# Тема 4

## 1. Двоични файлове, низове и символни списъци (Binaries, strings and char lists)

В „Основни типове“научихме за низовете и използвахме функцията is_binary/1 за проверки:

```
iex>string="hello"
"hello"
iex>is_binary string
true
```

В това упражнение ще разберем какво представляват двоичните файлове, как се свързват с низовете и какво означава в Elixir стойност в единични кавички, „ 'like this'.

### 1.1 UTF-8 и Unicode

Низът е UTF-8 кодиран двоичен файл. За да разберем какво точно се има предвид под това, трябва да разберем разликата между байтове и кодови точки.

Стандартът Unicode присвоява ASCII код на много от символите, които познаваме. Например буквата a има ASCII код 97, докато буквата ł има ASCII код 322. Когато записваме низа "hełło" на диск, трябва да преобразуваме този ASCII код в байтове. Ако приемем правило, според което един байт представлява един ASCII код, няма да можем да напишем "hełło", защото използва ASCII код 322 за ł, а един байт може да представлява само число от 0 до 255. Но разбира се, като се има предвид, че всъщност можем да четете "hełło" на екрана си, то трябва да бъде представено по някакъв начин. Тук идват кодировките.

Когато представяме ASCII код в байтове, трябва да го кодираме по някакъв начин. Elixir избра кодирането UTF-8 като основно и кодиране по подразбиране. Когато казваме, че низът е двоично кодиран в UTF-8, ние имаме предвид, че низът е байтове, организирани по начин да представят определен ASCII код, както е определено от кодирането на UTF-8.

Тъй като имаме ASCII код като ł, присвоени на числото 322, всъщност се нуждаем от повече от един байт, за да го представим. Ето защо виждаме разлика, когато изчисляваме byte_size/1 на низ спрямо неговия String.length/ 1:

```
iex>string="hełło"
"hełło"
iex>byte_size string
7
iex> String.length string
5
```

UTF-8 изисква един байт за представяне на ASCII код  h, e и o, но два байта за представяне на ł. В Elixir можете да получите стойността на ASCII код:

```
iex>?a
97
iex>?ł
322
```

Можете също да използвате функциите в модула „String“ </docs/stable/elixir/String.html>, за да разделите низ в неговите ASCII символи:

```
iex> String.codepoints("hełło")
["h","e","ł","ł","o"]
```

Ще видите, че Elixir има отлична поддръжка за работа със string. Той също така поддържа много от операциите на Unicode.

Ако низът е двоичен и сме използвали функцията is_binary/1, Elixir трябва да има основен тип упълномощаващ string (низове).

### 1.2 Двоични файлове (и бит string) Binaries (and bitstrings)

В Elixir можете да дефинирате двоичен файл, като използвате <<>>:

```
iex><<0,1,2,3>>
<<0,1,2,3>>
iex>byte_size <<0,1,2,3>>
4
```

Двоичният файл е просто последователност от байтове. Разбира се, тези байтове могат да бъдат организирани по всякакъв начин, дори в последователност, която не ги прави валиден низ:

```
iex> String.valid?(<<239,191,191>>)
false
```

Операцията за конкатенация на низове всъщност е двоичен оператор за конкатенация:

```
iex><<0,1>><><<2,3>>
<<0,1,2,3>>
```

Често срещан трик в Elixir е да се конкатенира нулевият байт <<0>> към низ, за да се види неговото вътрешно двоично представяне:

```
iex>"hełło"<><<0>>
<<104,101,197,130,197,130,111,0>>
```

Всяко число, дадено на двоичен файл е предназначен да представлява байт и следователно трябва да достигне до 255. Двоичните файлове позволяват да се дават модификатори за съхраняване на числа по-големи от 255 или за преобразуване на кодова точка в нейното utf8 представяне:

```
iex><<255>>
<<255>>
iex><<256>>  # truncated
<<0>>
iex><<256::size(16)>>    # use 16 bits (2 bytes) to store the number
<<1,0>>
iex><<256::utf8>>    # the number is a code point
"Ā"
iex><<256::utf8,0>>
<<196,128,0>>
```

Ако един байт има 8 бита, какво ще стане, ако предадем размер от 1 бит?

```
iex><<1::size(1)>>
<<1::size(1)>>
iex><<2::size(1)>>    # truncated
<<0::size(1)>>
iex>is_binary(<<1::size(1)>>)
false
iex>is_bitstring(<<1::size(1)>>)
true
iex>bit_size(<<1::size(1)>>)
1
```

Стойността вече не е двоичен, а bitstring – просто няколко битове! Така че двоичният файл е битов низ, при който броят на битовете се дели на 8!

