Erlang: Магия

Манипулируя битами и байтами, **Андрей Ушаков** колдует над строками: то перестроит их в другом порядке, то инвертирует...

Наш эксперт

Андрей Ушаков активно приближает тот день, когда функциональные языки станут мейнстримом.

В этом номере мы продолжаем наш практикум по функциональному программированию на языке **Erlang** и разберемся с«черной магией» битовых строк языка **Erlang**.

Давайте вспомним, что такое битовые строки и для чего они нужны (более подробно см. **LXF148**). Битовые строки – это тип данных, позволяющий работать с низкоуровневыми данными более структурированным, чем набор (массив) байт, способом. Это означает, что мы можем задавать или извлекать данные порциями (которые называются сегментами) произвольного размера, имеющими один из предопределенных типов. Задавать сегменты мы можем как в выражениях, так и в операциях соответствия шаблону [pattern-matching], а извлекать данные сегментами – только в операциях соответствия шаблону. Каждый сегмент состоит из значения или переменной (инициализированной либо неинициализированной), размера и списка спецификаторов, определяющих тип сегмента. Размер и список спецификаторов, определяющих тип сегмента, являются необязательными; если какой-то из этих атрибутов не задан, то для этого атрибута берется значение по умолчанию. У сегментов, которые сами являются битовыми строками (для них спецификатор типа будет либо **binary**, либо **bitstring**), размер по умолчанию – вся битовая строка. Поэтому в операции соответствия шаблону только последний сегмент в шаблоне может быть битовой строкой с размером по умолчанию (иначе непонятно, где должен закончиться сегмент, являющийся битовой строкой, и начаться следующий за ним сегмент).

> «Битовые строки позволяют работать с низкоуровневыми данными.»

Так, например, пусть **X** и **Y** – неинициализированные переменные. Тогда в результате выполнения выражения соответствия шаблону **<<X:1/binary, 2, Y/binary>> = <<1, 2, 3, 4>>**, переменная **X** будет иметь значение **<<1>>**, а переменная **Y** будет иметь значение **<<3, 4>>**.

Этот пример демонстрирует нам принцип последовательной обработки битовых строк: при помощи операции соответствия шаблону мы разделяем битовую строку на сегменты, причем последний сегмент имеет тип битовая строка и размер по умолчанию. Мы обрабатываем все сегменты, кроме последнего, после чего вызываем рекурсивно обработчик (при помощи хвостовой рекурсии), но уже для последнего сегмента; и так, пока не будет выбран нерекурсивный вариант для функции обработчика (например, вариант, обрабатывающий пустую битовую строку). Эта методика похожа на последовательную обработку списков (и там, и тут используются одни и те же принципы).

После краткого обзора теории пора перейти к практике. В качестве практики мы рассмотрим ряд задач, связанные с манипуляцией битами и байтами в битовых строках и целых числах. Способ решения, который мы будем использовать в наших примерах, отличается от способа решения таких задач в таких языках, как C/C++ и им подобные. Обычно такие задачи решаются с использованием побитовых операторов «И», «ИЛИ», «НЕ», «ИСКЛЮЧАЮЩЕЕ ИЛИ» и операторов сдвига. Такой вариант решения есть и у нас (язык Erlang содержит такой набор операторов). Но мы пойдем другим путем: будем решать эти задачи с использованием битовых строк и операций с ними. Но это не означает, что мы полностью отказываемся от использования побитовых операторов: там, где нам удобно их использовать, мы будем их использовать.

Начнем мы наш практикум с задачи определения минимального необходимого количества байт для хранения значения целого числа.. Определение минимального необходимого количества байт зависит от того, считаем ли мы исходное число числом со знаком или числом без знака.

Определить минимальное необходимое количество байт для целого числа без знака – достаточно тривиальная задача: мы берем исходное число, делим его на 256, после чего берем целую часть от деления и повторяем предыдущую операцию. Число операций, необходимых, чтобы получить 0 в качестве целой части результата деления, будет равно минимальному необходимому количеству байт для хранения положительного целого числа.

