

Erlang: Практика

Андрей Ушаков в третий раз приступает к упражнениям с функциями.



Наш
эксперт

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

В прошлый раз мы создали многозадачные версии функций `map` и `reduce`, реализующие порционную обработку исходных данных. Сегодня мы усложним наш пример и избавимся от главного недостатка предыдущей реализации.

Припомним, на чем мы тогда остановились. Мы выделили общую часть из функций `parallel_map:portion_pmap/3` и `parallel_reduce:portion_reduce/4` в виде функции `parallel_portion_helper:portion_core/4` и вспомогательной функции `portion_worker/4`, определенной в модуле `parallel_portion_helper`, но не экспортруемой из него. Функция `parallel_portion_helper:portion_core/4` выполняет всю основную работу: разбивает данные на порции, создает рабочие процессы и раздает им задания, собирает результаты трудов рабочих процессов и объединяет их в итоговый результат. Чтобы эту функцию можно было применять для построения любых функций, способных параллельно обрабатывать списки данных, нужно параметризовать ее. Для этого надо задать две функции: функцию `WorkerFun` для обработки порций данных в рабочих процессах и функцию `FinalAggrFun` для объединения обработанных данных в итоговый результат. Помимо параметров `WorkerFun` и `FinalAggrFun`, мы должны также передать в функцию `parallel_portion_helper:portion_core/4` исходный список с данными `SourceList` и размер порции `PortionSize`, обрабатываемой одним рабочим процессом.

```
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)).
```

Функция `portion_worker/4`, определенная в модуле `parallel_`

`portion_helper`, является телом каждого рабочего процесса, т. е. функцией, которую каждый рабочий процесс выполняет во время своей жизни. Так как на данный момент мы используем модель «одноразовых процессов», то функция `portion_worker/4` работает по очень простой схеме: обработать порцию исходных данных и отослать результат обработки обратно главному процессу. Естественно, что функция `portion_worker/4` должна быть параметризована функцией `Fun` для обработки порции исходных данных.

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

Параллельные версии функций `map` и `reduce`, созданные с использованием функции `parallel_portion_helper:portion_core/4` (и вспомогательной функции `portion_worker/4`), выглядят так:

```
portion_gen_pmap(_Fun, [], _PortionSize) -> [];
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).
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).
```

Видно, что функции `parallel_map:portion_gen_pmap/3` и `parallel_reduce:portion_gen_reduce/4` получаются достаточно просто. Они могли бы быть еще проще, если бы мы могли вынести обработку не параллельных вариантов в функцию `parallel_portion_helper:portion_core/4`; но как мы показали в прошлый раз, сделать это невозможно.

Внимательные читатели здесь могут вспомнить: мы говорили, что создание функции `parallel_portion_helper:portion_core/4` позволит избавиться от дублирования кода, а также позволит легко реализовывать многозадачные версии других функций. И если факт избавления от дублирования кода мы видели на примере функций `parallel_map:portion_gen_pmap/3` и `parallel_reduce:portion_gen_reduce/4`, то о возможности легко реализовать многозадачную версию какой-либо другой функции мы в прошлый раз только говорили (в связи с ограничением на размер статьи). Давайте покажем, что такая возможность действительно есть; и в качестве такой демонстрации создадим многозадачную версию для операции фильтрации (для функции `filter`). Операция

Флаг +Р

При создании экземпляра среды выполнения Erlang (программа `erl` на Linux и `erl.exe` на MS Windows) можно задать максимально возможное количество процессов, которые могут быть созданы одновременно; значение этого параметра должно лежать в диапазоне от 16 до 134217727. Значение по умолчанию – 32768 (именно столько процессов среда выполнения Erlang позволяет одновременно создать, если не задавать ключ +Р).

зать ключ “+P Number”. Параметр `Number` задает максимальное количество процессов, которые могут быть созданы одновременно; значение этого параметра должно лежать в диапазоне от 16 до 134217727. Значение по умолчанию – 32768 (именно столько процессов среда выполнения Erlang позволяет одновременно создать, если не задавать ключ +Р).

