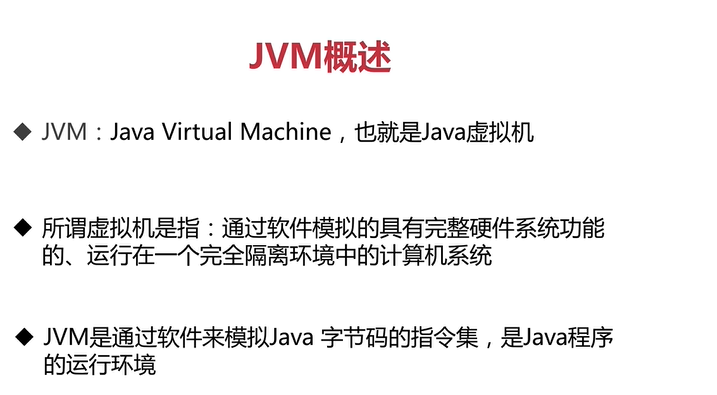
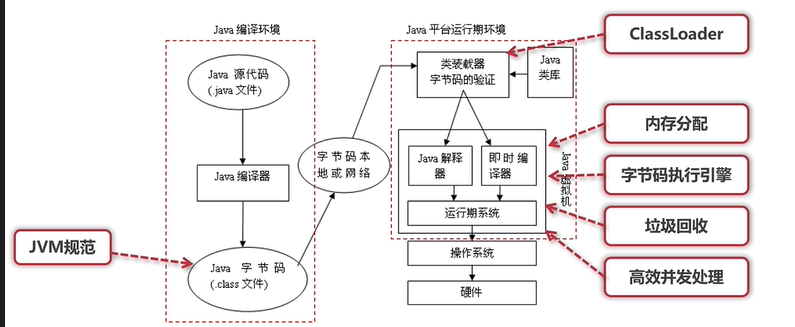
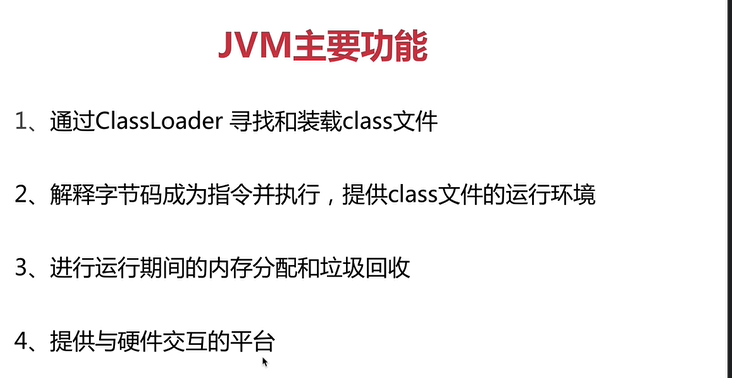
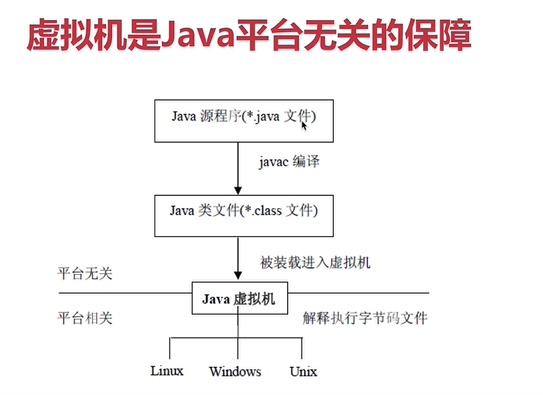
# JVM概述

