Erlang: Изучим

Андрей Ушаков увлечен взаимодействием между процессами.

Наш эксперт

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

Современный мир разработки ПО сложно представить без такого средства, как многозадачность. Многозадачность позволяет решать многие задачи за приемлемое время благодаря тому, что многие части одной задачи выполняются одновременно. Многозадачность – это одна из самых сложных областей программирования. Язык Erlang в свое время был специально разработан для решения большинства технических проблем, связанных с многозадачностью. Эта статья открывает большой цикл, посвященный созданию многозадачных и распределенных приложений средствами языка Erlang.

А начнем мы наш разговор с определения основных терминов, связанных с темой нашего разговора. Первым делом дадим определение термину «многозадачность». Многозадачность – это свойство языка программирования, среды выполнения и операционной системы по обеспечению одновременного (или псевдоодновременного) выполнения нескольких задач. Если язык программирования не является многозадачным, то и беспокоиться нам не о чем: все задачи у нас выполняются последовательно. Если среда выполнения или операционная система не являются многозадачными, то максимум, что мы сможем получить – это иллюзию многозадачности. Эта иллюзия заключается в том, что поток выполнения у нас в реальности всего один; среда выполнения или ОС переключают его на выполнение разных фрагментов кода.

Если и язык программирования, и среда выполнения, и ОС поддерживают многозадачность, то чтобы добиться одновременного выполнения нескольких задач, необходимо выполнить два условия. Во-первых, истинную многозадачность мы можем получить, запуская несколько задач на многопроцессорном или многоядерном компьютере, или на нескольких компьютерах, соединенных сетью. Во-вторых, и это, пожалуй, самое важное, для одновременного выполнения нескольких задач в рамках одной программы программа изначально должна разрабатываться как многозадачная.

Многозадачность бывает двух типов: основанная на процессах и основанная на потоках. Говоря о многозадачности, основанной на потоках, мы имеем в виду, что у нас есть несколько потоков выполнения кода в рамках одного процесса. При такой конфигурации несколько задач одновременно выполняются в одном адресном пространстве; это упрощает взаимодействие между задачами, но при неправильной реализации данного взаимодействия может привести к несогласованности и порче данных.

Многозадачность, основанная на процессах, подразумевает, что у нас есть несколько процессов на одном или нескольких компьютерах. При такой конфигурации у каждой задачи свое адресное пространство, что предотвращает многие случаи возможной порчи состояния одной задачи другой задачей, но усложняет взаимодействие между задачами. В дальнейшем мы будем различать ситуации, когда несколько процессов выполняются на одном компьютере и когда несколько процессов выполняются на разных компьютерах, соединенных сетью. Связано это с тем, что способы взаимодействия между процессами зависят от того, на одном или на разных компьютерах процессы выполняются.

Многозадачность, основанную на процессах, выполняющихся на разных компьютерах, мы будем рассматривать отдельно. Такой вариант многозадачности называют распределенной системой (а саму такую многозадачность – распределенными вычислениями). Главное преимущество распределенных систем перед нераспределенными в том, что их мощность (количество компьютеров, объединенных сетью) можно наращивать бесконечно. Минусы подобных систем в том, что их достаточно сложно создавать и поддерживать их работу (вполне очевидно, что чем больше такая распределенная система, тем это делать сложнее). Кроме того, далеко не всякую задачу имеет смысл решать в распределенной системе; но это тема для отдельной статьи.

Вместо этого давайте рассмотрим класс задач, решение которых в распределенной системе дает существенный выигрыш по сравнению с решением нераспределенным и однозадачным способом. Это задачи по поиску элемента из некоторого множества методом полного перебора (или методом «грубой силы», англ. brute force): мы перебираем все элементы из множества, пока не найдем удовлетворяющий нас элемент.

Очевидно, что если множество элементов упорядочено и существует связь между критерием поиска (близостью элемента к искомому) и порядком элементов, то метод поиска полным перебором в такой ситуации не оптимален. Также вполне очевидно, что если мы разделим множество элементов на N равных подмножеств и начнем поиск в этих подмножествах одновременно, то среднее время поиска будет в N раз меньше среднего времени поиска во всем множестве элементов. Поэтому вполне логично, что такой класс задач хорошо подходит для решения на распределенных системах. К такому классу задач относится, например, задача подбора пароля по хэшу MD5 (хотя для этой задачи существует и альтернативный подход к решению, с использованием радужных таблиц).

