# 第一部分Java技术体系

## java的"一次编译,到处运行"的机制是如何实现的?

java的所谓一次编译，到处运行，也是对于同一个版本的jre而言的，比如你用1.5版本编译出来的，在1.4版本的就不一定能运行了(jre包括 jvm以及运行时的类库)。但是，就同一个版本的jre而言，java的确做到一次编译，到处运行。因为java编译出来的并不是具体某个平台的二进制指令，而是一种伪代码，jre就是在运行java程序的时候，把这些伪代码解析为具体某个平台的二进制指令。

计算机只认识二进制，这点大家都知道的，对于不同的平台，每条指令的二进制代码不一定相同，比如move指令，windows下可能010，而unix下可能是001。java编译出来的并不是具体的 010或001二进制命令，而是一种可以让jre解释为010或001的伪代码。而C呢，编译出来的就直接是010或001这样的二进制指令了，所以假如你把 windows下编译好的C代码拿到unix下运行，unix在遇到010时不会认为它是move指令，可能把它当作push指令(假设unix下010 是push指令)，这样windows下编译出来的代码就不能在unix下正常被执行。所以，C是做不到编译一次，到处运行的。C在一个平台下编译一次，在多个相同的平台下到处运行，这是可以的。这个应该每种语言基本都能做得到。  
 不同的平台下，C的编译器也可能不同，C的lib，头文件等等也不同，那么，相同的程序就不一定能在不同的平台下编译，有可能需要改代码，否则就不存在移植这样的说法了。

## Java技术体系，JDK，JRE及JVM的关系

**Java技术体系：**Sun官方所定义的Java技术体系包括以下几个组成部分：

* Java程序语言
* Java API类库
* Java虚拟机
* Class文件格式规范
* 第三方Java类库

**JDK（Java Development Kit）**

我们把Java程序语言、Java API和Java虚拟机统称为JDK，JDK是用于支持Java程序开发的最小环境

JDK是面向开发人员使用的SDK（SDK是Software Development Kit 一般指软件开发包，可以包括函数库、编译程序等），它提供了Java的开发环境和运行环境两部分

基本上每个学java的人都会先在机器上装一个JDK，那他都包含哪几部分呢？让我们看一下JDK的安装目录。在目录C:\Program Files (x86)\java\jdk1.7.0\_11下面有六个文件夹，其中，真正在运行java时起作用的是以下四个文件夹：bin、include、lib、jre。现在我们可以看出这样一个关系，JDK包含JRE，而JRE包含JVM。

* bin：最主要的是编译器(javac.exe)，它将.java文件编译成.class文件
* include：我们知道JVM是C语言实现的，而该目录就是java和JVM交互用的头文件
* lib：类库
* jre：java运行环境，总的来说JDK是用于java程序的开发并且编译和执行程序的，而jre仅仅只是负责运行class，而没有编译的功能

eclipse、idea等其他IDE有自己的编译器而不是用JDK bin目录中自带的，所以在安装时你会发现他们只要求你选中jre路径就ok了。

**JRE（Java Runtime Environment）**

Java API类库中的Java SE API部分和Java 虚拟机这部分统称为JRE，JRE是支持Java程序运行的标准环境，是面向Java程序的使用者，而不是开发者，JDK是用于java程序的开发并且编译和执行程序的，而jre仅仅只是负责运行class，而没有编译的功能，所以在\jdk1.7.0\_1\bin目录下有Javac.exe编译器和Java.exe执行器，而在jre\bin目录下只有java.exe执行器。

**JVM（Java Virtual Machine）**

JVM是整个java实现跨平台的最核心的部分，所有的java程序会首先被编译为.class的类文件，这种类文件可以在虚拟机上执行，也就是说class并不直接与机器的操作系统相对应，而是经过虚拟机间接与操作系统交互，由虚拟机将程序解释给本地系统执行。JVM在解释执行.class文件的时候JVM需要调用解释所需要的类库lib。而jre目录里面有两个文件夹bin和lib,在这里可以认为bin里的就是JVM，lib中则是JVM工作所需要的类库，而JVM和 lib和起来就称为jre。所以在你写完java程序编译成.class之后，你可以把这个.class文件和jre一起打包发给朋友，这样你的朋友就可以运行你写程序了，因为jre里有运行.class的java.exe。

**JDK安装目录说明：**

如果安装了JDK，会发同你的电脑有两套JRE，一套位于 C:\Program Files (x86)\java\jre1.8.0\_60 另外一套位于 C:\Program Files (x86)\java\jdk1.8.0\_60\jre 目录下，后面这套比前面那套少了Server端的Java虚拟机，不过直接将前面那套的Server端Java虚拟机复制过来就行了。而且在安装JDK可以选择是否安装这个位于 C:\Program Files (x86)\java 目录下的公共JRE。JRE的地位就象一台PC机一样，我们写好的Win32应用程序需要操作系统帮我们运行，同样的，我们编写的Java程序也必须要JRE才能运行。所以当你装完JDK后，如果分别在硬盘上的两个不同地方安装了两套JRE，那么你可以想象你的电脑有两台虚拟的Java PC机，都具有运行Java程序的功能。所以我们可以说，只要你的电脑安装了JRE，就可以正确运行Java应用程序。

**问题1：为什么Sun要让JDK安装两套相同的JRE？**

这是因为JDK里面有很多用Java所编写的开发工具（如javac.exe、java.exe等），而且都放置在 C:\Program Files (x86)\java\jdk1.7.0\_1\lib\tools.jar 里。我们再dos中键入javac.exe命令与输入java -classpath tools.jar com.sun.tools.javac.Main是一样的，会得到相同的结果。从这里我们可以证明javac.exe只是一个包装器（Wrapper），而制作的目的是为了让开发者免于输入太长的指命。而且可以发现\lib目录下的程序都很小，从这里我们可以得出一个结论。就是JDK里的工具几乎是用Java所编写，所以也是Java应用程序，因此要使用JDK所附的工具来开发Java程序，也必须要自行附一套JRE才行，所以位于C:\Program Files\Java目录下的那套JRE就是用来运行一般Java程序用的。

**问题2：如果一台电脑安装两套以上的JRE，谁来决定呢？**

这个重大任务就落在java.exe身上。Java.exe的工作就是找到合适的JRE来运行Java程序。Java.exe依照底下的顺序来查找JRE：自己的目录下有没有JRE；父目录有没有JRE就查询注册表： [HKEY\_LOCAL\_MACHINE\SOFTWARE\JavaSoft\Java Runtime Environment] 所以java.exe的运行结果与你的电脑里面哪个JRE被执行有很大的关系。

## Java发展史

1991年4月，由James Gosling领导的绿色计划（Green Project）启动，此计划目的是开发一种能够在各种消费性电子产品（如机顶盒、冰箱、收音机）上运行的程序架构，这个计划的产品就是Java的前身Oak（橡树）。

1995年5月，Oak语言改名为Java，并正式发布Java 1.0版本。第一提出“Write Once，Run Anywhere”。

1996年1月，JDK 1.0发布，它提供了一个纯解释执行的Java虚拟机实现，代表技术包括：JVM、Applet、AWT等。

1997年2月，JDK 1.1发布，代表技术：jar文件格式、jdbc、Javabean、RMI。Java语法也有了一定的发展，如内部类和反射都是在这个时候出现的。

1998年12月，JDK 1.2发布，这个版本中Java技术拆分为3个方向，J2SE、J2EE和J2ME，代表性技术：EJB、Java Plug-in、Java IDL、Swing等，在语言和API上添加了strictfp关键字和Collections集合类。

2000年5月，JDK 1.3发布，主要对类库做了一些改进，并新加了JavaSound类库。

2002年2月，JDK 1.4发布，技术代表：正则表达式、异常链、NIO、日志类、XML解析器和XSLT转换器等，微软的.NET Framework也是这各时候发布的。

2004年9月，JDK 1.5发布，在Java语法的易用性上做改进，如：自动装箱、泛型、动态注解、枚举、可变长参数、遍历循环等语法，并提供Java.util.concurrent并发包等。

2006年12月，JDK 1.6发布，提供了动态语言支持，提供编译API和微型HTTP服务器API等，同时对JVM内部做了大量优化，如锁与同步、垃圾收集、类加载等算法。

2009年2月，JDK 1.7发布，它本应是一个包含许多改进的版本，其中Lambda项目（Lambda表达式、函数式编程），Jigsaw项目（JVM模块化支持），动态语言支持，GarbageFirst收集器和Coin项目（语言细节化）等，但由于开发期间Sun公司相继在技术竞争和商业竞争中陷入泥潭，股票市值下跌至仅有高峰期的3%，以无力推动JDK 1.7的研发工作。而在这个时候Oracle公司以74亿美元收购了Sun公司，大幅度裁剪了预定目标，将Lambda、Jigsaw和Coin项目的部分改进延迟到JDK 1.8发布。最终，JDK 1.7的主要改进包括：提供新的G1收集器、升级类加载架构等。

