

Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 24М.71-мм

*Трефилов Степан Захарович*

# Универсальный интерфейс JIT компилятора для OpenJDK

Отчёт по производственной практике  
в форме «Производственное задание»

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

Санкт-Петербург  
2026

# Оглавление

<b>1. Введение</b>	<b>3</b>
<b>2. Постановка задачи</b>	<b>5</b>
<b>3. Обзор предметной области</b>	<b>6</b>
3.1. Компиляторы C1 и C2 . . . . .	6
3.1.1. ciEnv интерфейс . . . . .	7
3.2. Компилятор Graal . . . . .	8
3.2.1. JVMCI интерфейс . . . . .	8
3.3. OpenJ9 JIT server . . . . .	10
3.4. Языки описания данных . . . . .	12
<b>4. Проектирование и реализация интерфейса</b>	<b>13</b>
4.1. Общая схема решения . . . . .	13
4.2. Описание интерфейса . . . . .	14
4.3. Реализация NVMCI компилятора . . . . .	16
4.4. Реализация интерфейса на стороне компилятора . . . . .	18
<b>5. Апробация интерфейса</b>	<b>19</b>
5.1. Тестирование . . . . .	19
<b>6. Заключение</b>	<b>20</b>
<b>Список литературы</b>	<b>21</b>

# 1. Введение

Java является широко используемым [16] индустриальным языком программирования. Java программы распространяются в виде class [11] файлов, содержащих байткод для виртуальной машины Java. Для достижения максимальной производительности исполнения Java байткода применяется техника JIT компиляции.

Популярной реализацией спецификации языка Java SE [17] является OpenJDK [13]. 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 [8] (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` [9].

Но что если мы захотим использовать использовать не Java, а, например, C++, для реализации стороннего компилятора? В таком случае Java слой становится лишним и неудобным. А если мы захотим убрать компилятор из виртуальной машины и запустить в отдельном процессе? Тогда JVMCI нам не подойдет совсем.

В рамках данной работы было решено разработать новый компиля-

торный интерфейс на основе, который будет предоставлять возможности, аналогичные JVMCI, но работать на основе технологий сериализации, а не Java.

## 2. Постановка задачи

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

1. Провести обзор существующих Java JIT-компиляторов и их интерфейсов.
2. Спроектировать и разработать новый компиляторный интерфейс.
3. Провести апробацию интерфейса.

## 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[3] 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[5] (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 метаданную, 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[10] 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 метаданную времени исполнения, которую накапливает HotSpot JVM во время работы каждого конкретного метода, удастся 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.

### 3.1.1. ciEnv интерфейс

Для доступа к сущностям виртуальной машины, а также для установки скомпилированного кода, C1 и C2 используют интерфейс, входной точкой которого является класс ciEnv<sup>1</sup>.

В частности, ciEnv позволяет:

- Получить информацию о любых классах, методах и полях в виде объектов типа ciKlass, ciMethod и ciField соответственно.
- Получить из constant pool [12] классы, методы поля и константы в виде ci оберток.
- Установить скомпилированный код в Code cache с помощью метода ciEnv::register\_method(...).
- Добавить спекулятивные зависимости к скомпилированному коду и т.д.

Сильной стороной данного интерфейса является то, что он, с одной стороны, предоставляет компилятору всю необходимую для компиляции информацию, а с другой стороны, создает минимум не нужных промежуточных сущностей (да и создает их на специальной компиляторной арене), тем самым не расходуя лишнюю память.

Минусом данного интерфейса является то, что он является приватным в HotSpot VM. Если мы захотим реализовать компилятор на базе данного интерфейса, то нам придется реализовывать его внутри кодовой базы OpenJDK, и мы не сможем иметь компилятор в виде отдельного приложения. Это плохо как с точки зрения того, что усложняет

---

<sup>1</sup>[hotspot/share/ci/ciEnv.hpp](https://hotspot/share/ci/ciEnv.hpp)

тестирование, так и с точки зрения того, что мы не сможем вынести компилятор в отдельный процесс (или даже на отдельный сервер), что может быть удобно, если мы захотим переиспользовать один компилятор для нескольких виртуальных машин.

## 3.2. Компилятор Graal

Graal[7] - JIT компилятор, написанный на Java для замены C2 в HotSpotVM. Во многом Graal похож на C2, так как тоже использует в качестве промежуточного представления Sea of nodes[15], однако является более продвинутым компилятором и имеет, в отличие от C2, проприетарную версию.

