Erlang: Сущности

Андрей Ушаков продолжает рассказ о программировании на Erlang.

Наш эксперт

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

Мы продолжаем обзор базовых сущностей языка **Erlang**. В предыдущем номере (**LXF145**) мы рассмотрели, что представляют собой функции. В этой статье мы поговорим о другой, не менее важной сущности – о кортежах (и основанных на них записях). Кортежи очень важны по той причине, что это основной строительный блок для структур данных (наряду со списками) во всех функциональных языках программирования, в том числе и в **Erlang**. Поэтому практически в любой программе вы столкнетесь с кортежами либо с основанными на них записями (то же самое можно сказать и про другие базовые сущности: функции и списки). Давайте рассмотрим их более подробно.

Кортеж [Tuple] – это контейнер для разнородных данных, т. е. для данных, обработка которых будет происходить по-разному. При этом не важно, имеют ли элементы кортежа один и тот же или разные типы. В этом кортеж принципиально отличается от списка: предполагается, что в списке все элементы будут обработаны одним способом, даже если у них разный тип (разговор о списках будет в следующей части учебника). Кортеж напоминает структуры языка C, за одним исключением: если доступ к данным в структуре осуществляется по имени, то доступ к данным в кортеже осуществляется по индексу. С другой стороны, упомянутый выше список напоминает вектор из стандартной библиотеки C++ (за одним исключением: в **Erlang** список неизменяемый, в отличие от вектора).

Как объявить переменную типа кортеж? Как мы помним (**LXF143**), **Erlang** – это язык со строгой динамической типизацией. Это означает, что тип переменной определяется в момент ее инициализации. Инициализатор для кортежа выглядит очень просто: внутри фигурных скобок “**{**” и “**}**” мы через запятую перечисляем значения элементов кортежа. Например, мы объявляем переменную типа кортеж с ее инициализацией следующим образом:

```
TupleVar = {1, abc}.
```

Размером кортежа называют количество элементов кортежа; в приведенном выше примере размер кортежа равен 2. Минимальный возможный размер кортежа – 0 (и это будет пустой кортеж). Максимальный возможный размер кортежа обусловлен только системными ограничениями и составляет 67 108 863 элементов.

После объявления кортежа, мы, скорее всего, захотим иметь доступ к отдельным элементам кортежа. Есть несколько способов, как это сделать, и один из них – использовать операцию соответствия шаблону [pattern matching]. Для этого мы создаем шаблон (очень похожий на инициализатор), который выглядит как список неинициализированных переменных либо конкретных значений, перечисленных через запятую, расположенный внутри фигурных скобок “{” и “}”. После этого шаблон сопоставляется кортежу (при помощи оператора соответствия шаблону “=”, причем шаблон стоит слева, а кортеж справа; см. **LXF143**). Кортеж будет соответствовать шаблону при выполнении следующих условий:

- Размеры кортежа и шаблона совпадают.
- Конкретные значения в шаблоне и значения элементов кортежа, стоящих на одной позиции, совпадают.

При этом неинициализированные переменные будут содержать значения элементов кортежа, стоящих в той же позиции. Вместо неинициализированной переменной может стоять специальный символ “_”, означающий, что значение элемента в данной позиции нас не интересует. Давайте рассмотрим несколько примеров, для пояснения этой далеко не самой простой операции. В следующих двух примерах операция соответствия шаблону выполняется успешно:

```
{A1, _ A2} = {1, abc, 2}.
{B1, abc, B2, xyz} = {1, abc, 2, xyz}.
```

В следующих двух примерах операция соответствия не выполняется, и все заканчивается ошибкой времени выполнения:

```
{C1} = {1, abc, 2}.
{D1, xyz, D2, abc} = {1, abc, 2, xyz}.
```

Для работы с кортежами, помимо операции соответствия шаблону, существует набор функций (API). Этот набор функций достаточно невелик (в отличие от набора функций для работы со списками); все функции определены в модуле **erlang**, и часть из них является встроенными – BIF [built-in functions]. Давайте рассмотрим их более подробно.

Выше уже упоминалось, что количество элементов в кортеже – это его размер. Единственный способ узнать размер кортежа – это воспользоваться одной из BIF: **size/1** или **tuple_size/1**. Вся разница между этими методами в том, что **tuple_size/1** работает только с кортежами, а **size/1** – с кортежами и двоичными дан-

языка. Кортежи

ными (рассказ про которые будет в одной из следующих частей). Для доступа к отдельным элементам кортежа, помимо операции соответствия шаблону, существует следующая BIF: **element(Index, Tuple)** (или **element/2**), где **Index** – индекс элемента, **Tuple** – кортеж. Данная BIF позволяет осуществить доступ к конкретному элементу кортежа (по его индексу), не используя шаблон, что особенно удобно в тех случаях, когда размер кортежа большой или может меняться.

