ОГЛАВЛЕНИЕ

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

ПО — программное обеспечение.

ЯП — язык программирования.

Целевая программа — программа, получающаяся в результате работы компилятора.

Целевой язык — язык, на котором составлена целевая программа.

JVM (Java Virtual Machine) — виртуальная машина ЯП Java — основная часть исполняющей системы Java, так называемой Java Runtime Environment (JRE). Виртуальная машина Java исполняет байт-код Java, предварительно созданный из исходного текста Java-программы компилятором Java.

Байт-код — стандартное промежуточное представление, в которое может быть переведена компьютерная программа автоматическими средствами. По сравнению с исходным кодом, удобным для создания и чтения человеком, байт-код — это компактное представление программы, уже прошедшей синтаксический и семантический анализ.

IDE (Integrated development environment, интеграированная среда разработки) — комплекс программных средств, используемый программистами для разработки ПО. Как правило, среда разработки включает в себя: текстовый редактор, компилятор и/или интерпретатор, средства автоматизации сборки, отладчик.

IntelliJ платформа — это платформа, разработанная для создания инструментов анализа кода на различных языках программирования в рамках IDE.

Дерево разбора PSI (Program Structure Interface) — это конкретное синтаксическое дерево определенного формата, используемое в IntelliJ платформе.

Профилировщик (профайлер) — инструмент, предназначеный для сбора характеристик программы, таких как время выполнения отдельных ее фрагментов (функций, строк) с целью оценки производительности программы и проведения её дальнейшей оптимизации.

ВВЕДЕНИЕ

В области разработки программного обеспечения довольно часто поднимается вопрос производительности разрабатываемых программ. Требование к производительности — одно из важнейших нефункциональных требований для большинства продуктов.

На пути от написания кода программы до исполнения соответствующего ей машинного кода на процессоре есть множество факторов, которые так или иначе могут влиять на производительность разрабатываемой программы.

Не углубляясь в конкретные факторы, выделим стадии с этими факторами от написания кода до исполнения машинного кода программы на процессоре:

- написание кода программистом,
- компиляция кода (некоторые стадии могут отсутствовать, либо быть совмещены):
    1. лексический анализ,
    2. синтаксический анализ,
    3. семантический анализ,
    4. оптимизации на абстрактном синтаксическом дереве,
    5. трансляция в промежуточное представление,
    6. машинно-независимые оптимизации,
    7. трансляция в конечное представление (машинный код),
    8. машинно-зависимые оптимизации.
- интерпретация или компиляция «на лету» промежуточного кода виртуальной машиной (при трансляции кода компилятором не в машинный код, а в код некоторой виртуальной машины),
- планирование исполнения машинного кода ядром операционной системы,
- исполнение кода программы на процессоре,

- исполнение кода функций библиотек поддержки времени исполнения, используемых в программе,

- исполнение кода системных вызовов, используемых в программе.

На каждой из перечисленных стадий существует множество факторов, способных в конечном счете повлиять на производительность программы. Под контролем программиста непосредственно целевой программы находится лишь одна группа факторов. За остальные группы факторов ответственны разработчики соответствующих инструментов и вспомогательных программ или их частей: виртуальных машин, библиотек поддержки времени исполнения, ядер операционных систем и непосредственно самих процессоров и других физических компонентов, способных влиять на производительность исполняемой программы.

В данной работе предлагается провести исследование, связанное с анализом влияния на производительность программ групп факторов в рамках стадий написания кода и его компиляции (в терминах теории компиляторов — в рамках стадии анализа, опуская стадию синтеза).

Хочется сразу отметить, что результаты такого анализа могут быть использованы как разработчиками целевых программ, так и разработчиками языка программирования, на котором эти программы составляются.

В качестве языка программирования, программы на котором и компилятор которого будут исследоваться, был выбрал Kotlin, как один из наиболее молодых, быстро развивающихся и ещё не достаточно исследованных языков.

Таким образом, целью данной магистерской диссертации является разработка набора инструментов для анализа исходного кода программ на ЯП Kotlin и выявления потенциальных проблем производительности в них, а также проведение исследования по данной теме с использованием разработанного набора инструментов.

Практическая значимость работы заключается в получении по результатам исследования и разработки:

1. сгруппированного списка файлов с исходным кодом из достаточно объемного набора данных, являющихся с точки зрения тех или иных алгоритмов аномальным;
2. набора инструментов, позволяющего получать аналогичный список файлов на заданном проекте.

