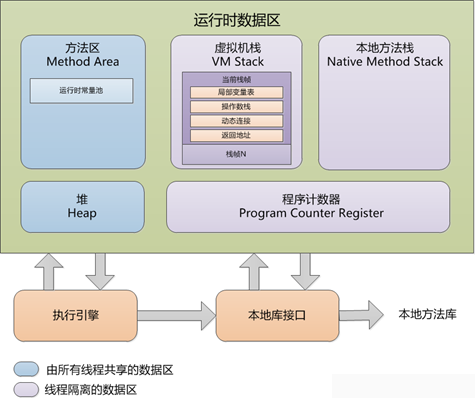
## http://www.cnblogs.com/prayers/p/5515245.html

## 第一章 Java内存区域与内存溢出异常

### ※1.1、运行是数据区域

JAVA虚拟机在执行Java程序的过程中会把它所管理的内存划分为若干个不同的数据区域。根据《Java虚拟机规范（Java SE 7 版）》的规定，Java虚拟机所管理的内存将会包括以下几个运行是数据区域：



#### 1.1.1.程序计数器

程序计数器（Program Counter Register）是一块较小的内存空间，字节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。虚拟机规范中指出，每一条线程都有一个独立的程序计数器，即为线程私有的。注意，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。

#### 1.1.2. Java虚拟机栈

Java虚拟机栈（VM Stack）也是线程私有的，它与Java线程同一时间创建，虚拟机栈用于存放局部变量表，操作数栈，动态链接，方法出口等信息。

局部变量表存放了编译期可知的**基本数据类型**(boolean、byte、char、short、int、float、long、double)，**对象引用**(可能是一个指向对象起始地址的引用指针，也可能指向一个代表对象的句柄或者其他与此对象相关的位置)**和returnAddress类型**(指向了一条字节码指令的地址)。局部变量表所需的内存空间在编译期间完成分配，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局部变量表的大小。

在Java虚拟机规范中，对这个区域规定了两种异常状况：①如果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；②如果虚拟机栈在动态扩展时无法申请到足够的内存，将抛出OutOfMemoryError异常。

#### 1.1.3. 本地方法栈

　　本地方法栈（Native Method Stack）和虚拟机栈作用相似，在HotSpot虚拟机将本地方法栈和虚拟机栈合二为一，它们的区别在于，虚拟机栈为执行Java方法服务，而本地方法栈则为虚拟机使用到的Native方法服务。本地方法栈也会抛出StackOverflowError异常和OutOfMemoryError异常

#### 1.1.4. Java堆

　　Java 堆（Java Heap）是被所有线程共享的内存区域，在虚拟机启动时创建。这个区域是用来存放对象实例的，几乎所有对象实例都会在这里分配内存。堆是Java垃圾收集器管理的主要区域（GC堆）。Java堆可以细分为：新生代和老年代；再细致一点的有Eden空间，From Survivor空间，To Survivor空间等。Java堆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，就像我们的磁盘空间一样。可以通过-Xmx和-Xms控制。

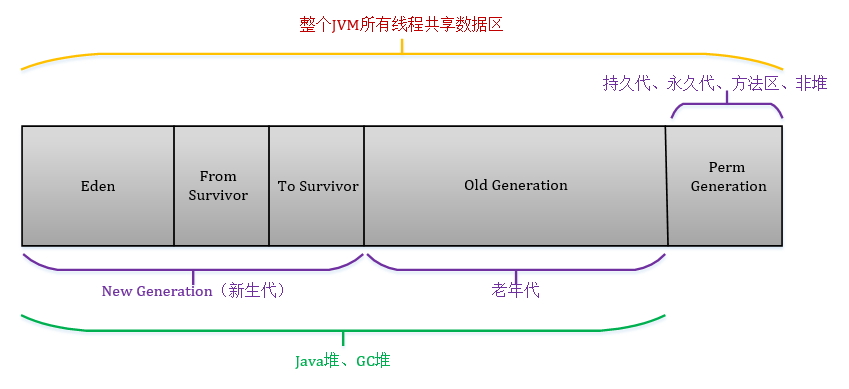
#### 1.1.5. 方法区

　　方法区（Method Area）也叫永久代，本质上并不等价，仅仅是因为HotSpot 虚拟机的设计团队选择把GC 分代收集扩展至方法区，或者说使用永久代来实现方法区而已。对于其他虚拟机（如BEA JRockit、IBM J9 等）来说是不存在永久代的概念的。

永久代也是被所有线程共享的内存区域，它用于存储已被虚拟机加载的**类信息、常量、静态变量**（JDK7中被移到Java堆）、**即使编译器编译后的代码**等数据。

**运行时常量池**，它是方法区的一部分，用于存放编译期生成的各种字面量和符号引用（其实就是八大基本类型的包装类型和String类型数据（JDK7中被移到Java堆））。运行期间也可能将新的常量放入池中，例如String.intern()方法。（官方文档说明： In JDK 7, interned strings are no longer allocated in the permanent generation of the Java heap, but are instead allocated in the main part of the Java heap (known as the young and old generations), along with the other objects created by the application）。从JDK7开始永久代的移除工作，贮存在永久代的一部分数据已经转移到了Java Heap或者是Native Heap。但**永久代仍然存在于JDK7，并没有完全的移除**：符号引用(Symbols)转移到了native heap;字面量(interned strings)转移到了java heap;类的静态变量(class statics)转移到了java heap。随着**JDK8的到来，JVM不再有PermGen**。但类的元数据信息（metadata）还在，只不过不再是存储在连续的堆空间上，而是移动到叫做“Metaspace”的本地内存（Native memory）中。

在JVM中共享数据空间划分如下图所示



上图中，刻画了Java程序运行时的堆空间,可以简述成如下2条

1.JVM中共享数据空间可以分成三个大区，新生代（Young Generation）、老年代（Old Generation）、永久代（Permanent Generation），其中JVM堆分为新生代和老年代

2.新生代可以划分为三个区，Eden区（存放新生对象），两个幸存区（From Survivor和To Survivor）（存放每次垃圾回收后存活的对象）。

### 1.2 HotPot虚拟机对象

#### 1.2.1对象的创建

虚拟机遇到一条new指令时，①首先将去检查这个指令的参数是否能再常量池中定位到一个类的符号引用，②而且检查这个符号引用代表的类是否已被加载、解析和初始化。③如果没有，那必须先执行相应的类加载过程。

在类加载检查通过之后，接下来虚拟机将为新生对象分配内存。两种分配方法：

①指针碰撞：假设Java堆中内存时绝对规整的，所有用过的内存都放在一边，空闲的内存放在另一边，中间放着一个指针作为分解点，这种分配方式就仅仅是把那个指针向空闲空间那边挪动一段与对象大小相等的距离。

②空闲列表：Java堆中的内存不是规整的，虚拟机需要维护一个列表，记录上哪块内存块是可用的，在分配的时候从列表中找到一块足够大的空间分给对象实例，并更新列表上的记录。

因此，使用Serial、ParNew等待compact过程的收集器时，系统采用的分配算法是指针碰撞；而使用CMS这种基于mark-sweep算法的收集器时，通常采用空闲列表。

