УДК: 681.3.06

АКТОРНОЕ РАСШИРЕНИЕ ЯЗЫКА JAVA В СРЕДЕ MPS

А.Р. Жукова, М.А. Мазин

Описанное в работе акторное расширение добавляет автоматически распараллеливающиеся функциональные конструкции в универсальный язык Java. Необходимость такого расширения вызвана слабой приспособленностью языка Java к написанию параллельных программ. Акторная модель, поддерживаемая расширением, успешно применяется в функциональном языке параллельного программирования Erlang. В качестве средства создания расширения была выбрана среда мета-программирования MPS, что позволило автоматически получить интегрированные средства разработки для применения расширения и, кроме того, достичь совместимости с другими языковыми расширениями, созданными в среде MPS.
Ключевые слова: акторная модель, проблемно-ориентированное расширение, параллельное программирование, языково-ориентированное программирование

Введение

В настоящее время растет актуальность разработки параллельных программ. Это связано с тем, что дальнейший рост производительности вычислительных систем более не может достигаться увеличением тактовой частоты процессоров, и главной тенденцией становится увеличение числа ядер в системах [1]. Но для того, чтобы программа выполнялась эффективно на нескольких ядрах, она должна быть написана с применением техник параллельного программирования [2].

Универсальные императивные языки программирования [3], наиболее распространенные на сегодняшний день, хорошо подходят для реализации последовательных алгоритмов, но плохо приспособлены для написания алгоритмов параллельных. Это связано с тем, что программа, написанная на императивном языке, представляет собой последовательность инструкций процессору, изменяющих состояние памяти [4]. Из-за того, что порядок этих команд зафиксирован, выполнение программы не может быть распределено между несколькими процессорами. Поэтому программисты вынуждены самостоятельно разбивать свою программу на параллельно выполняющиеся потоки [5].

Автоматическому распараллеливанию хорошо поддаются программы, написанные на функциональных языках программирования [6, 7]. Но функциональные языки, в отличие от императивных, значительно менее распространены. Тем не менее, некоторые функциональные конструкции в последнее время стали активно проникать в универсальные императивные языки программирования. Например, становится популярным использование в императивных языках замыканий [8] для обработки списков, в некоторых случаях [9] это позволяет выполнять автоматическое распараллеливание по данным [10].

В связи с этим естественно продолжить адаптацию абстракций параллельных функциональных программ для императивных языков программирования. К таким абстракциям, в частности, относится акторная модель [11], реализованная в языке программирования *Erlang* [12].

Акторная модель была предложена К. Хьюиттом, П. Бишопом и Р. Штайгером в 1973 г. Они ввели понятие актора – примитива параллельных вычислений, обладающего потоком выполнения и способного обмениваться сообщениями с другими акторами. Актор обрабатывает сообщения из собственной очереди сообщений и в ответ на них выполняет полезные действия.

Разработанное авторами языковое расширение позволяет использовать акторные конструкции при программировании на языке *Java* [13]. На рис. 1 приведены примеры объявления типа актора, объявления сообщения, создания нового актора и посылки сообщения. Из рисунка видно, что такой код синтаксически почти не отличается от объ-

явления класса, объявления метода, создания нового объекта и вызова метода в языке *Java*. Поэтому *Java*-программисту потребуются минимальные усилия для изучения синтаксиса акторного языкового расширения.

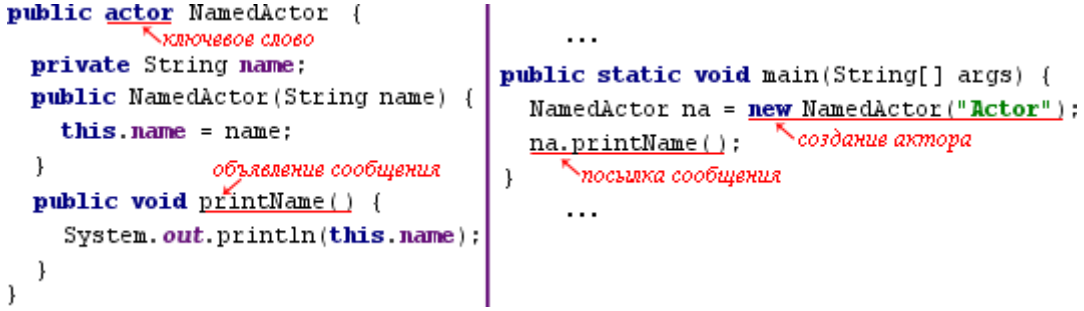

Рис. 1. Пример синтаксиса

Традиционно создание языка или языкового расширения [14] включает в себя разработку общей инфраструктуры языка: абстрактного и конкретного синтаксиса [15], системы типов [16], операционной семантики и т.д. Для этого реализуются лексический, синтаксический и семантический анализаторы, трансляторы и другие утилиты. Существуют различные инструменты, позволяющие частично автоматизировать эту деятельность [17, 18].

