Erlang: Базовые

Андрей Ушаков решительно настроен попрактиковаться и рассматривает ряд полезных примеров, в том числе про конечный автомат.

Наш эксперт

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

В прошлом номере (**LXF151**) мы пробовали применять знания о функциональном программировании на языке Erlang, которые изучали до этого. В этом номере мы продолжим этот процесс и на практике посмотрим, как нам строить свою работу, если необходимо сохранять состояние между вызовами функций (для чего нужны поля объектов в объектно-ориентированном программировании).

Основная идея функционального программирования в том, что вычисление значения функции должно зависеть только от входных аргументов этой функции и не предполагает хранения какого-либо состояния. Это означает, что при одних и тех же аргументах значение функции должно быть всегда одинаково. Понятно, что когда речь идет о функции, взаимодействующей с внешним миром (с файловой системой, с базами данных; даже просто с получением текущего времени), то это допущение будет неверным (мы не можем гарантировать неизменность внешних данных). Однако если мы рассматриваем функцию, которая реализует некоторый алгоритм без взаимодействия со внешним миром, то данное допущение справедливо; и более того, эта парадигма является одной из базовых при построении функциональных языков программирования (что справедливо и для Erlang). Как следует поступить с точки зрения функционального программирования, если необходимо сохранять некоторое состояние между вызовами одной или нескольких функций? Ответ достаточно очевиден: такие функции следует реализовать так, чтобы они принимали исходное состояние в качестве одного из аргументов и возвращали результирующее состояние в полученном значении функции.

После небольшого теоретического вступления перейдем к практике: на несложном примере рассмотрим, как можно сохранять состояние и работать с состоянием. Давайте создадим конечный автомат и применим его для создания лексического анализатора чисел. Не вдаваясь глубоко в теорию, напомним в двух словах, что такое конечные автоматы. Конечные автоматы – это формализованный способ описания систем с конечным числом состояний, переход между которыми осуществляется при возникновении некоторого события (например, если лексический анализатор строится на основе конечного автомата, то событием будет очередной символ из входного потока). Среди множества всех состояний конечного автомата можно выделить начальное состояние и множество конечных состояний. Конечный автомат начинает обработку событий всегда из начального состояния и переходит в некоторое конечное состояние по окончании работы. При помощи конечных автоматов большое количество задач может быть описано и решено понятным и красивым способом.

Мы рассмотрим два варианта конечных автоматов: вариант на основе таблицы переходов и автомат в стиле Erlang/OTP (Erlang/OTP – это библиотека языка Erlang для создания программ для телекома, поставляемая вместе с языком) с использованием метапрограммирования. В этом отношении, пример конечного автомата в стиле OTP будет также примером использования метапрограммирования на практике.

Так как язык Erlang, как и другие функциональные языки, не умеет сохранять состояние, то состояние после каждого перехода конечного автомата должны хранить мы – клиенты этого конечного автомата. Для этого мы определим запись **state** и поместим ее в подключаемый файл (**state.hrl**):

```
-record(state, {state_id, state_data, internal_state}).
```

Запись **state** содержит следующие три поля: **state_id** для хранения текущего состояния (идентификатора состояния), **state_data** для хранения данных и **internal_state** для хранения состояния самого конечного автомата (настроек, таких как таблица перехода).

Начнем с создания классического конечного автомата, который использует таблицу перехода. Первым делом объявляем модуль, в котором будет находиться весь наш код (при этом файл модуля должен иметь имя **state_machine.erl**):

```
-module(state_machine).
```

Затем определим запись для хранения состояния самого конечного автомата; нам нужно хранить начальное состояние (поле **init_state**), таблицу перехода (поле **transition_table**) и множество всех возможных состояний (поле **state_set**). Таблица перехода представляет собой словарь, в котором ключом является пара событие–состояние (кортеж), а значением, связанным с данным ключом, является действие, вызываемое, когда конечный автомат, находящийся в состоянии из ключа, обрабатывает событие из ключа. Множество всех возможных состояний нужно для проверки, что мы переходим в существующее состояние.

```
-record(internal_state, {init_state, transition_table = dict:new(),
state_set = sets:new()}).
-include("state.hrl").
```