Теперь давайте разберемся с целыми числами со знаком. Начнем с отрицательных чисел. Отрицательные числа хранятся в дополнительном коде, для которого справедливо следующее утверждение: старший байт числа должен быть в диапазоне от **2#10000000 = 128** до **2#11111111 = 255** (или, что то же самое, старший бит числа должен быть равен 1). Поэтому первым шагом мы можем разделить отрицательное число на -129 и взять целую часть; после этого с результатом целой части от деления поступаем точно так же, как и с целым числом без знака. Число операций, необходимых, чтобы получить 0 в качестве целой части результата деления, будет равно минимальному необходимому количеству байт для хранения отрицательного целого числа.

Перейдем к положительным целым числам. Если значение старшего байта положительного целого числа лежит в диапазоне от **2#10000000 = 128** до **2#11111111 = 255**, то такое положительное целое число мы не сможем отличить от отрицательного целого числа. Что мы можем сделать в такой ситуации – это добавить дополнительный байт со значением 0 в качестве старшего байта. Поэтому первым шагом мы можем разделить положительное число на 128, и взять целую часть; после этого с результатом целой части от деления поступаем точно так же, как и с целым числом

Побитовые операторы

Оператор унарного побитового «НЕ»	**bnot**
Оператор бинарного побитового «И»	**band**
Оператор бинарного побитового «ИЛИ»	**bor**
Оператор бинарного побитового «ИСКЛЮЧАЮЩЕЕ ИЛИ»	**bxor**
Оператор бинарного битового сдвига влево	**bsl**
Оператор бинарного битового сдвига вправо	**bsr**

битовых строк

без знака. Число операций, необходимых, чтобы получить 0 в качестве целой части результата деления, будет равно минимальному необходимому количеству байт для хранения положительного целого числа.

Приведенный выше алгоритм для целых чисел без знака реализован в функции **integer_size/1**. Эта функция является интерфейсной; вся работа по подсчету содержится в функции **integer_size/2**, реализующей приведенный выше алгоритм для целых чисел без знака. Следует отметить, что функция **integer_size/1** содержит отдельный вариант для 0; это сделано для удобства реализации подсчета.

```
integer_size(0) -> 1;
integer_size(Number) when is_integer(Number) ->
  integer_size(Number, 0).
integer_size(0, ByteCount) -> ByteCount;
integer_size(Number, ByteCount) ->
  integer_size(Number div 256, ByteCount + 1).
```

Практически то же самое можно сказать и о функциях **integer_signed_size/1** и **integer_signed_size/2**: первая является интерфейсной, а вторая реализует алгоритм для целых чисел со знаком. И точно так же, функция **integer_signed_size/1** содержит отдельный вариант для 0, который введен для удобства реализации подсчета.

```
integer_signed_size(0) -> 1;
integer_signed_size(Number) when is_integer(Number) ->
  integer_signed_size(Number, 0).
integer_signed_size(Number, ByteCount) when Number < 0 ->
  integer_size(Number div -129, ByteCount + 1);
integer_signed_size(Number, ByteCount) when Number > 0 ->
  integer_size(Number div 128, ByteCount + 1).
```

Давайте проверим, что написанные нами функции работают правильно. Для целого числа **127 = 2#01111111** минимальное необходимое для хранения количество байт равно 1 и для случая, когда мы считаем его числом без знака, и для случая, когда мы считаем его числом со знаком. Проверяем это: вызов **bit_utils:integer_size(127)** возвращает 1, вызов **bit_utils:integer_signed_size(127)** также возвращает 1.

