JVM

**JVM —** ядро экосистемы Java. Она позволяет программам на базе Java следовать принципу “написал один раз, запустил где угодно”. Вы можете написать Java-код на одной машине и запустить его на любой другой благодаря JVM.

**Виртуальная машина** — это виртуальное представление физического компьютера. Виртуальную машину можно назвать гостевой, а физический компьютер, на котором она работает, — хост-машиной.

**Что такое виртуальная машина Java?**

В языках программирования, таких как C и C++, код сначала компилируется в машинный для конкретной платформы. Эти языки называются компилируемыми языками.

С другой стороны, в таких языках, как JavaScript и Python, компьютер выполняет инструкции напрямую, без необходимости компиляции. Эти языки называются интерпретируемыми.

Java использует комбинацию обоих методов. Код Java сначала компилируется в байтовый код и генерирует файл класса (.class). Этот файл класса затем интерпретируется виртуальной машиной Java для базовой платформы. Один и тот же файл класса может выполняться на любой версии JVM, на любой платформе и операционной системе.

Подобно обычным виртуальным машинам, JVM создает изолированное пространство на хост-машине. Это пространство может использоваться для выполнения Java-программ независимо от платформы или операционной системы компьютера.

Архитектура виртуальной машины Java

**JVM состоит из трех отдельных компонентов:**

* **загрузчик классов**
* **область памяти/данных среды выполнения**
* **Система выполнения**

Загрузчик классов:

Когда вы компилируете исходный файл .java, он преобразуется в байт-код как файл .class. Когда вы вызываете этот класс у себя в программе, загрузчик классов загружает его в основную память.

**Процесс загрузки класса состоит из трех этапов:**

* **Загрузка**
* **Связывание**
* **инициализация**

Загрузка

Загрузка включает бинарную репрезентацию (байт-код) класса или интерфейса с определенным именем и создание на его основе исходного класса или интерфейса.

**В Java доступны три встроенных загрузчика классов:**

* **Загрузчик классов начальной загрузки (Bootstrap Class Loader)** — корневой загрузчик классов. Это суперкласс загрузчика классов расширений, который загружает стандартные пакеты Java, такие как java.lang, java.net, java.util, java.io и так далее. Эти пакеты находятся внутри rt.jar и других основных библиотек, присутствующих в каталоге $JAVA\_HOME/jre/lib.
* **Загрузчик классов расширений (Extension Class Loader)** — подкласс загрузчика классов начальной загрузки и суперкласс загрузчика классов приложений. Он загружает расширения стандартных библиотек Java, которые присутствуют в каталоге $JAVA\_HOME/jre/lib/ext.
* **Загрузчик классов приложений (Application Class Loader)** — конечный загрузчик классов и подкласс загрузчика классов расширений. Он загружает файлы, которые находятся в пути к классам (classpath). По умолчанию путь к классу устанавливается как текущий каталог приложения. Путь к классу также можно изменить, добавив параметр командной строки -classpath или -cp.

Связывание

После загрузки класса в память происходит процесс связывания. Связывание класса или интерфейса предполагает объединение различных элементов и зависимостей программы.

**Связывание включает следующие шаги:**

* **Проверка.** На этом этапе проверяется структурная корректность файла .class путем проверки его на соответствие набору ограничений и правил. Если проверка по какой-либо причине завершается неудачей, выбрасывается исключение VerifyException.
* **Подготовка.** На этом этапе JVM выделяет память для статических полей класса или интерфейса и инициализирует их значениями по умолчанию.
* **Решение.** На этом этапе символические ссылки заменяются прямыми, присутствующими в пуле констант времени выполнения.

Инициализация

Инициализация включает выполнение метода инициализации класса или интерфейса (известного как <clinit>). Сюда может входить вызов конструктора класса, выполнение статического блока и присвоение значений всем статическим переменным. Это заключительный этап загрузки класса.

Область данных среды выполнения:

**В области данных среды выполнения есть пять компонентов:**

Область метода

Здесь хранятся все данные уровня класса, такие как пул констант времени выполнения, данные полей и методов, а также код методов и конструкторов.

Если памяти, доступной в области метода, недостаточно для запуска программы, JVM выдает ошибку OutOfMemoryError.

Область кучи

Здесь хранятся все объекты и соответствующие им переменные экземпляра. Это область данных времени выполнения, из которой выделяется память для всех экземпляров классов и массивов.

Куча создается при запуске виртуальной машины, и на каждую виртуальную машину приходится только одна область кучи.

Примечание: поскольку области метода и кучи совместно используют одну и ту же память для нескольких потоков, данные, хранящиеся здесь, не потокобезопасны.

Область стека

Всякий раз, когда в JVM создается новый поток, одновременно создается отдельный стек среды выполнения. Все локальные переменные, вызовы методов и частичные результаты хранятся в области стека.

Если для обработки в потоке требуется больший размер стека, чем доступно, JVM выдает ошибку StackOverflowError.

Для каждого вызова метода в памяти стека делается одна запись, которая называется фреймом стека. Когда вызов метода завершен, фрейм стека уничтожается.

**Фрейм стека разделен на три части:**

* **Локальные переменные.** Каждый фрейм содержит массив переменных, известных как его локальные переменные. Здесь хранятся все локальные переменные и их значения. Длина этого массива определяется во время компиляции.
* **Стек операндов.** Каждый фрейм содержит стек последним-вошел-первым-вышел (last-in-first-out, LIFO), известный как стек операндов. Он действует как рабочая область среды выполнения для любых промежуточных операций. Максимальная глубина этого стека определяется во время компиляции.
* **Данные фрейма.** Здесь хранятся все символы, соответствующие методу. Здесь также хранится информация о блоке catch на случай исключений.

