JVM

需要解答的问题

- [x] ~~常量在类加载和实例化的过程中的变化~~
- [x] ~~不同的变量的初始化历程~~
- [x] ~~是编译一个，执行一个，用到了才编译，还是全部编译好后，在执行~~

模块讲解

方法区

> 当JVM使用类装载器定位class文件，并将其输入到内存中时。会 提取class文件的类型信息，并将这些信息存储到方法区中。同时放入方法区中的还有该类型中的 类静态变量 。

该类型的 全限定名 。如`java.io.FileOutputStream`

该类型的直接 超类的全限定名 。如`java.io.OutputStream`

该类型是 类类型 还是 接口类型 。

该类型的访问 修饰符(public、abstract、final)。

任何 直接超接口的全限定名 的有序列表。如`java.io.Closeable, java.io.Flushable`。

该类型的 常量池 。比如所有类型（Class）、方法、字段的符号、基本数据类型的直接数值（final）等。

字段信息： 对类型中声明的每个字段。

方法信息 。

类静态变量： 静态变量而不是放在堆里面，所以静态属于类，不属于对象。

指向ClassLoader类的引用。

指向Class类的引用。

方法表： 为了能快速定位到类型中的某个方法，JVM对每个装载的类型都会建立一个方法表，用于存储该类型对象可以调用的方法的直接引用，这些方法就包括从超类中继承来的。而这张表与Java动态绑定机制的实现是密切相关的。

常量池

> 常量池指的是在 编译期被确定，并被保存在已编译的.class文件中的一些数据。

除了包含代码中所定义的各种 基本数据类型 和 对象型（String及数组） 的常量值（final，在编译时确定，并且编译器会优化）

还包含一些以文本形式出现的符号引用（类信息），比如：

类和接口的全限定名

字段的名称和描述符

方法和名称和描述符

方法区与常量池

- 常量池 存在于方法区
- 虚拟机必须给每个被装载的 类 维护一个常量池。常量池就是该类型所用到常量的一个有序集合，包括直接常量（string、integer等）和其他类型，字段和方法的 符号引用

- 方法区是**多线程共享**的。也就是当虚拟机实例开始运行程序时，边运行边加载进class文件。不同的Class文件都会提取出不同类型信息存放在方法区中。同样，方法区中不再需要运行的类型信息会被垃圾回收线程丢弃掉

堆内存

> Java 程序在运行时创建的 所有类型对象和数组 都存储在堆中。JVM会根据new指令在堆中开辟一个确定类型的对象内存空间。但是堆中开辟对象的空间并没有任何人工指令可以回收，而是通过JVM的 垃圾回收器负责回收 。

堆中对象存储的是该对象以及对象所有超类的实例数据(但不是静态数据)。

其中一个 对象的引用可能在整个运行时数据区中的很多地方存在， 比如Java栈，堆，方法区等。

堆中对象还应该 关联一个对象的锁数据信息以及线程的等待集合（线程等待池） 。这些都是实现Java线程同步机制的基础。

java中数组也是对象，那么自然在堆中会存储数组的信息。

程序计数器（PC）

> 对于一个运行的Java而言，每一个线程都有一个PC寄存器。当线程执行Java程序时，PC寄存器的内容总是下一条将被执行的指令地址。
> 如果当前方法是 native 方法，那么PC 的值为 undefined

Java栈

> 每启动一个线程，JVM都会为它分配一个Java栈，用于存放方法中的 局部变量，操作数以及异常数据 等。当线程调用某个方法时，JVM会根据方法区中该方法的字节码组建一个栈帧。并将该栈帧压入Java栈中，方法执行完毕时，JVM会弹出该栈帧并释放掉。

注意：Java栈中的数据是线程私有的，一个线程是无法访问另一个线程的Java栈的数据。这也就是为什么多线程编程时，两个相同线程执行同一方法时，对方法内的局部变量是不需要数据同步的原因。

成员变量有默认值（被final修饰且没有static的必须显式赋值）， 局部变量不会自动赋值 。

每当启用一个线程时，JVM就为他分配一个Java栈，栈是以帧为单位保存当前线程的运行状态。某个线程正在执行的方法称为当前方法，当前方法　　使用的栈帧称为当前帧，当前方法所属的类称为当前类，当前类的常量池称为当前常量池。 当线程执行一个方法时，它会跟踪当前常量池 。

每当线程调用一个Java方法时，JVM就会在该线程对应的栈中压入一个帧，这个帧自然就成了当前帧。当执行这个方法时，它使用这个帧来存储参　数、局部变量、中间运算结果等等。（ 一帧一方法 ）