#### 1.2.2 对象的内存布局

对象在内存中存储的布局可以分为3块区域：对象头、实例数据和对齐填充。

对象头包括两部分信息：第一部分用于存储对象自身的运行时数据，第二部分是类型指针。如果对象是一个Java数组，那在对象头中还必须有一块英语记录数组长度的数据。

#### 1.2.3 对象的访问定位

1、使用句柄访问：从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。最大的好处就是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。

2、使用直接指针访问：速度快，hotSpot使用的是直接指针进行对象访问。

### 1.3 实战OutOfMemoryError异常

#### 1.3.1 Java堆溢出

**-XX:+HeapDumpOnOutOfMemoryError** 在内存溢出是dump出当前内存堆转储快照。

-Xms 最小堆大小

-Xmx 最大堆大小

#### 1.3.2 栈和本地方法栈溢出

-Xss 栈内存容量

#### 1.3.3 方法区和运行时常量池溢出

-XX:PermSize

-XX:MaxPermSize

String.intern()是一个Native方法，它的作用是：如果字符串常量池中已经包含一个等于此String对象的字符串，则返回代表池中这个字符串的String对象，否则，将此String对象包含的字符串添加到常量池中，并且返回此String对象的引用。

## 第二章、垃圾收集器与内存分配策略

Java内存运行时区域中**程序计数器、虚拟机栈、本地方法栈**是随着线程而生，随线程而灭，这几个区域的内存分匹配和回收都具备确定性，在方法结束或者线程结束时，内存自然就随着回收了。 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，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是Java堆和方法区这部分内存。

### 2.1.对象已死？

#### 2.1.1 引用计数算法

**思想：**给对象添加一个引用计数器，每当有一个地方引用它时，计数器值就加1；当引用失效时，计数器值就减1；任何时刻计数器为0的对象就是不可能再被使用的。

**缺点：**很难解决对象之间相互循环引用的问题。

虚拟机并不是通过引用计数算法来判断对象是否存活的。

#### 2.1.2 可达性分析算法

**思想：**通过一系列的成为“GC Roots”的对象作为起始点，从这些节点开始向下搜索，搜索所有走过的路径成为“引用链”，当一个对象到GC Roots没有任何引用链相连时，则证明此对象是不可用的。

**在Java中，可作为GC Roots的对象包括：**

1. 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引用的对象。

#### ※2.1.3 再谈引用

在JDK1.2之后，Java对引用的概念进行了扩充，将引用分为：①强引用；②软引用；③弱引用，④虚引用。这4中引用强度依次逐渐减弱。

①强引用：是指创建一个对象并把这个对象赋给一个引用变量。类似“Object obj = new Object()”这类的应用，强引用有引用变量指向时永远不会被垃圾回收。

1. **public void fun1() {**
2. Object object = **new Object();**
3. Object[] objArr = **new Object[1000];**
4. }

当fun1运行完之后，object和objArr都已经不存在了，所以它们指向的对象都会被JVM回收。如果想中断强引用和某个对象之间的关联，可以显示地将引用赋值为null。

②软引用：用来描述一些还有用但并非必需的对象。如果一个对象具有软引用，内存空间足够，垃圾回收器就不会回收它；如果内存空间不足了，就会回收这些对象的内存。只要垃圾回收器没有回收它，该对象就可以被程序使用。在JDK1.2之后，提供了SoftReference类来实现软应用。

③弱引用：弱引用也是用来描述非必需对象的，当JVM进行垃圾回收时，无论内存是否充足，都会回收被弱引用关联的对象，被弱引用关联的对象只能生存到下一次垃圾收集发生之前。在java中，用java.lang.ref.WeakReference类来表示。

④虚引用：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。提供了PhantomReference类来实现弱应用。

#### 2.1.4 生存还是死亡

即使在可达性分析算法中不可达的对象，也并非是“非死不可”的，这时候它们暂时处于“缓刑”阶段，要真正宣告一个对象死亡，至少要经历两次标记过程：如果对象在进行可达性分析后发现没有与GC Roots相连接的引用链，那它将会被第一次标记并且进行一次筛选，筛选的条件是此对象是否有必要执行finalize方法。Finalize方法是对象逃脱死亡命运的最后一次机会，如果对象要在finalize中成功拯救自己，只需要重新与引用链上的任何一个对象建立关联即可，譬如把自己（this关键字）复制给某个类变量或者对象的成员变量。

任何一个对象的finalize方法都只会被系统自动调用一次，如果对象面临下一次回收，它的finalize方法不会被再次执行。

#### 2.1.5 回收方法区

永久代的垃圾收集主要回收两部分内容：废弃常量和无用的类。无用的类“可以”被回收，并非必然回收。无用的类需要同时满足3个条件：

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。

### 2.2 垃圾收集算法

#### 2.2.1 标记-清除算法

算法分为“标记”和“清除”两个阶段：首先标记处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。

缺点：①效率问题：标记和清除两个过程的效率都不高；②空间问题：标记清除之后会产生大量不连续的内存碎片，空间碎片太多可能会导致需要分配较大对象时，无法找到足够的连续内存而不得不提前触发另一次垃圾收集动作。

#### 2.2.2 复制算法

它将可用内存按容量划分为大小相等的两块，每次只是用其中一块。当这块用完了就复制到另一块中。然后把使用过的内存空间一次清理掉。这样使得每次都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。只要移动退订指针，简单高效。

缺点：将内存缩小为原来的一半。

现在的商业虚拟机都采用这种收集算法来回收新生代。将内存分为块较大的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中还存活这的对象一次性地复制到另外一块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。也就是每次新生代中可用内存空间为整个新生代容量的90%，只有10%的内存会被“浪费”。我们没办法保证每次回收都只有不多于10%的对象存活，当survivor空间不够用时，需要依赖其他内存（这里指老年代）进行分配担保。

#### 2.2.3 标记-整理算法

复制算法在对象存活率较高时需要较多的复制操作，效率会边地，如果不想浪费50%的空间，就需要额外的空间进行分配担保。根据老年代的特点，提出了标记-整理算法。

标记-整理算法的标记过程和“标记-清除”一样，但后续步骤不是对可回收对象进行清理，而是**让所有存活的对象都向一端移动**，然后直接清理掉端边界以外的内存。

#### 2.2.4 分代收集算法

当前商业虚拟机的垃圾收集都采用“分代收集”算法。只是根据对象存活周期的不同将内存划分为几块（新生代和老年代，**当新创建对象时一般在新生代中分配内存空间**，当新生代垃圾收集器回收几次之后仍然存活的对象会被移动到年老代内存中，**当大对象在新生代中无法找到足够的连续内存时也直接在年老代中创建**），根据各个年代的特点采用最适当的手机算法。

新生代中，没有垃圾收集时只有少量对象存活，所以选择复制算法。Java虚拟机对新生代的垃圾回收称为Minor GC，次数比较频繁，每次回收时间也比较短。使用java虚拟机-Xmn参数可以指定新生代内存大小。

