# Java内存区域与内存溢出异常

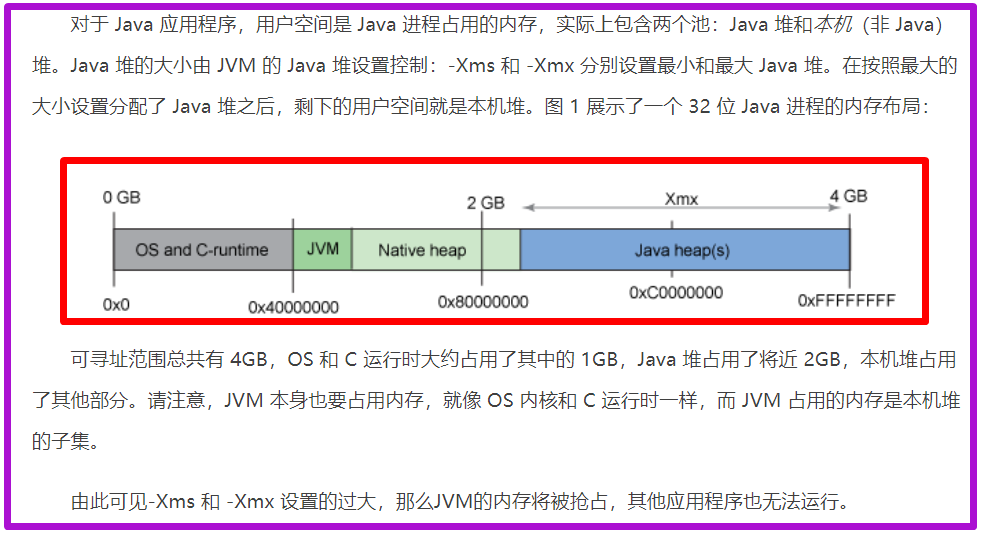
### JVM参见参数解析

-Xms512m: JVM初始分配的堆内存

-Xmx512m: JVM最大允许分配的堆内存，按需分配

**-XX:PermSize=64M JVM初始分配的非堆内存64M  
 -XX:MaxPermSize=128M JVM最大允许分配的非堆内存128M，按需分配**

### JVM内存分布解析



需要将-Xmx的值设置小一点,如-Xmx512;

# 虚拟机类加载机制

## 概述

### 什么是类加载机制

（1）类加载机制的定义

虚拟机把描述类的数据，从class文件加载到内存，通过校验、转换解析、初始化，转化为java虚拟机可以直接使用java类型，这个过程为类的加载机制。

（2）实际的应用

正是因为有了类的加载机制，各种用java语言写程序，最后发布只会发布编译好的class文件，出于保密考虑，常不会提供源码，但是因为有了类加载机制，class文件中已经有关于类的描述信息，只需要经过校验、转换解析、初始化等操作，即可得到jvm可以直接使用的java类，所以只需有class文件，因为有了类的加载机制，也可以正常运行。

（3）类加载机制和类加载器的区别

类加载器和类加载机制的区别是，类加载器只是实现从类的全限定名得到该类的二进制字节流这样的一个代码模块，是小范围的功能点；而类加载机制是：将描述该类的数据，从class文件中加载到内存中，经过校验、转化解析、初始化等过程，最终得到jvm可以直接使用的java类的这样一个整个流程，是一个大的范围，是机制，是大范围的概述。

### Java可动态扩展的原因

1. 原因

类的加载和连接过程都是在程序运行期间完成的;

1. 优缺点

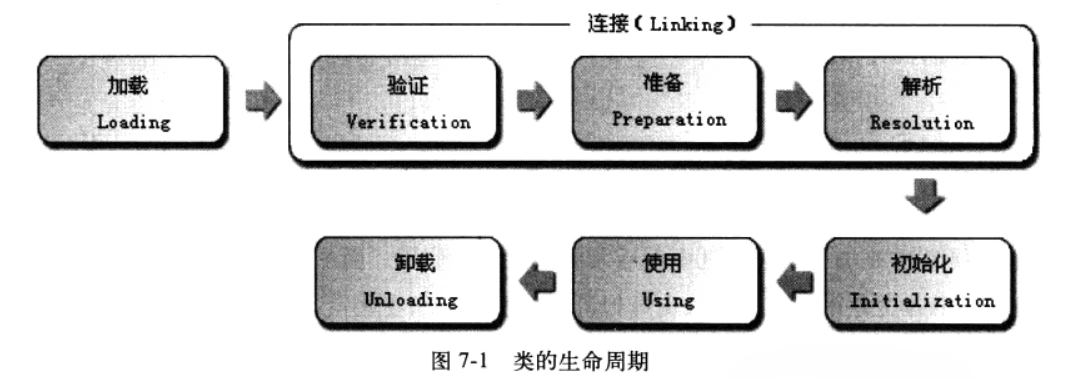
该特点在类加载时，稍微会增加一些性能开销，但是却能为java应用程序提供高度的灵活性。

1. 应用范举

编写一个使用接口的应用程序，可以等到运行时指定其实际的实现。

## 类加载的时机

### 类加载顺序



1.连接

验证、准备、解析称为连接;

2.顺序性

类的生命周期，从class文件加载到JVM中开始，从卸载出JVM结束，但是如上其他过程的顺序，能够确定的就是：加载->验证-准备->初始化->卸载,但是并不是依次进行，而是可能交叉进行，可能一个过程还没完成，就激活另一个过程。

解析和使用的顺序是没有确定的，java可支持动态绑定、晚期绑定，如上所述。

3.触发初始化的四个条件

1):执行了new、getstatic、putstatic和invokestatic字节码时

场景分别对应new一个对象、得到或设置一个静态变量的值(被final修饰的静态变量不算，因为它们已经在预编译期间，就被放到常量池中了)、执行一个静态方法时;

2):通过反射方法java.lang.reflect包中的方法操作一个类的时候;

3):初始化一个子类时，先初始化父类;

