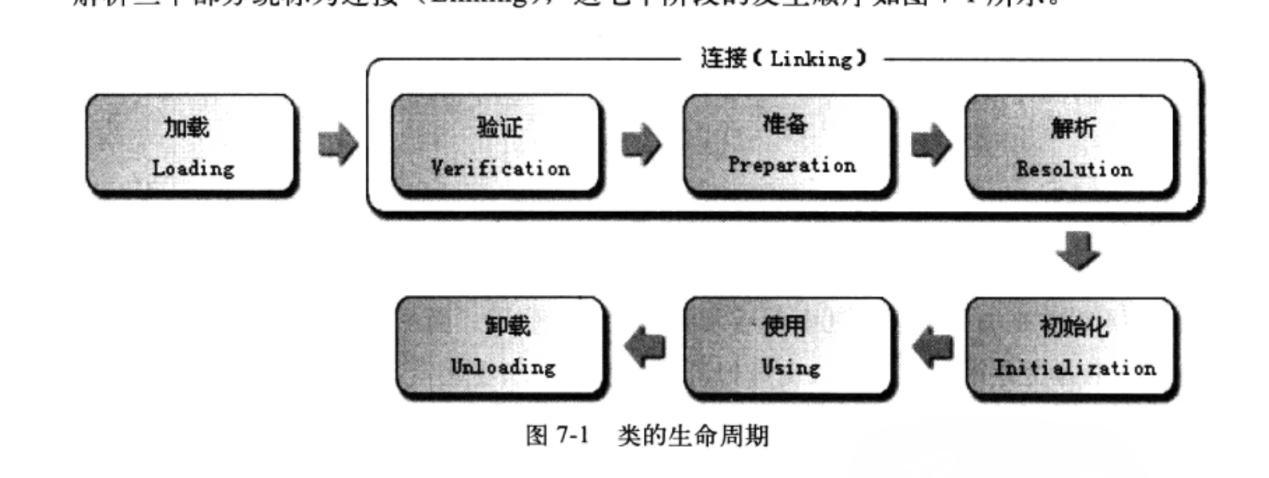
# 第六章--类文件结构

# 第七章--虚拟机类加载机制

类的加载和连接过程是在程序运行期间完成的。Java的动态扩展性就是依赖他的动态加载和动态连接特点。

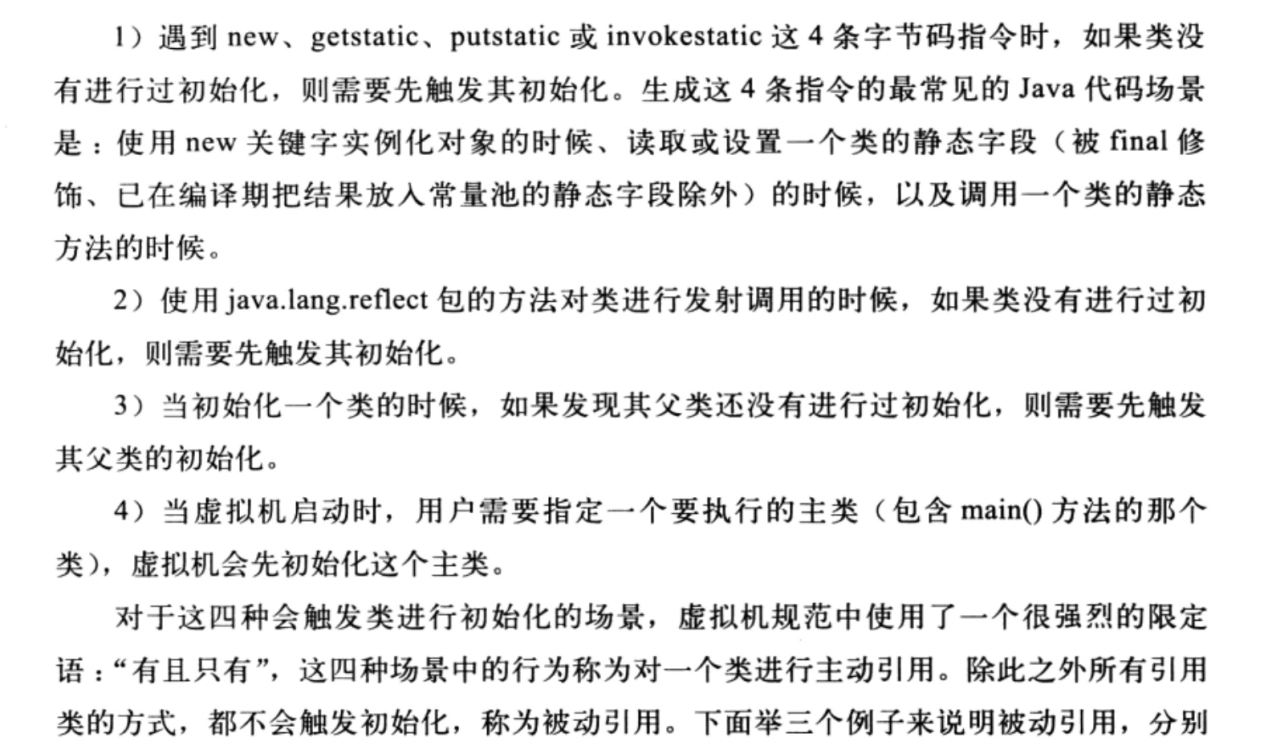
## 类加载时机



加载，验证，准备，初始化和卸载是严格按照这个顺序的。

而解析就不一定了，也可以在初始化之后再开始，这是为了动态绑定而设计的。

### 初始化条件



### 被动引用例子

例子一：

先定义两个类，父类和子类。父类中有一个静态常量，当通过子类访问这个静态常量时，只有父类的静态代码块会执行，其他的非静态代码块和构造方法都不会执行。

如果子类中也有一个静态常量，访问这个子类中的静态常量时，父类和子类的静态方法都会执行。无论怎么样，父类的静态方法一定会执行的。

例子二

通过数组定义来引用类，不会触发这个类的初始化。

User[] user = New User[10];

例子三

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，因此不会触发定义常量的类初始化。

Public static final String HELLO = “hello”;

