

Erlang: Практика

Андрей Ушаков не поступился принципом единоличной ответственности.
И вот куда это его завело...



Наш
эксперт

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

И так, мы продолжаем решать нашу большую задачу: создание многозадачных версий функций `map` и `reduce`. На этом уроке мы поговорим о том, как на основе функций, созданных в прошлый раз, реализовать распределенные (выполняющиеся на различных узлах) версии функций `map` и `reduce`. Также мы проанализируем допущения, принятые нами, когда мы реализовывали версии функций `map` и `reduce` на основе «многоразовых» процессов. Это позволит нам понять, куда идти дальше. Но сперва взглянем, где мы остановились в прошлый раз.

На прошлом уроке мы перешли к модели «многоразовых» рабочих процессов и ограничили их число. На основе этого подхода мы создали очередные многозадачные версии функций `map` и `reduce`: функции `parallel_map:limited_pmap/4` и `parallel_reduce:limited_reduce/5`. И, как обычно, при создании этих функций мы вынесли общую функциональность в ряд функций, располагающихся в модуле `parallel_limited_helper`. При реализации этой функциональности мы приняли два важных соглашения: во-первых, договорились, что ответственность за создание и уничтожение процессов ложится на вызывающую сторону. Во-вторых, все задания мы сразу распределяем между рабочими процессами, после чего только дожидаемся результатов их работы.

Еще раз рассмотрим эту общую функциональность. `parallel_limited_helper:limited_worker/1` является функцией, которую выполняет рабочий процесс во время своей жизни:

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

В этой функции рабочие процессы выполняют задания на обработку порций данных, которые они получают в виде сообщений, посланных рабочим процессам главным процессом. Естественно,

что результаты работы отсылаются обратно главному процессу. Мы экспортствуем эту функцию из модуля `parallel_limited_helper`, т.к. мы договорились, что рабочие процессы создает внешний код. Пара функций `send_worker_tasks/2` и `send_worker_tasks/3` используется для распределения заданий между рабочими процессами (функция `send_worker_tasks/2` является интерфейсом, а функция `send_worker_tasks/3` – реализацией данной функциональности):

```
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).
```

