|  |
| --- |
| Java 虚拟机 |
| chenhao |
| 2015.11.24 |

目录

[Java虚拟机 4](#_Toc436149308)

[第一章 Java内存区域 4](#_Toc436149309)

[1.1 运行时数据区域 4](#_Toc436149310)

[1.2 对象的创建 4](#_Toc436149311)

[1.3对象在内存中存储的布局 5](#_Toc436149312)

[1.4 对象访问定位 5](#_Toc436149313)

[第二章 异常处理 5](#_Toc436149314)

[2.1堆溢出 5](#_Toc436149315)

[2.2 栈溢出 6](#_Toc436149316)

[2.3方法区和运行时常量池溢出 6](#_Toc436149317)

[2.4本机直接内存溢出 6](#_Toc436149318)

[第三章 内存处理 6](#_Toc436149319)

[3.1判断对象生死 6](#_Toc436149320)

[3.2 引用种类 7](#_Toc436149321)

[3.3 方法区的垃圾回收 7](#_Toc436149322)

[3.4 垃圾收集算法 8](#_Toc436149323)

[3.5 垃圾回收中的重要名词 8](#_Toc436149324)

[3.6 垃圾收集器 9](#_Toc436149325)

[3.7 内存分配与回收策略 11](#_Toc436149326)

[C4虚拟机性能检测与故障处理 11](#_Toc436149327)

[C5调优案例分析 12](#_Toc436149328)

[5.1 案例分析 12](#_Toc436149329)

[第六章 类文件结构 12](#_Toc436149330)

[6.1 Class类文件的结构 12](#_Toc436149331)

[6.2 字节码指令简介 13](#_Toc436149332)

[第七章 虚拟机类加载机制 14](#_Toc436149333)

[7.1 概述 14](#_Toc436149334)

[7.2 类加载的时机 14](#_Toc436149335)

[第八章 虚拟机字节码执行引擎 18](#_Toc436149336)

[8.1 概述 18](#_Toc436149337)

[8.2 运行时栈帧结构 18](#_Toc436149338)

[8.3 方法调用 19](#_Toc436149339)

[8.4 基于栈的字节码解释执行引擎 21](#_Toc436149340)

[第九章 编译期优化 21](#_Toc436149341)

[9.1 概述 21](#_Toc436149342)

[9.2 Javac编译器 21](#_Toc436149343)

[9.3 Java语法糖 Syntactic Sugar 23](#_Toc436149344)

[第十章 运行期优化 24](#_Toc436149345)

[10.1 概述 24](#_Toc436149346)

[10.2 编译优化技术 25](#_Toc436149347)

[10.3 Java与C/C++编译器的对比 28](#_Toc436149348)

[第十一章 Java内存模型与线程 28](#_Toc436149349)

[11.1 缓存与乱序执行优化 28](#_Toc436149350)

[11.2 JMM Java内存模型 29](#_Toc436149351)

[11.3 volatile关键字 29](#_Toc436149352)

[11.4 JMM三大特性 30](#_Toc436149353)

[11.5 happens-before原则 30](#_Toc436149354)

[11.6 线程 31](#_Toc436149355)

[第十二章Java线程调度 32](#_Toc436149356)

[12.1 Cooperative Threads-Scheduling 协同式进程调度 32](#_Toc436149357)

[12.2 Preemptive Threads-Scheduling 抢占式进程调度 32](#_Toc436149358)

[12.3 状态转换 32](#_Toc436149359)

[第十三章 线程安全与锁优化 33](#_Toc436149360)

[13.1 线程安全的实现方法 34](#_Toc436149361)

[13.2 锁优化 35](#_Toc436149362)

# Java虚拟机

## 第一章 Java内存区域

### 1.1 运行时数据区域

线程隔离：

1.Program Counter Register **程序计数器**

线程所执行的字节码的行号指示器。

2.Java Virtual Machine Stacks Java**虚拟机栈**

* + 1. 每个方法在执行的时候都会创建一个Stack Frame栈帧，用于存储局部变量表、操作数栈、动态链接、方法出口灯信息。
    2. StackOverFlowError如果线程请求的栈深度大于虚拟机所允许的深度

3.Native Method Stack **本地方法栈**

与JVMStacks相似，但是为虚拟机使用到的Native方法服务。

线程共享：

1.Java Heap **堆**

* + 1. 是Java虚拟机所管理的内存中最大的一块，唯一目的就是存放对象实例。（栈上分配和标量替换优化技术将会导致一些微妙的变化发生）
    2. 可以处于物理上不连续的内存空间中，只要逻辑上是连续的即可。
    3. 可用过-Xmx和-Xms控制

2.Method Area **方法区**

* + 1. 存储已被虚拟机加载的类信息、常量、静态常量、即时编译器编译后的代码等数据。
    2. 别名Non-Heap 非堆 PermentGeneration 永久代
    3. 包含Runtime Constant Pool运行时常量池，用于存放编译期生成的各种字面量和符号引用，具备动态性，不要求常量一定只有编译器才能产生，比如String的intern

### 1.2 对象的创建

1. 分配内存；
2. 内存空间初始化为零（不包括对象头）；
3. 对对象进行必要的设置，将基本信息存放在对象的对象头中；
4. 调用<init>方法进行初始化

### 1.3对象在内存中存储的布局

1. Header 对象头

* 1. 存储对象本身的运行时数据，如HashCode、GC分布年龄等
  2. 类型指针，即对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例

1. Instance Data 实例数据

对象真正存储的有效信息，受到FieldsAllocationStyle（虚拟机分配策略参数）和字段在Java源码中定义顺序的影响。HotSpot虚拟机默认的分配策略是相同宽度的字段总是被分配在一起，父类中定义的变量会出现在子类之前，若CompactFields参数值为true，那么子类之中较窄的变量也会插入到父类变量的空隙之中。

1. Padding 对齐填充

占位符，因为要求对象的起始位置地址必须是8字节的整数倍，即对象的大小必须是8字节的整数倍。

### 1.4 对象访问定位

Java对象通过栈上的reference数据来操作栈上的具体对象，对象访问方式：

1. 句柄访问，堆中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，而句柄中包含了对象实例数据与类型数据各自的具体地址信息。
2. Reference中存储的是稳定的句柄地址，在对象被移动时只会改变句柄中的实例数据指针，而reference本身不需要修改。
3. 直接指针访问，堆对象的布局中放置访问类型数据，reference中存储的直接就是对象地址。
4. 速度更快，节省了依次指针定位的时间开销

## 第二章 异常处理

### 2.1堆溢出

-Xms –Xmx指令调整堆的最小值和最大值

OutOfMemoryError:Java heap space

1. 先通过内存镜像分析工具（Eclipse Analyzer）对Dump出来的堆转储快照进行分析，确认是Memory Leak还是Memory OverFlow
2. 如果是内存泄露，查看泄露对象到GC ROOTS的引用链，找到泄露对象是通过怎样的路径与GC Roots相关联并导致GC无法自动回收他们的。
3. 如果不存在内存泄露，就检查虚拟机的堆参数（-Xms -Xmx），与物理机器对比看是否还以可以调大，从代码上检查是否存在某些对象生命周期过长、持有时间过长的情况。

### 2.2 栈溢出

-Xoss或-Xss -Xss 128k

1. 线程请求的栈深度大于虚拟机所允许的最大深度，抛出StackOverflowError（单线程无限递归）

2. 虚拟机在扩展栈时无法申请到足够的内存空间，抛出StackOverflowError：unable to create new native thread（多线程无线递归）

