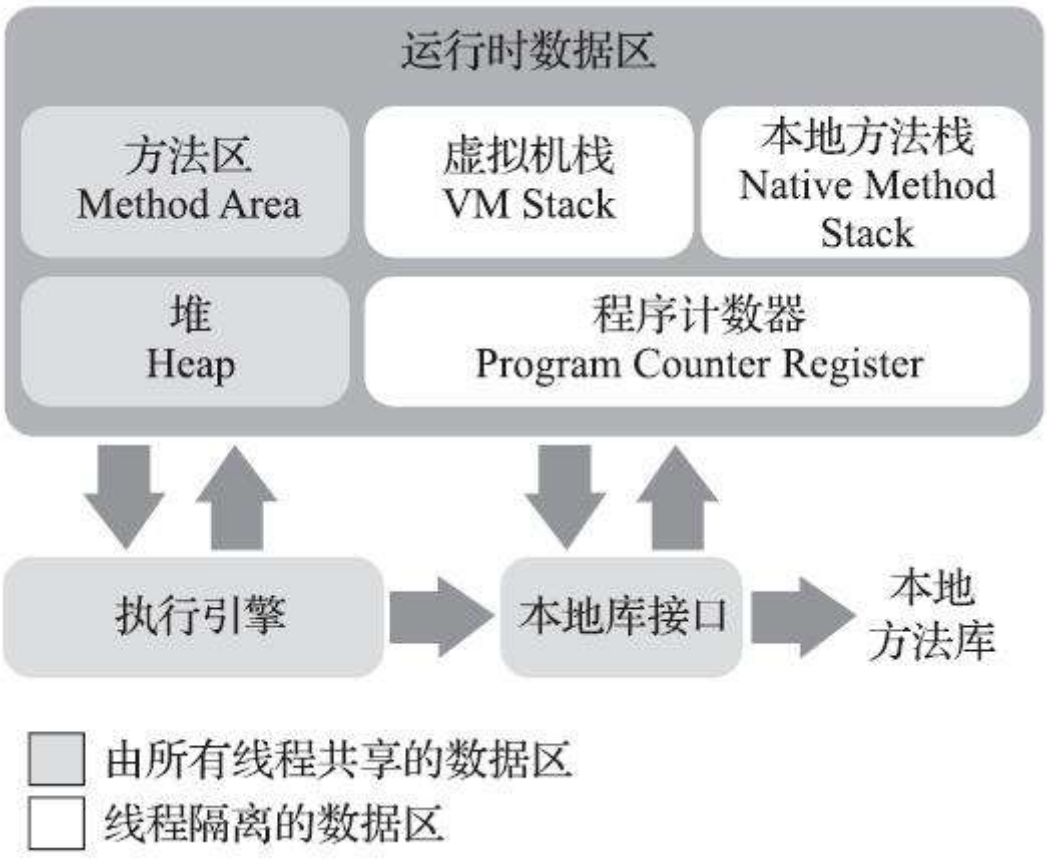
JVM内存区域



堆（Java Heap）

堆是所有类实例和数组的内存分配的运行时数据区。

虚拟机启动时创建。

堆可以为固定大小或动态扩展

堆的内存逻辑连续，物理不连续

堆是线程共享的

对象的堆存储由垃圾收集器回收

堆栈（Program Counter Register）

Java虚拟机堆栈存储帧。包含局部变量和部分结果，并在方法调用和返回中扮演角色。

每个线程都有一个私有Java虚拟机堆栈，同时创建线程的堆栈

堆栈可以为固定大小或动态扩展

堆栈是线程不共享的

本地方法栈（Native Method Stacks）

与虚拟机栈类似，为Native方法服务。一般是java调用其他语言时使用。

方法区（Method Area）

存储已被虚拟机加载的类信息、常量、静态变量、即时编译后的代码等数据。

可以为固定大小或动态扩展

方法区是线程共享的

方法区可以不被垃圾收集器回收

运行时常量池（Runtime Constant Pool）

存放编译器生成的各种字面量和符号引用以及直接引用。

方法区的一部分。

程序计数器（Program Counter Register）

当前线程所执行的字节码的行号指示器。

字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令。

直接内存（Direct Memory）

非虚拟机运行时内存

基于通道与缓冲区的NIO模式下，直接内存可以使用native函数库直接分配堆外内存，然后通过一个存储在java堆中的DirectByteBuffer对象作为这块内存的引用进行操作。避免java堆和natice堆中来回复制数据

内存模型特征对比图

内存区域	创建时期	动态扩展	线程共享	溢出
jvm堆	虚拟机启动	Y	Y	Y
jvm栈	线程创建	Y	N	Y
本地方法栈	线程创建	Y	N	Y
方法区	虚拟机启动	Y	Y	Y
运行时常量池	虚拟机启动	Y	N	Y
程序计数器	线程创建	\	N	N