Можем също да съвпадаме с шаблони на двоични/битови низове (binaries / bitstrings):

```
iex><<0,1, x>>=<<0,1,2>>
<<0,1,2>>
iex>x
2
iex><<0,1, x>>=<<0,1,2,3>>
** (MatchError) no match of right hand sidevalue: <<0,1,2,3>>
```

Обърнете внимание, че всеки запис в двоичния файл се очаква да съответства точно на 8 бита. Въпреки това можем да съпоставим с останалата част от двоичния (binary) модификатор:

```
iex><<0,1, x::binary>>=<<0,1,2,3>>
<<0,1,2,3>>
iex>x
<<2,3>>
```

Моделът по-горе работи само ако двоичният файл е в края на <<>>. Подобни резултати могат да бъдат постигнати с оператора за конкатенация на низове <>:

```
iex>"he"<>rest="hello"
"hello"
iex>rest
"llo"
```

Низът е двоично кодиран в UTF-8, а двоичният код е бит низ, при който броят на битовете се дели на 8.

### 1.3 Char lists

Списъкът със знаци (char lists) не е нищо повече от списък със знаци:

```
iex>'hełło'
[104,101,322,322,111]
iex>is_list'hełło'
true
iex>'hello'
'hello'
```

Можете да видите, че вместо да съдържа байтове, списъкът със знаци съдържа ASCII коде на знаците между единични кавички (обърнете внимание, че iex ще изведе ASCII код само ако някой от знаците е извън обхвата на ASCII). Така че докато двойните кавички представляват низ (т.е. двоичен файл), единичните кавички представляват списък със символи (т.е. списък).

На практика char списъците се използват най-вече при взаимодействие с Erlang, по-специално стари библиотеки, които не приемат двоични файлове като аргументи. Можете да конвертирате char списък в низ и обратно, като използвате функциите to_string/1 и to_char_list/1:

```
iex>to_char_list"hełło"
[104,101,322,322,111]
iex>to_string'hełło'
"hełło"
iex>to_string:hello
"hello"
iex>to_string1
"1"
```

Имайте предвид, че тези функции са полиморфни. Те не само преобразуват char списъци в низове, но и цели числа в низове, атоми в низове и т.н.

## 2. Keywords, maps и dicts

Досега не сме обсъждали никакви асоциативни структури от данни, т.е. структури от данни, които могат да асоциират определена стойност (или множество стойности) към ключ. Различните езици наричат тези различни имена като речници, хешове, асоциативни масиви, карти и т.н.

В Elixir имаме две основни асоциативни структури от данни: списъци с ключови думи и карти.

### 2.1. Keyword lists

В много функционални езици за програмиране е обичайно да се използва списък от кортежи с 2 елемента като представяне на асоциативна структура от данни. В Elixir, когато имаме списък с кортежи и първият елемент от кортежа (т.е. ключът) е атом, ние го наричаме keyword list (списък с ключови думи):

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

Както можете да видите по-горе, Elixir поддържа специален синтаксис за дефиниране на такива списъци, а отдолу те просто се преобразуват в списък с кортежи. Тъй като те са просто списъци, всички операции, достъпни за списъците, включително техните характеристики на ефективност, се прилагат и към списъците с ключови думи.

Например, можем да използваме ++, за да добавим нови стойности към списък с ключови думи:

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

Обърнете внимание, че стойностите, добавени в предната част, са тези, които се извличат при търсене:

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

Списъците с ключови думи са важни, защото имат три специални характеристики:

- Ключовете трябва да са атоми.
- Ключовете се поръчват, както е посочено от разработчика.
- Ключовете могат да се дават повече от веднъж.

