# 第 5 章

# 函数

很高兴看到你结束了第 1 级的学习。从本章开始，你将学习更高级的内容，包括处理错误、包管理、机器学习。

在本章中，你将学习：

- 什么是函数(functions)，如何使用函数；
- 函数是如何帮助用户减少错误，方便代码理解与维护的；
- 声明和调用函数；
- 具有返回值的函数；
- 具有可选关键字参数的函数；
- 在数组上应用函数；
- 通用函数；
- 递归地使用函数。

编写函数是所有编程语言最重要的特性，它有助于模块化我们创建的应用程序，使我们可以在需要进行修改和升级时回到那里。

## 5.1 函数及其使用

在进入下一级之前，你首先要了解一个基本概念——函数。编程中的函数是为完成一项任务或解决一个问题而编写的一组指令。函数会被命名，使得你可以任意多次地调用它们，并让它们执行程序中的任务；通过函数可以在整个程序中重用代码。例如，参考第 4 章构建的应用程序，它可以使你掌握借给朋友的所有物品的信息。

## 5.2 函数有助于减少错误，方便代码维护

请回忆一下，在第 4 章中，我们首先使用数组创建了一个应用程序，然后将其修改为使用字典运行。如果你仔细查看“借出”和“收回”的代码块，就会发现我们重复了前几行代码以询问用户谈论的是哪个朋友。这样做是有问题的，主要原因是会导致不可维护或不可扩展。我知道这是目前为止你编写的规模最大的应用程序，但你还会编写规模更大的应用程序。如果你要在所有应用程序中都重复这样的代码，那么你的编程之旅将不会走得太远。

想象一下，若要对这段代码做一个小小的改动，那么你就必须在很多地方做出修改！这正是函数旨在解决的问题，函数允许你封装或捆绑代码块，并在代码中的其他地方引用或调用这些代码块。

在继续之前，我要向你展示一个函数的简单示例。假设你希望构建一个应用程序，要求用户输入包含三条信息的一个列表：

① 谁是你最好的朋友；

② 谁是你的朋友；

③ 你不喜欢谁。

在这里，假设你可能有不止一个最好的朋友。每个字段都可以有任意数量的输入，因此，你当然需要使用数组。根据目前为止学到的知识，你可能会编写这样的代码：

```
println("Who are your best friends?")
best_friends = []
while true
    print("Name: ")
    user_input = readline()
    if user_input == "done"
        break
    end
    push!(best_friends, user_input)
end
println("Who are your friends? ")
friends = []
while true
    print("Name: ")
    user_input = readline()
    if user_input == "done"
        break
    end
    push!(friends, user_input)
end
println("Who do you not like? ")
not_friends = []
while true
    print("Name: ")
    user_input = readline()
    if user_input == "done"
        break
    end
    push!(not_friends, user_input)
end
```

这看起来代码量确实很少，但重复性非常高。尽管如此，它确实有

效，如果你要运行它，你可以这样与它交互：

```
Who are your best friends?
Name: Patrick
Name: Anna
Name: done
Who are your friends?
Name: Frank
Name: Jane
Name: done
Who do you not like?
Name: done
```

就这样，你有了 3 个数组，best_friends 数组包含

```
["Patrick", "Anna"]
```

friends 数组包含

```
["Frank", "Jane"]
```

而 not_friends 数组为空。

下面我将展示如何使用函数优化该示例，将代码行减少 2/3！但在此之前，你先要理解函数的基本概念。为此，让我们从示例中获取帮助。

## 5.3 声明和调用函数

假设你要创建一个代码块，它可以获取一个整数并显示该数字加 1 后的值。使用 Julia 可以这样做：

```
function add_one(original_number::Int64)
    println(original_number + 1)
end
```

哇！这是一个函数！让我们从第 1 行开始分解代码，我们将这行称

为函数声明行：

```
function add_one(original_number::Int64)
```

| 代码成分 | 作　用 |
| --- | --- |
| function | 第一个关键字告诉 Julia 要创建一个函数，它是 Julia 中的一个保留字，这意味着你不能将它用于任何其他目的，比如命名你的变量，你只能使用它开始声明你的函数 |
| add_one | 在本例中，这告诉 Julia 函数的名称是 add_one，但它可能是任何东西。任何一个有效的变量名都可以是函数的名称，例如 calc_area、find_least_number、squareOf |
| original_number | 这是一个由你决定的变量名，被称为参数，一个函数可能没有参数或有一个或多个参数。你将用这个名称引用你在函数的代码中获得的值 |
| :: Int64 | “::”部分的意思是“...的类型”，其后面是 Int64，它指定了函数接收的变量的类型，它们(original_number:: Int64)可以被读作“函数接收一个 Int64 类型的名为 original_number 的参数” |

从第 1 行代码之后输入的所有代码直到与第 1 行对应(或匹配)的 end 关键字都是函数的一部分。在本例中，在第 1 行和 end 关键字(第 3 行代码)之间只有一行代码。

该函数内的所有代码行都可以访问传递给该函数的参数。例如，如果你查看第 2 行代码：

```
println(original_number + 1)
```

虽然我们尚未设置 original_number 的值，但你仍然可以像访问任何整数一样访问它。这是因为当函数被调用时，original_number 将会被填充一个值，然后函数内部的代码将被运行，一旦遇到 end 关键字，original_number 变量将被删除，我们就会立即回到开始处。

