

# Erlang: Распределение

Андрей Ушаков осуществляет связь между узлами, где бы они ни были: и на одном компьютере, и в разных подсетях.



## Наш эксперт

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

В двух прошлых номерах мы говорили о многозадачности и об устойчивости к ошибкам в языке Erlang. Сегодня мы продолжим разговор о средствах построения промышленных приложений и поговорим о таком важном понятии, как распределенные системы.

Основными элементами для построения распределенных систем на языке Erlang являются узлы. Узел – это всего лишь именованный экземпляр среды выполнения Erlang. Чтобы сделать экземпляр среды выполнения узлом, достаточно при ее запуске задать один из параметров **-name** или **-sname** и имя, которое узел получит. Параметр **-sname** задает короткое имя; это имя может использоваться, если создаваемые узлы находятся либо на одном компьютере, либо на разных компьютерах в одной подсети. Кроме того, если сервис DNS не доступен для создаваемого узла, то создавать узел можно только с параметром **-sname**. Параметр **-name** позволяет задать длинное имя; это имя может использоваться во всех тех же случаях, что и короткое, а также в случаях, когда узлы располагаются на разных компьютерах в разных подсетях.

Пусть имя компьютера является полностью определенным доменным именем – например, **stdstring.example.com**. В этом случае, если мы создаем узел с ключом **-sname test**, имя узла будет **test@stdstring**, а с ключом **-name test - test@stdstring.example.com**. Теперь предположим, что имя компьютера является обычным именем (а не полностью определенным доменным именем), например, **stdstring**. Тогда, если мы создаем узел с ключом **-sname test**, имя узла будет **test@stdstring**. Если же попытаться создать узел с ключом **-name test**, мы получим ошибку при создании узла (при запуске экземпляра среды выполнения Erlang). Решение в этом случае достаточно простое: при создании узла с помощью ключа **-name** следует указать полное имя узла. Если, например, создавать узел мы будем с ключом **-name test@stdstring.cpp**, то аргумент ключа **test@stdstring.cpp** и будет именем созданного узла. В дальнейшем, когда мы будем использовать имя узла, его следует задавать как атом. Если атомы в своем составе содержат символы, отличные от строчных латинских

букв и символа **\_**, то тело атома должно быть заключено в одинарные кавычки. Так, например, для узла **test@stdstring.cpp** его имя мы будем задавать следующим образом: **'test@stdstring.cpp'**.

При назначении узлам имен следует учитывать следующее соображение: узлы с разными типами имен (т. е. с длинными и короткими) не могут взаимодействовать друг с другом через стандартный механизм обмена сообщениями. Если такое взаимодействие необходимо, задавайте создаваемым узлам имена одного типа. Отметим также, что если экземпляр среды выполнения Erlang не является узлом, то имя у такого экземпляра все равно есть: это атом **node@nohost**.

Помимо имени, при создании узла необходимо задавать также и некоторое «магическое» значение, называемое **magic cookie**. Данное значение является атомом и используется для процесса аутентификации узлов в момент установления соединения между ними. При этом происходит следующее: когда один узел пытается

установить соединение с другим узлом, среда выполнения Erlang сравнивает «магические» значения этих двух узлов; если они равны, то связь между узлами устанавливается, если же нет, связь между

узлами установлена не будет. После установления соединения между узлами их «магические» значения более не используются. А значит, после установления соединения с каким-либо узлом с одним «магическим» значением, мы, изменив «магическое» значение на текущем узле, можем установить соединение с другим узлом с другим «магическим» значением. Пусть, например, у нас есть два узла – **node1@comp1** и **node2@comp2**, для которых «магические» значения установлены в **A** и **B** соответственно. Перед тем, как установить соединение с узлом **node1@comp1**, мы устанавливаем «магическое» значение на текущем узле в **A**, после чего, желая установить соединение с узлом **node2@comp2**, мы устанавливаем «магическое» значение на текущем узле в **B**. «Магическое» значение на узле можно установить тремя способами, причем два из них позволяют установить «магическое» значение узла только при его создании. Самый простой способ – ничего не делать; при этом среда выполнения Erlang (при запуске ее как узла) сама прочитает значение из файла **\$HOME/.erlang.cookie** (а если такого файла нет, то среда выполнения Erlang его создаст). Чтобы при таком способе несколько узлов, расположенные на разных компьютерах, могли установить соединение друг с другом, на всех таких компьютерах содержимое файла **\$HOME/.erlang.cookie** должно быть одинаковым. Если же мы создаем несколько узлов на одном компьютере, то нам беспокоиться не о чем: у всех узлов, создаваемых таким способом, будет одно и то же «магическое» значение. Другой способ, позволяющий