# 第二部分JVM内存模型

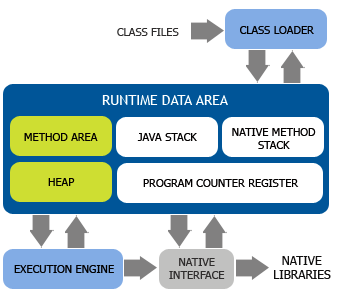
## 类装载机制和运行引擎机制

每个JVM都有两种机制：**类加载机制**和**执行引擎机制**。

**类加载器：**将java源程序编译后的.class文件加载到内存中，Class Loader加载的.class文件是有格式要求的，而不是随便建一个.class文件就能被加载。Class Loader只管加载，只要符合文件结构就加载，至于能不能运行是由Execution Engine负责的。

**执行引擎：**执行引擎也叫做解释器，负责将.class文件中的代码解析为具体某个平台的二进制指令，然后提交操作系统执行。

## JVM运行时数据区



每个JVM的都包含5个部分：**程序计数器、Java栈、本地方法栈、Java堆、方法域**，他们组成JVM的运行数据区。

### 程序计数器（PC Register）

每个线程都有一个程序计数器，就是一个指针，指向方法域中的方法字节码，以便执行引擎读取下一条命令。

### Java栈（Stack）

Java栈，是Java程序的运行区，它随线程创建而产生，对于栈来说不存在垃圾回收问题，因为只要线程结束栈内存也就释放；

每个方法执行时都会创建一个栈帧用于存储局部变量、操作数栈、动态链接、方法出口等信息；

每个方法从调用到执行结束的过程，就对应着一个栈帧在Java栈中入栈到出栈的过程；

Java栈所需的内存空间在编译期完成分配，当进入一个方法时，这个方法需要在帧中分配多大的局部变量空间是完全确定的，在方法运行期间不会改变该空间大小。

该区域规定了两种异常状况：

当线程请求的栈深度大于JVM所允许的深度时，抛出StackOverflowError异常；

当线程到达一定个数时无法再申请到足够的内存时，抛出OutOfMemoryError异常。

### 本地方法栈（Native Method Stack）

本地方法栈与Java栈的作用相似，他们的区别是Java栈为执行Java方法（字节码）服务，而本地方法栈为JVM使用到的Native方法服务；

有的虚拟机把本地方法栈和Java栈合二为一，本地方法栈和Java栈一样，也会抛出StackOverflowError和OutOfMemoryError异常。

**Native方法：**当java需要与一些底层系统如操作系统或某些硬件交换信息时，需用用到本地方法，它为我们提供了一个非常简洁的接口，而且我们无需去了解java应用之外的繁琐的细节。

### Java堆（Heap）

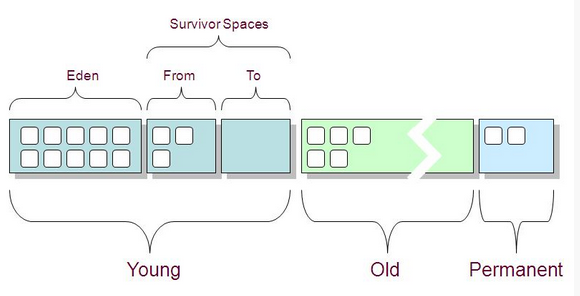
一个JVM实例只存在一个堆内存，它被所有线程共享，堆内存的大小是可以调节的，对于大多数应用，Java堆是JVM管理内存中最大的一块。

堆在JVM启动时创建，当类加载器读取了类文件后，需要把类、方法、常量、变量放到堆内存中，以方便执行器执行。几乎所以的对象实例都在这里分配内存。

Java堆是垃圾收集器管理的主要区域，因此很多时候也被称作“GC堆”。

Java堆可以细分为：新生代和老年代；在细致一点分为：Eden空间、From Survivor空间和To Survivor空间等。

如果堆中没有更多的内存完成实例分配，并且堆也无法扩展时，将会抛出OutOfMemoryError异常。



**上图是Java堆中各代分布：**

Young：主要是用来存放新生的对象。

Old：主要存放应用程序中生命周期长的内存对象。

Permanent：是指内存的永久保存区域，主要存放Class和Meta的信息,Class在被 Load的时候被放入PermGen space区域. 它和存放Instance的Heap区域不同,GC(Garbage Collection)不会在主程序运行期对PermGen space进行清理，所以如果你的APP会LOAD很多CLASS的话,就很可能出现PermGen space错误。

### 方法域（Method Area）

方法域与Java堆一样被所有线程共享，该区域保存所有字段和方法字节码，以及一些特殊方法如构造函数，接口代码也在此定义。

它用于存储已被JVM加载的类信息、常量、静态变量和即时编译器编译后的代码等数据。

JVM规范把方法域描述为Java堆的一个逻辑部分，但是他有一个别名Non-Heap（非堆），目的是与Java堆区分开。

**永久代（Permanent Generation）：**很多人喜欢把方法域称为”永久代”，但本质两者不等价。JVM对该区域的限制非常宽松，相对而言，垃圾收集行为在该区域比较少，但并非进入了方法域就如永久代的名字一样“永久”存在，这区域的内存回收主要是针对常量池的回收和类型的卸载，该区域回收的内存往往很少，但是回收却是必要的，当方法区无法满足内存分配时，将抛出OutOfMemoryError异常。

**运行时常量池（Runtime Constant Pool）：**是方法区的一部分。Class文件中除了类的版本、字段、方法、接口等描述信息外，还有一项信息是常量池，用于存放编译期生成的各种字面量和符号引用，这部分内容将会在类加载后进入方法区的运行时常量池中存放。运行时常量池具备动态性，Java语言并不要求常量一定要在编译期才能产生，运行期也能将常量放入池中，比如String类的intern()方法。当常量池无法再申请到内存时也会抛出OutOfMemoryError异常。

以上几个部分中，程序计数器、java栈、本地方法栈3个区域随线程而生，随线程而灭；栈中的栈帧随方法的进入和退出而有条不紊地执行着出栈和入栈操作。每一个栈帧中分配多少内存基本上是在类结构确定下来时就已知的，在这几个区域内就不需要过多的考虑内存回收的问题，因为方法结束或线程结束时，内存自然就跟随着回收了。而Java堆和方法区则不一样，一个接口中的多个实现类需要的内存可能不一样，一个方法中的多个分支需要的内存也可能不一样，我们只有在程序处于运行期间才能知道会创建哪些对象，这部分内存的分配和回收都是动态的，垃圾收集器所关注的都是这部分的内容。

## 对象的创建过程

①JVM遇到一条new指令时，首先去检查这条指令的参数是否能在常量池中定位到一个类的符号引用，并且检查这个符号引用代表的类是否已经被加载、解析和初始化过，如果没有执行相应的加载过程。

②类加载过后，JVM为新生对象分配内存。对象所需的内存大小在类加载完成后便可确定，然后JVM从Java堆中分配出确定大小的内存空间。

③对对象进行必要的设置，例如对象是哪个类的实例、如何才能找到类的元数据信息、对象的哈希码、对象的GC分代年龄等信息。这些信息存放在对象头中。

对象在内存中分为3个区域：对象头、实例数据和对齐填充。

**对象头**包括两部分：第一部分用于存储对象自身的运行时数据，如哈希码、GC分代年龄、锁状态标志、线程持有的锁等信息；第二部分是类型指针，即对象指向它的类元数据的指针，JVM通过这个指针来确定这个对象是哪个类的实例。

**实例数据**：该部分是对象真正存储的有效信息，也是程序代码中所定义的各种类型的字段内容。

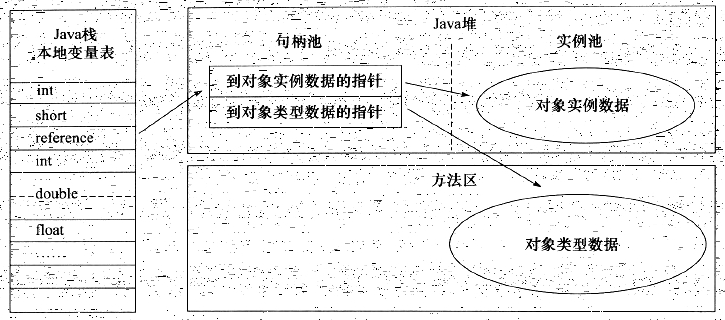
**对齐填充**：该部分不是必然存在的，也没有特别的含义，它仅仅起着占位符的作用。

## 使用Java栈中的reference来访问Java堆中的对象实例

在Java程序中，我们使用Java栈中的reference数据来操作堆上的具体对象。由于reference类型在JVM规范中只规定它是一个指向对象的引用，并没有定义这个引用通过何种方式去定位、访问堆中的对象的具体位置，所以对象访问方式取决于JVM的实现。目前主要使用句柄和直接指针两种。

