АКТОРНОЕ РАСШИРЕНИЕ ЯЗЫКА JAVA В СРЕДЕ MPS

|  |  |
| --- | --- |
| ЖУКОВА АННА РУСЛАНОВНА | МАЗИН МАКСИМ АЛЕКСАНДРОВИЧ |
| Санкт-Петербургский государственный университет | Санкт-Петербургский государственный университет информационных технологий, механики и оптики |
| anna.zhukova@math.spbu.ru | mazine@rain.ifmo.ru |
| ООО "ИнтеллиДжей Лабс" | |

**Ключевые слова:** акторная модель, проблемно-ориентированное расширение, параллельное программирование, языково-ориентированное программирование

**Аннотация** 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.

**1 Введение**

В настоящее время растет актуальность разработки параллельных программ. Это связано с тем, что дальнейшее увеличение производительности вычислительных систем более не может достигаться увеличением тактовой частоты процессоров, и главной тенденцией становится увеличения 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 году. 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 рисунке 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 объявления класса, объявления метода, создания нового объекта и вызова метода в языке *Java*. Поэтому *Java*-программисту потребуются минимальные усилия для изучения синтаксиса акторного языкового расширения.

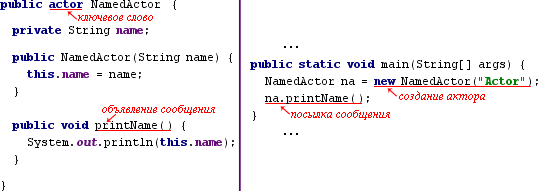


Рисунок 1. Пример синтаксиса

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.

В таблице 1 приведены существующие средства создания языковых расширений, с указанием поддержки совместимости разработанных с их помощью расширений и возможности создания языковой инфраструктуры.

|  |  |  |
| --- | --- | --- |
| **Название средства** | **Совместимость расширений** | **Языковая инфраструктура** |
| LISP | Есть | Нет |
| Внутренние языки в Ruby / Groovy | Есть | Нет |
| XText framework | Нет | Есть |
| JetBrains MPS | Есть | Есть |

Таблица 1. Существующие средства создания языковых расширений

Из приведенных средств лишь *JetBrains MPS* [14, 22, 23] одновременно позволяет 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.

**2 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 рисунке 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.



Рисунок 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 рисунке 3.

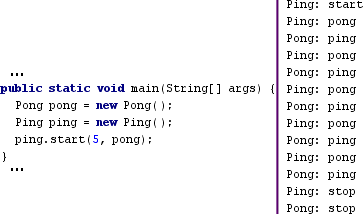


Рисунок 3. Запуск обмена сообщениями между акторами и вывод программы

**2.1 Обработка сообщений с задержкой**

Языковое расширение *Actor Language* позволяет при посылке сообщения указать минимальную задержку, с которой это сообщение 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 рисунке 4: код актора *Pong* отредактирован таким образом, чтобы посылка сообщения *pong* происходила не сразу, а с задержкой пять миллисекунд.

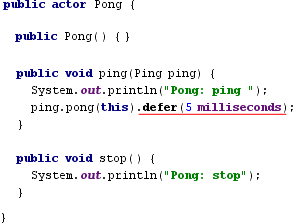


Рисунок 4. Отправка сообщения с задержкой

**2.2 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 организованно 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 выполнена, 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 окончанию 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, то есть 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 [26] – 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. На рисунке 5 приведен код актора *GraphReader*. Обработчик сообщения *read*, определенный для этого актора, принимает в качестве параметра файл, в котором хранится *XML*-представление графа, строит по нему модель графа и возвращает объект *GraphModel*. В процессе чтения и разбора файла могут быть сгенерированы исключения *SAXException* и *IOException*.

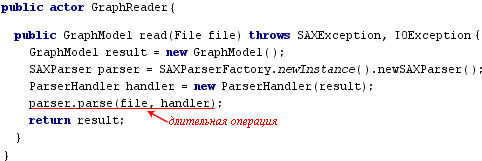


Рисунок 5. Актор *GraphReader*

Часть программы, использующая актор *GraphReader* (Рис. 6), посылает ему сообщение *read*, и 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*.

