Erlang: Коллекции

Андрей Ушаков принимается за новую базовую сущность: говорит – коллекции, подразумевает – списки. Но не только списки.

Наш эксперт

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

Мы продолжаем обзор базовых сущностей языка Erlang. Пришла пора поговорить о коллекциях. Когда мы говорим о коллекциях, то в первую очередь вспоминаем о списках. И действительно, списки – это наиболее часто используемая коллекция. Более того, списки часто являются базой для других коллекций и сущностей. Так, в языке Erlang списки являются базой для работы со строками (с не Unicode-строками; для Unicode-строк базой являются двоичные данные). Но разговор о строках будет в одном из следующих номеров.

Что же такое списки и для чего они нужны? Список – это контейнер для данных, обработка которых будет происходить одним и тем же способом. При этом количество данных (элементов списка) может быть любым. Не обязательно, чтобы тип всех элементов списка был одним и тем же: вполне возможна ситуация, когда список содержит данные различных типов. Главное, что мы всегда предполагаем, что все элементы списка будут обработаны одинаковым (или похожим) образом. Таким образом, списки нам нужны для хранения и обработки переменного количества похожих (с точки зрения обработки) данных. Каждый элемент списка располагается в определенной позиции, и получается, что элементы в списке упорядочены в соответствии с занимаемой ими позицией. Если мы добавляем элементы строго в конец списка, то в таком случае элементы списка будут упорядочены в соответствии с очередностью их добавления (в случае, например, множества это будет не так). Списки очень похожи на векторы из стандартной библиотеки C++, за одним исключением: списки в Erlang неизменяемые (как и все другие объекты).

Как объявить переменную типа список? Как мы помним (**LXF143**), Erlang – это язык со строгой динамической типизацией. Это означает, что тип переменной определяется в момент ее инициализации. Инициализатор для списка выглядит очень просто: внутри квадратных скобок “[” и “]” мы через запятую перечисляем значения элементов списка. Например, мы объявляем переменную типа список с ее инициализацией следующим образом: **ListVar = [12, “34”, 56].** Количество элементов в списке – это его размер. Минимальный размер списка – 0 элементов (пустой список). Максимальный размер списка ничем не ограничен (если быть точным, то максимальный размер списка ограничен только наличием – или отсутствием – свободной памяти).

Сразу же возникает вопрос: как получить доступ к отдельным элементам списка? Есть две возможности: использовать функцию **lists:nth/2** (о ней мы поговорим ниже) или операцию соответствия шаблону [pattern matching]. Операция соответствия шаблону (в отличие от кортежей) не позволяет получить доступ к произвольному элементу списка: она позволяет всего лишь обратиться к головному элементу и оставшейся части списка. Выглядит это следующим образом: **[Head | Tail] = [1, 2, 3, 4].** Здесь переменная **Head** получит значение 1, а переменная **Tail** – [2, 3, 4]. Очевидно, что в качестве головного в шаблоне можно задать несколько переменных; эти переменные будут сопоставляться (а если у них нет значения, то получать соответствующие значения) первым нескольким (по числу переменных в шаблоне) элементам списка. И вполне логично, что если мы зададим в головном шаблоне переменных больше, чем элементов в списке, то такой список операцию соответствия шаблону не пройдет. Так, например, следующая операция соответствия шаблону выполняется успешно: **[H1, H2 | Tail] = [1, 2, 3]** и переменные **H1**, **H2**, **Tail** получают значения 1, 2 и [3] соответственно. А следующие две операции соответствия шаблону не выполняются: **[1, H2 | Tail] = [2, 3, 4]** и **[H1, H2, H3 | Tail] = [1, 2]**. Подобный (и кажущийся на первый взгляд ограниченным) вид операции соответствия шаблону позволяет рекурсивно обрабатывать весь список поэлементно. Например, функцию нахождения суммы всех элементов в списке можно написать следующим образом:

```
list_sum(NumberList) -> list_sum_impl(NumberList, 0).
list_sum_impl([], Sum) -> Sum;
list_sum_impl([Number | Rest], Sum) -> list_sum_impl(Rest,
Sum + Number).
```

За счет операции соответствия шаблону мы рекурсивно поочередно обрабатываем (суммируем) значения элементов, а благодаря хвостовой рекурсии (см. **LXF145**), подсчет суммы элементов списка разворачивается в цикл.