访问这个变量不会触发该类的初始化。

### 接口初始化说明

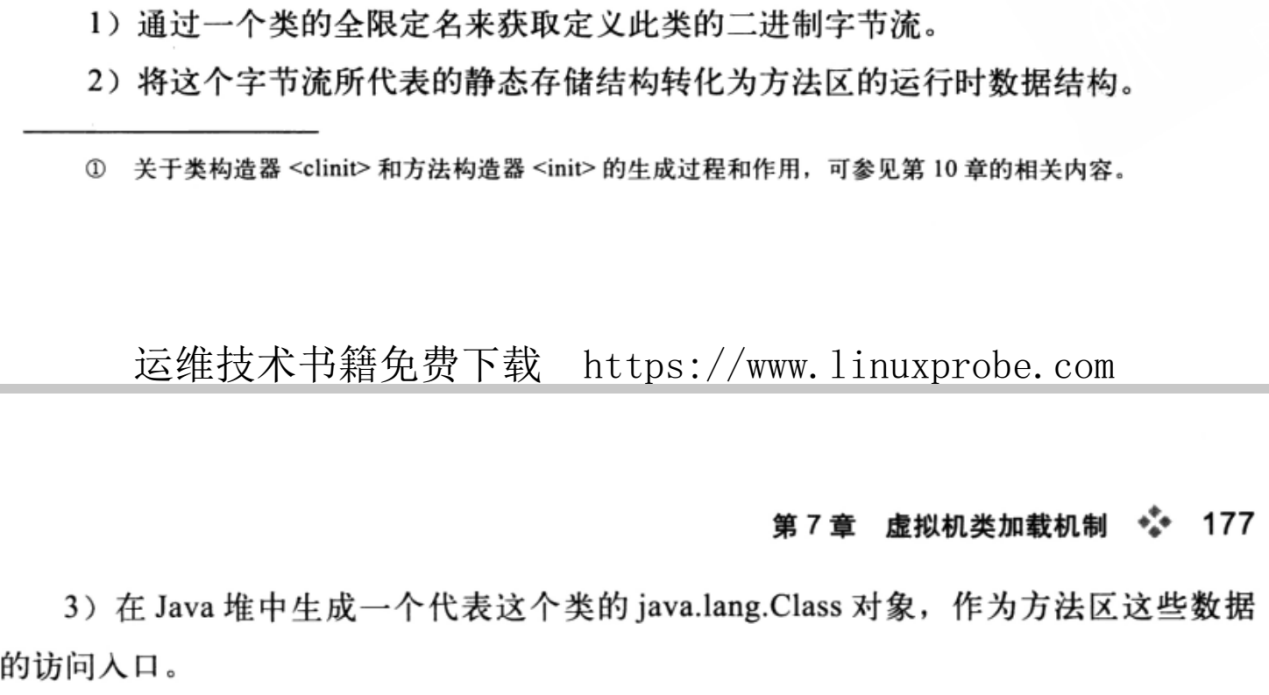
当一个类初始化的时候，必须要先初始化他的父类。但是接口不需要这样，只有使用到父接口的时候(比如引用接口中定义的常量)才会初始化。

### 类加载的全过程

：加载，验证，准备，解析，初始化--红色的是连接阶段

#### 加载

加载是类加载过程的一个阶段，



加载阶段（加载阶段获取类二进制字节流的阶段）是开发期可控性最强的阶段，因为加载阶段可以使用系统提供的类加载器，也可以使用自定义的类加载器。

加载阶段完成后，虚拟机外部的二进制流就会按照虚拟机所需的格式存放到方法区中。然后再堆中实例化一个java.lang.Class类的对象。

加载阶段与连接阶段的部分内容是交叉进行的。

#### 验证

验证是连接阶段的第一步，目的是确定class文件中的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。

文件格式验证、元数据验证、字节码验证、符号引用验证、

#### 准备

为类变量分配内存，并设置类变量初始值。这些内存都是在方法区进行分配。

这个时候分配内存仅仅指的是类变量（被static修饰的变量），而不包括实例变量。

Public static int a = 132；那么在准备阶段分配的初始值零值（这里是0），而不是132。

而将值赋为132的putstatic指令是在程序编译后，存放在类的构造器<clinit>方法中，赋值是在初始化阶段进行的。

基本数据类型的零值：



#### 解析

虚拟机将常量池中的符号引用替换为直接引用的过程。

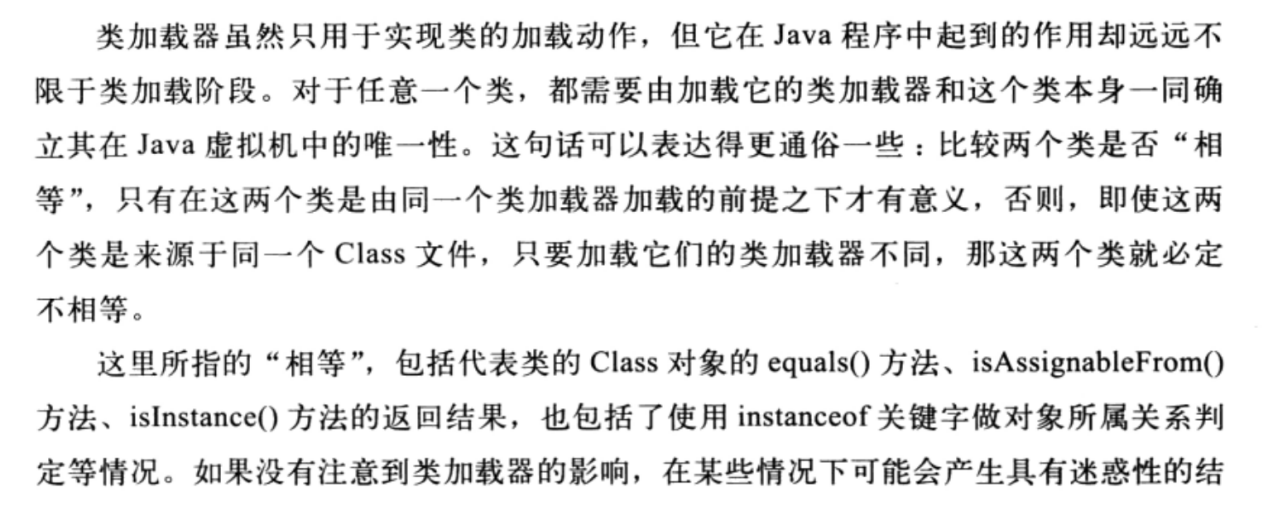
#### 初始化

类加载过程的最后一步，这个时候才真正开始执行类中的程序代码。

<clinit>()方法。在多线程下会正确的加锁和同步。

## 类加载器

类加载阶段的“通过一个全限定名来获取该类的二进制字节流”这个动作放到java虚拟机外部去实现。以便让程序自己决定如何去获取所需要的类，实现这个动作的代码就是类加载器。



