

Erlang: Опять

Андрей Ушаков вынужден будет приостановить полет мысли из-за недостатка места. Но уж в следующий раз мы все узнаем!



Наш
эксперт

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

Mы продолжаем решение нашей задачи: создание многозадачных версий функций `map` и `reduce`. Но на сей раз мы постараемся завершить начатое в предыдущей статье и не начнем ничего нового.

По традиции, вспомним, на чем мы остановились в прошлой статье. Мы поговорили о том, что явное управление назначением заданий рабочим процессам далеко не всегда эффективно. Мы помним, что раньше мы назначали задание на обработку порции данных тому рабочему процессу, который обработал некоторую порцию исходных данных и послал результаты обработки главному процессу. Предположим, что мы назначаем задания процессам 1, 2 и 3, располагающимся каждый на своем узле (а каждый из узлов — на отдельном компьютере). Предположим, что узел, на котором располагается процесс 3, сильно загружен, что выражается в более долгой обработке порций исходных данных этим процессом. В наиболее вырожденном случае процессы 1 и 2 могут обработать все остальные порции быстрее, чем процесс 3 обработает назначенную ему порцию. В итоге мы не получаем ожидаемого ускорения обработки исходных данных за счет ее распараллеливания, а в вырожденном случае обрабатываем данные даже медленнее, чем при обычной, не параллельной обработке. Конечно, если рабочие процессы выполняются на узлах, расположенных на одинаковых компьютерах с одинаковой загрузкой (как узлов, так и операционной системы в целом), то такая ситуация вряд ли возможна. Но уж если она имеет место, необходимо использовать какой-то другой механизм назначения заданий вместо назначения задания процессу, вернувшему результат обработки некоторой порции данных.

Проблема назначения задания некоторой единице выполнения кода (потоку и/или процессу) не является специфичной для языка Erlang. Во многих средах времени выполнения и операционных

системах вопрос о назначении задания некоторой единице выполнения кода решается при помощи пула потоков. Пул потоков — это набор некоторых уже созданных потоков, которые служат для выполнения того или иного пользовательского задания. Причем когда поток из пула выполнит пользовательское задание, он не прекращает свое существование (как это бывает с потоками, созданными специально для выполнения пользовательских заданий), а «возвращается» обратно в пул потоков. Существование подобного механизма связано с тем, что в таких средах времени выполнения и операционных системах создание новой единицы выполнения кода — достаточно дорогая операция. В среде времени выполнения Erlang ситуация с созданием новых процессов кардинально иная. Создание процессов является настолько легким, что рекомендуется создавать их столько, сколько нужно для решения той или иной задачи. Поэтому создание некоторого пула процессов языка Erlang не имеет особого смысла. С другой стороны, мы можем создать пул узлов, который будет выполнять пользовательские задания на одном из своих узлов (естественно, создавая для этого процесс) в зависимости от того или иного условия (например, от загрузки узлов). Очевидно, создание такого пула узлов имеет смысл; и такой пул узлов в стандартной библиотеке языка Erlang есть.

Разобравшись с возможным решением проблемы назначения очередного задания на обработку, мы перешли к реализации общей функциональности, на базе которой мы в дальнейшем реализуем очередные многозадачные версии функций `map` и `reduce`. Эту общую функциональность мы реализовали в модуле `parallel_pool_helper`. Из этого модуля мы экспортим две функции: точку входа в общую функциональность — функцию `parallel_pool_helper:pool_core/5` и тело рабочего процесса — функцию `parallel_pool_helper:pool_worker/4`. При реализации этой общей функциональности мы приняли важное решение, договорившись, что конкретизировать механизм назначения заданий мы будем в функциях, использующих нашу общую функциональность. Для этого в функцию `parallel_pool_helper:pool_core/5` в качестве одного из параметров мы передаем интерфейс (интерфейсную функцию) к механизму назначения заданий. При таком подходе наша общая функциональность не будет зависеть от конкретного механизма назначения заданий, специфицировать который мы будем в функциях, использующих нашу общую функциональность. Чтобы упростить интерфейс к механизму назначения заданий, мы решили экспортить функцию `parallel_pool_helper:pool_worker/4` из модуля `parallel_pool_helper`, хотя делать это не обязательно.