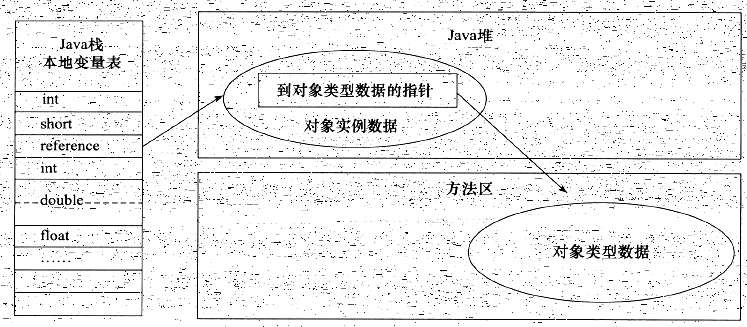
**句柄访问**

使用句柄访问的话，Java堆中将会划分出一块内存来作为句柄池，reference中存储的就是对象的句柄地址，而句柄包含了实例数据和类型数据的地址信息，如：



**直接指针访问**

使用直接指针访问的话，reference中存储的直接就是对象地址，如：



这两种访问方式各有优势，使用句柄的好处就是reference中存储的是稳定句柄地址，在对象被移动（垃圾收集时，常常需要移动对象）时只会改变句柄中的实例数据指针，而reference本身不需要修改；而使用直接指针访问就速度快，它节省了一次指针定位的时间开销。Sun HotSpot就是使用第二种方式的。

## 导致StackOverflowError和OutOfMemoryError的几种情况

### Java堆溢出

Java堆用于存储对象实例，只要不断的创造对象，并且保证GC Roots到对象之间有可达路径来避免垃圾回收机制清除这些对象，那么在对象数量到达最大堆的容量限制后就会产生内存溢出异常。

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** HeapOOM {

**static** **class** OOMObject{

}

**public** **static** **void** main(String[] args){

List<OOMObject> list = **new** ArrayList<OOMObject>();

**while**(**true**){

list.add(**new** OOMObject());

}

}

}

**运行结果：**

Exception in thread "main" java.lang.OutOfMemoryError: **Java heap space**

at java.util.Arrays.copyOf(Arrays.java:2245)

at java.util.Arrays.copyOf(Arrays.java:2219)

at java.util.ArrayList.grow(ArrayList.java:213)

at java.util.ArrayList.ensureCapacityInternal(ArrayList.java:187)

at java.util.ArrayList.add(ArrayList.java:411)

at HeapOOM.main(HeapOOM.java:12)

### Java栈和本地方法栈溢出

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

如果虚拟机在扩展栈时无法申请到足够的内存空间，则抛出OutOfMemoryError异常；

另外，在单线程中，不会出现OutOfMemoryError异常，因为在单个线程下，只要内存耗尽无法再分配的时候，JVM都将抛出StackOverflowError异常。

通过递归的方式来定义大量的本地变量，增大此方法帧中本地变量表的长度：

**public** **class** StackSOF {

**private** **int** stackLength = 1;

**public** **void** stackLeak(){

stackLength++;

stackLeak();

}

**public** **static** **void** main(String[] args) {

StackSOF stackSOF = **new** StackSOF();

stackSOF.stackLeak();

}

}

**运行结果：**

Exception in thread "main" java.lang.StackOverflowError

at StackSOF.stackLeak(StackSOF.java:6)

每个线程的栈分配的内存越大，可以建立的线程数量就越少，这样越容易把内存耗尽，所以越容易产生内存溢出异常，我们通过建立多个线程，当线程无法再申请到足够的栈内存空间时，抛出OutOfMemoryError异常：

运行以下程序会造成操作系统假死，请慎重

**public** **class** StackOOM {

**private** **void** dontStop(){

**while**(**true**){}

}

**public** **void** stackLeakByThread(){

**while**(**true**){

Thread thread = **new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

dontStop();

}

});

thread.start();

}

}

**public** **static** **void** main(String[] args) {

StackOOM stackOOM = **new** StackOOM();

stackOOM.stackLeakByThread();

}

}

运行结果：

Exception in thread "main" java.lang.OutOfMemoryError: unable to create new native thread

at java.lang.Thread.start0(Native Method)

at java.lang.Thread.start(Thread.java:691)

at StackOOM.stackLeakByThread(StackOOM.java:15)

at StackOOM.main(StackOOM.java:20)

### 方法区溢出

运行时常量池是方法区的一部分，方法区用于存放Class的相关信息，如类名、访问修饰符、常量池、字段描述、方法描述等。对于区域的测试，基本的思路是运行时产生大量的类去填满方法区，直到溢出。这里借助CGLib直接操作字节码运行时生成大量的动态类。当前很多框架，如Spring、Hibernate，在对类进行增强时，都会使用CGLib这类字节码技术，增强的类越多，就需要越大的方法区来保证动态生成的Class可以加载入内存。

**产生大量的类时方法区溢出：**

/\*\*

\* VM Args： -XX:PermSize=10M -XX:MaxPermSize=10M

\* **@author** zzm

\*/

**public** **class** MethodAreaOOM {

**public** **static** **void** main(String[] args) {

**while** (**true**) {

Enhancer enhancer = **new** Enhancer();

enhancer.setSuperclass(OOMObject.**class**);

enhancer.setUseCache(**false**);

enhancer.setCallback(**new** MethodInterceptor() {

**public** Object intercept(Object obj, Method method, Object[] args, MethodProxy proxy) **throws** Throwable {

**return** proxy.invokeSuper(obj, args);

}

});

enhancer.create();

}

}

**static** **class** OOMObject {

}

}

运行结果：

Exception in thread "main" java.lang.OutOfMemoryError: **PermGen space**

**让运行时常量池溢出：**

/\*\*

\* VM Args：-XX:PermSize=10M -XX:MaxPermSize=10M

\* **@author** zzm

\*/

**public** **class** RuntimeConstantPoolOOM {

**public** **static** **void** main(String[] args) {

// 使用List保持着常量池引用，避免Full GC回收常量池行为

List<String> list = **new** ArrayList<String>();

// 10MB的PermSize在integer范围内足够产生OOM了

**int** i = 0;

**while** (**true**) {

list.add(String.*valueOf*(i++).intern());

}

}

}

**运行结果：**

Exception in thread "main" java.lang.OutOfMemoryError: **Java heap space**

at java.util.Arrays.copyOf(Arrays.java:2245)

at java.util.Arrays.copyOf(Arrays.java:2219)

at java.util.ArrayList.grow(ArrayList.java:213)

at java.util.ArrayList.ensureCapacityInternal(ArrayList.java:187)

at java.util.ArrayList.add(ArrayList.java:411)

at RuntimeConstantPoolOOM.main(RuntimeConstantPoolOOM.java:16)

# 第3章 JVM的垃圾收集与内存分配

## 对象存活判定算法

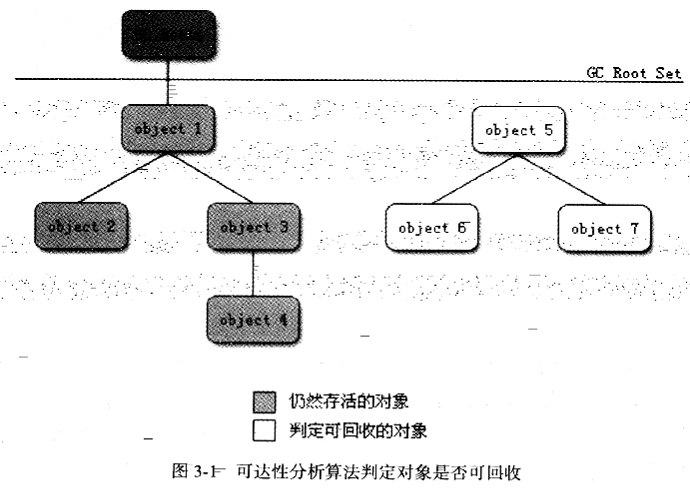
JVM运行时区域的程序计数器、虚拟机栈、本地方法栈3个区域随线程而生，随线程而灭；栈中的栈帧随着方法的进入和退出而有条不紊地执行着出栈和入栈操作。每个栈帧中分配的内存多少基本上是在类结构确定下来时就已知了，因此在这几个区域的内存分配和回收都具备确定性，在这几个区域内就不需要过多考虑回收的问题，因为方法结束或线程结束，内存自然就被回收了。而Java堆和方法区则不一样，一个接口中的多个实现类需要的内存可能不一样，一个方法中的多个分支需要的内存也可能不一样，我们只有在程序处于运行期间时才能知道会创建哪些对象，这部分内存的分配和回收都是动态的，垃圾收集器所关注的是这部分内存。

### 引用计数算法

给对象添加一个引用计数器，每当有一个地方引用它时，计数器加1；当引用失效时，计数器减1；任何时刻计数器为0的对象就是不可能再被使用的。主流的JVM都没有使用引用计数算法来管理内存，其中最主要的原因是它很难解决对象之间相互循环引用的问题。