### 加载器划分

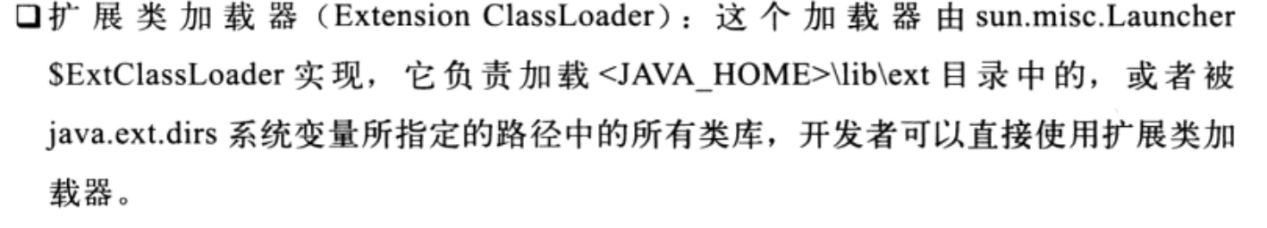
#### 启动类加载器

Bootstrap ClassLoader

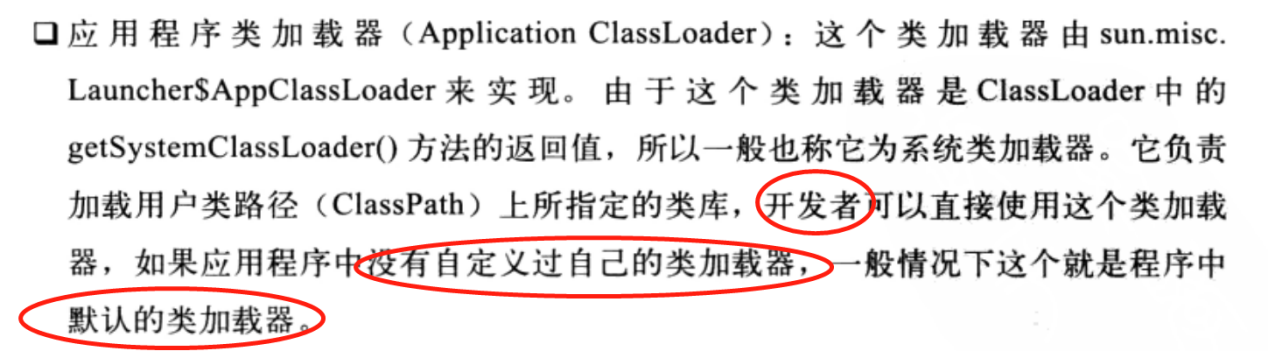
就是将<JAVA\_HOME>\lib目录中的或者是-XBootclasspath参数所指定的路径中的，并且是java虚拟机所识别的（仅按照文件名识别）类库加载到虚拟机中。启动类加载器无法被java应用程序直接引用。