Следующий шаг – объявление экспортируемых функций: нам необходимы функции для построения конечного автомата, для его инициализации и для обработки событий.

```
-export([build_init/1, build_transition/3, build_terminal/2, start/2,
send/2]).
```

Теперь можно перейти непосредственно к определению функций. Функция **build_init/1** нужна для указания, какое состояние является начальным. Она создает и возвращает экземпляр внутреннего состояния (экземпляр записи **internal_state**) с установленным начальным состоянием (и, что вполне естественно, это состояние добавляется во множество всех состояний).

```
build_init(InitStateId) ->
        #internal_state{init_state = InitStateId, state_set =
sets:add_element(InitStateId, sets:new())}.
```

Функция **build_terminal/3**, пожалуй, наиболее интересна среди функций построения конечного автомата. В этой функции мы, во-первых, проверяем, не был ли в таблицу переходов уже добавлен переход для данного состояния по данному событию. Если такой переход был добавлен, то это ошибка построения, и мы генерируем исключение времени выполнения. Если же такой переход добавлен не был, то мы этот переход в таблицу переходов добавляем, добавляем заданное состояние во множество состояний и возвращаем новое внутреннее состояние с обновленной таблицей переходов и множеством всех состояний.

```
build_transition({EventId, StateId}, Action, InternalState) ->
    Table = InternalState#internal_state.transition_table,
    StateSet = InternalState#internal_state.state_set,
    case dict:is_key({EventId, StateId}, Table) of
        true -> erlang:error(build_error);
```

Сущности снова

```
            false ->
                UpdatedTable = dict:store({EventId, StateId},
Action, Table),
                UpdatedStateSet = sets:add_element(StateId,
StateSet),
                InternalState#internal_state{transition_table =
UpdatedTable, state_set = UpdatedStateSet}
        end.
```

И, наконец, последняя функция для построения конечного автомата – **buld_terminal/2**. Она просто добавляет состояние во множество состояний и возвращает новое внутреннее состояние конечного автомата. В данной версии, конечный автомат никак не отличает конечные состояния от обычных, поэтому данная задача ложится на плечи пользователей нашего конечного автомата.

```
build_terminal(TerminalStateId, InternalState) ->
        StateSet = InternalState#internal_state.state_set,
        InternalState#internal_state{state_set = sets:add_
element(TerminalStateId, StateSet)}.
```

После построения конечного автомата (построения его внутреннего состояния) и перед обработкой событий конечный автомат необходимо инициализировать: задать в качестве текущего состояния начальное и создать экземпляр записи **state** (и вернуть этот экземпляр пользователю). Для этой цели мы определяем функцию **start/2**:

```
start(InitStateData, InternalState) ->
        #state{state_id = InternalState#internal_state.init_state,
state_data = InitStateData, internal_state = InternalState}.
```

Теперь перейдем к функции, являющейся сердцем конечного автомата: это функция обработки сообщений **send/2**. Для данного сообщения и состояния мы ищем в таблице переходов (которая находится во внутреннем состоянии автомата) обработчик. Если таковой найден, то конечный автомат способен обработать данное сообщение в данном состоянии; если нет, то данное событие мы считаем ошибочным и генерируем исключение. Когда обработчик данного события в данном состоянии найден (если найден), управление передается функции **process/3**, в которой и происходит вызов обработчика и формирование нового состояния.

```
send({EventId, EventData}, State) ->
        InternalState = State#state.internal_state,
        Table = InternalState#internal_state.transition_table,
        case dict:find({EventId, State#state.state_id}, Table) of
                {ok, Action} -> process_event({EventId,
EventData}, State, Action);
                error -> erlang:error(bad_event)
        end.
```