В контексте данной работы Graal нам интересен тем, что для его создания в OpenJDK был реализован JVMCI.

### 3.2.1. 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`<sup>2</sup>, в котором определен метод `compileMethod`. Каждый раз, когда OpenJDK требуется скомпилировать очередной метод, виртуальная машина вызывает компилятор с помощью `compileMethod`, передавая единственным аргументом объект типа `CompilationRequest`<sup>3</sup>, содержащий информацию о том, какой метод необходимо скомпилировать. Также для случаев, когда компилятор поддерживает OSR<sup>4</sup> 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` [9]. Структурно, основными частями данного модуля:

---

<sup>2</sup>[JVMCICompiler.java](#)

<sup>3</sup>[CompilationRequest.java](#)

<sup>4</sup>[On Stack Replacement](#)

- 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 [12];
  - 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<sup>5</sup> можно получить информацию о расширениях, поддерживаемых 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 самым свежим сборкам OpenJDK с

---

<sup>5</sup>[AMD64.java](#)

помощью набора флагов, без необходимости перекомпиляции или изменения кодовой базы OpenJDK. В частности, для Java 21 нужно сделать следующее:

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 *-Djvmtci.Compiler* для указания имени компилятора.

Существенными же минусами данного интерфейса является то, что:

1. Так как компилятор (либо обертка, которая обращается к компилятору через JNI) должен быть написан на Java, он создает ощутимую лишнюю нагрузку на этапе прогрева приложения. В частности, компилятор компилирует сам себя, и тем самым конкурирует за ресурсы с пользовательским приложением. Под ресурсами понимается и время процессора, и память, так как JVMCI-ные объекты создаются в обычной Java куче.
2. Компилятор может загрязнять профиль пользовательского приложения, тем самым ухудшая возможности оптимизировать данный код.

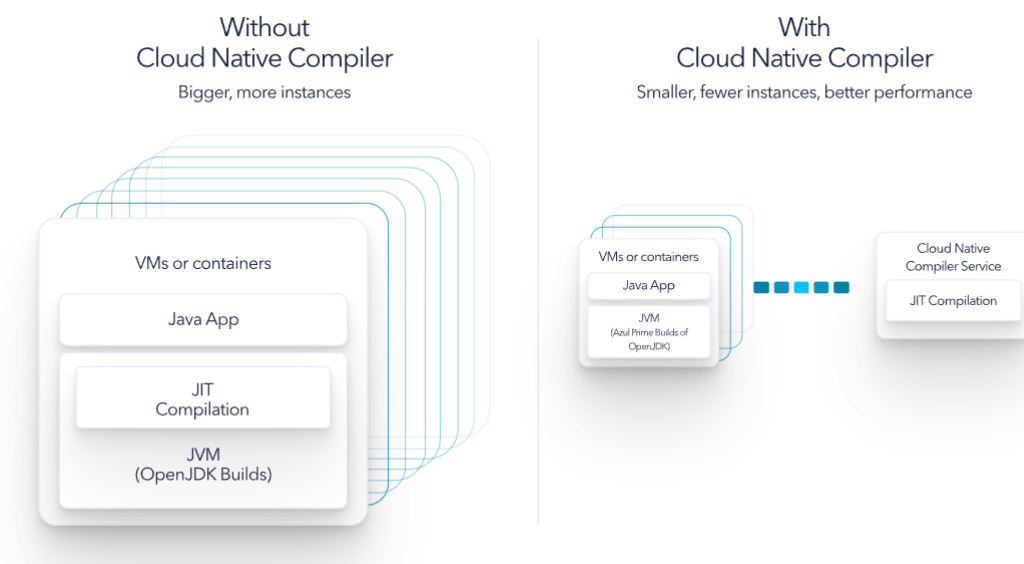
### 3.3. OpenJ9 JIT server

Кроме Graal, работающего с HotSpot VM через JVMCI, существуют и другие альтернативные JIT компиляторы для Java. В частности, существует OpenJ9 JIT Server - компилятор для Java, использующийся в виртуальной машине OpenJ9[4]. Данный компилятор для нас особенно интересен тем, что он может быть запущен отдельно от OpenJ9 VM на клиентской машине. Таким образом, можно создать некоторый сервер компиляции, пользователи которого будут подключаться через сетевой интерфейс.