#### 扩展类加载器

Extension Classloader

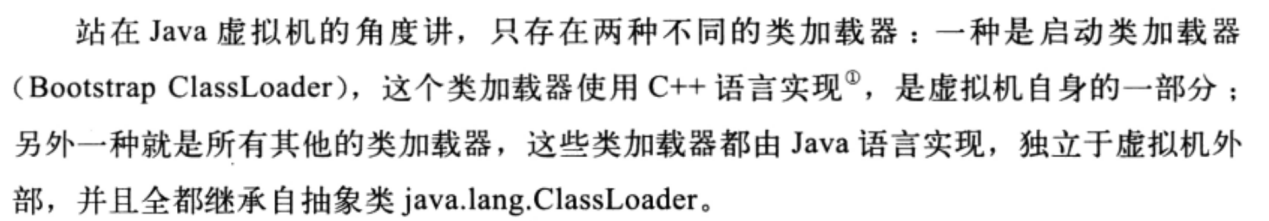


#### 应用程序类加载器



#### 自定义类加载器

### 双亲委派模型

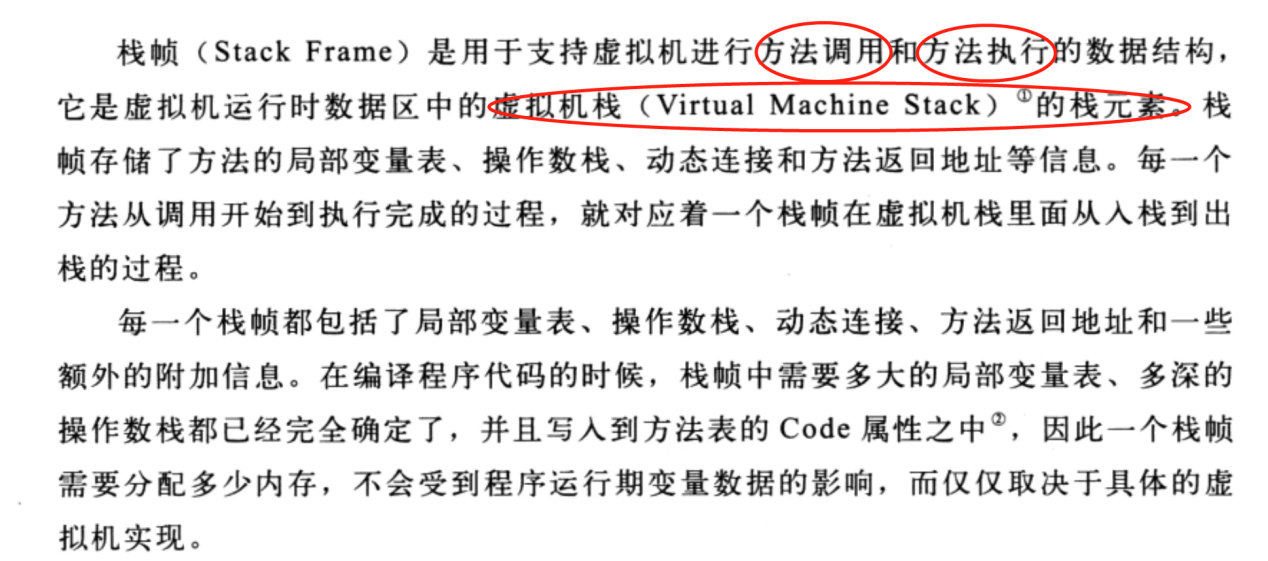


当一个类加载器收到加载请求时，首先不会自己去加载这个类，而是委派给父类加载器去完成加载，如果父类加载器加载失败，自己才会加载。

### 破坏双亲委派模型

# 第八章--虚拟机字节码执行引擎-没读懂

## 运行时栈帧结构

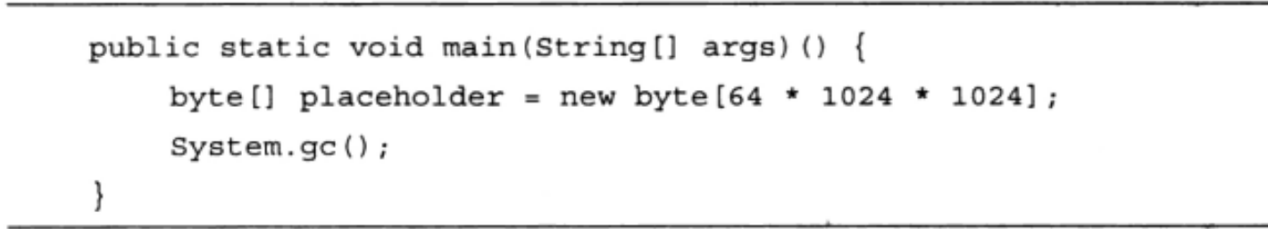


### 局部变量表

是一组变量值存储空间，用于存放方法参数和方法内部定义的局部变量。

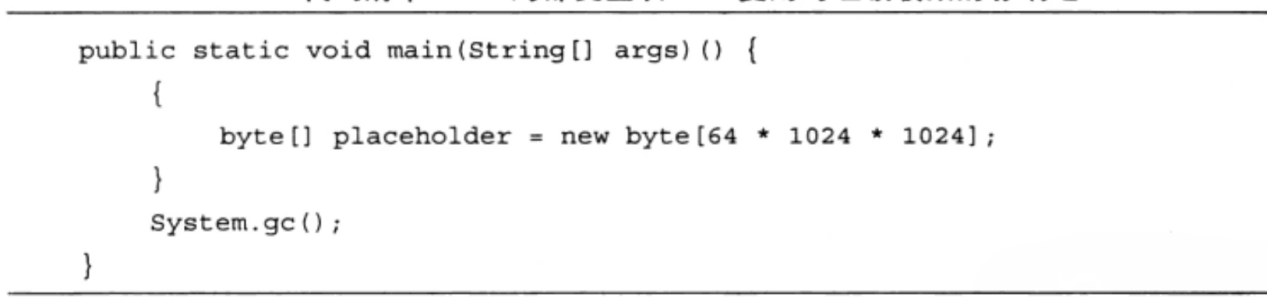
编译阶段，确定了该方法所需要分配的最大局部变量表的容量。

1



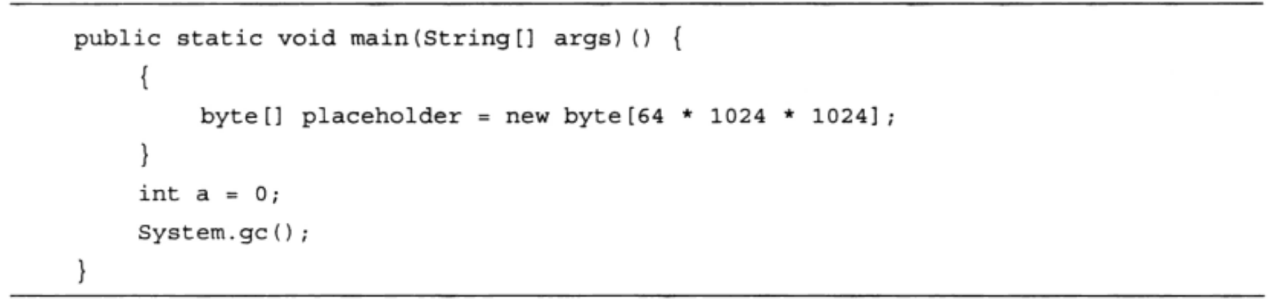
并没有回收这64M的内存，这个变量还处在作用域之内，虚拟机自然不会回收这部分内存。

2



从代码逻辑上看，这个变量不可能再被访问，但是还是没有回收掉。

3



这个时候就被回收了，问什么？

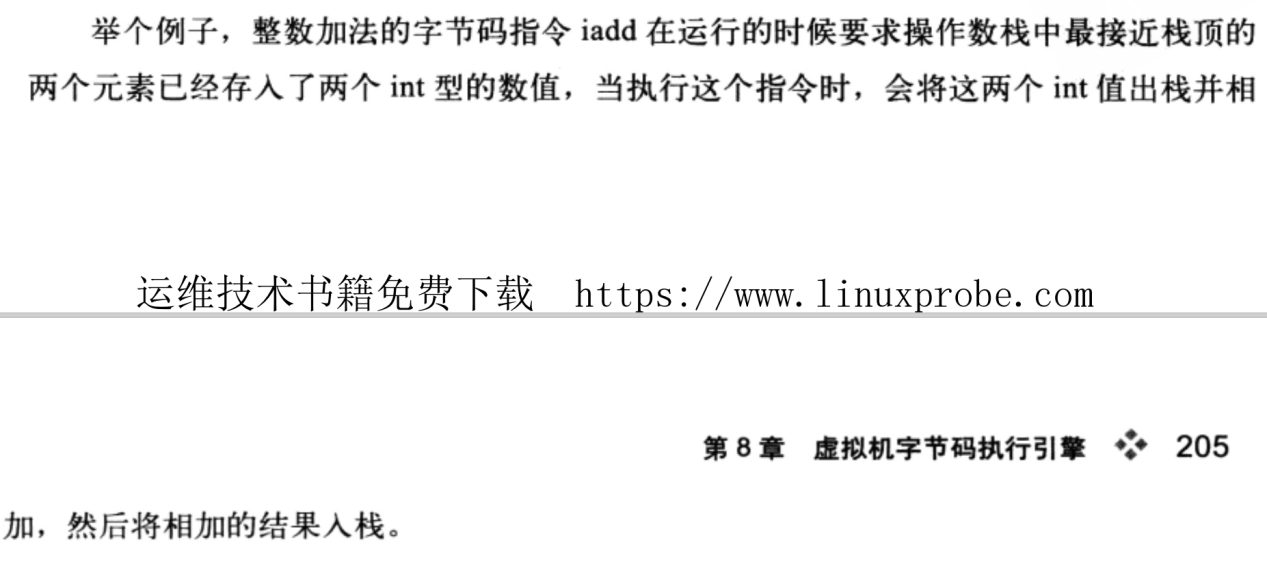
这个变量能不能被回收的根本原因是：局部变量表中的slot是否还存有这个对象的引用，2中代码虽然离开了这个变量的作用域，但在此之后，没有任何对局部变量表的读写操作，这个变量原本占用的slot还没有被其他变量所复用，所以作为GC Root一部分的局部变量表仍然保持着对他的关联。

### 操作数栈

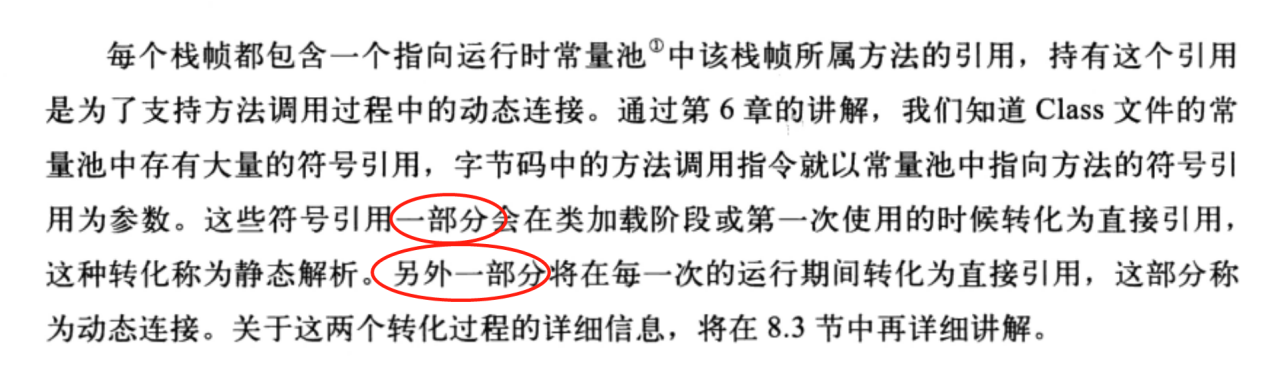
也被称为操作栈。

同局部变量表一样，操作数栈的最大深度也在编译的时候被写入到code属性的max\_stacks数据项中。

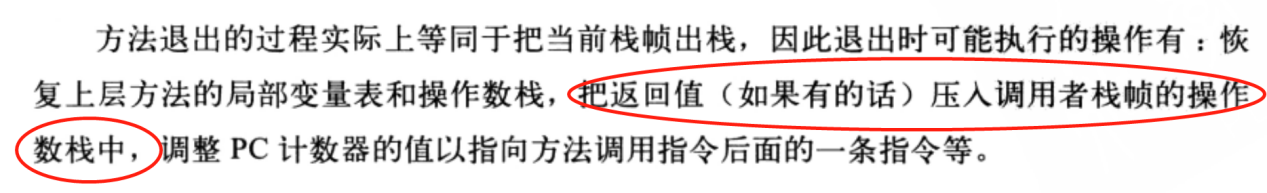
操作数栈的每一个元素可以是任意的java数据类型，包括long和double。32为数据类型所占的栈容量是1,64位占2。



