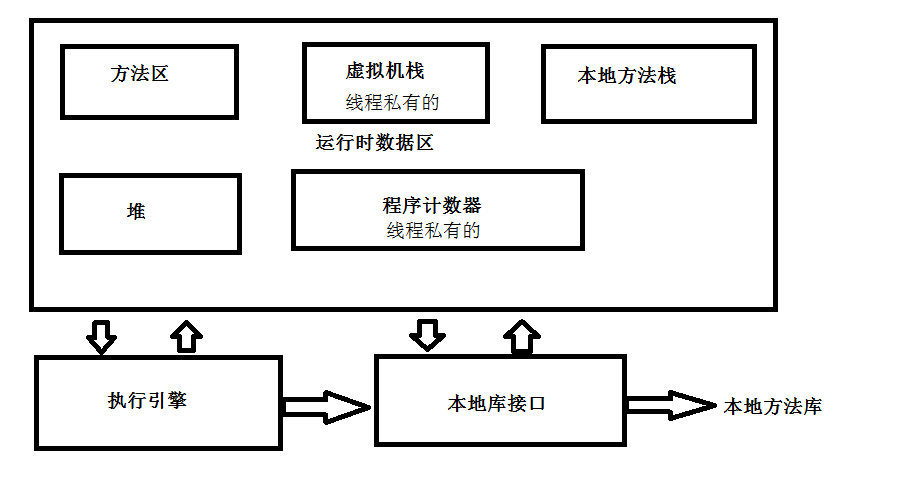
一、自动内存管理机制



1、程序计数器：当前线程所执行的字节码行号指示器。（如果线程正执行的是一个Java方法，则这个计数器记录的是正在执行的虚拟机字节码指令地址，如果是native方法，则这个计数器值为空；唯一一个在Java虚拟机规范里没有规定任何OutOfMemoryError情况的区域）。

2、Java虚拟机栈：每个方法被执行时都会创建一个栈帧用于存储局部变量表、操作栈、动态链接、方法出口等信息，每个方法调用就对应一个栈帧在虚拟机栈中从入栈到出栈的过程。

                局部变量表：存放编译器可知的各种基本数据类型（Boolean，byte，char，short，int，float，long，double，ref，returnAddres类型（指向了一条字节码指令的地址））；局部变量表在编译器完成分配，方法运行期间不会改变。

                            Slot：局部变量空间，long和double占两个，其余只占一个；

                            boolean 1字节（8位）

                            byte  1字节（8位）

                            char  2字节（16位）

                            short 2字节（16位）

                            int   4字节（32位）

                            float 4字节（32位）

                            long  8字节（64位）

                            double 8字节（64位）

    这个区域的两个异常：StackOverflowError：若线程请求的栈深度大于虚拟机所允许的深度时；

                                     OutOfMemoryError：虚拟机栈动态扩展时无法申请到足够的内存时；

3、本地方法栈：与虚拟机栈作用相似，只不过是为Native方法服务。（有的虚拟机如HotSpot直接把本地方法栈和虚拟机栈合二为一）也是有两个异常：StackOverflowError、OutOfMemoryError

