Erlang: Сущности

Никто не забыт, и ничто не забыто: завершая рассмотрение базовых сущностей, **Андрей Ушаков** берется за временно отложенные темы.

Наш эксперт

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

В предыдущих номерах мы поговорили обо всех больших базовых сущностях. На данный момент у нас не рассмотрен ряд тем (из категории базовые сущности), которые либо слишком малы, чтобы про них писать отдельную статью, либо для них не хватило места в одной из статей про базовые сущности (и они оказались менее приоритетными по сравнению с другими темами). Поэтому данная статья посвящена таким темам: атомам, целым и действительным числам, замыканиям и сериализации и десериализации.

Мы уже затрагивали тему атомов в одном из номеров (см. **LXF143**), теперь вернемся к ней снова. Итак, что такое атомы? В нашем случае, это не те атомы, которые присутствуют в химии и атомной физике; атомы – это аналог констант в языке Erlang. Но это не совсем обычные константы, т. к. обычные константы связывают имя константы с некоторым значением, а в случае атомов имя атома является одновременно и его значением. Так, например, атомами являются величины **true** и **false**.

Какие имена являются действительными именами и, соответственно, значениями атомов в языке Erlang? Если имя начинается со строчной буквы и содержит внутри только буквы (как строчные, так и заглавные), цифры, символ подчеркивания **"_"** и символ **"@"**, то такое имя является действительным именем атома. Для всех остальных имен, которые не являются действительными именами атомов, достаточно заключить их в одинарные кавычки, чтобы они стали действительны. Вполне очевидно, что, если имя **name** является действительным именем атома, то и имя **'name'** также является действительным именем атома. Мало того, имена **name** и **'name'** представляют один и тот же атом.

Что важно, атомы в языке Erlang, в отличие от констант, значения которых подставляются компилятором во время компиляции, являются полноправными объектами среды выполнения Erlang. Это означает, что мы можем проверить, является ли объект атомом (BIF **is_atom/1**), а также преобразовывать объекты других типов в атомы и наоборот. Преобразовывать в атомы и наоборот мы можем только строки; мы помним (см. **LXF148**), что строки у нас могут быть представлены либо в виде списков, либо в виде битовых строк. Поэтому преобразовывать атомы мы можем в списки и битовые строки (**atom_to_binary/2**, **atom_to_list/1**) и наоборот **(binary_to_atom/2**, **list_to_atom/1**). Что интересно, помимо обычных функций преобразования битовых строк и списков в атомы, определены версии этих функций, преобразующие битовые строки и списки в атомы, только если данные атомы уже созданы **(binary_to_existing_atom/2**, **list_to_existing_atom/1**). Если атома, соответствующего строковому представлению не существует, то будет брошена ошибка времени выполнения.

Подобная функциональность нужна для обеспечения безопасности при взаимодействии с внешним миром. Атомы, в отличие от других объектов, создаются навсегда и не уничтожаются сборщиком мусора, если на них нет ссылок. При создании нового атома создается запись в системной таблице атомов. Вполне очевидно, что количество записей в этой таблице ограничено. Поэтому, если неаккуратно десериализовать атомы из строковых данных, приходящих из внешнего мира, то может возникнуть ситуация, когда таблица атомов будет переполнена. После этого в системе нельзя будет создать ни одного атома. Конечно, есть возможность создавать атомы не напрямую, а при десериализации других структур данных, например, кортежей. Если существует такая опасность, то сериализацию и десериализацию объектов при взаимодействии с внешним миром лучше организовать при помощи BIF **term_to_binary/2** и **binary_to_term/2**. Если мы хотим иметь константу в том смысле, в котором она понимается в других языках программирования (т. е. некоторая постоянная величина, связанная с некоторым именем, которое заменяется этой величиной на этапе компиляции), то у нас есть один вариант: использовать директиву препроцессора **define**. Например, мы хотим определить некоторую константу **MAGIC_NUMBER** и затем ее использовать в коде программы; сделать это можно следующим образом:

```
-define(MAGIC_NUMBER, 666999666).
...
SomeFunc(Arg) -> SomeFuncImpl(Arg, ?MAGIC_NUMBER).
```

Видно, что для использования константы (а если быть более точным, то макроса), необходимо использовать специальный синтаксис: перед именем константы должен идти символ **"?"**. Видно, что привычные для нас константы являются ничем иным, как макроопределениями (как в языке C).