### 动态连接



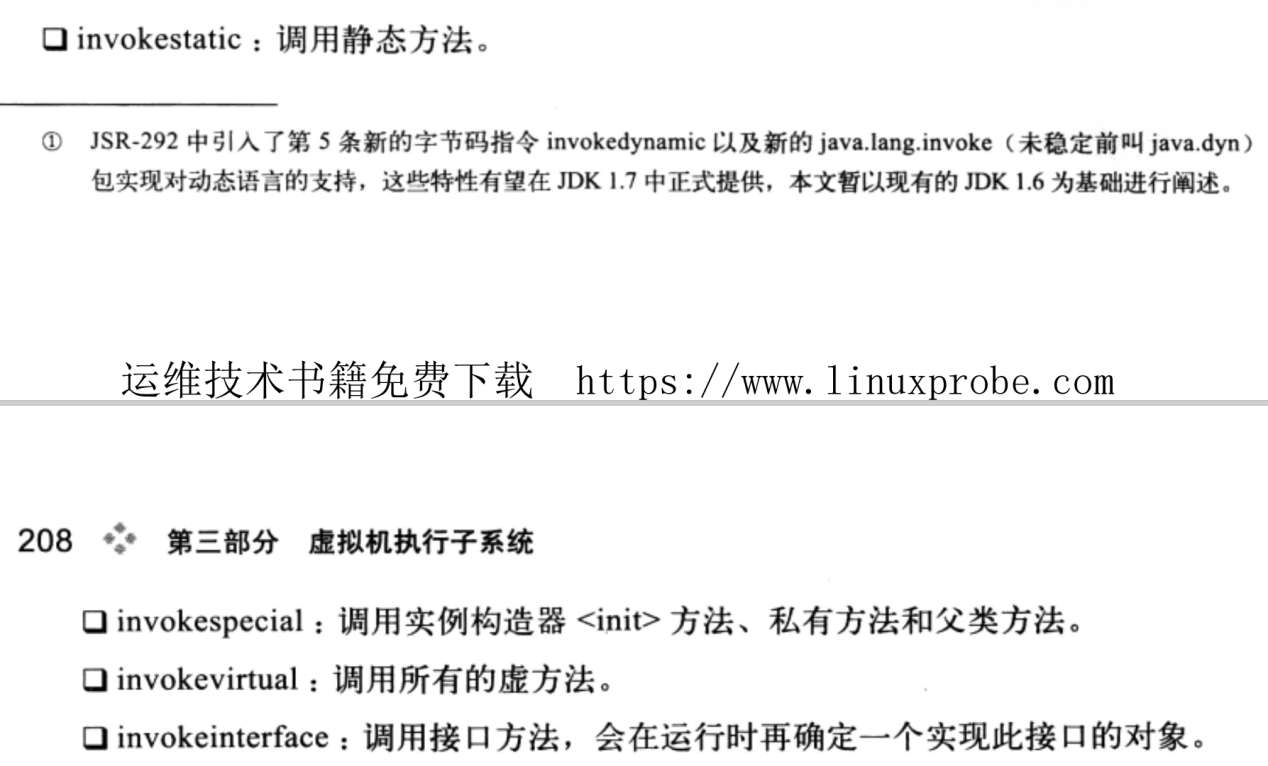
### 方法返回地址



### 附加信息

一般会把动态连接，方法返回地址，与附加信息全部归为一类，称为栈帧信息。

## 方法调用



### 解析

只要能被invokestatic和invokespecial指令调用的方法，都可以在解析阶段确定唯一的调用版本。符合这个条件的方法有：

静态方法

私有方法

实例构造器

父类方法

他们在类加载的时候就会把符号引用解析为该方法的直接引用。这些方法称为非虚方法。

另外final方法也被称为非虚方法。虽然他是被invokevirtual指令来调用的，但是他无法被覆盖。

其他方法都是虚方法。

解析调用是一个静态的过程，在编译期间就完全确定。

### 分派

#### 静态分派--方法重载

Human man = new Man();其中Man继承Human

Human称为静态类型或外观类型，Man为实际类型

静态类型和实际类型在程序中都会发生一些变化，区别是静态类型的变化仅仅在使用时发生，变量本身的静态类型不会被改变，并且最终的静态类型是在编译期可知的；而实际类型变化的结果在运行期才可确定，编译器在编译程序的时候并不知道一个对象的实际类型是什么。

虚拟机在重载时通过参数的静态类型作为判断依据。静态类型在编译器是可知的，所以在编译阶段编译器就会根据参数的静态类型决定使用哪个重载版本。

所有依赖静态类型而定位方法执行版本的分派动作，都称为静态分派。静态分派的最典型应用就是方法重载。

在选择重载方法时，编译器会选择一个更加合适的方法版本，例如参数是一个‘a’，方法形参有char，int，long，...，char的包装类型等。有限顺序是char->int->long->float->double->包装类型->包装类型实现的接口->object->可变长参数char...

静态方法会在类加载期就进行解析，而静态方法显然也是可以拥有多个重载版本的，选择重载版本的过程是通过静态分派完成的。

#### 动态分派--方法重写

动态分派和多态性的另外一个重要体现“重写”有很密切的联系。

Human man = new Man();

Human woman = new Woman();

首先是建立man和women的内存空间，调用Man和Woman类型的实例构造器，将这两个实例的引用存放在第一和第二个局部变量表Slot之中。

然后将这两个对象压到栈顶，这两个对象是将要执行的sayHello()方法的所有者，成为接受者(Receiver)；

然后方法的调用指令，这里就要从invokevirtual指令的多态查找过程说起，invokevirtual指令运行时解析过程大致分为以下几步：

找到操作数栈顶的第一个元素所指向的对象的实际类型，记作C

如果在C类型中找到与常量中描述符和简单名称都相符的方法，则进行访问权限校验，如果通过则返回这个方法的直接引用，查找过程结束。不通过则返回java.lang.IllegalAccessError异常。

否则按照继承关系从下往上依次对C的各个父类进行第二步的搜索和验证过程

如果始终没有找到合适的方法，则抛出java.lang,AbstractMethodError异常

由于invokevirtual指令执行的第一步就是在运行期确定接受者的实际类型，所以两次调用中的invokevirtual指令把常量池中的类方法符号引用解析到了不同的直接引用上，这就是重写的本质。把这种在运行期间根据实际类型确定方法执行版本的分派过程成为动态分派。