### 可达性分析算法

可达性分析算法是判断对象是否存活的主流实现，该算法的基本思路是通过一系列称为“GC Roots”的对象作为起始点，从这些节点开始向下搜索，搜索所走的路径称为引用链，当一个对象到GC Roots没有任务引用链时，则证明此对象是不可用的。对象object5、object6、object7虽然互相有关联，但是它们到GC Roots是不可达的，所以它们将会被判定为是可回收的对象。



在Java中，可作为GC Roots的对象包括下面几种：

* 虚拟机栈（栈帧中的本地变量表）中引用的对象；
* 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（Native方法）引用的对象；

## Finalize()方法

/\*\*

\* 此代码演示了两点：

\* 1.对象可以在被GC时自我拯救。

\* 2.这种自救的机会只有一次，因为一个对象的finalize()方法最多只会被系统自动调用一次

\* **@author** zzm

\*/

**public** **class** FinalizeEscapeGC {

**public** **static** FinalizeEscapeGC *SAVE\_HOOK* = **null**;

**public** **void** isAlive() {

System.***out***.println("yes, i am still alive :)");

}

@Override

**protected** **void** finalize() **throws** Throwable {

**super**.finalize();

System.***out***.println("finalize mehtod executed!");

FinalizeEscapeGC.*SAVE\_HOOK* = **this**;**//重新建立引用，实现自我拯救**

}

**public** **static** **void** main(String[] args) **throws** Throwable {

*SAVE\_HOOK* = **new** FinalizeEscapeGC();

//对象第一次成功拯救自己

*SAVE\_HOOK* = **null**;

System.*gc*();**//这里被回收前调用finalize()方法**

// 因为Finalizer方法优先级很低，暂停0.5秒，以等待它

Thread.*sleep*(500);

**if** (*SAVE\_HOOK* != **null**) {

*SAVE\_HOOK*.isAlive();

} **else** {

System.***out***.println("no, i am dead :(");

}

// 下面这段代码与上面的完全相同，但是这次自救却失败了

*SAVE\_HOOK* = **null**;

System.*gc*();

// 因为Finalizer方法优先级很低，暂停0.5秒，以等待它

Thread.*sleep*(500);

**if** (*SAVE\_HOOK* != **null**) {

*SAVE\_HOOK*.isAlive();

} **else** {

System.***out***.println("no, i am dead :(");

}

}

}

finalize方法在对象实例第一次被回收前执行，任何一个对象的finalize方法都只会被系统自动调用一次（有且只有一次），对象面临第二次回收，它的finalize方法不会被再次执行，因此第二段代码的自救行动失败。finalize方法是对象逃脱死亡的最后一次机会，只要重新与引用链上的任何一个对象建立关联即可。finalize能做的所有工作，使用try-finally或者其他方式都可以做得更好，所以笔者不建议使用该方法。

## 方法区的垃圾回收

很多人认为方法区（或者HotSpot中的永久代）是没有垃圾收集的，JVM规范中确实说过可以不要求虚拟机在方法区实现垃圾收集，因为在方法区中进行垃圾收集的性价比很低。但是永久代还是可以进行垃圾回收的，主要回收有两部分内容：**废弃常量**和**无用的类**。

**废弃常量的判定：**以常量池中字面量的回收为例，假如一个字符串“abc”已经进入了常量池，但是当前系统没有任何一个String对象是叫做“abc”的，实际上就是没有任何String对象引用常量池中的“abc”常量，同时也没有其他地方引用这个字面量，如果这时发生内存回收，而且必要的话，这个“abc”常量就会被系统清理出常量池。常量池中的其他类（接口）、方法、字段的符号引用也与此类似。

**无用的类的判断：**无用的类需要同时满足下面3个条件才能算是“无用的类”：

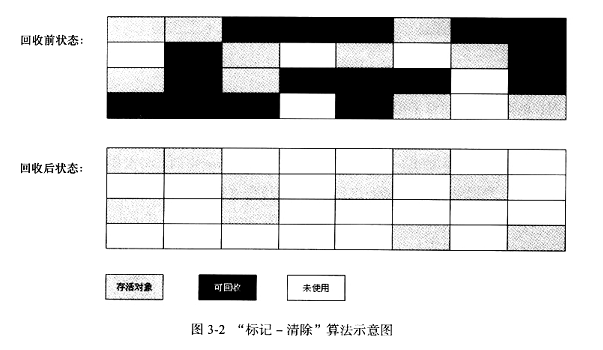
* 该类所有实例都已经被回收，也就是Java堆中不存在该类的任何实例；
* 加载该类的ClassLoader已经被回收；
* 该类对应的java.lang.Class对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法

在大量使用反射、动态代理、CGLib等ByteCode框架、动态生成JSP以及OSGi这类频繁自定义ClassLoader的场景都需要JVM具备类卸载的功能，以保证永久代不会溢出。

## 垃圾收集算法

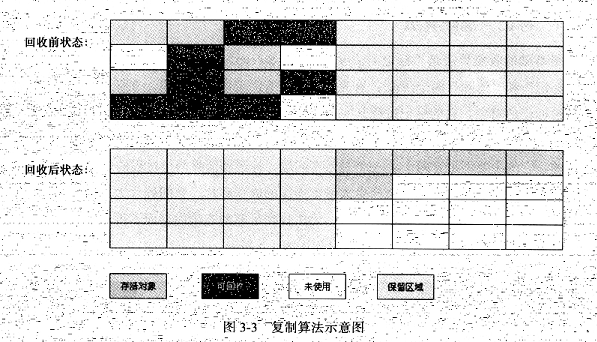
### 标记-清除算法（适用于老年代）

该算法是最基础的算法，算法分为“标记”和“清除”两个阶段：首先标记出所有需要回收的对象，在标记完成后统一回收所有被标记的对象。它的主要不足有两个：一个是效率问题，标记和清除两个过程的效率都不高；另一个是空间问题，标记清除之后会产生大量不连续的内存碎片，导致后续分配较大对象时，无法找到足够的连续内存而不得不提前再触发一次回收动作。



### 复制算法（适用于新生代）

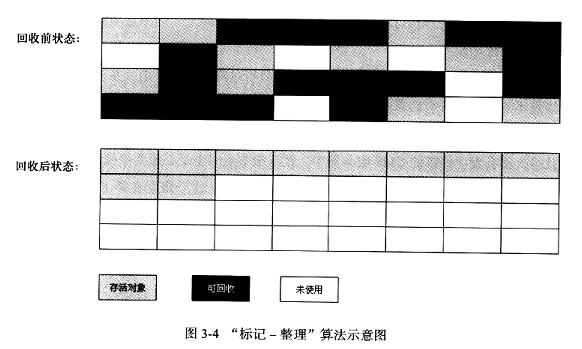
为解决“标记-清除算法”的效率问题，复制收集算法出现了，它将可用内存按容量划分为大小相等的两块，每次只使用其中一块，当这块内存用完了，就将还存活的对象复制到另一块内存，然后再回收使用过的内存空间一次清理掉。这样每次都是对整个半区进行内存回收，内存分配时就不用考虑内存碎片等复杂情况。这种算法的代价是将内存缩小为原来的一半。



由于Java堆的新生代中的对象98%是“朝生夕死”的，所以并不需要按照1:1的比例划分内存空间，而是将内存分为一块较大的Eden空间和两块较小的Survivor空间，每次使用Eden和其中一块Survivor。当回收时，将Eden和Survivor中还存活着的对象一次性地复制到另一块Survivor空间上，最后清理掉Eden和刚才用过的Survivor空间。HotSpot虚拟机默认Eden和Survivor的大小比例是8:1，也就是新生代中可用的内存空间为整个新生代容量的90%，只有10%的内存会被“浪费”。

### 标记-整理算法（适用于老年代）

复制收集算法在对象存活率较高时就要进行较多的复制操作，效率将会变低。所以在老年代一般不能直接选用这种算法。根据老年代的特点，有人提出了“标记-整理”算法，标记过程仍和“标记-清楚”算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活对象都向一端移动，然后直接清理掉边界以外的内存。



### 分代收集算法

当前商业虚拟机都使用“分代收集”算法，这种算法并没有什么新的思想，只是根据对象存活周期的不同将内存划分为几块。一般是把Java堆分为新生代和老年代，这样就可以根据各个年代的特点采用最适当的收集算法。一般在新生代中，每次收集时都会有大批对象死去，所以可以选用复制算法，只需要复制少量存活对象就可以。而老年代因为存活率较高，所以使用“标记-清除”或“标记-整理”算法。

## 几种垃圾收集器

**Serial收集器**

**ParNew收集器**

**Parallel Scavenge收集器**

**Serial Old收集器**

**Parallel Old收集器**

**GMS收集器**

**G1收集器**

## JVM的内存分配

Java自动内存管理最终可以归结为自动化的分配内存和回收内存。关于回收我们已经介绍了一些回收算法和一些回收器。接下来我们来讲讲如何给对象分配内存。

多数情况下，对象在新生代Eden区中分配。当Eden区没有足够的空间进行分配时，JVM将发起一次Minor GC。如果GC期间JVM发现存活的对象无法全部放入Survivor空间，则通过分配担保机制提前转移到老年代去。