Функция **process_event/3** заканчивает обработку события: в ней мы вызываем найденный обработчик, после чего обрабатываем его ответ. Обработка ответа включает следующие шаги: сначала мы проверяем, что функция-обработчик перешла в известное нам состояние (множество всех состояний у нас находится во внутреннем состоянии конечного автомата); после этого мы обновляем состояние конечного автомата и возвращаем его пользователю. Пару слов об интерфейсе обработчиков в таблице переходов: обработчик должен принимать два аргумента (первый аргумент – это кортеж, состоящий из события и данных события; второй аргумент – кортеж, состоящий из состояния и пользовательских данных) и возвращать кортеж, состоящий из нового состояния и обновленных пользовательских данных.

```
process_event({EventId, EventData}, State, Action) ->
        {NewStateId, NewStateData} = Action({EventId,
EventData}, {State#state.state_id, State#state.state_data}),
        InternalState = State#state.internal_state,
        StateSet = InternalState#internal_state.state_set,
        case sets:is_element(NewStateId, StateSet) of
                true -> State#state{state_id = NewStateId,
state_data = NewStateData};
                false -> erlang:error(bad_state)
        end.
```

Вот и весь код конечного автомата, построенного на основе таблицы перехода. Его тестирование мы отложим на чуть более поздний срок, и рассмотрим, как можно создать конечный автомат с использованием других принципов (в стиле Erlang/OTP).

Для использования конечного автомата, созданного выше, необходима фаза настройки, когда задаются начальное состояние, таблица переходов и конечные состояния (функции **state_machine:build_init/1**, **state_machine:build_transition/3** и **state_machine:build_terminal/2**). Понятно, что это не всегда удобно (в примере использования будет видно, насколько это »

Лексический анализ

В информатике, лексический анализ – процесс аналитического разбора входной последовательности символов (например, такой как исходный код на одном из языков программирования) с целью получения на выходе последовательности символов, называемых «токенами» (подобно группировке отдельных букв в слова). Группа символов входной последовательности, идентифицируемая на выходе процесса как токен, называется лексемой. В процессе лексического анализа производится распознавание и выделение лексем из входной последовательности символов.

Как правило, лексический анализ производится с точки зрения определенного формального языка или набора языков. Язык – а точнее, его грамматика – задает определенный набор лексем, которые могут встретиться на входе процесса.

Традиционно принято организовывать процесс лексического анализа, рассматривая входную последовательность символов как поток символов. При такой организации процесс самостоятельно управляет выборкой отдельных символов из входного потока.

Распознавание лексем в контексте грамматики обычно производится путем их идентификации (или классификации) согласно идентификаторам (или классам) токенов, определяемых грамматикой языка. При этом любая последовательность символов входного потока (лексема), которая согласно грамматике не может быть идентифицирована как токен языка, обычно рассматривается как специальный токен-ошибка.

Каждый токен можно представить в виде структуры, содержащей идентификатор токена (или идентификатор класса токена) и, если нужно, последовательность символов лексемы, выделенной из входного потока (строку, число и т.д.).

Цель такой конвертации обычно состоит в том, чтобы подготовить входную последовательность для другой программы – например для грамматического анализатора – и избавить его от определения лексических подробностей в контекстно-свободной грамматике (что привело бы к усложнению грамматики).

Так, например, входная последовательность ab = (1 + c)*d преобразуется в следующий набор токенов:

- Идентификатор ab;
- Операция “=”;
- Символ “(”;
- Константа 1;
- Операция “+”;
- Идентификатор c;
- Символ “)”;
- Операция “*”;
- Идентификатор d.

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

громоздко), особенно когда у нас есть модуль с экспортируемыми функциями, которые реализуют необходимую функциональность и имеют тот же интерфейс, что и обработчики в таблице перехода (в этом случае обработчики являются всего лишь ссылками на такие функции). Поэтому было бы очень удобно, если у нас была возможность не строить таблицу переходов, а просто сообщить конечному автомату, что такая-то функция ответственна за обработку событий в таком-то состоянии. В таких языках, как Java или C#, для этой цели мы могли бы использовать аннотации и атрибуты соответственно. В языке Erlang таких средств нет, но есть другое средство метапрограммирования: BIF **apply/3**. Эта BIF позволяет выполнить функцию по заданным модулю, имени функции и списку аргументов (при этом модуль и имя функции должны быть атомами). Чем же это нам поможет? Пусть на состояния накладывается следующее ограничение: состояние (идентификатор состояния) есть атом, являющийся допустимым именем функции. Тогда мы каждому состоянию можем сопоставить функцию в некотором модуле с таким же именем. Когда придет пора обрабатывать некоторое событие, мы, зная имя функции (оно же идентификатор состояния) и имя модуля, в котором эта функция определена, можем (при помощи BIF **apply/3**) вызвать ее для обработки события. Если функции в модуле с заданным именем нет, то вызов BIF **apply/3** приводит к генерации исключения. Но и, конечно, мы предполагаем, что функции, заданные таким образом, удовлетворяют вышеприведенному интерфейсу: принимают событие и старое состояние и возвращают новое состояние.

