Erlang: Устоим

Андрей Ушаков разбирается в хитросплетениях процессов, стараясь обеспечить их правильное взаимодействие и отказустойчивость.

Наш эксперт

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

Создание реальных распределенных систем невозможно без рассмотрения такой важной темы, как отказоустойчивость. А для этого нам необходимо знать ответы на следующие вопросы: как должна реагировать система, если во время выполнения какой-либо задачи что-то пошло не так? Как должна реагировать система, если во время выполнения задачи возникла необработанная (этой задачей) ошибка? Как должны реагировать другие задачи на ошибку в какой-либо задаче? Какая поддержка со стороны системы есть для очистки ресурсов? Ответы на эти вопросы применительно к языку Erlang мы и рассмотрим в данной статье.

Начнем же мы разговор с общих понятий и определений. Как мы знаем, любая задача состоит из последовательности шагов, которые необходимо выполнить для ее решения. Каждый шаг может зависеть от ряда предыдущих шагов, а может и не зависеть ни от какого предыдущего шага. В качестве примера зависимых друг от друга шагов можно привести следующий: открытие файла и последующее чтение данных из этого файла. Очевидно, что для успешного выполнения второго шага (чтения данных из файла) необходимо, чтобы первый шаг (открытие файла) выполнился успешно. Понятно, что если первый шаг по тем или иным причинам не выполнится, то нет смысла продолжать выполнение и переходить ко второму и последующим шагам. При этом возникают следующие, вполне логичные вопросы: кто будет прерывать поток выполнения и как в коде программы узнать, что какой-либо из шагов пошел не так? В связи с этими вопросами очень часто возникает еще и следующий вопрос: если мы на середине некоторой задачи понимаем, что она не ладится, и собираемся прервать выполнение задачи, то как «очистить» все выделенные до этого шага ресурсы? Понятие «очистка» ресурса зависит от самого ресурса: для памяти это ее возврат в кучу, для файла – его закрытие, для транзакции базы данных – ее откат, и т. д.

Рассмотрим, какие у нас есть подходы для решения этих задач. Наиболее простой подход – использование кода ошибки по выполнении той или иной операции. Выглядит это следующим образом: после выполнения той или иной операции мы проверяем специальную переменную, которая служит для хранения кода ошибки. Обычно такие операции являются функциями, по возвращаемому значению которых можно понять, что операция выполнилась с ошибкой, а уж по коду ошибки можно определить, что же конкретно пошло не так. Для примера давайте взглянем на POSIX API. Так, для открытия файла мы используем функцию **open**. Эта функция открывает или создает файл и возвращает дескриптор этого файла (целое неотрицательное число). Если что-то пойдет не так во время выполнения этой операции, функция **open** вернет -1. В этом случае мы можем обратиться к переменной **errno**, чтобы понять, что именно пошло не так (и, возможно, проинформировать об этом пользователя). Из нашего примера видно, что метод на самом деле очень прост. Но у этого метода есть ряд серьезных недостатков. Самый главный из них заключается в том, что ответственность за проверку и прерывание потока выполнения лежит исключительно на нас. Если мы забудем это сделать, то в итоге получим совсем не то, на что рассчитывали. Другой большой недостаток этого подхода состоит в том, что написание кода, «очищающего» ресурсы, становится непростой задачей (возможные методы решения этой задачи см. во врезке «Полезные заметки»). Есть у этого подхода и еще один недостаток, особенно заметный при его сравнении с подходом на основе исключений: работа с возвращаемым значением дает нам слишком мало информации о том, что именно не заладилось. Работая с исключениями, мы можем получить такие атрибуты, как тип ошибки (он же код ошибки); текст, описывающий проблему; место, где произошла ошибка. Всей этой дополнительной информации при работе с кодами ошибок у нас нет.

> «По коду ошибки можно определить, что конкретно пошло не так.»

Другой подход к работе с ошибками – использование исключений. Исключение – это объект, содержащий описание возникшей проблемы, место ее возникновения и, возможно, еще какие-то атрибуты, относящиеся к проблеме. Но главное отличие этого подхода от подхода с использованием кодов ошибок в том, что при возникновении исключения нормальное выполнение задачи будет прервано и начинается поиск первого подходящего обработчика возникшей исключительной ситуации.