**Minor GC和Full GC（或称Major GC）的区别：**

新生代GC（Minor GC）：指发生在新生代的垃圾收集动作，因为Java对象大多都具备朝生夕灭的特性，所以Minor GC非常频繁，回收速度也比较快。

老年代GC（Full GC/Major GC）：指发生在老年代的GC，出现了Full GC，经常会伴随至少一次的Minor GC（但也不绝对）。Full GC的速度都比较慢。

另外，Minor GC会把Eden中的所有活的对象都移到Survivor区域中，如果Survivor区中放不下，那么剩下的活的对象就被移到Old generation 中。

**大对象直接进入老年代**

所谓的大对象是指，需要大量连续内存空间的Java对象，最典型的大对象就是那种很长的字符串以及数组。大对象对JVM的内存分配来说就是一个坏消息，而比遇到大对象更加坏的消息就是遇到一群“朝生夕灭”的“短命大对象”，写程序的时候应当避免，经常遇到大对象容易导致内存还有不少空间，就提前触发垃圾收集以获取足够的连续空间来“安置”它们。

JVM提供了一个-XX:PretenureSizeThreshold参数，令大于这个设置值的对象直接在老年代分配。这样做的目的是避免在Eden区及两个Survivor区之间发生大量的内存复制，新生代采用复制算法来收集内存。

**长期存活的对象将直接进入老年代**

JVM采用了分代收集的思想来管理内存，那么内存回收时就必须能识别哪些对象应放在新生代，哪些对象应放在老年代中。为了做到这点，JVM给每个对象定义了一个对象年龄计数器。如果对象在Eden出生并经过第一次Minor GC后仍然存活，并且能被Survivor容纳的话，将被移动到Survivor空间中，并且对象年龄设为1，对象在Survivor区中每“熬过”一次Minor GC，年龄就增加1岁，当它的年龄增加到一定程度（默认为15），就会晋升到老年代。这个晋升值可以通过-XX:MaxTenuringThreshold设置。

**动态对象年龄判定**

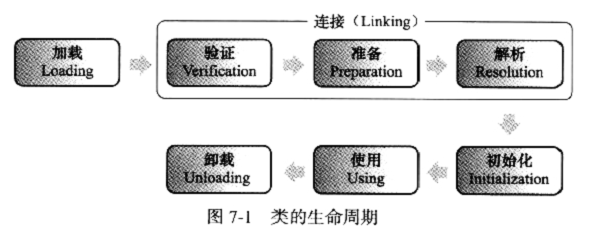
为了能更好地适应不同程序的内存状况，虚拟机并不是永远地要求对象的年龄必须达到了MaxTenuringThreshold才能晋升老年代，如果Survivor空间中相同年龄所以对象大小的总和大于Survivor空间的一半，年龄大于或等于该年龄的对象就可以直接进入老年代，无需等到MaxTenuringThreshold要求的年龄。

# 第7章 虚拟机类加载机制

JVM把Class文件加载到内存，并进行校验、解析和初始化等操作，然后形成可以被JVM直接使用的Java类型，这就是JVM的类加载机制。

注：这里的“Class文件”并非特定指某个存在于具体磁盘的文件，而是一串二进制的字节流，无论以何种形式存在。它可以从ZIP包中获取，如jar、rar、war等格式的包；从网络中获取，如Applet；运行时计算生成，如动态代理技术；从其他文件生成，如有JSP文件生成对应的Class类；从数据库中读取等等。

## 类加载的生命周期



* 类从被加载到内存，到卸载出内存的整个生命周期包括：加载、验证、准备、解析、初始化、使用和卸载7个阶段。
* 其中验证、准备和解析统称为连接。
* 加载、验证、准备、初始化和卸载这个5个阶段执行的开始顺序是确定的，但不一定按顺序完成，例如，加载阶段可能尚未完成，连接阶段就已经开始，但是开始的时间一定是保持先后顺序的。
* 解析阶段在某些情况下可以在初始化阶段之后，这是为了支持Java语言的运行时绑定（也称为动态绑定或晚期绑定）。

### 加载阶段

加载阶段，JVM完成以下3件事情：

* 通过类的全限定名获取类的二进制字节流；
* 将字节流代表的静态存储结构转化为方法区的运行时数据结构；
* 在内存中生成代表这个类的Java.lang.Class对象。

注：这里的二进制节流不一定要从Class文件获取，开发人员可以通过定义自己的类加载器去控制字节流的获取方式，即重写一个类加载器的loadClass方法。

### 验证阶段

该阶段的目的是为了确保二进制字节流（Class文件）中包含的信息是否符合当前JVM的要求，以确保JVM的安全。该验证是否严谨，决定了JVM是否能承受恶意代码的攻击，如果载入了有害的字节流会到导致系统崩溃。

验证阶段大致上会完成下面4个阶段的检验动作：

1. **文件格式验证**

验证字节流是否符合Class文件格式的规范以及能否被当前JVM所处理。该阶段的验证是基于二进制字节流进行的，只有通过了这个阶段的验证后，字节流才会进入内存的方法区进行存储，所以后面的3个验证阶段都基于方法区的存储结构进行的，不会再直接操作字节流。

1. **元数据验证**

对字节码描述信息进行语义分析，以保证其描述信息符合Java语言规范要求。

1. **字节码验证**

通过数据流和控制流分析，确定程序语义是否合法、符合逻辑。

1. **符号引用验证**

确保解析动作能正常执行，如果无法通过符号引用验证将抛出如java.lang.IllegalAccessError、java.lang.NoSuchFieldError、java.lang.NoSuchMethodError等异常。

### 准备阶段

该阶段正式为类变量分配内存并设置类变量的初始值，这些变量所使用的内存都将在方法区中进行分配。

注：这里分配的变量是类变量（即static修饰的变量），而不包括实例变量；设置类变量的初始值“通常情况”下是指数据类型的零值，如：

Public static int value = 123;

那变量value在准备阶段过后的初始值为0而不是123。在“特殊情况”下，如果是常量值如：

Public final int value = 123;

那编译时javac将会为value生成ConstantValue属性，并复制为123。

### 解析阶段

该阶段是将JVM将常量池内的符号引用替换为直接引用的过程。

符号引用：相当于标识ID。

直接引用：相当于引用地址，如直接指向目标的指针、相对偏移量等。

### 初始化阶段

类初始化阶段是累加载过程的最后一个，前面的类加载过程中，除了在加载阶段用户应用程序可以通过自定义类加载器参与之外，其余动作完全有JVM主导和控制。到了初始化阶段，才真正开始执行类中定义的Java程序代码（或者说是字节码）。在准备阶段，变量已经赋过一次系统要求的初始值，而在初始化阶段，则根据程序员通过程序制定的数据去初始化类变量和其他资源，或者可以从另外一个角度来表达：初始化阶段是执行类构造器<clinit>()方法的过程。

**<clinit>()方法有以下特点：**

1. <clinit>()方法是由编译器自动收集类中的所有类变量的赋值动作和静态语句块（static{}）中的语句合并产生的，编译器收集的顺序是源文件中出现的顺序锁决定的，静态语句块中只能访问到定义在静态语句块之前的变量，定义在它之后的变量，在前面的静态语句块可以赋值，但是不能访问，如：

**public class** Test {  
 **static** {  
 *i* = 0; *// 给变量复制可以正常编译通过* System.***out***.print(*i*); *// 这句编译器会提示“非法向前引用”* }  
 **static int** *i* = 1;  
}

1. JVM保证在子类的<clinit>()方法执行前，父类的<clinit>()方法已经执行完毕。因此JVM中第一个执行的<clinit>()方法的类肯定是java.lang.Object。
2. 由于父类的<clinit>()方法先执行，也就意味着父类中定义的静态语句块要优先于子类的变量赋值操作。
3. <clinit>()方法对于类或接口来说不是必须的，如果一个类中没有静态语句块，也没有对变量的赋值操作，那么编译器可以不为这个类生成<clinit>()方法。
4. 接口与类不同的是，执行接口的<clinit>()方法前不需要先执行父接口的<clinit>()方法，接口的实现类在初始化时也不会执行接口的<clinit>()方法。

## 类加载器

“通过一个类的全权限定名来获取描述此类的二进制字节流”，实现这个动作的代码称为“类加载器”。

任何一个类在JVM中的唯一性是由加载它的类加载器和这个类本身一同确定的。比较两个类是否“相等”，必须是由同一个类加载器加载才有意义，即使两个类源于同一个Class文件，被同一个JVM加载，但是只要类加载不同，那这两个类就必定不相等。（注意：这里的“相等”包括代表类的Class对象的equals()方法、isAssignableFrom()方法、isInstance()方法以及instanceof关键字判断关系的返回结果）。

/\*\*

\* 类加载器与instanceof关键字演示

\*/

**public** **class** ClassLoaderTest {

**public** **static** **void** main(String[] args) **throws** Exception {

ClassLoader myLoader = **new** ClassLoader() {

@Override

**public** Class<?> loadClass(String name) **throws** ClassNotFoundException {

**try** {

String fileName = name.substring(name.lastIndexOf(".") + 1) + ".class";