Помимо операции соответствия шаблону, для списков (списочных типов данных) определено еще две операции: “++” и “--”. Операция “++” – это операция сложения двух списков: она создает и возвращает новый список (мы помним, что все объекты в языке Erlang неизменяемые), в котором сначала идут все элементы из первого списка, а потом все элементы из второго. Операция “--” более сложна: она возвращает новый список, который содержит

как сущности

элементы, составляющие разность содержимого первого и второго списков. Алгоритм построения этой разности следующий: сначала копируются все элементы из первого списка, а потом для каждого элемента из второго списка ищется первое вхождение этого элемента в копию; если такое вхождение будет найдено, то из копии данный элемент удаляется, и дальнейшая работа продолжается уже с измененной копией. Об этом будет сказано ниже, но операция разности двух списков очень тяжелая: ее сложность составляет **O(M*N)**, где **M** – размер первого списка, **N** – размер второго.

Как мы видели выше, для создания списков можно использовать инициализаторы и операции сложения и разности. Но в общем случае это далеко не всегда удобно, хотя, используя эти операции и рекурсивные функции, можно создать любой список. Поэтому в языке Erlang существует техника, легко позволяющая создавать списки в соответствии с очень сложными правилами: эта техника называется конструирование списков [List Comprehensions]. Имеются две сущности, появляющиеся в выражении конструирования списков: генераторы и фильтры.

Генератор – это сущность, порождающая некоторую последовательность. В качестве генератора может выступать любое выражение, результатом выполнения которого является список. Чтобы иметь доступ к элементам генерируемого списка, с каждым генератором связывают переменную с уникальным (в пределах данного выражения конструирования) именем. Таким образом, выражение для генератора имеет следующий вид: **Pattern <- ListExpr**, где **Pattern** – шаблон для операции соответствия, **ListExpr** – выражение, результатом выполнения которого является список. Шаблоном **Pattern** обычно является имя переменной, используемой для доступа к элементам генерируемого списка; однако, если выражение **ListExpr** возвращает список «сложных» объектов (например, кортежей), то шаблон может быть более сложным.

Фильтр – это выражение, возвращающее **true** или **false**. Фильтр позволяет возвращать не все значения от генератора, а только те, которые удовлетворяют некому условию.

А теперь давайте соберем все вместе. Выражение конструирования списков имеет следующий вид: **[Expr || Qualifier1,...,QualifierN]**, где **Qualifier1**, …, **QualifierN** – это либо выражение генератора, либо выражение фильтра, а **Expr** – выражение для формирования элементов итогового списка, в которое могут входить любые переменные из области видимости выражения **Expr** (в том числе и все переменные, связанные с генераторами). Следует также упомянуть, что и в выражения фильтра также могут входить любые переменные, но уже из области видимости выражения фильтра: это означает, что в выражении фильтра могут использоваться только те переменные для генераторов, которые располагаются левее определения выражения фильтра. Как все это работает?

> «Генератор — сущность, порождающая последовательность.»

Вычисляются все возможные комбинации элементов из выражений генераторов (в том порядке, в котором эти выражения генераторы объявлены), после чего эти комбинации пропускаются через выражения фильтра. Все те комбинации, для которых все выражения фильтра вернули **true**, формируют элементы итогового списка при помощи вычисления выражения **Expr** для каждой такой комбинации. С первого взгляда кажется, что все это очень сложно, но на самом деле это не так. Давайте рассмотрим несколько примеров, чтобы концепции конструирования списков улеглись в голове. Для начала возьмем пример простого комбинирования элементов от двух генераторов без фильтров (так называемое Декартово произведение):

```
[{X, Y} || X <- [1, 2], Y <- [“a”, “b”]].
```

Результат вычисления этого выражения очевиден – это будет список

```
[{1, “a”}, {1, “b”}, {2, “a”}, {2, “b”}].
```

В следующем примере мы будем возвращать список пар чисел, чья сумма четная:

```
[{X, Y} || X <- [1, 2, 3], Y <- [1, 2, 3], (X+Y) rem 2 == 0].
```

