Aún no hemos visto ninguna estructura de datos asociativos, es decir, estructuras de datos que puedan asociar un cierto valor (o valores múltiples) a una clave. Diferentes idiomas llaman a estos diferentes nombres como diccionarios, hashes, matrices asociativas, etc.

Tenemos dos estructuras de datos asociativas principales: listas de palabras clave y mapas.

Lista de palabras claves (Keyword lists)

En Elixir, cuando tenemos una lista de tuplas y el primer elemento de la tupla (es decir, la clave) es un átomo, lo llamamos una lista de palabras clave:

```
iex> list = [{:a, 1}, {:b, 2}]
[a: 1, b: 2]
iex> list == [a: 1, b: 2]
true
```

Como puede ver arriba, Elixir admite una sintaxis especial para definir tales listas: [clave: valor]. Debajo se asigna a la misma lista de tuplas que la anterior. Como las listas de palabras clave son listas, podemos usar todas las operaciones disponibles para las listas. Por ejemplo, podemos usar ++ para agregar nuevos valores a una lista de palabras clave:

```
iex> list ++ [c: 3]
[a: 1, b: 2, c: 3]
iex> [a: 0] ++ list
[a: 0, a: 1, b: 2]
```

Ten en cuenta que los valores agregados al frente son los obtenidos en la búsqueda:

```
iex> new_list = [a: 0] ++ list
[a: 0, a: 1, b: 2]
iex> new_list[:a]
0
```

Las listas de palabras clave son importantes porque tienen tres características especiales:

- Las llaves deben ser átomos.
- Las claves están ordenadas, según lo especificado por el desarrollador.
- Las llaves se pueden dar más de una vez.

Por ejemplo, la biblioteca Ecto hace uso de estas características para proporcionar un DSL elegante para escribir consultas en la base de datos:

```
query = from w in Weather,
      where: w.prcp > 0,
      where: w.temp < 20,
     select: w
```

Estas características son las que llevaron a las listas de palabras clave a ser el mecanismo predeterminado para pasar opciones a funciones en Elixir. La macro if/2, mencionamos que la siguiente sintaxis es compatible:

```
iex> if false, do: :this, else: :that
:that
```

¡Los pares do: y else: forman una lista de palabras clave! De hecho, la llamada anterior es equivalente a:

```
iex> if(false, [do: :this, else: :that])
:that
```

Y tambien es lo mismo que:

```
iex> if(false, [{:do, :this}, {:else, :that}])
:that
```

En general, cuando la lista de palabras clave es el último argumento de una función, los corchetes son opcionales.

Aunque podemos hacer coincidir patrones en listas de palabras clave, rara vez se hace en la práctica, ya que la coincidencia de patrones en listas requiere la cantidad de elementos y su orden para que coincidan:

```
iex> [a: a] = [a: 1]
[a: 1]
iex> a
1
iex> [a: a] = [a: 1, b: 2]
** (MatchError) no match of right hand side value: [a: 1, b: 2]
iex> [b: b, a: a] = [a: 1, b: 2]
** (MatchError) no match of right hand side value: [a: 1, b: 2]
```

Para manipular las listas de palabras clave, Elixir proporciona el módulo de palabras clave. Sin embargo, recuerde que las listas de palabras clave son simplemente listas y, como tales, proporcionan las mismas características de rendimiento lineal que las listas. Cuanto más larga sea la lista, más tiempo llevará encontrar una clave, contar la cantidad de elementos, etc. Por esta razón, las listas de palabras clave se utilizan en Elixir principalmente para pasar

valores opcionales. Si necesita almacenar muchos artículos o garantizar asociados de una clave con un valor máximo, debe usar mapas en su lugar.

Mapas

Recomiendo la lectura de este articulo. Cuando termines este tema. Ya que se explica la sintaxis especial que trae Elixir.

Siempre que necesite un almacén de valores clave, los mapas son la estructura de datos “go to” en Elixir. Se crea un mapa utilizando la sintaxis %{}:

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

En comparación con las listas de palabras clave, hay dos diferencias:

- Los mapas permiten cualquier valor como clave.
- Las claves del mapas no siguen ningún pedido.

A diferencia de las listas de palabras clave, los mapas son muy útiles con la coincidencia de patrones. Cuando se usa un mapa en un patrón, siempre coincidirá en un subconjunto del valor dado:

```
iex> %{} = %{:a => 1, 2 => :b}
%{2 => :b, :a => 1}
iex> %{:a => a} = %{:a => 1, 2 => :b}
%{2 => :b, :a => 1}
iex> a
1
iex> %{:c => c} = %{:a => 1, 2 => :b}
** (MatchError) no match of right hand side value: %{2 => :b, :a =>
1}
```

Como se muestra arriba, un mapa coincide siempre y cuando las claves del patrón existan en el mapa dado. Por lo tanto, un mapa vacío coincide con todos los mapas.

