Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 24М.71-мм

Трефилов Степан Захарович

Создание универсального интерфейса на C++ для интеграции стороннего JIT компилятора в OpenJDK

Отчёт по учебной практике
в форме «Решение»

Научный руководитель:
Доцент кафедры системного программирования, к.ф.-м.н., Д. В. Луцив

Санкт-Петербург
2024

Оглавление

1. Введение

Java является широко используемым [13] индустриальным языком программирования. Java программы распространяются в виде class [8] файлов, содержащих байткод для виртуальной машины Java. Для достижения максимальной производительности исполнения Java байткода применяется техника JIT компиляции.

Популярной реализацией спецификации языка Java SE [14] является OpenJDK [11]. Он также является главной реализацией языка Java, и большое количество крупных компаний (например, Amazon, Red Hat, Azul) имеют свои дистрибутивы OpenJDK. Исходный код OpenJDK был открыт в 2007 году.

В OpenJDK применяется 2 компилятора: C1, также известный как клиентский, и C2, также известный как серверный. Несмотря на то, что C2 широко используется и работает хорошо, существуют ряд компаний, предлагающих альтернативные реализации Java с компиляторами, заменяющими C2. Данные альтернативные реализации, как правило, имеют компиляторы, генерирующие более производительный код, чем C2. Одной из таких альтернативных реализаций является GraalVM, разработчики которой добавили в OpenJDK интерфейс JVMCI.

Начиная с 9 версии языка Java в OpenJDK появился новый интерфейс JVMCI [5] (Java VM compiler interface) для встраивания в JDK стороннего компилятора, написанного на Java. Он позволяет относительно простым образом разрабатывать сторонние компиляторы никак не модифицируя OpenJDK, а также получать необходимую для компиляции информацию из HotSpot VM с помощью набора определенных классов и методов из модуля jdk.internal.vm.ci [6].

LLVM [10] является известным проектом с открытым исходным кодом, предоставляющем большой набор инструментов и модулей для разработки компиляторов. LLVM написан на C++, и, как следствие, работать с ним удобнее всего с помощью C++.

Но что если мы захотим использовать LLVM для реализации компилятора, совместимого с различными версиями Java? Можно либо изменить исходный код OpenJDK разных версий, добавив в него новый компилятор, либо использовать JVMCI и одну версию компилятора, но тогда придется делать вызовы из C++ в Java и наоборот для работы с JVMCI из LLVM компилятора. В таком случае, JVMCI становится лишним звеном между LLVM и виртуальной машиной.

В рамках данной работы было решено разработать новый компиляторный интерфейс вместо JVMCI, который будет предоставлять аналогичные возможности, но работать на уровне C++ кода виртуальной машины, а не Java кода.

2. Постановка задачи

Целью работы является разработка универсального компиляторного интерфейса в OpenJDK. Для реализации данной цели были поставлены следующие задачи.

1. Провести обзор JVMCI.
2. Спроектировать и разработать новый компиляторный интерфейс.
3. Провести аппробацию полученного решения.

3. Обзор предметной области

3.1. Компиляторы C1 и C2

C1 и C2 – стандартные компиляторы в OpenJDK. Они написаны на C++ и являются частью кодовой базы OpenJDK, в отличие от компиляторов на основе JVMCI (как, например, Graal Compiler), которые отделены от OpenJDK.

C1 (клиентский компилятор[2] HotSpot JVM) – JIT компилятор, нацеленный в первую очередь на быструю компиляцию и уменьшенное использование памяти (по сравнению с серверным компилятором HotSpot JVM). До версии Java 8, пользователь должен был выбирать, какой из двух компиляторов он хочет использовать, однако в современных версиях JDK оба компилятора, по умолчанию, работают совместно.

Многоуровневая компиляция[3] (multi–tiered compilation) – технология, направленная одновременно на ускорение работы HotSpot JVM и на улучшение качества производимого во время компиляции кода. Суть ее заключается в том, что сначала HotSpot JVM запускает клиентский компилятор, а далее, при необходимости, использует серверный. Подробнее, когда HotSpot JVM замечает, что метод часто используется, она отправляет его на компиляцию в C1. Это позволяет ускорить выполнение метода, а также собрать дополнительную метаинформацию, которая может помочь серверному компилятору произвести более качественный машинный код. Далее, если метод остается ”горячим”, он отправляется в C2 и компилируется для максимальной производительности.

