深入理解Java虚拟机

# Part 1. 走近Java

## 1.1 概述

## 1.2 java技术体系

## 1.3 java发展史

## 1.4 java虚拟机发展史

### 1.4.1 sun classic/exact vm

### 1.4.2 sun hostpot vm

### 1.4.3 sun mobile-embedded vm/meta-circular vm

### 1.4.4 bea jrockit/ibm j9 vm

### 1.4.5 azul vm/bea liquid vm

### 1.4.6 apache harmony/google android dalvik vm

### 1.4.7 microsoft jvm 及其它

## 1.5展望java技术的未来

### 1.5.1 模块化

### 1.5.2 混合语言

### 1.5.3 多核并行

### 1.5.4 进一步丰富语法

### 1.5.5 64位虚拟机

## 1.6实战：自己编译jdk

## 1.7本章小结

# Part 2. 自动内存管理机制

Java与C++之间有一堵由“内存动态分配”和“垃圾收集技术”所围成的“高墙”，墙外的人想进去，墙里面的人却想出来。

## 2 Java内存区域与内存溢出异常

### 2.1 概述

C、C++拥有每一个对象的“所有权”有承担着每一个对象声明开始到终结的维护责任。

Java由虚拟机管理内存，如容易出现“内存泄漏”和“内存溢出”问题。

### 2.2 运行时数据区域

Java虚拟机把管理的内存划分为若干个不同的数据区，各有用途，有的区域随着虚拟机进程的启动而存在，有些依赖用户线程的启动和结束而建立和销毁。Java SE 7版虚拟机管理的内存分为如下图中所示的几个运行时数据区域：

本地方法库

方法区

Method Area

堆

Heap

虚拟机栈

VM Stack

本地方法栈

Native Method

Stack

程序计数器

Program Counter Register

执行引擎

本地库接口

由所有线程共享的数据区

线程隔离的数据区

#### 2.2.1程序计数器

程序计数器是一夸较小的内存空间，可以看做是当前线程所执行的字节码的行号指示器，期解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，分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖这个计数器来完成。

每条线程都需要有一个独立的程序计数器，各条线程之间计数器之间计数器互不影响，独立存储----线程私有 的内存。

执行Java方法，计数器记录的是正在执行的虚拟机字节码指令地址；执行Native方法，技术及值为空。此内存区域是唯一一个在Java虚拟机规范中没有规定任何OutOfMemoryError情况的区域。

#### 2.2.2 Java虚拟机栈

Java虚拟机栈也是线程私有的，它的生命周期与线程相同。虚拟机栈描述的是Java方法执行的内存模型：每个方法执行的同时都会创建一个栈帧，用于存储局部变量表、操作数栈、动态链接、方法出口等信息。每一个方法从调用到执行完毕的过程，就对应着一个栈帧在虚拟机栈中入栈到出栈的过程。

局部变量表存放了编译器可知的各种基本数据类习惯、对象引用(指向对象其实地址的引用指针或代表对象的句柄或其他词对象相关的文职)。long和double占用2个局部变量，其余数据类型占1个。局部前量表所需的内存空间在编译期间完成分配，一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局部变量表的大小。

虚拟机栈规定了两种异常状况：线程请求的栈深度大于虚拟机所允许的深度，将抛出StackOverflowError异常；如虚拟机栈可以动态扩展，扩展时无法申请到足够的内存，就会抛出OutOfMemoryError异常。

#### 2.2.3 本地方法栈

本地方法栈与虚拟机栈所发挥的作用非常相似，虚拟机栈为虚拟机执行Java方法(字节码)服务；本地方法栈为虚拟机使用到的Native方法服务，甚至又饿虚拟机(如:Sun HotSpot虚拟机)直接把本地方法栈和虚拟机栈合二为一。与虚拟机栈一样，本地方法栈区域也会抛出StackOverflowError和OutOfMemoryError异常。

#### 2.2.4 堆

堆(Heap)是Java虚拟机所管理的内存中最大的一块。Java堆是被所有线程共享的一块区域，在虚拟机启动时创建。此内存区域的唯一目的就是存放对象实例，几乎所有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”(Garbage Collected Heap)，Java堆中还可以细分为：新生代和老年代；再细致点：Eden空间、From Survivor空间、To Survivor空间等。分区域是为了更好地回收内存或更快地分配内存。

Java堆可以处于物理上不连续的内存空间空，只要逻辑上是连续的即可。堆既可以是固定大小，也可以是可扩展的(-Xmx和-Xms控制)，如果在堆中没有内存弯沉实例分配，并且堆也无法再扩展时，将会抛出OutOfMemoryError异常。

#### 2.2.5 方法区

方法区也是各个线程共享的内存区域，用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。Java虚拟机规范把方法区描述为堆的一个逻辑部分，但是它还有一个别名叫做Non-Heap(非堆)，目的应该是与Java堆区分开来。