InputStream is = getClass().getResourceAsStream(fileName);

**if** (is == **null**) {

**return** **super**.loadClass(name);

}

**byte**[] b = **new** **byte**[is.available()];

is.read(b);

**return** defineClass(name, b, 0, b.length);

} **catch** (IOException e) {

**throw** **new** ClassNotFoundException(name);

}

}

};

Object obj = myLoader.loadClass("ClassLoaderTest").newInstance();

System.***out***.println(obj.getClass());

System.***out***.println(obj **instanceof** ClassLoaderTest);

}

}

运行结果：

class ClassLoaderTest

false

### 启动类加载器（Bootstrap ClassLoader）

从Java虚拟机来讲，只存在两种不同的类加载器：**启动类加载器（Bootstrap ClassLoader）和其他类加载器。**

启动类加载器使用C++语言实现，是虚拟机的一部分，它负责将存在JAVA\_HOME\lib目录中的、或者被-Xbootclasspath参数所指定路径中类库加载到JVM内存中。启动类加载器无法被java程序直接引用。

另外一种就是所有其它的类加载器，这些类加载器是由Java语言实现，独立于JVM外部，并且全部继承自抽象类java.lang.ClassLoader。

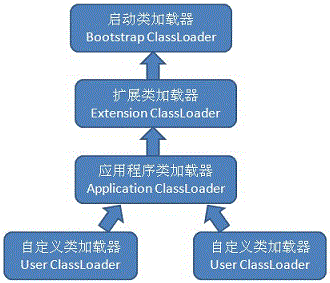
### 扩展类加载器（Extension ClassLoader）

该加载器器是用JAVA编写，且它的父类加载器是Bootstrap，是由sun.misc.Launcher$ExtClassLoader实现的，主要加载JAVA\_HOME/lib/ext目录中的类库。开发者可以自己使用扩展类加载器。

应用程序类加载器（Application ClassLoader）

该类加载器也称为系统类加载器，它负责加载用户类路径（Classpath）上所指定的类库，开发者可以直接使用该类加载器，如果应用程序中没有自定义过自己的类加载器，一般情况下这个就是程序中默认的类加载器。我们的应用程序都是由这三类加载器互相配合进行加载的，我们也可以加入自己定义的类加载器。

## 双亲委派模型

如图所示的类加载器之间的这种层次关系，就称为类加载器的双亲委派模型（Parent Delegation Model）。该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当有自己的父类加载器。子类加载器和父类加载器不是以继承（Inheritance）的关系来实现，而是通过组合（Composition）关系来复用父加载器的代码。

**双亲委派模型的工作过程为**：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，每一个层次的加载器都是如此，因此所有的类加载请求都会传给顶层的启动类加载器，只有当父加载器反馈自己无法完成该加载请求（该加载器的搜索范围中没有找到对应的类）时，子加载器才会尝试自己去加载。

类加载器双亲委派模型是从JDK1.2以后引入的，并且只是一种推荐的模型，不是强制要求的。使用这种模型来组织类加载器之间的关系的好处是Java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。例如java.lang.Object类，无论哪个类加载器去加载该类，最终都是由启动类加载器进行加载，因此Object类在程序的各种类加载器环境中都是同一个类。否则的话，如果不使用该模型的话，如果用户自定义一个java.lang.Object类且存放在classpath中，那么系统中将会出现多个Object类，应用程序也会变得很混乱。如果我们自定义一个rt.jar中已有类的同名Java类，会发现JVM可以正常编译，但该类永远无法被加载运行。  
在rt.jar包中的java.lang.ClassLoader类中，我们可以查看类加载实现过程的代码，具体源码如下：

**protected** **synchronized** Class loadClass(String name, **boolean** resolve) **throws** ClassNotFoundException {

// 首先检查该name指定的class是否有被加载

Class c = findLoadedClass(name);

**if** (c == **null**) {

**try** {

**if** (parent != **null**) {

// 如果parent不为null，则调用parent的loadClass进行加载

c = parent.loadClass(name, **false**);

} **else** {

// parent为null，则调用BootstrapClassLoader进行加载

c = findBootstrapClass0(name);

}

} **catch** (ClassNotFoundException e) {

// 如果仍然无法加载成功，则调用自身的findClass进行加载

c = findClass(name);

}

}

**if** (resolve) {

resolveClass(c);

}

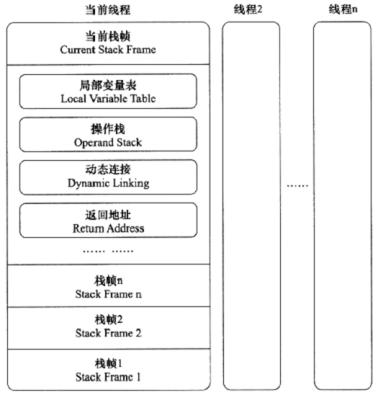
**return** c;

}

# 第8章 虚拟机字节码执行引擎

**栈帧的概念：**栈帧是用于支持JVM进行方法调用和方法执行的数据结构，栈帧存储了方法的局部变量表、操作数栈、动态链接和方法返回地址等信息。每一个方法从调用开始到执行完成对应着一个栈帧在虚拟机栈入栈到出栈的过程。

在编译程序代码的时候，栈帧中需要多大的局部变量表，多深的操作数栈都已经完全确定了，并且写入到方法表的Code属性之中，因此一个栈帧需要分配多少内存，不会受到程序运行期变量数据的影响，而仅仅取决于具体的虚拟机实现。



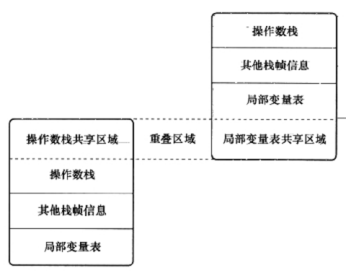
**局部变量表：**局部变量表是一组变量存储空间，用于存放方法参数和方法内部定义的局部变量。在java程序编译为Class文件时，就确定了该方法所需分配的局部变量表的最大容量。局部变量表以局部槽（Variable slot，下称slot）为最小单位。

为了尽可能节省栈帧空间，局部变量表中的slot是可以重用的。

我们已经知道类变量有两次赋初始值的过程，一次在准备阶段，赋予系统初始值；另外一次在初始化阶段，赋予程序员定义初始值。因此，即使在初始化阶段程序员没有为类变量赋值也没关系，类变量仍然具有一个确定的初始值。但局部变量就不一样，局部变量必须赋予初始值才能使用。

**操作数栈：**当一个方法刚刚开始执行的时候，这个方法的操作数栈是空的，在方法的执行过程中，会有各种字节指令往操作数栈中写入和提取内容，也就是出栈/入栈操作。它是方法执行过程临时数据的存放区域。

另外，在概念模型中，两个栈帧作为JVM的元素，是完全独立的。但在大多JVM的实现里都会做一些优化处理，让两个栈帧出现一部分重叠。如：让下面栈帧的部分操作数栈与上面栈帧的部分局部变量表重叠在一起，这样在进行方法调用时就可以共用一部分数据，无须额外进行参数复制传递。



Java虚拟机的解释执行引擎称为“基于栈的执行引擎”，其中所指的“栈”就是操作数栈。

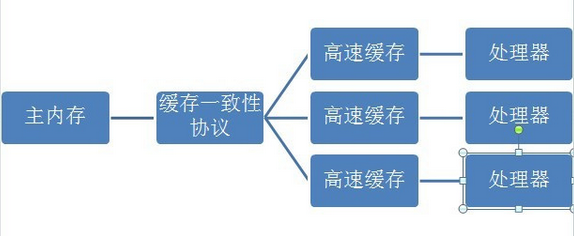
**动态连接：**每个栈帧都包含一个指向运行时常量池中该栈帧所属方法的引用，在方法调用过程中各个方法的引用就构成了动态连接。

我们知道Class文件的常量池中存放大量的符号引用，字节码中的方法调用指令就以常量池中指向方法的符号引用作为参数。这些符号引用一部分在类加载阶段或第一次使用时转化为直接引用，这种转化称为静态解析。另一部分将在每一次运行期间转化为直接引用，这部分称为动态连接。

# Java内存模型与线程

## 硬件的缓存

由于运算速度的差距，CPU和存储设备间加入多层的cache，但同时也引入了缓存一致性的问题。在多处理器系统中，每个处理器都有自己的高速缓存，而他们又共享一个主内存，如下图。当多个处理器的运算任务都涉及同一块主内存区域时，将可能导致各自的缓存数据不一致的问题，如果发生这种情况，那同步回到主内存时以谁的缓存数据为准呢？为解析此问题，需要各个处理器都遵循一些协议，在读写时根据协议来进行操作，这类协议有：MSI,MESI,MOSI,Synapse,Firefly和Dragon Protocol等。“内存模型”一词，可以理解为在特定操作协议下，对特定的内存或高速缓存进行读写访问的过程抽象。不同架构的物理机器可以拥有不一样的内存模型，而JVM也有自己的内存模型。这里我们介绍硬件的缓存访问操作和下面要介绍的内存访问操作具有很高的可比性。

