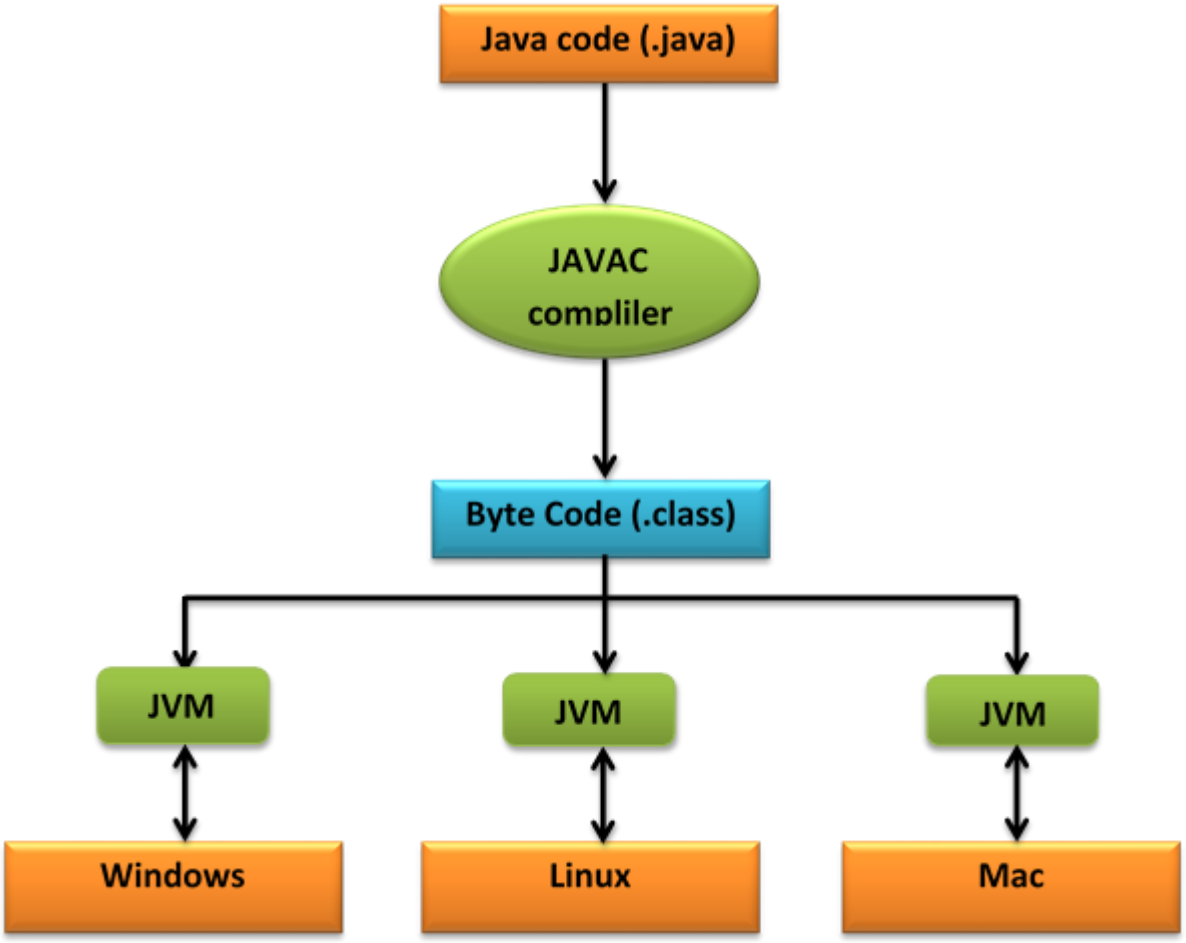
JAVA VIRTUAL MACHINE DOCUMENTATION

Lê Phương Thanh – Nguyễn Trọng Thuận

Nội dung

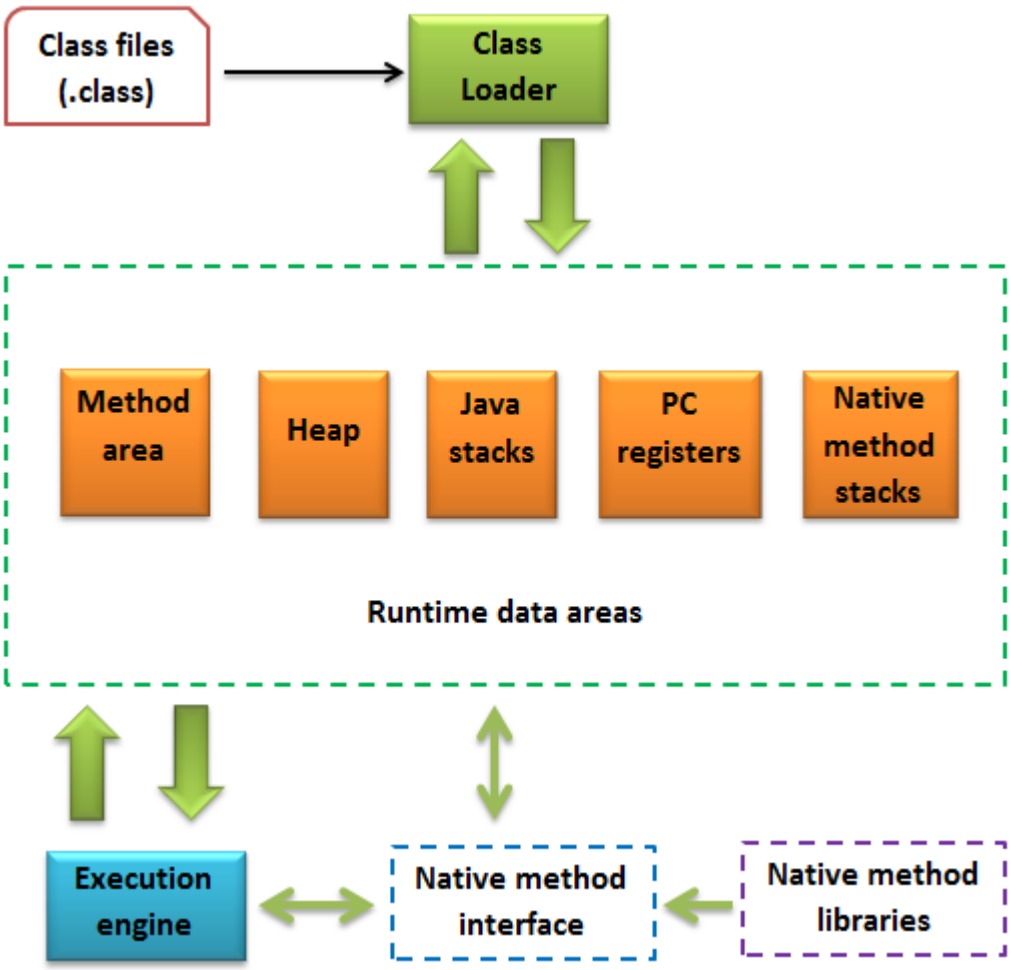
I. Java Virtual Machine là gì ?



Hình 1 Tổng quan về JVM

Java Virtual Machine (JVM) là một máy ảo, cung cấp môi trường runtime để **Java Bytecode** có thể thực thi được.