很多人叫方法区为“永久代”，垃圾收集器比较少出现在这个区，并非数据就会“永久”存在，此区域的内存回收主要是针对常量池的回收和对类型的卸载。

根据Java虚拟机规范的规定，当方法区无法满足内存分配需求是，将抛出OutOfMemoryError异常。

#### 2.2.6 运行时常量池

运行时常量池是方法区的一部分。Class文件中除了有类的版本、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。

运行时常量池是方法区的一部分，自然受到方法区内存的限制，当常量池无法再申请到内存时会抛出OutOfMemoryError异常。

#### 2.2.7 直接内存

不是虚拟机运行时数据区的部分，也不是Java虚拟机规范汇总定义的内存区域。但会频繁使用，也可能导致OutOfMemoryError异常。

NIO类，基于通道与缓冲区的I/O方式，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。在一些场景中提高性能，因为避免在Java堆和Naïve堆中来回复制数据。

直接内存的分配不会受到Java堆大小的限制，但是，既然是内存，肯定还是会受到本机总内存(RAM SWAP 分页文件)大小以及处理器寻址空间的限制。如果直接内存区域总和大于物理内存限制，则会导致动态扩展是出现OutOfMemoryError异常。

### 2.3 HotSpot虚拟机对象探秘

#### 2.3.1 对象的创建

虚拟机遇到一条new指令时，首先检查指令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。若果没有，那必须先执行相应的类加载过程。

接下来为新生对象分配内存。所需内存在类加载完成后便可完全确定，为对象分配空间等同于把一块内存从Java堆中划分出来。

方式一：“指针碰撞”，已用内存在一边，空闲内存在一边，指针在中间，为对象分配内存就是指针向空闲空间那边挪动一段与对象大小相等的距离；

方式二：“空闲列表”，虚拟机为何一个列表，记录哪些内存可用，在分配的时候从列表中找到一块足够大的空间划分给对象实例。

使用哪种方式和垃圾收集器是否带有压缩整理功能有关，使用Serial、ParNew等带Compact过程收集器采用方式一；使用CMS基于Mark-Sweep算法的收集器时，通常采用空闲列表。

对象创建时线程不安全的，在修改指针过程中，有可能其它对象使用了原来的指针来分配内存。解决方法有两种：

方式一：对分配内存空间的动作进行同步处理----实际上虚拟机采用CAS配上是爱重试的方式保证更新操作的原子性；

方式二：把内存分配的动作按照线程划分在不同的空间中进行，每个线程在Java堆中欲仙分配一小块内存，成为“本地线程分配缓冲(Thread Local Allocation Buffer--TLAB)”,只有在TLAB用完并分配新的TLAB时，才需要同步锁定，虚拟机是否使用TLAB，可以通过-XX:+/-UseTlab参数来指定。

内存分配完成后，虚拟机将内存空间全部初始化为零值(不包括对象头)，保证对象的实例字段在Java代码中可以不赋值就直接使用(默认值0)。

下一步，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分代年龄等信息。存放在对象头之中。根据虚拟机当前运行状态的不同，是否启用偏向锁等，对象头会有不同的设置方式。

此时，对象已经产生，但还没有<init>。所有的字段都还未零。对象创建完成之后执行<init>方法，把对象按照程序猿的意愿进行初始化，这样一个完整的对象才算完全产生出来。

#### 2.3.2 对象的内存布局

在HotSpot虚拟机中，对象在内存中存储布局可以分为3块区域：对象头、实例数据、对齐填充。

对象头包括两部分信息：：对象本身的运行时数据，如哈希码、GC分代年龄、锁状态、线程持有的锁、偏向线程ID、偏向时间戳等；类型指针，即对象指向它的类元数据的指针，数组对象还要包括数组的大小。

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

对齐填充：HopSpot，对象其实地址必须是8字节的整数倍，当对象实例数据部分没有对齐时，就需要通过对齐填充来补全。相当于占位符。

#### 2.3.3 对象的访问定位

方式一：“句柄访问”，Java堆中划分出一块内存作为句柄池，存储对象的句柄地址，包含对象实例数据与类型数据各自的具体地址信息，如下图所示：

Java栈本地变量表

int

…

reference

…

到对象实例数据的指针

到对象类型数据的指针

堆

实例池

方法区

方式二：直接指针访问，如下图所示：

Java栈本地变量表

int

…

reference

…

Java堆

到对象类型数据的指针

对象实例数据

方法区

Sun HotSpot采用第二种方式。

### 2.4 实战：OutOfMemoryError异常

除了程序计数器外，虚拟机内存其它几个运行时区域都会发生OutOfMemoryError异常的可能。

#### 2.4.1 Java堆溢出

指定-Xms和-Xmx将堆的最小值和最大值都设置为20M，例如：在idea中设置VM options：-Xms20M -Xmx20M

|  |
| --- |
| **public class** HeapOOP {  **static class** OOMObject {   }   **public static void** main(String[] args) {  List<OOMObject> list = **new** ArrayList<>();  **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:3210)  at java.util.Arrays.copyOf(Arrays.java:3181)  at java.util.ArrayList.grow(ArrayList.java:261)  at java.util.ArrayList.ensureExplicitCapacity(ArrayList.java:235)  at java.util.ArrayList.ensureCapacityInternal(ArrayList.java:227)  at java.util.ArrayList.add(ArrayList.java:458)  at com.daily.dailytest.jvm.HeapOOP.main(HeapOOP.java:27)  Process finished with exit code 1 |

