Elixir

Въведение в Elixir

Тема 1

1 Въведение

В тази тема ще се положи основата на Elixir, ще бъде представен синтаксиса на езика, дефиниране на модули, манипулиране на характеристиките на общите структури от данни и други. Тази тема ще се фокусира върху гарантирането, че Elixir е инсталиран и че можете успешно да стартирате Interactive Shell на Elixir, наречена IEx. Изискванията са:

- Erlang - Version 17.0 onwards
- Elixir - Version 1.0.0 onwards

1.1 Инсталация

Ако все още не сте инсталирали Elixir, стартирайте страница за инсталиране. След като сте готови, можете да стартирате elixir -v, за да получите текущата версия на Elixir.

1.2 Интерактивен режим

След като инсталирате Elixir, ще имате три нови изпълними файла: **iex, elixir и elixirc.** Ако сте компилирали Elixir от източник или използвате пакетирана версия, можете да ги намерите в директорията **bin.**

Засега нека започнем със стартиране на iex (или iex.bat, ако сте на Windows), което означава Interactive Elixir. В интерактивен режим можем да напишем произволен израз на Elixir и да получим неговия резултат. Например нека започнем с някои основни изрази, като:

```
InteractiveElixir  -press  Ctrl+C to exit (type h() ENTERfor  help)

iex>40+2
42
iex>"hello"<>"world"
"hello world"
```

Ще се използва интерактивната обвивка доста често в следващите теми, за да се запознаем малко по-добре с езиковите конструкции и основните типове, започвайки от следващата тема.

1.3 Изпълнение на скриптове

След като се запознаете с основите на езика, може да опитате да пишете прости програми. Това може да се постигне чрез поставяне на код на Elixir във файл и изпълнение с elixir:

```
$ cat simple.exs
IO.puts "Hello world
from Elixir"

$ elixir simple.exs
Hello world
from Elixir
```

По-късно ще научим как да компилираме код на Elixir (в тема 8) и как да използваме инструмента за изграждане на Mix (тема Mix & OTP). Засега да преминем към тема 2.

2 Основни типове

В тази глава ще научим повече за основните типове на Elixir: цели числа, плаващи числа, булеви числа, атоми, низове, списъци и кортежи. Някои основни типове са:

```
iex>1            # integer
iex>0x1F         # integer
iex>1.0          # float
iex>true         # boolean
iex>:atom        # atom / symbol
iex>"elixir"     # string
iex>[1,2,3]      # list
iex>{1,2,3}      # tuple
```

2.1 Основна аритметика

Отворете iex и въведете следните изрази:

```
iex>1+2
3
iex>5  * 5
25
iex>10/2
5.0
```

Забележете, че 10/2 върна float 5.0 вместо цяло число 5. Това е така, защото в Elixir операторът / винаги връща float. Ако искате да направите целочислено деление или да получите остатъка от деление, можете да извикате функциите div и rem:

```
iex>div(10,2)
5
iex>div10,2
5
iex>rem10,3
1
```

Забележете, че скоби не са необходими, за да се извика функция.

Elixir също поддържа нотации за бърз достъп за въвеждане на двоични, осмични и шестнадесетични числа:

```
iex>0b1010
10
iex>0o777
511
iex>0x1F
31
```

Числата с плаваща запетая изискват точка, последвана от поне една цифра, и също така и поддържат **e** (експонента) за числото на степен:

```
iex>1.0
1.0
iex>1.0e-10
1.0e-10
```

Float в Elixir са с 64-битова двойна точност. Можете да извикате функцията round, за да получите най-близкото цяло число до даден float, или функцията trunc, за да получите цялата част от float.

```
iex>round3.58
4
iex>trunc3.58
3
```

2.2 Булеви стойности

Elixir поддържа true и false като булеви:

```
iex>true
true
iex>true==false
false
```

Elixir предоставя куп предикатни функции за проверка за тип стойност. Например, функцията is_boolean/1 може да се използва за проверка дали дадена стойност е булева или не:

Забележка: Функциите в Elixir се идентифицират по име и по брой аргументи (т.е. arity). Следователно is_boolean/1 идентифицира функция с име is_boolean, която приема 1 аргумент. is_boolean/2 идентифицира различна (несъществуваща) функция със същото име, но различно arity.

```
iex>is_boolean(true)
true
```

```
iex>is_boolean(1)
false
```

Можете също да използвате is_integer/1, is_float/1 или is_number/1, за да проверите, съответно, дали

даден аргумент е цяло число, плаващо число или едно от двете.