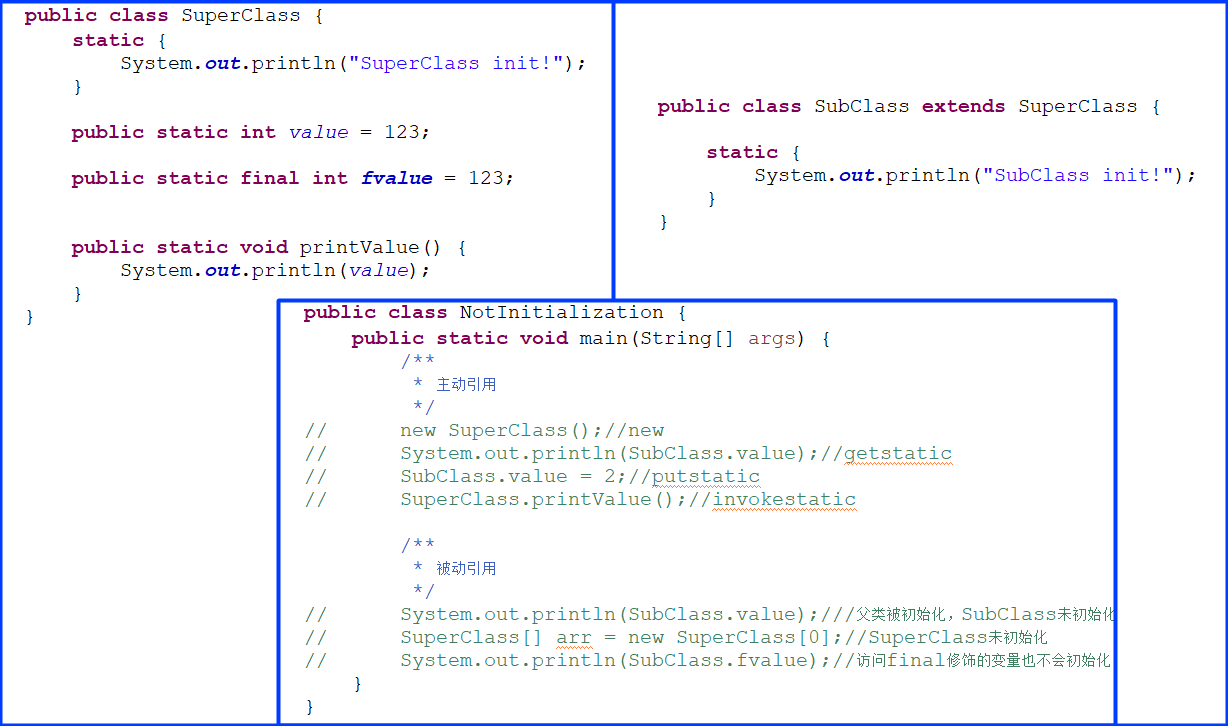
4):当虚拟机启动时，需要指定一个执行的主类,初始化它，一般是包含main方法的那个类;

以上四种情况称为主动引用，被动引用常见的三个例子如下

1):通过子类，调用父类的静态变量,只有父类被初始化了;

2):new一个类的一维数组，类没有初始化;

3):调用一个类的静态、但是被final修饰的变量时，类不会初始化;



### 接口初始化的特殊点

1.接口中不能使用static{}静态代码块;

2.接口不适用触发类初始化条件的第三点，只有当子接口使用到父接口的变量时，才会初始化父接口，它不会生成init<>方法，但会生成clinit<>构造方法。

## 类加载的过程

### 类的加载和加载的区别

类的加载包括加载、验证、准备、解析、初始化，而加载只不过是类的加载中的第一个环节，就是通过类的全限定名得到该类的二进制字节流的整个过程。

### 加载

1、通过类的全限定名获取描述该类的二进制字节流：

1):获取途径:

1]:获取二进制字节流可以多个途径得到，如zip包，如jar包、war包等；

2]:通过网络得到，序列化，反序列化,典型例子就是Applet，安卓应用等;

3]:运行时计算生成二进制字节流,典型例子就是动态代理技术:

4]:其他文件生成，如jsp;

2):实现功能体:

实现它的功能的东西叫类加载器，而且类的加载运行时、支持动态加载的，此 时虽然会耗点性能，但是却能极大地提高灵活性，像代码热部署、OSGi、代码加密 等都是这样的。

3):类的加载可以在JVM外部进行

2、将字节流中的静态数据，转换为JVM方法区中运行时的存储结构;

3、在JVM中创建一个该类的class对象，作为访问方法区中这些数据的入口;

JVM中，加载阶段，准确说是获取二进制字节流阶段，是可控性最强的阶段，既可以使用JVM定义的类加载器，也可以用自定义的；加载得到的二进制字节流是按照JVM所需的格式存储在方法区中的，JVM中的方法区数据存储格式是JVM定义的，但具体的数据结构没有规定;加载和连接阶段是交叉进行的，但是互相固定的顺序又是不变的。

活学活用:

1):Action.class 和 new Action().getClass()相等吗？为什么？

答: 相等;

字节码对象的生成是在类的加载过程中的加载阶段的第三步，在JVM中创建一个该类的字节码对象，作为访问方法区中，该类的静态存储数据的访问入口。

Action.class就是从方法区中获取数据静态数据,是加载阶段生成的，new Action()是在JVM中的堆中创建对象，是类加载过程中的 解析、初始化阶段，在加载阶段时，已经生成了该类的字节码对象，所以Action.class和new Action().getClass()都是获取方法区中，在加载过程生成的字节码对象，所以相等。所以他们获取的类加载器也相同。

2):JVM如何判断两个是否相同的标准是什么？

类的全限定名和类加载器；

### 验证

文件格式、元数据、字节码、符号引用四个方面进行验证。

### 准备

该阶段是正是为类变量分配内存并设置类变量初始值的阶段，这些内存都在方法区中进行分配。而类的实例变量会在初始化阶段，通过clinit方法在堆中分配空间和赋值。

注意，这些类变量此时都是赋予了零值，比如private static int a = 123,经过该阶段a被赋予零值0，但是如果private static final int a = 123,此时a直接会被赋值为123，但是如果a为引用类型的类成员变量，不管是否被final修饰，都只在初始化时，被赋予真实值。

