

Erlang: Опять

Андрей Ушаков возобновляет практикум по созданию многозадачных версий функций.



Наш
эксперт

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

Мы в очередной раз продолжаем в качестве практикума по многозадачности решать задачу по созданию многозадачных версий функций `map` и `reduce`. Но теперь мы создадим последний вариант этих функций и остановимся, чтобы оценить то, что мы сделали (правда, оценить наши результаты мы сможем только на следующем уроке), и пойти дальше.

Давайте посмотрим, на чем мы остановились в прошлый раз. На прошлом уроке мы переработали подход к созданию и распределению заданий для рабочих процессов при реализации первых многозадачных версий функций `map` и `reduce`. До переработки подхода к созданию и распределению заданий все исходные данные мы сразу же разбивали на порции и рассылали рабочим процессам. После переработки подхода мы стали создавать порции и отсылать их рабочим процессам только по требованию — в тот момент, когда рабочий процесс закончил обрабатывать некоторую предыдущую порцию данных и отоспал результаты обработки главному процессу.

Понятно, что при новом подходе мы не создаем одновременно большое количество новых объектов (порций данных, которые являются кортежем из двух элементов: индекс порции и собственно данные порции) и не отсылаем одновременно большое количество данных рабочим процессам. Однако следует заметить, что следующее фундаментальное свойство рабочих процессов сохраняется при переходе от старого подхода к новому: все рабочие процессы являются эквивалентными относительно распределения задач между ними. Это означает, что как только какой-либо рабочий процесс освободился, новое задание на обработку мы дадим именно этому процессу.

Здесь у читателя может возникнуть вопрос: насколько хорошо, что мы поступаем именно так при создании и назначении нового задания на обработку? Очевидно, что все зависит от расположения рабочих процессов. Если все рабочие процессы равномерно распределены между узлами (именованными экземплярами среды выполнения Erlang), причем нагрузка и среднее время выполнения задач на этих узлах примерно одинаковы, то в таком случае мы можем назначить новое задание на обработку освободившемуся процессу. При этом мы действительно получим выигрыш от использования многозадачности при обработке исходных данных, т.к. ситуация, при которой обработка некоторой порции одним из процессов тормозит весь процесс обработки исходных

данных, вряд ли возникнет. Если же конфигурация рабочих процессов такова, что выполнение одного и того же задания разными рабочими процессами сильно отличается, то подход, при котором мы новое задание назначаем освободившемуся процессу, неприменим (если мы не хотим получить вместо ускорения процесса обработки его замедление). Понятно, что в таком случае нам необходимо распределять задания между рабочими процессами каким-то другим способом. Иными словами, нам нужен некоторый механизм распределения заданий в зависимости от того или иного критерия.

Для начала рассмотрим, какие существуют механизмы для решения подобных проблем в других средах времени выполнения (и операционных системах). В большинстве таких сред создание новой единицы выполнения кода (потока либо процесса) является достаточно затратной операцией (по сравнению с процессами среды выполнения Erlang). Следует сказать, что затраты на создание новой единицы выполнения кода (потока либо процесса) различаются в зависимости от среды. Так, например, в операционной среде Linux затраты на создание нового процесса достаточно малы, поэтому в серверных приложениях для обработки запроса от клиента нередко создают новый процесс при помощи функции `fork(2)`. Более того, потоки в операционной системе Linux реализованы через процессы, разделяющие адресное пространство (и ряд других ресурсов) с родительским процессом. С другой стороны, если мы посмотрим на темную сторону силы (на операционные системы MS Windows), то в ней единицей выполнения кода является поток (а процесс является контейнером для потоков и ряда других ресурсов); причем создание потоков в операционных системах MS Windows является довольно дорогостоящей операцией. А для управляемых сред времени выполнения, таких как JVM, .NET Framework или Mono все более сложно: в них единицей выполнения кода является управляемый поток, который может соответствовать единицам выполнения кода операционной системы, а может и не соответствовать.

