Вградени функции

1) Integer - Функции за работа с цели числа.

Някои функции, които работят с цели числа, се намират в Kernel:

abs/1

div/2

div(dividend, divisor)
@spec div(integer(), neg_integer() | pos_integer()) :: integer()

Извършва целочислено деление.
Поражда ArithmeticErrorизключение, ако един от аргументите не е цяло число или когато divisore 0.
div/2извършва съкратено целочислено деление. Това означава, че резултатът винаги се закръгля към нула.
Ако искате да извършите целочислено деление на етажи (закръгляне към отрицателна безкрайност), използвайте Integer.floor_div/2вместо това.
Допуска се в тестове за защита. Вграден от компилатора.

Пример:

```
div(5, 2)
#=> 2
div(6, -4)
#=> -1
div(-99, 2)
#=> -49
div(100, 0)
** (ArithmeticError) bad argument in arithmetic expression
```

max/2

max(first, second)
@spec max(first, second) :: first | second when first: term(), second: term()

Връща най-големия от двата дадени термина според тяхното структурно сравнение.
Ако условията за сравнение са равни, връща се първият.
Това извършва структурно сравнение, при което всички термини на Elixir могат да бъдат сравнени един с друг. Вижте раздела „Сравнение на структурата“ за повече информация. Вграден от компилатора.

Пример:

```
max(1, 2)
2
max(:a, :b)
:b
```

min/2

min(first, second)
@spec min(first, second) :: first | second when first: term(), second: term()
Връща най-малкия от двата дадени термина според тяхното структурно сравнение.
Ако условията за сравнение са равни, връща се първият.
Това извършва структурно сравнение, при което всички термини на Elixir могат да бъдат сравнени един с друг. Вижте раздела „Сравнение на структурата“ за повече информация. Вграден от компилатора.

Пример:

```
min(1, 2)
1
min("foo", "bar")
"bar"
```

rem/2

rem(dividend, divisor)
@spec rem(integer(), neg_integer() | pos_integer()) :: integer()

Изчислява остатъка от целочислено деление.
rem/2използва съкратено деление, което означава, че резултатът винаги ще има знака на dividend.
Поражда ArithmeticErrorизключение, ако един от аргументите не е цяло число или когато divisore 0.
Допуска се в тестове за защита. Вграден от компилатора.

Пример:

```
rem(5, 2)
1
rem(6, -4)
2
```

НОВА ТЕМА

Enum

Осигурява набор от алгоритми за работа с изброими. В Elixir, enumerable е всеки тип данни, който имплементира Enumerable протокола. Lists ([1, 2, 3]), Maps (%{foo: 1, bar: 2}) и Ranges (1..3) са общи типове данни, използвани като изброими:

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

sort(enumerable)
@spec sort(t()) :: list()

Сортира изброимото според подреждането на термините на Erlang.
Тази функция използва алгоритъма за сортиране чрез сливане. Не използвайте тази функция за сортиране на структури, вижте sort/2 за повече информация.

https://hexdocs.pm/elixir/1.11.2/Enum.html#sort/2

Пример:

```
Enum.sort([3, 2, 1])
[1, 2, 3]
```

sort(enumerable, fun)

```
@spec sort(
  t(),
  (element(), element() -> boolean())
  | :asc
  | :desc
  | module()
  | {:asc | :desc, module()}
) :: list()
```

Сортира enumerable по дадената функция.

Тази функция използва алгоритъма за сортиране чрез сливане. Дадената функция трябва да сравнява два аргумента и да връща, trueако първият аргумент предхожда или е на същото място като втория.

Пример:

```
Enum.sort([1, 2, 3], &(&1 >= &2))
[3, 2, 1]
```

Алгоритъмът за сортиране ще бъде стабилен, докато дадената функция се връща trueза стойности, считани за равни:

```
Enum.sort(["some", "kind", "of", "monster"], &(byte_size(&1) <= byte_size(&2)))
["of", "some", "kind", "monster"]
```

Ако функцията не се върне trueза равни стойности, сортирането не е стабилно и редът на равните термини може да бъде разбъркан. Например:

```
Enum.sort(["some", "kind", "of", "monster"], &(byte_size(&1) < byte_size(&2)))
["of", "kind", "some", "monster"]
```

Възходящо и низходящо
sort/2 позволява на разработчика да премине :asc или :desc като функцията за сортиране, което е удобство за <=/2 и >=/2 съответно.

```
Enum.sort([2, 3, 1], :asc)
[1, 2, 3]
Enum.sort([2, 3, 1], :desc)
[3, 2, 1]
```

Пазачи (Guards)	
is_even(integer)	Определя дали integer е четно
is_odd(integer)	Определя дали integer е нечетно
Функции	
digits(integer, base \\ 10)	Връща подредените цифри за даденият integer
extended_gcd(n, n)	Връща разширения най-голям общ делител на двете дадени цели числа
floor_div(dividend, divisor)	Извършва floor (етажно) целочислено деление
gcd(integer1, integer2)	Връща най-големия общ делител на двете дадени цели числа
mod(dividend, divisor)	Изчислява остатъка по модул от целочислено деление
parse(binary, base \\ 10)	Анализира текстово представяне на цяло число
pow(base, exponent)	Изчислява base на степен exponent
to_charlist(integer, base \\ 10)	Връща списък със знаци, който съответства на текстовото представяне на integer в дадения base
to_string(integer, base \\ 10)	Връща двоичен файл, който съответства на текстовото представяне на integer в дадения base
undigits(digits, base \\ 10)	Връща цялото число, представено от подредения digits
Пазачи (Guards)	
is_even(integer)	Определя дали integer е четно. Връща, true, ако даденото integer е четно число, в противен случай връща false. Позволено в предпазни клаузи.
is_odd(integer)	Определя дали integere нечетно. Връща, true, ако даденото integer е нечетно число, в противен случай връща false. Позволено в предпазни клаузи.
Функции	
digits(integer, base \\ 10) @spec digits(integer(), pos_integer()) :: [integer(), ...]	Връща подредените цифри за даденото integer. Може да бъде предоставена незадължителна base стойност, представляваща основата за върнатите цифри. Това трябва да е цяло число >= 2.
extended_gcd(n, n)(от 1.12.0) @spec extended_gcd(integer(), integer()) :: {non_neg_integer(), integer(), integer()}	Връща разширения най-голям общ делител на двете дадени цели числа. Той използва разширения евклидов алгоритъм, за да върне кортеж от три елемента с и gcd коефициентите m и n на идентичността на Bézout, така че: gcd(a, b) = m*a + n*b По конвенция extended_gcd(0, 0)връща {0, 0, 0}.
floor_div(dividend, divisor)(от 1.4.0) @spec floor_div(integer(), neg_integer() | pos_integer()) :: integer()	Извършва етажно целочислено деление. Поражда Arithmetic Error изключение, ако един от аргументите не е цяло число или когато divisore 0. Integer.floor_div/2 извършва floor (етажно) целочислено деление. Това означава, че резултатът винаги е закръглен към отрицателна безкрайност. Ако искате да извършите съкратено целочислено деление (закръгляване към нула), използвайте Kernel.div/2 вместо това.
gcd(integer1, integer2)(от 1.5.0) @spec gcd(integer(), integer()) :: non_neg_integer()	Връща най-големия общ делител на двете дадени цели числа. Най-големият общ делител (НОД) на integer1 и integer2 е най-голямото положително цяло число, което дели integer1 и двете integer2, без да оставя остатък. По конвенция gcd(0, 0) връща 0.
mod(dividend, divisor)(от 1.4.0) @spec mod(integer(), neg_integer() | pos_integer()) :: integer()	Изчислява остатъка по модул от целочислено деление. Integer.mod/2 използва floor (етажно) деление, което означава, че резултатът винаги ще има знака на divisor. Поражда Arithmetic Error изключение, ако един от аргументите не е цяло число или когато divisor е 0.
parse(binary, base \\ 10) @spec parse(binary(), 2..36) :: {integer(), remainder_of_binary :: binary()} | :error	Анализира текстово представяне на цяло число. Може да се предостави опция base за съответното цяло число. Ако base не е дадено, ще се използва 10. При успех връща кортеж под формата на {integer, remainder_of_binary}. В противен случай :error. Извежда грешка, ако base е по-малко от 2 или повече от 36. Ако искате да конвертирате низово форматирано цяло число директно в цяло число String.to_integer/1 или String.to_integer/2 може да се използва вместо това.
pow(base, exponent)(от 1.12.0) @spec pow(integer(), non_neg_integer()) :: integer()	Изчислява base на степен exponent. base и exponent трябва да са цели числа. Показателят трябва да е нула или положителен. Вижте Float.pow/2 за степенуване на отрицателни показатели, както и числа с плаваща запетая.
to_charlist(integer, base \\ 10) @spec to_charlist(integer(), 2..36) :: charlist()	Връща списък със знаци, който съответства на текстовото представяне на integer в дадения base. base може да бъде цяло число между 2 и 36. Ако base е дадено не, по подразбиране е 10. Вграден от компилатора.
to_string(integer, base \\ 10) @spec to_string(integer(), 2..36) :: String.t()	Връща двоичен файл, който съответства на текстовото представяне на integer в дадения base. base може да бъде цяло число между 2 и 36. Ако base е дадено не, по подразбиране е 10. Вграден от компилатора.
undigits(digits, base \\ 10) @spec undigits([integer()], pos_integer()) :: integer()	Връща цялото число, представено от подредения digits. Може да бъде предоставена незадължителна base стойност, представляваща основата за digits. Основата трябва да е цяло число, по-голямо или равно на 2.