задать «магическое» значение при создании узла – использовать опцию командной строки **-setcookie** и желаемое «магическое» значение. Например, если при создании узла с коротким именем **test** (на компьютере с именем **stdstring**) мы хотим, чтобы «магическое» значение было **some\_value**, то сделать это мы можем сле-

## Замечание о примерах

В данной статье, когда мы обсуждаем примеры, очень часто встречаются инструкции наподобие: на узле с именем **NodeName** введите команду **Cmd** (например – на узле **n1@stdstring** введите команду **nodes()**). По этому поводу заметим следующее. Узел является именованным экземпляром среды выполнения Erlang. Создавая узел, мы запускаем в консоли среду выполнения

Erlang с ключами. Среда выполнения Erlang – интерактивное консольное приложение; значит, после ее запуска мы работаем уже в консоли Erlang. А при вводе команды в консоли Erlang каждую команду необходимо завершать символом **:**; команда отправляется на выполнение после нажатия клавиши **Enter**. Так что не забывайте ставить точку.

# дистанционные системы

дующим образом: `erl -sname test -setcookie some_value`. И наконец, последний способ установить «магическое» значение – это использовать функцию `erlang:set_cookie(Node, Cookie)` модуля `erlang`. Она позволяет установить «магическое» значение `Cookie` на узле `Node`. Например, чтобы установить «магическое» значение `SomeValue` на узле `n1@stdstring`, мы делаем следующий вызов: `erlang:set_cookie('n1@stdstring', 'SomeValue')`. Желая получить установленное «магическое» значение на текущем узле, мы должны использовать функцию `erlang:get_cookie/0` модуля `erlang`. Вполне понятна причина, по которой мы можем получить установленное «магическое» значение только на текущем узле; если бы мы могли получить его на любом узле, это сводило бы на нет систему аутентификации между узлами.

Проведем несколько экспериментов по установлению соединения между узлами. Для этого нам потребуется функция (BIF), о которой мы еще не говорили: `nodes/1`. Она позволяет получить список имен узлов определенного типа, так или иначе связанных с текущим узлом. Если в качестве аргумента передать атом `visible`, мы получим список видимых узлов, т. е. узлов, установивших обычное соединение с текущим узлом; если атом `hidden`, мы получим список невидимых узлов, т. е. узлов, установивших «невидимое» соединение с текущим узлом (мы поговорим об этом чуть позже); атом `connected` – список всех узлов, установивших соединение с текущим узлом; атом `this` – список, содержащий только имя текущего узла. И, наконец, если в качестве аргумента передать атом `known`, мы получим список всех узлов, известных текущему узлу, т. е. тех, которые когда-либо устанавливали с ним соединение (также этот список будет содержать и имя текущего узла). Кроме того, мы можем передать приведенные выше атомы в качестве списка аргументов; тогда мы получим список имен узлов, который является объединением списков имен узлов, полученных для каждого аргумента. Следует также сказать, что существует функция (BIF) `nodes/0`, являющаяся аналогом вызова `nodes(visible)`.

После этого небольшого вступления перейдем к собственно экспериментам. Для начала создадим три узла с короткими именами (на компьютере `stdstring`): `n1@stdstring`, `n2@stdstring` и `n3@stdstring`. Узлы в распределенной среде Erlang являются слабосвязанными; это означает, что пока к узлу не было обращения со стороны другого узла, соединение между этими узлами отсутствует. Если мы на узле `n1@stdstring` введем `nodes(visible)`, то получим пустой список; это означает, что узлы при создании не устанавливают соединение с уже созданными узлами. Для установления соединения между удаленным и текущим узлами необходимо использовать имя удаленного узла в одной из функций, обращающихся к заданному узлу. К таким функциям относятся, например, семейство функций `spawn/2,4` (о них мы поговорим чуть позже). Простейшей же функцией, которая устанавливает соединение между удаленным и текущим узлами, является функция `net_adm:ping/1` модуля `net_adm`. Эта функция проверяет, доступен ли удаленный узел для установления соединения (что ясно из названия этой функции); если да, то соединение между удаленным и текущим узлами устанавливается и возвраща-

