tproger.ru

Как работает виртуальная машина Java — взгляд изнутри

Никита Прияцелюк

12-17 минут

Рассказывает Роман Иванов

Каждому Java-разработчику будет очень полезно понимать, что из себя представляет JVM, как в неё попадает код и как он исполняется. Статья больше подойдёт новичкам, но найти в ней что-то новое смогут и более опытные программисты. В статье вкратце описано, как устроен class-файл и как виртуальная машина обрабатывает и исполняет байт-код.

Основной задачей разработчиков Java было создание переносимых приложений. JVM играет центральную роль в переносимости — она обеспечивает должный уровень

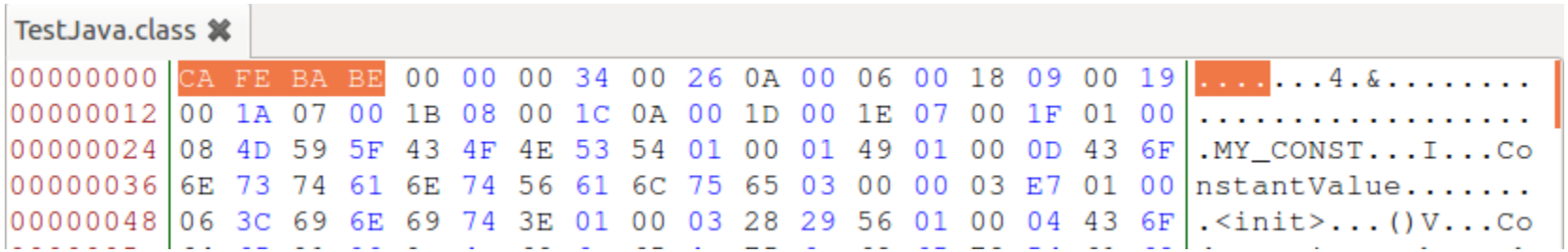
абстракции между скомпилированной программой и базовой аппаратной платформой и операционной системой. Несмотря на этот дополнительный «слой», скорость работы приложений необычайно высока, потому что байт-код, который выполняет JVM, и она сама отлично оптимизированы.

Рассмотрим схему работы JVM более подробно.

Структура class-файла

Напишем простейшее приложение и скомпилируем его. Компилятор заботливо создаст файл с расширением class и поместит туда всю информацию о нашем мини-приложении для JVM. Что мы увидим внутри? Файл поделён на десять секций, последовательность которых строго задана и определяет всю структуру class-файла.

Файл начинается со стартового (магического) числа: 0xCAFEBABE. Данное число присутствует в каждом классе и является обязательным флагом для JVM: с его помощью система понимает, что перед ней class-файл.



Следующие четыре байта class-файла содержат старший и младший номера версий Java. Они идентифицируют версию формата конкретного class-файла и позволяют JVM проверять, возможна ли его поддержка и загрузка. Каждая JVM имеет ограничение версии, которую она может загрузить — более поздние версии будут игнорироваться.

Как видно на примере файла выше, у нас major версия `0x34`, что соответствует Java SE 8. Для Java SE 11 будет значение `0x37`.

С девятого байта идёт пул констант, в котором содержатся все константы нашего класса. Так как в каждом классе их может быть различное количество, то перед массивом находится переменная, указывающая на его длину, то есть пул констант представляет из себя массив переменной длины. Каждая константа занимает один элемент в массиве. Во всём class-файле константы указываются целочисленным индексом, который обозначает их положение в массиве. Начальная константа имеет индекс 1, вторая константа — 2 и т. д.

Каждый элемент пула констант начинается с однобайтового тега, определяющего его тип. Это позволяет JVM понять, как правильно обработать идущую далее константу.

Всего зарезервировано 14 типов констант:

Тип константы	Значение тега
CONSTANT_Class	7
CONSTANT_Fieldref	9
CONSTANT_Methodref	10
CONSTANT_InterfaceMethodref	11
CONSTANT_String	8
CONSTANT_Integer	3
CONSTANT_Float	4
CONSTANT_Long	5
CONSTANT_Double	6
CONSTANT_NameAndType	12
CONSTANT_Utf8	1
CONSTANT_MethodHandle	15

CONSTANT_MethodType	16
CONSTANT_InvokeDynamic	18

Например, если тег указывает, что константа является строкой, JVM получает значение тега 1 и обрабатывает следующее за тегом число как длину массива байт, которые необходимо считать, чтобы получить нужную нам строку полностью.

Прочитав блок с константами, JVM переходит к следующим двум байтам — флагам доступа, которые определяют, описывает этот файл класс или интерфейс, общедоступный или абстрактный, является ли класс финальным.

Имена класса и его родительского класса хранятся в массиве констант, на которые указывают последующие 4 байта в файле.

Немного иначе обстоят дела с интерфейсами. Так как класс может наследоваться от множества интерфейсов одновременно, то хранить необходимо массив ссылок на пул констант. То есть за определением класса и его родительского класса идёт число, характеризующее размер массива интерфейсов, и сам массив. Все возможные значения кодов представлены ниже.

