

Erlang: Практика

Суха теория, мой друг... **Андрей Ушаков** предлагает приложить руки к практическим решениям.



Наш
эксперт

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

Последние несколько статей мы занимались тем, что изучали, что такое многозадачные и распределенные системы и какие сложности нас подстерегают на пути создания таких систем. Помимо этого, мы изучали, какие в языке Erlang есть средства для решения подобных задач. Но на одной теории далеко не уехать: без практического опыта не создать более-менее сложную, многозадачную, и, тем более, распределенную систему. Именно поэтому мы открываем цикл статей, посвященных практикуму по созданию многозадачных и распределенных систем средствами языка Erlang.

Нашей задачей в данном практикуме будет написание многозадачных и, с какого-то момента, распределенных версий таких широко распространенных функций, как **map** и **reduce**. Решать данную задачу мы будем постепенно: начнем с обычных, не многозадачных реализаций данных функций и закончим реализацией, работающей в распределенной среде, т.е. на нескольких узлах. Возникает закономерный вопрос: а почему именно эти функции выбраны для нашего практикума? Как мы увидим ниже, реализация этих функций в простейшем, не многозадачном случае очень ясна. Поэтому при реализации разнообразных многозадачных вариантов этих функций мы практически не будем отвлекаться на детали, не относящиеся к многозадачности.

Прежде чем начать работу с примерами, немного поговорим о том, что представляют собой функции **map** и **reduce** и какие ограничения мы накладываем на наши реализации этих функций.

Функция **map** вычисляет результат операции отображения, которая каждому элементу **a** из исходного множества **A** ставит в соответствие элемент **fun(a)** из результирующего множества для заданной функции отображения **fun**. Понятно, что в качестве исходного множества элементов можно взять любой набор элементов, как упорядоченный, так и неупорядоченный, но мы в нашем примере в качестве такого набора всегда будем применять только список.

Функция **reduce** вычисляет результат (некоторое значение) операции свертки (или агрегирования) для заданного множества, функции свертки и начального значения. Примером такой операции является операция нахождения суммы множества чисел. Как и в случае функции **map**, в качестве исходного набора элементов мы будем использовать списки. Результат выполнения операции свертки может быть разным и зависит от того, в каком порядке мы берем элементы из множества, поскольку операция свертки бывает некоммутативной. Действительно, если в качестве исходного множества мы возьмем список матриц, а в качестве функции свертки – операцию умножения этих матриц, результат будет зависеть от порядка обхода списка: слева направо или справа налево. Именно по этой причине модуль **lists** содержит две функции для операции свертки: **lists:foldl/3** и **lists:foldr/3**. Мы во всех наших примерах при реализации операции свертки будем обходить список слева направо (порядок обхода списка справа налево тривиально реализуется по аналогии).

Есть и еще один момент, который связан с операцией свертки: это ассоциативность данной операции. Ассоциативность операции определяет, зависит ли результат операции от того, в каком порядке мы вычисляем результат этой операции, т.е. расставляем скобки. Так, например, операция сложения чисел является ассоциативной: это означает, что значение выражения **(1+2)+3** равно значению выражения **1+(2+3)**. А операция вычитания чисел ассоциативной не является: значение выражения **(1-2)-3** равно **-4**, а значение выражения **1-(2-3)** равно **2**. И опять же для простоты мы полагаем, что имеем дело с ассоциативной операцией свертки (почему нам важна ассоциативность, мы увидим ниже).

Скажем пару слов и об организации наших примеров. Все экспортируемые функции, которые относятся к операции отображения, располагаются в модуле **parallel_map** (и, соответственно, в файле **parallel_map.erl**). Все экспортируемые функции, которые относятся к операции свертки, располагаются в модуле **parallel_reduce** (и, соответственно, в файле **parallel_reduce.erl**).

Помимо этих двух модулей, мы будем определять и использовать дополнительные модули по мере необходимости. Одним из таких модулей, который будет использоваться практически везде, является модуль **parallel_common** (располагающийся в файле **parallel_common.erl**). Как мы увидим, этот модуль содержит общие для наших примеров функции.