### 2.3方法区和运行时常量池溢出

-XX:PermSize -XX:MaxPermSize限制方法区大小

Intern方法：将String对象的字符串添加到常量池中，并且返回此String对象的引用。JDK1.7之后是记录首次出现的实例的引用

OutOfMemory：PermGen space

### 2.4本机直接内存溢出

DirectMomery

通过-XX：MaxDirectMemorySize指定，默认值与-Xms一样

## 第三章 内存处理

### 3.1判断对象生死

#### 3.1.1 ReferenceCounting引用计数法

当被引用时，计数器值加1，当引用失效时，计数器值减1

弊端：对象间相互循环引用时

#### 3.1.2 可达性分析算法

通过一系列的称为“GC Roots”的对象称为起始点，从这些节点开始向下搜索，走过的路径称为ReferenceChain（引用链）。

GC Roots包含：

1. 虚拟机栈（栈帧中的本地变量表）中引用的对象
2. 方法区中类静态属性引用的对象
3. 方法区中常量引用的对象
4. 本地方法栈中JNI（即一般说的Native方法）引用的对象

旁门小道：可以通过finalize()方法进行自救，即在finalize方法中与GC Roots建立连接，但是此法只能一次自救，因为finalize方法至多只会被调用一次。

### 3.2 引用种类

当内存空间还足够时，能够保存在内存之中；如果内存空间在进行垃圾收集后还是非常紧张，则可以抛弃这些对象。很多系统的缓存功能都符合这样的应用场景。

1. **强引用**

只要强引用还存在，垃圾收集器永远不会回收掉引用的对象（Object obj = new Object()）

1. **软引用**

用来描述一些还有用但并非必须的对象。在系统将要发生内存溢出异常之前，将会把这些对象列进回收范围之中进行第二次回收。如果这次回收还没有足够的内存，才会抛出内存溢出异常。

SoftReference类实现软引用。

1. **弱引用**

被弱引用关联的对象智能生存到下一次GC发生之前。当垃圾收集器工作时，无论当前内存是否足够，都会回收到只被弱引用关联的对象。

WeakReference类实现弱引用

1. **虚引用**

幽灵引用、幻影引用

是最弱的一种引用关系。一个对象是否有虚引用不会对其生存时间构成影响，也无法通过虚引用来取得一个对象实例。为一个对象设置虚引用关联的唯一目的就是能在这个对象被收集器回收时收到一个系统通知。

PhantomReference类实现虚引用

### 3.3 方法区的垃圾回收

在堆中，尤其是新生代中，常规应用进行一次垃圾收集一般可以回收七成以上空间，而方法区的垃圾收集效率远低于此。

主要回收两部分内容：废弃常量和无用的类。

1. **废弃常量**：与回收Java堆中的对象相似。以常量池中字面量的回收为例，假如字符串aaa已进入常量池，但是当前系统中没有任何一个String对象是叫做aaa的，如果这时候发生了GC且有必要的话将会将常量清理出常量池。
2. **无用类**：当满足以下三个条件时可以被回收
   * + 1. 该类所有的实例都已经被回收，也就是Java堆中不存在该类的任何实例
       2. 加载该类的ClassLoader已经被回收；
       3. 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

### 3.4 垃圾收集算法

#### 3.4.1 Mark-Sweep算法

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。

不足：标记和清除效率不高且容易产生大量不连续的内存碎片

#### 3.4.2 Copying算法

将内存容量分为大小相等的两块，每次只使用其中的一块。当这一块内存用完了，就将还存活着的对象赋值到另外一块上面，然后再把已使用过的内存空间一次清理掉。

这样使得每次都是对整个半区进行内存回收，内存分配时也就不用考虑内存碎片等情况。

代价是将内存缩小为了原来的一半。

扩展：

因为新生代中的对象98%都是朝生暮死的，所以并不需要按照1:1来进行内存空间划分，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。回收时将其还存活的对象一次性复制到另一块Survivor上，最后清理掉Eden和刚才用过的Survivor空间。

