Trường Đại Học Khoa Học Tự Nhiên
Đại Học Quốc Gia Hà Nội

Báo cáo môn Công Nghệ Phần Mềm

Đề tài: Tìm hiểu về cấu trúc Java Virtual Machine (JVM)

Giảng viên hướng dẫn: Bùi Sỹ Nguyên

Thực hiện: Nhóm 21.1
Nguyễn Công Hậu (18001127)
Đặng Quang Vinh (18001218)
Nguyễn Văn Huy (18001146)
Phạm Văn Khải (18001149)

Phòng thí nghiệm Khoa học dữ liệu
Khoa Toán – Cơ – Tin học

Hà Nội 11/2020

Lời Nói đầu

Tìm hiểu về cấu trúc Java Virtual Machine (JVM)

Thập kỷ 90 của thế kỷ 20 là sự phát triển nhanh như vũ bão của mạng Internet, kèm theo đó là vô vàn các ứng dụng trên các môi trường, hệ điều hành (OS) và các hệ xử lý (CPU) khác nhau. Tuy nhiên có một điểm hạn chế lớn là lập trình viên phải rất vất vả khi chuyển đổi các ứng dụng của mình để các hệ thống khác nhau có thể sử dụng được

23/5/1995, Sun Microsystems đã cho ra mắt công cụ lập trình **Java** với tiêu chí "**Viết một lần, chạy khắp nơi**" (Write Once, run anywhere), một ngôn ngữ lập trình hướng đối tượng (OOP) được thiết kế độc lập với hệ điều hành. Khác với phần lớn các ngôn ngữ lập trình thông thường, thay vì biên dịch mã nguồn thành mã máy hoặc thông dịch mã nguồn khi chạy, Java được thiết kế để biên dịch mã nguồn thành bytecode, sau đó sẽ được môi trường thực thi (runtime enviroment) chạy. Chương trình viết bằng Java có thể chạy trên mọi nền tảng (platform) khác nhau thông qua môi trường thực thi thích hợp hỗ trợ nền tảng đó. Môi trường thực thi của Java hiện hỗ trợ : Windows, Mac OS, Linux, Unix,...

Java Vitual Machine (JVM) bản chất là một chương trình có thể thực thi các đoạn mã lập trình của Java. Với máy ảo JVM, chương trình Java có thể chạy trên bất kỳ môi trường nào

Với vai trò to lớn và ứng dụng rộng rãi mà JVM mang lại cho thấy tầm quan trọng của JVM, song việc tìm hiểu JVM gặp rất nhiều khó khăn do chưa có tài liệu mô tả chi tiết, rõ ràng nên báo cáo này sẽ đóng góp một phần kiến thức về JVM cho bạn đọc

I. Giới thiệu chung về Java Virtual Machine (JVM)

1. Giới thiệu chung

a) Giới thiệu về Java Virtual Machine (JVM)

Tất cả các chương trình muốn thực thi được thì phải được biên dịch ra mã máy. Mỗi máy có hệ điều hành khác nhau (Windows, Mac Os, Linux,...) và kiến trúc CPU khác nhau (CPU intel, CPU macintosh,...) vì vậy trước đây mỗi chương trình chỉ có thể thực thi được trên một loại máy với hệ điều hành và kiến trúc CPU nào đó, như vậy muốn thực thi chương trình trên máy có cấu trúc khác thì phải chỉnh sửa và biên dịch lại mã nguồn.

Khi Java ra đời, nó nhờ vào máy ảo Java để khắc phục khó khăn đó. Một chương trình được viết bằng Java sẽ được biên dịch ra mã của máy ảo Java (bytecode). Sau đó máy ảo sẽ chuyển mã bytecode thành mã máy tương ứng

Máy ảo Java được sinh ra với 3 mục đích chính:

- Dịch mã Java ra mã máy chạy được trên các hệ điều hành khác nhau
- Tăng tốc độ
- Nâng cao độ bảo mật và tránh virus phá source code

b) Java Virtual Machine (JVM) là gì?

JVM bản chất là một chương trình có thể thực thi các đoạn mã lập trình của Java, và đặc điểm của các chương trình được viết bằng Java là đều có thể chạy trên bất kỳ môi trường nào miễn là có cài máy ảo JVM.