Примери:

Integer.is_even(10) **true** Integer.is_even(5) **false** Integer.is_even(-10) **true** Integer.is_even(0) **true**	Integer.is_odd(5) **true** Integer.is_odd(6) **false** Integer.is_odd(-5) **true** Integer.is_odd(0) **false**	Integer.digits(123) **[1, 2, 3]** Integer.digits(170, 2) **[1, 0, 1, 0, 1, 0, 1, 0]** Integer.digits(-170, 2) **[-1, 0, -1, 0, -1, 0, -1, 0]**
Integer.extended_gcd(240, 46) **{2, -9, 47}** Integer.extended_gcd(46, 240) **{2, 47, -9}** Integer.extended_gcd(-46, 240) **{2, -47, -9}** Integer.extended_gcd(-46, -240) **{2, -47, 9}**	Integer.floor_div(5, 2) **2** Integer.floor_div(6, -4) **-2** Integer.floor_div(-99, 2) **-50**	Integer.gcd(2, 3) **1** Integer.gcd(8, 12) **4** Integer.gcd(8, -12) **4** Integer.gcd(10, 0) **10**

Integer.extended_gcd(14, 21) **{7, -1, 1}** Integer.extended_gcd(10, 0) **{10, 1, 0}** Integer.extended_gcd(0, 10) **{10, 0, 1}** Integer.extended_gcd(0, 0) **{0, 0, 0}**		Integer.gcd(7, 7) **7** Integer.gcd(0, 0) **0**
Integer.mod(5, 2) **1** Integer.mod(6, -4) **-2**	Integer.parse("34") **{34, ""}** Integer.parse("34.5") **{34, ".5"}** Integer.parse("three") **:error** Integer.parse("34", 10) **{34, ""}** Integer.parse("f4", 16) **{244, ""}** Integer.parse("Awww++", 36) **{509216, "++"}** Integer.parse("fab", 10) **:error** **Integer.parse("a2", 38)** ** (ArgumentError) invalid base 38	Integer.pow(2, 0) **1** Integer.pow(2, 1) **2** Integer.pow(2, 10) **1024** Integer.pow(2, 11) **2048** Integer.pow(2, 64) **0x10000000000000000** Integer.pow(3, 4) **81** Integer.pow(4, 3) **64** Integer.pow(-2, 3) **-8** Integer.pow(-2, 4) **16** **Integer.pow(2, -2)** ** (ArithmeticError) bad argument in arithmetic expression
Integer.to_charlist(123) **'123'** Integer.to_charlist(+456) **'456'** Integer.to_charlist(-789) **'-789'** Integer.to_charlist(0123) **'123'** Integer.to_charlist(100, 16) **'64'** Integer.to_charlist(-100, 16) **'-64'** Integer.to_charlist(882_681_651, 36) **'ELIXIR'**	Integer.undigits([1, 2, 3]) **123** Integer.undigits([1, 4], 16) **20** Integer.undigits([]) **0**	

2) Float - Функции за работа с числа с плаваща запетая.

Функции на ядрото

В модула има Kernel и функции, свързани с числа с плаваща запетая. Ето списък с тях:

- Kernel.round/1: закръглява число до най-близкото цяло число.
- Kernel.trunc/1: връща цялата част от числото.

Има някои много добре известни проблеми с числата с плаваща запетая и аритметиката, поради факта, че повечето десетични дроби не могат да бъдат представени с двоичен код с плаваща запетая и повечето операции не са точни, а работят с приближения. Тези проблеми не са специфични за Elixir, те са свойство на самото представяне с плаваща запетая.

Например, числата 0,1 и 0,01 са две от тях, което означава, че резултатът от повдигане на квадрат 0,1 не дава 0,01 нито най-близкото представяне. Ето какво се случва в този случай:

- Най-близкото представяне е число до 0,1 е 0,1000000014
- Най-близкото представяне е число до 0,01 е 0,0099999997
- Извършването на 0,1 * 0,1 трябва да върне 0,01, но тъй като 0,1 всъщност е 0,1000000014, резултатът е 0,010000000000000002 и тъй като това не е най-близкото представяне на числото до 0,01, ще се получи грешен резултат за тази операция

Има и други известни проблеми като настилката или закръглянето на числата. Вижте round/2 и floor/2 за повече подробности за тях.

Функции	
ceil(number, precision \\ 0)	Закръглява float до най-малкото цяло число, по-голямо или равно на num.
floor(number, precision \\ 0)	Закръглява число с плаваща точка до най-голямото число, по-малко или равно на num.
parse(binary)	Анализира двоичен файл в плаващ.
pow(base, exponent)	Изчислява basena степен exponent.
ratio(float)	Връща двойка цели числа, чието съотношение е точно равно на оригиналния float и с положителен знаменател.
round(float, precision \\ 0)	Закръглява стойност с плаваща запетая до произволен брой дробни цифри (между 0 и 15).
to_charlist(float)	Връща списък със знаци, който съответства на текстовото представяне на дадения float.
to_string(float)	Връща двоичен файл, който съответства на текстовото представяне на дадения float.

ceil(number, precision \\ 0)

@spec ceil(float(), precision_range()) :: float()

Закръглява float до най-малкото цяло число, по-голямо или равно на num.
ceil/2 също приема прецизност за закръгляване на стойност с плаваща запетая до произволен брой дробни цифри (между 0 и 15).
Операцията се извършва върху двоичната стойност с плаваща запетая, без преобразуване в десетична.
Поведението на **ceil/2** за floats може да бъде изненадващо.

Пример:

```
iex> Float.ceil(-12.52, 2)
-12.51
```

Човек може да е очаквал достигне до -12,52. Това не е грешка. Повечето десетични дроби не могат да бъдат представени като числа с двоична плаваща запетая и следователно числото по-горе е вътрешно представено като -12,51999999, което обяснява поведението по-горе.
Тази функция винаги връща плаващи числа. Kernel.trunc/1 може да се използва вместо това за съкращаване на резултата до цяло число след това.

Пример:

```
iex> Float.ceil(34.25)
35.0
iex> Float.ceil(-56.5)
-56.0
iex> Float.ceil(34.251, 2)
34.26
```

floor(number, precision \\ 0)
@spec floor(float(), precision_range()) :: float()

Закръгля число с плаваща точка до най-голямото число, по-малко или равно на num.
floor/2 също приема прецизност за закръгляване на стойност с плаваща запетая до произволен брой дробни цифри (между 0 и 15). Операцията се извършва върху двоичната стойност с плаваща запетая, без преобразуване в десетична.
Тази функция винаги връща float. Kernel.trunc/1 може да се използва вместо това за съкращаване на резултата до цяло число след това.
Поведението на floor/2 за floats може да бъде изненадващо.

Пример:

```
iex> Float.floor(12.52, 2)
12.51
```

Човек може да е очаквал да падне до 12,52. Това не е грешка. Повечето десетични дроби не могат да бъдат представени като двоична плаваща запетая и следователно числото по-горе е вътрешно представено като 12,51999999, което обяснява поведението по-горе.

Пример:

```
iex> Float.floor(34.25)
34.0
iex> Float.floor(-56.5)
-57.0
iex> Float.floor(34.259, 2)
34.25
```

parse(binary)
@spec parse(binary()) :: {float(), binary()} | :error

Анализира двоичен файл в плаващ.
При успех връща кортеж под формата на {float, remainder_of_binary}; когато двоичният файл не може да бъде преобразуван във валиден float и се връща атомът :error.
Ако размерът на float надвишава максималния размер на 1.7976931348623157e+308, ArgumentError се повдига изключение.
Ако искате да преобразувате низово форматиран float директно в float, String.to_float/1 може да се използва вместо това.
Пример:

```
iex> Float.parse("34")
{34.0, ""}
iex> Float.parse("34.25")
{34.25, ""}
iex> Float.parse("56.5xyz")
{56.5, "xyz"}
iex> Float.parse("pi")
:error
```

pow(base, exponent)
@spec pow(float(), number()) :: float()

Изчислява base на степен exponent.
base трябва да е число с плаваща запетая и exponent може да бъде произволно число. Въпреки това, ако са дадени отрицателна основа и дробен показател, това повишава ArithmeticError.
Винаги връща float. Integer.pow/2 за степенуване, връща цели числа.