### 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，主要针对四大对象进行解析，分别是类或接口、字段、类方法、接口方法。

### 初始化

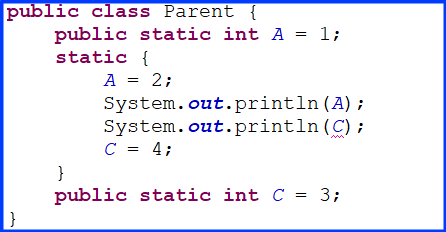
该过程是类加载过程的最后一步，是执行类构造器<clinit>()的过程，需要区别的就是实例构造器<init>,clinit有一个原则就是从上到下，先属性(static{}和字段)后方法，先静态后实例。其中有下面几个特点

1.clinit组成:

编译器 自动收集 类变量的赋值动作和静态代码块 汇集而成;

2.static访问规则:

静态代码块能赋值访问定义在它之前的静态变量，但是只能对它之后的静态变量赋值，而不能访问。



2.父类是否先执行：

clinit没有init那么严格，不需要显示调用父类构造器，因为此时虚拟机会保证父类已经执行完了clinit方法，而接口甚至都不需要虚拟机保证执行过了，只有子接口用到了父接口中的成员时才执行clinit方法。

接口实现类初始化时也不会执行接口的clinit方法。

由此可以得出一个结论就是，java.lang.Object一定是第一个执行clinit方法的对象。所以父类静态代码块先与子类执行。

3.clinit的必要性

clinit并不是必要的，如果一个类中没有static{}和对静态变量的赋值操作，可以不需要有clinit方法的创建，所以反推的话，只要有static{}或者静态变量的赋值操作，就会创建clinit方法。

4.线程安全性

一个类的clinit会在多线程环境下，会被正确的加锁和同步。

### 实战演练



## 类加载器

### 什么是类加载器

1. 理解

类加载阶段中的第一步：加载功能的实现，通过类的全限定名，得到类的二进制字节流，实现这样功能的代码模块，就叫做是类的加载器。

2、实际

类加载器这个动作常放在jvm的外部去实现，以便让应用程序自己去获取所需要的类。

3、发展

类层次划分、OSGi、热部署、代码加密等领域。

### 双亲委派模式

1、JVM视角，只存在两种类加载器

1):启动类加载器-Bootstrap ClassLoader

1]:实现语言：

C++

2]:是否属于JVM:

JVM自身的一部分

2):所有其他的类加载器

1]:实现语言：

Java,且全继承抽象类java.lang.ClassLoader

2]:是否属于JVM:

独立于JVM外部

2、Java开发人员视角，三种系统提供的类加载器

启动类加载器-Bootstrap ClassLoader

扩展类加载器-Extendsion ClassLoader

应用程序类加载器-Application ClassLoader

3、双亲委派模型

类加载器都应有自己的父类加载器,除了顶层的启动类加载器，子类加载器应该都是组合的方式来复用父类加载器的代码该模型是java在jdk1.2,设计者推荐的、非强制性的一个约束模型。

