

Erlang: И снова

Андрей Ушаков считает, что одного практикума по многозадачности маловато – и проводит второй.



Наш
эксперт

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

В прошлый раз в нашем практикуме по многозадачности мы решили создать многозадачные версии таких популярных в программировании функций, как `map` (операция отображения) и `reduce` (операция свертки). Эти функции были выбраны не случайно: их обычные, не многозадачные реализации настолько просты, что при создании их многозадачных версий нечестно сильно отвлекаться на детали, не связанные с многозадачностью. Простоту их мы показали в прошлый раз, создав их обычную (не многозадачную) реализацию. Также мы создали простую многозадачную версию функции `map`, использующую для отображения каждого элемента свой рабочий процесс. При таком подходе (по процессу на элемент) создание многозадачной версии функции `reduce` не имеет смысла. Для создания многозадачной версии функции `reduce` (и более гибкой многозадачной версии функции `map`) надо разбивать данные на порции и эти порции обрабатывать параллельно, после чего формировать итоговый результат. На этом мы и остановились в прошлый раз.

Снова рассмотрим вспомогательные функции, созданные в прошлый раз для работы с порциями данных. Эти функции импортируются из модуля `parallel_common`. Функция `calc_portion_count/2` позволяет вычислить количество порций данных по заданным размерам порции и исходного списка с данными:

```
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.
```

Пара функций `prepare_data/2` (интерфейсная функция) и `prepare_data/3` (функция, реализующая данную функциональность) разбивают исходные данные на порции:

```
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).
```

Исходный список разбивается на список пар (кортежей), состоящих из индекса порции и собственно самой порции с данными.

Теперь мы готовы двигаться дальше. В многозадачной версии функции `reduce` у нас будет один главный процесс и несколько вспомогательных рабочих процессов. Главный процесс (в котором мы инициируем выполнение нашей версии функции `reduce`) отвечает за разбиение данных на порции, создание необходимого количества рабочих процессов и заданий для них, сбор результатов работы от всех рабочих процессов и, наконец, свертку результатов работы вспомогательных процессов в итоговый результат. Как и при реализации функции `parallel_map: simple_pmap/2`, мы используем «одноразовые» рабочие процессы. При таком подходе можно раздать задания сразу же при создании процессов. А рабочая функция «одноразовых» процессов крайне проста:

вычислить результат свертки для порции и отослать его главному процессу. После создания всех «одноразовых» рабочих процессов все взаимодействие с ними сводится только к получению от них результатов и сохранению этих результатов в промежуточном буфере (в массиве). Для этого мы можем использовать функцию `parallel_common: collect_result/2`, только вместо сохранения отдельных элементов после отображения (что мы делали, когда реализовывали простейший многозадачный вариант функции `map`) мы будем сохранять результаты свертки для группы элементов. Подведя итог вышесказанному, мы можем написать рабочую функцию главного процесса (она же точка входа в многозадачную версию функции `reduce`) следующим образом:

```
portion_reduce(_Fun, [], _InitValue, _PortionInitValue), _PortionSize) -> InitValue;
portion_reduce(Fun, SourceList, {InitValue, _PortionInitValue}, PortionSize)
when length(SourceList) <= PortionSize ->
lists:foldl(Fun, InitValue, SourceList);
portion_reduce(Fun, SourceList, {InitValue, PortionInitValue}, PortionSize) ->
process_flag(trap_exit, true),
MasterPid = self(),
PortionCount = parallel_common:calc_portion_count(length(SourceList), PortionSize),
PreparedData = parallel_common:prepare_data(PortionSize, SourceList),
lists:foreach(fun({Index, Portion}) -> spawn_link(fun() ->
portion_worker(Fun, Portion, PortionInitValue, Index, MasterPid) end) end, PreparedData),
EmptyStorage = array:new([{size, PortionCount}, {fixed, true}, {default, none}]),
FullStorage = parallel_common:collect_result(EmptyStorage, PortionCount),
process_flag(trap_exit, false),
lists:foldl(Fun, InitValue, array:to_list(FullStorage)).
```

Эта функция содержит три варианта. Первый обрабатывает наиболее тривиальный случай, когда исходный список пуст, и мы возвращаем начальное значение операции свертки. Второй вариант обрабатывает ситуацию, когда количество элементов в списке не больше размера порции; тогда нет смысла распараллеливать выполнение функции, и операция свертки выполняется при помощи функции `lists:foldl/3`. И, наконец, последний вариант обрабатывает наиболее общий случай, который и является многозадачным. При этом не стоит забывать, что при помощи рабочих процессов мы формируем только результаты операции свертки по подгруппам; финальную операцию свертки промежуточных результатов мы выполняем в конце рабочей функции главного процесса. И, конечно, мы помним, что функция `portion_reduce/3` является экспортируемой функцией модуля `parallel_reduce`.