Возьмем более сложный пример. Для целого числа **128 = 2#10000000**, когда мы считаем его числом без знака, минимальное необходимое для хранения количество байт равно 1, а в случае, когда мы считаем его числом со знаком, минимальное необходимое для хранения количество байт равно уже 2. Проверяем это: вызов **bit_utils:integer_size(128)** возвращает 1, а вызов **bit_utils:integer_signed_size(128)** возвращает 2. Теперь проверим для отрицательных чисел. Для хранения отрицательного числа **-128 = 2#10000000** достаточно 1 байта, тогда как для хранения отрицательного числа **-129 = 2#1111111101111111** уже необходимо 2 байта. Проверяем это: вызов **bit_utils:integer_signed_size(-128)** возвращает 1, а вызов **bit_utils:integer_signed_size(-129)** возвращает 2.

После решения предыдущей задачи мы можем решить другую, связанную с ней задачу: создать битовую строку, содержащую значение исходного целого числа. Для решения этой задачи мы должны знать количество байт, необходимое для представления исходного целого числа. Если при создании сегмента (с типом **integer**), который должен содержать целое число, мы не зададим размер сегмента, то будет использован размер по умолчанию (для целых чисел это 8 бит), и сегмент будет содержать только младший байт целого числа. Так, например, битовая строка **<<256/integer>>** будет равна битовой строке **<<0>>**. Поэтому при построении битовой строки, содержащей значение целого числа, знание минимального необходимого для хранения этого целого числа размера является обязательным. Обязательным является также знание порядка хранения байт и является ли исходное число числом со знаком. Оба эти параметра независимые и передаются в качестве исходных параметров в нашу функцию. Следует сказать, что параметр, определяющий, является ли исходное число числом со знаком, влияет на то, какой алгоритм вычисления минимального необходимого количества байт для хранения целого числа будет использован.

Эта задача реализована в функции **integer_to_binary/3**. Функция достаточно тривиальна: в зависимости от исходных параметров, определяющих порядок байт, и является ли исходящее целое число числом без знака, мы вычисляем минимальное необходимое количество байт для хранения исходного целого числа и конструируем соответствующую битовую строку.

```
integer_to_binary(Number, Signedness, ByteOrder) when is_
integer(Number) ->
  case {Signedness, ByteOrder} of
    {signed, big} ->
      IntegerSize = 8 * integer_signed_size(Number),
      <<Number:IntegerSize/signed-integer-big>>;
    {signed, little} ->
      IntegerSize = 8 * integer_signed_size(Number),
      <<Number:IntegerSize/signed-integer-little>>;
    {unsigned, big} ->
      IntegerSize = 8 * integer_size(Number),
      <<Number:IntegerSize/unsigned-integer-big>>;
    {unsigned, little} ->
      IntegerSize = 8 * integer_size(Number),
      <<Number:IntegerSize/unsigned-integer-little>>
  end.
```

Давайте проверим, что наше решение правильно. Для числа 128, если считать это число числом без знака, мы должны получить следующую битовую строку: **<<128>>**. Вызов **bit_utils:integer_to_binary(128, unsigned, big)** возвращает нам **<<128>>**. Если же мы считаем число 128 числом со знаком, то мы должны получить битовую строку **<<0, 128>>**, при условии, что порядок байт у нас от старшего байта к младшему [big-endian]. Вызов **bit_utils:integer_to_binary(128, signed, big)** возвращает нам **<<0, 128>>**. Для числа -128 мы должны получить битовую строку **<<128>>**, а для числа -129 – битовую строку **<<255, 127>>** (при условии, что порядок байт у нас от старшего байта к младшему). Вызов **bit_utils:integer_to_binary(-128, signed, big)** возвращает нам **<<128>>**; вызов **bit_utils:integer_to_binary(-129, signed, big)** возвращает **<<255, 127>>**. »

Пойдем дальше: займемся теперь задачами, связанными непосредственно с битовыми строками. В качестве первой такой задачи мы возьмем следующую: изменение порядка следования байт в битовой строке с размером, кратным 8 бит. Решение этой задачи тривиальное: мы преобразуем битовую строку в список целых чисел (каждый элемент списка является представлением соответствующего 8-битного сегмента в виде целого числа), меняем порядок следования элементов в списке на обратный и преобразуем список целых обратно в битовую строку.

```
reverse_bytes(Binary) when is_binary(Binary) ->
  list_to_binary(lists:reverse(binary_to_list(Binary))).
```

Теперь мы можем проверить, что наша функция корректно работает: вызов **bit_utils:reverse_bytes(<<1, 2, 3>>)** дает вполне ожидаемый результат **<<3, 2, 1>>**.