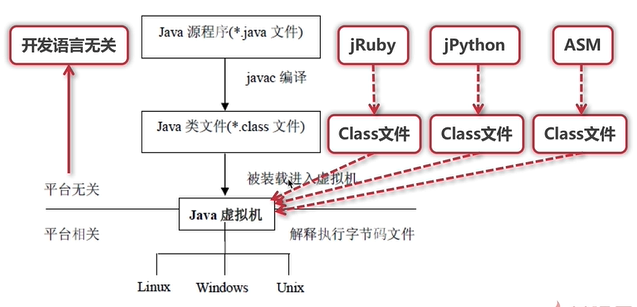




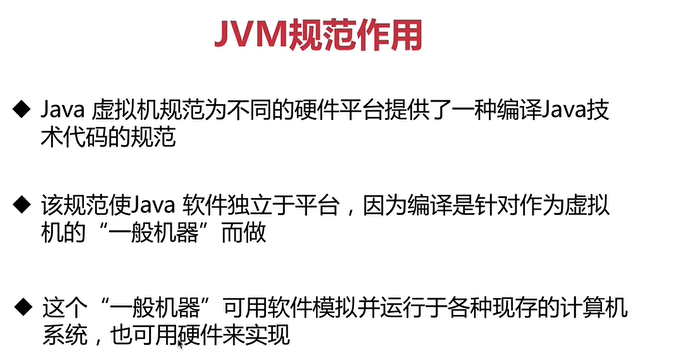




**jvm只认\*.class文件，与开发语言无关**



# JVM规范作用



# ASM开发

## ASM概述：

\* ASM是一个java字节码操纵框架，它能被用来动态生成类或者增强既有类的功能

\* ASM可以直接产生二进制class文件，也可以在类被加载入虚拟机之前动态改变类的行为，ASM从类文件中读入信息后，能够改变类行为，分析类信息，甚至能根据要求生成新类

\* 目前许多框架如cglib，hibernate，spring都直接或间接的使用ASM操作字节码

## ASM编程模型：

\* core api :提供了基于事件形式的变成模型。该模型不需要一次性将整个类的结构读取到内存中，因此这种方式更快，需要更少的内存，但这种编程方式难度较大

\* tree api：提供了基于树形的编程模型。该模型需要一次性将一个类的完整结构全部读取到内存当中，所以这种方法需要更多的内存，这种编程方式较简单

## ASM的Core API-1

\* ASM Core API中操纵字节码的功能基于CLassVisitor接口，这个接口中的每个方法对应了class文件中的每一项

\* ASM提供了三个基于ClassVisitor接口的类来实现class文件的生成和转换

1：classReader解析一个类的class字节码

2：ClassAdapter：ClassAdapeter是ClassVisitor的实现类

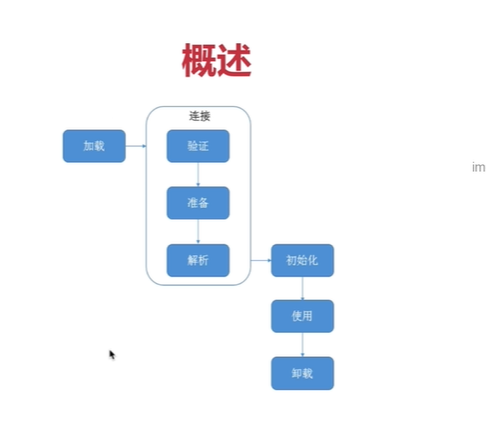
3：ClassWriter：ClassWriter也是ClassVisitor的实现类，可以用来输出变化后的字节码

