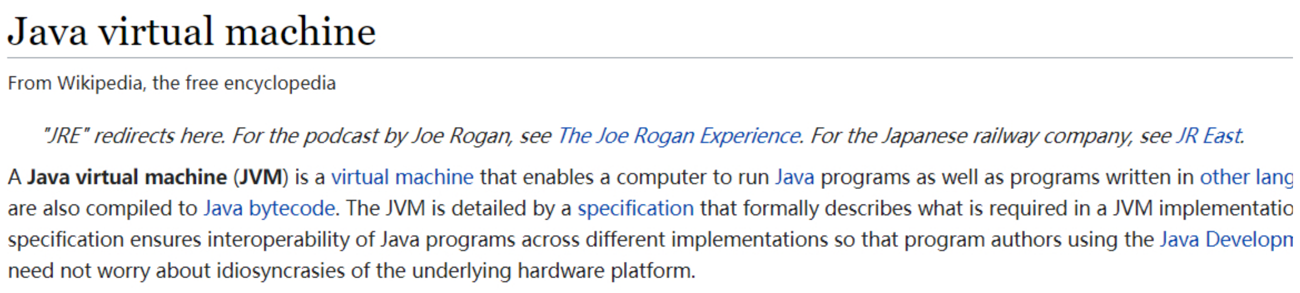
01 Introduction of JVM

1.1 What is JVM

> From Wikipedia



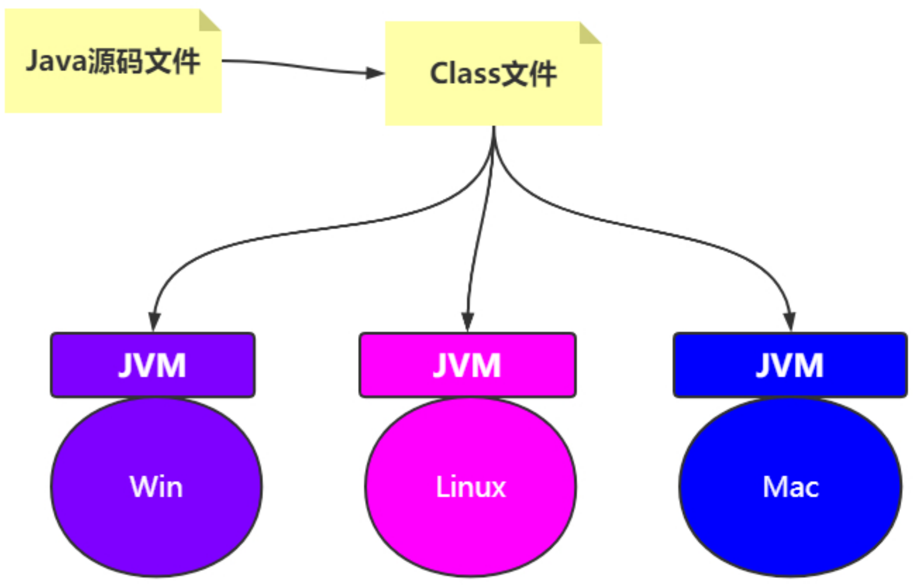
Java virtual machine

From Wikipedia, the free encyclopedia

"JRE" redirects here. For the podcast by Joe Rogan, see The Joe Rogan Experience. For the Japanese railway company, see JR East.

A **Java virtual machine** (**JVM**) is a virtual machine that enables a computer to run Java programs as well as programs written in other lang
are also compiled to Java bytecode. The JVM is detailed by a specification that formally describes what is required in a JVM implementatio
specification ensures interoperability of Java programs across different implementations so that program authors using the Java Developn
need not worry about idiosyncrasies of the underlying hardware platform.