默认Eden和Survivor的大小比例是8：1

当然不能保证每次回收都只有一成以下对象存活，当Survivor空间不够用时，需要依赖其他内存（老年代）进行**Handle Promotion**（分配担保）

#### 3.4.3 Mark-Compact算法

标记后让所有或者的对象都向一端移动，然后直接清理掉端边界以外的内存。

#### 3.4.4 Generational Collection算法

因材施教，新生代选用Copying算法，老生代选用Mark-Compact算法。

### 3.5 垃圾回收中的重要名词

#### 3.5.1 枚举根节点

GC进行时需要停顿所有Java执行线程，为了使其不必要一个不漏的检查完所有执行上下文和全局的引用位置，引入OopMap记录哪些地方存放着哪些引用。

#### 3.5.2 SafePoint安全点

程序执行时并非在所有的地方都能停顿下来进行GC，只有在到达安全点时才能暂停。

安全点的选定基本上是以程序“是否具有让程序长时间执行的特征”为标准进行选定的。最明显特征就是指令序列复用，例如方法调用、循环跳转、异常跳转等。

#### 3.5.3 SafeRegion安全区域

当程序不执行，即程序没有分配CPU时间，比如线程处于Sleep状态或者Blocked状态时，无法走入safePoint，所以需要SafeRegion来进行GC。

### 3.6 垃圾收集器

Young Generation： Serial、ParNew、ParallelScavenge

Gl

Tenured Generation: CMS 、Serial Old（MSC）、ParallelOld

#### 3.6.1 Serial收集器

单线程收集器，在进行垃圾收集时，必须暂停其他所有的工作线程，直到它收集结束。

简单而高效。对于运行在Client模式下的虚拟机来说是一个很好的选择。

#### 3.6.2 ParNew收集器

实际上就是Serial收集器的多线程版本，

#### 3.6.3 Parallel Scavenge收集器

目的是达到一个可控制的吞吐量。适合在后台运算而不需要太多交互的任务。

#### 3.6.4 Serial Old收集器

是Serial收集器的老年代版本。

#### 3.6.5 Parallel Old收集器

是Parallel Scavenge收集器的老年代版本。

#### 3.6.6 CMS收集器 Concurrent Mark Sweep

以获取最短回收停顿时间为目标的收集器。适用于重视服务的响应速度，希望**系统停顿时间最短**，以给用户带来较好的体验。比如互联网站或B/S系统的服务端。

主要分四步：

1. Initial mark 初始标记

标记GC Roots能直接关联到的对象。

1. Concurrent mark 并发标记

进行GC Roots Tracing的过程。

1. Remark 重新标记

修正b期间因为用户程序继续运作而导致标记产生的变动

1. Concurrent sweep 并发清除

初始标记、重新标记仍然需要Stop the world。

并发收集、低停顿。

缺点：

1. 对CPU资源敏感（面向并发设计的程序都对CPU资源比较敏感，在并发阶段会导致应用程序变慢，总吞吐量降低；
2. 无法处理浮动垃圾。

Floating Garbage是指标记过程之后出现的新垃圾

1. 内存碎片

#### 3.6.7 G1 收集器 Garbage-First

面向服务端应用的垃圾收集器。

* + - 1. 并行与并发

缩短StopTheWorld的停顿时间

* + - 1. 分代收集
      2. 空间整合

整体上基于标记-整理算法，局部上基于复制算法

使得收集后能提供规整的可用内存

* + - 1. **可预测的停顿**
      2. 化整为零

将整个Java堆划分为多个大小相等的独立区域Region

步骤：

1. Initial Marking 初始标记
2. Concurrent Marking 并发标记
3. Final Marking 最终标记
4. LiveDataCounting and Evacuation 筛选回收

对各个Region的回收价值和成品进行排序，根据用户所期望的GC停顿时间来制定回收计划。

### 3.7 内存分配与回收策略

1. 对象优先在Eden区分配
2. 大对象直接进入老年代

大对象指需要大量连续内存空间的Java对象，比如很长的字符串以及数组。

这样做避免在Eden区及两个Survivor区之间发生大量的内存复制

1. 长期存活的对象将进入老年代

设置MaxTenuringThreshold

1. 动态对象年龄判定

如果在Survivor空间中相同年龄所有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。

1. 空间分配担保

发生MinorGC之前，虚拟机会先检查老年代最大可用的连续空间是否大于新生代所有对象总空间，以此确保安全性。

## C4虚拟机性能检测与故障处理

|  |  |
| --- | --- |
| 名称 | 主要作用 |
| **jps** | JVM Process Status Tool  显示指定系统内所有的HotSpot虚拟机进程 |
| **jstat** | JVM Statistics Mornitoring Tool  用于收集HotSpot虚拟机各方面的运行数据 |
| **jinfo** | Configuration Info for Java  显示虚拟机配置信息 |
| **jmap** | Memory Map for Java  生成虚拟机的内存转储快照（heapdump文件） |
| **jhat** | JVM Heap Dump Browser  用于分析dump文件，会建立一个HTTP/HTML服务器，让用户可以在浏览器上看分析结果 |
| **jstack** | Stack Trace for Java  显示虚拟机的线程快照 |

|  |  |
| --- | --- |
| JDK下的可视化工具 |  |
| JConsole  Java Monitoring and Management Console |  |
| VisualVM  All-in-One Java Troubleshooting Tool |  |

## C5调优案例分析

### 5.1 案例分析

#### 5.1.1 高性能硬件上的程序部署策略

过大的堆内存进行垃圾回收时会带来长时间的停顿。

在高性能硬件上部署程序，主要有两种方式：

1. 使用64位JDK来使用大内存
2. 使用若干个32位虚拟机建立逻辑集群来利用硬件资源

##### 5.1.1.1 使用64位JDK来使用大内存

1. 需要控制Full GC频率，保障老年代空间的稳定，不能有成批量的、长生存时间的大对象产生
2. 内存回收导致长时间停顿
3. 现阶段64位JDK性能测试结果普遍低于32位
4. 需要保证程序足够稳定，因为如果产生堆溢出几乎就无法产生堆转储快照
5. 因为指针膨胀和数据类型对齐等元素，相同程序在64位JDK消耗的内存比32位大

##### 5.1.1.2 使用若干个32位虚拟机建立逻辑集群来利用硬件资源

在一台物理机器上启动多个应用服务器进程，每个服务器进程分配不同端口，然后在前端搭建一个负载均衡器，以反向代理的方式来分配访问请求。

1. 尽量避免节点竞争全局的资源，最典型的是磁盘竞争，容易导致IO异常
2. 很难最高效率的利用某些资源池
3. 节点仍然不可避免的收到32位的内存限制
4. 大量使用本地缓存（比如HashMap作为K/V缓存）的应用，在逻辑集群中会造成内存浪费，需要考虑把本地缓存改为集中式缓存。

## 第六章 类文件结构

实现语言无关性的基础是虚拟机和字节码存储格式，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。

### 6.1 Class类文件的结构

任何一个Class文件都对应着唯一一个类或接口的定义信息，但类或接口并不一定都得定义在文件里（可以通过类加载器直接生成）。只是通俗的将任意一个有效的类或接口所应当满足的格式称为“Class文件格式”。

1. Class文件是一组以8位字节为基础单元的二进制流，各个数据项目紧凑排列，无分隔符；
2. 只有两种数据类型：无符号数和表
3. 无符号数可以用来描述数字、索引应用、数量值或者按照UTF-8编码构成字符串值
4. 表是由多个无符号数或者其他表作为数据项构成的符合数据类型，用于描述有层次关系的符合结构的数据

主要形式：

1. 每个Class文件的前四个字节称为Magic Number，0xCAFEBABE
2. 2字节的Minor Version，2字节的Major Version
3. 常量池

因为常量数量不确定，所以在入口需要防止一项u2类型的数据，代表容量。计数为从1开始。

主要保存：

1. 字面量 Literal

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

1. 符号引用 Symbolic References

属于编译原理方面的概念，包括 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

1. 2字节代表访问标志，用于识别一些类或者接口层次的访问信息
2. 类索引、父类索引、接口索引集合
3. 类索引用来确定这个类的全限定名
4. 父类索引用来确定父类的全限定名
5. 接口索引集合用来描述这个类实现了哪些接口
6. 字段表集合

字段表用于描述接口或者类中声明的变量

1. 方法表集合
2. 属性表集合
3. Code属性

Args\_size在非static情况下回包含this

### 6.2 字节码指令简介

#### 6.2.1 字节码指令

Java虚拟机的指令由Opcode（操作码）和Operands（操作数）构成，Java虚拟机采用面向操作数栈而不是寄存器的架构，因此大多数的指令都不包含操作数，唯有操作码。

字节码指令集，操作码总数不可能超过256条，Class文件格式放弃了编译后代码的操作数长度对齐，意味着虚拟机处理那些超过1个字节数据的时候，不得不在运行时从字节中重建出具体数据的结构。但是这样使得编译代码短小精干。

Load:

将一个局部变量加载到操作栈

局部变量→操作栈

Store：

将一个数值从操作数栈存储到局部变量表

操作数栈→局部变量表

Push Tconst：

将一个变量加载到操作数栈

→操作数栈

#### 6.2.2 类型转换指令

1. Widening Numeric Conversions

宽化类型转换，即小范围类型向大范围类型的安全转换

int到long、float、double

long到float、double

float到double

1. Narrowing Numeric Conversions

窄化类型转换，比如显式的使用转换指令来完成，可能导致转换结果产生不同的正负号、不同的数量级的情况，很可能会导致数值的精度丢失。但不会导致虚拟机抛出异常。

如果浮点值是NaN，转换的int或long就是0

如果浮点值不是无穷大的话，使用向零舍入模式取整，如果该整数在目标类型的表示范围之内，就直接转换，否则根据符号转换为所能表示的最大或最小正数。

## 第七章 虚拟机类加载机制

### 7.1 概述

代码编译的结果从本地机器码转换为字节码，是存储格式发展的一小步，却是编程语言发展的一大步。

虚拟机的类加载机制，就是把描述类的数据从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。

Java中类型的加载、连接和初始化过程都是在程序运行期间完成的，虽然会令类加载时稍微增加一些性能开销，但是会为Java应用程序提供高度的灵活性。如：如果编写一个面向接口的应用程序，可以等到运行时再指定其实际的实现类，也可以通过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。

### 7.2 类加载的时机

类从被加载到虚拟机内存中开始，到卸载出内存位置，整个生命周期包括：

1. 加载 Loading
2. 验证 Verification
3. 准备 Preparation
4. 解析 Resolution
5. 初始化 Initialization
6. 使用 Using
7. 卸载 Unloading

其中验证、准备、解析三个部分统称为**连接**。

这些阶段通常都是互相交叉混合式的进行的，通常会在一个阶段执行的过程中调用、激活另一个阶段，故解析和使用的顺序是不一定的。

JVM规定有且只有五种情况必须立即对类进行初始化：

1. 遇到new、getstatic、putstatic、invokestatic这四条字节码指令时，如果类没有进行过初始化，则需要先触发其初始化；
2. 使用java.lang.reflect包的对象对类进行反射调用的时候
3. 初始化一个类的时候它的父类
4. 包含main方法的主类在虚拟机启动时要先进行初始化
5. 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。