\*ASM给我们提供了ASMifier工具来帮助开发，可使用ASMigier工具生成ASM结构来对比

# 类加载，连接和初始化

## 类加载和类加载器

类从被加载到jvm开始到卸载出内存，整个生命周期如图



\* 加载：查找并加载类文件的二进制数据

\* 连接：将已经读入内存的类的二进制数据合并到jvm运行时环境中去，包含如下几个步骤

1验证：确保被加载类的正确性

2准备：为类的静态变量分配内存，并初始化他们

3 解析：把常量池中的符号引用转换成直接引用

\* 初始化：为类的静态变量赋初始值

### 类加载要完成的功能

1 通过类的全限定名来获取该类的二进制字节流

2 把二进制字节流转化为方法区的运行时数据结构

3 在堆上创建一个java.lang.Class对象，用来封装类在方法区内的数据结构，并向外提供了访问方法区内数据结构的接口

### 加载类的方式

\* 最常见的方式：本地文件系统中加载，从jar等归档文件中加载

\* 动态的方式：将java源文件动态编译成class

\* 其他方式：网络下载，从专有数据库中加载等等

### 类加载器

java虚拟机自带的加载器包括如下几种：

启动类加载器（BootstrapClassLoader）

扩展类加载器 （ExtensionClassLoader）

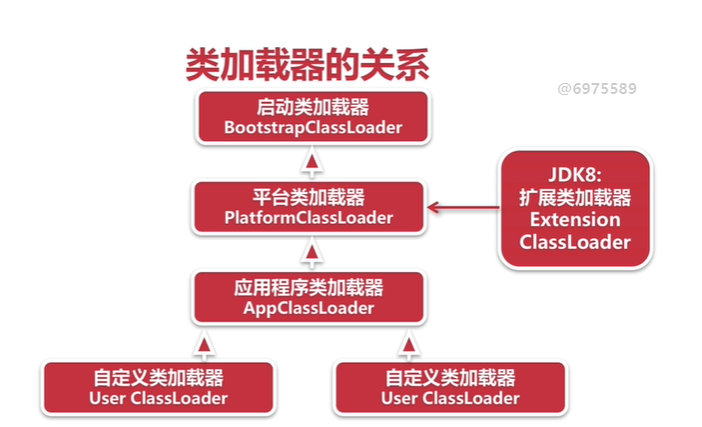
应用程序类加载器（AppClassLoader）

java9之后为了支持“模块化”将扩展类加载器改为平台类加载器（PlatformClassLoader）

用户自定义的加载器，是java.lang.ClassLoader的子类

用户可以定制类的加载方式；只不过自定义类加载器是在所有系统类加载器的最后

### 类加载器的关系



\*启动类加载器：用于加载启动的基础模块类，比如：java.base，java.management,java.xml等等

\*平台类加载器：用于加载一些平台相关的模块，比如：java.scripting,java.compiler\*,java.corba\*等等

\*应用程序加载器：用于加载应用级别的模块，比如：jdk.compiler,jdk.jartool,jdk.jshell等等；还加载classpath路径中的所有类库

\*JDK8：启动类加载器：负责将<JAVA\_HOME>/lib，或者Xbootclasspath参数指定的路径中的，且是虚拟机识别的类库加载到内存中（按照名字识别，比如rt.jar，对于不能识别的文件不予装载）

\*JDK8：扩展类加载器：负责加载<JRE\_HOME>/lib/ext,或java.ext.dirs系统变量所指定路径中的所有类库

\*JDK8：应用程序类加载器：负责加载classpath路径中的所有类库

\*java程序不能直接引用启动类加载器，直接设置classloader为null，默认就使用启动类加载器

\*类加载器并不需要等到某个类“首次主动使用”的时候才加载它，jvm规范允许类加载器在预料到某个类将要被使用的时候就预先加载它

\*如果在加载的时候.class文件缺失，会在改类首次主动使用的时候报告linkageError错误，如果一直没有被使用，就不会报错

## 双亲委派模型

### 双亲委派模型的原理

双亲委派模式是在Java 1.2后引入的，其工作原理的是，如果一个类加载器收到了类加载请求，它并不会自己先去加载，而是把这个请求委托给父类的加载器去执行，如果父类加载器还存在其父类加载器，则进一步向上委托，依次递归，请求最终将到达顶层的启动类加载器，如果父类加载器可以完成类加载任务，就成功返回，倘若父类加载器无法完成此加载任务，子加载器才会尝试自己去加载，这就是双亲委派模式

### 双亲委派模型的优势

· 采用双亲委派模式的好处是Java类随着它的类加载器一起具备了一种带有优先级的层次关系，通过这种层级关可以避免类的重复加载，当父亲已经加载了该类时，就没有必要子ClassLoader再加载一次。