Очевидно, что результатом будет следующий список:

```
[{1, 1}, {1, 3}, {2, 2}, {3, 1}, {3, 3}].
```

А теперь приведем примеры сложнее. Первый пример – это реализация алгоритма быстрой сортировки [quick sort]:

```
sort([Middle | T]) -> sort([X || X <- T, X < Middle]) ++ [Middle] ++
sort([X || X <- T, X >= Middle]);
sort([]) -> [].
```

Второй пример – это реализация методов фильтрации [filter] и преобразования [map] списков:

```
map(Fun, List) -> [Fun(X) || X <- List].
filter(Pred, List) -> [X || X <- List, Pred(X)].
```

И, наконец, следует заметить следующее: в выражениях конструирования списков помимо генераторов списочных данных могут появляться и генераторы битовых данных (битовых строк), результат которых преобразуется в списочные данные. Но разговор о битовых данных (битовых строках) – это тема одной из следующих статей.

Выше мы показали в качестве примеров реализацию методов сортировки, фильтрации и преобразования списков. В реальных программах так делать не стоит: все эти и большое количество других функций уже реализованы в стандартных (поставляемых вместе с компилятором и средой выполнения) библиотеках языка Erlang. Часть этих функций является BIF. В первую очередь из этих BIF следует упомянуть следующие: **hd/1** возвращает головной элемент списка, **tl/1** возвращает остаток списка без головного элемента, **length/1** возвращает количество элементов списка. Помимо них, есть еще следующие BIF: **is_list/1** позволяет определить, является ли тип выражения списком, а набор функций **list_to_xxx/1** позволяет преобразовать список во что-то еще – например, в кортеж. Следует упомянуть, что следующие BIF можно применять в выражениях охраны (см. **LXF143**): **hd/1**, **tl/1**, **length/1**, **is_list/1**. Также существует достаточно большой набор функций (определенных в модуле **lists**) для более сложных задач работы со списками, включая фильтрацию, преобразование, вычисление агрегатов и многое другое. »

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

Давайте рассмотрим некоторые, как мне кажется, наиболее полезные функции из этого модуля. Функция **lists:filter(Pred, SourceList)** фильтрует исходный список **SourceList**: на выходе мы будем иметь список, содержащий только те элементы исходного списка, для которых предикат **Pred/1** вернет значение **true**. Следующий пример возвратит список только четных чисел:

```
lists:filter(fun(Number) -> Number rem 2 == 0 end, [1, 2, 3, 4]).
```

Функция **lists:map(TransformFun, SourceList)** преобразует исходный список **SourceList** поэлементно: каждый элемент исходного списка преобразуется при помощи функции **TransformFunc/1**. Так, следующий пример возвращает список квадратов чисел из исходного списка:

```
lists:map(fun(Number) -> Number*Number end, [1, 2, 3, 4]).
```

Функция **lists:foldl(FoldFun, Acc0, SourceList)** позволяет вычислить агрегат по исходному списку **SourceList.** Вычисление происходит следующим способом: начиная с начального значения агрегата **Acc0**; элементы списка обходятся слева направо (в прямом порядке) и для каждого элемента и предыдущего значения агрегата вычисляется новое значение агрегата (при помощи функции **FoldFun(Element, PrevAcc)**); значение агрегата, вычисленное для последнего элемента, становится результатом, возвращаемым функцией **lists:foldl/3**. Существует вариант функции **lists:foldl/3**, в котором обход элементов при вычислении агрегата осуществляется не слева направо, а справа налево (в обратном порядке) – **lists:foldr/3**. Ее наличие оправдывается ситуациями, когда значение получаемого агрегата может различаться в зависимости от направления обхода элементов (когда операция построения агрегата некоммутативна). Например, такая ситуация возникает, если мы хотим построить произведение матриц, расположенных в списке (операция произведения матриц, как известно, некоммутативна). Попробуем вычислить произведение чисел, заданных в исходном списке:

```
lists:foldl(fun(Number, Product) -> Number*Product end, 1, [1, 2, 3, 4]).
```

> «Библиотека языка Erlang содержит две реализации множеств.»