对象创建

创建过程

1.检查是否加载过

虚拟机解析new指令，首先检查常量池是否有类的符号引用，并且检查是否已经加载解析和初始化，没有就执行类加载过程

2.分配内存

虚拟机在java堆中分配空间。如果内存是规整的，虚拟机通过“指针碰撞”分配内存、如果内存不规整的，虚拟机维护一个内存块使用情况表，通过“空闲列表”分配内存。

3.初始化

内存空间初始化为零值。这也是有些对象不赋值就可以使用的原因。

4.设置对象头

在对象的对象头中设置：对象是哪个类的实例，如何找到类的元数据信息、对象的哈希码、对象的Gc分带年龄等信息。

5.执行init指令

以上4步一个新的对象已经产生，但是所有的字段都是零。接着字节码解析器调用init指令，这个对象才算完全产生出来。

对象的内存布局

1.对象头

1.1、运行时数据包括哈希码（HashCode）、GC分代年龄、锁状态标志、线程持有的锁、偏向锁ID和偏向时间戳等，这部分数据在32位和64位虚拟机中的长度分别为32bit和64bit，官方称为"Mark Word"。Mark Word被设计成非固定的数据结构，以实现在有限空间内保存尽可能多的数据。

32位的虚拟机中，对象未被锁定的状态下，Mark Word的32bit中25bit存储对象的HashCode、4bit存储对象分代年龄、2bit存储锁标志位、1bit固定为0，具体如下：

其它状态（轻量级锁定、重量级锁定、GC锁定、可偏向锁）下Mark Word的存储内容如下：

1.2、对象头的类型指针指向该对象的类元数据，虚拟机通过这个指针可以确定该对象是哪个类的实例。

2.实例数据

实例数据就是在程序代码中所定义的各种类型的字段，包括从父类继承的，这部分的存储顺序会受到虚拟机分配策略和字段在源码中定义顺序的影响。

3.对齐填充

由于HotSpot的自动内存管理要求对象的起始地址必须是8字节的整数倍，即对象的大小必须是8字节的整数倍，对象头的数据正好是8的整数倍，所以当实例数据不够8字节整数倍时，需要通过对齐填充进行补全。

溢出分析

溢出区域

内存溢出分为内内存溢出和栈内存溢出

如果虚拟机在扩展栈时无法申请到足够大的内存空间时，就会抛出outOfMemoryError

如果线程请求的栈深度大于虚拟机所允许的最大深度，则抛出StackOverflowError

jvm参数设置

栈溢出代码

```
 public static void main(String[] args) {
    a();
}
public static String a(){
    return a();
}
```

堆溢出代码

为了更好检测和查看溢出原因在运行前先设置jvm参数

-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/Users/xianghan/Desktop/bak -Xmx2m -Xms2m

jvm参数含义请参考：http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html

生成的二进制文件格式为hprof,可以使用jprofile打开分析具体导致异常的代码

```
public static void main(String[] args) {
    List<Integer> list=new ArrayList<>();
    while (true){
        list.add(1);
    }
}
```

线程数过多导致堆溢出

==在执行前一定小心保存当前工作内容，防止机器卡死。==

```
public static void main(String[] args) {
    try {
        while (true){
            Thread thread=new Thread(new Runnable() {
                @Override
                public void run() {
                    while (true){
                    }
                }
            });
            System.out.println(thread.getName());
            thread.start();
        }
    } catch (Exception e) {
                e.printStackTrace();
        }
}
```

常量池溢出

```
// -Xmx10m -Xms10m
public static void main(String[] args) {
     List<String> list = new ArrayList<String>();
    int i = 0;
    while (true) {
        list.add(String.valueOf(i++).intern());
    }
}
```

String.intern()是一个Native方法,它的作用是:如果字符串常池中已包含一个等于此String对象的字符串,则返回代表池中这个字符串的String对象,否则将此String对象包含的字符串添加到常池中,并且返回此string对象的引用。在JDK1.6及之前的版本中,由于常池分配在永久代内,我们可以通过-XX:PermSize=10M -XX:MaxPermSize=10M。MaxPermSize限制方法区大小从而间接限制其中常量池的容量

本机直接内存溢出

如果不指定MaxDirectMemorySize那么直接内存与堆内存一样大

==运行此代码风险较大直接死机==

```
import sun.misc.Unsafe;
import java.lang.reflect.Field;
public class Test {
    public static int _1MB=1024*1024;
    //-Xmx20M -XX:MaxDirectMemorySize=10M
    public static void main(String[] args) throws Exception{
        Field unsafeField=Unsafe.class.getDeclaredFields()[0];
        unsafeField.setAccessible(true);
        Unsafe unsafe =(Unsafe) unsafeField.get(null);
        while (true){
            unsafe.allocateMemory(_1MB);
        }
    }
```

```
}
```