МНОГОЗАДАЧНОСТИ

Расправа с неизвестными

Каждый процесс в языке Erlang имеет свою собственную очередь сообщений, в которую и попадают сообщения, адресованные этому процессу. Когда процесс инициирует получение сообщения (при помощи выражения `receive`), он просматривает по очереди все сообщения в очереди сообщений и выбирает первое, которое подходит под одно из условий, заданных в выражении `receive`. Соответственно, все сообщения, просмотренные до того, как было выбрано подходящее сообщение, остаются в очереди сообщений процесса. Вполне воз-

можно, что такие сообщения будут обработаны позже, но не исключена ситуация, когда процессу приходят не известные ему сообщения. Такие сообщения будут оставаться в очереди сообщений, замедляя поиск подходящих сообщений в выражении `receive` и увеличивая память, потребляемую процессом (своего рода утечка памяти). Чтобы избежать подобных проблем, необходимо всегда принимать (и отбрасывать) неизвестные сообщения. Делается это следующим образом: в цикле обработки сообщений в конец выражения `receive` необ-

ходимо добавить условие, которому соответствуют все выражение и в котором мы ничего не делаем, а переходим на начало цикла обработки сообщений. Например, если цикл обработки сообщений содержитя в функции **MessageHandler**, то выражение **receive**, которое обрабатывает также и неизвестные сообщения, будет выглядеть так:

```
receive  
...  
_Other -> MessageHandler(...)  
end
```

фильтрации для набора исходных элементов (в нашем случае, для списка) возвращает только элементы, удовлетворяющие некоторому условию, заданному функцией-предикатом. В многозадачной версии операции фильтрации мы должны разбить исходный список на порции, к каждой порции применить операцию фильтрации (именно это действие мы можем выполнять параллельно), после чего полученные результаты объединить в итоговый.

При этом следует учесть два момента. Во-первых, результатом операции фильтрации является список такого же или меньшего размера (в качестве результата мы можем получить и пустой список). Во-вторых, операция фильтрации должна сохранять порядок отфильтрованных элементов относительно друг друга, так как мы фильтруем элементы из упорядоченного списка. С учетом сказанного выше, для реализации операции фильтрации с помощью функции `parallel_portion_helper:portion_core/4` параметр `WorkerFun` должен быть функцией, созданной на основе функции `lists:filter/2` и функции-предиката, а параметр `FinalAggrFun` должен быть функцией `lists:append/1`. Кому-то может показаться неочевидным выбор функции `lists:append/1` в качестве параметра `FinalAggrFun`. Однако причины такого выбора аналогичны таковым для многозадачной реализации функции `map` (см. [LXF165/166](#)).

Перейдем к реализации многозадачной версии операции фильтрации. Это будет функция `parallel_filter:portion_gen_filter/3`, экспортная из модуля `parallel_filter`:

```
portion_gen_filter(_Fun, [], _PortionSize) -> []
portion_gen_filter(Fun, SourceList, PortionSize)
when length(SourceList) <= PortionSize ->
    lists:filter(Fun, SourceList);
portion_gen_filter(Fun, SourceList, PortionSize)
    WorkerFun = fun(SourcePortion) -> lists:filter(
        Fun, SourcePortion) end,
parallel_portion_helper:portion_core(Worker
    lists:append(1, SourceList, PortionSize).
```

Как и в случае функций `parallel_map:portion_gen_pmap/3` и `parallel_reduce:portion_gen_reduce/4`, тело функции `parallel_filter:portion_gen_filter/3` состоит из трех вариантов. Первый вариант обрабатывает ситуацию пустого списка исходных данных. Второй обрабатывает ситуацию, когда размер исходных данных

не превышает размера порции, на которые разбиваются исходные данные. И, наконец, третий вариант является общим вариантом и обрабатывает все остальные ситуации.