Давайте усложним предыдущую задачу: будем менять порядок следования бит в битовой строке на обратный. Заметим, что следующие действия дают тот же результат, что и решение «в лоб»: разделяем битовую строку на 8-битовые сегменты (и возможный остаток, размер которого меньше 8 бит), меняем порядок следования сегментов на обратный и, для каждого сегмента, меняем порядок следования битов в нем на обратный. После чего собираем список сегментов обратно в битовую строку. Разделить битовую строку на 8-битовые сегменты и возможный остаток (размер которого будет меньше 8 бит) мы можем с помощью BIF **bitstring_to_list/1**.

```
reverse_bits(Bitstring) when is_bitstring(Bitstring) ->
  Segments = bitstring_to_list(Bitstring),
  ReversedSegments = lists:map(fun reverse_segment/1,
lists:reverse(Segments)),
  list_to_bitstring(ReversedSegments).
```

Для изменения порядка следования бит в каждом элементе списка сегментов мы используем функцию **reverse_segment/1** (эту функцию мы используем из **lists:map/2** для преобразования сегментов с нормальным порядком бит в сегменты с обратным порядком бит). Эта функция имеет вариант для целых чисел, т.к. обычные 8-битные сегменты в списке у нас представлены в виде целых чисел, и варианты для обработки битовых сегментов размером от 1 до 8 бит.

```
reverse_segment(Number) when is_integer(Number) ->
  reverse_segment(<<Number:8>>);
reverse_segment(<<B7:1, B6:1, B5:1, B4:1, B3:1, B2:1, B1:1, B0:1>>)
->
  <<B0:1, B1:1, B2:1, B3:1, B4:1, B5:1, B6:1, B7:1>>;
reverse_segment(<<B6:1, B5:1, B4:1, B3:1, B2:1, B1:1, B0:1>>) ->
  <<B0:1, B1:1, B2:1, B3:1, B4:1, B5:1, B6:1>>;
reverse_segment(<<B5:1, B4:1, B3:1, B2:1, B1:1, B0:1>>) ->
  <<B0:1, B1:1, B2:1, B3:1, B4:1, B5:1>>;
reverse_segment(<<B4:1, B3:1, B2:1, B1:1, B0:1>>) ->
  <<B0:1, B1:1, B2:1, B3:1, B4:1>>;
reverse_segment(<<B3:1, B2:1, B1:1, B0:1>>) ->
  <<B0:1, B1:1, B2:1, B3:1>>;
reverse_segment(<<B2:1, B1:1, B0:1>>) ->
  <<B0:1, B1:1, B2:1>>;
reverse_segment(<<B1:1, B0:1>>) ->
  <<B0:1, B1:1>>;
reverse_segment(<<B0:1>>) ->
  <<B0:1>>.
```

Теперь пришла пора проверить, что наша реализация алгоритма изменения порядка бит в битовом слове работает правильно. В качестве исходной битовой строки возьмем строку **<<1, 2, 3:4>>**. В двоичном представлении она имеет следующий вид: **<<2#00000001, 2#00000010, 2#0011>>**. После изменения порядка следования бит на обратный эта битовая строка в двоичном представлении примет вид **<<2#1100, 2#01000000, 2#10000000>>**. После перегруппировки сегментов согласно правилам представления битовых строк в языке Erlang (когда остаток, не кратный 8 бит, идет в конце) битовая строка в двоичном представлении будет иметь вид **<<2#11000100, 2#00001000, 2#0000>>**, а в десятичном представлении – **<<196, 8, 0:4>>**. Вызов **bit_utils:reverse_bits(<<1, 2, 3:4>>)** дает ожидаемый результат **<<196, 8, 0:4>>**. Это дает нам право считать, что наша реализация работает правильно.

