Erlang: Много-

Андрей Ушаков завершает серию уроков по Erlang рассмотрением проблемы распараллеливания задач.

Наш эксперт

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

О важности многозадачности в современных приложениях сказано уже так много, что смысла повторяться, пожалуй, нет. Многозадачность применима не всегда: в некоторых случаях сам алгоритм не поддерживает распараллеливание, в некоторых случаях побочные затраты на поддержку многозадачности (например, время переключения контекста в многопоточных приложениях) больше, чем получаемая от нее выгода. Если же многозадачность возможна и от нее есть выгода, то существует проблема разделяемого состояния: если с совместно используемыми объектами работать неграмотно, то их состояние может «испортиться».

Конечно, при аккуратной (и правильной) работе с разделяемым состоянием проблем не возникает, но увеличивается сложность кода по сравнению с однозадачной версией.

Большинства этих проблем в Erlang просто нет. Конечно, если алгоритм не поддерживает распараллеливание, то исправить это можно только выбором другого алгоритма. И многозадачная версия практически всегда будет больше и сложнее, чем однозадачная. С другой стороны, создание и уничтожение процессов (это процессы самого языка Erlang, а не ОС) – очень быстрая операция, памяти и ресурсов процессы потребляют мало, создавать их можно много (максимальное количество процессов по умолчанию – 32768, но, используя флаг **+P**, максимальное количество можно довести до 134217727). И, что является большим плюсом, процессы в Erlang полностью независимы и не содержат совместно используемых объектов. Если двум процессам необходимо взаимодействие, это взаимодействие выглядит следующим образом: один процесс посылает сообщение, а другой процесс это сообщение принимает. Посылка и обработка сообщений происходят очень быстро. Поэтому, если перед вами встает вопрос, использовать ли многозадачность в Erlang – ответ всегда один: если алгоритм позволяет это, то да.

> «Посылка и обработка сообщений происходят очень быстро.»

Рассмотрим примитивы, которые мы будем применять для построения многозадачной программы.

» **Pid = spawn(Fun)** Создает новый процесс, который будет выполнять функцию Fun и возвращать идентификатор этого процесса (process ID). Идентификатор процесса PID – один из типов данных в Erlang; он используется для взаимодействия процессов друг с другом (см. далее).

» **Pid! Message** Посылка сообщения **Message** процессу с идентификатором **Pid**. Сообщение посылается асинхронно, и отправитель не ждет, когда получатель получит сообщение. Если необходимо, чтобы процесс-получатель мог послать отправителю что-нибудь обратно, то процесс-получатель должен знать идентификатор отправителя. Самый удобный способ сделать это – отослать **Pid** отправителя в сообщении, например, так: **ReseiverPID! {Message, self()}** (функция self() позволяет процессу узнать свой **Pid**).

» **receive ... end** Получение и обработка процессом сообщений из своей очереди.

Конструкция **receive** выглядит следующим образом:

```
receive
        Pattern1 [when Guard1] -> Expression1;
        Pattern2 [when Guard2] -> Expression2;
        …
[after Time -> AfterExpression]
end
```

Когда сообщение доставляется процессу, оно помещается в очередь сообщений данного процесса. Когда выполнение процесса попадает в конструкцию **receive,** процесс просматривает свою очередь сообщений и последовательно проверяет каждое сообщение на соответствие одному из шаблонов **PatternN** (и на соответствие guard-выражению **GuardN**, если такое есть). Как только соответствие будет установлено, на этом поиск оканчивается, и конструкция **receive** возвратит значение – соответствующее выражение **ExpressionN**. Если для данного сообщения ни одного соответствия не найдено, то данное сообщение откладывается обратно в очередь (оно будет просмотрено в следующей конструкции **receive**), и для просмотра берется следующее из очереди. Если все сообщения из очереди просмотрены и ни для одного из них не найдено соответствия, то, что будет происходить дальше, зависит от наличия секции **after**. Если секции **after** нет, то конструкция **receive** будет ждать, пока в очереди не появится новое сообщение, если же секция **after** есть, то через **Time** миллисекунд конструкция **receive** возвратит значение выражения **AfterExpression**.