一些常见现象：

1. 通过子类引用父类的静态字段，不会导致子类初始化。

对于静态字段，只有直接定义这个字段的类才会被初始化。

1. 通过数组定义来引用类，不会触发此类的初始化,将会触发一个新类的初始化，”[XXX”

Java语言中对数组的访问比C\C++相对安全是因为这个类封装了数组元素的访问方法，而C/C++直接翻译为对数组指针的移动

1. 常量在编译阶段会调入类的常量池中，本质上并没有直接饮用到定义常量的类，因此不会触发定义常量的类的初始化

编译期会由常量传播优化，若A类饮用了B类的常量c，则已经在编译阶段将c的值存储到了A类的常量池中

1. 接口也有初始化过程，与类有所区别的是接口在初始化时不要求父接口也完成了初始化，只有在真正使用到父接口的时候（如引用接口中定义的常量）才会初始化。

#### 7.2.1 类加载过程

##### 7.2.1.1 加载

1. 通过一个类的全限定名来获取定义此类的二进制字节流；
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

##### 7.2.1.2 验证

目的是确保Class文件的字节流中包含的信息符合当前虚拟机的要求，是虚拟机对自身保护的一项重要工作

##### 7.2.1.3 准备

正式为类变量分配内存并设置类变量初始值的阶段，这些变量所使用的内存将在方法区进行分配。

不包括实例变量，而且只是设置初始值并没有进行赋值，通常情况下是零值。如果类字段的字段属性表中存在ConstantValue属性，那在准备阶段变量value就会被初始化为ConstantValue所指定的值（final static）

##### 7.2.1.4 解析

解析阶段是虚拟机将常量池内的符号引用替换为直接引用的过程。

Symbolic References 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，只要使用时能无歧义的定位到目标即可。与虚拟机实现的内存布局无关。

Direct References 直接引用

直接指向目标的指针、相对偏移量或一个能间接定位到目标的句柄。和虚拟机实现的内存布局相关。

包括：

类或接口的解析；

字段解析；

类方法解析；

##### 7.2.1.5 初始化

真正开始执行类中定义的Java程序代码（字节码）。即执行类构造器<clinit>()方法的过程。

关于clinit方法

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由语句在与那文件中出现的顺序所决定的。
2. 与类的构造方法不同，不需要显示的调用父类构造器，虚拟机会保证；
3. 对于类或接口来说不是必须的‘
4. 接口中不能使用静态语句块，但仍有变量初始化的赋值操作，因此接口与类一样都会生成clinic方法。但接口clinit方法不需要先执行父接口的clinit方法，只有当父接口中定义的变量使用时，才会初始化父接口；
5. 会保证一个类的clinic方法在多线程环境中被正确的加锁、同步，如果多个线程同时去初始化一个类，那么只有一个线程去执行这个类的clinic方法，其他线程都需要阻塞等待。

#### 7.2.2 类加载器

通过一个类的全限定名来获取描述此类的二进制字节流，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称为“类加载器”。

##### 7.2.2.1 类与类加载器

对于任意一个类，都需要由加载它的类加载器和这个类本身一同确立其在Java虚拟机中的唯一性。这里的唯一性包括代表Class对象的equal()、isAssignableForm()、isInstance()方法，也包括instanceof关键字对对象所属关系判定的情况。

##### 7.2.2.2 双亲委派模型

1. 启动类加载器 Bootstrap ClassLoader
2. 扩展类加载器 Extension ClassLoader
3. 应用程序类加载器 Application ClassLoader

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,而是把请求委派给父类加载器去完成。只有当父加载器反馈自己无法完成这个加载请求，子加载器才会尝试自己去加载。

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。比如Object类在程序的各种类加载器环境中都是同一个类。

##### 7.2.2.3 破坏双亲委派模型

1. 向ClassLoader中添加findClass方法，当loadClass方法的逻辑里如果父类加载失败，则会调用自己的findClass方法完成加载
2. 由于自己缺陷：无法调用用户的代码，比如JNDI服务

引入了线程上下文加载器

1. 由于用户对程序动态性的追求所致，比如代码热替换Hotswap，模块热部署HotDeployment等

## 第八章 虚拟机字节码执行引擎

### 8.1 概述

执行引擎是Java虚拟机**最核心**的组成部分之一，在不同的虚拟机实现里，执行引擎在执行Java代码的时候可能会有解释执行（通过解释器执行）和编译执行（通过即时编译器产生本地代码执行）两种选择。

但是从外观上看，所有的Java虚拟机的执行引擎都是一致的：输入的是字节码文件，处理过程是字节码解析的等效过程，输出的是执行结果。

### 8.2 运行时栈帧结构

Stack Frame是用于支持虚拟机进行方法调用和方法执行的数据结构，是虚拟机运行时数据区中的虚拟机栈的栈元素。

栈帧存储了方法的局部变量表、操作数栈、动态连接方法和方法返回地址等信息。每一个方法从调用开始至执行完成的过程，都对应着一个栈帧在虚拟机栈里面从入栈到出栈的过程。

栈帧的大小在编译程序代码的时候已经确定，并且写入到方法表的Code属性之中。

执行引擎的所有字节码指令都只针对当前栈帧进行操作。

#### 8.3 局部变量表 Local Variable Table

存放方法参数和方法内部定义的局部变量。

Slot：变量槽，局部变量表的容量以变量草为最小单位。

局部变量表中的Slot是可以重用的，方法体中定义的变量，其作用域不一定会覆盖整个方法区，如果当前字节码PC计数器的值已经超出了某个变量的作用域，那么这个变量对应的Slot就可以交给其他变量使用。但是Slot的复用有时候会直接影响到系统的垃圾收集行为。所以Practice Java中把“不适用的对象应手动赋值为null”作为推荐的编码规则（对象占用内存大、方法的栈帧长时间不能被回收、方法调用次数达不到JIT的编译条件极端条件下），但更优雅的解决方法是用恰当的变量作用域来控制变量回收时间。

类变量有两次赋初始值的过程，而局部变量定义了但没有赋初始值是不能使用的。

#### 8.4 操作数栈 Operand Stack

最大深度在编译的时候就已经写入到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。

操作数栈中元素的数据类型必须与字节码指令的序列严格匹配。

在概念模型中，两个栈帧作为虚拟机栈的元素是完全独立的。但是在大多数虚拟机的实现里都会做一些优化处理，令两个栈帧出现一部分重叠，这样在进行方法调用时就可以共用一部分数据，无须进行额外的参数复制传递。

#### 8.5 动态链接 Dynamic Linking

每个栈帧都包含了一个指向运行时常量池中该栈帧所属方法的引用，持有这个引用就是为了支持方法调用过程中的动态链接。

Class文件的常量池中存有大量的符号引用，字节码中的方法调用指令就是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会在类加载阶段或者第一次使用的时候就转化为直接引用，这种转化称为静态解析。另外一部分将在每一次运行期间转化为直接引用，这部分称为动态链接。

#### 8.6 方法返回地址

方法开始执行后，只有两种方式可以退出这个方法：

1. 正常完成出口Normal Method Invocation Completion

执行引擎遇到任意一个方法返回的字节码指令

1. 异常完成出口 Abrupt Method Invocation Completion

方法执行过程中遇到了异常，并且这个异常没有在方法体内得到处理

### 8.3 方法调用

方法调用阶段唯一的任务就是确定被调用方法的版本，还不涉及方法内部的具体运行过程。

#### 8.3.1 解析 Resolution

调用在程序代码写好、编译器进行编译时就确定下来的方法，将这些方法的符号引用转换为直接引用。

静态方法、私有方法、实例构造器、父类方法、final方法。称为非虚方法，其他的方法被称为虚方法

#### 8.3.2 分派

##### 8.3.2.1 静态分派（重载Overload）

Human man = new Man();