Например библиотеката Ecto използва и двете функции, за да предостави елегантен DSL за писане на заявки към база данни:

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

Тези функции накараха списъците с ключови думи да бъдат механизмът по подразбиране за предаване на опции към функциите в Elixir. В предното упражнение, когато обсъждахме макроса if/2, споменахме, че се поддържа следният синтаксис:

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

Двойките do: и else: са списъци с ключови думи! Всъщност обаждането по-горе е еквивалентно на:

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

Като цяло, когато списъкът с ключови думи е последният аргумент на функция, квадратните скоби са незадължителни.

За да манипулира списъците с ключови думи, Elixir предоставя модула „Ключова дума“ </docs/stable/elixir/Keyword.html>‘. Колкото по-дълъг е списъкът, толкова повече време ще отнеме да се намери ключ, да се преброи броят на елементите и т.н. Поради тази причина списъците с ключови думи се използват в Elixir главно като опции. Ако трябва да съхранявате много артикули или да гарантирате съдружници с един ключ с максимум една стойност, вместо това трябва да използвате map.

Рядко се прави на практика съвпадение по шаблони в списъци с ключови думи, тъй като съвпадението на шаблони в списъците изисква броя на елементите и техния ред да съответстват.

```
iex>[a: a]=[a:1]
[a:1]
iex>a
1
iex>[a: a]=[a:1,b:2]
```

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

### 2.2. Maps

Всеки път, когато имате нужда от съхранение на ключова стойност, maps са структурата от данни за „go to“ в Elixir. Създава се map с помощта на %{}синтаксис:

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

В сравнение със списъците с ключови думи вече можем да видим две разлики:

- Map позволяват всяка стойност да е ключ.
- клювете при Map-а не следват никаква подредба.

Ако подадете дублирани ключове при създаване на Map, последният ще печели:

```
iex>%{1=>1,1=>2}
%{1=>2}
```

Когато всички ключове в Map-а са атоми, можете да използвате синтаксиса на ключовата дума (keywords) за удобство:

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

За разлика от списъците с ключови думи, Map са много полезни със съвпадение на шаблони:

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

Както е показано по-горе, Map съвпада, стига дадените ключове да съществуват в даденият Map. Следователно, празна карта съвпада с всички карти.

Модулът „Map“ </docs/stable/elixir/Map.html>‘ предоставя много подобен API на модула за ключови думи с удобни функции за манипулиране на Map:

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

Едно интересно свойство на картите е, че те предоставят определен синтаксис за актуализиране и достъп до атомни ключове:

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

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

iex>%{map|:a=>2}
%{:a=>2,2=>:b}
iex>%{map|:c=>3}
** (ArgumentError) argument error
```

И двата синтаксиса за достъп и актуализиране по-горе изискват дадените ключове да съществуват. Например, достъп и актуализиране на :c ключът е неуспешен, няма :c в картата.

Разработчиците на Elixir обикновено предпочитат да използват синтаксиса на map.field и съвпадението на шаблона вместо функциите в модула Map, когато работят с карти, защото те водят до асертивен стил на програмиране.

Забележка: Map бяха въведени наскоро във VM Erlang с EEP 43. Erlang 17 предоставя частично изпълнение на EEP, където се поддържат само „малки карти“. Това означава, че картите имат добри характеристики само при съхранение на максимум няколко десетки ключа. За да запълни тази празнина, Elixir предоставя и „модула HashDict“ </docs/stable/elixir/HashDict.html>, който използва алгоритъм за хеширане, за да предостави речник, който поддържа стотици хиляди ключове с добра производителност.

### 2.3. Dicts

В Elixir и списъците с ключови думи, и картите се наричат речници. С други думи, речникът е като интерфейс (наричаме ги поведения в Elixir) и както списъците с ключови думи, така и модулите с карти реализират този интерфейс.

Този интерфейс е дефиниран в модула „Dict“ </docs/stable/elixir/Dict.html>“, който също предоставя API, който делегира основните реализации:

```
iex>keyword=[]
[]
iex>map=%{}
%{}
iex> Dict.put(keyword,:a,1)
[a:1]
iex> Dict.put(map,:a,1)
%{a:1}
```

Модулът Dict позволява на всеки разработчик да приложи своя собствена вариация на Dict, със специфични характеристики, и да се присъедини към съществуващия код на Elixir. Модулът Dict също така предоставя функции, които са предназначени да работят в различни речници. Например, Dict.equal?/2 може да сравнява два речника от всякакъв вид.

Въпреки това, може би се чудите кой от модулите за Keyword, Map или Dict трябва да използвате във вашия код? Отговорът е: зависи.

Ако вашият код очаква една конкретна структура от данни като аргумент, използвайте съответния модул, тъй като това води до по-асертивен код. Например, ако очаквате ключова дума като аргумент, използвайте изрично модула Ключова дума вместо Dict. Въпреки това, ако вашият API е предназначен да работи с който и да е речник, използвайте модула Dict (и не забравяйте да напишете тестове, които предават различни реализации на dict като аргументи).

## 3. Modules

В Elixir групираме няколко функции в модули. Вече използвахме много различни модули в предишното упражнение, като модула „String“ </docs/stable/elixir/String.html>‘ :

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

За да създадем наши собствени модули в Elixir, използваме макроса defmodule. Ние използваме макроса def, за да дефинираме функции в този модул:

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

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

В следващите упражнения примерите ще станат по-сложни и може да е трудно да ги въведете всички в обвивката. Крайно време е да се научим как да компилираме код на Elixir, както и как да изпълняваме скриптове на Elixir.

### 3.1. Компилация (Compilation)

През повечето време е удобно да се записват модули във файлове, за да могат да бъдат компилирани и използвани повторно. Да приемем, че имаме файл с име math.ex със следното съдържание:

```
defmoduleMathdo
  def sum(a, b) do
    a+b
  end
end
```

Този файл може да бъде компилиран с помощта на elixirc:

```
$ elixirc math.ex
```

Това ще генерира файл с име Elixir.Math.beam, съдържащ байт кода за дефинирания модул. Ако стартираме iex отново, нашата дефиниция на модула ще бъде налична (при условие, че iex се стартира в същата директория, в която се намира файлът с байт код):

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

Проектите на Elixir обикновено са организирани в три директории:

• ebin - съдържа компилирания байткод

• lib - съдържа код на еликсир (обикновено .ex файлове)

• test - съдържа тестове (обикновено .exs файлове)

Когато работите върху реални проекти, инструментът за изграждане, наречен mix, ще отговаря за компилирането и настройката на правилните пътища за вас. За учебни цели Elixir също поддържа скриптов режим, който е по-гъвкав и не генерира никакви компилирани артефакти.

### 3.2. Scripted mode

В допълнение към файловото разширение Elixir .ex, Elixir поддържа и .exs файлове за скриптове. Elixir третира и двата файла абсолютно по същия начин, единствената разлика е в намерението. .ex файловете са предназначени да бъдат компилирани, докато .exs файловете се използват за скриптове, без да е необходимо компилиране. Например, можем да създадем файл, наречен math.exs:

```
defmoduleMathdo
  def sum(a, b) do
    a+b
  end
end
IO.puts Math.sum(1,2)
```

И го изпълнете като:

```
$ elixir math.exs
```

Файлът ще бъде компилиран в паметта и ще бъде изпълнен, като резултатът ще бъде отпечатан "3". Няма да бъде създаден файл с байткод. В следващите примери ви препоръчваме да напишете своя код в

скриптови файлове и да ги изпълните, както е показано по-горе.

### 3.3. Наименувани функции (Named functions)

Вътре в модул можем да дефинираме функции с def/2 и частни функции с defp/2. Функция, дефинирана с def/2 може да се извиква от други модули, докато частна функция може да се извиква само локално.

```
defmoduleMathdo
  def sum(a, b) do
    do_sum(a, b)
  end

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

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

Декларациите на функции също поддържат защитни (guards) и multiple клаузи. Ако дадена функция има няколко клаузи, Elixir ще опита всяка клауза, докато не намери такава, която съвпада. Ето реализация на функция, която проверява дали даденото число е нула или не:

```
defmoduleMathdo
  def zero?(0) do
    true
  end

  def zero?(x) when is_number(x) do
    false
  end
end

Math.zero?(0)   #=> true
Math.zero?(1)   #=> false
```

```
Math.zero?([1,2,3])
#=> ** (FunctionClauseError)
```

Даването на аргумент, който не съответства на нито една от клаузите, води до грешка.