И, наконец, мы посмотрели, что же представляет собой пул узлов из стандартной библиотеки языка Erlang. Пул узлов определен в модуле `pool`. Этот модуль может как самостоятельно создавать узлы на основе имен хостов, определенных в файле `.hosts.erlang`, так и использовать заранее созданные узлы. Для начала работы с пулом узлов необходимо вызвать одну из функций `pool:start/1` или `pool:start/2`; при этом пул узлов создает узлы на хостах, определенных в файле `.hosts.erlang`, если есть такая возможность. Чтобы использовать заранее созданные узлы,

Исходники библиотеки Erlang

На данном уроке мы заглянем в исходные коды модуля `pool`. Однако мы не затронули вопрос, а где вообще лежат исходные коды библиотек языка Erlang (и в частности, модуля `pool`). Сейчас мы этот вопрос разберем. Корневая директория среды выполнения Erlang (та, в которую мы устанавливаем среду выполнения Erlang) определяется переменной среды окружения `$OTPROOT`; также она может быть определена через вызов функции `code:root_dir/0`. Директория `$OTPROOT/lib` содержит все библиотеки, идущие вместе со средой выполнения. Согласно принципам OTP, все библиотеки принадлежат некоторому приложению; поэтому директория `$OTPROOT/lib` содержит поддиректории, соответствующие таким приложениям.

Каждая из директорий приложений всегда содержит следующие поддиректории: `doc`, `ebin`, `src`. Директория `doc` содержит документацию на приложение (на модули, которые данное приложение содержит). Директория `ebin` содержит все модули в откомпилированном виде (в виде файлов `*.beam`). И, наконец, директория `src` содержит исходные коды всех модулей. Следует также учесть, что директории приложений помимо имени приложения содержат также его версию. Так, например, на Ubuntu `$OTPROOT` будет `/usr/lib/erlang`, модуль `pool` принадлежит приложению `stdlib`, поэтому полный путь до исходного кода модуля `pool` будет следующий `/usr/lib/erlang/lib/stdlib-1.18.1/src/pool.erl`.

практикуемся

необходимо каждый из таких узлов добавить в пул при помощи функции `pool:attach/1`. При этом временем жизни такого переданного узла будет управлять пул узлов. Для выполнения некоторого задания (вычисления функции, определенной в некотором модуле, с заданными аргументами) используется одна из функций `pool:pspawn/3` 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 и связь между созданным процессом и вызывающим процессом. И, наконец, чтобы закончить работу с пулом узлов, необходимо вызвать функцию `pool:stop/0`; при этом все узлы пула будут уничтожены.

А теперь мы можем пойти дальше и реализовать очередные многозадачные версии функций `map` и `reduce`. Эти функции будут использовать общую функциональность, определенную нами в прошлый раз; причем в качестве механизма назначения заданий мы будем использовать пул узлов, определенный в модуле `pool`. При этом нам необходимо учесть следующий факт: в зависимости от операционной системы и окружения пул узлов либо сможет создать узлы на хостах, определенных в файле `.hosts.erlang`, либо не сможет. Кроме того, в каких-то ситуациях пользователи наших функций `map` и `reduce` могут захотеть использовать созданные ими узлы в пуле узлов. Поэтому наши очередные версии функций `map` и `reduce` должны принимать в качестве одного из параметров список узлов, который будет передан пулу узлов. Соответственно, в начале выполнения функций `map` и `reduce` мы должны инициализировать (при помощи одной из функций `pool:start/1` или `pool:start/2`) пул узлов, а в конце выполнения этих функций завершить его работу (при помощи функции `pool:stop/0`). Если же в силу тех или иных причин вам необходимо использовать уже инициализированный пул узлов и нет нужды его останавливать, то наша реализация функций `map` и `reduce` вам не подойдет. На самом деле это не является проблемой, но об этом мы поговорим, когда рассмотрим реализацию функций `map` и `reduce`.

