

Erlang: Почти

Пятый практикум по многозадачности **Андрей Ушаков** посвящает сбережению системных ресурсов.



Наш
эксперт

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

В очередной раз займемся практикумом по многозадачности в языке Erlang – в виде работы над многозадачными версиями функций `map` и `reduce`. Сегодня мы продолжим создавать версии функций `map` и `reduce`, которые бережно относятся к ресурсам компьютера: процессорному времени, памяти, сетевой подсистеме. Это означает, что новые версии функций `map` и `reduce` не создают лишних объектов, в роли которых выступали порции данных, и не нагружают сетевую подсистему одновременно посылкой всех этих объектов.

По традиции вспомним, на чем мы остановились в прошлом номере. Мы говорили о решении (принятом при построении наших примеров) разбить все исходные данные на порции и разослать эти порции данных (как задания по обработке) рабочим процессам сразу, а затем только собирать результаты обработки этих порций рабочими процессами. Мы показали, что это приводит к резкому скачку потребляемой памяти и нагрузки на процессор и, возможно, на сеть (если рабочие процессы выполняются на разных узлах). Ведь если мы хотим разбить все исходные данные на порции, то необходимо под эти порции данных выделить память, размер которой чуть больше размера исходных данных (т.к. каждая порция – это кортеж, состоящий из самой порции данных и ее индекса), а также потратить некоторое процессорное время. Далее мы все эти данные передаем по сети нашим рабочим процессам, увеличивая нагрузку на сеть. А при достаточно большом объеме входных данных нам просто может не хватить размера адресного пространства (процесса операционной системы, содержащего среду выполнения Erlang, наш пример и данные) для того, чтобы хранить одновременно как исходные данные, так и порции данных.

Разберемся, как решить эту проблему. В прошлый раз мы рассмотрели возможное ее решение: «отрезать» новую порцию для обработки какому-либо рабочему процессу только по мере необходимости. Т.е. только после того, как какой-либо рабочий процесс выполнит свою задачу (обработает порцию данных) и отшлет результат обработки главному процессу, он получит следующее задание (следующую порцию) для выполнения. Правда, за это решение нам придется платить, т.к. усложняется взаимодействие между рабочими процессами и главным процессом. Если раньше главный процесс разбивал исходные данные на порции, распределяя эти порции (как задания) между рабочими процессами и ожидал результатов обработки от всех рабочих процессов, то при новом подходе необходимо в главном процессе делать следующее: создать и раздать начальные задания для рабочих процессов, потом собрать полученные результаты обработки и при наличии еще не обработанных данных создавать новую порцию и отсыпать ее на обработку. С другой стороны, для рабочих процессов ничего не изменилось: мы все так же получаем задание на обработку порции данных, обрабатываем эту порцию, отсылаем результаты обработки обратно главному процессу, после чего получаем очередное задание, если оно есть. И абсолютно не важно, сразу ли все задания на обработку отправляются рабочим процессам или же постепенно, в ответ на результат обработки предыдущего задания.

А теперь давайте посмотрим на то, что из реализации данного подхода в прошлый раз мы успели сделать. Во-первых, мы объ-

явили, что всю общую функциональность (на основе которой мы сможем реализовать многозадачные версии функций `map` и `reduce`) мы помещаем в модуль `parallel_smartmsg_helper`. Вторых, мы ввели ряд определений записей для того, чтобы сделять наш код более понятным:

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

Здесь запись `task_descr` определена для хранения данных о процессе обработки исходного списка, запись `task_request` представляет запрос на обработку очередной порции данных (сообщение, которое получает рабочий процесс от главного процесса), а запись `task_result` представляет результат обработки очередной порции данных рабочим процессом (сообщение, которое рабочий процесс посылает главному процессу). Следует заметить, что мы не помещаем эти определения записей в отдельный заголовочный файл (с расширением `.hrl`), т.к. эти определения используются только внутри модуля `parallel_smartmsg_helper`. И, наконец, мы определили функцию, которую выполняет рабочий процесс во время своей жизни. Это экспортруемая 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.
```

Как мы уже говорили, для рабочего процесса логика работы абсолютно не зависит от того, сразу ли все задания на обработку порций данных он получит или же постепенно, в ответ на результат обработки некоторой порции данных. Поэтому эта функция ничем не отличается от аналогичных функций, которые выполняют рабочие процессы более ранних примеров, за исключением того, что здесь для работы с сообщениями мы используем определенные ранее записи. Эта функция является экспортруемой – по причине того, что за создание рабочих процессов у нас ответственен внешний код.

После повторения мы можем смело идти дальше. Для начала давайте создадим пару вспомогательных функций, которые будут инкапсулировать такие операции, как создание и назначение новой задачи рабочему процессу и сохранение результата обработки порции в промежуточном хранилище. Начнем со вспомогательной функции, которая сохраняет результат обработки порции в промежуточном хранилище. Как уже говорилось в предыдущих статьях, в качестве промежуточного хранилища мы используем массив (коллекцию, доступ к элементам которой осуществляется по индексу), а результат обработки порции состоит из самого результата и его индекса (который совпадает с индексом порции при ее создании). Поэтому функция для сохранения результатов

с очередями

обработки порции должна в качестве входных параметров принимать массив с результатами обработки порций, сам результат обработки порции и его индекс, а возвращать обновленный массив с результатами обработки порций (мы помним, что в языке Erlang мы не изменяем существующий объект, а создаем новый). Это функция `collect_result/3`, которая определена в модуле `parallel_smartmsg_helper`, но не экспортируется из него:

```
collect_result(Result, Index, Storage) ->
    array:set(Index, Result, Storage).
```

По сути, она очень проста – это всего лишь альтернатива (алиас) для функции `array:set/3`; основное ее предназначение – сделать код, который сохраняет результаты обработки порции в промежуточное хранилище, более ясным и понятным.

В качестве следующего шага мы создадим вспомогательную функцию для назначения нового задания рабочему процессу. Это будет функция `assign_task/4`, определенная в модуле `parallel_smartmsg_helper`, но не экспортируемая из него:

```
assign_task(Worker, SourceList, PortionSize, Index)
when length(SourceList) =< PortionSize ->
    Worker ! #task_request{master = self(), index = Index, portion =
SourceList},
    [];
assign_task(Worker, SourceList, PortionSize, Index) ->
    {Portion, Rest} = lists:split(PortionSize, SourceList),
    Worker ! #task_request{master = self(), index = Index, portion =
Portion},
    Rest.
```

Видно, что эта функция выглядит сложнее функции `collect/3`. В ней нам необходимо обрабатывать два случая (для этого определено два варианта функции `assign_task/4`): когда размер необработанного остатка меньше или равен размеру порции данных и когда размер необработанного остатка больше размера порции. В первом случае мы просто отсылаем этот остаток рабочему процессу на обработку и возвращаем пустой список; это означает, что больше нет исходных данных для обработки. Во втором случае мы «отрезаем» порцию заданного размера, отсылаем эту порцию рабочему процессу, после чего возвращаем остаток после «отрезания» порции обратно, т.к. он содержит не обработанные еще данные. В качестве параметров эта функция принимает идентификатор рабочего процесса, которому мы собираемся назначить задание, остаток необработанных исходных данных, размер порции и индекс создаваемой порции данных.

А мы можем пойти дальше. Если раньше мы «разбивали» исходные данные на порции и распределяли эти порции (в виде заданий на обработку) среди рабочих процессов сразу, то сейчас мы «отрезаем» очередную порцию и отдаём её рабочему процессу только после того, как он пришлет главному процессу результаты обработки какой-либо порции данных. Но для того, чтобы такая схема взаимодействия главного процесса и рабочих процессов работала, необходимо инициализировать рабочие процессы. Процесс инициализации рабочих процессов состоит в следующем: для каждого рабочего процесса мы «отрезаем» от исходных данных (а точнее, от необработанного остатка) порцию и отправляем эти порции на обработку рабочим процессам. После такой инициализации мы можем использовать описанный выше механизм взаимодействия, т.е. ожидать результат обработки порции от какого-либо рабочего процесса, после чего подго-

тавливать и отсылать ему очередное задание (очередную порцию исходных данных). Данный процесс инициализации рабочих процессов мы реализуем при помощи функции `distribute_init_tasks/3`, которая определена в модуле `parallel_smartmsg_helper`:

```
distribute_init_tasks(#tasks_descr{created=Created, rest=[]}, _PortionSize, _WorkerList) ->
    #tasks_descr{created = Created, rest = []};
distribute_init_tasks(TasksDescr, _PortionSize, []) -> TasksDescr;
distribute_init_tasks(#tasks_descr{created=Created, rest=Source}, PortionSize, [Worker|Workers]) ->
    Rest = assign_task(Worker, Source, PortionSize, Created),
    TasksDescr = #tasks_descr{created=Created+1, rest=Rest},
    distribute_init_tasks(TasksDescr, PortionSize, Workers).
```

Эта функция на вход принимает три параметра: описатель процесса обработки исходных данных (который является экземпляром записи `tasks_descr`), размер порции исходных данных и список идентификаторов рабочих процессов (точнее, остаток списка идентификаторов рабочих процессов); возвращает эта функция описатель процесса обработки исходных данных после инициализации рабочих процессов. В функции `distribute_init_tasks/3` мы должны обрабатывать три разных случая (поэтому функция `distribute_init_tasks/3` содержит три варианта): когда закончились входные данные, когда мы инициализировали все рабочие процессы, и общий случай – когда есть как необработанные входные данные, так и неинициализированные рабочие процессы. Первый случай возможен, если количество рабочих процессов больше или равно количеству порций (с размером `PortionSize`). В первом и втором варианте функции `distribute_init_tasks/3` дальнейшая инициализация уже невозможна, поэтому мы возвращаем описатель процесса обработки данных. В последнем же варианте (когда есть как входные данные, так и неинициализированные рабочие процессы) мы инициализируем очередной рабочий процесс порцией (при помощи функции `assign_task/4`), после чего вызываем рекурсивно (при помощи хвостовой рекурсии) метод `distribute_init_tasks/3` для оставшихся рабочих процессов и входных данных. Также следует сказать, что функция `distribute_init_tasks/3` не экспортируется из модуля `parallel_smartmsg_helper`, т.к. она используется только внутри него.

Следующий шаг, который нам необходимо реализовать – это взаимодействие между рабочими процессами и главным процессом. Как мы уже говорили, взаимодействие между рабочими процессами и главным процессом выглядит следующим образом: после инициализации рабочих процессов (о чем мы говорили выше) главный процесс ждет сообщения от любого рабочего процесса с результатами обработки назначенному ему порции данных. При получении им такого сообщения (экземпляра записи `task_result`) главный процесс сохраняет результаты обработки в промежуточное хранилище, а его дальнейшие действия зависят от того, есть на момент получения сообщения необработанные данные или нет. Если на момент получения сообщения необработанные данные еще есть, то главный процесс «отрезает» от этих данных порцию, отсылает эту порцию рабочему процессу в виде задания на обработку (в виде экземпляра записи `task_request`), после чего продолжает ждать сообщения от рабочих процессов. Если же на момент получения сообщения необработанных данных не осталось, то главный процесс просто продолжает ждать сообщений от других процессов. Это ожидание заканчивается тогда,

когда приходит последнее сообщение с результатом обработки какой-либо порции исходных данных. Чтобы отследить это последнее сообщение, мы используем описатель процесса обработки данных. Как мы уже говорили, описатель процесса обработки данных (экземпляр записи **tasks_descr**) содержит три поля: количество созданных задач на обработку, количество обработанных задач и остаток необработанных исходных данных. Очевидно, что последним будет такое сообщение от одного из рабочих процессов, после которого количество созданных задач равно количеству выполненных задач, а остаток необработанных исходных данных пуст. Таким образом, видно, что функция, которая будет реализовывать это взаимодействие, должна иметь три варианта: один вариант – для окончания взаимодействия (и дальнейшего вычисления итогового результата), другой вариант – для ситуации, когда необработанные исходные данные уже закончились, но некоторые рабочие процессы еще выполняют свои задания, и, наконец, последний вариант для общего случая. Это будет функция **handle_workers/3**, которая определена в модуле **parallel_smartmsg_helper**, но не экспортируется из него:

```
handle_workers(#tasks_descr{created=N, processed=N, rest=[]}, Storage, _PortionSize) ->
    Storage;
handle_workers(#tasks_descr{created=Created, processed=Processed, rest=[]}, Storage, PortionSize) ->
    receive
        #task_result{index=Index, result=Dest} ->
            UpdatedStorage = collect_result(Dest, Index, Storage),
            TasksDescr = #tasks_descr{created=Created,
                processed=Processed+1, rest=[]},
            handle_workers(TasksDescr, UpdatedStorage, PortionSize);
        _ -> handle_workers(#tasks_descr{created=Created,
            processed=Processed, rest=[]}, Storage, PortionSize)
    end;
handle_workers(#tasks_descr{created=Created, processed=Processed, rest=Source}, Storage, PortionSize) ->
    receive
        #task_result{worker=Worker, index=Index, result=Dest} ->
            UpdatedStorage = collect_result(Dest, Index, Storage),
            Rest = assign_task(Worker, Source, PortionSize, Created),
            TasksDescr = #tasks_descr{created=Created+1,
                processed=Processed+1, rest=Rest},
            handle_workers(TasksDescr, UpdatedStorage, PortionSize);
        _ -> handle_workers(#tasks_descr{created=Created,
            processed=Processed, rest=Source}, Storage, PortionSize)
    end.
```

Функция **handle_workers/3** принимает три параметра: описатель процесса обработки данных (экземпляр записи **tasks_descr**), хранилище промежуточных данных и размер порции, а возвращает (после того, как все исходные данные будут «разбиты» на порции, обработаны рабочими процессами и собраны) хранилище промежуточных данных с результатами обработки всех порций. Следует обратить внимание на то, какие сообщения мы обрабатываем в функции **handle_workers/3**. Во-первых, мы обрабатываем сообщения от рабочих процессов с результатами обработки очередной порции данных; эти сообщения являются экземплярами записи **task_result**. Во-вторых, мы обрабатываем все остальные виды сообщений, не делая при этом ничего. Мы это делаем для того, чтобы очередь сообщения процесса не засорялась «мусорными» сообщениями.

Теперь мы можем собрать все вместе и создать точку входа для общей функциональности, на основе которой мы потом можем сделать соответствующие версии функций **map** и **reduce**. Это будет функция **parallel_smartmsg_helper:smartmsg_core/4**, определенная в модуле **parallel_smartmsg_helper** и экспортируемая из него:

```
smartmsg_core(FinalAggrFun, SourceList, PortionSize,
WorkerList) ->
    process_flag(trap_exit, true),
    TasksDescr = distribute_init_tasks(#tasks_descr{rest=SourceList}, PortionSize, WorkerList),
    PortionCount = parallel_common:calc_portion_count(length(SourceList), PortionSize),
    EmptyStorage = array:new([{size, PortionCount}, {fixed, true},
    {default, none}]),
    FullStorage = handle_workers(TasksDescr, EmptyStorage, PortionSize),
    process_flag(trap_exit, false),
    FinalAggrFun(array:to_list(FullStorage)).
```

В этой функции мы делаем следующее: инициализируем рабочие процессы (раздавая им начальные задания), инициализируем хранилище промежуточных данных, инициируем общение с рабочими процессами, после чего из данных, находящихся в хранилище промежуточных данных, формируем итоговый результат. Как уже говорилось выше, общение с рабочими процессами заканчивается тогда, когда все порции исходных данных обработаны и результаты их обработки получены и сохранены в хранилище промежуточных данных. Поэтому при формировании итогового результата из данных, находящихся в хранилище промежуточных данных, все результаты обработки порций в этом хранилище уже присутствуют (т.е. ровно так же, как и в предыдущих версиях, созданных нами). Следует также заметить, что при инициализации рабочих процессов (при вызове функции **distribute_init_tasks/3**) в качестве одного из параметров мы передаем начальное значение описателя процесса обработки данных. При формировании этого начального значения в качестве остатка необработанных исходных данных мы передаем сами эти исходные данные (при этом количество созданных и выполненных задач, очевидно, равно 0).

И, наконец, мы можем создать очередные многозадачные версии функций **map** и **reduce**: это будут функции **parallel_map:smartmsg_pmap/4** и **parallel_reduce:smartmsg_reduce/5**, определенные в модулях **parallel_map** и **parallel_reduce** соответственно.

Начнем мы с очередной версии многозадачной функции **map**:

```
smartmsg_pmap(_Fun, [], _PortionSize, _WorkerCount) -> [];
smartmsg_pmap(Fun, Source, PortionSize, _WorkerCount)
when length(Source) =< PortionSize ->
    lists:map(Fun, Source);
smartmsg_pmap(Fun, Source, PortionSize, WorkerCount) ->
    WorkerFun = fun(Portion) -> lists:map(Fun, Portion) end,
    Workers = [spawn_link(fun() -> parallel_smartmsg_
helper:smartmsg_worker(WorkerFun) end) || _Index <- lists:seq(1,
WorkerCount)],
    Result = parallel_smartmsg_helper:smartmsg_core(fun
lists:append/1, Source, PortionSize, Workers),
    lists:foldl(fun(Worker, _Aggr) -> exit(Worker, normal) end, true,
Workers),
    Result.
```

Легко увидеть, что ничего особо не поменялось по сравнению с предыдущей версией многозадачной функции **map**, кроме одного: мы используем функцию **parallel_smartmsg_helper:smartmsg_core/4** в качестве точки входа в общую функциональность. Теперь рассмотрим очередную многозадачную версию функции **reduce**:

```
smartmsg_reduce(_Fun, [], {Init, _PortionInit}, _PortionSize, _WorkerCount) -> Init;
smartmsg_reduce(Fun, Source, {Init, _PortionInit}, PortionSize,
_WorkerCount)
when length(Source) =< PortionSize ->
    lists:foldl(Fun, Init, Source);
smartmsg_reduce(Fun, Source, {Init, PortionInit}, PortionSize,
WorkerCount) ->
    ReduceFun = fun(List) -> lists:foldl(Fun, Init, List) end,
```

```

PortionReduceFun = fun(List) -> lists:foldl(Fun, PortionInit, List)
end,
Workers = [spawn_link(fun() -> parallel_smartmsg_
helper:smartmsg_worker(PortionReduceFun) end) || _Index <-
lists:seq(1, WorkerCount)],
Result = parallel_smartmsg_helper:smartmsg_core(ReduceFun,
Source, PortionSize, Workers),
lists:foldl(fun(Worker, _Aggr) -> exit(Worker, normal) end, true,
Workers),
Result.

```

И в этом случае ничего не поменялось по сравнению с предыдущей версией, кроме точки входа в общую функциональность.

Давайте проверим, что очередные многозадачные версии функций **map** и **reduce** работают правильно. Для этого мы компилируем соответствующие модули и запускаем консоль среды выполнения языка **Erlang**. Начнем с проверки работы функции **parallel_map:smartmsg_pmap/4**. Вызов **parallel_map:smartmsg_pmap(fun(Item)->lists:reverse(Item)) end, [], 2, 2** возвращает пустой список. Этим проверяется первый вариант функции **parallel_map:smartmsg_pmap/4**. Вызов **parallel_map:smartmsg_pmap(fun(Item)->lists:reverse(Item)) end, ["ab", "cd"], 4, 2** возвращает список **["ba", "dc"]**. Так как размер порции 4, а список с данными содержит всего 2 элемента, то мы проверяем второй вариант функции **parallel_map:smartmsg_pmap/4**. Наконец, вызов **parallel_map:smartmsg_pmap(fun(Item)->lists:reverse(Item)) end, ["ab", "cd", "ef", "gh"], 2, 2** возвращает список **["ba", "dc", "fe", "hg"]**. Так как размер порции 2, а список с данными содержит 4 элемента, то мы проверяем общий вариант функции

parallel_map:smartmsg_pmap/4. При этом будет создано 2 рабочих процесса, и оба эти процессы будут загружены, т.к. список с данными разбивается на 2 порции данных. Теперь перейдем к проверке работы функции **parallel_reduce:smartmsg_reduce/5**. Вызов **parallel_reduce:smartmsg_reduce(fun(Item, Agg)->Agg ++ Item) end, [], ("++", ""), 2, 2** возвращает строку **"++"**. Этот вызов проверяет первый вариант функции **parallel_reduce:smartmsg_reduce/5**. Вызов **parallel_reduce:smartmsg_reduce(fun(Item, Agg)->Agg ++ Item) end, ["aa", "bb"], ("++", ""), 4, 2** возвращает строку **"++aabbb"**. Так как размер порции 4, а список с данными содержит всего 2 элемента, то мы проверяем второй вариант функции **parallel_reduce:smartmsg_reduce/5**. Наконец, вызов **parallel_reduce:smartmsg_reduce(fun(Item, Agg)->Agg ++ Item) end, ["aa", "bb", "cc", "dd"], ("++", ""), 2, 2** возвращает строку **"++aabbcddd"**. Так как размер порции 2, а список с данными содержит 4 элемента, то мы проверяем общий вариант функции **parallel_reduce:smartmsg_reduce/5**. При этом будет создано 2 рабочих процесса, и оба эти процессы будут загружены, т.к. список с данными разбивается на 2 порции данных.

Итак, мы создали почти очередные многозадачные версии функций **map** и **reduce**. Может показаться, что их уже некуда улучшать и пора остановиться. Однако мы все-таки можем сделать следующее: отказаться от ручного распределения заданий на обработку (мы распределяем задания так, чтобы каждый рабочий процесс был загружен). Тогда мы переложим эту задачу на некоторый пул процессов (а точнее, узлов), который бы и распределял задания на обработку между процессами по заданным критериям. Чем мы и займемся в следующий раз. [\[xf\]](#)

Альтернатива инициализации

В данной статье при реализации очередных многозадачных версий функций **map** и **reduce** мы применили подход, при котором порции создаются по требованию, т.е. только в ответ на результат обработки некоторой предыдущей порции данных. При этом нам необходимо инициализировать рабочие процессы, чтобы данный подход работал. Для этого мы создаем и отсылаем каждому рабочему процессу порцию данных, после чего все остальные порции данных создаются по требованию. Этот алгоритм инициализации мы реализовали в функции **distribute_init_tasks/3**, которая определена в модуле **parallel_smartmsg_helper** (эта функция используется только внутри модуля **parallel_smartmsg_helper**, поэтому мы не экспортствуем ее). Очевидно, что при таком подходе мы создаем как в функции **distribute_init_tasks/3** (при инициализации рабочих процессов), так и в функции **handle_workers/3** (при взаимодействии между рабочими процессами и главным процессом) порции данных, которые определены в модуле **parallel_smartmsg_helper**.

Если же мы хотим локализовать создание порций только во время взаимодействия между рабочими процессами и главным процессом, следует применить другой подход.

Мы можем инициализировать рабочие процессы некоторыми «фейковыми» заданиями, которые ничего полезного не делают. Чтобы отличать такие «фейковые» задания от обычных заданий, мы будем задавать в качестве индекса порции число -1 (в качестве самой порции – пустой список). При этом в функции **handle_workers/3** необходимо такие «фейковые» задания обрабатывать

отдельно от основных заданий (при обработке «фейковых» заданий мы не будем заносить результат их обработки в промежуточное хранилище, а также не будем учитывать такие задания в количестве обработанных заданий описателя процесса обработки). Давайте посмотрим на альтернативную реализацию этих двух функций. Функция **distribute_init_tasks/3** будет иметь следующий вид:

```

distribute_init_tasks(#tasks_descr{rest=[]}, _PortionSize, _WorkerList) ->
    #tasks_descr{rest=[]};
distribute_init_tasks(#tasks_descr{rest=Source}, _PortionSize, Workers) ->
    lists:foreach(fun(Worker) -> assign_task(Worker, [], PortionSize, -1) end, Workers),
    #tasks_descr{rest=Source}.

```

Функция **handle_workers/3** будет иметь следующий вид:

```

handle_workers(#tasks_descr{created=N, processed=N, rest = []}, Storage, _PortionSize) ->
    Storage;
handle_workers(#tasks_descr{created=Crt, processed=Proc, rest=[]}, Storage, PortionSize) ->
    receive
        #task_result{index=-1, result=_Dest} ->
            TasksDescr = #tasks_descr{created=Crt, processed=Proc, rest=[]},
            handle_workers(TasksDescr, Storage, PortionSize);
        #task_result{index=Index, result=Dest} ->
            UpdatedStorage = collect_result(Dest, Index, Storage),
            TasksDescr = #tasks_descr{created=Crt, processed=Proc+1, rest=Rest},
            handle_workers(TasksDescr, UpdatedStorage, PortionSize);
    end.

```

```

_Other -> handle_workers(#tasks_descr{created=Crt, processed=Proc, rest=Src}, Storage, PortionSize)
end;
handle_workers(#tasks_descr{created=Crt, processed=Proc, rest=Rest}, Storage, PortionSize) ->
    receive
        #task_result{worker=Worker, index=-1, result=_Dest} ->
            Rest = assign_task(Worker, Src, PortionSize, Crt),
            TasksDescr = #tasks_descr{created=Crt+1, processed=Proc, rest=Rest},
            handle_workers(TasksDescr, Storage, PortionSize);
        #task_result{worker=Worker, index=Index, result=Dest} ->
            UpdatedStorage = collect_result(Dest, Index, Storage),
            Rest = assign_task(Worker, Src, PortionSize, Crt),
            TasksDescr = #tasks_descr{created=Crt+1, processed=Proc+1, rest=Rest},
            handle_workers(TasksDescr, UpdatedStorage, PortionSize);
    end.
_Other -> handle_workers(#tasks_descr{created=Crt, processed=Proc, rest=Src}, Storage, PortionSize)
end.

```

Какой из вариантов реализации использовать – это дело вкуса, поэтому выбор остается за читателями.