Посмотрим, как это реализуется на практике. Первым шагом, как обычно, объявляем модуль:

```
-module(otp_state_machine).
```

Далее, определяем внутреннее состояние нашего конечного автомата: в него в этом варианте реализации входят имя модуля и идентификатор начального состояния (он же – имя функции, которая будет вызвана первой).

```
-record(internal_state, {init_state, module_name}).
-include("state.hrl").
```

Следующий шаг – определение экспортируемых функций: нам нужны функция для создания внутреннего состояния конечного автомата, для его инициализации и для обработки событий.

```
-export([build/2, start/2, send/2]).
```

Перейдем к рассмотрению функций. Функция **build/2** создает внутреннее состояние конечного автомата (в которое входят имя модуля, где определены обработчики и начальное состояние).

```
build(InitStateId, ModuleName) ->
        #internal_state{init_state = InitStateId, module_name =
ModuleName}.
```

Функция **start/2** инициализирует конечный автомат перед обработкой событий. Инициализация заключается в создании состояния конечного автомата (экземпляра записи **state**) и установке начального состояния (из внутреннего состояния конечного автомата) в качестве текущего.

```
start(InitStateData, InternalState) ->
        #state{state_id = InternalState#internal_state.init_state,
state_data = InitStateData, internal_state = InternalState}.
```

Функция **send/2**, пожалуй, самая интересная среди функций данного варианта конечного автомата. Но и она не делает никаких сложных вещей: берет в качестве имени функции обработчика идентификатор текущего состояния, формирует список аргументов и, при помощи BIF **apply/3**, вызывает этот обработчик. При этом предполагается, что обработчик имеет следующий интерфейс: принимает два аргумента (первый аргумент – это кортеж, состоящий из события и данных события, второй аргумент – кортеж, состоящий из состояния и пользовательских данных) и возвращает кортеж, состоящий из нового состояния и обновленных пользовательских данных.

```
send({EventId, EventData}, State) ->
        InternalState = State#state.internal_state,
        Args = [{EventId, EventData}, {State#state.state_id,
State#state.state_data}],
        {NewStateId, NewStateData} =
apply(InternalState#internal_state.module_name, State#state.
state_id, Args),
        State#state{state_id = NewStateId, state_data =
NewStateData}.
```

Мы создали конечные автоматы в двух вариантах: в классическом (с использованием таблицы переходов) и в стиле OTP. Естественное желание – убедиться, что эти автоматы работают, и работают правильно. Для этого решим следующую задачу: реализуем лексический анализатор для чисел (целых и действительных). Скажем пару слов о том, что такое лексический анализ. Это процесс разбора входной последовательности символов с целью распознавания и группировки подпоследовательностей символов в лексемы (для целей дальнейшего анализа). Распознавание и группировка производятся на основе правил, показывающих, какие символы могут появиться после текущего распознанного. Такие правила задаются при помощи регулярных выражений. Соответственно, лексический анализатор – это программа, выполняющая лексический анализ. Почему же для лексического анализа необходимо строить конечный автомат, а не использовать соответствующие регулярные выражения? Ответ достаточно очевиден: мы обрабатываем входную строку посимвольно, пока не найдем подпоследовательность максимальной длины, удовлетворяющую некоторому правилу; при этом часть этой подпоследовательности может этому правилу не удовлетворять. Поэтому обычно строят конечный автомат на основе регулярного выражения и обрабатывают строку посимвольно (каждый символ при этом является событием). Когда конечный автомат входит в состояние, называемое принимающим, это значит, что он нашел подпоследовательность, удовлетворяющую регулярному выражению. Но автомат может продолжить поиск, чтобы найти более длинную подпоследовательность, если это позволяет сделать регулярное выражение. Для чисел регулярное выражение будет таким: **number = sign?(digit)+**, где **sign = [+-]**, **digit = [0-9]**. Построение конечного автомата по этому регулярному выражению – отдельная большая тема, и мы приведем сразу готовый результат.