> How to understand

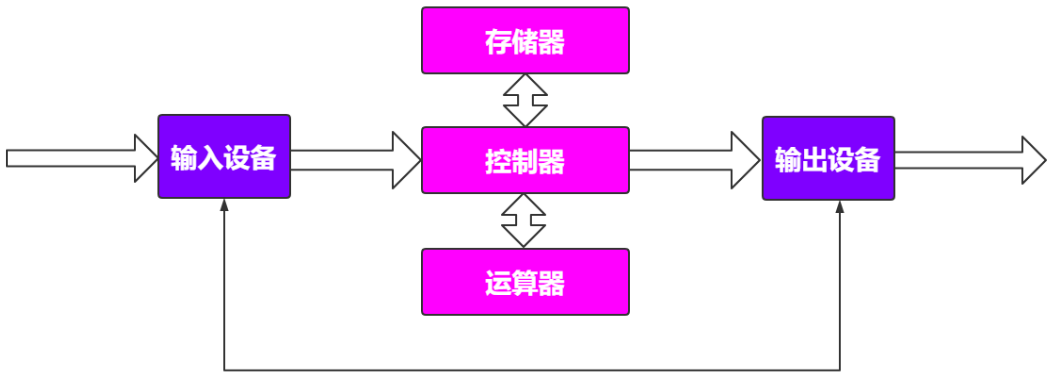


> Java虚拟机与物理机

Class文件类比输入设备

CPU指令集类比输出设备

JVM类比存储器、控制器、运算器等



1.2 JVM products

> 最常用的目前是HotSpot，可以通过java -version命令查看

Oracle：HotSpot、JRockit

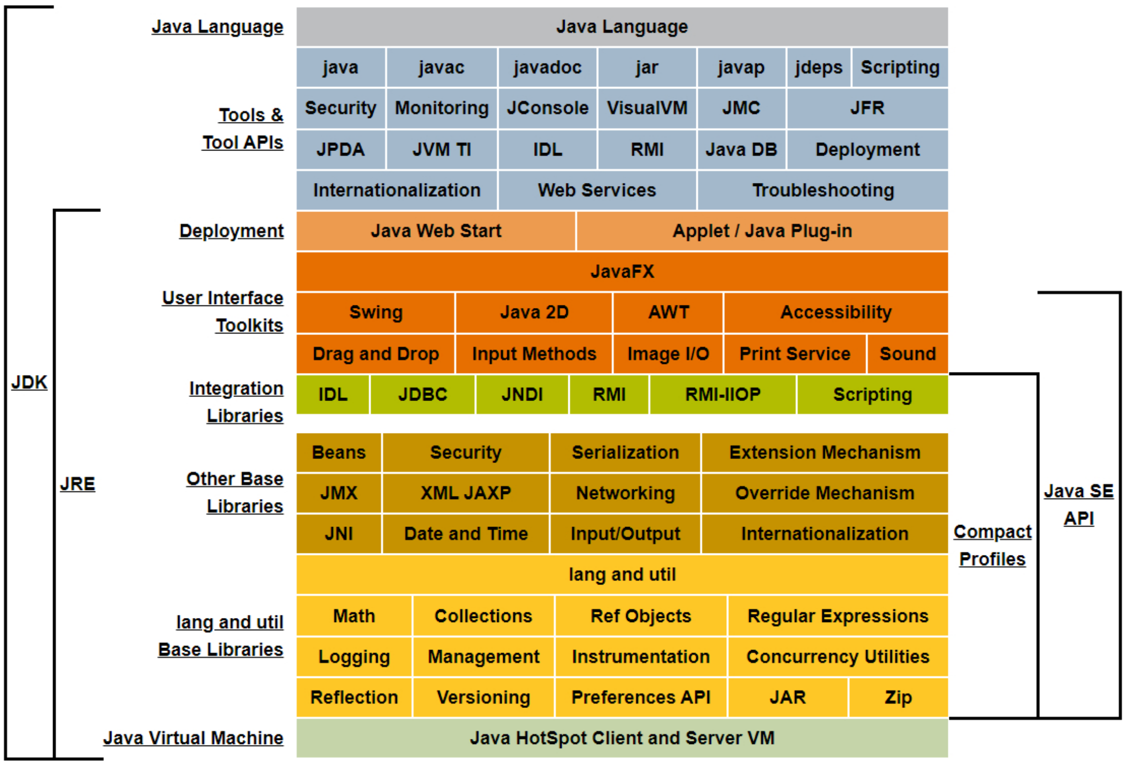
IBM：J9 VM

Ali：TaobaoVM

Zual：Zing

1.3 JDK JRE JVM

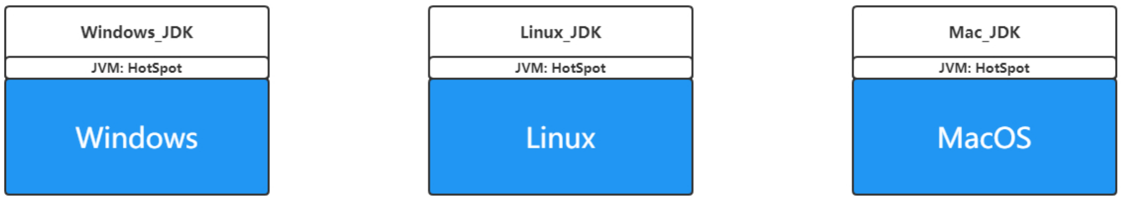
> 官网: https://docs.oracle.com/javase/8/docs/index.html