4、Java堆：存放实例对象和数组。（JIT编译器以及逃逸分析技术、栈上分配、标量替换等优化技术的出现导致所有对象都在堆上分配不再那么绝对了）

                  新生代，老年代，再细一点的有Eden空间、From Survivor空间、To Survivor空间等。

                  堆是可扩展的：通过-Xmx  -Xms控制（在堆中没有完成实例分配，并且堆也无法再扩展时抛出OutOfMemoryError）

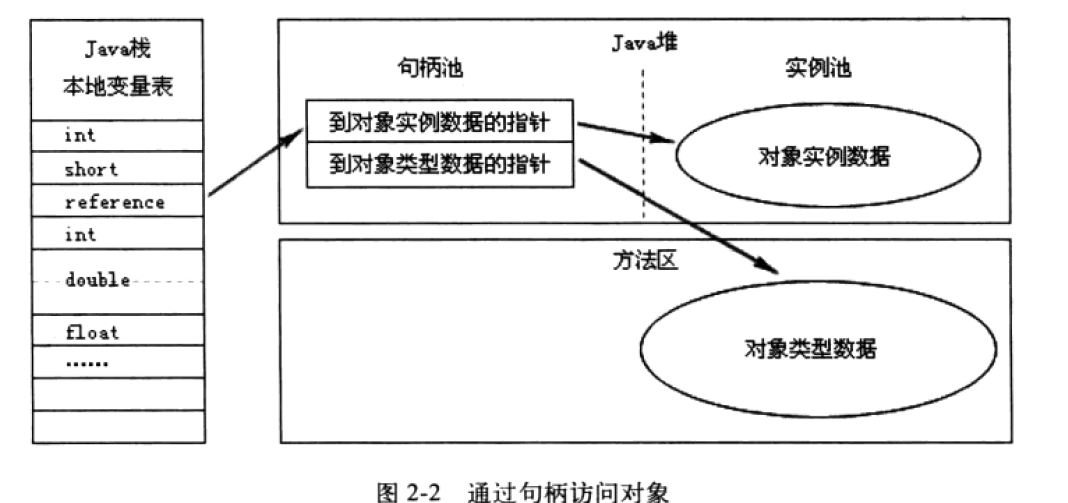
5、方法区：存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后代码等数据。（非堆NonHeap，java虚拟机规范把方法区描述为堆的一个逻辑部分）（永久代-->Native Memory）

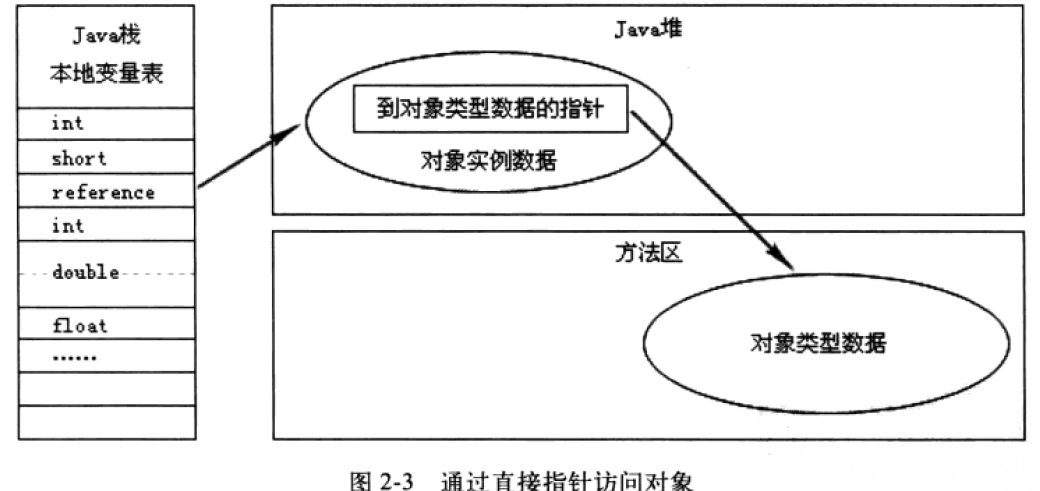
         运行时常量池：是方法区的一部分，用于存放编译器生成的各种字面量和符号引用（class加载之后将编译器生成的各种字面量和符号引用、直接引用存放到方法区的运行时常量池中）

                       运行时常量池相对于Class文件常量池的另一个重要特征是具备动态性（也就是并不要求常量一定只能在编译期产生，并非预置入Class文件中常量池的内容才能进入方法区运行时常量池，运行期间也可以，比如String类的intern()方法）。有OutOfMemoryError异常

6、直接内存：不是虚拟机运行时数据区的一部分，也不是Java虚拟机规范中定义的内存区域。jdk1.4引入的NIO引入了一种基于通道与缓冲区的IO方式，可以直接使用Native函数库直接分配堆外内存，然后通过Java堆里面的DirectByteBuffer对象作为这块内存的引用进行操作（避免了再Java堆和Native堆中来回复制数据）