Забележка: Във всеки един момент можете да въведете h в обвивката, за да отпечатате информация как да използвате обвивката. h помощникът може да се използва и за достъп до документация за всяка функция. Например, въвеждането на h is_integer/1 ще отпечата документацията за функцията is_integer/1. Освен това работи с оператори и други конструкции (опитайте h ==/2).

2.3 Атоми

Атомите са константи, на които името им е тяхната собствена стойност. Някои други езици ги наричат символи:

```
iex>:hello
:hello
iex>:hello==:world
false
```

Булевите стойности true и false са всъщност атоми:

```
iex>true==:true
true
iex>is_atom(false)
true
iex>is_boolean(:false)
true
```

2.4 Низ

Низовете в Elixir се вмъкват между двойни кавички и са кодирани в UTF-8:

```
iex> "hellö"
"hellö"

Note: if you are running on Windows, there is a chance your terminal does not use UTF-8
by default. You can change the encoding of your current session by running ``chcp
65001``.
```

Elixir също поддържа интерполация на низове:

```
iex>"hellö  #{:world}"
"hellö world"
```

Низовете могат да имат прекъсвания на редове в тях или да ги въвеждат с помощта на escape последователности:

```
iex>"hello
...> world"
"hello\nworld"
iex>"hello \nworld"
"hello\nworld"
```

Можете да отпечатате низ с помощта на функцията IO.puts/1 от IO модула:

```
iex> IO.puts"hello \nworld"
hello
world
:ok
```

Забележете, че функцията IO.puts/1 връща **atom :ok** като резултат след отпечатване. Низовете в Elixir са представени вътрешно от двоични файлове, които са поредици от байтове:

```
iex>is_binary("hellö")
true
```

Можем също да получим броя на байтовете в низ:

```
iex>byte_size("hellö")
6
```

Забележете, че броят на байтовете в този низ е 6, въпреки че има 5 знака. Това е така, защото символът „ö“ отнема 2 байта, за да бъде представен в UTF-8. Можем да получим действителната дължина на низа въз основа на броя на знаците, като използваме функцията String.length/1:

```
iex> String.length("hellö")
5
```

Unicode: Модулът String съдържа куп функции, които работят върху низове, както е дефинирано в стандарта:

```
iex> String.upcase("hellö")
"HELLÖ"
```

2.5 Анонимни функции

Функциите са ограничени от ключовите думи `fn` и `end`:

```
iex>add=   fn a, b->a+b     end
#Function<12.71889879/2 in :erl_eval.expr/5>
iex>is_function(add)
true
iex>is_function(add,2)
true
iex>is_function(add,1)
false
iex>add.(1,2)
3
```

Функциите са „функции от първи класа“ в Elixir, което означава, че могат да бъдат предавани като аргументи на други функции, точно както цели числа и низове. В примера сме предали функцията в променливата add на функцията is_function/1, която правилно върна true. Можем също да проверим arity на функцията, като извикаме is_function/2.

Забележете, че точка (.) между променливата и скобата е необходима за извикване на анонимна функция.

Анонимните функции са затворени и като такива могат да имат достъп до променливи, които са в обхват, когато функцията е дефинирана:

```
iex>add_two=   fn a->add.(a,2)     end
#Function<6.71889879/1 in :erl_eval.expr/5>
iex>add_two.(2)
4
```

Имайте предвид, че променлива, присвоена във функция, не засяга заобикалящата я среда:

```
iex>x=42
42
iex>( fn ->x=0     end).()
0
iex>x
42
```

2.6 (Свързани) списъци

Elixir използва квадратни скоби, за да посочи списък със стойности. Стойностите могат да бъдат от всякакъв тип:

```
iex>[1,2,true,3]
[1,2,true,3]
iex>length [1,2,3]
3
```

Два списъка могат да бъдат конкатенирани и изведени с помощта на операторите ++/2 и --/2:

```
iex>[1,2,3]++[4,5,6]
[1,2,3,4,5,6]
iex>[1,true,2,false,3,true]--[true,false]
[1,2,3,true]
```

