



 enhorse / java-interview Public[Code](#) [Issues 9](#) [Pull requests](#) [Actions](#) [Projects](#) [Security](#) [Insights](#) master ▾

...

[java-interview](#) / [jvm.md](#)

DanilPogozhev Убрано сокращение слова. Убрана лишняя буква в слове progr...

 History 4 contributors 238 lines (169 sloc) | 27.2 KB

...

JVM

- [За что отвечает JVM](#)
- [Classloader](#)
- [Области данных времени выполнения](#)
- [Frames](#)
- [Execution Engine](#)
- [Полезные ссылки](#)

За что отвечает JVM:

- Загрузка, проверка и исполнение байт кода;
- Предоставление среды выполнения для выполнения байт-кода;
- Управление памятью и очисткой мусора (Garbage collection);

Виртуальная машина Java (Java Virtual Machine) - это механизм, предоставляющий среду выполнения для управления Java-кодом или приложениями. Виртуальная машина является независимой оболочкой исполнения кода, благодаря которой возможен её запуск на любой ОС, без влияния ОС на выполняемую программу.

JVM работает с 2мя типами данных: примитивные типы (**primitive types**) и ссылочные типы (**reference types**).

Примитивы

JVM работает с примитивными значениями (целыми числами и числами с плавающей точкой). По сути, JVM - это 32-битная машина. Типы `long` и `double`, которые являются 64-битными, поддерживаются изначально, но занимают две единицы памяти в `frame's local` или стеке операндов, поскольку каждая единица составляет 32 бита. Типы `boolean`, `byte`, `short` и `char` имеют расширенный знак (кроме `char` с нулевым расширением) и работают как 32-разрядные целые числа, так же, как и типы `int`. Меньшие типы имеют только несколько специфических для типа инструкций для загрузки, хранения и преобразования типов. `boolean` значение работает как 8-битное `byte` значения, где 0 представляет значение `false`, а 1 - значение `true`.

Типы ссылок и значения

Существует три типа ссылочных типов: типы классов, типы массивов и типы интерфейсов. Их значения являются ссылками на динамически создаваемые экземпляры классов, массивы или экземпляры классов, которые реализуют интерфейсы соответственно.

[к оглавлению](#)

Classloader

Загрузчик классов является частью JRE, которая динамически загружает Java классы в JVM. Обычно классы загружаются только по запросу. Система исполнения в Java не должна знать о файлах и файловых системах благодаря загрузчику классов. Делегирование является важной концепцией, которую выполняет загрузчик. Загрузчик классов отвечает за поиск библиотек, чтение их содержимого и загрузку классов, содержащихся в библиотеках. Эта загрузка обычно выполняется «по требованию», поскольку она не происходит до тех пор, пока программа не вызовет класс. Класс с именем может быть загружен только один раз данным загрузчиком классов.

При запуске JVM, используются три загрузчика классов:

- **Bootstrap class loader** (Загрузчик класса Bootstrap)
- **Extensions class loader** (Загрузчик класса расширений)

- **System class loader (Системный загрузчик классов)**

Загрузчик класса Bootstrap загружает основные библиотеки Java, расположенные в папке `<JAVA_HOME>/jre/lib`. Этот загрузчик является частью ядра JVM, написан на нативном коде.

Загрузчик класса расширений загружает код в каталоги расширений (`<JAVA_HOME>/jre/lib/ext`, или любой другой каталог, указанный системным свойством `java.ext.dirs`).

Системный загрузчик загружает код, найденный в `java.class.path`, который сопоставляется с переменной среды `CLASSPATH`. Это реализуется классом `sun.misc.Launcher$AppClassLoader`.

Загрузчик классов выполняет три основных действия в строгом порядке:

- **Загрузка:** находит и импортирует двоичные данные для типа.
- **Связывание:** выполняет проверку, подготовку и (необязательно) разрешение.
 - **Проверка:** обеспечивает правильность импортируемого типа.
 - **Подготовка:** выделяет память для переменных класса и инициализация памяти значениями по умолчанию.
 - **Разрешение:** преобразует символические ссылки из типа в прямые ссылки.
- **Инициализация:** вызывает код Java, который инициализирует переменные класса их правильными начальными значениями.

Пользовательский загрузчик классов

Загрузчик классов написан на Java. Поэтому возможно создать свой собственный загрузчик классов, не понимая тонких деталей JVM. У каждого загрузчика классов Java есть родительский загрузчик классов, определенный при создании экземпляра нового загрузчика классов или в качестве системного загрузчика классов по умолчанию для виртуальной машины.

Что делает возможным следующее:

- **загружать или выгружать классы во время выполнения** (например, динамически загружать библиотеки во время выполнения, даже из ресурса HTTP). Это важная особенность для:
 - реализация скриптовых языков;
 - использование bean builders;
 - добавить пользовательскую расширение;

- позволяя нескольким пространствам имен общаться. Например, это одна из основ протоколов CORBA / RMI;
- **изменить способ загрузки байт-кода** (например, можно использовать зашифрованный байт-код класса Java);
- **модифицировать загруженный байт-код** (например, для переплетения аспектов во время загрузки при использовании аспектно-ориентированного программирования);

[к оглавлению](#)

Области данных времени выполнения

Run-Time Data Areas. JVM выделяет множество областей данных во время выполнения, которые используются во время выполнения программы. Некоторые участки данных созданы JVM во время старта и уничтожаются во время её выключения. Другие создаются для каждого потока и уничтожаются, когда поток уничтожается.