### 3.4. Function capturing

През целия този урок използвахме нотацията name/arity за препращане към функции. Случва се тази нотация действително да се използва за извличане на наименувана функция като тип функция. Нека стартираме iex и стартираме файла math.exs, дефиниран по-горе:

```
$ 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
```

Локални или импортирани функции, като is_function/1, могат да бъдат заснети без модула:

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

Обърнете внимание, че синтаксисът на улавяне може да се използва и като пряк път за създаване на функции:

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

&1 представлява първия аргумент, предаден във функцията. &(&1+1) по-горе е точно същото като fn x -> x

+ 1 end. Синтаксисът по-горе е полезен за кратки дефиниции на функции.

По същия начин, ако искате да извикате функция от модул, можете да направите &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) е същото като писането на fn(list, tail) -> List.flatten(list, tail) end. Можете да прочетете повече за оператора за улавяне и в документацията на „Kernel.SpecialForms“.

</docs/stable/elixir/Kernel.SpecialForms.html#&/1>‘ .

### 3.5. Default arguments

Наименуваните функции в Elixir също поддържат аргументи по подразбиране:

```
defmoduleConcatdo
  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
```

Всеки израз е разрешен да служи като стойност по подразбиране, но няма да бъде оценен по време на дефинирането на функцията; той просто ще бъде съхранен за по-късна употреба. Всеки път, когато функцията бъде извикана и трябва да се използва някоя от нейните стойности по подразбиране, изразът за тази стойност по подразбиране ще бъде оценен:

```
defmoduleDefaultTestdo
  def dowork(x \\ IO.puts"hello")  do
    x
  end
end
```

```
iex> DefaultTest.dowork
hello
:ok
iex> DefaultTest.dowork123
123
iex> DefaultTest.dowork
hello
:ok
```

Ако функция със стойности по подразбиране има множество клаузи, се препоръчва да се създаде head функция (без действително тяло), само за деклариране на стойности по подразбиране:

```
defmoduleConcatdo
  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
```

Когато използвате стойности по подразбиране, трябва да внимавате да избегнете припокриване на дефиниции на функции. Помислете за следния пример:

```
defmoduleConcatdo
  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
```

Ако запазим кода по-горе във файл с име “concat.ex” и го компилираме, Elixir ще изведе следното предупреждение:

```
concat.ex:7: this clause cannot match because a previous clause at line2always
→matches
```

Компилаторът ни казва, че извикването на функцията за присъединяване с два аргумента винаги ще избере първата дефиниция за присъединяване, докато втората ще бъде извикана само когато се подадат три аргумента:

```
$ iex concat.exs
```

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

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

С това завършва въведението в модулите.

## 4. Recursion

### 4.1. Цикли чрез рекурсия (Loops through recursion)

Поради неизменимостта, циклите в Elixir (и във всеки функционален език за програмиране) се записват различно от императивните езици. Например, на императивен език (като C в следващия пример), човек ще напише:

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

В примера по-горе, ние мутираме както масива, така и променливата i. Мутирането не е възможно в Elixir. Вместо това функционалните езици разчитат на рекурсия: функцията се извиква рекурсивно, докато се достигне условие, което спира рекурсивното действие от продължаване. В този процес не се променят данни. Помислете за примера по-долу, който отпечатва низ произволен брой пъти:

```
defmoduleRecursiondo
  def print_multiple_times(msg, n) when n<=1   do
    IO.puts msg
  end

  def print_multiple_times(msg, n) do
    IO.puts msg print_multiple_times(msg,
    n-1)
  end
end

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

Подобно на case, функцията може да има много клаузи. Конкретна клауза се изпълнява, когато аргументите, предадени на функцията, съвпадат с моделите на аргументите на клаузата и нейната защита се оценява като true.

Когато print_multiple_times/2 първоначално е извикан в примера по-горе, аргументът n е равен на 3.

Първата клауза има „защита“, която казва „използвайте тази дефиниция, ако и само ако n е по-малко или равно на 1“. Тъй като случаят не е такъв, Elixir преминава към дефиницията на следващата клауза.

Втората дефиниция съответства на шаблона и няма защита, така че ще бъде изпълнена. Първо отпечатва нашето съобщение и след това се извиква, като предава n - 1 (2) като втори аргумент.

Нашето съобщение се отпечатва и print_multiple_times/2 се извиква отново този път с втория аргумент, зададен на 1. Тъй като n вече е настроен на 1, защитата в нашата първа дефиниция на print_multiple_times/2 се оценява на true и ние изпълняваме тази конкретна дефиниция. Съобщението се отпечатва и повече нямаме нищо за изпълнение.

Дефинирахме print_multiple_times/2, така че без значение какво число е подадено като втори аргумент, то или задейства първата ни дефиниция (известна като „базов case“), или задейства втората ни дефиниция, която ще гарантира, че ще се приближим точно с една стъпка до нашият основен случай.

### 4.2. Алгоритми за „намаляване“ и „картографиране“ (“Reduce” and “map” algorithms)

Нека сега да видим как можем да използваме силата на рекурсията, за да сумираме списък с числа:

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

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

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

Извикваме sum_list със списъка [1, 2, 3] и първоначалната стойност 0 като аргументи. Ще опитаме всяка клауза, докато не намерим такава, която съвпада според правилата за съвпадение на шаблона. В този случай списъкът [1, 2, 3] съвпада с [head|tail], което свързва head с 1 и tail с [2, 3]; акумулаторът е настроен на 0.

Когато списъкът е празен, той ще съответства на крайната клауза, която връща крайния резултат от 6.

След това добавяме главата на списъка към главата на акумулатора + акумулатора и извикваме sum_list отново, рекурсивно, предавайки

опашката на списъка като негов първи аргумент. Опашката отново ще съвпада с [head|tail], докато списъкът е празен, както се вижда по-долу:

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

Процесът на вземане на списък и „намаляването“ му до една стойност е известен като алгоритъм за „намаляване“ и е централен за функционалното програмиране.

Ами ако вместо това искаме да удвоим всички стойности в нашия списък?

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

  def double_each([]) do
    []
  end
end

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

Тук сме използвали рекурсия, за да преминем през списък, удвоявайки всеки елемент и връщайки нов списък. Процесът на вземане на списък и „картографиране“ върху него е известен като “map”алгоритъм.

Рекурсията и оптимизацията на опашката са важна част от Elixir и обикновено се използват за създаване на цикли. Въпреки това, когато програмирате в Elixir, рядко ще използвате рекурсия, както по-горе, за да манипулирате списъци.

```
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]
```

Или, като използвате синтаксиса на улавяне:

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

Нека да разгледаме по-задълбочено Enumerables и Streams.

## 5. Enumerables and Streams

### 5.1. Enumerables

Elixir предоставя концепцията за enumerables и модула „Enum“ </docs/stable/elixir/Enum.html>‘ за работа с тях. Вече научихме две изброими числа: списъци и карти (lists and maps).

```
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]
```

Модулът Enum предоставя огромен набор от функции за трансформиране, сортиране, групиране, филтриране и извличане на елементи от изброим тип. Това е един от модулите, които разработчиците използват често в своя код на Elixir.

Elixir също предлага диапазони:

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

Тъй като модулът Enum е проектиран да работи с различни типове данни, неговият API е ограничен

до функции, които са полезни за много типове данни. За специфични операции може да се наложи да достигнете до модули, специфични за типовете данни. Например, ако искате да вмъкнете елемент на дадена позиция в списък, трябва да използвате функцията List.insert_at/3 от модула "List' </docs/stable/elixir/List.html>' , като би имало малко смисъл да се вмъква стойност в, например, диапазон.

Казваме, че функциите в модула Enum са полиморфни, защото могат да работят с различни типове данни. По-специално, функциите в модула Enum могат да работят с всеки тип данни, който имплементира протокола „Enumerable“ </docs/stable/elixir/Enumerable.html>‘ . Ще обсъдим протоколите в по-късна глава, засега ще преминем към специфичен вид изброими, наречени потоци.

### 5.2. Нетърпелив срещу мързелив (Eager vs Lazy)

Всички функции в модула Enum са нетърпеливи. Много функции очакват изброяване и връщат списък обратно:

