Erlang: Язык для

Андрей Ушаков знакомит вас с концепцией функционального программирования и одним из многочисленных языков на основе этой концепции.

Наш эксперт

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

Функциональное программирование звучит как академическая дисциплина в ВУЗе: нечто заумное и оторванное от реальной жизни. Еще бы не сложиться такому впечатлению – стоит только посмотреть на языки программирования из мейнстрима: C, C++, Java, C#, ... И какой из этих языков программирования функциональный? Но стоит только присмотреться повнимательней, как вы увидите элементы функционального программирования и в этих языках (по крайней мере, в большинстве). Так, например, в функциональном программировании есть концепция функций высшего порядка (более подробно см. далее) – это функции, которые принимают функции в качестве параметра, либо возвращаемое значение есть функция. Но подобные функции высшего порядка у нас встречаются и в императивных языках: так, в программах на C, C++ мы можем принимать либо возвращать указатель на функцию; в программах на C++, Java, C# мы можем принимать либо возвращать интерфейс с одним методом, и т.д.

> «Функциональный подход более абстрактен, чем императивный.»

Более того, многие императивные языки вводят элементы, поддерживающие функциональное программирование: например, в C# для подобной поддержки были введены делегаты и лямбды. В настоящее время появились и развиваются функциональные языки, которые можно отнести к мейнстриму: F#, Scala, ... Данная статья посвящена обзору одного из таких функциональных языков – Erlang, который можно описать следующей формулой: функциональный язык + процессы.

Элементы программирования

Функциональное программирование – очень большая тема, которой посвящно много книг. Давайте коротко рассмотрим особенности, преимущества и недостатки функциональных языков (желающие изучить эту тему более подробно могут обратиться к соответствующей литературе). Что же отличает функциональное программирование от императивного (от программирования, например, на языке C)? Главным образом то, как трактуется процесс вычисления: вычисление есть вычисление значений функций в математическом понимании последних. А это означает следующее: значение функции определяется только значениями переданных ей аргументов, и нет никакого внешнего состояния, от которого может зависеть значение функции.

Этот подход сильно отличается от подхода императивного программирования, в котором вычисление трактуется как последовательность изменений состояния программы. И, в отличие от функционального подхода, вычисление в императивном программировании может иметь побочные эффекты, то есть значение функции, например, зависит уже не только от значения аргументов, но и от внешнего состояния. Понятно, что если бы в функциональных языках были только вычисления, не зависящие от внешнего состояния, они вряд ли бы когда-либо вышли за рамки академического интереса. Пользовательский ввод и вывод, работа с файлами, с базами данных и т.п. – все это вычисления, зависящие от внешнего состояния (и с некоторыми побочными эффектами). И во всех функциональных языках есть те или иные средства для работы с такими вычислениями (так, в Erlang просто не все функции являются независимыми от внешнего состояния).

В чем же плюсы функционального подхода? В первую очередь, в повышении надежности кода: если нет внешнего состояния и побочных эффектов, мы легко можем предсказать результат работы программы. Отсутствие внешнего состояния и побочных эффектов позволяет нам писать более простые и надежные модульные тесты, что опять же повышает надежность кода. В многопоточной среде вычисления, которые не зависят от внешнего состояния и не имеют побочных эффектов, легко могут быть распараллелены.

Теперь поговорим о минусах. Связаны они с тем, что функциональный подход более абстрактен, чем императивный, и реализации функциональных языков вынуждены достаточно много делать «под капотом», что приводит к дополнительным расходам ресурсов компьютера, в некоторых случаях достаточно существенным. В первую очередь это необходимость наличия сборки мусора, во вторую – поддержка «ленивых» вычислений.

Стоит упомянуть еще пару концепций, которые широко используются в функциональном программировании: функции высших

процессов

порядков и анонимные функции (лямбда-выражения). Функции высшего порядка – это функции, которые принимают в качестве аргументов одну или несколько функций, либо возвращаемое значение есть функция. Так, функция, вычисляющая значение определенного интеграла, является примером функции высшего порядка, принимающей в качестве аргумента другую функцию; а функция, возвращающая функцию – решение дифференциального уравнения является примером функции высшего порядка, возвращаемое значение которой есть функция. Анонимная функция – это функция, определенная только в том месте, где она используется. Поэтому, соответственно, анонимная функция обходится без имени и не участвует в процедуре поиска функции по имени в момент компиляции.

