Enumerables

Elixir proporciona el concepto de enumerables y el módulo Enum para trabajar con ellos. Ya hemos aprendido dos enumerables: listas y mapas.

```
iex> Enum.map([1, 2, 3], fn x -> x * 2 end)
[2, 4, 6]
iex> Enum.map(%{1 => 2, 3 => 4}, fn {k, v} -> k * v end)
[2, 12]
```

El módulo Enum proporciona una amplia gama de funciones para transformar, ordenar, agrupar, filtrar y recuperar elementos de enumerables. Es uno de los módulos que los desarrolladores usan con frecuencia en su código Elixir.

Elixir también proporciona rangos:

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

Las funciones en el módulo Enum se limitan a, como su nombre lo indica, enumerar valores en estructuras de datos. Para operaciones específicas, como insertar y actualizar elementos particulares, es posible que deba buscar módulos específicos para el tipo de datos. Por ejemplo, si desea insertar un elemento en una posición dada en una lista, debe usar la función List.insert_at/3 del módulo List, ya que tendría poco sentido insertar un valor en, por ejemplo, un rango.

Decimos que las funciones en el módulo Enum son polimórficas porque pueden trabajar con diversos tipos de datos. En particular, las funciones en el módulo Enum pueden funcionar con cualquier tipo de datos que implemente el protocolo Enumerable.

El operador pipe

En el ejemplo siguiente tiene una pipeline de operaciones. Comenzamos con un rango y luego multiplicamos cada elemento en el rango por 3. Esta primera operación ahora creará y devolverá una lista con 100_000 elementos. Luego mantenemos todos los elementos impares de la lista, generando una nueva lista, ahora con 50_000 elementos, y luego sumamos todas las entradas.

```
iex> total_sum = 1..100_000 |> Enum.map(&(&1 * 3)) |> Enum.filter(odd?) |> Enum.sum
7500000000
```

El símbolo |> utilizado en el fragmento de arriba es el operador de tubería: toma la salida de la expresión en su lado izquierdo y la pasa como el primer argumento para la llamada a la función en su lado derecho. Es similar al Unix | operador. Su propósito es resaltar los datos que están siendo transformados por una serie de funciones. Para ver cómo puede hacer que el código sea más limpio, eche un vistazo al ejemplo anterior reescrito sin usar el operador |>:

```
iex> Enum.sum(Enum.filter(Enum.map(1..100_000, &(&1 * 3)), odd?))
7500000000
```

Puedes encontrar más sobre el operador de tubería en su documentación.

Streams

Como alternativa a Enum, Elixir proporciona el módulo Stream que admite operaciones diferidas:

```
iex> 1..100_000 |> Stream.map(&(&1 * 3)) |> Stream.filter(odd?) |> Enum.sum
7500000000
```

Los flujos son enumerables perezosos y componibles.

En el ejemplo anterior, 1..100_000 |> Stream.map (& (& 1 * 3)) devuelve un tipo de datos, un flujo real, que representa el cálculo del mapa en el rango 1..100_000:

```
iex> 1..100_000 |> Stream.map(&(&1 * 3))
#Stream<[enum: 1..100000, funs: [#Function<34.16982430/1 in Stream.map/2>]]>
```

Además, son componibles porque podemos canalizar muchas operaciones de flujo:

```
1..100_000 |> Stream.map(&(&1 * 3)) |> Stream.filter(odd?)
#Stream<[enum: 1..100000, funs: [...]]>
```

En lugar de generar listas intermedias, las secuencias crean una serie de cálculos que se invocan solo cuando pasamos la secuencia subyacente al módulo Enum. Las secuencias son útiles cuando se trabaja con colecciones grandes, posiblemente infinitas.

```
iex> stream = Stream.cycle([1, 2, 3])
#Function<15.16982430/2 in Stream.unfold/2>
iex> Enum.take(stream, 10)
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1]
```

Por otro lado, Stream.unfold/2 puede usarse para generar valores a partir de un valor inicial dado:

```
iex> stream = Stream.unfold("hełło", &String.next_codepoint/1)
#Function<39.75994740/2 in Stream.unfold/2>
iex> Enum.take(stream, 3)
["h", "e", "ł"]
```

Otra función interesante es Stream.resource/3, que se puede utilizar para envolver los

recursos, garantizando que se abran justo antes de la enumeración y se cierren después, incluso en caso de fallas. Por ejemplo, File.stream!/1 se basa en Stream.resource/3 para transmitir archivos:

```
iex> stream = File.stream!("path/to/file")
%File.Stream{
  line_or_bytes: :line,
  modes: [:raw, :read_ahead, :binary],
  path: "path/to/file",
  raw: true
}
iex> Enum.take(stream, 10)
```

El ejemplo anterior buscará las primeras 10 líneas del archivo que ha seleccionado. Esto significa que las transmisiones pueden ser muy útiles para manejar archivos grandes o incluso recursos lentos como los recursos de red.

La cantidad de funcionalidad en los módulos Enum y Stream puede ser desalentadora al principio, pero se familiarizará con ellos caso por caso. En particular, enfóquese primero en el módulo Enum y solo muévase a Stream para los escenarios particulares donde se requiere pereza, ya sea para manejar recursos lentos o grandes colecciones, posiblemente infinitas.