```
iex>odd?=&(rem(   &1,2)!=0)
#Function<6.80484245/1 in :erl_eval.expr/5>
iex> Enum.filter(1..3, odd?)
[1,3]
```

Това означава, че при извършване на множество операции с Enum, всяка операция ще генерира междинен списък, докато стигнем до резултата:

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

Примерът по-горе има конвейер от операции. Започваме с диапазон и след това умножаваме всеки елемент в диапазона по 3. Тази първа операция сега ще създаде и върне списък със 100_000 елемента. След това запазваме всички нечетни елементи от списъка, генерирайки нов списък, сега с 50_000 елемента, и след това сумираме всички записи.

#### 5.2.1. The pipe operator

Символът |>, използван във фрагмента по-горе, е операторът pipe: той просто взема изхода от израза от лявата му страна и го предава като вход към извикването на функция от дясната му страна. Той е подобен на Unix | оператор. Целта му е да подчертае потока от данни, който се трансформира чрез серия от функции. За да видите как може да направи кода по-чист, вижте примера по-горе, пренаписан без да използвате оператора |>:

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

Повече за оператора на pipe може да намерите „като прочетете неговата документация

<http://elixir-lang.org/docs/stable/elixir/Kernel.html#|>/2>“.

### 5.3. Streams

As an alternative to Enum, Elixir provides *the ‘‘Stream‘* module </docs/stable/elixir/Stream.html>‘_which supports lazy operations:

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

Потоците са мързеливи, композируеми изброявания. Вместо да генерират междинни списъци, потоците създават серия от изчисления, които се извикват само когато ги предадем на модула Enum. Потоците са полезни при работа с големи, вероятно безкрайни колекции.

Те са мързеливи, защото, както е показано в примера по-горе, 1..100_000 |> Stream.map(&(&1 * 3)) връща тип данни, действителен поток, който представлява изчислението на картата в диапазона 1..100_000 :

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

Освен това, те са композируеми, тъй като можем да предаваме много операции с поток:

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

Много функции в модула Stream приемат всеки изброим като аргумент и връщат поток като резултат. Той също така предоставя функции за създаване на потоци, вероятно безкрайни. Например, Stream.cycle/1 може да се използва за създаване на поток, който циклизира дадено изброимо число безкрайно. Внимавайте да не извиквате функция като Enum.map/2 в такива потоци, тъй като те ще циклят завинаги:

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

От друга страна, Stream.unfold/2 може да се използва за генериране на стойности от дадена първоначална стойност:

```
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"]
```

Друга интересна функция е Stream.resource/3, която може да се използва за обвиване на ресурси, като гарантира, че те се отварят точно преди изброяването и се затварят след това, дори в случай на неуспехи. Например, можем да го използваме за поточно предаване на файл:

```
iex>stream=   File.stream!("path/to/file")
#Function<18.16982430/2 in Stream.resource/3>
iex> Enum.take(stream,10)
```

Примерът по-горе ще извлече първите 10 реда от избрания от вас файл. Това означава, че потоците могат да бъдат много полезни за работа с големи файлове или дори бавни ресурси като мрежови ресурси.

Количеството функции и функционалност в „Enum </docs/stable/elixir/Enum.html>“ и „Stream

</docs/stable/elixir/Stream.html>‘ модулите могат да бъдат обезсърчителни в началото, но ще се запознаете с тях от всеки отделен случай. По-специално, първо се съсредоточете върху модула Enum и преминете към Stream само за конкретните сценарии, при които се изисква мързел или за справяне с бавни ресурси или големи, вероятно безкрайни колекции.

След това ще разгледаме функция, централна за Elixir, Processes, която ни позволява да пишем едновременни, паралелни и разпределени програми по лесен и разбираем начин.