Java栈上的所有数据都是私有的。任何线程都不能访问另一个线程的栈数据。所以我们不用考虑多线程情况下栈数据访问同步的情况。

像方法区和堆一样， Java栈和帧在内存中也不必是连续的,帧可以分布在连续的栈里， 也可以分布在堆里

Java栈的组成元素——栈帧

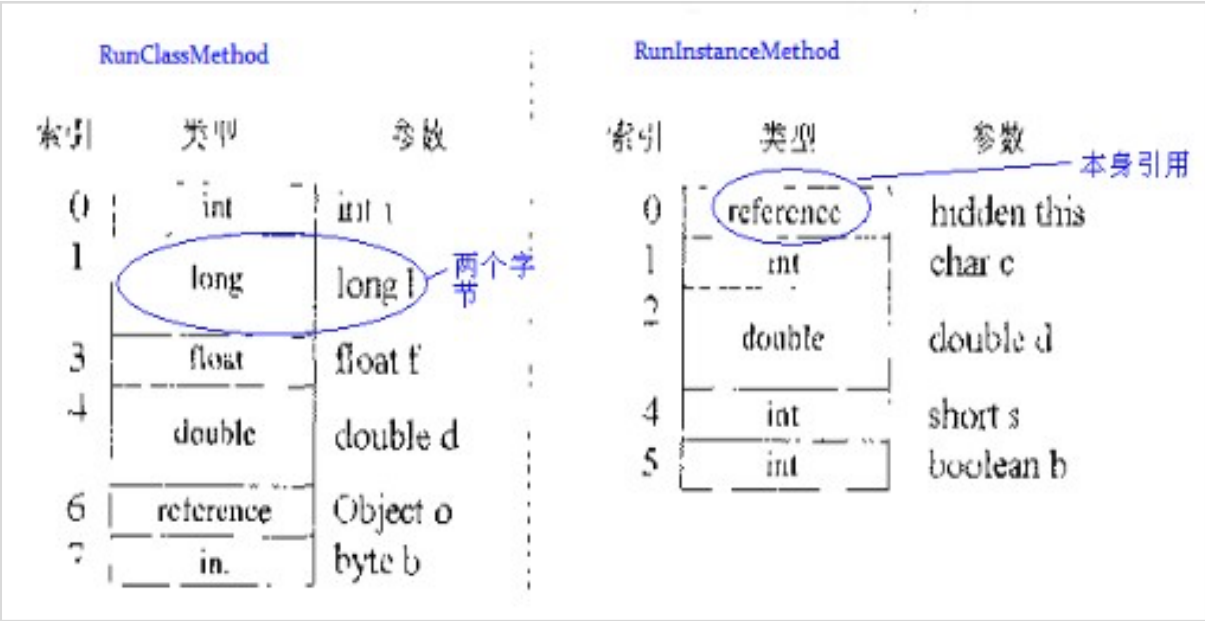
栈帧由三部分组成： 局部变量区、操作数栈、帧数据区 。局部变量区和操作数栈的大小要视对应的方法而定，他们是 按字长计算 的。但调用 一个方法时，它从 类型信息中得到此

方法局部变量区和操作数栈大小，并据此分配栈内存，然后压入Java栈 。

局部变量区 局部变量区被组织为以一个字长为单位、从0开始计数的数组，类型为short、byte和char的值在存入数组前要被转换成int值，而long和 double在数组中占据连续的两项，在访问局部变量中的long或double时，只需取出连续两项的第一项的索引值即可,如某个long值在局部变量区中 占据的索引时3、4项，取值时，指令只需取索引为3的long值即可。

下面就看个例子，好让大家对局部变量区有更深刻的认识。这个图来自《深入JVM》：

```
public static int runClassMethod(int i,long l,float f,double d,Object o,byte b)
{
        return 0;
}
public int runInstanceMethod(char c,double d,short s,boolean b) {
        return 0;
}
```



上面这个图没什么好说的，大家看看就会懂。但是，在这个图里，有一点需要注意：

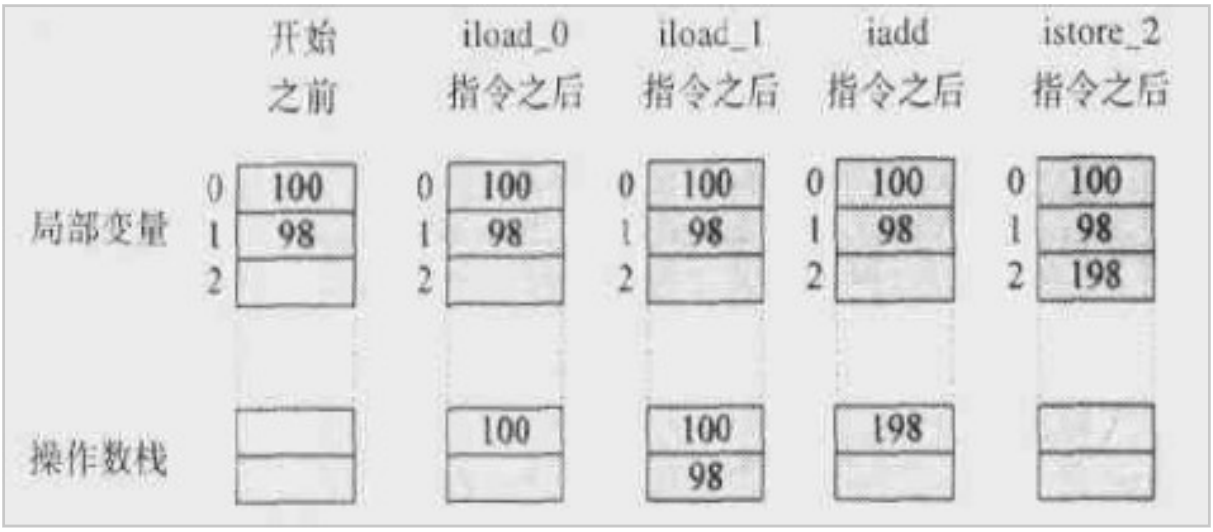
runInstanceMethod的局部变量区 第一项是个reference （this引用），它指定的就是对象本身的引用，也就是我们常用的this,但是在 runClassMethod方法中，没这个引用，那是因为runClassMethod是个静态方法。

