УДК 658.012.011.56:378](477)

Н.Н. ГЛИБОВЕЦ, С.С. ГОРОХОВСКИЙ, М.С. СТУКАЛО

РАСШИРЕНИЕ ЯЗЫКА SCALA СРЕДСТВАМИ ПАРАЛЛЕЛИЗМА И РАСПРЕДЕЛЕННОСТИ С ПОМОЩЬЮ КООРДИНАЦИОННОЙ СИСТЕМЫ LINDA

Ключевые слова: *распределенные вычисления, координационные системы*.

ВВЕДЕНИЕ

Компонентно-ориентированный подход к проектированию и реализации сложных программных систем в определенном смысле развивает объектно-ориентированную парадигму. Его использование становится более производительным для разработки масштабных проектов и распределенных систем, например корпоративных применений [1, 2].

Использование потенциала масштабных распределенных систем требует наличия программных моделей, явно оперирующих понятиями параллельного сотрудничества между большим числом активных сущностей, составляющих единственное применение. Такая потребность привела к проектированию и внедрению нескольких координационных моделей вместе с отдельными языками, поддерживающими эти модели. Они созданы для предоставления разработчикам каркаса (framework), который улучшал бы модуль, способствовал повторному использованию компонентов (последовательных или уже параллельных), повышал кроссплатформность и языковую совместимость. Однако эти модели отличаются именно понятием координации: что конкретно координируется, какими средствами достигается и какие метафоры применяются для представления этих понятий [3].

Модели координации ориентированы на задачи (такие как Linda), используют общее пространство данных. Такой подход предоставляет разработчикам простой и эффективный способ достижения многих целей, главная из них — оптимальное взаимодействие между процессами. Здесь коммуникация процессов реализуется введением небольшого числа операций, которые легко могут использовать программисты.

В середине 1980-х годов Д. Гелернтер из Йельского университета предложил скорее подход, чем отдельный язык, параллельного программирования Linda [4]. В нем параллельная программа состоит из многих процессов, каждый работает как обычная последовательная программа. Эти процессы имеют доступ к общей памяти, единицей хранения в которой является кортеж. Для связи процессов с общей памятью существует всего шесть базовых операций.

Кроме стилистических отличий между координационными моделями, влияющими на степень разделения вычислительных и координационных частей, выделяют и разное применение.

Модель *data-driven* координации используется в основном для распараллеливания вычислительных задач. Подход *control-driven* координации обычно применяется для моделирования систем. Это можно объяснить тем, что в рамках конфигурационного компонента программист имеет больший контроль над данными в случае использования языков, которые поддерживают *data-driven* координацию, чем в случае *control-driven* координации. Представители первой категории, как правило, пытаются координировать данные, а представители второй — координировать сущности (могут быть не только обычными процессами, но и устройствами, компонентами системы и т.п.).

Многие исследователи считают, что частичное отсутствие прогресса в компонентном программном обеспечении объясняется недостатками языков программирования, используемых для определения и интеграции компонентов. Большинство языков предлагают лишь ограниченную поддержку абстрагирования и композиции компонентов. Это касается, в частности, таких статически типизирующих языков, как Java и C#.

Настоящая статья направлена на решение этой проблемы с использованием базовых идей координационной модели Linda относительно улучшения интеграционных механизмов языков компонентного программирования на примере расширения Scala.

1. ЯЗЫК ПРОГРАММИРОВАНИЯ SCALA

Язык Scala создан в 2001–2004 гг. в лаборатории методов программирования EPFL [5] в рамках улучшения языковой поддержки компонентного программного обеспечения. В основном эти исследования направлены на реализацию двух идей. Во-первых, язык компонентных применений должен быть масштабированным, т.е. должна существовать возможность с помощью одних и тех же концепций описывать как малые, так и крупные части применений. Поэтому разработчики языка сконцентрировались на эффективной реализации механизмов абстракции, композиции и декомпозиции вместо введения большого количества примитивов, полезных только на каком-то одном уровне масштабирования. Во-вторых, вполне естественно создать язык, унифицирующий и обобщающий объектно-ориентированное и функциональное программирование. Некоторые из основных технических нововведений Scala — это концепция, объединяющая эти парадигмы программирования.