Давайте перейдем к следующей задаче: посчитаем количество бит со значением 1 в битовой строке (читатель может обобщить эту задачу и реализовать функцию по подсчету количества бит с определенным значением в битовой строке). Само по себе решение этой задачи достаточно тривиально: необходимо рекурсивно (при помощи хвостовой рекурсии) обработать битовую строку, выделяя первый бит и остаток. Реализация состоит из двух функций: **setbit_count/1** и **setbit_count/2**. Первая является интерфейсной функцией, а вторая при помощи разных вариантов определения функции реализует решение данной задачи.

```
setbit_count(Bitstring) when is_bitstring(Bitstring) ->
  setbit_count(Bitstring, 0).
setbit_count(<<>>, Count) -> Count;
setbit_count(<<1:1, Rest/bitstring>>, Count) ->
  setbit_count(Rest, Count + 1);
setbit_count(<<0:1, Rest/bitstring>>, Count) ->
  setbit_count(Rest, Count).
```

Можно эту задачу решить другим способом: с помощью выражения конструирования битовой строки [Bitstring Comprehensions]. В этом выражении источником будет исходная битовая строка, из которой мы будем извлекать сегменты размером 1 бит, а фильтром будет выражение, пропускающее только те сегменты, которые содержат 1. Количество бит со значением 1 будет равно длине полученной битовой строки, которую мы можем подсчитать при помощи BIF **bit_size/1**. Выглядит это выражение следующим образом (здесь **Bitstring** – переменная, содержащая исходную битовую строку): **bit_size(<< <<Bit:1>> || <<Bit:1>> <= Bitstring, Bit == 1 >>)**.

Теперь можно проверить работу нашей функции. Битовая строка **<<2#110010101:9>>** содержит 5 бит со значением 1. Вызов **bit_utils:setbit_count(<<2#110010101:9>>)** дает нам число 5.

Давайте рассмотрим еще пару подобных задач: подсчитаем количество ведущих и завершающих битов со значением 0 в битовой строке. Эти задачи схожи с предыдущей решением: мы рекурсивно (при помощи хвостовой рекурсии) обрабатываем битовую строку, выделяя первый или последний (в зависимости от задачи) бит и остаток. Отличает эти задачи от предыдущей условие прекращения работы: если в предыдущей задаче мы полностью проходили по битовой строке, то в рассматриваемых в данный момент задачах мы прекращаем обработку, как только нам встретится бит со значением 1. Что интересно, данные две задачи решить с помощью выражения конструирования битовых строк [Bitstring Comprehensions] невозможно.

Задача по подсчету количества ведущих бит со значением 0 решается в следующих двух функциях: **head_unsetbit_count/1** и **head_unsetbit_count/2**. Первая функция является интерфейсной, вторая функция реализует алгоритм по подсчету количества ведущих бит со значением 0; условием окончания работы является факт появления бита со значением 1 на месте обрабатываемого, либо обработка всей битовой строки.

```
head_unsetbit_count(Bitstring) when is_bitstring(Bitstring) ->
  head_unsetbit_count(Bitstring, 0).
head_unsetbit_count(<<>>, Count) -> Count;
```

» **Пропустили номер?** Узнайте на с. 108, как получить его прямо сейчас.

```
head_unsetbit_count(<<0:1, Rest/bitstring>>, Count) ->
  head_unsetbit_count(Rest, Count + 1);
head_unsetbit_count(<<1:1, _Rest/bitstring>>, Count) -> Count.
```

