Erlang: Магия

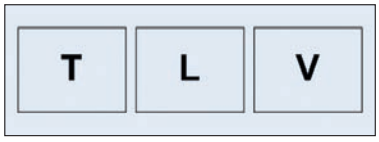
Пришла пора опробовать «черную магию» битовых строк на большом примере, воодушевился **Андрей Ушаков**.

Наш эксперт

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

В этом номере мы продолжаем наш практикум по функциональному программированию: пришла пора опробовать «черную магию» битовых строк на большом примере. В качестве такого большого примера мы реализуем ASN.1-совместимую сериализацию и десериализацию объектов языка **Erlang**.

Что же такое **ASN.1**? Это набор стандартов для описания абстрактного синтаксиса данных в области телекоммуникаций и компьютерных сетей. Стандарты **ASN.1** описывают структуры данных для представления, кодирования, передачи и декодирования данных. Они слишком многочисленны, чтобы рассматривать их полностью; мы кратко остановимся на той их части, что касается кодирования и декодирования данных. Для нашей задачи мы применим правила кодирования и декодирования **ASN.1 BER** [basic encoding rules]. В соответствии с ними, закодированное значение любого элемента данных состоит из 3-х частей: описателя типа данных (тэга), длины закодированного значения элемента данных и собственно закодированного значения элемента данных. Описатель типа данных (тэг) содержит идентификатор типа данных, класс описателя (одно из следующих значений: универсальный тип данных, специфичный для приложения, специфичный для контекста, приватный тип данных) и форму данных (одно из следующих значений: простые данные, составные данные). Все части состоят из целого числа октетов (в стандарте **ASN.1** применяется термин не байты, а октеты).



› Рис. 1.

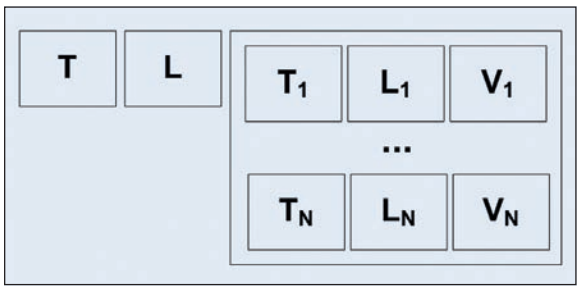
Для стандартных типов данных (таких как целые числа, действительные числа, битовые строки и т. д.) правила кодирования содержимого содержатся в **ASN.1 BER** (о некоторых из этих правил мы поговорим далее); для остальных типов правила кодирования могут быть любыми. Если тип данных является составным (то есть включает несколько элементов данных), то его содержимое – закодированные значения элементов данных, составляющих тип данных; каждое закодированное значение содержит тройку тэг, длина, содержимое. Длина содержимого составного типа данных равняется сумме длин закодированных значений элементов данных. Пример составного типа данных – последовательность (список элементов, в терминах **ASN.1**). Эти концепции правил кодирования **ASN.1 BER** показаны на рис. 1 и 2: рис. 1 показывает пример простого типа данных, рис. 2 – составного типа данных (здесь **T** – это тэг, **L** – длина, **V** – содержимое).

Про **ASN.1** можно сказать еще следующее: **ASN.1** – это аналог XML для двоичных протоколов. Чем же плох XML, если для двоичных протоколов применяется другое, в чем-то аналогичное ему решение? Главный недостаток XML в том, что это текстовое представление данных, и, соответственно, его размер больше (в грубых оценках, где-то на порядок) двоичного представления данных. Другой большой недостаток XML – тот факт, что определение типов данных (например, с использованием схем XSD) оторвано от самих данных. С другой стороны, **ASN.1** – это набор стандартов для кодирования двоичных данных, обработка которых, в целом, более сложна. К тому же для работы с XML существует целый ряд технологий (таких как XQuery, XSLT), которых нет для **ASN.1**.

Давайте перейдем непосредственно к примеру. Наша задача – написать сериализацию и десериализацию (кодирование и декодирование) объектов языка **Erlang** в соответствии с правилами **ASN.1 BER**. Вполне очевидно, что данная задача состоит из двух практически независимых друг от друга частей: из кодирования и декодирования данных. Также вполне очевидно, что начнем мы с части, отвечающей за кодирование данных.