Язык Scala не предоставляет никаких примитивов для параллельного программирования. Ядро языка построено так, чтобы упростить создание библиотек поддержки разных моделей параллелизма, надстроенных над текущей моделью языка-основы [6].

Отметим также ключевые аспекты Scala. Программы во многом похожи на Java-программы и могут свободно взаимодействовать с Java-кодом; язык включает унифицированную объектную модель в том смысле, что любое значение является объектом, а любая операция — вызовом метода; функциональность языка выражается в том, что функции — это полноправные значения. Введены мощные унифицированные концепты абстракций для типов и значений; имеем гибкие симметричные конструкции примесей (mixin) для композиции классов и коллекций методов *traits*; ввиду чего декомпозируют объекты, сравнивая с образцом (уже реализовано на платформах Java и .NET).

Основные материалы по языку см. http://www.scala-lang.org/, среди них выделим книгу одного из соавторов языка М. Одерского «Programming in Scala» [6].

2. LINDA ДЛЯ SCALA

Имплементация Linda для Scala, описанная в этой работе, базируется на объектно-ориентированном подходе и заключается в создании специализированной библиотеки. Именно концептуальная элегантность Linda, в том числе ассоциативная структура пространства кортежей, вызвала определенные трудности в реализации [7]. Вместо встроенных типов данных Scala (язык со статической типизацией) для представления типов данных применяют классы. Абстрактный базовый класс — *LindaType*. Каждый класс, использованный в кортежах, должен наследоваться от базового класса, который позволяет разработчику использовать типы данных, определенные в этой библиотеке и, что важно, создавать новые типы данных.

В *LindaType* определена лишь булева переменная *formal*, которая указывает, является класс актуальным или формальным параметром. Для передачи по сети применяется встроенная в Java сериализация объектов.

2.1 Типы данных. Каждый тип данных библиотеки наследуется от базового класса *LindaType*. Это позволяет реализовать простое превращение определенных в Scala классов *TupleN* (*n*-мерный кортеж, $1 \le n \le 22$) к более удобному с точки зрения обработки кортежа в виде списка *List*[*LindaType*]. Ограничение количества числом 22 связано со способом реализации кортежей в Scala. Необходимость в таком представлении возникла потому, что класс *TupleN* не поддерживает итерацию, которая значительно осложняет сравнение кортежа с антикортежом. Еще одним аргументом для использования списка как структуры данных для кортежа было то, что в Scala определено много функций для работы со списками.

Для сравнения типов данных, производных от *LindaType*, используется переопределенный оператор равенства (==) и вспомогательный метод *canEqual*. Определение метода равенства для *LindaInteger* см. на рис. 1.

Имплементация Linda для Scala имеет набор заранее определенных типов данных, которые приведены в табл. 1.

```
override def equals (other : Any) = other match {
 case that: LindaInteger =>
  (that canEqual this) &&
  (this.Value == that.Value)
 case _ =>
  false
 }
 def canEqual (other: Any) = other.isInstanceOf[LindaInteger]
```

Рис. 1

Таблица 1

Класс	Тип данных
LindaDouble	Double
LindaFloat	Float
LindaInteger	Int
LindaLong	Long
LindaString	String

Например, фрагмент кода на рис. 2 показывает использование типа данных *LindaInteger.*

```
val lint:LindaInteger = 20     //фактический тип

val lint2 = new LindaInteger (15)   //фактический тип
lint.toFormal      //теперь lint — это формальный параметр, который можно применить
                   для операции in или rd
```

Рис. 2

Последняя строчка кода показывает важную особенность библиотеки. Любой фактический параметр можно превратить в формальный и, наоборот, с помощью методов *toFormal* и *toActual*, определенных в базовом классе *LindaType*.