#### 2.4.2 虚拟机栈和本地方法栈溢出

HotSpot不区分虚拟机栈和本地方法栈。

▪ 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。

测试代码：

|  |
| --- |
| **public class** JavaVMStackSOF {  **private int stackLength** = 1;   **private void** stackLeak() {  **stackLength** ++;  stackLeak();  }   **public static void** main(String[] args) {  JavaVMStackSOF oom = **new** JavaVMStackSOF();  **try** {  oom.stackLeak();  } **catch** (Throwable e) {  System.***out***.println(**"stack length:"** + oom.**stackLength**);  **throw** e;  }  } } |

运行结果：

|  |
| --- |
| stack length:985  Exception in thread "main" java.lang.StackOverflowError  at com.daily.dailytest.jvm.JavaVMStackSOF.stackLeak(JavaVMStackSOF.java:20)  at com.daily.dailytest.jvm.JavaVMStackSOF.stackLeak(JavaVMStackSOF.java:21)  at com.daily.dailytest.jvm.JavaVMStackSOF.stackLeak(JavaVMStackSOF.java:21)  at com.daily.dailytest.jvm.JavaVMStackSOF.stackLeak(JavaVMStackSOF.java:21)  at com.daily.dailytest.jvm.JavaVMStackSOF.stackLeak(JavaVMStackSOF.java:21)  at com.daily.dailytest.jvm.JavaVMStackSOF.stackLeak(JavaVMStackSOF.java:21) |

在递归中可能会出现这个异常。

|  |
| --- |
| **public class** JavaVMStackOOM {  **private void** dontStop() {  **while** (**true**) {   }  }  **public void** stackLeakByThread() {  **while** (**true**) {  Thread thread = **new** Thread(**new** Runnable() {  @Override  **public void** run() {  dontStop();  }  });  thread.start();  }  } } |

运行结果：

#### 2.4.3 方法区和运行时常量池溢出

#### 2.4.4 本机直接内存溢出

### 2.5 本章小结

## 3 垃圾收集器与内存分配策略

### 3.1 概述

### 3.2 对象已死吗

#### 3.2.1 引用计数算法

#### 3.2.2 可达性分析算法

#### 3.2.3 再谈引用

#### 3.2.4 生存还是死亡

#### 3.2.5 回收方法区

### 3.3 垃圾收集算法

#### 3.3.1 标记-清除算法

#### 3.3.2 复制算法

#### 3.3.3 标记-整理算法

#### 3.3.4 分代收集算法

### 3.4 Hotspot的算法实现

#### 3.4.1 枚举根节点

#### 3.4.2 安全点

#### 3.4.3 安全区域

### 3.5 垃圾收集器

#### 3.5.1 serial收集器

#### 3.5.2 parnew收集器

#### 3.5.3 parallel scavenge收集器

#### 3.5.4 serial old收集器

#### 3.5.5 parallel old收集器

#### 3.5.6 cms收集器

#### 3.5.7 g1收集器

#### 3.5.8 理解gc日志

#### 3.5.9 垃圾收集器参数总结

### 3.6 内存分配与回收策略

#### 3.6.1 对象优先在Eden分配

#### 3.6.2 大对象直接进入老年代

#### 3.6.3 长期存活的对象将进入老年代

#### 3.6.4 动态对象年龄判定

#### 3.6.5 空间分配担保

### 3.7 本章总结

## 4 虚拟机性能监控与故障处理工具

### 4.1 概述

### 4.2 jdk的命令行工具

#### 4.2.1 jps：虚拟机进程状况工具

#### 4.2.2 jstat：虚拟机统计信息监视工具

#### 4.2.3 jinfo：java配置信息工具

#### 4.2.4 jmap：java内存映像工具

#### 4.2.5 jhat：虚拟机堆转储快照分析工具

#### 4.2.6 jstack：java堆栈跟踪工具

#### 4.2.7 hsdis：jit生成代码反汇编

### 4.3 jdk的可视化工具

#### 4.3.1 jconsole：java监视与管理控制台

#### 4.3.2 visualvm：多合

### 4.4 本章小结

## 5 调优案例分析与实战

### 5.1 概述

### 5.2 案例分析

#### 5.2.1 高性能硬件上的程序部署策略

#### 5.2.2 集群间同步导致的内存溢出

#### 5.2.3 堆外内存导致的溢出错误

#### 5.2.4 外部命令导致系统缓慢

#### 5.2.5 服务器jvm进程崩溃

#### 5.2.6 不恰当数据结构导致内存占用过大

#### 5.2.7 由windows虚拟内存导致的长时间停顿

### 5.3 实战：eclipse运行速度调优

#### 5.3.1 调优前的程序运行状态

#### 5.3.2 升级jdk1.6的性能变化及兼容问题

#### 5.3.3 编译时间和类加载时间的优化

#### 5.3.4 调整内存设置控制垃圾收集频率

#### 5.3.5 选择收集器降低延迟

### 5.4 本章小结

# Part 3. 虚拟机执行子系统

## 6.1 概述

## 6.2 无关性的基石

**一次编译，到处运行**

各种平台的虚拟机与所有平台都统一使用的程序存储格式----字节码(ByteCode)是构成“平台无关性”的基石。

设计者在发展之初，就刻意将Java的规范拆分为Java语言规范《The Java Language Specification》和Java虚拟机规范《The Java Virtual Machine Specification》，这为其它语言运行在Java虚拟机上提供了可能性。