Тип данных (он же тэг данных) – величина трехкомпонентная: он состоит из класса типа данных, формы типа данных и идентификатора типа данных. Поэтому для него логично определить соответствующую запись (и поместить ее в файл **asn1_tag.hrl**):

```
-record(tag, {class, form, tag_value}).
```



› Рис. 2.

битовых строк 2

Как и во всех других примерах, в качестве первого шага мы определяем модуль (и не забываем, что имя файла – это имя модуля с расширением **.hrl**), подключаем файлы с определениями и задаем список экспортируемых функций. Экспортируемых функций у нас всего две: функция **build/1** для построения функции диспетчера для выбора подходящей функции кодирования данных и функция **encode/2** для кодирования объектов **Erlang**.

```
-module(asn1_encoder).
-include("asn1_tag.hrl").
-export([build/1, encode/2]).
```

Функция **build/1**, как уже говорилось выше, служит для создания функции-диспетчера для выбора подходящей функции кодирования данных. Для этого она использует список пар (кортежей из двух значений) из двух анонимных функций (лямбда-выражений): первая функция проверяет, может ли переданный ей объект быть закодирован при помощи второй функции в этой паре.

Решение о том, подходит ли объект (может ли функция для кодирования закодировать данный объект), принимается не только на основании типа объекта, но и на основании значения объекта. Это связано с тем, что объекты **Erlang** одного и того же типа в зависимости от значения объекта должны кодироваться по-разному; так, например, атомы **true** и **false** являются логическими значениями и должны кодироваться отличным от атомов образом.

При создании функции-диспетчера мы используем как внутренний (заданный нами) список пар функций, так и внешний (задаваемый пользователем функции через единственный параметр) список пар функций. Внутренний список пар функций обрабатывает общие ситуации кодирования (когда объект является списком, кортежем, целым числом и т. д.); внешний список (задаваемый пользователем) служит для обработки специфичных ситуаций кодирования – например, если мы хотим кодировать записи отличным от обычных кортежей образом. Для этого пары функций из внешнего списка идут всегда перед парами функций из внутреннего списка – это означает, что у них более высокий приоритет. Более того, если одна пара функций идет перед другой, то это означает, что приоритет у этой пары функций выше, т. к. эта пара функций будет использована в процедуре выбора подходящей функции кодирования первой. Для пар из внутреннего списка в качестве первой функции (функции, которая проверяет, может ли быть закодирован объект другой функцией из пары) используется BIF **is_XXX/1**, где **XXX** – тип кодируемого объекта (для функций проверки из внешнего списка может использоваться любой алгоритм).

```
build(ExternalEncoders) when is_list(ExternalEncoders) ->
  InternalEncoders =
  [
    {fun is_boolean/1, fun encode_boolean/2},
    {fun is_integer/1, fun encode_integer/2},
    {fun is_float/1, fun encode_real/2},
    {fun is_binary/1, fun encode_octetstring/2},
    {fun is_bitstring/1, fun encode_bitstring/2},
    {fun is_list/1, fun encode_sequence/2},
    {fun is_tuple/1, fun encode_tuple/2},
    {fun is_atom/1, fun encode_atom/2}
  ],
  EncodersList = ExternalEncoders ++ InternalEncoders,
  fun(Value, Dispatcher) -> first(EncodersList, Value, Dispatcher)
end.
```

Функция **encode/2** использует функцию-диспетчер кодирования (которую мы построили при помощи функции **build/1**) для кодирования объекта, передаваемого в качестве первого параметра. Работа этой функции полностью основана на использовании функции-диспетчера кодирования: мы вызываем функцию-диспетчер, передавая в качестве параметров кодируемый объект и саму функцию-диспетчер кодирования (т. к. аналога указателя **this** из языка **Java** и ему подобным у нас нет). Если передаваемый объект может быть закодирован, то будет возвращен кортеж, состоящий из атома **ok** и закодированного исходного объекта (в виде битовой строки); если же передаваемый объект не может быть закодирован, то будет возвращен атом **false**. В последнем случае, мы генерируем исключение времени выполнения.

```
encode(Value, EncodeDispatcher) ->
  case EncodeDispatcher(Value, EncodeDispatcher) of
    {ok, Result} -> Result;
    false -> erlang:error(unsuitable_value)
  end.
```