操作数栈和局部变量区一样，操作数栈也被组织成一个以字长为单位的数组。但和前者不同的是，它不是通过索引来访问的，而是通过入栈和出栈 来访问的。可把操作数栈理解为存储计算时，临时数据的存储区域。下面我们通过一段简短的程序片段外加一幅图片来了解下操作数栈的作用。

int a = 100;

int b = 98;

int c = a+b;



从图中可以得出：操作数栈其实就是个 临时数据存储区域， 它是通过入栈和出栈来进行操作的。

帧数据区除了局部变量区和操作数栈外，Java栈帧还需要一些数据来支持常量池解析、正常方法返回以及异常派发机制。这些数据都保存在Java 栈帧的帧数据区中。

当JVM执行到需要常量池数据的指令时，它都会通过 帧数据区中指向常量池的指针 来访问它。

除了处理常量池解析外，帧里的数据还要处理Java方法的正常结束和异常终止。 如果是通过return正常结束，则当前栈帧从Java栈中弹出，恢复 发起调用的方法的栈。如果方法又返回值，JVM会把返回值压入到发起调用方法的操作数栈。

为了处理Java方法中的异常情况，帧数据区还必须保存一个对此方法异常引用表的引用。当异常抛出时，JVM给catch块中的代码 。如果没发 现，方法立即终止，然后JVM用帧区数据的信息恢复发起调用的方法的帧 。然后再发起调用方法的上下文重新抛出同样的异常。

动态链接

每个栈帧都有一个 运行时常量池的引用 。 这个引用指向栈帧当前运行方法所在类的常量池 。通过这个引用支持动态链接（dynamic linking）。

在 Java中， 链接(连接)阶段是运行时动态完成 的。

当 Java 类文件编译时，所有变量和方法的引用都被当做符号引用存储在这个类的常量池中。 符号引用是一个逻辑引用，实际上并不指向物理内 存地址。 JVM 可以选择符号引用解析的时机， 一种是当类文件加载并校验通过后， 这种解析方式被称为饥饿方式。 另外一种是符号引用在 第一次使用的时候被解析， 这种解析方式称为惰性方式。无论如何， JVM 必须要在第一次使用符号引用时完成解析并抛出可能发生的解析错 误。绑定是将对象域、方法、类的符号引用替换为直接引用的过程。 绑定只会发生一次。一旦绑定，符号引用会被完全替换 。如果一个类的符 号引用还没有被解析，那么就会载入这个类。每个直接引用都被存储为相对于存储结构（与运行时变量或方法的位置相关联的）偏移量。

栈的整个结构

在前面就描述过：栈是由栈帧组成，每当线程调用一个Java方法时，JVM就会在该线程对应的栈中压入一个帧，而帧是由局部变量区、操作数栈和　帧数据区组成。那在一个代码块中，栈到底是什么形式呢？下面是我从《深入JVM》中摘抄的一个例子，大家可以看看：

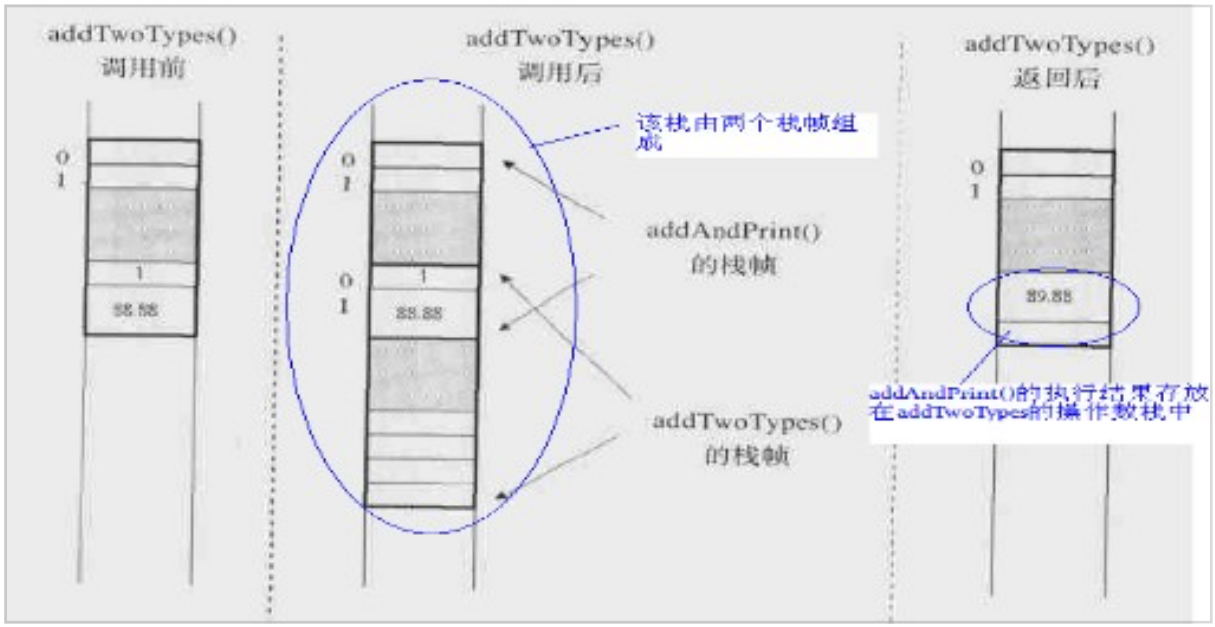
```
class Example3c {

    public static void addAndPrint() {
        double result = addTwoTypes(1, 88.88);
        System.out.println(result);
    }

    public static double addTwoTypes(int i, double d) {
        return i + d;
    }
}
```