老年代中因为对象存活率高，使用“标记-清理”或者“标记-整理”算法。Java虚拟机对年老代的垃圾回收称为MajorGC/Full GC。

永久代也使用**标记-整理算法**进行垃圾回收，java虚拟机参数-XX:PermSize和-XX:MaxPermSize可以设置永久代的初始大小和最大容量。

### 2.3 HotSpot的算法实现

#### 2.3.1 枚举根节点

为了满足分析结果准确性，即在整个分析期间整个执行系统看起来就像被冻结在某个时间点上，所以GC进行时必须停顿所有Java执行线程。当执行系统停顿下来后，并不需要一个不漏地检查完所有执行上下文和全局的引用位置，虚拟机应该有办法直接得知哪些地方存放着对象引用。HotSpot使用OopMap的数据结构来实现，在类加载完成的时候，HotSpot就把对象内什么偏移量上是什么类型的数据计算出来，也会在特定的位置记录下栈和寄存器中哪些位置是引用。

#### 2.3.2 安全点

在OopMap的协助下，HotSpot可以快速并准确地完成GC Roots枚举。但是并不会为每条指令都生成OopMap，只是在“特定的位置”记录下这些信息，这些位置成为安全点，即程序执行时并非在所有地方都能停顿下来开始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。“长时间执行”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，所以具有这些功能的指令才会产生“安全点”。

在发生GC时，让所有线程都在安全点上停顿下来的方案有：抢先式中断和主动式中断。

抢先式中断：在GC发生时，首先把所有线程全部中断，如果发现有线程中断的地方不在安全点上，就回复线程，让他“跑”到安全点上。**现在几乎没有虚拟机采用这种方式。**

主动式中断：在GC发生时，不直接对线程操作，仅仅简单地设置一个标志，各个线程执行时主动去轮询这个标志，发现中断标志为真时就自己中断挂起。

#### 2.3.3 安全区域

程序“不执行”的时候（即线程处于sleep状态或者blocked状态），这时候线程无法相应JVM的中断请求，就需要安全区域来解决。

在线程执行到安全区域中的代码时，首先标识自己已经进入了安全区域，这样，当JVM要发起GC时，就不用管标识自己为安全区域状态的线程。在线程要离开安全区域时，他要检查系统是否已经完成了根节点枚举。

### 2.4 垃圾收集器

#### 2.4.1 Serial收集器

Serial收集器曾经是虚拟机新生代收集的唯一选择，是一个单线程的收集器，在进行收集垃圾时，必须暂停其他所有的工作线程，它是虚拟机运行在Client模式下的默认新生代收集器。

#### 2.4.2 ParNew收集器

ParNew收集器是Serial收集器的多线程版本，许多运行在Server模式下的虚拟机中首选的新生代收集器，除Serial外，只有它能与CMS收集器配合工作。

ParNew在单CPU的环境中绝对不会比Serial收集器更好的效果，两个CPU也不能百分之百的保证可以超越Serial。

#### 2.4.3 Parallel Scavenge收集器

Parallel Scavenge收集器也是新生代收集器，使用复制算法又是并行的多线程收集器，它的目标是达到一个可控制的吞吐量（吞吐量=运行用户代码跟/（运行用户代码+垃圾收集时间））。

可以使用参数-XX:+UseAdaptiveSizePolicy，进行GC自适应的调节策略。

**自适应调节策略也是Parallel Scavenge与ParNew的重要区别。**

#### 2.4.4 Serial Old收集器

Serial Old是Serial收集器的老年代版本，同样是单线程收集器，使用“标记-整理算法”。

#### 2.4.5 Parallel Old收集器

Parallel Old 是Parallel Scavenge收集器的老年代版本，使用多线程和标记整理算法。

#### 2.4.6 CMS收集器

Concurrent Mark Sweep 收集器是一种以**获得最短回收停顿时间为目标**的收集器，基于标记清除算法。

过程如下：初始标记，并发标记，重新标记，并发清除。

**初始标记、重新标记**这两个步骤**仍然需要“** **stop the world”**。初始标记仅仅只是标记一下 GC Roots能直接关联到的对象，速度很快。并发标记阶段就是进行 GC Roots Tracing的过程。而重新标记阶段则是为了修正并发标记期间因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录。整个过程中耗时最长的并发标记和并发清楚过程收集器线程都可以与用户线程一起工作。

优点： **并发收集、低停顿**；

缺点：① **CMS对CUP资源非常敏感** 。在并发阶段，占用部分线程而导致应用程序变慢， **总吞吐量降低**。 CMS默认启动的回收线程数为：（ CPU数量+3 ）/4.

②**CMS无法处理浮动垃圾**。**可能出现Concurrent Mode Failure** **失败而导致另一次** **fullGC的产生**。CMS不能像其他收集器那样等到老年代几乎完全被填满了再进行收集，需要预留一部分空间提供并发收集时的程序运作使用。在 JDK1.6中，启动阈值为92%。要是 CMS运行期间预留的内存无法满足程序需要就会出现 Concurrent Mode Failure失败，这是就会临时启用 Serial old收集器来重新进行老年代的垃圾收集，这样停顿时间就更长了。

③CMS是基于“标记-清除”算法实现的收集器，**在收集结束时会有大量空间碎片产生**。如果无法找到足够大的连续空间来分配当前对象，不得不提前出发一次 fullGc。CMS 默认开启了再 FullGC时开启内存碎片的合并整理过程，默认每次 FullGC都进行碎片整理。

#### 2.4.7 G1收集器

         JDK1.7 中使用G1收集器。 G1是一款面向服务端应用的垃圾收集器。

特点：①并行和并发；②分带收集；③空间整合： G1从整体来看是基于“标记 -整理”算法实现的收集器，从局部（两个 region之间）上看来是基于“复制”算法实现的。④可预测额停顿： G1除了追求低停顿外，还能建立可预测的停顿时间模型，能让使用者明确指定在一个长度为 M毫秒的时间片段内，消耗在垃圾收集上的时间不得超过 N毫秒。

在G1之前的其他收集器进行收集的范围都是整个新生代或者老年代。而 G1不在是这样。使用G1收集器时，它将整个 Java堆划分为多个大小相等的独立区域（ region），新生代和老年代不再是物理隔离的了，他们都是一部分 region的结合。

G1跟踪各个region 里面的垃圾堆积的价值大小（回收获得的空间大小以及回收所需时间的经验值），在后台维护一个优先列表，每次根据允许的收集时间，有限回收价值最大的 region。

为了可达性判定确定对象是否存活的时候保证准确性，使用 Remembered Set来避免全堆扫描。G1的每个 region都有一个与之对应的Remembered Set，在 GC根节点的枚举范围中加入 Remembered Set及可保证不对全堆扫描也不会有遗漏。

G1运作步骤：初始标记，并发标记，最终标记，筛选回收。

### 2.5 内存分配合回收策略

#### 2.5.1 对象优先在Eden 分配

大多数情况下，对象在新生代 Eden区中分配。当Eden区没有足够空间进行分配时，虚拟机将发起一次 Minor GC。新生代中Eden 区与一个 survivor区的空间比例为8:1。新生代可用空间为 Eden+一个survivor ；

新生代GC（Minor GC）；老年代GC（Full GC）

#### 2.5.2 大对象直接进入老年代

典型的大对象就是那种很长的字符串以及数组。

参数 -XX:PretenureSizeThreshold 直接让大对象在老年代分配。避免在Eden区以及两个Surivor区之间发生大量的内存复制。

#### 2.5.3 长期存活的对象将进入老年代

虚拟机给每个对象定义了一个对象年龄计数器：如果对象经过一次Minor GC，年龄就增加1岁，当年龄增加到一定程度MaxTenuringThreshold（默认15岁），就会被晋升到老年代。