· 其次是考虑到安全因素，java核心api中定义类型不会被随意替换，假设通过网络传递一个名为java.lang.Integer的类，通过双亲委托模式传递到启动类加载器，而启动类加载器在核心Java API发现这个名字的类，发现该类已被加载，并不会重新加载网络传递的过来的java.lang.Integer，而直接返回已加载过的Integer.class，这样便可以防止核心API库被随意篡改。

· 可能你会想，如果我们在classpath路径下自定义一个名为java.lang.SingleInterge类(该类是胡编的)呢？该类并不存在java.lang中，经过双亲委托模式，传递到启动类加载器中，由于父类加载器路径下并没有该类，所以不会加载，将反向委托给子类加载器加载，最终会通过系统类加载器加载该类。但是这样做是不允许，因为java.lang是核心API包，需要访问权限，强制加载将会报出异常

### 双亲委派模型说明

1 双亲委派模型对于保证java程序的稳定运作很重要

2 实现双亲委派的代码在java.lang.ClassLoader的loadClass()方法中，如果自定义类加载器的话，推荐覆盖实现findClass()

3 如果有一个类加载器能加载某个类，称为定义类加载器，所有能成功返回该类的Classs的类加载器都被称为初始类加载器

4 如果没有指定父加载器，默认就是启动加载器

5 每个类加载器都有自己的命名空间，命名空间由该加载器及其所有父加载器所加载的类构成，不同的命名空间，可以出现类的全路径名相同的情况

6 运行时包由同一个类加载器的类构成，决定两个类是否属于同一个运行时包，不仅要看全路径名是否一样，还要看定义类加载器是否相同。只有属于同一个运行时包的类才能实现相互包内可见

### 破坏双亲委派模型

\* 双亲模型有个问题：父加载器无法向下识别子加载器加载的资源

\* 为了解决这个问题，引入了线程上下文类加载器，可以通过Thread的setContextClassLoader（）进行设置

\* 另外一种典型情况就是实现热替换，比如OSGI的模块化热部署，它的类加载器就不再是严格按照双亲委派模型，很多可能就在平级的类加载器中执行了

## 类连接和初始化

### 类验证的主要验证的内容

\* 类文件结构检查：按照jvm规范规定的类文件结构进行

\* 元数据验证： 对字节码描述的信息进行语义分析，保证其符合java语言规范要求

\* 字节码验证： 通过对数据流和控制流进行分析，确保程序语义是合法和符合逻辑的。这里主要对方法进行校验

\* 符号引用验证：对类自身以外的信息，也就是常量池中的各种符号引用，进行匹配校验

### 类连接中的解析

\* 所谓解析就是把常量池中的符号引用转换成直接引用的过程，包括:符号引用：以一组无歧义的符号来描述锁引用的目标，与虚拟机的实现无关

\* 直接引用：直接指向目标的指针，相对偏移量，或是能间接定位到目标的句柄，是和虚拟机实现相关的

\* 主要针对：类，接口，字段，类方法，接口方法，方法类型，方法句柄，调用点限定符

### 类的初始化

\* 类的初始化就是为类的静态变量赋初始值，或者说是执行类构造器<clinit>方法的过程

1 如果类还没有加载和连接，就先加载和连接

2 如果类存在父类，且父类没有初始化，就先初始化父类

3 如果类中存在初始化语句，就依次执行这些初始化语句

4 如果是接口的话：

1. 初始化一个类的时候，并不会先初始化它实现的接口
2. 初始化一个接口时，并不会初始化它的父接口
3. 只有程序首次使用接口里面的变量或者是调用接口方法的时候，才会导致接口初始化

5 调用Classloader类的loadClass方法来装载一个类，并不会初始化这个类，不是对类的主动使用

### 类的初始化时机

\* java 程序对类的使用方式分成：主动使用和别动使用，JVM必须在每个类或接口“首次主动使用”时才初始化他们；被动使用类不会导致类的初始化，主动使用的情况：

