Erlang: Битовые

Базовые сущности продолжаются: **Андрей Ушаков** переходит к битовым строкам. Они помогают телекоммуникациям!

Наш эксперт

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

В данной статье мы рассмотрим очередную базовую сущность языка Erlang – битовые строки. Отличие битовых строк от других базовых сущностей, таких как кортежи, списки и функции, в том, что те пришли в язык Erlang из его функциональной сущности, тогда как битовые строки пришли из области применения – телекоммуникаций. И действительно, как будет показано ниже, битовые строки упрощают нам жизнь при работе с двоичными данными, которые сплошь и рядом встречаются в телекоммуникациях, будь то двоичные протоколы или файлы двоичных форматов. Конечно, такую большую тему за один урок не охватить, и наша статья – всего лишь введение в работу с двоичными данными; мы еще не раз вернемся к этой теме в статьях про язык Erlang.

Первым делом, давайте разберемся с тем, для чего нужны битовые строки. Битовые строки – это средство для работы с низкоуровневыми данными более структурированным, чем набор (массив) байт, способом. Почему мы говорим про низкоуровневые данные? Потому что для представления высокоуровневых данных нам не важно, сколько байт (или бит) занимает то или иное поле структуры данных или в каком порядке располагаются байты. Мы вообще не заботимся о байтах и битах – вместо этого мы работаем с типами данных языка: атомами, функциями, целыми числами, списками, кортежами и т.д. Всеми вещами, связанными с представлением в памяти, порядком следования байт и т.д., занимается среда выполнения языка Erlang. И нам нет необходимости заниматься всеми этими вопросами, пока мы не выходим за пределы языка. Однако, как только у нас возникает необходимость взаимодействовать с внешним миром, перед нами встают все вопросы, связанные с низкоуровневым представлением данных (даже при чтении пользовательского ввода с консоли). Если же нам требуется реализовать взаимодействие по сети в соответствии с некоторым двоичным протоколом или прочитать/записать файл, содержащий данные в некотором двоичном формате, то помимо работы с двоичными данными как с набором байт необходимо что-то еще (иначе наша жизнь может превратиться в ад). В нашем случае это не проблема: язык Erlang создавался как язык для написания приложений для телекоммуникаций – области, в которой существует большое количество двоичных протоколов взаимодействия по сети. Для эффективной реализации этих протоколов нужно нечто большее, чем работа с набором байт. И в языке Erlang такое средство есть – это битовые строки!

Давайте рассмотрим определение выражений, являющихся битовыми строками. Выражение для битовой строки определяется следующим образом: **<<E1, ..., En>>** (**<<>>** определяет пустую битовую строку). В этом определении **E1, ..., En** – это последовательно идущие сегменты битовой строки. Каждый сегмент – это значение вместе с необязательным набором спецификаторов (в набор спецификаторов могут входить спецификатор длины и спецификатор типа). Другими словами, каждый сегмент может выглядеть одним из следующих образов: **Value**, **Value:Size**, **Value/TypeSpecifierList**, **Value:Size/TypeSpecifierList**, где **Value** – это значение, **Size** – спецификатор размера, **TypeSpecifierList** – спецификатор типа. Значением **Value** может быть любое выражение, тип которого – целое число, действительное число или битовая строка. Значение **TypeSpecifierList** – это список спецификаторов типа (идущих в любом порядке), разделенный дефисом "-". Список спецификаторов типа может включать следующие значения:

» Одно из следующих значений типа: **integer**, **float**, **binary**, **bytes**, **bitstring**, **bits**, **utf8**, **utf16**, **utf32**. Значение типа по умолчанию – **integer**. Типы **bytes** и **binary** являются одним и тем же типом; то же справедливо и для типов **bits** и **bitstring**. Типы **utf8**, **utf16**, **utf32** определяют формат кодирования символов Юникод (об этом мы поговорим подробнее на одном из следующих уроков).