Все значения в языке **Erlang** (согласно концепции функциональных языков) неизменяемы; применительно к кортежам это означает, что если мы хотим изменить какие-либо элементы кортежа, то должны создать кортеж заново. Это не очень удобно, если мы хотим изменить всего один (или несколько, в случае, когда размер кортежа достаточно большой) элемент. В этом случае удобно применять следующую BIF: **setelement(Index, OldTuple, NewValue)** (или **setelement/3**), где **Index** – индекс изменяемого элемента, **OldTuple** – исходный кортеж, **NewValue** – новое значение элемента. Данная BIF возвращает новый кортеж (копию кортежа **OldTuple**), у которого элемент с индексом **Index** установлен в значение **NewValue**, а значения остальных элементов совпадают с соответствующими элементами кортежа **OldTuple**.

> «Для работы с кортежами существует набор функций.»

Создание кортежа при помощи инициализатора не всегда удобно, особенно когда размер кортежа большой и мы хотим для некоторых (или для всех) элементов задать значения по умолчанию. В этом случае нам приходят на помощь следующие функции (следует особо заметить, что эти функции – не BIF): **make_tuple(Size, InitialValue)** (или **make_tuple/2**) и **make_tuple(Size, Default, InitList)** (или **make_tuple/3**). Функция **make_tuple/2** создает кортеж размером **Size**, все элементы которого имеют значение **InitialValue**. Функция **make_tuple/3** более хитрая: она создает кортеж размером **Size** и заполняет его элементы в соответствии со списком инициализации **InitList**. Список **InitList** является списком пар позиция–значение (каждая пара – кортеж). При вызове функции **make_tuple/3** те элементы кортежа, позиции которых есть в этом списке, устанавливаются в соответствующие значения; а элементы, позиций которых нет в списке **InitList**, принимают значение **Default**. Давайте приведем пример. Так, вызов

```
erlang:make_tuple(2, abc).
```

создает кортеж **{abc, abc}**, а вызов

```
erlang:make_tuple(3, abc, [{1, xyz}, {3, uvw}]).
```

создает кортеж **{xyz, abc, uvw}**. У нас осталась еще пара BIF: **tuple_to_list/1** и **is_tuple/1**. Первая (**tuple_to_list/1**) позволяет преобразовать кортеж в список, вторая (**is_tuple/1**) – проверить, является ли некоторое значение кортежем. Ну и напоследок про функции: **size/1**, **tuple_size/1**, **element/2** и **is_tuple/1** можно применять в охранных выражениях (см. **LXF145**).

Главный недостаток кортежей в том, что доступ к их элементам осуществляется по индексу. Очень легко забыть, какой элемент в какой позиции находится, и перепутать несколько элементов. Это приведет к ошибке в логике работы с данными, понять причины которой очень сложно. Если кортеж используется только внутри одного модуля, то правильность его использования можно отследить; но если кортеж используется как входной параметр или возвращаемое значение экспортируемой функции, то отследить правильность его использования становится невозможно. И остается только одно: задокументировать структуру кортежа и молиться суровым северным богам, чтобы эту документацию все-таки прочитали и использовали кортеж в соответствии с ней. Для решения этих проблем в языке **Erlang** были введены записи – контейнеры для разнородных данных, доступ к элементам которых осуществляется по имени.

Перед использованием записи следует ее определить. Определение записи выглядит следующим образом:

```
-record(Name, {Field1 [= Value1], ... FieldN [= ValueN]}).
```

Здесь **Name**, **Field1**, **FieldN** – это имена записи и полей соответственно (имена являются атомами – см. **LXF143**). Каждое поле может иметь значение по умолчанию (значения **Value1**, **ValueN**); если значение по умолчанию не задано, таковым становится атом **undefined**. Для совместного использования записи в нескольких модулях ее определение удобно вынести во внешний подключаемый файл (файл с расширением **.hrl** и подключаемый директивой **-include**). Возникает вполне логичный вопрос: как внутри устроены записи, если мы говорим про них в рамках статьи о кортежах? Ответ достаточно очевиден: записи являются лишь «синтаксическим сахаром» компилятора **Erlang** и являются на самом деле кортежами вида **{Name, Field1Value, ... FieldNValue}**. Следует сказать, что, несмотря на наличие специального синтаксиса, с записями можно работать как с обычными кортежами.