#### 2.5.4 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总和大于空间的一般，年龄大于或等于该年龄的对象就直接进入老年代，不用等到MaxTenuringThreshold要求的年龄。

#### 2.5.5 空间分配担保

在发生Minor GC之前，虚拟机先检查**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**。

如果不成立，则查看HandlePromotionFailure设置值是否允许担保失败。

如果允许，检查老年代最大可用的连续空间是否大于历次晋升到老年代的平均大小

如果大于，尝试进行Minor GC

如果小于或者不允许冒险，进行Full GC。

大部分情况下，都允许冒险，避免Full GC 过于频繁。

JDK6 update24之后，只要老年代的连续空间大于新生代对象总大小或者历次晋升的平均大小就会进行Minor GC，否则进行Full GC。

## 第三章 虚拟机性能监控与故障处理工具

### 3.1 JDK命令行工具

|  |  |
| --- | --- |
| 名称 | 主要作用 |
| Jps | 显示系统内所有的HotSpot虚拟机进程 |
| jstat | 收集HotSpot虚拟机各方面的运行数据 |
| jinfo | 显示虚拟机配置信息 |
| Jmap | 生成虚拟机的内存转储快照 |
| jhat | 在浏览器查看分析内存转储快照 |
| jstack | 显示虚拟机的线程快照 |

#### 3.1.1 jps:虚拟机进程状况工具

jps –l 输出主类的全名，如果进程执行的是jar包，输出jar路径

jps –q 省略主类的名称

jsp –v 输出虚拟机进程启动时JVM参数

jsp –m 虚拟机启动时传递给主类main()函数的参数

#### 3.1.2 jstat:虚拟机统计信息监控工具

**jstat -gc pid [查询间隔 查询次数** ] 监视Java堆状态，包括Eden、survivor、老年代、永久代等的容量，已用空间、GC时间合计等信息。

Jstat –gccapacity 监视内容与-gc基本相同，但关注的是各个区域使用到的最大、最小空间

Jstat –gcutil 关注的是已使用空间占总空间的百分比

Jstat –gcnew 监视新生代GC状态

Jstat –gcold 监视老年代GC状态

#### 3.1.3 jinfo:java配置信息工具

Jinfo –v 虚拟机启动时显示指定的参数列表

Jinfo –flag 虚拟机未被显示指定的参数的系统默认值

Jinfo –flag name=value 修改虚拟机参数

#### 3.1.4 jmap:Java内存映像工具

Jmap –dump:format=b,file=eclipse.bin pid 生成一个正在运行的Eclipse的dump快照文件。

#### 3.1.5 jstack:java堆栈跟踪工具

Jstack –l pid 除堆栈外，显示关于锁的附加信息

### 3.2 JDK可视化工具

JConsole：Java监视与管理控制台

VisualVM：多合一故障处理工具

### 3.3 修改后的eclipse.ini配置

-Dosgi.requiredJavaVersion=1.6

-Duser.name="chengling"

-Xms1024m

-Xmx1024m

-Xmn256m

-XX:PermSize=192M

-XX:MaxPermSize=192M

-Dfile.encoding＝utf-8

## 第四章 类文件结构

### 4.1 Class类文件的结构

Class文件是一组以8位字节为基础单位的二进制流。各个数据项目严格按照顺序紧凑地排列在Class文件之中，中间**没有任何分隔符**。

Class文件格式采用一种类似于C语言结构体的微结构来存储数据，只有两种数据类型：无符号数和表。

#### 4.1.1 魔数与Class文件的版本

每个class文件的头4个字节成为魔数，它的唯一作用是确定这个文件是否是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?)。

紧跟魔数的4个字节存储的是Class文件的版本号：第5和第6是此版本号，第7和第8是主版本号。高版本可以向下兼容以前的版本，但是不能运行以后的版本。

#### 4.1.2 常量池

紧接着主次版本号之后的是常量池入口。

由于常量池中常量的数量是不固定的，所以在常量池的入口需要放置意向u2（2个字节）类型数据，代表常量池容量计数值。这个容量计数是从1而不是0开始的。常量池主要存放两大类常量：字面量（文本字符串、申明为final的常量值等）和符号引用（①类和接口的全限定名；②字段的名称和描述符；③方法的名称和描述符）。

#### 4.1.3 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，是否定义为abstract等。

#### 4.1.4 类索引、弗雷索引与接口索引集合

类索引和父类索引都是一个u2类型的数据；接口索引集合是一组u2类型数据的集合，这三项数据确定这个类的继承关系。

#### 4.1.5字段表集合

字段表用于描述接口或者类中申明的变量，包括类级变量以及实例级变量，但不包括在方法内部申明的局部变量。

#### 4.1.6 方法表集合

方法表的结构依次是：访问标志、名称所以、描述符索引、属性表集合。

#### 4.1.7 属性表集合

### 4.2 字节码与数据类型

#### 4.2.1 字节码与数据类型

操作码助记符： l=long s=short b=byte c=char f=float d=double a=reference

大部分指令都没有支持整数类型byte、char和short，甚至没有任何指令支持Boolean，编译器在编译期或者运行期将byte和short类型的数据带符号扩展为相应的**int**类型数据，将Boolean和char类型数据扩展为int。

#### 4.2.2 加载和存储指令

①局部变量加载到操作栈：x**load** 、x**load\_<n>**

②操作数栈存储到局部变量表：x**store**、x**store\_<n>**

注意：用操作码助记符替换x，<n>代表操作数隐含在指令中。iload\_0 等价于操作数为0的iload指令。

#### 4.2.3 运算指令

加法：add；

减法：sub；

乘法：mul

除法：div

求余：rem

取反：neg

没有直接支持byte、short、char和Boolean类型的算术指令，用int替代。

#### 4.2.4 对象创建于访问指令

①创建类实例：new

②创建数组：newarray、anewarray、multianewarray

③访问类字段（static字段）和实例字段：getfield、putfield、getstatic、putstatic

④数组元素加载到操作数栈：Xaload

⑤操作数栈存储到数组元素：Xastore

⑥数组长度：arraylength

⑦检查类实例类型：instanceof、checkcast

#### 4.2.5 方法调用和返回

invokevirtual：调用对象的实例方法

invokeinterface：调用接口

invokespecial：调用需要特殊处理的实例方法

invokestatic：调用类方法

invokedynamic：运行时动态解析。

## 第五章 虚拟机类加载机制

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。在Java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。

### 5.1 类加载的时机

类被加载到虚拟机内存中开始，到卸载为止，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、解析统称为连接。

**加载、验证、准备、初始化和卸载**这5个阶段的顺序是确定的，类的加载过程必须按照这种顺序按部就班地**开始**，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，这个是为了支持Java语言运行时绑定（也成为动态绑定或晚期绑定）。

