МОДЕЛЬ ПАМЯТИ JAVA

[1]Бакунова Оксана Михайловна,
[2]Хмелевская Алина Леонидовна,
[3]Ефимук Никита Андреевич,
[3]Дмитриев Алексей Викторович,
[3]Бацкель Дмитрий Викторович,
[3]Кобяк Даниил Ромуальдович

Беларусь, ИИТ БГУИР,
[1]старший преподаватель, исследователь технических наук, магистр технических наук;
[2]ассистент;
[3]студент

DOI: https://doi.org/10.31435/rsglobal_wos/12062018/5737

ARTICLE INFO

Received: 10 May 2018
Accepted: 02 June 2018
Published: 12 June 2018

KEYWORDS

JVM,
java memory model,
HotSpot virtual machine,
garbage collector

ABSTRACT

A Java virtual machine (JVM) is an implementation of JVM specification that formally described what should be done in the realization of JVM. Installed JVM allows running programs, which have already compiled to the Java bytecode. JVM also can use for compiling other programming languages. For example, if a code was written in Scala or Python programming language, it can be compiled in Java bytecode and run using JVM. When Java process starts, the operating system allocates some space of memory for Java process. Afterward, this space will be organized by JVM for such areas as heap, stack, method area and program counter register. This memory distribution describes Java memory model. In this article, we will discuss how threads in Java program interact with memory and how exactly it allocates the memory for itself.

Citation: Бакунова О. М., Хмелевская А. Л., Ефимук Н. А., Дмитриев А. В., Бацкель Д. В., Кобяк Д. Р. (2018) Модель памяти JAVA. *Web of Scholar. 6(24),* Vol.1. doi: 10.31435/rsglobal_wos/12062018/5737

Copyright: © 2018 **Бакунова О. М., Хмелевская А. Л., Ефимук Н. А., Дмитриев А. В., Бацкель Д. В., Кобяк Д. Р.** This is an open-access article distributed under the terms of the **Creative Commons Attribution License (CC BY)**. The use, distribution or reproduction in other forums is permitted, provided the original author(s) or licensor are credited and that the original publication in this journal is cited, in accordance with accepted academic practice. No use, distribution or reproduction is permitted which does not comply with these terms.

Введение. При компиляции исходного текста программы, написанной на Java создается файл с расширением «.class» с таким же именем, который содержит байт-код, и, в дальнейшем, отдается на исполнение JVM (java virtual machine). JVM – виртуальная машина, основная часть которой исполняет байт-код предварительно созданный из исходного текста Java-программы компилятором. Байт-код можно использовать на многих платформах, так как существуют различные реализации JVM (для Windows, Linux, MacOS, Android и так далее). Это позволяет реализовывать одно из главных преимуществ языка Java - «скомпилировано однажды, запускается везде». Существует большое количество имплементаций JVM, таких как: HotSpot, OpenJ9, JRockit, Dalvik. Также есть спецификация, в которой описано как производители должны реализовывать свои JVM, в этой статье мы будем отталкиваться от данной спецификации и от спецификации, которая описывает The Java HotSpot Virtual Machine, а также от Java версии 9.

Результаты и обсуждение. Виртуальная машина — программа, написанная на смеси языков C/C++. Она, являясь механизмом выполнения байт-кода Java предоставляет средства выполнения Java, такие как синхронизация потоков и объектов в различных операционных

системах и архитектурах. Она включает в себя динамические компиляторы, которые адаптивно компилируют байт-код Java в оптимизированные машинные инструкции и управляют областью памяти heap с помощью сборщика мусора (garbage collector).

Когда запускается процесс Java операционная система выделяет некоторую область памяти, распределением которой занимается JVM. С точки зрения операционной системы, это такой же процесс, как и Word, Excel или Skype. Если учесть, что в процессе выполняется один поток, то графически эту область можно представить так, как показано на рисунке 1.

Дадим описание областей памяти:

– Stack – Каждый поток имеет свой стек, который создается в тоже время, когда и создается поток. Стек содержит фреймы, которые создаются при каждом вызове метода и хранят локальные переменные и промежуточные результаты, возвращают значения для методов и выбрасывают исключения, если это необходимо. Фрейм разрушается, когда вызов метода завершается, неважно является это завершение успешным или с исключением.

