Erlang: Базовые

Базовыми сущностями языка функционального программирования, естественно, являются функции. **Андрей Ушаков** начинает новую серию уроков...

Наш эксперт

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

В предыдущих выпусках (**LXF 143**, **144**) была напечатана статья, посвященная введению в язык программирования Erlang. В новом цикле статей я продолжу рассказ о языке Erlang и концепциях функционального программирования. Данная статья посвящена одной из фундаментальных сущностей вообще всех языков программирования (в том числе и Erlang) – функциям.

Во многих языках программирования функции не являются типом данных. Это означает, что я не могу объявить переменную и присвоить ей функцию, либо передать функцию как аргумент вызова другой функции. Конечно, не все так плохо, но для работы с функциями как с типами данных приходится совершать дополнительные действия. Так, например, в C++ для этого мы вводим указатель на функцию, либо вместо функций используем функторы; в Java используем типы-обертки (например, анонимные классы) либо ссылки на метаданные. Понятно, что хочется работать с функциями, как с другими типами данных: было бы удобно иметь возможность легко объявить переменную и присвоить ей в качестве значения функцию, либо передать одну функцию в качестве аргумента другой. Язык Erlang, так же как и другие функциональные языки программирования, это позволяет. Давайте поговорим о функциях более подробно.

> «Хочется работать с функциями как с другими типами данных.»

Все функции всегда определены в модулях. Объявление модуля – это всегда первая строка в файле. Модуль объявляется при помощи директивы **module**, содержащей имя модуля. Имя модуля есть атом (и поэтому должно начинаться с маленькой буквы), совпадающий с именем файла без расширения. Все функции в модуле делятся на экспортируемые и неэкспортируемые. По умолчанию функция является неэкспортируемой и будет видна только внутри модуля. Чтобы сделать ее экспортируемой, необходимо ее сигнатуру (сигнатура функции – это имя функции, после которого идет знак '/' и арность функции) прописать в директиве **export**. Директива **export** содержит список сигнатур экспортируемых функций. Например, объявление модуля и экспортируемых функций может выглядеть следующим образом (имя файла, соответственно, **example1.erl**):

```
-module(example1).
-export([func1/1, func2/2, func3/0]).
```

Использовать функции из того же модуля, в котором они объявлены, просто: достаточно обратиться к ним по имени (и передать соответствующий список аргументов). А что же с функциями из других модулей (понятно, что речь идет только об экспортируемых функциях)? У нас есть два варианта решения этого вопроса (так же как и в большинстве языков программирования): использовать полное имя функции либо импортировать функции из другого модуля. Полное имя функции – это имя модуля, в котором функция определена, после которого идут знак '**:**' и имя функции, заданное при определении. Например, обращение к функции **seq**, определенной в модуле **lists**, будет выглядеть следующим образом: **lists:seq(1, 10).** Импорт функций, с другой стороны, позволяет использовать функции из других модулей по их имени, заданному при определении. Директива импорта выглядит следующим образом:

```
-import(ModuleName, FuncList).
```

Здесь **ModuleName** – имя модуля, из которого производится импорт; **FuncList** – список импортируемых функций. Например, импорт функций **seq/2** и **seq/3** из модуля **lists** будет выглядеть следующим образом:

```
-import(lists, [seq/2, seq/3]).
```

Так же, как и в большинстве языков программирования, в Erlang есть автоматически импортируемые функции. Эти функции называются BIF и импортируются они (не все) из модуля **erlang**.

Обратим свой взор теперь к объявлению функции. А точнее, на объявление нескольких вариантов одной и той же функции. Как вы помните из предыдущих статей, при вызове функции поиск подходящего варианта осуществляется при помощи двух механизмов (которые могут работать совместно в одном варианте): операции соответствия шаблону [pattern-matching] и выражения охраны [guards]. И если с операцией соответствия шаблону все достаточно просто, то с выражением охраны ситуация более интересная. Выражение охраны – это булевское выражение (на самом деле выражение охраны может возвращать любой атом, но истинным значением бу-

СУЩНОСТИ

дет значение **true**), и вопрос заключается в том, какие операции я как разработчик могу использовать. Например, вправе ли я написать объявление такого варианта функции:

```
calculate(X) when math:sin(X) > math:cos(X) → … ;
```

Из документации видно, что в выражениях охраны могут появляться только следующие операции: атомы, операции сравнения, арифметические выражения, логические выражения и ограниченный набор BIF'ов. В этот набор входят все функции, проверяющие тип аргумента: **is_*/1** (например, **is_atom/1**), **abs/1**, **bit_size/1**, **byte_size/1**, **element/2**, **float/1**, **hd/1**, **length/1**, **node/0**, **node/1**, **round/1**, **self/0**, **size/1**, **tl/1**, **trunc/1**, **tuple_size/1**. И ответ на приведенный выше вопрос будет следующий: мы не вправе написать подобное объявление варианта функции – его не пропустит компилятор.

