Оглавление

- Java Virtual Machine Stacks
- Heap
- Область метода (Method Area)
- Run-Time Constant Pool
- Native Method Stacks

Frames

- Локальные переменные
- Стеки операндов (Operand Stacks)
- Динамическое связывание (Dynamic Linking)
- Нормальное завершение вызова метода
- Резкое завершение вызова метода

Основная идея языка (кроссплатформенность)

Изначально язык задумывался для программирования бытовых приборов (кофемашины, парковочные автоматы). Поэтому в основу легли две основные идеи — *платформонезависимость* (кроссплатформенность) и *встроенный сборщик мусора* (у бытовых приборов не было стандартизации в плане архитектуры и не было интерфейсов). На тот момент эти две идеи оказались хороши не только для бытовых приборов (апплеты, телефоны).

Сейчас эти особенности языка нам не очень интересны. Мы пользуемся не столько этими основами, заложенными в язык, сколько их результатами. За это время язык хорошо развился.

Под кроссплатформенностью (платформонезависимостью) предполагается, что одно и то же приложение, запущенное на разных платформах, будет работать одинаково.

Кроссплатформенность в Java опирается на существование JVM. Программисты пишут код, который компилируется в байт-код. Этот байт-код считывает JVM и превращает его в команды для компьютера. Т. о., мы не компилируем код для конкретной машины,

а компилируем в инструкции для JVM. Реализации JVM есть для всех платформ. Обычно их даже не надо устанавливать.

Источники:

Курс на степике «Разработка веб сервиса на Java (часть 1)»:

https://stepik.org/course/146/syllabus

Преимущества и недостатки Java

Преимущества Java

· кроссплатформенность;
· встроенный сборщик мусора;
· объектно-ориентированный язык;
· простой, понятный, похожий на C++ синтаксис;
· прекрасная документация;
· язык постоянно развивался, т. е. постоянные улучшения и доработки;
· многопоточность;
· стабильность и сообщество;
· высокая востребованность языка (лидирующие положения по индексу TIOBE с 2000 года) — огромное количество готовых библиотек и инструментов, а также источников информации;
· много вакансий и специалистов.

Недостатки Java

· плата за коммерческое использование с 2019 года;
· производительность (сборщик мусора);
· многословность;
· отсутствие нативного дизайна.

Байт-код, JVM, JRE и JDK

Код программиста при помощи компилятора компилируется в байт-код, который может исполнить JVM.

JVM (виртуальная машина Java) — основная часть JRE, исполняющая байт-код. Может выполнять программы, написанные на других языках.

JRE (среда времени выполнения Java) — минимальный набор для запуска приложений. JVM + Java SE (набор стандартных библиотек, Classloader).

JDK (комплект разработки на Java) — JRE + набор инструментов разработчика (*компилятор*, примеры, документация, утилиты).

Источники:

https://github.com/enhorse/java-interview/blob/master/core.md#%D0%A7%D0%B5%D0%BC-

%D1%80%D0%B0%D0%B7%D0%BB%D0%B8%D1%87%D0%B0%D1%8E%D1%82%D1%81%D1%8F-jre-jvm-%D0%B8-jdk

Интерпретатор и JIT-компилятор

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

Интерпретатор (interpreter) интерпретирует байт-код быстро, но медленно выполняется (при повторном вызове одного метода — новая интерпретация).

JIT-компилятор (Just-in-time Compiler) использует интерпретатор, но повторный код изменяет на свой, который использует при последующих вызовах методов.

· *Генератор промежуточного кода*. Производит промежуточный код.

· *Code Optimizer*. Отвечает за оптимизацию промежуточного кода, сгенерированного выше.

· *Генератор целевого кода*. Отвечает за генерацию машинного кода или родной код.

· *Профилировщик*. Специальный компонент, отвечающий за поиск горячих точек, т. е., вызывается ли метод несколько раз или нет.

Источники:

https://github.com/enhorse/java-interview/blob/master/jvm.md#Execution-Engine

Загрузчик классов

Загрузчик классов (Classloader) 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:

Загрузчик класса Bootstrap (bootstrap class loader):

· загружает основные библиотеки Java (<JAVA_HOME>/jre/lib);

· является частью ядра JVM и написан на нативном коде.

Загрузчик класса расширений (extensions class loader):

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

Системный загрузчик классов (system class loader):

· загружает код из `java.class.path`, который сопоставляется с переменной среды `CLASSPATH`;

· 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. 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).

Источники:

https://github.com/enhorse/java-interview/blob/master/jvm.md#Classloader

Сборщик мусора

Делает две вещи:

· находить мусор;

· освобождать память от мусора.

Что является мусором — определяется подходом к его обнаружению.