字节码指令

java文件

```
package com.naixue.vip.p6.jvm.bytecode;

/**
 * @Description
 * @Author 向寒 奈学教育
 * @Date 2020/7/29 14:32
 **/
public class Hello {
    private int n=0;
}
```

编译class

```
javac Hello.java
# 得到 Hello.class文件
```

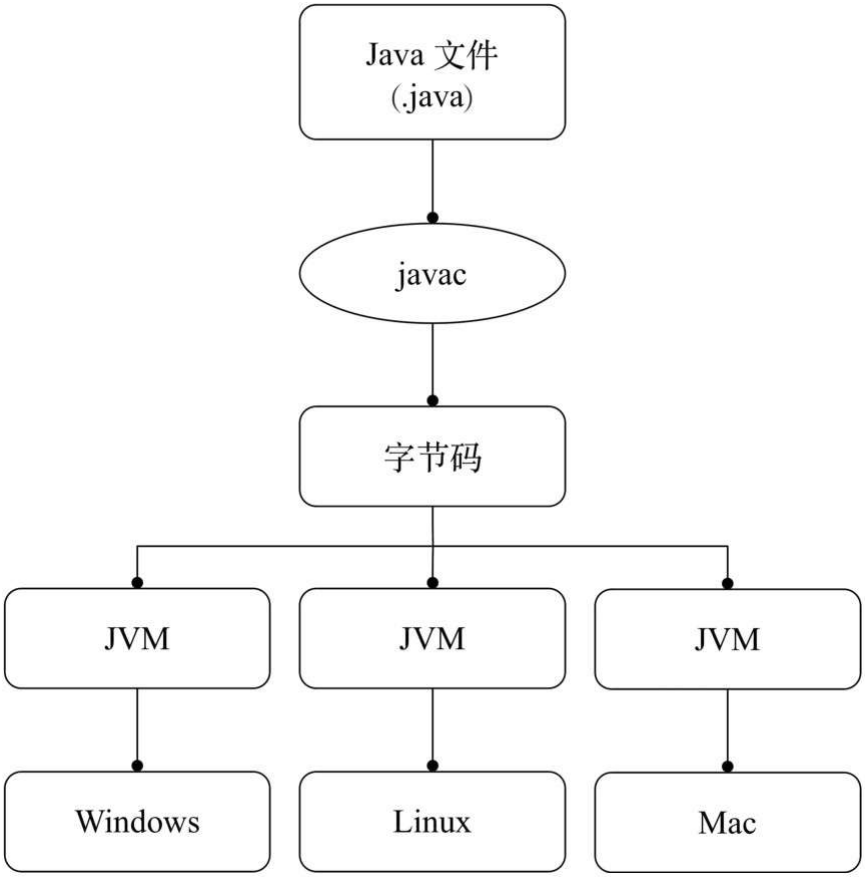
查看class文件

```
cafe babe 0000 0034 0011 0a00 0400 0d09
0003 000e 0700 0f07 0010 0100 016e 0100
0149 0100 063c 696e 6974 3e01 0003 2829
5601 0004 436f 6465 0100 0f4c 696e 654e
756d 6265 7254 6162 6c65 0100 0a53 6f75
7263 6546 696c 6501 000a 4865 6c6c 6f2e
6a61 7661 0c00 0700 080c 0005 0006 0100
2463 6f6d 2f6e 6169 7875 652f 7669 702f
7036 2f6a 766d 2f62 7974 6563 6f64 652f
4865 6c6c 6f01 0010 6a61 7661 2f6c 616e
672f 4f62 6a65 6374 0021 0003 0004 0000
0001 0002 0005 0006 0000 0001 0001 0007
0008 0001 0009 0000 0026 0002 0001 0000
000a 2ab7 0001 2a03 b500 02b1 0000 0001
000a 0000 000a 0002 0000 0008 0004 0009
0001 000b 0000 0002 000c
```

初探class文件

“一次编写，到处运行”即Java编译生成的二进制文件能够在不做任何改变的情况下运行于多个平台

Java是平台无关的语言，但JVM却不是跨平台的，不同平台的JVM帮我们屏蔽了平台的差异。通过这些虚拟机加载和执行同一种平台无关的字节码，我们的源代码就不用根据不同平台编译成不同的二进制可执行文件