Зачем сделано такое ограничение на выражения охраны? Затем, чтобы гарантировать, что выражения охраны свободны от побочных эффектов. Что же делать, если мне необходимо выбрать вариант функции в зависимости от более сложного условия, чем позволяет задать разрешенный набор операций в выражениях охраны (как в приведенном примере)?

Ответ достаточно очевиден: использовать одно из выражений **if** или **case**. Рассмотрим приведенный выше пример: пусть **calculateWhenX/1** и **calculateWhenY/1** представляют две ветви выполнения (то, что мы хотели записать как два варианта функции), тогда функция **calculate/1** будет иметь следующий вид:

```
calculate(X) →
        if
                math:sin(X) > math:cos(X) →
calculateWhenX(X);
                true → calculateWhenY(X)
        end.
```

И последнее про выражения охраны: выражение охраны на самом деле может быть списком выражений, использующим в качестве разделителя либо символ ';', либо символ ','. В первом случае выражение охраны истинно, если истинно хотя бы одно выражение из списка; во втором случае – если истинны все выражения.

Перейдем к рассмотрению рекурсии. Рекурсия – это возможность функции вызывать саму себя. Реализуется она через стек, и не случайно, что при большой глубине рекурсивных вызовов наступает переполнение стека. Есть рекурсия и в Erlang, но помимо обычной рекурсии в Erlang присутствуют рекурсивные вызовы специального типа – хвостовая рекурсия. Если в результате выполнения тела функции последней операцией будет рекурсивный вызов самой себя, то такой рекурсивный вызов называется прямой хвостовой рекурсией. Возможна ситуация, когда в результате выполнения функции **A** последней операцией будет вызов функции **B**, а в результате выполнения функции **B** последней операцией будет вызов функции **A**. И в этом случае подобный рекурсивный вызов будет являть хвостовую рекурсию, только подобная рекурсия называется непрямой хвостовой рекурсией.

Особенность хвостовой рекурсии в том, что компилятор умеет обрабатывать такой случай нерекурсивным образом и, следовательно, хвостовая рекурсия может быть сколь угодно глубокой и при этом не вызовет переполнения стека. Рекурсия важна при реализации рекурсивных алгоритмов и/или при работе с рекурсивными структурами данных (например, с деревьями). В Erlang рекурсия (точнее, хвостовая рекурсия) еще важна и потому, что это единственный механизм построения циклических структур управления (циклических алгоритмов). Так, например, в Erlang отсутствует цикл **for**; предположим, что нам необходима структура управления, эмулирующая этот цикл (для простоты предположим, что нам необходимо эмулировать цикл **for**, который просто выполняется заданное число раз). Тогда мы можем эмулировать цикл **for** следующим образом:

```
for(Start, Count, Func, Acc) → for_impl(Start, 0, Count, Func, Acc).
for_impl(_, Index, Count, _, Acc) when Index >= Count → Acc;
for_impl(Start, Index, Count, Func, Acc) →
                for_impl(Start, Index+1, Count, Func, Func(Start + Index,
Acc)).
```

и далее, мы можем создать список из чисел от 1 до 10 следующим образом:

```
for(1, 10, fun(Number, Acc) → Acc ++ [Number] end, []).
```

В приведенном выше примере я использовал объявление анонимной функции. Давайте рассмотрим их более подробно. Анонимная функция (или лямбда) – это безымянная функция, определяемая локально по месту использования. В чем преимущества определения по месту таких безымянных функций? Такое определение более наглядно, чем определение отдельной функции, когда объявляемая функция небольшая. Если определение анонимной функции используется только в одном месте, то такой способ не «загрязняет» исходный код еще одним определением функции и не влияет на операцию поиска функции по сигнатуре, выполняемую компилятором. Определение анонимной функции в общем случае имеет следующий вид:

```
fun
        (Pattern11, …, Pattern1N) [when Guard1] → Body1;
        … ;
        (PatternK1, …, PatternKN) [when GuardK] → BodyK
end
```

Из общего определения видно, что и в анонимной функции мы можем задавать несколько ее вариантов, выбор которых может происходить как за счет операции соответствия шаблону, так и за счет выражений охраны. Если анонимная функция нужна только для передачи вызова обычной, неанонимной функции, то вместо написания подобного кода:

```
fun(X1, …, XN) → module:func_name(X1, …, XN) end
```

мы можем написать более коротко: **fun module:func_name/N** (ну или **fun func_name/N** – для функции из текущего модуля). Такая короткая запись обычно используется, чтобы создать ссылку на существующую неанонимную функцию.