Las variables se pueden usar al acceder, hacer coincidir y agregar claves de mapa:

```
iex> n = 1
1
iex> map = %{n => :one}
%{1 => :one}
iex> map[n]
:one
iex> %{^n => :one} = %{1 => :one, 2 => :two, 3 => :three}
%{1 => :one, 2 => :two, 3 => :three}
```

El módulo Map proporciona una API muy similar al módulo Keyword con funciones convenientes para manipular mapas:

```
iex> Map.get(%{:a => 1, 2 => :b}, :a)
1
```

```
iex> Map.put(%{:a => 1, 2 => :b}, :c, 3)
%{2 => :b, :a => 1, :c => 3}
iex> Map.to_list(%{:a => 1, 2 => :b})
[{2, :b}, {:a, 1}]
```

Los mapas tienen la siguiente sintaxis para actualizar el valor de una clave:

```
iex> map = %{:a => 1, 2 => :b}
%{2 => :b, :a => 1}

iex> %{map | 2 => "two"}
%{2 => "two", :a => 1}
iex> %{map | :c => 3}
** (KeyError) key :c not found in: %{2 => :b, :a => 1}
```

La sintaxis anterior requiere que exista la clave dada. No se puede usar para agregar nuevas claves. Por ejemplo, al usarlo con la tecla: c falló porque no hay: c en el mapa.

Cuando todas las claves en un mapa son átomos, puede usar la sintaxis de palabras clave por conveniencia:

```
iex> map = %{:a => 1, 2 => :b}
%{2 => :b, :a => 1}

iex> map.a
1
iex> map.c
** (KeyError) key :c not found in: %{2 => :b, :a => 1}
```

Los desarrolladores de Elixir generalmente prefieren usar la sintaxis map.field y la coincidencia de patrones en lugar de las funciones en el módulo Map cuando trabajan con mapas porque conducen a un estilo de programación asertivo. Esta publicación de blog proporciona información y ejemplos sobre cómo obtener un software más conciso y rápido escribiendo código asertivo en Elixir.

Estructuras de datos anidados

A menudo tendremos mapas dentro de los mapas, o incluso listas de palabras clave dentro de los mapas, y así sucesivamente. Elixir proporciona comodidades para manipular estructuras de datos anidados a través de put_in/2, update_in/2 y otras macros que brindan las mismas comodidades que encontraría en lenguajes imperativos mientras mantiene las propiedades inmutables del lenguaje.

Imagina que tienes la siguiente estructura:

```
iex> users = [
  john: %{name: "John", age: 27, languages: ["Erlang", "Ruby",
"Elixir"]},
  mary: %{name: "Mary", age: 29, languages: ["Elixir", "F#",
"Clojure"]}
]
[john: %{age: 27, languages: ["Erlang", "Ruby", "Elixir"], name:
"John"},
 mary: %{age: 29, languages: ["Elixir", "F#", "Clojure"], name:
"Mary"}]
```

Tenemos una lista de palabras clave de usuarios donde cada valor es un mapa que contiene

el nombre, la edad y una lista de lenguajes de programación que le gusta a cada usuario. Si quisiéramos acceder a la edad de John, podríamos escribir:

```
iex> users[:john].age
27
```

Sucede que también podemos usar esta misma sintaxis para actualizar el valor:

```
users = put_in users[:john].age, 31
[john: %{age: 31, languages: ["Erlang", "Ruby", "Elixir"], name:
"John"},
 mary: %{age: 29, languages: ["Elixir", "F#", "Clojure"], name:
"Mary"}]
```

La macro update_in/2 es similar pero nos permite pasar una función que controla cómo cambia el valor. Por ejemplo, eliminemos “Clojure” de la lista de idiomas de Mary:

```
iex> users = update_in users[:mary].languages, fn languages ->
List.delete(languages, "Clojure") end
[john: %{age: 31, languages: ["Erlang", "Ruby", "Elixir"], name:
"John"},
 mary: %{age: 29, languages: ["Elixir", "F#"], name: "Mary"}]
```

Hay más para aprender sobre put_in/2 y update_in/2, incluido get_and_update_in/2 que nos permite extraer un valor y actualizar la estructura de datos de una vez. También hay put_in/3, update_in/3 y get_and_update_in/3 que permiten el acceso dinámico a la estructura de datos.

Esto concluye nuestra introducción a las estructuras de datos asociativas en Elixir.

Descubrirás que, dadas las listas de palabras clave y los mapas, siempre tendrá la herramienta adecuada para abordar los problemas que requieren estructuras de datos asociativas en Elixir.