Прежде чем приступить к долгожданной реализации функций `map` и `reduce`, давайте реализуем пару функций, упрощающих нам работу с пулом узлов. Это будут функции `pool_helper:start_pool/2` и `pool_helper:stop_pool/0`, экспортируемые из модуля `pool_helper`:

```
start_pool(Prefix, PoolNodes) ->
    pool:start(Prefix),
    lists:foreach(fun(Node) -> pool:attach(Node) end, PoolNodes).
stop_pool() ->
    pool:stop().
```

Функция `pool_helper:start_pool/2` инициализирует пул узлов, после чего добавляет узлы из списка `PoolNodes` в пул узлов. Функция `pool_helper:stop_pool/0` завершает работу пула узлов. Видно, что функция `pool_helper:stop_pool/0` всего лишь вызывает функцию `pool:stop/0`. Может показаться, что введение функции `pool_helper:stop_pool/0` излишне и ее стоит удалить. На самом деле для создания этой функции есть две причины. Во-первых, для симметрии — чтобы у функции инициализации пула узлов

`pool_helper:start_pool/2` была функция-антитипод для завершения работы узлов; во-вторых, для возможных будущих изменений, когда при завершении работы пула узлов нам необходимо будет выполнять еще какие-либо действия.

Пришла долгожданная пора реализовать очередные версии функций `map` и `reduce`. Начнем мы по традиции с многозадачной версии функции `map`, основанной на использовании пула узлов. В этой функции мы инициализируем пул узлов (при помощи функции `pool_helper:start_pool/2`), вычисляем результат операции отображения исходных данных, после чего завершаем работу пула узлов (при помощи функции `pool_helper:stop_pool/0`) и возвращаем результат. В нашем случае это будет функция `parallel_map:pool_pmap/5`, определенная в модуле `parallel_map` и экспортная из него:

```
pool_pmap(Fun, Source, PortionSize, WorkerCount, PoolNodes) ->
    pool_helper:start_pool(pmap, PoolNodes),
    Result = pool_pmap_impl(Fun, Source, PortionSize,
    WorkerCount),
    pool_helper:stop_pool(),
    Result.
```

Результат операции отображения исходных данных вычисляется (естественно, многозадачным образом) в функции `parallel_map:pool_pmap_impl/4`:

```
pool_pmap_impl(_Fun, [], _PortionSize, _WorkerCount) -> [];
pool_pmap_impl(Fun, Source, PortionSize, _WorkerCount)
when length(Source) =< PortionSize -> lists:map(Fun, Source);
pool_pmap_impl(Fun, Source, PortionSize, WorkerCount) ->
    WorkerFun = fun(Portion) -> lists:map(Fun, Portion) end,
    AssignFun = fun(Portion, Index, Master) -> pool:pspawn_link(parallel_pool_helper, pool_worker, [WorkerFun, Portion,
    Index, Master]) end,
    parallel_pool_helper:pool_core(fun lists:append/1, Source,
    PortionSize, AssignFun, WorkerCount).
```