C2 (серверный компилятор[7] HotSpot JVM, top-tier компилятор) – JIT компилятор, нацеленный в первую очередь на скорость работы производимого кода. Он затрачивает больше ресурсов, чем клиентский компилятор HotSpot JVM, однако используя метаинформацию времени исполнения, которую накапливает HotSpot JVM во время работы каждого конкретного метода, удается добиться относительно хорошей производительности производимого кода.

Несмотря на то, что C2 способен компилировать сравнительно хороший машинный код, существуют различные проекты, целью которых является замена C2 на другой, лучший компилятор. В частности, компилятор Graal, работающий в OpenJDK с помощью JVMCI интерфейса, является одной из таких замен.

3.2. JVMCI-интерфейс

Как уже было упомянуто ранее, JVMCI — интерфейс для встраивания в Java стороннего компилятора, написанного на Java. С точки зрения данного интерфейса, компилятор это наследник абстрактного класса JVMCICompiler[1], в котором определен

[1] JVMCICompiler.java

метод compileMethod. Каждый раз, когда OpenJDK требуется скомпилировать очередной метод, виртуальная машина вызывает компилятор с помощью compileMethod, передавая единственным аргументом объект типа CompilationRequest[2], содержащий информацию о том, какой метод необходимо скомпилировать. Также для случаев, когда компилятор поддерживает OSR[3] компиляции, CompilationRequest содержит номер байткода в методе, от которого нужно сделать компиляцию метода.

Для получения информации из виртуальной машины, компилятор может использовать классы, определенные в модуле jdk.internal.vm.ci [6]. Структурно, основными частями данного модуля:

- Директории meta и hotspot - содержат интерфейсы для работы с сущностями виртуальной машины, а также реализации этих интерфейсов для виртуальной машины HotSpot. Например:
  - HotSpotConstantPool - сущность для доступа к constant pool [9];
  - HotSpotResolvedJavaMethod - сущность для доступа к информации о конкретном методе;
  - HotSpotResolvedObjectType - сущность для доступа к информации о конкретном классе.
- Директория code - содержит классы, необходимые для установки скомпилированных методов в Code cache. Например:
  - BytecodeFrame - сущность, позволяющая описать, в каких регистрах или стековых слотах находятся конкретные локальные переменные, стековые переменные и мониторы. Данная информация необходима для деоптимизаций и сборки мусора;
  - CodeCacheProvider - сущность, с помощью которой можно взаимодействовать с Code Cache. Данная абстракция необходима для добавления скомпилированного кода для Java методов.
- Директории по названиям архитектур (в частности, amd64 и aarch64) - содержат классы для предоставления архитектурно специфичной информации. Например, с помощью методов класса AMD64.java[4] можно получить информацию о расширениях, поддержанных процессором, а также номера регистров архитектуры X86 с точки зрения HotSpot VM.

Большой плюс данного интерфейса заключается в том, что он позволяет подключить Java компилятор к любой актуальной сборке OpenJDK с помощью набора флагов. В частности, для Java 21 нужно сделать следующее:

[2] CompilationRequest.java
[3] On Stack Replacement
[4] AMD64.java

1. Собрать свой Java компилятор в виде Jar архива.

2. Добавить его в *classpath* целевого приложения.

3. Использовать флаги $-XX : +UseJVMCICompiler$ для включения JVMCI и $-Djvmci.Compiler$ для указания имени компилятора.

На текущий момент не все LTS версии OpenJDK имеют одинаковый JVMCI. В частности, Java 8 и 11 до сих пор активно используются и поддерживаются некоторыми большими компаниями, но JVMCI данных версий является сильно устаревшим, а для Java 8 вообще является экспериментальным и не является частью официального репозитория. Кроме того, JVMCI для версий Java ниже 21 не имеет поддержки архитектуры RISCV, хотя для Java 17 имеется реализация JEP 422.

Таким образом, хотя JVMCI и реализован во всех версиях Java начиная с 11, не для всех версий API и возможности интерфейса являются одинаковыми, что не позволяет назвать JVMCI универсальным интерфейсом.

4. Исследование накладных расходов JVMCI

Для определения того, насколько перспективно портировать в старые версии JDK современного JVMCI, было решено провести эксперимент.