Вот в принципе и все необходимое (для начала) знание, чтобы создавать многозадачные приложения на Erlang.

Следует сказать пару слов о создании распределенных приложений. При создании распределенных приложений используются те же самые примитивы, что и при создании многозадачных приложений. Новым тут является только понятие узла – экземпляра виртуальной машины, запущенной локально либо удаленно (для того, чтобы экземпляр виртуальной машины считался узлом, при запуске необходимо указать его имя, например, при помощи ключей **-sname** либо **-name**). После создания узла **Node** на нем можно создать новый процесс при помощи функции **spawn**: **Pid = spawn(Node, Fun)**. После этого вся работа с процессом на удаленном узле строится точно так же, как и с локальными процессами: используется идентификатор созданного процесса **Pid** для взаимодействия с ним.

Пример

Для демонстрации всего рассказанного выше (и чтобы не заскучать), давайте создадим простую распределенную систему – а точнее, создадим обычный и распределенный вариант одной и той же задачи и сравним их. Возьмем в качестве примера задачу поиска пароля по хэшу MD5 и решим ее обычным перебором. Для простоты ограничим набор символов, используемых в пароле, только цифровыми символами (0 ... 9).

много задач

Для начала введем несколько вспомогательных функций, которые будут использоваться в обоих вариантах. Во время работы мы захотим оценить время, затрачиваемое тем или иным вариантом. Текущее время в формате **{MegaSeconds, Seconds, MicroSeconds}** мы можем получить, вызвав функцию **now()**. Вспомогательный метод **calc_work_time/2** позволит вычислить количество секунд между двумя измерениями:

```
calc_work_time(Now1, Now2) ->
        {MegaSecs1, Secs1, MicroSecs1} = Now1,
        {MegaSecs2, Secs2, MicroSecs2} = Now2,
        (MegaSecs2-MegaSecs1)*1000000+(Secs2-
Secs1)+(MicroSecs2-MicroSecs1)*1.0e-6.
```

Для поиска нам нужно уметь генерировать очередную строку, после чего вычислять для нее хэш MD5 и сравнивать с исходным. Для генерации мы каждой строке сопоставим целочисленный номер так, чтобы номера и строки располагались в одном порядке: строке “0” будет соответствовать номер 0, строке “1” – номер 1, …, строке “00” – номер 10, строке “01” – номер 11, и т. д. Вспомогательные методы **generate_string_by_number/2**, **generate_string_by_number/4 и correct_number/3** реализуют данную функциональность. С точки зрения внешнего (относительно этих методов) кода, метод **generate_string_by_number/2** является интерфейсом к данной функциональности.

```
generate_string_by_number(0, Alphabet) -> [lists:nth(1,
Alphabet)];
generate_string_by_number(Number, Alphabet) ->
    {CorrectNumber, StringLength} = correct_number(Number,
length(Alphabet), 1),
    generate_string_by_number(CorrectNumber, StringLength,
Alphabet, []).
generate_string_by_number(0, StringLength, [First|_],
GeneratedPart) ->
    lists:duplicate(StringLength-length(GeneratedPart), First) ++
GeneratedPart;
generate_string_by_number(Rest, StringLength, Alphabet,
GeneratedPart) ->
    Index = (Rest rem length(Alphabet)),
    NewRest = Rest div length(Alphabet),
    generate_string_by_number(NewRest, StringLength,
Alphabet, [lists:nth(Index+1, Alphabet)]++GeneratedPart).

correct_number(Number, AlphabetCount, CheckStringLength) ->
    StringCountInRange = trunc(math:pow(AlphabetCount,
CheckStringLength)),
    if
        Number < StringCountInRange -> {Number,
CheckStringLength};
        true -> correct_number(Number-StringCountInRange,
AlphabetCount, CheckStringLength+1)
    end.
```