字节码构成

class文件由下面十个部分组成：

❑ 魔数（Magic Number）

❑ 版本号（Minor&Major Version）

❑ 常量池（Constant Pool）

❑ 类访问标记（Access Flag）

❑ 类索引（This Class）

❑ 超类索引（Super Class）

❑ 接口表索引（Interface）

❑ 字段表（Field）

❑ 方法表（Method）

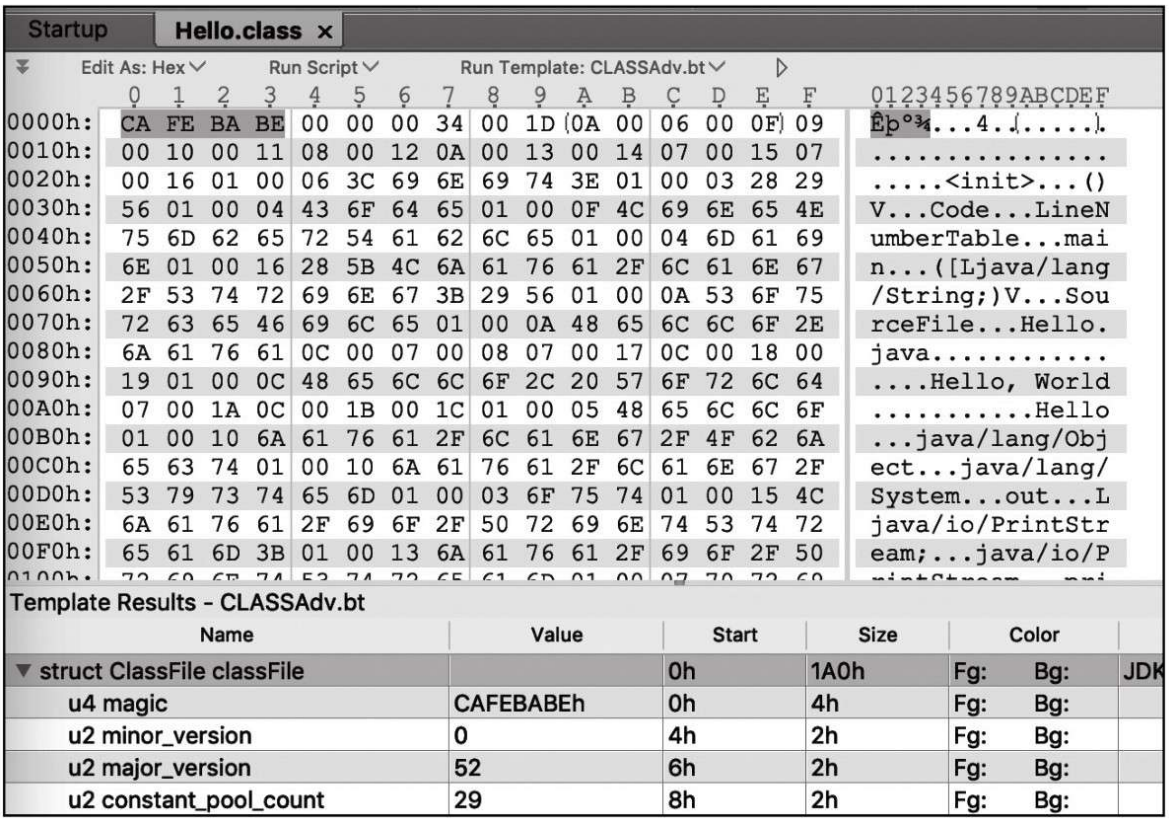
❑ 属性表（Attribute）

魔数

使用文件名后缀来区分文件类型很不靠谱，后缀可以被随便修改，可以用魔数（Magic Number）实现，根据文件内容本身来标识文件的类型

很多文件都以固定的几字节开头作为魔数，比如PDF文件的魔数是 %PDF-（十六进制0x255044462D），png文件的魔数是 \x89PNG（十六进制0x89504E47）。