На атомы в языке Erlang возложена еще одна важная ответственность: атомы **true** и **false** представляют соответствующие логические значения (отдельного типа данных для логических значений в языке Erlang нет). Это означает, что везде, где ожидается появление некоторой логической величины, должен в ито-

рассмотрены

ге появиться один из этих атомов (непосредственно либо в результате вычисления выражения). Это касается таких областей, как результат сравнения двух объектов, логические выражения, возвращаемое значения предикатов. Давайте рассмотрим всех их по очереди.

Все объекты в языке Erlang можно сравнивать не только на равенство (оператор ==) и неравенство (оператор /=), но и на упорядоченность друг относительно друга (операторы =<, <, =>, >). При этом списки сравниваются поэлементно; у кортежей сначала сравниваются размеры, и если размеры одинаковы, то кортежи сравниваются поэлементно. А что будет, если мы попытаемся сравнить два элемента разного типа? Элементы разных типов считаются не равными, за одним исключением: если сравниваются целые и действительные числа, то целое число преобразуется в действительное, после чего и происходит сравнение. Так, например, выражение **1 == 1.0** вернет **true**.

Если мы будем сравнивать два элемента разных типов на упорядоченность, то все зависит от типов сравниваемых объектов. В языке Erlang, в отличие от многих других языков, сравнение объектов разных типов разрешено и определено. Разные типы данных имеют следующий порядок по возрастанию: числа, атомы, ссылки, порты, идентификаторы процессов PId, кортежи, списки, битовые строки. Раз уж мы упомянули здесь все типы данных в языке Erlang, то следует сказать пару слов о незнакомых нам типах. Ссылка [reference] – это тип данных для хранения некоторой уникальной метки; для получения этой метки используется BIF **make_ref/0** (этот BIF начинает давать неуникальные значения примерно через 2^{82} вызовов). Порт (или идентификатор порта) – это тип данных для взаимодействия с внешним миром. О портах мы поговорим в будущих статьях. А теперь вернемся к нашим баранам: к упорядочению данных различных типов.

> «Сравнение объектов разных типов разрешено и определено.»

Рассмотрим несколько примеров. Выражение **12 < aaa** вернет **true**, т. к. объекты типа число всегда меньше объектов типа атом. Выражение **{aa, 12} < {12, aa, 12}** вернет **true**, т. к. мы сравниваем кортежи разных размеров и кортеж, стоящий слева от оператора, имеет меньший размер; т. е. он меньше кортежа, стоящего справа от оператора. Выражение **{12, aa} < {aa, aa}** вернет **true**, т. к. кортежи имеют равный размер, поэтому мы сравниваем их поэлементно; при этом первый элемент кортежа, стоящего слева от оператора, меньше, чем первый элемент кортежа, стоящего справа от оператора. Выражение **[aa, 2] < [1, 2, 3]** вернет **false**, т. к. списки сравниваются поэлементно независимо от их размера; при этом первый элемент списка, стоящего слева от оператора, больше, чем первый элемент списка, стоящего справа. Выражение **[1, 2] < [1, 2, 3]** вернет **true**, т. к. все элементы списка, расположенного слева от оператора, равны первым двум элементам списка, расположенного справа от оператора, и список, расположенный справа, имеет больший размер.

Язык Erlang имеет еще одну пару операторов для сравнения объектов: это оператор точного равенства **=:=** и оператор точного неравенства **=/=**. Разница между этими операторами и операторами равенства и неравенства в том, что операторы равенства и неравенства могут преобразовать тип одного из аргументов (целое число в действительное), а операторы точного равенства и неравенства никогда этого не делают. Так, например, выражение **1 == 1.0** равно **true**, а выражение **1 =:= 1.0** равно **false**.