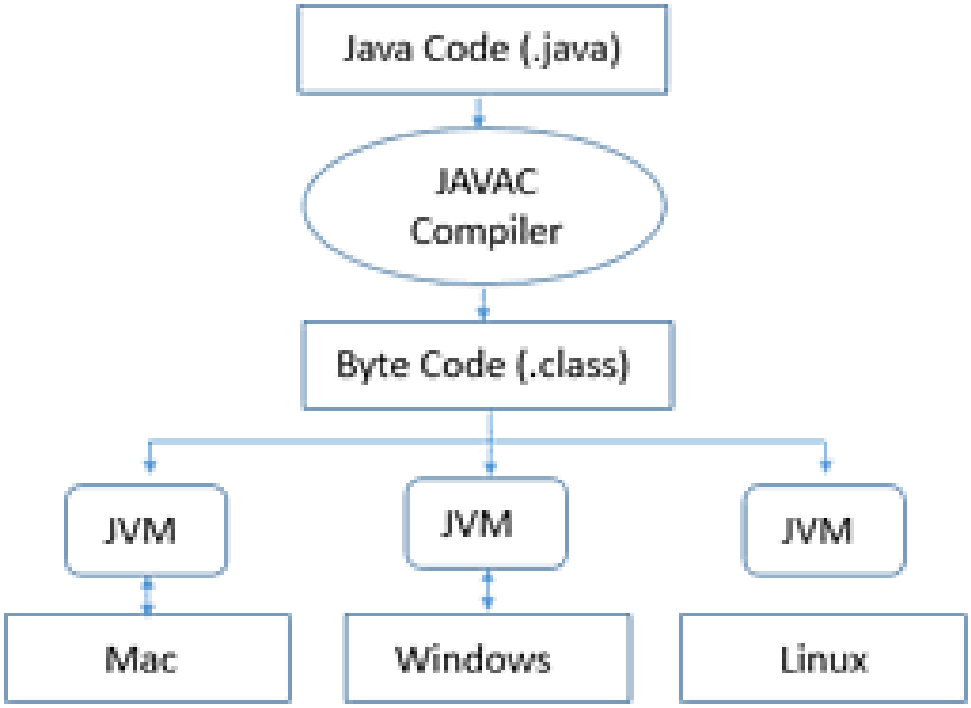
JVM quản lý bộ nhớ hệ thống và cung cấp môi trường thực thi cho ứng dụng Java, nó có hai chức năng chính là cho phép chương trình Java chạy trên mọi thiết bị, nền tảng khác nhau và quản lý, tối ưu bộ nhớ chương trình.

Trong hầu hết các trường hợp, các ngôn ngữ lập trình khác, trình biên dịch tạo ra mã cho một Hệ điều hành cụ thể nhưng trình biên dịch Java chỉ tạo Bytecode cho **Máy ảo Java** . Khi bạn chạy một chương trình Java, chương trình này sẽ chạy như một chuỗi trong quy trình JVM. JVM có trách nhiệm tải các tệp lớp của bạn, xác minh mã, diễn giải và thực thi chúng. Khi bạn phát hành một lệnh như java, JVM tải định nghĩa lớp cho lớp cụ thể đó và gọi phương thức chính của lớp đó.

Khi các nhà phát triểu nói về JVM họ thường nghĩ tới các chương trình thực thi trong máy, đặc biệt là máy chủ, nó kiểm soát việc sử dụng tài nguyên cho ứng dụng Java. Điều đó khác với định nghĩa kỹ thuật của JVM, nó tuân thủ theo một đặc tả cho chương trình phầm mềm, miêu tả những yêu cầu sửa dụng chương trình và cung cấp môi trường thực thi code.

II. Kiến trúc của JVM

Sơ đồ đơn giản về máy ảo Java:

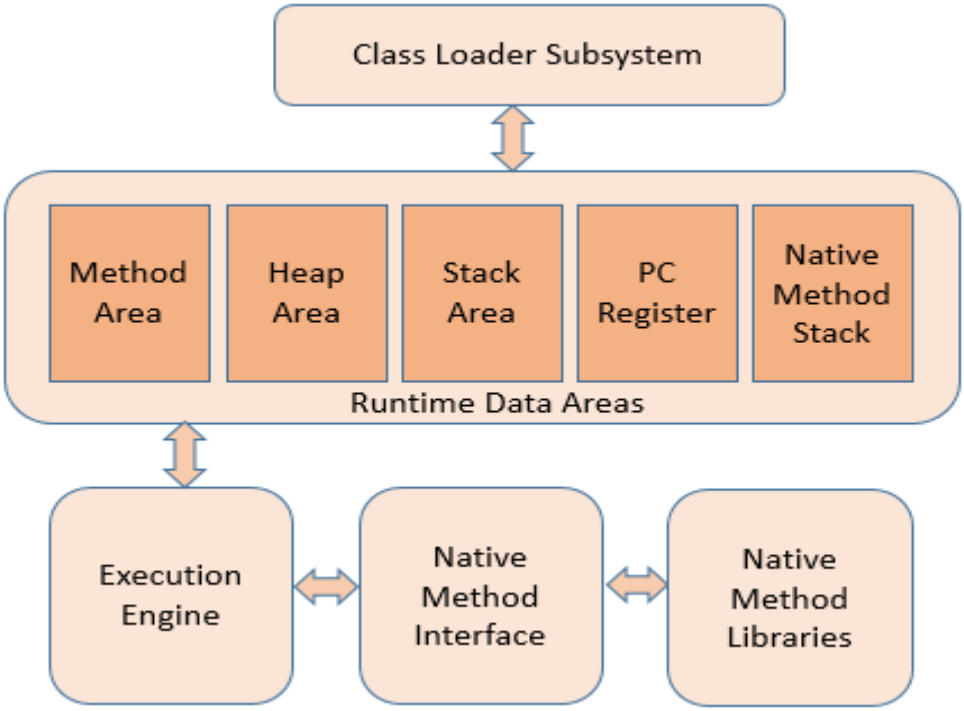


Sau khi biên dịch file .java sang file .class máy ảo JVM sẽ chuyển hóa thành các mã máy ứng với các hệ điều hành tường ứng.

Các bước mà JVM vận hành:

- Loads code (Tải mã)
- Verifies code (Xác minh mã)
- Executes code (Thực thi mã)

Đi sâu vào nội bộ của máy ảo Java



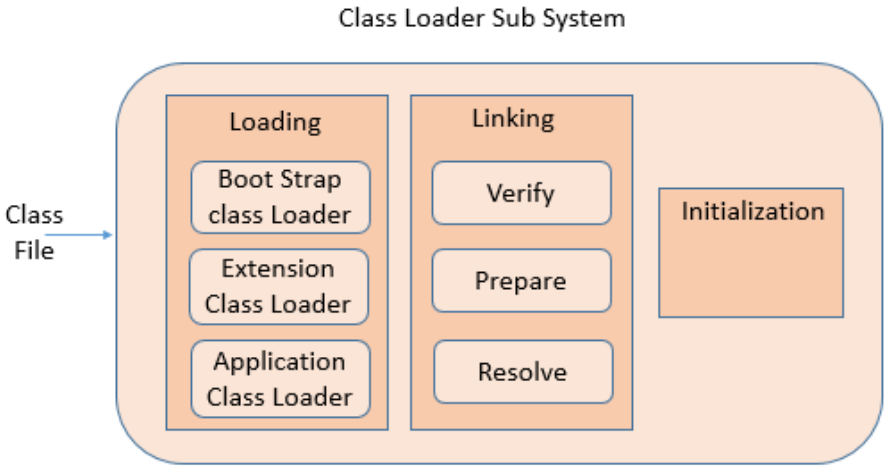
JVM được chia ra làm 3 hệ thống con chính:

Class Loader Subsystem: Tìm kiếm và nạp các file .class vào vùng nhớ Java

Runtime Data Area: Vùng nhớ hệ thống cấp phát cho JVM

Execution Engine: Chuyển các lệnh trong file .class thành mã máy tương ứng

1. Class Loader Subsystem

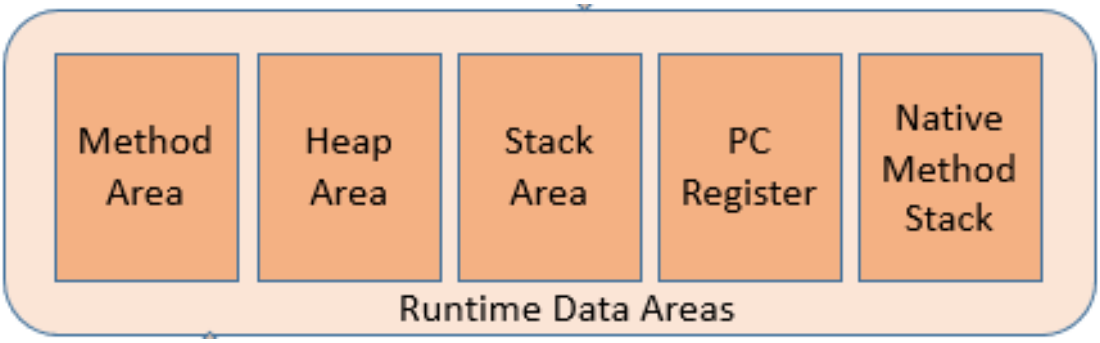


Class Loader Subsystem chịu trách nhiệm **load**, **linking** và **initialization** (khởi tạo) file **.class**:

- **Loading:** là hoạt động đầu tiên, nó tải các file **.class** (nhị phân) vào vùng **method area.** Gồm 3 class loader mặc định tuân theo sự phân cấp:
  - **Boot Strap class Loader:** chịu trác nhiệm load các class từ classpath, đầu tiên là các file **.jar**. Ưu tiên cao nhất được trao cho class loader này
  - **Extension Class Loader:** chịu trách nhiệm load các class nằm trong các thư mục mở rộng của **JRE** như **lib/ext**
  - **Application Class Loader:** chịu trác nhiệm load các class được đề cập đến ở trong biến môi trường
- **Linking:** là quá trình kết hợp các đoạn mã bytecode đã tải :
  - **Verify:** Bytecode verifier (trình xác minh bytecode) sẽ kiểm tra xem bytecode có được tạo ra phù hợp hay không, nếu không sẽ thông báo lỗi verify
  - **Prepare:** Đối với tất cả các bộ nhớ cho biến tĩnh sẽ được phân bổ và gán giá trị mặc định.
  - **Resolve:** các bộ nhớ tham chiếu tượng trưng được thay thế bằng các tham chiếu ban đầu từ **method area**
- **Initialization:** Đây là giai đoạn cuối của Class Loading. Trong giai đoạn này các biến tĩnh (**static variables)** sẽ được gán với các giá trị ban đầu và static block sẽ được thực thi

2. Runtime Data Areas

- Các máy ảo Java (JVM) định nghĩa Runtime Data Areas khác nhau được sử dụng trong thực thi một chương trình. Một số vùng dữ liệu này được tạo khi khởi động máy ảo Java và chỉ bị hủy khi thoát khỏi máy ảo Java. Mỗi vùng dữ liệu là mỗi luồng khác nhau. Các vùng dữ liệu của mỗi luồng được tạo khi một luồng được tạo và bị hủy khi luồng đó thoát



Runtime Data Area được chia làm 5 thành phần chính:

- **Method Area:** là phần bộ nhớ được tạo ngay khi khởi động máy ảo Java. Chứa code đã biên dịch, các phương thức, dữ liệu và các trường của nó (VD: static variables,..)
- **Heap Area:** Tất cả các đối tượng, các biến, arrays được lưu trữ tại đây. Mỗi khi một đối tượng được tạo trong Java, nó sẽ đi vào vùng nhớ **Heap**
- **Stack Area:** là nơi bộ nhớ lưu trữ các phương thức và các biến cục bộ. Biến tham chiếu (tham chiếu nguyên thủy hoặc tham chiếu đối tượng) được lưu trữ trong **Stack**
- **PC Register:** Với mỗi luồng sẽ được chia vào PC Register riêng để giữ địa chỉ của lệnh hiện tại sau khi lệnh đó được thực thi

- **Native Method Stack:** Native method là những phương thức được viết bằng ngôn ngữ khác Java (C,C++…). **JVM Implementations** không thể không thể load các native method và không thể dựa vào các ngăn xếp thông thường.