Для понимания того, как происходит поиск, разберемся, как осуществляется выполнение какого-либо приложения. Единицей выполнения всегда является функция. Выполнение приложения начинается с некоторой функции, называемой точкой входа [entry point]; в процессе работы эта функция может вызывать другие функции, а те, в свою очередь, могут вызывать еще какие-либо функции, и т. д. В любой момент времени состояние выполнения программы содержит, помимо указателя команд, и информацию о том, какие функции были вызваны перед тем, как выполнение программы пришло в текущую точку (на которую указывает указатель команд). Если во время выполнения программы происходит исключение, нормальное выполнение программы приостанавливается (средой выполнения) и начинается раскрутка [unwind] стека в поисках первого подходящего обработчика возникшего исключения. Поиск идет следующим образом: сначала мы просматриваем список зарегистрированных обработчиков в текущей точке выполнения. Если среда выполнения нашла подходящий обработчик, то выполнение программы передается на него; после обработки исключения (если во время обработки не было сгенерировано никакого исключения) выполнение программы продолжается в нормальном режиме. Если среда выполнения подходящий обработчик не нашла или если во время работы обработчика было сгенерировано это же или какое-либо другое исключение, среда выполнения переходит в точку программы, из которой была вызвана данная функция, и продолжает поиск там. Эта точка программы принадлежит некоторой родительской функции, т. е. функции, в ходе выполнения которой была вызвана данная функция.

перед отказами

Полезные заметки

Давайте рассмотрим ближе проблему грамотной очистки ресурсов в ситуации, когда о том, что произошла ошибка, мы можем узнать только при помощи возвращаемых значений и кодов ошибки. Для этого напишем несколько вариантов небольшого фрагмента кода на языке **C** под операционную систему **Linux**, в котором мы откроем два файла и выполним с этими открытыми файлами некоторую работу, после чего закроем их. Для определенности будем считать, что этот фрагмент находится внутри функции с типом возвращаемого значения **void**. Первый вариант – обрабатывать проблемы при открытии файла, и если файл открыть не удалось, передавать управление из фрагмента кода наружу. Если при открытии второго файла у нас возникнут какие-либо проблемы, то мы выйдем из фрагмента кода, оставив незакрытым первый файл. Таким образом, налицо потенциальная утечка ресурсов.

```
int descr1, descr2;
descr1 = open("file1.dat", 0_RDWR);
if (-1 == descr1) return;
descr2 = open("file2.dat", 0_RDWR);
if (-1 == descr2) return;
some_task(descr1, descr2);
close(descr2);
close(descr1);
```

Переделаем этот фрагмент, устранив потенциальную утечку ресурсов. Новый вариант работает следующим образом: мы пытаемся открыть второй файл, только если первый файл открылся успешно; выполняем некоторую работу с файлами, только если оба файла открылись успешно, и т. д. Для этого применим ряд вложенных блоков **if**. Сразу же можно увидеть потенциальный недостаток этого варианта: вложенность блоков **if** сильно разрастается при наличии нескольких ресурсов, подлежащих «очистке».

```
int descr1, descr2;
descr1 = open("file1.dat", 0_RDWR);
if (-1 == descr1) {
  descr2 = open("file2.dat", 0_RDWR);
  if (-1 == descr2) {
    some_task(descr1, descr2);
    close(descr2);
  }
  close(descr1);
}
```

Сделаем еще одну попытку реализовать фрагмент так, чтобы избежать утечки ресурсов. Давайте, если нам не удалось открыть второй файл, закрывать первый перед выходом из фрагмента кода. Именно этот подход и используется в следующей реализации. Эта реализация тоже далека от идеала: у нас возникает дублирование кода по очистке ресурсов в конце фрагмента и в ситуации, когда второй файл открыть не удалось.

```
int descr1, descr2;
descr1 = open("file1.dat", 0_RDWR);
if (-1 == descr1) return;
descr2 = open("file2.dat", 0_RDWR);
if (-1 == descr2) {
  close(descr1);
  return;
}
some_task(descr1, descr2);
close(descr2);
close(descr1);
```