Для работы нашего лексического анализатора нам необходимо некоторое состояние: для этого мы объявляем запись **state_data** и помещаем ее в отдельный файл (который должен называться **state_data.hrl**). Это состояние содержит следующие атрибуты: обработанную часть строки, распознанную подпоследовательность и флаг, показывающий, находимся ли мы в данный момент в принимающем состоянии.

```
-record(state_data, {pull = "", recognized = "", is_complete =
false}).
```

Затем создадим модуль для функций лексического анализатора.

```
-module(number_lex).
```

После этого подключаем определения записей; т.к. лексический анализатор мы строим на основе конечного автомата, то нам нужно подключить как определение состояния самого лексического анализатора (из файла **state.hrl)**, так и определение данных, специфичных для лексического анализа (из файла **state_data.hrl)**

```
-include("state.hrl").
-include("state_data.hrl").
```

Следующий шаг – объявление экспортируемых функций. Нам нужны функции для фазы инициализации (создание внутреннего

состояния конечного автомата) и для обработки входной строки. Так как мы привели два варианта реализации конечного автомата, то и функций инициализации тоже должно быть две – чтобы была возможность проверить оба варианта.

```
-export([build_lex_fsm/0, build_lex_otpfsm/0, process/3]).
```

Перейдем теперь к рассмотрению функций. Перед использованием лексического анализатора его необходимо инициализировать, то есть инициализировать используемый им конечный автомат. Для инициализации конечного автомата на основе таблицы перехода используется метод **build_lex_fsm/0**:

```
build_lex_fsm() ->
        IntState0 = state_machine:build_init(start),
        IntState1 = state_machine:build_transition({char, start}, fun integer_fsm:start/2, IntState0),
        IntState2 = state_machine:build_transition({char, sign}, fun integer_fsm:sign/2, IntState1),
        IntState3 = state_machine:build_transition({char, digit}, fun integer_fsm:digit/2, IntState2),
        state_machine:build_terminal(unrecognized, IntState3).
```

Для инициализации конечного автомата в стиле OTP используется метод **build_lex_otpfsm/0**:

```
build_lex_otpfsm() -> otp_state_machine:build(start, integer_fsm).
```

Для обработки входной строки используется метод **process/3**. В качестве параметров он принимает входную строку, имя модуля, реализующего конечный автомат, и внутреннее состояние конечного автомата. Имя модуля используется для вызова функций конечного автомата с использованием BIF **apply/3** и позволяет протестировать работу обоих вариантов конечного автомата. Функция **procress/3** передает управление функции **process_impl/3**, которая и выполняет всю реальную работу.

```
process(Source, FSMModule, InternalState) -> process_impl(Source, FSMModule, apply(FSMModule, start, [#state_data{}, InternalState])).
```

И, наконец, мы дошли до функции, которая и выполняет реальную обработку – **process_impl/3**. В этой функции мы посимвольно обрабатываем входную строку. Для этой обработки мы используем конечный автомат, событием для которого является каждый очередной символ. Обработка заканчивается либо когда обработана вся входная строка, либо когда конечный автомат переходит в состояние **unrecognized**. Результатом обработки является кортеж, содержащий распознанную (как целое число) строку и остаток входной строки.

```
process_impl("", _FSMModule, State) ->
    StateData = State#state.state_data,
    {StateData#state_data.recognized, string:sub_string(StateData#state_data.pull, length(StateData#state_data.recognized) + 1)};
process_impl([Char | Rest], FSMModule, State) ->
    case apply(FSMModule, send, [{char, Char}, State]) of
        #state{state_id = unrecognized, state_data = NewStateData} ->
            {NewStateData#state_data.recognized, string:sub_string(NewStateData#state_data.pull, length(NewStateData#state_data.recognized) + 1) ++ [Char] ++ Rest};
        NewState -> process_impl(Rest, FSMModule, NewState)
    end.
```