Мы реализовали многозадачную версию операции фильтрации (функцию `parallel_filter:portion_gen_filter/3`); теперь пора проверить, что наша реализация работает правильно. Как уже говорилось выше, функция `parallel_filter:portion_gen_filter/3` содержит три варианта; их мы и должны проверить. Для этого скомпилируем соответствующие модули и запустим консоль среды выполнения языка `Erland`.

Проверим сначала первый вариант: вызов

```
parallel_filter:portion_gen_filter(fun(Item) -> length(Item) > 2 end,  
[1, 4])
```

возвращает пустой список, как и ожидается. Теперь проверим второй вариант: вызов

`parallel_filter:portion_gen_filter(fun(Item) -> length(Item) > 2 end,
[«а», «bbb», «cc»], 4)`

возвращает список строк **[[«bbb»]]**, длина которых больше 2. При этом размер исходного списка меньше размера порции. Это означает, что для выполнения будет выбран второй вариант функции `parallel_filter:portion_gen_filter/3`. И, наконец, проверим третий вариант вызов:

```
parallel_filter:portion_gen_filter(fun(Item) -> length(Item) > 2 end,  
[«a», «bbb», «cc», «dd», «eee»], 2)
```

возвращает список строк `["bbb", "eee"]`, длина которых больше 2. Так как размер списка с исходными данными 5, а размер порции 2, то для выполнения вызова будет создано 3 рабочих процесса.

А теперь давайте внимательно рассмотрим наше решение. Сле-

А теперь давайте внимательно рассмотрим наше решение. Оно имеет следующий вид: мы разбиваем исходящие данные на порции фиксированного размера, для обработки каждой порции создаем свой процесс, обрабатываем все порции данных много-задачным образом, собираем результаты обработки от всех процессов и объединяем все собранные данные в итоговый результат.

Легко увидеть главный недостаток нашего решения: мы разбиваем исходные данные на порции фиксированного размера и для обработки каждой порции создаем собственный процесс (модель «одноразовых процессов»). Очевидно, что при таком подходе количество созданных процессов будет пропорционально размеру

» Не хотите пропустить номер? Подпишитесь на www.linuxformat.ru/subscribe/!

Очистка очереди сообщений процесса

Возможна ситуация, когда один и тот же процесс используется для выполнения разных задач. В этом случае также возможно, что перед началом выполнения очередной задачи очередь сообщений процесса содержит сообщения, оставшиеся от выполнения одной из предыдущих задач. Эти сообщения могут повлиять на выполнение очередной задачи, если оно полагается на обмен

сообщениями с другими процессами. Конечно, это говорит о плохом проектировании тех задач, после которых остается «мусор» в очереди сообщений. Однако если мы хотим, чтобы подобные эффекты не влияли на выполнение задачи, необходимо очистить очередь сообщений процесса перед выполнением задачи. Сделать это можно, написав самим достаточно простой код, но библиотека язы-

ка Erlang уже содержит функцию для решения этой задачи – `lib:flush_receive/0` из модуля `lib`. И напоследок: использование сообщений для взаимодействия между процессами – это правильный способ, а для взаимодействия между задачами, последовательно выполняющимися на одном процессе – неправильный. Такое взаимодействие вводит неявные зависимости между задачами.

исходных данных. Хотя основная идеология языка Erlang подразумевает, что мы можем создавать ровно столько процессов, сколько нужно для решения задачи, все же максимальное количество процессов, которое может быть одновременно создано в одном экземпляре среды выполнения Erlang (на одном узле), ограничено. По умолчанию – не более 32768 процессов. Но если мы создаем экземпляр среды выполнения Erlang (узел) с флагом `+P`, то мы можем одновременно создать до 134217727 процессов (в зависимости от значения параметра, переданного с флагом `+P`). Это достаточно большое, но все же конечное число, и вполне возможна ситуация, когда мы не сможем создать очередной процесс в экземпляре среды выполнения Erlang (как рабочий процесс одной из наших реализаций, так и какой-либо сторонний процесс).