Имя флага	Код	Определение
ACC_PUBLIC	0x0001	Объявлен публичным
ACC_FINAL	0x0010	Объявлен финальным
ACC_SUPER	0x0020	Специальный флаг, введённый в версии Java 1.1 для совместимости при вызове методов родителя
ACC_INTERFACE	0x0200	Объявлен интерфейсом
ACC_ABSTRACT	0x0400	Объявлен абстрактным
ACC_SYNTHETIC	0x1000	Зарезервированное определение
ACC_ANNOTATION	0x2000	Объявлен аннотацией
ACC_ENUM	0x4000	Объявлен перечислением

Подобную структуру имеет и следующий блок — *Fields*.

Этот блок начинается с двухбайтового параметра количества полей в этом классе или интерфейсе. За ним идёт массив структур переменной длины. Каждая структура содержит информацию об одном поле: имя поля, тип, значение, если это, например,

финальная переменная. В списке отображаются только те поля, которые были объявлены классом или интерфейсом, определённым в файле. Поля от родительских классов и имплементированных интерфейсов здесь не присутствуют, они задаются в своих class-файлах.

Далее мы переходим к самому важному месту в любом классе — его методам, именно в них сосредоточена вся логика любой программы, весь исполняемый байт-код.

Ситуация абсолютно аналогична описанным выше полям. В массиве переменной длины содержатся структуры, в которые входит полное описание сигнатуры метода: модификаторы доступа, имя метода и его атрибуты, которые также представляют из себя структуру, так как их может быть множество и каждый из них может принадлежать разным типам.

В последнем блоке идёт дополнительная мета-информация, например имя файла, который был скомпилирован. Она может присутствовать, а может и нет. В случае каких-то проблем JVM просто игнорирует этот блок.

Мы рассмотрели структуру файлов и готовы перейти к следующей части — загрузке class-файла в JVM и последующему выполнению байт-кода из этого класса. В качестве закрепления полученных знаний по структуре class-файла можете воспользоваться

встроенным декомпилятором Java и посмотреть результат его выполнения с ключами `-c -verbose` (`javap -c -verbose TestJava.class`).

Загрузка классов

Теперь, разобравшись с общей структурой файла, посмотрим, как JVM его обрабатывает.

Чтобы попасть в JVM, класс должен быть загружен. Для этого существуют специальные классы-загрузчики:

1. *Bootstrap* — базовый загрузчик, загружает платформенные классы. Этот загрузчик является родителем всех остальных классов и частью платформы.
2. *Extension ClassLoader* — загрузчик расширений, потомок Bootstrap-загрузчика. Загружает классы расширений, которые по умолчанию находятся в каталоге `jre/lib/ext`.
3. *AppClassLoader* — системный загрузчик классов из *classpath*, который является непосредственным потомком *Extension ClassLoader*. Он загружает классы из каталогов и jar-файлов, указанных переменной среды `CLASSPATH`, системным свойством `java.class.path` или параметром командной строки `-classpath`.

4. Собственный загрузчик — у приложения могут быть свои собственные загрузчики.

Главный класс приложения всегда загружается системным загрузчиком, остальные же классы могут быть загружены различными пользовательскими загрузчиками. Стоит упомянуть, что имя загрузчика создаёт уникальное пространство имён, то есть в программе может существовать несколько классов с одним и тем же полным именем, если они обрабатывались разными загрузчиками.

Поэтому каждый загрузчик делегирует свои полномочия родителю, то есть перед поиском класса для загрузки он попытается узнать, не был ли загружен нужный класс раньше.

После загрузки класса начинается этап линковки, который делится на три части.

1. Верификация байт-кода. Это статический анализ кода, выполняется один раз для класса. Система проверяет, нет ли ошибок в байт-коде. Например, проверяет корректность инструкций, переполнение стека и совместимость типов переменных.

2. Выделение памяти под статические поля и их инициализация.

3. Разрешение символьных ссылок — JVM подставляет ссылки на другие классы, методы и поля. В большинстве случаев это происходит лениво, то есть при первом обращении к

классу.

Класс инициализируется, и JVM может начать выполнение байт-кода методов.

JVM получает один поток байтовых кодов для каждого метода в классе. Байт-код метода выполняется, когда этот метод вызывается в ходе работы программы. Поток байт-кода метода — это последовательность инструкций для виртуальной машины Java. Каждая инструкция состоит из однобайтового кода операции, за которым может следовать несколько операндов. Код операции указывает действие, которое нужно предпринять. Всего на данный момент в Java более 200 операций. Все коды операций занимают только 1 байт, так как они были разработаны компактными, поэтому их максимальное число не может превысить 256 штук.

В основе работы JVM находится стек — основные инструкции работают с ним.