Следует отметить, что для вычисления суммы элементов списка использовать функцию **lists:foldl/3** не обязательно: для этой операции специально введена функция **lists:sum/1**. Следующие две функции позволяют проверить список в целом на выполнение некоторого условия: **lists:all(Pred, SourceList)** и **lists:any(Pred, SourceList)**. Функция **lists:all/2** возвращает **true**, если для всех элементов исходного списка выполняется условие, заданное функцией-предикатом **Pred**; функция **lists:any/2** возвращает **true**, если хотя бы для одного элемента исходного списка выполняется условие, заданное функцией-предикатом **Pred**. Вот как можно проверить, что все числа в списке больше 0:

```
lists:all(fun(Number) -> Number > 0 end, [1, 2, 3, 4]).
```

или – что хотя бы одно число больше 0:

```
lists:any(fun(Number) -> Number > 0 end, [-1, 2, -3, 4]).
```

В модуле **lists** есть еще масса полезных функций. Например, функции для сортировки: **lists:sort/1**, **lists:sort/2**, функции для сортировки без дубликатов: **lists:usort/1**, **lists:usort/2**, функция для доступа к элементу списка по индексу **lists:nth/2** и т.д. Перечисление всех их заняло бы слишком много места, но желающие могут посмотреть описание всех функций из модуля **lists** в документации по языку Erlang.

А мы пойдем дальше. Помимо списков, есть еще другие виды коллекций. В первую очередь это множества и словари. Зачем же они нужны, если аналоги множеств и словарей можно реализовать при помощи списков? Затем, что набор API для работы со списками не предназначен для работы со списками как со словарями и множествами. Особенно это касается операций добавления и удаления элементов: обычно множества и словари представляют собой либо деревья, либо хэш-таблицы, и добавление и удаление элементов должно производиться особым способом. Кроме того, временные характеристики операций в списках и во множествах и словарях отличаются. Так, например, операция поиска для списка имеет сложность **O(N)** (для отсортированного списка операция двоичного поиска имеет сложность **O(logN)**, но при этом возникают накладные расходы на сортировку и сложности добавления и удаления элементов в такой список). А для множеств и словарей операция поиска имеет сложность либо **O(logN)**, либо **O(1)**, в зависимости от структуры данных, использовавшейся при реализации. Поэтому использование списков в тех случаях, когда данные по своей природе представляют собой либо множество, либо словарь, не разумно (хотя и возможно в случае коллекций небольшого размера).

Рассмотрим поближе реализацию этих коллекций (множеств и словарей) в библиотеке языка Erlang. Библиотека языка Erlang содержит две реализации множеств, причем их интерфейс одинаков (под интерфейсом понимается множество экспортируемых из модуля функций): в модулях **sets** и **ordsets**. Между этими реализациями есть два различия. Во-первых, элементы сравниваются на равенство при помощи разных операторов: в реализации в модуле **ordsets** используется оператор "==", а в реализации в модуле **sets** – оператор "=:=". Разница между этими операторами в том, что оператор "=:=" сравнивает два операнда такими, какие они есть, а оператор "==" может преобразовать один операнд к другому, если они имеют разные типы. Так, например, **1 =:= 1.0** вернет **false**, т.к. операнды имеют разные типы, а **1 == 1.0** вернет **true**, т.к. целочисленный операнд 1 может быть преобразован в вещественный операнд 1.0. Во-вторых, реализация множеств в модуле **sets** не определена и может быть любой (на данный момент – с помощью хэш-таблицы), а множества в модуле **ordsets** реализованы с помощью упорядоченного списка. Точно так же, в языке Erlang есть две реализации словарей: в модуле **dict** и в модуле **orddict**. Реализация словарей в модуле **dict** не определена и может быть любой (на данный момент – с помощью хэш-таблицы); для сравнения элементов используется оператор "=:=". Словари в модуле **orddict** реализованы с помощью списка пар (кортежей) ключ-значение, упорядоченных по ключу; для сравнения элементов используется оператор "==".