Пример:

```
iex> Float.pow(2.0, 0)
1.0
iex> Float.pow(2.0, 1)
2.0
```

```
iex> Float.pow(2.0, 10)
1024.0
iex> Float.pow(2.0, -1)
0.5
iex> Float.pow(2.0, -3)
0.125
iex> Float.pow(3.0, 1.5)
5.196152422706632
iex> Float.pow(-2.0, 3)
-8.0
iex> Float.pow(-2.0, 4)
16.0
iex> Float.pow(-1.0, 0.5)
** (ArithmeticError) bad argument in arithmetic expression
```

ratio(float)
@spec ratio(float()) :: {integer(), pos_integer()}
Връща двойка цели числа, чието съотношение е точно равно на оригиналния float и с положителен знаменател.

Пример:

```
iex> Float.ratio(0.0)
{0, 1}
iex> Float.ratio(3.14)
{7070651414971679, 2251799813685248}
iex> Float.ratio(-3.14)
{-7070651414971679, 2251799813685248}
iex> Float.ratio(1.5)
{3, 2}
iex> Float.ratio(-1.5)
{-3, 2}
iex> Float.ratio(16.0)
{16, 1}
iex> Float.ratio(-16.0)
{-16, 1}
```

round(float, precision \\ 0)
@spec round(float(), precision_range()) :: float()
Закръгля стойност с плаваща запетая до произволен брой дробни цифри (между 0 и 15).
Посоката на закръгляне винаги е свързана с половината нагоре. Операцията се извършва върху двоичната стойност с плаваща запетая, без преобразуване в десетична.
Тази функция приема само плаващи числа и винаги връща плаващи числа. Използвайте Kernel.round/1, ако искате функция, която приема както плаващи, така и цели числа и винаги връща цяло число.
Поведението на round/2 за floats може да бъде изненадващо.

Пример:

```
iex> Float.round(5.5675, 3)
5.567
```

Човек може да е очаквал да закръгли до половината нагоре 5,568. Това не е грешка. Повечето десетични дроби не могат да бъдат представени като двоична плаваща запетая и следователно числото по-горе е вътрешно представено като 5,567499999, което обяснява поведението по-горе. Ако искате точно закръгляване за десетични числа, трябва да **използвате десетична библиотека.** Поведението по-горе също е в съответствие с референтни реализации, като например "Правилно закръглени двоично-десетични и десетично-двоични преобразувания" от Дейвид М. Гей.

Пример:

```
iex> Float.round(12.5)
13.0
iex> Float.round(5.5674, 3)
5.567
iex> Float.round(5.5675, 3)
5.567
iex> Float.round(-5.5674, 3)
-5.567
iex> Float.round(-5.5675)
-6.0
iex> Float.round(12.341444444444441, 15)
12.341444444444441
```

to_charlist(float)
@spec to_charlist(float()) :: charlist()

Връща списък със знаци, който съответства на текстовото представяне на дадения float.
Той използва най-краткото представяне според алгоритъма, описан в „Бързо и точно отпечатване на числа с плаваща запетая“ в докладите на конференцията SIGPLAN '96 за проектиране и внедряване на език за програмиране.

Пример:

```
iex> Float.to_charlist(7.0)
'7.0'
```

to_string(float)
@spec to_string(float()) :: String.t()

Връща двоичен файл, който съответства на текстовото представяне на дадения float.
Той използва най-краткото представяне според алгоритъма, описан в „Бързо и точно отпечатване на числа с плаваща запетая“ в докладите на конференцията SIGPLAN '96 за проектиране и внедряване на език за програмиране.
Пример:

```
iex> Float.to_string(7.0)
"7.0"
```

4) String - Низовете в Elixir са UTF-8 кодирани двоични файлове.

Низовете в Elixir са поредица от Unicode символи, обикновено написани между низове с двойни кавички, като "hello"и "héllò".
В случай, че низ трябва да има двойни кавички в себе си, двойните кавички трябва да бъдат екранирани с обратна наклонена черта.
Пример: "this is a string with \"double quotes\"".

Можете да свържете два низа с <>/2оператора:
"hello" <> " " <> "world"
"hello world"

Интерполация
Низовете в Elixir също поддържат **интерполация**. Това позволява да поставите някаква стойност в средата на низ, като използвате синтаксиса **#{}:**
name = "joe"
"hello #{name}"
"hello joe"
Всеки израз на Elixir е валиден вътре в интерполацията. Ако е даден низ, низът се интерполира такъв, какъвто е. Ако е дадена друга стойност, Elixir ще се опита да я преобразува в низ, използвайки протокола String.Chars. Това позволява, например, да се изведе цяло число от интерполацията:

```
"2 + 2 = #{2 + 2}"
"2 + 2 = 4"
```

В случай, че стойността, която искате да интерполирате, не може да бъде преобразувана в низ, защото няма човешко текстово представяне, ще се появи грешка в протокола.

Escape символи
Освен че позволяват двойни кавички да бъдат екранирани с обратна наклонена черта, низовете поддържат и следните екраниращи знаци:

\a - Bell
\b - Backspace
\t - Horizontal tab
\n - Line feed (New lines)
\v - Vertical tab
\f - Form feed
\r - Carriage return
\e - Command Escape
\# - Returns the # character itself, skipping interpolation
\xNN - A byte represented by the hexadecimal NN
\uNNNN - A Unicode code point represented by NNNN

Обърнете внимание, че обикновено не се препоръчва да се използва \xNNв низове на Elixir, тъй като въвеждането на невалидна последователност от байтове би направило низа невалиден. Ако трябва да въведете знак чрез неговото шестнадесетично представяне, най-добре е да работите с Unicode кодови точки, като например \uNNNN. Всъщност разбирането на кодовите точки на Unicode може да бъде от съществено значение, когато се извършват манипулации на низ на ниско ниво, така че нека ги разгледаме подробно по-нататък.

Кодови точки и клъстер от графеми
Функциите в този модул работят в съответствие със стандарта Unicode, версия 13.0.0.

Съгласно стандарта кодовата точка е единичен Unicode знак, който може да бъде представен от един или повече байта.
Например, въпреки че кодовата точка "é" е един знак, нейното основно представяне използва два байта:

```
String.length("é")
1
byte_size("é")
2
```

Освен това, този модул също така представя концепцията за клъстер от графеми (отсега нататък наричани графеми). Графемите могат да се състоят от множество кодови точки, които могат да се възприемат като един знак от читателите. Например „é“ може да бъде представено или като единична кодова точка „e с акут“ или като буквата „e“, последвана от „комбиниращ акут“ (две кодови точки):

```
string = "\u0065\u0301"
byte_size(string)
3
String.length(string)
1
String.codepoints(string)
["e", "́"]
String.graphemes(string)
["é"]
```

Въпреки че примерът по-горе е съставен от два знака, той се възприема от потребителите като един.
Графемите също могат да бъдат два знака, които се тълкуват като един от някои езици. Например, някои езици може да разглеждат "ch" като един знак. *Въпреки това, тъй като тази информация зависи от локалната инсталация на компютъра, тя не се взема предвид от този модул.*
Като цяло функциите в този модул разчитат на стандарта Unicode, но не съдържат никакво специфично за *локалната инсталация на компютъра* поведение. Повече информация за графемите можете да намерите в стандартното приложение на Unicode #29 .
За конвертиране на двоичен файл в различно кодиране и за механизми за нормализиране на Unicode вижте :unicode модула на Erlang.

Низови и двоични операции
За да действат в съответствие със стандарта Unicode, много функции в този модул се изпълняват в линейно време, тъй като трябва да преминат през целия низ, като вземат предвид правилните кодови точки на Unicode.
Например, String.length/1 ще отнеме повече време, с нарастване на входните данни. От друга страна, Kernel.byte_size/1 винаги работи в постоянно време (т.е. независимо от размера на входа).

Това означава, че често има разходи за производителност при използването на функциите в този модул, в сравнение с операциите на по-ниско ниво, които работят директно с двоични файлове:
Kernel.binary_part/3- извлича част от двоичния файл
Kernel.bit_size/1и Kernel.byte_size/1- функции, свързани с размера
Kernel.is_bitstring/1и Kernel.is_binary/1- функция за проверка на типа

Плюс редица функции за работа с двоични файлове (байтове) в :binary модула
Има много ситуации, при които използването на String модула може да се избегне в полза на двоични функции или съпоставяне на шаблони. Например, представете си, че имате низ prefix и искате да премахнете този префикс от друг низ с име full.

Човек може да се изкуши да напише:

```
take_prefix = fn full, prefix ->
  base = String.length(prefix)
  String.slice(full, base, String.length(full) - base)
end
take_prefix.("Mr. John", "Mr. ")
"John"
```

Въпреки че функцията по-горе работи, тя се представя зле. За да изчислим дължината на низа, трябва да го обходим изцяло, така че преминаваме през двата prefix низа full и след това нарязваме единия full, като го обхождаме отново.