7、对象访问：





8、内存泄露（有没被回收掉的对象），内存溢出（内存不够分配了）

     如果是建立过多线程导致的内存溢出，在不能减少线程数或更换64位虚拟机时就只能通过减少最大堆和减少栈容量来换取更多线程。

     -Xmx:堆最大size

     -Xms：堆最小size

     -Xoss：本地方法栈大小

     -Xss：栈容量

     -XX:PermSize=10M, -XX:MaxPermSize=10M：限制方法区大小

     -XX:MaxDirectMemorySize：指定本机直接内存大小（若不指定默认与Java堆的最大值-Xmx一样）

二、GC

1、根搜索算法：通过一系列的“GC Roots”对象作为起始点，从这些节点开始往下搜索，搜索所走过的路径成为引用链，当一个对象到GC roots没有任何引用链相连时此对象是不可用的；

            GC Roots：

             虚拟机栈（栈帧中的本地变量表）中的引用的对象；

             方法区中的类静态属性引用的对象；

             方法区中的常量引用的对象；

             本地方法栈中JNI（Native方法）的引用对象。

    （1）引用：1）强引用：Object ojb = new Object(); 只要强引用还在，垃圾收集器就不会回收被引用的对象；

                      2）软引用：还有用但非必须得，对于软引用关联着的对象，在系统将发生内存溢出之前会把这些对象列进回收范围并进行第二次回收，若回收后还没有足够内存才会抛内存溢出异常（SoftReference来实现）

                      3）弱引用：每次垃圾收集都会回收的（用WeakReference类来实现）

                      4）虚引用（幻影引用）：无法通过这种引用来获取一个对象，虚引用的唯一目的就是希望在这个对象被回收时收到一个系统通知。

     （2）回收要经过两次标记的过程：当发现没有与GC Roots连接的对象时会进行第一次标记，并进行一次筛选（筛选条件是此对象有无必要执行finalize()方法）。如果有必要执行（覆盖了finalize()并且没被执行过），则这个对象会被放置在一个名为F-Queue队列中，稍后由一条由虚拟机自动建立的低优先级的Finalizer线程去执行（虚拟机触发这个方法，但并不承诺会等待它运行结束）

     （3）回收方法区：Java虚拟机规范不要求在方法区实现垃圾回收（效率低，堆中一次GC可以收回70%-95%）。

                    永久代的GC主要回收两部分：废弃的常量和无用的类。

                     1）回收废弃常量与回收堆的对象类似，例如常量池中的一个字面量“abc”，但是当前系统中没有任何一个String对象是引用它，那就清理出常量池（常量池中的其他类、接口、方法、字段的符号引用也是类似）

                     2）回收无用的类：该类所有的实例都已经被回收；

                                                 加载该类的ClassLoader已经被回收；

                                                 该类对应的Java.lang.Class对象没有在任何地方呗引用，无法在任何地方通过反射访问该类的方法。

                           这时**可以**进行回收。

                           -Xnoclassgc:是否对类进行回收；

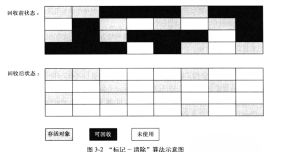
                           -verbose:class及-XX:+TraceClassLoading、-XX:+TraceClassUnLoading查看类的加载和卸载信息；