Перейдем теперь к следующему понятию функционального программирования – функциям высшего порядка. Функции высшего порядка отличаются от обычных функций только тем, что они принимают в качестве аргумента функцию, либо их возвращаемое значение есть функция (либо и то, и другое). Достаточно очевидны случаи, когда нам нужно, чтобы одна функция »

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

принимала другую в качестве параметра. Например, если вы разрабатываете библиотеку для численного интегрирования, то вам нужно передавать в функцию интегрирования – в качестве одного из параметров – функцию, по которой считается определенный интеграл. Несложно придумать еще массу примеров, когда будет необходимость в параметризации одной функции другой. Более того, взглянув на стандартную библиотеку, поставляемую с Erlang, можно увидеть массу функций, которые ожидают, что один (или несколько) из параметров будет другой функцией. Придумать пример функции, у которой возвращаемое значение – другая функция, несколько более сложно, но тоже возможно. Таким примером может быть функция-фабрика (паттерн «фабрика»), создающая и возвращающая другую функцию (подобную функцию-фабрику мы создадим в примере ниже).

С функциями высшего порядка тесно связано еще одно понятие функционального программирования – карринг [currying] (или каррирование функции). Карринг – это преобразование функции от пары аргументов в функцию, берущую свои аргументы по одному. Другими словами, это преобразование функции от пары аргументов в функцию от одного аргумента, возвращающую функцию от одного аргумента. Возникает вопрос: а зачем вообще это нужно? Для частичного задания аргументов функции прямо сейчас! Например, у нас есть функция, позволяющая вести поиск одного фрагмента текста в другом. Очевидно, что это функция двух аргументов (минимум двух аргументов). Предположим, что набор фрагментов текста, которые мы ищем, заранее предопределен, а фрагмент текста, в котором ведется поиск, заранее неизвестен. В таком случае было бы удобно иметь предопределенный набор функций от одного параметра, для поиска предопределенного фрагмента текста в заданном. Но создавать новую функцию для каждого предопределенного фрагмента текста (по которому будет вестись поиск) будет неправильно. И вот тут вступает в дело карринг – мы преобразуем функцию от двух аргументов в функцию, берущую свои аргументы по одному, и задаем один из аргументов. Пусть функция поиска одного фрагмента в другом называется **search_text/2**, тогда карринг функции будет выглядеть следующим образом:

```
SearchFun = fun(Search) → fun(Source) → search_text(Source, Search) end end,
SearchFragment1 = SearchFun("fragment1"),
```

Переменная **SearchFun** содержит ссылку на каррированную функцию **search_text/2** (т. е. ссылку на функцию одного аргумента, возвращающую функцию одного аргумента), а переменная **SearchFragment1** просто содержит ссылку на функцию одного аргумента, которая в итоге будет искать в заданном фрагменте текста строку **"fragment1"**.

> «Мы можем преобразовать арифметическое выражение в дерево.»

Настало время применить полученные знания на практике: давайте напишем простой парсер арифметических выражений. На вход он будет получать арифметическое выражение в виде строки, которая может содержать целочисленные константы, имена переменных и все арифметические действия. На выход он будет выдавать функцию одного аргумента, содержащую распарсенное выражение. В дальнейшем, передавая полученной функции список пар «имя переменной – значение» (список кортежей, в котором первый элемент – имя переменной в виде строки, второй элемент – значение этой переменной), мы можем вычислять значение этого выражения для конкретных значений переменных. Для простоты наш парсер будет содержать следующие допущения: считаем, что арифметическое выражение не содержит скобки, знаки перед константами и переменными, исходную строку не чистим от пробельных символов, не вводим полноценную обработку ошибок.