1 创建类实例

2 访问某个类或接口的静态变量

3 调用类的静态方法

4 反射某个类

5 初始化某个类的子类，而父类还没有初始化

6 JMV启动的时候运行的主类

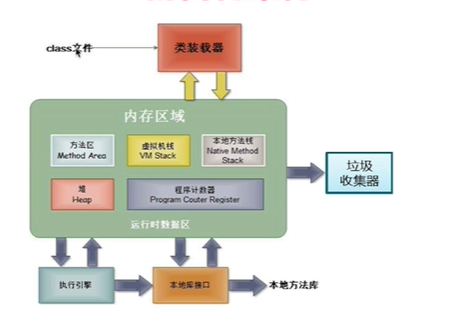
7 定义了default方法的接口，当接口实现类初始化时

### 类的卸载

\* 当代表一个类的Class对象不再被引用，那么Class对象的生命周期就结束了，对应的在方法区中的数据也会被卸载

\* JVM自带的类加载器装载的类，是不会卸载的，由用户自定义的类加载器加载的类是可以卸载的

# JVM内存分配



## JVM的简化架构和运行时数据区

### 运行时数据区

包括：PC寄存器，Java虚拟机栈，Java堆，方法区，运行时常量池，本地方法栈等

\* PC(Program Counter)寄存器说明：

1 每个线程拥有一个PC寄存器，是线程私有的，用来存储指向下一条指令的地址

2 在创建线程的时候，创建响应的PC寄存器

3 执行本地方法是，PC寄存器的值为undefined

4 是一块较小的内存空间，是唯一一个在JVM规范中没有规定OutOfMemoryError的内存区域（不会内存溢出）

\* Java栈

1 栈是由一系列帧组成（因此Java栈也叫做帧栈），是线程私有的

2 帧用来保存一个方法的局部变量，操作数栈（java没有寄存器，所有参数传递使用操作数栈），常量池指针，动态链接，方法返回值等

3 每一次方法调用创建一个帧，并压栈，退出方法的时候，修改栈顶指针就可以把栈帧中的内容销毁

4 局部变量表存放了编译期可知的各种基本数据类型和引用类型，每个slot存放32位的数据，long,double占两个槽位

5 栈的优点：存取速度比堆快，仅次于寄存器

6 栈的缺点：存在栈中的数据大小，生存期是编译期决定的缺乏灵活性

\*Java 堆

1 用来存放应用系统创建的对象和数组，所有线程共享java堆

2 GC主要就管理堆空间，对分代GC来说，堆也是分代的

3 堆的优点：运行期动态分配内存大小，自动进行垃圾回收；

堆的缺点：效率相对较慢

\* 方法区

1 方法区是线程共享的，通常用来保存装载的类的结构信息

2 通常和元空间关联在一起，但具体的跟JVM实现和版本有关

3 JVM规范把方法区描述为堆的一个逻辑部分，但它有个别名为Non-heap（非堆），应是为了与java堆区分开

