ПРОГРАМНО-ТЕХНІЧНІ КОМПЛЕКСИ

В.О. ЛАРИН, О.В. БАНТЫШ, А.В. ГАЛКИН, А.И. ПРОВОТАР

УДК 004.434:004.75

ПРЕДМЕТНО-ОРИЕНТИРОВАННЫЙ ЯЗЫК STRUMOK ДЛЯ ОПИСАНИЯ АКТОРНЫХ СИСТЕМ С ОБЩЕЙ ПАМЯТЬЮ

Аннотация. Предложен предметно-ориентированный язык Strumok для описания межакторного взаимодействия и работы с общей памятью. Проведена оценка эффективности языка Strumok с использованием общей памяти по сравнению с фреймворком Java Vert.x.

Ключевые слова: модель акторов, параллельные системы с общей памятью, предметно-ориентированный язык программирования, Akka, Java.

ВВЕДЕНИЕ

Современный процесс разработки параллельных и распределенных систем нетривиальный и ресурсоемкий. Для упрощения систем формируют различные концептуальные подходы и пакеты инструментария. Для преодоления барьера сложности проектирования предложен, например, подход с использованием отдельных функциональных сущностей — модель акторов. Наиболее совершенным на данный момент средством для разработки и развертывания подобных систем является программный инструментарий Akka. Однако он также имеет определенные недостатки: описание взаимодействия и поведения акторов в нем низкоуровневое. Как следствие, для систем со многими акторами и соответственно сценариями общения между ними текст описания становится неочевидным и сложно поддерживаемым. Кроме того, в акторах нет встроенного механизма композиции и они имеют ограниченные возможности статической типизации сообщений. Основная проблема акторов (но при этом и преимущество) — надежная изоляция их внутреннего состояния, которая приводит к запрету на общую память. На практике, в реальных задачах, часто возникает оправданная необходимость в общих данных — как неизменяемых в рамках акторной модели, так и с возможностью произвольного изменения акторами. Для преодоления указанных недостатков сформулирована концепция предметно-ориентированного языка Strumok, которая с помощью высокоуровневых примитивов межакторного взаимодействия и концепта общих межакторных данных потенциально упрощает создание сложных акторных систем.

АКТОРНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ И AKKA

Цель работы — определение семантики и синтаксиса предметно-ориентированного языка акторного программирования Strumok и основных принципов его трансляции в совместимый с Akka исходный Java-код, а также проверка эффективности полученных результатов в сравнении с другими решениями.

Модель акторов представляет собой математическую модель, трактующую понятие «актор» как универсальный примитив для параллельных вычислений: в ответ на сообщения, которые он получает, актор может изменить свое локаль-

ное состояние, создать новые акторы и отправлять сообщения тем акторам, которых он знает. Модель акторов впервые была описана в 1973 г. Она использовалась как основа для понимания вычисления процессов и теоретическая база для ряда практических реализаций параллельных систем [1].

Развязка отправителя и получателя отправленных сообщений стала фундаментальным достижением модели акторов. Она обеспечила асинхронную связь и управление структурами как прототип передачи сообщений [2].

Akka — фреймворк на JVM, полноценно реализующий модель акторов [3]. Акторы в Akka выполняются в рамках так называемых «легких» потоков, что дает следующие преимущества:

- акторы защищены от вмешательства других акторов;
- отсутствие различных блокирующих механизмов при реализации логики акторов;
- за счет малой ресурсоемкости возможно параллельное выполнение большого количества акторов.

Одной из важных особенностей Akka является реализация так называемой системы супервизора (supervising). При возникновении нештатных ситуаций супервизор в соответствии с указанной стратегией перезагружает актор или останавливает его. Состояние актора также можно автоматически восстановить до состояния перезапуска с сохранением полученных сообщений и их воспроизведением после перезагрузки. Это дает возможность самовосстановления системы. Кроме корневого супервизора, каждый актор может наблюдать за другими акторами, формируя при этом иерархию супервизоров. Родительский актор получает специальное сообщение, включающее информацию об акторе, который непредсказуемо завершил работу. Таким образом, родительский актор может выбрать дальнейшую стратегию действий.

