

Erlang: о пользе

Андрей Ушаков проходится по функциям, сильно облегчающим жизнь многозадачным и распределенным системам.



Наш
эксперт

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

В нескольких последних номерах мы говорили о принципах построения многозадачных, отказоустойчивых и распределенных систем. И мы, естественно, затрагивали тему о библиотечных функциях, упрощающих нам жизнь. Данная статья посвящена как раз этой теме: мы рассмотрим большую часть библиотечных функций, о которых пока не говорилось.

Вернемся немного назад и обсудим хранение состояния между вызовами функций. Одно из основных положений функционального программирования – то, что результат выполнения любой функции зависит только от ее аргументов. Передав одни и те же аргументы несколько раз, мы должны получить один и тот же результат. А значит, функция не должна помнить какое-либо состояние между ее вызовами. Что делать, если некоторое состояние между вызовами функции все же необходимо хранить? Ответ очевиден: переложить ответственность по хранению состояния на вызывающую сторону и передавать состояние в функцию как один из аргументов. Основной строительной единицей многозадачных приложений в языке Erlang являются процессы. «Сердце» процесса Erlang – обычно цикл обработки сообщений, представляющий собой функцию – обработчик сообщений. Эта функция вызывает сама себя (это единственный способ в языке Erlang создать конечный или бесконечный цикл) при помощи хвостовой рекурсии. Соответственно, в таком случае цикл – обработчик сообщений будет и вызывающей, и вызываемой сторонами, и за передачу состояния будет ответственна функция-обработчик. Естественно, что при создании процесса и первом вызове функции-обработчика сообщений (произведенного извне), ответственность за передачу начального состояния лежит на стороне, создающей процесс.

Описанный подход к хранению состояния является строго функциональным. Среда выполнения Erlang содержит и другой подход к хранению состояния: это использование памяти, принадлежащей процессу. С каждым процессом связан некий объем памяти в виде изменяемого словаря; доступ к этой памяти имеет только код, выполняемый процессом. Если два разных процесса выполняют один и тот же код, который обращается к памяти, связанной с процессом, то этот код будет обращаться к разным областям памяти, в зависимости от того, какой из процессов его выполняет. Для работы с такой памятью в языке Erlang определен целый ряд функций (а точнее, BIF). Сохранить некоторое значение **Value** и ассоциированный с ним ключ **Key** (при помощи которо-

го будет осуществляться доступ к этому значению в дальнейшем) можно при помощи функции **put(Key, Value)**. Если с данным ключом уже было ассоциировано какое-либо значение, оно будет возвращаемым значением функции **put/2** (иначе возвращаемым значением будет атом **undefined**). Для получения значения **Value** по ключу **Key** служит функция **get(Key)**, которая возвращает либо значение **Value**, ассоциированное с ключом **Key**, либо атом **undefined**. Мы можем получить все данные сразу, в виде пар ключ–значение: для этого служит функция **get/0**. Помимо этого, можно получить список всех ключей, ассоциированных со значением **Value**, при помощи функции **get_keys(Value)**. И, наконец, чтобы удалить некоторое значение по ключу (и ассоциированный с ним ключ), используется функция **erase(Key)**; чтобы полностью очистить память, связанную с процессом – функция **erase/0**.

Следует сразу сказать, что использование памяти, связанной с процессом и этим набором функций, имеет два больших минуса по сравнению с «классическим» функциональным подходом. Во-первых, можно легко написать функцию, возвращающее значение которой при одних и тех же аргументах будет различаться, что нарушает положение функционального программирования о том, что результат выполнения функции должен зависеть только от ее аргументов. Конечно, есть ряд функций, которые это положение нарушают, но все эти функции являются интерфейсом к внешнему миру – например, функция **now/0**, возвращающая текущее время. Во-вторых, использование памяти, связанной с процессом, создает неявную связь как между функцией и некоторым состоянием, так и между несколькими функциями. Это увеличивает связность и сложность приложения. В общем, использовать память, связанную с процессом, или нет – решать, конечно, вам.

Раз уж мы затронули функции, определенные в модуле **erlang** (а как мы помним, BIF также определены в этом модуле), давайте кратко рассмотрим те функции, о которых мы еще не говорили (естественно, это касается только функций, интересных с точки зрения создания многозадачных и распределенных приложений).