А начнем мы наши примеры с обычных, не многозадачных версий функций **map** и **reduce**. Для реализации обычной версии функции **map** мы воспользуемся техникой конструирования списков [List comprehension]:

```
usual_map(_Fun, []) -> [];
usual_map(Fun, SourceList) -> [Fun(Element) || Element
-<- SourceList].
```

Как видно, реализация этой функции достаточно тривиальна (этую функцию можно было бы реализовать и рекурсивным образом, но реализация при этом стала бы несколько больше). Обычную версию функции **reduce** таким образом реализовать не получится – для этого нам потребуется рекурсивно работать с исходным списком:

```
usual_reduce(_Fun, [], InitValue) -> InitValue;
usual_reduce(Fun, [H | Rest], InitValue) ->
    NewAgg = Fun(H, InitValue),
    usual_reduce(Fun, Rest, NewAgg).
```

И опять же, реализация этой функции достаточно тривиальна.

Давайте проверим, что наши функции работают правильно. Для этого откомпилируем соответствующие модули и для вызовов функций, приведенных ниже, проверим, что результат их вызовов соответствует приведенным результатам. Вызов **parallel_map:usual_map(fun(X) -> X*X*2 end, [1, 2, 3, 4])** вернет список **[2, 4, 6, 8]**. Вызов **parallel_map:usual_map(fun(Str) -> " " ++ Str ++ " " end, ["aa", "bb", "cc"])** вернет список **["-aa-", "-bb-", "-cc-"]**. Те же самые результаты мы получим и при использовании функции **lists:map/2** из модуля **lists**. Результатом вызова **parallel_reduce:usual_**

Многозадачности

`reduce(fun(Item, Agg) -> Item + Agg end, [1, 2, 3, 4], 1)` будет число **11**. Результатом вызова `parallel_reduce:usual_reduce(fun(Item, Agg) -> Agg ++ Item end, ["aa", "bb"], "+")` будет строка **"++aabb"**. Такие же результаты мы получим и при использовании функции `lists:foldl/3` из модуля `lists`.

Реализовав обычные версии функций `map` и `reduce`, создадим их многозадачные версии. Мы начнем с простейшего случая для функции `map`: когда для отображения каждого элемента (т. е. для вычисления результирующего элемента `fun(a)` для каждого элемента `a` из исходного списка) из исходного множества мы используем отдельную задачу. В многозадачной версии функции `map` у нас будет один главный процесс, который создает дочерние рабочие процессы, раздает им задания и собирает результаты их работы, и несколько рабочих процессов (в нашем случае количество рабочих процессов равно количеству элементов в списке). Главный процесс (в котором мы инициируем выполнение нашей функции `map`) должен сделать следующее: создать задания для рабочих процессов, создать необходимое количество рабочих процессов, раздать всем этим процессам задание, получить результаты от всех рабочих процессов и объединить эти результаты в результирующем списке. Рабочие процессы в данной версии функции `map` являются «одноразовыми»: они получают задание, выполняют его, возвращают его и заканчивают свою работу. Создание задания для рабочих процессов заключается в преобразовании исходного списка в список пар (кортежей из двух элементов), состоящих из порядкового номера элемента и самого элемента. Мы задаем порядковые номера элементов, начиная с **0**; почему мы так делаем и зачем вообще нужны порядковые номера элементов, будет ясно чуть ниже. Для создания заданий для рабочих процессов мы создаем пару вспомогательных функций `simple_prepare_data/2` (интерфейсная функция) и `simple_prepare_data/3` (функция, решающая данную задачу) в модуле `parallel_map`:

```
simple_prepare_data([]) -> [];
simple_prepare_data(SourceList) -> simple_prepare_data(0, SourceList, []);
simple_prepare_data(Index, [Element], PreparedData) ->
    lists:reverse([{Index, Element}] ++ PreparedData);
simple_prepare_data(Index, [ElementRest], PreparedData) ->
    simple_prepare_data(Index + 1, Rest, [{Index, Element}] ++ PreparedData).
```

Принцип работы этой функции основан на рекурсии (а точнее, функции `simple_prepare_data/3`); при этом сама функция достаточно тривиальна, и детально разговаривать про нее мы не будем. Создание рабочих процессов и раздача им заданий можно объединить в нашем случае, т. к. у нас каждый процесс служит для выполнения только одного задания: применения функции отображения к одному из элементов исходного множества.