Работа функции диспетчера основана на функции **first/3**. Эта функция последовательно проверяет пары функций, и как только проверка для пары будет положительна (проверка осуществляется при помощи первой функции из пары), исходный объект будет закодирован при помощи второй функции из пары (в виде кортежа из атома **ok** и результата кодирования). Если же объект не удовлетворяет ни одной паре, то будет возвращен атом **false**.

```
first([], _Value, _EncoderDispatcher) -> false;
first([{Predicate, Encoder} | Rest], Value, EncoderDispatcher) ->
  case Predicate(Value) of
    true -> {ok, Encoder(Value, EncoderDispatcher)};
    false -> first(Rest, Value, EncoderDispatcher)
  end.
```

Полезные заметки: Стандарты ASN.1

- ITU-T Rec. X.680 | ISO/IEC 8824-1. Спецификация на базовую нотацию.
- ITU-T Rec. X.681 | ISO/IEC 8824-2. Спецификация на информационные объекты.
- ITU-T Rec. X.682 | ISO/IEC 8824-3. Спецификация на ограничения.
- ITU-T Rec. X.683 | ISO/IEC 8824-4. Спецификация на параметризацию ASN.1.
- ITU-T Rec. X.690 | ISO/IEC 8825-1. Спецификация на BER (Basic encoding rules), CER (Canonical encoding rules) и DER (Distinguished encoding rules).
- ITU-T Rec. X.691 | ISO/IEC 8825-2. Спецификация на PER (Packed encoding rules).
- ITU-T Rec. X.692 | ISO/IEC 8825-3. Спецификация на ECN (Encoding control notation).
- ITU-T Rec. X.693 | ISO/IEC 8825-4. Спецификация на XER (XML Encoding rules).
- ITU-T Rec. X.694 | ISO/IEC 8825-5. Спецификация на отображение на XSD.
- ITU-T Rec. X.695 | ISO/IEC 8825-6. Спецификация на регистрацию и применение инструкций кодирования PER (Packed encoding rules).

Теперь перейдем непосредственно к кодированию данных. Начнем с кодирования типа данных (он же тэг). Он у нас состоит из трех частей (и для его представления мы используем запись типа **tag**). Поэтому мы отдельно кодируем класс (и получаем битовую строку размером 2 бита), форму (и получаем битовую строку размером 1 бит) и идентификатор типа данных, после чего склеиваем три полученных битовых строки в одну при помощи BIF **list_to_bitstring/1**.

```
encode_tag(#tag{class = Class, form = Form, tag_value = Value})
->
  list_to_bitstring([encode_tag_class(Class), encode_tag_
form(Form), encode_tag_value(Value)]).
```

Функция **encode_tag_class/1** отвечает за кодирование класса типа данных (тэга). В качестве значения класса используется множество предопределенных атомов. Принцип работы этой функции тривиален; стоит лишь отметить, что возвращает она битовую строку размером 2 бита.

```
encode_tag_class(universal) -> <<2#00:2>>;
encode_tag_class(application) -> <<2#01:2>>;
encode_tag_class(context_specific) -> <<2#10:2>>;
encode_tag_class(private) -> <<2#11:2>>.
```

Функция **encode_tag_form/1** отвечает за кодирование формы данных. В качестве значения формы используется множество предопределенных атомов. Принцип работы этой функции также тривиален; отметим, что возвращает она битовую строку размером 1 бит.

```
encode_tag_form(primitive) -> <<0:1>>;
encode_tag_form(constructed) -> <<1:1>>.
```