调用方法

执行过程中的三个快照：



上面所给的图，只想说明两件事情，我们也可用此来理解Java中的栈：

1、只有在调用一个方法时，才为当前栈分配一个帧，然后将该帧压入栈。

2、帧中存储了对应方法的局部数据，方法执行完，对应的帧则从栈中弹出，并把返回结果存储在调用方法的帧的操作数栈中。

Native栈

并非所有的 JVM 实现都支持本地（native）方法（不是Native栈哦）

场景：当对一些底层的如操作系统或某些硬件交换信息时，我们使用java来编程实现起来不容易，再者使用java来编程效率也很低下。这就不得不　　需要调用本地方法来解决这一

问题。

其不受JVM控制， 都不是Java编写的， 可能是C语言。

本地方法栈则是为JVM使用到的Native方法服务

栈（Java栈）与Native栈的区别

区别不过是栈为虚拟机执行Java方法（也就是字节码）服务，而本地方法栈则是为JVM使用到的Native方法服务。

JVM栈与局部变量

直接引用于符号引用

对于指向"类型" 【Class对象】、类变量、类方法 的直接引用可能是指向方法区的 本地指针 。指向 实例变量、实例方法 的直接引用都是 偏移量 。实例变量的直接引用可能是从对象的映像开始算起到这个实例变量位置的偏移量。实例方法的直接引用可能是方法表的偏移量。

子类中方法表的偏移量和父类中的方法表的偏移量是一致的 。比如说父类中有一个say()方法的偏移量是7，那么子类中say方法的偏移量也是7。（动态引用）

而通过"类引用"来调用一个方法的时候，直接通过偏移量就可以找到要调用的方法的位置了。

【 因为子类中的方法的偏移量跟父类中的偏移量是一致的 】

所以，通过接口引用调用方法会比类引用慢一些。

下面介绍下什么是接口引用。

```
interface A{void say();}
class B implements A{}
class C{public static void main(String []s){A a=new B();a.say()}}
```

在上面的第三行代码中，就是用"接口引用"来调用方法。

————

符号引用是一个 字符串， 它给出了被引用的内容的名字并且可能会包含一些其他关于这个被引用项的信息– –这些信息必须足以唯一的识别一个类、字段、方法。这样，对于其他类的符号引用必须给出类的全名 。对于其他类的字段，必须给出类名、字段名以及字段描述符。对于其他类的方法的引用必须给出类名、方法名以及方法的描述符

~~符号引用保存于常量池，直接引用在方法区~~

执行引擎

运行Java的每一个线程都是一个独立的虚拟机执行引擎的实例。从线程生命周期的开始到结束，他 要么在执行字节码，要么在执行本地方法 。一个线程可能通过解释或者使用芯片级指令直接执行字节码，或者间接通过JIT（即时编译器）执行编译过的本地代码。

注意：JVM是进程级别，执行引擎是线程级别。

指令集

实际上，class文件中方法的字节码流就是有JVM的指令序列构成的。 每一条指令包含一个单字节的操作码，后面跟随0个或多个操作数 。

指令由 一个操作码和零个或多个操作数 组成。

```
iload_0    // 把存储在局部变量区中索引为0的整数压入操作数栈。
iload_1    // 把存储在局部变量区中索引为1的整数压入操作数栈。
iadd         // 从操作数栈中弹出两个整数相加，在将结果压入操作数栈。
istore_2   // 从操作数栈中弹出结果
```

很显然，上面的指令反复用到了Java栈中的某一个方法栈帧。实际上执行引擎运行Java字节码指令很多时候都是在不停的操作Java栈，也有的时候需要在堆中开辟对象以及运行系统的本地指令等。但是Java栈的操作要比堆中的操作要快的多，因此反复开辟对象是非常耗时的。这也是为什么Java程序优化的时候，尽量减少new对象。

编译

> 编译：源码要运行，必须先 转成二进制 的机器码。这是编译器的任务。

源文件由编译器编译成字节码。 创建完源文件之后，程序会先被编译为.class文件。Java编译一个类时，如果这个类所依赖的类还没有被编译，编译器就会先编译这个被依赖的类，然后引用，否则直接引用。如果java编译器在指定目录下找不到该类所其依赖的类的.class文件或者.java源文件的话，编译器话报“cant find symbol”的错误。

编译后的字节码文件格式主要分为两部分： 常量池和方法字节码 。常量池记录的是代码出现过的所有token(类名，成员变量名等等)以及符号引用（方法引用，成员变量引用等等）；方法字节码放的是类中各个方法的字节码。

这个过程就是静态绑定

————

将.java文件转为.class二进制文件

在java编译好的class文件中,有个区域称为Constant Pool,他是一个由数组组成的表,类型为cp_info constant_pool[],用来存储程序中使用的各种常量,包括Class*String*Integer等各种基本Java数据类型，所以编译阶段不是直接存储到方法区下面的常量池中