Human为静态类型(Static Type)，或外观类型(Apparent Type)

Man为实际类型(Actual Type)

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，而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。

依赖静态类型来定位方法执行版本的分派动作称为静态分配，静态分配的典型应用是方法的重载。虚拟机（更准确的说是编译器）在重载时是通过参数的静态类型而不是实际类型作为判断依据的。

##### 8.3.2.2 动态分派（重写 Override）

在运行期根据实际类型确定方法执行版本的分派过程。

invokevirtual指令的运行时解析原理：找到操作数栈顶的第一个元素所指向的对象的实际类型，然后从下往上进行回溯。

虚拟机动态分配最常用的“稳定优化”手段就是为类在方法区中建立一个虚方法表，使用虚方法表索引来代替元数据查找以提高性能。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毕。

##### 8.3.2.3 单分派与多分派

方法的接收者与方法的参数统称为方法的宗量，根据分派基于多少种宗量，可以将分派划分为单分派和多分派两种。单分派是根据一个宗量对目标方法进行选择，多分派则是根据多于一个宗量对目标方法进行选择。

现在的Java语言是一门静态多分派、动态单分派的语言

#### 8.3.3 动态类型语言支持

##### 8.3.3.1 MethodHandle与Reflection区别

1. 从本质上讲都是在模拟方法调用，但reflection是在模拟Java代码层次的方法调用，而MethodHandle是在模拟字节码层次的方法调用，MethodHandles.lookup中的三个方法findStatic(),findVirtual(),findSpecial()正是为了对应于invokestatic、invokevirtual、invokeinterface和invokespecial这几条字节码指令的执行权限校验行为
2. Reflection中的Method对象远比MethodHandle对象所包含的信息多，前者是方法在Java一端的全面影像，包含了方法的签名、描述符以及方法属性表中的各种属性的Java端表示方式。而后者仅仅包含了与执行该方法相关的信息
3. 由于MethodHandle是对字节码的方法指令调用的模拟，所以理论上虚拟机在这方面做的各种优化，在MethodHandle上也应当采用类似思路去支持。

##### 8.3.3.2 Invokedynamic

### 8.4 基于栈的字节码解释执行引擎

#### 8.4.1解释执行

只有确定了谈论对象是某种具体的Java实现版本和执行引擎运行模式时，谈解释执行还是编译执行才会比较确切。

在执行前先对程序源码进行词法分析和语法分析处理，把源码转化为抽象语法树。

#### 8.4.2基于栈的指令集与基于寄存器的指令集

两套指令集各有优势。

基于栈的指令集：

1. 可移植，寄存器由硬件直接提供，程序直接依赖会受到硬件的约束
2. 可以由虚拟机把一些访问最频繁的数据放到寄存器中以获取更好性能
3. 代码相对紧凑
4. 编译器实现更加简单（不需考虑空间分配的问题）
5. 执行速度会慢一些，完成相同指令的操作数会多一些
6. 会有频繁的内存访问

## 第九章 编译期优化

### 9.1 概述

编译期其实是一段不确定的操作过程。

1. 前端编译器

\*.java转化为\*.class

1. JIT编译器

字节码转变为机器码

1. AOT编译器

直接把\*.java编译成本地机器代码

### 9.2 Javac编译器

1. 解析与填充符号表过程

Parse and Enter

1. 插入式注解处理器的注解处理过程

Annotation Processing

1. 分析与字节码生成过程

Analyse and Generate

#### 9.2.1 Parse and Enter 解析与填充符号表

1. **词法分析**

将源代码的字符流转变为标记集合Token，单个字符是程序编写的最小元素，而标记则是编译过程的最小元素（关键字、变量名、字面量、运算符）

如int a = b + 2 有6个标记

1. **语法分析**

根据Token序列构造抽象语法树（Abstract Syntax Tree）的过程，抽象语法树是一种用来描述程序代码语法结构的树形表示方式，语法树的每一个节点都代表着程序代码中的一个语法结构，如包、类型、修饰符、运算符、接口、返回值甚至代码注释。

1. **填充符号表**

符号表是一组由符号地址和符号信息构成的表格。在语义分析中，符号表所登记的内容将用于语义检查和产生中间代码。在目标代码生成阶段，当对符号名进行地址分配时，符号表是地址分配的依据。

#### 9.2.2 注解处理器

注解与普通的Java代码一样，是在运行期间发挥作用的。JDK提供了一组插入式注解处理器的标准API在编译期间对注解进行处理，可以把它看作的一组编译器的插件，可以读取、修改、添加抽象语法树的任意元素。如果这些插件在处理注解期间对语法树进行了修改，编译器将回到解析及填充符号表的过程重新处理，直到所有插入式注解处理器都没有再对语法树进行修改为止。

#### 9.2.3 语义分析与字节码生成

语法树能表示一个结构正确的源程序的抽象，但无法保证源程序是符合逻辑的。而语义分析的主要任务是对结构上正确的源程序进行上下文有关性质的审查。

1. **标注检查**

包括诸如变量使用前是否已被声明、变量与赋值之间的数据类型是否能够匹配等。

还包括**常量折叠**

即int a = 1+2 经过折叠后与int a = 3一致，减少程序运行期的运算量

1. **数据及控制流分析**

是对程序上下文逻辑更进一步的验证，可以检查出诸如程序局部变量在使用前是否有赋值、方法的每条路径是否都有返回值、是否所有的受查异常都被正确处理等问题。

1. **解语法糖**

Syntactic Sugar，**语法糖** 指在计算机语言中添加的某种语法，这种语法对计算机的性能并没有什么影响，但是更方便程序员使用。可以增加程序的可读性，减少程序代码出错的机会。

如：Java中的泛型、变长参数、自动拆装箱等

1. **字节码生成**

不仅仅把前面各个步骤所生成的信息（语法树、符号表）转化成字节码写到磁盘中，编译器还进行了少量的代码添加和转换工作。如实例构造器<init>()方法和类构造器<clinit>()方法

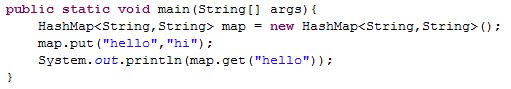
### 9.3 Java语法糖 Syntactic Sugar

#### 9.3.1 泛型与类型擦除

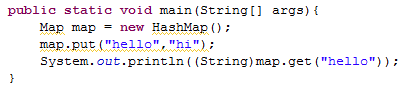
本质是**参数化类型**（Parametersized Type）的应用，也就是说所操作的数据类型被指定为一个参数。这种参数类型可以用在类、接口和方法的创建中，分别称为泛型类、泛型接口和泛型方法。

C#的泛型无论在程序源码中、编译后的中间语言（IL,Intermediate Language）中，还是在运行期的CLR中，都是切实存在的，List<int>与List<String>就是两个不同的类型，他们的系统运行期生成，有自己的虚方法表和类型数据，这种实现称为**类型膨胀**。

Java语言中的泛型只在程序源码中存在，在编译后的字节码文件中，经替换为原来的原生类型（Raw Type），并且在响应的地方插入了强制转型代码，因此对于运行期Java来说ArrayList<int>与ArrayList<String>是同一个类。Java中泛型实现方式称为**类型擦除**。