Для общения через сетевой интерфейс в OpenJ9 определены сообщения<sup>6</sup>, которые передаются по специальному CommunicationStream<sup>7</sup>. CommunicationStream же строится поверх файлового дескриптора. Возможности данного интерфейса аналогичны JVMCI и ciEnv, описанным выше.

Можно также упомянуть, что кроме OpenJ9 подобную технологию отделения JIT компилятора в отдельный сервер поддерживает Azul Platform Prime[1] - еще одна альтернативная реализация Java VM Specification. В Azul Platform Prime данная технология носит название Cloud Native Compiler.

Рис. 1: Демонстрация Cloud Native Compiler



Несмотря на то, что реализация подобного сетевого компиляторного интерфейса не является целью данной работы, разработанный интерфейс также может быть использован для сетевого JIT компилятора, так как работает с сериализованными сообщениями, подобно OpenJ9 и Azul Platform Prime.

Именно по аналогии с данными Java реализациями было принято решение делать новый интерфейс как общение с помощью сообщений между виртуальной машиной и компилятором, где сообщения пред-

<sup>6</sup>[runtime/compiler/net/MessageTypes.hpp](#)

<sup>7</sup>[doc/compiler/jitserver/Networking.md](#)

ставляют из себя сериализованные структуры виртуальной машины.

### 3.4. Языки описания данных

Для реализации нового универсального интерфейса было 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[14]. 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[2]. 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.

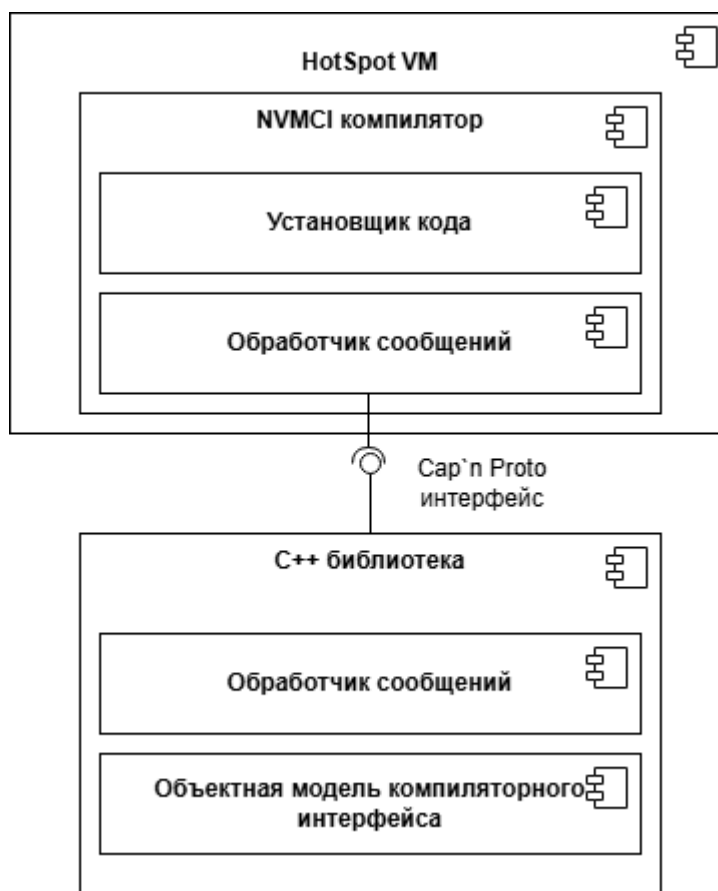
Также в качестве языка для описания данных рассматривался Flatbuffers[6]. Аналогично Cap'n Proto, Flatbuffers позволяет сэкономить за счет применения Zero-Copy подхода, а также имеет поддержку в большом количестве языков.

## 4. Проектирование и реализация интерфейса

### 4.1. Общая схема решения

Новый универсальный компиляторный интерфейс было решено реализовывать в виде набора связанных компонентов:

Рис. 2: Общая диаграмма компонентов решения



- NVMCI компилятор – входная точка для нового компиляторного интерфейса. С точки зрения HotSpot VM, это обычный компилятор, который работает точно так же как C1 или C2. На текущий момент это позволяет переиспользовать большое количество кода, в частности, весь `ciEnv`. В ходе поддержки NVMCI компилятора также были реализованы:
  - Обработчик сообщений – компонента, получающая от компилятора запросы и формирующая на них ответы.