Пойдем дальше и остановимся на еще одном понятии, без которого невозможно создавать более-менее сложные многозадачные программы: это взаимодействие между задачами (обычно называемое взаимодействием между процессами или IPC). Средства, применяемые для такого взаимодействия, зависят от вида используемой многозадачности (основанной на потоках, основанной на процессах на одном компьютере или основанной на использовании распределенных вычислений), а также от языка программирования (если такие средства встроены в язык), среды выполнения и ОС. Вполне логично, что средства, применяемые для взаимодействия процессов в распределенной системе, могут быть применены и для взаимодействия процессов на одном компьютере, и для взаимодействия потоков в одном процессе. А средства, применяемые для взаимодействия процессов на одном компьютере, пригодны и для взаимодействия потоков внутри процесса.

Возникает вопрос: зачем нам использовать более общие средства взаимодействия в случае, когда можно применить более подходящие для данного типа многозадачности? Одна из причин такого подхода состоит в том, что, разрабатывая изначально взаимодействие между задачами таким способом, мы получаем возможность использовать наш код для любых видов многозадачности. Кроме того, при таком подходе мы можем избежать или минимизировать возникновение одновременного доступа к данным (точнее говоря, нам надо избегать одновременного доступа

МНОГОЗАДАЧНОСТЬ

к данным, если мы хотим изменять эти данные) и, как следствие, возможного повреждения и несогласованности данных.

Теперь давайте разберемся, какие есть средства для создания многозадачных программ. В качестве примера такого средства рассмотрим ОС Linux. Первый вопрос, который встает перед нами – как создавать новые задачи. Начнем с создания потоков. В Linux для работы с потоками у нас есть библиотека *pthreads* (интересно, что потоки в Linux – это процессы, разделяющие ресурсы с процессом, который их создал); для создания новых потоков используется функция **pthread_create**. Перейдем к созданию процессов. Для создания процессов у нас есть следующие библиотечные функции: функция **fork** для создания нового процесса, семейство функций **exec** для запуска в рамках процесса другой программы, функция **system** для выполнения команд. Следует сказать, что создавать новые процессы мы можем только на локальном компьютере; возможности создать новый процесс (и запустить в нем какую-либо программу) у нас нет. Поэтому при построении распределенной системы необходимо предусмотреть автоматический старт (при старте системы) некоторого процесса для взаимодействия узлов этой распределенной системы.

Пришла пора посмотреть, какие есть средства взаимодействия между задачами в ОС Linux. Начнем с многозадачности, основанной на потоках; для взаимодействия задач у нас есть мьютексы, блокировки чтения-записи, условные переменные, возможность ожидания завершения потока. Все эти средства основаны на использовании блокировок потока выполнения. Множество средств для многозадачности, основанной на локальных процессах, достаточно разнообразно: именованные и неименованные каналы, файлы, очереди сообщений, семафоры, разделяемая память, сокеты домена Unix, возможность ожидания завершения процесса. Если для взаимодействия процессов мы используем разделяемую память или файлы, то при неаккуратной работе с данными (например, когда несколько процессов одновременно пытаются эти данные изменить) данные могут испортиться; в этом случае необходимо использовать дополнительные средства синхронизации, такие как блокировка записей в файле, семафоры, обмен сообщениями. И, наконец, для многозадачности в распределенной среде у нас есть лишь одно средство взаимодействия между задачами – сокеты (правда, конфигурация сокетов может быть разнообразной – например, сокеты с установлением соединения и без).

После общего обзора, что такое многозадачность, и обзора средств для создания многозадачных приложений на примере конкретной платформы мы можем перейти непосредственно к основной теме статьи – многозадачности в языке Erlang. Язык Erlang изначально был разработан для создания многозадачных и распределенных приложений, поэтому средства для создания задач и взаимодействия между ними имеют поддержку на уровне языка (в чем очень скоро мы убедимся).