Първият опит за подобряване може да бъде с диапазони:

```
take_prefix = fn full, prefix ->
  base = String.length(prefix)
  String.slice(full, base..-1)
end
take_prefix.("Mr. John", "Mr. ")
"John"
```

Въпреки че това е много по-добро (не преминаваме full два пъти), все пак може да се подобри. В този случай, тъй като искаме да извлечем подниз от низ, можем да използваме Kernel.byte_size/1и Kernel.binary_part/3 тъй като няма шанс да изрежем в средата на кодова точка, съставена от повече от един байт:

```
take_prefix = fn full, prefix ->
  base = byte_size(prefix)
  binary_part(full, base, byte_size(full) - base)
end
take_prefix.("Mr. John", "Mr. ")
"John"
```

Или просто използвайте съвпадение на шаблони:

```
take_prefix = fn full, prefix ->
  base = byte_size(prefix)
  <<_::binary-size(base), rest::binary>> = full
  rest
end
take_prefix.("Mr. John", "Mr. ")
"John"
```

От друга страна, ако искате динамично да разделите низ въз основа на цяло число, тогава използването String.slice/3 е най-добрият вариант, тъй като гарантира, че няма да разделим неправилно валидна кодова точка на множество байтове.

Целочислени кодови точки

Въпреки че кодовите точки са представени като цели числа, този модул представя кодовите точки в техния кодиран формат като низове.

Пример:

```
String.codepoints("olá")
["o", "l", "á"]
```

Има няколко начина за извличане на кодовата точка на символа. Може да се използва ? конструкцията:

```
?o
111

?á
225
```

Или също чрез съвпадение на шаблони:

```
<<aacute::utf8>> = "á"
aacute
225
```

Както видяхме по-горе, кодовите точки могат да бъдат вмъкнати в низ чрез техния шестнадесетичен код:

```
"ol\u00E1"
"olá"
```

И накрая, за да преобразувате низ в списък с целочислени кодови точки, известни като "charlists" в Elixir, можете да извикате String.to_charlist:

```
String.to_charlist("olá")
[111, 108, 225]
```

Самосинхронизация

UTF-8 кодирането се самосинхронизира. Това означава, че ако бъдат открити неправилно формирани данни (т.е. данни, които не са възможни според дефиницията на кодирането), само една кодова точка трябва да бъде отхвърлена.

Този модул разчита на това поведение, за да игнорира такива невалидни знаци. Например, length/1 ще върне правилен резултат, дори ако в него е въведена невалидна кодова точка.

С други думи, този модул очаква невалидни данни да бъдат открити другаде, обикновено при извличане на данни от външен източник. Например, драйвер, който чете низове от база данни, ще отговаря за проверката на валидността на кодирането. String.chunk/2 може да се използва за разделяне на низ на валидни и невалидни части.

Компилиране на двоични модели

Много функции в този модул работят с шаблони. Например, String.split/3 може да раздели низ на множество низове, дадени на шаблон. Този модел може да бъде низ, списък от низове или компилиран шаблон:

```
String.split("foo bar", " ")
```

```
["foo", "bar"]
String.split("foo bar!", [" ", "!"])
["foo", "bar", ""]
pattern = :binary.compile_pattern([" ", "!"])
String.split("foo bar!", pattern)
["foo", "bar", ""]
```

Компилираният модел е полезен, когато едно и също съпоставяне ще се прави отново и отново. Обърнете внимание обаче, че компилираният шаблон не може да бъде съхранен в атрибут на модул, тъй като шаблонът се генерира по време на изпълнение и не оцелява след компилиране.

Функции	
codepoint()	Единична Unicode кодова точка, кодирана в UTF-8. Може да бъде един или повече байта.
grapheme()	Множество кодови точки, които могат да се възприемат като един знак от читателите
pattern()	Шаблон, използван във функции като replace/4 и split/3
t()	UTF-8 кодиран двоичен файл.
at(string, position)	Връща графемата в position дадения UTF-8 string. Ако position е по-голямо от дължината на string, тогава се връща nil.
bag_distance(string1, string2)	Изчислява разстоянието между два низа.
capitalize(string, mode \\ :default)	Преобразува първия знак в дадения низ в главни букви, а остатъка в малки според mode.
chunk(string, trait)	Разделя низа на части от знаци, които споделят обща черта.
codepoints(string)	Връща списък с кодови точки, кодирани като низове.
contains?(string, contents)	Проверява дали string съдържа някое от дадените contents.
downcase(string, mode \\ :default)	Преобразува всички знаци в дадения низ в малки букви според mode.
duplicate(subject, n)	Връща низ, subject повтарящ се n пъти.
ends_with?(string, suffix)	Връща, true ако string завършва с някой от дадените наставки.
equivalent?(string1, string2)	Връща true, ако string1е канонично еквивалентен на string2.
first(string)	Връща първата графема от UTF-8 низ, nilако низът е празен.
graphemes(string)	Връща Unicode графеми в низа според алгоритъма за разширен клъстер на графеми.
jaro_distance(string1, string2)	Изчислява Jaro разстоянието (сходството) между два низа.
last(string)	Връща последната графема от UTF-8 низ, nil ако низът е празен.
length(string)	Връща броя на Unicode графемите в UTF-8 низ.
match?(string, regex)	Проверява дали string съответства на дадения регулярен израз.
myers_difference(string1, string2) next_codepoint(string)	Връща списък с ключови думи, който представлява скрипт за редактиране. Връща следващата кодова точка в низ.
next_grapheme(binary)	Връща следващата графема в низ.
next_grapheme_size(string)	Връща размера (в байтове) на следващата графема.
normalize(string, form)	Преобразува всички знаци във string форма за нормализиране на Unicode, идентифицирана от form.
pad_leading(string, count, padding \\ [" "])	Връща нов низ, подплатен с водещ пълнител, който е направен от елементи от padding.
pad_trailing(string, count, padding \\ [" "])	Връща нов низ, подплатен със завършващ пълнител, който е направен от елементи от padding.
printable?(string, character_limit \\ :infinity)	Проверява дали даден низ съдържа само печатаеми знаци до character_limit.
replace(subject, pattern, replacement, options \\ [])	Връща нов низ, създаден чрез замяна на срещания на pattern в subject с replacement.
replace_leading(string, match, replacement)	Заменя всички водещи срещания на matchc replacementна matchв string.
replace_prefix(string, match, replacement)	Заменя префикса в string с replacement, ако съвпада match.
replace_suffix(string, match, replacement)	Заменя суфикса в string с replacement, ако съвпада match.
replace_trailing(string, match, replacement)	Заменя всички завършващи срещания на match с replacement в string.
reverse(string)	Обръща графемите в даден низ.
slice(string, range)	Връща подниз от отместването, дадено от началото на диапазона, до отместването, дадено от края на диапазона.
slice(string, start, length)	Връща подниз, започващ от отместването start и на дадения length.
split(binary)	Разделя низ на поднизове при всяко появяване на интервали в Unicode, като празните интервали в началото и края се игнорират. Групите празни интервали се третират като едно събитие. Деленията не се появяват върху празно пространство без прекъсване.
split(string, pattern, options \\ [])	Разделя низ на части въз основа на модел.
split_at(string, position)	Разделя низ на две при указаното отместване. Когато даденото отместване е отрицателно, местоположението се брои от края на низа.
splitter(string, pattern, options \\ [])	Връща изброимо, което разделя низ при поискване.
starts_with?(string, prefix)	Връща, trueако string започва с някой от дадените префикси.
to_atom(string)	Преобразува низ в атом.
to_charlist(string)	Преобразува низ в списък със знаци.
to_existing_atom(string)	Преобразува низ в съществуващ атом.
to_float(string)	Връща float, чието текстово представяне е string.
to_integer(string)	Връща цяло число, чието текстово представяне е string.

to_integer(string, base)	Връща цяло число, чието текстово представяне е stringв base base.
trim(string)	Връща низ, където всички начални и завършващи интервали в Unicode са премахнати.
trim(string, to_trim)	Връща низ, където всички водещи и завършващи to_trim знаци са премахнати.
trim_leading(string)	Връща низ, където всички водещи бели интервали в Unicode са премахнати.
trim_leading(string, to_trim)	Връща низ, където всички водещи to_trimзнаци са премахнати.
trim_trailing(string)	Връща низ, където всички бели интервали в Unicode в края са премахнати.
trim_trailing(string, to_trim)	Връща низ, където всички to_trim знаци в края са премахнати.
upcase(string, mode \\ :default)	Преобразува всички знаци в дадения низ в главни букви според mode.
valid?(arg1)	Проверява дали string съдържа само валидни знаци.
codepoint() @type codepoint() :: t()	Единична Unicode кодова точка, кодирана в UTF-8. Може да бъде един или повече байта.
grapheme() @type grapheme() :: t()	Множество кодови точки, които могат да се възприемат като един знак от читателите
pattern() @type pattern() :: t() | [t()] | :binary.cp()	Шаблон, използван във функции като replace/4 и split/3
t() @type t() :: binary()	UTF-8 кодиран двоичен файл.

Типовете String.t() и binary() са еквивалентни на инструментите за анализ. Въпреки че за тези, които четат документацията, String.t() това предполага, че това е UTF-8 кодиран двоичен файл.