The pc Register (PCR)

Виртуальная машина Java может поддерживать много потоков исполнения одновременно. Каждый поток виртуальной машины Java имеет свой собственный регистр PC (program counter). В любой момент каждый поток виртуальной машины Java выполняет код одного метода, а именно текущий метод для этого потока. Если этот метод не является native, регистр pc содержит адрес инструкции виртуальной машины Java, выполняемой в настоящее время.

Коротко говоря: для одного потока, существует один PCR, который создается при запуске потока. PCR хранит адрес выполняемой сейчас инструкции JVM.

Java Virtual Machine Stacks

Каждый поток в JVM имеет собственный стек, созданный одновременно с потоком. Стек в JVM хранит frames. Стеки в JVM могут иметь фиксированный размер или динамически расширяться и сжиматься в соответствии с требованиями вычислений.

Heap

JVM имеет heap (кучу), которая используется всеми потоками виртуальной машины Java. Куча - это область данных времени выполнения, из которой выделяется память для всех экземпляров и массивов классов. Куча создается при запуске виртуальной машины. Хранилище для объектов восстанавливается автоматической системой управления данными (известной как сборщик мусора); объекты никогда не освобождаются явно. JVM не предполагает какого-либо конкретного типа системы автоматического управления хранением данных, и метод управления может быть выбран в соответствии с системными требованиями разработчика. Куча может иметь фиксированный размер или может быть расширена в соответствии с требованиями вычислений и может быть сокращена, если большая куча становится ненужной. Память для кучи не должна быть смежной.

Method Area

JVM имеет область методов, которая является общей для всех потоков. Она хранит структуры для каждого класса, такие как пул констант, данные полей и методов, а также код для методов и конструкторов, включая специальные методы, используемые при инициализации классов и экземпляров, и инициализации интерфейса. Хотя область метода является логически частью кучи, простые реализации могут не обрабатываться сборщиком мусора. Область метода может иметь фиксированный размер или может быть расширена в соответствии с требованиями вычислений и может быть сокращена, если большая область метода становится ненужной.

Run-Time Constant Pool

A run-time constant pool существует для каждого класса или интерфейса в рантайме и представленно constant_pool таблицей в *.class файле. Он содержит несколько видов констант: от числовых литералов, известных во время компиляции, до ссылок на методы и поля, которые должны быть разрешены во время выполнения. Сам run-time constant pool выполняет функцию, аналогичную функции таблицы символов для обычного языка программирования, хотя он содержит более широкий диапазон данных, чем типичная таблица символов. Каждый run-time constant pool отделён от JVM's method area. JVM создаёт run-time constant pool вместе с созданием class или interface.

Native Method Stacks

Реализация виртуальной машины Java может использовать обычные стеки, обычно называемые «стеки Си», для поддержки native methods (методов, написанных на языке, отличном от языка программирования Java).

[к оглавлению](#)

Frames

Frame используется для хранения данных и частичных результатов, а также для выполнения динамического связывания, возврата значений для методов и отправки исключений. Новый frame создается каждый раз, когда вызывается метод. Frame уничтожается, когда завершается вызов метода, является ли это завершение нормальным или резким (он генерирует неперехваченное исключение). Frames выделяются из стека потока, создающего frame. Каждый frame имеет свой собственный массив локальных переменных, свой собственный стек операндов и ссылку на пул констант во время выполнения класса текущего метода. Размеры массива локальных переменных и стека операндов определяются во время компиляции и предоставляются вместе с кодом для метода, связанного с фреймом. Таким образом, размер структуры данных, frame-а зависит только от реализации виртуальной машины Java, и память для этих структур может быть выделена одновременно при вызове метода.

Только один frame активен в любой точке данного потока управления - метода выполнения, и это frame называется текущим, а его метод известен как текущий метод. Класс, в котором определен текущий метод, является текущим классом. Операции над локальными переменными и стеком операндов обычно выполняются со ссылкой на текущий frame.

Frame перестает быть текущим, если его метод вызывает другой метод или если его метод завершается. Когда метод вызывается, новый frame создается и становится текущим, когда управление переходит к новому методу. При возврате метода текущий frame передает результат вызова метода, если таковой имеется, в предыдущий frame. Текущий frame затем отбрасывается, так как предыдущий frame становится текущим. Обратите внимание, что frame, созданный потоком, является локальным для этого потока и на него не может ссылаться ни один другой поток.