Рассмотрим пример умножения двух чисел. Ниже представлен байт-код метода:

```
0: iconst_1 // взять число 1, положить в стек
1: istore_1 // сохранить это число в переменную 1 стека метода
2: iconst_5 // взять число 5, положить в стек
3: istore_2 // сохранить его в переменную 2 стека метода
```

```
4: iload_1 // положить в стек переменную 1
5: iload_2 // положить в стек переменную 2
6: imul // достать из стека два числа, умножить их и положить в стек
7: istore_3 // взять из стека число и сохранить его в переменную 3
```

На Java это будет выглядеть так:

```
int a = 1;
int b = 5;
int c = a * b
```

По листингу выше можно заметить, что коды операций сами по себе указывают тип и значение. Например, код операции `iconst_1` указывает JVM на целочисленное значение, равное единице. Такие байт-коды определены для самых часто используемых констант. Эти инструкции занимают 1 байт и введены специально для повышения эффективности выполнения байт-кода и уменьшения размера его потока. Подобные короткие константы также присутствуют и для других типов данных.

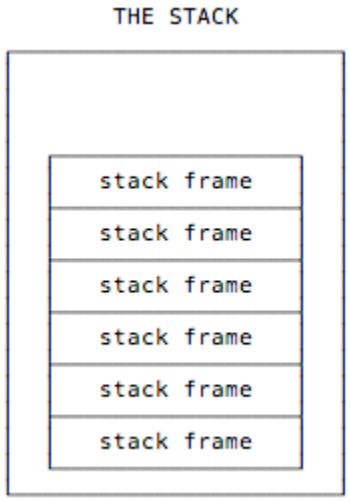
Всего JVM поддерживает семь примитивных типов данных: byte, short, int, long, float, double и char.

Если бы мы хотели положить в переменную a другое значение, например 11112, то нам пришлось бы использовать инструкцию `sipush`:

```
0: sipush 11112

3: istore_1
```

Данные операции выполняются в так называемом фрейме стека метода. У каждого метода есть некоторая своя часть в общем стеке. Таким образом в нашем главном потоке исполнения программы создаются множество подстеков на каждый вызов метода. Более наглядно это представлено на картинке ниже:



В каждом стек-фрейме хранится массив локальных переменных, который позволяет сохранять и доставать локальные переменные, как мы сделали в примере выше,

поместив значения 1 и 5 в переменные 1 и 2. Стоить отметить, что здесь компилятор также оптимизировал байт-код, используя однобайтовые инструкции. Если бы переменных в нашем методе было много, использовался бы код операции сохранения значения вместе с указанием позиции переменной в массиве.

Чтобы достучаться до пула констант класса и получить нужное значение, используются инструкции `lcd` и `lcd_w`. При этом `lcd` может ссылаться только на константы с индексами от 1 до 255, поскольку размер её операнда составляет всего 1 байт. `Lcd_w` имеет 2-байтовый индекс, поэтому может ссылаться на более широкий диапазон индексов.

Вызовы методов

Java предоставляет два основных вида методов: методы экземпляра и методы класса. Методы экземпляра используют динамическое (позднее) связывание, тогда как методы класса используют статическое (раннее) связывание.

Виртуальная машина Java вызывает метод класса, выбирая его на основании типа ссылки на объект, который всегда известен во время компиляции. С другой стороны, когда виртуальная машина вызывает метод экземпляра, она выбирает метод для

вызова на основе фактического класса объекта, который может быть известен только во время выполнения. Поэтому для вызова методов используются разные инструкции: `invokevirtual` и `invokestatic`. Данные функции ссылаются на запись в пуле констант в виде полного пути к необходимой функции. Виртуальная машина снимает нужное количество переменных со стека и передает в метод.

Возвращаемое методом значение кладётся на стек. Типы возвращаемых значений методов указаны ниже:

Операция	Описание
ireturn	Помещает на стек значение типа int
lreturn	Помещает на стек значение long,
freturn	Помещает на стек значение float
dreturn	Помещает на стек значение double
areturn	Помещает на стек значение object
return	Не изменяет стек

Циклы

Осталось рассмотреть последнюю часто используемую конструкцию языка — циклы.

Посмотрим, во что превратится код, представленный ниже:

```
int i = 1000;

while (i < 9999) {

   i += 10;

}
```

Байт-код:

```
0: sipush        1000

3: istore_1

4: iload_1

5: sipush        9999

8: if_icmpge     17

11: iinc          1, 10

14: goto          4

17: return
```

Каждый раз на стеке оказывается два числа, которые сравниваются, и если `i > 9999`, происходит выход из цикла. При этом для создания цикла используется конструкция на основе `goto`, которая запрещена в самом языке Java, хотя ключевое слово и зарезервировано.

Заключение

Изначально байт-код интерпретируется в большинстве JVM, но как только система замечает, что некоторый код используется очень часто, она подключает встроенный компилятор, который компилирует байт-коды в машинный код, тем самым значительно ускоряя работу приложения.

Таким образом, мы поверхностно рассмотрели жизненный цикл байткода в JVM: class-файлы, их загрузку и выполнение байт-кода и базовые инструкции.