DOI: 10.18137/RNU.V9187.23.04.P.171 УДК 004.432.42

В.А. Телегин

ТЕХНОЛОГИИ AHEAD-OF-TIME И JUST-IN-TIME

Аннотация. Статья посвящена исследованию AOT- и JIT-технологий. Обосновывается актуальность и значимость темы исследования. Существует два способа компиляции приложения: своевременная компиляция (JIT) или компиляция с опережением времени (AOT). Первый вид – это режим по умолчанию, который используется виртуальной машиной Java Hotspot для преобразования байт-кода в машинный код во время выполнения. Второй вид поддерживается новым компилятором GraalVM и позволяет статически компилировать байт-код непосредственно в машинный код во время сборки. Рассматриваются основные различия между этими двумя стратегиями компиляции.

Ключевые слова: компилятор, программирование, код, программа, языки программирования, своевременная компиляция, JIT-технология, компиляция с опережением времени, AOT-технология, приложения.

V.A. Telegin

AHEAD-OF-TIME AND JUST-IN-TIME TECHNOLOGIES

Abstract. The article focuses on the study of Ahead-of-Time and Just-in-Time technologies. The author substantiates the relevance and significance of the research topic. There are two ways to compile an application using JIT or AOT compilation. The first is the default mode and is used by the Java Hotspot virtual machine to convert bytecode to machine code at runtime. The latter is supported by the new GraalVM compiler and allows bytecode to be statically compiled directly into machine code at build time. In this article, the author discusses the main differences between these two compilation strategies.

Keywords: compiler, programming, code, program, programming languages, JIT technology, AOT technology, applications.

Введение

Компиляция программы означает преобразование исходного кода с языка программирования высокого уровня, такого как Java или Python, в машинный код. Машинный код – это низкоуровневые инструкции, адаптированные для выполнения в конкретном микропроцессоре. Компиляторы – это программы, предназначенные для эффективного выполнения этой задачи. Цель компилятора – создать согласованный исполняемый файл скомпилированной программы. Согласованный исполняемый файл соответствует спецификации, написанной в исходном коде, выполняется быстро и является безопасным [1].

Компиляторы выполняют несколько оптимизаций на этапе генерации машинного кода. Например, большинство компиляторов выполняют встраивание, развертывание цикла и частичную оценку во время компиляции, и это лишь некоторые из них. Количество и сложность этих оптимизаций значительно увеличились за последние десятилетия.

В Java есть две основные стратегии компиляции: компиляция точно в срок и компиляция с опережением времени. Термины Ahead-of-Time (далее – AOT) и Just-in-Time (далее – JIT) относятся к тому, когда выполняется компиляция: время (time), упоминаемое в этих терминах, является временем выполнения, то есть JIT-компилятор компилирует программу во время ее выполнения, AOT-компилятор компилирует программу до ее запуска.

Опишем различия между этими двумя подходами.

Телегин Валентин Александрович
технический директор департамента по разработке мобильных приложений, ООО «Ростелеком Информационные Технологии», Москва. Сфера научных интересов: ИТ, разработка ПО, разработка мобильных приложений. Автор четырех опубликованных научных работ.
Электронный адрес: valentin.telegin.it@gmail.com

Технологии AOT и JIT

Все технологии на опережение – это процесс компиляции языка более высокого уровня или промежуточного языка в собственный машинный код, который зависит от системы.

При использовании **AOT-технологии** компиляция происходит только один раз – во время сборки проекта, и не нужно отправлять HTML-шаблоны и компилятор Angular всякий раз, когда вводится новый компонент. К примеру, он также может минимизировать размер приложения [2]. Браузеру не нужно компилировать код во время выполнения, он может непосредственно отображать приложение немедленно, не дожидаясь предварительной компиляции приложения, таким образом, он обеспечивает более быстрый рендеринг компонентов. Компилятор AOT обнаруживает ошибку шаблона раньше. Он обнаруживает ошибки привязки шаблона и сообщает о них на этапах сборки, прежде чем пользователи смогут их увидеть. AOT обеспечивает лучшую безопасность. Он компилирует HTML-компоненты и шаблоны в файлы JavaScript задолго до их отправки на дисплей клиента. Таким образом, нет шаблонов для чтения и нет рискованной оценки HTML или JavaScript на стороне клиента, что уменьшит вероятность атак с использованием SQL-инъекций [3].

Для запуска приложения сборку исходного кода делают в три этапа, которые включают анализ кода, генерацию кода и проверку типа шаблона. В конце этого процесса размер пакета будет намного меньше размера пакета JIT-компилятора.