                           在大量使用反射、动态代理、CGLIB等需要频繁自定义ClassLoader场景都需要虚拟机具备类卸载功能，以保证永久代不会溢出。

2、GC算法：

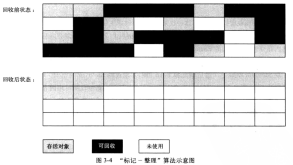
    （1）标记-清除算法：先标记一遍，然后统一回收掉被标记的对象。

             缺点：一是效率不高，标记和清除效率都不高；二是空间问题，标记清除之后会产生大量不连续的内存碎片。



     （2）复制算法（针对新生代的GC）：将内存分为一块较大的Eden空间和两块较小的Survivor空间（8:1），每次使用Eden和其中的一块Survivor，回收时将Eden和Survivor中还存活的对象一次性的copy到另外一块Survivor上，最后清理掉Eden和刚刚用过的Survivor。（Survivor空间不够用时需要依赖其他内存（老年代）进行分配担保）

     （3）标记-整理算法（针对老年代的GC）：第一步和标记清除一样，后面不是清除，而是将存活对象都向一端移动，然后直接清理掉端边界以外的内存；



      （4）分代收集算法：将内存划分为几块，一般为新生代、老年代，新生代使用复制算法，老年代使用标记清理或者标记整理算法。

堆：新生代（eden、Survivor空间：复制算法）、老年代（标记清理或标记整理）

方法区：永久代

GCLogViewer

3、垃圾收集器：

       （1）Serial收集器：单线程收集器（stop the world）

       （2）ParNew收集器：Serial的多线程版本

       （3）Parallel Scavenge收集器：利用复制算法的新生代收集器（关注吞吐量）；

       （4）Serial Old收集器：单线程收集器，使用标记-整理算法；

       （5）Parallel Old收集器：是Parallel Scavenge的老年代版本，使用多线程和标记整理算法；

       （6）CMS（Concurrent Mark Sweep）收集器：以获取最短回收停顿时间为目标的收集器，使用标记清除算法；

       （7）G1收集器：使用标记整理算法，不会产生空间碎片。

4、内存分配策略：

       （1）对象优先在Eden分配；

       （2）大对象直接进入老年代；

       （3）长期存活的对象进入老年代；每进行一次GC没被回收的对象年龄加一，增加到一定程度（默认15岁）时晋升到老年代中；

       （4）动态年龄判定；虚拟机并不要求一定大于MaxTenuringThreshold才晋升老年代，若在Survivor空间中相同年龄所有对象大小总和大约Survivor空间的一半，则大于等于该年龄的直接进老年代

       （5）空间分配担保；虚拟机会检测之前每次晋升到老年代的平均大小是否大于老年代的剩余空间大小，如果大于，则改为Full GC，若小于，怎查看HandlePromotionFailure是否允许担保失败，若允许则只进行Minor GC，若不允许则进行一次Full GC；