- Установщик кода – компонента, переводящая сериализованный скомпилированный код в формат привычный для HotSpot VM, а затем устанавливающая скомпилированный код для исполнения.
- C++ библиотека. Внутри данной библиотеки были реализованы:
  - Обработчик сообщений – компонента, переводящая сериализованные Cap'n Proto сообщения в C++ объекты.
  - Объектная модель компиляторного интерфейса. Содержательно это просто набор классов, во многом аналогичны `ciEnv`, с которым компилятору удобно работать.

Общую диаграмму решения можно увидеть на Рис. 2.

## 4.2. 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 СНА девиртуализовать виртуальный метод и т.д.

Во многом описанный интерфейс дублирует JVMCI, однако есть и некоторые существенные различия:

- Используется меньшее количество различных сущностей. JVMCI активно использует наследование, что затрудняет использование

интерфейса. Судя по всему, изначально это было сделано для гибкости, так как подразумевалось, что JVMCI может быть реализован не только для виртуальной машины HotSpot. Так как мы хотим работать только с HotSpot, было решено избавиться от большого количества промежуточных сущностей в новом интерфейсе.

- Так как новый интерфейс основан на сериализации, а не Java объектах, каждой сущности представляющий объект, класс, метод и поле было добавлено специальное *id* поле, позволяющее однозначно идентифицировать некоторую сущность. Во время общения с виртуальной машиной компилятор должен запоминать *id* сущностей, так как некоторые ответы виртуальной машины могут ссылаться друг на друга по *id*. На текущий момент данные *id* выдаются объекту только на время одной компиляции, что создает необходимость на каждую новую компиляцию запрашивать кучу одинаковых объектов. Данный недостаток планируется исправить в будущем.

В результате был разработан файл в формате Cap'n Proto, содержащий полное описание интерфейса. Пример описания одного из сообщений можно видеть на Рис. 3.

Рис. 3: Описание сообщения типа ResolvedJavaType на языке Cap'n Proto

---

```
struct ResolvedJavaType {
  id @0: ProtoID;
  javaType @1: JavaType;
  superClassId @2: ProtoID; # ResolvedJavaType id
  interfaceIds @3: List(ProtoID); # ResolvedJavaType ids
  layoutHelper @4: Int32;
  superCheckOffset @5: Int32;
  union {
    instanceType @6: ResolvedInstanceType;
    arrayType @7: ResolvedArrayType;
  }
  accessFlags @8: AccessFlags;
}
```

---

### 4.3. Реализация NVMCI компилятора

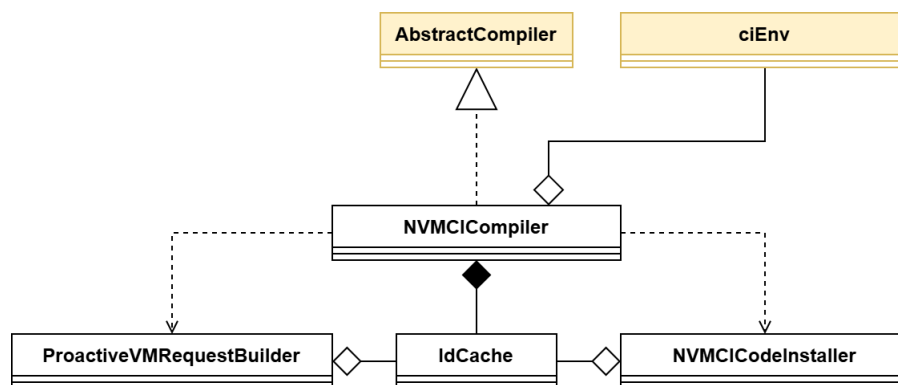
Для подключения компилятора к OpenJDK было решено реализовать NVMCI компилятор как объект, который, с одной стороны, реализует API стандартного абстрактного компилятора<sup>8</sup>, а с другой стороны, отправляет все запросы в реальный компилятор.

Таким образом, NVMCI компилятор выполняет 4 функции:

1. Создается в виртуальной машине вместо C2, реализуя аналогичный интерфейс.
2. Загружает реальный компилятор в виде динамической библиотеки.
3. Сериализует запросы и ответы между реальным компилятором из динамической библиотеки и виртуальной машиной.
4. Устанавливает скомпилированный код в виртуальную машину.