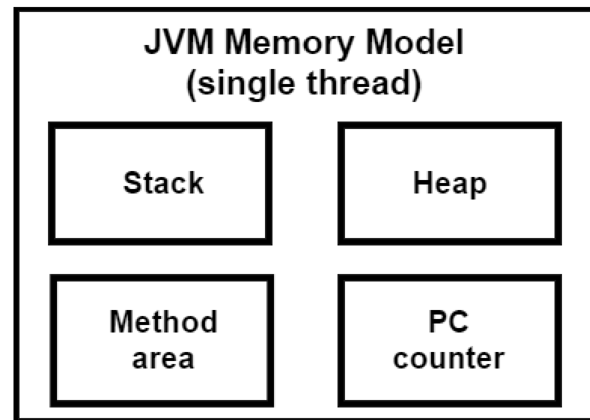

Рис. 1. Графическое представления разделения памяти внутри однопоточного процесса Java

Ниже приведен пример кода:

```
public class Memory {
 public static void main(String[] args) {
 main(args);
 }
}
```

Его результатом будет исключение StackOverflowError, потому что этот код бесконечно вызывает сам себя, соответственно память в стеке заканчивается. Существуют возможности увеличить размер стека, для этого необходимо при запуске добавить аргумент для виртуальной машины –Xss1024k. Это установит размер стека равным 1 мегабайту.

– Heap – создается в момент запуска виртуальной машины, это область памяти в которой хранятся все созданные в процессе работы программы ссылочные типы данных. Он существует только один и разделяется между всеми потоками, существующими в программе. Для очистки от более неиспользуемых объектов (те объекты, на которые никто больше не ссылается) используется сборщик мусора (garbage collector). Heap состоит из частей, которые представлены на рисунке 2:

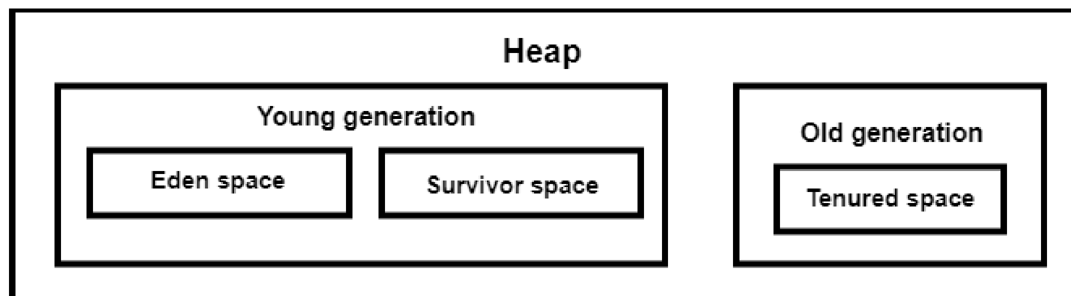

Рис. 2. Разделение области памяти heap

Young generation – область где объект находится короткий период времени и делится на два пространства:

1. Eden space – в этом пространстве выделяется память для объекта при создании используя ключевое слово new.
2. Survivor space – пространство используется для хранения объектов которые выжили после сборки мусора в пространстве Eden.

Old generation – область для хранения объектов которые выжили после сборки мусора в Young generation. Включает:

1. Tenured space – содержит объекты которые выжили после многократной сборки мусора в Survivor space.

Область памяти heap разделена на такие части в связи с работой сборщика мусора (garbage collector). Он опирается на тот факт, что в большинстве программ подавляющее большинство объектов (часто более 95 %) очень недолговечны (например, они используются как временные структуры данных), а остальные объекты живут довольно продолжительное количество времени. В современной JVM HotSpot поддерживается параллельный сборщик мусора в нем трассировка и копирование живых объектов выполняется несколькими потоками, работающими параллельно. При перемещении объектов он пытается сохранить связанные объекты вместе, что приводит к тому что связанные объекты располагаются в памяти рядом. Объекты, которые выживают после сборки мусора в одном пространстве, помещаются в пространство на уровень ниже. Таким образом выжившие объекты перемещаются из young generation в old generation.

Ниже приведен пример кода:

```
public class Memory {
 public static void main(String[] args) {
 Object ref = new long[Integer.MAX_VALUE / 2];
 }
}
```

Результатом работы данного кода на среднестатистической машине будет исключение OutOfMemoryError, потому что данный код пытается создать массив данных long (8 байт) размером ~ 1.070.000.000. Это требует примерно 8.560.000.000 байт памяти, что равно 8560 мегабайтам. Существуют следующие команды для конфигурации размера кучи:

1. –Xms – устанавливает начальный размер кучи;
2. –Xmx – устанавливает максимальный размер кучи;
3. –Xmn – устанавливает размер young generation.

– Program counter register – У каждого потока есть свой собственный PC register. В любой момент поток может выполнять код только одного метода. Если этот метод не native (т.е. метод, который написан на другом языке программирования), то регистр содержит адрес инструкции, которая выполняется в данный момент. Если метод все же native, то значение регистра не определено.

– Method area – Существует всего одна область методов для всех потоков, в ней хранятся структуры каждого класса, данные полей и методов и код для методов и конструкторов, включая специальные методы инициализации класса, интерфейса и экземпляра, а также статические переменные. Эта область известна как Metaspace или Permanent generation. Начиная с Java 8 эта область памяти при необходимости динамически расширяется. Также в этой области находится Run-Time Constant Pool.

Учитывая информацию, которая была дана выше и то, что большинство приложений, написанных на Java работают в нескольких потоках представим модель памяти многопоточного процесса, она находится на рисунке 3.