1.4 结合JDK看JVM

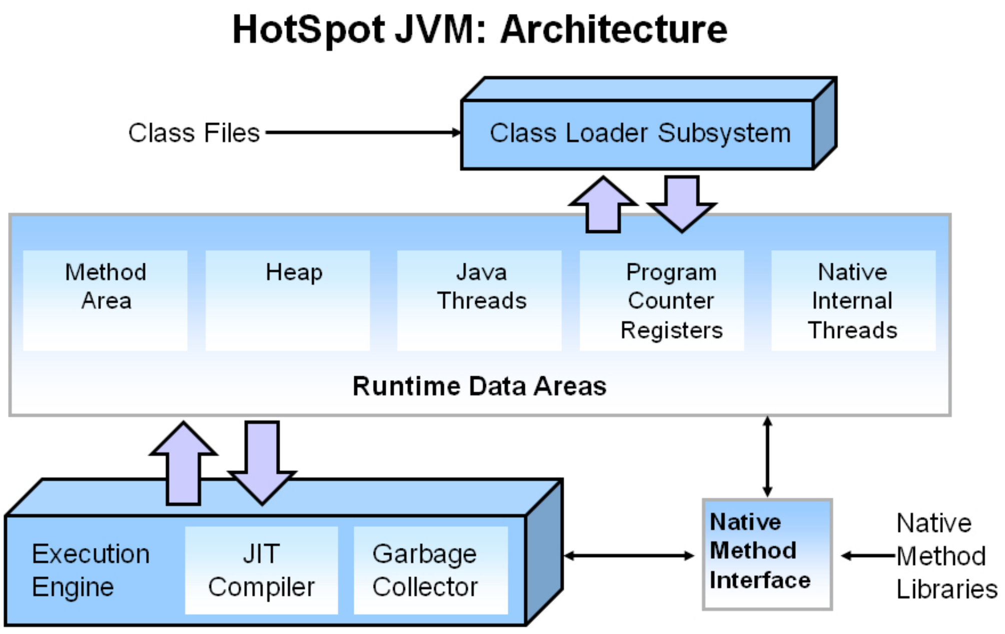
（1）能够把Class文件翻译成不同平台的CPU指令集

（2）也是Write Once Run Anywhere的保证



1.5 HotSpot JVM Architecture

> https://www.oracle.com/technetwork/tutorials/tutorials-1876574.html



02 Class file

2.1 Java Source

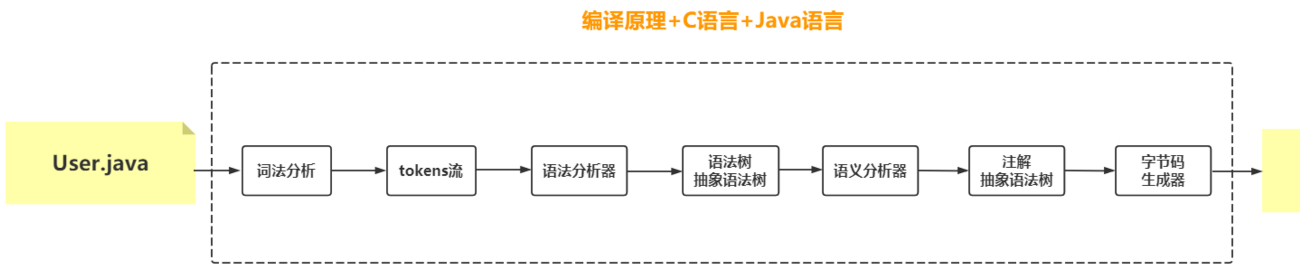
> Java源码文件、Kotlin源码文件、Groovy源码文件等

```java
public class User {
 private Integer age;
 private String name="Jack";
 private Double salary=100.0;
 private static String address;
 public void say(){
 System.out.println("Jack Say...");
 }
 public static Integer calc(Integer op1,Integer op2){
  op1=3;
  Integer result=op1+op2;
  return result;
  }

  public static void main(String[] args) {
  System.out.println(calc(1,2));
  }
 }
```

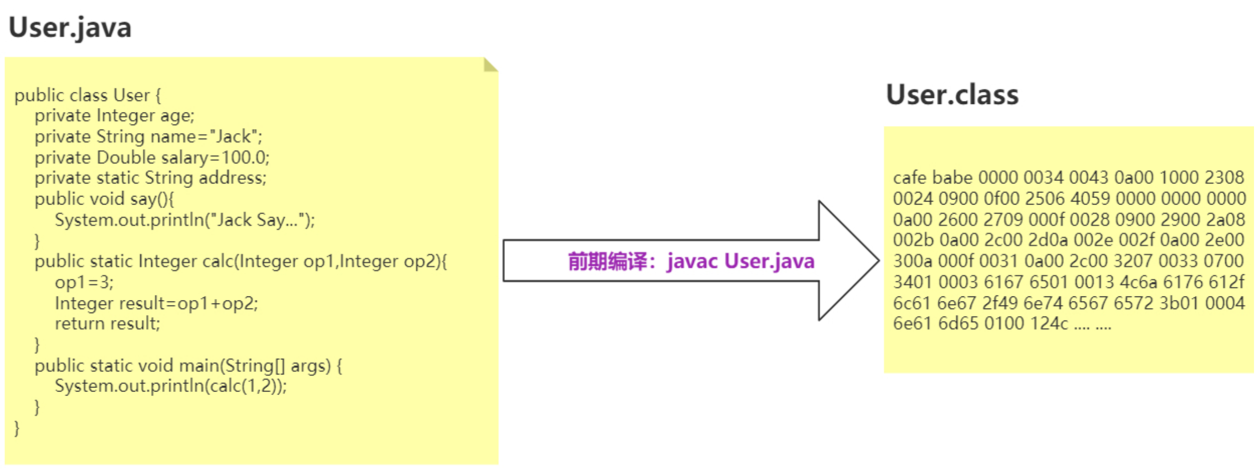
2.2 Early compile

> javac User.java->User.class



2.3 Class format

> https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html



```
public class User {
    private Integer age;
    private String name="Jack";
    private Double salary=100.0;
    private static String address;
    public void say(){
        System.out.println("Jack Say...");
    }
    public static Integer calc(Integer op1,Integer op2){
        op1=3;
        Integer result=op1+op2;
        return result;
    }
    public static void main(String[] args) {
        System.out.println(calc(1,2));
    }
}
```

```
cafe babe 0000 0034 0043 0a00 1000 2308
0024 0900 0f00 2506 4059 0000 0000 0000
0a00 2600 2709 000f 0028 0900 2900 2a08
002b 0a00 2c00 2d0a 002e 002f 0a00 2e00
300a 000f 0031 0a00 2c00 3207 0033 0700
3401 0003 6167 6501 0013 4c6a 6176 612f
6c61 6e67 2f49 6e74 6567 6572 3b01 0004
6e61 6d65 0100 124c .... ....
```

```
ClassFile {
 u4 magic;
 u2 minor_version;
 u2 major_version;
 u2 constant_pool_count;
 cp_info constant_pool[constant_pool_count-1];
 u2 access_flags;
 u2 this_class;
 u2 super_class;
 u2 interfaces_count;
 u2 interfaces[interfaces_count];
 u2 fields_count;
 field_info fields[fields_count];
 u2 methods_count;
 method_info methods[methods_count];
 u2 attributes_count;
 attribute_info attributes[attributes_count];
}
```