#### 单分派和多分派

方法的接受者与方法的参数统称为方法的宗量，根据分派基于多少种总量，可以将分派分为单分派和多分派

单分派是根据一个宗量对目标方法进行选择，多分派是根据多于一个的宗量对目标方法进行选择。

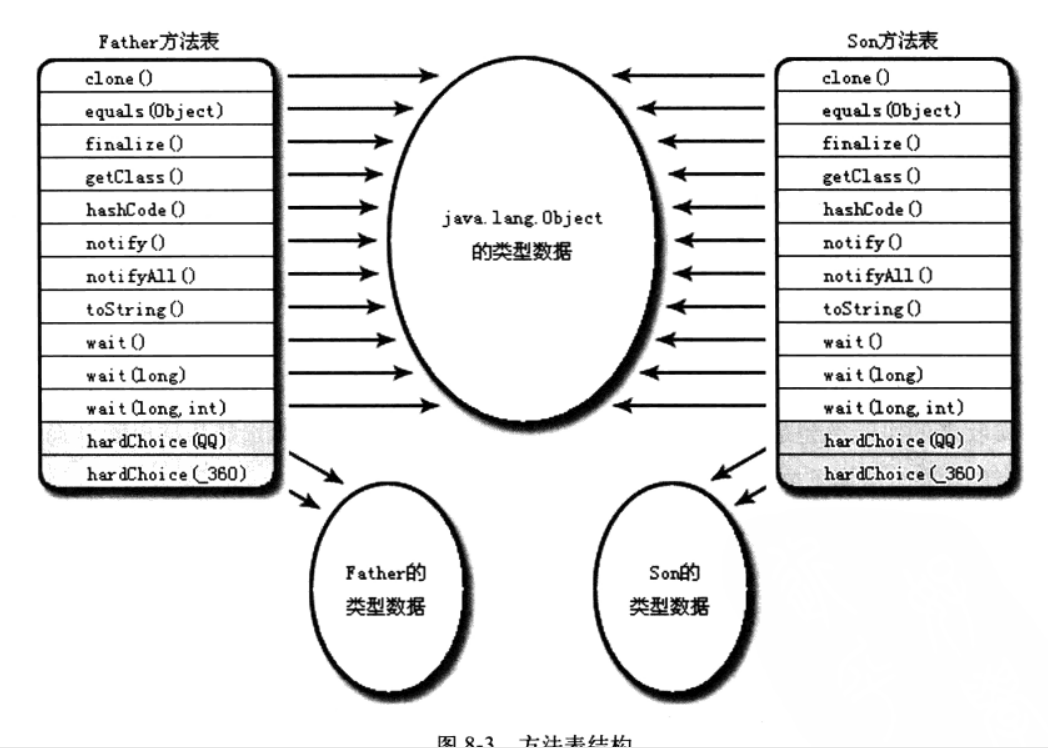
静态多分派

动态单分派

#### 虚拟机动态分派的实现

虚方法表--Vtable的概念，于此对应invokeinterface执行时也会用到接口方法表interface-method-table（itable）,使用虚方法表索引来代替元数据查找以提高性能。

虚方法表结构：



具有相同签名的方法，在父类、子类的虚方法表中都应当具有一样的索引序号，这样当类型变幻时，仅需变更查找的方法表，就可以从不同的虚方法表中按索引转换出所需的入口地址。

方法表一般在类加载的连接阶段进行初始化，准备了类的变量初始值后，虚拟机会把该类的方法表也初始化完成。

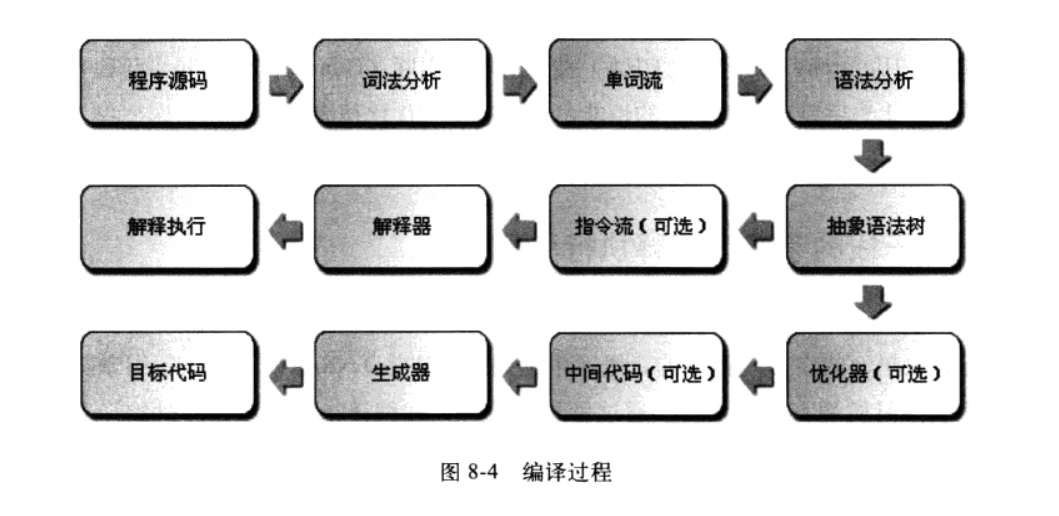
方法表是“稳定优化”手段，虚拟机除了使用方法表之外，在条件允许的情况下，还会使用内联缓存（Inline Cache）和基于“类型继承关系分析”技术的守护内联（Guarded Inline）两种非稳定的“激进优化”手段获取更高的性能，关于这两种优化的技术原理和运作过程，可以参考本书11章的内容。

## 基于栈的字节码解释执行引擎

如何调用方法上面已经说过了，现在开始探讨虚拟机是如何执行方法里面的字节码指令的。

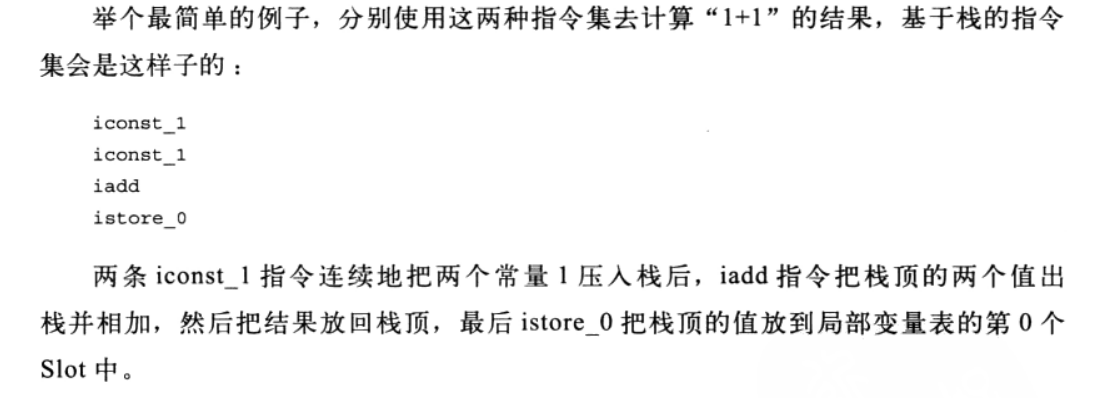
Java虚拟机的执行引擎在执行java代码的时候都有解释执行（解释器执行）和编译执行（通过即时编译器产生本地代码执行）两种选择。