Особенности языка

Рассмотрим некоторые элементы функционального программирования применительно к языку Erlang.

Переменные являются настолько фундаментальными объектами, что они присутствуют, пожалуй, во всех языках. И Erlang в этом плане не исключение – переменные в нем есть. Erlang является языком со строгой динамической типизацией. Поэтому при объявлении переменной ее тип не задается (в отличие, например, от C), а динамически выводится во время выполнения. При объявлении переменной очень важно, чтобы ее имя начиналось с заглавной буквы – это ее отличает от других объектов. Например, **SomeVariable** – это переменная. Значение переменной может быть присвоено один и только один раз, и после присвоения ее значение не может быть изменено. Таким образом, мы всегда знаем, в каком месте программы переменная получила свое значение, и это очень сильно упрощает отладку.

Присваивая значение переменной, мы используем оператор = (например, **SomeVariable = 1**). Но этот оператор не является оператором присвоения (что очень неожиданно для человека, имеющего опыт программирования на императивных языках вроде C): это оператор соответствия [pattern matching]. При вычислении выражения **Left = Right** происходит проверка, соответствуют ли **Left** и **Right** друг другу. **Left** и **Right** будут соответствовать друг другу в следующих случаях:

- **Left** и **Right** имеют одно и тоже значение одного и того же типа. Например, **1 = 1**.
- Объявляется переменная **Left**. В этом случае, т. к. переменная **Left** никакого значения не имеет, ей присваивается значение **Right**.

Операция соответствия появляется не только при использовании оператора =. В любом месте программы, где возможно появление операндов **Left** и **Right**, будет происходить проверка на их соответствие. Например, пусть у нас есть следующее объявление функции (более подробно о функциях см. далее):

```
f(0) -> true;
f(Number) -> f(Number-1).
```

В первой строке содержится конкретизирующий вариант функции **f()**. При вызове функции будет происходить поиск в порядке объявления вариантов. При этом заданное значение параметра будет проверяться на соответствие значению, определенному в варианте. Так, например, вызов **f(0)** приведет к вызову первого варианта, а вызов **f(1)** – к вызову второго, причем параметру **Number** будет присвоено значение **1**.

Что будет, если при проверке двух операндов **Left** и **Right** выявится, что они не соответствуют друг другу? Если мы использовали оператор =, то получим ошибку времени выполнения. Если это было, например, при просмотре очередного варианта объявления функции, то поиск перейдет к следующему варианту.

Типы данных

Давайте поговорим теперь о типах данных Erlang. Наиболее очевидные типы – это целые и действительные числа (было бы странно, если бы их не было в языке программирования). Другой, чуть менее очевидный тип данных – это атомы. Аналог атомов – константы, но, в отличие от констант, атомы не содержат связанных с ними значений. Объявление атомов всегда начинается с маленькой буквы: например, **atom_sample**. »

Другие функциональные языки

- **Scala** – мультипарадигмальный язык, спроектированный кратким и типобезопасным для простого и быстрого программирования. В нем сочетаются возможности функционального и объектно-ориентированного подходов. Основной целью разработки был язык, обладающий хорошей поддержкой компонентного ПО.
- **Haskell** – стандартизованный чистый функциональный язык программирования общего назначения. Является одним из самых распространенных языков с поддержкой отложенных вычислений.
- **Lisp** – семейство языков программирования, данные в которых представляются системами линейных списков символов. Язык является функциональным, но многие поздние версии обладают также чертами императивности; к тому же, имея полноценные средства символьной обработки, становится возможным реализовать объектно-ориентированность. Наиболее популярные в наши дни диалекты языка Lisp – это Common Lisp и Scheme.
- **ML** – семейство строгих языков функционального программирования с развитой полиморфной системой типов и параметризуемыми модулями. Это не чисто функциональные языки, так как включают и императивные инструкции. ML преподается во многих западных университетах (в некоторых даже как первый язык программирования).
- **Miranda** – функциональный язык программирования, имеющий строгую полиморфную систему типов и поддерживающий типы данных пользователя. Как и язык ML, преподается во многих университетах.
- **OCaml** – современный объектно-ориентированный язык функционального программирования общего назначения, который был разработан с учетом безопасности исполнения и надежности программ. Язык OCaml поддерживает функциональную, императивную и объектно-ориентированную парадигмы программирования.
- **F#** – функциональный язык программирования общего назначения. Структура F# во многом схожа со структурой Ocaml, с той лишь разницей, что язык F# реализован поверх библиотек и среды исполнения .NET.