В тази тема ще говорим много за главата и опашката на списъка. Главата е първият елемент от списъка, а опашката е останалата част от списъка. Те могат да бъдат извлечени с функциите hd/1 и tl/1. Нека присвоим списък на променлива и да го извлечем head-a and tail-a (главата и опашката):

```
iex>list=[1,2,3]
iex>hd(list)
1
iex>tl(list)
[2,3]
```

Получаването на главата или опашката на празен списък извежда грешка:

```
iex>hd []
** (ArgumentError) argument error
```

Понякога ще се наложи да създадете списък и той ще върне стойност в единични кавички. Например:

```
iex>[11,12,13]
'\v\f\r'
iex>[104,101,108,108,111]
'hello'
```

Когато Elixir види списък с ASCII числа за печат, той го отпечата като списък със символи (буквално списък със знаци). Списъците със символи са доста често срещани при взаимодействие със съществуващ Erlang код.

Имайте предвид, че представянето в единични и двойни кавички не са еквивалентни в Elixir, тъй като са представени от различни типове:

```
iex>'hello'=="hello"
false
```

Единичните кавички са списъци със знаци, двойните кавички са низове. Ще говорим повече за тях в темата „Бинарни файлове, низове и списъци със знаци“.

2.7 Кортежи (Tuples)

Elixir използва къдрави скоби, за да дефинира кортежи. Подобно на свързаните списъци, кортежите могат да съдържат всяка стойност:

```
iex>{:ok,"hello"}
{:ok,"hello"}
iex>tuple_size {:ok,"hello"}
2
```

Кортежите съхраняват елементи последователно в паметта. Това означава, че достъпът до елемент на кортежа по индекс или получаването на размера на кортежа е бърза операция (индексите започват от нула):

```
iex>tuple={:ok,"hello"}
{:ok,"hello"}
iex>elem(tuple,1)
"hello"
iex>tuple_size(tuple)
2
```

Възможно е също да зададете елемент с определен индекс в кортеж с put_elem/3:

```
iex>tuple={:ok,"hello"}
{:ok,"hello"}
iex>put_elem(tuple,1,"world")
{:ok,"world"}
iex>tuple
{:ok,"hello"}
```

Забележете, че put_elem/3 върна нов кортеж. Оригиналният кортеж, съхранен в променливата на кортежа, не е променен, тъй като типовете данни на Elixir са неизменни. Тъй като е неизменим, кодът на Elixir е по-лесен за разсъждение, тъй като никога не е нужно да се притеснявате, ако определен код мутира вашата структура от данни на място.

Като е неизменим, Elixir също така помага да се елиминират често срещаните случаи, при които паралелният код има условия за състезание, тъй като два различни субекта, се опитват да променят структура от данни едновременно.

2.8 Списъци или кортежи?

Каква е разликата между списъци и кортежи?

Списъците се съхраняват в паметта като свързани списъци, което означава, че всеки елемент в списъка запазва стойността си и сочи към следващия елемент, докато се достигне краят на списъка. Ние наричаме всяка двойка стойности и указатели **cons cell**:

```
iex>list=[1|[2|[3|[]]]]
[1,2,3]
```

Това означава, че достъпът до дължината на списък е линейна операция: трябва да преминем през целия списък, за да разберем неговия размер. Актуализирането на списък е бързо, стига да добавим елементи:

```
iex>[0]++list
[0,1,2,3]
iex>list++[4]
[1,2,3,4]
```

Първата операция е бърза, защото просто добавяме **нови cons**, които сочат към останалия списък. Вторият е бавен, защото трябва да възстановим целия списък и да добавим нов елемент в края.

Кортежите, от друга страна, се съхраняват последователно в паметта. Това означава, че получаването на размера на кортежа или достъпът до елемент по индекс е бърз. Въпреки това, актуализирането или добавянето на елементи към кортежи е скъпо, защото изисква копиране на целия кортеж в паметта.

Тези характеристики на производителност диктуват използването на едната или другата структура от данни. Един много често срещан случай на използване на кортежи е използването им за връщане на допълнителна информация от функция. Например File.read/1 е функция, която може да се използва за четене на съдържанието на файла и връща кортежи:

```
iex> File.read("path/to/existing/file")
{:ok,"... contents ..."}
iex> File.read("path/to/unknown/file")
{:error,:enoent}
```

Ако пътят, даден на File.read/1, съществува, той връща кортеж с **atom :ok** като първи елемент и съдържанието на файла като втори. В противен случай той връща кортеж с **:error** и описанието на грешката.