ется атом `pong`, иначе возвращается атом `pang`. Давайте на узле `n1@stdstring` введем `net_adm:ping('n2@stdstring')`; после этого на узлах `n1@stdstring` и `n2@stdstring` введем `nodes(visible)`. На узле `n1@stdstring` мы получим список `[n2@stdstring]`, а на узле `n2@stdstring` – список `[n1@stdstring]`. Это означает, что соединение между этими узлами было установлено. Давайте на узле `n1@stdstring` введем `net_adm:ping('n3@stdstring')`; после этого на узле `n1@stdstring` введем `nodes(visible)` – и получим список `['n2@stdstring', 'n3@stdstring']`, что ожидаемо. Теперь на узлах `n2@stdstring` и `n3@stdstring` введем `nodes(visible)`; мы получим списки `['n1@stdstring', 'n3@stdstring']` и `['n1@stdstring', 'n2@stdstring']` соответственно. Этот несколько неожиданный для нас результат означает, что установление соединения – процесс транзитивный. Т. е. когда узел **A** устанавливает соединение с узлом **B**, уже имеющий установленное соединение с узлом **C**, то узел **A** попытается установить соединение и с узлом **C**.

Транзитивность процесса установления соединения является поведением по умолчанию; однако его можно изменить, создавая узел с ключом `-connect_all false`. Если такой узел участвует в процессе установления соединения, установится только одно соединение: между инициатором и явно указанным узлом (если вдруг «магические» значения у этих узлов не совпадут, то никакого соединения не будет). Предположим, что при создании узла `n3@stdstring` мы указали флаг `-connect_all false`, после чего на узле `n1@stdstring` последовательно сделали два вызова `net_adm:ping('n2@stdstring')` и `net_adm:ping('n3@stdstring')`. В этом случае, вызов `nodes(visible)` на узле `n1@stdstring` вернет `['n2@stdstring', 'n3@stdstring']`, вызов `nodes(visible)` на узле `n2@stdstring` вернет `['n1@stdstring']`, то же самое вернет и вызов `nodes(visible)` на узле `n3@stdstring`.

Подобное поведение при установлении соединения (отсутствие транзитивности) можно получить, если при создании узла указать, что это узел должен быть невидимым. Для этого при создании узла необходимо указать ключ `-hidden`. При установлении соединения между узлами, если хотя бы один из узлов является невидимым, будет установлено т. н. невидимое соединение. Это означает, что в результатах вызова `nodes()` или `nodes(visible)` узлы, соединенные с текущим узлом невидимым соединением, будут отсутствовать. Чтобы получить узлы, соединенные с данным узлом при помощи невидимых соединений, необходимо использовать вызов `nodes(hidden)`; чтобы получить все узлы, соединенные с данным узлом – вызов `nodes(connected)`. Для чего нужны невидимые узлы? Для того, чтобы вести проверку некоторых частей системы (располагающихся на одном или нескольких узлах), не влияя на всю систему целиком (не устанавливая соединения со всеми остальными узлами).

Давайте вернемся к нашему примеру. Предположим, что при создании узла `n3@stdstring` мы указали флаг `-hidden`, после чего на узле `n1@stdstring` последовательно сделали два вызова `net_adm:ping('n2@stdstring')` и `net_adm:ping('n3@stdstring')`. В этом случае, вызов `nodes(visible)` на узле `n1@stdstring` вернет `['n2@stdstring']`, вызов `nodes(hidden)` вернет `['n3@stdstring']`, а вызов `nodes(connected)` вернет `['n2@stdstring', 'n3@stdstring']`.