Пришла пора более интересной функции: **encode_tag_value/1**, которая применяется для кодирования идентификатора типа данных. Кодирование идентификатора зависит от того, меньше его значение 31 или нет. Если значение идентификатора меньше 31, то идентификатор кодируется как битовая строка размером 5 бит. Если значение идентификатора больше или равно 31, то он кодируется более сложным способом: сначала идет сегмент размером 5 бит, содержащий число 31 (или 2#11111), после чего идут сегменты размеров 8 бит, содержащие закодированное значение идентификатора. Значение идентификатора кодируется следующим образом: сначала идентификатор кодируется как битовая строка, состоящая из сегментов размером 7 бит; после чего каждый сегмент увеличивается до 8 бит добавлением в качестве старшего бита 1, если это не последний сегмент в битовой строке, и 0 – в противном случае. Таким образом, при декодировании значения идентификатора мы сможем понять, когда нам необходимо остановиться. Следует добавить еще следующее: размер битовой строки, содержащей закодированное значение типа данных, всегда будет кратен 8 битам, в чем легко может убедиться каждый.

```
encode_tag_value(Value) when (Value >= 0) and (Value =< 30) ->
<<Value:5>>;
encode_tag_value(Value) when Value >= 31 ->
  SegmentCount = (Value div 128) + 1,
  SegmentList = encode_tag_value(<<TagValue:(SegmentCount *
7)>>, []),
  list_to_bitstring([<<2#11111:5>>] ++ lists:reverse(SegmentList)).
```

Функция **encode_tag_value/2** занимается увеличения сегментов размером 7 бит до 8 бит при помощи добавления в качестве старшего бита 1, если соответствующий сегмент размером 7 бит в битовой строке не последний, и 0 – в противном случае.

```
encode_tag_value(<<Segment:7>>, SegmentList) -> [<<0:1,
Segment:7>>] ++ SegmentList;
encode_tag_value(<<Segment:7, Rest/bitstring>>, SegmentList) ->
  encode_tag_value(Rest, [<<1:1, Segment:7>>] ++ SegmentList).
```

Следующий необходимый шаг при кодировании данных – кодирование длины (или количества октетов, необходимых для сохранения данных) кодируемых данных. Кодирование длины проще кодирования типа, но и тут у нас есть два варианта, в зависимости от того, меньше ли значение длины 128 или нет. В первом случае мы кодируем длину одним сегментом, размером 8 бит, старший бит которого равен 0, а младшие 7 бит содержат значение длины. Во втором случае мы кодируем длину несколькими сегментами размером 8 бит: при этом у первого сегмента старший бит равен 1, младшие 7 бит содержат количество октетов, необходимых для кодирования длины, а все остальные октеты содержат закодированное значение длины.

```
encode_length(LengthValue) when (LengthValue >= 0) and
(LengthValue =< 127) -> <<0:1, LengthValue:7>>;
encode_length(LengthValue) when LengthValue >= 128 ->
  OctetCount = (LengthValue div 256) + 1,
  list_to_binary([<<1:1, OctetCount:7>>] ++ [binary:encode_
unsigned(LengthValue, big)]).
```

Теперь переходим непосредственно к кодированию самих данных (объектов **Erlang**). Начнем с логических значений (в языке **Erlang** логические значения представлены атомами **true** и **false**). Логические значения кодируются следующим образом: тип данных имеет значение 1 (класс – **universal**, форма – **primitive**, идентификатор – 1), длина – 1 октет, логическое значение **false** кодируется значением 0, логическое значение **true** – любым ненулевым значением (мы будем кодировать значением **2#11111111=255**).

```
encode_boolean(true, _EncodeDispatcher) ->
  Tag = encode_tag(#tag{class = universal, form = primitive, tag_
value = 1}),
  list_to_binary([Tag, encode_length(1), <<2#11111111:8>>]);
encode_boolean(false, _EncodeDispatcher) ->
  Tag = encode_tag(#tag{class = universal, form = primitive, tag_
value = 1}),
  list_to_binary([Tag, encode_length(1), <<2#00000000:8>>]).
```

Следующий тип данных, кодирование которого мы рассмотрим – это целые числа. Целые числа кодируются следующим образом: тип данных имеет значение 2 (класс – **universal**, форма – **primitive**, идентификатор – 2), длина ничем не ограничена.

```
encode_integer(Number, _EncodeDispatcher) ->
  Tag = encode_tag(#tag{class = universal, form = primitive, tag_
value = 2}),
  NumberBinary = encode_integer_value(Number),
  list_to_binary([Tag, encode_length(size(NumberBinary)),
NumberBinary]).
```