В отличие от большинства языков и платформ, в языке Erlang существует только один тип задач, называемый процессами. Задачи в языке Erlang не являются истинными процессами, т. к. не обладают собственным адресным пространством. Это означает, что несколько процессов могут выполняться в пределах одного экземпляра среды выполнения Erlang (на одной виртуальной машине Erlang). С другой стороны – и это роднит процессы Erlang с истинными процессами – процессы Erlang изолированы друг от друга (в отличие от, например, потоков в случае многозадачности, основанной на потоках). Изолированность процессов друг от друга заключается в следующих аспектах. Во-первых, данные одного процесса не доступны никому, если этого не пожелал сам процесс. Во-вторых, если во время работы возникает необрабатываемое исключение, то процесс Erlang будет завершен (так же как и некоторые связанные с этим процессом процессы; но об этом – на одном из следующих уроков). При этом завершение сбойного процесса Erlang не коснется других (не связанных с ним) процессов, вне зависимости от того, выполняются ли они в этом же экземпляре среды выполнения Erlang или нет. Такое поведение отличается от поведения при возникновении необрабатываемого исключения в одном из потоков выполнения какого-либо процесса: в этом случае обычно завершается процесс целиком, в том числе и другие потоки выполнения.

Процессы Erlang легковесны, их создание и завершение достаточно дешево (по сравнению, например, с созданием процессов и потоков в Linux), поэтому вопрос о том, использовать или нет многозадачность при решении той или иной задачи, должен »

Типы псевдопараллельной многозадачности

» **Невытесняющая многозадачность** Тип многозадачности, при котором операционная система может загрузить более одного приложения в память, но время процессора предоставляется только одному из них – основному приложению. Остальные приложения являются фоновыми. Для предоставления процессорного времени фоновому приложению его необходимо активизировать. Подобная многозадачность может быть реализована не только в ОС, но и с помощью программ – переключателей задач.

» **Кооперативная многозадачность** Тип многозадачности, при котором следующая задача выполняется только после того, как текущая задача явно провозгласит себя готовой отдать процессорное время другим задачам. Как частный случай, такое объявление подразумевается при попытке захвата уже занятого объекта блокировки, а также при ожидании поступления следующего сообщения от подсистемы пользовательского интерфейса.

» **Вытесняющая многозадачность** Вид многозадачности, в котором ОС сама передает управление от одной выполняемой программы другой в случае завершения операций ввода-вывода, возникновения событий в аппаратуре компьютера, истечения таймеров и квантов времени или же поступлений тех или иных сигналов от одной программы к другой. В этом виде многозадачности процессор может быть переключен с исполнения одной программы на исполнение другой без всякого пожелания первой программы и буквально между любыми двумя инструкциями в ее коде. Распределение процессорного времени осуществляется планировщиком процессов. К тому же каждой задаче может быть назначен пользователем или самой операционной системой определенный приоритет, что обеспечивает гибкое управление распределением процессорного времени между задачами (например, можно снизить приоритет ресурсоемкой программе, снизив тем самым скорость ее работы, но повысив производительность фоновых процессов). При этом обеспечивается более быстрый отклик на действия пользователя.

Проблемы в многозадачных средах

» **Голодание [starvation]** Задержка времени от пробуждения потока до его вызова на процессор, в течение которой он находится в списке потоков, готовых к исполнению. Возникает по причине присутствия потоков с бо́льшими или равными приоритетами, которые исполняются все это время. Негативный эффект заключается в том, что возникает задержка времени от пробуждения потока до исполнения им следующей важной операции, что задерживает исполнение этой операции, а следом за ней и работу многих других компонентов. Голодание создает узкое место в системе и не дает выжать из нее максимальную производительность, ограничиваемую только аппаратно обусловленными узкими местами.

» **Гонки [race condition]** Недетерминированный порядок исполнения двух путей кода, работающих с одними и теми же данными и исполняемыми в двух различных задачах. Приводит к зависимости порядка и правильности исполнения от случайных факторов.