Начнем с разговора о лидере группы процессов. Все процессы в языке Erlang принадлежат той или иной группе, и в каждой группе есть процесс, называемый лидером этой группы. Весь ввод/вывод внутри группы направляется из/в лидера группы. Когда один процесс создает другой процесс, то у нового процесса будет тот же самый лидер группы (и та же самая группа), что и у родительского процесса. Для получения идентификатора лидера группы, которой принадлежит текущий процесс, следует использовать функцию **group_leader/0**. Чтобы для группы, которой принадлежит процесс с идентификатором **Pid**, задать нового лидера, следует использовать функцию **group_leader(GroupLeaderPid, Pid)**, где **GroupLeaderPid** – идентификатор процесса, который станет новым лидером группы.

Отметим также, что для управления группами процессов следует использовать функции из модулей **pg** или **pg2**. Модуль **erlang**, помимо вышеупомянутых функций, содержит также функции, реализующие функциональность таймера: **erlang:start_timer/3**, **erlang:read_timer/1**, **erlang:cancel_timer/1** и **erlang:send_after/3** (о них мы поговорим подробнее в одном из практикумов).

Функции, использующие состояние

Примером функций, использующих состояние, сохраненное между их вызовами вызывающей стороной, являются функции для расчета значения хэша MD5 (определенные в модуле **erlang**). Функция **erlang:md5_init()** инициализирует состояние, называемое контекстом **Context**, и возвращает его.

Функция **erlang:md5_update(Context, Data)** принимает в качестве аргументов старое состояние **Context** и очередную порцию данных **Data**, и возвращает обновленное состояние. Функция **erlang:md5_final(Context)** принимает состояние **Context** и возвращает значение хэша MD5.

библиотек

Следующий модуль, которым мы займемся – небольшой модуль **lib**, содержащий разнообразные полезные функции. Из этих функций для нас интересны (в ключе нашей статьи) в первую очередь две: **lib:flush_receive/0** и **lib:sendw/2** (также этот модуль содержит функцию **lib:send/2**, полностью идентичную функции **erlang:send/2**). Функция **lib:flush_receive/0** служит для очистки очереди сообщений процесса от необработанных сообщений. Мы помним, что каждый процесс содержит собственную очередь сообщений. Когда мы посыпаем сообщение некоторому процессу, оно помещается в очередь сообщений этого процесса.

Выражение **receive** извлекает первое подходящее сообщение из очереди сообщений, оставляя остальные сообщения в очереди. Поэтому вполне возможно возникновение ситуации, когда количество сообщений в очереди какого-либо процесса постоянно увеличивается, что, в конечном итоге, приведет к замедлению вычисления (выполнения) выражения **receive** и переполнению очереди сообщений. Такая ситуация возможна, если процесс получает сообщения, не подходящие ни для одного выражения **receive**, либо если сообщения приходят быстрее, чем процесс их обрабатывает. Первого случая можно избежать, если в выражении **receive** всегда добавлять в конец вариант, обрабатывающий все сообщения. Однако этот подход не будет работать, если вид выражения **receive** (варианты обрабатываемых сообщений) может изменяться (например, в зависимости от некоторого состояния). В этом случае (а также в случае переполнения очереди сообщений) можно воспользоваться функцией **lib:flush_receive/0**. Чтобы узнать, сколько же сообщений содержится в очереди сообщений процесса, необходимо использовать BIF **process_info(Pid, [message_queue_len])**, где **Pid** – идентификатор процесса, размер очереди сообщений которого мы хотим узнать.

Мы также можем получить список всех необработанных сообщений в очереди сообщений процесса, вызвав **process_info(Pid, [messages])**. Функция **lib:sendw(To, Msg)** отсылает сообщение процессу **To** и ожидает ответного сообщения, которое она и возвращает. Эта функция позволяет упростить написание кода для синхронизации взаимодействия между процессами, когда требуется послать сообщение процессу, дождаться от него ответа и продолжить выполнение задачи.

У этой функции есть два недостатка, о которых необходимо помнить при ее использовании. Во-первых, она ждет ответа от другого процесса бесконечно долго; если ответ от другого процесса по каким-либо причинам не придет ожидающему процессу, то ожидающий процесс будет заблокирован навсегда. Во-вторых, эта функция обрабатывает первое сообщение от любого процесса, которое придет после посылки сообщения какому-то определенному процессу. Вполне возможна ситуация, что мы посыпаем сообщение одному процессу, первым приходит сообщение от другого процесса, и именно его нам функция **lib:sendw/2** и возвратит.

Теперь поговорим про один из самых, пожалуй, полезных модулей для создания многозадачных приложений: это модуль **rpc**.