Компилятор JIT обеспечивает компиляцию во время выполнения программы, то есть code get компилируется, когда это необходимо, а не во время сборки. Компилятор JIT компилирует каждый файл отдельно в основном в браузере. Таким образом, нет необходимости создавать проект заново после изменения кода. Бóльшая часть компиляции выполняется на стороне браузера, поэтому компиляция займет меньше времени. Компилятор JIT лучше всего подходит, когда, к примеру, приложение находится в локальной разработке. Изначально компилятор отвечал за преобразование языка высокого уровня в машинный язык, который затем был бы преобразован в исполняемый код. Компилятор JIT компилирует код во время выполнения, что означает, что вместо интерпретации байтового кода во время сборки он будет компилировать байтовый код при вызове этого компонента. В случае JIT не весь код компилируется в начальный момент; будут скомпилированы только необходимые компоненты, которые понадобятся при запуске приложения. Затем, если функциональность необходима в проекте, а ее нет в скомпилированном коде, эта функция или компонент будут скомпилированы. Этот процесс поможет снизить нагрузку на центральный процессор и ускорить рендеринг вашего приложения. Также можно просматривать исходный код и ссылаться на него в режиме проверки.

Разница между выполнением приложений JIT и AOT

Виртуальная машина Java выполняет байт-код, который был сгенерирован на основе кода, написанного на Java (или на других поддерживаемых языках, таких как Kotlin). Этот байт-код обычно упаковывается в файл JAR, который среда выполнения может выполнить на любой платформе.

Во время выполнения часто используемые методы (горячие точки) идентифицируются и компилируются в машинный код. Это делается с помощью двух компиляторов: C1 (быстрая компиляция, низкая оптимизация) и C2 (медленная компиляция, высокая оптимизация). При таком подходе всегда компилируется наиболее производительный машинный код для конкретного варианта использования и платформы, на которой выполняется приложение.

Недостатком этого подхода является то, что при запуске виртуальная машина выполняет (медленный) байт-код во время «прогрева», пока не определит горячие точки и не достигнет идеального собственного скомпилированного кода. Этот процесс повторяется при каждом запуске.

Код Java также может быть скомпилирован в собственные приложения, например, с помощью GraalVM. При таком подходе ваш Java-код компилируется статически, и компилятор создает машинный код в исполняемом файле для конкретной платформы. Преимущество заключается в том, что байт-код не нужно интерпретировать во время выполнения, не нужно определять горячие точки и не требуется загрузка процессора для компиляции. Таким образом, приложение будет запускаться на полной скорости с заранее созданным собственным скомпилированным кодом.

Недостатки заключаются в том, что необходимо скомпилировать код для всех платформ, на которых необходимо запустить свое приложение, а среда выполнения не содержит исходного кода и не может использовать его для дальнейшей оптимизации поведения приложения на основе фактического использования [4].

Таким образом, мы имеем разные подходы и разную производительность.

Если весь код компилируется перед его запуском (AOT), как он может работать хуже, чем скомпилированный JIT-код? В этом заключается истинная сила подхода JIT – он может адаптировать скомпилированный машинный код для обработки данных или выполнять свои действия на основе реальных потребностей.

Это всего лишь несколько примеров того, что известно как спекулятивная оптимизация.

Если код был создан для обработки нескольких вариантов if или switch, но некоторые опции никогда не используются, они не будут скомпилированы в машинный код, что делает его меньше и быстрее [5].

Если позже выяснится, что необходимы некоторые из нескомпилированных опций, компилятор может создать новый машинный код для обработки метода наилучшим возможным способом. Компилятор заменит частные переменные, где это возможно, общедоступными, чтобы уменьшить потребность в методе получения для возврата значения атрибута. Таким образом, стоит сказать, что такая спекулятивная оптимизация может привести к увеличению производительности до 50 %.

Рассмотрим в Таблице плюсы и минусы AOT и JIT.

Таблица

Обзор минусов и плюсов AOT и JIT

AOT	JIT
Загрузка класса предотвращает встраивание метода (–)	Может использовать встраивание агрессивных методов (+)
Нет генерации байт-кода во время выполнения (–)	Может использовать генерацию байт-кода во время выполнения (+)
Отражение сложно (–)	Отражение (относительно) простое (+)
Невозможно использовать спекулятивную оптимизацию (–)	Может использовать спекулятивную оптимизацию (+)
Общая производительность, как правило, будет ниже (–)	Общая производительность, как правило, будет выше (+)
Полная скорость с самого начала (+)	Требуется время на прогрев (–)
Отсутствие затрат процессора на компиляцию кода во время выполнения (+)	Нагрузка на процессор при компиляции кода во время выполнения (–)

Источник: таблица составлена на основе [6].

Таким образом, и AOT, и JIT предоставляют хорошие способы выполнения кода Java. Но хотя AOT предлагает ряд преимуществ в отношении запуска, влияние JIT-компилятора не следует недооценивать для достижения наилучшей производительности кода во время выполнения.