Пойдем дальше – поговорим о логических выражениях. Логические выражения строятся из других логических выражений, атомов **true** и **false** и выражений, выполнение которых дает один из атомов **true** или **false** (как, например, выражения сравнения). Для построения логических выражений используются следующие операторы: унарное логическое НЕ **not**, логическое И **and**, логическое ИЛИ **or** и логическое ИСКЛЮЧАЮЩЕЕ ИЛИ **xor**. Помимо этих операторов, есть еще операторы «сокращенного» логического И **andalso** и «сокращенного» логического ИЛИ **orelse**. Отличие этих операторов от обычных операторов И и ИЛИ в следующем: операторы **and** и **or** всегда вычисляют оба своих операнда; оператор **andalso** вычисляет второй операнд, только если значение первого операнда равно **true**; оператор **orelse** вычисляет второй операнд, только если значение первого операнда равно **false**. Так, например, предположим, что определена переменная **X** и ее значение равно **0**; тогда логическое выражение **(X /= 0) andalso (1/X > 2)** вернет **false**, а выражение **(X /= 0) and (1/X > 2)** вызовет ошибку времени выполнения (деление на ноль).

И последнее, с чем связаны атомы **true** и **false**: предикаты. Многие библиотечные функции (например, **lists:filter/2**) в качестве одного из своих аргументов ожидают предикат. Предикат – это функция одного аргумента (рассматриваемого в данный момент), которая возвращает либо **true**, либо **false**. Так, например, если мы хотим получить список всех чисел в диапазоне от 1 до 1000, которые делятся на 13 без остатка, то сделать это мы можем следующим способом:

```
lists:filter(fun(Number) -> Number rem 13 == 0 end, lists:seq(1,
1000)).
```

В этом вызове функции **lists:filter/2** первым аргументом стоит анонимная функция-предикат, которая возвращает **true**, если текущее рассматриваемое число делится на 13 без остатка.

А теперь остановим взгляд на целых и действительных числах. Пожалуй, про действительные числа особо и нечего сказать: они занимают в памяти 16 байт (4 слова) в 32-битной версии и 24 байта (3 слова) в 64-битной версии среды выполнения. Для действительных чисел определены стандартные арифметические операторы. И, наконец, в стандартной библиотеке языка Erlang находится модуль **math**, который содержит стандартные математические функции, такие как синус, косинус и т. д.

С целыми числами все намного интереснее. Целые числа делятся на два типа: обычные и большие, причем это деление про- »

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

исходит «под капотом» (в отличие, например, от языка **Java**, в котором есть обычные типы целых чисел и есть тип **java.math.BigInteger** для больших целых чисел). Мы работаем с большими целыми числами точно так же, как и с обычными. Обычные целые числа занимают 4 байта (одно слово) в 32-битной версии среды выполнения (а если быть точным, то 28 бит); в 64-битной версии среды выполнения они занимают 8 байт (одно слово) – а если быть точным, то 60 бит. Большие целые числа занимают минимум 12 байт в 32-битной версии среды выполнения и минимум 24 байта в 64 битной версии среды выполнения. Сверху большие целые числа ничем не ограничены (точнее, ограничены размером адресного пространства процесса). Такая архитектура целых чисел в языке Erlang позволяет легко решать разные задачи, в которых требуется работа с произвольными целыми числами, например, из области computer science, теории чисел и т.д.

Давайте рассмотрим небольшой пример: предположим, перед нами стоит задача посчитать сумму всех цифр числа **100!** (так, например, **6! = 720**, а сумма цифр числа **6!** равна **9**). Для начала определим функцию вычисления значения факториала (причем определим с использованием хвостовой рекурсии):

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

С помощью функции **factorial/1** мы можем вычислить значение **100!**. Чтобы осознать, насколько это число велико (и убедиться, что с использованием обычных целых чисел эту задачу на других языках программирования так просто не решить), давайте вычислим величину **1.0*factorial(100)**. Получится действительная величина $\mathbf{9{,}33*10^{157}}$. Теперь вычислим сумму цифр этого числа:

```
lists:sum(lists:map(fun(Char) -> Char-$0 end,
integer_to_list(factorial(100)))).
```

В результате мы получим величину **648**. С той же легкостью мы можем вычислить сумму цифр еще большего числа – **1000!** (это число уже не представимо в виде действительного числа); результатом будет **10539**. Столь же легко решаются и другие задачи с использованием больших целых чисел на языке Erlang.