Цель акторов — обработка сообщений. Канал, соединяющий отправителя и получателя, является почтовым ящиком актора (mailbox): каждый актор имеет один почтовый ящик, в который поступают все сообщения. Поступления упорядочиваются по времени отправки. Это означает, что сообщения, отправленные от одного актора к другому, обрабатываются в порядке отправления, но при этом порядок обработки сообщений от разных акторов не гарантирован.

Модель акторов также имеет определенные недостатки. Акторы не компонуются: в модели не предусмотрено прозрачного механизма для компоновки нескольких акторов. Akka только частично решает эту проблему: реализованная иерархия супервизоров и паттерн ask позволяют унифицировать опрос и контроль дочерних акторов, но в то же время отсутствие внешнего интерфейса доступных методов актора усложняет его повторное использование [4].

Кроме того, Akka имеет ограниченные возможности статической (на этапе компиляции) типизации сообщений. Разработчик не может, предварительно не протестировав систему, узнать, все ли связи между акторами согласованные. При усложнении конфигурации их отследить трудно, в то время как проверка типов позволила бы сразу выявлять значительное количество ошибок.

Наиболее значимой особенностью акторов, и Akka в частности, является отсутствие механизма общей межакторной памяти. Для защиты от deadlocks и блокировок в целом внутреннее состояние актора можно представить только упаковкой в сообщении. Эта концепция имеет принципиальный недостаток: когда данные нужны для чтения лишь нескольким акторам, они не имеют возможности их параллельно прочитать, а могут только отправить сообщения, которые будут последовательно обработаны. В случае доступа на запись возникает проблема с универсальным контроллером данных, т.е. инкапсулирующий актор, содержащий данные, должен полностью обрабатывать все сценарии работы с ними. Это усложняет его повторное использование и разносит логику алгоритма по разным акторам, что нежелательно.

Модель акторов имела ограниченное применение до появления системы Akka Framework. Akka отличается высокой производительностью и надежностью (50 млн сообщений в с; ~ 2.7 млн акторов на 1 ГБ ОЗУ) за счет использования «легких» потоков и fail-safe механизма сообщений [3]. Конфигурация системы вынесена за рамки исходного кода в config-файл; кроме того, Akka из коробки поддерживает кластеризацию.

В настоящее время фреймворк Akka широко используется в финансовой сфере, медиа, социальных и развлекательных сервисах. Однако, как отмечено выше, она имеет определенные недостатки.

Для эффективного решения описанных проблем разработан предметно-ориентированный язык Strumok.

РЕАЛИЗАЦИЯ МОДЕЛИ АКТОРОВ В STRUMOK

Предметно-ориентированный язык (DSL) — специализированный язык программирования, применяемый в конкретной предметной области. Построение такого языка или структуры данных ориентировано на специфику отрасли, к которой относится задача. Выделим некоторые его особенности по сравнению с обычными языками:

- абстракции DSL обеспечивают определение концепций и абстрактных понятий в предметной области;
- проверка описания в DSL требует статических анализаторов, выявляющих больше ошибок, чем анализаторы общего назначения, и выдающих сообщение о них на этом же языке, что понятнее специалистам предметной области;
- оптимизация кода в DSL базируется на знаниях, недоступных компиляторам языков общего назначения.

По результатам исследования предметно-ориентированных языков в контексте модели акторов выделим Erlang [5], SALSA [6], E [7], AmbientTalk [8], Kilim [9] и Shacl [10]. Каждый из них предоставляет высокие гарантии безопасности: низкоуровневое состояние гонки исключено в самом дизайне системы. Тем не менее в [11] показано, что 56% программ на Scala использовали модель акторов в нераспределенной конфигурации только для параллелизма, а в 68% приложений модель акторов применялась с другими механизмами параллельных вычислений. Это обусловило необходимость поддержки общих данных на уровне самой модели. Если в классических чистых акторных языках предполагается инкапсулирование данных в отдельном акторе, что лишает возможности их параллельного чтения, то новые решения [12] предлагают введение отдельных концептов для представления общих данных в рамках нераспределенного приложения. Цель языка Strumok — представить полноценную реализацию акторной модели, при этом используя механизм общих данных.