А теперь давайте поговорим о том, зачем нам нужно связывать с каждым из элементов его порядковый номер и почему мы нумерацию элементов начинаем с **0**. Обычная, не многозадачная версия функции `map` работает следующим образом: последовательно обходит все элементы исходного списка, для каждого элемента вычисляет значение функции отображения от этого эле-

Что такое список в языке Erlang

Если вы работали с такими языками программирования, как C#, Java, Python, где списки являются контейнерами с произвольным доступом (по индексу) к их элементам, то вы можете ожидать от списков в языке Erlang такого поведения, которого здесь у списков нет. Список в Erlang – это структура данных для хранения элементов (которые предполагается обрабатывать одинаковым образом), дающая доступ к головному элементу (или к нескольким головным элементам) и остатку. Это осуществляется посредством операции соответствия шаблону [pattern-matching] **[Head | Tail]** (или **[Head1,...,HeadN | Tail]**), если нам нужен доступ к нескольким головным элементам сразу), где **Head** – головной эле-

мент, **Tail** – остаток списка без головного элемента. Понятно, что работа со списком предполагает использование рекурсивного подхода (хвостовой рекурсии), когда в некоторой функции у списка выделяются головной элемент и остаток, головной элемент обрабатывается, и происходит вызов функции уже для остатка списка (и так, пока весь список не будет обработан). Список в языке Erlang по поведению подобен реализациям интерфейса `IEnumerable` в языке C#, `Iterable` в языке Java, итераторам в C++ и т. д. Если же в языке Erlang необходима структура данных с произвольным доступом к элементам (по индексу), следует использовать массивы, определенные в модуле `array`.

мента, и полученное значение добавляется к результирующему списку. Очень важно понять, что все это происходит последовательно! В случае же многозадачной версии, мы не можем гарантировать, что все задачи пришлют нам свои результаты работы в правильном порядке, даже если (как в нашем случае) мы их создавали (и запускали на выполнение) в правильном порядке. Решение этой проблемы достаточно простое и очевидное: необходимо с каждым элементом передавать его порядковый номер. Если функция рабочего процесса написана так, что она вместе с результатом работы возвращает и этот исходный порядковый номер, то мы сможем сохранять получаемые результаты по его порядковому номеру в некоторое хранилище. В момент готовности всех результатов работы их можно будет извлечь из хранилища и поместить в подходящую структуру данных. Хранилищами, которые позволяют сохранить некоторые значение по его порядковому номеру, являются массивы (определенные в модуле `array`). Так как нумерация элементов в массиве начинается с **0**, то именно по этой причине мы также начинаем нумеровать элементы из исходного списка с **0**. Ну, а подходящей структурой данных, как мы уже говорили, является список.

Функциональность (или средство), которая приостанавливает выполнение одной задачи, пока не будут получены все необходимые результаты от других задач, и собирает эти результаты, называется барьером. Данная функциональность у нас будет общей для нескольких вариантов реализаций функций `map` и `reduce`, поэтому ее реализация находится в паре функций `collect_result/2` (интерфейсная функция) и `collect_result/3` (функция, решающая данную задачу) модуля `parallel_common`:

```
collect_result(ResultStorage, TotalCount) -> collect_result(ResultStorage, TotalCount, 0).
collect_result(ResultStorage, TotalCount, TotalCount) -> ResultStorage;
```

>>

» Не хотите пропустить номер? Подпишитесь на [www.linuxformat.ru/subscribe/!](http://www.linuxformat.ru/subscribe/)

Сообщения об окончании жизни процесса

Если два процесса являются связанными и один из них – супервизор, то этот процесс-супервизор получит сообщение вида `{'EXIT', From, Reason}`, когда второй процесс закончит свою работу. Здесь `From` – идентификатор процесса, закончившего работу, `Reason` – причина, по которой процесс закончил работу. Если второй процесс закончит свою работу

естественным образом, т.е. выполнив свою рабочую функцию, то причиной будет атом `normal`; в противном случае `Reason` будет содержать информацию о произошедшей ошибочной ситуации. Если же два процесса являются связанными, и ни один из них не является супервизором, то в случае естественного завершения одного из процессов