» **Инверсия приоритета** Пусть поток L имеет низкий приоритет, поток M – средний, поток H – высокий. Пусть поток L захватил объект блокировки (например, мьютекс) и, выполняясь с удержанием объекта блокировки, прерывается пробудившимся по какой-то причине потоком M, который имеет более высокий приоритет. Пусть поток H также пытается захватить этот объект блокировки. В такой ситуации поток H ждет завершения текущей работы потоком M, т. к., пока поток M исполняется, низкоприоритетный поток L не получает управления и не может освободить захваченный объект блокировки.

решаться только исходя из факта, можно ли распараллелить алгоритм задачи или нет. Конечно, следует сказать, что возможны ситуации, когда применение многозадачного решения будет неоправданно (когда, например, необходимо нескольким задачам работать одновременно с большим набором данных, расположенных в памяти), но этому мы посвятим один из последующих уроков.

Как уже говорилось, в языке Erlang у нас есть всего один тип задач на все случаи жизни – процессы Erlang. Мы уже поняли, что при желании использовать многозадачность в рамках одного экземпляра среды выполнения Erlang достаточно создать необходимое количество процессов Erlang. Но что делать, если мы хотим создать процессы Erlang на нескольких экземплярах среды выполнения Erlang – как на локальной машине, так и в случае распределенной системы? Для понимания этого, давайте введем понятие узла: узел – это именованный экземпляр среды выполнения Erlang. Для создания узла достаточно при ее запуске задать короткое (при помощи ключа **-sname**) или длинное (при помощи ключа **-name**) имя узла (этими ключами и разницей между ними мы более подробно займемся на одном из следующих уроков). После того, как узел с определенным именем создан, мы можем создать на нем необходимое нам количество процессов с некоторого другого узла. При этом создание процесса на каком-либо узле практически ничем не отличается от создания процесса в локальной среде выполнения Erlang: для создания процесса на узле нам необходимо лишь указать его имя, дополнив рядом параметров.

Может показаться, что такая возможность удаленного создания процессов – это большая дыра в системе безопасности. На самом деле это не так: средствами среды выполнения Erlang мы можем управлять тем, кто и какой код может запускать на удаленном узле, но об этом мы поговорим на одном из следующих уроков.

Для понимания того, как создавать многозадачные программы в языке Erlang, нам осталось поговорить еще на одну тему – взаимодействие между задачами. Взаимодействие между задачами в языке Erlang осуществляется при помощи обмена сообщений. Процесс может послать любому известному ему процессу любое сообщение – любой объект языка Erlang. Что означает, что один процесс знает о другом процессе? Это означает, что у процесса есть идентификатор (называемый **Pid**) другого процесса, либо другой процесс зарегистрировал свое имя, которое знает данный процесс. Идентификатор другого процесса мы можем знать, если мы сами его создали либо этот другой процесс послал нам сообщение, содержащее его идентификатор (каждый процесс может легко узнать свой идентификатор). Сообщение от одного процесса другому посылается асинхронно, т. е. мы не блокируем процесс-отправитель, пока это сообщение не будет доставлено процессу-получателю. Посылка сообщения от одного процесса другому процессу является надежной; это означает, что среда выполнения Erlang гарантирует доставку сообщения получателю.

Давайте посмотрим, как у нас обстоят дела со стороны получения сообщений. Каждый процесс в языке Erlang имеет связанную с ним очередь сообщений. Как только сообщение доставляется средой выполнения Erlang до процесса, это сообщение размещается в конце очереди сообщений данного процесса. Когда процесс пытается получить сообщение, удовлетворяющее некоторым критериям, то он просматривает последовательно свою очередь сообщений и извлекает из нее первое удовлетворяющее критериям отбора сообщение. Если удовлетворяющего критериям отбора сообщения нет, то процесс, инициировавший получение, переходит в состояние ожидания. Это ожидание длится до тех пор, пока среда выполнения Erlang не доставит процессу какое-либо новое сообщение. Как только новое сообщение будет доставлено, процесс поиска сообщения, удовлетворяющего критериям отбора, будет запущен заново. Понятно, что процесс может бесконечно ожидать удовлетворяющее его критериям сообщение; однако мы можем повлиять на максимальное время ожидания, задав его в выражении получения сообщения. Обмен сообщениями между процессами, несмотря на унификацию многозадачности в языке Erlang, не единственное средство для взаимодействия между задачами. Так как, помимо общения друг с другом, задачи предназначены и для взаимодействия с внешним миром, то в наши руки попадают такие средства, как сокеты, файлы и т. д.