И последняя попытка решить проблему с очисткой ресурсов. Сделаем следующие обязательные шаги. Все переменные, содержащие ресурсы, должны быть инициализированы значениями, означающими отсутствие ресурса (для дескрипторов файлов это значение **-1**, для динамически выделяемой памяти – **NULL**). Всю очистку ресурсов мы помещаем в конец фрагмента и помечаем меткой; при этом при очистке каждого ресурса проверяем, был ли этот ресурс выделен (например, что дескриптор файла не равен **-1**). И, наконец, если какой-либо ресурс выделить не удалось, то в этой ситуации мы переходим напрямую к секции очистки ресурсов. В результате код получается простым и разделенным с кодом для очистки ресурсов. Минус этого решения только один: использование оператора **goto** для перехода к секции очистки ресурсов.

```
int descr1, descr2;
descr1 = -1;
descr2 = -1;
descr1 = open("file1.dat", 0_RDWR);
if (-1 == descr1) goto CLEANUP;
descr2 = open("file2.dat", 0_RDWR);
if (-1 == descr1) goto CLEANUP;
some_task(descr1, descr2);
CLEANUP:
if (-1 != descr2) close(descr2);
if (-1 != descr1) close(descr1);
```

Вполне возможна ситуация, что во время поиска подходящего обработчика исключений среда выполнения пришла в функцию, являющуюся точкой входа, и не нашла в ней специализированного обработчика для данного исключения. В такой ситуации будет вызван обработчик исключений по умолчанию, который завершит данное приложение. Возникает вполне очевидный вопрос: как в этом случае гарантировать очистку ресурсов? Для решения этой проблемы мы можем зарегистрировать специальный обработчик для очистки ресурсов (как вместе с регистрацией обработчиков исключений, так и сам по себе), который будет всегда вызван (что гарантируется средой выполнения) во время раскрутки стека.

Давайте посмотрим, что же для обработки ошибок и «очистки» ресурсов у нас есть в языке Erlang. А в языке Erlang одинаково широко используются оба подхода: и коды ошибок и исключения времени выполнения. Подход с использованием кода ошибки в языке Erlang несколько отличается от «классического» подхода: вместо использования отдельной глобальной переменной для хранений кода ошибки, многие BIF и функции в случае успеха и неудачи сохраняют разные объекты. Различить ситуации успешного и неуспешного выполнения легко при помощи операции соответствия шаблону [pattern-matching]. Так, например, функция **file:open/2** модуля **file** используется для того, чтобы открыть какой-либо файл перед тем, как начать работу с этим файлом. Если эта операция выполнится успешно, то будет возвращен кортеж вида **{ok, IoDevice}**, где **IoDevice** – некоторый описатель файла, применяемый для дальнейшего доступа к нему. Если эта операция выполнится с ошибкой, будет возвращен кортеж вида **{error, Reason}**, где **Reason** – некоторый объект, описывающий, что пошло не так. С другой стороны, многие BIF и функции генерируют исключение, если что-то пошло не так во время их выполнения. Так, например, если мы возьмем BIF **integer_to_list/1** и передадим ей в качестве аргумента атом **ab**, то мы получим в результате ошибку времени выполнения **badarg**, означающую, что данный аргумент для функции не корректен.

Использование кодов ошибок достаточно очевидно и не требует специального синтаксиса, в отличие от работы с исключениями. »

В языке Erlang исключения бывают трех разных классов: **error**, **exit**, **throw**. Класс исключения определяется тем, при помощи какого BIF было сгенерировано исключение. Если исключение сгенерировано при помощи одной из BIF **error/1** или **error/2**, то класс исключения будет **error**; если исключение сгенерировано при помощи BIF **exit/1**, то класс исключения будет **exit**; если исключение сгенерировано при помощи BIF **throw/1**, то класс исключения будет **throw**. Исключения, генерируемые средой выполнения Erlang, BIF и библиотечными функциями, всегда имеют класс **error**; мы же в своем коде можем генерировать исключения любого из трех классов. Генерировать исключения самостоятельно мы уже научились; следующий шаг – понять, как мы можем работать со сгенерированными исключениями.