Подходы к обнаружению мусора:

· подсчет ссылок;

· трассировка.

Подсчет ссылок. Суть подхода: каждый объект имеет счетчик, который хранит количество ссылок на данный объект. Если значение счетчика равно нулю, то объект считается мусором. Минусы подхода: сложность обеспечения точности счетчика и сложность выявления циклических зависимостей.

Главная идея подхода *трассировки* состоит в утверждении, что живыми могут считаться только те объекты, до которых мы можем добраться из корневых точек (GC Root) и те

объекты, которые доступны с живого объекта. Все остальное — мусор.

Существует 4 типа корневых точки:
· локальные переменные и параметры методов;
· потоки;
· статические переменные;
· ссылки из JNI.

Самое простое java приложение будет иметь корневые точки:
· локальные переменные внутри `main()` метода и параметры main() метода;
· поток который выполняет main();
· статические переменные класса, внутри которого находится main() метод.

Т. о., если мы представим все объекты и ссылки между ними как дерево, то нам нужно будет пройти с корневых узлов (точек) по всем ребрам. При этом узлы, до которых мы сможем добраться — не мусор, все остальные — мусор. При таком подходе циклические зависимости легко выявляются. HotSpot VM использует именно такой подход.

Основные методы для очистки памяти от мусора:
· Copying collectors;
· Mark-and-sweep.

При *copying collectors* подходе память делится на две части: «from-space»

и «to-space». Принцип работы:
· объекты создаются в «from-space»;
· когда «from-space» заполняется, приложение приостанавливается;
· запускается сборщик мусора, находятся живые объекты в «from-space» и копируются в «to-space»;
· когда все объекты скопированы «from-space» полностью очищается;
· «to-space» и «from-space» меняются местами.

Плюсы:
· объекты плотно забивают память.

Минусы:
· требуется остановка приложения (для полного прохождения цикла сборки мусора);
· в худшем случае (все объекты живые) обе части памяти будут одинакового размера.

Алгоритм работы *mark-and-sweep*:
· объекты создаются в памяти;
· в момент, когда нужно запустить сборщик мусора приложение приостанавливается;
· сборщик проходится по дереву объектов, помечая живые объекты;
· сборщик проходится по всей памяти, находя все не отмеченные куски памяти и сохраняя их в «free list»;
· когда новые объекты начинают создаваться,

они создаются в памяти доступной во «free list».

Минусы:
· приложение не работает пока происходит сборка мусора;
· время остановки напрямую зависит от размеров памяти и количества объектов;
· если не использовать «compacting» память будет использоваться не эффективно.

Сборщики мусора HotSpot VM используют комбинированный подход Generational Garbage Collection, который позволяет использовать разные алгоритмы для разных этапов сборки мусора. Этот подход опирается на том, что:
· большинство создаваемых объектов быстро становятся мусором;
· существует мало связей между объектами, которые были созданы в прошлом и только что созданными объектами.

Принцип работы

Механизм сборки мусора — это процесс освобождения места в куче, для возможности добавления новых объектов.

Объекты создаются посредством оператора `new`, тем самым присваивая объекту ссылку. Для окончания работы с объектом достаточно просто перестать на него ссылаться, например, присвоив переменной ссылку

на другой объект или значение null; прекратить выполнение метода, чтобы его локальные переменные завершили свое существование естественным образом. Объекты, ссылки на которые отсутствуют, принято называть мусором, который будет удален.

Виртуальная машина Java, применяя механизм сборки мусора, гарантирует, что любой объект, обладающий ссылками, остается в памяти — все объекты, которые недостижимы из исполняемого кода, ввиду отсутствия ссылок на них, удаляются с высвобождением отведенной для них памяти. Точнее говоря, объект не попадает в сферу действия процесса сборки мусора, если он достижим посредством цепочки ссылок, начиная с корневой (GC Root) ссылки, т. е. ссылки, непосредственно существующей в выполняемом коде.

Память освобождается сборщиком мусора по его собственному «усмотрению». Программа может успешно завершить работу, не исчерпав ресурсов свободной памяти или даже не приблизившись к этой черте и поэтому ей так и не потребуются «услуги» сборщика мусора.

Мусор собирается системой автоматически, без вмешательства пользователя

или программиста, но это не значит, что этот процесс не требует внимания вовсе. Необходимость создания и удаления большого количества объектов существенным образом сказывается на производительности приложений и если быстродействие программы является важным фактором, следует тщательно обдумывать решения, связанные с созданием объектов, — это, в свою очередь, уменьшит и объем мусора, подлежащего утилизации.

Разновидности

Java HotSpot VM предоставляет разработчикам на выбор четыре различных сборщика мусора:

Последовательный — самый простой вариант для приложений с небольшим объемом данных и не требовательных к задержкам. На данный момент используется сравнительно редко, но на слабых компьютерах может быть выбран виртуальной машиной в качестве сборщика по умолчанию. Использование Serial GC включается опцией -XX:+UseSerialGC.