II. Kiến trúc JVM



Hình 2 Mô hình cấu trúc của JVM

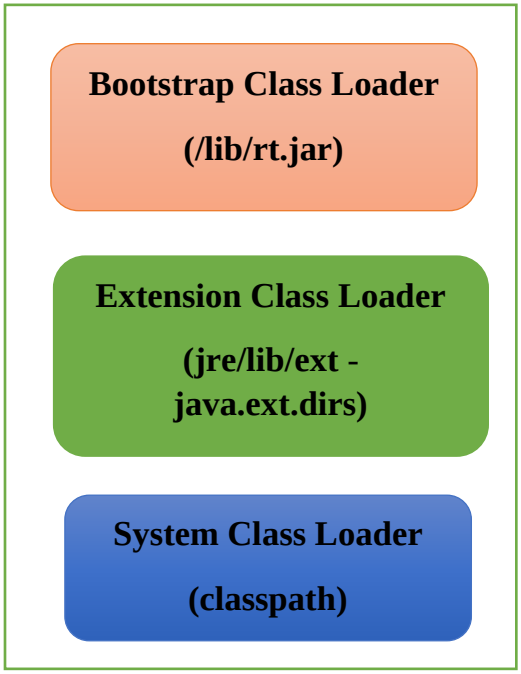
JVM gồm 3 thành phần chính:

- **Class loader**.
- **Runtime Data Area**.
- **Execution Engine**.

Ngoài các thành phần chính như trên, **JVM** còn có thêm hai thành phần khác là **Native Method Interface** (Java Native Interface – JNI), **Native Method Library**.

1. Class loader

Class loader là một Subsystem trong **JVM** có nhiệm nạp các **class file (.class)** vào JVM.



Hình 3 Mô hình cấu trúc của Class Loader Sybsystem

Class Loader gồm 3 Class Loader:

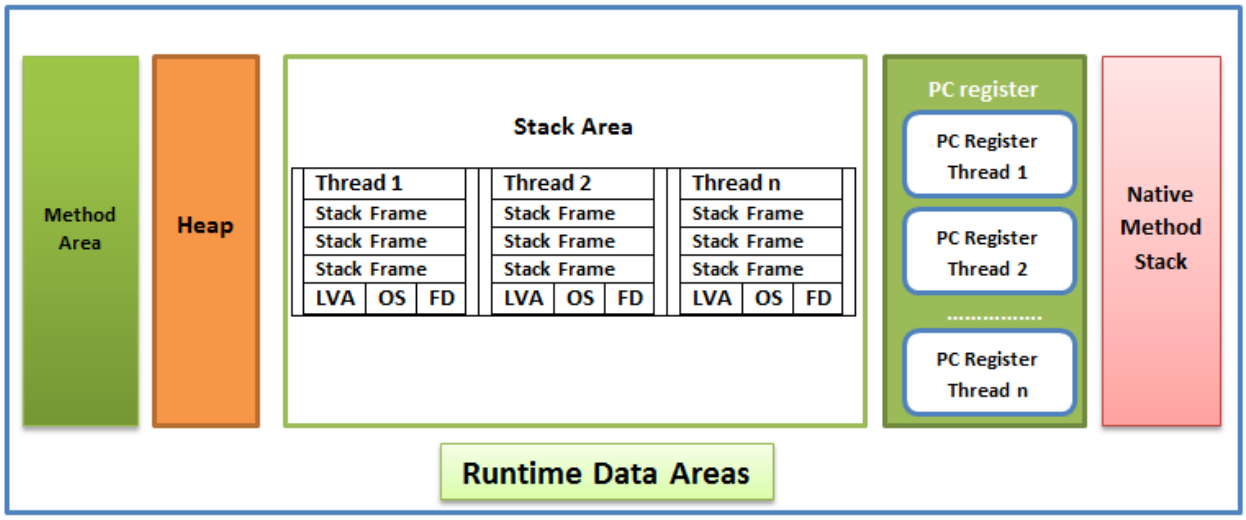
- **Bootstrap Class Loader** (Bộ nạp lớp khởi động): được tạo ra khi JVM bắt đầu hoạt động, có nhiệm vụ nạp các class của Java cơ bản từ file **/lib/rt.jar** như *java.lang, java.ulti,…*
- **Extension Class Loader** (Bộ nạp lớp mở rộng): có nhiệm vụ nạp tất cả các file class trong **JRE/lib/ext** hoặc **java.ext.dirs**.
- **System Class Loader** (Bộ nạp lớp hệ thống): có nhiệm vụ nạp các lớp java có sẵn trong file **classpath** được nạp bởi **Bootstrap Class Loader**.