Опишем, как создать новые типы данных и использовать их в Linda для Scala. Для начала нужно определить новый тип данных (рис. 3) как производный от *LindaType.*

```
case class LindaPoint (coord1 : Int, coord2 : Int) extends LindaType {
 val x = coord1
 val y = coord2

 override def getType = "point" //метод для получения сигнатуры типа
 override def toString = x+" "+v //метод для превращения значений в строку
 }
```

Рис. 3

Ключевое слово *case* перед определением класса указывает, что для этого класса применимый встроенный в Scala механизм поиска соответствий (pattern matching). Для сохранения кортежей в базе данных необходимо получить сигнатуру каждого элемента кортежа с помощью функции *getType*. Затем из этих сигнатур формируется строка, необходимая для нахождения кортежей в базе данных. Функция получения такой строки определена в объекте-синглетоне (singleton object) *LindaTuple* (рис. 4).

```
def getSignature (tuple : List[LindaType]) : String ={
 var output = ""
 tuple.foreach(arg => output = output +" " + arg.getType)
 output.drop(1)
 }
```

Рис. 4

Для новых классов также нужно определить метод сравнения объектов этого класса между собой аналогично рис. 1.

2.2. Особенности реализации операций. Координационная модель Linda содержит небольшое количество операций коммуникации процессов и распределение пространства данных. Поскольку коммуникационные операции должны принимать на вход любой тип данных и иметь простой синтаксис, то логично объявить, что входным параметром этих операций должна быть *Any* — базовый тип в иерархии классов Scala. Фактически на вход операций подается кортеж Scala (*TupleN*), который необходимо преобразовать в кортеж Linda. Поэтому на количество входящих объектов накладывается ограничение в 22, что связано со способом реализации кортежей в Scala. Преобразование кортежа в нужную форму проводится с помощью функции *createLindaTupl*, находящейся в объекте-синглетоне *LindaTuple.* Эта функция используется в системе один раз, когда клиент передает запрос от процесса на сервер. После этого кортеж представляет собой список значений типа *LindaType.*

Для связи процесса с пространством кортежей он должен создать новый экземпляр клиента и с его помощью работать с серверной базой данных (рис. 5).

```
package org.linda.Test
import org.linda.LindaTypes._
import org.linda.Client._

object process extends Application {
 val i1 = new LindaInteger (28) //создание новых объектов данных
 val i2 = new LindaInteger (34)
 val i3 = new LindaInteger (14)
 val s = new LindaString ("hello")

 val client = new Client //создание нового клиента
 client.out (i1, i2, s) //выполнение операции out
 client.out (i1, i3, s)
 i2.toFormal                    //преобразование фактического параметру в формальный
 client.in(i1, i2, s)           //выполнение операции in
}
```

Рис. 5

На рис. 5 продемонстрированы основные моменты работы с клиентом — создание новых объектов значений, создание клиента, выполнение операций. Отметим, что последняя операция *in* возвратит один из двух кортежей: (28, 34, "hello") или (28, 14, "hello"), поскольку актуальный параметр *i*2 превратился в формальный, реально стал шаблоном типа *<Int>*.

2.3. База данных для кортежей. Связка клиент-сервер используется для передачи запросов к базе данных, которая реализует общее распределенное пространство кортежей и отвечает за поиск кортежей в этом пространстве. Структура для хранения данных — отображения *Map*, вида *[signature : String –> List [List [Lindatype]]]*, где каждой строке-сигнатуре ставится в соответствие список кортежей, которые имеют такую же сигнатуру. Такой способ организации кортежей упрощает поиск нужного кортежа. Для нахождения соответствия с антикортежом сначала получается сигнатура антикортежа, затем из отображения выбирается список кортежей с соответствующей сигнатурой, с помощью функции *find* ищутся все кортежи, которые удовлетворяют шаблону, полученному из антикортежа. Если кортежей больше одного, то из них функцией *choosetuple* случайным образом выбирается один. Затем он удаляется из базы данных, обновленный список записывается в отображение. Детальнее этот процесс описан на рис. 6.