4、工作过程理解

类加载器收到一个类加载的请求，先传给自己的父类，父类无法加载才给子类，每一个层次的类加载器都是如此。

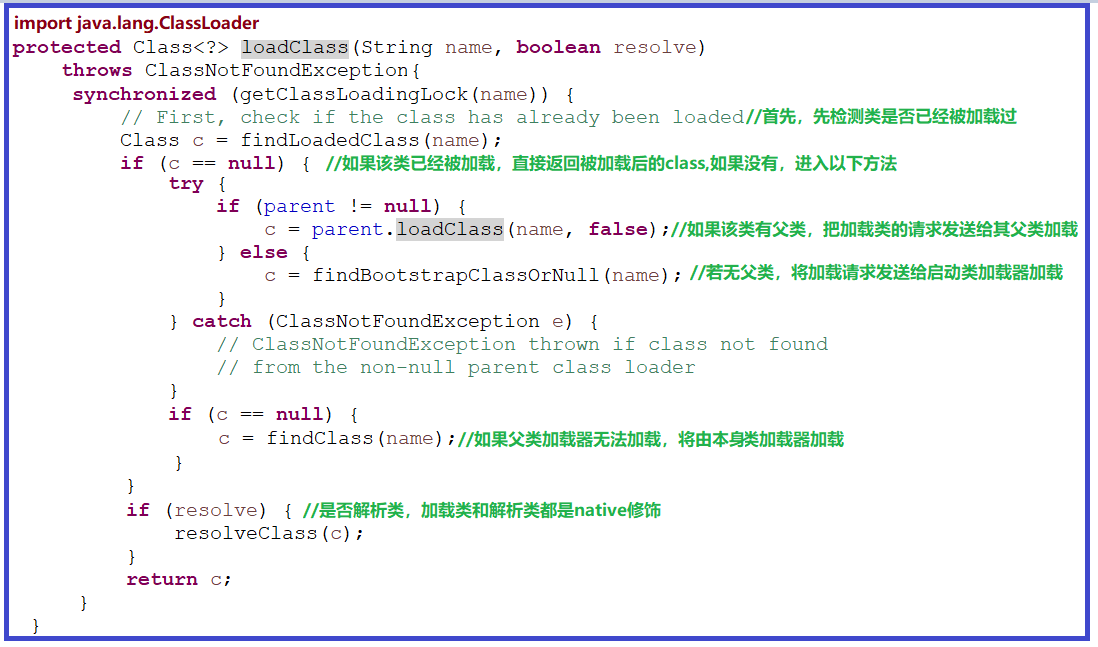
5、工作过程特点

所有的加载请求都会传送到启动类加载器。

6、优点

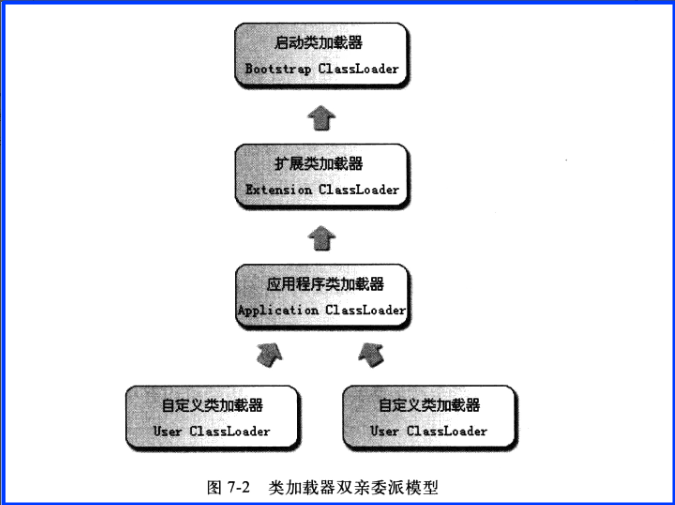
具备优先级的特点，保证底层的类，如Object,加载出来都是同一个类。

7、底层代码实现

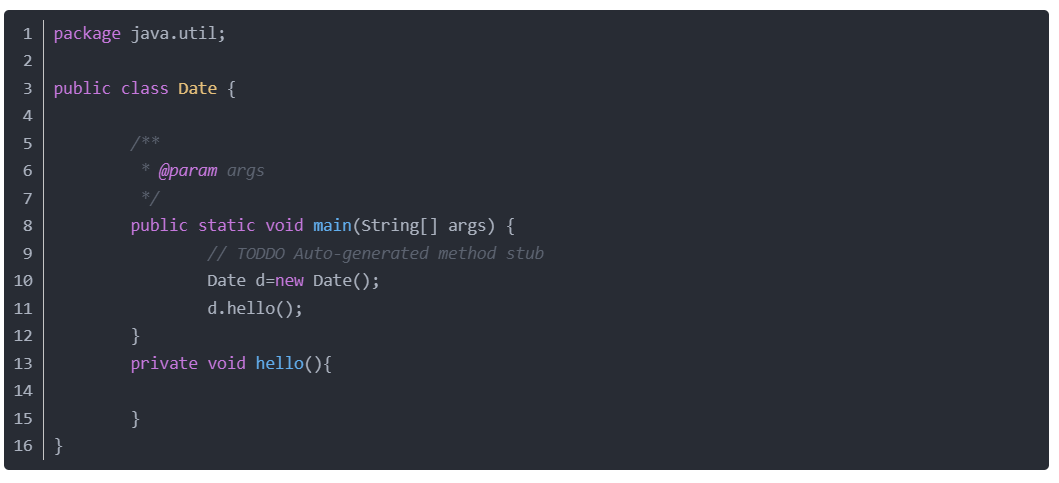


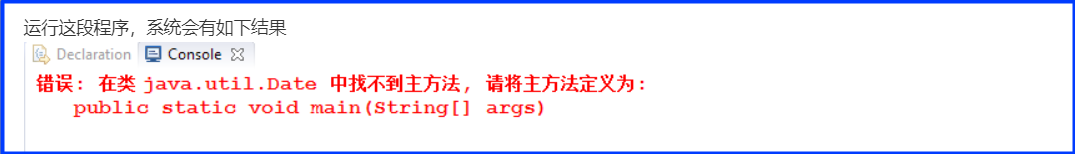
8、判断是否同一个对象

标准之一就是该类是否是同一个类加载器加载的，不认Object不相等



9、类的加载顺序-举例





首先系统会把应用程序类加载器发起的类加载请求，发送给扩展类加载器，转而发送给启动类加载器，因为启动类加载器已经加载了java.util.Date类，所以，应用程序类加载器就不会加载java.util.Date类，而系统已经加载的java.util.Date类中无main函数，所以报如上错误。

10、覆盖jar包中的类

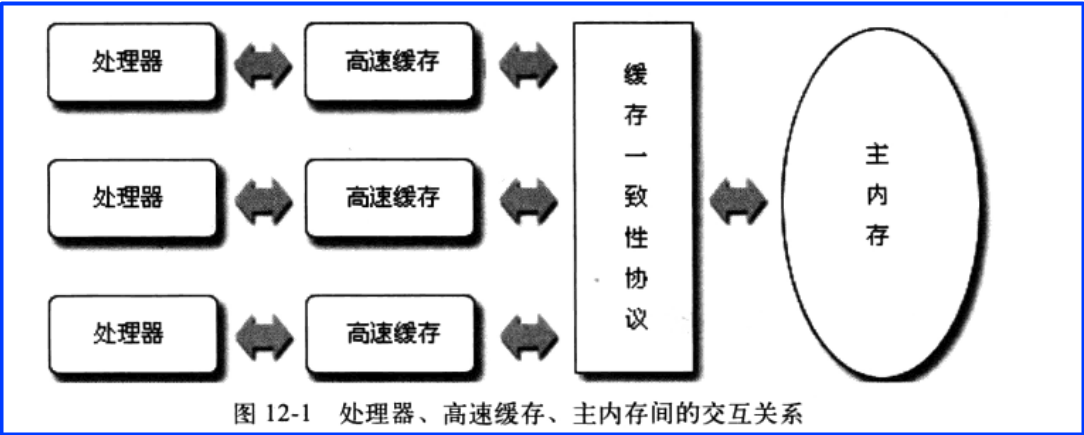
有些jar包中的类，有些方法不符合我们的需求，此时可以根据：工程路径下的文件和 jar包里面的文件重复时，优先加载工程路径下的类 这一特点，在相同的包路径下拷贝类，然后修改。因为jar包中的类既不是启动类加载器加载的，也不是扩展类加载器加载的，所以，该特点属于在应用程序类加载器加载类的顺序上的一个特点，为覆盖jar包中的方法，提供了依据。