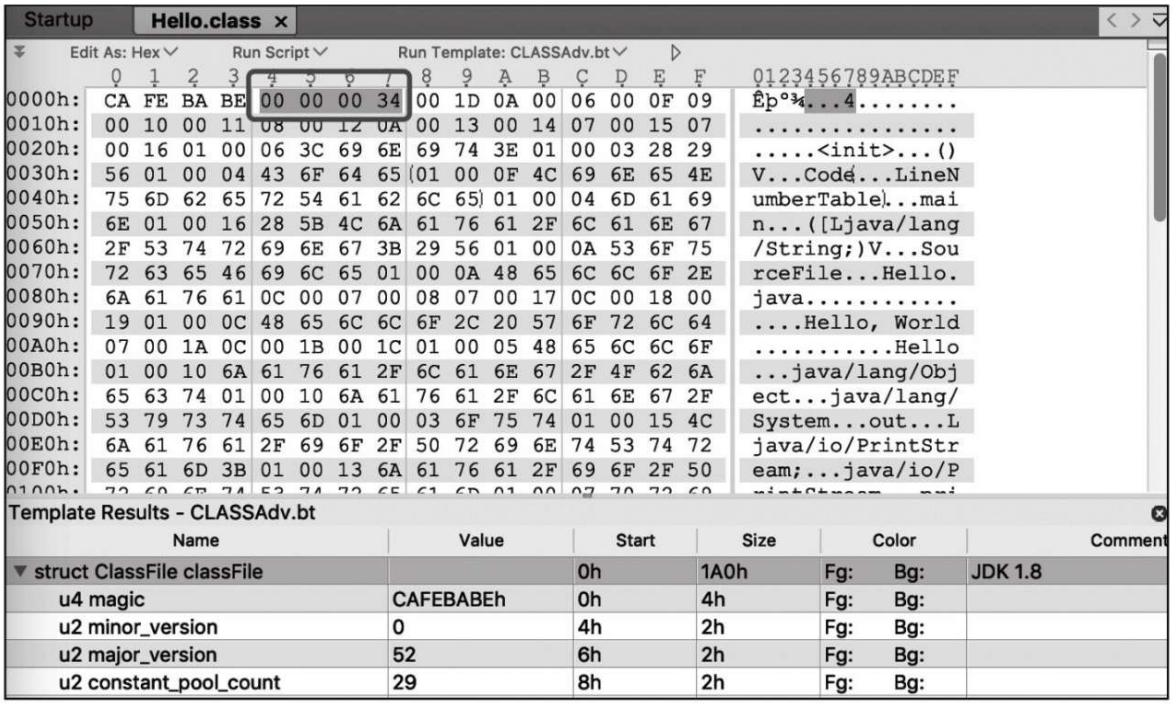
使用十六进制工具打开class文件，首先看到的是充满浪漫气息的魔数0xCAFEBABE（咖啡宝贝）



版本号

在魔数之后的四个字节分别表示副版本号（Minor Version）和主版本号（MajorVersion）

主版本号是52（0x34），虚拟机解析这个类时就知道这是一个Java 8编译出的类



常量池

常量池是类文件中最复杂的数据结构。对于JVM字节码来说，如果操作数是很常用的数字，比如0，这些操作数是内嵌到字节码中的。如果是字符串常量和较大的整数等，class文件则会把这些操作数存储在常量池（Constant Pool）中，当使用这些操作数时，会根据常量池的索引位置来查找。

```
javap -v Hello
```

```
警告: 二进制文件Hello包含com.naixue.vip.p6.jvm.bytecode.Hello
Classfile
/Users/xianghan/work/naixue/gitlab/jvm/src/main/java/com/naixue/vip/p6/jvm/bytec
ode/Hello.class
  Last modified 2020-7-29; size 250 bytes
  MD5 checksum 64f87b18667dac2661dbfc292cfad7eb
  Compiled from "Hello.java"
public class com.naixue.vip.p6.jvm.bytecode.Hello
  minor version: 0
  major version: 52
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
   #1 = Methodref          #4.#13         // java/lang/Object."<init>":()V
   #2 = Fieldref           #3.#14         //
com/naixue/vip/p6/jvm/bytecode/Hello.n:I
   #3 = Class              #15            //
com/naixue/vip/p6/jvm/bytecode/Hello
   #4 = Class              #16            // java/lang/Object
   #5 = Utf8               n
   #6 = Utf8               I
   #7 = Utf8               <init>
   #8 = Utf8               ()V
   #9 = Utf8               Code
  #10 = Utf8               LineNumberTable
  #11 = Utf8               SourceFile
  #12 = Utf8               Hello.java
  #13 = NameAndType        #7:#8          // "<init>":()V
  #14 = NameAndType        #5:#6          // n:I
  #15 = Utf8               com/naixue/vip/p6/jvm/bytecode/Hello
  #16 = Utf8               java/lang/Object
{
  public com.naixue.vip.p6.jvm.bytecode.Hello();
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
      stack=2, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1                  // Method java/lang/Object."
<init>":()V
         4: aload_0
         5: iconst_0
         6: putfield      #2                  // Field n:I
         9: return
      LineNumberTable:
        line 8: 0
        line 9: 4
}
SourceFile: "Hello.java"
```