В одном из недавних номеров (см. **LXF145**) мы говорили о функциях и связанными с ними концепциях функционального программирования. В этом разговоре (в связи с ограничениями на размер статьи) мы отложили в сторону одну важную концепцию – замыкания. У читателей может возникнуть вопрос: если эта концепция важна, то почему она осталась в стороне? Понимание этой концепции и деталей реализации важно, с точки зрения автора, в языках с изменяемыми типами данных, таких как C#, C++, JavaScript, Java. В чисто функциональных языках (и в частности, в Erlang), где изменяемые типы данных отсутствуют, данная концепция тривиальна и не содержит подводных камней.

Итак, что такое замыкание? Замыкание – это «захват» внешнего контекста при создании локального объекта. Под локальным объектом в языке Erlang подразумеваются анонимные функции (в C# это анонимные делегаты и лямбды, в C++ 11 – анонимные функторы, в Java – локальные и анонимные классы, в JavaScript – анонимные функции). Внешним контекстом являются локальные переменные и параметры функции, в которой объявляется данный локальный объект (в объектно-ориентированных языках к контексту также относятся поля объекта, которому принадлежит эта внешняя функция).

Рассмотрим небольшой пример, иллюстрирующий эту концепцию. Мы объявляем фабричную функцию, которая конструирует и возвращает анонимную функцию, складывающую два числа:

```
factory(Number) -> fun(X) -> Number + X end.
```

Фабричная функция возвращает анонимную функцию одного аргумента, но эта функция складывает два числа. Откуда же берется второе число? А второе число берется из внешнего по отношению к анонимной функции контекста: вторым числом является аргумент фабричной функции **Number**. Мы можем присвоить построенную функцию переменной **AddFun = factory(10)** и использовать ее: выражение **AddFun(5)** вернет **15**, выражение **AddFun(25)** вернет **35**, и т.д.

Теперь рассмотрим другой пример, когда внешним контекстом является не аргумент внешней функции, а уже объявленная локальная переменная. В этом примере мы объявляем фабричную функцию, которая конструирует и возвращает анонимную функцию для проверки строки на вхождение в нее одной из известных подстрок:

```
factory() ->
  PredefinedStrings = [“abc”, “nmo”, “xyz”],
    fun(String) ->
          lists:any(fun(PredefinedString) -> string:str(String,
PredefinedString) > 0 end, PredefinedStrings)
end.
```

Список известных подстрок определяется в локальной переменной **PredefinedStrings** фабричной функции и входит во внешний контекст для конструируемой анонимной функции. Теперь (как и в первом примере) мы можем присвоить построенную функцию переменной **StrFun = factory()** и использовать ее: выражение **StrFun(“acddc”)** вернет **false**, выражение **StrFun(“aabcc”)** вернет **true**.

С первого взгляда кажется, что все достаточно просто: при создании внутренней функции мы просто копируем значения внешних переменных и параметров функций (если переменная содержит ссылку на объект в куче, то, естественно, копируется ссылка). И для языка Erlang (впрочем, как для любого другого языка с неизменяемым состоянием) это поведение справедливо. Мы могли бы на этом закончить разговор о замыканиях, но давайте пойдем дальше и для большей ясности поговорим о том, какие есть сложности с замыканиями в языках с изменяемым состоянием (таких как C#, C++, JavaScript, Java). И в качестве языка, на котором мы продемонстрируем, как нам все-таки повезло с Erlang, мы возьмем C#. Для всех примеров на C# автор использует Mono как среду времени выполнения и MonoDevelop как IDE для разработки (кто находится на «темной стороне силы», для примеров на C# может использовать Visual Studio 2008 и выше). Следует сделать еще пару замечаний по поводу компилятора C# в плане замыканий: компилятор C# создает объект некоторого генерируемого класса, который содержит внешний контекст; компилятор C# генерирует класс, который охватывает минимально возможный внешний контекст.