В настоящее время для повышения эффективности труда программисты используют интегрированные среды разработки [19], обладающие знаниями о языке программирования и значительно ускоряющие разработку. Поэтому создание нового языка предполагает также создание инфраструктуры среды разработки [20] с редактором, автодополнением кода, подсветкой ошибок, рефакторингами, системами версионирования и др. Помимо того, что эта задача сама по себе является ресурсоемкой и требует высококвалифицированных разработчиков, возникает проблема совместимости различных языковых расширений при использовании в рамках одной программы. Под совместимостью понимается возможность совместного использования компонентов, даже если они созданы независимо [21]. Например, различные библиотеки для языка *Java* совместимы, если их классы расположены в разных пакетах. Обеспечить совместимость языковых расширений несколько сложнее.

Из существующих средств создания языковых лишь *JetBrains MPS* [14, 22, 23] позволяет одновременно создавать как совместимые языковые расширения, так и языковую инфраструктуру для них. Поэтому именно среда *MPS* была выбрана для создания акторного расширения языка *Java*. Среда *MPS* поставляется с большим количеством уже готовых языков и языковых расширений, поддерживающих дополнительные конструкции: коллекции, замыкания, регулярные выражения, работу с датами и др.

Базовые понятия расширения Actor Language

Созданное авторами языковое расширение позволяет использовать акторы наряду с обычными объектами при программировании на языке *Java*. Декларация типа актора, поддерживаемая расширением, синтаксически совпадает с декларацией java-класса с той лишь разницей, что вместо ключевого слова «*class*» используется ключевое слово «*actor*». Для типа актора могут быть определены конструкторы, поля и объявления типов обрабатываемых сообщений. Синтаксис последних полностью совпадает с синтаксисом объявлений обычных методов в языке *Java*. Аналогично, синтаксис отправки сообщений актору не отличается от синтаксиса вызова методов у объекта. Но, в отличие

от вызова методов, отправка сообщений происходит асинхронно: сообщения помещаются в индивидуальную очередь актора и последовательно актором обрабатываются.

Содержание потока в языке *Java* – ресурсоемкий процесс, поэтому для обработки сообщений актору не выделяется отдельный экземпляр класса «*java.lang.Thread*». Вместо этого отправка всех сообщений всем акторам происходит посредством добавления сообщений в очередь общего диспетчера. Диспетчер обладает пулом потоков [24], размер которого либо задается программистом явно, либо вычисляется, исходя из числа доступных ядер в системе. По мере появления новых сообщений в очереди диспетчера и освобождения потоков из пула сообщения последовательно извлекаются из очереди для обработки акторами. При этом гарантируется, что сообщения, направленные одному и тому же актору, не могут обрабатываться параллельно двумя разными потоками. Таким образом, эмулируется обладание актором потоком управления.

На рис. 2 приведен пример объявления двух типов акторов: «*Ping*» и «*Pong*». При отправке актору «*Ping*» сообщения «*start*» между акторами начинается обмен сообщениями «*ping*» и «*pong*». После отправки заданного параметром «*pingCount*» числа сообщений актор «*Ping*» отправляет актору «*Pong*» сообщение «*stop*» и прекращает работу. В ответ на сообщение «*stop*» актор «*Pong*» также завершает работу.

```
public actor Ping {
  private int pingsLeft;
  public Ping() { }
  public void pong(Pong pong) {
    if (this.pingsLeft-- >= 0) {
      System.out.println("Ping: pong");
      pong.ping(this);
    } else {
      System.out.println("Ping: stop");
      pong.stop();
    }
  }
  public void start(int count, Pong pong) {
    System.out.println("Ping: start");
    this.pingsLeft = count;
    this.pong(pong);
  }
}
```

```
public actor Pong {
  public Pong() { }
  public void ping(Ping ping) {
    System.out.println("Pong: ping ");
    ping.pong(this);
  }
  public void stop() {
    System.out.println("Pong: stop");
  }
}
```

Рис. 2. Пример объявления акторов

Для запуска обмена сообщениями необходимо создать акторы «*Ping*» и «*Pong*» и отправить актору «*Ping*» сообщение «*start*». Код, выполняющий эти действия, и результаты работы этого кода приведены на рис. 3.

```
  ...
public static void main(String[] args) {
  Pong pong = new Pong();
  Ping ping = new Ping();
  ping.start(3, pong);
}
  ...
```

```
Ping: start
Ping: pong
Pong: ping
Ping: pong
Pong: ping
Ping: pong
Pong: ping
Ping: stop
Pong: stop
```

Рис. 3. Запуск обмена сообщениями между акторами и вывод программы

Обработка сообщений с задержкой

Языковое расширение *Actor Language* позволяет при посылке сообщения указать минимальную задержку, с которой оно должно быть обработано. Для этого в языковое расширение введена конструкция *defer*, которая может быть применена к оператору посылки сообщения. В качестве параметра конструкция *defer* принимает величину минимальной задержки, для задания которой используется встроенное в среду *MPS* языковое расширение *Dates*.