以上代码类型擦除后相当于：



擦除法所谓的擦除，仅仅是对方法的Code属性中的字节码进行擦除，实际上元数据中还是保留了泛型信息，这也是我们能通过反射手段取得参数化类型的根本依据。

#### 9.3.2 自动装箱、拆箱与遍历循环

是Java里使用的最多的语法糖。

#### 9.3.3 条件编译

许多程序设计语言都提供了条件编译的途径，如C、C++中使用预处理器指示符#ifdef来完成条件编译，而在Java语言中并没有使用预处理器，因为Java编译器并非一个个地编译Java文件，而是将所有编译单元的语法树顶级节点输入到待处理列表后再进行编译，因此各个文件之间能够互相提供符号信息。

使用条件为常量的if语句能达到条件编译。编译器会把分支中不成立的代码块消除掉，这一工作在解除语法糖阶段完成。

## 第十章 运行期优化

### 10.1 概述

**JIT编译器**，Just In Time Compiler即时编译器，在运行时将热点代码Hot Spot code编译成与本地平台相关的机器码，并进行各种层次的优化。

#### 10.1.1 解释器与编译器

大部分主流的虚拟机采用解释器与编译器共存的架构。

解释器可以使程序迅速启动和执行，省去编译的时间，立即执行。当程序运行环境中内存限制较大时可以使用解释执行节约内存。

编译器能获取更高的执行效率。解释器也能帮助编译器根据概率选择一些大多数时候都能提升运行速度的优化手段。

-Xint Interpreted Mode 解释模式

-Xcomp Compiled Mode 编译模式

编译器分为Client Compiler和Server Compiler

Client Compiler是一个简单快速的三段式编译器，主要关注点在于局部性的优化，放弃了许多耗时较长的全局优化手段；

Server Compiler是专门面向服务器的典型应用并未服务器端的性能配置特别调整过的编译器，也是一个充分优化过的高级编译器，速度缓慢，但是可以减少本地代码执行时间。

经典代码优化手段：

无用代码消除 Dead Code Elimination

循环展开 Loop Unrolling

循环表达式外提 Loop Expression Hoisting

消除公共子表达式Common Subexpression Elimination

常量传播 Constant Propagation

基本块重排序 Basic Block Reordering

范围检查消除 Range Check Elimination

空值检查消除 Null Check Elimination

守护内联 Guarded Inlining

分支频率预测 Branch Frequency Prediction

#### 10.1.2 编译对象与触发条件

热点代码主要有两类：

1. 被多次调用的方法
2. 被多次执行的循环体

编译器会以整个方法（不是单独的循环体）作为编译对象，因为编译发生在方法执行过程之中，因此被称为栈上替换OnStackReplacement，简称OSR编译。

如何判断多次？

1. 基于采样的热点探测 Sample Based Hot Spot Detection

周期性检查各个线程的栈顶，如果发现某个方法经常出现在栈顶，就是热点方法。

简单、高效，容易获取方法调用关系，但是不精确、

1. 基于计数器的热点探测 Counter Based Hot Spot Detection

为每个方法建立计数器，统计方法的执行次数。

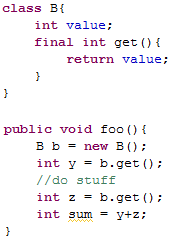
实现起来麻烦一些，但是结果精确、严谨。

当超过一定的时间限度，方法的调用次数仍然不足以让它提交给即时编译器编译，那这个方法的调用计数器就会较少一般，这个过程称为方法调用计数器热度的衰减（Counter Decay），而这段时间称为半衰周期（Counter Half Life Time）

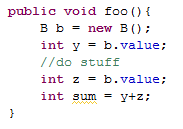
### 10.2 编译优化技术

#### 10.2.1 概览

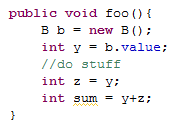
Java程序员有一个共识，以编译方式执行本地代码比解释方式更快，不仅因为虚拟机解释执行字节码时额外消耗时间的原因外，还有一个原因是虚拟机设计团队几乎把对代码的所有优化措施都集中在了即时编译器中。因此一般来说，即时编译器产生的本地代码会比Javac长生的字节码更加优秀。



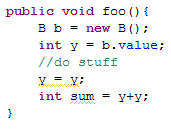
方法内联 Method Inlining



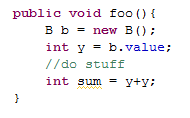
冗余访问消除 Redundant Loads Elimination



复写传播 Copy Propagation



无用代码消除 Dead Code Elimination



#### 10.2.2 公共子表达式消除：语言无关的经典优化技术

如果一个表达式E已经计算过了，并且从先前的计算到现在E中所有变量的值都没有发生变化，那么E的这次出现就称为了公共子表达式。可以直接使用前面计算过的表达式结果代替E就行了。

#### 10.2.3 数组边界检查消除：语言相关的经典优化技术

Java语言中访问数组元素系统将会自动进行上下界的范围检查，对于拥有大量数组访问的程序代码，这无疑也是一种性能负担。故在编译器就提前进行检查。

#### 10.2.4 方法内联：最重要的优化技术之一

除了消除方法调用的成本之外，它更重要的意义是为其他优化手段建立良好的基础。

如果是非虚方法，那么可以直接进行内联。如果是虚方法，则会向CHA**类型继承关系分析**（Class Hierarchy Analysis）查询在当前程序下是否有多个目标版本可供选择，如果查询结果只有一个版本，则进行内联，如果程序在后续执行过程中，虚拟机如果加载了导致继承关系发生变化的新类，就需要抛弃已经编译的代码，退回到解释状态执行，或者重新进行编译。称为**激进优化**。

**内联缓存Inline Cache**，在未发生方法调用前，内联缓存状态为空，当第一次调用发生后，缓存记录下方法接收者的版本信息，并且每次进行方法调用时都直接比较接收者版本，如果每次调用的方法接收者版本都是一样的，那这个内联还可以一直用下去。如果发生了方法接收者不一致的情况，就说明程序真正使用了虚函数的多态特性，这时才会取消内联，查找虚方法表进行方法分配。

#### 10.2.5 逃逸分析：最前沿的优化技术之一 Escape Analysis

并不是直接优化代码的手段，而是为其他优化手段提供依据的分析技术。

逃逸分析的基本行为就是分析对象动态作用域：当一个对象在方法中被定义后，它可能被外部方法所引用，例如作为调用参数传递到其他方法中，称为**方法逃逸**。甚至可能被外部线程访问到，譬如赋值给类变量或可以在其他线程中访问的实例变量，称为**线程逃逸**。

如果能证明一个对象不会逃逸到方法或线程之外，则可能为这个变量进行一些高效的优化：

1. **栈上分配 Stack Allocation**

在虚拟机中，一般在Java堆上分配创建对象，垃圾收集系统可以回收堆中不再使用的对象。如果确定一个对象不会逃逸出方法之外，那让这个对象在栈上分配内存将会是一个很不错的主意，对象所占用的内存空间就可以随栈帧出栈而销毁。在一般应用中，不会逃逸的局部对象所占的比例很大，如果能使用栈上分配，那大量的对象就会随着方法的结束而自动销毁了，垃圾收集系统的压力将会小很多。

1. **同步消除 Synchronization Elimination**

对非逃逸变量可以消除同步措施

1. **标量替换 Scalar Replacement**

标量指一个数据已经无法再分解成更小的数据来表示了（如int、long等），相对的，如果一个数据可以继续分解，那么他就称作**聚合量**（Aggregate）

如果逃逸分析证明一个对象不会被外部访问，并且这个对象可以被拆散的话，那程序真正执行的时候将可能不创建这个对象，而改为直接创建它的若干个被这个方法使用到的成员变量来代替。将对象拆分后，除了可以让对象的成员变量在栈上分配和读写外，还可以为后续进一步的优化手段创建条件。