虚拟机规范规定有且只有5种情况必须立即对类进行初始化：

①.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；

②.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；

③.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；

④.当虚拟机启动时候，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；

⑤.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。

**被动引用：**

1.通过子类引用父类的静态字段，不会导致子类初始化。

2.通过数组定义来引用类，不会触发此类的初始化。

3.常量在编译阶段会存入调用类的常量池中，本质上并没有直接引用到定义常量的类，因此不会触发定义常量的类的初始化。

4.接口的初始化：接口在初始化时，并不要求其父接口全部完成类初始化，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。

### 5.2 类加载的过程

#### 5.2.1 加载

1)通过一个类的全限定名类获取定义此类的二进制字节流

2)将这字节流所代表的静态存储结构转化为方法区运行时数据结构

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。

**怎么获取二进制字节流？**

1)从ZIP包中读取，这很常见，最终成为日后JAR、EAR、WAR格式的基础

2)从网络中获取，这种场景最典型的应用就是Applet

3)运行时计算生成，这种常见使用得最多的就是动态代理技术

4)由其他文件生成，典型场景就是JSP应用

5)从数据库中读取，这种场景相对少一些（中间件服务器）

**数组类**本身不通过类加载器创建，它是由Java虚拟机直接创建的。**数组类的创建过程遵循以下规则**：

1)如果数组的组件类型(指的是数组去掉一个维度的类型)是**引用类型**，那就递归采用上面的加载过程去加载这个组件类型，数组C将在加载该组件类型的类加载器的类名称空间上被标识；

2)如果数组的组件类型不是引用类型(列如int[]组数)，Java虚拟机将会把数组C标识为与引导类加载器关联

3)数组类的可见性与它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

#### 5.2.2 验证

验证是连接阶段的第一步，这一阶段的目的是为了确保Class文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。

验证阶段会完成下面4个阶段的检验动作：文件格式验证，元数据验证，字节码验证，符号引用验证。

**1.文件格式验证**

第一阶段要验证字节流是否符合Class文件格式的规范，并且能被当前版本的虚拟机处理。这一阶段可能包括：

①.是否以魔数oxCAFEBABE开头

②.主、次版本号是否在当前虚拟机处理范围之内

③.常量池的常量中是否有不被支持的常量类型(检查常量tag标志)

… …

这个阶段的验证时基于**二进制字节流**进行的，只有通过类这个阶段的验证后，**字节流才会进入内存的方法区进行存储**，所以后面的3个验证阶段全部是基于方法区的存储结构进行的，不会再直接操作字节流.

**2.元数据验证**

第二个阶段是对字节码描述的信息进行**语义分析**，保证其描述的信息符合Java语言规范的要求，这个阶段包括：

①.这个类是否有父类(除了java.lang.Object之外,所有的类都应当有父类)

②.这个类的父类是否继承了不允许被继承的类（被final修饰的类）

③.如果这个类不是抽象类，是否实现类其父类或接口之中要求实现的所有方法

④.类中的字段、方法是否与父类产生矛盾(列如覆盖类父类的final字段,或者出现不符合规则的方法重载，列如方法参数都一致，但返回值类型却不同等)

**3.字节码验证**

第三阶段主要目的是通过数据流和控制流分析，确定程序语言是合法的、符合逻辑的。**这个阶段将对类的方法体进行校验分析**，保证被校验类的方法在运行时不会做出危害虚拟机安全的事件。

①.保证任意时刻操作数栈的数据类型与指令代码序列都能配合工作，列如，列如在操作数栈放置类一个int类型的数据，使用时却按long类型来加载入本地变量表中

②.保证跳转指令不会跳转到方法体以外的字节码指令上

③.保证方法体中的类型转换时有效的，列如可以把一个子类对象赋值给父类数据类型，这个是安全的，但是吧父类对象赋值给子类数据类型，甚至把对象赋值给与它毫无继承关系、完全不相干的一个数据类型，则是危险和不合法的

**4.符号引用验证**

这个阶段发生在虚拟机将符号引用转化为直接引用的时候，这个转化动作将在连接的第三阶段——解析阶段中发生。

①.符号引用中通过字符串描述的全限定名是否能找到相对应的类

②.在指定类中是否存在符合方法的字段描述符以及简单名称所描述的方法和字段

③.符号引用中的类、字段、方法的访问性是否可被当前类访问。

#### 5.2.3 准备

**准备阶段是正式为类变量分配内存并设置类变量初始值的阶段，这些变量都在方法区中进行分配。**这个时候进行内存分配的仅包括类变量(被static修饰的变量)，而不包括实例变量，实例变量将会在对象实例化时随着对象一起分配在Java堆中。其次，这里说的初始值通常下是数据类型的零值。

假设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，因为这时候尚未开始执行任何Java方法，而把value赋值为123的putstatic指令是程序被编译后，存放于类构造器<clinit>()方法之中，所以把value赋值为123的动作将在初始化阶段才会执行，但是如果使用**final**修饰，则在这个阶段其初始值设置为123。

#### 5.2.4解析

解析阶段是虚拟机将常量池内符号引用替换为直接引用的过

#### 5.2.5 初始化

类的初始化阶段是类加载过程的最后一步，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，其余动作完全由虚拟机主导和控制。到了初始化阶段，才正真开始执行类中定义的Java程序代码(或者说是字节码)。

### 5.3 类的加载器

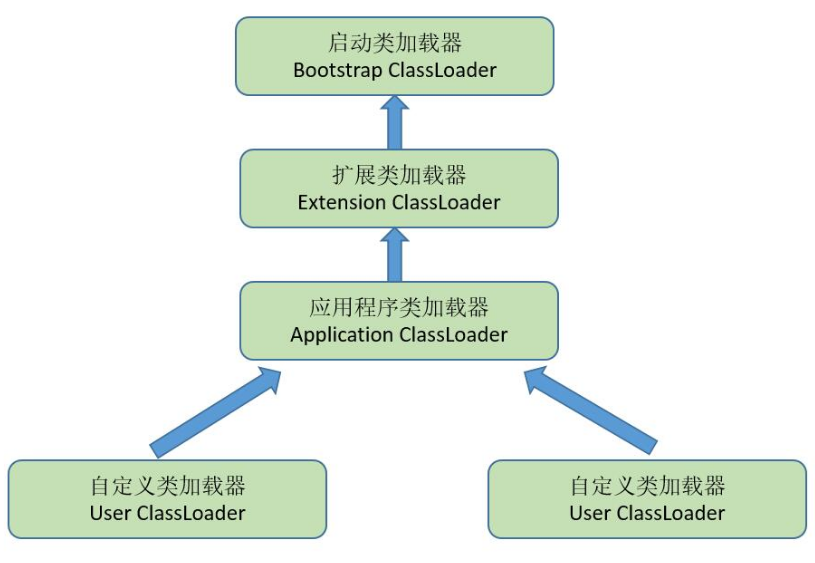
#### 5.3.1 双亲委派模型：

只存在两种不同的类加载器：启动类加载器（Bootstrap ClassLoader），使用C++实现，是虚拟机自身的一部分。另一种是所有其他的类加载器，使用JAVA实现，独立于JVM，并且全部继承自抽象类java.lang.ClassLoader。

①启动类加载器（Bootstrap ClassLoader），负责将存放在<JAVA+HOME>\lib目录中的，或者被-Xbootclasspath参数所制定的路径中的，并且是JVM识别的（仅按照文件名识别，如rt.jar，如果名字不符合，即使放在lib目录中也不会被加载），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。