# Java内存模型与线程

## 硬件的效率与一致性

由于主内存的读写和处理器的运算速度差距太大，主内存直接和处理器交互非常影响性能，所以在主内存和处理器中间加了一个高速缓存，满足主内存和处理器速度不匹配问题，但是现在计算机都是多处理器，当多个处理器都完成了计算任务，将要把计算结果同步到同一个主内存中时，到底同步哪一个呢？此时在高速缓存和主内存中间又添加了一个缓存一致性协议来处理这个问题。

处理器为了充分使用每一个计算单元，会对输入的代码进行乱序执行优化，对计算后的结果进行重组，使结果和顺序执行的结果一致，但是并不保证程序中的各个语句计算的顺序和输入的顺序一致。JVM中即时编译器中的指令重排序类似。



## Java内存模型

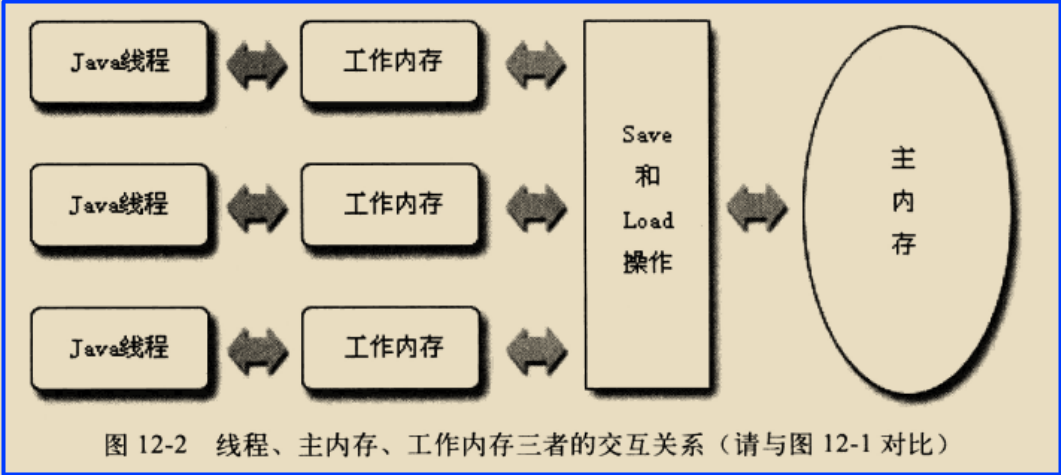
### 主内存和工作内存

Java内存模型的主要目标，就是定义程序中变量的访问规则，即定义虚拟机中变量存储到内存和从内存中取出变量这样的底层操作。

此处的变量包括了对象中的实例变量和静态变量等存在竞争的变量，但是对于局部变量和参数列表中的变量等线程私有变量就不包括。局部变量中的Reference的变量虽然在堆内存中是内存共享存在竞争的，但是Reference本身在java栈的局部变量表中，就是线程私有的，所以Reference就是线程私有的，符合上述描述。

假如说要主内存和工作内存和JVM内存空间相比较的话，那么主内存比较类似于JVM堆内存中对象实例数据部分，而工作内存相当于JVM栈内存中的部分区域。

工作内存和高速缓存为了获得更好的运行效率，都会优先存储。



### 内存间交互操作

1、lock-锁定

作用于主内存的变量,**它把一个变量标识为线程独占的状态。**

2、unlock-解锁

作用于主内存的变量,**它把一个处于锁定状态的变量释放出来。**

3、read-读取

作用于主内存的变量,**它把一个变量的值从主内存传输到工作内存中**，以便后续的load加载。

4、load-加载

作用于工作内存的变量，它把read操作从主内存中获取的变量**加载到工作内存的变量副本中。**

5、use-使用

作用于工作内存的变量，**将工作内存中变量的值传递给执行引擎，当虚拟机执行了一条需要使用变量值的字节码指令时**，会执行该use操作。

6、assign-赋值

作用于工作内存的变量，**它把从执行引擎中返回的值赋给工作内存中的变量**，当虚拟机执行一条给变量赋值的字节码指令时，执行该操作。

1. store-存储

作用于工作内存中的变量，它将工作内存当中变量的值传输到主内存中，以便于后续write操作使用。

1. write-写入

作用于主内存中的变量，它将store操作传来的变量写入到主内存中的变量中。

***其中只要求read、load和store、write两组操作顺序执行，其中read和load之间，store和write之间都可以插入其他指令，比如将变量a,b从主内存加载到工作内存中read a,read b,load b, load a***

1. read、load操作和store、write操作必须成对出现。
2. assign操作了，必须回写到主内存中，如果没有assign操作，就不能无缘由地回写到主内存中。

3、一个新的变量只能在主内存中诞生，在工作内存中未经过初始化的操作的，不经进行，初始化常指load和assign操作，即use操作前必须经过load操作呢，store操作前必须经过assign操作。