2.4. Linda-клиент. Клиент отвечает за передачу запросов от процесса к серверу и за возвращение полученного результата. Для связи между сервером и клиентом используются Java-сокеты. При выполнении какого-то метода клиента, например операций *in* или *out*, создаются новый сокет для соединения с сервером, потоки (*streams*) для записи и чтения объектов из сервера, передается запрос на сервер,

ожидается ответ и процессу возвращаются полученные значения. Реализация операции *in* на клиенте описана на рис. 7.

```
def in (antiTuple : List[LindaType]) : List[LindaType]= {
 println(" db in")
 val signature = LindaTuple.getSignature(antiTuple) //получение сигнатуры кортежа
 val tuple = chooseTuple(find(antiTuple)) //поиск и выбор кортежа
 if (db(signature).length == 1) //если выбранный кортеж единственный с такой сигнатурой
        db - signature //то удалить сигнатуру из отображения
 else
 db(signature) = deleteTuple(tuple,db(signature)) //удаление кортежа из базы данных
 tuple //вернуть кортеж
}
```

Рис. 6

```
def in (input : Any) : List[LindaType] = {
val socket = new Socket(ia, port) //создание нового сокета
val outStream = new ObjectOutputStream(new DataOutputStream(socket.getOutputStream()))  //
                создание нового выходного потока
val inStream = new ObjectInputStream (new DataInputStream(socket.getInputStream()))
                //создание нового входного потока
val antiTuple = LindaTuple.createLindaTuple(input)          //создание кортежа Linda
                outStream.writeObject(("in","",antiTuple))   //передача запроса на сервер
                val tuple = inStream.readObject().asInstanceOf[List[LindaType]]   //чтение
                объекта от сервера
                outStream.close()   //закрытие потоков
                inStream.close()
                tuple   //возвращение найденного кортежа процесса}
```

Рис. 7

2.5. Linda-сервер. Задача сервера — ожидать от клиентов запросы и выполнять их. Для выполнения каждого запроса запускается отдельный поток, который и делает запросы к базе данных. Потоки могут выполняться параллельно. Каждый поток открывает *streams* для чтения и записи объектов с/на клиент. Именно в потоках реализовано, что такие операции, как *in* или *rd* приостанавливают выполнение процесса до момента, пока в базе данных не появится нужный кортеж. Синхронизация между потоками происходит по объекту базы данных с помощью метода Java syncronized: *db.synchronized* {…}, а когда в базе данных нет соответствующего кортежа, выполнение потока приостанавливается по команде *db.wait*(). Реализация операции *in* на сервере описана на рис. 8.

```
case "in" => {while (db.find(tuple).isEmpty)
        db.wait()
   outStream.writeObject(db.in(tuple))
}
```

Рис. 8

2.6. Локальное пространство кортежей. Локальное пространство кортежей используется для поддержки дополнительных операций наподобие *collect* или *copy-collect*, когда из распределенного пространства кортежей возвращается не отдельный кортеж, а список кортежей, которые необходимо сохранить. Общая схема взаимодействия между клиентом и сервером, учитывая локальное пространство кортежей, представлена на рис. 9, откуда видно, что локальное пространство данных и сервер ис-

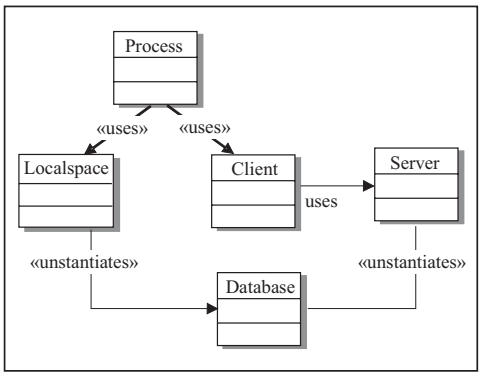

Рис. 9

пользуют один и тот же класс базы данных для создания пространства кортежей. Для процесса локальное пространство выполняет те же функции, что и сервер для клиента, а именно, передает запросы процесса к базе данных.