Наиболее простая конструкция для этой цели – **catch Expr**, где **Expr** – произвольное выражение. Если во время вычисления выражения **Expr** никакого исключения не будет сгенерировано, то значением выражения **catch Expr** будет значение выражения **Expr**. Если же во время вычисления выражения будет сгенерировано исключение, то значение выражения будет зависеть от класса сгенерированного исключения. Если исключение имеет класс **error**, то выражение **catch Expr** вернет **{'EXIT', {Reason, Stack}}**, где **Reason** – причина возникновения исключения (или аргумент вызова **error/1**), **Stack** – возврат по стеку [stacktrace] до места возникновения исключения. Если исключение имеет класс **exit** (это означает, что где-то в коде был сделан вызов **exit(Term)**), то выражение **catch Expr** вернет **{'EXIT', Term}**. Если исключение имеет класс **throw** (это означает, что где-то в коде был сделан вызов **throw(Term)**), то выражение **catch Expr** вернет **Term**. Очевидно, что использование выражения **catch Expr** позволяет покрыть все возможные проблемы работы с исключениями, но использовать это выражение не всегда удобно. Поэтому в языке Erlang есть улучшенный вариант выражения **catch Expr**: это выражение **try/catch**. Базовая версия этого выражения имеет следующий вид:

```
try Exprs
catch
  [Class1:]ExceptionPattern1 [when ExceptionGuard1] →
CatchBody1;
...
  [ClassN:]ExceptionPatternN [when ExceptionGuardN] →
CatchBodyN
end
```

Здесь **Exprs** – последовательность выражений, в которых может быть сгенерировано исключение, $\mathbf{Class}_i$ – один из трех возможных классов исключений, $\mathbf{ExceptionPattern}_i$ – выражение соответствия шаблону для отлавливаемого исключения, $\mathbf{ExceptionGurad}_i$ –выражение охраны для отлавливаемого исключения, $\mathbf{CatchBody}_i$ – тело обработчика исключения. Класс исключения и выражения охраны являются необязательными элементами при задании обработчика. Значение выражения **try/catch** вычисляется следующим образом. Если во время вычисления последовательности выражений **Exprs** никакого исключения не было сгенерировано, то значением выражения **try/catch** будет значение последовательности выражений **Exprs**. Если во время вычисления последовательности выражений сгенерируется исключение, то среда выполнения Erlang будет искать первый подходящий обработчик последовательно среди обработчиков блока **catch**. По нахождении первого подходящего обработчика поиск среди обработчиков блока **catch** прекращается, и значение выражения тела обработчика $\mathbf{CatchBody}_i$ будет значением всего выражения **try/catch**. Если же подходящего обработчика среди обработчиков блока **catch** не обнаружится, то сгенерированное исключение выйдет за пределы выражения **try/catch**. Существует более сложный вариант выражения **try/catch**, который является гибридом выражения **try/catch** в простом варианте и выражения **case** для значения последовательности выражений **Exprs**. Об этом варианте выражения **try/catch** мы поговорим более подробно во время одного из практикумов (интересующиеся читатели могут посмотреть документацию к языку Erlang).

У нас остался еще один не затронутый пока вопрос, связанный с работой с исключениями: «очистка» ресурсов. Для этого выражение **try/catch** содержит специальный блок (в выражении **catch Expr** такого блока нет, по логике работы этого выражения он там и не нужен): это блок **after**. Логика вычисления выражения **try/catch** не меняется в зависимости от того, есть блок **after** в этом выражении или нет. Это означает, что блок **after** нужен только для очистки ресурсов и значение последовательности выражений в блоке **after** «теряется» после выполнения этого блока. Давайте на небольшом примере рассмотрим, как использовать блок **after**:

```
{ok, File} = file:open(FileName, [read, binary])
try
        {ok, Data} = file:read(File, Size),
        binary_to_term(Data)
after
        file:close(File)
end
```

В этом примере мы открываем файл (до начала выражения **try/catch**), читаем из файла двоичные данные некоторого предопределенного размера, десериализуем из этих данных некоторый объект Erlang и закрываем файл. Если во время чтения данных из файла или десериализации будет сгенерировано исключение, то файл будет закрыт благодаря тому, что код закрытия находится в блоке **after**. Если все операции выполнятся успешно, то файл также будет закрыт.

До сих пор, говоря о работе с исключениями, мы подразумевали, что все действия происходят в одном процессе Erlang. Теперь давайте поговорим о работе с исключениями в ситуации, когда у нас выполняется несколько процессов Erlang – как на одном узле (или в пределах одного экземпляра среды выполнения Erlang), так и на нескольких. Очевидно, что если в каком-либо процессе Erlang во время выполнения кода будет сгенерировано исключение, и в том же процессе это исключение будет обработано, то для других процессов Erlang данное происшествие окажется незамеченным. Поэтому в дальнейшем мы будем рассматривать только ситуации, когда исключения в каком-либо процессе генерируются и не обрабатываются. Возникает вполне логичный вопрос: а что в таком случае произойдет с самим процессом и другими процессами? Процесс, в котором будет сгенерировано необрабатываемое исключение, будет завершен средой выполнения Erlang, что очевидно. Все остальные процессы, если они не были связаны с завершенным процессом, продолжают свою работу независимо от того, в одном ли экземпляре среды выполнения Erlang они работали или в разных. Более того, они даже никак не узнают об этом событии, если только сами не запросят информацию о процессе (используя для этого одну из BIF **process_info/1** или **process_info/2**). Таким образом, можно сказать, что процессы в языке Erlang независимы друг от друга с точки зрения необработанных исключений.