在Java虚拟机发展到JDK1.7~1.8的时候，JVM设计者通过JSR-292基本上兑现了这个承若，如Clojure、Groovy、JRuby、Jython、Scala等刻意在Java虚拟机上运行。

实现语言无关性的基础仍然是虚拟机和字节码储存格式。Java虚拟机不和包括Java在内的任何语言绑定，它只与“Class”文件这种特定的二进制文件格式所关联，Class文件中包含了Java虚拟机指令集和符号表以及若干其他辅助信息，Java虚拟机规范要求在Class文件中使用许多强制性的语法和结构化约束，但任一门功能性语言都可以表示为一个能被Java虚拟机所接受的有效的Class文件。作为一个通用的、机器无关的执行平台，任何其他语言的实现者都可以将Java虚拟机作为语言的产品交付媒介。例如：使用Java编译器可以把Java代码编译为存储字节码的Class文件；使用JRuby等其他语言的编译器一样可以把程序代码编译成Class文件，虚拟机并不关心Class的来源是何种语言，如下图所示：

Java程序

(\*.java)

JRuby程序

(\*.rb)

Groovy程序

(\*.rb)

其它语言程序

javac编译器

jruby编译器

groovy编译器

对应的编译器

字节码

(\*.class)

字节码

(\*.class)

Java 语言中的各种变量、关键字和运算符的语义最终都是由多条字节码命令组合而成的，因此字节码命令所能提供的语义描述能力肯定会比Java语言本身更加强大。因此，有一些Java语言本身无法有效支持的语言特性不代表字节码本身无法有效支持，这也为其它语言实现一些有别于Java的语言特性提供了基础。

## 6.3类文件结构

本章关于Class文件结构的讲解中，将以《Java虚拟机规范(第二版)》(1994年发布，对应于JDK1.4时代的Java虚拟机)中的定义为主线，同时也会JDK1.5~JDK1.7中添加的内容为支线进行较为简略的、介绍性的讲解。

注意：任何一个Class文件都对应着唯一一个类或接口的定义信息，但反过来说，类或接口并不一定都得定义在文件里(如类和接口也可以通过类加载器直接生成)。

Class文件是一组以8位字节为基础的二进制流，各个数据项目严格按照顺序紧凑地排列在Class文件中，中间没有添加任何分隔符，使得整个Class文件中存储的内容几乎全部是程序运行的必要数据，没有空隙存在。当遇到需要占用8字节以上空间的数据项时，则会按照高位在前(Big-Endian)的方式分割成若干个8位字节进行存储。

根据Java虚拟机规定，Class文件格式采用一种类似于C语言结构体的伪结构来存储数据，这种伪结构中只有两种数据类型：无符号数和表，后面的解析都要以这两种数据类型为基础，所以这里要先介绍着两个概念。

无符号数属于基本的数据类型，以u1,u2,u4,u8来分别代表1个字节、2个字节、4个字节和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、数量值或者按照UTF-8编码构成字符串值。

表是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，所有表都习惯性地以“\_info”结尾。表用于描述有层次关系的复合的数据，整个Class文件本质上就是一张表，如下表所示的数据项构成：

|  |  |  |
| --- | --- | --- |
| 类型 | 名称 | 数量 |
| u4 | magic | 1 |
| u2 | minor\_version | 1 |
| u2 | major\_version | 1 |
| u2 | constant\_pool\_count | 1 |
| cp\_info | constant\_pool | constant\_pool\_count-1 |
| u2 | access\_flags | 1 |
| u2 | this\_class | 1 |
| u2 | super\_class | 1 |
| u2 | interfaces\_count | 1 |
| u2 | interfaces | interfaces\_count |
| u2 | fields\_count | 1 |
| field\_info | fields | fields\_count |
| u2 | methods\_count | 1 |
| method\_info | methods | methods\_count |
| u2 | attributes\_acount | 1 |
| attribute\_info | attributes | attributes\_count |

无论是无符号数还是表，当需要描述同一类型但数量不定的多个数据时，经常会使用一个前置的容量计数器加若干个连续的数据项的形式，这是称这一系列连续的某一类型的数据为某一类的集合。

Class的结构不像XML等描述语言，由于它没有任何分隔符号，所以无论是顺序还是数量，甚至数据存储的字节序(Byte Ordering,Class文件中字节序为Gig-Endian)这样的细节，都是被严格限定的，哪个字节代表什么含义，长度是多少，先后顺序如何，都不允许改变。接下来我们将一起看看这个表中各个数据项的具体含义。