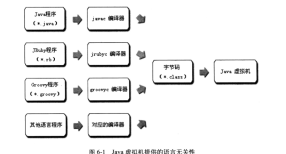
     Minor GC：新生代GC；

     Major GC（Full GC）：老年代GC；比新生代GC慢10倍

三、虚拟机性能监控与故障处理工具

四、调优案例分析与实战

五、类文件结构

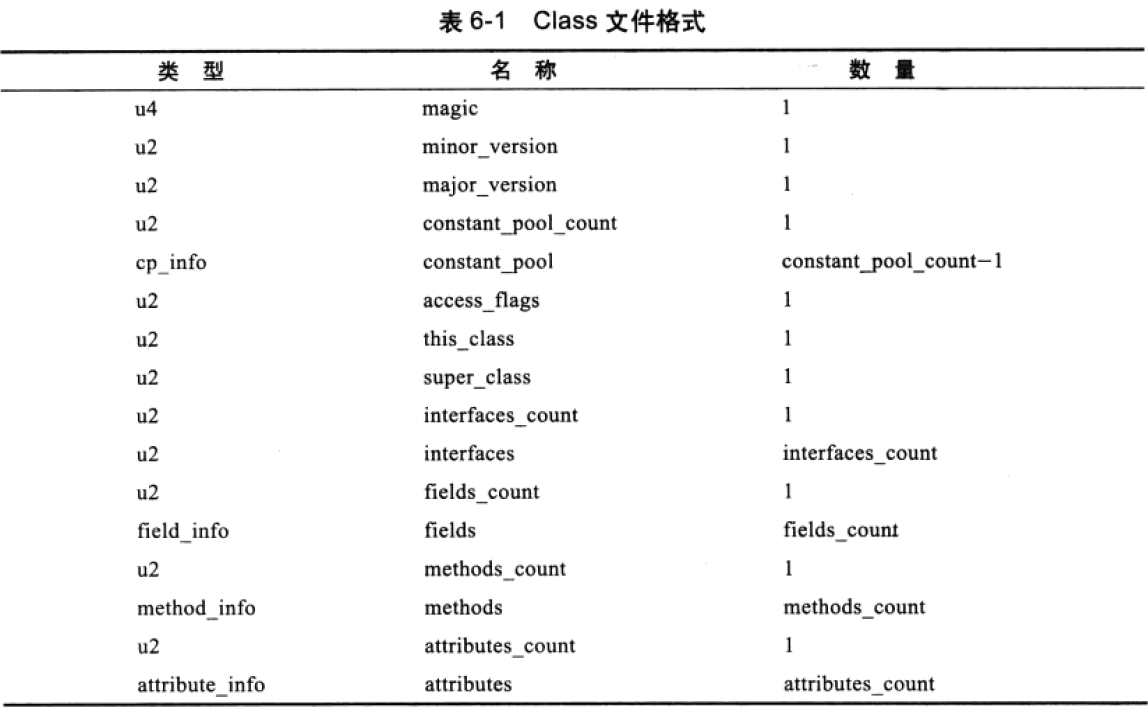


1、class类文件结构：

    （1）Class文件是一组以8位字节为基础单位的二进制流（当遇到需要占用8位字节以上空间的数据项时则会按照高位在前的方式分割成若干个8位字节进行存储）。Class文件格式采用一种类似C语言结构体的伪结构来存储，只有两种类型：无符号数和表。

             无符号数：属于基本的数据类型，u1,u2,u4,u8分别代表1个字节，2个字节，4个字节，8个字节的无符号数，无符号数可以用来描述数字、索引引用、数量值、按UTF-8编码的字符串值；

             表：由多个无符号数或其他表作为数据项构成的复合数据类型（“\_info”结尾）



     1）魔数与Class文件版本：

          每个Class文件的头4个字节称为魔数，唯一用途就是用于确定这个是否是一个能被虚拟机接受的class文件（很多文件标准比如gif，jpeg等都用魔数识别），0xCAFEBABE

          紧接着魔数的4个字节是class的版本号，第5、6个字节是次版本号，第7、8个字节是主版本号。（jdk1.7的为51.0）

     2）常量池：

          紧接着版本号之后的就是常量池入口（表类型数据项目）。常量池入口处放置一项u2类型数据，代表常量池容量计数值（唯一从1开始的，因为0用来表示“不引用任何一个常量池项目”）。

          主要存放：字面量和符号引用

                           字面量：文本字符串、声明为final的常量值等；

                           符号引用：类和接口的全限定名； 字段的名称和描述符； 方法的名称和描述符；

          常量池中每一项常量都是一个表，共有11种结构各不相同的表结构数据（开始的第一位是一个u1类型的标志位tag）



          UTF-8缩略编码：'\u0001'->'\u007f'用一个字节表示

                                   '\u0080'->'\u07ff'用两个字节表示

                                   '\u0800'->'\uffff'用三个字节表示（普通UTF-8）

     3)访问标志：常量池之后紧接着的2个字节。用来识别：这个Class是类还是接口，是否定义为public类型，是否定义为abstract类型，如果是类的话是否被声明为final等等；

     4）类索引、父类索引与接口索引集合：

        类索引与父类索引都是一个u2类型的数据，而接口索引是一组u2类型的数据的集合，Class文件中由这三项数据来确定这个类的继承关系。类索引用于确定这个类的全限定名，父类索引用于确定这个类的父类的全限定名，由于Java不允许多继承，所以父类索引只有一个（除了Java.lang.Object之外，所有Java类的父类索引都不为0）；