3. Execution Engine

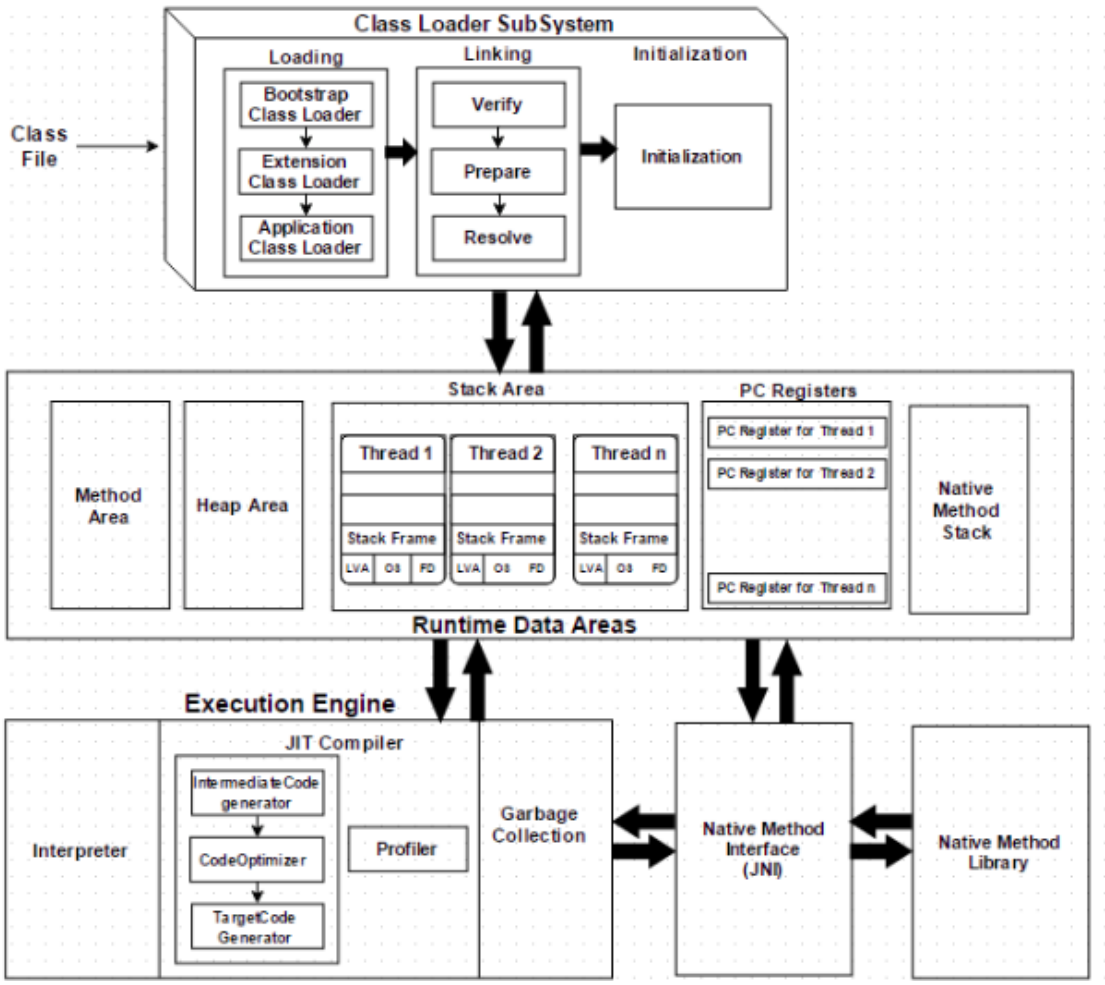
Đây là cốt lõi của JVM. Execution Engine có thể giao tiếp với các vùng nhớ khác nhau của JVM. Mỗi luồng của một ứng dụng Java đang chạy là một execution riêng biệt. bytecode được gán cho **Runtime Data Areas** trong JVM thông qua **Class Loader** được thực thi bởi execution Engine. Execution Engine đọc bytecode từng mảng một

- **Interperter:** Đọc, diễn giải và thực hiện từng bytecode một dẫn đến việc thông dịch bytecode nhanh nhưng thực thi lại chậm. Nhược điểm của trình thông dịch là khi phương thức được gọi nhiều lần, mỗi lần cần một thông dịch mới.
- **JIT Compiler:** Trình biên dịch JIT vô hiệu hóa nhược điểm của thông dịch(**Interperter). Execution Engine** sẽ sử dụng sự trợ giúp của thông dịch trong việc chuyển đổi bytecode, nhưng khi thấy mã lặp lại, nó sẽ sử dụng trình biên dịch **JIT.** Nó biên dịch toàn bộ bytecode thành mã gốc , mã này được sử dụng khi các phương thức bị gọi lặp lại nhiều lần, điều này giúp cải thiện hiệu năng hệ thống.
- **Garbage Collector:** thu thập và loại bỏ các đối tượng được khởi tạo nhưng không sử dụng.

4. Native Method Interface và Native Method Libraries

- **Native Method Interface:** cho phép bạn sử dụng code từ các ngôn ngữ khác như C, C++, … trong code Java của bạn khi java không cung cấp các chức năng mà bạn cần
- **Native method Libraries:** là tập hợp các thư viện **Native method** mà **Execution Engine** yêu cầu

■ Cấu trúc JVM đầy đủ:



Cùng với JDK (Java Development Kit) và JRE (Java Runtime Environment), thì JVM (Java Virtual Machine) là một thành phần quan trọng của Java, giúp hỗ trợ phát triển và thực thi các ứng dụng Java. Chúng đã góp phần làm Java mạnh mẽ như ngày hôm nay.

Mục lục