将class文件中的类名（类和接口的全限定名）、成员变量名和方法、成员变量等名称的 符号引用 存放在Constant Pool

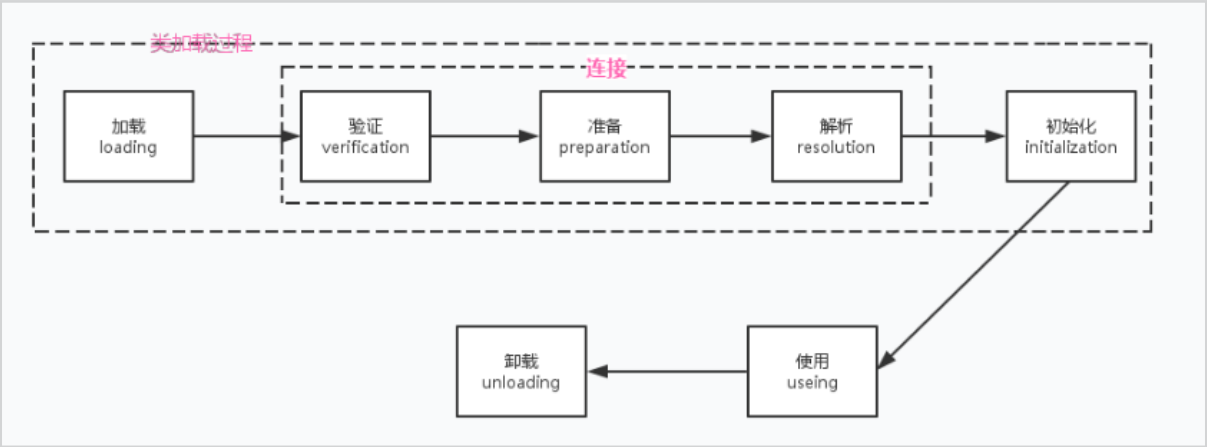
对final修饰的static常量进行Constant Pool指定，即使是在静态代码块中进行初始化赋值的，在编译之后，其实是一句话

类加载

JVM主要包含三大核心部分： 类加载器，运行时数据区和执行引擎 。

> 虚拟机将描述类的数据从class文件加载到内存，并对数据进行 校验，准备，解析和初始化 ，最终就会形成可以被虚拟机使用的java类型，这就是一个虚拟机的类加载机制。java在类中的类是动态加载的，只有在运行期间使用到该类的时候，才会将该类加载到内存中，java依赖于运行期动态加载和动态链接来实现类的动态使用。

一个类的生命周期：



加载，验证，准备，初始化和卸载 在开始的顺序上是固定的，但是可以交叉进行 。

在Java中，对于类有且仅有四种情况会对类进行“初始化”。

使用new关键字实例化对象的时候，读取或设置一个类的静态字段时候（除final修饰的static外），类只有在第一次使用的时候才会被初始化

使用reflect包对类进行反射调用的时候，如果类没有进行初始化，则先要初始化该类。

当初始化一个类的时候，如果其父类没有初始化过，则先要触发其父类初始化。

虚拟机启动的时候，会初始化一个有main方法的主类。

注意：

子类引用父类静态字段，只会初始化父类不会初始化子类

通过数组定义来引用类，也不会触发该类的初始化

常量在编译阶段会存入调用类的常量池中， 本质上没有直接引用到定义常量的类，因此也不会触发定义常量的类的初始化

被final修饰的static静态字段，必须要在类加载的时候默认值（可以是直接赋值，也可以在静态代码块中初始值）

加载

> 加载阶段主要完成三件事，即通过一个类的全限定名来获取定义此类的二进制字节流， 将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构，在Java堆中生成一个代表此类的Class对象，作为访问方法区这些数据的入口。 这个加载过程主要就是靠类加载器实现的，这个过程可以由用户自定义类的加载过程。

验证

> 这个阶段目的在于确保才class文件的字节流中包含信息符合当前虚拟机要求，不会危害虚拟机自身安全。

主要包括四种验证：

文件格式验证：基于字节流验证，验证字节流是否符合Class文件格式的规范，并且能被当前虚拟机处理。

元数据验证：基于方法区的存储结构验证，对字节码描述信息进行语义验证。

字节码验证：基于方法区的存储结构验证，进行数据流和控制流的验证。

符号引用验证：基于方法区的存储结构验证，发生在解析中，是否可以将符号引用成功解析为直接引用。

准备

仅仅为类变量（即static修饰的字段变量）分配内存并且设置该类变量的初始值即零值，这里不包含用final修饰的static，因为final在编译的时候就会分配了（编译器的优化），同时这里也不会为实例变量分配初始化。 类变量会分配在方法区中，而实例变量是会随着对象一起分配到Java堆中 。

解析

解析主要就是将 常量池中的符号引用替换为直接引用 的过程。符号引用就是一组符号来描述目标，可以是任何字面量，而直接引用就是直接指向目标的指针、相对偏移量或一个间接定位到目标的句柄。有类或接口的解析，字段解析，类方法解析，接口方法解析。

初始化

初始化阶段 依旧是初始化类变量和其他资源， 这里将 执行用户的static字段和静态语句块的赋值操作 。这个过程就是执行类构造器< clinit >方法的过程。

< clinit >方法是由编译器收集类中所有类变量的赋值动作和静态语句块的语句生成的，类构造器< clinit >方法与实例构造器< init >方法不同，这里面不用显示的调用父类的< clinit >方法，父类的< clinit >方法会自动先执行于子类的< clinit >方法 。即父类定义的静态语句块和静态字段都要优先子类的变量赋值操作。