        接口索引入口第一项u2类型的数据为接口计数器，表示索引表的容量

     5）字段表集合：用于描述接口或类中声明的变量。包括类级变量、实例级变量但不包括方法内部声明的变量；info：字段作用域（public，private，protected）、类级还是实例级、可变现（final）、并发可见性（volatile，是否强制从主内存读写）、是否可序列化（transient）、字段数据类型（基本类型、对象、数组）、字段名称。

          全限定名：org.fenixsoft.clazz.TestClass   -> org/fenixsoft/clazz/TestClass

          简单名称：方法inc()  -> inc   字段m -> m

           对象：用L加对象的全限定名表示（Ljava/lang/Object;）

           数组：int[]  -> [I,  String[] -> [[Ljava/lang/String;

     6)方法表集合：与字段表类似，依次包括了访问标志、名称索引、描述符索引、属性表集合等（volatile和transient不能修饰方法，所以没有ACC\_VOLATILE, ACC\_TRANSIENT标志，增加了synchronize，native，strictfp，和abstract关键字的标志）

     7）属性表集合：限制稍微宽松了点，不要求各个属性表具有严格的顺序，并且只要不与已有的属性名重复。（应该是为了方便jvm内存对齐吧）

          Code属性：

          Exceptions属性：

          LineNumbertable属性：

          LocalVariableTable属性：

          SourceFile属性：

          ConstantValue属性：

          InnerClasses属性：

          Deprecated及Synthetic属性：

六、虚拟机类加载机制：

**加载**->**验证**->**准备**->解析->**初始化**->使用->**卸载**

加载：虚拟机规范没有强制约束什么时候进行；

验证、准备

初始化：遇到new,getstatic,putstatic,invokestatic这4条字节码时如果类没有初始化则先进行初始化；

             使用java.lang.reflect包的方法对类进行反射调用时，若还没初始化则进行初始化；

             当初始化一个类时其父类还没初始化，则先触发其父类的初始化；(接口的初始化不要求)

             当虚拟机启动时，用户需要指定一个要执行的主类（main()），虚拟机会先初始化这个主类；

    1）对于静态字段，只有直接定义这个字段的类才会被初始化

         public class SuperClass{

             static{system.out.println("Super class init!");}

             public static int value = 123;

         }

         public class SubClass extends SuperClass{

             static{system.out.println("sub class init!");}

         }

        调用SubClass.value时只会输出"Super class init!"

        调用SuperClass[] sc = new SuperClass[10];时没有输出"Super class init!",说明没有触发SuperClass的初始化，但是触发了一个[Lorg.fenixsoft.classloading.SuperClass类的初始化（虚拟机自动生成的，直接继承于java.lang.Object，创建动作由字节码指令newarray触发，这个类代表了类型为SuperClass的一维数组，数组应有的属性和方法都实现在这个类里）

        public class ConstClass{

            static{system.out.println("ConstClass init!");}

            public static final String HELLOWORLD = "hello world";

        }

        当调用ConstClass.HELLOWORLD时不会输出"ConstClass init!",因为在编译阶段会将此常量的值“hello world!”存储到了调用处的那个类的常量池中。

1、加载过程：

    （1）加载：

            1）通过一个类的全限定名获取此类的二进制字节流；（不一定从class文件，也可以从zip，jar，ear，war，网络，文件，数据库等，也可动态生成->动态代理技术）

            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Java堆中生成一个代表这个类的java.lang.Class对象，作为方法区这些数据的访问入口；

    （2）验证：是连接阶段的第一步，为了确保Class文件的字节流包含信息符合虚拟机要求，不会危害虚拟机自身安全（class文件不一定是Java源码编译来的，也可其他方式比如16进制编辑器写）

           具体验证过程JVM规范没有规定，大致上：

            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虚拟机处理范围之内；

                                        常量池中的常量是否有不被支持的常量类型（检测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者不符合类型的常量；