Эта пара функций всего лишь определена в модуле `parallel_limited_helper`, но не экспортируется из него, т.к. инкапсулирует один из внутренних шагов. И, наконец, функция `parallel_limited_helper:limited_core/4` является сердцем всех реализаций, основанных на этой функции:

```
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 разбиваем исходные данные на порции, равномерно распределяем задания по обработке порций данных между созданными рабочими потоками, собираем результаты обработки порций данных рабочими потоками и объединяем результаты их работы в итоговый результат.

Рассмотрим соглашения, принятые нами при реализации функций из модуля `parallel_limited_helper`. Начнем с соглашения о том, что ответственность за создание и уничтожение рабочих процессов лежит на вызывающей стороне. С первого взгляда может показаться, что это решение – не из оптимальных: почему нельзя просто передать число рабочих процессов в функцию `parallel_limited_helper:limited_core/4` (вместо списка рабочих процессов)? Для понимания причин давайте поставим более общую задачу: нам необходимо создать распределенные версии функций `map` и `reduce`, т.е. версии функций, рабочие процессы которых выполнялись бы на заранее заданных узлах. При этом мы ограничиваем максимальное число рабочих процессов на каждом

Я отвечаю за все? А вот и нет!

Термин Single Responsibility Principle (SRP) переводится как «принцип одной ответственности». Это принцип объектно-ориентированного программирования, и он означает, что каждый создаваемый класс должен отвечать за что-то одно и эта ответственность должна быть данным классом полностью инкапсулирована. Заменив термин «класс» на термин «сущность», то есть слегка обобщив (сущностью может быть и функция), этот принцип можно использовать и в функциональном программировании. Пусть для обработки некоторых данных необходимо сделать

несколько подготовительных шагов. Если код, выполняющий эти шаги, и код по обработке данных будут находиться в одной функции, то это, очевидно, нарушение SRP-принципа. Наиболее логичным подходом (приводящим к понятному и легкому в поддержке результату) будет разнести все подготовительные шаги и обработку данных по отдельным функциям. После такого разделения также необходимо будет создать функцию, содержащую вызовы – как функций, инкапсулирующих подготовительные шаги, так и функции, которая выполняет обработку данных.

Многозадачности

Что такое узлы и как их именуют

Узлом называется именованный экземпляр среды выполнения Erlang. Чтобы создать узел, достаточно при запуске среды выполнения Erlang указать ключ `-sname` или `-name` и имя создаваемого узла. При указании ключа `-sname` создается узел с коротким именем, при указании ключа `-name` – узел с длинным именем. Если полное имя компьютера будет `CompName.DomainName`, то при создании узла с коротким именем `Name` его полное имя

будет `Name@CompName`, а при создании узла с длинным именем `Name` его полное имя будет `Name@CompName.DomainName`. Если при создании узла задать имя в виде `Part1@Part2`, то это имя будет именем узла вне зависимости от типа создаваемого узла и имени компьютера. Функция `node/0` (BIF) позволяет получить имя текущего узла в виде атома. Если среда выполнения Erlang создана не именованной, то функция `node/0` в таком слу-

чае вернет атом `nonode@nohost` (этот атом можно использовать как обычное имя узла).

Желая использовать экземпляр среды выполнения Erlang для создания распределенной среды выполнения, мы должны его создать с длинным или коротким именем, т.е. сделать узлом. Отметим, что узлы с разным типом имен (т.е. узлы с длинными и короткими именами) не могут взаимодействовать друг с другом.

узле, что дает в результате ограничение на общее число рабочих процессов. Распределяя рабочие процессы по всем узлам равномерно, мы все же можем их создавать в нашей обобщенной функции (которая аналогична функции `parallel_limited_helper:limited_core/4`), передавая как параметры максимальное число рабочих процессов на каждом узле и список доступных узлов. А если мы хотим создавать на разных узлах разное число рабочих процессов, мы уже должны передавать список пар (кортежей из двух элементов) «узел – максимальное число процессов на узле».

В чем минусы такого подхода? Во-первых, тогда уменьшается количество сценариев использования данной функции. Действительно, при передаче в функцию списка рабочих процессов (которые создала нам вызывающая сторона) нам без разницы, созданы ли эти процессы на одном узле с главным процессом или же нет. Нам также без разницы, ограничено ли время жизни рабочих процессов многозадачной (или распределенной) версией функции `map` и `reduce` или же они являются долгоживущими (например, из некоторого пула процессов). С другой стороны, решив, что процессы должна создавать сама наша функция, мы получим ситуацию, когда, скажем, наша функция может создать рабочие процессы только на локальном узле (или с еще какими-либо ограничениями). Во-вторых, при таком подходе нарушается принцип SRP: функция содержит и реализацию многозадачной обработки списка, и функциональность по созданию рабочих процессов.

Конечно, когда мы использовали «одноразовые» рабочие процессы, их создание в нашей обобщенной функции было оправдано, т.к. было неотъемлемой частью алгоритма, и только наша функция знала об этих процессах. В случае же «многоразовых» рабочих процессов их создание неотъемлемой частью алгоритма не является. К тому же об этих процессах знает внешний код (т.к. он задает ограничения на их количество), и вполне логично, что именно он будет управлять временем жизни этих процессов. Если же нам необходимо, чтобы у нас на разных узлах было разное количество рабочих процессов, то при обсуждаемом подходе мы получим ситуацию, когда код для создания этих рабочих процессов находится как на вызывающей стороне, так и на вызываемой стороне. Действительно, на вызывающей стороне мы будем вычислять для каждого узла максимальное число рабочих процессов, и формировать список пар «узел – максимальное число рабочих процессов на этом узле». А на вызываемой сто-

роне – создавать рабочие процессы в соответствии с переданным списком. Очевидно, что поддерживать и расширять подобную реализацию будет тяжело.

А теперь давайте решим поставленную выше задачу: создадим распределенные версии функций `map` и `reduce`. Как уже говорилось, функция `parallel_limited_helper:limited_core/4` передает ответственность за управление жизнью рабочих процессов вызывающей стороне. От вызывающей стороны функция `parallel_limited_helper:limited_core/4` ожидает список рабочих процессов, которые выполняют функцию `parallel_limited_helper:limited_worker/1` или ей подобную, т.е. с таким же протоколом взаимодействия. Это означает, что для создания распределенных вариантов функций `map` и `reduce` мы можем использовать функцию `parallel_limited_helper:limited_core/4`. Функции `parallel_map:distributed_pmap/5` и `parallel_reduce:distributed_reduce/6` будут реализациями распределенных версий функций `map` и `reduce`. В этих функциях мы создаем рабочие процессы на заданных узлах, используем созданные процессы для распределенной обработки исходного списка (при помощи вызова функции `parallel_limited_helper:limited_core/4`) и завершаем работу созданных рабочих процессов. Функция `parallel_map:distributed_pmap/5` имеет следующий вид:

```
distributed_pmap(_Fun, [], _PortionSize, _NodeList, _WorkerCount) -> [];
distributed_pmap(Fun, SourceList, PortionSize, NodeList,
    _WorkerCount)
when length(SourceList) =< PortionSize ->
    lists:map(Fun, SourceList);
distributed_pmap(Fun, SourceList, PortionSize, NodeList,
    WorkerCount) ->
    WorkerFun = fun(SourcePortion) -> lists:map(Fun,
        SourcePortion) end,
    WorkerList = [spawn_link(Node, fun() -> parallel_limited_
        helper:limited_worker(WorkerFun) end) || Node <- NodeList,
        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.
```

>>

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

Записи в языке Erlang

Записи в языке Erlang не являются отдельным типом данных: это всего лишь «синтаксический сахар» для доступа к полям кортежа по именам (заданным при определении записи), а не по индексам. Записи определяются при помощи директивы “-record”. Так, например, директива -record(somerecord, {f1=[]},

{f2=0}) определяет запись с именем somerecord и двумя полями: f1 со значением по умолчанию [] (пустой список) и f2 со значением по умолчанию 0. Создать экземпляр записи можно следующим способом: RecordInstance = #somerecord{f2=666}. При этом в реальности создается следующий кортеж:

{somerecord, [], 666}. Получить значение поля записи по имени можно так: RecordInstance#somerecord.f2.

Определение записи, которое должно быть доступно нескольким модулям, выносят в отдельный файл (с расширением .hrl) и подключают директивой “-include” там, где это нужно.

Как и в предыдущих случаях, функция parallel_map:distributed_pmap/5 содержит три варианта. Первый вариант предназначен для обработки ситуации, когда исходный список пуст, второй вариант – для обработки ситуации, когда размер исходных данных не превышает размера порции, а третий – для обработки всех остальных случаев. Функция parallel_reduce:distributed_reduce/6 выглядит следующим образом:

```
distributed_reduce(_Fun, [], {Init, _PortionInit}, _PortionSize, _NodeList, _WorkerCount) -> InitValue;
distributed_reduce(Fun, SourceList, {Init, _PortionInit}, PortionSize, _NodeList, _WorkerCount) when length(SourceList) =< PortionSize ->
    lists:foldl(Fun, Init, SourceList);
distributed_reduce(Fun, SourceList, {Init, PortionInit}, PortionSize, NodeList, WorkerCount) ->
    ReduceFun = fun(List) -> lists:foldl(Fun, Init, List) end,
    PortionReduceFun = fun(List) -> lists:foldl(Fun, PortionInit, List) end,
    WorkerList = [spawn_link(Node, fun() -> parallel_limited_helper:limited_worker(PortionReduceFun) end) || Node <- NodeList, _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.
```

Функция parallel_reduce:distributed_reduce/6, как и функция parallel_map:distributed_pmap/5, содержит три варианта для обработки точно таких же ситуаций: пустого исходного списка, исходного списка малого размера (не больше размера порции) и для всех остальных случаев. В этих функциях мы задаем список узлов NodeList, на которых могут создаваться рабочие процессы, и максимальное количество рабочих процессов на каждом узле WorkerCount; тем самым мы ограничиваем общее количество рабочих процессов.

Внимательный читатель легко заметит, что в качестве параметров мы передаем в функции parallel_map:distributed_pmap/5 и parallel_reduce:distributed_reduce/6 список узлов NodeList и максимальное количество процессов на каждом узле WorkerCount. Может возникнуть вопрос, не противоречит ли такое решение всему вышеизказанному. Если мы используем долгоживающие рабочие процессы (например, из некоторого пула процессов), такое решение будет просто неправильным. Во всех остальных случаях мы хотим просто выполнить распределенную операцию map (или reduce) на определенном наборе узлов, ограничив максимальное количество рабочих процессов на этих узлах. Если и тогда мы будем возлагать ответственность по созданию рабочих процессов на внешнюю сторону, то это не совсем то, что ожидает от нас вызывающая сторона (да и неудобно для вызывающей стороны). Конечно, можно было бы выделить операции по созданию рабочих процессов и завершению их работы в отдельные методы,

что слегка повысило бы читаемость. Но это действие мы оставим для тех читателей, кому оно интересно.

Пора проверить, что наши распределенные реализации функций map и reduce (функции parallel_map:distributed_pmap/5 и parallel_reduce:distributed_reduce/6) работают правильно. Для начала давайте создадим три узла: два узла с именами slave1 и slave2 для создания рабочих процессов и узел с именем master для главного процесса (и запуска функций parallel_map:distributed_pmap/5 и parallel_reduce:distributed_reduce/6 на выполнение). На компьютере автора (при создании узлов с ключом -sname) полные имена узлов будут следующими: slave1@stdstring, slave2@stdstring и master@stdstring (именно эти имена узлов мы будем использовать в нашем примере). Все действия в примере мы будем производить на узле master@stdstring.

Начнем с функции parallel_map:distributed_pmap/5: мы помним, что эта функция имеет три варианта. Вызов parallel_map:distributed_pmap(fun(Item) -> 3*Item end, [], 2, ['slave1@stdstring', 'slave2@stdstring'], 2) проверяет первый вариант (когда исходный список данных пуст) и возвращает пустой список, как и ожидается. Вызов parallel_map:distributed_pmap(fun(Item) -> 3*Item end, [2, 3], 4, ['slave1@stdstring', 'slave2@stdstring'], 2) возвращает следующий список [6, 9]. Так как размер исходного списка меньше размера порции, то мы проверяем второй случай. И, наконец, вызов parallel_map:distributed_pmap(fun(Item) -> 3*Item end, [2, 3, 5, 6, 8, 1, 7, 2], 2, ['slave1@stdstring', 'slave2@stdstring'], 2) возвращает следующий список: [6, 9, 15, 18, 24, 3, 21, 6]. Очевидно, что этот вызов проверяет третий вариант функции parallel_map:distributed_pmap/5, т. к. размер исходного списка больше размера порции. У нас выделено два узла под рабочие процессы, на каждом узле мы создаем по два процесса; в итоге у нас четыре рабочих процесса. Размер исходного списка – 8, размер порции данных для обработки – 2; легко видеть, что список будет разбит на четыре порции, и в результате все четыре рабочих процесса будут загружены.

А теперь проверим работу функции parallel_reduce:distributed_reduce/6 (эта функция также имеет три варианта). Вызов parallel_reduce:distributed_reduce(fun(Item, Agg) -> Item + Agg end, [], {1, 0, 2, 1}, ['slave1@stdstring', 'slave2@stdstring'], 2) проверяет первый вариант (когда исходный список данных пуст) и возвращает 1, т. е. начальное значение операции свертки. Вызов parallel_reduce:distributed_reduce(fun(Item, Agg) -> Item + Agg end, [1, 2], {1, 0, 4, 1}, ['slave1@stdstring', 'slave2@stdstring'], 2) возвращает значение 4. Размер исходного списка – 2, размер порции – 4; это означает, что данный вызов проверяет второй вариант функции parallel_map:distributed_pmap/5. И, наконец, вызов parallel_reduce:distributed_reduce(fun(Item, Agg) -> Item + Agg end, [1, 2, 3, 4, 5, 6, 7, 8], {1, 0, 2, 1}, ['slave1@stdstring', 'slave2@stdstring'], 2) возвращает значение 37. Этот вызов проверяет третий вариант функции parallel_map:distributed_pmap/5, т. к. размер исходного списка больше размера порции. Как и в предыдущем случае, у нас четыре рабочих процесса на двух узлах (по два рабочих процесса

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

на узел). Размер исходного списка – 8, размер порции – 2, так что список будет разбит на четыре порции и все четыре рабочих процесса будут загружены. Проверку других сценариев работы этих функций (например, когда не для всех рабочих процессов будут созданы задания) мы оставляем читателям.

Займемся соглашением раздавать все задачи рабочим процессам сразу, при вызове функции **parallel_limited_helper:limited_core/4**. При вызове этой функции, мы разбиваем исходный список с данными на порции; порции являются списком пар (кортежей из двух элементов) «индекс – часть исходного списка с данными». Очевидно, что размер всех порций данных несколько больше размера исходных данных; действительно, размер всех порций равен размеру исходного списка, плюс размер на индексы всех порций, плюс накладные расходы на создание пары (кортежа из двух элементов) для каждой порции. И если исходный список очень велик, нам просто может не хватить памяти для создания всех порций. Например, на 32-разрядной системе размер адресного пространства процесса (в операционной системе Linux с обычным ядром) – 3 ГБ; если размер исходного списка больше 1,5 ГБ, то создать все порции данных для этого списка, очевидно, не получится. Даже если места для создания всех порций хватит, все равно их создание приводит к одномоментным накладным расходам на выделение памяти, создания объектов, копированиями данных и к последующим накладным расходам на сборку мусора, когда все эти объекты станут не нужны. Создав все порции, мы сразу же распределяем их между рабочими процессами, т.е. отправляем рабочим процессам сообщения, содержащие данные для обработки порции для всех порций данных. При этом все порции данных в один момент времени оказываются в сети, вне зависимости от их объема. Понятно, что при большом объеме полученных порций (то есть большом объеме исходных данных) взаимодействие узлов по сети может стать узким местом, тормозящим всю систему (всю нашу многозадачную обработку).

На эту проблему стоит взглянуть несколько с другой стороны: так ли нам надо сразу разбивать все исходные данные на порции и отправлять их по сети рабочим процессам? Очевидно, что такой подход упрощает исходный код наших реализаций; более того, такой подход позволяет во всех наших реализациях (созданных до этого момента) использовать одну и ту же функцию для сборки результатов обработки: **parallel_common:collect_result/2**. Мы решили, что отказываемся от подхода, когда все делается сразу, и предпочли подход, когда порции формируются и отсылаются рабочему процессу по мере необходимости. Это решение подводит нас к тому, что мы не можем отделить стадию формирования и отсылки заданий рабочим процессам от стадии сбора результата их работы (как это было у нас раньше). Поэтому наше взаимодействие с рабочими процессами будет выглядеть следующим образом.

Мы получаем результат обработки какой-либо порции данных от какого-то рабочего процесса, сохраняем этот результат (в массиве, как мы это делали и раньше), после чего «отсыпываем» от оставшихся необработанных данных порцию, формируем новое задание на обработку и отсылаем это задание рабочему процессу, с которым начали взаимодействие. И так до тех пор, пока мы не обработаем весь список исходных данных (пока список оставшихся необработанных данных не опустеет) и не получим все результаты обработки (понятно, что список оставшихся необработанных данных опустеет раньше, чем мы действительно обработаем все данные). Ну и, естественно, работу мы должны начать с того, чтобы каждому рабочему процессу раздать по заданию. Понятно, что такой подход несколько усложняет реализацию по сравнению с реализацией, в которой мы сразу разбиваем все данные на порции и отсылаем эти данные рабочим процессам.

А теперь давайте реализуем новый вариант многозадачных функций **map** и **reduce** на основе всего вышеизложенного. Как

и раньше, мы выделяем общую функциональность (на основе которой мы сможем реализовать многозадачные версии функций **map** и **reduce**) и помещаем ее в отдельный модуль; в нашем случае это будет модуль **parallel_smartmsg_helper**.

Чтобы получить более понятную и гибкую реализацию, введем несколько определений записей:

```
-record(tasks_descr, {created = 0, processed = 0, rest = []}).  
-record(task_request, {master, index, portion}).  
-record(task_result, {worker, index, result}).
```

Экземпляр записи **task_descr** хранит данные о процессе обработки исходного списка; поле **created** содержит количество созданных заданий на обработку; поле **processed** содержит количество заданий на обработку, выполнение которых закончилось; поле **rest** содержит необработанный остаток исходного списка. Экземпляр записи **task_request** содержит данные запроса на обработку (посыпаем главным процессом одному из рабочих процессов); поле **master** содержит идентификатор главного процесса; поле **index** содержит индекс порции исходных данных; поле **portion** содержит саму порцию исходных данных. Следует заметить, что вместо передачи идентификатора главного процесса в сообщении его можно было бы передать рабочему процессу одним из параметров функции, которую этот рабочий процесс выполняет. И, наконец, экземпляр записи **task_result** содержит данные с результатами обработки порции (посыпаемые одним из рабочих процессов главному); поле **worker** содержит идентификатор рабочего процесса; поле **index** содержит индекс исходной порции данных; поле **result** содержит результат обработки этой исходной порции данных.

Следующий шаг, который мы сделаем в рамках нашей реализации – создадим функцию, которую должны выполнять рабочие процессы. Так как мы создаем рабочие процессы снаружи нашей обобщенной функции обработки списка данных (аналог функции **parallel_limited_helper:limited_core/4**), на основе которой мы потом создадим очередные версии функций **map** и **reduce**, очевидно, что эта функция должна быть экспортруемой. В нашей реализации это будет функция **parallel_smartmsg_helper:smartmsg_worker/1**:

```
smartmsg_worker(Fun) ->  
    receive  
        #task_request{master=MasterPid, index=Index,  
        portion=SourcePortion} ->  
            Dest = Fun(SourcePortion),  
            MasterPid ! #task_result{worker=self(), index=Index,  
            result=Dest},  
            smartmsg_worker(Fun);  
        _Other -> smartmsg_worker(Fun)  
    end.
```

В функции, которую выполняют рабочие процессы, мы обрабатываем два типа сообщений: во-первых, сообщения, являющиеся экземпляром записи **task_request** – это задания на обработку порции данных; во-вторых, все остальные сообщения, чтобы в очереди сообщений рабочего процесса не накапливались необработанные неизвестные сообщения. Следует сказать, что функция **parallel_smartmsg_helper:smartmsg_worker/1** практически идентична функции **parallel_limited_helper:limited_worker/1**, которую создали в предыдущей реализации. Мы не стали использовать в нашей новой реализации функцию **parallel_limited_helper:limited_worker/1**, чтобы не смешивать использование нескольких модулей и не смущать читателя этим.

На этом практикуме мы создали распределенные версии функций **map** и **reduce** и протестировали их. Мы также разобрались, что создание всех порций исходных данных и распределение их среди рабочих потоков в один момент времени – не лучшая идея, и начали реализовывать более сложный подход, в котором мы формируем задачи и отсылаем их рабочим процессам по мере необходимости. В следующий раз мы продолжим эту работу. [lxr](#)