Эта функция содержит три варианта и обрабатывает, помимо общего случая, еще пару граничных случаев. Первый вариант функции `parallel_map:pool_pmap_impl/4` обрабатывает случай, когда исходные данные отсутствуют (т. е. исходные данные — пустой список). Второй вариант этой функции обрабатывает случай, когда размер исходных данных не больше размера порции данных; в этом случае нет смысла обрабатывать исходные данные многозадачным способом. И, наконец, третий вариант функции `parallel_map:pool_pmap_impl/4` обрабатывает общий случай, когда мы используем нашу многозадачную реализацию из модуля `parallel_pool_helper`. В этом варианте функции `parallel_map:pool_pmap_impl/4` мы создаем интерфейс (интерфейсную функцию `AssignFun`) к механизму назначения заданий на базе пула узлов (в нашем случае — к функции `pool:pspawn/3`), после чего используем общую функциональность, реализованную в модуле `parallel_pool_helper`. Если сравнить функции `parallel_map:pool_pmap/5` и `parallel_map:pool_pmap_impl/4`, то видно, что функция `parallel_map:pool_pmap_impl/4` вычисляет результат операции отображения уже с использованием готового пула узлов и не управляет его жизнью, в отличие от функции

>>

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

Создать условия пулу

При инициализации (которая происходит при помощи вызова одной из функций `pool:start/1` или `pool:start/2`) пул узлов может создавать узлы на хостах, определенных в файле `.hosts.erlang`. Для создания узлов на удаленных хостах используется утилита `rsh`. Можно использовать альтернативу данной утилиты; для этого необходимо среди выполнения Erlang запускать с ключом `-rsh` и именем альтернативы, отделенным пробелом. Естественно, что когда мы используем

темную сторону силы (одну из операционных систем семейства Microsoft Windows), то создавать узлы на удаленных хостах мы не можем. Тем не менее, можно создавать узлы на локальном хосте. Однако тут существует небольшая хитрость. Если при задании локального хоста (в файле `.hosts.erlang`) мы используем имя `localhost`, то узел создан не будет. Чтобы узел на локальном хосте был создан, необходимо использовать имя этого хоста, например, `stdstring`.

parallel_map:pool_pmap/5. Таким образом, мы экспортствуем из модуля `parallel_map` обе функции `parallel_map:pool_pmap/5` и `parallel_map:pool_pmap_impl/4`. Первую функцию мы используем тогда, когда нам не нужно управлять временем жизни пула узлов и нас устраивает сценарий, что пул узлов будет создан в начале ее выполнения и завершен в конце ее выполнения. Вторую же функцию мы используем тогда, когда у нас есть некоторый пул узлов и мы желаем управлять его временем жизни (в этом случае, в отличие от предыдущего, у нас получается долгоживущий пул узлов).

После реализации многозадачной версии функции `map` перейдем к рассмотрению многозадачной версии функции `reduce`, основанной на использовании пула узлов. В этой функции мы инициализируем пул узлов (при помощи функции `pool_helper:start_pool/2`), вычисляем результат операции свертки исходных данных, после чего завершаем работу пула узлов (при помощи функции `pool_helper:stop_pool/0`) и возвращаем результат. В нашем случае это будет функция `parallel_reduce:pool_reduce/6`, определенная в модуле `parallel_reduce` и экспортствуемая из него:

```
pool_reduce(Fun, Source, {InitValue, PortionInitValue},  
PortionSize, WorkerCount, Nodes) ->  
    pool_helper:start_pool(pmap, Nodes),  
    Result = pool_reduceImpl(Fun, Source, {InitValue,  
PortionInitValue}, PortionSize, WorkerCount),  
    pool_helper:stop_pool(),  
    Result.
```

Результат операции свертки исходных данных вычисляется (естественно, многозадачным образом) в функции `parallel_reduce:pool_reduceImpl/5`:

```
pool_reduceImpl(_Fun, [], {InitValue, _PortionInitValue}, _  
PortionSize, _WorkerCount) ->  
    InitValue;  
pool_reduceImpl(Fun, Source, {InitValue, _PortionInitValue},  
PortionSize, _WorkerCount)  
when length(Source) =< PortionSize -> lists:foldl(Fun, InitValue,  
Source);  
pool_reduceImpl(Fun, Source, {InitValue, PortionInitValue},  
PortionSize, WorkerCount) ->  
    WorkerFun = fun(Portion) -> lists:foldl(Fun, PortionInitValue,  
Portion) end,  
    AssignFun = fun(Portion, Index, Master) -> pool:pspawn_  
link(parallel_pool_helper, pool_worker, [WorkerFun, Portion,  
Index, Master]) end,  
    FinalAggFun = fun(List) -> lists:foldl(Fun, InitValue, List) end,
```

```
parallel_pool_helper:pool_core(FinalAggFun, Source,  
PortionSize, AssignFun, WorkerCount).
```