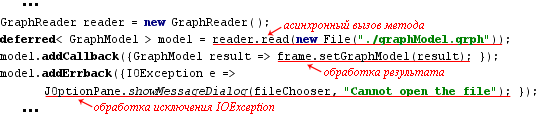


Рисунок 6. Действия по окончанию обработки сообщения

**3 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 это среды.

В дальнейшем планируется развить языковое расширение *Actor Language* конструкциями для неблокирующего асинхронного ввода / вывода. Это позволит эффективно использовать акторы в разработке веб-приложений [27], а также в приложениях интенсивно работающих с файловой системой.

**Литература**

1. S.V. Adve et al. (November 2008). ["Parallel Computing Research at Illinois: The UPCRC Agenda"](http://www.upcrc.illinois.edu/documents/UPCRC_Whitepaper.pdf" \o "http://www.upcrc.illinois.edu/documents/UPCRC_Whitepaper.pdf) (PDF). Parallel@Illinois, University of Illinois at Urbana-Champaign. "The main techniques for these performance benefits – increased clock frequency and smarter but increasingly complex architectures – are now hitting the so-called power wall. The computer industry has accepted that future performance increases must largely come from increasing the number of processors (or cores) on a die, rather than making a single core go faster."
2. В.В.Воеводин, Вл.В.Воеводин "Параллельные вычисления"-СПб.: БХВ-Петербург, 2002.
3. Себеста Р. Основные концепции языков М.: Вильямс, 2001.
4. последовательность инструкций процессору, изменяющих состояние памяти
5. программисты вынуждены самостоятельно разбивать свою программу на параллельно выполняющиеся потоки
6. Непейвода Н.Н. Стили и методы программирования. М.: Интернет-Университет Информационных технологий, 2005.
7. на что-нибудь про распараллеливание функ. программ
8. использование в императивных языках замыканий
9. ссылка на asParallel в Linq
10. ссылка про распараллеливание по данным
11. Hewitt C., Bishop P., Steiger R. A Universal Modular Actor Formalism for Artificial Intelligence. // Proc. of the Third Int. Joint Conf. on AI (IJCAI-73). – Stanford, CA: Stanford University: 1973. – P. 235–245.
12. Armstrong J. Programming ERLANG: Software for a Concurrent World. Pragmatic Bookshelf, 2007.
13. Eckel B. Thinking in Java. NJ: Prentice Hall, 2006.
14. Дмитриев С. Языко-ориентированное программирование: следующая парадигма. //RSDN Magazine. 2005. №5. http://www.rsdn.ru/article/philosophy/LOP.xml.
15. Ахо А., Сети Р., Ульман Дж. Компиляторы. Принципы, технологии, инструменты. М.: Вильямс, 2003.
16. Luo Z. Computation and Reasoning: A Type Theory for Computer Science. Oxford University Press, 1994.
17. antlr
18. jflex
19. Fowler M. PostIntelliJ <http://martinfowler.com/bliki/PostIntelliJ.html>
20. Fields D. K., Saunders S., Belyaev E. IntelliJ IDEA in Action. Manning Publications, 2006.
21. Solomatov K. DSL Adoption with JetBrains MPS <http://architects.dzone.com/articles/solving-problem-dsls-adoption>
22. Fowler M. A Language Workbench in Action – MPS <http://martinfowler.com/articles/mpsAgree.html>
23. Фаулер М. Языковой инструментарий: новая жизнь языков предметной области <http://www.maxkir.com/sd/languageWorkbenches.html>
24. Гетц Б. Теория и практика Java: Пулы потоков и очередь действий <http://www.ibm.com/developerworks/ru/library/j-jtp0730/index.html>
25. ссылка на GoF
26. ссылка про замыкания
27. разработке веб-приложений