Следующий шаг – это создание записи. Запись создается точно так же, как и кортеж: при помощи инициализатора. Инициализатор для записи имеет следующий вид:

```
#RecordName{Field1=Expr1, ..., FieldK=ExprK}.
```

где **RecordName**, **Field1**, **FieldK** – имена записей и полей. Поля в инициализаторе можно задавать в любом порядке, и любое поле в инициализаторе можно пропустить: тогда оно получит значение по умолчанию. Если мы хотим, чтобы все не упомянутые в инициализаторе поля имели одно и то же конкретное значение (**DefaultExpr**), то это можно сделать следующим образом: »

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

```
#RecordName{Field1=Expr1, ..., FieldK=ExprK, _=DefaultExpr}.
```

Возникает вполне логичный вопрос: как получить доступ к конкретному полю в записи? Для этого есть немного неочевидный синтаксис (не очевидный с точки зрения таких языков, как C):

```
RecordExpr#RecordName.Field.
```

Что интересно, простое выражение **#RecordName.Field** дает позицию поля в записи (точнее, позицию поля в кортеже, представляющем запись).

Следующая важная операция, которая может нам потребоваться – это изменение существующего экземпляра записи. Мы помним, что под капотом записи – это кортежи, поэтому изменение означает создание копии записи (кортежа), в которой изменены одно или несколько полей по сравнению с исходной записью. Операция изменения выглядит как инициализатор, примененный к экземпляру записи, в котором задаются только изменяемые поля:

```
RecordExpr#RecordName{Field1=Expr1,...,FieldK=ExprK}.
```

И последнее развлечение с синтаксисом записей: операция соответствия шаблону. Здесь (так же, как и в случае с кортежами, что неудивительно), мы используем шаблон, который имеет тот же вид, что и инициализатор (только он стоит слева от оператора соответствия). В шаблоне (см. выше) **Expr1**, **ExprK** могут быть как конкретными значениями, так и неинициализированными переменными. Алгоритм проверки на соответствие точно такой же, как и в случае кортежа (что опять же неудивительно). Но есть одно отличие: в шаблоне для кортежа мы вынуждены перечислить все поля (того кортежа, который стоит справа в операции соответствия шаблону), а в случае записи – только те, которые интересуют нас (при этом не перечисленные поля никакой роли в операции соответствия шаблону не играют).

> «Отсутствие сокрытия данных не является фатальной вещью.»

Чтобы не запутаться в синтаксисе операций с записями, давайте рассмотрим несколько примеров. Первый шаг, который мы должны сделать, это определить запись:

```
-record(demo, {left = "", middle = null, right}).
```

В данном случае мы определяем запись с именем **record**, которая содержит три поля: поле **left** со значением по умолчанию **""**, поле **middle** со значением по умолчанию **null** и поле **right** со значением по умолчанию **undefined**. Дальше мы создаем экземпляра записи со следующими значениями полей: **left** – **"lvalue"**, **middle** – значение по умолчанию (**null**), **right** – **rvalue**:

```
SimpleRecord = #demo{left = "lvalue", right = rvalue}.
```

Теперь создадим еще один экземпляр записи, задав всем полям одно и то же значение **none**:

```
EmptyRecord = #demo{_ = none}.
```

Далее, получим доступ к полям: **SimpleRecord#demo.left** возвратит нам значение **"lvalue"**, а **#demo.left** вернет нам позицию поля **left** в кортеже, которым является запись **demo**, а именно **2**. В следующем примере мы изменим экземпляр записи **SimpleRecord** (мы помним, что экземпляр **SimpleRecord** не меняется, а вместо этого создается новая запись, отличающаяся от **SimpleRecord** лишь значением поля **middle**):

```
OtherSimpleRecord = SimpleRecord#demo{middle = 333}.
```

Экземпляр записи **OtherSimpleRecord** – это результат операции изменения экземпляра записи **SimpleRecord**; он содержит следующие значения полей: **left** – **"lvalue"**, **middle** – **333**, **right** – **rvalue**.