Функции из этого модуля упрощают выполнение таких операций, как вызов какой-либо функции (экспортируемой из некоторого модуля) на удаленной стороне, в отдельном процессе на некотором узле (или группе узлов). При создании любого узла на нем автоматически запускается **rpc**-сервер, с которым и взаимодействуют функции из этого модуля.

Начнем с функций, позволяющих выполнить некоторую функцию на удаленной стороне и вернуть результат выполнения этой функции: **rpc:call(Node, Module, Func, Args)** и **rpc:call(Node, Module, Func, Args, Timeout)**. Обе эти функции на узле **Node** в отдельном процессе выполняют функцию **apply(Module, Func, Args)**, т. е. вызывают функцию **Module:Func** с аргументами **Args** и возвращают результат этого вызова. Если вызов функции был успешен, то результат вызова возвращается напрямую; если же во время вызова возникла ошибка (исключение времени выполнения), то эти функции в качестве результата вернут кортеж **{badrpc, Reason}**, где **Reason** – причина возникновения ошибки. Обе эти функции синхронные – то есть вызывающий процесс блокируется до получения результата. Единственное отличие между этими двумя функциями – первая функция ожидает результата бесконечно

долго, тогда как вторая – в течение отрезка времени, заданного параметром **Timeout**.

Функции **rpc:block_call/4** и **rpc:block_call/5** аналогичны двум предыдущим функциям, за одним исключением:

в отличие от предыдущих, они не создают новый процесс на узле **Node**, а выполняют вызов в процессе **rpc**-сервера. Это приводит к тому, что **rpc**-сервер на узле **Node** блокируется до окончания выполнения исходного вызова. Получается, что пока на узле **Node** вызов функции **rpc:block_call/4** или **rpc:block_call/5** не завершится, никакой другой вызов какой-либо функции из модуля **rpc** на узле **Node** выполняться не начнет. Однако есть некоторые тонкости, связанные с этими функциями. Предположим, что на узле **Node** мы инициировали выполнение функции **rpc:call/4** (или любой другой функции, отличной от функций **rpc:block_call/4** и **rpc:block_call/5**). После этого из другого процесса мы инициировали выполнение функции **rpc:block_call/4** (или **rpc:block_call/5**) опять же на узле **Node**, что приведет к блокированию **rpc**-сервера на узле **Node**. Но при этом вызов функции **rpc:call/4** продолжит выполняться на узле **Node**, т. к. он выполняется в своем процессе (а не в процессе **rpc**-сервера). Далее предположим, что вызов функции **rpc:call/4** завершается раньше вызова функции **rpc:block_call/4**.

По завершении вызова функции **rpc:call/4** вызывающая сторона должна получить результат работы этой функции; однако результат передается не напрямую, а через **rpc**-сервер, что приведет к тому, что результат вызова функции **rpc:call/4** будет передан вызывающей стороне только после завершения вызова функции **rpc:block_call/4**. Ну и никто не запрещает нам создавать процессы на узле **Node** при помощи одной из функций семейств **spawn/2,4** и **spawn_link/2,4**, даже если в этот момент на узле **Node** выполняется вызов одной из функций **rpc:block_call/4** и **rpc:block_call/5**.

>>

«Полезный модуль для многозадачных приложений — **rpc**.»

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

«Неподходящие» объекты языка

При использовании функций семейства `global:set_lock/1,2,3` для установления блокировки применяется идентификатор блокировки, имеющий вид `{ResourceId, LockRequesterId}`, где `ResourceId` – идентификатор запрашиваемого ресурса, `LockRequesterId` – идентификатор стороны, запрашивающей блокировку на ресурс. В качестве как `ResourceId`, так и `LockRequesterId` может выступать любой объект языка Erlang. Но на величину `ResourceId` есть ограничения: атомы `global`, `dist_ac`, `mnesia_adjust_log_writes`, `pg2`, `mnesia_table_lock` не годятся как идентификаторы ресурсов при установлении блокировки, и их применять не рекомендуется.

