Учреждение образования
Белорусский государственный университет
информатики и радиоэлектроники

УДК 004.432

Глущенко
Сергей Вадимович

Ускорение Scalaколлекций с помощью макросов и рефлексии

АВТОРЕФЕРАТ
на соискание степени магистра технических наук
по специальности 1-40 80 04 Математическое моделирование,
численные методы и комплексы программ

_____

Научный руководитель
Стержанов Максим Валерьевич
доцент, к.т.н.

_____

Минск 2015

ОБЩАЯ ХАРАКТЕРИСТИКА РАБОТЫ

Цель и задачи исследования

Целью диссертационной работы является увеличение производительности коллекций языкаScala.

Для достижения поставленной цели необходимо решить следующие задачи:

1. Рассмотреть архитектуру Scala коллекций.
2. Проанализировать инструменты метапрограммирования (макросы и рефлексия) доступные в языке Scalaи их применимость для увеличения производительности.
3. Разработать набор макросов, разворачивающих вызовы методов Scalaколлекций в циклы, не использующие создаваемых динамически анонимных классов.
4. Произвести сравнительный анализ влияния разработанных макросов на производительность Scalaколлекций.

Объектом исследования являются языки программирования.

Предметом исследования являются язык программирования Scala, JVMи метапрограммирование.

Основной *гипотезой*, положенной в основу диссертационной работы, является возможность при помощи техник метапрограммирования, доступных в языке Scala, преобразовывать вызовы методов коллекций, принимающих функции высшего порядка в циклы таким образом, что они могут быть успешно оптимизированы компилятором и JVM.

Личный вклад соискателя

Результаты, приведенные в диссертации, получены соискателем лично. Вклад научного руководителя М. В. Стержанова, заключается в формулировке целей и задач исследования.

Структура и объем диссертации

Диссертация состоит из введения, общей характеристики работы, трех глав, заключения и списка использованных источников. В первой главе

представлен анализ предметной области, рассмотрена архитектура Scalaколлекций и возможности метапрограммирования, предоставляемые языком Scala. Вторая глава посвящена разработке макросов, обеспечивающих ускорение работы у методов коллекций. В третьей главе произведён сравнительный анализ влияния разработанных макросов на производительность коллекций.

Общий объем работы составляет 50 страниц, из которых основного текста – 40 страниц, 22 рисунка на 10 страницах, 1 таблица на 1 странице, список использованных источников из 24 наименований на 2 страницах.

ВВЕДЕНИЕ

Новые языки, доступные под JVM, и новые программные парадигмы (такие как ForkJoin) показали слабость текущих методов оптимизации, использующих встраивание кода (inlining). Зачастую встраивание не происходит на важнейших участках кода, и это отрицательно влияет на производительность программного обеспечения. Проблема заключается в том, что компилятору тяжело получить необходимый объем информации о часто вызываемых внутренних циклах, как правило, содержащих в основном лишь мегаморфный виртуальный вызов без какого-либо дополнительного кода. Мегаморфными виртуальными вызовами называются вызовы, в которых вызывающая сторона может ожидать объекты различного типа во время выполнения. На Рисунке 1.1 показана тривиальная реализация метода map, которая применяет некоторую заранее определённую функцию ко всем элементам исходного массива и сохраняет результат в целевой массив. В данном случае каждый элемент массива увеличивается на 1.

```
// Thefunctionintheinnerloop
long add1(long a) { return a + 1;}
 // The iterator function
void map(long[] dst, long[] src) {
for(inti=0; i<dst.len; i++) // simple loop
dst[i] = add1(src[i]); // around a simple loop body
}
```

Рисунок 1.1 Пример тривиальной реализации map в языке Java

В данном случае встраивание функции add1 играет важнейшую роль в увеличении производительности. Без этого компилятор не будет знать, что происходит в теле цикла (т.к. в общем случае может быть реализована любая логика), а если код add1 будет встроен в цикл, то компилятор сможет полностью проанализировать функцию и увидеть, что это простой цикл, итерирующий элементы массива и сохраняющий результат в другой массив. В этот момент JITкомпилятор может убрать из генерируемого кода проверку на выход за границы массива, заменить цикл последовательностью выражений, произвести предварительную загрузку данных и другие оптимизации.