Но вне зависимости от среды выполнения (или операционной системы), мы не можем одновременно создать большое количество единиц выполнения кода, т.к. в худшем случае у нас закончатся системные ресурсы, а в лучшем — мы придем к ситуации, когда среда выполнения большую часть времени тратит на переключение контекста единиц выполнения кода. Поэтому в ситуации, когда необходимо создать единицу выполнения кода для выполнения некоторой задачи, рекомендуется использовать пул единиц выполнения кода. Обычно для планирования выполнения заданий используется пул потоков. В некоторых ситуациях можно использовать и пул процессов (если таковой есть); так, например, в языке Python для распараллеливания вычислений рекомендуется использовать пул процессов (воспользовавшись стандартным модулем `multiprocessing`) из-за GIL (Global Interpreter Lock, см. врезку).

А теперь вернемся к нашей проблеме: нам необходим некоторый механизм назначения заданий рабочим процессам в зависимости от того или иного критерия. В отличие от большинства других сред времени выполнения (и операционных систем), идеология среды выполнения Erlang подразумевает, что мы можем

Global Interpreter Lock

Выражение Global Interpreter Lock (GIL) переводится как «глобальная блокировка интерпретатора». Это средство синхронизации, используемое потоком интерпретатора некоторого языка программирования, чтобы избежать совместного выполнения небезопасного (с точки зрения многопоточности) кода другими потоками. Как

следует из названия, данная блокировка существует в единственном экземпляре в пределах процесса интерпретатора. Следствие ее применения — тот факт, что для параллельного выполнения некоторых задач надо использовать процессы, а не потоки. Примерами интерпретаторов языков с GIL являются CPython и CRuby.

МНОГО ЗАДАЧ

создавать столько процессов, сколько нужно (однако не следует забывать, что есть ограничение на количество процессов для каждого экземпляра среды выполнения Erlang), не опасаясь роста накладных расходов на поддержание их работы. Поэтому для нас нет смысла в пуле процессов (языка Erlang): нам проще создать новый процесс, чем иметь набор предопределенных процессов.

С другой стороны, если на одном узле (в одном экземпляре среды выполнения Erlang) мы будем создавать все новые и новые процессы, то начиная с какого-то момента остановимся без прироста производительности (вызванного одновременным выполнением задач). Связано это с тем, что количество процессоров и/или ядер процессоров ограничено и обычно достаточно мало. Максимальное количество одновременно работающих единиц выполнения кода равно сумме всех ядер всех процессоров в системе (или произведению количества процессоров на количество ядер на одном процессоре, если у всех процессоров одинаковое количество ядер). И если количество единиц выполнения кода, запланированных для выполнения, превышает это максимальное количество, то некоторые из единиц выполнения кода будут простаивать, а система должна будет затрачивать время на переключение с одной единицы выполнения кода на другую, чтобы все единицы выполнения кода получили процессорное время.

Конечно, для среды времени выполнения Erlang временем переключения процессов можно пренебречь, но все остальное будет справедливо: чем больше процессов мы создадим, тем больше их будет простаивать в ожидании шанса на выполнение.

Очевидно, что в такой ситуации мы должны создавать процессы на разных узлах (именованных экземплярах среды выполнения Erlang), причем эти узлы должны располагаться на разных компьютерах, иначе все процессы будут выполняться на одних и тех же физических ядрах и процессорах, что будет эквивалентно ситуации с одним экземпляром среды выполнения. Поэтому наш механизм назначения заданий должен выбирать (или уметь создавать в определенных ситуациях) некоторый узел, на нем создавать процесс и выполнять с его помощью интересующее нас задание. Обычно для работы такого механизма назначения задач необходимо создать некоторый предопределенный набор узлов. Из этого набора узлов механизм назначения заданий будет по какому-либо критерию выбирать наиболее подходящий узел и выполнять задание на нем (по такому же принципу работают и другие пулы, например, пул потоков). И, соответственно, мы будем называть этот механизм пулом узлов.

В нашем очередном примере в качестве механизма назначения заданий мы будем использовать пул узлов. Это означает, что наш пример будет абстрагирован от выбора процесса для выполнения очередного задания. Более того, при правильной реализации примера мы сможем легко поменять этот механизм назначения заданий на какой-либо другой, не затрагивая код примера. При реализации нашего примера мы поступим точно так же, как поступали раньше: создадим модуль (в нашем примере это будет модуль *parallel_pool_helper*), содержащий общую функциональность, на основе которой мы в дальнейшем реализуем многозадачные версии функций *map* и *reduce*.