Функции

at(string, position)

@spec at(t(), integer()) :: grapheme() | nil

Връща графемата в positionдадения UTF-8 string. Ако position е по-голямо от дължината на string, тогава се връща nil.

Пример:

```
String.at("elixir", 0)
"e"
String.at("elixir", 1)
"l"
String.at("elixir", 10)
nil
String.at("elixir", -1)
"r"
String.at("elixir", -10)
nil
```

bag_distance(string1, string2)(от 1.8.0)

@spec bag_distance(t(), t()) :: float()

Изчислява разстоянието между два низа.
Връща плаваща стойност между 0 и 1, представляваща разстоянието на чантата между string1и string2.
Разстоянието на чантата е предназначено да бъде ефективно приближение на разстоянието между две струни за бързо изключване на струни, които са до голяма степен различни.
Алгоритъмът е описан в статията „Съпоставяне на низове с метрични дървета с помощта на приблизително разстояние“ от Илария Бартолини, Паоло Чача и Марко Патела.

Пример:

```
String.bag_distance("abc", "")
0.0
String.bag_distance("abcd", "a")
0.25
String.bag_distance("abcd", "ab")
0.5
String.bag_distance("abcd", "abc")
0.75
String.bag_distance("abcd", "abcd")
1.0
```

capitalize(string, mode \\ :default)

@spec capitalize(t(), :default | :ascii | :greek | :turkic) :: t()

Преобразува първия знак в дадения низ в главни букви, а остатъка в малки според mode.
mode може да бъде :default, :ascii или :greek. :turkic. Режимът :default взема предвид всички безусловни трансформации, описани в стандарта Unicode. :ascii изписва с главни букви само буквите от A до Z. :greek включва чувствителните към контекста съпоставяния, открити на гръцки. :turkic правилно борави с буквата i с варианта без точка.

Пример:

```
String.capitalize("abcd")
```

```
"Abcd"
String.capitalize("fin")
"Fin"
String.capitalize("olá")
"Olá"
```

chunk(string, trait)
@spec chunk(t(), :valid | :printable) :: [t()]

Разделя низа на части от знаци, които споделят обща черта.
Характеристиката може да бъде една от двете опции:

:valid - низът се разделя на части от валидни и невалидни символни последователности
:printable - низът се разделя на части от печатаеми и непечатаеми символни последователности
Връща списък от двоични файлове, всяка от които съдържа само един вид знаци.
Ако даденият низ е празен, се връща празен списък.

Пример:

```
String.chunk(<<?a, ?b, ?c, 0>>, :valid)
["abc\0"]
String.chunk(<<?a, ?b, ?c, 0, 0xFFFF::utf16>>, :valid)
["abc\0", <<0xFFFF::utf16>>]
String.chunk(<<?a, ?b, ?c, 0, 0x0FFF::utf8>>, :printable)
["abc", <<0, 0x0FFF::utf8>>]
```

codepoints(string)
@spec codepoints(t()) :: [codepoint()]

Връща списък с кодови точки, кодирани като низове.
За да извлечете кодови точки в тяхното естествено цяло число, вижте to_charlist/1. За подробности относно кодовите точки и графемите вижте String документацията на модула.

Пример:

```
String.codepoints("olá")
["o", "l", "á"]
String.codepoints("оптими зации")
["о", "п", "т", "и", "м", "и", " ", "з", "а", "ц", "и", "и"]
String.codepoints("α̈ΗΩ")
["α̈", "̈Η", "Ω"]
String.codepoints("\u00e9")
["é"]
String.codepoints("\u0065\u0301")
["e", "́"]
```

contains?(string, contents)
@spec contains?(t(), pattern()) :: boolean()
Проверява дали string съдържа някое от дадените contents.
contents може да бъде или низ, списък от низове или компилиран модел.

Пример:

```
String.contains?("elixir of life", "of")
true
String.contains?("elixir of life", ["life", "death"])
true
String.contains?("elixir of life", ["death", "mercury"])
false
```

Аргументът може също да бъде компилиран модел:

```
pattern = :binary.compile_pattern(["life", "death"])
String.contains?("elixir of life", pattern)
true
```

Празен низ винаги ще съответства:

```
String.contains?("elixir of life", "")
true
String.contains?("elixir of life", ["", "other"])
true
```

Имайте предвид, че тази функция може да съвпада в рамките на или извън границите на графема. Например, вземете графемата "é", която е съставена от буквите "e" и острото ударение. Следните връщания true:

```
String.contains?(String.normalize("é", :nfd), "e")
true
```

Въпреки това, ако "é" е представено от единичен знак "e с остро ударение", тогава ще се върне false:

```
String.contains?(String.normalize("é", :nfc), "e")
false
downcase(string, mode \\ :default)
```

@spec downcase(t(), :default | :ascii | :greek | :turkic) :: t()
Преобразува всички знаци в дадения низ в малки букви според mode.

modeможе да бъде :default, :ascii или :greek. :turkic Режимът :default взема предвид всички безусловни трансформации, описани в стандарта Unicode. :ascii малки букви само буквите от A до Z. :greek включва чувствителните към контекста съпоставяния, открити на гръцки. :turkic правилно борави с буквата i с варианта без точка.

Пример:

```
String.downcase("ABCD")
"abcd"
String.downcase("AB 123 XPTO")
"ab 123 xpto"
String.downcase("OLÁ")
"olá"
```

Режимът :ascii игнорира Unicode символи и осигурява по-ефективна реализация, когато знаете, че низът съдържа само ASCII знаци:

```
String.downcase("OLÁ", :ascii)
"olÁ"
```

Режимът :greek правилно обработва чувствителната към контекста сигма на гръцки:

```
String.downcase("ΣΣ")
"σσ"
String.downcase("ΣΣ", :greek)
"σς"
```

И :turki с правилно обработва буквата i с варианта без точка:

```
String.downcase("İİ")
"ii"
String.downcase("İİ", :turkic)
"ii"
```

duplicate(subject, n)
@spec duplicate(t(), non_neg_integer()) :: t()

Връща низ, subjectповтарящ се nпъти.
Вграден от компилатора.

Пример:

```
String.duplicate("abc", 0)
""
String.duplicate("abc", 1)
"abc"
String.duplicate("abc", 2)
"abcabc"
```

ends_with?(string, suffix)
@spec ends_with?(t(), t() | [t()]) :: boolean()

Връща, true ако string завършва с някой от дадените наставки.
suffixes може да бъде или един суфикс, или списък от суфикси.

Пример:

```
String.ends_with?("language", "age")
true
String.ends_with?("language", ["youth", "age"])
true
String.ends_with?("language", ["youth", "elixir"])
false
```

Празен суфикс винаги ще съответства:

```
String.ends_with?("language", "")
true
String.ends_with?("language", ["", "other"])
true
```

equivalent?(string1, string2)
@spec equivalent?(t(), t()) :: boolean()

Връща true, ако string1е канонично еквивалентен на string2.
Той извършва канонично разлагане на форма за нормализиране (NFD) върху низовете, преди да ги сравни. Тази функция е еквивалентна на: **String.normalize(string1, :nfd) == String.normalize(string2, :nfd)**
Ако планирате да сравнявате множество низове, няколко пъти подред, можете да ги нормализирате предварително и да ги сравнявате директно, за да избегнете множество преминавания за нормализиране.

Пример:

```
String.equivalent?("abc", "abc")
true
String.equivalent?("man\u0303ana", "mañana")
true
String.equivalent?("abc", "ABC")
false
String.equivalent?("nø", "nó")
false
first(string)
```

@spec first(t()) :: grapheme() | nil
Връща първата графема от UTF-8 низ, nilако низът е празен.

Пример:

```
String.first("elixir")
"e"
String.first("ʇnqɟh")
"ʇ"
String.first("")
nil
```

graphemes(string)
@spec graphemes(t()) :: [grapheme()]

Връща Unicode графеми в низа според алгоритъма за разширен клъстер на графеми.
Алгоритъмът е описан в стандартното приложение на Unicode #29, сегментиране на текст на Unicode .
За подробности относно кодовите точки и графемите вижте Stringдокументацията на модула.

Пример:

```
String.graphemes("Ńaïve")
["Ń", "a", "ï", "v", "e"]
String.graphemes("\u00e9")
["é"]
String.graphemes("\u0065\u0301")
["é"]
```

jaro_distance(string1, string2)
@spec jaro_distance(t(), t()) :: float()

Изчислява Jaro разстоянието (сходството) между два низа.
Връща плаваща стойност между 0.0(равнява се на липса на сходство) и 1.0(е точно съвпадение), представляваща Jaro разстоянието между string1и string2.
Метриката за разстояние на Jaro е проектирана и най-подходяща за къси низове като имена на хора. Самият Elixir използва тази функция, за да предостави "Имате предвид?" функционалност. Например, когато извиквате функция в модул и имате правописна грешка в името

на функцията, ние се опитваме да предложим най-подобното налично име на функция, ако има такова, въз основа на резултата jaro_distance/2.