» **Пропустили номер?** Узнайте на с. 107, как получить его прямо сейчас.

Следующие два типа данных являются фундаментальными в большинстве (если не во всех) функциональных языков программирования: это кортежи и списки. Кортеж [Tuple] – это контейнер для хранения объектов разных типов. Его аналогом являются, например, структуры языка C, но, в отличие от структур, поля кортежа не имеют имени. Доступ к ним можно осуществить либо по индексу поля, либо с использованием операции соответствия. Например, мы объявляем следующий кортеж: **TupleSample = {1, atom_sample}**. Тогда доступ к первому полю осуществляется следующим образом: **element(1, TupleSample)**, а для доступа к первому полю через операцию соответствия мы используем следующий синтаксис: **{First, Second} = TupleSample**, после чего переменная **First** содержит значение из первого поля кортежа **TupleSample**. Список – это контейнер для хранения объектов одного типа. Аналогами списка являются, например, массивы языка C, класс ArrayList языка Java. Список объявляется следующим образом: **[Element1, Element2, ...]**, например, **[1, 2, 3]**. Существует специальный синтаксис для доступа к головному и хвостовому элементам списка через операцию соответствия: **[Head | Other]** и **[Other | Tail]**, где **Other** – оставшаяся часть списка. Для создания списков также существует специальная методика, аналогов которой в императивных языках программирования нет – List Comprehensions. Не углубляясь во все возможности этого мощного механизма, просто приведем пару примеров.

» Генерация списка квадратов чисел от 1 до 10:

```
[N*N | | N <- lists:seq(1, 10)].
```

» Генерация списка квадратов чисел от 1 до 10 только для четных чисел:

```
[N*N | | N <- lists:seq(1, 10), N rem 2 == 0].
```

А как же строки, спросите вы. Неужели в Erlang сделали шаг назад в отношении работы со строковыми данными? В Erlang отдельного типа данных для строк нет: строки представляются в виде списков целых чисел. Но существует «синтаксический сахар» для работы со строками, позволяющий записывать строковые данные в привычном виде. Например, литерал “123” синтаксически корректен и соответствует значению [49, 50, 51]. К сожалению, в привычном для нас виде могут быть записаны строки только в кодировке Latin-1 (ISO-8859-1). Поэтому в других кодировках необходимо работать напрямую с данными в списках.

Помимо перечисленных здесь типов данных, в языке Erlang существуют и другие типы. Наиболее интересный и значимый из них – это тип двоичных данных. Но углубляться в эти типы данных мы не будем.

Функции

Пожалуй, последнее и самое важное, что следует рассказать об Erlang – это объявление функций. В самом простом случае, функция объявляется обычным образом: имя функции (оно должно начинаться с маленькой буквы), после которого следует список аргументов (через запятую) в скобках (для объявления аргументов действуют те же правила именования, что и для переменных), а за ним следует тело функции. Например, объявление функции, вычисляющей квадрат своего аргумента, будет выглядеть так:

```
square(Number) -> Number*Number.
```

Функции (сигнатуры функций, если быть более точным) различаются не только по имени, но и по числу аргументов, которые они принимают (арность функции). Поэтому мы можем иметь две или более функции с одинаковым именем, но разным числом аргументов.

> «Какой вариант будет подходящим, зависит от аргументов.»

Но на этом возможности определения функции не заканчиваются. Мы можем определить два и более варианта одной и той же функции. При вызове функции будет происходить последовательный просмотр всех объявленных вариантов (в порядке их объявления) и поиск первого подходящего. Какой вариант будет подходящим, зависит от переданных аргументов. Если же не будет найдено ни одного подходящего варианта, мы получим ошибку времени выполнения.