» Для типа **integer** можно указать, что значение выражения идет со знаком (**signed**) или без знака (**unsigned**). Значение по умолчанию – **unsigned**.

» Для типов **integer**, **float**, **utf16**, **utf32** можно указать порядок следования байт: определяемый процессором в момент выполнения (**native**), от старшего байта к младшему или big-endian (**big**), от младшего байта к старшему или little-endian (**little**). Значение по умолчанию – **big**.

» Размер «единицы», задаваемый в следующем виде: **unit:UnitSize**, где **UnitSize** – число из диапазона 1...256. По умолчанию, размер «единицы» равен 1 для типов **integer**, **float** и **bitstring**, и равен 8 для типа **binary**. Для типов **utf8**, **utf16** и **utf32** размер «единицы» не задается.

И, наконец, значение **Size** задает размер сегмента в «единицах» (т.е. истинный размер сегмента в битах будет равен произведению значения **Size** на величину размера «единицы»). Для сегмента типа **binary** истинный размер должен быть кратен 8. Зна-

строки

чение **Size** по умолчанию зависит от типа сегмента: для типа **integer** значение по умолчанию 8, для типа **float** – 64, для **binary** (**bytes**) и **bitstring** (**bits**) – это размер данного **binary** или **bitstring**. Ну и, наконец, следует сказать о том, что для типов **utf8**, **utf16** и **utf32** значение **Size** не задается (что не удивительно).

Что же в результате такого объявления получается? А получается набор бит, сгруппированный побайтно (с возможным наличием некратного размеру байта сегментом на конце, если количество бит в битовой строке не кратно 8). Рассмотрим несколько примеров. Объявление **<<1:8>>** дает нам битовую строку, состоящую из одного байта – **<<1>>**. Объявление **<<1:1, 1:1>>** упаковывает два сегмента в один и дает нам битовую строку, состоящую из одного битового сегмента – **<<3:2>>**. В следующем объявлении **<<9:2>>** размер сегмента слишком мал, чтобы хранить значение целого числа целиком, поэтому происходит усечение наиболее значимых битов числа, и в результате мы имеем следующую битовую строку: **<<1:2>>**. Число 9 в двоичном представлении имеет вид 2#101; усекаем до 2 битов и в результате получаем число 2#01, которое в десятичном представлении имеет вид 1. Объявление **<<1:5, 1:5, 1:6>>** упаковывает три сегмента в два байта и дает нам битовую строку вида **<<16, 65>>**. Объявление **<<65, 66>>** дает битовую строку, состоящую из двух байт и имеющую следующий вид: **<<"AB">>**.

> «Некоторые комбинации дают ошибку времени выполнения.»

Как и в случае со списками (о них см. в **LXF147**), язык Erlang поддерживает «синтаксический сахар» автоматического преобразования набора байт в строку в кодировке Latin-1 (ISO-8859-1) и обратно. Давайте пойдем дальше. Объявление **<<1000:16/big>>** дает нам битовую строку с порядком следования байт от старшего к младшему: **<<3, 232>>**, а объявление **<<1000:16/little>>** дает битовую строку с порядком следования байт от младшего к старшему: **<<232, 3>>**. Объявление **<<22:2/integer-unit:2>>** даст такой же результат, что и объявление **<<22:4>>** – **<<6:4>>**. Объявление **<<1.0>>** даст ошибку времени выполнения; все потому, что тип сегмента по умолчанию – **integer**, а тип выражения **1.0** – **float**. Чтобы все работало, данное определение следует записать так: **<<1.0/float>>**; его результатом будет битовая строка из 8 байт – **<<63, 240, 0, 0, 0, 0, 0, 0>>**. И наконец, объявление **<< <<1:4>>/bits, <<1:4>>/bits >>** упаковывает два 4-битовых сегмента в один, состоящий из одного байта: **<<17>>**.