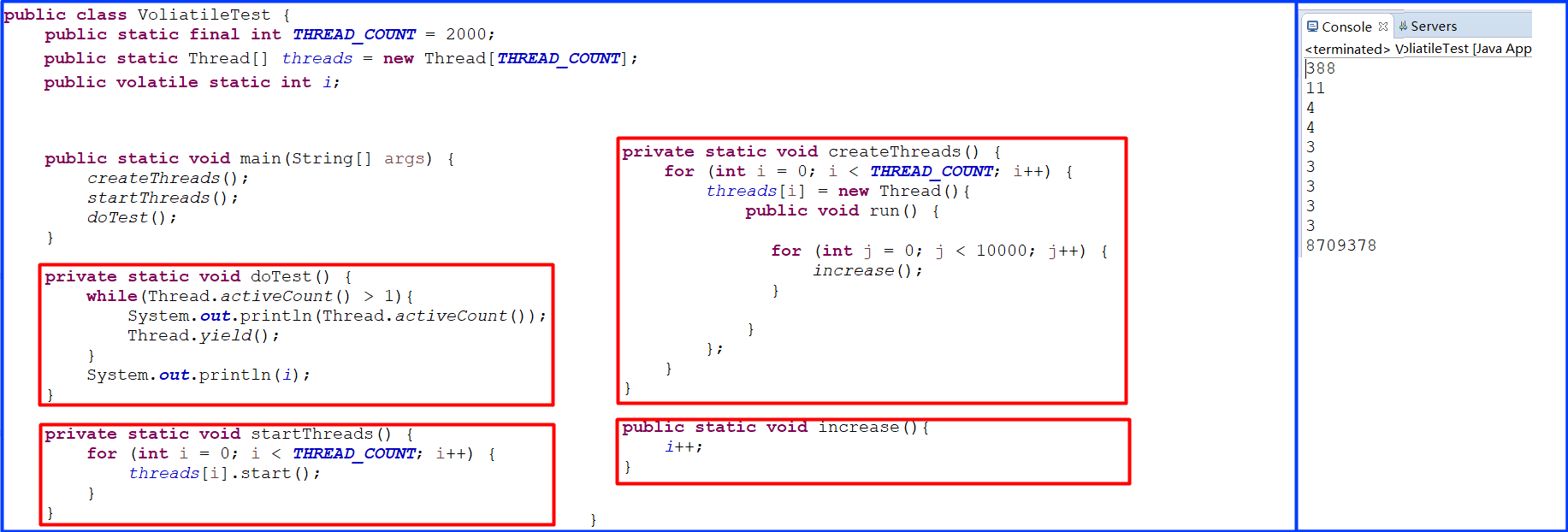
4、同一时刻，一个主内存中的变量只能被一条线程锁定，但是该线程可以对变量多次上锁，如果要解锁的话需要相同次数的解锁操作才能解锁。

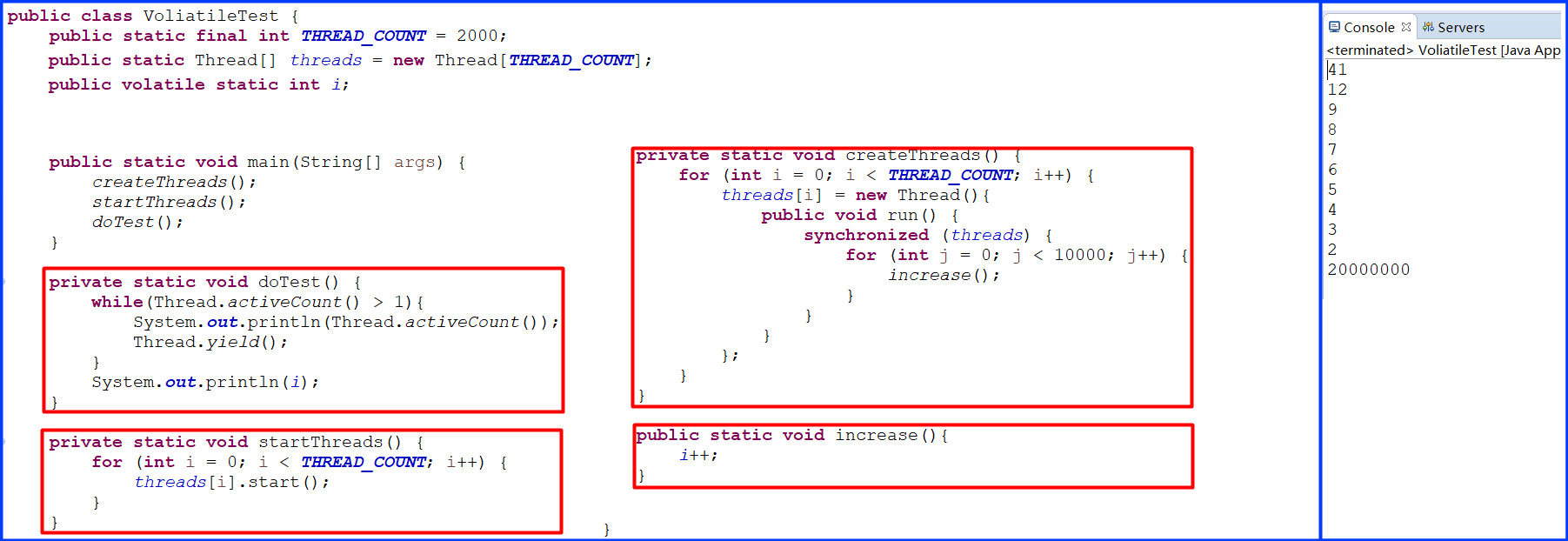
5、lock操作后，工作内存中的值会被清空，此时需要执行初始化操作load、assign后才能执行相应的use和store操作。