Итак, начнем. Первое, что мы должны сделать – это разбить полученную строку на лексемы. В нашем случае лексемы могут быть следующие: целочисленная константа, имя переменной, знак арифметического действия. Знак арифметического действия, помимо того, что является лексемой, является также и разделителем, разбивающим выражение на лексемы (т. к. знаки перед константами и переменными не поддерживаются). Например, строка **2+a** разобьется на следующие лексемы: **2**, **+**, **a**. Для разбития исходной строки на лексемы служит функции **get_tokens/2** и **get_tokens_impl/2**:

```
get_tokens(InputStr, Delimiters) ->
        get_tokens_impl(InputStr, Delimiters, []).
get_tokens_impl([], _, TokenList) -> lists:reverse(TokenList);
get_tokens_impl(InputStr, Delimiters, TokenList) ->
        {Token, Rest} = lists:splitwith(fun(Char) -> not lists:member(Char, Delimiters) end, InputStr),
        if
                Rest /= [] ->
                        [DelimiterChar | NextRest] = Rest,
                        get_tokens_impl(NextRest, Delimiters, [[DelimiterChar]] ++ [Token] ++ TokenList);
                Rest == [] ->
                        get_tokens_impl([], Delimiters, [Token]++TokenList)
        end.
```

После разбиения строки на лексемы наступает стадия парсинга. В результате парсинга мы должны получить некую структуру данных, позволяющую вычислять значение функции при заданных значениях переменной. Что же должна представлять собой эта структура данных, с учетом того факта, что в результате она должна быть обернута в функцию от одного аргумента? Рассмотрим для начала константу. Константу можно представить в виде функции от двух аргументов: списка пар «имя переменной – значение» и значения константы, возвращающей значение константы. Во время парсинга, при помощи карринга и частичного задания аргументов (задавая значение константы), мы можем преобразовать эту функцию в функцию от одного аргумента. Рассмотрим теперь переменную. Переменную можно представить в виде функции от двух аргументов: списка пар «имя переменной – значение» и имени переменной, возвращающей значение переменной по ее имени. Точно так же, как и в случае константы, во время парсинга мы можем преобразовать эту функцию в функцию от одного аргумента.

Рассмотрим, наконец, какую-либо бинарную операцию – например, сложение. Подобную операцию можно представить в виде функции трех аргументов: списка пар «имя переменной – значение», левого операнда от одного аргумента и правого операнда от одного аргумента, возвращающую результат выполнения операции. И точно так же, как в случае константы и переменной, мы можем преобразовать эту функцию в функцию от одного аргумента. Действуя подобным образом, мы можем преобразовать все арифметическое выражение в дерево из преобразованных (при помощи карринга и частичного задания аргументов) функций от одного аргумента. Функция **constant_fun/2** служит для представления константы, функция **variable_fun/2** – для представления переменной, функции **addition_fun/3**, **subtraction_fun/3**,

multiplication_fun/3, **division_fun/3** – для представления арифметических действий. В статье приводим объявление только для функции **addition_fun/3**, т. к. объявление остальных функций для арифметических действий аналогичное:

```
constant_fun(_, Value) -> Value.
variable_fun(VarList, Name) ->
        FindResult = lists:keyfind(Name, 1, VarList),
        if
                FindResult == false -> erlang:error({variable_
not_found, Name});
                true -> element(2, FindResult)
        end.
addition_fun(VarList, LeftMember, RightMember) ->
        LeftMember(VarList) + RightMember(VarList).
```

Следующий шаг – преобразование этих функций в функции от одного аргумента при помощи карринга и частичного задания аргументов. Это происходит при создании операндов (функции **build_operand/1**, **build_operand_impl/2**) и операций (функция **build_operator/3**):

```
build_operand(Operand) ->
        build_operand_impl(Operand,
string:to_integer(Operand)).
build_operand_impl(Operand, {error, _}) ⟶
        fun(VarList) -> variable_fun(VarList, Operand) end;
build_operand_impl(_, {Int, []}) ⟶
         fun(VarList) -> constant_fun(VarList, Int) end.
build_operator("*", LeftOperand, RightOperand) ->
        fun(VarList) -> multiplication_fun(VarList, LeftOperand,
RightOperand) end;
build_operator("/", LeftOperand, RightOperand) ->
        fun(VarList) -> division_fun(VarList, LeftOperand,
RightOperand) end;
build_operator("+", LeftOperand, RightOperand) ->
        fun(VarList) -> addition_fun(VarList, LeftOperand,
RightOperand) end;
build_operator("-", LeftOperand, RightOperand) ->
                fun(VarList) -> subtraction_fun(VarList,
LeftOperand, RightOperand) end.
```

Следует отметить, что создавать операнды нужно только для констант и переменных; распарсенная часть выражения и так является операндом для текущей операции.