Пример:

```
String.jaro_distance("Dwayne", "Duane")
0.8222222222222223
String.jaro_distance("even", "odd")
0.0
String.jaro_distance("same", "same")
1.0
```

last(string)
@spec last(t()) :: grapheme() | nil
Връща последната графема от UTF-8 низ, nilако низът е празен.

Пример:

```
String.last("elixir")
"r"
String.last("ʇnqɭh")
"h"
```

length(string)
@spec length(t()) :: non_neg_integer()
Връща броя на Unicode графемите в UTF-8 низ.

Пример:

```
String.length("elixir")
6

String.length("ʇnqɭh")
5
```

match?(string, regex):: b
@spec match?(t(), Regex.t()) oolean()
Проверява дали string съответства на дадения регулярен израз.

Пример:

```
String.match?("foo", ~r/foo/)
true
String.match?("bar", ~r/foo/)
false
```

myers_difference(string1, string2)(от 1.3.0)
@spec myers_difference(t(), t()) :: [{:eq | :ins | :del, t()}]

Връща списък с ключови думи, който представлява скрипт за редактиране.
Проверете List.myers_difference/2за повече информация.

Пример:

```
string1 = "fox hops over the dog"
string2 = "fox jumps over the lazy cat"
String.myers_difference(string1, string2)
[eq: "fox ", del: "ho", ins: "jum", eq: "ps over the ", del: "dog", ins: "lazy cat"]
```

next_codepoint(string)
@spec next_codepoint(t()) :: {codepoint(), t()} | nil
Връща следващата кодова точка в низ.

Резултатът е кортеж с кодовата точка и остатъка от низа или nilв случай, че низът е достигнал своя край.
Както при другите функции в Stringмодула, next_codepoint/1работи с двоични файлове, които са невалидни UTF-8. Ако низът започва с поредица от байтове, която не е валидна в UTF-8 кодиране, първият елемент на върнатия кортеж е двоичен с първия байт.

Пример:

```
String.next_codepoint("olá")
{"o", "lá"}
invalid = "\x80\x80OK" # first two bytes are invalid in UTF-8
```

```
{_, rest} = String.next_codepoint(invalid)
{<<128>>, <<128, 79, 75>>}
String.next_codepoint(rest)
{<<128>>, "OK"}
```

Сравнение със съпоставяне на двоичен модел

Съпоставянето на двоичен модел осигурява подобен начин за разлагане на низ:

```
<<codepoint::utf8, rest::binary>> = "Elixir"
"Elixir"
codepoint
69
rest
"lixir"
```

въпреки че не е напълно еквивалентен codepoint, защото идва като цяло число и моделът няма да съвпада с невалиден UTF-8.
Съпоставянето на двоични шаблони обаче е по-просто и по-ефективно, така че изберете опцията, която е по-подходяща за вашия случай на употреба.

next_grapheme(binary)
@spec next_grapheme(t()) :: {grapheme(), t()} | nil

Връща следващата графема в низ.
Резултатът е кортеж с графемата и остатъка от низа или nilв случай, че низът е достигнал своя край.

Пример:

```
String.next_grapheme("olá")
{"o", "lá"}
String.next_grapheme("")
nil
```

next_grapheme_size(string)
@spec next_grapheme_size(t()) :: {pos_integer(), t()} | nil

Връща размера (в байтове) на следващата графема.
Резултатът е кортеж със следващия размер на графема в байтове и остатъка от низа или nilв случай, че низът е достигнал своя край.

Пример:

```
String.next_grapheme_size("olá")
{1, "lá"}
String.next_grapheme_size("")
nil
```

normalize(string, form)

Преобразува всички знаци във stringформа за нормализиране на Unicode, идентифицирана от form.
Невалидните Unicode кодови точки се пропускат и останалата част от низа се конвертира. Ако искате алгоритъмът да спре и да се върне при невалидна кодова точка, използвайте :unicode.characters_to_nfd_binary/1, :unicode.characters_to_nfc_binary/1, :unicode.characters_to_nfkd_binary/1и :unicode.characters_to_nfkc_binary/1вместо това.
Формите за нормализиране :nfkcне :nfkdтрябва да се прилагат сляпо към произволен текст. Тъй като те изтриват много различия във форматирането, те ще предотвратят двупосочно преобразуване към и от много наследени набори от знаци.

Форми

Поддържаните форми са:
:nfd- Нормализираща форма на канонично разлагане. Символите се разлагат чрез канонична еквивалентност и множество комбиниращи знаци се подреждат в определен ред.

:nfc- Нормализация Форма Каноничен състав. Символите се разлагат и след това се прекомпозират чрез канонична еквивалентност.

:nfkd- Декомпозиция на съвместимост на форма за нормализиране. Знаците се разлагат чрез еквивалентност на съвместимост и множество комбиниращи знаци се подреждат в определен ред.

:nfkc- Състав за съвместимост на формата за нормализиране. Символите се разлагат и след това се прекомпозират чрез еквивалентност на съвместимост.

Примери:

```
String.normalize("yêṩ", :nfd)
"yêṩ"
String.normalize("leña", :nfc)
```

```
"leña"
String.normalize("fi", :nfkd)
"fi"
String.normalize("fi", :nfkc)
"fi"
```

pad_leading(string, count, padding \\ [" "])
@spec pad_leading(t(), non_neg_integer(), t() | [t()]) :: t()
Връща нов низ, подплатен с водещ пълнител, който е направен от елементи от padding.

Предаването на списък от низове paddingще вземе един елемент от списъка за всеки липсващ запис. Ако списъкът е по-кратък от броя на вложките, попълването ще започне отново от началото на списъка. Предаването на низ paddinge еквивалентно на предаването на списъка с графеми в него. Ако paddinge дадено не, по подразбиране се използва интервал.
Когато counte по-малка или равна на дължината на string, даденото stringce връща.
Повишава ArgumentError, ако даденото paddingсъдържа елемент, който не е низ.

Пример:

```
String.pad_leading("abc", 5)
"  abc"
String.pad_leading("abc", 4, "12")
"1abc"
String.pad_leading("abc", 6, "12")
"121abc"
String.pad_leading("abc", 5, ["1", "23"])
"123abc"
```

pad_trailing(string, count, padding \\ [" "])
@spec pad_trailing(t(), non_neg_integer(), t() | [t()]) :: t()

Връща нов низ, подплатен със завършващ пълнител, който е направен от елементи от padding.
Предаването на списък от низове paddingще вземе един елемент от списъка за всеки липсващ запис. Ако списъкът е по-кратък от броя на вложките, попълването ще започне отново от началото на списъка. Предаването на низ paddinge еквивалентно на предаването на списъка с графеми в него. Ако paddinge дадено не, по подразбиране се използва интервал.
Когато counte по-малка или равна на дължината на string, даденото stringce връща.
Повишава ArgumentError, ако даденото paddingсъдържа елемент, който не е низ.

Пример:

```
String.pad_trailing("abc", 5)
"abc  "
String.pad_trailing("abc", 4, "12")
"abc1"
String.pad_trailing("abc", 6, "12")
"abc121"
String.pad_trailing("abc", 5, ["1", "23"])
"abc123"
```

printable?(string, character_limit \\ :infinity)
@spec printable?(t(), 0) :: true
@spec printable?(t(), pos_integer() | :infinity) :: boolean()

Проверява дали даден низ съдържа само печатаеми знаци до character_limit.
Взема незадължителен character_limitкато втори аргумент. Ако character_limite 0, тази функция ще се върне true.

Пример:

```
String.printable?("abc")
true
String.printable?("abc" <> <<0>>)
false
String.printable?("abc" <> <<0>>, 2)
true
String.printable?("abc" <> <<0>>, 0)
true
```

replace(subject, pattern, replacement, options \\ [])
@spec replace(t(), pattern() | Regex.t(), t() | (t() -> t() | iodata()), keyword()) ::
  t()

Връща нов низ, създаден чрез замяна на срещания на patternв subjectс replacement.

Винаги е subject низ.
Може patternда бъде низ, списък от низове, регулярен израз или компилиран модел.
Може replacementда е низ или функция, която получава съответстващия модел и трябва да върне замяната като низ или iodata.
По подразбиране той замества всички срещания, но това поведение може да се контролира чрез :globalопцията; вижте раздела "Опции" по-долу.

Настроики
:global- (булев) ако true, всички срещания на patternce заменят с replacement, в противен случай се заменя само първото срещане. По подразбиране е true

Пример:

```
String.replace("a,b,c", ",", "-")
"a-b-c"
String.replace("a,b,c", ",", "-", global: false)
"a-b,c"
```

Моделът може също да бъде списък от низове и замяната може също да бъде функция, която получава съвпаденията:

```
String.replace("a,b,c", ["a", "c"], fn <<char>> -> <<char + 1>> end)
"b,b,d"
```

Когато моделът е регулярен израз, човек може да даде \Nили \g{N}в replacementниза за достъп до конкретно улавяне в регулярния израз:

```
String.replace("a,b,c", ~r/,(.)/, ",\\1\\g{1}")
"a,bb,cc"
```

Обърнете внимание, че трябваше да екранираме обратната наклонена черта (т.е. използвахме \\Nвместо само \Nда екранираме обратната наклонена черта; същото нещо за \\g{N}). Като даде \0, може да се инжектира цялото съвпадение в заместващия низ.