4.1. Тестовые данные

Для обнаружения накладных расходов от использования JVMCI было решено поставить следующий эксперимент: на одних и тех же тестовых данных мы пробуем использовать JVMCI компилятор без ограничения компиляций, а также с ограничением компиляции JVMCI методов до уровня C1 компилятора. Данный эксперимент позволит приблизительно оценить, сколько вычислительных ресурсов можно сэкономить используя вместо JVMCI некоторый Ahead-of-Time скомпилированный интерфейс.

4.2. Схема экспериментов

Эксперимент осуществлялся на компьютере с процессором Intel Xeon Gold 6326 (частота 2,9 ГГц). В качестве тестовой Java использовался OpenJDK (версия OpenJDK 21.0.4).

Для осуществления замеров использовались нагрузки из SpecJVM (compiler.compiler) и Renaissance (dotty). Измерялись зависимость времени на выполнение одной итерации целевой нагрузки от общего времени выполнения приложения (время прогрева), а также общее количество методов, скомпилированных JVMCI компилятором.

Кроме того, отдельно проводились тесты для двух конфигураций:

1. Использовались все доступные 32 ядра процессора.
2. Использовались только 4 ядра процессора. Данная конфигурация ближе к средней виртуальной машине.

4.3. Анализ результатов

4.3.1. Количество скомпилированных методов

Рис. 1: Прогрев compiler.compiler, 4 ядра

JVMCI	Бенчмарк	Кол-во top-tier методов
По умолчанию	compiler.compiler	5004
В C1	compiler.compiler	4345
По умолчанию	dotty	15127
В C1	dotty	13550

После запуска целевых нагрузок в предложенных конфигурациях были получены результаты, которые можно видеть на таблице (рисунок 1). Не трудно заметить, что использование JVMCI может создавать на компилятор лишнюю нагрузку от 700 до 1500 методов. На выбранных нагрузках это порядка 10% от общего количества всех скомпилированных методов.

4.3.2. Время прогрева приложения

Как можно видеть на графиках (рисунки 2 и 3), на конфигурации с 4 ядрами наблюдается существенная разница в прогреве целевых нагрузок. В случае dotty время выполнения без накладных расходов на компиляцию JVMCI в среднем на 10% меньше, а в случае compiler.compiler приложение прогревается примерно на 10% быстрее. Также можно обратить внимание, что хотя при использовании 32 ядер время прогрева и не меняется в зависимости от компиляции JVMCI в C1 или в top-tier, использование процессора во время прогрева меньше, если компилировать JVMCI в C1 (рисунки 4 и 5).