И, наконец, приведем несколько примеров операции соответствия шаблону для записей. В следующих примерах операция соответствия шаблону выполняется успешно:

```
#demo{left = "lvalue"} = SimpleRecord.
#demo{left = "lvalue"} = OtherSimpleRecord.
{demo, "lvalue", Middle, Right} = OtherSimpleRecord.
```

В следующих примерах операция соответствия шаблону не выполняется, и все заканчивается ошибкой времени выполнения:

```
#demo{left = "lvalue", middle = 333} = SimpleRecord.
#demo{left = "rvalue"} = OtherSimpleRecord.
{demo, Left, Right} = OtherSimpleRecord.
```

Выше мы познакомились с такими сущностями, как кортежи и основанные на них записи. После этого знакомства возникает вопрос: а как обстоят дела с инкапсуляцией данных? Ответ достаточно очевиден – никакой инкапсуляции данных нет (она не поддерживается моделью представления данных в кортежах и записях). Давайте разберемся, насколько это плохо для нас как для разработчиков. Для чего нужна инкапсуляция? Для того, чтобы скрывать детали реализации, и для поддержания целостности данных. Рассмотрим целостность данных. Кортежи и записи в языке **Erlang** неизменяемы. Поэтому, если мы изначально создали кортеж или запись с правильными данными, то этот кортеж или запись так и останутся с правильными данными во время своей жизни (пока не будут собраны сборщиком мусора). Защититься же от неправильного созданного кортежа или записи просто: достаточно проверить на корректность передаваемый кортеж или запись в качестве аргумента функции. Теперь перейдем к вопросу о сокрытии деталей реализации. Детали реализации в языке **Erlang** никакими средствами не скрыть (это особенно хорошо заметно, когда мы начинаем работать со словарями из модуля **dict**; но об этом в следующей части). В нашем случае это означает, что все поля кортежа либо записи доступны любому коду. К тому же, язык **Erlang** никакого состояния не хранит (ибо он функциональный язык программирования), поэтому мы часто вынуждены передавать кортежи, записи и данные других типов, которые содержат все необходимые детали для обработки (как, например, с вышеупомянутыми словарями). Нельзя однозначно сказать, что это плохо, т. к. в большинстве случаев барьер этого сокрытия данных не так уж сложно преодолеть (в тех языках, где он есть). Поэтому отсутствие сокрытия данных (по мнению автора) не является фатальной вещью и при развитой культуре программирования проблем не представляет.

При работе с кортежами и записями возникает еще один вопрос: а можно ли как-то связать данные и код, их обрабатывающий (как это сделано в объектно-ориентированных языках программирования), или же данные у нас сами по себе, а код – сам по себе? Сначала кажется, что связать данные и код в языке **Erlang** невозможно: не хватает соответствующих языковых конструкций (таких как классы). Но давайте подумаем более тщательно (а также вспомним тему про функции в **LXF145**). В языке **Erlang** функции являются полноправными типами данных, поэтому нам никто не мешает создать кортеж либо запись, одним (или несколькими) из членов которого будет функция. Таким образом, мы связываем в пределах кортежа либо записи данные и методы для их обработки.

Возникает вполне естественный вопрос: а как метод обработки данных узнает о том, с какими данными он связан? В традиционных объектно-ориентированных языках у любого метода класса существует указатель на экземпляр (например, **this** в C++). В нашем же случае ничего подобного нет, что естественно. В этом случае, его стоит эмулировать, передавая в качестве первого пара-

метра экземпляр кортежа либо записи, с которым связан метод обработки (при этом мы не забываем, что переданный экземпляр мы изменить не сможем).

Давайте рассмотрим пример. Для начала определим запись, которая будет содержать данные и метод для их обработки:

```
-record(class, {data, method = fun(This) -> This end}).
```

После этого создаем экземпляр записи и вызываем метод-обработчик, который изменяет экземпляр записи; таким образом мы и получаем новый экземпляр записи:

```
Object = #class{data = none, method = fun(This) -> #class{data =
modified, method = This#class.method} end}.
ProcessMethod = Object#class.method.
NewObj = ProcessMethod(Object).
```

Следует заметить, что подобные конструкции напоминают скорее не обычные классы, а объекты – прототипы (например, из языка JScript).