Може да се даде и компилиран модел:

```
pattern = :binary.compile_pattern(",")
String.replace("a,b,c", pattern, "[]")
"a[]b[]c"
```

Когато празен низ е предоставен като pattern, функцията ще го третира като косвен празен низ между всяка графема и низът ще бъде разпръснат. Ако е предоставен празен низ, replacementще subjectбъде върнато:

```
String.replace("ELIXIR", "", ".")
".E.L.I.X.I.R."
String.replace("ELIXIR", "", "")
"ELIXIR"
```

replace_leading(string, match, replacement)
@spec replace_leading(t(), t(), t()) :: t()

Заменя всички водещи срещания на matchc replacementна matchв string.
Връща низа недокоснат, ако няма срещания.
Ако matche "", тази функция предизвиква ArgumentErrorизключение: това се случва, защото тази функция замества всички срещания на matchв началото на stringи е невъзможно да се заменят „множество“ появявания на "".

Пример:

```
String.replace_leading("hello world", "hello ", "")
"world"
String.replace_leading("hello hello world", "hello ", "")
"world"
String.replace_leading("hello world", "hello ", "ola ")
"ola world"
String.replace_leading("hello hello world", "hello ", "ola ")
"ola ola world"
```

replace_prefix(string, match, replacement)
@spec replace_prefix(t(), t(), t()) :: t()

Заменя префикса в stringc replacement, ако съвпада match.
Връща низа недокоснат, ако няма съвпадение. Ако matche празен низ (""), replacementпросто се добавя към string.

Пример:

```
String.replace_prefix("world", "hello ", "")
```

```
"world"
String.replace_prefix("hello world", "hello ", "")
"world"
String.replace_prefix("hello hello world", "hello ", "")
"hello world"
String.replace_prefix("world", "hello ", "ola ")
"world"
String.replace_prefix("hello world", "hello ", "ola ")
"ola world"
String.replace_prefix("hello hello world", "hello ", "ola ")
"ola hello world"
String.replace_prefix("world", "", "hello ")
"hello world"
```

replace_suffix(string, match, replacement)
@spec replace_suffix(t(), t(), t()) :: t()

Заменя суфикса в stringc replacement, ако съвпада match.
Връща низа недокоснат, ако няма съвпадение. Ако matche празен низ (""), replacementпросто се добавя към string.

Пример:

```
String.replace_suffix("hello", " world", "")
"hello"
String.replace_suffix("hello world", " world", "")
"hello"
String.replace_suffix("hello world world", " world", "")
"hello world"
String.replace_suffix("hello", " world", " mundo")
"hello"
String.replace_suffix("hello world", " world", " mundo")
"hello mundo"
String.replace_suffix("hello world world", " world", " mundo")
"hello world mundo"
String.replace_suffix("hello", "", " world")
"hello world"
```

replace_trailing(string, match, replacement)
@spec replace_trailing(t(), t(), t()) :: t()
Заменя всички завършващи срещания на matchc replacementв string.

Връща низа недокоснат, ако няма срещания.
Ако matche "", тази функция предизвиква ArgumentErrorизключение: това се случва, защото тази функция замества всички срещания на matchв края на stringи е невъзможно да се заменят „множество“ появявания на "".

Пример:

```
String.replace_trailing("hello world", " world", "")
"hello"
String.replace_trailing("hello world world", " world", "")
"hello"
String.replace_trailing("hello world", " world", " mundo")
"hello mundo"
String.replace_trailing("hello world world", " world", " mundo")
"hello mundo mundo"
```

reverse(string)

@spec reverse(t()) :: t()
Обръща графемите в даден низ.

Пример:

```
String.reverse("abcd")
"dcba"

String.reverse("hello world")
"dlrow olleh"

String.reverse("hello ∂og")
"go∂ olleh"
```

Имайте предвид, че обръщането на същия низ два пъти не води непременно до оригиналния низ:

```
"̀e"
"è"
String.reverse("̀e")
"è"
String.reverse(String.reverse("̀e"))
"è"
```

В първия пример ударението е пред гласната, така че се счита за две графеми. Когато обаче го обърнете веднъж, имате гласната, последвана от ударението, което се превръща в една графема. Обръщането му отново ще го запази като една единствена графема.

slice(string, range)
@spec slice(t(), Range.t()) :: t()

Връща подниз от отместването, дадено от началото на диапазона, до отместването, дадено от края на диапазона.
Ако началото на диапазона не е валидно отместване за дадения низ или ако диапазонът е в обратен ред, връща "".
Ако началото или краят на диапазона е отрицателен, първо се преминава през целия низ, за да се преобразуват отрицателните индекси в положителни.
Не забравяйте, че тази функция работи с Unicode графеми и счита, че срезовете представляват отмествания на графеми. Ако искате да разделите на необработени байтове, проверете Kernel.binary_part/3вместо това.

Пример:

```
String.slice("elixir", 1..3)
"lix"
String.slice("elixir", 1..10)
"lixir"
String.slice("elixir", -4..-1)
"ixir"
String.slice("elixir", -4..6)
"ixir"
```

За диапазони start > stop, където трябва изрично да ги маркирате като нарастващи:

```
String.slice("elixir", 2..-1//1)
"ixir"
String.slice("elixir", 1..-2//1)
"lixi"
```

Ако стойностите са извън границите, той връща празен низ:

```
String.slice("elixir", 10..3)
""
String.slice("elixir", -10..-7)
""
String.slice("a", 0..1500)
"a"
String.slice("a", 1..1500)
""
```

slice(string, start, length)
@spec slice(t(), integer(), non_neg_integer()) :: grapheme()
Връща подниз, започващ от отместването starти на дадения length.
Ако отместването е по-голямо от дължината на низа, то връща "".
Не забравяйте, че тази функция работи с Unicode графеми и счита, че срезовете представляват отмествания на графеми. Ако искате да разделите на необработени байтове, проверете Kernel.binary_part/3вместо това.

Пример:

```
String.slice("elixir", 1, 3)
"lix"
String.slice("elixir", 1, 10)
"lixir"
String.slice("elixir", 10, 3)
""
String.slice("elixir", -4, 4)
"ixir"
String.slice("elixir", -10, 3)
""
String.slice("a", 0, 1500)
"a"
String.slice("a", 1, 1500)
```

```
""
String.slice("a", 2, 1500)
""
```

split(binary)
@spec split(t()) :: [t()]

Разделя низ на поднизове при всяко появяване на интервали в Unicode, като празните интервали в началото и края се игнорират. Групите празни интервали се третират като едно събитие. Деленията не се появяват върху празно пространство без прекъсване.

Пример:

```
String.split("foo bar")
["foo", "bar"]

String.split("foo" <> <<194, 133>> <> "bar")
["foo", "bar"]

String.split(" foo   bar ")
["foo", "bar"]

String.split("no\u00a0break")
["no\u00a0break"]
```

split(string, pattern, options \\ [])
@spec split(t(), pattern() | Regex.t(), keyword()) :: [t()]

Разделя низ на части въз основа на модел.
Връща списък с тези части.
Може pattern да бъде низ, списък от низове, регулярен израз или компилиран модел.
По подразбиране низът е разделен на възможно най-много части, но може да се контролира чрез опцията :parts.
Празните низове се премахват от резултата само ако :trimопцията е зададена на true.
Когато използваният модел е регулярен израз, низът се разделя с помощта на Regex.split/3.
Настроики
:parts(цяло положително число или :infinity) - низът се разделя на най-много толкова части, колкото задава тази опция. Ако :infinity, низът ще бъде разделен на всички възможни части. По подразбиране е :infinity.
:trim(булев) - ако true, празните низове се премахват от получения списък.
Тази функция също приема всички опции, приети от Regex.split/3if patterne регулярен израз.