②扩展类加载器，由sun.misc.Launcher$ExtClassLoader实现，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。

③应用程序类加载器（Application ClassLoader），由sun.misc.Launcher$AppClassLoader来实现。由于这个类加载器是ClassLoader中的getSystemClassLoader()方法的返回值，所以一般称它为系统类加载器。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，一般情况下这个就是程序中默认的类加载器。



这张图表示**类加载器的双亲委派模型**（Parents Delegation model）. 双亲委派模型要求除了顶层的启动加载类外，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，而是使用组合关系来复用父类加载器的代码。

**双亲委派模型的工作过程是**：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，每一个层次的类加载器都是如此，因此所有的加载请求最终都是应该传送到顶层的启动类加载器中，只有当父类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。

**使用双亲委派模型的好处就是：**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，即就是保证某个范围的类一定是被某个类加载器所加载的，这就保证在程序中同 一个类不会被不同的类加载器加载。例如类java.lang.Object,它存放在rt.jar中，无论哪一个类加载器要加载这个类，最终都是委派给处于模型最顶端的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。相反，如果没有使用双亲委派模型，由各个类加载器自行去加载的话，如果用户自己编写了一个称为java.lang.object的类，并放在程序的ClassPath中，那系统中将会出现多个不同的Object类，Java类型体系中最基础的行为也就无法保证，应用程序也将会变得一片混乱。

**双亲委派模型的实现：**实现双亲委派模型代码都集中在java.lang.ClassLoader的loadClass()方法之中，首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，再调用自己的findClass()方法进行加载。

## 第六章 Java内存模型与线程

### 6.1 Java内存模型

#### 6.1.1 主内存与工作内存

Java内存模型的主要目标是定义程序中各个变量的访问规则，即在虚拟机中将变量存储到内存和从内存中取出变量这样的底层细节。此处的变量包括：实例字段、静态字段和构成数组对象的元素，但不包括局部变量与方法参数。

Java内存模型规定了所有的变量都存储在主内存中。每条线程还有自己的工作内存，线程的工作内存中保存了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）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。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。

#### 6.1.2 内存间的交互操作

关于主内存与工作内存之间的具体交互协议，即**一个变量如何从主内存拷贝到工作内存、如何从工作内存同步到主内存之间**的实现细节，Java内存模型定义了以下八种操作来完成，每一种操作都是原子的、不可再分的：

①**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状态。

②**unlock（解锁）**：作用于主内存变量，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。

③**read（读取）**：作用于主内存变量，把一个变量值从主内存传输到线程的工作内存中，以便随后的load动作使用

④**load（载入）**：作用于工作内存的变量，它把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。

⑤**use（使用）**：作用于工作内存的变量，把工作内存中的一个变量值传递给执行引擎，每当虚拟机遇到一个需要使用变量的值的字节码指令时将会执行这个操作。

⑥**assign（赋值）**：作用于工作内存的变量，它把一个从执行引擎接收到的值赋值给工作内存的变量，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。

⑦**store（存储）**：作用于工作内存的变量，把工作内存中的一个变量的值传送到主内存中，以便随后的write的操作。

⑧**write（写入）**：作用于主内存的变量，它把store操作从工作内存中一个变量的值传送到主内存的变量中。

　　如果要把一个变量从主内存中复制到工作内存，就需要按顺寻地执行**read和load**操作，如果把变量从工作内存中同步回主内存中，就要按顺序地执行**stor和write**操作。Java内存模型只要求上述操作**必须按顺序执行，而没有保证必须是连续执行**。也就是read和load之间，store和write之间是可以插入其他指令的。

**Java内存模型还规定了在执行上述八种基本操作时，必须满足如下规则：**

①不允许read和load、store和write操作之一单独出现，既不允许一个变量从主内存读取了但工作内存不接受，或者从工作内存发起回写了但主内存不接受的情况出现。

②不允许一个线程丢弃它的最近assign的操作，即变量在工作内存中改变了之后必须同步到主内存中。

③不允许一个线程无原因地（没有发生过任何assign操作）把数据从工作内存同步回主内存中。

④一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。即就是对一个变量实施use和store操作之前，必须先执行过了assign和load操作。

⑤一个变量在同一时刻只允许一条线程对其进行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unlock操作，变量才会被解锁。lock和unlock必须成对出现

⑥如果对一个变量执行lock操作，将会清空工作内存中此变量的值，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

⑦如果一个变量事先没有被lock操作锁定，则不允许对它执行unlock操作；也不允许去unlock一个被其他线程锁定的变量。

⑧对一个变量执行unlock操作之前，必须先把此变量同步到主内存中（执行store和write操作）。

6.2 重排序

在执行程序时为了提高性能，编译器和处理器经常会对指令进行重排序。重排序分成三种类型：

1.编译器优化的重排序。编译器在不改变单线程程序语义放入前提下，可以重新安排语句的执行顺序。

2.指令级并行的重排序。现代处理器采用了指令级并行技术来将多条指令重叠执行。如果不存在数据依赖性，处理器可以改变语句对应机器指令的执行顺序。

3.内存系统的重排序。由于处理器使用缓存和读写缓冲区，这使得加载和存储操作看上去可能是在乱序执行。

从Java源代码到最终实际执行的指令序列，会经过下面三种重排序：

为了保证内存的可见性，Java编译器在生成指令序列的适当位置会插入内存屏障指令来禁止特定类型的处理器重排序。Java内存模型把内存屏障分为LoadLoad、LoadStore、StoreLoad和StoreStore四种：

#### 6.1.3 对于volatile型变量的特殊规则

关键字volatile可以说是Java虚拟机提供的最轻量级的同步机制。

**当一个变量定义为volatile之后，它将具备两种特性：**

第一：保证此变量对所有线程的可见性。

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。普通变量的值在线程间传递需要通过主内存来完成。Volatile变量在各个线程的工作内存中不存在一致性问题，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（例如：i++）。**由于valatile只能保证可见性，在不符合一下两条规则的运算场景中，我们仍要通过加锁来保证原子性：**

1.运算结果并不依赖变量的当前值，或者能够确保只有单一的线程修改变量的值。

2.变量不需要与其他的状态变量共同参与不变约束。

第二：禁止指令重排序。

**Volatile屏蔽指令重排序的语义在JDK1.5中才被完全修复。**

#### 6.1.4 对于long和double型变量的特殊规则

Java模型要求lock、unlock、read、load、assign、use、store、write这8个操作都具有原子性，但是对于64为的数据类型（long和double），在模型中特别定义了一条相对宽松的规定：允许虚拟机将没有被volatile修饰的64位数据的读写操作分为两次32为的操作来进行，即允许虚拟机实现选择可以不保证64位数据类型的load、store、read和write这4个操作的原子性。目前虚拟机都把64位数据的读写操作作为原子操作来对待。

#### 6.1.5 原子性、可见性和有序性

JAVA内存模型是围绕着并发过程中如何处理原子性、可见性和有序性来建立的：

**①原子性：**即一个操作或者多个操作 要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行。基本数据类型的访问读写是具备原子性的（例外是long和double），在synchronized块之间的操作也具备原子性。