类加载器

类加载器

```
//MainApp.java
public class MainApp {
    public static void main(String[] args) {
        Animal animal = new Animal("Puppy");
        animal.printName();
    }
}
//Animal.java
public class Animal {
    public String name;
    public Animal(String name) {
        this.name = name;
    }
    public void printName() {
        System.out.println("Animal ["+name+"]");
    }
}
```

在编译好java程序得到MainApp.class文件后，在命令行上敲java MainApp。系统就会启动一

个jvm进程，jvm进程从classpath路径中找到一个名为AppMain.class的二进制文件，将MainApp的类信息加载到运行时数据区的方法区内，在堆中生成一个MainApp的class对象，用于类信息获取，这个过程叫做MainApp类的加载

————

然后JVM找到AppMain的主函数入口，开始执行main函数。

main函数的第一条命令是Animal animal = new Animal("Puppy");就是让JVM创建一个Animal对象，但是这时候方法区中没有Animal类的信息，所以JVM马上加载Animal类，把Animal类的类型信息（数据结构）放到方法区中。

校验Animal类

在方法区中，为static变量（除了final）分配初始值

将Animal类的常量池中的符号引用替换为直接引用

如果存在父类，则进行父类的初始化操作

执行Animal中的static字段和静态语句块的赋值操作

————

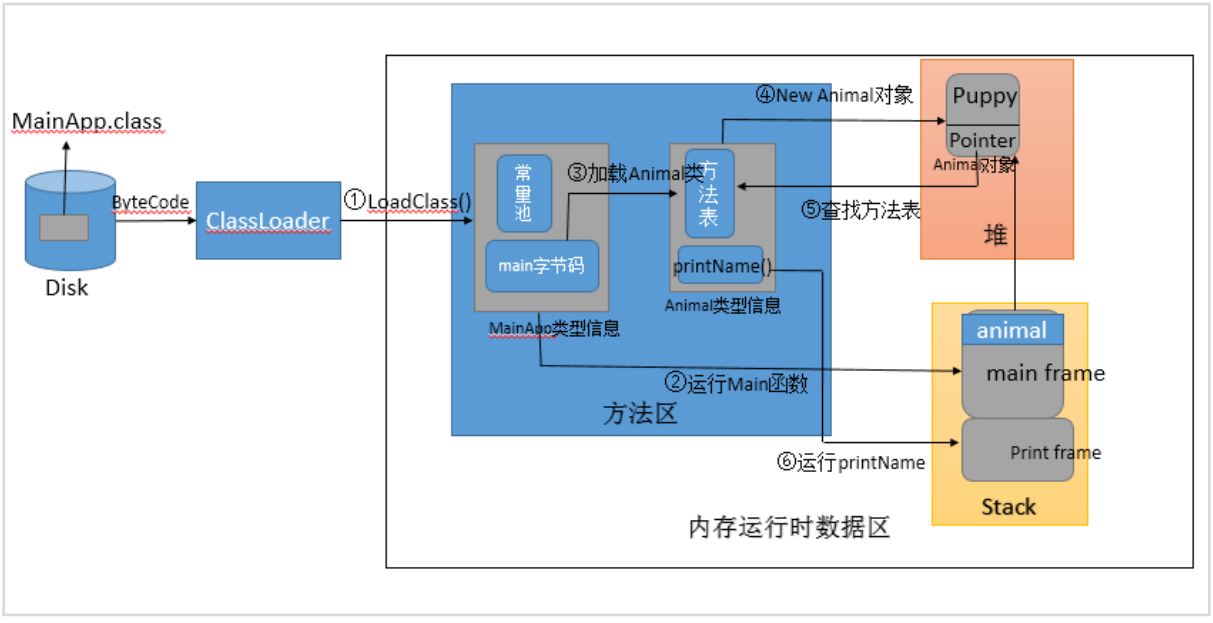
加载完Animal类之后，Java虚拟机做的第一件事情就是在堆区中为一个新的Animal实例分配内存，

如果存在父类，则先调用父类默认构造函数实例化父类，

然后调用构造函数初始化Animal实例，这个Animal实例与堆内存中的Animal的class对象建立联系，从而持有指向方法区的Animal类的类型信息（其中包含有方法表，java动态绑定的底层实现）的引用。

当使用animal.printName()的时候，JVM根据animal引用找到Animal对象，然后根据Animal对象持有的引用定位到方法区中Animal类的类型信息的方法表，获得printName()函数的字节码的地址。