другой об этом никак не узнает, а в случае завершения одного из процессов из-за ошибки второй процесс также будет завершен. Естественно, что все изложенное выше справедливо и для случая, когда связанных процессов несколько (при этом процессы-супервизоры будут получать сообщения об окончании жизни, а обычные процессы – нет).

```
collect_result(ResultStorage, TotalCount, ProcessedCount) ->
receive
    {'EXIT', _From, normal} -> collect_result(ResultStorage,
TotalCount, ProcessedCount);
    {'EXIT', _From, Reason} -> error(internal_error, Reason);
    {result, Index, DestElement} -> UpdatedResultStorage
= array:set(Index, DestElement, ResultStorage), collect_
result(UpdatedResultStorage, TotalCount, ProcessedCount + 1);
    _Other -> collect_result(ResultStorage, TotalCount,
ProcessedCount)
end.
```

Для сбора результатов работы рабочих процессов наша барьерная функция должна уметь взаимодействовать с рабочими процессами, а если точнее – принимать от них сообщения с результатами работы. В этих сообщениях нам необходимо знать индекс исходного элемента и результирующий объект; идентификатор рабочего процесса нам не нужен, т.к. используемые рабочие процессы являются «одноразовыми». Поэтому мы ожидаем от рабочих процессов сообщения вида `{result, Index, DestElement}`, где `Index` – индекс исходного элемента, `DestElement` – результирующий элемент. Помимо этого сообщения, мы также обрабатываем сообщения об изменении состояния рабочих процессов: завершился ли рабочий процесс обычным образом или из-за ошибки. В первом случае мы ничего не делаем, во втором – завершаем главный процесс с ошибкой. И, наконец, мы обрабатываем все остальные сообщения; т.к. они не имеют для нас смысла, мы их просто извлекаем из очереди сообщений главного процесса и ничего не делаем.

Теперь можно перейти непосредственно к телу рабочей функции главного процесса. Это функция `simple_pmap/2`, определенная в модуле `parallel_map`. Эта функция является и точкой входа в реализуемый нами вариант, т.е. экспортную:

```
simple_pmap(_Fun, []) -> [];
simple_pmap(Fun, SourceList) ->
process_flag(trap_exit, true),
MasterPid = self(),
ElementCount = length(SourceList),
PreparedData = simple_prepare_data(SourceList),
lists:foreach(fun({Index, Element}) -> spawn_link(fun() ->
simple_worker(Fun, Element, Index, MasterPid) end) end,
PreparedData),
EmptyStorage = array:new([{size, ElementCount}, {fixed, true},
{default, none}]),
FullStorage = parallel_common:collect_result(EmptyStorage,
ElementCount),
process_flag(trap_exit, false),
array:to_list(FullStorage).
```

Как уже говорилось выше, в рабочей функции главного процесса мы создаем задания для рабочих процессов, создаем необходимое количество рабочих процессов и раздаем им задания,

получаем результаты работы всех рабочих процессов и объединяем эти результаты в итоговом списке. Помимо этого, перед созданием рабочих процессов мы делаем главный процесс супервизором, а после сбора результатов работы рабочих процессов (после барьера) мы делаем главный процесс обычным процессом. Главный процесс обычно делают супервизором для того, чтобы отслеживать завершение вспомогательных процессов и, при необходимости, реагировать на это (например, если вспомогательный процесс завершился из-за ошибки, то перезапустить его). В нашем случае мы для процессов, завершившихся с ошибкой, генерируем ошибку более высокого уровня (но содержащую исходную ошибку в качестве дополнительной информации). Так делают, чтобы абстрагироваться от деталей реализации, но, тем не менее, позволяя эти детали получить при необходимости.

Нам осталось рассмотреть, что делают рабочие процессы для выполнения своего задания. Функция `simple_worker/4` из модуля `parallel_map` является рабочей функцией таких процессов:

```
simple_worker(Fun, SourceElement, Index, MasterPid) ->
DestElement = Fun(SourceElement),
MasterPid ! {result, Index, DestElement}.
```

Так как рабочие процессы у нас «одноразовые», то их рабочая функция имеет очень простой вид: выполнить задание и послать результат выполнения задания обратно главному процессу. Для нашей задачи, задание рабочего процесса – это просто вычислить значение функции отображения для заданного исходного элемента.