В языковом расширении *Dates* предусмотрены специальные проблемно-ориентированные конструкции для работы с датами, временем и промежутками времени. В частности, оно позволяет записывать периоды времени наглядно, указывая их величину и размерность: «5 minutes», «1 year».

Пример использования конструкции *defer* приведен на рис. 4: код актора *Pong* отредактирован таким образом, чтобы посылка сообщения *pong* происходила не сразу, а с задержкой в пять миллисекунд.

```
public actor Pong {
  public Pong() { }
  public void ping(Ping ping) {
    System.out.println("Pong: ping ");
    ping.pong(this).defer(5 milliseconds);
  }
  public void stop() {
    System.out.println("Pong: stop");
  }
}
```

Рис. 4. Отправка сообщения с задержкой

Отложенный результат

В традиционном объектно-ориентированном программировании некоторые методы в результате своего выполнения возвращают вычисленные значения. Эти значения становятся доступны вызывающему коду сразу по окончанию синхронного выполнения таких методов. Так может быть организовано взаимодействие между вызываемым и вызывающим кодом.

При программировании с использованием акторов нет возможности использовать возвращаемые значения, так как к тому моменту, когда операция асинхронной посылки сообщения оказывается выполненной, само сообщение может быть еще не обработано, а возвращаемое значение не вычислено.

Для того чтобы вызываемый код мог оповестить вызывающий код об окончании обработки события и передать вычисленное значение, можно передавать вызывающий актор в качестве параметра сообщения. По окончании обработки сообщения вызываемый актор должен послать вызывающему актору ответное сообщение. Именно так организованно взаимодействие в приведенном выше примере (рис. 2). Однако при таком подходе код обрабатывающего сообщение актора должен «знать» о структуре посылающего сообщение актора, т.е. возникает нежелательная сильная связанность кода [25].

Чтобы обойти эту проблему и позволить использовать возвращаемые обработчиками сообщений значения, в языковом расширении *Actor Language* существует поддержка механизма отложенных результатов. Для любого сообщения актора, так же как и для любого метода в языке *Java*, могут быть заданы тип возвращаемого значения и набор генерируемых исключений. Код обработчика сообщения должен либо вычислить значение соответствующего типа, либо сгенерировать исключение.

Оператор посылки сообщения имеет специальный тип *deferred*, параметризованный типом возвращаемого сообщением значения. Например, если обработчик сообщения возвращает значение типа *String*, то асинхронная посылка этого сообщения вернет значение типа *deferred<String>*. Используя это значение, код, посылающий сообщение, может определить действия как в связи с успешной обработкой сообщения, так и в связи с возникновением исключительной ситуации. Действия задаются в виде замыканий – анонимных функций, языковая поддержка которых осуществляется встроенным в среду *MPS* расширением *Closures*.

Использование отложенного значения может быть продемонстрировано на примере программы редактора графов. Одной из функций такой программы является чтение графов из файлов, реализованное актором *GraphReader*. При получении сообщения *read* актор *GraphReader* начинает выполнять длительную операцию чтения и разбора файла. Результатом этой операции является экземпляр класса *GraphModel*, но в процессе выполнения операции может возникнуть исключение *IOException*. Часть программы, использующая актор *GraphReader* (рис. 5), посылает ему сообщение *read*, а полученное отложенное значение сохраняет в переменной *model*. Далее с помощью оператора *addCallback* назначается действие в связи с успешной обработкой сообщения, а с помощью оператора *addErrback* – обработчик исключения *IOException*.

```
...
GraphReader reader = new GraphReader();          // асинхронный вызов метода
deferred< GraphModel > model = reader.read(new File("./graphModel.grph"));
model.addCallback({GraphModel result => frame.setGraphModel(result); });
                                                 // обработка результата
model.addErrback({IOException e =>
    JOptionPane.showMessageDialog(fileChooser, "Cannot open the file"); });
...                                              // обработка исключения IOException
```

Рис. 5. Действия по окончанию обработки сообщения

Заключение

Разработанное авторами языковое расширение *Actor Language* позволяет в привычной для *Java*-программистов манере использовать автоматически распараллеливающиеся функциональные конструкции. Расширение упрощает создание программ, которые эффективно используют возможности многоядерных компьютеров.