Задача по подсчету количества завершающих бит со значением 0 решается в следующих двух функциях: **tail_unsetbit_count/1** и **tail_unsetbit_count/2**. Как и во всех подобных задачах, первая функция является интерфейсной, а вторая реализует алгоритм задачи. Отличие этой задачи от предыдущей заключается в том, с какого конца битовой строки берутся биты на обработку. В предыдущей задаче мы брали биты с начала битовой строки, а в текущей задаче – с конца битовой строки. Когда мы рекурсивно обрабатываем битовую строку с начала, разделяя ее на сегмент и остаток, то в этом случае мы имеем достаточно простое выражение соответствия шаблону, т. к. мы можем не задавать размер остатка. Когда мы рекурсивно обрабатываем битовую строку с конца, разделяя ее на сегмент и остаток, то в этом случае выражение соответствия шаблону более сложное, т. к. мы обязаны вычислить и задать размер остатка явно. Разницу в реализации обработки битовой строки с начала и с конца можно увидеть в реализации предыдущей и текущей задач (в качестве совета: когда есть такая возможность, всегда реализуйте обработку битовой строки с ее начала).

```
tail_unsetbit_count(Bitstring) when is_bitstring(Bitstring) ->
  tail_unsetbit_count(Bitstring, 0).
tail_unsetbit_count(<<>>, Count) -> Count;
tail_unsetbit_count(Bitstring, Count) ->
  RestSize = bit_size(Bitstring) - 1,
  <<Rest:RestSize/bitstring, TailBit:1>> = Bitstring,
  case TailBit of
    0 -> tail_unsetbit_count(Rest, Count + 1);
    1 -> Count
  end.
```

Как обычно, после реализации задачи проверим правильность этой реализации. В битовой строке **<<2#000110000:9>>** – 3 ведущих бита со значением 0 и 4 завершающих бита со значением 0. Пусть переменная **Bitstring** содержит значение **<<2#000110000:9>>**; тогда вызов **bit_utils:head_unsetbit_count(Bitstring)** возвращает 3, а вызов **bit_utils:tail_unsetbit_count(Bitstring)** возвращает 4.

В качестве последней задачи на сегодня, рассмотрим задачу инвертирования содержимого битовой строки. Эту задачу можно решить двумя разными способами: с использованием хвостовой рекурсии и с использованием выражений конструирования битовых строк [Bitstring Comprehensions].

Давайте решим эту задачу с использованием выражений конструирования битовых строк; очевидно, что выражение получится тривиальное, т. к. у нас только один источник, из которого мы будем доставать данные сегментами размером в 1 бит, и отсутствуют какие-либо фильтры.

Единственный момент, который может вызвать вопрос – это как получить инвертированное значение бита. В языке Erlang есть оператор **bnot**, который является побитовым оператором «НЕ». Это означает, что **bnot 0** равен **-1**, а **bnot 1** равен **-2**. Конечно, можно использовать и этот оператор с учетом обрезания значения до одного бита, когда мы будем создавать сегмент размером 1 бит. Но мы воспользуемся другим битовым оператором, который без всякого обрезания позволяет нам инвертировать значения битов: это побитовый оператор «ИСКЛЮЧАЮЩЕГО ИЛИ» или **bxor**.

```
inverse(Bitstring) when is_bitstring(Bitstring) ->
  << <<(Bit bxor 1):1>> || <<Bit:1>> <= Bitstring >>.
```

Осталось только проверить правильность реализации нашего алгоритма: для битовой строки **<<2#01101001>>** инвертированное значение будет **<<2#10010110>>** или **<<150>>**. Вызов **bit_utils:inverse(<<2#01101001>>)** возвращает **<<150>>**.