Попытаемся па нескольких примерах понять и объяснить получаемое поведение. Начнем с простого примера, в котором мы создаем несколько анонимных делегатов в цикле, после чего поочередно вызываем их:

```
const int count = 2;
System.Action[] actions = new System.Action[count];
for (int i = 0; i < count; ++i)
{
  actions[i] = () => System.Console.WriteLine(i);
}
foreach (System.Action action in actions)
  action();
```

Мы хотим, чтобы каждый созданный делегат вывел на консоль номер итерации цикла, на которой он был создан (т.е. мы ожидаем,

что выведутся числа 0 и 1). Но на самом деле вывод будет совсем другой: а именно, два числа 2 (значение константы **count**).

Сначала получаемое поведение кажется немного странным, но давайте вспомним замечание про поведение компилятора. В данном случае, минимально возможным контекстом является цикл **for**: компилятор генерирует класс, содержащий в качестве поля переменную **i**, по которой происходят итерации цикла, после чего перед циклом создает объект этого сгенерированного класса, и в анонимных делегатах вместо переменной **i** используется поле этого созданного объекта. Поскольку выполнение делегатов происходит уже после цикла, то естественно, что все делегаты получают одно и то же значение – то, которое получило поле **i** сгенерированного объекта, т.е. число 2.

Чтобы получить поведение, которое мы ожидаем, необходимо заставить компилятор минимально возможным внешним контекстом считать тело цикла. Сделать это можно следующим способом: объявить в теле цикла переменную, которой присваивать значение переменной цикла. Это вынудит компилятор сгенерировать класс, который в качестве поля будет содержать эту переменную и создавать объект этого класса (и связывать этот объект с соответствующим анонимным делегатом) для каждой итерации цикла.

```
const int count = 2;
System.Action[] actions = new System.Action[count];
for (int i = 0; i < count; ++i)
{
  int j = i;
  actions[i] = () => System.Console.WriteLine(j);
}
foreach (System.Action action in actions)
  action();
```

В данном варианте поведение будет совпадать с ожидаемым: выведутся числа 0 и 1.

Помимо этого, весьма неочевидного случая, возможна более очевидная ситуация. Предположим, мы создаем некий объект (например, список), затем создаем анонимный делегат, после чего меняем состояние объекта (например, добавляем в список дополнительные элементы). Когда мы вызовем созданный анонимный делегат, то будем иметь дело с ссылкой на измененный объект. Чтобы такого не происходило, нужно создать копию объекта (до создания анонимного делегата) и использовать ее. Пожалуй, это вся «черная магия», связанная с замыканиями в языках с изменяемым состоянием. Но для нас (для тех, кто использует язык Erlang либо другие языки с неизменяемым состоянием) все гораздо проще, и вышеперечисленных подводных камней нет.

А теперь обратимся к сериализации и десериализации данных. Мы уже упоминали о сериализации и десериализации в одном из недавних номеров (LXF148); давайте вспомним, о чем мы там говорили. Атомы и списки, состоящие из целых чисел, битовых строк и подобных же списков мы можем преобразовывать в обычные битовые строки. Атомы преобразуются в строку; списки же «расплющиваются» и превращаются в битовую строку, которая последовательно содержит все данные в том же порядке, что и в исходном списке.

Вновь рассмотрим несколько примеров. Выражение **atom_to_binary(abc, utf8)** возвращает строковое представление атома в виде битовой строки **<<"abc">>**. Выражение **list_to_binary([1, [<<12, 13>>, 2], <<14>>])** «расплющивает» список и возвращает следующую битовую строку **<<1, 12, 13, 2, 14>>**. Битовые строки можно преобразовывать и обратно – и опять же только в атомы и списки. Никакого прямого преобразования в другие типы данных нет. На самом деле отсутствие такой возможности не является чем-то страшным: для работы с низкоуровневыми данными в двоичном формате этих операций, операции создания битовых строк и операции соответствия шаблону [pattern matching] для битовых строк оказывается достаточно (например, для создания клиента, общающегося через сокеты с внешним сервером). Но предположим, что перед нами стоит задача о сохранении объектов любого типа на внешнем носителе или передача их по сети. Что нам делать в этом случае – разрабатывать свой формат сериализации?