Подобная независимость процессов сохраняется до тех пор, пока между процессами отсутствуют связи. Связь между процессами означает, что один процесс начинает узнавать обо всех изменениях в жизни другого процесса (т. е. начинает получать события, если другой процесс завершается, как вследствие завершения выполнения своего кода, так и вследствие возникновения необработанных исключений) и реагировать на эти изменения. Связь между процессами является двусторонней: после того как

» **Пропустили номер?** Узнайте на с. 106, как получить его прямо сейчас.

один процесс установил связь с другим процессом, они оба будут получать известия обо всех изменениях в жизни процесса на другом конце связи. Связи могут быть установлены с любым количеством других процессов.

Наиболее интересен для нас вопрос, что происходит со связанными процессами, если один из них заканчивает свою жизнь (как в случае обычного завершения работы, так и в случае необработанного исключения). Для ответа на этот вопрос следует сделать следующее замечание: все процессы можно разделить на две группы: обычные процессы и процессы-супервизоры. Когда какой-либо процесс завершает свою работу, то процесс-супервизор (связанный с этим процессом) всегда получает сообщение **{'EXIT', FromPid, Reason}**. Здесь **FromPid** – это идентификатор процесса, завершившего свою работу; **Reason** – причина, по которой процесс завершил работу. Если процесс завершил свою работу естественным способом (когда завершается выполнение функции процесса), то причина **Reason** завершения процесса будет **normal**. Точно такая же причина будет, если процесс сгенерирует необработанное исключение вызовом **exit(normal)**; или если какой-либо другой процесс сделает вызов **exit(Pid, normal)**, где **Pid** – идентификатор завершаемого процесса (все это считается естественным завершением работы процесса). Если процесс сам завершил свою работу, сгенерировав необработанное исключение вызовом **exit(Reason)**; или если какой-либо другой процесс сделает вызов **exit(Pid, Reason)**, то причиной завершения процесса будет соответствующий аргумент BIF **exit/1** или **exit/2**. Если процесс сгенерирует необработанное исключение класса **error** или класса **throw**, то причина завершения процесса будет иметь следующий более сложный вид: **{Reason, Stack}**. Здесь **Reason** – причина завершения, а **Stack** – стектрейс [stacktrace], указывающий на место генерации этого исключения.

Теперь давайте поговорим об обычных процессах. Если процесс, связанный с обычным процессом, завершается естественным образом, то с таким обычным процессом ничего не происходит: он продолжает свое выполнение и никаких сообщений не получает. Если же процесс, связанный с обычным процессом, завершается из-за необработанного исключения или если этот процесс завершает какой-либо другой процесс вызовом BIF **exit/2** с причиной завершения, отличной от **normal**, то связанный обычный процесс также будет завершен. Такое завершение связанного обычного процесса не является естественным (это означает, что причина завершения связанного обычного процесса отлична от **normal**).

Рассмотрим небольшой пример, иллюстрирующий описанное выше поведение. Пусть у нас процесс **A** связан с процессами **B** и **C**, а процесс **D** связан с процессом **B**. Пусть все процессы являются обычными. Рассмотрим ситуацию, когда процесс **B** завершается естественным способом. В этом случае процессы **A**, **C** и **D** остаются «живыми», причем процессы **A** и **C** остаются связанными, а процесс **D** становится обособленным. Теперь рассмотрим ситуацию, когда процесс **B** завершается способом, отличным от естественного завершения. В этом случае все процессы, связанные с **B** (это **A и D**), завершаются, приводя к тому, что завершаются и все процессы, связанные с **A** или **D** – это процесс **C**. Видно, что если у нас есть граф связанных обычных процессов, то в случае неестественного завершения одного из процессов завершатся все процессы в этом графе.