Чтобы все приведенные здесь примеры работали, необходимо организовать их в модуль и создать список экспортируемых функций. Приведенные ниже строки делают это (и не забываем, что имя файла должно быть имя модуля с расширением "**.erl**"):

```
-module(bit_utils).
-export([reverse_bytes/1, reverse_bits/1]).
-export([integer_size/1, integer_signed_size/1,
integer_to_binary/3]).
-export([setbit_count/1, head_unsetbit_count/1, tail_unsetbit_
count/1, inverse/1]).
```

Сегодня мы использовали «черную магию» битовых строк для решения небольших задач, связанных с работой с целыми числами и битовыми строками. Овладение этой «магией» позволяет решать многие такие задачи быстро, красиво и эффективно. Но мы не заканчиваем практикум по битовым строкам. В следующей статье мы рассмотрим использование битовых строк на гораздо большем примере. LXF

Определение сегментов в битовых строках

Битовая строка состоит из сегментов: **<<E1, ..., En>>** (возможна ситуация, когда в битовой строке нет ни одного сегмента; такая битовая строка называется пустой – **<<>>**) Каждый сегмент **Ei** состоит из значения либо переменной (инициализированной или неинициализированной), необязательного размера и необязательного списка спецификаторов формата. Таким образом, сегмент битовой строки **Ei** может иметь один из следующих видов:

- **Ei = Value**
- **Ei = Value:Size**
- **Ei = Value/TypeSpecifierList**
- **Ei = Value:Size/TypeSpecifierList**

Здесь **Value** – значение либо переменная (как инициализированная, так и неинициализированная), **Size** – размер сегмента в некоторых единицах (см. далее), **TypeSpecifierList** – список спецификаторов типа сегмента.

Значение **Value** должно быть одного из следующих типов: целое число, действительное число, битовая строка. Для размера сегмента **Size** значение по умолчанию для сегментов с типом **integer** – 8, для сегментов с типом **float** – 64, для сегментов с типом **binary** и **bitstring** – вся битовая строка. Размер сегмента **Size** не задается для сегментов с одним из следующих типов: **utf8**, **utf16**, **utf32**. Список спецификаторов типа сегмента **TypeSpecifierList** – это список из следующих значений, разделенных дефисом "-", которые могут идти в любом порядке:

- Тип сегмента, одно из следующих значений: **integer**, **float**, **binary**, **bytes**, **bitstring**, **bits**, **utf8**, **utf16**, **utf32**. Значение по умолчанию для типа сегмента – **integer**. Типы **binary** и **bytes** эквивалентны; аналогично, типы **bitstring** и **bits** эквивалентны.
- Наличие знака у значения **Value**, одно из следующих значений: **signed**, **unsigned**. Применим только для сегментов с типом **integer**. Значение по умолчанию – **unsigned**.
- Порядок байт, одно из следующих значений: **big**, **little**, **native**. Применим только для сегментов с одним из следующих типов: **integer**, **utf16**, **utf32**, **float**. Значение по умолчанию – **big**.
- Размер единицы, в которых задается размер сегмента **Size**. Имеет следующий вид **unit:IntegerLiteral**, где значение **IntegerLiteral** должно быть в диапазоне от 1 до 256. Неприменим для сегментов одного из следующих типов: **utf8**, **utf16**, **utf32**. Значение по умолчанию – 1 для сегментов с одним из следующих типов: **integer**, **float**, **bitstring**; значение по умолчанию – 8 для сегментов с типом **binary**. Реальное значение размера сегмента (в битах) равняется **Size * unit**. Для сегментов типа **binary** (**bytes**) реальный размер должен быть кратен 8.

В выражении соответствия шаблону [pattern matching] только крайний справа сегмент может иметь тип **bitstring** или **binary** и значение размера сегмента по умолчанию. Для всех остальных сегментов с типом **binary** или **bitstring** размер сегмента должен быть задан явно.

Так, например, следующее выражение: **<< X:1/binary, Y/binary >> = <<1, 2, 3>>** является корректным, а выражение **<< X/binary, Y/binary >> = <<1, 2, 3>>** корректным не является.