Давайте посмотрим на практике, как работать со словарями и множествами. В следующем примере мы сначала создаем множество, потом добавляем в него два элемента, потом удаляем отсутствующий во множестве элемент и в конце получаем список всех элементов множества. Заметьте, что каждая операция, изменяющая множество, на самом деле создает новый объект (т.к. все объекты в языке **Erlang** неизменяемые).

```
Set0 = ordsets:new().
Set1 = ordsets:add_element(abc, Set0).
Set2 = ordsets:add_element(bac, Set1).
Set3 = ordsets:del_element(ccc, Set2).
List0 = ordsets:to_list(Set3).
```

Теперь рассмотрим пример со словарями: мы так же, как в предыдущем примере, сначала создаем словарь, потом добавляем две пары ключ – значение, потом удаляем данные из словаря по отсутствующему ключу и в конце получаем список всех пар ключ – значение, хранящихся в словаре.

```
Dict0 = orddict:new().
Dict1 = orddict:append(ccc, ccc_333, Dict0).
Dict2 = orddict:append(aaa, aaa_111, Dict1).
Dict3 = orddict:erase(bbb, Dict2).
List0 = orddict:to_list(Dict3).
```

Как мы увидели, списки, пожалуй, наиболее универсальная коллекция: на ее базе можно реализовать все другие коллекции. Однако у списков в языке **Erlang** имеется один минус: нет простого способа установить значение элемента в определенной позиции списка (пусть даже с созданием нового) – хотя в таких языках программирования, как C, C++, Java, C#, списки и их аналоги (векторы, массивы и т. д.) легко позволяют нам сделать это. Тем не менее разработчикам не придется писать свою реализацию списка с возможностью установки значения элемента в произвольной позиции: в библиотеке языка **Erlang** подобная реализация существует. Она называется массив и находится в модуле **array**. В массивах, в отличие от списков, индексация элементов начинается с 0, а не с 1; кроме того, массив можно создать фиксированного, а не динамического размера. Ну и, конечно, массивы позволяют задать значение элемента в определенной позиции; правда, как и в случае других объектов в языке Erlang, при этом будет создан новый объект. Покажем это на примере – создадим массив фиксированного размера (с размером 10 и значением по умолчанию -1), потом установим значение первого элемента (по индексу 0) равным 100, после чего получим значение второго элемента (по индексу 1) и, наконец, получим представление массива в виде списка:

```
Array0 = array:new(10, [fixed, {default, -1}]).
Array1 = array:set(0, 100, Array0).
Value0 = array:get(1, Array1).
List0 = array:to_list(Array1).
```

Мы рассмотрели списки и другие коллекции: теперь пришло время поговорить об эффективности использования тех или иных операций над коллекциями. Начнем разговор со следующих операций над списками: “++” и “--”. Операция “++” реализована следующим образом: она создает новый список, копируя все элементы из левого операнда, но элементы из правого операнда не копируются, а вместо этого создается ссылка на правый операнд. Все объекты в языке Erlang неизменяемые, и можно быть уверенным, что правый операнд будет всегда содержать один и тот же набор элементов. Поэтому, когда мы создаем новый список операцией “++”, добавляя элементы один за другим, очень важно с точки зрения производительности, чтобы элементы добавлялись спереди (во избежание ненужных операций копирования). Что же делать, если нам требуется обратный порядок построения списка? Все очень просто: построим список в порядке наибольшей эффективности, после чего при помощи функции **lists:reverse/1** сменим порядок следования элементов на обратный.

Теперь займемся операцией “--”. Эта операция строит список, являющийся разностью левого и правого операнда. Выше был рассмотрен алгоритм построения разности; сложность этой операции составляет **O(M*N)**, где **M** – размер первого списка, **N** – размер второго. Очевидно, что в случае больших списков эта операция займет много времени, и ее следует избегать (как и других операций из теории множеств, например, пересечения двух множеств).

Сложность операций над коллекциями неодинакова. Скажем, сложность функции **lists:max/1** будет **O(N)** (т. к. мы не располагаем информацией об упорядоченности списка), а сложность операции **lists:sort/1** – **O(NlogN)**.