Решение этой проблемы достаточно очевидно: нужно отказаться от модели «одноразовых» процессов (т.е. от создания собственного процесса для обработки каждой порции данных) и использовать «многоразовые» процессы. Другими словами, мы должны ограничить число создаваемых рабочих процессов некоторым постоянным значением, не зависящим от размера исходных данных. Это означает, что мы должны создать предопределенный набор процессов до начала обработки исходных данных, раздать подготовленные для обработки порции данных этим процессам и собрать результаты обработки с этих процессов.

В отличие от ситуации «одноразовых» рабочих процессов, взаимодействие между предопределенными рабочими процессами и главным процессом должно быть более сложным. Действительно, мы должны дать задание рабочему процессу на обработку очередной порции, после чего ожидать от него результата работы, дать задание на обработку другой порции... и так, пока все порции данных не будут обработаны. Но можно несколько упростить это взаимодействие: после разбиения исходных данных на порции мы можем сразу раздать все задания всем рабочим процессам, после чего остается только собрать результаты их работы. Такой подход имеет свои минусы (о которых мы поговорим далее), но позволяет упростить переход на модель «многоразовых» рабочих процессов. На данном этапе для построения очередных многозадачных версий этих функций мы выбираем именно этот подход. Преимущество его в том, что для сбора результатов работы рабочих процессов мы можем использовать старый сборщик результатов (реализованный в функции `parallel_common:collect_result/2`). Действительно, не все ли равно, какие процессы и сколько раз («одноразовые» один раз или «многоразовые» несколько раз) будут нам присыпать результаты своей работы, если нам никак не надо отвечать на эти сообщения.

Приступим к реализации нашей очередной многозадачной версии функций `map` и `reduce` с учетом всего сказанного выше. Следует также напомнить о важной договоренности, введенной на прошлом уроке: мы договорились, что будем сначала писать набор общих методов, на основе которых строить конкретные

реализации для функций `map` и `reduce`. Пример такого подхода мы уже видели, реализуя методы `parallel_map:portion_gen_rmap/3` и `parallel_reduce:portion_gen_reduce/4` на основе функции `parallel_portion_helper:portion_core/4`, содержащей общую функциональность. В нашей реализации мы поступим точно так же: всю общую функциональность мы будем определять (и экспортить в случае необходимости) в модуле `parallel_limited_helper`.

Начнем с реализации функции `parallel_limited_helper:limited_worker/1`, которую будут выполнять рабочие процессы. Наши рабочие процессы являются «многоразовыми»; это означает, что мы в цикле (в рекурсивно-хвостовом вызове этой же функции) должны получать все сообщения, которые приходят в рабочий процесс и обрабатывать их. Сообщения вида `{task_request, MasterPid, Index, SourcePortion}`, где `MasterPid` – идентификатор главного процесса, `Index` – номер обрабатываемой порции, `SourcePortion` – порция исходных данных, иницируют обработку порции (и последующий возврат результатов обработки главному процессу); все остальные сообщения мы выбрасываем. Итак, с учетом всего сказанного, функция `parallel_limited_helper:limited_worker/1` имеет следующий вид:

```
limited_worker(Fun) ->
    receive
        {task_request, MasterPid, Index, SourcePortion} ->
            Dest = Fun(SourcePortion),
            MasterPid ! {result, Index, Dest},
            limited_worker(Fun);
        _Other -> limited_worker(Fun)
    end.
```

Здесь параметр `Fun` – это функция для обработки порции исходных данных; ее задают при создании рабочих потоков. Следует также сказать, что мы экспортствуем функцию `parallel_limited_helper:limited_worker/1` из модуля `parallel_limited_helper`. Это отличается от прошлой реализации функций `map` и `reduce`, когда мы не экспортствовали функцию, выполняемую рабочими процессами. Связано это с тем, что в модели «многоразовых» рабочих процессов ответственность за создание этих процессов мы выносим наружу (вскоре мы увидим, в чем плюс такого решения).