Естественно, что кроме такой реализации map, программисту потребуется иметь в своем распоряжении и другие похожие функции, add2,

mul3, filteri т.д. В конечном итоге потребуется некоторый способ передать в метод mapфункцию, которая и будет описывать действие над элементами массива. В языке Java наиболее частой реализацией является использование объекта, реализующего интерфейс Callable или Runnable. На Рисунке 1.2 показано использование Callableдля этих целей.

```
// Asampleiteratorfunction
2 void map(CallableOneArg fcn1arg, long[] dst, long[] src)
{
3 for(inti=0; i<dst.len; i++)
4 dst[i] = fcn1arg.call(src[i]);
5 }
```

Рисунок 1.2 Пример реализации общей функции mapв языке Java

При такой реализации в качестве аргумента fcn1argможет выступать практически любая функция одного параметра. К сожалению, при такой реализации внутренний цикл, в который должен бы быть встроен код fcn1arg для дальнейшей оптимизации цикла, содержит вызов fcn1arg.call, у которого могут быть десятки различных реализаций. Поскольку все эти функции вызываются в различное время, JITне может знать, какую из них и когда нужно встроить. В таком случае JIT не делает оптимизаций, заточенных ни под одну из возможных функций, а вместо этого пытается оптимизировать виртуальный вызов. Сам по себе виртуальный вызов не имеет отрицательного влияния на производительность, однако недостаточность информации о происходящем внутри вызова делает невозможным произведение важных оптимизаций, описанных выше.

Существует несколько способов решения описанной выше проблемы, например, можно сделать вызов внутренней функции вызовом статического метода, который потом может быть встроен JIT. Однако большинство подобных решений имеют существенные недостатки: например, при описанном подходе пришлось бы создать отдельные итераторы для каждой из функций add1, add2, mult3 и т.д., что в итоге привело бы к наличию большого числа идентичных итераторов, различающихся лишь применяемой внутри функцией. Все эти итераторы рано или поздно стали бы переполнять кэш инструкций ЦП, не говоря уже о чисто архитектурной тяжести поддерживать десятки практически одинаковых итераторов.

Вместо этого, данная работа сфокусирована на использовании техник метапрограммирования, предоставляемых языком Scala, и добавляет в

коллекции методы, реализованные через макросы, например, macroМари macroForeach. Они предоставляют такую же функциональность, как и обычные mapи foreach, однако, как будет показано далее, работают значительно быстрее, поскольку обходят обозначенную выше проблему. В большинстве случаев такие методы можно считать заменой обычных методов коллекций стандартной библиотеки.

КРАТКОЕ СОДЕРЖАНИЕ

1.1 Описание архитектуры коллекций языка Scala и техник метапрограммирования

Scala — мультипарадигмальный язык программирования, спроектированный кратким и типобезопасным для простого и быстрого создания компонентного программного обеспечения, сочетающий возможности функционального и объектно-ориентированного программирования. Набор коллекций в стандартной библиотеке языка богат и разнообразен, однако все коллекции организованы в простую и понятную иерархию благодаря продуманной архитектуре. Также стандартная библиотека языка предоставляет средства для удобного и быстрого быстрого создания коллекций и написания своих структур данных.

Что касается предоставляемых техник метапрограммирования, на уровне синтаксиса языка доступны только негигиеничные нетипобезопасные макросы. Однако, используя эти макросы как основу, язык программирования Scalaпредоставляет полностью типобезопасные макросы, которые позволяют сделать процесс преобразования ASTкратким и безопасным. Макросы в Scalaдобавляются на более высоком уровне, чем уровень стандартной библиотеки языка, так что для внедрения макросов в стандартную библиотеку приходится использовать специальную технологию FastTrack, которая применяется в компиляторе scalac.