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

## На темной стороне силы (на Microsoft Windows)

Когда мы создаем узел (именованный экземпляром среды выполнения Erlang) и не указываем (при помощи ключа `-setcookie`) «магическое» значение, оно берется из файла `$HOME.erlang.cookie`, а если такого файла нет, то среда выполнения Erlang создает его. Пусть мы хотим создать несколько узлов, располагающиеся на разных компьютерах, которые должны быть соединены друг с другом, и не хотим при этом задавать «магическое» значение при помощи ключа `-setcookie` или вызова `erlang:set_cookie/2`. Тогда надо обеспечить, чтобы

содержимое файла `$HOME.erlang.cookie` на всех компьютерах было одинаковым. Как мы знаем, язык и среда Erlang являются кроссплатформенными, и возможна ситуация, что часть узлов в такой сети будут создаваться на компьютерах под управлением ОС Microsoft Windows. Возникает вполне логичный вопрос, где в таком случае располагается файл, хранящий «магическое» значение по умолчанию. В Microsoft Windows этот файл точно так же называется `.erlang.cookie`, и располагается он в корне пользовательской директории. Найти расположение

этой директории нам помогут две переменные окружения: `%HOMEPATH` и `%HOMEPATH%`. Первая возвращает букву диска, на котором располагается эта директория (в Microsoft Windows не существует единого дерева каталогов, как в Linux), а вторая – путь до пользовательской директории (без буквы диска). Поэтому полный путь до файла с «магическим» значением по умолчанию в ОС Microsoft Windows будет `%HOMEPATH%\.erlang.cookie`. Да пребудет с вами сила, и не обратитесь вы на ее темную сторону.

`stdstring']`. На узле `n2@stdstring` вызовы `nodes()`, `nodes(visible)`, `nodes(connected)` вернут `['n1@stdstring']`, а вызов `nodes(hidden)` вернет пустой список. На узле `n3@stdstring` вызовы `nodes()`, `nodes(visible)` вернут пустой список, вызовы `nodes(hidden)`, `nodes(connected)` вернут `['n1@stdstring']`.

Разорвать соединение между узлами можно двумя способами. Во-первых, можно завершить работу узла средствами операционной системы. Во-вторых, можно разорвать соединение между текущим и некоторым другим узлами при помощи функции (BIF) `disconnect_node/1`, которая в качестве аргумента принимает имя узла на другой стороне установленного соединения. В контексте разговора о разрыве соединений, поговорим также об известных узлах (узлах, которые мы можем получить вызовом `nodes(known)`). К известным относительно текущего узла относятся узлы, когда-либо устанавливавшие соединение с текущим узлом. Это означает, что, если соединение между текущим и каким-либо другим узлом уже разорвано, то этого узла в списке, который возвращает вызов `nodes(connected)`, уже не будет, а в списке, который возвращает вызов `nodes(known)`, он будет присутствовать. Если же соединение между текущим и каким-либо другим узлом не разорвано, то этот узел будет присутствовать как в списке, возвращаемом вызовом `nodes(known)`, так и в списке, возвращаемом вызовом `nodes(connected)`.

Опять вернемся к примеру. Пусть мы создали три узла `n1@stdstring`, `n2@stdstring` и `n3@stdstring` обычным способом (без всяких дополнительных флагов). Затем на узле `n1@stdstring` мы последовательно сделали три вызова `net_adm:ping('n2@stdstring')`, `net_adm:ping('n3@stdstring')`, `disconnect_node('n3@stdstring')`. Вызов функции `nodes(connected)` на узле `n1@stdstring` вернет список `['n2@stdstring']`, а вызов функции `nodes(known)` вернет список `['n1@stdstring', 'n2@stdstring', 'n3@stdstring']` (мы помним, что список известных узлов содержит также и имя текущего узла). На узле `n3@stdstring` все будет аналогично: вызов функции `nodes(connected)` вернет список `['n2@stdstring']`, а вызов функции `nodes(known)` вернет список `['n1@stdstring', 'n2@stdstring', 'n3@stdstring']`. И, наконец, на узле `n2@stdstring` (с которым никто из узлов соединение не разрывал) вызов функции `nodes(connected)` вернет список `['n1@stdstring', 'n3@stdstring']`, а вызов функции `nodes(known)` вернет список `['n1@stdstring', 'n2@stdstring', 'n3@stdstring']`.

Мы достаточно много говорили об узлах, но главный вопрос пока остался за кадром: как использовать узлы для создания

### «Разорвать соединение между узлами можно двумя способами.»

распределенных приложений. Ответ на этот вопрос достаточно прост: имя узла нам нужно только для создания процесса одной из функций из семейств `spawn/2,4`, `spawn_link/2,4`, `spawn_opt/3,5`, которые принимают имя узла в качестве первого параметра. И все. Эти функции возвращают идентификатор созданного процесса, для работы с которым имя узла не нужно. Для взаимодействия процессов опять же используются их идентификаторы, и знать узлы, на которых эти процессы располагаются, не нужно. Таким образом, видно, что при написании многозадачных приложений, выполняющихся в одном экземпляре среды выполнения Erlang и распределенных приложений, все отличие будет только при создании процессов. Это замечательный факт, и мы к нему еще вернемся при реализации практических задач.