После обсуждения принципов реализации многозадачности в языке Erlang пришла пора посмотреть на конкретные средства ее реализации. Для создания новых процессов у нас есть целое семейство BIF: **spawn/1**, **spawn/2**, **spawn/3**, **spawn/4**. Функции **spawn/1** и **spawn/3** предназначены для создания новых процессов на локальном экземпляре среды выполнения, функции **spawn/2** и **spawn/4** – на удаленном узле. Когда мы создаем новый процесс (одной из этих функций), мы должны указать задачу, которую этот процесс будет выполнять. Задачей всегда является некоторая функция. Функции **spawn/1** и **spawn/2** задают выполняемую задачу в виде ссылки на произвольную функцию. Функции **spawn/3** и **spawn/4** задают выполняемую задачу в виде **MFA**, где **M** – это модуль, **F** – некоторая экспортируемая из модуля **M** функция, **A** – список аргументов, передаваемых данной функции. Разница между этими подходами при задании выполняемой задачи состоит в том, что при помощи ссылки на функцию мы можем задать как экспортируемую функцию, так и анонимную функцию и не экспортируемую функцию из текущего модуля; при помощи подхода **MFA** мы всегда задаем экспортируемую функцию.

Для создания новых процессов у нас также имеется семейство BIF **spawn_link**. Разница между этим семейством функций и семейством функций **spawn** в том, что семейство функций **spawn_link** не только создает новый процесс, но и создает связь между новым процессом и текущим (разговор о связи между процессами тоже ждет нас в будущем). Все функции обоих семейств возвращают идентификатор процесса или **Pid**.

Для отправки сообщения от одного процесса другому процессу мы используем выражение **Process! Message**, где **Process** – выражение, определяющий целевой процесс (значением этого

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

выражения является либо идентификатор процесса, либо имя зарегистрированного процесса), а **Message** – отправляемое сообщение (любой объект языка Erlang). Результатом этого выражения отправки сообщения является значение самого отправляемого сообщения **Message**. Помимо этого выражения отправки сообщений, существует еще целое семейство функций, определенных в модуле **erlang** для данной задачи. К этому семейству относятся функции **erlang:send/2**, **erlang:send/3**, **erlang:send_after/3**, **erlang:send_nosuspend/2**, **erlang:send_nosuspend/3**. Эти функции (за исключением функции **erlang:send/2**, которая идентична выражению отправки сообщения) позволяют настроить процесс отправки сообщений. Для решения задачи отправки сообщения у нас есть несколько возможных вариантов, а для получения сообщения только один – выражение **receive**. Выражение для получения сообщений **receive** имеет следующий вид:

```
receive
  Pattern1 [when GuardSeq1] -> Body1;
  ...;
  PatternN [when GuardSeqN] -> BodyN
[after ExprT -> BodyT]
end
```

В этом выражении получения сообщения каждое подвыражение вида **$Pattern_i$ [when $GuardSeq_i$]** определяет критерий, по которому то или иное сообщение из очереди сообщений процесса может быть выбрано для обработки. В подвыражении **$Pattern_i$ [when $GuardSeq_i$]** часть **$Pattern_i$** является выражением соответствия шаблону [pattern-matching], а часть **when $GuardSeq_i$** является выражением охраны. Если будет найдено сообщение, удовлетворяющее одному из критериев **$Pattern_i$ [when $GuardSeq_i$]**, то значением выражения **receive** будет подвыражение **$Body_i$**, соответствующее подвыражению критерия выбора. Выражение **after ExprT -> BodyT** задает поведение, если в очереди сообщений не оказалось подходящего сообщения: в этом случае выполнение процесса приостанавливается максимум на **ExprT** миллисекунд. Если в течение **ExprT** миллисекунд в очереди сообщений процесса не появится сообщения, удовлетворяющего перечисленным выше критериям, то процесс закончит ожидание подходящего сообщения, и значением выражения **receive** будет подвыражение **BodyT**. Значением подвыражения **ExprT** также может быть атом **infinity**; в этом случае процесс будет ждать сообщения, удовлетворяющего одному из критериев, бесконечно долго. Выражение **after ExprT -> BodyT** не является обязательным; если оно отсутствует в выражении **receive**, то это эквивалентно случаю бесконечного ожидания подходящего сообщения процессом.