Как говорилось в предыдущей статье, при реализации параллельного варианта функции `reduce` нам необходимо задавать пару начальных значений, в отличие от обычного варианта функции `reduce`. Одно значение из этой пары является начальным

практикум

значением всей операции свертки, другое же является «нулем» операции свертки; оно необходимо для выполнения операции свертки в подгруппах. Так, например, если операцией свертки является операция суммирования чисел, то «нулем» будет число **0**, а если операцией свертки будет операция конкатенации строк, «нулем» операции будет пустая строка. В общем случае, что будет являться «нулем», для операции свертки не вычислить. Поэтому пользователь нашей многозадачной версии функции **reduce** должен передавать как начальное значение операции свертки, так и «нуль» этой операции. В нашей реализации мы ожидаем, что эти два значения будут переданы в виде пары (кортежа) **{InitValue, PortionInitValue}**, где **InitValue** – начальное значение всей операции свертки, **PortionInitValue** – «нуль» операции свертки.

Теперь посмотрим на то, чем же занимаются у нас рабочие процессы (т. е. на функцию, выполняемую рабочими процессами). Это функция **portion_worker/5**, определенная в модуле **parallel_reduce** (но не экспортируемая из этого модуля):

```
portion_worker(Fun, SourcePortion, InitValue, Index, MasterPid)->
    AggrValue = lists:foldl(Fun, InitValue, SourcePortion),
    MasterPid ! {result, Index, AggrValue}.
```

Как мы уже говорили ранее, рабочие процессы у нас «одноразовые», т. е. выполняют свою задачу и заканчивают работу. Именно поэтому функция, которую выполняют рабочие процессы, настолько проста: в ней мы вычисляем значение операции свертки для группы и посылаем сообщение главному процессу с вычисленным значением. Сообщение имеет вид **{result, Index, AggrValue}**, где **Index** – индекс исходной порции данных, **AggrValue** – значение операции свертки для исходной порции. В качестве начального значения **InitValue** для операции свертки порции данных мы передаем «нуль» операции свертки (при создании рабочих процессов).

Пришла пора проверить, что наша многозадачная версия функции **reduce** работает правильно. Для этого компилируем соответствующие модули и запускаем консоль среды выполнения языка Erlang. Так как у нас функция **parallel_reduce:portion_reduce/4** имеет три варианта, все эти три варианта было бы неплохо проверить. Для начала в качестве операции свертки возьмем операцию сложения чисел. Вызов **parallel_reduce:portion_reduce(fun(Item, Agg) -> Item + Agg end, [], {3, 0}, 5)** вернет нам в качестве значения число **3**, т. е. начальное значение операции свертки. Вызов **parallel_reduce:portion_reduce(fun(Item, Agg) -> Item + Agg end, [1, 2, 3], {3, 0}, 5)** вернет число **9**. Это сумма всех чисел из списка **[1, 2, 3]** с величиной **3** в качестве начального значения суммы. Так как количество элементов в списке – **3**, а размер порции – **5**, то сумма всех чисел вычисляется при помощи второго варианта функции **parallel_reduce:portion_reduce/4**, т. е. не многозадачным способом. Выполняем вызов **parallel_reduce:portion_reduce(fun(Item, Agg) -> Item + Agg end, [1, 2, 3, 4, 5, 6], {3, 0}, 2)** и получаем в результате значение **24**. Легко проверить, что сумма всех чисел из списка с начальным значением этой суммы **3** будет равна **24**. В этом вызове размер порции равен **2**, а количество элементов в списке – **6**; так что при вызове будут созданы **3** рабочих процесса.