Следует упомянуть о следующем: не все возможные комбинации (возможные с точки зрения синтаксиса) спецификатора размера **Size** и спецификаторов типов **TypeSpecifierList** работоспособны: некоторые комбинации дают ошибку времени выполнения (в данной версии среды выполнения Erlang). Так, документация не запрещает для сегмента типа **float** задать размер, отличный от размера по умолчанию (от 64 бит); например, следующее определение не запрещено: **<<1.0:63/float>>**, но на практике это приведет к ошибке времени выполнения. Что можно сказать в качестве совета: если вы сомневаетесь, будет ли работать ваше определение битовой строки или нет, то просто проверьте ваше определение в консоли среды выполнения Erlang.

Выражения для битовых строк мы определять научились; следующий шаг – разобраться с операцией соответствия шаблону [pattern matching]. Шаблон для этой операции имеет вид **<<E1, …, En>>**, где **E1, …, En** – шаблоны сегментов битовой строки. Шаблоны сегментов могут принимать, как и в случае с выражениями, виды **Value**, **Value:Size**, **Value/TypeSpecifierList**, **Value:Size/TypeSpecifierList**. Чем же отличаются шаблоны сегментов от выражений? Во-первых, **Value** может быть либо переменной (инициализированной или неинициализированной), либо выражением одного из следующих типов: целое число, действительное число, строка. Во-вторых, **Size** может быть либо целочисленным выражением, либо переменной, инициализированной таким выражением. И в-третьих, значение по умолчанию для **Size** может использоваться только для последнего сегмента в шаблоне; для всех остальных сегментов значение **Size** должно быть задано явно.

Давайте разберем, как работает операция соответствия шаблону для битовых строк. Что нужно для того, чтобы операция соответствия шаблону для битовой строки выполнилась успешно? Нужно выполнение двух условий. Во-первых, размер шаблона должен соответствовать размеру выражения. Почему мы говорим, что размеры соответствуют, а не совпадают? А потому, что для последнего сегмента может быть использован размер по умолчанию; если его тип будет **binary** (**bytes**) или **bitstring** (**bits**), для которого размер по умолчанию равен размеру этого **binary** или **bitstring**, то последний сегмент будет соответствовать остатку выражения (после учета предыдущих сегментов шаблона). Это означает, что размер выражения должен быть не меньше размера шаблона (размеры выражения и шаблона должны быть равны, если тип последнего сегмента шаблона не **binary** (**bytes**) или

Полезные заметки

Нотации для целых чисел

В языке **Erlang** существуют две специфические нотации для записи целых чисел. Во-первых, выражение **$char** возвращает число, равное ASCII-коду символа **char**. Так, например, выражение **$A** равно **65**. Во-вторых, выражение **base#value** равно целому числу, значение которого по основанию **base** равно величине **value**. Для основания **base** допустимы любые значения из диапазона 2…36. Так, например, выражение **2#101** равно 5, выражение **16#EF** равно 239, а значение **36#XX** равно 1221.

Битовые операторы

В языке **Erlang** определены следующие битовые операторы над целыми числами:

- **bnot** – унарная побитовая операция НЕ;
- **band** – побитовая операция И;
- **bor** – побитовая операция ИЛИ;
- **bxor** – побитовая операция ИСКЛЮЧАЮЩЕЕ ИЛИ;
- **bsl** – побитовый сдвиг влево;
- **bsr** – побитовый сдвиг вправо.

Синтаксическая ошибка в операции соответствия шаблону для битовых строк

Компилятором **Erlang** объявление **Var=<<1>>** интерпретируется как **Var =<<1>>**, что является ошибкой синтаксиса. Для преодоления этой проблемы достаточно поставить пробел после знака равно "=": **Var= <<1>>**.

» **Пропустили номер?** Узнайте на с. 100, как получить его прямо сейчас.

bitstring (**bits**)). Во-вторых, для каждого сегмента шаблона и соответствующего ему сегмента выражения должна успешно выполняться операция соответствия шаблону.