技术尚未成熟，主要是不能保证逃逸分析的性能受益必定高于它的消耗。

### 10.3 Java与C/C++编译器的对比

Java与C/C++的编译器对比实际上代表了最经典的即时编译器与静态编译器的对比。

1. 即时编译器运行占用的是用户的运行时间，具有很大的**时间压力**，能提供的优化手段也严重受制于编译成本。所以即时编译器不敢随便引入大规模的优化技术，而编译的时间成本在静态优化编译器中并不是主要的关注点；
2. Java是**动态的类型安全语言**，意味着需要虚拟机动态检查，尽管编译器会努力进行优化，但是总体上仍然要消耗不少的运行时间；
3. Java中使用**虚函数**的频率远远大于C/C++语言，运行时对方法接收者进行多台选择的频率要远远大于C/C++语言，也意味着即时编译器在进行一些优化（方法内联）时的难度要远大于C/C++的静态语言优化器；
4. Java语言是**动态扩展**的语言，运行时加载新的类可能改变程序类型的继承关系，使得很多全局的优化都难以进行（只能以激进优化的方式完成），编译器不得不时刻注意并随着类型的变化而在运行时撤销或重新进行一些优化；
5. Java语言中对象的内存分配都是在堆上进行的，只有方法中的局部变量才能在栈上分配。而C/C++的对象则有多重内存分配方式，可以在栈上分配线程私有的对象，减轻内存回收的压力。另外C/C++中主要由用户程序代码来回收分配的内存，这就不存在无用对象筛选的过程，运行效率上也比垃圾收集机制高。

总而言之，Java语言性能上的劣势都是为了换取开发效率上的优势，动态安全、动态扩展、垃圾回收这些特性都会Java语言的开发效率做出了巨大的贡献。

另外由于C/C++编译器所有优化都在编译期完成，以运行期性能监控为基础的优化措施它都无法进行，如调用频率检测（Call Frequency Prediction）、分支频率预测（Branch Frequency Prediction）、裁剪未被选择的分支（Untaken Branch Pruning）等。

## 第十一章 Java内存模型与线程

### 11.1 缓存与乱序执行优化

由于计算机的存储设备与处理器的运算速度有几个数量级的差距，所以现代计算机系统都不得不加入一层读写速度尽可能接近处理器运算速度的高速缓存Cache来作为内存与处理器之间的缓冲。

在多处理器系统中，每个处理器都有自己的Cache，而它们共享同一主内存Main Memory，将可能导致各自的缓存数据不一致。

Out-Of-Order Execution,处理器还会对输入代码进行乱序执行优化，对代码的先后顺序进行调整。

### 11.2 JMM Java内存模型

Java Memory Model的目标是定义程序中各个变量的访问规则，这里的变量不包括线程私有的局部变量、方法参数。

内存间交互操作主要有：

1. Lock
2. Unlock
3. Read

主内存→工作内存

1. Load

Read的apply

1. Use

工作内存→执行引擎

每当虚拟机遇到一个需要使用到变量的值的字节码指令时执行这个操作

1. Assign

执行引擎→工作内存

每当虚拟机遇到一个给变量赋值的字节码指令时执行这个操作

1. Store

工作内存→主内存

1. Write

Store操作的apply

### 11.3 volatile关键字

确保本条指令不会因编译器的优化而省略，且要求使用时每次都重新读取、写入这个变量的值而不是使用保存在寄存器里的备份。

（为了提高存取速度，编译器优化有时会先把变量读取到一个寄存器中，以后再取变量值时，就直接从寄存器中取值。而当变量在别的线程改变了值，该寄存器的值不会相应的改变）

Java语言规范中说：为了获得最佳速度，允许线程保存共享成员变量的私有拷贝，而且只当线程进入或离开同步代码块时才与共享成员变量的原始值对比。所以建议在两个或更多的线程访问的成员变量上使用volatile，当要访问的变量已在synchronized代码块中时或为常量时，不必使用。

常用在

1. 并行设备的硬件寄存器，如：状态寄存器
2. 一个中断服务子程序中会访问到的非自动变量
3. 多线程应用中被几个任务共享的变量

volatile具有synchronized的可见性特征，但是不具备原子特性。在某些情况下，如果读操作远远大于写操作，volatile变量还可以提供优于锁的性能优势。

注意使用条件：

1. 对变量的写操作不依赖于当前值，如x++
2. 该变量没有包含在具有其他变量的不变式中

long，double虽然64位，导致load read和store write不具有原子性，但是大多数商用虚拟机几乎都把64位数据的读写操作作为院子操作来对待。

### 11.4 JMM三大特性

#### 11.4.1 Atomicity 原子性

通过read、load、assign、use、store、write保证原子性

#### 11.4.2 Visibility 可见性

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。

Volatile的特殊规则保证了新值能立即同步到主内存，以及每次使用前立即从主内存中刷新。

Synchronized块，对一个变量执行unlock操作之前，必须先把此变量同步回主内存中；

Final，被final修饰的字段在构造器中一旦初始化完成，并且构造器没有把this引用传递出去，那在其他线程中就可以看见final字段的值。

#### 11.4.3 Ordering 有序性

本线程内观察，所有的操作都是有序的。

如果在一个线程中观察另一个线程，所有的操作都是无序的。

### 11.5 happens-before原则

先行发生是Java内存模型中定义的两项操作之间的偏序关系。

Java中无序任何同步手段保障就能成立的先行发生规则：

1. Program Order Rule 程序次序规则

在一个线程内，按照程序代码顺序，书写在前面的操作先行发生于书写在后面的操作

1. Monitor Lock Rule 管程锁定规则

一个unlock操作先行发生于后面对同一个锁的lock操作

1. Volatile Variable Rule volatile变量规则

对一个volatile变量的写操作先行发生于后面对这个变量的读操作

1. Thread Start Rule 线程启动规则

Thread对象的start()方法先行发生于此线程的每一个动作

1. Thread Termination Rule 线程终止规则

线程中的所有操作都先行发生于对此线程的终止检测

1. Thread Interruption Rule 线程中断规则

对线程interrupt()方法的调用先行发生于被终端线程的代码检测到中断事件的发生

1. Finalizer Rule 对象终结规则

一个对象的初始化完成先行发生于它的finalize()方法的开始

1. Transitivity 传递性

A先于B，B先于C，则A先于C

### 11.6 线程

并发不一定要依赖多线程（如PHP中很常见的多进程并发）

线程是比进程更轻量级的调度执行单位，线程的引入，可以把一个进程的资源分配和执行调度分开，各个线程既可以共享进程资源（内存地址、文件IO等），又可以独立调度（线程是CPU调度的最小单位）。

实现线程的三种方式：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。

#### 11.6.1 使用内核线程实现

Kernel-Level Thread，KLT 内核线程，就是直接由操作系统内核支持的线程，这种线程由内核来完成线程切换，内核通过Scheduler操纵调度器对线程进行调度，并负责将线程的任务映射到各个处理器上。每个内核线程可以视为内核的一个分身。

Light Weight Process LWP 轻量级进程

程序一般不会直接去使用内核线程，而是去使用轻量级进程，这是内核线程的一种高级接口。每个轻量级线程都由一个内核线程支持，因此只有先支持内核线程，才能有轻量级进程。

缺点：

1. 基于内核线程实现，各种线程操作（创建、西沟、同步）都需要进行系统调用，而系统调用代价较高，需要在用户态和内核态中来回切换
2. 每个轻量级线程需要有一个内核线程的支持，需要消耗一定的内核资源（如内核线程的栈空间）