Следующая функция – та, что раздает задания нашим «многоразовым» рабочим процессам. Когда мы использовали модель «одноразовых» процессов, создание этих процессов и раздача им заданий реализовывалась крайне просто: проходом по списку порций исходных данных с помощью функции `lists:foreach/2`. Для «многоразовых» процессов все несколько сложнее: каждый такой процесс получает несколько заданий, и распределить задания между ними необходимо более-менее равномерно. Итак, наша задача заключается в равномерном распределении `N` заданий по `M` процессам. Одно из возможных решений этой задачи выглядит следующим образом: мы одновременно проходим как по списку заданий, так и по списку процессов, назначая текущее задание

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

текущему процессу. Если в какой-то момент времени мы дошли до конца списка процессов, а до конца списка заданий не дошли, то проходить список процессов мы начинаем сначала. Если же в какой-то момент времени мы дошли до конца списка задач, а до конца списка процессов не дошли, то мы заканчиваем работу, т.к. все задания уже распределены. Все это мы реализуем в паре функций `send_worker_tasks/2` и `send_worker_tasks/3` (мы не экспортим эти функции из модуля `parallel_limited_helper`):

```
send_worker_tasks(PreparedData, WorkerList) ->
    send_worker_tasks(PreparedData, WorkerList, 1).
send_worker_tasks([], _WorkerList, _WorkerIndex) -> complete;
send_worker_tasks(PreparedData, WorkerList, WorkerIndex)
when WorkerIndex > length(WorkerList) ->
    send_worker_tasks(PreparedData, WorkerList, 1);
send_worker_tasks([{Index, Portion} | Rest], WorkerList,
WorkerIndex) ->
    Worker = lists:nth(WorkerIndex, WorkerList),
    Worker ! {task_request, self(), Index, Portion},
    send_worker_tasks(Rest, WorkerList, WorkerIndex + 1).
```

Функция `send_worker_tasks/2` является интерфейсом; работа выполняется в функции `send_worker_tasks/3`. Обход по списку заданий осуществляется при помощи рекурсивно-хвостового вызова функцией `send_worker_tasks/3` самой себя. При этом по списку заданий (списку порций исходных данных) мы идем при помощи операции соответствия шаблону [pattern-matching], обрабатывая головной элемент и передавая хвостовую часть в рекурсивно-хвостовой вызов функции `send_worker_tasks/3`. С другой стороны, для прохода по списку процессов мы используем индекс текущего задания, увеличивая его на единицу при рекурсивно-хвостовом вызове функции `send_worker_tasks/3`. Реализовано это таким способом потому, что мы должны обходить список процессов циклически, если заданий больше, чем процессов.

Теперь наконец-то мы можем реализовать сердце нашего примера – функцию `parallel_limited_helper:limited_core/4`:

```
limited_core(FinalAggrFun, SourceList, PortionSize, WorkerList) ->
    process_flag(trap_exit, true),
    PortionCount = parallel_common:calc_portion_count(length(SourceList), PortionSize),
    PreparedData = parallel_common:prepare_data(PortionSize, SourceList),
    send_worker_tasks(PreparedData, WorkerList),
    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)).
```

В этой функции мы выполняем всю основную работу по разбиению исходных данных на порции, распределению заданий на обработку порций исходных данных равномерно по рабочим процессам, сбору результатов обработки порций исходных данных и объединению полученных результатов в итоговый результат. Отметим, что мы не создаем и не уничтожаем рабочие процессы в общей функции `parallel_limited_helper:limited_core/4`, а отдаляем эту ответственность вызывающему коду. В связи с этим, функции `parallel_limited_helper:limited_core/4`, помимо списка исходных данных `SourceList` и размера порции `PortionSize`, мы передаем список созданных рабочих процессов `WorkerList` и функцию `FinalAggrFun` для объединения результатов работы в итоговый результат. При этом функцию для обработки порций исходных данных мы задаем при создании рабочих процессов при помощи функции `parallel_limited_helper:limited_worker/1`.