Далее наступает самое интересное: преобразование списка лексем в дерево функции. Если бы все операции имели одинаковый приоритет, то подобное преобразование было бы тривиальной операцией: иди себе просто по списку лексем и преобразовывай по мере прохождения. В нашем случае все немного сложнее – приоритет операции имеют разный. Пусть мы идем по списку лексем и преобразуем его в дерево по мере прохождения. Пусть есть уже преобразованная часть (левый операнд), и текущая операция – низкоприоритетная. Тогда наши действия зависят от того, какая операция стоит после правого операнда. Возможны три случая: после правого операнда больше операций нет (конец списка); после правого операнда стоит низкоприоритетная операция; и после правого операнда стоит высокоприоритетная операция. В первых двух случаях все хорошо, и мы можем преобразовать текущую операцию в дерево функций, связывающее левый и правый операнды. В третьем же случае мы запоминаем левый операнд и текущую операцию и начинаем строить новое дерево, начиная с высокоприоритетной операции. Когда мы встретим низкоприоритетную операцию или конец списка, мы объединяем два дерева функций в одно при помощи запомненной операции (левый операнд – запомненное дерево, правый операнд – новое дерево) и идем далее. Преобразованием в дерево функций занимаются функции **build_fun/1**, **build_fun_impl/2** и **process_operands/4**:

```
build_fun(TokenList) ->
[LeftOperand | Rest] = TokenList,
        build_fun_impl(Rest, build_operand(LeftOperand)).
build_fun_impl([], {PrevOperand, Operator, LeftOperand}) ->
        build_operator(Operator, PrevOperand, LeftOperand);
build_fun_impl([], LeftOperand) -> LeftOperand;
build_fun_impl(TokenList, {PrevOperand, PrevOperator,
LeftOperand}) ->
        [Operator | Rest] = TokenList,
        [RightOperand | NextRest] = Rest,
        CurrentOperand = build_operator(Operator,
LeftOperand, build_operand(RightOperand)),
        process_operands(PrevOperand, PrevOperator,
CurrentOperand, NextRest);
build_fun_impl(TokenList, LeftOperand) ->
        [Operator | Rest] = TokenList,
        [RightOperand | NextRest] = Rest,
        process_operands(LeftOperand, Operator, build_
operand(RightOperand), NextRest).
process_operands(LeftOperand, Operator, RightOperand, []) ->
        build_fun_impl([], build_operator(Operator,
LeftOperand, RightOperand));
process_operands(LeftOperand, Operator, RightOperand,
TokenList) ->
        NextOperator = hd(hd(TokenList)),
        IsPriorityOperator = lists:member(NextOperator, "*/"),
        if
                IsPriorityOperator -> build_fun_
impl(TokenList, {LeftOperand, Operator, RightOperand});
                true -> build_fun_impl(TokenList, build_
operator(Operator, LeftOperand, RightOperand))
        end.
```

Ну что же, осталось совсем немного: объявление модуля, списка экспортируемых функций и экспортируемой функции **parse/1**:

```
-module(arith_parse).
-export([parse/1]).
parse(InputStr) ->
        build_fun(get_tokens(InputStr, "*/+-")).
```

Сохраняем исходный код в файле с именем **arith_parse.erl**, запускаем среду выполнения Erlang. В консоли Erlang запускаем компиляцию: командой **c(arith_parse).**, после чего можно приступать к тестированию. Сначала получаем распарсенное дерево функции:

```
F1 = arith_parse("3-2*a+b/4*3-1").
```

После, мы можем посчитать значение арифметического выражения для конкретных значений переменных:

```
F1([{"a", 9}, {"b", 13}]).
```

Для значений переменных a = 9 и b = 13 получаем ответ –6.25, правильность которого легко проверить вручную

В порядке заключения: способы использования функций в функциональных языках и похожи, и не похожи на те, к которым мы привыкли. Мы привыкли, что функции – это контейнеры для исполняемого кода, и если мы хотим их использовать как-то иначе, то вынуждены делать дополнительные и не всегда удобные действия. В функциональных языках, как мы увидели, все гораздо проще: функции одновременно являются и контейнерами для кода, и полноценным типом данных. Более того, некоторые техники делают использование функций более удобным и гибким, по сравнению с использованием в императивных языках. А в следующей статье мы рассмотрим другую базовую сущность функциональных языков (и Erlang в том числе) – кортежи. LXF

» **Через месяц** Рассмотрим применение кортежей в Erlang.