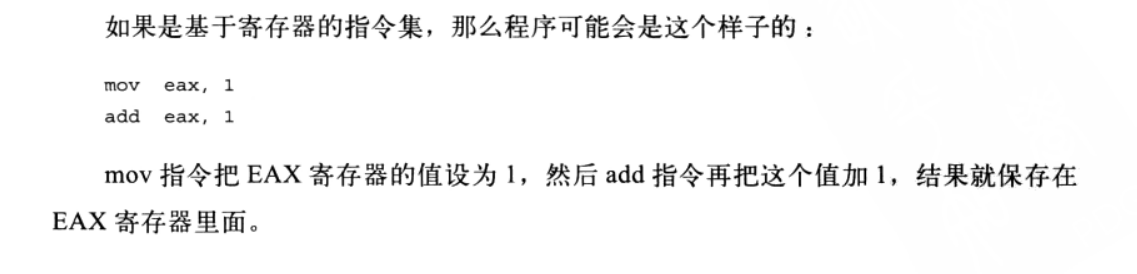
### 解释执行



### 基于栈的指令集与基于寄存器的指令集

Java编译器输出的指令流，基本上是一种基于栈的指令集架构，指令流里面的指令大部分都是零地址指令。他们依赖操作数栈进行工作。





基于栈的指令集优点是可移植性，代码相对紧凑、编译器实现更加简单。

但是执行速度相对来说稍微慢些

基于寄存器的指令集速度快。

如果使用栈架构的指令集，那就可以由虚拟机决定把一些访问最频繁的数据（程序计数器、栈顶缓存等）放到寄存器中以获取更好的性能。

### 基于栈的解释器执行过程

用一段代码进行演示

|  |
| --- |
| **public** **int** calc(){  **int** a = 100;  **int** b = 200;  **int** c = 300;    **return** (a+b)\*c;  } |

Javap执行：

|  |
| --- |
| public int calc();  Code:  0: bipush 100  2: istore\_1  3: sipush 200  6: istore\_2  7: sipush 300  10: istore\_3  11: iload\_1  12: iload\_2  13: iadd  14: iload\_3  15: imul  16: ireturn |

bipush 将100压入栈

istore\_1 将操作数栈顶的整形值出栈并存放到第一个局部变量Slot中

...

iload\_1 将变量表中的第一个Slot中的数值复制到操作数栈顶

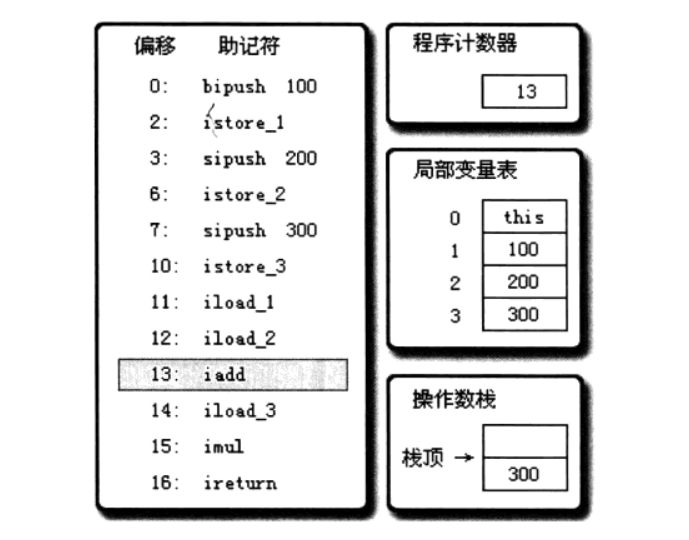
iload\_2 将变量表中的第二个Slot中的数值复制到操作数栈顶

iadd 将栈中前两个数值取出相加得出的结果入栈。

iload\_3 将第三个Slot中的数值复制一份到操作栈顶

imul 将栈中前两个数值出栈相乘得出的结果入栈

ireturn 将栈顶数据返回



这是一种概念模型，虚拟机最终会对执行过程做出一些优化来提高性能。实际的运作过程不一定完全符合概念模型的描述，确切的说实际情况会和上面描述的概念模型有非常大的差距。

虚拟机中的解析器和即时编译器都会对输入的字节码进行优化。

# 第九章--类加载及执行子系统的案例与实战

## 概述

虚拟机直接控制：

class文件以何种格式存储，类型何时加载，如何连接，虚拟机如何执行字节码指令。 这些都是由虚拟机直接控制的行为，用户无法对其进行改变。

通过程序进行操作：

字节码生成和类加载器。

这两点已经出现了许多值得欣赏和借鉴的思路。

## 案例分析

### Tomcat--正统的类加载器架构

主流的web服务器，都实现了自己定义的类加载器（一般都不止一个），web服务器需要解决如下几个问题

1. 部署在同一个服务器上的两个应用所使用的java类库可以实现相互隔离，这是最基本的要求，两个不同的应用程序可能会依赖同一个第三方类库的不同版本，服务器应该保证两个应用程序的类库可以互相独立使用。
2. 部署在同一个服务器上的两个应用所使用的java类库可以互相共享。例如用户可能有10个使用spring组织的应用程序部署在同一台服务器上，如果10份都分别存放在各个应用程序的隔离目录中，将会造成很大的资源浪费--这里指的不是磁盘空间，而是指类库在使用时都要被加载到服务器内存，如果类库不能共享，虚拟机的方法区很容易就会出现过度膨胀的风险。
3. 服务器需要尽可能的保证自身的安全不受部署的web应用程序影响。Web服务器本身就是用java语言实现的，因此服务器本身也有类库依赖的问题，一般来说，基于安全考虑，服务器使用的类库应该与应用程序的类库互相独立。
4. 支持jsp的web服务器，十有八九都需要支持HotSwap功能。Jsp文件最终都要被编译成java class才能被虚拟机执行。

由于存在上述问题，在部署web应用时，单独的一个ClassPath就无法满足需求了，所以各种web服务器都不约而同的提供了好几个ClassPath路径供用户存放第三方类库，这些路径一般都以“lib”或“classes”命名，被放置到不用路径中的类库，具备不同的访问范围和服务对象，通常，每一个目录都会有一个相应的自定义类加载器去加载放置在里面的java类库。

Tomcat目录结构中，有三组目录（“/common/\*”、“/server/\*”、“/shared/\*”）可以存放java类库，另外还可以加上web应用程序自身的目录“/WEB-INF/\*”