Давайте проверим, что созданный нами многозадачный вариант функции `map` работает правильно. Для этого компилируем соответствующие модули и запускаем консоль среды выполнения языка Erlang. В ней набираем `parallel_map:simple_pmap(fun(Item) -> 3*Item end, [1, 2, 5, 8])` и получаем в результате список, состоящий из утроенных элементов исходного списка `[3, 6, 15, 24]`. Аналогичным образом набираем `parallel_map:simple_pmap(fun(Item) -> lists:reverse(Item) end, ["str13", "str666"])` и получаем список, состоящий из обратных строк `["31rts", "666rts"]`. Таким образом, мы можем сделать вывод, что данный вариант многозадачной реализации функции `map` работает правильно.

С простой многозадачной версией функции `map` мы закончили. Теперь возникает вопрос: можем ли мы, руководствуясь теми же принципами, написать столь же простую многозадачную версию функции `reduce`? Для ответа на этот вопрос давайте рассмотрим принципиальное отличие операции отображения (метод `map`) от операции свертки (метод `reduce`). В операции отображения (`map`) мы обрабатываем все элементы независимо друг от друга; именно поэтому мы можем обрабатывать их все параллельно. В операции свертки (`reduce`) мы вычисляем одно значение по всем элементам; это означает, что мы не можем взять какой-либо элемент и работать с ним независимо от остальных элементов. Рассмотрим, например, как мы вычисляем сумму элементов в списке. У нас есть начальное значение суммы (обычно это `0`);

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

мы берем значение первого элемента и складываем его с начальным значением, потом полученный результат складываем со значением второго элемента и т. д. Видно, что мы не можем взять и одновременно обработать первый и второй элементы.

Может показаться на первый взгляд, что мы не сможем создать многозадачную версию функции **reduce**, но не стоить отчаиваться: нам поможет такое свойство операции, как ассоциативность. Как уже говорилось выше, ассоциативность операции определяет, зависит ли ее результат от того, в каком порядке мы его вычисляем. Другими словами, если операция ассоциативна, то мы можем расставить скобки (выделить подмножества элементов) так, как нам хочется, вычислить результат операции согласно расставленным скобкам, и получим в качестве результата всегда одно и то же значение. Так, например, сумма чисел **1+2+3+4+5+6** равна как сумме **(1+2)+(3+4)+(5+6)**, так и сумме **(1+2)+(3+4+5+6)**. Следует также сказать, что когда мы выделяем подгруппы для операции свертки, мы должны также задать начальное значение для операции свертки в подгруппе, или «ноль». Для нахождения суммы чисел этот «ноль» является числом **0**, для нахождения произведения чисел – **1**, для конкатенации строк – «» (или **[]**), для нахождения произведения матриц – единичная матрица, и т. д. Понимание этого факта важно по той причине, что такие функции свертки, как **lists:foldl/3** и **lists:foldr/3**, позволяют задать начальное значение для всей операции свертки, которое может отличаться от «ноля». Итак,

видно, что для создания многозадачной версии функции **reduce** нам необходимо сделать следующее: разбить исходной список на порции, порции обработать параллельно, после чего результаты параллельной обработки свернуть в итоговое значение.

Давайте еще поговорим о том, что делать, если операция свертки не ассоциативна. Возьмем, например, следующую разность: **1-2-3-4**, значение которой равно **-8**. Если мы сгруппируем элементы так, как мы это делали для суммы, то получим совсем другое значение. Так, например, группировка **(1-2)-(3-4)** дает значение **0**. Но с точки зрения арифметики это неправильно, а правлен один из следующих вариантов: **(1-2)-(3+4)** или **(1-2)+(-3-4)**. Давайте перепишем эти варианты следующим образом: **1-(2)-(3+4)** и **1+(-2)+(-3-4)**. Становится ясно, что для выполнения операции свертки по группам (как для вычитания чисел, так и в общем случае) нам необходимы две операции свертки элементов: исходная и некоторая дополнительная. При этом мы либо применяем исходную операцию свертки для свертки внутри групп, а дополнительную для свертки результатов свертки для групп, либо наоборот. Также видно, что первый элемент в операции свертки не входит ни в какую группу. Как итог: написание операции свертки с группами для неассоциативной операции свертки приводит к дополнительным сложностям, никак не связанным с многозадачностью. Чтобы избежать этих сложностей, реализовывать многозадачный вариант функции **reduce** для таких операций свертки мы не будем, как уже говорилось выше.