Предположим, что перед нами стоит задача выполнить удаленный вызов функции и получить результат этого вызова сразу на нескольких узлах. Конечно, мы можем ее решить, последовательно вызывая функцию `rpc:call/4` для каждого узла, где необходимо удаленно выполнить функцию и получить результат. Вполне очевидно, что такое решение неэффективно: каждый следующий вызов функции `rpc:call/4` мы сможем сделать только когда предыдущий вызов вернет значение, а задача, что очевидно, легко параллелируется. Но нам нет нужды реализовывать такое параллельное решение; для этого модуль `rpc` содержит функции `rpc:multicall/4` и `rpc:multicall/5`. Функция `rpc:multicall/Nodes, Module, Func, Args, Timeout` вычисляет значение функции `Module:Func` с аргументами `Args` на узлах из списка `Nodes` и возвращает результат в виде `{ResL, BadNodes}`, где `ResL` – список полученных результатов, `BadNodes` – список узлов, на которых вычисление функции окончилось неудачей. Параметр `Timeout` в функции `rpc:multicall/5` устанавливает максимальное время ожидания ответа от каждого из узлов. Если за время `Timeout` от какого-либо узла ответа не получено, узел попадает в список `BadNodes`, т. е. мы считаем, что вычисление функции на данном узле окончилось неудачей. Список полученных результатов `ResL` содержит результаты вычисления функции `Module:Func` с аргументами `Args` в произвольном порядке (относительно порядка узлов в списке `Nodes`) и без привязки к узлу, на котором этот результат был получен.

Если такая привязка нужна, то сама функция должна ее возвращать – например, в виде кортежа, состоящего из имени узла и вычисленного результата. Понятно, что функция `rpc:multicall/5` (и `rpc:multicall/4`) возвращает управление только тогда, когда будет сформирован результат, т. е. является синхронной функцией.

Все вышеописанные функции из модуля `rpc` были синхронными функциями, но вызов синхронной функции приводит к блокированию вызывающего процесса, что не всегда желательно. А значит, необходим асинхронный вызов функции на удаленном узле: для этого мы сначала вызываем функцию `rpc:acsync_call/4`, которая возвращает ключ, используемый потом для получения результата вызова при помощи одной из функций `rpc:yield/1`, `rpc:nb_yield/1` или `rpc:nb_yield/2`. Функция `rpc:yield/1` блокирует вызывающий процесс до получения результата (если результат вызова уже готов, то никакой блокировки не будет). Функция `rpc:nb_yield(Key, Timeout)` ждет готовности результата вызова согласно параметру `Timeout` и возвращает либо результат (точнее, кортеж `{Value, Value}`, где `Value` – результат вызова), либо атом `timeout`; функция `rpc:nb_yield/1` эквивалентна вызову `rpc:nb_yield(Key, 0)`.

Давайте коротко поговорим об оставшихся функциях из модуля `rpc`. Если нам возвращаемое значение не нужно (обычно это

функции с некоторым побочным эффектом), то для удаленного вызова можно воспользоваться функцией `rpc:cast/4`, которая не блокирует выполнение вызывающего потока. Если необходимо вычислить значение функции, возвращаемое значение которой нас не интересует, на нескольких узлах, для этого можно воспользоваться одной из функций `rpc:eval_everywhere/3,4`.

Предположим, что на некотором наборе узлов у нас есть процессы (на каждом узле) с одинаковым локальным зарегистрированным именем `Name`. Если мы хотим послать всем этим процессам сообщение синхронно, то для этого модуль `rpc` содержит функцию `rpc:sbcast(Nodes, Name, Msg)` (а также функцию `rpc:sbcast/2`). Эта функция шлет сообщение `Msg` процессам с локальным именем `Name` на узлах из списка `Nodes` и возвращает кортеж `{GoodNodes, BadNodes}`, где `GoodNodes` – список узлов, на которых существует процесс с именем `Name`, `BadNodes` – список узлов, на которых процесса с именем `Name` нет. Эта функция гарантирует только, что сообщение `Msg` будет послано всем существующим процессам с локальным именем `Name` на узлах из списка `Nodes`, но не обработано этими процессами. Если мы хотим выполнить ту же самую задачу, но асинхронно, то к нашим услугам в модуле `rpc` есть функции `rpc:abcast/2` и `rpc:abcast/3`.

Предыдущие функции позволяли просто послать некоторое сообщение удаленному процессу и не ожидали никакого ответа от него (хотя, включив в сообщение идентификатор процесса отправителя, вполне можно реализовать и получение ответного сообщения от удаленного процесса). Если же наша задача состоит в том, чтобы послать сообщение и получить ответ, то для этой цели модуль `rpc` содержит еще ряд функций. Функция `rpc:server_call(Node, Name, ReplyWrapper, Msg)` предназначена для синхронной посылки сообщения `Msg` процессу с локальным именем `Name`, зарегистрированным на узле `Node`. При этом удаленный процесс должен ожидать сообщение в виде `{From, Msg}`, где `From` – идентификатор процесса инициатора, и отсыпать ответное сообщение обратно в виде `{ReplyWrapper, Node, Reply}`. Здесь `Reply` – это результат выполнения запроса; это либо просто некоторый

объект языка Erlang, либо кортеж `{error, Reason}` в случае возникновения ошибки. Если мы хотим отослать сообщения нескольким процессам с локальным именем `Name`, зарегистрированным на разных узлах, то для этого в модуле `rpc` определены функции `rpc:multi_server_call/2` и `rpc:multi_server_call/3`.