Параллельный — наследует подходы к сборке от последовательного сборщика, но добавляет параллелизм в некоторые операции, а также возможности по автоматической подстройке под требуемые параметры производительности. Параллельный

сборщик включается опцией -XX:+UseParallelGC.

Concurrent Mark Sweep (CMS) — нацелен на снижение максимальных задержек путем выполнения части работ по сборке мусора параллельно с основными потоками приложения. Подходит для работы с относительно большими объемами данных в памяти. Использование CMS GC включается опцией -XX:+UseConcMarkSweepGC.

Garbage-First (G1) — создан для замены CMS, особенно в серверных приложениях, работающих на многопроцессорных серверах и оперирующих большими объемами данных. G1 включается опцией Java -XX:+UseG1GC.

Последовательный сборщик мусора

Последовательный сборщик мусора был одним из первых сборщиков мусора в HotSpot VM. Во время работы этого сборщика приложения приостанавливается и продолжает работать только после прекращения сборки мусора.

Память приложения делится на три пространства:

· *Young generation.* Объекты создаются именно в этом участке памяти.

· *Old generation.* В этот участок памяти перемещаются объекты, которые переживают «minor garbage collection».

· *Permanent generation.* Тут хранятся метаданные об объектах, Class data sharing (CDS), пул строк. Permanent область делится на две: только для чтения и для чтения-записи. Очевидно, что в этом случае область только для чтения не чистится сборщиком мусора никогда.

Область памяти *Young generation* состоит из трех областей: Eden и двух меньших по размеру Survivor spaces — To space и From space. Большинство объектов создаются в области Eden, за исключением очень больших объектов, которые не могут быть размещены в ней и поэтому сразу размещаются в Old generation. В Survivor spaces перемещаются объекты, которые пережили по крайней мере одну сборку мусора, но еще не достигли порога «старости» (tenuring threshold), чтобы быть перемещенными в Old generation.

Когда Young generation заполняется, то в этой области запускается процесс легкой сборки (minor collection), в отличие от процесса сборки, проводимого над всей кучей (full collection). Он происходит следующим образом: в начале работы одно

из Survivor spaces — To space, является пустым, а другое — From space, содержит объекты, пережившие предыдущие сборки. Сборщик мусора ищет живые объекты в Eden и копирует их в To space, а затем копирует туда же и живые «молодые» (т. е. не пережившие еще заданное число сборок мусора) объекты из From space. Старые объекты из From space перемещаются в Old generation. После легкой сборки From space и To space меняются ролями, область Eden становится пустой, а число объектов в Old generation увеличивается.

Если в процессе копирования живых объектов To space переполняется, то оставшиеся живые объекты из Eden и From space, которым не хватило места в To space, будут перемещены в Old generation, независимо от того, сколько сборок мусора они пережили.

Поскольку при использовании этого алгоритма сборщик мусора просто копирует все живые объекты из одной области памяти в другую, то такой сборщик мусора называется copying (копирующий). Очевидно, что для работы копирующего сборщика мусора у приложения всегда должна быть свободная область памяти, в которую будут копироваться живые объекты, и такой

алгоритм может применяться для областей памяти сравнительно небольших по отношению к общему размеру памяти приложения. Young generation как раз удовлетворяет этому условию (по умолчанию на машинах клиентского типа эта область занимает около 10% кучи (значение может варьироваться в зависимости от платформы)).

Однако, для сборки мусора в *Old generation*, занимающем большую часть всей памяти, используется другой алгоритм.

В Old generation сборка мусора происходит с использованием алгоритма mark-sweep-compact, который состоит из трех фаз. В фазе Mark (пометка) сборщик мусора помечает все живые объекты, затем, в фазе Sweep (очистка) все не помеченные объекты удаляются, а в фазе Compact (уплотнение) все живые объекты перемещаются в начало Old generation, в результате чего свободная память после очистки представляет собой непрерывную область. Фаза уплотнения выполняется для того, чтобы избежать фрагментации и упростить процесс выделения памяти в Old generation.

Когда свободная память представляет собой непрерывную область, то для выделения памяти под создаваемый объект можно использовать очень быстрый (около десятка

машинных инструкций) алгоритм bump-the-pointer: адрес начала свободной памяти хранится в специальном указателе, и когда поступает запрос на создание нового объекта, код проверяет, что для нового объекта достаточно места, и, если это так, то просто увеличивает указатель на размер объекта.