### 6.3.1魔数与Class文件的版本

魔数(Magic Number)，每个Class文件的头4个字节称为魔数，唯一作用是确定这个文件是否为一个能被虚拟机接收的Class文件。Class文件的魔数值为：0xCAFEBABE(咖啡宝贝)还被称为“Oak”。

紧接着魔数的4个字节存储的是Class文件的版本号：第5和第6个字节是次版本号(Minor Version)，第7和第8个字节是主版本号(Major Version)。Java的版本号是从45开始的，JDK1.1之后的每个大版本主版本号向上加1，高版本的JDK能向下兼容以前版本的Class文件，但不能运行以后版本的Class文件，即是文件格式并未发生任何变化，虚拟机也必须拒绝执行超过其版本号的Class文件。

### 6.3.2常量池

常量池可以理解为Class文件之中的资源仓库，它是Class文件结构中与其它项目关联最多的数据类型，也是占用Class文件空间最大的数据项目之一，同时它还是在Class文件中第一个出现的表类型数据项目。

常量池大小不固定，入口需要放置一项u2类型的数据，代表常量池容量计数值。从1开始而不是从0开始，0项有特殊考虑，“不引用任何一个常量池的项目”时把索引值置为0来表示。Class文件结构中只有常量池的容量计数是从1开始，对于其它集合类型，包括接口索引集合、字段表集合、方法表集合等的容量计数都与一般习惯相同，是从0开始的。

常量池中主要存放两大类常量：字面量和符号引用。字面量比较接近于Java语言层面的常量概念，如文本字符串、声明为final的常量值等；符号引用则属于编译原理方面的概念，包括了以下三类常量：

▪ 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。

### 6.3.3访问标志

在常量池结束之后，紧接着的两个字节代表访问标志，这个标志用于识别一些类或者接口层次的访问信息，包括：这个Class是类还是接口；是否定义为public类型；是否定义了abstract类型；如果是类的话，是否被声明为final等。具体的标志位以及标志的含义如下图：

|  |  |  |
| --- | --- | --- |
| 标志名称 | 标志值 | 含义 |
| ACC\_PUBLIC | 0x0001 | 是否为public类型 |
| ACC\_FINAL | 0x0010 | 是否被声明为final，只有类可设置 |
| ACC\_SUPER | 0x0020 | 是否允许使用invokespecial字节码指令的新语义，invokespcial指令的语义在JDK1.0.2发生过改变，为了区别这条指令使用哪种语义，JDK1.0.2之后编译出来的类的这个标志都必须为真 |
| ACC\_INTERFACE | 0x0200 | 标识这是一个接口 |
| ACC\_ABSTRACT | 0x0400 | 是否为abstract类型，对于接口或者抽象类来说，此标志值为真，其它类值为假 |
| ACC\_SYNTHETIC | 0x1000 | 标识这个类并非由用户代码产生的 |
| ACC\_ANNOTATION | 0x2000 | 标识这是一个注解 |
| ACC\_ENUM | 0x4000 | 标识这是一个枚举 |

access\_flags中一共有16个标志位可以使用，当前只定义了其中8个，没有使用到的标志位要求一律为0。

### 6.3.4类索引、父类索引、接口索引集合

### 6.3.5字段表集合

### 6.3.6方法表集合

### 6.3.7属性表集合

## 6.4字节码指令简介

### 6.4.1字节码与数据类型

### 6.4.2加载和存储指令

### 6.4.3运算指令

### 6.4.4类型转换指令

### 6.4.5对象创建与访问指令

### 6.4.6操作数栈管理指令

### 6.4.7控制转移指令

### 6.4.8方法调用和返回指令

### 6.4.9异常处理指令

### 6.4.10同步指令

## 6.5共有设计和私有实现

## 6.6 class文件结构的发展

## 6.7 本章小结

## 7虚拟机类加载机制

### 7.1概述

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，这就是虚拟机的类加载机制。

与那些在编译时需要进行连接工作的语言不同，在J啊啊语言里面，类型的加载、连接和初始化过程都是在程序运行期间完成的。虽然类加载时稍微增加一些性能开销，但是为J啊啊应用程序提供高度的灵活性，Java天生可以动态扩展的语言特性就是依赖运行期动态加载和动态连接这个特点实现的。用户可以通过Java预定义的和自定义类加载器，让一个本地的应用程序可以在运行时从网络或其他地方加载一个二进制流作为程序代码的一部分，这种组装应用程序的方式目前已管饭应用于Java程序之中。从最基础的Applet、JSP到相对复杂的OGSI技术，都使用了Java语言运行期类加载的特性。

### 7.2类加载的时机

类从被加载到虚拟机内存中开始，到卸载出内存为止，整个生命周期包括：加载(Loading)、验证(Verification)、准备(Preparation)、解析(Resolution)、初始化(Initialization)、使用(Using)、卸载(Unloading) 7 个阶段。其中验证、准备、解析三个部分统称为连接(Linking)，这 7 个阶段的发生顺序如下图所示：

加载

Loading

连接(Linking)