访问标记（Access flags）

访问标记（Access flags），用来标识一个类为final、abstract等，由两个字节表示，总共有16个标记位可供使用

继承关系

this_class表示类索引，super_name表示直接父类的索引，interfaces表示类或者接口的直接父接口

字段表

字段表（fields），类中定义的字段会被存储到这个集合中，包括静态和非静态的字段

方法表

类中定义的方法会被存储在方法表，方法表也是一个变长结构。

- 方法method_info结构
- 方法访问标记
- 方法名与描述符
- 方法属性表

属性表

属性表使用两个字节表示属性的个数attributes_count，接下来是若干个属性项的集合，可以看作是一个数组，数组的每一项都是一个属性项attribute_info，数组的大小为attributes_count

- ConstantValue属性ConstantValue属性出现在字段field_info中，用来表示静态变量的初始值
- Code属性Code属性是类文件中最重要的组成部分，它包含方法的字节码，除native和abstract方法以外，每个method都有且仅有一个Code属性

javap查看class

```
javap [options] <classes>
```

javap会显示访问权限为public、protected和默认级别的方法。如果想要显示private方法和字段，就需要加上 -p选项

javap还有一个好用的选项 -s，可以输出类型描述符签名信息

加上 -c选项可以对类文件进行反编译，可以显示出方法内的字节码

加上 -v选项可以显示更加详细的内容，比如版本号、类访问权限、常量池相关的信息

加上-l选项，可以用来显示行号表和局部变量表，实测并没有输出局部变量表，只显示了行号表

加 -g选项，生成所有的调试信息选项，加上 -g选项编译javac -g HelloWorld.java以后重新执行javap -l命令就可以看到局部变量表

概述

Java虚拟机的指令由一个字节长度的操作码（opcode）和紧随其后的可选的操作数（operand）构成，如下所示。

> [,]

比如将整型常量100压栈到栈顶的指令是bipush 100，其中bipush是操作码，100是操作数

比如字节码getfield 0002，表示的是getfiled 0x00 < < 8 | 0x02（getfield #2）

根据字节码的不同用途，字节码指令可以大概分为如下几类：

❑ 加载和存储指令，比如iload将一个整型值从局部变量表加载到操作数栈；

❑ 控制转移指令，比如条件分支ifeq；

❑ 对象操作，比如创建类实例的指令new；

❑ 方法调用，比如invokevirtual指令用于调用对象的实例方法；

❑ 运算指令和类型转换，比如加法指令iadd；

❑ 线程同步，比如monitorenter和monitorexit这两条指令用于支持synchronized关键字的语义；

❑ 异常处理，比如athrow显式抛出异常。

类加载机制

定义

Java虚拟机把描述类的数据从Class文件加载到内存，并对数据进行校验、转换解析和初始化，最终形成可以被虚拟机直接使用的Java类型，这个过程被称作虚拟机的类加载机制

一个类型从被加载到虚拟机内存中开始，到卸载出内存为止，它的整个生命周期将会经历加载（Loading）、验证（Verification）、准备（Preparation）、解析（Resolution）、初始化（Initialization）、使用（Using）和卸载（Unloading）七个阶段

加载阶段

“加载”（Loading）阶段是整个“类加载”（Class Loading）过程中的一个阶段

Java虚拟机需要完成以下三件事情：

1）通过一个类的全限定名来获取定义此类的二进制字节流。

2）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。

3）在内存中生成一个代表这个类的java.lang.Class对象，作为方法区这个类的各种数据的访问入口。

验证阶段

验证是连接阶段的第一步，这一阶段的目的是确保Class文件的字节流中包含的信息符合《Java虚拟机规范》的全部约束要求，保证这些信息被当作代码运行后不会危害虚拟机自身的安全

1.文件格式验证

·是否以魔数0xCAFEBABE开头。

·主、次版本号是否在当前Java虚拟机接受范围之内。

·常量池的常量中是否有不被支持的常量类型（检查常量tag标志）。

·指向常量的各种索引值中是否有指向不存在的常量或不符合类型的常量。

·CONSTANT_Utf8_info型的常量中是否有不符合UTF-8编码的数据。

·Class文件中各个部分及文件本身是否有被删除的或附加的其他信息。

2.元数据验证

第二阶段是对字节码描述的信息进行语义分析，以保证其描述的信息符合《Java语言规范》的要求，这个阶段可能包括的验证点如下：

·这个类是否有父类（除了java.lang.Object之外，所有的类都应当有父类）。

·这个类的父类是否继承了不允许被继承的类（被final修饰的类）。

·如果这个类不是抽象类，是否实现了其父类或接口之中要求实现的所有方法。

3.字节码验证

第三阶段是整个验证过程中最复杂的一个阶段，主要目的是通过数据流分析和控制流分析，确定程序语义是合法的、符合逻辑的

·保证任意时刻操作数栈的数据类型与指令代码序列都能配合工作，例如不会出现类似于“在操作栈放置了一个int类型的数据，使用时却按long类型来加载入本地变量表中”这样的情况。