Предыдущая операция свертки была коммутативной операцией; давайте проверим работу нашей многозадачной версии

Арность функции

Арность функции – это количество аргументов, которые необходимо передать функции во время ее вызова. В языке Erlang арность функции обозначается следующим образом: **func_name/func_arity**, где **func_name** – имя функции, а **func_arity** – арность функции. Так, например, **some_func/0** обозначает функцию, арность которой **0** (это означает, что при ее вызове не надо передавать никаких аргументов), а **some_func/3** – функцию, арность которой **3** (это означает, что при вызове функции

необходимо передать **3** аргумента). Так как Erlang – язык динамический (то есть типы переменных определяются в нем во время выполнения), то перегрузка некоторой функции возможна, только если функции с одним и тем же именем (определенные в одном модуле) имеют разное количество аргументов, т. е. арность. Следует также добавить, что некоторая функция может состоять из нескольких вариантов, что тоже в своем роде некоторая перегрузка этой функции.

функции **reduce** в случае, когда операция свертки коммутативной не является. В качестве такой некоммутативной операции свертки возьмем операцию конкатенации строк. Как и в предыдущем случае, необходимо проверить работу всех трех вариантов функции **parallel_reduce:portion_reduce/4**. Вызов **parallel_reduce:portion_reduce(fun(Item, Agg) -> Agg ++ Item end, [], {"++"}, 2)** вернет нам в качестве значения строку **“++”**, т. е. начальное значение операции конкатенации. Результатом выполнения вызова **parallel_reduce:portion_reduce(fun(Item, Agg) -> Agg ++ Item end, ["aa", "bb"], {"++"}, 5)** будет строка **“++aabbb”**. Легко проверить, что конкатенация всех строк из исходного списка с начальным значением **“++”** даст нам строку **“++aabbb”**. При этом количество элементов в списке равно **2**, а размер порции равен **5**; это означает, что результат конкатенации строк мы получили при помощи второго варианта функции **parallel_reduce:portion_reduce/4**. И, наконец, результатом выполнения вызова **parallel_reduce:portion_reduce(fun(Item, Agg) -> Agg ++ Item end, ["aa", "bb", "cc", "dd", "ee", "ff"], {"++"}, 2)** будет строка **“++aabbcdeeff”**. Очевидно, что эта строка является результатом конкатенации всех строк из исходного списка, с начальным значением **“++”**. Так как размер списка – **6** элементов, а размер порции данных – **2**, то этот вызов будет обработан третьим вариантом функции **parallel_reduce:portion_reduce/4**; при этом будет создано **3** рабочих потока.

Теперь создадим многозадачную версию функции **map**, рабочие процессы которой будут обрабатывать не отдельные элементы исходного списка, а порции из элементов. Как и раньше, у нас будет один главный процесс и несколько вспомогательных рабочих процессов. В главном процессе (в котором мы инициируем выполнение нашей версии функции **map**) мы будем разбивать данные на порции, создавая необходимое количество рабочих процессов и заданий для них, собирать результаты работы от всех рабочих процессов и, наконец, преобразовывать собранные промежуточные результаты в итоговый список. При этом рабочие процессы у нас продолжают быть «одноразовыми», т. е. они выполняют свою задачу, отсылают результаты работы главному процессу и заканчивают свое существование в этом бренном

>>

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

мире. Поэтому мы можем не останавливаться подробно на принципах работы, а сразу посмотреть код соответствующих методов. Рабочая функция главного процесса (она же точка входа в многозадачную версию функции `map`) `portion_pmap/3` определена в модуле `parallel_map` (и, естественно, объявлена экспортруемой функцией из этого модуля):

```
portion_pmap(_Fun, [], _PortionSize) -> [];
portion_pmap(Fun, SourceList, PortionSize)
when length(SourceList) =< PortionSize ->
    lists:map(Fun, SourceList);
portion_pmap(Fun, SourceList, PortionSize) ->
    process_flag(trap_exit, true),
    MasterPid = self(),
    PortionCount = parallel_common:calc_portion_count(length(SourceList), PortionSize),
    PreparedData = parallel_common:prepare_data(PortionSize, SourceList),
    lists:foreach(fun({Index, Portion}) -> spawn_link(fun() ->
        portion_worker(Fun, Portion, Index, MasterPid) end) end,
    PreparedData),
    EmptyStorage = array:new([{size, PortionCount}, {fixed, true},
    {default, none}]),
    FullStorage = parallel_common:collect_result(EmptyStorage,
    PortionCount),
    process_flag(trap_exit, false),
    lists:append(array:to_list(FullStorage)).
```

Как и в случае функции `parallel_reduce:portion_reduce/4`, мы определяем три варианта функции. Первый обрабатывает случай пустого списка; второй – случай, когда количество элементов в списке меньше размера порции (в этом случае нет смысла распараллеливать задачу); и, наконец, третий – общий случай.