Альтернатива для rsh

Для создания узлов на удаленных хостах среда времени выполнения Erlang использует утилиту *rsh*. Если данной утилиты у пользователя нет и/или есть желание использовать альтернативу этой про-

грамммы, то это можно сделать следующим способом. При старте экземпляра среды выполнения Erlang следует добавить ключ “-rsh Program”, где *Program* — аналог утилиты *rsh*.

В качестве следующего шага, определим ряд записей, чтобы сделать код более понятным (причем эти записи определены только на уровне модуля *parallel_pool_helper*):

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

Запись **task_descr** предназначена для хранения данных о процессе обработки исходного списка: поле **created** содержит количество созданных заданий, поле **processed** — количество выполненных заданий, поле **rest** — остаток необработанных данных. Очевидно, что когда в поле **rest** пустой список, новых заданий создавать не надо; если поля **created** и **processed** содержат одно и то же значение — значит, мы обработали все исходные данные и результаты обработки сохранили в промежуточном хранилище. Запись **task_result** определена для передачи результатов обработки исходных данных: поле **index** содержит индекс порции данных, поле **result** — результат обработки порции исходных данных. Отметим, что, в отличие от предыдущего примера (из LXF170), запись **task_result** не содержит идентификатор рабочего процесса, обработавшего данную порцию данных. Дело в том, что там мы этому рабочему процессу назначали новую порцию данных; а сейчас мы перекладываем эту ответственность на внешний механизм назначения заданий (на пул узлов, в нашем случае).

Теперь давайте создадим функцию, которую будут выполнять рабочие процессы. Это будет экспортруемая функция *parallel_pool_helper:pool_worker/3*:

```
pool_worker(Fun, Portion, Index, Master) ->  
    Result = Fun(Portion),  
    Master ! #task_result{index=Index, result=Result}.
```

Она проще соответствующей функции для рабочих процессов из предыдущего примера: в ней нам нет необходимости организовывать цикл обработки сообщений от родительского процесса. Связано это с тем фактом, что в данном примере мы не управляем рабочими процессами напрямую, а полагаемся на некоторый механизм назначения заданий (например, на пул узлов). Это, в частности, означает, что мы не знаем ничего о рабочих процессах, которые обрабатывают наши порции данных (не знаем идентификаторы этих процессов). Как уже говорилось, эта функция экспортруется из модуля *parallel_pool_helper*, а в модуле *parallel_pool_helper* мы хотим полностью абстрагироваться от механизма назначения заданий; поэтому клиенты нашего модуля должны знать, какую функцию должны выполнять рабочие процессы при назначении конкретного механизма распределения заданий. Конечно, мы можем убрать эту зависимость клиентов от функции

»

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

Про файл .hosts.erlang

Файл `.hosts.erlang` содержит имена хостов, записанные в виде выражений языка Erlang. Формат его таков: каждая строка содержит только одно имя хоста, заключенное в одиночные кавычки, и заканчивается точкой (т.к. это выражение языка Erlang). Последняя строка файла должна быть пустой. Вот пример такого файла:

```
'host1.somecorp.ru'.
```

```
'host2.somecorp.ru'.
```

```
<пустая строка>
```

Файл `.hosts.erlang` может располагаться в следующих местах системы: в текущем рабочем каталоге, в домашнем каталоге текущего пользователя и в корневом каталоге Erlang/OTP (в `$OTP_ROOT`). Именно в таком порядке происходит поиск файла `.hosts.erlang`, пока он не будет найден.

для выполнения рабочими процессами и передавать ее через определенный нами интерфейс взаимодействия (этот интерфейс является просто функцией) с механизмом назначения заданий, но при этом код станет более сложным, хотя при создании реальных программ указанный подход предпочтительнее.