И, наконец, для параллельного решения типичных задач в модуле `rpc` определены еще две функции: `rpc:parallel_eval/1` и `rpc:pmapper/3`. Функция `rpc:parallel_eval/1` позволяет параллельно вычислить значения функций на узлах, соединенных с данным узлом, и возвращает результат этого вычисления в том же порядке, в каком шли исходные функции. Функция `rpc:pmapper/3` является версией функции `lists:map/2`, работающей параллельно.

Займемся работой с узлами. В прошлом номере мы говорили о модуле `global`, который содержит функции для работы с глобальными зарегистрированными именами процессов (это функции `global:register_name/2,3`, `global:re_register_name/2,3`, `global:registered_names/0`, `global:unregister_name/1`, `global:whereis_name/1` и `global:send/2`). Помимо этого, данный модуль содержит функциональность для работы с глобальными блокировками. Глобальная блокировка – это блокировка на какой-либо ресурс (заданный идентификатором) либо на уровне узла, либо на уровне всех узлов, соединенных друг с другом. Для управления блокировками используется специальный идентификатор, который имеет

«Необходим асинхронный вызов функции на удаленном узле.»

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

вид **{ResourceId, LockRequesterId}**, где **ResourceId** – некоторый идентификатор ресурса, **LockRequesterId** – идентификатор того, кто запрашивает. И **ResourceId**, и **LockRequesterId** могут быть любыми объектами языка Erlang; идентификатор запрашиваемой стороны не обязательно должен быть идентификатором или зарегистрированным именем процесса. Мы поговорим более подробно о глобальных блокировках на следующем уроке, а сейчас просто рассмотрим предназначенные для этого функции. Чтобы установить блокировку, используется семейство функций **global:set_lock/1,2,3**. Чтобы снять блокировку, используется семейство функций **global:del_lock/1,2,3**. Наконец, у нас есть возможность установить блокировку и после успешного установления вычислить значение некоторой заданной функции: для этого определено семейство функций **global:trans/2,3,4**. Если при установлении блокировки окажется, что блокировка на запрашиваемый ресурс уже установлена кем-то еще, то процесс, во время выполнения которого мы устанавливаем блокировку, перейдет в состояние ожидания. Это состояние закончится, как только блокировка на запрашиваемый ресурс будет снята другим процессом и установлена нашим процессом. А если установление блокировки на ресурс ожидает несколько процессов, то блокировку установит любой из них (мы не можем полагаться на какой-либо порядок установления блокировки ожидающими процессами). Состояние ожидания может также закончиться, если будет превышено предельное время ожидания, передаваемое в качестве параметра (на самом деле передается не предельное время ожидания, а количество попыток установления блокировки), если мы не указали, что установления блокировки процесс должен ожидать бесконечно долго.

Последняя наша сегодняшняя тема – это создание и работа с вспомогательными дочерними узлами [slave nodes]. Пусть для создания распределенной системы нам необходимы несколько узлов, расположенных на разных компьютерах (хостах). Понятно, что развертывание такой инфраструктуры потребует некоторых добавочных усилий по ее администрированию и поддержке. Но если нас устраивает ситуация (а в большинстве случаев это так), что время жизни всех разворачиваемых узлов будет зависеть от времени жизни некоторого главного узла, и весь ввод/вывод будет идти через этот главный узел (консольный и файловый ввод/вывод), то мы можем сильно упростить себе жизнь.