ОСНОВНЫЕ ЭЛЕМЕНТЫ ЯЗЫКА STRUMOK

Основная идея языка заключается в добавлении дополнительного уровня абстракции межакторного взаимодействия. При этом определяются аргументы и типы межакторных операций, что позволило на этапе компиляции проверить соответствие типов, компоновать вызовы различных акторов в методах и гарантировать безопасность общего доступа к данным. Элементы конфигурации (в том числе поддержка routing, supervising) Akka вынесены в аннотации и ключевые слова языка. На основе этой информации создано представление высокоуровневого актора, на базе которого транслятор Strumok формирует реализацию на Java. Благодаря дополнительному уровню абстракции созданные акторы можно использовать в других модулях, без необходимости изменения исходного кода. Для решения проблемы типизации межакторного взаимодействия в Strumok введено понятие акторного метода, предусматривающее указания типов и количества входных и выходных аргументов, на основе которых генерируются классы акторов на Java с динамической проверкой и приведением сообщений к данному типу. Приведем пример простейшей акторной программы:

```
singleton actor Controller {
  Controller() {
       this→ping();
  }
  message ping(out String pong) {
       send "Hello!" :pong;
  }
}
```

Входная точка программы на Strumok — это определения акторов, их экземпляров и общих данных. В примере первой строкой определяется экземпляр актора Controller, существующего в единственном экземпляре. В конструкторе актора Controller отправляем единственное сообщение ping.

Отдельно рассмотрим тип экземпляра, представленный данной конструкцией. Всего вводится четыре типа акторных объявлений. Отметим два наиболее важных типа экземпляров.

Первый тип акторного объявления (тривиальный) **singleton** создает один актор в момент инициализации акторной ссылки. Данный тип ссылки позволяет использовать соотношение один к одному, что удобно при определении глобальных контроллеров или менеджеров выполнения.

Второй тип акторного объявления **group** позволяет создавать набор акторов заданного типа с контролирующим актором, который в соответствии с заданной логикой диспетчеризации сообщений (routing в Akka) пересылает сообщения созданным акторам.

Далее рассмотрим объявление актора Controller. После ключевого слова actor и названия следует перечень всех методов и полей актора. В приведенном примере показан конструктор без параметров актора Controller (автоматически вызываемый при создании экземпляра актора, прямой аналог конструктора класса в Java), который, в свою очередь, асинхронно вызывает обработчик сообщения ping у себя же (ключевое слово this). Заметим, что синтаксис вызова метода актора в Strumok отличается от вызова метода объекта/класса Java использованием пары символов →, а также тем, что вызов акторного метода происходит асинхронно и, как результат, возвращает последовательность (continuation), которую затем можно использовать как для ожидания и определения асинхронного порядка, так и обработки полученных результатов.

Рассмотрим пример акторного метода, или обработчика сообщения:

```
message cycle (int amount, out int res, out string exception)
```

Данный метод определяет обработчик сообщения cycle с входным параметром целочисленного типа amount и двумя выходными результатами: целочисленный res и строковый exception. Для определения исходящего аргумента используется модификатор out. В отличие от метода объекта акторный метод может возвращать произвольное количество результатов в исходящие подписки/потоки (subscriptions).

Тело обработчика сообщений может содержать произвольное количество инструкций, из которых выделим следующие группы: управление потоком асинхронного исполнения, поддержка общих данных и привязка Strumok к Akka и Java.

УПРАВЛЕНИЕ ПОТОКОМ АСИНХРОННОГО ИСПОЛНЕНИЯ В STRUMOK

Для управления порядком асинхронного выполнения в Strumok вводится понятие последовательности (continuation), заимствованное из языка SALSA [6]. Последовательность — это результат произвольной асинхронной операции в Strumok. Каждую последовательность можно использовать в дальнейших инструкциях как в целях синхронизации, так и для применения результата в последующих вычислениях. Рассмотрим пример кода акторного метода:

```
var cont = actorInstance→cycle(10);
await (cont) actorInstance→cycle(11);
```

В приведенном фрагменте вызывается метод cycle у экземпляра актора с идентификатором actorInstance, присваивается полученная последовательность идентификатору cont, а затем ожидается завершение выполнения оператора await. Отметим, что оператор await является лишь вспомогательной конструкцией. Так, например, использование последовательности в качестве аргумента вызова метода актора или объекта автоматически создает точку синхронизации в данном месте:

```
var cont = actorInstance→cycle(10);
actorInstance→cycle(cont.res);
```