В качестве целевой версий OpenJDK для реализации NVMCI компилятора была выбрана наиболее свежая LTS версия Java под номером 25<sup>9</sup>.

Рис. 4: Диаграмма классов NVMCI компилятора



Общую диаграмму компилятора можно увидеть на Рис. 4. Желтые классы на данной диаграмме являются встроенными в HotSpot VM и

<sup>8</sup><share/compiler/abstractCompiler.hpp>

<sup>9</sup>[jdk25u на GitHub](#)

не были реализованы в рамках данной работы. Центральный класс реализованного NVMCI компилятора носит название NVMCICompiler. Он наследуется от стандартного HotSpot класса AbstractCompiler и реализует API аналогичный тому, что предоставляют C1 и C2. В частности, NVMCICompiler ответственен за:

1. Загрузку настоящего компилятора в виде динамической библиотеки. Для этого применяется обычный `dlopen`<sup>10</sup>.
2. Перенаправление запросов на компиляцию в настоящий компилятор, а также ответы на вопросы компилятора к виртуальной машине. Как раз для сериализации данных ответов и применяется класс ProactiveVMRequestBuilder, который переводит сущности из `ciEnv` в сериализованные Cap'n Proto сообщения.
3. Конвертацию Cap'n Proto сериализованного скомпилированного кода в структуры HotSpot VM и установку этого кода. Это делается с помощью класса NVMCICodeInstaller.

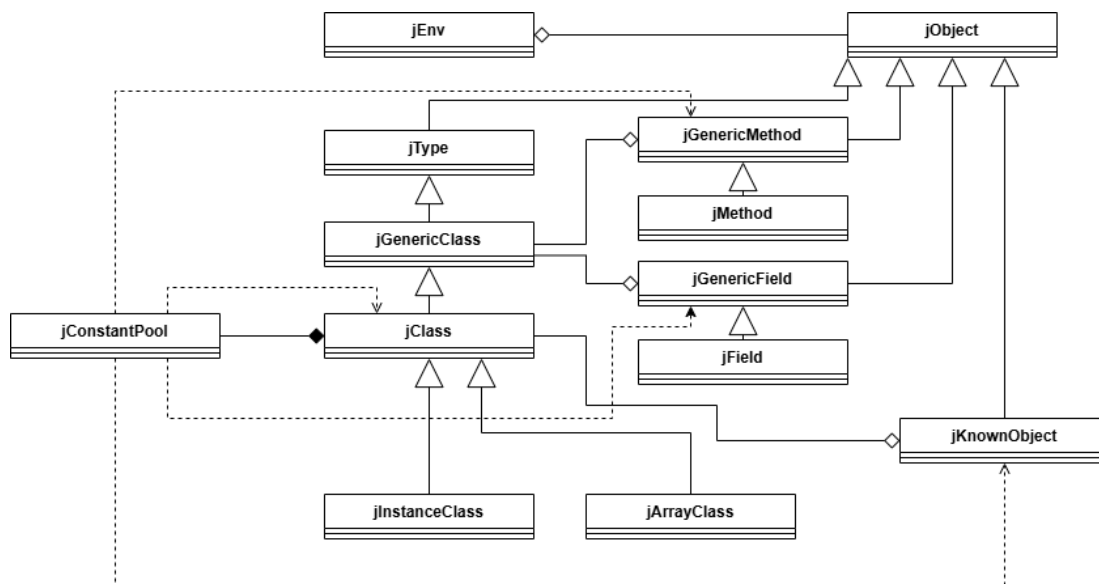
Так как внутри Cap'n Proto протокола для идентификации объектов применяются *id*, также NVMCICompiler содержит в себе IdCache, который просто сохраняет сопоставление между *id* и `ciEnv` объектами.

---

<sup>10</sup>[dlopen man](#)

## 4.4. Реализация интерфейса на стороне компилятора

Рис. 5: Диаграмма классов объектной модели компилятора



Для работы с предложенным интерфейсом на C++ была реализована библиотека, которая способна:

- Сериализовывать запросы к виртуальной машине.
- Десериализовывать приходящие ответы в объектную модель, подходящую для непосредственно компиляции Java программ (диаграмма на Рис. 6).

Данная объектная модель почти полностью является калькой с аналогичной из `ciEnv`, применяемой в OpenJDK C1 и C2 компиляторами.