Метод **encode_integer_value/1** кодирует непосредственно значение целого числа. Кодирование целых чисел, пожалуй, является самой сложной операцией, в связи со способом кодирования положительных и отрицательных целых чисел. Положительные целые числа кодируются следующим образом: целое число сохраняется как битовая строка с размером, кратным 8 бит (с порядком записи байт **big-endian**); если старший бит битовой строки равен 1, то к битовой строке слева дописывается октет, содержащий 0. Отрицательные целые числа кодируются в дополнительном коде представления числа, при этом количество октетов размером 8 бит и значением **16#FF** должно быть минимально необходимым. Это означает (для кодирования отрицательного числа), например, что для кодирования числа **-128 = 16#80** достаточно одного октета, а для кодирования числа **-129 = 16#FF7F** уже нужно два октета.

```
encode_integer_value(Number) when Number >= 0 ->
  OctetCount = get_octet_count(Number, 0),
  NumberBinary = <<Number:(8 * OctetCount)/integer-signed-big>>,
  <<OldestBit:1, _Rest/bitstring>> = NumberBinary,
```

» **Пропустили номер?** Узнайте на с. 104, как получить его прямо сейчас.

```
  if
    OldestBit == 1 -> list_to_binary([<<0:8>>, NumberBinary]);
    OldestBit == 0 -> NumberBinary
  end;
encode_integer_value(Number) when Number < 0 ->
  OctetCount = get_octet_count(Number, 0),
  <<Number:(8 * OctetCount)/integer-signed-big>>.
```

Метод **get_octet_count/2** служит для подсчета количества октетов, необходимых для кодирования целого числа. Принцип его работы тривиален.

```
get_octet_count(0, 0) -> 1;
get_octet_count(0, Count) -> Count;
get_octet_count(Number, 0) when Number < 0 -> get_octet_
count(Number div -129, 1);
get_octet_count(Number, Count) -> get_octet_count(Number div
256, Count + 1).
```

Перейдем теперь к действительным числам. С ними все проще, чем с целыми: действительные числа (по основанию 10) кодируются в строковом представлении. При этом тип данных имеет значение 9 (класс – **universal**, форма – **primitive**, идентификатор – 9).

```
encode_real(0.0, _EncodeDispatcher) ->
  Tag = encode_tag(#tag{class = universal, form = primitive, tag_
value = 9}),
  list_to_binary([Tag, <<0:8>>]);
encode_real(Number, _EncodeDispatcher) ->
  Tag = encode_tag(#tag{class = universal, form = primitive, tag_
value = 9}),
  NumberStr = float_to_list(Number),
  list_to_binary([Tag, encode_length(length(NumberStr) + 1),
<<2#00000011>>, NumberStr]).
```

Разберемся с кодированием более сложных типов данных. Начнем с битовых строк – в данном контексте под битовой строкой мы понимаем последовательность бит, количество которых не кратно 8. Битовая строка кодируется следующим образом: она разбивается на сегменты размером 8 бит и остаток, размер которого меньше 8 бит. После чего справа добавляем сегмент такого размера (от 1 до 7 бит), чтобы остаток и этот сегмент в сумме имели размер 8 бит, и заполняем этот сегмент значением 0. Затем перед битовой строкой дописываем октет, содержащий количество бит добавленного справа сегмента (от 1 до 7). Это нужно потому, что длина данных задается в количестве используемых октетов. Соответственно, длина закодированной битовой строки будет на единицу больше числа используемых для хранения битовой строки октетов. Для битовых строк тип данных имеет значение 3 (класс – **universal**, форма – **primitive**, идентификатор – 3).

```
encode_bitstring(BitString, _EncodeDispatcher) ->
  OctetCount = (bit_size(BitString) div 8) + 1,
  UnusedBitCount = 8 - bit_size(BitString) rem 8,
  Tag = encode_tag(#tag{class = universal, form = primitive, tag_
value = 3}),
  EncodedValue = list_to_bitstring([BitString,
<<0:UnusedBitCount>>]),
  list_to_binary([Tag, encode_length(OctetCount + 1),
<<UnusedBitCount:8>>, EncodedValue]).
```