验证

Verificat

准备

Preparation

解析

Resolution

初始化

Initialization

使用

Using

卸载

Unloading

类的声明周期

加载、验证、准备、初始化和卸载这5个阶段的顺序是确定的，，而解析阶段则不一定：它在某些情况下可以在初始化阶段之后再开始，这是为了支持Java语言的运行时绑定(动态绑定或晚期绑定)。

什么情况下开始类加载过程的第一阶段：加载？Java虚拟机规范中并没有进行强制约束，有虚拟机的具体实现来自由把握。但是对于初始化阶段，虚拟机规范则是严格规定了有且只有5中情况IXUS立即对类进行初始化(加载、验证、准备需在初始化之前开始)。

▪ 遇到new、getstatic、putstatic、或invokestatic这 4 条字节码指令时，如果类没有进行过初始化，则需要先触发其初始化。生成这 4 条指令的最常见的Java代码场景是：使用new关键字实例化对象的时候、读取或设置一个类的静态字段(被final修饰、已在编译期把结果放入常量池的静态字段除外)的时候，以及调用一个类的静态方法的时候。

▪ 使用java.lang.reflect包的方法对类进行反射调用的时候，如果类没有进行过初始化，则需要先触发其初始化。

▪ 当初始化一个类的时候，如果发现其父类还没有进行过初始化，则需要先触发其父类的初始化。

▪ 当虚拟机启动时，用户需要指定一个要执行的主类(包含main()方法的那个类)，虚拟机会先初始化这个主类。

▪ 当使用JDK1.7的动态语言支持时，如果一个java.lang.invoke.MethodHandle实例最后的解析结果REF\_getStatic、REF\_putStatic、REF\_invokeStatic的方法句柄，并且这个方法句柄所对应的类没有进行过初始化，则需要先触发其初始化。

### 7.3类加载的过程

#### 7.3.1加载

“加载”是“类加载”(Class Loading)过程的一个阶段，希望读者没有混淆这两个看起来很相似的名词。在加载阶段，虚拟机需要完成以下 3 件事情；

▪ 通过一个类的全限定名来获取定义此类的二进制字节流；

▪ 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；

▪ 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。

通过类的全限定名获取类的二进制字节流，虚拟机规范中没有指出从哪里获取此字节流，非常开放，例如：

▪ 从ZIP包中获取，这很常见，最终成为日后JAR、EAR、WAR格式的基础；

▪ 从网络中获取，这种场景最典型的应用就是Applet；

▪ 运行时计算生成，这种场景使用最多的就是动态代理技术，在java.lang.reflect.Proxy中就是用了ProxyGenerator.generateProxyClass来为特定接口生成形式为“\*$Proxy”的代理类的二进制字节流。

▪ 有其它文件生成，典型场景是JSP应用，即由JSP文件生成对应的Class类

▪ 从数据库中读取，这种场景相对少见些，例如有些中间件服务器(如 SAP Netweaver)可以选择把程序安装到数据库中来完成程序代码在集群间的分发。

相对于其它阶段类加载过程的其它阶段，一个非数组类的加载阶段(准确地说，是加载阶段中获取类的二进制字节流的动作)是开发人员可控性最强的，因为加载阶段既可以使用系统提供的引导加载器来完成，也可以有用户自定义的类加载器去完成，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)。

对于数组，情况有所不同，数组类本身不通过类加载器创建，而又Java虚拟机直接创建的。但数组类与类加载器仍然有很密切的关系，因为数组类的元素类型(Element Type, 指的是数组去掉所有维度的类型)最终是要靠类加载器去创建，一个数组类(下面成为C)创建过程就遵循以下规则：

▪ 如果数组的组件类型(Component Type，指的是数组去掉一个维度的类型)是引用类型，那就递归采用本节中定义的加载过程去加载这个组件类型，数组类将在加载该组件类型的类加载器的类名称空间上被标识(一个类必须与类加载器一起确定唯一性)。

▪ 如果数组的组件类型不是引用类型(如 int[]数组)，Java虚拟机将会把数组类标记为与引导类加载器关联。

▪ 数组类的可见性与他的组件类型的可见性一致，如果组件类型不是引用类型，那数组类的可见性将默认为public。

加载阶段完成后，虚拟机外部的二进制字节流就按照虚拟机所需的格式存储在方法区之中，方法区中的数据存储格式有虚拟机实现自行定义，虚拟机规范为规定此区域的具体数据结构。然后在内存中实例化一个java.lang.Class类的对象(并没有明确规定是在Java堆中，对于HotSpot虚拟机而言，Class对象比较特殊，它虽然是对象，但是存放在方法区里面)，这个对象将作为程序访问方法区中的这些类型数据的外部接口。

加载阶段与连接阶段的部分内容(如一部分字节码文件格式验证动作)是较差进行的，加载阶段尚未完成，连接阶段可能已经开始，但这些夹在加载阶段之中进行的动作仍然属于连接阶段的内容，这两个阶段的开始时间仍然保持着固定的先后顺序。