Пусть теперь у нас процесс **A** является супервизором. Рассмотрим ситуацию, когда процесс **B** завершается естественным способом. В этом случае процессы **A**, **C** и **D** остаются «живыми», при этом процессы **A** и **C** остаются связанными, а процесс **D** становится обособленным процессом. Процесс **A** получит сообщение **{'EXIT', Pid_B, normal}**, где **Pid_B** – идентификатор процесса **B**. Теперь рассмотрим ситуацию, когда процесс **B** завершается способом, отличным от естественного завершения. В этом случае все обычные процессы, связанные с **B**, завершаются – это процесс **D**; при этом процесс **A** получит сообщение **{'EXIT', Pid_B, $Reason_B$}** (здесь **$Reason_B$** – причина завершения процесса **B**), а процесс **C** «останется в живых» (и не получит никаких сообщений). Видно, что процесс-супервизор, помимо возможности получения уведомлений о завершении связанных с ним процессов и возможной реакции на эти уведомления, экранирует процесс из одной ветви графа связанных процессов от изменений в жизни процессов из других ветвей графа связанных процессов. Разница в поведении между обычными процессами и процессами-супервизорами заключается еще и в реакции на попытку каким-либо процессом завершить данный процесс при помощи BIF **exit/2**. Если мы пытаемся завершить обычный процесс при помощи вызова **exit(Pid, Reason)**, то этот процесс завершается с причиной завершения **Reason**. Если же мы пытаемся завершить процесс-супервизор при помощи вызова **exit(Pid, Reason)**, то этот процесс получит сообщение **{'EXIT', FromPid, Reason}** и продолжит свое выполнение. Здесь **FromPid** – идентификатор процесса, пытавшегося завершить данный процесс при помощи вызова **exit(Pid, Reason)**. Если мы хотим завершить процесс-супервизор из другого процесса, то делать это следует при помощи следующего вызова: **exit(Pid, kill)**. Тогда причина завершения такого процесса (независимо от того, является ли он супервизором или нет) будет следующая: **killed**.

Теперь давайте поговорим о функциях (точнее, о BIF), которые мы будем применять для создания связей и управления, является ли процесс супервизором или обычным процессом. Вызов **process_flag(trap_exit, true)** позволяет процессу указать, что этот процесс должен быть супервизором; вызов **process_flag(trap_exit, false)** указывает, что он должен быть обычным процессом. Существует вариант BIF **process_flag/3**, позволяющий определить, является ли процесс супервизором или обычным процессом для любого другого процесса. Создать связь между текущим процессом и процессом, заданным по его идентификатору, можно при помощи BIF **link/1**; разрывается данная связь при помощи BIF **unlink/1**. Создать новый процесс и сразу же связать его с текущим процессом можно при помощи семейства BIF **spawn_link/1,2,3,4**. Главное отличие функций семейства **spawn_link/1,2,3,4** от последовательного использования функций семейства **spawn/1,2,3,4** и функции **link/1**, т.е. от последовательного создания нового процесса и связи между текущим и новым процессами, в том, что функции семейства **spawn_link/1,2,3,4** создают новый процесс и связь между новым и текущим процессами атомарно.

В языке Erlang существует альтернатива связям – это мониторы. Монитор – это однонаправленная связь между процессами, служащая только для передачи сообщения о прекращении работы процесса. Это означает, что вне зависимости от того, является ли процесс, создавший монитор к другому процессу, супервизором или обычным процессом, он всегда будет только получать сообщение о прекращении работы другого процесса. Это сообщение имеет следующий вид: **{'DOWN', Ref, process, Pid2, Reason}**, где **Ref** – это объект (типа ссылка), получаемый при создании монитора, **Pid2** – идентификатор процесса, к которому создан монитор, **Reason** – причина завершения процесса. Если процесс создает монитор к несуществующему монитору, этот процесс немедленно получит приведенное выше сообщение с причиной **noproc**. Для создания монитора используется BIF **monitor/2**; для уничтожения монитора используется одна из BIF **demonitor/1** или **demonitor/2**. Для атомарного создания нового процесса и монитора к нему используется семейство BIF **spawn_monitor/1,2**.

В данной статье мы рассмотрели такую важную часть для построения распределенных систем, как отказоустойчивость. А в следующем номере мы непосредственно перейдем к рассмотрению темы о создании распределенных приложений на языке Erlang. LXF