\* 运行时常量池

1 是class文件中每个类或接口的常量池表，在运行期间的表示形式，通常包括：类的版本，字段，方法，接口等信息

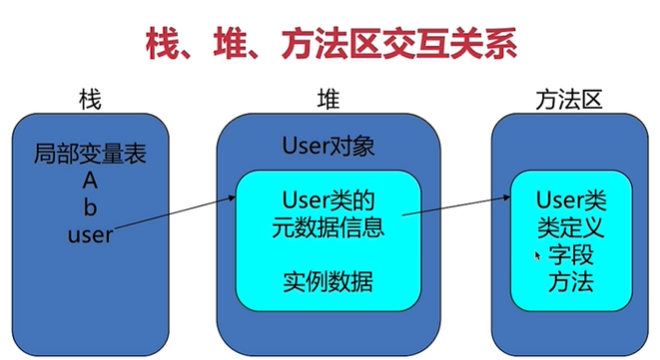
2 在方法区中分配

3 通常在加载类和接口到JVM后，就创建响应的运行时常量池

\* 本地方法栈

1 在JVM中用来支持native方法执行的栈就是本地方法栈

### 栈 堆 方法去交互关系



## Java堆内存模型和分配

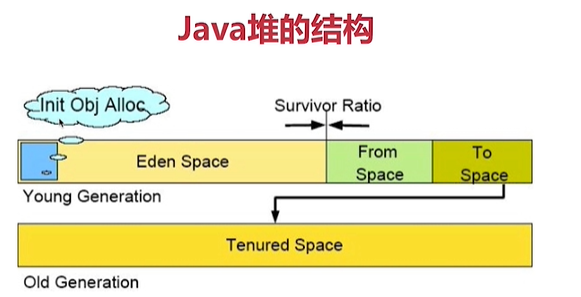
### java堆内存概述

1 java堆用来存放应用系统创建的对象和数组，所有线程共享java堆

2 java堆是在运行期动态分配内存大小，自动进行垃圾回收

3 java垃圾回收（GC）主要就是回收堆内存，对分代GC来说，堆也是分代的

### java堆的结构



\* 新生代用来放新分配的对象；新生代中经过垃圾回收，没有回收掉的对象，被复制到老年代

\* 老年代存储对象比新生代存储对象的年龄大得多

\* 老年代存储一些大对象

\* 整个堆大小 = 新生代+老年代

\* 新生代= eden+存活区

\* 从前的持久代，用来存放Class，Method等元信息的区域，从JDK8开始去掉了，取而代之的是元空间，元空间并不在虚拟机里面，而是直接使用本地内存

### 对象的内存布局

\* 对象的内存中存储的布局（这里毅Hotspot虚拟机为例来说明），分为：对象头，实例数据和对齐填充

\* 对象头包含两部分：

1 Mark Word：存储对象自身的运行数据，如：HashCode,GC分代年龄，锁状态标志等

2 类型指针：对象指向它的类元数据的指针

\* 实例数据：真正存放对象示例数据的地方

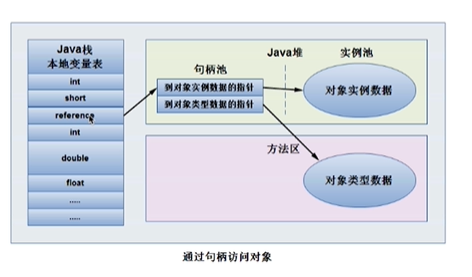
\* 对齐填充：这部分不一定存在，也没有什么特别的含义，仅仅是占位符，因为HotSpot要求对象起始地址都是8字节的整数倍，如果不是，就对齐

### 对象的访问定位

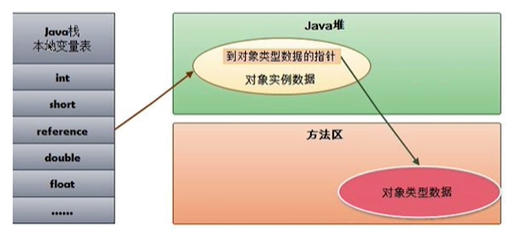
\* 对象的访问定位:在jvm规范中只规定了reference类型是一个指向对象的引用，但没有规定这个引用具体如何去定位，访问堆中对象的具体位置