През повечето време Elixir ще ви напътства да направите правилното нещо. Например, има функция elem/2 за достъп до кортеж елемент, но няма вграден еквивалент за списъци:

```
iex>tuple={:ok,"hello"}
{:ok,"hello"}
iex>elem(tuple,1)
"hello"
```

Когато преброява "броя на елементите в структурата от данни", Elixir спазва и едно просто правило: функцията трябва да бъде наречена (size) размер, ако операцията е в постоянно време (т.е. стойността е предварително изчислена) или (length) дължина, ако операцията изисква изрично броене.

Например досега сме използвали 4 функции за броене: byte_size/1 (за броя на байтовете в низ), tuple_size/1 (за размера на кортежа), length/1 (за дължината на списъка) и String.length/ 1 (за броя на знаците в низ). Въпреки това ние използваме byte_size, за да получим броя на байтовете в низ, което е евтино, но извличането на броя на символите в Unicode използва String.length, тъй като целият низ трябва да бъде повторен.

Elixir също така предоставя Port, Reference и PID като типове данни (обикновено използвани в процесната комуникация) и те ще се разглеждат, когато говорим за процеси. Засега нека да разгледаме някои от основните оператори, които вървят с нашите основни типове.

3 Основни оператори (Basic operators)

В предишната тема видяхме, че Elixir предоставя +, -, *, / като аритметични оператори, плюс функциите div/2 и rem/2 за целочислено деление и остатък. Elixir също така предоставя ++ и -- за манипулиране на списъци:

```
iex>[1,2,3]++[4,5,6]
[1,2,3,4,5,6]
iex>[1,2,3]--[2]
[1,3]
```

Конкатенацията на низове се извършва с <>:

```
iex>"foo"<>"bar"
"foobar"
```

Elixir също така предоставя три булеви оператора: `or`, `and` и `not`. Тези оператори са строги в смисъл, че очакват boolean (`true` или `false`) като първи аргумент:

```
iex>true  and true
true
iex>false  or is_atom(:example)
true
```

Предоставянето на не-булеви ще предизвика изключение:

```
iex>1  and true
** (ArgumentError) argument error
```

`or` и `and` са оператори на късо съединение. Те изпълняват само дясната страна, ако лявата страна не е достатъчна за определяне на резултата:

```
iex> false and error("This error will never be raised")
false

iex> true or error("This error will never be raised")
true

Note: If you are an Erlang developer, ``and`` and ``or`` in Elixir
actually map to the ``andalso`` and ``orelse`` operators in Erlang.
```

Освен тези булеви оператори, Elixir предоставя и ||, && и ! които приемат аргументи от всякакъв тип. За тези оператори всички стойности с изключение на false и nil ще бъдат оценени на true:

```
# or
iex>1||true
1
iex>false||11
11

# and
iex>nil&&13
nil
iex>true&&17
17

# !
iex>!true
false
iex>!1
false
iex>!nil
true
```

Като правило, използвайте `and`, `or` и `not`, когато очаквате булеви стойности. Ако някой от аргументите не е булев, използвайте &&, || и !.

Elixir също така предоставя ==, !=, ===, !==, <=, >=, < и > като оператори за сравнение:

```
iex>1==1
true
iex>1!=2
true
iex>1<2
true
```

Разликата между == и === е, че последното е по-строго при сравняване на integer и float:

```
iex>1==1.0
true
iex>1===1.0
false
```

В Elixir можем да сравним два различни типа данни:

```
iex>1<:atom
true
```

Причината да сравняваме различни типове данни е прагматизмът. Алгоритмите за сортиране не трябва да се тревожат за различни типове данни, за да сортират. Общият ред на сортиране е дефиниран по-долу:

```
number<atom<reference<functions<port<pid<tuple<maps<list<bitstring
```

Всъщност не е нужно да запомняте тази подредба, но е важно просто да знаете, че съществува. В следващата тема ще се обсъдят някои основни функции, преобразувания на типове данни и малко контролен поток.

4 Съвпадение на шаблон (Pattern matching)

В тази тема ще покажем как операторът = в Elixir всъщност е оператор за съвпадение и как да го използваме за съвпадение на шаблони в структури от данни. И накрая, ще научим за оператора pin ^, използван за достъп до предишно свързани стойности.