                                        CONSTANT\_Utf8\_info型的常量中是否有不符合UTF8编码的数据；

                                        Class文件中各个部分或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：这个类是否有父类（除Object之外其他的都该有）；

                                     这个类的父类是否集成了不允许集成的类（final修饰的类）；

                                     这个类如果不是抽象类，是否实现了其父类或接口中要求实现的所有的方法；

                                     类中的字段、方法是否与父类产生了矛盾（覆盖父类的final字段，出现不合规则的方法重载：例如方法参数都一样，但返回值类型却不一样）；

            3）字节码验证：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（例如不会出现在栈中放了一个int，使用时却按long类型来加载到本地变量表中）；

                                     保证跳转指令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是有效的（例如把一个子类对象赋给一个父类数据类型是安全的，把父类的赋给子类或者其他类型则是危险的不合法的）；

                     数据流验证的高复杂性-->优化：给方法体的Code属性属性表中增加了一项名为“StackMapTable”属性，此属性描述了方法体中所有基本块开始时本地变量表和操作栈应有的状态，从而将字节码验证的类型推导转变为类型检查从而节省时间；

            4）符号引用验证：发生在解析阶段中，在将符号引用转化为直接引用的时候。

                                     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、字段和方法的访问性（private、protected、public、default）是否可被当前类访问；

     （3）准备（正式为类变量分配内存并设置类变量初始值阶段：类变量而不是实例变量（比如int a=123,是设为0而不是123的阶段），特殊情况：若类字段的字段属性表中存在ConstantValue属性，则直接初始化为ConstantValue属性的值，例如public static final int a=123;直接初始化为123）