Для этого нам достаточно создать только один главный узел руками, после чего в коде инициализации (или где-то еще) мы можем создавать вспомогательные дочерние узлы на определенных компьютерах (хостах). Это означает, что множество компьютеров и узлов на них мы зададим при помощи некоторого файла конфигурации. Вся эта функциональность реализована в модуле **slave**. Мы можем создать новый дочерний вспомогательный узел на компьютере (хосте) **Host**, с именем **Name** (имя создаваемого

узла будет **Name@Host**) и параметрами командной строки для запуска среди времени выполнения Erlang (процесса **erl**) при помощи семейства функций **slave:start/1,2,3**. Мы можем сделать то же самое при помощи семейства функций **slave:start_link/1,2,3**, но тогда время жизни созданного вспомогательного узла будет зависеть от вызывающего процесса. И, наконец, можно завершить работу узла при помощи функции **slave:stop/1**. Следует сказать, что для создания узлов на компьютерах (хостах), отличных от того, на котором запущен главный узел, используется утилита **rsh** (или при запуске главного узла при помощи ключа **-rsh Program** задается альтернатива утилите **rsh**).

С дочерними вспомогательными узлами (и модулем **slave**) связана еще одна близкая тема: пулы узлов. Это предопределенные наборы уже созданных узлов, предназначенные для обслуживания запросов клиентов на выполнение той или иной функции; при этом выбором узла и созданием процесса для выполнения занимается сам пул узлов. В качестве такого узла выбирается узел с наименьшей загрузкой. Создать пул узлов с именем **Name** можно при помощи семейства функций **pool:start/1,2**. Узел, на котором происходит вызов одной из функций **pool:start/1,2**, становится главным узлом пула. Вспомогательные узлы пула создаются при помощи одной из функций **slave:start/2,3**; файл **.hosts.erlang** определяет хосты, где будут созданы вспомогательные узлы (на каждом хосте из этого файла по одному узлу). Если файл **.hosts.erlang** в системе отсутствует, создание пула узлов завершится с ошибкой. Мы можем присоединить уже созданный узел к пулу при помощи функции **pool:attach/1**. При помощи функции **pool:stop/0** мы можем завершить работу пула (и «убить» все дочерние узлы). При помощи функции **pool:get_nodes/0** мы получим список всех узлов, составляющих пул, а при помощи функции **pool:get_node/0** – наименее загруженный узел из пула.

И, наконец, при помощи функций **pool:pspawn/3** и **pool:pspawn_link/3** создается процесс для выполнения функции **Module:Func** с аргументами **Args**; при этом выбором узла и созданием процесса занимается пул потоков. Функции **pool:pspawn/3** и **pool:pspawn_link/3** возвращают управление сразу после создания процесса; при этом они возвращают идентификатор созданного процесса. Это означает, что если надо получить значение функции, об этом следует позаботиться самому; например, в качестве **Module:Func** передавать некоторую функцию-обертку, которая внутри выполняет интересующую нас функцию и посыпает нам ее значение через механизм сообщений.

В этой статье мы поговорили о библиотеках и функциях, которые предназначены для упрощения жизни нам, разработчикам многозадачных и распределенных приложений. По-моему, создателям языка Erlang это удалось. А в следующий раз мы рассмотрим интересную и важную тему синхронизации задач. **LXF**

Хранение состояния в цикле обработки сообщений.

Рассмотрим пример некоторой гипотетической функции сообщений, демонстрирующий, как хранить состояние в цикле обработки сообщений:

```
message_handler(State) ->
    receive
        {m1, From} ->
            ReturnValue = some_func1(State),
            From ! {r1, ReturnValue},
            message_handler(State);
        {m2, Data} ->
            NewState = some_func2(State, Data),
            message_handler(NewState);
            _Other -> message_handler(State)
    end.
```

Приведенный пример содержит три обработчика сообщений: для сообщений **{m1, From}**, для сообщений **{m2, Data}** и для всех остальных сообщений.

При обработке сообщения **{m1, From}** мы вычисляем некоторое значение с использованием текущего состояния, возвращаем его инициатору запроса **From**, и вызываем рекурсивно сами себя, передавая текущее состояние **State** в качестве параметра. Видно, что в данном обработчике состояние не меняется.

При обработке сообщения **{m2, Data}** мы вычисляем новое состояние **NewState** с использованием старого состояния **State** и некоторых данных **Data**, после чего рекурсивно вызываем сами себя,

передавая новое состояние **NewState** в качестве параметра.

Последний обработчик демонстрирует нам, как при помощи механизма соответствие шаблону (pattern matching) мы можем обрабатывать (в нашем случае просто извлекать из очереди сообщений) все остальные сообщения. В этом обработчике мы с сообщением ничего не делаем, а просто вызываем рекурсивно сами себя, передавая текущее состояние **State** в качестве параметра.

И, наконец, при создании процесса мы задаем начальное состояние для обработчика сообщений следующим образом: **spawn(fun() -> message_handler.initState) end**.