Эта функция содержит три варианта и обрабатывает, помимо общего случая, еще пару граничных случаев. Первый вариант функции `parallel_reduce:pool_reduceImpl/5` обрабатывает случай, когда исходные данные отсутствуют (когда исходные данные — пустой список). Второй вариант этой функции обрабатывает случай, когда размер исходных данных не больше размера порции данных; тогда нет смысла многозадачным способом обрабатывать исходные данные. И, наконец, третий вариант функции `parallel_reduce:pool_reduceImpl/5` обрабатывает общий случай, когда мы используем нашу многозадачную реализацию из модуля `parallel_pool_helper`. В этом варианте функции `parallel_reduce:pool_reduceImpl/5` мы создаем интерфейс (интерфейсную функцию `AssignFun`) к механизму назначения заданий на базе пула узлов (в нашем случае — к функции `pool:pspawn/3`), после чего используем общую функциональность, реализованную в модуле `parallel_pool_helper`. Про отношение между функциями `parallel_reduce:pool_reduce/6` и `parallel_reduce:pool_reduceImpl/5` можно сказать то же, что и про отношение между функциями `parallel_map:pool_pmap/5` и `parallel_map:pool_pmapImpl/4`. Первая функция позволяет не задумываться об управлении жизнью пула узлов и берет ответственность за это на себя (за создание пула узлов, добавление необходимых узлов к пулу узлов и завершение его работы). Вторая функция перекладывает ответственность за управление жизнью пула узлов на вызывающую сторону и полагается во время своей работы на то, что пул узлов создан и нужным образом проинициализирован. Естественно, что мы экспортствуем из модуля `parallel_reduce` обе эти функции.

Давайте проверим, что наши реализации многозадачных версий функций `map` и `reduce` на базе пула узлов работают. Прежде чем начать проверку, давайте слегка модифицируем функцию `parallel_pool_helper:pool_worker/4` следующим образом: добавим в начало тела этой функции вывод имени узла на консоль, на которой функция в данный момент выполняется. Выглядеть наша модификация будет следующим образом:

```
pool_worker(Fun, Portion, Index, Master) ->  
    io:format("~p~n", [node()]),  
    Result = Fun(Portion),  
    Master ! #task_result{index = Index, result = Result}.
```

При помощи данной модификации мы сможем убедиться, что у нас действительно создаются и выполняются процессы на узлах из пула узлов.

Теперь можно перейти непосредственно к проверке наших реализаций функций `map` и `reduce`. Первым делом давайте создадим три узла со следующими короткими именами: `node1`, `node2`, `node3`. Полные имена созданных узлов будут такими (с учетом, что имя компьютера автора — `stdstring`): `node1@stdstring`, `node2@stdstring`, `node3@stdstring`. Узлы с именами `node2` и `node3` мы будем использовать в качестве дополнительных узлов для пула узлов; узел с именем `node1` будет у нас основным — на нем мы будем производить проверку наших реализаций функций `map` и `reduce`. Как всегда, начнем проверку с проверки реализации функции `map`. Вызовом `parallel_map:pool_pmap(fun(X) -> X+0.1 end, [], 2, 3, ['node1@stdstring', 'node2@stdstring', 'node3@stdstring'])` мы проверяем работу функции `parallel_map:pool_pmap/5` в ситуации, когда список исходных данных пуст. Другими словами, мы проверяем работу первого варианта функции `parallel_map:pool_pmapImpl/4`. Этот вызов, как и ожидается, вернет нам пустой список. Очевидно, что при этом рабочие процессы на узлах из пула узлов создаваться не будут; мы это можем увидеть по отсутствию в выводе на узле `node1` имен узлов из пула