Благодаря тому, что расширение создано в среде *MPS*, программисты имеют возможность использовать его совместно с другими языковыми расширениями этой среды. В дальнейшем планируется развить языковое расширение *Actor Language* конструкциями для неблокирующего асинхронного ввода / вывода. Это позволит эффективно использовать акторы в разработке веб-приложений [26], а также в приложениях, интенсивно работающих с файловой системой.

Литература

1. Adve S. Parallel Computing Research at Illinois: The UPCRC Agenda. – 2008.
2. Воеводин В.В. Параллельные вычисления / В.В. Воеводин, Вл.В. Воеводин. – БХВ-Петербург, 2004.
3. Sebesta R. Concepts of Programming Languages / Robert W. Sebesta. – Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.
4. von Neumann J. First Draft of a Report on the EDVAC / John von Neumann // IEEE Annals of the History of Computing. – 1993. – Vol. 15. – № 4. – P. 28–75.
5. Эндрюс Г. Основы многопоточного, параллельного и распределенного программирования / Грегори Р. Эндрюс. – Вильямс, 2003.

6. Непейвода Н.Н. Стили и методы программирования. Курс лекций: Учебное пособие. – М.: Интернет-университет информационных технологий, 2005.
7. Trinder P.W. GUM: a portable parallel implementation of Haskell. – 1996.
8. Fowler, M. Closure. – 2004 [Электронный ресурс]. – Режим доступа: http://martinfowler.com/bliki/Closure.html, своб.
9. Duy J. PARALLEL LINQ: Running Queries On Multi-Core Processors / Joe Duy, Ed Essey // MSDN Magazine. – 2007.
10. Hillis D. Data Parallel Algorithms / W. Daniel Hillis, Guy L. Jr. Steele // Commun. ACM. – 1986. – Vol. 29. – № 12. – P. 1170–1183.
11. Hewitt C. Universal Modular ACTOR Formalism for Articial Intelligence / Carl Hewitt, Peter Bishop, Richard Steiger // IJCAI. – 1973. – P. 235–245.
12. Armstrong, J. Programming Erlang: Software for a Concurrent World / Joe Armstrong. – Pragmatic Bookshelf, 2007.
13. Эккель Б. Философия Java. – СПб: Питер, 2009.
14. Дмитриев С. Языково-ориентированное программирование: следующая парадигма // RSDN Magazine. – 2005. – № 5.
15. Компиляторы. Принципы, технологии и инструментарий / А. Ахо, М. Лам, Р. Сети, Дж. Ульман. – Вильямс, 2008.
16. Luo Z. Computation and Reasoning: A Type Theory for Computer Science // International Series of Monographs on Computer Science. № 11. – Oxford University Press, 1994.
17. Bovet J. ANTLRWorks: an ANTLR grammar development environment / Jean Bovet, Terence Parr // Softw. Pract. Exper. – 2008. – Vol. 38. – № 12.
18. Sevenich R. Compiler Construction Tools // Linux Gazette. – 1999. – Vol. 39 [Электронный ресурс]. – Режим доступа: http://linuxgazette.net/issue39/sevenich.html, своб.
19. Fowler M. Crossing Refactoring's Rubicon / Martin Fowler // ThoughtWorks. – 2001. [Электронный ресурс]. – Режим доступа: http://martinfowler.com/articles/refactoringRubicon.html.
20. Давыдов С. IntelliJ IDEA. Профессиональное программирование на Java. Наиболее полное руководство / С. Давыдов, А. Ефимов. – СПб: БХВ-Петербург, 2005.
21. Solomatov K. DSL Adoption with JetBrains MPS / Konstantin Solomatov // DZone, Architect Zone. – 2009.
22. Fowler M. A Language Workbench in Action. – MPS, 2005.
23. Fowler M. Language Workbenches: The Killer-App for Domain-Specic Languages. — 2005. — jun. [Электронный ресурс]. – Режим доступа: http://www.martinfowler.com/articles/languageWorkbench.html.
24. Goets B. Java theory and practice: Thread pools and work queues / Brian Goets // Developer Works, 2002.
25. Приемы объектно-ориентированного проектирования. Паттерны проектирования / Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес. Библиотека программиста. – СПб: Питер, 2007.
26. Моделирование контроллера Web-приложений с использованием UML / Е.А. Горшкова, Б.А. Новиков, Д.Д. Белов et al. // Программирование. – 2005. – № 1. – Р. 44–51.

Жукова Анна Руслановна – Санкт-Петербургский государственный университет, студент, anna.zhukova@math.spbu.ru

Мазин Максим Александрович – Санкт-Петербургский государственный университет информационных технологий, механики и оптики, аспирант, mazine@rain.ifmo.ru