Кроме того, последовательности поддерживают в том числе множественную или нелинейную синхронизацию:

```
var contA = actorInstance→cycle(10);
var contB = actorInstance→cycle(11);
actorInstance →cycle(contA.res + contB.res);
```

Для удобства описания длинных линейных последовательностей реализован оператор >>, который переносит полученную последовательность на последующий оператор или блок:

```
actorInstance→cycle(10) >> actorInstance→cycle(token.res);
```

В контексте данного оператора текущая последовательность определяется ключевым словом token. Она эквивалентна определению последовательности с именем token для первого вызова.

Возврат полученных результатов акторного метода выполняется с помощью оператора send. Этот процесс не прерывает дальнейшего выполнения обработчика и может осуществляться произвольное количество раз за один проход. Для указания выходной точки назначения применяется семантика: **<имя исходного аргумента>**. На фрагменте кода показано использование оператора send для возврата результата в исходящий поток result:

```
actorInstance→cycle(10) >> send token.res:result;
```

Возникает вопрос: каким образом провести обработку множественных выходных аргументов? В Strumok данной задаче соответствует оператор асинхронного соответствия on:

```
actorInstance→cycle(10) >> {
  on (r :res) send r :result;
  on (ex :exception) send "Something is wrong!" :exception;
}
```

Оператор on задает асинхронные обработчики исходящих аргументов. Каждый обработчик является полноценным блоком, который может содержать произвольное количество инструкций и захватывать внешние последовательности.

Рассмотрим механизм прерывания дальнейшей обработки сообщения. Прерывание (досрочное или плановое) асинхронной обработки сообщения является ключевым механизмом системы Strumok, поддерживающим целостность программы, благодаря которой возможна сборка мусора, гарантия корректной работы метода, а также реализация механизмов пакетной обработки сообщений. Чистые акторные языки не вводят специальных операторов прерывания. Более универсальные системы, например OrcO [13], поддерживают досрочное завершение асинхронного метода с помощью оператора stop и комбинатора trim, что существенно упрощает написание асинхронного кода. Для прерывания обработки сообщения и всех его незавершенных ветвей в Strumok вводится оператор return:

```
actorInstance→cycle(-1) >> {
  on (r :res) send r :result;
```

```
    on (ex :exception) {
        send "Something is wrong!" :exception;
        return;
    }
}
```

Приведенный выше пример отличается от последнего тем, что в нем прерывается дальнейшая обработка данного сообщения после получения ответного сообщения на исходящий аргумент exception. В таком случае результаты на аргументе res, даже если и поступили, обрабатываться далее не будут.

Заметим, что оператор непрямой подписки subscribe, хотя и не является необходимым для полноты контроля потока исполнения, позволяет «отвязать» получателей сообщения от отправителя, что повышает повторную используемость кода. Применение оператора subscribe представим следующим образом:

```
subscribe actorInstance→cycle on (r :res) send r :result;
```

В данном фрагменте продемонстрирована привязка на возвращаемые результаты метода cycle, экземпляра actorInstance и последующий возврат результата.

Таким образом, Strumok формирует полный набор операторов управления асинхронным потоком исполнения.

ОБЩИЕ ДАННЫЕ В ЯЗЫКЕ STRUMOK

Как отмечено ранее, отсутствие общей памяти с параллельным доступом является препятствием для использования чистой акторной модели в прикладных задачах. Для объявления общей переменной в Strumok предусмотрен модификатор типа shared:

```
shared int counter = 0;
```

С помощью данного модификатора можно объявить произвольную общую переменную в области видимости объемлющего актора. Требование к типу переменной — реализация интерфейса serializable или использование базового типа Java. Общую переменную можно применить в любом месте акторного кода. Система при чтении такой переменной автоматически определит, на одном ли узле кластера размещены акторы: если да, то выполнит параллельное чтение, если нет, то отправит сообщение с запросом значения общей переменной. Для записи значения будет также отправлено сообщение с командой на изменение значения. Для обеспечения синхронизации и запрета состояния гонки при записи общей переменной в блоке из нескольких команд Strumok переносит контекст вызова всего синхронного блока изменения переменной в анонимный служебный актор общей переменной. Таким образом, все блоки изменения общей переменной последовательно обрабатываются анонимным служебным актором, что обеспечивает потокобезопасность.