·保证任何跳转指令都不会跳转到方法体以外的字节码指令上。

·保证方法体中的类型转换总是有效的，例如可以把一个子类对象赋值给父类数据类型，这是安全的，但是把父类对象赋值给子类数据类型，甚至把对象赋值给与它毫无继承关系、完全不相干的一个数据类型，则是危险和不合法的

4.符号引用验证

最后一个阶段的校验行为发生在虚拟机将符号引用转化为直接引用[插图]的时候，这个转化动作将在连接的第三阶段——解析阶段中发生。符号引用验证可以看作是对类自身以外（常量池中的各种符号引用）的各类信息进行匹配性校验，通俗来说就是，该类是否缺少或者被禁止访问它依赖的某些外部类、方法、字段等资源

·符号引用中通过字符串描述的全限定名是否能找到对应的类。

·在指定类中是否存在符合方法的字段描述符及简单名称所描述的方法和字段。

·符号引用中的类、字段、方法的可访问性（private、protected、public、）是否可被当前类访问。

准备阶段

准备阶段是正式为类中定义的变量（即静态变量，被static修饰的变量）分配内存并设置类变量初始值的阶段

首先是这时候进行内存分配的仅包括类变量，而不包括实例变量，实例变量将会在对象实例化时随着对象一起分配在Java堆中。其次是这里所说的初始值“通常情况”下是数据类型的零值

解析阶段

解析阶段是Java虚拟机将常量池内的符号引用替换为直接引用的过程

1.类或接口的解析

2.字段解析

3.方法解析

4.接口方法解析

初始化阶段

类的初始化阶段是类加载过程的最后一个步骤

直到初始化阶段，Java虚拟机才真正开始执行类中编写的Java程序代码，将主导权移交给应用程序

类加载器

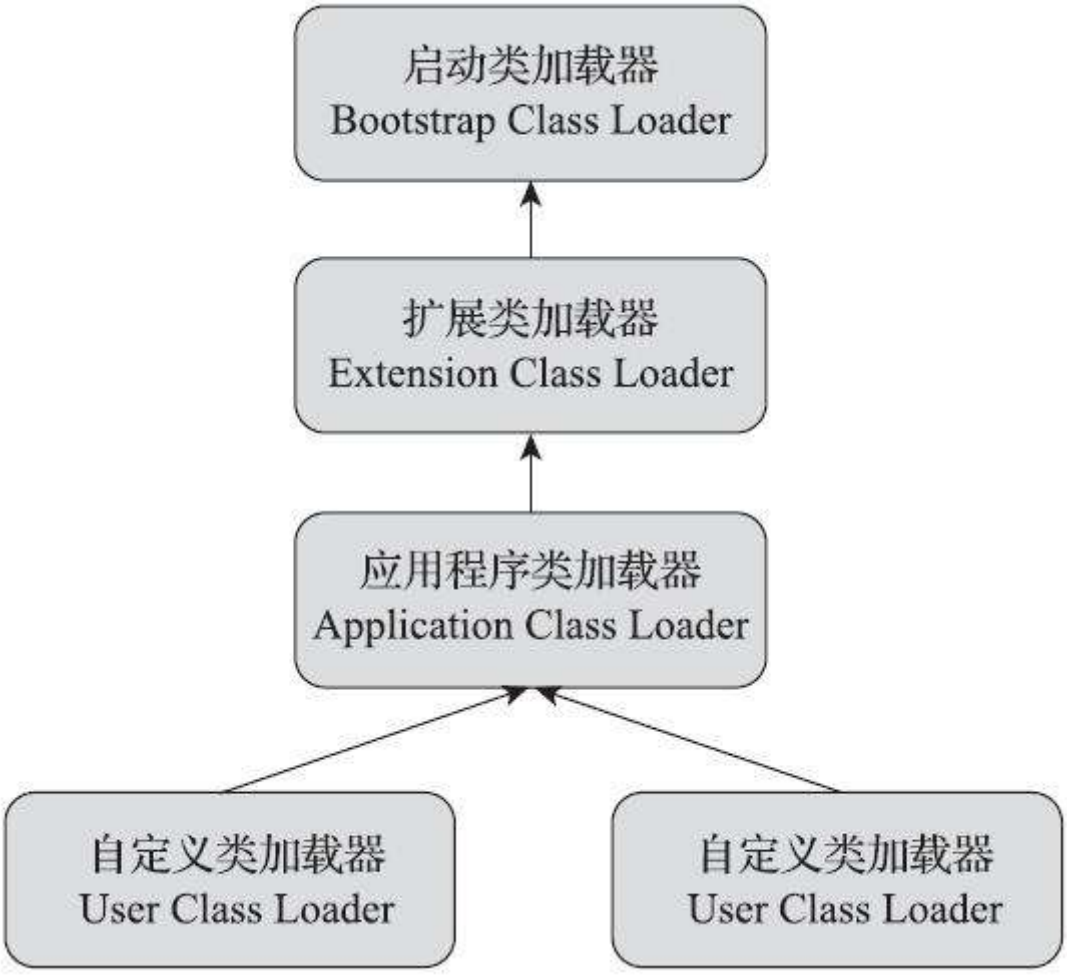
只存在两种不同的类加载器：一种是启动类加载器（Bootstrap ClassLoader），这个类加载器使用C++语言实现，是虚拟机自身的一部分；另外一种就是其他所有的类加载器，这些类加载器都由Java语言实现，独立存在于虚拟机外部，并且全都继承自抽象类java.lang.ClassLoader