На самом деле ничего подобного делать не надо: в языке Erlang есть сериализация и десериализация объектов любого типа. Для сериализации используется одна из BIF: **term_to_binary/1**, **term_to_binary/2**. Эти BIF сериализуют произвольный объект (при сериализации есть возможность задать уровень сжатия и минимальную версию) и возвращают так называемую расширенную битовую строку. Расширенная битовая строка – это битовая строка, которая помимо самих данных содержит еще и метаданные (и, что важно, длину битовой строки). Так, например, вызов **term_to_binary(1)** вернет следующую битовую строку – **<<131, 97, 1>>**; видно, что битовая строка помимо данных содержит и метаданные. Для десериализации используется одна из BIF: **binary_to_term/1**, **binary_to_term/2**. Эти BIF десериализуют производный объект, позволяя при этом производить безопасную десериализацию. Безопасной называется десериализация, при которой не создаются новые объекты, не собираемые сборщиком мусора. К таким объектам относятся атомы и ссылки на внешние функции. Предположим, например, что в консоли среды выполнения Erlang мы ввели следующее выражение **term_to_binary(ab)**; это выражение будет равно <<**131, 100, 0, 2, 97, 98>>**. Теперь в другой консоли среды выполнения Erlang введем выражение **binary_to_term(<<131,100,0,2,97,98>>, [safe])**. Нашим результатом будет ошибка времени выполнения, т.к. второй экземпляр среды выполнения Erlang ничего не знает про атом **ab**.

В данной статье мы обсудили ряд тем, которые мы еще не рассматривали до этого. Мы прошли большой путь в изучении языка Erlang, но это всего лишь первый шаг. Особая магия языка Erlang заключена все же в области построения многозадачных и распределенных приложений. Тем не менее, без понимания базовых сущностей невозможно понимать и более продвинутые темы. Поэтому следующая статья будет посвящена повторению и закреплению всех тем про базовые сущности; в ней мы сосредоточим свои усилия на практике и на примерах рассмотрим все базовые сущности еще раз. LXF

Полезные заметки

Директивы препроцессора

- **-include(File)** Директива для включения содержимого файла **File** в исходный код. Имя файла **File** может быть как абсолютным путем, так и относительным. Имя файла **File** может начинаться с **$VAR**, где **VAR** – это имя переменной среды, при этом **$VAR** заменяется значением этой переменной; если переменная **VAR** не определена, то в имени файла подстрока **$VAR** остается неизменной. Обычно подключаемые файлы содержат определения записей и макросов. Рекомендуется, чтобы имена подключаемых файлов имели расширение **.hrl**.
- **-include_lib(File)** То же самое, что и директива **include**, только имя файла **File** – всегда путь относительно первого компонента пути, который считается именем приложения (для получения директории для приложения используется вызов **code:lib_dir/1**).
- **-define(Const, Replacement)** Определение макроса. Используется следующим образом:**?Const**.
- **-define(Func(Var1, ..., VarN), Replacement)** Определение макроса-функции. Используется следующим образом:**?Func(Var1, ..., VarN)**.

Предопределенные макросы

- **?MODULE** Имя текущего модуля в виде атома
- **?MODULE_STRING** Имя текущего модуля в виде строки
- **?FILE** Имя файла текущего модуля
- **?LINE** Номер текущей строки
- **?MACHINE** Имя машины, 'BEAM'

Макросы, влияющие на компиляцию

- **-undef(Macro)** Отмена определения макроса **Macro**.
- **-ifdef(Macro)** Компиляция последующих строк только в том случае, если макрос **Macro** определен.
- **-ifndef(Macro)** Компиляция последующих строк, только если макрос **Macro** не определен.
- **-else** Разрешено применять только после директив **ifdef** или **ifndef**. Если условие (заданное в директиве **ifdef** или **ifndef**) ложное, то компилируются строки после директивы **else**.
- **-endif** Определяет конец действия директив **ifdef** или **ifndef**.