Хочется отдельно остановиться на сложности доступа к элементам во множествах и словарях. У нас есть две реализации множеств (модули **sets** и **ordsets**) и словарей (модули **dict** и **orddict**). Реализации в модулях **ordsets** и **orddict** используют сортированный список, поэтому доступ к элементу (паре ключ–значение) будет иметь сложность **O(NlogN)**. Реализации в модулях **sets** и **dict** стандартом не определены, но на данный момент используются хэш-таблицы, поэтому доступ к элементу (к паре ключ–значение) будет иметь сложность **O(1)**. Однако тут есть одно «но»: вполне возможна ситуация, когда мы сохраняем в хэш-таблицу элементы, хэш-коды которых попадают в одну ячейку; в этом случае сложность доступа к элементу возрастает до **O(N)**.

В данной статье мы рассмотрели списки и другие коллекции. Но это не последняя статья про них: в одном из грядущих номеров статья будет про строки, базой которых являются списки; отдельную статью также планируется посвятить некоторым аспектам работы со строками, например, конструированию списков. А следующая статья будет посвящена такой интересной сущности языка Erlang, как битовые строки (двоичные данные). LXF

Полезные заметки

» O-нотация

O-нотация – математические обозначения для сравнения асимптотического поведения функций. Используются в различных разделах математики, но активнее всего – в математическом анализе, теории чисел и комбинаторике, а также при оценке сложности алгоритмов. В частности, фраза «сложность алгоритма есть O(F(n))» означает, что при больших значениях n время работы алгоритма (или общее количество операций) не более чем C*F(n), где C – некая положительная константа, n – объем входной информации алгоритма. Если утверждается, что сложность алгоритма O(n), то это означает, что время работы алгоритма растет линейно с ростом **n. За**пись O(1) означает, что время работы алгоритма не зависит от n.

» Хэш-код

Хэширование – это преобразование входного массива данных произвольной длины в выходную битовую строку фиксированной длины. Такие преобразования также называются хэш-функциями, а их результаты называют хэшем или хэш-кодом. Функция хэширования должна обладать следующим свойством: если два объекта считаются равными, то их хэш-коды должны быть одинаковыми; для неравных объектов хэш-коды могут быть как одинаковыми, так и разными. Ситуация, когда два разных объекта имеют одинаковые хэш-коды, называется коллизией. Хорошая хэш-функция должна давать как можно меньше коллизий. Очевидно, что для достижения этой цели результат хэш-функции должен быть равномерно распределен на всем множестве битовых строк фиксированной длины. Например, если хэш-функция возвращает для всех объектов одно и то же значение, то ее вряд ли можно назвать хорошей.

» Хэш-таблица

Хэш-таблица – это структура данных, реализующая интерфейс ассоциативного массива, а именно, она позволяет хранить пары (ключ, значение) и выполнять три операции: операцию добавления новой пары, операцию поиска и операцию удаления пары по ключу. Хэш-таблица содержит некоторый массив H, служащий для хранения пар ключ – значение. Выполнение операции в хэш-таблице начинается с вычисления хэш-функции от ключа. Вычисленный хэш-код является индексом в массиве H (обычно одному индексу в массиве H соответствует некоторый диапазон хэш-кодов, поэтому более правильно говорить, что по хэш-коду определяется индекс в массиве H). Затем выполняемая операция (добавление, удаление или поиск) перенаправляется объекту, который хранится в соответствующей ячейке массива H.

Ситуация, когда для различных ключей получается один и то же индекс, называется коллизией. Такие события не так уж и редки – например, при вставке в хэш-таблицу размером 365 ячеек всего лишь 23-х элементов вероятность коллизии уже превысит 50 % (если каждый элемент может равновероятно попасть в любую ячейку). Поэтому механизм разрешения коллизий – важная составляющая любой хэш-таблицы.

Число хранимых элементов, деленное на размер массива H, называется коэффициентом заполнения хэш-таблицы [load factor]. Это важный параметр: от него зависит среднее время выполнения операций.

Важное свойство хэш-таблиц состоит в том, что, при некоторых разумных допущениях, все три операции (поиск, вставка, удаление элементов) в среднем выполняются за время O(1). Но при этом не гарантируется, что время выполнения отдельной операции мало. Это связано с тем, что при достижении некоторого значения коэффициента заполнения необходимо осуществлять перестройку индекса хэш-таблицы: увеличить значение размера массива H и заново добавить в пустую хэш-таблицу все пары.