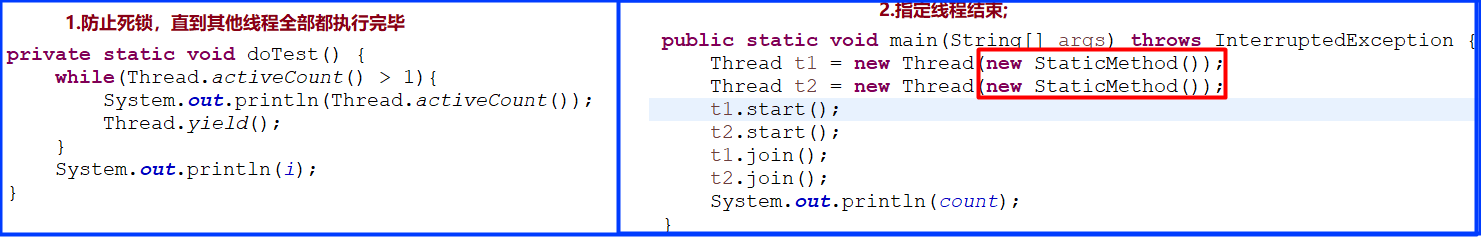
6、unlock操作执行前，必须将工作内存中的值回写到主内存中。

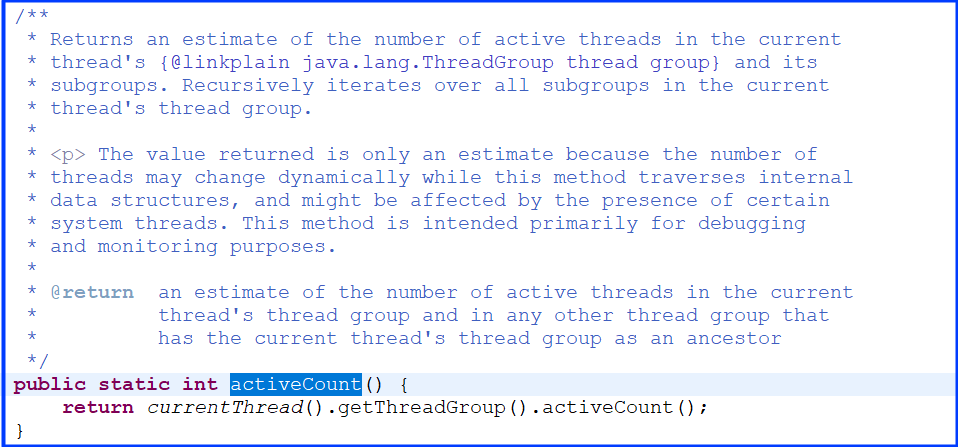
7、一个未被锁定的变量不能对它执行unlock操作，非同一个线程的变量不允许执行unlock操作。

### 对于volatile型变量的特殊规则









### 对于long和double型变量的特殊规则

JVM允许将没有被volatile修饰的64位数据的读写操作划分成两次32数据的操作进行，即不保证64位数据的read、load、store、write操作的原子性，这就是非原子性协定。但是一般的虚拟机建议也已经将64位数据的读写操作实现了其原子性。

### 原子性、可见性和有序性

1.原子性

小范围：Java内存模型直接保证原子性变量操作包括read、load、use、assign、store、write，因此基本可以认为基本数据类型的访问读写都具备原子性，long和double类型变量的的非原子性协定一般不考虑；

大范围：而lock和unlock操作一般不对用户开放，但是JVM提供了更高级的字节码指令monitorenter和monitorexit，反映到代码上就是synchronized代码块，他能够提供更大范围的原子性操作，所以synchronized也具备原子性。

2.可见性

volatile的可见性是在基本操作规则，read和load与store和write必须成对出现的前提下，添加了load和use以及assign和store必须前后紧密关联，保证工作内存中的变量新值能理解同步到共享内存中，使用变量时能理解刷新共享内存中的变量到工作内存中。 synchronized和final也就有可见性，前者是由规则“ unlock操作之前必须将工作内存中的变量同步回主内存中”而得到的。final不知道为什么在构造器中初始化后没有把this传递出去，其他线程可访问变量。无需同步就能正确访问？

2.有序性

线程内表现为有序，线程内看其他线程无序，前者表现为”线程内表现为串行的语义”，后者为”指令重排序和主内存和工作内存同步延迟”;volatile本身语义之一就有禁止指令重排序优化，而synchronized的有序性是由规则“同一时刻，只能有一条线程对一个变量上锁”得到的。

### 先行发生原则

顺序、锁、volatile、线程启动、线程终止、线程中断、对象初始化和finalize()、传递性八个;

是否是先行发生和时间顺序执行无关,就算是第一个“顺序”,只要不影响先行发生原则的正确性，处理器还是会择优乱序执行代码的。

## Java与线程

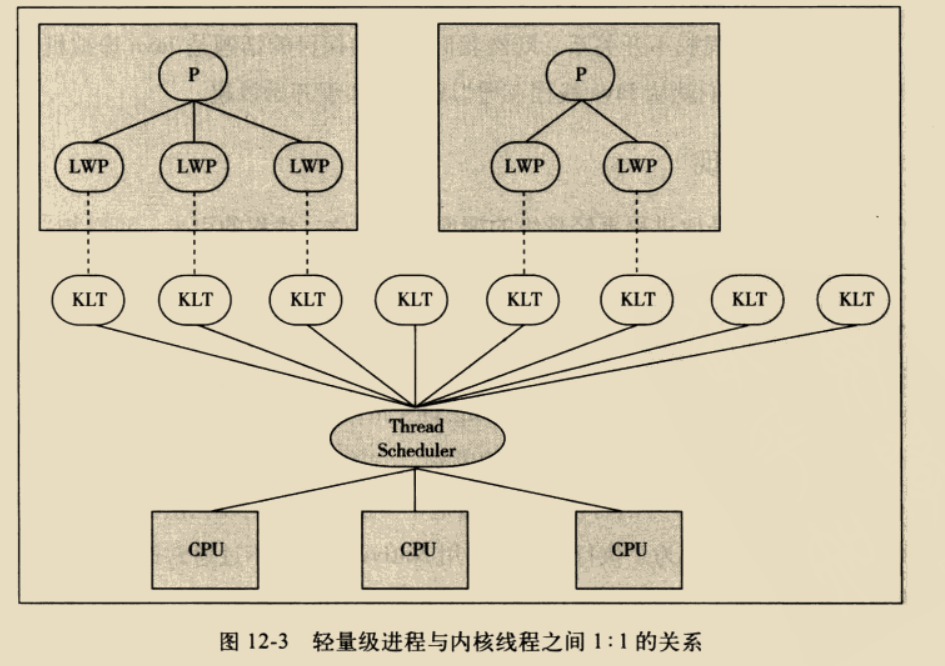
### 线程的实现

1、使用内核线程实现

线程是cpu调度的最小单位;

线程实现一般有三种方式，内核线程、用户线程、用户线程和轻量级进程的混合实现;