ПРИВЯЗКА STRUMOK К AKKA И JAVA

Поскольку на выходе препроцессора Strumok генерируется валидный Java-код, в котором все сущности реализованы с использованием Akka, важным аспектом является обеспечение совместимости Strumok и Akka, а также возможность вызова разработчиком сторонних Java методов.

Вызов стороннего Java метода возможен непосредственно в коде на Strumok:

```
f.someMethod("abc", instance);
```

Для объявления дополнительных переменных внутреннего состояния и логики из Java можно воспользоваться импортом статических методов, объявлением readonly переменных в акторах или локальных переменных в обработчиках сообщений:

```
readonly MessageGenerator gen = new MessageGenerator()
message init() {
    this→executeTasks(gen.generate());
}
```

Относительно привязки Strumok к Akka выделим две области: конфигурация и функциональность. С точки зрения функциональности Strumok полностью реализует все базовые возможности Akka, в том числе поддержку супервизоров. Для задания стратегии обработки нештатной ситуации введен тип элемента актора catch, в котором на каждый тип исключения задается своя стратегия:

```
catch (IllegalFormatException ex) { restart; }
catch (MyCustomException ex) {
  if (ex.isFatal) {
      escalate;
  } else {
      resume;
  }
}
```

Таким образом, реализуется механизм fail-safe обработки сообщений Akka. Одним из преимуществ Akka является конфигурация. Сгенерированные с помощью Strumok акторы поддерживают конфигурирование как через стандартный конфигурационный файл Akka, так и с помощью специальных аннотаций AkkaRoutedPool, AkkaRoutedGroup, AkkaActor:

```
@AkkaRoutedPool (strategy = "RoundRobin", amount = 4)
group Producer producers;
```

В данном фрагменте объявлена группа акторов producers и в качестве их конфигурации задан пул из четырех элементов с типом диспетчеризации Round Robin (каждый актор выбирается по очереди по кругу). Таким же образом можно задать произвольный тип диспетчеризации, количества объектов и другие специфичные параметры Akka.

ПОДДЕРЖКА РАСПРЕДЕЛЕННЫХ ВЫЧИСЛЕНИЙ В STRUMOK

Поскольку наиболее эффективно модель акторов работает в распределенных приложениях, важным аспектом является поддержка кластеризации. Akka из коробки поддерживает кластеризацию. Используя ранее приведенные аннотации, покажем кластерное расположение для любого экземпляра актора:

```
@AkkaActor (location = "127.0.0.1:5302")
singleton Controller controller;
```

С помощью @AkkaActor или параметра location в аннотациях AkkaRoutedPool, AkkaRoutedGroup для каждого экземпляра актора можно указать его конфигурацию в Akka-кластере. В дальнейшем данную конфигурацию возможно изменить, отредактировав файл application.conf.

Если рассматривать механизм общих данных в Akka, возникает вопрос, как обеспечить его работу в кластерном режиме. Для этого предназначен специальный параметр акторных аннотаций **domain**, позволяющий группировать акторные экземпляры и общие данные в рамках одного узла кластера, за счет чего генератор кода транслятора формирует корректные ссылки на общие данные как в рамках одного узла кластера, так и между разными узлами:

```
@AkkaShared (domain = "worker")
shared bool completed = true;
@AkkaShared (domain = "worker")
shared int counter = 0;
@AkkaShared (domain ="worker")
singleton Controller contrl;
group Consumer consumers;
```

Пример группирования домена общих данных узла кластера показывает, как объединить несколько общих переменных с экземпляром актора. С одной стороны, актор Controller будет использовать специальную транслированную версию с прямым доступом к переменным completed и counter, с другой — consumers как

представители группы акторов Consumer будут внутренне получать доступ к counter и completed через служебные акторы-посредники.