2.4 Analyse

(1) cafebabe

```
magic:The magic item supplies the magic number identifying the class file format
```

(2) 0000+0034：minor_version+major_version

```
16进制的34等于10进制的52，表示JDK的版本为8
```

(3) 0043：constant_pool_count

```
The value of the constant_pool_count item is equal to the number of entries in the constant_pool table plus one
16进制的43等于10进制的67，表示常量池中常量的数量是66
```

(4) cp_info：constant_pool[constant_pool_count-1]

```
The constant_pool is a table of structures representing various string constants, class and interface names, field names, and other constants that are referred to within the ClassFile structure and its substructures. The
format of each constant_pool table entry is indicated by its first "tag" byte.
The constant_pool table is indexed from 1 to constant_pool_count - 1.
字面量：文本字符串，final修饰的常量等
符号引用：类和接口的全限定名、字段名称和描述符、方法名称和描述符
```

(5) The constant pool

> 官网: https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.4

All constant_pool table entries have the following general format:

```
cp_info {
 u1 tag;
```

```
  u1 info[];
}
```

Table 4.4-A. Constant pool tags

Constant Type	Value
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

（6）First constant

由0a可以知道第一个常量的类型对应的10进制为10，所以这表示一个方法引用，查找方法引用对应的structure

```
CONSTANT_Methodref_info {
  u1 tag;
  u2 class_index; // 代表的是class_index，表示该方法所属的类在常量池中的索引
  u2 name_and_type_index; // 代表的是name_and_type_index，表示该方法的名称和类型的索引
}
```

经过分析可以得出第一个常量表示的形式为

```
#1 = Methodref #16,#35
```

（7）Second constant

由08可以知道第一个常量的类型对应的10进制为8，所以这表示一个String引用，查找方法引用对应的structure

```
CONSTANT_String_info {
  u1 tag;
  u2 string_index; // 代表的是string_index
}
```

经过分析可以得出第1、2个常量表示的形式为

```
#1 = Methodref #16,#35
#2 = String #36
```

2.5 反汇编

> JDK自带的命令
> javap -h

```
javap -v -c -p User.class > User.txt 进行反编译，查看字节码信息和指令等信息
```

JVM相对class文件来说可以理解为是操作系统；class文件相对JVM来说可以理解为是汇编语言或者机器语言。

03 类加载机制

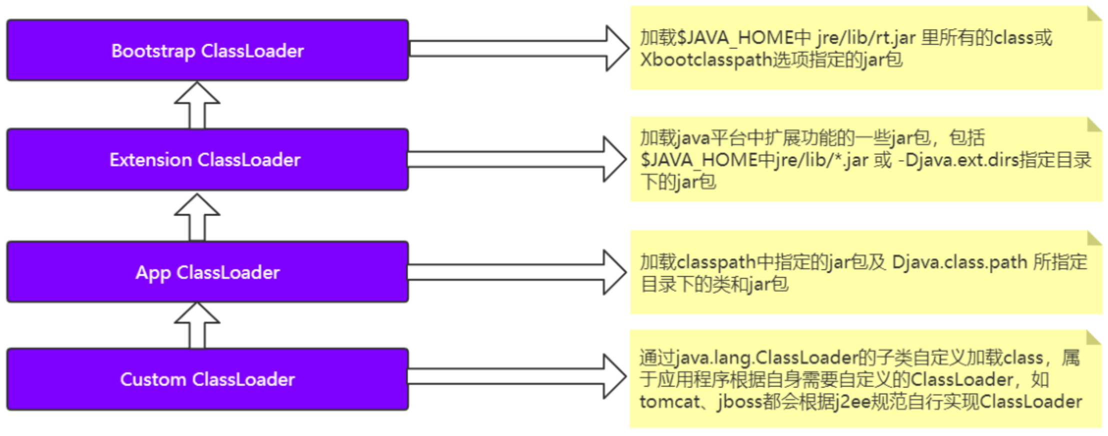
官网：https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-5.html

```
The Java Virtual Machine dynamically loads, links and initializes classes and interfaces. Loading is the process of finding the binary representation of a class or interface type
```

```
with a particular name and creating a class or interface from that binary representation. Linking is the process of taking a class or interface and combining it into the run-time state of the Java Virtual Machine
so that it can be executed. Initialization of a class or interface consists of executing the class or interface initialization method <clinit>
```