·启动类加载器（Bootstrap Class Loader）：前面已经介绍过，这个类加载器负责加载存放在<JAVA_HOME>\lib目录，或者被-Xbootclasspath参数所指定的路径中存放的，而且是Java虚拟机能够识别的（按照文件名识别，如rt.jar、tools.jar，名字不符合的类库即使放在lib目录中也不会被加载）类库加载到虚拟机的内存中

·扩展类加载器（Extension Class Loader）：这个类加载器是在类sun.misc.Launcher$ExtClassLoader中以Java代码的形式实现的。它负责加载<JAVA_HOME>\lib\ext目录中，或者被java.ext.dirs系统变量所指定的路径中所有的类库

·应用程序类加载器（Application Class Loader）：这个类加载器由sun.misc.Launcher$AppClassLoader来实现。由于应用程序类加载器是ClassLoader类中的getSystem-ClassLoader()方法的返回值，所以有些场合中也称它为“系统类加载器”。它负责加载用户类路径（ClassPath）上所有的类库

双亲委派机制



各种类加载器之间的层次关系被称为类加载器的“双亲委派模型（Parents Delegation Model）”。双亲委派模型要求除了顶层的启动类加载器外，其余的类加载器都应有自己的父类加载器

工作过程

双亲委派模型的工作过程是：如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成，每一个层次的类加载器都是如此，因此所有的加载请求最终都应该传送到最顶层的启动类加载器中，只有当父加载器反馈自己无法完成这个加载请求（它的搜索范围中没有找到所需的类）时，子加载器才会尝试自己去完成加载

作用

- 使用双亲委派模型来组织类加载器之间的关系，一个显而易见的好处就是Java中的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
- 解决安全问题

破坏双亲委派机制

双亲委派模型并不是一个具有强制性约束的模型，而是Java设计者推荐给开发者们的类加载器实现方式

1. 双亲委派模型的第一次“被破坏”其实发生在双亲委派模型出现之前——即JDK 1.2面世以前的“远古”时代
2. 双亲委派模型的第二次“被破坏”是由这个模型自身的缺陷导致的

   双亲委派很好地解决了各个类加载器协作时基础类型的一致性问题，但程序设计往往没有绝对不变的完美规则，如果有基础类型又要调用回用户的代码

   一个典型的例子便是JNDI服务

   JNDI存在的目的就是对资源进行查找和集中管理，它需要调用由其他厂商实现并部署在应用程序的ClassPath下的JNDI服务提供者接口（Service Provider Interface，SPI）的代码，启动类加载器是绝不可能认识、加载这些代码的

   为了解决这个困境，Java的设计团队只好引入了一个不太优雅的设计：线程上下文类加载器（Thread ContextClassLoader）。这个类加载器可以通过java.lang.Thread类的setContext-ClassLoader()方法进行设置，如果创建线程时还未设置，它将会从父线程中继承一个，如果在应用程序的全局范围内都没有设置过的话，那这个类加载器默认就是应用程序类加载器

   JNDI服务使用这个线程上下文类加载器去加载所需的SPI服务代码，这是一种父类加载器去请求子类加载器完成类加载的行为，这种行为实际上是打通了双亲委派模型的层次结构来逆向使用类加载器，已经违背了双亲委派模型的一般性原则，但也是无可奈何的事情
3. 双亲委派模型的第三次“被破坏”是由于用户对程序动态性的追求而导致的，这里所说的“动态性”指的是一些非常“热”门的名词：代码热替换（Hot Swap）、模块热部署（HotDeployment）等

   OSGi实现模块化热部署的关键是它自定义的类加载器机制的实现，每一个程序模块（OSGi中称为Bundle）都有一个自己的类加载器，当需要更换一个Bundle时，就把Bundle连同类加载器一起换掉以实现代码的热替换。在OSGi环境下，类加载器不再双亲委派模型推荐的树状结构，而是进一步发展为更加复杂的网状结构

类加载过程代码