В качестве следующего шага создадим ряд функций для инкапсуляции таких операций, как создание и назначение новой задачи рабочему процессу и сохранение результата обработки порции в промежуточном хранилище. Это будут функции `collect_result/3` и `assign_task/4`, определенные в модуле `parallel_pool_helper`, но не экспортируемые из него:

```
collect_result(Result, Index, Storage) ->
    array:set(Index, Result, Storage).
assign_task(Source, PortionSize, Index, AssignFun) when
    length(Source) =< PortionSize ->
    AssignFun(Source, Index, self()),
    [];
assign_task(Source, PortionSize, Index, AssignFun) ->
    {Portion, Rest} = lists:split(PortionSize, Source),
    AssignFun(Portion, Index, self()),
    Rest.
```

Функция `collect_result/3` сохраняет результат обработки порции в промежуточном хранилище (массиве). Эта функция ничем не отличается от подобной функции из предыдущего примера. Функция `assign_task/4` служит для создания и назначения задания на обработку порции данных рабочему процессу. На вход она принимает необработанный остаток исходных данных `Source`, размер порции `PortionSize`, индекс очередной порции данных `Index` и интерфейс (интерфейсную функцию) к механизму назначения заданий `AssignFun`. В функции `assign_task/4` мы выделяем очередную порцию данных, отаем ее в виде задания на обработку рабочему процессу и возвращаем остаток исходных данных после создания задания. Понятно, что при этом в функции `assign_task/4` мы должны обрабатывать два возможных случая: когда размер остатка исходных данных больше размера порции данных и когда размер остатка исходных данных меньше или равен размеру порции данных. В первом случае мы разбиваем остаток исходных данных на порцию, которую будет обрабатывать рабочий процесс, и новый остаток; во втором случае весь остаток будет обработан, а новым остатком будет пустой список. Для назначения задания рабочему процессу мы используем интерфейс к механизму назначения заданий, который является функцией `AssignFun`. В нее мы передаем очередную порцию для обработки, индекс этой порции и идентификатор главного процесса. В этом заключается отличие функции `assign_task/4` от подобной функции, определенной в предыдущем примере.

В нашем примере мы используем следующий подход к назначению заданий рабочим процессам: как только мы получаем

результат обработки какой-либо порции, мы назначаем новое задание на обработку очередной порции. При этом количество рабочих процессов, обрабатывающих порции исходных данных, ограничено (и гораздо меньше количества этих порций). Но чтобы этот подход работал, необходимо одно предусловие: мы должны создать и назначить рабочим процессам определенное количество (равное количеству одновременно выполняющихся рабочих процессов) заданий на обработку. Для этого мы создадим функцию `distribute_init_tasks/4` в модуле `parallel_pool_helper` (но не экспортируем ее из этого модуля):

```
distribute_init_tasks(#tasks_descr{created=Created, rest=[]}, _PortionSize, _AssignFun, _WorkerCount) ->
    #tasks_descr{created=Created, rest=[]};
distribute_init_tasks(#tasks_descr{created=WorkerCount, rest=Rest}, _PortionSize, _AssignFun, WorkerCount) ->
    #tasks_descr{created=WorkerCount, rest=Rest};
distribute_init_tasks(#tasks_descr{created=Created, rest=Source}, PortionSize, AssignFun, WorkerCount) ->
    Rest = assign_task(Source, PortionSize, Created, AssignFun),
    TasksDescr = #tasks_descr{created=Created+1, rest=Rest},
    distribute_init_tasks(TasksDescr, PortionSize, AssignFun, WorkerCount).
```

При работе функции `distribute_init_tasks/4` возможно возникновение следующих 3-х ситуаций, которые эта функция должна уметь обрабатывать (именно поэтому она содержит 3 варианта). Во-первых, у нас остаток необработанных исходных данных может быть пустым. Во-вторых, мы можем создать необходимое количество порций данных и раздать их рабочим процессам в качестве заданий на обработку. И, наконец, возможна ситуация, когда у нас есть еще исходные данные для обработки, и мы не создали необходимого количества заданий для обработки.

Очевидно, что в первых двух ситуациях следует завершить выполнение функции `distribute_init_tasks/4`; в этом случае мы возвращаем соответствующие данные о процессе обработки исходного списка (экземпляр записи `task_descr`). В последнем случае, мы создаем очередную порцию, назначаем ее в качестве задания какому-то рабочему процессу и рекурсивно вызываем сами себя. Так как при каждом вызове третьего варианта функции `distribute_init_tasks/4` у нас объем необработанных исходных данных уменьшается (на размер порции данных), а количество созданных задач на обработку увеличивается на 1, то, в конце концов, мы придем к вызову либо первого варианта, либо второго варианта функции `distribute_init_tasks/4`. Это означает, что функция `distribute_init_tasks/4` написана корректно и не приведет к бесконечному рекурсивному вызову самой себя.