3.1 Loading

3.1.1 ClassLoader


Bootstrap ClassLoader
加载$JAVA_HOME中 jre/lib/rt.jar 里所有的class或Xbootclasspath选项指定的jar包
Extension ClassLoader
加载java平台中扩展功能的一些jar包，包括$JAVA_HOME中jre/lib/*.jar 或 -Djava.ext.dirs指定目录下的jar包
App ClassLoader
加载classpath中指定的jar包及 Djava.class.path 所指定目录下的类和jar包
Custom ClassLoader
通过java.lang.ClassLoader的子类自定义加载class，属于应用程序根据自身需要自定义的ClassLoader，如tomcat、jboss都会根据j2ee规范自行实现ClassLoader

3.1.2 双亲委派机制

（1）检查某个类是否已经加载

自底向上，从Custom ClassLoader到BootStrap ClassLoader逐层检查，只要某个Classloader已加载，就视为已加载此类，保证此类只所有ClassLoader加载一次。

（2）加载的顺序

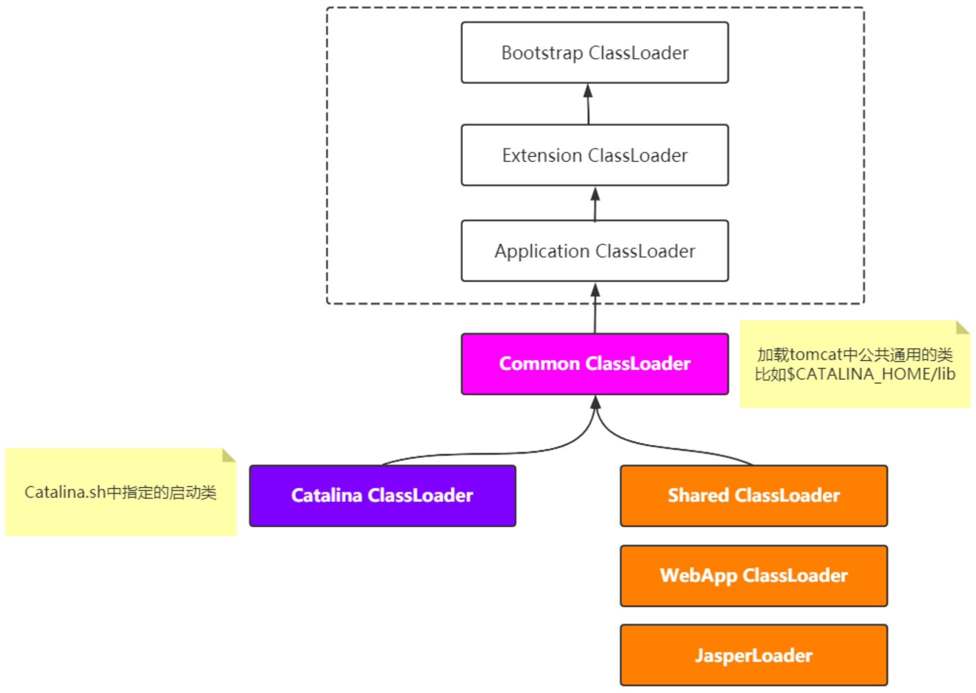
自顶向下，也就是由上层来逐层尝试加载此类。

3.1.3 代码体验

```java
public class Demo {
 public static void main(String[] args) {
 // App ClassLoader
 System.out.println(new Worker().getClass().getClassLoader());
 // Ext ClassLoader
 System.out.println(new Worker().getClass().getClassLoader().getParent());
 // Bootstrap ClassLoader
 System.out.println(new Worker().getClass().getClassLoader().getParent().getParent());
 System.out.println(new String().getClass().getClassLoader());
  }
}
```

3.1.4 破坏双亲委派

（1）tomcat



（2）SPI机制

（3）OSGi

3.2 Linking

3.2.1 Verification

保证被加载类的正确性

3.2.2 Preparation

为类的静态变量分配内存，并将其初始化为默认值

3.2.3 Resolution

把类中的符号引用转换为直接引用

3.3 Initialization

对类的静态变量，静态代码块执行初始化操作

04 运行时数据区

官网：https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html#jvms-2.5

```
The Java Virtual Machine defines various run-time data areas that are used during execution of a program.
Some of these data areas are created on Java Virtual Machine start-up and are destroyed only when the Java Virtual Machine exits.
Other data areas are per thread. Per-thread data areas are created when a thread is created and destroyed when the thread exits.
```

4.1 Method Area(方法区)

> JVM运行时数据区是一种规范，真正的实现在JDK 8中就是Metaspace，在JDK6或7中就是Perm Space

方法区是各个线程共享的内存区域，在虚拟机启动时创建，虽然Java虚拟机规范把方法区描述为堆的一个逻辑部分，但是它却又一个别名叫做Non-Heap(非堆)，目的是与Java堆区分开来。

用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。当方法区无法满足内存分配需求时，将抛出OutOfMemoryError异常。

4.1.1 常量池和运行时常量池

The Constant Pool：https://docs.oracle.com/javase/specs/jvms/se17/html/jvms-4.html#jvms-4.4

The Run-time Constant Pool：https://docs.oracle.com/javase/specs/jvms/se17/html/jvms-2.html#jvms-2.5.5