Рассмотрим на примере кода как взаимодействуют стек и область памяти heap:

```
public class Memory {
 public static void main(String[] args) {
 int i = 21;
 int[] array = {10, 20, 30};
 short s = 25;
 Object object = new Object();
 }
}
```

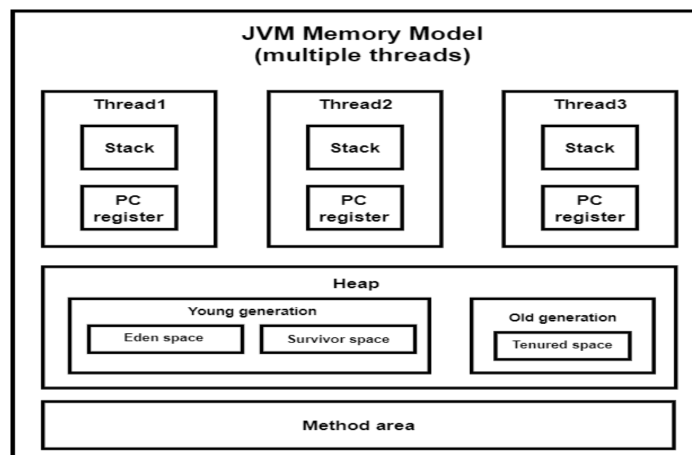

Рис. 3. Графическое представление разделения памяти внутри процесса Java с несколькими потоками

Как упоминалось выше, в стеке хранятся примитивные типы данных, которых в Java восемь штук: byte, short, char, int, long, float, double, boolean. Исходя из этого, фрейм этого метода будет содержать два значения типа int и

short и три ссылки (включая аргумент метода main) на объекты, которые хранятся в куче (массив примитивного типа в Java это тоже объект). Графическое представление находится на рисунке 4.

В Java объекты передаются по ссылке, а примитивные типы данных по значению. Рассмотрим следующий код:

```
public class Memory {
 public static void main(String[] args) {
 int number = 10;
 int[] array = {10, 20, 30};
 change(number, array);
 System.out.println("Number: " + number + "; Array: " + Arrays.toString(array));
 }
 private static void change(int number, int[] arg) {
 number = 100;
 arg[0] = 100;
 }
}
```

Содержимое фреймов представлено на рисунке 5.

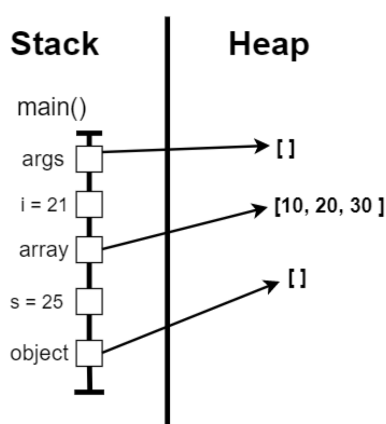

Рис. 4. Фрейм метода main()

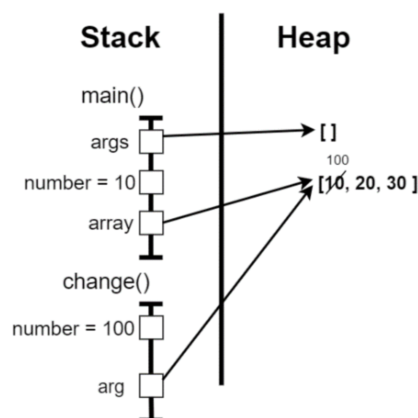

Рис. 5. Содержимое фреймов методов main() и change()

Результатом работы примера будет следующая строка: Number: 10; Array: [100, 20, 30]. Как видно в результате, переменная типа int не изменилась потому что в фрейм метода change была передана копия переменной. В случае с массивом, в метод мы передали ссылку на объект в куче, где расположен этот массив и успешно изменили значение нулевого элемента.

Выводы. Такие технологии как сборщик мусора и динамическое распределение памяти существенно упрощает процесс программирования, но это может создать иллюзию того, что программисту вопросами выделения и освобождения памяти вообще не надо уделять внимания. Во многих случаях, системы с динамическим выделением памяти значительно ограничивают возможности программиста, сложно реализуются алгоритмы, которые требуют прямого доступа к оперативной памяти. В некоторых случаях такие системы проигрывают и по производительности, и по объему используемой памяти. Несмотря на это Java успешно заняла свое место на рынке разработки ПО.

ЛИТЕРАТУРА

1. The Java® Virtual Machine Specification. Java SE 9 Edition. [Электронный ресурс]. URL: https://docs.oracle.com/javase/specs/jvms/se9/jvms9.pdf (дата обращения: 11.03.2018)

2. Java SE HotSpot at a Glance. [Электронный ресурс]. URL: http://www.oracle.com/technetwork/articles/javase/index-jsp-136373.html (дата обращения: 11.03.2018)