И что еще нам нужно из вспомогательных методов – это метод, позволяющий получить максимальный целочисленный номер для заданной максимальной длины строки. Это делает метод **generate_number_by_string_length/2**:

```
generate_number_by_string_length(MaxStringLength,
AlphabetCount) ->
        (AlphabetCount*(1-trunc(math:pow(AlphabetCount,
MaxStringLength))) div (1-AlphabetCount))-1.
```

Со вспомогательными функциями все, и теперь можно перейти к основным функциям. Рассмотрение мы начнем со случая простого последовательного поиска. В этом случае нам понадобятся всего два метода: для запуска поиска (**search/0**) и для просмотра очередного варианта (**search/4**). Обратите внимание, что поиск не содержит явного цикла для просмотра вариантов: вместо этого метод просмотра очередного варианта (**search/4**) вызывает рекурсивно сам себя для просмотра следующего варианта. А благодаря тому, что в этом методе рекурсия хвостовая, этот метод разворачивается в цикл. Очень элегантно, не правда ли?

```
search(SourceMD5, _, CurrentNumber, MaxNumber)
when CurrentNumber > MaxNumber -> {cant_find, SourceMD5};
search(SourceMD5, Alphabet, CurrentNumber, MaxNumber) ->
    GeneratedString = generate_string_by_
number(CurrentNumber, Alphabet),
    GeneratedStringMD5 = erlang:md5(GeneratedString),
    if
        SourceMD5 == GeneratedStringMD5 -> GeneratedString;
        true -> search(SourceMD5, Alphabet, CurrentNumber+1,
MaxNumber)
    end.
```

»

История Erlang

- **1982–1985** Эксперименты в Ericsson Computer Science Laboratory по программированию в области телекоммуникаций на более чем 20 языках. Вывод: нужен высокоуровневый символический язык для достижения высокой производительности труда (наподобие Lisp, Prolog, Parlog и т. д.).
- **1985–1986** Эксперименты с Lisp, Prolog, Parlog и т. д. Вывод: язык должен содержать примитивы для поддержки параллелизма и восстановления после сбоев. Он должен также поддерживать детализацию параллелизма, чтобы один асинхронный процесс телефонии соответствовал одному процессу в языке. Т.о., было принято решение разработать свой собственный язык, основываясь на Lisp, Prolog и Parlog, но с поддержкой параллелизма и восстановления после сбоев на уровне языка.
- **1987** Первые эксперименты с Erlang.
- **1988** Фаза 1: Прототип показан внешним пользователям. Erlang вышел за пределы лаборатории.
- **1989** Фаза 2: Воссоздана 1/10 полной MD-110 системы. Итог: создание программ более чем в 10 раз эффективнее, чем в PLEX.
- **1990** Erlang представлен на ISS’90, что привело к появлению новых пользователей, например, Bellcore.
- **1991** Версия Erlang выпущена для пользователей. Erlang представлен на Telecom’91. Появилась новая функциональность, такая как ASN/1 – компилятор, графический интерфейс и т. д.
- **1992** Появление большого числа новых пользователей Erlang. Erlang портирован на большинство платформ: VxWorks, PC, Macintosh и т. д.
- **1993** В Erlang добавлена поддержка распределенных вычислений. Принято решение продавать реализацию Erlang внешним организациям.
- **1998** Реализация Erlang становится open-source.
- **2006** Поддержка симметричной многопроцессорности встроена в исполняющую среду и виртуальную машину Erlang.

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

```
search() ->
    Alphabet = [$0, $1, $2, $3, $4, $5, $6, $7, $8, $9],
    Source = "01234321",
    SourceMD5 = erlang:md5(Source),
    Now1 = now(),
    Result = search(SourceMD5, Alphabet, 0, generate_number_
by_string_length(10, length(Alphabet))),
    Now2 = now(),
    {calc_work_time(Now1, Now2), Result}.
```

Осталось только привести объявления модуля и экспортируемых функций:

```
-module(md5_sequential_search).
-export([search/0]).
```

Вот и все с последовательным поиском. Запускаем среду выполнения Erlang, в консоли Erlang запускаем сначала компиляцию **c(md5_sequential_search).**, а потом и выполнение нашей программы **md5_sequential_search:search().** При запуске на моей машине (ноутбук Acer Aspire 7520G: процессор AMD Turion64×2 TL-58 $5,9 ГГц, 2ГБ ОЗУ), приложение находит искомую строку **"01234321"** по ее хэшу MD5 за 158,234 секунд.