## 5. Аппробация интерфейса

### 5.1. Тестирование

Для аппробации интерфейса был реализован компилятор-заглушка, способный получать и запрашивать сериализованные сообщения. На базе данной заглушки было написано порядка 300 различных тестов, проверяющих различные сценарии возможного взаимодействия между виртуальной машиной и компилятором.

Кроме того, для упрощения тестирования, был реализован специальный класс, позволяющий производить некоторые подготовительные действия с HotSpot VM перед инициализацией компиляции. Например, данная обертка для запуска тестов позволяет:

- Запустить тестовый метод в интерпретаторе некоторое количество раз.
- Скомпилировать тестовый метод в C1 компиляторе для сбора профиля.
- Вызвать компиляцию тестового метода как On Stack Replacement

Для верификации того, что сериализатор и десериализатор отработали как ожидается была реализована возможность распечатать все отправленные сообщения, а также содержимое *jEnv* – объекта содержащего все принятые от виртуальной машины сообщения в виде объектной модели компилятора. Проверка того, что вывод компилятора соответствует ожидаемому, осуществляется с помощью FileCheck<sup>11</sup>.

На текущий момент еще не все написанные тесты проходят. Это планируется исправить в следующем семестре.

---

<sup>11</sup>[LLVM FileCheck](#)

## 6. Заключение

В ходе работы на текущий момент были получены следующие результаты.

1. Проведен обзор предметной области. Были рассмотрены существующие JIT-компиляторы для Java (C1, C2, Graal), JVMCI интерфейс, а также языки описания данных Protobuf и Cap'n Proto.
2. Реализован новый компиляторный интерфейс:
  - Реализован NVMCI компилятор внутри HotSpot VM, сериализующий сущности виртуальной машины в Cap'n Proto сообщения и отправляющий их реальному компилятору.
  - Реализована C++ библиотека, позволяющая десериализовать Cap'n Proto сообщения в C++ объекты.
3. Написаны автоматические тесты, позволяющие проверить работоспособность NVMCI компилятора.

Планы на следующий семестр:

1. Добиться 100% прохождения написанных тестов.
2. Сравнить NVMCI компилятор с реализующим аналогичный интерфейс JVMCI агентом для демонстрации уменьшения накладных расходов.

## Список литературы

- [1] Azul Platform Prime. — URL: <https://www.azul.com/products/prime/> (online; accessed: 2024-4-24).
- [2] Cap'n Proto введение. — URL: <https://capnproto.org/> (online; accessed: 2024-12-19).
- [3] 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).
- [4] Eclipse OpenJ9: A Java Virtual Machine for OpenJDK that's optimized for small footprint, fast start-up, and high throughput. — URL: <https://github.com/eclipse-openj9/openj9> (online; accessed: 2025-3-16).
- [5] 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).
- [6] Flatbuffers документация. — URL: <https://flatbuffers.dev/> (online; accessed: 2024-12-19).
- [7] Graal Compiler. — URL: <https://www.graalvm.org/latest/reference-manual/java/compiler/> (online; accessed: 2025-3-22).
- [8] JEP 243: Java-Level JVM Compiler Interface. — 2022. — URL: <https://openjdk.java.net/jeps/243> (online; accessed: 2024-12-19).
- [9] JVMCI модуль OpenJDK. — URL: <https://github.com/openjdk/jdk17u-dev/tree/master/src/jdk.internal.vm.ci/share/classes> (online; accessed: 2024-12-19).
- [10] The Java HotSpot Server Compiler. — 2001. — URL: [https://www.usenix.org/legacy/event/jvm01/full\\_papers/paleczny/paleczny.pdf](https://www.usenix.org/legacy/event/jvm01/full_papers/paleczny/paleczny.pdf) (online; accessed: 2024-12-19).

- [11] 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).
- [12] 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).
- [13] OpenJDK project. — URL: <https://github.com/openjdk/> (online; accessed: 2024-12-19).
- [14] Protocol Buffers документация. — URL: <https://protobuf.dev/> (online; accessed: 2024-12-19).
- [15] Semantic reasoning about the sea of nodes. — URL: <https://www.graalvm.org/latest/reference-manual/java/compiler/> (online; accessed: 2025-3-22).
- [16] TIOBE Index for December 2024. — URL: <https://www.tiobe.com/tiobe-index/> (online; accessed: 2024-12-19).
- [17] Спецификации Java SE. — URL: <https://docs.oracle.com/javase/specs/jls/se20/html/index.html> (online; accessed: 2024-12-19).