Как мы выбираем для шаблона сегмента соответствующий ему сегмент выражения? Все очень просто: начинаем мы с первого сегмента шаблона, вычисляем его размер в битах и выбираем сегмент такого же размера из выражения; точно так же для второго сегмента шаблона вычисляем его размер в битах и выбираем сегмент такого же размера из остатка выражения (после первой выборки), и т.д. Ну и не забываем, что последний сегмент шаблона может соответствовать остатку выражения (если тип сегмента шаблона **binary** (**bytes**) или **bitstring** (**bits**)).

Давайте посмотрим на практике, как работает операция соответствия шаблону для битовых строк. Начнем с простого примера: **<<X>> = <<1:4, 1:4>>**. В данном примере операция соответствия шаблону проходит успешно, т.к. размер по умолчанию для сегмента шаблона – 8 (т.к. тип сегмента шаблона по умолчанию – **integer**), а размер выражения – также 8. Переменная **X** в результате операции соответствия шаблону получает значение 17. В следующих двух примерах – **<<X>> = <<1:8, 1:8>>** и **<<X>> = <<1:1, 1:1>>** — операция соответствия шаблону выполняется с ошибкой, т.к. размер сегмента шаблона по умолчанию – 8 (т.к. тип по умолчанию – **integer**), а размер выражения для битовой строки в первом примере – 16, а во втором примере – 3. Чтобы в этих примерах операция соответствия шаблону проходила успешно, их нужно переписать следующим образом: **<<X:16>> = <<1:8, 1:8>>** и **<<X:2>> = <<1:1, 1:1>>**. Переменная **X** в первом случае получит значение **<<1, 1>>**, а во втором случае – **<<3:2>>**. В следующем примере операция соответствия шаблону пройдет успешно: **<<X:7, Y/bits>> = <<1, 2, 3>>**, при этом переменная **X** будет содержать первые 7 бит исходного выражения (значение 0), а переменная **Y** – остаток исходного выражения (значение **<<129, 1, 1:1>>**), т.к. переменная **Y** находится в последнем сегменте шаблона и его тип – **bits**. В следующем примере операция соответствия шаблону выполнится с ошибкой: **<<X:3, 1:1, Y:4>> = 43**, т.к. 5-й бит числа 43 (43 = 2#00101011) равен 0, при этом в шаблоне сегмент, соответствующий 5-му биту, равен 1. Как уже говорилось выше, невозможно при помощи объявления выражения для битовой строки получить младшие *N* бит сегмента типа **float** (старшие *N* бит можно получить при объявлении, например, так: **<< <<1/float>>:8/bits >>**). При помощи операции соответствия шаблону это можно; в следующих двух примерах мы получаем 8 первых и 8 последних бит битовой строки, содержащего сегмент типа **float**: **<<Lead:8, _/bits>> = <<1/float>>** и **<<_:56, Tail:8>> = <<1/float>>**.

> «Битовые строки по своей природе очень похожи на списки.»

В предыдущей статье (**LXF147**) было показано, что для создания списков существует мощная и гибкая техника – выражения конструирования списков (или List Comprehensions). Битовые строки по своей природе очень похожи на списки, поэтому подобная техника определена и для них – это конструирование битовых строк (или Bit String Comprehensions). Выражение конструирования для битовых строк очень похоже на выражение конструирования для списков и имеет следующий вид: **<<BitString || Qualifier1, …, QualifierN>>**. Здесь **BitString** – это выражение, формирующее результирующую битовую строку, **Qualifier1, …, QualifierN** – это либо выражение генератора, либо выражение фильтра.