Перейдем теперь к распределенному варианту. В нем мы также используем вспомогательные функции **calc_work_time/2** и **generate_string_by_number/2**. Но, в отличие от обычного варианта, мы введем несколько ролей, которые будут соответствовать разным компонентам, выполняющимися в разных Erlang-процессах. Это следующие роли: инициатор, координатор, обработчики.

Инициатор создает необходимое количество обработчиков (каждый в своем процессе), после чего создает координатор (тоже в своем процессе) и передает ему список идентификаторов процессов обработчиков. Координатор проходится по списку обработчиков и каждому из них посылает сообщение (**{are_you_ready, CurrentPID, SourceMD5, Alphabet}**) с требованием подтвердить свою готовность. Обработчик, получая данное сообщение, отправляет координатору сообщение с подтверждением готовности (**{ready_master, HandlerPID}**).

Координатор, после получения подтверждения о готовности, посылает сообщение с заданием на поиск хэша MD5 для строк, чей номер лежит в диапазоне [FromNumber, ToNumber] (**{search, FromNumber, ToNumber}**).

Обработчик при получении данного сообщения начинает поиск: если для какой-либо строки будет найдено соответствие с искомым MD5-хэшем, то координатору будет послано сообщение о том, что строка найдена (**{found, GeneratedString}**); если же обработчик в заданном ему диапазоне ничего не найдет, то будет послано соответствующее сообщение координатору (**{not_found, HandlerPID}**). Если координатор получает сообщение, что искомая строка найдена, он это сообщение пересылает инициатору и останавливает свою работу и работу обработчиков. Получив от обработчика сообщение, что в заданном диапазоне ничего не найдено, координатор посылает обработчику новое задание с новым диапазоном. Если обработчиками просмотрено все множество строк (из нашего ограничения на длину и набор символов) и не найдено ни одной строки, MD5-хэш которой совпадает с искомым, то инициатору будет послано соответствующее сообщение (**{not_found}**).

Вот и все о разных компонентах и их взаимодействии.

Давайте теперь посмотрим, как это все реализовано – и начнем с обработчиков:

```
start_search_handler() ->
    receive
        {are_you_ready, MasterPID, SourceMD5, Alphabet} ->
            MasterPID ! {ready_master, self()},
            search_handler(MasterPID, SourceMD5, Alphabet)
    end.
search_handler(MasterPID, SourceMD5, Alphabet) ->
    receive
        {search, FromNumber, ToNumber} ->
            portion_search(MasterPID, SourceMD5,
FromNumber, ToNumber, Alphabet),
            search_handler(MasterPID, SourceMD5, Alphabet)
    end.
portion_search(MasterPID, _, ToNumber, ToNumber, _) ->
MasterPID!{not_found, self()};
portion_search(MasterPID, SourceMD5, FromNumber, ToNumber,
Alphabet) ->
    GeneratedString = generate_string_by_
number(FromNumber, Alphabet),
    GeneratedStringMD5 = erlang:md5(GeneratedString),
    if
        SourceMD5 == GeneratedStringMD5 ->
MasterPID!{found, GeneratedString};
        true -> portion_search(MasterPID, SourceMD5,
FromNumber+1, ToNumber, Alphabet)
    end.
```

Метод **start_search_handler/0** используется для запуска обработчика, метод **search_handler/3** – обработчик сообщений от координатора, в методе **portion_search/5** происходит поиск хэша MD5 для строк, чей номер лежит в диапазоне [FromNumber, ToNumber].