2.7. Дополнительные операции. В этой имплементации Linda для Scala реализовано такие дополнительные примитивы или операции: *collect*, *copy-collect*, *query*, *copyquery*. Операции *copy-collect* и *collect* описывались ранее. Две последние операции также имеют предикатную форму, которая не приостанавливает выполнение процесса, а возвращает значение, равное *true* или *false*. Операция *query* похожа на операцию *collect* с различием в том, что здесь указывается количество элементов, которое необходимо возвратить из пространства кортежей.

Соответственно операция *copyquery* несет ту же самую функциональность, что и предыдущие, за исключением того, что она не удаляет найденные кортежи из пространства кортежей.

ЗАКЛЮЧЕНИЕ

Очевидно, что для нахождения решения сложных и масштабных вычислительных проблем должны использоваться и усовершенствоваться методы распараллеливания и координации отдельных вычислительных задач. Эффективное программное решение их реализации предложил Д. Гелертер. Разработанную им координационную модель Linda можно добавлять практически к любому языку программирования.

Модель Linda базируется на концепции общего пространства данных, для доступа к которому у процессов есть небольшой, но достаточный набор примитивов или операций, с помощью которых процесс может разместить, прочитать или изъять кортеж из общего пространства данных. Процессы общаются лишь через общее пространство данных, которое позволяет разнести их в пространстве и времени (space and time decoupling).

В настоящей работе описано созданную нами библиотеку классов, направленную на имплементацию (http://www.ukma.kiev.ua/~gor/LindaforScala/) Linda для языков программирования Scala (языковой поддержки компонентного программного обеспечения). Библиотека использует клиент-серверную архитектуру, базу данных для представления пространства кортежей. В ней имплементирован механизм поиска соответствий, «примитивные» и дополнительные операции. Библиотека может использоваться для разработки распределенных параллельных приложений.

Описанная библиотека нуждается в среде разработки Eclipse [7] с установленным плагином для работы с языком Scala или отдельный пакет с интерпретатором командной строки и компилятором, который можно найти на сайте Scala [8].

Разработанная библиотека кросплатформная, поскольку язык Scala использует Java для работы и компилируется в байт-код Java. Библиотека разработана в среде Mac OS X версии 10.5.7 с установленной Java 2 Runtime Environment версии 1.5.0_16, версией Eclipse SDK 3.4.2 и Scala 2.7.4.

СПИСОК ЛИТЕРАТУРЫ

1. Моделі, методи та технології паралельного програмування / П.І. Андон, А.Ю. Дорошенко, О.А. Летичевський, О.Л. Перевозчикова та ін. // Стан та перспективи розвитку інформатики в Україні. — Київ: Наук. думка, 2010. — С. 242–258.
2. http://folding.stanford.edu/
3. Глибовец Н.Н., Федорченко В.М. Упрощенная инфраструктура для трансформации XML-моделей // Кибернетика и системный анализ. — 2010. — № 1. — С. 105–111.
4. Gelernter D. Generative communication in Linda // ACM Transact. on Programming Languages and Systems (TOPLAS). — 1985. — **7**, N 1. — P. 80–112, http://www.ece.rutgers.edu/~parashar/Classes/03-04/ece572/papers/gencommlinda.pdf
5. http://www.epfl.ch
6. Odersky M., Spoon L., Venners B. Programming in Scala, http://www.artima.com/shop/programming_in_scala
7. http://www.eclipse.org/
8. http://www.scala-lang.org/

Поступила 09.04.2010