1.2 Разработка программного средства для ускорения Scalaколлекций

Разработанное преобразование ASTзаменяет вызовы методов, выполняющих операции над коллекциями и использующих функции как аргументы, на локальные выражения, представляющие собой циклы, а передаваемые в эти методы функции – на вызовы локально определённых свободных методов. Таким образом, данное преобразование позволяет компилятору и JVMполностью проанализировать как построение коллекции, так и логику передаваемых функций, что даёт им возможность производить высокоэффективные оптимизации кода.На уровне стандартной библиотеки

языка преобразование представлено набором макросов, добавленных коллекции стандартной библиотеки. Каждый из этих макросов является заменой определённого метода коллекции. В силу того, что на уровне стандартной библиотеки языка макросы недоступны, они были добавленыколлекции стандартной библиотеки языка при помощи технологии FastTrack. В зависимости от типа коллекции - IndexedSeq, LinearSeq, Traversable, применяются различные виды преобразования, каждое из которых учитывает специфику работы и структуру соответствующей коллекций для достижения лучших результатов в смысле ускорения методов коллекции. Для проведения преобразования требуется, чтобы передаваемая в метод коллекции функция не содержала более одной конструкции return, прерывающей выполнение этой функции. В ходе работы над преобразованием были выявлены ограничения, которые планируется устранить в будущем: иногда теряется возможность переопределить методы коллекций при наследовании, а также преобразование всегда происходит согласно статическому типу коллекции, хотя преобразование согласно реальному типу иногда может ускорить метод значительнее.

1.3 Проведенные измерения

Произведённые измерения показывают, что при использовании разработанного преобразования ASTускорение времени работы методов коллекций в среднем составляет 40%. Наилучших результатов удалось добиться на коллекциях типа IndexedSeq, благодаря использованию специфических свойств этих коллекций, таких как константное время чтения и изменения элемента по индексу, а также того, что иногда при создании таких коллекций компилятор может преобразовать тип элементов коллекции в примитивы JVM. Наименьшее ускорение достигнуто для преобразования коллекций типа Traversable, поскольку эта группа коллекций является наибольшей и включает в себя самые разнообразные коллекции, что позволяет использовать в преобразовании только наиболее общие методы, присутствующие во всех коллекциях этой группы. Вместе с тем, рассмотренные в способы выбора типа преобразования могут дать возможность значительнее ускорить методы коллекций этого типа.

Анализируя результаты проведённых измерений, можно сделать вывод, что удалось реализовать на практике гипотезу, положенную в основу работы, а именно: при помощи техник метапрограммирования, доступных в языке Scala, можно преобразовывать вызовы методов коллекций, принимающих функции высшего порядка в циклы таким образом, что они могут быть успешно оптимизированы компилятором и JVM.

ЗАКЛЮЧЕНИЕ

В результате работы над предметной областью достигнуты положительные результаты по всем сформулированным направлениям. Используя техники метапрограммирования, предоставляемые языком Scala начиная с версии 2.10, удалось ускорить методы коллекций из стандартной библиотеки языка, такие как map, foreach, filter, fold и т.д., сделав их работающими в среднем на 40% быстрее. Ускорение производительности было достигнуто благодаря разработанному преобразованию AST. Данное преобразование заменяет вызовы методов, выполняющих операции над коллекциями и использующих функции как аргументы, на локальные выражения, представляющие собой циклы, а передаваемые в эти методы функции – на вызовы локально определённых свободных методов. Такое преобразование позволяет компилятору и JVM полностью проанализировать как построение коллекции, так и логику передаваемых функций, что даёт им возможность производить высокоэффективные оптимизации кода. В итоге получилось раскрыть тему магистерской диссертации и на практике реализовать описанные идеи.

Исследование результатов показало необходимость доработки алгоритмов по следующим направлениям:

1. Из-за преобразования в некоторых случаях нельзя переопределить методы у коллекций, что нарушает правило подстановки Лисков. [9]

2. В коде вида *Traversable[Int] tr = List(1,2,3); tr.macroMap(_+1);* операция *macroMap* будет развернута согласно статическому типу *tr*, т.е. будет применено преобразование для *Traversable*, а не для *List*. Однако, согласно полученным результатам, преобразование для *List* улучшает производительность значительнее, чем преобразование для *Traversable*. Наилучшим решением в данной ситуации была бы возможность выбора наиболее оптимального среди всех доступных вариантов преобразований, что, как показано выше, на текущий момент не поддерживается.

В дальнейшем планируется развивать и усовершенствовать алгоритмы преобразования AST, а также работать над более точным выбором способа преобразования.