现在，运行你的代码，这令人非常兴奋！但是……什么都没有发生，它什么也不打印。你知道这是为什么吗？

这是因为我们已经定义了一个代码块，它应该接收一些输入，并对该输入进行操作。但是，我们从未对该函数进行任何输入。

因此，在函数的末尾，请输入以下代码：

```
add_one(3)
```

现在，在运行代码时，应该会看到以下输出：

```
4
```

非常棒！但是请稍等，在函数的第 2 行，我们访问 original_number 变量并对其加 1，但我们不会将此新值存储回 original_number 变量中，这意味着该变量的值没有改变。所以，如果要打印 original_number，我们应该会看到 3，对吧？在这行代码之后：

```
add_one(3)
```

输入以下代码：

```
println(original_number)
```

非常棒！继续运行你的程序，你应该会看到这样的输出：

```
ERROR: LoadError: UndefVarError: original_number not defined
Stacktrace:
[1] top-level scope at none:0
[2] include at ./boot.jl:326 [inlined]
[3] include_relative(::Module, ::String) at ./loading.jl:1038
[4] include(::Module, ::String) at ./sysimg.jl:29
[5] exec_options(::Base.JLOptions) at ./client.jl:267
[6] _start() at ./client.jl:436
```

等等，这是什么？出现了错误，为什么会是这样呢？

还记得我提到过这个变量在 end 关键字后被删除了吗？这就是出现这个错误的原因。我们试图打印一个不存在的变量，但它“超出了作

用域”。让我来解释一下。

看看以下代码：

```
1   function add_one(original_number::Int64)
2       . . .
3       . . .
.       . . .
.       println(original_number)
.       . . .
98      . . .
99      . . .
100 end

add_one(3)
println(original_number)
```

唯一可以访问 original_number 变量的代码行是 function 关键字与其对应的 end 行之间的代码行。假设这是一个包含 100 行代码的程序，可以在第 2～99 行中使用函数。当函数结束时之所以无法访问该变量，是因为该变量是局部变量。

这应该具有相同的输出：

```
function add_one(original_number::Int64)
    added_number = original_number + 1
    println(added_number)
end
add_one(3)
println(added_number)
```

代码的行为应该相同，它将打印一个 4，但仍然会给出一个错误提示。

你可能会问：“我们并非打印函数的参数，而是打印一个变量，为什么仍然会提示错误？”这是因为我们在函数内部定义了这个变量，当函数结束时，函数内部的所有声明都将不复存在，数据可以流入（通过参数），

但无法输出。

## 5.4 具有返回值的函数

只接收输入的函数很简单，我们还希望函数能够处理我们输入的数据，然后返回处理结果。所以这次我们不让函数加 1 并打印出值，而是让它加 1 并返回值，然后在函数外打印出值。

输入以下代码：

```
function add_one(original_number::Int64)::Int64
    return original_number + 1
end
```

你可能会立即注意到第 1 行中的不同：在函数声明的末尾，我们添加了::Int64。这会告诉 Julia，这个函数将返回一个 Int64 类型的值。换句话说，当有人使用该函数时，其值将始终被解析为一个整数。

现在，你可以像过去使用其他函数一样使用该函数。例如：

```
println(add_one(3))
```

请注意，调用 add_one 本身就是一个表达式，而该表达式将解析为一个整数，然后将该整数传递给 println，最终结果会被打印到屏幕上。所以，如果运行该应用程序，应该会看到输出为

```
4
```

你也可以通过使用以下变量进行打印：

```
input = 10
result = add_one(input)
println(result)
```

这将会打印出

```
11
```

因此，一开始看起来复杂难懂的函数，现在已经被揭开了神秘面纱，现在你也知道如何使用它们了。但要真正理解函数的工作原理，我们还必须构建一些功能更强大的程序。让我们看看第一个例子，请注意，这是我们原来的代码：

```
println("Who are your best friends?")
best_friends = []
while true
    print("Name: ")
    user_input = readline()
    if user_input == "done"
        break
    end
    push!(best_friends, user_input)
end
println("Who are your friends? ")
friends = []
while true
    print("Name: ")
    user_input = readline()
    if user_input == "done"
        break
    end
    push!(friends, user_input)
end
println("Who do you not like? ")
not_friends = []
while true
    print("Name: ")
    user_input = readline()
    if user_input == "done"
        break
    end
    push!(not_friends, user_input)
end
```

首先，让我们找出重复的代码（本例中非常简单）：

```
println("Who are your best friends?")
best_friends = []
while true
    print("Name: ")
    user_input = readline()
    if user_input == "done"
        break
    end
    push!(best_friends, user_input)
end
println("Who are your friends? ")
friends = []
while true
    print("Name: ")
    user_input = readline()
    if user_input == "done"
        break
    end
    push!(friends, user_input)
end
println("Who do you not like? ")
not_friends = []
while true
    print("Name: ")
    user_input = readline()
    if user_input == "done"
        break
    end
    push!(not_friends, user_input)
end
```

好了，我们已经确定了要封装在一个函数中的模式了，以下就是它所做的事情：

① 创建一个空数组（字符串类型）；

② 创建一个无限循环；