4.1 Оператор за съвпадение (The match operator)

Използвахме оператора = няколко пъти, за да присвоим променливи в Elixir:

```
iex>x=1
1
iex>x
1
```

В Elixir операторът = всъщност се нарича оператор за съвпадение. Нека видим защо:

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

Забележете, че 1 = x е валиден израз и съвпада, защото и лявата, и дясната страна са равни на 1. Когато страните не съвпадат, се повдига MatchError.

Променлива може да бъде присвоена само от лявата страна на =:

```
iex>1=unknown
** (RuntimeError) undefinedfunction: unknown/0
```

Тъй като няма предварително дефинирана неизвестна променлива, Elixir си представи, че се опитвате да извикате функция с име unknown/0, но такава функция не съществува.

4.2 Съвпадение на шаблона (Pattern matching)

Операторът за съвпадение не се използва само за съпоставяне с прости стойности, но е полезен и за деструктуриране на по-сложни типове данни. Например, съвпадение по шаблон на кортежи:

```
iex>{a, b, c}={:hello,"world",42}
{:hello,"world",42}
iex>a
:hello
iex>b
"world"
```

Съвпадението на шаблона ще даде грешка в случай, че страните не могат да съвпаднат. Това е например случаят, когато кортежите имат различни размери:

```
iex>{a, b, c}={:hello,"world"}
** (MatchError) no match of right hand sidevalue: {:hello,"world"}
```

А също и при сравняване на различни типове:

```
iex>{a, b, c}=[:hello,"world","!"]
** (MatchError) no match of right hand sidevalue: [:hello,"world","!"]
```

По-интересното е, че можем да съпоставим по конкретни стойности. Примерът по-долу показва, че лявата страна ще съвпада с дясната страна само, когато дясната страна е кортеж, който започва с atom :ok:

```
iex>{:ok, result}={:ok,13}
{:ok,13}
iex>result
13

iex>{:ok, result}={:error,:oops}
** (MatchError) no match of right hand sidevalue: {:error,:oops}
```

Можем да има съвпадение по шаблони в списъци:

```
iex>[a, b, c]=[1,2,3]
[1,2,3]
iex>a
1
```

Списъкът също така поддържа съвпадение на собствените му head и tail (глава и опашка):

```
iex>[head|tail]=[1,2,3]
[1,2,3]
iex>head
1
iex>tail
[2,3]
```

Подобно на функциите hd/1 и tl/1, не можем да съпоставим празен списък с шаблон на главата и опашката:

```
iex>[h|t]=[]
** (MatchError) no match of right hand sidevalue: []
```

[head | tail] форматът не се използва само за съвпадение на шаблони, но и за добавяне на елементи към списък:

```
iex>list=[1,2,3]
[1,2,3]
iex>[0|list]
[0,1,2,3]
```

Съвпадението на шаблоните позволява на разработчиците лесно да деструктурират типове данни, като кортежи и списъци. Както ще видим в следващите теми, това е една от основите на рекурсията в Elixir и се прилага и за други типове, като maps и двоични файлове.

4.3 pin оператор (The pin operator)

Променливите в Elixir могат да бъдат възстановени:

```
iex>x=1
1
iex>x=2
2
```

Операторът pin ^ може да се използва, когато няма интерес от повторно свързване на променлива, а по-скоро от съпоставяне с нейната стойност преди съвпадението:

```
iex>x=1
1
iex>^x=2
** (MatchError) no match of right hand sidevalue:2
iex>{x,^x}={2,1}
{2,1}
iex>x
2
```

Забележете, че ако променлива е спомената повече от веднъж в шаблон, всички препратки трябва да се свързват към един и същ модел:

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

В някои случаи не ви пука за конкретна стойност в модел. Обичайна практика е тези стойности да се свързват с долното черта, _. Например, ако само главата на списъка има значение за нас, можем да присвоим опашката с подчертаване:

```
iex>[h|_]=[1,2,3]
[1,2,3]
iex>h
1
```

Променливата _ е специална с това, че никога не може да бъде прочетена. Опитът за четене от нея дава грешка с несвързана променлива:

```
iex>_
** (CompileError) iex:1: unbound variable_
```

Въпреки че съпоставянето на шаблони ни позволява да изграждаме мощни конструкции, използването му е ограничено. Например, не можете да извършвате извиквания на функции от лявата страна на съвпадението. Следният пример е невалиден:

```
iex>length([1,[2],3])=3
** (CompileError) iex:1: illegal pattern
```