После того, как мы научились инициализировать рабочие процессы необходимым количеством заданий, пришла пора организовать взаимодействие между рабочими процессами и главным. Это взаимодействие заключается в следующем: мы должны ожидать сообщения от рабочих процессов с результатами обработки порций данных. При получении такого сообщения мы должны сохранять результаты обработки порций в промежуточное хранилище, после чего, если еще есть необработанные данные, создавать новое задание и отправлять его на обработку. Обрабатывать это задание будет не обязательно тот процесс, от которого пришло сообщение с результатом обработки некоторой предыдущей порции. Соответственно, у нас возможны следующие 3 ситуации: у нас есть еще исходные данные для обработки; исходные данные для обработки закончились, но не все рабочие процессы закончили выполнение своих заданий (обработку порций исходных данных); и, наконец, у нас закончились все исходные данные и все рабочие процессы завершили свою работу. Очевидно, что

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

в первом случае мы будем создавать новые задания на обработку исходных данных; во втором случае мы этого делать не будем; а в третьем случае — закончим работу (и вернем хранилище промежуточных данных, которое будет содержать результаты обработки всех порций данных). Это взаимодействие между рабочими процессами и главным процессом с описанным выше поведением мы реализуем в функции **handle_workers/4**, определенной в модуле *parallel_pool_helper*, но не экспортруемой из него:

```
handle_workers(#tasks_descr{created=N, processed=N, rest=[]}, Storage, _PortionSize, _AssignFun) ->
    Storage;
handle_workers(#tasks_descr{created=C, processed=P, rest=[]}, Storage, PortionSize, AssignFun) ->
    receive
        #task_result{index=Index, result=Dest} ->
            UpdatedStorage = collect_result(Dest, Index, Storage),
            TasksDescr = #tasks_descr{created=C, processed=P+1, rest=[]},
            handle_workers(TasksDescr, UpdatedStorage, PortionSize, AssignFun);
        _ -> handle_workers(#tasks_descr{created=C, processed=P, rest=[]}, Storage, PortionSize, AssignFun)
    end;
handle_workers(#tasks_descr{created=C, processed=P, rest=Src}, Storage, PortionSize, AssignFun) ->
    receive
        #task_result{index=Index, result=Dest} ->
            UpdatedStorage = collect_result(Dest, Index, Storage),
            Rest = assign_task(Src, PortionSize, C, AssignFun),
            TasksDescr = #tasks_descr{created=C+1, processed=P+1, rest=Rest},
            handle_workers(TasksDescr, UpdatedStorage, PortionSize, AssignFun);
        _ -> handle_workers(#tasks_descr{created=C, processed=P, rest=Src}, Storage, PortionSize, AssignFun)
    end.
```

Видно, что функция **handle_workers/4** содержит 3 варианта, каждый из которых обрабатывает одну из 3-х возможных ситуаций во время взаимодействия между главным процессом и рабочими процессами.

А теперь пришла пора объединить все вместе: давайте создадим точку входа для общей функциональности, на основе которой создадим очередные версии функций **map** и **reduce**. Это будет функция **parallel_pool_helper:pool_core/5**, определенная в модуле *parallel_pool_helper* и, что естественно, экспортруемая из него:

```
pool_core(FinalAggrFun, Source, PortionSize, AssignFun,
WorkerCount) ->
    process_flag(trap_exit, true),
    PortionCount = parallel_common:calc_portion_
count(length(Source), PortionSize),
    TasksDescr = distribute_init_tasks(#tasks_descr{rest=Source},
PortionSize, AssignFun, WorkerCount),
    EmptyStorage = array:new([{size, PortionCount}, {fixed, true},
{default, none}]),
    FullStorage = handle_workers(TasksDescr, EmptyStorage,
PortionSize, AssignFun),
    process_flag(trap_exit, false),
    FinalAggrFun(array:to_list(FullStorage)).
```