Теперь перейдем к координатору:

```
main_search_handler(MasterPID, SourceMD5, Alphabet,
PortionSize, MaxNumber, HandlerPIDList) ->
    process_flag(trap_exit, true),
    CurrentPID = self(),
    lists:foreach(fun(HandlerPID) ->
            link(HandlerPID),
            HandlerPID ! {are_you_ready, CurrentPID,
SourceMD5, Alphabet}
        end, HandlerPIDList),
    main_search_handler(MasterPID, SourceMD5, Alphabet, 0,
PortionSize, MaxNumber, 0).
main_search_handler(MasterPID, _, _, _, _, MaxNumber,
ResponseCount)
when ResponseCount >= MaxNumber ->
    MasterPID!{not_found},
    exit(stop_work);
main_search_handler(MasterPID, SourceMD5, Alphabet,
CurrentNumber, PortionSize, MaxNumber, ResponseCount)
when CurrentNumber >= MaxNumber ->
    receive
        {stop} -> exit(stop_work);
        {found, GeneratedString} -> MasterPID!{found,
GeneratedString},
                                    exit(stop_work);
        {not_found, _} -> main_search_handler(MasterPID,
SourceMD5, Alphabet, MaxNumber, PortionSize, MaxNumber,
ResponseCount+PortionSize)
    end;
main_search_handler(MasterPID, SourceMD5, Alphabet,
CurrentNumber, PortionSize, MaxNumber, ResponseCount) ->
    receive
        {stop} -> exit(stop_work);
        {ready_master, HandlerPID} ->
            ToNumber = min(CurrentNumber+PortionSize,
MaxNumber+1),
```

```
                HandlerPID!{search, CurrentNumber, ToNumber},
                main_search_handler(MasterPID, SourceMD5,
Alphabet, ToNumber, PortionSize, MaxNumber, ResponseCount);
            {found, GeneratedString} -> MasterPID!{found,
GeneratedString},
                                        exit(stop_work);
            {not_found, HandlerPID} ->
                ToNumber = min(CurrentNumber+PortionSize,
MaxNumber+1),
                HandlerPID!{search, CurrentNumber, ToNumber},
                main_search_handler(MasterPID,
SourceMD5, Alphabet, ToNumber, PortionSize, MaxNumber,
ResponseCount+PortionSize)
        end.
```

Метод **main_search_handler/6** используется для запуска координатора и отсылки сообщений обработчикам с требованием подтвердить свою готовность; метод **main_search_handler/7** используется для взаимодействия с обработчиками.

И, наконец, инициатор. На самом деле у нас два инициатора: один (метод **start_search/0**) – для запуска простого многозадачного поиска на данном узле (экземпляре виртуальной машины), другой (метод **start_distributed_search/0**) – для запуска распределенного поиска на разных узлах (на одном или разных компьютерах). Самая большая разница между ними в том, как (и где) создаются обработчики (координатор создается на том же узле, что и инициатор). При простом многозадачном поиске обработчики создаются вызовом **spawn/1** (версия **spawn**, в которой не указывается узел). При распределенном поиске обработчики создаются вызовом **spawn/2** (версия **spawn**, в которой указывается узел, где создается процесс). В нашем модельном инициаторе список узлов, на которых будут создаваться процессы, задается прямо в теле метода; в реальном же приложении список узлов будет, скорее всего, браться из конфигурационного файла.