Первый результат должен стать важным и полезным в первую очередь разработчикам языка программирования Kotlin; второй же — пользователям языка — программистам, использующих для разработки своих проектов язык программирования Kotlin.

1 АНАЛИЗ ПРДЕМЕТНОЙ ОБЛАСТИ И ПОСТАНОВКА ЗАДАЧИ

1.1 Анализ предметной области

Предметной областью, как уже было сказано, является анализ исходного кода на языке программирования Kotlin (и других представлений этого исходного кода) с целью обнаружения потенциальных проблем производительности в программах, написанных на нём.

Компилятор ЯП Kotlin транслирует исходный код в промежуточный код (называемый также байт-кодом) — набор инструкций для виртуальной машины ЯП Java — JVM. Исследование предлагается проводить в рамках представлений исходного кода, ответственность за которые лежит непосредственно на стороне компилятора Kotlin (дальнейшая интепретация или компиляция «на лету» в машинный байт-код JVM и исполнение кода рассматриваться не будут, т. к. данные стадии лежат за рамками ответственности компилятора Kotlin).

В компиляторе Kotlin производятся следующие преобразования исходного кода:

1. Преобразование исходного кода в набор лексем (токенизация или лексический анализ);
2. Построение дерева разбора на основе набора лексем (в компиляторе Kotlin данное дерево именуется как PSI — Program Structure Interface — по общепринятой терминологии его можно назвать конкретным синтаксическим деревом);
3. Генерация байт-кода JVM на основе дерева разбора PSI.

Стоит отметить, что построение дерева разбора PSI возможно на любом исходном коде. Для синтаксически некорректных конструкций (тех, которые не вписываются в грамматику ЯП Kotlin) в дереве разбора будет создаваться узел специального типа; в то время, как генерация байт-кода JVM по исходному коду с синтаксически и/или семантически некорректными конструкциями невозможна.

К сожалению, на данный момент в компиляторе Kotlin нет представления исходного кода в виде абстрактного синтаксического дерева при компиляции в

целевой язык — JVM байт-код. Генерация JVM байт-кода производится напрямую по дереву разбора PSI.

Дерево разбора PSI, как уже было сказано, представляет из себя конкретное синтаксическое дерево. В отличие от абстрактного синтаксического дерева в нём в том числе содержаться узлы, несущие исключительно синтаксический характер (не несущие семантический характер с точки зрения целевого языка — например, отступы, пробелы, комментарии и т. д.). Листья данного дерева, как правило, соответствуют конкретным лексемам (таким образом, список листьев данного дерева является набором лексем, которые в свою очередь являются результатом работы лексического анализатора в компиляторе Kotlin). Корень дерева разбора PSI является узлом с типом FILE, который представляет весь исходный код анализируемого файла. Таким образом, по данному дереву разбора может быть однозначно восстановлен исходный код программы.

1.2 Постановка задачи

После освещения предметной области — того, как происходит компиляция кода на ЯП Kotlin в компиляторе, и в каких представлениях он может находиться, можно определить задачу для исследования и разработки.

Задачей будет являться анализ исходного кода на ЯП Kotlin в представлении в виде дерева разбора PSI, а также сгенерированного по нему JVM байт-кода. Такой анализ должен будет быть направлен на обнаружение потенциальных проблем производительности в программах с анализируемым кодом.

Потенциальные проблемы производительности могут быть выражены в слишком объемном сгенерированном JVM байт-коде, в большом количестве повторяющихся JVM инструкций или наборов инструкций, в слишком объёмном и/или глубоком дереве разбора PSI, в большом количестве повторяющихся узлов или наборов узлов, а также в нетипичном (по меркам анализируемого набора данных) JVM байт-коде или дереве разрбора PSI. Будет называть файлы с исходным кодом, которым соответствует такое дерево разбора PSI и/или JVM байт-код, аномальными.

В первую очередь предлагается провести исследование на некотором эталонном наборе данных: определить способы анализа, оптимальные параметры алгоритмов, обсудить полученные результаты с экспертами, получить результаты при некоторых заданных ограничениях и других параметрах алгоритмов.

На втором этапе предлагатеся разработать набор инструментов для запуска поиска аномальных файлов на заданном проекте (наборе исходых кодов). Предполагается, что такие файлы будут интересны программистам, занимающихся разработкой этого проекта, как файлы имеющие потенциальные проблемы производительности.