**②可见性：**可见性是指当一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。除了volatile之外，Java还有两个关键字能实现可见性：synchronized和final。

**③有序性：**即程序执行的顺序按照代码的先后顺序执行。Java提供了volatile和synchronized两个关键字来保证线程之间操作的有序性。

#### 6.1.6  先行发生原则

这些先行发生关系无须任何同步就已经存在，如果不再此列就不能保障顺序性，虚拟机就可以对它们任意地进行重排序：

1.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。准确的说，应该是控制顺序而不是程序代码顺序。

2.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。这里必须强调的是同一个锁，而后面的是指时间上的先后顺序。

3.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，这里的后面同样是指时间上的先后顺序

4.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

5.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，我们可以通过Thread.joke()方法结束、ThradisAlive()的返回值等手段检测到线程已经终止执行

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时间的发生，可以通过Thread.interrupted()方法检测到是否有中断发生

7.对象终结规则：一个对象的初始化完成(构造函数执行结束)先行发生于它的finalize()方法的开始

8.传递性：如果操作A先行发生于操作B，操作B先行发生于操作C，那就可以得出操作A先行发生于操作C的结论

### 6.2 Java与线程

#### 6.2.1  Java线程调度

协同式调度：线程的执行时间由线程本身控制

抢占式调度：线程的执行时间由系统来分配

#### 6.2.2 状态转换

1.新建

2.运行：可能正在执行。可能正在等待CPU为它分配执行时间

3.无限期等待：不会被分配CUP执行时间，它们要等待被其他线程显式唤醒

①没有设置Timeout参数的Object.wait()方法

②没有设置Timeout参数的Thread.join()方法

③LockSupport.park()方法

4.限期等待：不会被分配CUP执行时间，它们无须等待被其他线程显式唤醒，一定时间会由系统自动唤醒。

①Thread.sleep();

②设置Timeout参数的Object.wait()方法

③设置Timeout参数的Thread.join()方法

④LockSupport.parkNanos()方法

⑤LockSupport.parkUntil()方法

5.阻塞：阻塞状态在等待这获取到一个排他锁，这个时间将在另一个线程放弃这个锁的时候发生；等待状态就是在等待一段时间，或者唤醒动作的发生

6.结束：已终止线程的线程状态，线程已经结束执行

## 第七章 线程安全与锁优化

### 7.1 线程安全

#### 7.1.1 Java语言中的线程安全

按照线程安全的“安全程度”有强至弱来排序，Java语言中各种操作共享的数据分为以下五类：不可变、绝对线程安全、相对线程安全、线程兼容和线程对立。

1不可变：**不可变的对象一定是线程安全的**、无论是对象的方法实现还是方法的调用者，都不需要再采取任何的线程安全保障。

①Final修饰的基本数据类型；

②对象中带有状态的变量都声明为final

③String

④Long、Double等数值包装类，BigInteger和BigDecimal等大数据类型

⑤AutomicInteger、AutomicLong则不是不可变的

2绝对线程：Vector并非是绝对线程安全的。

3、相对线程安全：相对的线程安全就是我们通常意义上所讲的线程安全，它需要保证对这个对象单独的操作是线程安全的，我们在调用的时候不需要做额外的保障措施，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

4、线程兼容：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使用

5、线程对立：是指无论调用端是否采取了同步措施，都无法在多线程环境中并发使用的代码。

#### 7.1.2 线程安全的实现方法

**1.互斥同步：**

同步是指在多个线程并发访问共享数据时，保证共享数据在同一个时刻只被一个（或者是一些，使用信号量的时候）线程使用。而互斥是实现同步的一种手段，临界区、互斥量和信号量都是主要的互斥实现方式。在Java中，最基本的互斥同步手段就是synchronized关键字，它经过编译之后，会**在同步块的前后分别形成monitorenter和monitorexit**这两个字节码指令，这两个字节码都需要一个reference类型的参数来指明要锁定和解锁的对象。

如果Java程序中的synchronized明确指定了对象参数，那就是这个对象的reference；如果没有指明，那就根据synchronized修饰的是实例方法还是类方法，去取对应的对象实例或Class对象来作为锁对象。

在执行monitorenter指令时，首先要尝试获取对象的锁。如果这个对象没有被锁定，或者当前线程已经拥有了那个对象的锁，把锁的计数器加1，对应的在执行monitorexit指令时会将锁计数器减1，当计数器为0时，锁就被释放。如果获取对象锁失败，那当前线程就要阻塞等待，直到对象锁被另外一个线程释放为止。

首先，Synchronized同步块对同一条线程来说是可重入的，不会出现自己把自己锁死的问题；其次，同步块在已进入的线程执行完之前，会阻塞后面其他线程的进入。

除了Synchronized之外，还可以使用ReentrantLock来实现同步，并且增加了一些高级功能：**等待可中断、可实现公平锁以及锁可以绑定多个条件。**

1.等待可中断：是指当持有锁的线程长期不释放锁的时候，正在等待的线程可以选择放弃等待，改为处理其他事情，可中断特性对处理执行时间非常长的同步块很有帮助。

2.公平锁：是指多个线程在等待同一个锁时，必须按照申请锁的时间顺序来依次获得锁；非公平锁则不能保证这一点，在锁被释放时，任何一个等待锁的线程都有机会获得锁。Synchronized中的锁是非公平的，ReentrantLock默认情况下也是非公平的，但可以通过带布尔值的构造函数要求使用公平锁。

3.锁绑定多个条件是指一个ReentrantLock对象可以同时绑定多个Condition对象，而在synchronized中，锁对象的wait()和notify()或notifyAll()方法可以实现一个隐含的条件，如果要和多余一个的条件关联的时候，就不得不额外地添加一个锁，而ReentrantLock则无须这样做，只需要多次调用newCondition方法即可。

**2.非阻塞同步**

互斥同步最主要的问题就是进行线程阻塞和唤醒锁带来的性能问题，因此这种同步也称为阻塞同步。互斥同步属于一种悲观的并发策略。

基于冲突检测的乐观并发策略，通俗地说，就是先进性操作，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。这种乐观的同步操作就是非阻塞同步。

比较交换（CAS），JDK1.5之后才能使用，该操作由sun.misc.Unsafe类中的compareAndSwapInt()和compareAndSwapLong()等几个方法包装提供。Unsafe的代码中限制了只有启动类加载器加载的class才能访问它，所以不用反射手段，只能通过其他的Java API来间接使用，例如JUC包里面的整数院子类，compareAndSet和getAndIncrement等方法都使用了Unsafe类的CAS操作。

**3.无同步方案**

**可重入代码：**也叫纯代码，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，原来的程序不会出现任何错误。所有的可重入代码都是线程安全的，但是并非所有的线程安全的代码都是可重入的。