\* 因此对象的访问方式取决于JVM的实现，目前主流的有：使用句柄或使用指针两种方式

\* 使用句柄：java堆中会划分出一块内存来作为句柄池，reference中存储句柄的地址句柄中存储对象的实例数据和类元数据的地址，如下图所示：



\* 使用指针：java堆中会存放访问类元数据的地址，reference存储的就直接是对象的地址，如下图所示



\* 使用指针的方式比使用句柄的方式要更快

## Trace跟踪和Java堆的参数配置

### Trace跟踪参数

\* 可以打印GC的简要信息： -Xlog:gc

\* 打印GC详细信息： -Xlog:gc\*

\* 指定GC log的位置，以文件输出 -Xlog:gc:garbage-collection.log

\* 每一次GC后，都打印堆信息：-Xlog:gc+heap=debug

### GC日志格式

\* GC发生的时间，也就是JVM从启动以来经过的秒数

\* 日志级别信息和日志类型标记

\* GC识别号

\* GC类型和说明GC的原因

\* 容量：GC前容量->GC后容量（该区域总容量）

\* GC持续时间，单位秒。有的收集器会有更详细的描述，比如：user表示应用程序消耗的时间，sys表示系统内核消耗的时间，real表示操作从开始到结束的时间

## Java堆参数

\* Xms：初始堆大小，默认物理内存的1/64

\* Xmx: 最大堆大小，默认为物理内存的1/4

\* Xmn: 新生代大小，默认整个堆的3/8

\* -XX:+HeapDumpOnOutOfMemoryError:OOM时导出堆到文件

\* -XX:+HeapDumpPath:导出OOM的路径

\* -XX:OnOutOfMemoryError:在OOM时，执行一个脚本

\*-XX:NewRatio:老年代和新生代的比值

如果xms-xmx，且设置了xmn的情况下，该参数不用设置

\*-XX:SurvivorRatio:Eden区和Survivor区的大小比值，设置为8，则两个Survivor去与一个Eden区的比值为2:8，一个Survivor占整个新生的1/10

## Java栈的参数

\*-Xss:通常只有几百k，决定了函数调用的深度

## 元空间的参数

-XX:MetaspaceSize:初始空间大小

-XX:MaxMetaspaceSize:最大空间，默认是没有限制的

-XX:MinMetaspaceFreeRatio:在GC之后，最小的Metaspace剩余空间容量的百分比

-XX:MaxMetaspaceFreeRatio:在GC之后，最大的Metaspace剩余空间容量的百分比

# 字节码执行引擎

## 字节码执行引擎概述

\* JVM的字节码执行引擎，功能基本就是输入字节码文件，然后对字节码进行解析并处理，最后输入执行的结果。

\*实现方式可能有通过解释器直接解释执行字节码，或者是通过即时编译器产生本地代码，也就是编译执行，当然也可能两者皆有

## 栈帧概述

\*栈帧是用于支持JVM进行方法调用和方法执行的数据结构

\*栈帧随着方法调用而创建，随着方法结束而销毁

\*栈帧里面存储了方法的局部变量，操作数栈，动态连接，方法返回地址等信息

#### 栈帧的概念结构



#### 局部变量表

\* 局部变量表：用来存放方法参数和方法内部定义的局部变量的存储空间

1 以变量槽slot为单位，目前一个slot存放32位以内的数据类型

2 对于64位的数据栈两个slot

3 对于实例方法，第0位slot存放的是this，然后从1到n，依次分配给参数列表

4 根据方法体内部定义的变量顺序和作用域来分配slot  
· 5 slot是复用的，以节省栈帧的空间，这种设计可能会影响到系统的垃圾收集行为

#### 操作数栈

\*操作数栈：用来存放方法运行期间，各个指令操作的数据

1 操作数栈中的元素的数据类型必须和字节码指令的顺序严格匹配

2 虚拟机在实现栈帧的时候可能会做一些优化，让两个栈帧出现部分重叠区域，以存放公用的数据

#### 动态连接

\* 动态连接：每个栈帧持有一个指向运行时常量池中该栈帧所属方法的引用，以支持方法调用过程的动态连接

1 静态解析：类加载的时候，符号引用就转化成直接引用

2 动态连接：运行期间转换为直接引用

#### 方法返回地址

\* 方法返回地址：方法执行后返回的地址

#### 方法调用

\* 方法调用：方法调用就是确定具体调用哪一个方法，并不涉及方法内部的执行过程

1 部分方法是直接在类加载的解析阶段，就确定了直接引用关系

2 但是对于实例方法，也成虚方法，因为重载和多态，需要运行期动态委派

#### 分派

\*分派：又分为静态分派和动态分派

1 静态分派：所有依赖静态类型来定位方法执行版本的分派方式。比如：重载方法

2 动态分派：根据运行期的实际类型来定位方法执行版本的分派方式，比如：覆盖方法

\*单分派和多分派：就是按照分派思考的维度，多余一个的就算多分派，只有一个的成为单分派

\*如何执行方法中的字节码指令：JVM通过基于栈的字节码解释执行引擎来执行指令，JVM的指令集也是基于栈的