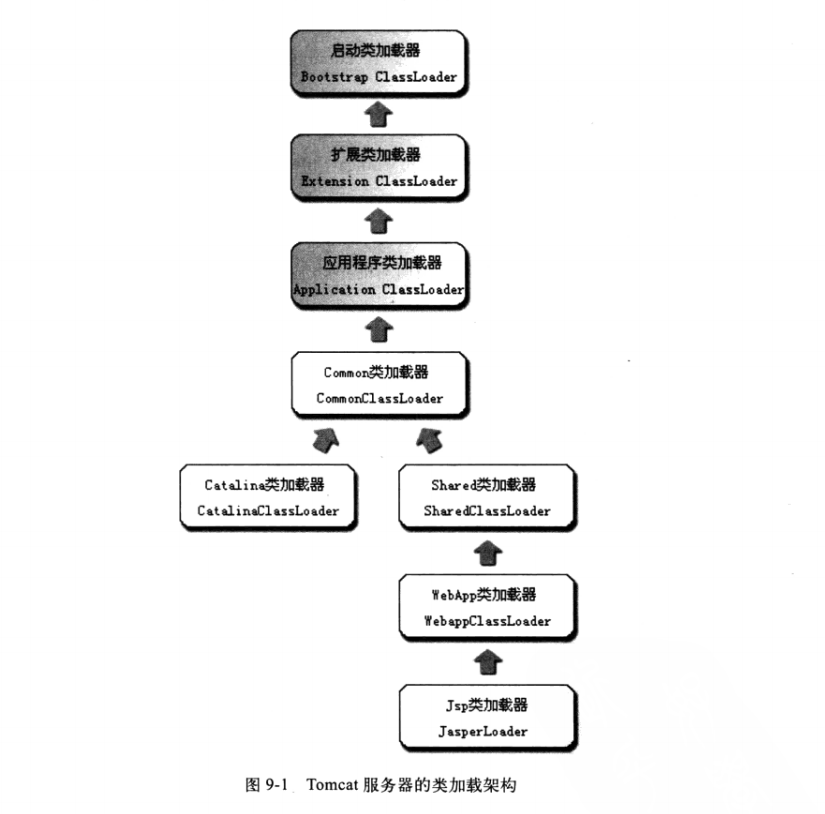
Common：类库可被tomcat和所有的web应用程序共同使用。

Server：可以被tomcat使用，多所有的web应用程序都不可见

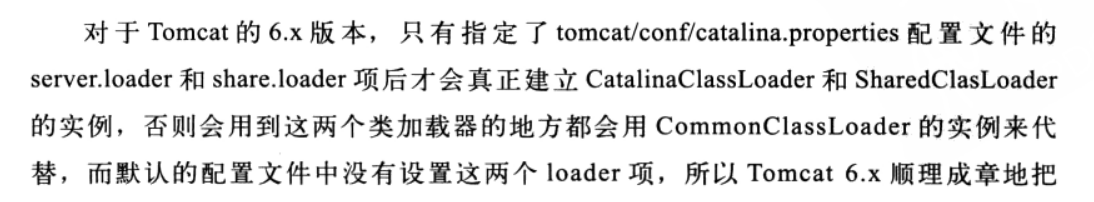
Shared：可被所有的应用程序共同使用，但对tomcat自己不可见

WebApp/WEB-INF：仅仅可以被此web应用程序使用

为了支持这套目录结构，并对目录里面的类库进行加载和隔离，Tomcat自定义了多个类加载器，这些类加载器按照经典的双亲委派模型来实现。



每一个web应用程序对应一个WebApp类加载器，每一个jsp文件对应一个jsp类加载器。

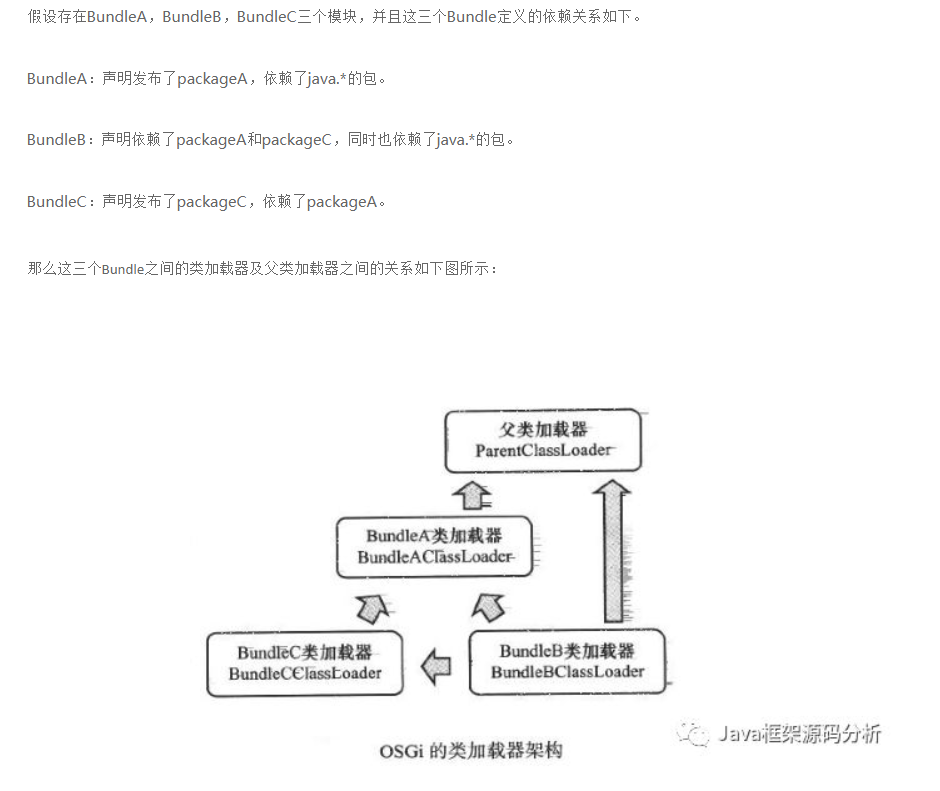


Tomcat 6.x把/common、/server、/shared三个目录默认合并到一起变成一个/lib目录。这个目录里的类库相当于以前/common目录中类库的作用。

### OSGI--灵活的类加载器架构

流传的一个观点“学习JEE规范，去看JBOSS源码；学习类加载器，去看OSGI源码”。

例如：某个Bundle声明了一个它依赖的Package，如果有其它的Bundle声明发布了这个Package，那么所有对这个Package的类加载动作都会委派给发布它的Bundle类加载器取完成。





### 字节码生成技术与动态代理的实现

### Retrotranslator：跨越JDK版本

把jdk1.5中编写的代码放到jdk1.4或者1.3的环境中运行，为了解决这个问题，“Java逆向移植”的工具诞生，Retrotranslator是其中最出色的一个。

JDK的升级中会提供哪些新的功能，新增的功能大致分为四类。

1. 在编译层面所做的改进，如自动装箱，实际上就是编译器在程序中使用到包装对象的地方自动插入了很多Integer.valueOf()和Float.valueOf()之类的代码；变长参数在编译之后就会被自动转换成了一个数组完成参数传递；泛型的信息则在编译阶段就已经被擦除掉了（但是在元数据中还保留着），相应的地方被编译器自动插入了类型转换代码。
2. Java API的代码增强，JDK1.2时代引入了java.util.Collections等一些列的集合类。
3. 需要在字节码中进行支持的改动。如在1.7里面加入的语法特性：动态语言支持，就需要在虚拟机中新增一条invokeddynamic字节码指令来实现相关的调用功能。
4. 虚拟机内部的改进。重新定义了内存模型，CMS收集器之类的改动。

Retrotranslator只能模拟前两类，第二类模拟相对更容易一些，