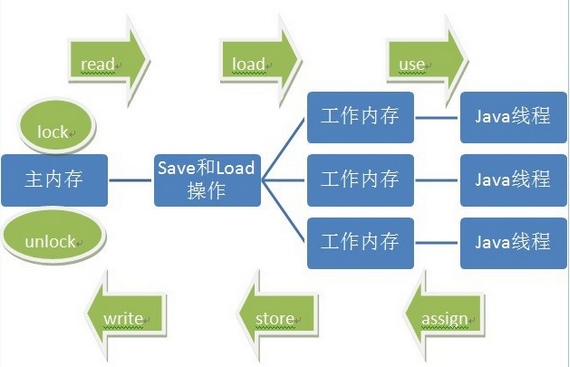


另外，为了使内部运算单元能尽量利用，CPU可能会对输入代码进行乱序执行优化。

## Java内存模型

JVM规范中试图定义中Java内存模型（Java Memory Model，JMM）来屏蔽掉各种硬件和操作系统的内存访问差异，以实现让Java程序在各种平台下都能达到一致的内存访问效果。在此之前，主流程序语言（如C/C++等）直接使用物理硬件和操作系统的内存模型，因此，会由于不同平台内存模型的差异，有可能导致程序在一套平台上并发完全正常，而在另一个平台上并发访问却经常出错，因此在某些场景就必须针对不同的平台来编写程序。

内存模型的主要目标是定义程序中各个变量的访问规则，即在JVM中将变量存储到主内存和工作内存底层细节。java内存模型规定，所有变量都存储在主内存中。每条线程还有自己的工作内存（类比高速缓存）。线程工作内存中保存了被该线程使用到的变量的主内存副本拷贝，线程对变量的所有操作（读取、赋值等）都必须在工作内存中进行，而不能直接读写主内存中的变量。线程间变量值的传递均需通过主内存中的变量。不同的线程之间也无法直接访问对方工作内存中的变量，线程间变量值的传递均需要通过主内存来完成，线程、主内存、工作内存三者的交互关系如下：



注意：上述的变量它包括实例字段，静态字段和构成数组对象的元素，但不包括局部变量和方法参数，因为后者是线程私有的，不会被共享。

## 内存间的基本交互操作和操作规则

**8种基本操作**

**Java内存模型定义了8种基本操作来完成主内存和工作内存间的交互：**

* lock（锁定）：作用于主内存的变量，把一个变量标识为一条线程独占的状态。
* read（读取）：作用于主内存的变量，把一个变量的值从主内存传输到线程的工作内存，以便随后的load动作使用。
* load（载入）：作用于工作内存的变量，把read操作从主内存中得到的变量值放入工作内存的变量副本中。
* use（使用）：作用于工作内存的变量，把工作内存中一个变量的值传递给执行引擎。
* assign（赋值）：作用于工作内存的变量，把执行引擎接收到的值赋给工作内存的变量。
* store（存储）：作用于工作内存的变量，把工作内存中一个变量的值传送给主内存中，以便随后的write操作使用。
* write（写入）：作用于主内存的变量，把store操作从工作内存中得到的变量的值放入主内存的变量中。
* unclock（解锁）：作用于主内存的变量，把一个处于锁定状态的变量释放出来，释放后的变量才可以被其他线程锁定。

如果要把一个变量从主内存复制到工作内存，那就要顺序执行read和load操作，如果要从工作内存同步会主内存就要顺序执行store和write操作。这里只要保正是顺序操作而不一定是连续操作。

**Java内存模型规定了执行以上8中操作时必须满足以下规则：**

* 不允许read和load、store和write操作之一单独出现，即不允许一个变量从主内存中读取了但工作内存不接受，或者工作内存发起了回写但主内存不接受的情况出现。
* 不允许一个线程丢弃它的最近的assign操作，即变量在工作内存中改变了之后必须把该变化同步到主内存。
* 不允许一个线程无原因地（没有发生任何assign操作）把数据从线程的工作内存同步到主内存。
* 一个新的变量只能在主内存中诞生，不允许在工作内存中直接使用一个未被初始化（load或assign）的变量，换句话说，就是对一个变量use、store操作之前，必须先执行过了assign和load操作。
* 一个变量在同一时刻只允许一个线程对其进行lock操作，但lock操作可以被同一条线程重复执行多次，多次执行lock后，只有执行相同次数的unclock操作，变量才会被解锁。
* 如果对一个变量执行lock操作，那么会清空工作内存次变量的值，在执行引擎使用这个变量前，需要重新执行load或assign操作初始化变量的值。
* 如果对一个变量执行lock操作，那么会清空工作内存次变量的值，在执行引擎使用这个变量前，需要重新执行load或assign操作初始化变量的值。
* 对一个变量执行unclock之前，必须先把此变量同步回主内存中（执行store、write操作）。

## volatile变量的特殊规则

关键volatile可以说是JVM提供的最轻量级的同步机制，但是它不容易完全被正确、完整地理解，以至于许多程序员都习惯不去使用它，遇到需要处理多线程数据竞争问题的时候一律使用synchronized来进行同步。了解volatile变量的语义可以很好的帮助我们理解多线程操作特性。

当一个变量定义为volatile之后，它将具备两种特性：**线程间的可见性**和**禁止指令重排序优化**。

**第一种特性 线程间的可见性**：这里的“可见性”是指当一条线程修改了这个变量的值，新值对于其他线程来说是可以立即得知的。而普通变量不能做到这一点，普通变量的值在线程间传递均需通过主内存来完成。例如，线程A改修一个普通变量的值，然后向主内存回写，另外一条线程B要在线程A回写完成之后再从主内存进行读取操作，新变量值才会对线程B可见。关于volatile变量的可见性经常会被误解，很多人会认为：”volatile变量对所有线程是立即可见的，对volatile变量所有的写操作都能立刻反应到其他线程之中，换句话就是volatile变量在各个线程中是一致的，~~所以基于volatile变量的运算在并发下是安全的~~”。前面说的都没错，但是“~~基于volatile变量的运算在并发下是安全的~~”这个结论是错的。为什么呢？其实在各个线程的工作内存中，volatile变量也可以存在不一致的情况，但由于每次使用volatile修饰的变量之前都要先刷新，所以执行引擎看不到不一致的情况，因此认为不存在一致性问题。但是Java里面的运算并非原子操作，导致volatile变量的运算在并发下一样是不安全的。我们通过如下测试代码来说明原因：

*//volatile变量自增运算测试***public class** VolatileTest {  
  
 **public static volatile int** *race* = 0;  
 **private static final int *THREADS\_COUNT*** = 20;  
  
 **public static void** main(String[] args) {  
 Thread[] threads = **new** Thread[***THREADS\_COUNT***];  
 **for** (**int** i = 0; i < ***THREADS\_COUNT***; i++) {  
 threads[i] = **new** Thread(**new** Runnable() {  
 @Override  
 **public void** run() {  
 **for** (**int** i = 0; i < 1000; i++) {  
 *increase*();  
 }  
 }  
 });  
 threads[i].start();  
 }  
  
 **try** {  
 *//主线程停止10秒，让20个累加线程都执行完成后打印在打印race* Thread.*sleep*(10000);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 System.***out***.println(*race*);  
 }  
  
 **public static void** increase() {  
 *race*++;  
 }  
  
}

这段代码发起了20个线程，每个线程对race变量进行1000自增，如果这段代码能够正确并发的话，正确的结果应该是20000，但实际运行结果总是小于20000。为什么呢？问题就出现在自增运算“race++”，race++并不是一个原子操作，我们可以把它在细分为rece+1运算和赋值两部分运算操作。所以，volatile变量只能保证可见性，在不符合以下两条规则的运算场景，我们仍然要通过加锁（或使用synchronized或java.util.concurrent中的原子类）来保证原子性。

1. 写入变量不依赖此变量的值（例如：i++，实质上是i=i+1;我们对i进行赋值时不能使用到i它本身），或者只有一个线程修改此变量
2. 变量的状态不需要与其它变量共同参与不变约束

**volatile的使用场景**：我们可以使用布尔类型volatile变量来控制并发，如下：

**public boolean shutdownRequested**;  
  
**public void** shutdown(){  
 **shutdownRequested** = **true**;  
}  
  
**public void** doWork(){  
 **while** (!**shutdownRequested**){  
 *//do samething ...* }  
}

**第二种特性 禁止指令重排序优化：**

volatile的变量附近的代码不会被指令重排序打乱。

volatile屏蔽指令重排序的语义在JDK1.5中才被完全修复，此前的JDK中即使将变量声明为volatile也仍然不能完全避免重排序所导致的问题（主要是volatile变量前后的代码仍然存在排序问题），这点也是JDK1.5之前的Java中无法安全地使用DCL（双检锁）来实现单例模式的原因。