Локальные переменные

Каждый frame содержит массив переменных, известных как его локальные переменные. Длина массива локальных переменных frame определяется во время компиляции и предоставляется в двоичном представлении класса или интерфейса вместе с кодом для метода, связанного с frame-ом. Единичная локальная переменная может хранить значение типа: `boolean`, `byte`, `char`, `short`, `int`, `float`, `reference`, or `returnAddress`. Пара локальных переменных может хранить значение типов: `long` или `double`.

Локальные переменные адресуются путем индексации. Индекс первой локальной переменной равен нулю.

Значение типа `long` или типа `double` занимает две последовательные локальные переменные.

JVM использует локальные переменные для передачи параметров при вызове метода. При вызове метода класса все параметры передаются в последовательных локальных переменных, начиная с локальной переменной 0. При вызове метода экземпляра локальная переменная 0 всегда используется для передачи ссылки на объект, для которого вызывается метод экземпляра (`this` в Java). Любые параметры впоследствии передаются в последовательных локальных переменных, начиная с локальной переменной 1.

Стеки операндов (Operand Stacks)

Каждый frame содержит стек «последний вошел - первый вышел» (LIFO), известный как стек операндов. Максимальная глубина стека операндов frame-а определяется во время компиляции и предоставляется вместе с кодом для метода, связанного с frame-ом.

Стек операнда пуст при создании frame-а, который его содержит. JVM предоставляет инструкции для загрузки констант или значений из локальных переменных или полей в стек операндов. Другие инструкции JVM берут операнды из стека операндов, оперируют с ними и помещают результат обратно в стек операндов. Стек операндов также используется для подготовки параметров для передачи в методы и для получения результатов метода.

Для примера, инструкция `iadd` суммирует два `int`-вых значения. От стека операндов требуется, чтобы два `int`-вых значения были наверху стека. Значения удаляются из стека, операция `pop`. Суммируются и их сумма помещается в стек операндов.

Динамическое связывание (Dynamic Linking)

Каждый frame содержит ссылку на `run-time constant pool` для типа текущего метода для поддержки динамического связывания кода метода. Доступ к вызываемым методам и переменным осуществляется через символические ссылки из `class` файла. Динамическое связывание преобразует эти символьные ссылки на методы в конкретные ссылки на методы, загружая классы по мере необходимости для разрешения пока еще не определенных символов, и преобразует обращения к переменным в соответствующие смещения в структурах хранения, связанных с расположением этих переменных во время выполнения.

Позднее связывание методов и переменных вносит изменения в другие классы, которые метод использует с меньшей вероятностью нарушить этот код.

Нормальное завершение вызова метода

Вызов метода завершается нормально, если этот вызов не вызывает исключение, либо непосредственно из JVM, либо в результате выполнения явного оператора `throw`. Если вызов текущего метода завершается нормально, то значение может быть возвращено вызывающему методу. Это происходит, когда вызванный метод выполняет одну из инструкций возврата, выбор которых должен соответствовать типу возвращаемого значения (если оно есть).

Текущий `frame` используется в этом случае для восстановления состояния инициатора, включая его локальные переменные и стек операндов, с соответствующим образом увеличенным программным счетчиком инициатора, чтобы пропустить инструкцию вызова метода. Затем выполнение обычно продолжается в `frame` вызывающего метода с возвращенным значением (если оно есть), помещаемым в стек операндов этого `frame`.

Резкое завершение вызова метода

Вызов метода завершается преждевременно, если при выполнении инструкции JVM в методе выдает исключение, и это исключение не обрабатывается в методе. Выполнение команды `throw` также приводит к явному выбрасыванию исключения, и, если исключение не перехватывается текущим методом, приводит к неожиданному завершению вызова метода. Вызов метода, который завершается внезапно, никогда не возвращает значение своему вызывающему.

[к оглавлению](#)

Execution Engine

Байт-код, назначенный **run-time data areas**, будет выполнен **execution engine**. Механизм выполнения считывает байт-код и выполняет его по частям.

Interpreter

Интерпретатор интерпретирует байт-код быстро, но выполняется медленно. Недостаток интерпретатора заключается в том, что, когда один метод вызывается несколько раз, каждый раз требуется новая интерпретация.

JIT Compiler

JIT-компилятор устраняет недостатки интерпретатора. Механизм выполнения будет использовать помощь интерпретатора при преобразовании байт-кода, но когда он находит повторный код, он использует JIT-компилятор, который компилирует весь байт-код и изменяет его на собственный код. Этот нативный код будет использоваться непосредственно для повторных вызовов методов, которые улучшают производительность системы.

- Генератор промежуточного кода (Intermediate Code Generator). Производит промежуточный код.
- Code Optimizer. Отвечает за оптимизацию промежуточного кода, сгенерированного выше.
- Генератор целевого кода (Target Code Generator). Отвечает за генерацию машинного кода или родной код.
- Профилировщик (Profiler). Специальный компонент, отвечающий за поиск горячих точек, то есть, вызывается ли метод несколько раз или нет.

Garbage Collector

[к оглавлению](#)

Полезные ссылки:

- <https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-2.html>
- <https://www.developer.com/java/data/understanding-the-jvm-architecture.html>
- <https://dzone.com/articles/understanding-jvm-internals>

[к оглавлению](#)