内核线程就是由内核负责调度和线程切换的，但是程序一般不会直接调用内核线程，而是调用内核线程的一种高级接口：轻量级进程Light Weight Process(LWP),LWP其实就是线程，一个内核线程对应一个LWP，但是使用内核线程有两个局限，一个是代价高，因为需要经常系统调用和用户态与内核态来回切换，一个是资源有限，因为内核线程的数量是有限的，内核线程和轻量级进程属于1对1的线程模型，如下图



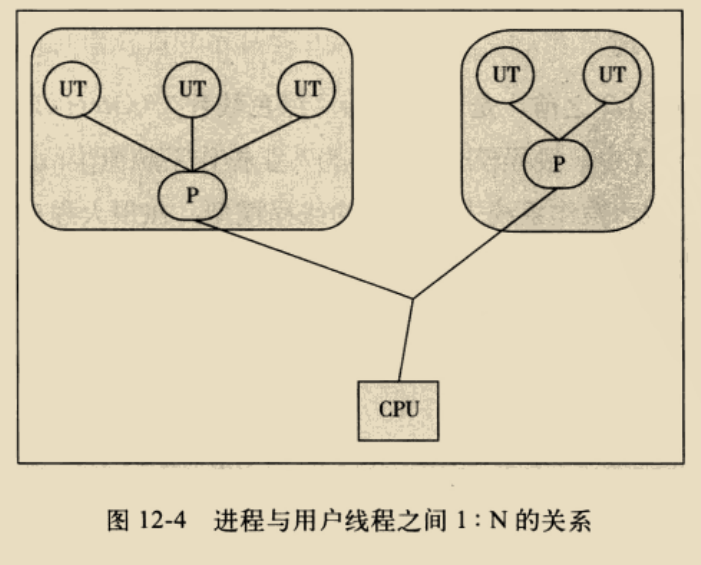
2、使用用户线程实现

从广义上讲，不属于内核线程的都属于用户线程，此时轻量级进程就属于用户线程，但是轻量级进程是基于内核实现的；

从狭义上来讲，只要是在用户空间的线程库中实现的线程都属于用户线程，用户线程的调度等一系列操作对内核线程是无感知的，也不需要内核线程的参与

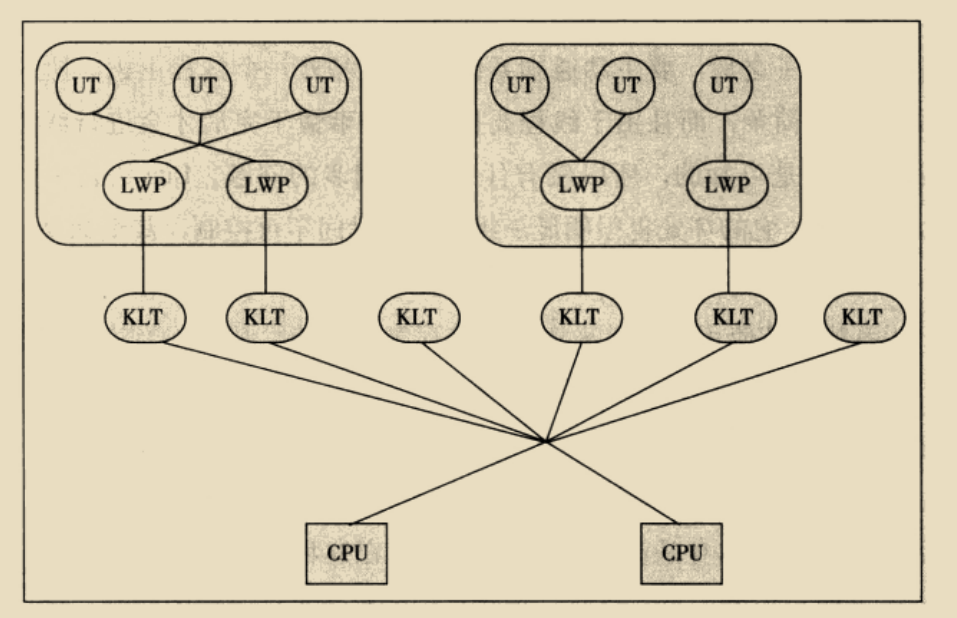
用户线程的优点和缺点都在缺少内核线程的支持，操作系统只负责将处理器的资源分配到进程，而阻塞、多处理器环境下如何将线程映射到其他处理器等操作变得很难实现，很多语言如java都已经放弃单纯使用用户线程实现。

这种进程对用户线程1对N的线程模型，如下图



3、混合实现

用户线程和轻量级进程的方式共同实现。此时用户线程依然在用户空间中，线程的创建的操作依然廉价，此时，关联用户线程和内核线程的桥梁就是轻量级进程，此时用户线程的阻塞、线程调度以及在不同处理器上的映射等问题都可以通过轻量级进程得以解决，此时轻量级进程和用户线程是M:N的关系，如下图



### Java线程调度

线程的调度就是操作系统为线程分配处理器使用权的过程;

线程的调度一般由两种，一种是协同式线程调度，一种是抢占式线程调度;

协同式线程调度的特点是，线程执行的时间由线程决定，优点是实现简单，由于线程结束时线程能够感知，所以也不存在同步问题了，它的缺点也是由于时间不可控，假如说代码错误导致该线程一直不肯让出cpu而阻塞，整个系统都要崩溃了;

抢占式线程调度，线程的调度由操作系统决定，当然此时需要给某个线程多分配点时间的话存在优先级，java线程中提供了十个线程优先级，但是仅仅依靠优先级还不能判断哪个线程先执行，因为java中的每一个线程都是要一一映射到操作系统的原生线程中的，但是某些操作系统的优先级比java线程中定义的还少，这就导致了java中的某些优先级重复等级了，当然还有其他的影响因素，比如windows中还有一个“优先级推进器”的东西，某些线程越勤奋，它会给该线程多分配点时间，所以单看线程优先级不一定能判断哪个线程先执行。

### 状态转换