> «Язык Erlang позволяет создавать любое ПО любой сложности.»

```
start_distributed_search() ->
    Alphabet = [$0, $1, $2, $3, $4, $5, $6, $7, $8, $9],
    Source = "01234321",
    SourceMD5 = erlang:md5(Source),
    MaxNumber = generate_number_by_string_length(10,
length(Alphabet)),
    ProcessCount = 4,
    PortionSize = 100000,
    NodeList = ['node1@beerzone2', 'node2@beerzone2'],
    {HandlerPIDList, _} = lists:mapfoldl(fun(_, CurrentNodeList) ->
            [NodeHead | NodeOther] = CurrentNodeList,
            HandlerPID = spawn(NodeHead, fun() -> start_search_
handler() end),
            {HandlerPID, NodeOther++[NodeHead]}
        end, NodeList, lists:seq(1, ProcessCount)),
    Now1 = now(),
    CurrentPID = self(),
    MainHandlerPID = spawn(fun() -> main_search_
handler(CurrentPID, SourceMD5, Alphabet, PortionSize,
MaxNumber, HandlerPIDList) end),
    Result = process_response(),
    MainHandlerPID!{stop},
    Now2 = now(),
    {calc_work_time(Now1, Now2), Result}.
start_search() ->
    Alphabet = [$0, $1, $2, $3, $4, $5, $6, $7, $8, $9],
    Source = "01234321",
    SourceMD5 = erlang:md5(Source),
    MaxNumber = generate_number_by_string_length(10,
length(Alphabet)),
    ProcessCount = 2,
    PortionSize = 100000,
    HandlerPIDList = [spawn(fun() -> start_search_handler() end)
|| _ <- lists:seq(1, ProcessCount)],
    Now1 = now(),
    CurrentPID = self(),
    MainHandlerPID = spawn(fun() -> main_search_
handler(CurrentPID, SourceMD5, Alphabet, PortionSize,
MaxNumber, HandlerPIDList) end),
    Result = process_response(),
    MainHandlerPID!{stop},
    Now2 = now(),
    {calc_work_time(Now1, Now2), Result}.
process_response() ->
    receive
        Response -> Response
    end.
```

Осталось только привести объявления модуля и экспортируемых функций:

```
-module(md5_distributed_search).
-export([start_search/0, start_distributed_search/0]).
```

Чтобы запустить распределенный поиск, необходимо сделать следующее. Предположим, что имя компьютера – beerzone2 (как у меня). Мы хотим запустить обработчики на узлах node1@beerzone2, node2@beerzone2. В теле программы, в методе **start_distributed_search/0** устанавливаем список узлов **NodeList** в **['node1@beerzone2', 'node2@beerzone2']**. После этого мы создаем три экземпляра консоли: в двух мы запускаем виртуальную машину с ключами **-sname node1** и **-sname node2**, а в третьей – виртуальную машину с ключом **-sname main** (очень важно, чтобы все взаимодействующие узлы в распределенной системе имели имена одного типа: либо короткие, либо длинные). В главной консоли (запущенной с ключом **-sname main**) запускаю сначала компиляцию **c(md5_distributed_search).**, а потом и выполнение нашей программы **md5_distributed_search:start_distributed_search().** При запуске на моей машине (ноутбук Acer Aspire 7520G: процессор AMD Turion64 x2 TL-58 1,9 ГГц, 2 ГБ ОЗУ), приложение находит искомую строку **"01234321"** по ее хэшу MD5 за 76,984 секунды.

В качестве заключения

Итак, написать распределенную систему для решения любой задачи легко. Язык Erlang позволяет создавать любое серверное и распределенное ПО любой сложности, по производительности не уступающее такому же ПО, написанному на других языках, а по качеству кода и надежности сильно превосходящее их. LXF

Полезные сайты и книги

- http://www.erlang.org/ – главный сайт (с документацией и исходным кодом среды).
- http://www.trapexit.org/ – сайт Erlang-сообщества (форум, вики, решения, учебные пособия, справочные материалы).
- http://erlanger.ru/ – сайт русского Erlang-сообщества.
- http://groups.google.com/group/erlang-russian – русское Erlang-сообщество на Google.
- http://www.tryerlang.org/ – онлайн-интерпретатор Erlang.

- Martin Logan, Eric Merritt, and Richard Carlsson "*Erlang and OTP in Action*".
- Francesco Cesarini, Simon Thompson "*Erlang Programming A Concurrent Approach to Software Development*".
- Joe Armstrong "*Programming Erlang: Software for a Concurrent World*".