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

узлов. Вызов `parallel_map:pool_pmap(fun(X) -> X+0.1 end, [5, 13], 2, 3, ['node1@stdstring', 'node2@stdstring', 'node3@stdstring'])` проверяет работу функции `parallel_map:pool_pmap/5` в ситуации, когда размер списка исходных данных не больше размера порции данных. Другими словами, мы проверяем работу второго варианта функции `parallel_map:pool_pmap_impl/4`. Этот вызов вернет нам список элементов `[5.1, 13.1]`, что совпадает с ожидаемым результатом. Точно так же, как и в прошлый раз, рабочие процессы на узлах из пула узлов создаваться не должны; мы это можем увидеть по отсутствию в выводе на узле `node1` имен узлов из пула узлов. И, наконец, вызов `parallel_map:pool_pmap(fun(X) -> X+0.1 end, [1, 3, 4, 7, 8, 11, 2, 19], 2, 3, ['node1@stdstring', 'node2@stdstring', 'node3@stdstring'])` проверяет работу функции `parallel_map:pool_pmap/5` в ситуации, когда мы список исходных данных обрабатываем с использованием многозадачности. Другими словами, мы проверяем работу третьего варианта функции `parallel_map:pool_pmap_impl/4`. Этот вызов вернет нам список элементов `[1.1, 3.1, 4.1, 7.1, 8.1, 11.1, 2.1, 19.1]`, что совпадает с ожидаемым результатом. Исходный список содержит 8 элементов, а размер порции данных — 2 элемента, поэтому исходный список при обработке будет разбит на 4 порции. Так как рабочих процессов может быть не больше 3, а у нас 4 порции исходных данных, то при обработке исходных данных должны быть задействованы все узлы (и один из узлов дважды). И действительно, вывод на узле `node1` будет содержать имена всех трех узлов, причем имя одного из узлов встретится дважды.

Следующий шаг, который следовало бы сделать — это проверить работу реализации функции `reduce` (в нашем случае это будут функции `parallel_reduce:pool_reduce/6` и `parallel_reduce:pool_reduce_impl/5`). Однако давайте эту проверку оставим читателям, кому интересно ее провести (или поверьте на слово автору, что эти функции проверены и работают), и несколько углубимся в то, как работает пул узлов (реализованный в модуле `pool`). Для этого придется рассмотреть исходный код модуля `pool`, а также документацию к этому модулю.

В документации утверждается, что функции `pool:pspawn/3` и `pool:spawn_link/3` создают процесс на узле (из пула) с минимальной ожидаемой загрузкой узла; также, такой узел с минимальной загрузкой возвращает функция `pool:get_node/0`. Загрузка узла вычисляется при помощи функции `statistics(run_queue)`, которая возвращает количество процессов, готовых к выполнению. Однако если мы посмотрим на исходный код модуля `pool`, то мы найдем там следующую информацию, которая, что естественно, специфична для данной реализации пула узлов (а реализация может измениться в любой момент). Пул узлов хранит каждый узел вместе со значением загрузки узла в списке пар (кортежей), причем этот список отсортирован по значению загрузки узла по возрастанию. Это означает, что голова этого списка всегда содержит узел (точнее, пару узел—загрузка) с минимальной загрузкой. Значение загрузки каждого узла периодически пересчитывается (в данной реализации, каждые 2 секунды), после чего список пар перестраивается в соответствии с пересчитанными значениями загрузки узлов. При обращении к одной из функций `pool:pspawn/3`, `pool:spawn_link/3` или `pool:get_node/0` пара узел — значение загрузки этого узла из головы переносится в хвост списка пар, при этом значение загрузки узла увеличивается на единицу.