Рассмотрим теперь базовый набор функций, предназначенный для работы с узлами (большую часть из этих функций мы уже видели и использовали). Для управления «магически

ми» значениями у нас есть две функции: `erlang:get_cookie/0` и `erlang:set_cookie/2`. Первая возвращает «магическое» значение текущего узла, вторая позволяет задать «магическое» значение произвольному

узлу. Чтобы разорвать соединение между текущим и заданным узлами, используется функция (BIF) `disconnect_node/1`. Функция (BIF) `is_alive/0` позволяет определить, может ли текущий узел (а точнее, экземпляр среды времени выполнения Erlang) быть частью распределенной системы узлов. Функции `monitor_node/2` и `erlang:monitor_node/3` позволяют включать, выключать и устанавливать опции мониторинга жизненного цикла узла. Если узел прекратит свою работу (или если узел не существует), текущему узлу будет послано сообщение `{nodedown, Node}`, где `Node` – имя узла, за которым осуществлялся мониторинг. Функция `node/0` позволяет получить имя текущего узла; функция `node/1` – имя узла, на котором расположен объект (процесс, порт, ссылка). Функция `nodes/1` возвращает список узлов, которые так или иначе связаны с текущим узлом и удовлетворяют определенным критериям (передаваемым в качестве аргумента); функция `nodes/0` является аналогом вызова `nodes(visible)`. И, наконец, семейства функций `spawn/2,4`, `spawn_link/2,4`, `spawn_opt/3,5` позволяют создать процесс на узле, имя которого идет первым аргументом этих функций.

Следующая тема – регистрация имен процессов. Чтобы взаимодействовать с каким-либо процессом (послать ему сообщение и, возможно, получить ответ), необходимо знать его идентификатор. Если один процесс создал другой процесс, то проблем

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

с взаимодействием нет: родительский процесс, зная идентификатор дочернего процесса, может послать ему в сообщении свой идентификатор, на что дочерний процесс может ответить родительскому, и т.д. Однако возможна и другая ситуация – когда мы создаем некоторый сервисный процесс обособленным способом, после чего этот сервисный процесс является точкой доступа для получения какой-либо услуги. Обычно, когда некоторый сервис предоставляет свои услуги, доступ к нему осуществляется при помощи некоторого стабильного имени. Идентификатор процесса для этого не подходит, т.к. не является стабильной величиной (это означает, что каждый раз, создавая такой сервисный процесс, я буду получать разные идентификаторы этого сервисного процесса).

Именно для этого и нужна регистрация имен процессов. Мы можем зарегистрировать некоторое имя (которое является атомом) и связать его с определенным процессом. В дальнейшем, при взаимодействии с этим процессом, мы можем использовать зарегистрированное имя вместо идентификатора процесса. Для регистрации имени процесса служит функция (BIF) **register(RegName, Pid)**, где **RegName** – регистрируемое имя процесса, **Pid** – идентификатор соответствующего процесса. Помимо регистрации имени процесса, мы можем эту регистрацию убрать (при помощи функции **unregister/1**), получить список всех зарегистрированных имен (при помощи функции **registered/0**) и получить по зарегистрированному имени процесса его идентификатор (при помощи функции **whereis/1**). У имен процессов, регистрируемых подобным образом, есть только один, но существенный минус: они регистрируются только на одном определенном узле (т.е. на двух разных узлах могут оказаться два процесса с одинаковыми зарегистрированными именами). Этот недостаток можно обойти и обращаться к процессу на узле **Node** по его зарегистрированному имени **Name** при помощи следующего синтаксиса: **{Name, Node}**.