Последовательный сборщик мусора отлично подходит для большинства приложений, использующих до 200 мегабайт кучи, работающих на машинах клиентского типа и не предъявляющих жестких требований к величине пауз, затрачиваемых на сборку мусора. В то же время модель «stop-the-world» может вызвать длительные паузы в работе приложения при использовании больших объемов памяти. Кроме того, последовательный алгоритм работы не позволяет оптимально использовать вычислительные ресурсы компьютера, и последовательный сборщик мусора может стать узким местом при работе приложения на многопроцессорных машинах.

Источники:

https://github.com/enhorse/java-interview/blob/master/core.md#%D0%94%D0%BB%D1%8F-%D1%87%D0%B5%D0%B3%D0%BE-%D0%BD%D1%83%D0%B6%D0%B5%D0%BD-

%D1%81%D0%B1%D0%BE%D1%80%D1%89%D0%B8%D0%BA-%D0%BC%D1%83%D1%81%D0%BE%D1%80%D0%B0

Heap и Stack

Чем похожи:
· используют оперативную память;
· изменяют размеры в зависимости от требований вычисления.

Чем отличаются:
· heap создается при запуске JVM, stack — с созданием потока;
· heap является глобальной памятью, stack существует внутри каждого потока;
· heap хранит экземпляры и массивы классов, stack — frames;
· разные исключения при переполнении: `java.lang.OutOfMemoryError: Java Heap Space` и `java.lang.StackOverflowError`;
· используются разные опции JVM для определения размера памяти (Xms/Xmx и Xss);
· в heap работает сборщик мусора;
· стековая память меньше и быстрее, работает по схеме LIFO (последний зашел, первый вышел).

Утверждение, что примитивные типы данных всегда хранятся в стеке, а экземпляры ссылочных типов данных в куче не совсем верно, т. к. примитивное поле будет хранится в куче.

Источники:

https://github.com/enhorse/java-interview/blob/master/core.md#%D0%A7%D1%82%D0%BE-%D1%82%D0%B0%D0%BA%D0%BE%D0%B5-heap-%D0%B8-stack-%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C-%D0%B2-java-%D0%9A%D0%B0%D0%BA%D0%B0%D1%8F-%D1%80%D0%B0%D0%B7%D0%BD%D0%B8%D1%86%D0%B0-%D0%BC%D0%B5%D0%B6%D0%B4%D1%83-%D0%BD%D0%B8%D0%BC%D0%B8

https://github.com/enhorse/java-interview/blob/master/concurrency.md#%D0%92-%D1%87%D0%B5%D0%BC-%D0%B7%D0%B0%D0%BA%D0%BB%D1%8E%D1%87%D0%B0%D1%8E%D1%82%D1%81%D1%8F-%D1%80%D0%B0%D0%B7%D0%BB%D0%B8%D1%87%D0%B8%D1%8F-%D0%BC%D0%B5%D0%B6%D0%B4%D1%83-c%D1%82%D0%B5%D0%BA%D0%BE%D0%BC-stack-%D0%B8-%D0%BA%D1%83%D1%87%D0%B5%D0%B9-heap-%D1%81-%D1%82%D0%BE%D1%87%D0%BA%D0%B8-%D0%B7%D1%80%D0%B5%D0%BD%D0%B8%D1%8F-%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE%D0%BF%D0%BE%D1%82%D0%BE%D1%87%D0%BD%D0%BE%D1%81%D1%82%D0%B8

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 ее выключения. Другие создаются для каждого потока и уничтожаются, когда поток уничтожается.

Регистр PC (Programm Counter Register)

JVM может поддерживать много потоков исполнения одновременно. Каждый поток JVM имеет свой собственный *регистр PC*. В любой момент каждый поток JVM выполняет код одного метода, а именно текущий метод для этого потока. Если этот метод не является native, *регистр PC* содержит адрес инструкции JVM, 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 — 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.

Область метода (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

Существует для каждого класса или интерфейса в рантайме и представлено 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 отделен от JVM's method area. JVM создает 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).

Источники:

https://github.com/enhorse/java-interview/blob/master/jvm.md#%D0%9E%D0%B1%D0%BB%D0%B0%D1%81%D1%82%D0%B8-%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85-%D0%B2%D1%80%D0%B5%D0%BC%D0%B5%D0%BD%D0%B8-%D0%B2%D1%8B%D0%BF%D0%BE%D0%BB%D0%BD%D0%B5%D0%BD%D0%B8%D1%8F

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. Т. о., размер структуры данных, frame 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, т. к. 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. 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 определяется во время компиляции и предоставляется вместе с кодом для метода, связанного с frame.

Стек операнда пуст при создании frame, 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` значения. От стека операндов требуется, чтобы два `int` 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 вовремя 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 athrow 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.

Источники:

https://github.com/enhorse/java-interview/blob/master/jvm.md#Frames