#### 7.3.2验证

1. 文件格式验证
2. 元数据验证
3. 字节码验证
4. 符号引用验证

#### 7.3.3准备

#### 7.3.4解析

1. 类或接口的解析
2. 字段解析
3. 类方法解析
4. 接口方法解析

#### 7.3.5初始化

### 7.4类加载器

虚拟机设计团队把类加载阶段中的“通过一个类的全限定名来获取描述此类的二进制字节流”这个动作放到Java虚拟机外部去实现，以便让应用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模块成为“类加载器”。

类加载器可以说是Java语言的一项创新，也是Java语言流行的重要原因之一，它最初是为了满足Java Applet的需求而开发出来的。虽然目前Java Applet技术基本上已经“死掉”，但类加载器却在类层次划分、OSGI、热部署、代码加密等领域大放异彩，成为了Java技术体系中一块总要的基石，可谓是失之桑榆收之东隅。

#### 7.4.1类与类加载器

类加载器只用于实现类的加载动作，但去哦用却不限于类加载阶段。任意一个类，都需要由加载它的类加载器和这个类本身一起确立其在Java虚拟机中的唯一性，每个类加载器，都拥有独立的类名称空间。如：比较两个类是否“相等”，只有在同一个Class文件和同一个类加载器加载的前提下才相等。

手动编写类加载器：

|  |
| --- |
| **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 = 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(**"com.daily.dailytest.jvm.ClassLoaderTest"**).newInstance();  System.***out***.println(obj.getClass());  System.***out***.println(obj **instanceof** com.daily.dailytest.jvm.ClassLoaderTest);  } } |

运行结果：

|  |
| --- |
| class com.daily.dailytest.jvm.ClassLoaderTest  true |

#### 7.4.2双亲委派模型

从Java虚拟机的角度来讲，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；另一种是所有其它的类加载器，这些类加载器都有Java语言实现，独立于虚拟机外部，并且全都继承自抽象类java.lang.ClassLoader。

*这里只限于HotSpot，向MRP、Maxine等虚拟机，整个虚拟机本身都由Java程序写的，自然Bootstrap ClassLoader也是由Java语言而不是C++实现的，退一步讲，除了HotSpot意外的其它两个高性能虚拟机JRockit和J9都有一个代表Bootstrap ClassLoader的Java类存在，但是关键方法的实现上仍然是使用JNI回调到C(注意不是C++)的实现上，这个Bootstrap ClassLoader的实例也无法被用户获取到。*

在Java开发人员的角度来看，类加载器还可以划分的更细致一些，绝大部分Java程序都会使用到以下3种系统提供的类加载器。

▪ 启动类加载器(Bootstrap ClassLoader)：这个类将负责将存放在<JAVA\_HOME>\lib目录中的，或者被-Xbootclasspath参数所指定的路径中，并且是虚拟机识别的(仅按照文件名识别，如rt.jar，名字不符合的类库即使放在lib目录中也不会被加载)类库加载到虚拟机内存中。启动类加载器无法被Java程序直接引用，用户在编写自定义类加载器时，如果需要把加载请求委派给引导类加载器，那直接使用null代替即可，如代码清单为：java.lang.ClassLoader.getClassLoader方法的代码片段：

|  |
| --- |
| **public** ClassLoader run() **throws** Exception {  String cls = System.*getProperty*(**"java.system.class.loader"**);  **if** (cls == **null**) {  **return parent**;  }   Constructor<?> ctor = Class.*forName*(cls, **true**, **parent**)  .getDeclaredConstructor(**new** Class<?>[] { ClassLoader.**class** });  ClassLoader sys = (ClassLoader) ctor.newInstance(  **new** Object[] { **parent** });  Thread.*currentThread*().setContextClassLoader(sys);  **return** sys; } |

▪ 扩展类加载器(Extension ClassLoader)：这个加载器由sun.misc.Launcher$ExtClassLoader实现，它负责加载<JAVA\_HOME>\lib\ext目录中的，或者被java.ext.dirs系统变量所指定的路径中的所有类库，开发者可以直接使用扩展类加载器。

▪ 应用程序类加载器(Application ClassLoader)：这个类加载器由sun.miscLauncher$App-ClassLoader实现。由于这个类加载器是ClassLoader中的getSystemClassLoader()方法的返回值，左移一般也称它为系统类加载器。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这个类加载器，如果应用程序中没有自定义过自己的类加载器，一般情况这个就是程序中默认的类加载器。

类加载器之间的层级关系如下图所示：

启动类加载器

Bootstrap ClassLoader

扩展类加载器

Extension ClassLoader

应用程序类加载器

Application ClassLoader

自定义类加载器

User ClassLoader

自定义类加载器

User ClassLoader

上图中类加载器之间的这种层次关系，成为类加载器的双亲委派模式(Parents Delegation Model)。双亲委托模式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。这里类加载器之间的父子关系一般不会以继承的关系来实现，而是都使用组合关系来复用父类加载器的代码。

双亲委派模式的工作过程是：如果一个类加载器收到了类加载器的请求，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加载。

使用双亲委托模式来组织类加载器之间的关系，有一个显而易见的好处就是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。同一个类在程序的各种类加载器环境中都是同一个类。