СРАВНЕНИЕ ПРОИЗВОДИТЕЛЬНОСТИ STRUMOK И VERT.x НА ПРИМЕРЕ РАБОТЫ С ОБЩИМ СЧЕТЧИКОМ

Как описано выше, Strumok представляет механизм общих данных для акторной модели. Для оценки эффективности реализации механизма общих данных системы Strumok выполнен замер затраченного времени на решение задачи генерации N сообщений L акторами, с условием, что каждый актор генерирует не более K сообщений в секунду. В качестве аналога для сравнения использован асинхронный фреймворк Vert.x [14], реализующий парадигму асинхронной событийной шины с возможностью подписки на произвольные события, что в приближении данной задачи совпадает с возможностями представленной модели акторов. Отметим, что инструментарий Vert.x поддерживает механизм общих данных в системе.

Приведем исходный код Strumok для решения задачи со счетчиком:

```
import static Controller.*;
import static Range.create;

actor Consumer {
   readonly ControllerImpl impl;

   Consumer () {
      subscribe producers→produce
         on (t :task) impl.collect(t);
   }
}

actor Producer {
   readonly ProducerImpl impl;

   message produce(out Task task) {
      var tmr = Timer→wait(1000);
      var gen = for (int i:create(0, N)) {
         if (counter < K)) {
            counter++;
            send impl.gen(counter) :task;
         }
         else {
            completed = true;
            return;
         }
      }

      await (gen, tmr) {
         if (!completed) {
            await this→produce();
         }
      }
   }
}
```

```
singleton actor Controller {

   @AkkaShared(domain = "w1")
   shared bool completed = true;
   @AkkaShared(domain = "w1")
   shared int counter = 0;

   @AkkaRoutedPool(size = 4, domain = "w1")
   group Producer producers;

   @AkkaRoutedPool(size = 4, domai n ="w1")
   group Consumer consumers;

   readonly ControllerImpl impl;

   Controller () {
      this→executeTasks();
   }

   message executeTasks(out string
                           notify) {

      var prod = broadcast
               producers→produce();

      await (prod) {
         if (completed) {
            send "OK!" :notify;
         }
         else {
            send "Smth is :(" :notify;
         }
      }
   }
}
```

Данный пример демонстрирует решение рассматриваемой задачи. В нем определяется три типа акторов: Controller, Producer и Consumer. Актор Controller содержит конструктор и метод executeTasks. Конструктор без параметров, используемый по умолчанию при создании экземпляра актора, вызывает метод executeTasks, который с помощью оператора broadcast вызывает метод produce группы экземпляров актора Producer и ожидает завершения работы метода у всех акторов с помощью оператора await. Проверяя состояние флага, актор может установить, корректно ли выполнен метод. В свою очередь, метод produce запускает таймер на одну секунду, задавая полученную последовательность идентификатору tmr, а затем с помощью асинхронного цикла for и встроенной функции

Таблица 1

Размер задачи	Vert.x	Strumok
K = 10 , N = 1000	26.021	24.008
K = 1000 , N = 10 000	3.142	2.262
K = 5000 , N = 100 000	6.798	4.165

Range.create генерирует последовательность целых чисел от одного до N, где для каждого шага выполняется проверка на превышение размера сгенерированных сообщений (counter < K). Если ограничение еще не выполнено, значение таймера увеличивается на единицу и сгенерированное задание возвращается, иначе флагу completed выставляется значение «истина» и обработчик сообщения прерывается оператором return. Далее в блоке await выполняется ожидание завершения работы таймера, генератора сообщений и перезапуск сообщения produce для последующего такта с помощью ключевого слова this для обращения к собственному экземпляру. Актор consumer подписывается на результат метода produce не напрямую, а с помощью привязки subscribe. Каждое полученное задание он обрабатывает внутренним методом collect. Отметим также размещение экземпляров акторов Controller и Producer в одном домене w1 с общими переменными counter и completed, что позволяет эффективно использовать механизм параллельного прямого чтения.

С фиксированным параметром L = 4 и характеристиками стендовой машины: Intel Core i7 3770K 3.5GHz, 16GB 2133 MHz RAM, 128 GB SSD на десяти тестовых прогонах получены усредненные результаты. В табл. 1 приведены результаты сравнения скорости (в секундах) решения задачи со счетчиком; каждая строка таблицы соответствует времени решения при определенных размерах входных данных.