На данный момент мы написали всю необходимую функциональность и готовы реализовать многозадачные версии функций `map` и `reduce` с использованием модели «многоразовых»

рабочих процессов (да, мы пока больше не будем делать многозадачную версию функции `filter`). Начнем с версии функции `map`, которая для данного примера реализации называется `parallel_map:limited_pmap/4`:

```
limited_pmap(_Fun, [], _PortionSize, _WorkerCount) -> [];
limited_pmap(Fun, SourceList, PortionSize, _WorkerCount)
when length(SourceList) =< PortionSize ->
    lists:map(Fun, SourceList);
limited_pmap(Fun, SourceList, PortionSize, WorkerCount) ->
    WorkerFun = fun(SourcePortion) -> lists:map(Fun,
SourcePortion) end,
    WorkerList =
    [spawn_link(fun() -> parallel_limited_helper:limited_
worker(WorkerFun) end) || _WorkerIndex <- lists:seq(1,
WorkerCount)],
    Result = parallel_limited_helper:limited_core(fun lists:append/1,
SourceList, PortionSize, WorkerList),
    lists:foldl(fun(Worker, _Aggr) -> exit(Worker, normal) end, true,
WorkerList),
    Result.
```

Функция `parallel_map:limited_pmap/4` содержит три варианта: первый вариант обрабатывает ситуацию пустого списка исходных данных, второй вариант – когда размер исходных данных меньше размера порции, а третий вариант обрабатывает общий случай. Как говорилось выше, ответственность за создание и завершение рабочих процессов мы возлагаем на код, который использует функцию `parallel_limited_helper:limited_core/4`. Поэтому перед использованием этой функции мы создаем `WorkerCount` «многоразовых» рабочих процессов, а после получения результата мы завершаем работу этих рабочих процессов.

Перейдем к версии функции `reduce`, которая для данного примера реализации называется `parallel_reduce:limited_reduce/5`:

```
limited_reduce(_Fun, [], {InitValue, _PortionInitValue}, _PortionSize, _WorkerCount) -> InitValue;
limited_reduce(Fun, SourceList, {InitValue, _PortionInitValue}, PortionSize, _WorkerCount) when length(SourceList) =< PortionSize ->
    lists:foldl(Fun, InitValue, SourceList);
limited_reduce(Fun, SourceList, {InitValue, PortionInitValue}, PortionSize, WorkerCount) ->
    ReduceFun = fun(List) -> lists:foldl(Fun, InitValue, List) end,
    PortionReduceFun = fun(List) -> lists:foldl(Fun, PortionInitValue, List) end,
    WorkerList =
    [spawn_link(fun() -> parallel_limited_helper:limited_
worker(PortionReduceFun) end) || _WorkerIndex <- lists:seq(1,
WorkerCount)],
    Result = parallel_limited_helper:limited_core(ReduceFun,
SourceList, PortionSize, WorkerList),
    lists:foldl(fun(Worker, _Aggr) -> exit(Worker, normal) end, true,
WorkerList),
    Result.
```

Как видно, функция `parallel_reduce:limited_reduce/5` реализована с использованием тех же принципов, что и функция `parallel_map:limited_pmap/4`. Они отличаются только функциями, которые обрабатывают порции исходных данных и объединяют результаты обработки порции в итоговый результат.

Сегодня мы сделали очередной шаг в нашем практикуме: мы реализовали версии функций `map` и `reduce`, которые используют ограниченное количество рабочих процессов (используют модель «многоразовых» процессов). К сожалению, из-за того, что место под статью конечно, мы не успели протестировать созданные нами функции; эту задачу мы оставляем читателям. А в следующий раз мы продолжим наш практикум: на очереди распределенная версия функций `map` и `reduce` и многое другое. 