Рассмотрим функцию, которую будут выполнять рабочие процессы. Это функция `portion_worker/4`, определенная в модуле `parallel_map` (но не экспортруемая из него):

```
portion_worker(Fun, SourcePortion, Index, MasterPid) ->
    DestPortion = lists:map(Fun, SourcePortion),
    MasterPid ! {result, Index, DestPortion}.
```

Так как рабочие процессы у нас опять же «одноразовые», то выполняемая ими функция очень проста: мы вычисляем результат операции отображения порции и возвращаем его главному процессу при помощи посылки сообщения вида `{result, Index, DestPortion}`, где `Index` – индекс исходной порции данных, `DestPortion` – результат применения операции отображения к исходной порции.

Проверим, что новая многозадачная версия функции `map` работает правильно. Для этого скомпилируем соответствующие модули и запустим консоль среды выполнения языка Erlang. Так как у функции `parallel_map:portion_pmap/3` определено три варианта, необходимо проверить работу их всех. Начнем с первого варианта, обрабатывающего случай пустого исходного списка. Вызов `parallel_map:portion_pmap(fun(Item) -> lists:reverse(Item) end, [], 4)` возвращает пустой список, как и ожидалось. Второй вариант – когда количество элементов в списке меньше размера порции (тогда дополнительных процессов мы не создаем, а все вычисления выполняем в вызывающем процессе). Вызов `parallel_map:portion_pmap(fun(Item) -> lists:reverse(Item) end, ["str13", "str667"], 4)` вернет список `[<31rts>, "766rts"]`, как и ожидается. Так как список содержит 2 элемента, а размер порции 4, то мы можем быть уверены, что наш вызов будет выполнен вторым вариантом функции `parallel_map:portion_pmap/3`. И, наконец, проверим последний вариант функции `parallel_map:portion_pmap/3`. Вызов `parallel_map:portion_pmap(fun(Item) -> lists:reverse(Item) end, ["str13", "str67", "str667", "str909"], 2)` вернет нам список `[<31rts>, "76rts", "766rts", "909rts"]`. Очевидно, что это значение является правильным. При последнем вызове размер порции – 2, а исходный список содержит 4 элемента; то есть создутся два рабочих потока для выполнения операции отображения каждой порции данных.

Давайте внимательно посмотрим на созданные нами выше многозадачные версии функций `map` и `reduce` и сравним, как эти методы реализуют свою функциональность (сравнивать, что эти методы делают, очевидно, не имеет смысла). В обоих случаях у нас есть главный процесс (он же процесс, в котором инициировался вызов функции) и несколько вспомогательных «одноразовых» рабочих процессов. В обоих случаях в главном процессе мы разбиваем данные на порции, создаем нужное количество рабочих процессов и заданий для них, собираем результаты работы от всех рабочих процессов и, наконец, преобразовываем собранные промежуточные результаты в итоговый результат. А в рабочих процессах, опять же в обоих случаях, мы выполняем задание, отсылаем результат работы обратно главному процессу, и все.

Видно, что оба эти метода организуют свою работу одинаковым образом. Они отличаются только функцией, обрабатывающей порции данных в рабочих процессах, и функцией, преобразовывающей промежуточные результаты в итоговый результат. В реализации функции `map` для обработки порций данных в рабочем процессе используется функция `lists:map/2`, а для преобразования промежуточных результатов в итоговый результат –

Зачем нужен `append`

Почему мы используем `list:append/1` для объединения промежуточных результатов в итоговый результат в методе `parallel_map:portion_pmap/3` (как и в методе `parallel_map:portion_gen_pmap/3`)?

Порция исходных данных (которую обрабатывает рабочий процесс) является списком, после обработки порции рабочим процессом мы также получаем список (т. к. мы реализовываем многозадачную версию операции отображения), поэтому коллекция промежуточных результатов (а это у нас массив) хранит списки в качестве своих элементов. Функция `array:to_list/1` возвращает список элементов, которые хранились в массиве; в нашем случае это будет список, элементами которого будут списки. Соответственно, возникает задача о пре-

образовании такого «глубокого» списка в плоский список. В общем случае эта задача решается при помощи функции `lists:flatten/1`, но в нашем случае (когда у нас список, элементами которого являются списки из элементов) эту задачу также можно решить при помощи функции `lists:append/1`.