Благодаря каким механизмам варианты функции будут отличаться друг от друга? Таких механизмов два: операция соответствия и охранные выражения [guards]. Механизм соответствия действует следующим образом: при объявлении варианта функции для одного или нескольких аргументов мы вместо самих аргументов задаем какие-либо значения. И при поиске подходящего варианта эти значения будут сравниваться со значениями аргументов, а если между ними будет соответствие, то будет выбран этот вариант функции. Например, мы можем объявить функцию для вычисления факториала следующим образом:

```
factorial(0) -> 1;
factorial(1) -> 1;
factorial(N) -> N*factorial(N-1).
```

Другой механизм – это охранные выражения. Охранное выражение – это логическое выражение с аргументами функции. Если при поиске подходящего варианта охранное выражение истинно, то выбирается этот вариант. Охранное выражение объявляется при помощи ключевого слова **when**. Например, функцию для вычисления факториала мы можем объявить при помощи охранного выражения следующим образом:

```
factorial(N) when N == 0 or N == 1 -> 1;
factorial(N) -> N*factorial(N-1).
```

Понятно, что при объявлении функции у нас могут быть варианты, использующие соответствие, или использующие охранные выражения, или использующие и то, и другое.

С функциями связано еще одно фундаментальное понятие – хвостовая рекурсия. Если при выполнении функции последней операцией будет вызов самой себя (с теми же либо с другими аргументами), то этот вызов будет не рекурсивным (этот вызов будет представлять собой простой переход на начало функции). Если же вызов самой себя будет располагаться где-то внутри тела (не будет последней операцией), то это будет обычный рекурсивный вызов. Так, например, в функции **factorial()** будет хвостовая рекурсия, а в функции **factorial2()** – нет:

```
factorial(N) when N == 0 or N == 1 -> 1;
factorial(N) -> N*factorial(N-1).
factorial2(N) when N == 0 or N == 1 -> 1;
factorial2(N) →
        N*factorial(N-1),
        log(N).
```

Хвостовая рекурсия важна тем, что она позволяет организовать цикл (конечный или бесконечный) и не приводит к заполнению стека (которое возникает при обычном рекурсивном вызове).

Пример

Хватит теории – давайте решим простую задачу для демонстрации некоторых концепций, изложенных выше. В качестве примера возьмем задачу номер 17 с проекта Project Euler (http://projecteuler.net/index.php?section=problems&id=17). Условие этой задачи звучит следующим образом:

Если числа от 1 до 5 записать английскими словами (one, two, three, four, five), то будет использовано 3 + 3 + 5 + 4 + 4 = 19 букв. Если все число от 1 до 1000 включительно записать английскими словами то сколько будет использовано букв?

Примечание: Пробелы и дефисы при подсчете не учитывать. Например, 342 (three hundred and forty-two) содержит 23 буквы, 115 (one hundred and fifteen) – 20 букв. Использование союза "and" соответствует британскому варианту.

Нашей задачей будет написать программу, которая считает количество букв, необходимое для записи английскими словами чисел от 1 до N, при условии, что N не превышает 1000. По правилам британского варианта английского языка, между сотнями и десятками (если таковые есть) ставится артикль "and" (например, числу 115 соответствует строка one hundred and fifteen). Поэтому для удобства мы разделим обработку десятков и сотен в нашей программе. Метод **less_hundred/4** обрабатывает числа, которые меньше 100. В этом методе мы подсчитываем количество символов, необходимых для записи числа, меньшего 100, английскими словами:

```
less_hundred(Number, _, _, _) when Number >= 100 ->
erlang:error(badarg);
less_hundred(Number, From0To9, _, _) when Number =< 9 ->
        length(lists:nth(Number+1, From0To9));
less_hundred(Number, _, From10To19, _) when (Number > 9) and
(Number < 20) ->
        length(lists:nth(Number-9, From10To19));
less_hundred(Number, From0To9, _, OtherTens) ->
        length(lists:nth((Number div 10)-1, OtherTens)) +
length(lists:nth((Number rem 10)+1, From0To9)).
```

Обратите внимание, как мы определяем несколько вариантов функции **less_hundred/4** при помощи охранных выражений.

Следующий шаг – обработка чисел, не превышающих 1000. Для этого служит метод **parse_number/4**:

```
parse_number(1000, _, _, _) -> length("one" ++ "thousand");
parse_number(Number, From0To9, From10To19, OtherTens) when
Number >= 100 ->
LessHundred = less_hundred(Number rem 100, From0To9,
From10To19, OtherTens),
        if
                LessHundred == 0 ->
length(lists:nth((Number div 100)+1, From0To9)) +
length("hundred");
                LessHundred /= 0 -> length(lists:nth((Number
div 100)+1, From0To9)) + length("hundred" ++ "and") +
LessHundred
        end;
parse_number(Number, From0To9, From10To19, OtherTens) ->
        less_hundred(Number rem 100, From0To9, From10To19,
OtherTens).
```