Các class loader có mối quan hệ phân cấp với nhau, class loader có thể nạp các file class từ một cấp độ nào đó trên hệ thống phân cấp. Mức thứ nhất là **Bootstrap Class Loader**, mức thứ hai là **Extension Class Loader** và mức thứ ba là **System Class**

Loader. Nếu như file class không tìm thấy để nạp thì hệ thống sẽ đưa ra lỗi ***ClassNotFoundException***.

2. Runtime Data Areas

Runtime Data Areas là vùng nhớ do hệ thống cung cấp cho JVM để lưu trữ dữ liệu khi JVM hoạt động. Vùng nhớ này được tạo ra khi JVM khởi động và bị hủy khi JVM kết thúc công việc.



Hình 4 Mô hình cấu trúc của Runtime Data Areas

Runtime Data Areas gồm các thành phần chính như sau:

a. Method Area

Là nơi lưu trữ dữ liệu của class, được tạo ra khi JVM hoạt động. Mỗi JVM chỉ có một **Method Area** để dùng chung cho tất cả các tiến trình.

Method Area chứa các thông tin của **class** và **interface** như:

- **Runtime constant pool**: Chứa địa chỉ tham chiếu của các thuộc tính và phương thức của class hoặc interface. Khi một thuộc tính hay phương thức được gọi thì JVM sẽ tìm kiếm địa chỉ của thuộc tính hoặc phương thức đó Runtime constant pool cần nhiều bộ nhớ hơn khả năng cung cấp của hệ thống thì sẽ đưa ra lỗi **OutOfMemoryError**.

- Thông tin của **field**: tên, kiểu dữ liệu, access control modifier (public, default, protected, private).
- Thông tin của phương thức (**method**): tên, kiểu trả về, số lượng và kiểu dữ liệu của tham số, access control modifier (public, default, protected, private).
- **Biến static**: tên, kiểu dữ liệu, access control modifier (public, default, protected, private).
- Bytecode của phương thức.

b. Heap

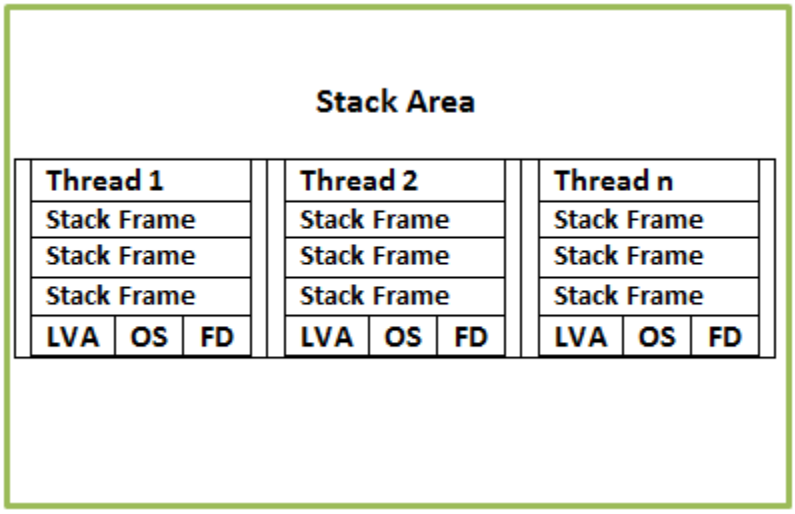
Heap được tạo ra khi JVM hoạt động, dùng để lưu thông tin của các đối tượng được tạo ra. Tất cả các tiến trình trong JVM đều sử dụng chung bộ nhớ Heap.

Đối tượng được tạo trong bộ nhớ heap có phạm vi truy cập toàn cục (có thể truy cập đối tượng đó bất kì đâu trong ứng dụng).