Возникает вопрос: почему мы не использовали для решения этой задачи более очевидную функцию `lists:flatten/1`, а использовали функцию `lists:append/1`? Ответ на этот вопрос достаточно неожиданный: все дело в строках. Как мы помним (см. [LXF149](#)), строки не являются отдельным типом данных; вместо этого строки являются списками, элементами которых являются символы (или целые положительные числа, что то же самое). Поэтому,

если элементами являются строки, то результат выполнения функции `lists:flatten/1` даст совсем не тот результат, который ожидается. Так, например, результатом выражения `lists:flatten([["aa", "bb"], ["cc"]])` будет строка `«aabbcc»`, тогда как мы ожидали список из строк `[«aa», «bb», «cc»]`. Именно поэтому мы и используем функцию `lists:append/1`: она не имеет подобных побочных эффектов при работе со строками. Мораль всего этого следующая: необходимо с осторожностью подходить к использованию некоторых функций для работы со списками (таких как `lists:flatten/1`), если элементами списка могут быть строки, т. к. вы можете получить не тот результат, на который рассчитывали.

» **Пропустили номер?** Узнайте на с. 108, как получить его прямо сейчас.

функция `lists:append/1`. В реализации функции `reduce` в обоих случаях применяется функция `lists:foldl/3`, только с разными параметрами. Логично спросить: а зачем мы сейчас сравниваем наши реализации функций `map` и `reduce`? Ответ, пожалуй, очевиден: во-первых, мы хотим избежать дублирования кода в уже существующих реализациях методов `map` и `reduce`, а во-вторых, хотим в дальнейшем создавать параллельные версии еще каких-либо методов с минимальными затратами. В нашем случае видно, что мы можем выделить общую часть в реализации методов `parallel_map:portion_pmap/3` и `parallel_reduce:portion_reduce/4`. Все выделяемые общие методы мы помещаем в модуль `parallel_portion_helper` (и, соответственно, в файл `parallel_portion_helper.erl`).

Начнем с функции, которую выполняют рабочие процессы. Для создания ее общей версии достаточно заменить конкретные функции, используемые для обработки порций в рабочих процессах на некоторую функцию, передаваемую как параметр:

```
portion_worker(Fun, SourcePortion, Index, MasterPid) ->
    DestPortion = Fun(SourcePortion),
    MasterPid ! {result, Index, DestPortion}.
```

Здесь параметр `Fun` и является той самой функцией, которая обрабатывает входную порцию данных. Следует заметить, что эта функция не экспортируется из модуля `parallel_portion_helper`. Теперь выделим общую часть из рабочих функций главных процессов (и, соответственно, точек входа) для наших реализаций методов `map` и `reduce`. Как уже говорилось, рабочие функции главных процессов отличаются только следующими аспектами: функцией, которая обрабатывает порции данных в рабочих процессах, и функцией, которая преобразовывает промежуточные результаты в итоговый результат. Очевидно, что общая часть должна содержать обе эти функции в качестве параметров. С учетом всего сказанного, общая часть рабочей функции главного потока должна выглядеть следующим образом (естественно, что эта функция является экспортируемой из модуля `parallel_portion_helper`):

```
portion_core(WorkerFun, FinalAggrFun, SourceList, PortionSize) ->
    process_flag(trap_exit, true),
    MasterPid = self(),
    PortionCount = parallel_common:calc_portion_count(length(SourceList), PortionSize),
    PreparedData = parallel_common:prepare_data(PortionSize, SourceList),
    lists:foreach(fun({Index, Portion}) -> spawn_link(fun() ->
        portion_worker(WorkerFun, Portion, Index, MasterPid) end) end,
    PreparedData),
    EmptyStorage = array:new([{size, PortionCount}, {fixed, true},
    {default, none}]),
    FullStorage = parallel_common:collect_result(EmptyStorage, PortionCount),
    process_flag(trap_exit, false),
    FinalAggrFun(array:to_list(FullStorage)).
```