Выражение генератора – это выражение, связывающее выражение, результатом которого является либо список, либо битовая строка, с шаблоном для получения доступа к элементам генерируемой сущности. Элементами генерируемой сущности являются либо элементы списка, либо сегменты битовой строки, доступ к которым осуществляется через операцию соответствия шаблону. Для этого в шаблоне объявляются одна или несколько уникальных в пределах данного выражения конструирования переменных, через которые и осуществляется доступ к элементам генерируемой сущности. Выражение для генераторов списков имеет следующий вид: **Pattern <- ListExpr**, где **ListExpr** – выражение, результатом которого является список элементов, **Pattern** – шаблон для извлечения элементов (через операцию соответствия шаблону). Выражение для генераторов битовых строк имеет похожий вид: **Pattern <= BitStringExpr**, где **BitStringExpr** – выражение, результатом которого является битовая строка, **Pattern** – шаблон для извлечения сегментов битовой строки (через операцию соответствия шаблону). Элементы генерируемой сущности, которые не соответствуют шаблону, отбрасываются.

Выражение-фильтр – это выражение, возвращающее **true** или **false**. В выражении-фильтре можно использовать любую переменную из генераторов, уже определенных на момент определения фильтра (т.е. находящихся левее выражения фильтра). Выражения-фильтры служат для фильтрации значений, получаемых от генераторов: те значения или комбинации значений, которые не проходят фильтр (для которых выражение-фильтр возвращает **false**), отбрасываются. Итоговое выражение **BitString** формирует сегменты конструируемой битовой строки; в это итоговое выражение могут входить все уникальные (в пределах данного выражения конструирования) переменные, объявленные в шаблонах генераторов.

Как все это работает? Вычисляются все возможные комбинации элементов генераторов (все возможные комбинации, доступные через переменные, объявленные в шаблонах), после чего к этим комбинациям применяются фильтры: комбинации, для которых все фильтры вернули **true**, попадают в итоговое выражение **BitString** и формируют сегменты конструируемой битовой строки. На самом деле все гораздо проще, чем кажется после прочтения; давайте рассмотрим несколько примеров, чтобы эта концепция стала понятнее.

О примерах в статье

Сделаем следующие замечания по поводу примеров в статье: предположим, мы запускаем примеры в консоли. Пусть у нас есть два примера на соответствие шаблону для битовых строк: **<<X:8>> = <<1:8>>** и **<<_:4, X:4>> = <<2:8>>**. Если их запустить в одном экземпляре консоли Erlang, то второй пример выполнится с ошибкой, т.к. после выполнения первого примера переменная **X** уже инициализирована и имеет значение, не совпадающее со значением, которое ей пытаются сопоставить во втором примере. Чтобы примеры выполнялись, нужно либо в каждом примере менять имя переменной на ранее неиспользуемое, либо запускать каждый пример в новом экземпляре консоли Erlang.

В тексте статьи во всех примерах используется одно и то же имя, с целью единообразия. Пусть у нас есть два следующих примера на конструирование битовых строк: **<< <<X>> || <<X>> <= <<1>> >>** и **<< <<X>> || <<X>> <= <<1, 2>> >>**. Если их запустить в одном экземпляре консоли Erlang, то оба примера выполнятся успешно, поскольку область видимости переменной **X** – это операция конструирования. Поэтому можно написать выражение **X = << <<X>> || <<X>> <= <<1>> >>**, и оно будет работать. И не забываем, что в консоли, чтобы выполнить введенное выражение, необходимо поставить после выражения точку "." и нажать на клавишу Enter.