Рис. 2: Прогрев compiler.compiler, 4 ядра

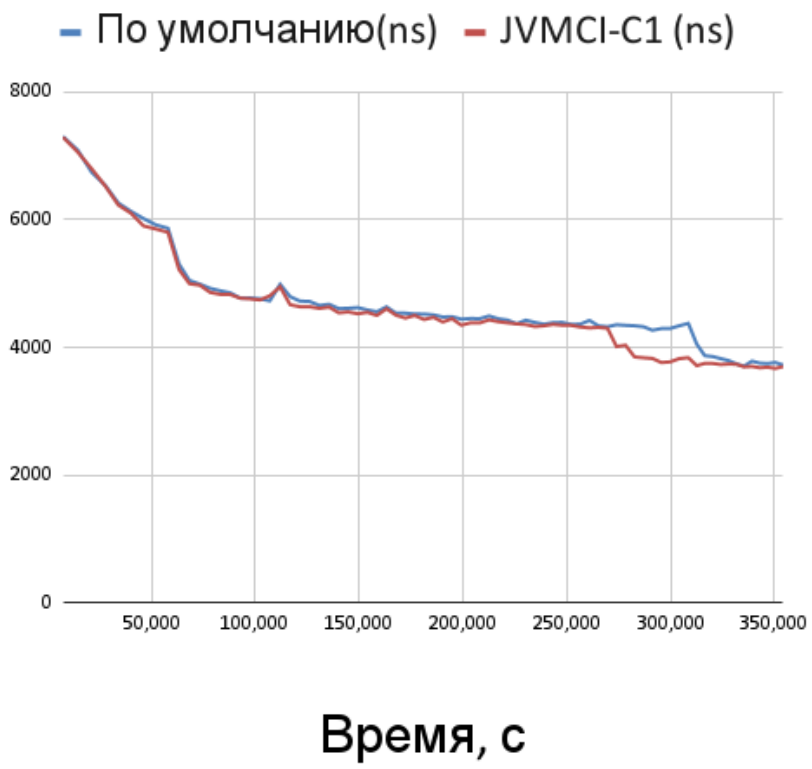


Рис. 3: Прогрев dotty, 4 ядра

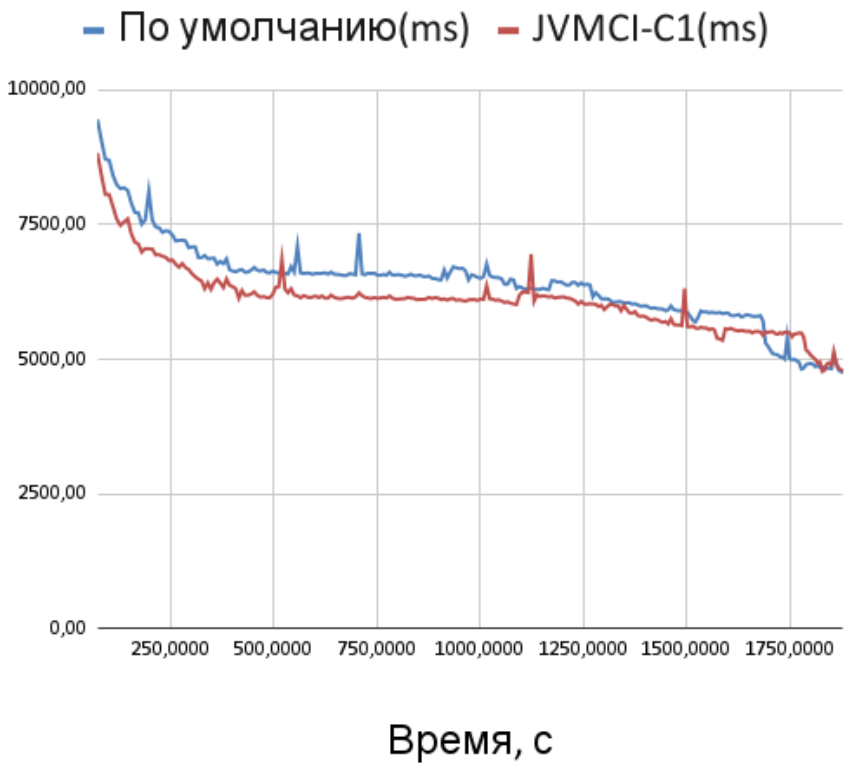


Рис. 4: Утилизация CPU compiler.compiler, 32 ядра

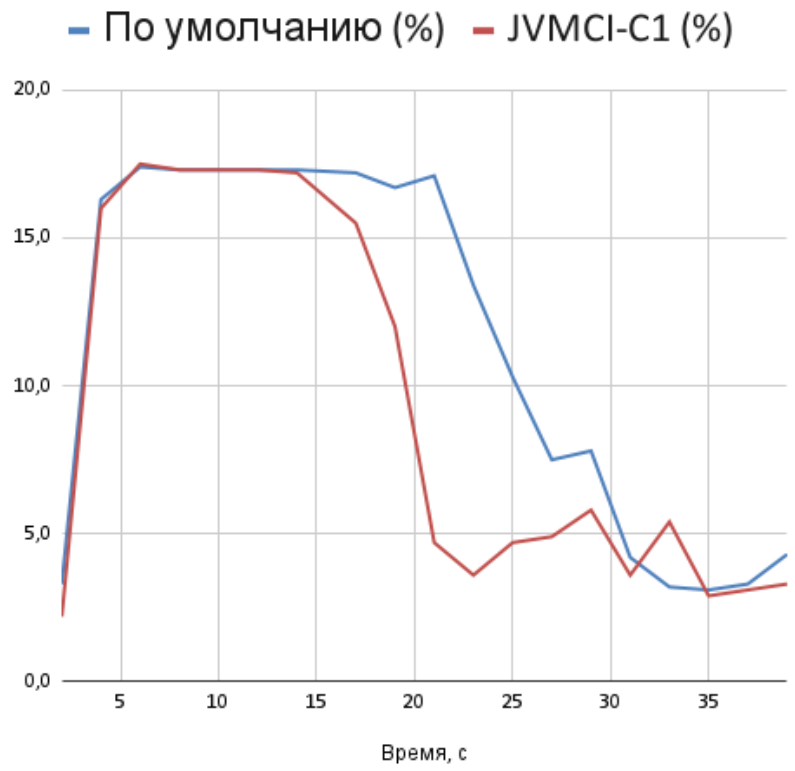
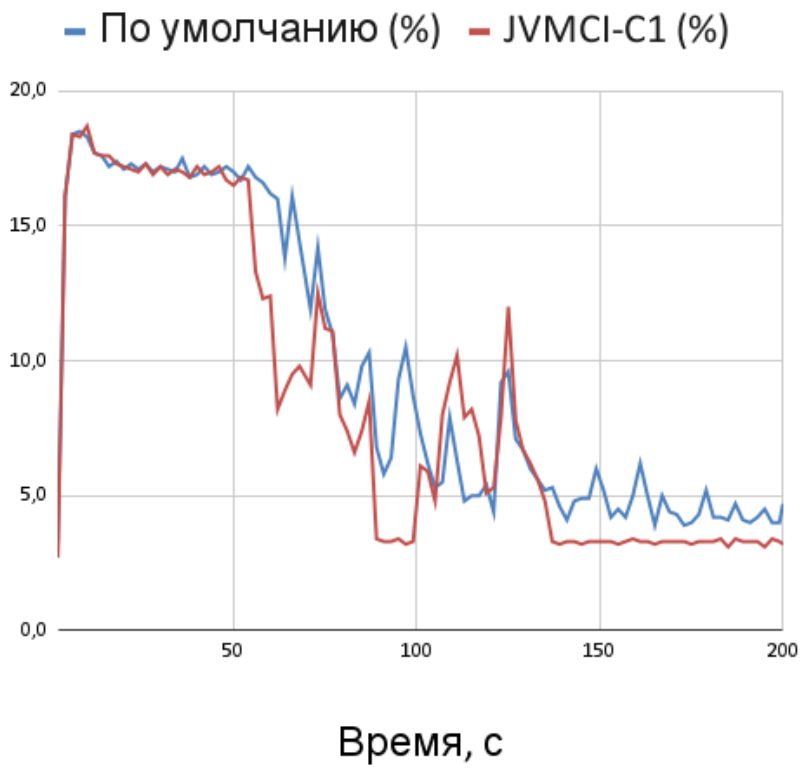


Рис. 5: Утилизация CPU dotty, 32 ядра



5. Проектирование интерфейса

Как можно видеть из эксперимента, проведенного выше, так как JVMCI является Java интерфейсом, это создает дополнительную нагрузку на JIT компилятор, работающий через JVMCI. В связи с этим, было принято решение не портировать актуальную версию JVMCI во все актульные версии Java, а разработать свой легковесный компиляторный интерфейс между виртуальной машиной и JIT компилятором, который, при этом, позволит разрабатывать и компилировать JIT компилятор отдельно от OpenJDK, как это позволяет делать JVMCI.

5.1. Языки описания данных

Для реализации нового универсального интерфейса необходимо выбрать некоторый формат данных, с помощью которого компилятор и виртуальная машина будут общаться. Так как JIT-компилятор должен работать параллельно с приложением и максимально быстро, вариант описания данных с помощью JSON или любого другого текстового представления был сразу отброшен. Было решено использовать некоторый независимый от языка бинарный протокол.

Первым в качестве такого формата рассматривался Protobuf[12]. Protocol Buffers – это не зависящий от языка расширяемый механизм сериализации структурированных данных.

Были выделены следующие плюсы использования Protobuf:

1. Удобный формат описания интерфейса в виде отдельного файла со структурами данных.

2. Хорошая поддержка во множестве языков, что, в частности, позволяет удобно общаться с виртуальной машиной из C++ компилятора.

Хотя Protobuf хорошо подходит для выбранной задачи, вместо него было решено использовать Cap‘n Proto[1]. Cap‘n Proto это очень похожая на Protobuf технология от того же автора, но является более легковесной. Это достигается за счет применения Zero-Copy подхода: Cap‘n Proto использует одно и то же представление данных при записи, чтении и передаче, что позволяет сэкономить время на сериализации и десериализации данных.

Также в качестве языка для описания данных рассматривался Flatbuffers[4]. Аналогично Cap‘n Proto, Flatbuffers позволяет сэкономить за счет применения Zero-Copy подхода, а также имеет поддержку в большом количестве языков.