另外在性能上，volatile与普通变量上在写操作可能会慢些，因为需要插入内存屏障来保证不乱序执行。不过比sync高效很多。

具体操作细节：

A：read，load和use必须一起出现。保证使用变量前必须先从主内存刷新最新的值

B：assign，store和write必须一起出现。保证改变变量的时候刷新回去主内存。

C：被volatile修饰的变量不会被指令重排序优化，保证代码的执行顺序与程序的顺序相同

## 原子性，可见性和有序性

Java 内存模型是围绕着在并发过程中如何处理原子性、可见性和有序性3个特征来建立的：

* **原子性：**Java内存模型保证read、load、use、assign、store、write这个6个变量操作的操作都具有原子性。如何应用场景需要一个更大范围的原子性保证，Java内存模型提供了lock和unlock来满足需求，它是通过synchronized语法来加锁的。
* **可见性：**可见性是指当一个线程修改了共享变量的值，其他线程能够立即得知这个修改。Java内存模型是通过在变量修改后将新值同步会主内存，volatile的特殊规则是保证了新值能立即同步到主内存，以及每次调用前立即从主内存刷新。因此，可以说volatile保证了多线程操作时的可见性，而普通变量则不能保证这一点。除volatile外，final和synchronized 也能实现可见性。
* **有序性：“**如果在本线程内观察，所有操作都是有序的；如果在一个线程观察另一个线程，所有的操作都是无序的”，前半句是指“线程内表现为串行的语义”，后半句指“指令重排序”现象和“工作内存与主内存同步延迟”现象。Java语言提供了volatile和synchronized两个关键字来保证线程之间操作的有序性，volatile关键字本身包含了禁止重排序指令的语义，而synchronized则是由“一个变量在同一个时刻只允许一条线程对其进行lock操作”这条规则实现的，这条规则决定了持有同一个锁的两个同步块只能串行地进入。

## 先行发生原则

如果Java内存模型中所有有序性都靠volatile和synchronized来实现，那么编写代码时将会很繁琐，然后事实并非如此，这是因为Java中有一个“先行发生”的原则。下面是Java内存模型下一些“天然的”先行发生规则，可以在编码中直接使用。

1. 程序次序规则：同一个线程内，按照代码出现的顺序，前面的代码先行于后面的代码，准确的说是控制流顺序，因为要考虑到分支和循环结构。

2. 管程锁定规则：一个unlock操作先行发生于后面（时间上）对同一个锁的lock操作。

3. volatile变量规则：对一个volatile变量的写操作先行发生于后面（时间上）对这个变量的读操作。

4. 线程启动规则：Thread的start()方法先行发生于这个线程的每一个操作。

5. 线程终止规则：线程的所有操作都先行于此线程的终止检测。可以通过Thread.join()方法结束、Thread.isAlive()的返回值等手段检测线程的终止。

6. 线程中断规则：对线程interrupt()方法的调用先行发生于被中断线程的代码检测到中断事件的发生，可以通过Thread.interrupt()方法检测线程是否中断

7. 对象终结规则：一个对象的初始化完成先行于发生它的finalize()方法的开始。

8. 传递性：如果操作A先行于操作B，操作B先行于操作C，那么操作A先行于操作C。

## 线程的实现

主流的操作系统都提供了线程实现，Java语言则提供了在不同硬件和操作系统平台下对线程操作的统一处理，每个已经执行start()且还未结束的java.lang.Thread类的实例就代表了一个线程。我们注意到Thread类与大部分的Java API有显著的差别，它的所有关键方法都是声明为Native的。在Java API中，一个Native方法往往意味着这个方法没有使用或无法使用与平台无关的手段来实现。

实现线程主要有3种方式：

* 使用内核线程实现
* 使用用户线程实现
* 使用用户线程加轻量级进程混合实现

# 线程安全与锁优化

## 5种线程安全等级

为了更加深入地理解线程安全，在这里我们不把线程安全当做一个非真即假的二元排他项看待，我们按线程的“安全程度”由强至弱来排序，它们分别是：**不可变**、**绝对线程安全**、**相对线程安全**、**线程兼容**和**线程对立**。

### 不可变（final）

Java语言中，如果共享数据是一个基本数据类型，那么只要在定义时使用final关键字修饰它就可以保证它是不可变的；保证对象不可变的方法有很多种，其中最简单就是把对象中带有状态的变量都声明为final。这里的“不可变”是指引用不可变，内容还是可以修改了的，例如声明一个Collection的集合类，我们仍然可以向其添加或删除元素。

### 绝对线程安全（只能通过加锁或synchronized来实现）

在Java　API中标注自己是线程安全的类，大多数都不是绝对的线程安全。就比如java.util.Vector：

**import** java.util.ArrayList;  
**import** java.util.List;  
**import** java.util.Vector;  
**public class** VectorTest {  
  
 **private static** Vector<Integer> *vector* = **new** Vector<Integer>();  
  
 **public static void** main(String[] args) {  
 **while** (**true**) {  
 **for** (**int** i = 0; i < 10; i++) {  
 *vector*.add(i);  
 }  
  
 Thread removeThread = **new** Thread(**new** Runnable() {  
 @Override  
 **public void** run() {  
 **for** (**int** i = 0; i < *vector*.size(); i++) {  
 *vector*.remove(i);  
 }  
 }  
 });  
  
 Thread printThread = **new** Thread(**new** Runnable() {  
 @Override  
 **public void** run() {  
 **for** (**int** i = 0; i < *vector*.size(); i++) {  
 System.***out***.println((*vector*.get(i)));  
 }  
 }  
 });  
  
 removeThread.start();  
 printThread.start();  
  
 *//不要同时产生过多的线程，否则会导致操作系统假死* **while** (Thread.*activeCount*() > 20);  
 }  
 }  
  
}

**执行结果：(截取一部)**

...

7

9

Exception in thread "Thread-32523" java.lang.ArrayIndexOutOfBoundsException: Array index out of range: 3

at java.util.Vector.get(Vector.java:744)

at VectorTest$2.run(VectorTest.java:31)

at java.lang.Thread.run(Thread.java:722)

0

1

...

尽管在这里使用到的Vector的get()、remove()、和size()都是同步方法，但是在多线程环境中，如果不在方法调用端做额外操作的同步措施的话，该代码仍然是不安全的，因为另一个线程恰好在错误的时间里删除了一个元素，导致序号i已经不在可用。比如在printThread 线程中i自增完以后就切换到另一个线程removeThread 中去，removeThread 线程把当前i的元素除了，当线程printThread 再次被唤醒时*vector*.get(i)元素已经不存在了

**改进方法：**

Thread removeThread = **new** Thread(**new** Runnable() {  
 @Override  
 **public void** run() {  
 **synchronized** (*vector*) {  
 **for** (**int** i = 0; i < *vector*.size(); i++) {  
 *vector*.remove(i);  
 }  
 }  
 }  
});  
  
Thread printThread = **new** Thread(**new** Runnable() {  
 @Override  
 **public void** run() {  
 **synchronized** (*vector*) {  
 **for** (**int** i = 0; i < *vector*.size(); i++) {  
 System.***out***.println((*vector*.get(i)));  
 }  
 }  
 }  
});

**执行结果：不会再出现异常情况**

### 相对线程安全（Java API中所谓的线程安全类，如Vector、HashTable等）

相对线程安全就是我们通常意义上说的线程安全，他需要保证对这个对象单独的操作是线程安全的，我们调用的时候不需要做额外保障措施，但是对于一些特定顺序的连续调用，就可能需要在调用端使用额外的同步手段来保证调用的正确性。在Java中，大部分线程安全类都属于这种类型，例如Vector、HashTable、Collections集合等。

### 线程兼容（本身不是线程安全但可以通过同步手段实现线程安全的类）

线程兼容指对象本身并不是线程安全的，但是可以通过在调用端正确地使用同步手段来保证对象在并发环境中可以安全地使用，我们通常说的一个类不是线程安全的，绝大多数是指这种情况。

### 线程对立（如：suspend、resume方法）

线程对立是指无论调用端是否采取了同步措施，都无法再多线程环境中并发使用的代码。例如Thread类的suspend和resume方法，如果suspend中断的线程就是即将要执行resume方法的那个线程，那就肯定要产生死锁。也正是由于这个原因，suspend和resume已经被JDK废弃。线程对立这种排斥多线程的代码是很少出现的，而且通常都是有害的，应尽量避免。

## 锁优化

Java的线程是映射到操作系统的原生线程之上的，如果要阻塞或唤醒一个线程，都需要操作系统来帮忙完成，这就需要从用户态转换到核心态中，因此状态转换需要耗费很多的处理器时间。对于简单的同步代码块，状态转换消耗的时间有可能比用户代码执行时间还要长。

以下是几种锁优化方案：

**自旋锁**

**自适应自旋锁**

**锁消除**

**锁粗化**

**轻量级锁**

**偏向锁**