开始运行printName()函数的字节码（可以把字节码理解为一条条的指令）。



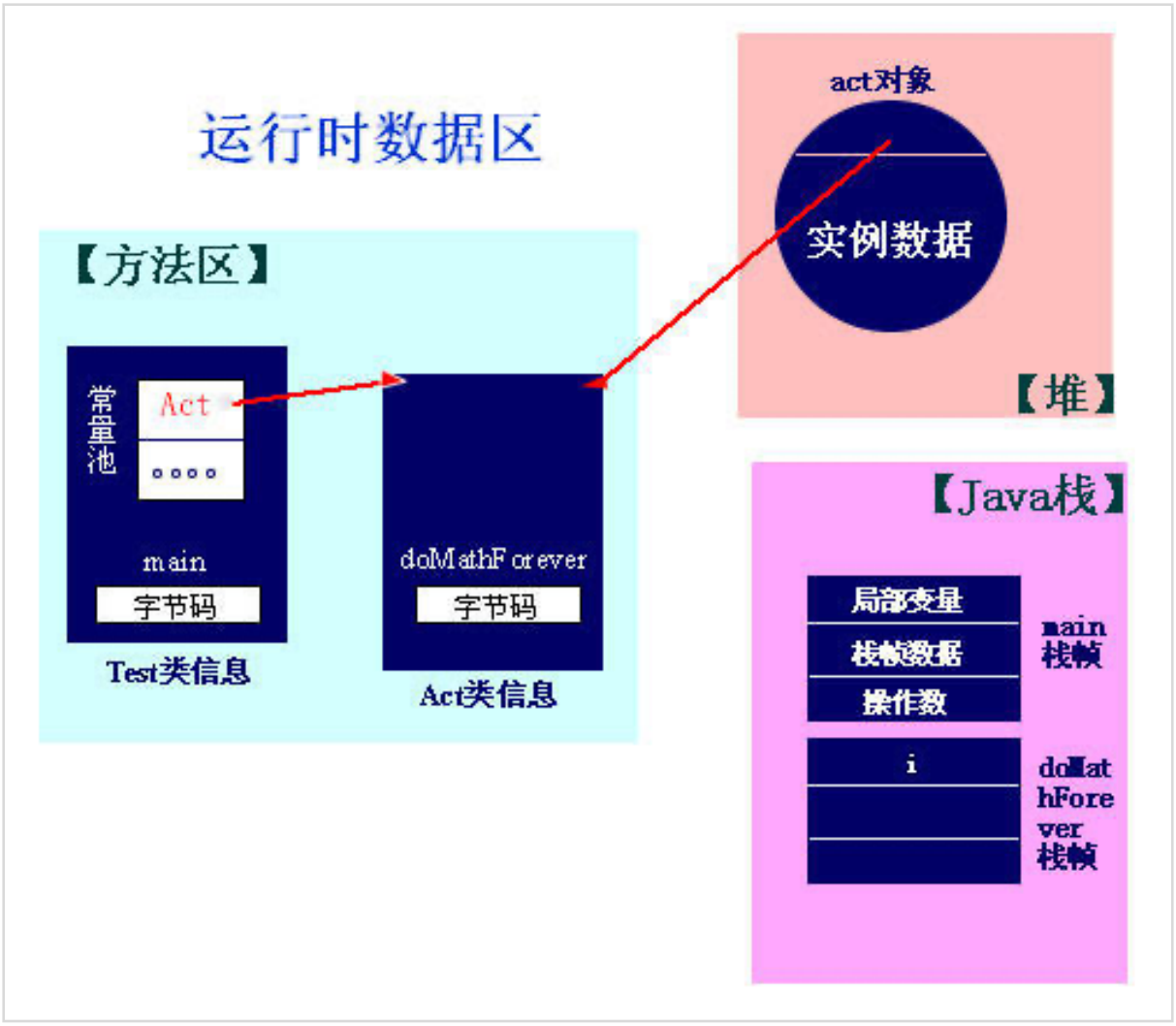
方法执行流程

```
//源代码 Test.java
package edu.hr.jvm;

import edu.hr.jvm.bean;
public class Test{
	public static void main(String[] args){
		 Act act=new Act();
		 act.doMathForever();
	}
}

//源代码 Act.java
package edu.hr.jvm.bean;

public class Act{
	public void doMathForever(){
		int i=0;
		for(;;){
			i+=1;
			i*=2;
		}
	}
}
```



首先OS会创建一个JVM实例(进行必要的初始化工作，比如：初始启动类装载器，初始运行时内存数据区等。

然后通过自定义类装载器加载Test.class。并提取Test.class字节码中的信息存放在方法区 中(具体的信息在上面已经讲过)。上图展示了方法区中的Test类信息，其中在常量池中有一个符号引用"Act"(类的全限定名，注意：这个引用目前还没有真正的类信息的内存地址)。

接着JVM开始从Test类的main字节码处开始解释执行。在运行之前，会在Java栈中组建一个main方法的栈帧， 如上图Java栈所示。JVM需要运行任何方法前，通过在Java栈中压入一个帧栈。在这个帧栈的内存区域中进行计算。

现在可以开始执行main方法的第一条指令 —— JVM需要为常量池的第一项的类（符号引用Act）分配内存空间。但是Act类此时还没有加载进JVM（因为常量池目前只有一个"Act"的符号引用）。

JVM加载进Act.class，并提取Act类信息放入方法区中。然后以一个直接指向方法区Act类信息的直接引用（在栈中）换开始在常量池中的符号引用"Act"，这个过程就是常量池解析 。以后就可以直接访问Act的类信息了。

此时JVM可以根据方法区中的Act类信息，在堆中开辟一个Act类对象act。

接着开始执行main方法中的第二条指令调用doMathForever方法。这个可以通过堆中act对象所指的方法表中查找，然后定位到方法区中的Act类信息中的doMathForever方法字节码。在运行之前，仍然要组建一个doMathForever栈帧压入Java栈。（注意：JVM会根据方法区中