Обратите внимание, что при определении нескольких вариантов функции **parse_number/4** мы используем как механизм соответствия, так и охранные выражения. Следует сказать, что в данном методе мы применяем конструкцию, которую в обзоре не обсуждали – конструкцию **if**. Эта конструкция состоит из последовательности пар выражений (разделенных оператором **->**) и возвращает некоторое значение. Возвращаемое значение определяется следующим образом: последовательно просматриваются пары выражений, в каждой паре вычисляется значение первого выражения, и если значение первого выражения равно **true**, то значение второго выражения становится возвращаемым (либо будет брошена ошибка времени выполнения, если поиск завершится не успешно).

Осталось сделать совсем немного – обработать все числа от 1 до некоторого максимально числа (не превышающего 1000).

```
parse_numbers(MaxNumber, From0To9, From10To19, OtherTens)
->
        parse_numbers(MaxNumber, 1, 0, From0To9,
From10To19, OtherTens).

parse_numbers(MaxNumber, CurrentNumber, LetterCount, _, _, _)
        when CurrentNumber > MaxNumber -> LetterCount;
parse_numbers(MaxNumber, CurrentNumber, LetterCount,
From0To9, From10To19, OtherTens) ->
        parse_numbers(MaxNumber, CurrentNumber +
1, LetterCount + parse_number(CurrentNumber, From0To9,
From10To19, OtherTens), From0To9, From10To19, OtherTens).
```

Для такой обработки мы вводим две функции: **parse_number/4** и **parse_number/6**. Эти функции являются разными за счет того, что у них разное количество аргументов (разная арность) – это вариант перегрузки функций в Erlang. Про эти функции следует заметить следующее: **parse_number/4** является интерфейсной функцией к **parse_number/6**, а **parse_number/6**, соответственно, функцией реализации. Это удобный и широко применяемый в Erlang подход, когда функция с меньшим числом аргументов объявляется для удобства использования, а функция с тем же самым именем и большим числом аргументов выполняет всю работу (подобные примеры интерфейсных объектов и объектов реализаций можно легко найти в императивных языках).

Ну и, наконец, экспортируемая функция, для взаимодействия с нашей функциональностью (**solve/1**):

```
solve(MaxNumber) when MaxNumber > 1000 ->
erlang:error(badarg);
solve(MaxNumber) ->
        From0To9 = ["", "one", "two", "three", "four", "five",
"six", "seven", "eight", "nine"],
        From10To19 = ["ten", "eleven", "twelve", "thirteen",
"fourteen", "fifteen", "sixteen", "seventeen", "eighteen", "nineteen"],
        OtherTens = ["twenty", "thirty", "forty", "fifty", "sixty",
"seventy", "eighty", "ninety"],
        parse_numbers(MaxNumber, From0To9, From10To19,
OtherTens).
```

Осталось только привести объявления модуля и экспортируемых функций:

```
-module(problem_017).
-export([solve/1]).
```

Вот и все решение задачи. Запускаем в консоли среду выполнения (запуском исполняемого файла **erl**; надеюсь, пути у вас настроены), в консоли Erlang запускаем сначала компиляцию **c(problem_017).**, а потом и выполнение нашей программы **problem_017:solve(1000)**, и получаем следующий результат: 21124. Если зайти на сайт проекта Project Euler, в раздел задачи номер 17, то можно убедиться, что это правильный ответ (http://projecteuler.net/index.php?section=problems&id=17).

О многозадачности и взаимодействии между процессами мы поговорим в следующей части этого учебника. LXF

Полезные ссылки и книги

- http://www.erlang.org/ – главный сайт (с документацией и исходным кодом среды).
- http://www.trapexit.org/ – сайт Erlang-сообщества (форум, вики, решения, учебные пособия, справочные материалы).
- http://erlanger.ru/ – сайт русского Erlang-сообщества.
- http://groups.google.com/group/erlang-russian – русское Erlang-сообщество на Google.
- http://www.tryerlang.org/ – онлайн-интерпретатор Erlang.

- Martin Logan, Eric Merritt, and Richard Carlsson "*Erlang and OTP in Action*".
- Francesco Cesarini, Simon Thompson "*Erlang Programming A Concurrent Approach to Software Development*".
- Joe Armstrong "*Programming Erlang: Software for a Concurrent World*".