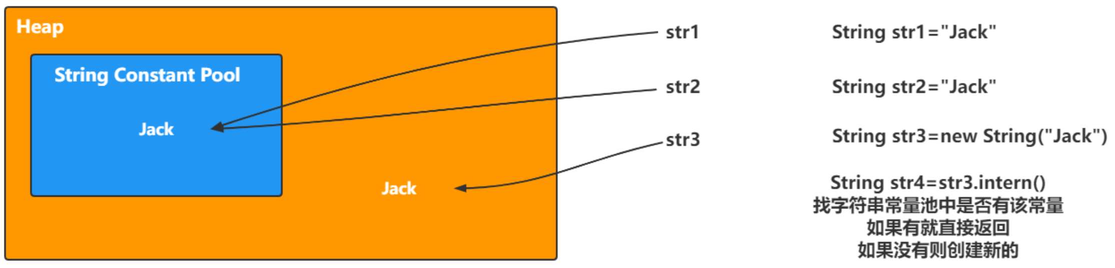
4.1.2 String常量到底存在哪

```java
public class SCPDemo {
public static void main(String[] args) {
String str1="Jack"; // 这个常量一定会放到字符串常量池中
String str2="Jack";
String str3=new String("Jack");
String str4=str3.intern(); // 找字符串常量池中是否有该常量，如果有就直接返回，如果没有再创建

// equals 只会比较值 == 会比较地址
System.out.println(str1.equals(str2)); // true
System.out.println(str1==str2); // true

System.out.println(str1.equals(str3)); // true
System.out.println(str1==str3); // false

System.out.println(str1.equals(str4)); // true
System.out.println(str1==str4); // true
}
}
```



4.2 Heap(堆)

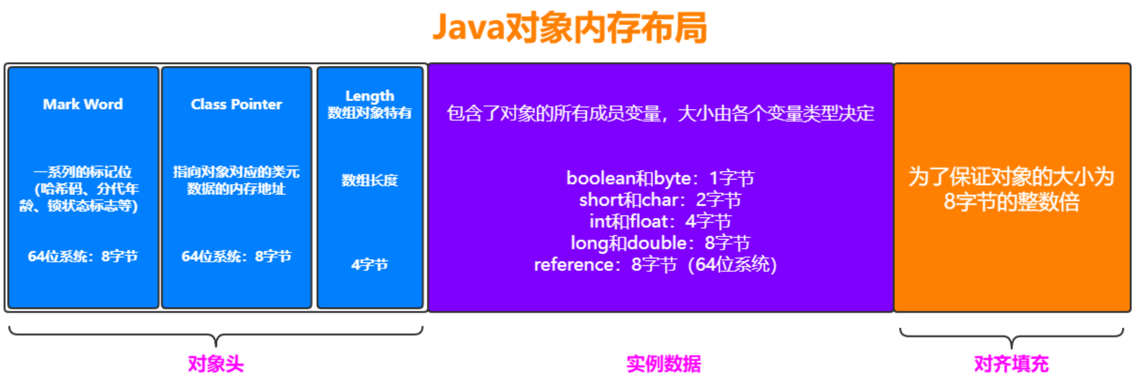
（1）Java堆是Java虚拟机所管理内存中最大的一块，在虚拟机启动时创建，被所有线程共享