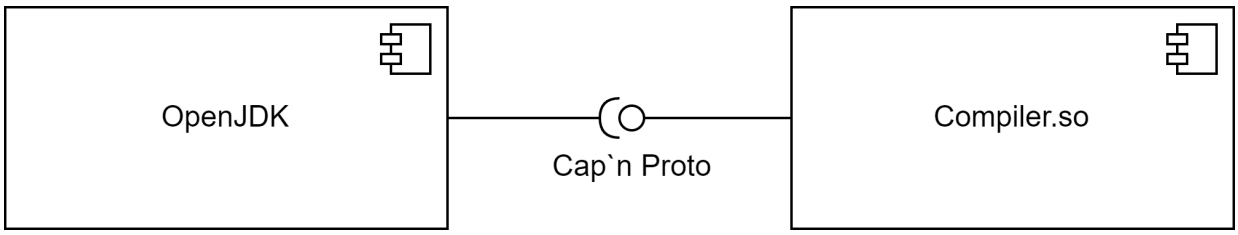
5.2. Описание интерфейса

Используя язык описания Cap‘n Proto, было создано описание нового компиляторного интерфейса для OpenJDK. В частности, было необходимо описать запросы для следующих сущностей виртуальной машины:

- Классы, методы и поля.
- Сущности из constant pool.
- Запросы на создания различных зависимостей для скомпилированного кода. Например, unique concrete method зависимость, которая позволяет с помощью CHA девиртуализовать виртуальный метод.
- Некоторые другие запросы, как, например, константное значение поля, если поле является константой.

Планируется, что компилятор компилятор будет подключаться к OpenJDK в виде динамической библиотеки.

Рис. 6: Взаимосвязь OpenJDK и компилятора через новый интерфейс



6. Заключение

В ходе работы на текущий момент были получены следующие результаты.

1. Проведен обзор предметной области.
2. Проведены эксперименты по оценке накладных расходов JVMCI при компиляции.
3. Проведены подготовительные работы по реализации интерфейса: выбрана технология реализации интерфейса и создано описание интерфейса.

В весеннем семестре планируется:

1. Разработать новый компиляторный интерфейс.
2. Провести аппробацию полученного решения: сравнить разработанное решение с JVMCI.

Список литературы

[1] Cap'n Proto введение. — URL: https://capnproto.org/ (online; accessed: 2024-12-19).

[2] Design of the Java HotSpot Client Compiler for Java 6. — 2008. — URL: https://dl.acm.org/doi/pdf/10.1145/1369396.1370017 (online; accessed: 2024-12-19).

[3] Efficient Code Management for Dynamic Multi-Tiered Compilation Systems. — 2014. — URL: https://dl.acm.org/doi/10.1145/2647508.2647513 (online; accessed: 2024-12-19).

[4] Flatbuffers документация. — URL: https://flatbuffers.dev/ (online; accessed: 2024-12-19).

[5] JEP 243: Java-Level JVM Compiler Interface. — 2022. — URL: https://openjdk.java.net/jeps/243 (online; accessed: 2024-12-19).

[6] JVMCI модуль OpenJDK. — URL: https://github.com/openjdk/jdk17u-dev/tree/master/src/jdk.internal.vm.ci/share/classes (online; accessed: 2024-12-19).

[7] The Java HotSpot Server Compiler. — 2001. — URL: https://www.usenix.org/legacy/event/jvm01/full_papers/paleczny/paleczny.pdf (online; accessed: 2024-12-19).

[8] Java SE спецификация. The class File Format. — URL: https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html (online; accessed: 2024-5-02).

[9] Java VM Specification глава 4. — URL: https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.4 (online; accessed: 2024-12-19).

[10] LLVM project. — URL: https://llvm.org/ (online; accessed: 2024-12-19).

[11] OpenJDK project. — URL: https://github.com/openjdk/ (online; accessed: 2024-12-19).

[12] Protocol Bruffers документация. — URL: https://protobuf.dev/ (online; accessed: 2024-12-19).

[13] TIOBE Index for December 2024. — URL: https://www.tiobe.com/tiobe-index/ (online; accessed: 2024-12-19).

[14] Спецификации Java SE. — URL: https://docs.oracle.com/javase/specs/jls/se20/html/index.html (online; accessed: 2024-12-19).