Начнем с простого примера без фильтров: **<< <<X>> || <<X:4>> <= <<1, 2, 1:1>> >>**. Здесь выражение генератора извлекает 4-битовые сегменты из исходного выражения для битовой строки, эти сегменты расширяются до 8-битовых и входят в итоговую битовую строку **<<0, 1, 0, 2>>**. Следует заметить, что в исходном выражении для битовой строки количество бит не кратно 4; поэтому оставшийся сегмент не проходит операцию соответствия шаблону (которая требует, чтобы сегмент был 4-битовый) и отбрасывается (т.е. не попадает в итоговую битовую строку). В следующем примере: **<< <<(X+Y)>> || <<X>> <= <<1, 2, 3>>, <<Y>> <= <<4, 5, 6>>, Y+X < 7 >>** выражения генераторы извлекают 8-битовые сегменты из исходных выражений для битовых строк, а в итоговую битовую строку попадают только такие сегменты, для которых их сумма меньше 7 (в результате мы получим следующую битовую строку: **<<5, 6, 6>>**). В качестве последнего примера, мы модифицируем предыдущий пример – пусть первый генератор будет генератором списка: **<< <<(X+Y)>> || X <- [1, 2, 3], <<Y>> <= <<4, 5, 6>>, Y+X < 7 >>**. Результат выражения от этого не изменится и будет равен **<<5, 6, 6>>**, как и в предыдущем примере.

Давайте теперь вернемся к выражениям конструирования списков (которые мы обсуждали в предыдущем номере, LXF147). Как уже говорилось, и в выражениях конструирования списков в качестве генераторов мы можем использовать генераторы битовых строк. В следующем примере мы возвращаем из генератора битовой строки только младшие 2 бита 8-битовых сегментов исходной битовой строки; а результирующий список состоит из целых чисел, до которых расширяются соответствующие 2-битовые сегменты: **[X || <<_:6, X:2>> <= <<124, 125, 127, 63>>]**. Результатом этого выражения будет следующий список: **[0, 1, 3, 3]**.

Теперь пришла пора поговорить о функциях для работы с битовыми строками. Часть функций определена в модуле **erlang** и является BIF [built-in function, встроенной функцией]. В первую очередь следует упомянуть BIF для преобразования типов языка Erlang в битовые строки и обратно. Самые простые и полезные из этих BIF – это **term_to_binary/1** и **binary_to_term/1**, позволяющие сериализовать объект языка Erlang в битовую строку и десериализовать из битовой строки обратно в объект языка Erlang. Это стандартное средство сериализации и десериализации в языке Erlang.

Сериализованный методом **term_to_binary/1** объект содержит помимо данных еще и метаданные; в этом можно убедиться, сделав, например, вызов **term_to_binary(11)** и получив в результате битовую строку **<<131, 97, 11>>**. Этот формат битовых строк называется внешними битовыми строками. У BIF **term_to_binary/1** и **binary_to_term/1** есть перегруженные версии (**term_to_binary/2** и **binary_to_term/2**), принимающие в качестве второго параметра список опций сериализации или десериализации. Так, при сериализации (при вызове метода **term_to_binary/2**) можно задать, использовать ли сжатие (опция **compressed**), уровень сжатия (опция **{compressed, Level}**), минимальный номер версии для сериализации (опция **{minor_version, VersionNumber}**). Например, следующий вызов при сериализации действительного числа задает максимальный уровень сжатия и задает минимальную версию для сериализации: **term_to_binary(1.0, [{compressed, 9}, {minor_version, 1}])**. Т.к. все эти опции записываются в метаданные, то при десериализации их задавать не нужно. Поэтому при десериализации (при вызове метода **binary_to_term/2**) можно задать лишь одну опцию: приходят ли данные из доверяемого источника (опция safe). Если данные приходят не из доверяемого источника (например, через сокет), то использование этой опции позволяет избежать атак на среду выполнения Erlang.