Bộ nhớ mặc định của Heap trong Java là 64MB, vùng nhớ của Heap sẽ tăng dần kích thước mỗi khi đối tượng được cấp vùng nhớ vật lý. Vì thế, JVM cung cấp giải pháp thu dọn rác (**Garbage Collection**-GC) để giải phóng vùng nhớ bằng cách hủy những đối tượng không dùng đến do đó thời gian sống của đối tượng phụ thuộc vào Garbage Collection của java. Với cách thức này thì vùng nhớ trong Heap sẽ được tái sử dụng cho các đối tượng khác.

Đối với đối tượng tham là Thread thì GC chỉ được xem xét hủy nó khi trạng thái của thread object *isAlive()=false*. Ngoài ra, GC không đóng các file, kết nối mạng, recordstore, media player,..

c. Stack



Hình 5 Cấu trúc của Stack

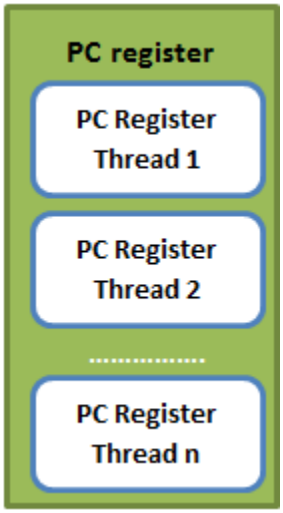
Khi mỗi tiến trình hoạt động thì JVM sẽ tạo ra một JVM stack. JVM stack là nơi lưu trữ các **frame** (frame là nơi lưu trữ dữ liệu và kết quả trả về của phương thức. Khi một phương thức được gọi để thực thi thì một frame sẽ được tạo mới, sau khi phương thức thực hiện xong thì frame sẽ bị hủy).

Mỗi **frame** sẽ bao gồm:

- **Local variable Array** (mảng các biến cục bộ): là một mảng dùng để lưu trữ các biến cục bộ, độ dài của mảng phụ thuộc vào thời gian biên dịch của chương trình.
- **Operand Stack**: là ngăn xếp chứa các toán hạng, hoạt động dựa trên cơ chế Last-In-First-Out (**LIFO**) của stack. Khi mới khởi tạo thì Operand Stack rỗng, sau đó JVM sẽ đưa các biến trong mảng biến cục bộ vào Operand Stacks. Tiếp theo, JVM sẽ thực hiện các phương thức tính toán trên các biến này, cuối cùng kết quả sẽ được đẩy vào Operand Stacks theo cơ chế LIFO.
- **Frame data**: tham chiếu đến **run-time constant pool** của class có phương thức đang được thực thi.

Bộ nhớ của JVM stack có thể rời rạc, không liên tục, do đó kích thước của JVM stack có thể cố định hoặc được mở rộng tùy vào nhu cầu sử dụng.

d. PC registers (Program counter register)