Итак, мы пришли к такому понятию, как разбиение данных на порции. Мы обычно разбиваем исходный набор элементов на порции тогда, когда есть возможность обработать данные параллельно, но обработка одного элемента данных невыгодна. В случае операции свертки (функции **reduce**), мы можем разбить список исходных данных на группы из одного элемента и обработать их, после чего полученные результаты свернуть в итоговый результат. Очевидно, что это невыгодно и бессмысленно. В случае операции отображения (функции **map**), если функция отображения простая (например, удвоение аргумента), то параллельная обработка всех элементов также невыгодна. И в этом случае гораздо выгоднее обрабатывать параллельно порции данных.

В связи с этим возникает вполне естественный вопрос: а как выбирать размер таких порций данных? Хотя мы и можем теоретически прикинуть размеры порций, обычно их размеры ищут экспериментальным путем на примерах типичных данных. Мы об этом еще поговорим в нашем практикуме.

Пора двигаться дальше. Но прежде чем браться за реализацию параллельной версии функции **reduce**, следует создать несколько полезных функций для разбиения данных на порции. Нас интересуют две задачи: вычисление количества порций данных по размеру одной порции (и размеру исходных данных) и собственно разбиение исходных данных на порции. Первая задача реализована в функции **calc_portion_count/2** модуля **parallel_common**, которая весьма тривиальна:

```
calc_portion_count(TotalSize, PortionSize) when TotalSize rem PortionSize == 0 ->
    TotalSize div PortionSize;
calc_portion_count(TotalSize, PortionSize) when TotalSize rem PortionSize /= 0 ->
    (TotalSize div PortionSize) + 1.
```

Эта функция учитывает тот факт, что если размеры исходных данных и порции не кратны, то у нас появляется остаток (размер которого меньше размера порции), который также необходимо учитывать. Разбиение исходного списка данных на порции реализовано в функциях **prepare_data/2**

(интерфейсная функция)

и **prepare_data/3** (функция, решающая данную задачу) модуля **parallel_common**:

```
prepare_data(_PortionSize, []) -> [];
prepare_data(PortionSize, SourceList) -> prepare_data(0, PortionSize, SourceList, []);
prepare_data(Index, PortionSize, SourceList, PreparedData)
when length(SourceList) =< PortionSize ->
    lists:reverse([{Index, SourceList}] ++ PreparedData);
prepare_data(Index, PortionSize, SourceList, PreparedData) ->
    {Portion, Rest} = lists:split(PortionSize, SourceList),
    prepare_data(Index + 1, PortionSize, Rest, [{Index, Portion}] ++ PreparedData).
```

Так же, как и в ситуации с простой многозадачной реализацией функции **map**, нам придется собирать результаты со всех рабочих процессов. Это означает, что нам точно так же необходимо связывать с порциями данных индексы, начинающиеся с **0** (только это уже будут индексы порций). Именно это и делают функции **prepare_data/2** и **prepare_data/3** – они создают список из пар (кортежей, состоящих из двух элементов: индекс порции и собственно порция), который мы используем в дальнейшем. Следует сказать, что все дальнейшие варианты многозадачных реализаций функций **map** и **reduce** будут использовать порции в качестве единицы работы для рабочих процессов.

На этом мы, пожалуй, сегодня остановимся: к сожалению, место в журнале для статьи ограничено. Давайте подведем промежуточный итог: мы увидели, что даже в самом простейшем случае (на примере многозадачной версии функции **map**) многозадачная версия больше и сложнее соответствующей не многозадачной версии функции. Мы увидели, что не всегда возможно подойти к задаче распараллеливания процесса вычисления в лоб: мы не всегда можем распараллелить обработку каждого элемента исходных данных. И мы начали рассматривать случай, когда параллельно у нас обрабатываются не единичные элементы, а порции. В следующем номере мы продолжим наш практикум и, в том числе, закончим создание многозадачных версий функций **map** и **reduce**, которые параллельно обрабатывают порции исходных данных. **LXF**