Регистры счетчика программ

JVM поддерживает многопоточность. Каждый поток имеет собственный регистр счетчика программ для хранения адреса выполняемой в данный момент инструкции JVM. Как только инструкция выполнена, регистр обновляется следующей инструкцией.

Стеки нативных методов

JVM содержит стеки, которые поддерживают нативные методы, то есть такие методы, которые написаны на языке, отличном от Java, например C или C++. Для каждого нового потока также выделяется отдельный стек нативных методов.

Система выполнения:

Как только байт-код загружен в основную память, и подробная информация становится доступна в области данных среды выполнения, наступает следующий этап — запуск программы. Механизм выполнения делает это, выполняя код из каждого класса.

Однако перед выполнением программы байт-код необходимо преобразовать в инструкции машинного языка. В качестве механизма выполнения JVM может задействовать интерпретатор или JIT-компилятор.

Интерпретатор

Интерпретатор считывает и выполняет инструкции байт-кода строка за строкой. Из-за построчного выполнения интерпретатор работает сравнительно медленнее.

Еще один недостаток интерпретатора — при многократном вызове метода каждый раз требуется новая интерпретация.

JIT-компилятор

JIT-компилятор преодолевает недостаток интерпретатора. Механизм выполнения сначала использует интерпретатор для выполнения байт-кода, но когда он находит какой-то повторяющийся код, то задействует JIT-компилятор.

Затем JIT-компилятор компилирует весь байт-код и изменяет его на собственный машинный код. Этот собственный машинный код используется непосредственно для повторных вызовов методов, что повышает производительность системы.

**JIT-компилятор содержит следующие компоненты.**

* **Генератор промежуточного кода** — генерирует промежуточный код.
* **Оптимизатор кода** — оптимизирует промежуточный код для повышения производительности.
* **Генератор целевого кода** — преобразует промежуточный код в собственный машинный код.
* **Профилировщик** — находит горячие точки (код, который выполняется повторно).

Сборщик мусора

Сборщик мусора (Garbage Collector, GC) собирает и удаляет объекты без ссылок из области кучи. Это процесс автоматического восстановления неиспользуемой памяти во время выполнения путем уничтожения мусорных объектов.

Сборка мусора делает память Java эффективной, потому что удаляет объекты без ссылок из памяти кучи и освобождает место для новых объектов. **Она включает два этапа:**

* **Пометка** — на этом этапе GC идентифицирует неиспользуемые объекты в памяти.
* **Очистка** — на этом этапе GC удаляет объекты, идентифицированные на предыдущем этапе.

Сборка мусора выполняется JVM автоматически через регулярные промежутки времени и не требует отдельной обработки. Ее также можно запустить вызовом System.gc(), но выполнение не гарантируется.

**JVM содержит три различных типа сборщиков мусора:**

* **Последовательная сборка мусора.** Это самая простая реализация GC. Она предназначена для небольших приложений, работающих в однопоточных средах. Для сборки мусора используется один поток. Запуск приводит к событию “остановки мира”, когда все приложение приостанавливает работу. Аргумент JVM для запуск последовательного сборщика мусора: -XX:+UseSerialGC.
* **Параллельная сборка мусора.** Это реализация GC по умолчанию, также известная как сборщик пропускной способности. Для сборки мусора в нем используется несколько потоков, но работа приложения все равно приостанавливается при запуске. Аргумент JVM для параллельного сборщика мусора: -XX:+UseParallelGC.
* **Garbage First (G1).** G1 был разработан для многопоточных приложений с большим доступным размером кучи (более 4 ГБ). Он разбивает кучу на набор областей одинакового размера и использует несколько потоков для их сканирования. G1-сборщик определяет регионы с наибольшим количеством мусора и сначала выполняет сбор мусора в них. Аргумент JVM для этого сборщика мусора: -XX:+UseG1GC.

Нативный интерфейс Java

Иногда необходимо задействовать в работе нативный (не Java) код (например, написанный на C/C++). К примеру, в тех случаях, когда нужно взаимодействовать с физическим оборудованием или преодолевать ограничения по управлению памятью и производительности в Java. Java поддерживает выполнение нативного кода через нативный интерфейс Java (JNI).

JNI действует как мост для предоставления вспомогательных пакетов другим языкам программирования, таким как C, C++ и так далее. Это особенно полезно в тех случаях, когда нужно написать код, который не полностью поддерживается Java, например, некоторые специфичные для платформы функции могут быть написаны только на C.

Вы можете воспользоваться ключевым словом native, чтобы указать, что реализация метода будет предоставлена нативной библиотекой. Также потребуется вызвать System.LoadLibrary(), чтобы загрузить общую нативную библиотеку в память и сделать ее функции доступными для Java.

Нативные библиотеки методов

Нативные библиотеки методов — это библиотеки, написанные на других языках программирования, таких как C, C++ и ассемблер. Эти библиотеки обычно представлены в виде файлов .dll или .so. Такие библиотеки можно загружать через JNI.

Распространенные ошибки JVM

* **ClassNotFoundException.** Происходит, когда загрузчик классов пытается загрузить классы с помощью Class.forName(), ClassLoader.loadClass() или ClassLoader.findsystemclass(), но определение класса с указанным именем не найдено.
* **NoClassDefFoundError.** Происходит, когда компилятор успешно скомпилировал класс, но загрузчик классов не может найти файл класса во время выполнения.
* **OutOfMemoryError.** Происходит, когда JVM не может выделить объект из-за нехватки памяти, и сборщик мусора не может предоставить больше памяти.
* **StackOverflowError.** Происходит, если в JVM не хватает места при создании новых кадров стека во время обработки потока.