En Elixir agrupamos varias funciones en módulos. Ya hemos usado muchos módulos diferentes en los capítulos anteriores, como el módulo de cadena:

```
iex> String.length("hello")
5
```

Para crear nuestros propios módulos en Elixir, utilizamos la macro defmodule. Usamos la macro def para definir funciones en ese módulo:

```
iex> defmodule Math do
...>   def sum(a, b) do
...>     a + b
...>   end
...> end

iex> Math.sum(1, 2)
3
```

Compilación

Ya es hora de que aprendamos cómo compilar el código de Elixir y también cómo ejecutar los scripts de Elixir.

La mayoría de las veces es conveniente escribir módulos en archivos para que puedan compilarse y reutilizarse. Supongamos que tenemos un archivo llamado math.ex con los siguientes contenidos:

```
defmodule Math do
  def sum(a, b) do
    a + b
  end
end
```

Este archivo se puede compilar usando elixirc:

```
$ elixirc math.ex
```

Esto generará un archivo llamado Elixir.Math.beam que contiene el código de bytes para el módulo definido. Si iniciamos iex nuevamente, la definición de nuestro módulo estará disponible (siempre que iex se inicie en el mismo directorio en el que se encuentra el archivo de código de bytes):

```
iex> Math.sum(1, 2)
3
```

Los proyectos de elixir generalmente se organizan en tres directorios:

- ebin – contiene el bytecode compilado
- lib: contiene código de elixir (generalmente archivos .ex)
- prueba: contiene pruebas (generalmente archivos .exs)

Cuando trabaje en proyectos reales, la herramienta de compilación llamada mix será responsable de compilar y configurar las rutas adecuadas para usted. Con fines de aprendizaje, Elixir también es compatible con un modo de script que es más flexible y no genera ningún artefacto compilado.

Modo Scripted

Además de la extensión de archivo Elixir .ex, Elixir también admite archivos .exs para secuencias de comandos. Elixir trata ambos archivos exactamente de la misma manera, la única diferencia está en la intención. Los archivos .ex están destinados a ser compilados mientras que los archivos .exs se utilizan para la creación de scripts. Cuando se ejecutan, ambas extensiones compilan y cargan sus módulos en la memoria, aunque solo los archivos .ex escriben su código de bytes en el disco en el formato de archivos .beam.

Por ejemplo, podemos crear un archivo llamado math.exs:

```
defmodule Math do
  def sum(a, b) do
    a + b
  end
end

IO.puts Math.sum(1, 2)
```

Y ejecutarlo como:

```
$ elixir math.exs
```

El archivo se compilará en la memoria y se ejecutará, imprimiendo “3” como resultado. No se creará ningún archivo de código de bytes.

Funciones nombradas

Dentro de un módulo, podemos definir funciones con def/2 y funciones privadas con defp/2. Una función definida con def/2 puede invocarse desde otros módulos, mientras que una función privada solo puede invocarse localmente.

```
defmodule Math do
  def sum(a, b) do
    do_sum(a, b)
  end

  defp do_sum(a, b) do
    a + b
  end
end

IO.puts Math.sum(1, 2)    #=> 3
IO.puts Math.do_sum(1, 2) #=> ** (UndefinedFunctionError)
```

Las declaraciones de funciones también admiten guards y múltiples cláusulas. Si una función tiene varias cláusulas, Elixir probará cada cláusula hasta que encuentre una que coincida. Aquí tenemos una implementación de una función que verifica si el número dado es cero o no:

```
defmodule Math do
  def zero?(0) do
    true
  end
```

```
  def zero?(x) when is_integer(x) do
    false
  end
endSiiiii......imágenes cuanto menos, impactantes

IO.puts Math.zero?(0)         #=> true
IO.puts Math.zero?(1)         #=> false
IO.puts Math.zero?([1, 2, 3]) #=> ** (FunctionClauseError)
IO.puts Math.zero?(0.0)       #=> ** (FunctionClauseError)
```

¿El signo de interrogación final en cero? significa que esta función devuelve un valor booleano. Ver convenciones de nomenclatura.

Dar un argumento que no coincide con ninguna de las cláusulas genera un error.

Similar a construcciones como if, las funciones con nombre admiten la sintaxis de bloque do: y do/end, como aprendimos, do/end es una sintaxis conveniente para el formato de lista de palabras clave. Por ejemplo, podemos editar math.exs para que se vea así:

```
defmodule Math do
  def zero?(0), do: true
  def zero?(x) when is_integer(x), do: false
end
```

Y proporcionará el mismo comportamiento. Puede usar do: para líneas simples, pero siempre use do/end para funciones que abarcan varias líneas.