Для примера создадим простой модуль (с именем **simple\_message\_handler**), который содержит только одну экспортную функцию **message\_loop/0**. Эта функция представляет собой просто бесконечный цикл обработки сообщений, для каждого полученного сообщения выводящий его на экран. Вот тело этой функции:

```
message_loop() ->
receive
    Message -> io:format("Message: ~p~n", [Message]),
    message_loop()
end.
```

Откомпилируем этот модуль и создадим два узла: **n1@stdstring** и **n2@stdstring**. На узле **n1@stdstring** создадим процесс, рабочей функцией которого будет функция **simple\_message\_handler:message\_loop/0** (бесконечный цикл обработки сообщений): **ProcessPid = spawn(fun simple\_message\_handler: message\_loop/0)**. После этого зарегистрируем имя для вновь созданного процесса: **register(simple\_service, ProcessPid)**. Теперь мы (все еще находясь на узле **n1@stdstring**) можем послать сообщение вновь созданному процессу при помощи зарегистрированного имени: **simple\_service!(simple\_message, message\_data)**. В результате мы увидим следующий вывод на консоль (на узле **n1@stdstring**):

```
Message: {simple_message, message_data}.
```

Это означает, что сообщение созданным нами процессом получено и обработано. Теперь попробуем на узле **n2@stdstring** обратиться к созданному процессу по зарегистрированному имени: **simple\_service!(simple\_message, message\_data)**. В результате мы получим ошибку времени выполнения, т.к. зарегистрированные имена являются локальными относительно узла. Попробуем обратиться к созданному процессу по зарегистрированному имени еще раз, но используя специальный синтаксис: **{simple\_service,**

**'n1@stdstring'!{simple\_message, message\_data}**. В результате на узле **n2@stdstring** выполнение этого выражения будет успешно, а в консоли на узле **n1@stdstring** мы увидим следующий вывод:

```
Message: {simple_message, message_data}.
```

Как мы увидели, несмотря на то, что регистрация имен локальная для каждого узла, мы можем использовать зарегистрированные имена на любых узлах при помощи специального синтаксиса. Но, во-первых, этот синтаксис не очень удобен, т.к. вынуждает задавать два имени вместо одного, а во-вторых (и это достаточно серьезное ограничение), этот синтаксис раскрывает детали внутренней реализации системы: на каком узле и с каким именем располагается сервис. Чтобы преодолеть эти недостатки, нам нужен механизм, позволяющий регистрировать имена глобально (на уровне используемой сети из узлов, образующих приложение). Для этих целей служит модуль **global**; более подробно о функциях из этого модуля мы поговорим в следующем номере, а пока ограничимся лишь глобальной регистрацией имен. В модуле **global** определены следующие функции (полный аналог функций, регистрирующих имена процессов локально): **global:register\_name/2**, **global:unregister\_name/1**, **global:whereis\_name/1**. Но при использовании этих функций существуют и некоторые различия: во-первых, узел, желающий воспользоваться глобальным зарегистрированным именем, д.б. известен узлу, который эту регистрацию осуществил; во-вторых, для отсылки сообщения при помощи глобального зарегистрированного имени следует использовать функцию **global:send/2**. Обычная функция (BIF) **send/2** и оператор **!** с глобальными зарегистрированными именами не работают. Наш пример в этом случае будет выглядеть так. На узле **n1@stdstring** мы вводим команды

```
ProcessPid = spawn(fun simple_message_handler:
```

```
message_loop/0).
```

```
global:register_name(simple_service, ProcessPid).
```

А на узле **n2@stdstring** мы вводим команды

```
net_adm:ping('n1@stdstring').
```

```
global:send(simple_service, {simple_message, message_data}).
```

В результате на узле **n1@stdstring** на консоль дважды выводится следующая строка:

```
Message: {simple_message, message_data}.
```

Сегодня мы рассмотрели средства для построения распределенных приложений на языке Erlang. Мы увидели, что специфика построения распределенных приложений в основном касается инфраструктуры; основной код приложения при этом особо не меняется, по сравнению с обычным многозадачным кодом. Как следствие, это упрощает написание распределенных приложений (что мы еще увидим на практике). Мы пока не прощаемся с темой, посвященной многозадачности и распределенным приложениям: в следующий раз мы поговорим о том, какие у нас есть библиотечные средства для построения многозадачных и распределенных приложений. [\[x\]](#)

## Ошибка в предыдущем номере

В предыдущем номере в заметках я приводил несколько примеров решения проблемы с очисткой ресурсов на языке C. Второй пример содержит досадную ошибку: везде вместо сравнения на неравенство **!="** написано сравнение на равенство **"=="**. Ниже приводится правильный вариант этого примера:

```
int descr1, descr2;
```

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