В качестве финального аккорда, давайте рассмотрим небольшой пример и применим часть знаний на практике. Пусть у нас есть иерархические данные (например, XML), которые мы представляем в виде дерева в памяти. Наша задача – отфильтровать и обработать эти иерархические данные. Иерархические данные мы представляем в виде дерева узлов; для этого мы определяем запись следующего типа:

```
-record(node, {value = "", children = [], attr = []}).
```

Из определения видно, что каждый узел содержит некоторое значение, список дочерних узлов и список атрибутов (предполагаем, что значение, связанное с узлом – строка, а атрибутом может быть любой объект). Следующая и самая важная часть – это сам метод для фильтрации и обработки узлов. Несмотря на свою важность, он выглядит очень просто:

```
process_node(Node, Filter, Map) ->
        Children = lists:map(fun(Child) -> process_node(Child,
Filter, Map) end, lists:filter(Filter, Node#node.children)),
Map(Node#node{children = Children}).
```

В этом методе мы сначала фильтруем и обрабатываем список дочерних узлов для текущего узла, а потом обрабатываем и сам текущий узел (фильтровать текущий узел не надо, т. к. он уже отфильтрован, когда был в списке дочерних узлов родительского узла). Вся работа по фильтрации списка дочерних узлов и обработке отдельного узла вынесена в аргументы метода **Filter** и **Map**, которые, очевидно, являются функциями. Следующий шаг – генерация тестовых данных, для демонстрации работы нашего метода. Понятно, что в реальной системе мы бы парсили XML-файл и преобразовывали его в нашу структуру. В нашем случае достаточно объявить метод, создающий жестко заданные тестовые данные:

```
test_data() ->
        NodeList2 = [#node{value = "most_inner", attr = ["attr1",
"attr2"]}, #node{value = "most_inner"}],
        NodeList1 = [#node{value = "inner", children =
NodeList2, attr = []}, #node{value = "inner", attr = ["attr3"]}],
        #node{value = "doc", children = NodeList1}.
```

Теперь нам нужен метод, который все собирает вместе и запускает обработку (и который мы экспортируем из модуля). В методе мы объявляем функции для фильтрации и обработки:

```
go() ->
        Doc = test_data(),
        Filter = fun(Node) -> length(Node#node.attr) == 0 end,
        Map = fun(Node) -> Node#node{value = Node#node.
value ++ " processed"} end,
        process_node(Doc, Filter, Map).
```

Остался финальный штрих – объявление модуля и списка экспортируемых функций:

```
-module(hierarchy_demo).
-export([go/0]).
```

Сохраняем исходный код в файле с именем **hierarchy_demo.erl**, запускаем среду выполнения **Erlang**. В консоли **Erlang** запускаем компиляцию: командой **c(hierarchy_demo).**, после чего запускаем (**hierarchy_demo:go()**). и наблюдаем результат фильтрации и обработки.

> «Достаточно объявить метод, создающий тестовые данные.»

В данной статье мы рассмотрели и обсудили, что такое кортежи и записи и как их правильно «готовить». Мы увидели, что кортежи (наряду с функциями и списками) – это фундаментальные строительные блоки для создания структур данных, и без них никуда. А в следующей статье мы рассмотрим следующую базовую сущность функциональных языков (и **Erlang** в том числе) – а именно, списки. LXF

Полезные заметки

» Обозначение **fun_name/arity** (например, **size/1**) означает функцию с именем **fun_name**, у которой количество аргументов равно **arity**. Две функции могут иметь одно и то же имя, но различаться при этом по числу аргументов.

» Чтобы в консоли среды выполнения **Erlang** создать запись, вместо директивы **–record** следует использовать команду **rd(Name, Definition)**. Здесь **Name** – имя записи, **Definition** – список полей и их значений по умолчанию. Для работы с записями в консоли среды выполнения **Erlang** есть еще несколько полезных команд: **rl()** для вывода всех определений записей, **rf()** – для удаления всех определений записей и **rf(Names)** – для удаления всех определений всех записей, имена которых находятся в списке **Names**. Каждая команда в консоли должна завершаться точкой.

» Помимо функции **tuple_to_list/1**, преобразующей кортеж в список, есть обратная функция **list_to_tuple/1**, преобразующая список в кортеж.

» В модуле **erlang** определена пара BIF, позволяющих узнать, является ли кортеж записью: **is_record/2** и **is_record/3**.

» Функция **filter(Pred, List)** из модуля **lists** фильтрует список **List**: в результате она возвращает список, который содержит только те элементы из **List**, для которых предикат **Pred** возвращает **true**.

» Функция **map(Fun, List)** из модуля **lists** возвращает список, составленный из результатов применения функции **Fun** к каждому элементу из списка **List**.

» Компилятор языка **Erlang** в каждом модуле, использующем записи, объявляет две следующие псевдофункции: **record_info(fields, Record)** и **record_info(size, Record)**. Первая возвращает список имен полей, вторая – размер кортежа, лежащего в основании записи (количество полей в записи + 1). Здесь **fields** и **size** – атомы, **Record** – имя типа записи.