doMathForever的字节码来创建栈帧的局部变量区和操作数栈的大小）
接下来JVM开始解释运行Act.doMathForever字节码的内容了。

垃圾回收

Java堆内存
分代收集
新生代（Young Generation）
Eden空间（Eden space，任何实例都通过Eden空间进入运行时内存区域）
S0 Survivor空间（S0 Survivor space，存在时间长的实例将会从Eden空间移动到S0 Survivor空间）
S1 Survivor空间 （存在时间更长的实例将会从S0 Survivor空间移动到S1 Survivor空间）
老年代（Old Generation）实例将从S1提升到Tenured（终身代）
永久代（Permanent Generation）包含类、方法等细节的元信息
永久代空间在Java SE8特性中已经被移除。
新生代： 使用标记复制清理算法，解决内存碎片问题。因为在年轻代会有大量的内存需要回收，GC比较频繁。通过这种方式来处理内存碎片化，然后在老年代中通过标记清理算法来回收内存，因为在老年代需要被回收的内存比较少，提高效率。
Eden 区： 当一个实例被创建了，首先会被存储在堆内存年轻代的 Eden 区中。
Survivor 区（S0 和 S1）： 作为年轻代 GC（Minor GC）周期的一部分，存活的对象（仍然被引用的）从 Eden区被移动到 Survivor 区的 S0 中。类似的，垃圾回收器会扫描 S0 然后将存活的实例移动到 S1 中。总会有一个空的survivor区。
老年代： 老年代（Old or tenured generation）是堆内存中的第二块逻辑区。当垃圾回收器执行 Minor GC 周期时（对象年龄计数器），在 S1 Survivor 区中的存活实例将会被晋升到老年代，而未被引用的对象被标记为回收。老年代是实例生命周期的最后阶段。Major GC 扫描老年代的垃圾回收过程。如果实例不再被引用，那么它们会被标记为回收，否则它们会继续留在老年代中。
内存碎片： 一旦实例从堆内存中被删除，其位置就会变空并且可用于未来实例的分配。这些空出的空间将会使整个内存区域碎片化。为了实例的快速分配，需要进行碎片整理。基于垃圾回收器的不同选择，回收的内存区域要么被不停地被整理，要么在一个单独的GC进程中完成
根可达性算法
Java语言规范没有明确地说明JVM使用哪种垃圾回收算法，但是任何一种垃圾收集算法一般要做2件基本的事情：
发现无用信息对象
回收被无用对象占用的内存空间，使该空间可被程序再次使用。
GC Roots
根集就是正在执行的Java程序可以访问的引用变量的集合（包括局部变量、参数、类变量）
GC Roots的对象包括
虚拟机栈中所引用的对象（本地变量表）

方法区中类静态属性引用的对象
方法区中常量引用的对象
本地方法栈中JNI引用的对象（Native对象）

可达性算法分析

通过一系列称为"GC Roots"的对象作为起点，从这些节点开始向下搜索，搜索所有走过的路径称为引用链，当一个对象到GC Roots没有任何引用链相连时(从GC Roots到此对象不可达)，则证明此对象是不可用的，应该被回收。

垃圾回收算法

引用计数法

引用计数法是唯一没有使用根集（GC Roots）的垃圾回收的法，该算法使用引用计数器来区分存活对象和不再使用的对象。堆中的每个对象对应一个引用计数器。当每一次创建一个对象并赋给一个变量时，引用计数器置为1。当对象被赋给任意变量时，引用计数器每次加1，当对象出了作用域后(该对象丢弃不再使用)，引用计数器减1，一旦引用计数器为0，对象就满足了垃圾收集的条件。

唯一没有使用根可达性算法的垃圾回收算法。

缺陷：不能解决循环引用的回收。

tracing算法（tracing collector）

tracing算法是为了解决引用计数法的问题而提出，它使用了根集（GC Roots）概念。垃圾收集器从根集开始扫描，识别出哪些对象可达，哪些对象不可达，并用某种方式标记可达对象，例如对每个可达对象设置一个或多个位。在扫描识别过程中，基于tracing算法的垃圾收集也称为标记和清除(mark-and-sweep)垃圾收集器。

compacting算法（Compacting Collector）

为了解决堆碎片问题，在清除的过程中，算法将所有的对象移到堆的一端，堆的另一端就变成了一个相邻的空闲内存区，收集器会对它移动的所有对象的所有引用进行更新，使得这些引用在新的位置能识别原来的对象。在基于Compacting算法的收集器的实现中，一般增加句柄和句柄表。

copying算法（Coping Collector）

该算法的提出是为了克服句柄的开销和解决堆碎片的垃圾回收。它开始时把堆分成 一个对象面和多个空闲面，程序从对象面为对象分配空间，当对象满了，基于coping算法的垃圾收集就从根集中扫描活动对象，并将每个活动对象复制到空闲面（使得活动对象所占的内存之间没有空闲洞），这样空闲面变成了对象面，原来的对象面变成了空闲面，程序会在新的对象面中分配内存。

generation算法（Generational Collector）：现在的java内存分区

stop-and-copy垃圾收集器的一个缺陷是收集器必须复制所有的活动对象，这增加了程序等待时间，这是coping算法低效的原因。在程序设计中有这样的规律：多数对象存在的时间比较短，少数的存在时间比较长。因此，generation算法将堆分成两个或多个，每个子堆作为对象的一代 （generation）。由于多数对象存在的时间比较短，随着程序丢弃不使用的对象，垃圾收集器将从最年轻的子堆中收集这些对象。在分代式的垃圾收集器运行后，上次运行存活下来的对象移到下一最高代的子堆中，由于老一代的子堆不会经常被回收，因而节省了时间。

adaptive算法（Adaptive Collector）

> 在特定的情况下，一些垃圾收集算法会优于其它算法。基于Adaptive算法的垃圾收集器就是监控当前堆的使用情况，并将选择适当算法的垃圾收集器