Помимо методов для сериализации и десериализации во внешние битовые строки, в модуле **erlang** еще есть BIF для преобразования объектов в обычные битовые строки и обратно: **binary_to_atom/2**, **binary_to_existing_atom/2**, **binary_to_list/3**, **atom_to_binary/2**, **list_to_binary/1** и другие. Чтобы узнать размер битовой строки, у нас есть три BIF: **bit_size/1**, **byte_size/1** и **size/1**. Метод **bit_size/1** возвращает количество бит в битовой строке, метод **byte_size/1** возвращает количество байт, необходимых для хранения битовой строки, метод **size/1** возвращает количество целых байт в битовой строке. Так, например, **size(<<1:7, 1:7>>)** вернет 1, **bit_size(<<1:7, 1:7>>)** вернет 14, **byte_size(<<1:7, 1:7>>)** вернет 2. Реализация более сложных алгоритмов работы с битовыми строками содержится в следующих BIF: **binary_part/2**, **binary_part/3**, **decode_packet/3**, **split_binary/2** и во всех методах из модуля **binary**. Но об этих функциях и BIF мы поговорим в одной из следующих статей о языке Erlang.

Давайте перейдем от теории к практике и рассмотрим небольшой пример. В качестве примера, предположим, что у нас есть байт данных, в который упаковано несколько полей данных (часто встречающаяся ситуация при работе с двоичными форматами и протоколами). Для определенности, пусть биты 0, 4 и 7 будут флагами **F0**, **F4** и **F7** соответственно, биты 1–3 будут значением **V13**, биты 5 и 6 будут значением **V56**. Наша задача состоит в том, чтобы создать байт данных, который будет содержать все вышеперечисленные поля. При помощи битовых строк эта задача решается элементарно:

```
<<Packed>> = <<F7:1, V56:2, F4:1, V13:3, F0:1>> .
```

(предполагаем, что **F0**, **F4**, **F7**, **V13**, **V56** – это инициализированные переменные, содержащие необходимые значения). Если переменные содержат значения **F0 = 1**, **V13 = 5**, **F4 = 0**, **V56 =2**, **F7 = 1**, то упакованное значение (в переменной **Packed**) будет равно 203. Каждый может посчитать вручную и убедиться, что мы правильно упаковали поля в байт.

Теперь рассмотрим обратную задачу: необходимо из байта данных извлечь значения индивидуальных полей. При помощи битовых строк и эта задача решается элементарно:

```
<<F7:1, V56:2, F4:1, V13:3, F0:1>> = <<Packed>>.
```

(предполагаем, что **Packed** – инициализированная переменная, содержащая упакованное значение). Если упакованное значение (в переменной **Packed**) равно 183, то переменные для полей содержат следующие значения **F0 = 1**, **V13 = 3**, **F4 = 1**, **V56 = 1**, **F7 = 1**.

Если не использовать битовые строки, то упаковка значений полей в один байт будет выглядеть следующим образом: **Packed = F0 bor (V13 bsl 1) bor (F4 bsl 4) bor (V56 bsl 5) bor (F7 bsl 7)**, а извлечение значений отдельных полей из упакованного байта – следующим образом: **F0 = Packed band 2#1**, **V13 = (Packed bsr 1) band 2#111**, **F4 = (Packed bsr 4) band 2#1**, **V56 = (Packed bsr 5) band 2#11**, **F7 = (Packed bsr 7) band 2#1**. Гораздо менее удобно, чем использование битовых строк, не правда ли?

Итак, мы рассмотрели базовые принципы работы с двоичными данными с использованием битовых строк. Мы еще не раз вернемся к этой теме, т.к. как только возникает вопрос о взаимодействии с внешним миром, так сразу нам приходится работать с двоичными данными (будь то строки, протоколы взаимодействия или файлы). А следующая статья будет посвящена работе со строковыми данными в языке Erlang. LXF

Интересный факт

Термины big-endian и little-endian (характеризующие порядок следования байт в памяти) первоначально не имели отношения к информатике. В сатирическом произведении Джонатана Свифта «Путешествия Гулливера» описываются вымышленные государства Лилипутия и Блефуску, долгие годы ведущие между собой войну из-за разногласия по поводу того, с какого конца следует разбивать вареные яйца. Тех, кто считает, что их нужно разбивать с тупого конца, в произведении называют “Big-endians” («тупоконечники»); тех, кто считает, что их нужно разбивать с острого конца, называют “Little-endians” («остроконечники»).