判断一个代码是否具备可重入性：如果一个方法，它的返回结果是可预测的，只要输入了相同的数据，就都能返回相同的结果，那它就满足可重入性的要求。

**线程本地存储：**如果一段代码中所需要的数据必须与其他代码共享，那就看看这些共享数据的代码是否能保证在同一个线程中执行？如果能保证，我们就可以把共享数据的可见范围限制在同一个线程之内，这样，无须同步也能保证线程之间不出现数据争用的问题。

Java语言中，如果一个变量要被多线程访问，可以使用volatile关键字声明他为“易变的”，还可以通过threadLocal类来实现线程本地存储的功能。

#### 7.2锁优化

**高效并发是从JDK1.5 到 1.6 的一个重要改进，如适应性自旋、锁消除、锁粗化、轻量级锁和偏向锁，为了在线程之间更高效地共享数据，以及解决竞争问题。**

#### 7.2.1 自旋锁与自适应自旋

**自旋锁：由于共享数据的锁定状态只会持续很短的一段时间，**为了这段时间去挂起和恢复线程并不值得。如果物理机器上有一个以上的处理器，能让两个或以上的线程同时并行执行，我们就可以让后面请求锁的那个线程稍等一下，但不放弃处理器的执行时间，看看持有锁的线程是否很快就会释放锁。为了让线程等待，我们只需让线程执行一个忙循环（自旋），这项技术就是所谓的自旋锁。

在JDK1.6中默认开启自旋锁，自旋等待的时间必须要有一定的限度，如果自旋超过了限定的次数仍然没有成功获得锁，就应当使用传统的方式去挂起线程，自旋的默认值是10次，可以使用参数 –XX：PreBlockSpin来修改。

**自适应自旋转：**是由前一次在同一个锁对象上，自旋等待刚刚成功获得过锁，并且持有锁的线程正在运行中，那么虚拟机就会认为这次自旋也很有可能再次成功，进而它将允许自旋等待持续相对更长的时间。如果对于某个锁，自旋很少成功获得过，那在以后要获取这个锁时将可能省略掉自过程，以避免浪费处理器资源。

#### 7.2.2 锁消除

锁消除是指虚拟机即时编辑器在运行时，对一些代码上要求同步，但是被检测到不可能存在共享数据竞争的锁进行消除。如果在一段代码中。推上的所有数据都不会逃逸出去从而被其他线程访问到，那就可以把它们当作栈上数据对待，认为它们是线程私有的，同步加锁自然就无须进行

#### 7.2.3锁粗化

如果虚拟机检测到有一串零碎的操作都是对同一对象的加锁，将会把加锁同步的范围扩展（粗化）到整个操作序列的外部

#### 7.2.4 轻量级锁

轻量级锁是JDK1.6之中加入的新型锁机制，他名字中的“轻量级”是相对于使用操作系统互斥量来实现的传统锁而言的，因此传统的锁机制就成为“重量级”锁。

HotSpot虚拟机的对象头分为两部分信息：第一部分用于存储对象自身的运行时数据，如哈希码、GC分代年龄等，这部分长度分别为32bit和64bit，官方称它为“Mark Word”；另外一部分用于存储指向方法区对象类型数据的指针，如果是数组对象的话，还会有一个额外的部分用于存储数组长度。

**轻量级锁的加锁执行过程：**在代码进入同步块的时候，如果此同步对象没有被锁定（锁标志位为“01”状态），虚拟机首先将在当前线程的栈帧中建立一个名为锁记录的空间，用于存储锁对象目前的Mark Word的拷贝。然后虚拟机使用CAS操作尝试将对象Mark Word更新为指向Lock Record 的指针，如果成功，这个线程就拥有了该对象的锁，将锁标志位这只为“00”，表示此对象处于轻量级锁定状态。如果更新失败，虚拟机首先检查对象的Mark Word是否指向当前线程的栈帧，如果是说明当前线程已经拥有了这个对象的锁，那就直接进入同步块继续执行，否则说明这个锁对象已经被其他线程抢占。

**轻量级锁的解锁执行过程：通过CAS操作操作，如果对象的**Mark Word仍然指向了线程的锁记录，那就用CAS操作把对象当前的Mark Word和线程中渎职的Displaced Mark Word替换回来，如果成功，整个过程完成，如果失败，则说明有其他线程尝试过获取该锁，那就在释放锁的同时，唤醒被挂起的线程。

如果有两条以上的线程争用同一个锁，那轻量级锁就不再有效，要膨胀为重量级锁，锁标志的状态变为“10”。

如果存在锁竞争，除了互斥量的开销外，还额外放生了CAS操作，因此在有竞争的情况下，轻量级锁会比传统的重量级锁更慢。

#### 7.2.5 偏向锁

偏向锁是JDK1.6中引入的一项锁优化。它的目的是消除无竞争情况下的同步原语，进一步提高程序的运行性能。如果轻量级锁是在无竞争的情况下使用CAS操作去消除同步使用的互斥量，那偏向锁就是在无竞争的情况下把这个同步都消除掉，CAS操作都不做了。

这个锁会偏向于第一个获得它的线程，如果接下来的执行过程中，该锁没有被其他线程获取，则持有偏向锁的线程将永远不需要在进行同步。当另一个线程去尝试获取这个锁时，偏向模式就宣告结束。

## 八、逃逸分析

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，成为方法逃逸。甚至还可能被外部线程访问到，比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。

如果一个对象不会逃逸到方法或线程之外，也就是别的方法或线程无法通过任何途径访问到这个对象，则可能为这个变量进行一些高效的优化。

**栈上分配：**如果确定一个对象不会逃逸出方法外，那让这个对象在栈上分配内存将会是一个不错的注意，对象所占用的内存空间就可以随栈帧出栈而销毁。如果能使用栈上分配，那大量的对象就随着方法的结束而销毁了，垃圾收集系统的压力将会小很多

**同步消除：**如果确定一个变量不会逃逸出线程，无法被其他线程访问，那这个变量的读写肯定就不会有竞争，对这个变量实施的同步措施也就可以消除掉

**标量替换：**标量就是指一个数据无法在分解成更小的数据表示了，int、long等及refrence类型等都不能在进一步分解，它们称为标量。

如果一个数据可以继续分解，就称为聚合量，Java中的对象就是最典型的聚合量

如果一个对象不会被外部访问，并且这个对象可以被拆散的化，那程序正整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

**二、Java内存分配参数**

**1.堆内存**

-Xmx 最大堆内存

-Xms 最小堆内存

-XX:MinHeapFreeRatio 设置堆空间最小空闲比例

-XX:MaxHeapFreeRatio 设置堆空间最大空闲比例

**2.新生代**

-Xmn 设置新生代

3.JVM垃圾回收机制采用“分代收集”：新生代采用复制算法，老年代采用标记清理算法。

在JVM运行时，可以通过配置以下参数改变整个JVM堆的配置比例

1.Java heap的大小（新生代+老年代）

　　-Xms堆的最小值

　　-Xmx堆空间的最大值

2.新生代堆空间大小调整

　　-XX:NewSize新生代的最小值

　　-XX:MaxNewSize新生代的最大值

　　-XX:NewRatio设置新生代与老年代在堆空间的大小

　　-XX:SurvivorRatio新生代中Eden所占区域的大小

3.永久代大小调整

　　-XX:MaxPermSize

4.其他

　 -XX:MaxTenuringThreshold,设置将新生代对象转到老年代时需要经过多少次垃圾回收，但是仍然没有被回收

**对复制算法进一步优化：使用Eden/S0/S1三个分区**

平均分成A/B块太浪费内存，采用Eden/S0/S1三个区更合理，空间比例为Eden:S0:S1==8:1:1，有效内存（即可分配新生对象的内存）是总内存的9/10。