（2）Java对象实例以及数组都在堆上分配

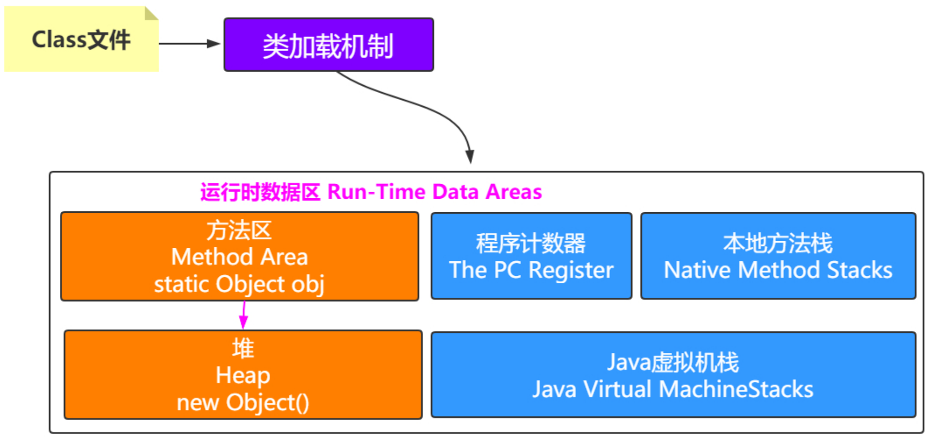
（3）堆内存空间不足时，也会抛出OOM

4.2.1 Java对象内存布局

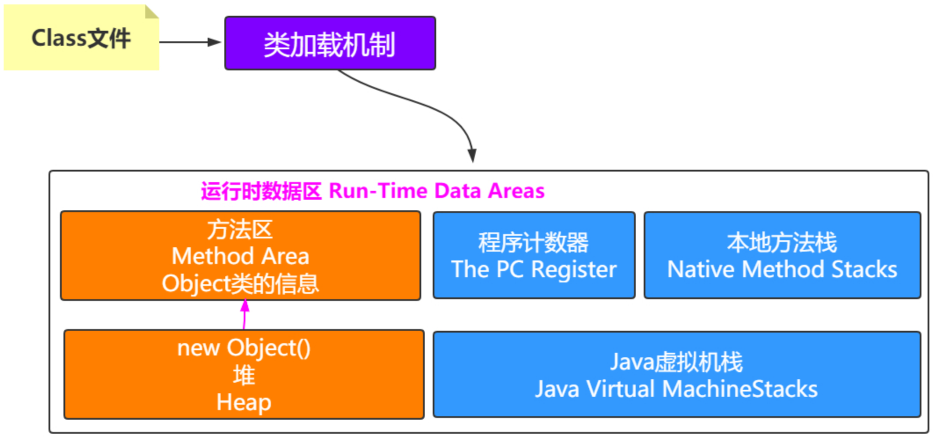
> 一个Java对象在内存中包括3个部分：对象头、实例数据和对齐填充



4.2.2 方法区引用指向堆



4.2.3 堆指向方法区



4.3 Java Virtual Machine Stacks(Java虚拟机栈)

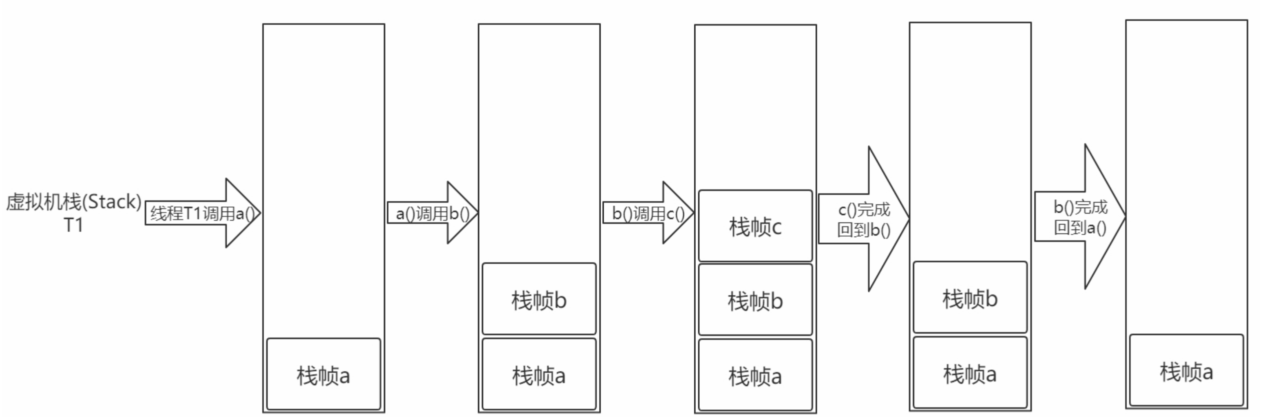
（1）虚拟机栈是一个线程执行的区域，保存着一个线程中方法的调用状态。换句话说，一个Java线程的运行状态，由一个虚拟机栈来保存，所以虚拟机栈肯定是线程私有的，独有的，随着线程的创建而创建。

（2）每一个被线程执行的方法，为该栈中的栈帧，即每个方法对应一个栈帧。调用一个方法，就会向栈中压入一个栈帧；一个方法调用完成，就会把该栈帧从栈中弹出。

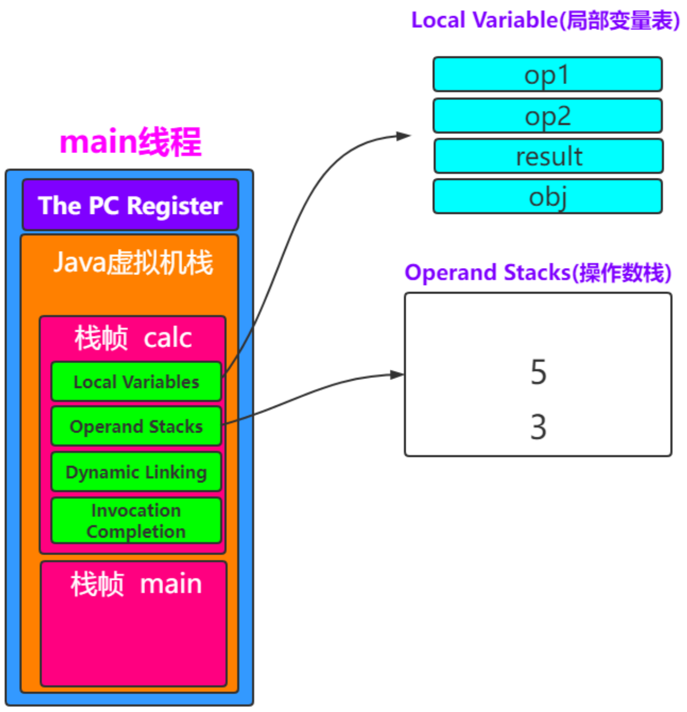
4.3.1 代码

```
void a(){
b();
}
void b(){
c();
}
void c(){
}
```