     （4）解析：将常量池中的符号引用替换为直接引用的过程；

                        **符号引用：以一组符号来描述所引用的目标，符号可以是任何形式的字面量，与内存布局无关，不一定加载到内存中了；**

**直接引用：可以是直接指向目标的指针、相对偏移量或是一个能间接定位到目标的句柄，与内存布局相关，已经加载到了内存；**

             1）类或接口解析：假设当前代码类为D，要把符号引用N解析为一个类或接口C的直接引用：

                                         a、若C不是数组类型，JVM会把代表N的全限定名传递给D的类加载器去加载这个类C；

                                         b、若C是数组类型并且数组元素类型为对象，则会按照a加载数组元素类型；

                                         c、之后进行符号引用验证，确认C是否具备对D访问权限；

             2）字段解析：先解析出类方法表的class\_index项中索引的方法所属的类或接口的符号引用。假设这个字段所属的类或接口为C

                                  a、若C本身就包含了简单名称和字段描述都与目标相匹配的字段，则返回这个字段的直接引用，查着结束；

                                  b、若C中实现了接口，则从上到下递归搜索各个接口和它的父接口，若接口中包含了简单名称和字段描述都与目标相匹配的的字段，则返回这个字段的直接引用，查着结束；

                                  c、若C不是Object将按照从上到下递归搜索其父类，若在父类中含了简单名称和字段描述都与目标相匹配的的字段，则返回这个字段的直接引用，查着结束；

             3）类方法解析：先解析出类方法表的class\_index项中索引的方法所属的类或接口的符号引用。假设这个字段所属的类为C

                                  a、类方法和接口方法符号引用的常量类型定义是分开的，若在类方法表中发现class\_index中索引的C是个接口直接抛异常；

                                  b、在类C中查着是否有简单名称和描述符都与目标相匹配的方法，若有则返回这个方法的直接引用；

                                  c、否则在类C的父类中递归查找。。

                                  d、否则在类C实现的接口列表及他们的父接口列表中递归查找。。。

             4）接口方法解析：与上面差不多，只不过是查找接口

                                  a、类方法和接口方法符号引用的常量类型定义是分开的，若在类方法表中发现class\_index中索引的C是个类直接抛异常；

                                  b、在接口C中查着是否有简单名称和描述符都与目标相匹配的方法，若有则返回这个方法的直接引用；

                                  c、否则在接口C的父接口中递归查找。。

       （5）初始化：。。

2、类加载器

     两个类相同是指类和类加载器都相同。

     1、双亲委派模型

         （1）启动类加载器：加载JAVA\_HOME/lib中的类。无法被Java程序直接引用；

         （2）扩展类加载器：加载JAVA\_HOME/lib/ext中的类。开发者可以使用；

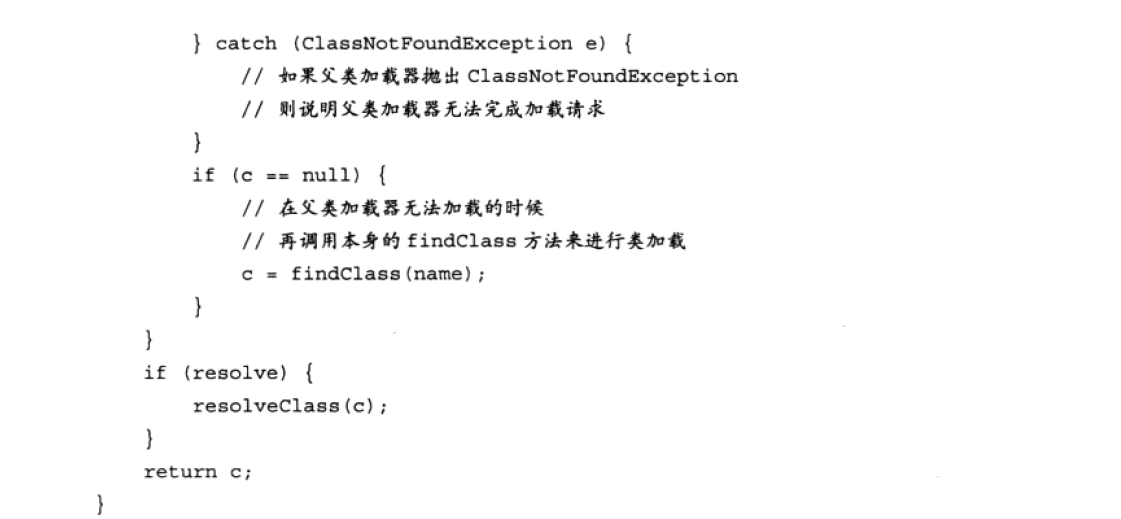
         （3）应用程序类加载器（系统类加载器）：ClassLoader中的getSystemClassLoader()方法的返回值，负责加载用户类路径上所指定的类，开发者可以直接使用，是默认的加载器。



      除顶层的启动类加载器之外其余加载器都有自己的父加载器（不是继承方式实现，而是组合方式实现的），若一个类加载器收到了类加载请求首先会委派给父类加载器去完成，只有当父加载器无法完成时（搜索自己的范围没找到这个类），子加载器才会去加载。

      实现：实现双亲委派的code都在**java.lang.ClassLoader的loadClass()**方法中，先检查是否已经加载过，若没有加载则调用父加载器的loadClass()方法，若父加载器为空则默认使用启动类加载器作为父加载器，若父加载器加载失败则在抛出ClassNotFoundException之后再调用自己的findClass()方法进行加载。





2、破坏双亲委派模型

七、编译优化：

1、方法内联：去除方法调用成本，如建立栈帧、入栈、出栈等；

2、公共字表达式消除：计算过的表达式不再重复计算（中间没有变量被改）；

3、数组范围检查消除；

4、逃逸分析：当一个对象在方法里被定义之后可能被外部方法所引用

         方法逃逸

         线程逃逸

    如果分析一个对象不会发生逃逸的话可以进行一些优化：

    （1）栈上分配

    （2）同步消除

    （3）标量替换：标量比如int long ref等不能进一步分解的，聚合量是指可分解的比如对象；

                             直接访问或创建需要的某些成员变量，而不是创建整个对象；

八、锁

    （1）自旋锁

    （2）锁粗化

    （3）轻量级锁

    （4）偏向锁