Hình 6 PC register

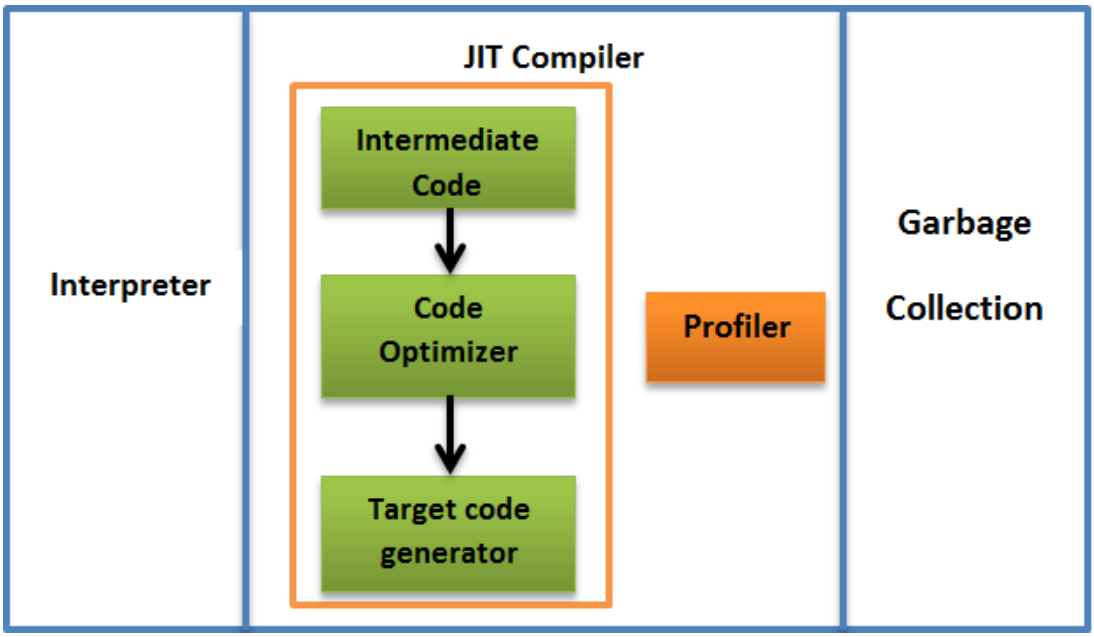
JVM có thể hỗ trợ nhiều tiến trình hoạt động cùng một lúc, mỗi tiến trình của JVM có một PC register, PC register được tạo ra khi có một tiến trình mới khởi tạo. Mỗi tiến trình sẽ thực thi code của một phương thức, nếu phương thức viết bằng ngôn ngữ Java (**không phải native method**) thì PC register chứa địa chỉ của JVM hướng dẫn việc thực thi phương thức đó.

e. Native Method Stack

Native method stack của JVM hỗ trợ việc thực thi các phương thức viết bằng ngôn ngữ Java và kết hợp với ngôn ngữ khác chẳng hạn như C (gọi là native method) thông qua Java Native Interface (ở mục 4).

3. Execution Engine

Execution Engine có nhiệm vụ thực thi bytecode chứa trong các file class (.class)



Hình 7 Mô hình cấu trúc của Execution Engine

Execution Engine gồm các thành phần:

- **Interpreter** (Trình thông dịch)

  Trình thông dịch có nhiệm vụ thông dịch bytecode. Nhược điểm của trình thông dịch là thực thi chậm vì nếu như một phương thức lặp lại nhiều lần thì trình thông dịch sẽ thông dịch bấy nhiêu lần.

- **JIT Compiler**

  Execution Engine dùng trình thông dịch (Interpreter) để thông dịch bytecode, khi gặp các code trùng lặp thì sẽ dùng JIT Compiler. JIT Compliler có nhiệm vụ biên dịch toàn bộ bytecode thành native code. Native code này dùng để thay thế các lời gọi hàm bị trùng lặp do đó hiệu năng của việc thực thi bytecode được cải thiện đáng kể.

  JIT Compliler thực hiện gồm các bước sau:

  - ✓ **Intermediate Code Generator**: tạo ra code trung gian
  - ✓ **Code Optomizer**: tối ưu code.

- ✓ **Target Code Gernerator**: tạo ra mã máy (machine language) hoặc native code.
- ✓ **Profiler**: tìm kiếm các phương thức lặp đi lặp lại nhiều lần.

➢ **Garbage Collection**

Garbage Collection có nhiệm vụ tìm kiếm và thu dọn các đối tượng đã tạo ra nhưng không được tham chiếu đến. Ta có thể kích hoạt thủ công bộ Garbage Collection (GC) thông qua lệnh "System.gc()".

4. Native Method Interface (Java Native Interface – JNI), Native Method Library

JNI - Java Native Interface sẽ tương tác với Native Method Libraries và cung cấp Native Libraries cần thiết cho Execution Engine.

III. Tài liệu tham khảo

[1] https://dzone.com/articles/understanding-jvm-internals

[2] http://www.artima.com/insidejvm/ed2/jvmP.html

[3] https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-2.html