В этой функции мы создаем и назначаем рабочим процессам определенное количество заданий (равное значению параметра **WorkerCount**), после чего взаимодействуем с рабочими процессами, пока все исходные данные не будут обработаны рабочими процессами (и все результаты обработки не будут собраны в промежуточном хранилище); и, наконец, из результатов обработки порций формируем итоговый результат. В качестве одного из параметров,

в функцию **parallel_pool_helper:pool_core/5** мы передаем интерфейс (а точнее, интерфейсную функцию) к механизму назначения заданий. Этим параметром является параметр **AssignFun**. Это означает, что определенная нами общая функциональность не зависит от используемого механизма назначения заданий, и любой клиент, использующий эту нашу общую функциональность, может использовать любой удобный ему механизм. После создания необходимой общей функциональности, определенной в модуле *parallel_pool_helper*, мы можем перейти к созданию на основе этой функциональности очередных версий функций **map** и **reduce**. Но прежде чем их создавать, следует выбрать механизм назначения заданий. Как уже говорилось выше, логично выбрать некоторый пул узлов. В стандартной библиотеке языка Erlang такой пул узлов есть: он определен в модуле *pool*. Этот пул узлов может как самостоятельно создавать узлы на определенных хостах, так и использовать некоторые уже созданные узлы.

Для создания узлов на хостах используется программа *rsh* (или ее альтернатива). Поэтому, если у клиента (на каких-либо узлах) не установлена эта программа (или не указано использование ее альтернативы) или если клиент находится на темной стороне силы (использует операционную систему семейства Microsoft Windows), то пул узлов создавать узлы на этих хостах не сможет. В этом случае клиент может сам создать необходимое количество узлов на этих хостах и приказать пулу узлов использовать их.

Для начала работы с пулом узлов необходимо вызвать одну из функций **pool:start/1** или **pool:start/2**. Если есть такая возможность (см. врезку про программу *rsh* на стр. 85), на каждом хосте, описанном в файле **.hosts.erlang**, будет создан свой узел. Если при старте пул узлов создать узлы не смог или же есть необходимость добавить дополнительные узлы в пул, можно воспользоваться функцией **pool:attach/1**, которая добавляет в пул узлов узел, переданный в качестве аргумента.

Для выполнения некоторого задания (вычисления функции, определенной в некотором модуле, с заданными аргументами) используется одна из функций **pool:pspawn/1** или **pool:pspawn_link/3**. Обе эти функции создают процесс на одном из узлов пула, где ожидается наименьшая загрузка, и возвращают идентификатор созданного процесса, что позволяет дополнительно с ним взаимодействовать. Их отличие только в том, что первая функция просто создает процесс, а вторая, помимо процесса, создает еще и связь между созданным и вызывающим процессом. И, наконец, чтобы закончить работу с пулом узлов, необходимо вызвать функцию **pool:stop/0**; при этом все узлы пула будут уничтожены.

В очередной раз мы вынуждены остановиться из-за того, что место, выделяемое для статьи, ограничено. В следующий раз мы закончим пример на основе пула узлов, который начали в этом номере, а также подведем итоги решения задачи по созданию многозадачных версий функций **map** и **reduce**. [LXF170](#)

Альтернативная инициализация

Пример, приведенный в данной статье, по своей структуре похож на пример из [LXF170](#). Как и там, задания на обработку создаются у нас в двух местах: во-первых, в функции **distribute_init_tasks/4** для создания необходимого количества рабочих процессов (для их инициализации); во-вторых, в функции **handle_workers/4** для организации взаимодействия между рабочими процессами и главным процессом (и для обработки оставшихся данных, естественно). Если кого-то не устраивает,

что создание заданий на обработку у нас «размазано» по двум функциям, можно создать альтернативный вариант инициализации рабочих процессов: инициализировать их некоторыми «псевдозаданиями». Пример такой альтернативной инициализации мы приводили в [LXF170](#). Похожую альтернативную инициализацию рабочих процессов можно сделать и для текущего примера, но мы оставим такую реализацию в качестве домашнего задания для тех, кому это интересно.