Для нас это означает следующее: если у нас пул узлов содержит `N` узлов с разной загрузкой, и мы захотим достаточно быстро выполнить при помощи этого пула `N` заданий, то задания будут распределены между всеми узлами. Давайте это продемонстрируем. Для этого мы создадим модуль `load_sample` с тремя экспортируемыми функциями `load_sample:create_load/1`, `load_sample:ping/0` и `load_sample:pong/0`:

```
create_load(0) -> true;
create_load(Count) ->
```

Как зовут тебя, узел?

Узел — это именованный экземпляр среди выполнения Erlang. Для создания узла при запуске среди выполнения Erlang необходимо задать его имя, используя один из ключей `-sname` или `-name`. Ключ `-sname` задает короткое имя узла; ключ

`-name` задает длинное имя узла. При работе с узлами следует помнить следующее правило: необходимо всегда создавать узлы с одним типом имен. Узлы с разными типами имен не смогут взаимодействовать друг с другом.

```
PingPID = spawn(load_sample, ping, []),
PongPID = spawn(load_sample, pong, []),
PingPID ! {ping, PongPID},
create_load(Count-1).
ping() ->
receive
    {ping, PID} -> PID ! {pong, self()}, ping()
end.
pong() ->
receive
    {pong, PID} -> PID ! {ping, self()}, pong()
end.
```

Видно, что для создания загрузки мы создаем `N` процессов, отправляющих сообщения друг другу. Очевидно, что при создании этих `N` пар процессов вызов `statistics(run_queue)` вернет нам значение, очень близкое к `N`.

А теперь посмотрим, как работает пул узлов при их неравномерной загрузке. Как и раньше, создадим 3 узла: `node1`, `node2` и `node3`. На узле `node3` мы создадим нагрузку вызовом `load_sample:create_load(10000)`. То, что нагрузка создалась, можно проверить при помощи вызова функции `statistics(run_queue)` на узле `node3`. После этого на узле `node1` создаем пул узлов вызовом `pool_helper:start_pool(test, ['node1@stdstring', 'node2@stdstring', 'node3@stdstring'])`. Сейчас у нас есть пул узлов, причем один из узлов нагружен сильнее, чем два других. На узле `node1` выполним следующее выражение: `lists:foreach(fun(_) -> io:format("~-p~n", [pool:get_node()]) end, lists:seq(1, 9))`. Это выражение выводит на экран имя наименее загруженного узла 9 раз. Так как после обращения к функции `pool:get_node/0` головной элемент переносится в конец списка узлов, а наш вызов происходит практически моментально, то мы видим, что все наши 3 узла выводятся на экран одинаковое количество раз. А сейчас на узле `node1` давайте выполним следующее выражение: `lists:foreach(fun(_) -> io:format("~-p~n", [pool:get_node()]), timer:sleep(4000) end, lists:seq(1, 9))`. Это выражение отличается от предыдущего только тем, что после каждого вызова `pool:get_node/0` мы ждем достаточно большое время (4 секунды), чтобы произошел пересчет статистики по загрузке узлов на пуле узлов. Результат выполнения этого выражения будет отличаться от предыдущего: на экране мы увидим только имена узлов `node1` и `node2`. Из данной демонстрации можно сделать следующий вывод: пул узлов из модуля `pool` не очень хорошо подходит в ситуации, когда нам необходимо в достаточно короткий промежуток времени создать много задач на обработку при неравномерной загрузке узлов пула.

На нашем уроке мы закончили с очередной многозадачной реализацией функций `map` и `reduce` (в этот раз на основе пула узлов) и проверили их работоспособность. Также мы разобрались с ограничениями реализации пула узлов из библиотеки языка Erlang (модуль `pool`). Автор в прошлый раз обещал, что мы подведем итоги решения задачи о реализации многозадачных версий функций `map` и `reduce`. К сожалению, из-за ограниченности места журнала автор сделал этого не смог (хотя желание, чтобы весь журнал целиком посвятили одному ему, у него есть). Обещанные итоги мы подведем в следующий раз. [LXF](#)