Здесь параметр `WorkerFun` является функцией, которая применяется для обработки порций исходных данных в рабочих процессах, а параметр `FinalAggrFun` – функция для преобразования промежуточных результатов в итоговый результат. Помимо этих двух параметров, эта функция принимает также исходный список `SourceList` и размер порции `PortionSize`. Может возникнуть вопрос: можем ли мы вынести в метод `parallel_portion_helper:portion_core/4` дополнительные варианты, которые были в методах `parallel_map:portion_pmap/3` и `parallel_reduce:portion_reduce/4` и предназначались для обработки ситуаций, когда исходный список либо пуст, либо его размер не больше размера порции? Для ответа сравним варианты функций `parallel_map:portion_pmap/3` и `parallel_reduce:portion_reduce/4`, которые обрабатывают ситуацию пустого списка с данными. Функция `parallel_map:portion_pmap/3` в этом случае возвращает пустой список, а функция `parallel_reduce:portion_reduce/4` – начальное зна-

чение операции свертки. Видно, что для этого варианта возвращаемые значения специфичны. Точно так же можно увидеть специфику и в вариантах функций `parallel_map:portion_pmap/3` и `parallel_reduce:portion_reduce/4`, которые обрабатывают ситуацию, когда размер списка не превышает размера порции. Именно по этой причине таких вариантов нет в функции `parallel_portion_helper:portion_core/4`.

Теперь посмотрим, как можно переписать функции `parallel_map:portion_pmap/3` и `parallel_reduce:portion_reduce/4`, используя созданную функцию `parallel_portion_helper:portion_core/4`. Новые (переписанные) функции будут называться `parallel_map:portion_gen_pmap/3` и `parallel_reduce:portion_gen_reduce/4`, соответственно. Реализация `parallel_map:portion_gen_pmap/3` функции выглядит следующим образом:

```
portion_gen_pmap(_Fun, [], _) -> [];
portion_gen_pmap(Fun, SourceList, PortionSize)
when length(SourceList) =< PortionSize ->
    lists:map(Fun, SourceList);
portion_gen_pmap(Fun, SourceList, PortionSize) ->
    WorkerFun = fun(SourcePortion) -> lists:map(Fun, SourcePortion) end,
    parallel_portion_helper:portion_core(WorkerFun, fun
lists:append/1, SourceList, PortionSize).
```

Здесь функция `WorkerFun` создается на основе функции `lists:map/2`, а функцией `FinalAggrFun` является функция `lists:append/1`. Аналогичным образом выглядит реализация функции `parallel_reduce:portion_gen_reduce/4`:

```
portion_gen_reduce(_Fun, [], {InitValue, _PortionInitValue}, _PortionSize) -> InitValue;
portion_gen_reduce(Fun, SourceList, {InitValue, _PortionInitValue}, PortionSize)
when length(SourceList) =< PortionSize ->
    lists:foldl(Fun, InitValue, SourceList);
portion_gen_reduce(Fun, SourceList, {InitValue, PortionInitValue}, PortionSize) ->
    ReduceFun = fun(List) -> lists:foldl(Fun, InitValue, List) end,
    PortionReduceFun = fun(List) -> lists:foldl(Fun, PortionInitValue, List) end,
    parallel_portion_helper:portion_core(PortionReduceFun, ReduceFun, SourceList, PortionSize).
```

Здесь и функция `WorkerFun`, и функция `FinalAggrFun` создаются на основе функции `lists:foldl/3`; главное отличие между этими построениями в том, что в качестве параметра `Acc0` (начальное значение) в первом случае берется «нуль» операции свертки, а во втором – начальное значение операции свертки.

Для тестирования работы функций `parallel_map:portion_gen_pmap/3` и `parallel_reduce:portion_gen_reduce/4` мы можем использовать те же сценарии, что и для тестирования функций `parallel_map:portion_pmap/3` и `parallel_reduce:portion_reduce/4`. Поэтому дублировать результаты тестирования функций `parallel_map:portion_gen_pmap/3` и `parallel_reduce:portion_gen_reduce/4` мы здесь не будем (желающие могут провести его сами и убедиться, что все работает, как надо).

Сегодня мы сделали очередной шаг в нашем практикуме: создали многозадачные версии функций `map` и `reduce`, которые обрабатывают порции исходных данных параллельно. Настраивая размер порции, мы можем управлять производительностью наших функций (мы об этом поговорим в одном из следующих номеров). Более того, мы смогли найти и выделить общее ядро у соответствующих версий функций `map` и `reduce`. Важность этого результата в том, что при создании всех дальнейших многозадачных версий функций `map` и `reduce` мы будем создавать для них некоторое общее ядро. Это поможет избежать дублирования кода и облегчит создание многозадачных версий каких-либо еще функций (но об этом мы поговорим в следующий раз). [\[XF\]](#)