Заключение

Вышесказанное позволяет сделать объективное заключение, что преимущество компиляторов в том, что сгенерированный код выполняется быстрее, потому что бóльшая часть работы должна быть выполнена только один раз во время компиляции (например, лексический анализ, проверка типов, оптимизация), поэтому во время выполнения это исключается, и результаты доступны при каждом выполнении [7]. Интерпретатору приходится каждый раз переделывать эту работу.

Проблема компиляторов AOT заключается в том, что им часто не хватает информации для выполнения агрессивной оптимизации; эта информация доступна только во время выполнения.

Компилятор JIT стремится осуществлять лучшую, более агрессивную, оптимизацию только во время выполнения. Однако полный цикл компиляции является дорогостоящим (отнимающим много времени).

Ключевым моментом является то, что время выполнения программы, как правило, соответствует принципу Парето: бóльшая часть времени выполнения программы (обычно) тратится на выполнение крошечного фрагмента кода. Такой код называется горячим. Принцип Парето говорит нам, что (обычно) программа содержит некоторый горячий код с информацией, доступной во время выполнения, и мы можем агрессивно оптимизировать такой горячий код и добиться значительного ускорения. Остальное интерпретируется. Медлительность интерпретации не имеет значения, потому что это лишь часть времени выполнения программы.

Литература

1. *Ахмедов Р.Х.* Обзор технологий JIT-компиляций // Инновации и инвестиции. 2023. № 4. С. 278–280. EDN ROBMIX.
2. *Lindholm T., Yellin F., Bracha G., Buckley A.* The Java® Virtual Machine Specification, Java SE. 8th edition. San Francisco. CA: Addison-Wesley Professional, 2014. 600 p. ISBN 978-0133905908.
3. *Duimering P.R., Safayeni F., Purdy L.* Integrated Manufacturing: Redesign the Organization before Implementing Flexible Technology // Sloan Management Review. 1993. Vol. 34 No. 4. Pp. 47–56. URL: https://sloanreview.mit.edu/article/integrated-manufacturing-redesign-the-organization-before-implementing-flexible-technology/ (дата обращения: 03.11.2023).
4. *Gunawardan K.D.* Introduction of Advanced Manufacturing Technology: A Literature Review // Sabaragamuwa University Journal. 2006. Vol. 6. No. 1. Pp. 116–134. URL: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=2932029 (дата обращения: 03.11.2023).
5. *Постнов С.С.* Обзор технологий JIT-компиляции // International Journal of Open Information Technologies. 2020. Т. 8. № 9. С. 8–17. EDN UFNVMC.
6. *Григорьев Е.А., Климов Н.С.* Использование Ahead-Of-Time компиляции в платформе .NET как альтернатива Just-In-Time компиляции // E-Scio. 2019. № 11 (38). С. 364–371. EDN EOMDHH.
7. *Жуйков Р., Мельник Д., Бучацкий Р.* Методы динамической и предварительной оптимизации программ на языке JavaScript // Труды Института системного программирования РАН. 2014. Т. 26. № 1. С. 297–314. EDN RUMWZD.

References

1. Akhmedov R.H. (2023) Overview of JIT compilation technologies. *Innovations and investments.* No. 4. Pp. 278–280. (In Russian).
2. Lindholm T., Yellin F., Bracha G., Buckley A. (2014) *The Java® Virtual Machine Specification, Java SE*. 8th edition. San Francisco. CA : Addison-Wesley Professional, 2014. 600 p. ISBN 978-0133905908.
3. Duimering P.R., Safayeni F., Purdy L. (1993) Integrated Manufacturing: Redesign the Organization before Implementing Flexible Technology. *Sloan Management Review*. Vol. 34 No. 4. Pp. 47–56. URL: https://sloanreview.mit.edu/article/integrated-manufacturing-redesign-the-organization-before-implementing-flexible-technology/ (accessed 03.11.2023).
4. Gunawardan K.D. (2006) Introduction of Advanced Manufacturing Technology: A Literature Review. *Sabaragamuwa University Journal*. Vol. 6. No. 1. Pp. 116–134. URL: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=2932029 (accessed 03.11.2023).
5. Postnov S.S. (2020) Review of JIT compilation technologies. *International Journal of Open Information Technologies*. Vol. 8. No. 9. Pp. 8–17. URL: https://elibrary.ru/UFNVMC (accessed 03.11.2023).
6. Grigoriev E.A., Klimov N.C. (2019) Using Ahead-Of-Time compilation in the platform .NET as an alternative to Just-In-Time compilation. *E-SCIO*. No. 11 (38). Pp. 364–371. URL: https://elibrary.ru/EOMDHH (accessed 03.11.2023).
7. Zhuikov R., Melnik D., Buchatsky R. (2014) Methods of dynamic and preliminary optimization of programs in the Java Script language. *Proceedings of the Institute for System Programming of the RAS*. Vol. 26. No. 1. Pp. 297–314. URL: https://elibrary.ru/RUMWZD (accessed 03.11.2023).