Строки октетов кодируются гораздо проще: сначала идет тип данных, равный 4 (класс – **universal**, форма – **primitive**, идентификатор – 4), потом длина закодированных данных (в нашем случае, количество октетов в строке), после чего идет сама строка.

```
encode_octetstring(OctetString, _EncodeDispatcher) ->
  Tag = encode_tag(#tag{class = universal, form = primitive, tag_
value = 4}),
  list_to_binary([Tag, encode_length(size(OctetString)),
OctetString]).
```

Теперь займемся кодированием составных типов данных: списков и кортежей. Как кодируется содержимое составных типов данных? Ответ очевиден: мы берем первый элемент содержимого и кодируем у него последовательно тип данных, длину и содержимое, затем то же самое делаем для второго элемента, и так до тех пор, пока все элементы содержимого не будут закодированы. Длина (или, что то же самое, число октетов), полученная в результате кодирования содержимого, становится длиной закодированного составного элемента. Для списков значение типа данных равняется 48 (класс – **universal**, форма – **constructed**, идентификатор – 16)

```
encode_sequence(Sequence, EncodeDispatcher) ->
  Tag = encode_tag(#tag{class = universal, form = constructed,
tag_value = 16}),
  {ContentLength, ContentBinary} = encode_sequence_
content(Sequence, EncodeDispatcher),
  list_to_binary([Tag, encode_length(ContentLength),
ContentBinary]).
```

Кортежи, как говорилось выше, кодируются точно так же, как и списки. Только для кортежей значение типа данных равняется **16160 = 2#0011111100100000** (класс – **universal**, форма – **constructed**, идентификатор – 32). Следует сказать про значение идентификатора следующее: до этого все типы данных кодировались с использованием стандартных идентификаторов типов, но нотация **ASN.1** не позволяет различать такие типы данных, как списки и кортежи. Поэтому для кортежей был выбран идентификатор, равный 32, но не являющийся стандартным. Поэтому, когда мы будем использовать наш пример универсальным образом, с кодированием и декодированием кортежей, скорее всего, будут проблемы. Выходов из этой ситуации два: либо не использовать нашу систему универсальным способом, либо ограничится стандартными типами данных (что означает – вместо кортежей использовать списки).

```
encode_tuple(Tuple, EncodeDispatcher) ->
  Tag = encode_tag(#tag{class = universal, form = constructed,
tag_value = 32}),
  {ContentLength, ContentBinary} = encode_sequence_
content(tuple_to_list(Tuple), EncodeDispatcher),
  list_to_binary([Tag, encode_length(ContentLength),
ContentBinary]).
```

Метод **encode_sequence_content/2** реализует алгоритм кодирования содержимого объекта составного типа данных, о котором мы говорили выше. В этой реализации мы считаем, что составной тип данных является списком, поэтому для всех других составных типов данных необходимо преобразовывать их содержимое в список (что делается, например, в методе **encode_tuple/2**).

```
encode_sequence_content(Sequence, EncodeDispatcher) ->
  lists:foldl(fun(Element, {Length, Binary}) ->
    EncodedElement = encode(Element, EncodeDispatcher),
    EncodedSize = size(EncodedElement),
    {Length + EncodedSize, list_to_binary([Binary,
EncodedElement])}
  end, {0, <<>>}, Sequence).
```

Последний поддерживаемый в нашем примере тип данных – атом. Он кодируется по тем же принципам, что и строка октетов; для этого атом преобразуется в битовую строку при помощи BIF **atom_to_binary/2** в кодировке **utf8**. Для атомов значение типа данных равняется **16161 = 2#0011111100100001** (класс – **universal**, форма – **constructed**, идентификатор – 33)

```
encode_atom(Atom, _EncodeDispatcher) ->
  Tag = encode_tag(#tag{class = universal, form = primitive, tag_
value = 33}),
  AtomBinary = atom_to_binary(Atom, utf8),
  list_to_binary([Tag, encode_length(size(AtomBinary)),
AtomBinary]).
```

Вот и все с кодированием данных, но остается еще задача, как закодированные нами данные потом раскодировать. Об этом мы поговорим в следующей статье – завершающей цикл «Практикум функционального программирования на языке **Erlang**». LXF