Полученные результаты показывают эффективность реализации акторной модели в Akka, а также трансляции программы на Strumok в Java-код. За счет предварительной трансляции возможно добиться максимального использования статических классов сообщений вместо более громоздких замыканий для лямбда-выражений в Vert.x, что также приводит к выигрышу в скорости. При этом реализация общего счетчика в Strumok не уступает по своей эффективности Vert.x и позволяет не терять в скорости.

ДРУГИЕ ПОДХОДЫ ПРИ РЕШЕНИИ ПРОБЛЕМЫ ОБЩЕЙ ПАМЯТИ В ПАРАЛЛЕЛЬНЫХ И РАСПРЕДЕЛЕННЫХ СИСТЕМАХ

В процессе работы над языком Strumok были рассмотрены другие варианты решения проблемы сложности асинхронного взаимодействия и общей памяти.

Концепция асинхронного вызова сообщений и их ожиданий воплощена в await/async и реализована в классических объектно-ориентированных языках (например C#) [15]. Хотя данный подход не предоставляет всех возможностей акторной модели и не отражает истинной асинхронности методов объекта, он хорошо себя зарекомендовал при решении прикладных интерактивных задач (пользовательские интерфейс, веб-сервисы и т.д.).

Возможный вариант решения проблемы непрозрачности композиции акторов и их взаимодействия — использование визуального подхода [16]. Такой подход обеспечивает высокую прозрачность при правильном использовании, но при этом построение диаграмм является более нетривиальной задачей, чем написание программного кода. Кроме того, визуальное программирование на данный момент имеет недостаточно изученные средства для эффективного выражения сложных взаимосвязей.

Для реализации параллельного доступа к состоянию актора предложены параллельные акторные мониторы [17]. Такой подход позволяет использовать ресурсы системы даже при инкапсуляции данных в акторе, но при этом требует от

разработчика вручную и с дополнительными ограничениями разделять независимые области данных в акторе, что в языке Strumok происходит автоматически.

Перспективна предложенная в [12] модель доменов данных. Домены — отдельные контейнеры данных, декларирующие определенные правила доступа к себе. Всего определено четыре типа доменов: изолированные, неизменные, наблюдаемые, общие. Каждый тип домена оптимизирует работу согласно указанным правилам, вследствие чего достигается параллельное чтение, а также неблокирующая запись для наблюдаемых (observable) доменов, поскольку только один актор имеет право записи. На данный момент при определенных преимуществах эта модель и язык Shacl, в котором она реализована, в отличие от Strumok не поддерживают распределенность.

ЗАКЛЮЧЕНИЕ

Рассмотрены основы и принципы модели акторов, а также фреймворк Akka, который ее реализует. Отмечены их недостатки, в частности отсутствие возможностей композиции акторов, механизмов общей межакторной памяти с параллельным доступом, статической типизации и интерфейса обрабатываемых сообщений. Предложен предметно-ориентированный акторный язык Strumok, в котором нет указанных ограничений. Описаны его основные элементы и выполнено сравнение с функциональным аналогом.