Для полноты картины приведем код модуля **integer_fsm**, использующийся для построения конечных автоматов. В подробности вдаваться не будем (это отдельная большая тема) – отметим следующее: все функции (являющиеся обработчиками) в этом модуле имеют варианты, которое ведут в состояние **unrecognized**, означающее, что очередной символ не может быть распознан.

```
-module(integer_fsm).
-include("state_data.hrl").
-export([start/2, sign/2, digit/2]).
start({char, Char}, {start, _StateData}) when Char == $+; Char == $- ->
        {sign, #state_data{pull = [Char]}};
start({char, Char}, {start, _StateData}) when Char >= $0, Char =< $9 ->
        {digit, #state_data{pull = [Char], recognized = [Char], is_complete = true}};
start({char, _Char}, {start, _StateData}) ->
        {unrecognized, #state_data{}}.
sign({char, Char}, {sign, StateData}) when Char >= $0, Char =< $9 ->
        {digit, #state_data{pull = StateData#state_data.pull ++ [Char], recognized = StateData#state_data.pull ++ [Char], is_complete = true}};
sign({char, _Char}, {sign, StateData}) ->
        {unrecognized, StateData}.
digit({char, Char}, {digit, StateData}) when Char >= $0, Char =< $9 ->
        {digit, #state_data{pull = StateData#state_data.pull ++ [Char], recognized = StateData#state_data.pull ++ [Char], is_complete = true}};
digit({char, _Char}, {digit, StateData}) ->
        {unrecognized, StateData}.
```

Нам осталось совсем немного перед проверкой: откомпилировать все необходимые модули (**state_machine**, **otp_state_machine**, **integer_fsm**, **number_lex**). После этого необходимо создать внутреннее состояние тестируемых конечных автоматов: **IntState1 = number_lex: build_lex_fsm()** для конечного автомата на основе таблицы переходов и **IntState2 = number_lex: build_lex_otpfsm()** для конечного автомата в стиле OTP. Теперь все готово к лексическому анализу. Вызов **number_lex:process("+666", state_machine, IntState1)** вернет **{"+666",[]}**; это означает, что вся строка распознана как число. Вызов **number_lex:process("-333abc", state_machine, IntState1)** вернет **{"+333", "abc"}**; это означает, что лишь часть строки распознана как число. Аналогично для конечного автомата в стиле OTP: вызов **number_lex:process("+666", otp_state_machine, IntState2)** вернет **{"+666",[]}**, вызов **number_lex:process("-333abc", otp_state_machine, IntState2)** вернет **{"+333", "abc"}**.

Сегодня мы увидели, что работа с состоянием и сохранение состояния между вызовами – это очень простая задача. Единственное отличие от императивных языков программирования заключается в том, что ответственность за сохранение состояния лежит целиком на вызывающей стороне, тогда как в императивных языках программирования эта ответственность может возлагаться как на вызывающую, так и на вызываемую сторону. На этом мы рассмотрение практических задач не заканчиваем – продолжение в следующей статье. LXF

Лексический анализатор

Лексический анализатор – это программа или часть программы, выполняющая лексический анализ. Лексический анализатор обычно работает в две стадии: сканирование и оценка.

На первой стадии, сканировании, лексический анализатор обычно реализуется в виде конечного автомата, определяемого регулярными выражениями. В нем кодируется информация о последовательностях символов, которые могут встречаться в токенах. Например, токен «целое число» может содержать любую последовательность десятичных цифр. Нередко первый непробельный символ используется для определения типа следующего токена, после чего входные символы обрабатываются один за другим, пока не встретится символ, не входящий во множество допустимых символов для данного токена. В некоторых языках правила разбора лексем несколько сложнее и требуют возвратов назад по читаемой последовательности.

Полученный таким образом токен содержит необработанный исходный текст (строку). Чтобы получить токен со значением, соответствующим типу (напр. целое или дробное число), выполняется оценка этой строки – проход по символам и вычисление значения.

Токен с типом и соответственно подготовленным значением передается на вход синтаксического анализатора.