Пример:
Разделяне с модел на низ:

```
String.split("a,b,c", ",")
["a", "b", "c"]
String.split("a,b,c", ",", parts: 2)
["a", "b,c"]
String.split(" a b c ", " ", trim: true)
["a", "b", "c"]
```

Списък с модели:

```
String.split("1,2 3,4", [" ", ","])
["1", "2", "3", "4"]
Регулярен израз:
String.split("a,b,c", ~r{,})
["a", "b", "c"]
String.split("a,b,c", ~r{,}, parts: 2)
["a", "b,c"]
String.split(" a b c ", ~r{\s}, trim: true)
["a", "b", "c"]
String.split("abc", ~r{b}, include_captures: true)
["a", "b", "c"]
```

Компилиран модел:

```
pattern = :binary.compile_pattern([" ", ","])
String.split("1,2 3,4", pattern)
["1", "2", "3", "4"]
Разделянето на празен низ връща графеми:
String.split("abc", "")
```

```
["", "a", "b", "c", ""]
String.split("abc", "", trim: true)
["a", "b", "c"]
String.split("abc", "", parts: 1)
["abc"]
String.split("abc", "", parts: 3)
["", "a", "bc"]
```

Имайте предвид, че тази функция може да се раздели в рамките на или през границите на графема. Например, вземете графемата "é", която е съставена от буквите "e" и острото ударение. Следното ще раздели низа на две части:

```
String.split(String.normalize("é", :nfd), "e")
["", ""]
```

Ако обаче "é" е представено от единичния знак "e с остро ударение", то ще раздели низа само на една част:

```
String.split(String.normalize("é", :nfc), "e")
["é"]
```

split_at(string, position)
@spec split_at(t(), integer()) :: {t(), t()}

Разделя низ на две при указаното отместване. Когато даденото отместване е отрицателно, местоположението се брои от края на низа.
Отместването е ограничено до дължината на низа. Връща кортеж с два елемента.
Забележка: имайте предвид, че тази функция се разделя на графеми и за това трябва да премине линейно през низа. Ако искате да разделите низ или двоичен файл въз основа на броя байтове, използвайте Kernel.binary_part/3вместо това.

Пример:

```
String.split_at("sweetelixir", 5)
{"sweet", "elixir"}
String.split_at("sweetelixir", -6)
{"sweet", "elixir"}
String.split_at("abc", 0)
{"", "abc"}
String.split_at("abc", 1000)
{"abc", ""}
String.split_at("abc", -1000)
{"", "abc"}
```

splitter(string, pattern, options \\ [])
@spec splitter(t(), pattern(), keyword()) :: Enumerable.t()

Връща изброимо, което разделя низ при поискване.
Това е в контраст с split/3това, което разделя целия низ отпред.
Тази функция не поддържа регулярни изрази по дизайн. Когато използвате регулярни изрази, често е по-ефективно регулярните изрази да преминават през низа наведнъж, отколкото на части, както прави тази функция.

Настроики
:trim - когато true, не излъчва празни шаблони

Пример:

```
String.splitter("1,2 3,4 5,6 7,8,...,99999", [" ", ","]) |> Enum.take(4)
["1", "2", "3", "4"]
String.splitter("abcd", "") |> Enum.take(10)
["", "a", "b", "c", "d", ""]
String.splitter("abcd", "", trim: true) |> Enum.take(10)
["a", "b", "c", "d"]
```

Може да се даде и компилиран модел:

```
pattern = :binary.compile_pattern([" ", ","])
String.splitter("1,2 3,4 5,6 7,8,...,99999", pattern) |> Enum.take(4)
["1", "2", "3", "4"]
```

starts_with?(string, prefix)

@spec starts_with?(t(), pattern()) :: boolean()
Връща, trueако stringзапочва с някой от дадените префикси.

prefixможе да бъде или низ, списък от низове или компилиран модел.

Пример:

```
String.starts_with?("elixir", "eli")
true
String.starts_with?("elixir", ["erlang", "elixir"])
true
String.starts_with?("elixir", ["erlang", "ruby"])
false
```

Може да се даде и компилиран модел:

```
pattern = :binary.compile_pattern(["erlang", "elixir"])
String.starts_with?("elixir", pattern)
true
```

Празен низ винаги ще съответства:

```
String.starts_with?("elixir", "")
true
String.starts_with?("elixir", ["", "other"])
true
```

to_atom(string)
@spec to_atom(t()) :: atom()

Преобразува низ в атом.
Предупреждение: тази функция създава атоми динамично и атомите не се събират. Следователно stringне трябва да бъде ненадеждна стойност, като например въвеждане, получено от сокет или по време на уеб заявка. Помислете за използване to_existing_atom/1вместо това.
По подразбиране максималният брой атоми е 1_048_576. Тази граница може да бъде увеличена или намалена чрез опцията VM +t.
Максималният размер на атома е 255 Unicode кодови точки.
Вграден от компилатора.

Пример:

```
String.to_atom("my_atom")
:my_atom
```

to_charlist(string)
@spec to_charlist(t()) :: charlist()
Преобразува низ в списък със знаци.
По-конкретно, тази функция приема UTF-8 кодиран двоичен файл и връща списък с неговите цели кодови точки. Подобен е на този, codepoints/1с изключение на това, че последният връща списък с кодови точки като низове.
В случай, че трябва да работите с байтове, вижте :binaryмодула .

Пример:

```
String.to_charlist("æß")
'æß'
```

to_existing_atom(string)
@spec to_existing_atom(t()) :: atom()

Преобразува низ в съществуващ атом.
Максималният размер на атома е 255 Unicode кодови точки.
Вграден от компилатора.

Пример:

```
_ = :my_atom
String.to_existing_atom("my_atom")
:my_atom
```

to_float(string)
@spec to_float(t()) :: float()

Връща float, чието текстово представяне е string.
stringтрябва да бъде низовото представяне на число с плаваща запетая, включително десетична точка. За да анализирате низ без десетична запетая като float тогава Float.parse/1трябва да се използва. В противен случай ArgumentErrorще бъде повдигнато.

Вграден от компилатора.

Пример:

```
String.to_float("2.2017764e+0")
2.2017764
String.to_float("3.0")
3.0
String.to_float("3")
** (ArgumentError) argument error
```

to_integer(string)
@spec to_integer(t()) :: integer()
Връща цяло число, чието текстово представяне е string.

string трябва да бъде низово представяне на цяло число. В противен случай ArgumentErrorще бъде повдигнато. Ако искате да анализирате низ, който може да съдържа неправилно форматирано цяло число, използвайте Integer.parse/1.
Вграден от компилатора.

Пример:

```
String.to_integer("123")
123
```

Предаването на низ, който не представлява цяло число, води до грешка:

```
String.to_integer("invalid data")
** (ArgumentError) argument error
```

to_integer(string, base)
@spec to_integer(t(), 2..36) :: integer()

Връща цяло число, чието текстово представяне е stringв base base.
Вграден от компилатора.

Пример:

```
String.to_integer("3FF", 16)
1023
```

trim(string)
@spec trim(t()) :: t()

Връща низ, където всички начални и завършващи интервали в Unicode са премахнати.

Пример:

```
String.trim("\n  abc\n  ")
"abc"
```

trim(string, to_trim)
@spec trim(t(), t()) :: t()

Връща низ, където всички водещи и завършващи to_trimзнаци са премахнати.
Пример:

```
String.trim("a  abc  a", "a")
"  abc  "
```

trim_leading(string)
@spec trim_leading(t()) :: t()
Връща низ, където всички водещи бели интервали в Unicode са премахнати.

Пример:

```
String.trim_leading("\n  abc   ")
"abc   "
```

trim_leading(string, to_trim)

@spec trim_leading(t(), t()) :: t()
Връща низ, където всички водещи to_trimзнаци са премахнати.

Пример:

```
String.trim_leading("__ abc _", "_")
```

```
" abc _"
String.trim_leading("1 abc", "11")
"1 abc"
```

trim_trailing(string)
@spec trim_trailing(t()) :: t()

Връща низ, където всички бели интервали в Unicode в края са премахнати.

Пример:

```
String.trim_trailing("   abc\n  ")
"   abc"
trim_trailing(string, to_trim)
```

@spec trim_trailing(t(), t()) :: t()
Връща низ, където всички to_trimзнаци в края са премахнати.

Пример:

```
String.trim_trailing("_ abc __", "_")
"_ abc "
String.trim_trailing("abc 1", "11")
"abc 1"
```

upcase(string, mode \\ :default)
@spec upcase(t(), :default | :ascii | :greek | :turkic) :: t()

Преобразува всички знаци в дадения низ в главни букви според mode.
mode може да бъде :default, :asciiили :greek. :turkicРежимът :defaultвзема предвид всички безусловни трансформации, описани в стандарта Unicode. :asciiглавни букви само от a до z. :greekвключва чувствителните към контекста съпоставяния, намерени на гръцки. :turkicправилно борави с буквата i с варианта без точка.

Пример:

```
String.upcase("abcd")
"ABCD"
String.upcase("ab 123 xpto")
"AB 123 XPTO"
String.upcase("olá")
"OLÁ"
```

Режимът :ascii игнорира Unicode символи и осигурява по-ефективна реализация, когато знаете, че низът съдържа само ASCII знаци:

```
String.upcase("olá", :ascii)
"OLá"
```

И :turki с правилно обработва буквата i с варианта без точка:

```
String.upcase("ıi")
"II"
String.upcase("ıi", :turkic)
"Iİ"
```

valid?(arg1)
@spec valid?(t()) :: boolean()
Проверява дали string съдържа само валидни знаци.

Пример:

```
String.valid?("a")
true
String.valid?("ø")
true
String.valid?(<<0xFFFF::16>>)
false
String.valid?(<<0xEF, 0xB7, 0x90>>)
true
String.valid?("asd" <> <<0xFFFF::16>>)
false
```