Разработчикам же ЯП Kotlin предполагается, что будут интересны следующие результаты:

- полный набор найденных файлов-аномалий с исходным кодом на эталонном наборе данных: предлполагается, что такие файлы могут способствовать переосмыслению некоторых дизайнерских решений в конструкциях языка, либо разработке новых конструкций — вероятно, код таких файлов будет трактоваться как нерациональное использование конструкций языка;

- набор файлов с исходным кодом, сгенерированный JVM байт-код по которым считается аномальным, а дерево разбора PSI не превышает некоторые заданные ограничения (на глубину, либо на количество узлов), такой случай может соответствовать одному из двух вариантов:

    1. по заданной конструкции было либо сгенерировано больше JVM байт-кода, чем ожидалось, либо был сгенерирован нетипичный (по меркам анализируемого набора данных) JVM байт-код, что может свидетельствовать о наличии каких-либо проблем в кодогенераторе компилятора Kotlin;
    2. по заданной конструкции было сгенерировано большое количество JVM байт-кода вследствие удачной реализации данной конструкции: код, который предполагает выполнение большого числа действий, удалось

записать достаточно коротко (данный случай, хоть и имеет позитивный характер, также должен быть интересен разработчикам компилятора Kotlin — можно перенять опыт реализации данных конструкций для других конструкций);

- набор файлов с исходным кодом, дерево разбора PSI которых считается аномальным, а размер JVM байт-кода не превышает некоторые заданные ограничения: гипотетически такой случай также должен нести позитивный характер — по аномальному дереву разбора PSI был сгенерирован достаточно небольшой JVM байт-код, что безусловно должно позитивно сказаться на производительности программы (но такие файлы также могут быть интересны разработчикам компилятора Kotlin: можно также перенять опыт реализации данных конструкций, и, помимо этого, слишком большая разница в размере дерева разбора PSI и JVM байт-кода может гипотетически соответствовать некорректной генерации кода по каким-либо конструкциям).

Предполагается и следует из вышеизложенного, что анализ исходного кода и JVM байт-кода будет статическим (без фактического запуска программы), т. к. анализ поведения программы во время исполнения выходит за обнозначенные рамки: целью исследования является анализ исходного кода в разных представлениях.

Для анализа поведения программы во время исполнения существует множество инструментов. Одни из самых популярных — профилировщики. Они осуществляют сбор характеристик работы программы: времени выполнения отдельных фрагментов, числа верно предсказанных условных переходов, числа кэш-промахов и т. д. Данные характеристики могут быть так же использованы для оценки производительности программы в целом и для осуществления её дальнейшей оптимизации. Но в данном способе поиска потенциальных проблем производительности по сравнению с предложенным способом есть ряд отличий:

- необходимость фактического запуска программы в тестовой среде, что, как правило, является более сложно организуемым, чем статический анализ кода;

- оценка лишь фактического времени выполнения части кода (как правило, замеряется время и другие характеристики процесса исполнения соответствующего JVM байт-кода), без привязки к конкретной стадии преобразования анализируемого кода, которая могла повлечь найденную проблему (по этой причине динамический анализ кода будет мало полезен для разработчиков языка);

- более позднее обнаружение проблемы, как правило, может повлечь большие убытки различного характера (стадия написания кода и его компиляции в отличие от стадии исполнения является самой ранней в этапе программирования при разработке ПО).

По вышеизложенным причинам для решения обозначенной задачи и был выбран статический анализ кода.

2 ОБЗОР ИЗВЕСТНЫХ МЕТОДОВ И СРЕДСТВ РЕШЕНИЯ ЗАДАЧИ

2.1 Определение области решения задачи

Как уже было отмечено, объектом поиска будут являться файлы с исходным кодом на ЯП Kotlin, PSI дерево разбора и сгенерированный JVM байт-код которых, по тем или иным параметрам являются аномальными (нетипичными), поскольку предполагается, что аномальность будет заключается в слишком объемном JVM байт-коде и слишком объемном и/или глубоком дереве разбора PSI.

Задача обнаружения аномалий уже была поставлена в области машинного обучения. Для её решения существует множество техник. Рассмотрим эти техники, оценим их пригодность для поставленной задачи и определим условия для их применения.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. …