Captura de funciones

Hemos estado usando el nombre/aridad de notación para referirnos a funciones. Sucede que esta notación se puede utilizar para recuperar una función con nombre como un tipo de función. Inicie iex, ejecutando el archivo math.exs definido anteriormente:

```
$ iex math.exs
```

```
iex> Math.zero?(0)
true
iex> fun = &Math.zero?/1
&Math.zero?/1
iex> is_function(fun)
true
iex> fun.(0)
true
```

Recuerde que Elixir hace una distinción entre funciones anónimas y funciones con nombre, donde las primeras deben invocarse con un punto (.) Entre el nombre de la variable y los paréntesis. El operador de captura cierra esta brecha al permitir que las funciones con nombre se asignen a variables y se pasen como argumentos de la misma manera que asignamos, invocamos y pasamos funciones anónimas.

Las funciones locales o importadas, como is_function/1, se pueden capturar sin el módulo:

```
iex> &is_function/1
&:erlang.is_function/1
```

```
iex> (&is_function/1).(fun)
true
```

hay que tener en cuenta que la sintaxis de captura también se puede utilizar como acceso directo para crear funciones:

```
iex> fun = &(&1 + 1)
#Function<6.71889879/1 in :erl_eval.expr/5>
iex> fun.(1)
2

iex> fun2 = &"Good #{&1}"
#Function<6.127694169/1 in :erl_eval.expr/5>
iex)> fun2.("morning")
"Good morning"
```

El &1 representa el primer argumento pasado a la función. &(&1 + 1) arriba es exactamente lo mismo que fn x -> x + 1 final. La sintaxis anterior es útil para definiciones breves de funciones.

Si desea capturar una función desde un módulo, puede hacer &Module.function():

```
iex> fun = &List.flatten(&1, &2)
&List.flatten/2
iex> fun.([1, [[2], 3]], [4, 5])
[1, 2, 3, 4, 5]
```

&List.flatten(&1, &2) es lo mismo que escribir fn(list, tail) -> List.flatten (list, tail) final, que en este caso es equivalente a &List.flatten/2. Puede leer más sobre el operador de captura y en

la documentación Kernel.SpecialForms.

Argumentos por defecto

Las funciones con nombre en Elixir también admiten argumentos predeterminados:

```
defmodule Concat do
  def join(a, b, sep \\ " ") do
    a <> sep <> b
  end
end

IO.puts Concat.join("Hello", "world")      #=> Hello world
IO.puts Concat.join("Hello", "world", "_") #=> Hello_world
```

Se permite que cualquier expresión sirva como valor predeterminado, pero no se evaluará durante la definición de la función. Cada vez que se invoca la función y se debe utilizar cualquiera de sus valores predeterminados, se evaluará la expresión para ese valor predeterminado:

```
defmodule DefaultTest do
  def dowork(x \\ "hello") do
    x
  end
end
```

```
iex> DefaultTest.dowork
```

```
"hello"
iex> DefaultTest.dowork 123
123
iex> DefaultTest.dowork
"hello"
```

Si una función con valores predeterminados tiene varias cláusulas, es necesario crear un encabezado de función (sin un cuerpo real) para declarar valores predeterminados:

```
defmodule Concat do
  def join(a, b \\ nil, sep \\ " ")

  def join(a, b, _sep) when is_nil(b) do
    a
  end

  def join(a, b, sep) do
    a <> sep <> b
  end
end

IO.puts Concat.join("Hello", "world")      #=> Hello world
IO.puts Concat.join("Hello", "world", "_") #=> Hello_world
IO.puts Concat.join("Hello")               #=> Hello
```

El guión bajo inicial en _sep significa que la variable se ignorará en esta función. Ver convenciones de nomenclatura.

Cuando se usan valores predeterminados, se debe tener cuidado para evitar la superposición de definiciones de funciones. Considere el siguiente ejemplo:

```
defmodule Concat do
  def join(a, b) do
    IO.puts "***First join"
    a <> b
  end

  def join(a, b, sep \\ " ") do
    IO.puts "***Second join"
    a <> sep <> b
  end
end
```

Si guardamos el código anterior en un archivo llamado “concat.ex” y lo compilamos, Elixir emitirá la siguiente advertencia:

```
warning: this clause cannot match because a previous clause at line
2 always matches
```

El compilador nos dice que invocar la función de combinación con dos argumentos siempre elegirá la primera definición de combinación, mientras que la segunda solo se invocará cuando se pasen tres argumentos:

```
$ iex concat.ex
```

```
iex> Concat.join "Hello", "world"
***First join
"Helloworld"
```

```
iex> Concat.join "Hello", "world", "_"
***Second join
"Hello_world"
```