双亲委派模型对于保证Java程序的稳定运作很重要，代码如下所示：

|  |
| --- |
| **protected** Class<?> loadClass(String name, **boolean** resolve)  **throws** ClassNotFoundException {  **synchronized** (getClassLoadingLock(name)) {  *// First, check if the class has already been loaded* Class<?> c = findLoadedClass(name);  **if** (c == **null**) {  **long** t0 = System.*nanoTime*();  **try** {  **if** (**parent** != **null**) {  c = **parent**.loadClass(name, **false**);  } **else** {  c = findBootstrapClassOrNull(name);  }  } **catch** (ClassNotFoundException e) {  *// ClassNotFoundException thrown if class not found  // from the non-null parent class loader* }   **if** (c == **null**) {  *// If still not found, then invoke findClass in order  // to find the class.* **long** t1 = System.*nanoTime*();  c = findClass(name);   *// this is the defining class loader; record the stats* sun.misc.PerfCounter.*getParentDelegationTime*().addTime(t1 - t0);  sun.misc.PerfCounter.*getFindClassTime*().addElapsedTimeFrom(t1);  sun.misc.PerfCounter.*getFindClasses*().increment();  }  }  **if** (resolve) {  resolveClass(c);  }  **return** c;  } } |

如上图所示，先检查是否已经被加载过，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。如果父类加载失败，抛出ClassNotFoundException异常后，在调用自己的findClass()方法进行加载。

#### 7.4.3破坏双亲委派模型

## 8虚拟机字节码执行引擎

### 8.1 概述

#### 8.2 运行时栈帧结构

##### 8.2.1 局部变量

##### 8.2.2 操作数栈

##### 8.2.3 动态连接

##### 8.2.4 方法返回地址

##### 8.2.5 附加信息

#### 8.3 方法调用

##### 8.3.1 解析

##### 8.3.2 分派

##### 8.3.3 动态类型语言支持

#### 8.4 基于栈的字节码解释执行引擎

##### 8.4.1 解释执行

##### 8.4.2 基于栈的指令集与基于寄存器的指令集

##### 8.4.3 基于栈的解释器执行过程

#### 8.5 本章小结

## 9类加载及执行子系统的案例与实战

### 9.1 概述

### 9.2 案例分析

#### 9.2.1 tomcat：正统的类加载器架构

#### 9.2.2 osgi：灵活的类加载器架构

#### 9.2.3 字节码生成技术与动态代理的实现

#### 9.2.4 retrotranslator：跨越jdk版本

### 9.3 实战：自己动手实现远程执行功能

#### 9.3.1 目标

#### 9.3.2 思路

#### 9.3.3 实现

#### 9.3.4 验证

### 9.4 本站小结

# Part 4. 程序编译与代码优化

## 10早期(编译期)优化

### 10.1 概述

### 10.2 javac编译器

#### 10.2.1 javac的源码与调试

#### 10.2.2 解析与填充符号表

#### 10.2.3 注解处理器

#### 10.2.4 语义分析与字节码生成

### 10.3 java语法糖的味道

#### 10.3.1 泛型与类型擦除

#### 10.3.2 自动装箱、拆箱与遍历循环

#### 10.3.3 条件编译

### 10.4 实战：插入式注解处理器

#### 10.4.1 实战目标

#### 10.4.2 代码实现

#### 10.4.3 运行与测试

#### 10.4.4 其它应用案例

### 10.5 本章小结

## 11晚期(运行期)优化

### 11.1 概述

### 11.2 Hotspot虚拟机内的即时编译器

#### 11.2.1 解释器与编译器

#### 11.2.2 编译对象与触发条件

#### 11.2.3 编译过程

#### 11.2.4 查看及分析即时编译结果

### 11.3 编译优化技术

#### 11.3.1 优化技术概览

#### 11.3.2 公共子表达式消除

#### 11.3.3 数组边界检查消除

#### 11.3.4 方法内联

#### 11.3.5 逃逸分析

### 11.4 java与C/C++的编译器对比

### 11.5 本章小结

# Part 5. 高效并发

## 12 Java内存模型与线程

### 12.1概述

### 12.2硬件的效率一致性

### 12.3 java内存模型

#### 12.3.1 主内存与工作内存

#### 12.3.2 内存间交互操作

#### 12.3.3 对于volatile型变量的特殊规则

#### 12.3.4 对于long和double型变量的特殊规则

#### 12.3.5 原子性、可见性、有序性

#### 12.3.6 先行发生于那则

### 12.4 java与线程

#### 12.4.1 线程的实现

#### 12.4.2 java线程调度

#### 12.4.3 状态转换

### 12.5 本章小结

## 13 线程安全与锁优化

### 13.1概述

### 13.2线程安全

#### 13.2.1 java语言中的线程安全

#### 13.2.2线程安全的实现方法

### 13.3锁优化

#### 13.3.1自旋锁与自适应自旋

#### 13.3.2锁消除

#### 13.3.3锁粗化

#### 13.3.4轻量级锁

#### 13.3.5偏向锁

### 13.4 本章小结