СПИСОК ЛИТЕРАТУРЫ

1. Mitchell J.C. Concepts in programming languages. Cambridge: University Press, 2002. 441 p.
2. Hewitt C. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*. 1977. Vol. 8, Iss. 3. P. 323–364.
3. Typesafe Inc. Akka. 2017. URL: http://akka.io/.
4. Welsh N. Why I don't like Akka actors. 2013. URL: http://noelwelsh.com/programming/2013/03/04/why-i-dont-like-akka-actors/.
5. Armstrong J., Virding R., Wikstrom C., Williams M. Concurrent programming in ERLANG. 2nd ed. Hertfordshire: Prentice Hall International (UK) Ltd, 1996. P. 68–77.
6. Varela C., Agha G. Programming dynamically reconfigurable open systems with SALSA. *SIGPLAN Notices*. 2001. Vol. 36, N 12. P. 20–34.
7. Miller M.S., Tribble E.D., Shapiro J. Concurrency among strangers: Programming in E as plan coordination. *Proc. of the 1st Intern. Conf. on Trustworthy Global Computing* (*TGC'05*). Berlin; Heidelberg: Springer-Verlag, 2005. P. 195–229.
8. Van Cutsem T., Mostinckx S., Gonzalez Boix E., Dedecker J., De Meuter W. AmbientTalk: Object-oriented event-driven programming in mobile ad hoc networks. *Proc. of the 26th Intern. Conf. of the Chilean Society of Computer Science* (*SCCC'07*). *IEEE Computer Society.* Washington, 2007. P. 3–12.
9. Srinivasan S., Mycroft A. Kilim: Isolation-typed actors for Java. *Proceedings of the 22nd European Conference on Object-Oriented Programming* (*ECOOP'08*). Berlin; Heidelberg: Springer-Verlag, 2008. P. 104–128.
10. De Koster J. The shacl programming language. URL: http://soft.vub.ac.be/~{}jdekoste/shacl/. 2014.
11. Tasharofi S., Dinges P., Johnson R.E. Why do scala developers mix the actor model with other concurrency models? *Proc. of the 27th European conf. on Object-Oriented Programming* (*ECOOP'13*). Berlin; Heidelberg: Springer-Verlag, 2013. P. 302–326.
12. De Koster J., Marr S., Van Cutsem T., D'Hondt T. Domains: Sharing state in the communicating event-loop actor model. *Comput. Lang. Syst. Struct*. 2016. Vol. 45. P. 132–160.
13. Peters A., Kitchin D., Thywissen J., Cook W. OrcO: a concurrency-first approach to objects. *Proc. of the 2016 ACM SIGPLAN Intern. Conf. on Object-Oriented Programming, Systems*. *Languages, and Applications Pages*. 2016. Vol. 51, Oct. P. 548–567.
14. Eclipse Vert.x 2017. URL: http://vertx.io/.

15. Microsoft Corp. Asynchronous programming with async and await (C# and Visual Basic). 2015. URL: https://msdn. microsoft.com/en-us/library/hh191443.aspx.

16. Ларін В.О., Бантиш О.В., Білецький С.С., Галкін О.В. Візуальне прототипування та програмування акторів на Akka. *Проблеми інформаційних технологій*. 2015. № 18. Р. 133–138.

17. Scholliers C., Tanter E., De Meuter W. Parallel actor monitors: Disentangling task-level parallelism from data partitioning in the actor model. *Sci. Comput. Program*. 2014. Vol. 80, Feb. P. 52–64.

Надійшла до редакції 19.10.2017

В.О. Ларін, О.В. Бантиш, О.В. Галкін, О.І. Провотар
ПРЕДМЕТНО-ОРІЄНТОВАНА МОВА STRUMOK ДЛЯ ОПИСУ АКТОРНИХ СИСТЕМ ЗІ СПІЛЬНОЮ ПАМ'ЯТТЮ

Анотація. Запропоновано предметно-орієнтовану мову Strumok для опису межакторної взаємодії і роботи зі спільною пам'яттю. Проведено оцінювання ефективності мови Strumok з використанням загальної пам'яті у порівнянні з фреймворком Java Vert.x.

Ключові слова: модель акторів, паралельні системи зі спільною пам'яттю, предметно-орієнтована мова програмування, Akka, Java.

V.O. Larin, O.V. Bantysh, O.V. Galkin, O.I. Provotar
STRUMOK DSL FOR DEFINING ACTOR-ORIENTED SYSTEMS WITH SHARED MEMORY SUPPORT

Abstract. Strumok DSL is defined for describing actor interaction and shared system support. Performance of the Strumok DSL which uses shared memory is compared with that of Java Vert.x framework.

Keywords: actor model, parallel systems with shared memory, DSL, Akka, Java.

Ларин Владислав Олегович,
аспирант Киевского национального университета имени Тараса Шевченко, e-mail: vlarinmain@gmail.com.

Бантыш Олег Витальевич,
аспирант Киевского национального университета имени Тараса Шевченко, e-mail: iambantysh@gmail.com.

Галкин Александр Владимирович,
кандидат физ.-мат. наук, доцент Киевского национального университета имени Тараса Шевченко, e-mail: galkin@unicyb.kiev.ua.

Провотар Александр Иванович,
доктор физ.-мат. наук, профессор, заведующий кафедрой Киевского национального университета имени Тараса Шевченко; профессор Жешувского университета, Польша, e-mail: aprowata@unicyb.kiev.ua.