4.3.2 压栈出栈



4.3.3 Frame(栈帧)



4.3.4 字节码

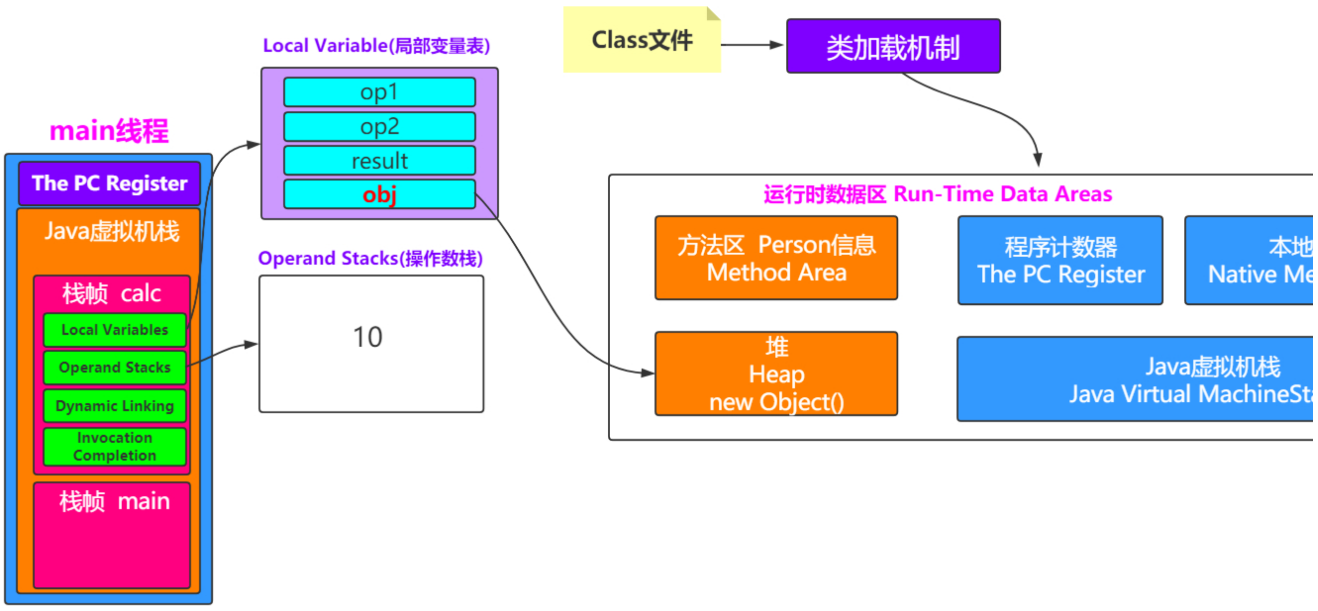
```
public static java.lang.Integer calc(java.lang.Integer, java.lang.Integer);
 descriptor: (Ljava/lang/Integer;Ljava/lang/Integer;)Ljava/lang/Integer;
 flags: ACC_PUBLIC, ACC_STATIC
 Code:
 stack=2, locals=3, args_size=2
 0: iconst_3 // 将int类型常量3压入[操作数栈]
 1: invokestatic #11 // Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
 4: astore_0 // 将int类型值存入[局部变量0]
 5: aload_0 // 从[局部变量0]中装载int类型值入栈
  6: invokevirtual #12 // Method java/lang/Integer.intValue:()I
  9: aload_1 // 从[局部变量1]中装载int类型值入栈
  10: invokevirtual #12 // Method java/lang/Integer.intValue:()I
  13: iadd // 将栈顶元素弹出栈，执行int类型的加法，结果入栈
  14: invokestatic #11 // Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
  17: astore_2 // 将栈顶int类型值保存到[局部变量2]中
  18: aload_2 // 从[局部变量2]中装载int类型值入栈
  19: areturn // 从方法中返回int类型的数据
  LineNumberTable:
  line 11: 0
  line 12: 5
  line 13: 18
```

4.3.5 index为0还是1

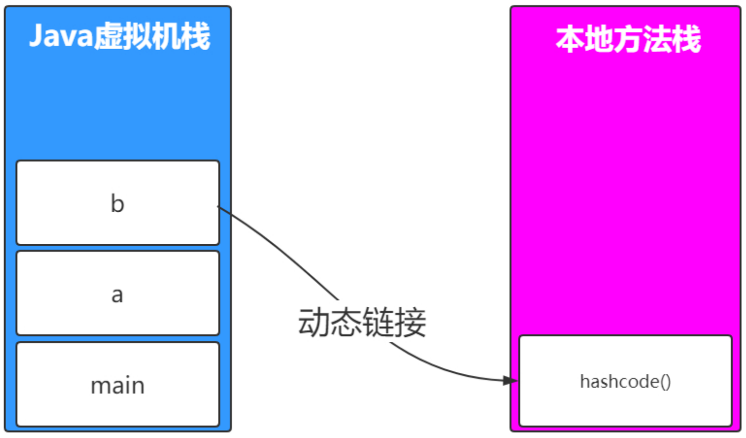
对于Java虚拟机栈中的Local Variables，到底是从0开始还是1开始，要看当前方法是static还是实例方法。

```
The Java Virtual Machine uses local variables to pass parameters on method invocation. On class method invocation, any parameters are passed in consecutive local variables starting from local variable 0.
On instance method invocation, local variable 0 is always used to pass a reference to the object on which the instance method is being invoked (this in the Java programming language).
Any parameters are subsequently passed in consecutive local variables starting from local variable 1.
```

4.3.6 栈引用指向堆



4.4 Native Method Stacks(本地方法栈)



4.5 The pc Register

如果线程正在执行Java方法，则计数器记录的是正在执行的虚拟机字节码指令的地址。如果正在执行的是Native方法，则这个计数器为空。