А теперь пришла пора практики: давайте на простом примере посмотрим, как работать с многозадачностью в языке Erlang. В этом примере мы из основного процесса создадим рабочий процесс и будем взаимодействовать с ним при помощи сообщений. Давайте рассмотрим сначала функцию (выполняемую задачу) рабочего процесса **example_worker/1**:

```
example_worker(Master) ->
  receive
    {ping, From } ->
      io:format("ping from master ~n"),
      From ! {pong, self()},
      example_worker(Master);
    _Other ->
      io:format("unknown message ~n"),
      example_worker(Master)
  after 1000 ->
    io:format("timeout ~n"),
    erlang:send(Master, timeout)
  end.
```

Эта функция содержит выражение для получения сообщений **receive**. Выражение **receive** умеет получать сообщения вида **{ping, From}**, где **From** – любой объект языка Erlang (мы надеемся, что **From** – это идентификатор процесса отправителя, но при необходимости это можно проверить явно при помощи BIF **is_pid/1**) и все остальные сообщения. Обычно все остальные сообщения (т.е. сообщения, не подпадающие ни под один другой критерий) получают для того, чтобы эти «мусорные» сообщения не накапливались в очереди сообщения процесса. Кроме того, выражение **receive** содержит секцию **after**, которая задает максимальное время ожидания в 1000 миллисекунд.

При получении любого сообщения мы выводим информацию на экран и, при помощи хвостовой рекурсии, зацикливаем выполнение функции **example_worker/1**. Таким образом, функция **example_worker/1** является функцией обработки сообщений. Кроме того, при получении сообщения **{ping, From}** мы отсылаем процессу, пославшему нам это сообщение, сообщение **{pong, self()}**. BIF **self/0** возвращает идентификатор текущего процесса, но у нас нет никакого способа узнать идентификатор процесса, пославшего нам сообщение, кроме как включить этот идентификатор в само сообщение. Именно этого мы ожидаем при получении сообщения **{ping, From}**, и именно это мы делаем, когда отсылаем сообщение **{pong, self()}**.

И, наконец, при наступлении таймаута в выражении receive мы отсылаем сообщение **timeout** главному процессу при помощи функции **erlang:send/2** (для демонстрации такой возможности) и обрываем цикл обработки сообщений.

Теперь давайте взглянем на функцию главного процесса **example/0**:

```
example() ->
  Master = self(),
  Worker = spawn(fun() -> example_worker(Master) end),
  Worker ! hello,
  Worker ! {ping, Master},
  Worker ! hi,
  receive
    timeout -> io:format("timeout in worker ~n")
  end,
  receive
    {pong, Worker} -> io:format("pong from worker ~n")
  end.
```

В этой функции мы создаем рабочий процесс (при помощи BIF **spawn/1**, которая возвращает идентификатор созданного процесса), после чего посылаем рабочему процессу ряд сообщений. Среди этих сообщений находится как известное рабочему процессу сообщение **{ping, Master}**, так и неизвестные **hello** и **hi**. После этого, при помощи выражений **receive**, мы принимаем сообщения от рабочего потока. Заметьте, что принимать сообщения мы можем в любом порядке, а не в том, в котором они были посланы. Если мы запустим на выполнение функцию **example/0** (создав модуль с экспортируемой функцией **example/0**), то мы получим следующий вывод в консоли среды выполнения Erlang:

```
unknown message
ping from master
unknown message
timeout
timeout from worker
pong from worker
```

Мы сегодня сделали первый, но очень важный шаг в сторону многозадачности в языке Erlang. Написание многозадачных программ – достаточно сложная область, как в плане алгоритмов, так и в плане технических средств. Язык Erlang позволяет снять сложность технического плана и позволяет нам сосредоточиться только на соответствующих многозадачных алгоритмах. Мы это увидели сегодня (пускай это было не очень очевидно), мы это увидим еще не раз. А на следующем уроке мы поговорим о средствах для создания устойчивых к ошибкам программ. LXF