举`模块已经为使用列表提供了许多便利。例如，上面的例子可以写成：

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

或者，使用捕获语法：

```
iex> Enum.reduce([1, 2, 3], 0, &+/2)
6
iex> Enum.map([1, 2, 3], &(&1 * 2))
[2, 4, 6]
```

让我们更深入地看看，当我们在它的时候，它的懒惰对应物，`.Enumerable` `Stream`

← 上一页　返回页首　下一→