#### 11.6.2 使用用户线程实现

User Thread

广义上说，一个线程只要不是内核线程，就可以认为是用户进程。

狭义上的用户线程是完全建立在用户空间的线程库上，系统内核不能感知线程存在的实现。用户线程的建立、同步、销毁和调度完全在用户态中完成。操作非常快速和低消耗，也支持规模更大的线程数量。但是线程的创建、切换和调度都是需要考虑的问题，而且由于操作系统只把处理器资源分配到进程，那诸如“阻塞如何处理”、“多处理器系统中如何将线程映射到其他处理器上”这类问题解决起来将非常困难。

#### 11.6.3 使用用户线程加轻量级进程混合实现

## 第十二章Java线程调度

### 12.1 Cooperative Threads-Scheduling 协同式进程调度

线程的执行时间由线程本身来控制。

### 12.2 Preemptive Threads-Scheduling 抢占式进程调度

线程的执行时间由操作系统来进行分配

### 12.3 状态转换

#### 12.3.1 New 新建

创建后尚未启动的线程

#### 12.3.2 Runable 运行

包括操作系统中的Running和Ready

#### 12.3.3 Waiting 无限期等待

等待被其他线程显示的唤醒，可以使用以下方法：

1. Object.wait()
2. Thread.join()
3. LockSupport.park()

#### 12.3.4 Timd Waiting 限期等待

一定时间后会由系统自动唤醒

1. Thread.sleep()
2. Object.wait( time)
3. Thread.join(time)
4. LockSupport.parkNanos()
5. LockSupport.parkUntil()

#### 12.3.5 Blocked 阻塞

等待着获取到一个拍他所

#### 12.3.6 Terminated 终止

## 第十三章 线程安全与锁优化

当多个线程访问一个对象时，如果不用考虑这些线程在运行时环境下的调度和交替执行，也不需要进行额外的同步，或者在调用方进行任何其他的协调操作，调用这个对象的行为都可以获得正确的结果，那这个对象是线程安全的。

这要求线程安全的代码本身封装所有必要的正确性保障手段（如互斥同步）

将Java语言中的各种操作共享数据分为5类：

1. 不可变
2. 绝对线程安全

不管运行环境如何，都不需要任何额外的同步措施。

这是非常严格的条件！！！！

Vector都不会绝对线程安全的。

1. 相对线程安全

Vector、HashTable、Collections

1. 线程兼容

本身不是线程安全的，但是可以通过在调用端正确的使用同步手段来保证对象在并发环境中可以被安全的使用

ArrayList HashMap

1. 线程对立

无论如何都无法在多线程环境中并发使用

Thread的suspend() resume()

### 13.1 线程安全的实现方法

#### 13.1.1 互斥同步 Mutual Exclusion & Synchronization

##### 13.1.1.1 Synchronized

最基本的手段，经过编译后会在同步块的前后形成monitorenter和monitorexit两个字节码指令。

是Java语言中一个Heavyweight重量级操作，在确实必要的情况下才使用这种操作。因为Java的线程是映射到操作系统的原生线程上的，如果要阻塞或者唤醒一个线程，都需要切换到和心态。

##### 13.1.1.2 重入锁 ReentrantLock

是API层面的互斥锁，synchronized是原生语法层面的互斥锁。

增加了高级功能：等待可中断、可实现公平锁、锁可以绑定。

1. 等待可中断

等待线程可以放弃等待，处理其他事情

1. 可实现公平锁

必须按照申请锁的事件顺序来获得锁，默认非公平，需要通过带布尔值的构造函数

1. 锁可以绑定

一个ReentrantLock对象可以同时绑定多个Condition对象，而在synchronized中，锁对象的wait()，norify(),norifyAll()方法可以实现一个隐含的条件，如果要和多于一个的条件关联的时候，就不得不额外的添加一个锁，而ReentrantLock只需要多次调用newCondition即可

#### 13.1.2 非阻塞同步 Non-Blocking Synchronization

阻塞同步属于悲观的并发策略，总是认为只要不去做正确的同步措施，就一定会发生问题，所以无论共享数据是否真的会出现竞争，它都要进行加锁。

乐观并发策略：先进行操作，如果没有其他线程争用共享数据，那操作就成功了。如果共享数据有争用，产生了冲突，那就再采取其他的补偿措施（最常见的是不断重试直到成功）。这种乐观并发策略的很多实现都不需要把线程挂起，因此被称为非阻塞同步。

需要硬件，因为操作和冲突检测这两个步骤需要具备原子性。

#### 13.1.3 无同步方案

##### 13.1.3.1 Reentrant Code 可重入代码

也被称为PureCode，可以在代码执行的任何时刻中断它，转而去执行另外一段代码（包括递归调用它本身），而在控制权返回后，原来的程序不会发生任何错误。

共同特征：

1. 不依赖存储在堆上的数据和公用的系统资源
2. 用到的状态量都由参数传入
3. 不调用非可重入的方法

如果一个方法它的返回结果是可以预测的，只要输入了相同的数据就都能返回相同的结果，它就满足可重入性的要求。

##### 13.1.3.2 Thread Local Storage 线程本地存储

一段代码中所需要的数据必须与其他代码共享，如果能保证这些共享数据的代码在同一个线程中执行，就把共享数据的可见范围限制在同一个线程之内，这样，无需 同步也能保证线程之间不出现数据争用的问题。

### 13.2 锁优化

#### 13.2.1 适应性自旋 Adaptive Spinning

互斥同步对性能最大的影响是阻塞的实现，挂起线程和恢复线程需要转入内核态中实现，给系统的并发性能带来了巨大的压力。

自旋锁是让锁的线程执行一个忙循环，但不放弃处理器的执行时间。

如果锁被占用的时间很短，自旋等待的效果就非常好。自适应意味锁的时间不是固定的 了，而是由前一次在同一个锁上的自旋时间及锁的拥有者的状态来决定。

#### 13.2.2 锁消除 Lock Elimination

指虚拟机即时编译器在运行时，对一些代码上要求同步，但是检测到不可能存在共享数据竞争的锁进行消除。判断方法是：逃逸分析技术。

#### 13.2.3 锁粗化 Lock Coarsening

如果虚拟机探测到有这样一串零碎的操作都对同一个对象加锁，将会把加锁同步的范围扩展到整个操作序列的外部。防止频繁的进行互斥同步操作导致的不必要的性能消耗。

#### 13.2.4 轻量级锁 Lightweight Locking

通过对象头中最后两位标志锁。若有两条以上的线程征用同一个锁，那轻量级锁就不再有效，要膨胀为重量级锁。核心是CAS操作将对象头与线程的栈同步。

依据是“对于绝大部分的锁，在整个同步周期内都是不存在竞争的”。如果没有竞争，轻量级锁使用CAS操作避免了使用互斥量的开销。

#### 13.2.5 偏向锁 Biased Locking

消除数据在无竞争情况下的同步原语。意思是锁会偏向于第一个获得它的线程，如果在接下来的执行过程中，该锁没有被其他的线程获取，则持有偏向锁的线程将永远不需要再进行同步。

原理类似轻量级锁，设置标志位并通过CAS操作将线程的ID记录在对象的Mark Word中，持有偏向锁的线程每次进入这个锁相关的同步块，都不进行任何同步操作（update，lock，unlock）等。但是，当有另一个线程去尝试获取这个锁时，偏向模式宣告结束。

偏向锁可以提高带有同步但无竞争的程序性能。它同样是一个TradeOff带有效益均衡性质的优化，并不一定总是对程序的运行有利。