第2章 Java内存区域与内存溢出异常

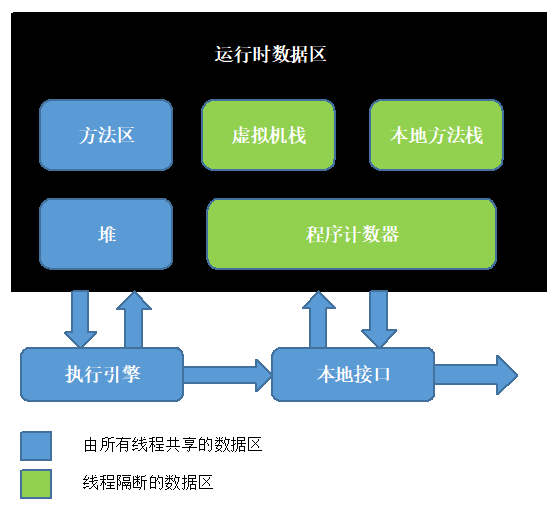
2.2 运行时数据区域

图2-1 Java虚拟机运行时数据区

2.2.1 程序计数器

程序计数器（Program Counter Register）是一块较小的内存空间，它可以看做是当前线程所执行的字节码的行号指示器。

由于Java虚拟机的多线程是通过线程轮流切换并分配处理器执行时间的方式来实现的，在任何一个确定的时刻，一个处理器（对于多核处理器来说是一个内核）都只会执行一条线程中的指令。因此，为了线程切换后能恢复到正确的执行位置，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互不影响，独立存储，我们称这类内存区域为“线程私有”的内存。

2.2.2 Java虚拟机栈

与程序计数器一样，Java虚拟机栈（Java Virtual Machine Stacks）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。每个方法从调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中入栈到出栈的过程。

2.2.3 本地方法栈

本地方法栈与虚拟机栈所发挥的作用是非常相似的，它们之间的区别不过是虚拟机栈为虚拟机执行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。

2.2.4 Java堆

Java堆是被所有线程共享的一块内存区域，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。

2.2.5 方法区

方法区与Java堆一样，是各个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、即时编译器编译后的代码等。

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。

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，首先将去检查这个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被加载、解析和初始化过。如果没有，那必须先执行相应的类加载过程。

2.3.2 对象的内存布局

在HotSpot虚拟机中，对象在内存中存储的布局可以分为三块区域：**对象头**、**实例数据**和**对齐填充**。

HotSpot虚拟机的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，如HashCode、GC分代年龄、锁状态标志、线程持有的锁、偏向线程、偏向时间戳等，在部分数据的长度在32位和64位虚拟机中分别为32bit和64bit，官方称它为“Mark Word”。

对象头的另一部分是类型指针，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。

接下来的实例数据部分是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。

第三部分对齐填充并不是必然存在的，它仅仅起着占位符的作用。

2.3.2 对象的访问定位

Java程序需要通过栈上的reference数据来操作堆上的具体对象，目前主流的访问方式有使用句柄和直接指针两种。

* 如果使用句柄多话，那么堆上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，而句柄中包含了对象实例数据与类型数据各自的具体地址信息。

Java栈

本地变量表

int

short

reference

int

double

float

**Java堆**

到对象实例数据的指针

到对象类型数据的指针

**句柄池**

**实例池**

**方法区**

图2-2 通过句柄访问对象

* 如果使用直接指针访问，那么Java堆对象的布局中就必须考虑如何放置访问类型数据的相关信息，而reference中存储的直接就是对象地址。

图2-3 通过直接指针访问

**方法区**

Java栈

本地变量表

int

short

reference

int

double

float

**Java堆**

到对象类型数据的指针

这两种对象访问方式各有优势，使用句柄来访问的最大好处就是reference中存储的是稳定的句柄地址，在对象被移动（垃圾收集时移动对象是非常普遍的行为）时只会改变句柄中的实例数据指针，而reference本身不需要修改。

使用直接指针访问方式的最大好处就是速度更快，它节省了一次指针定位的时间开销，由于对象的访问在Java中非常频繁，因此这类开销积少成多后也是一项非常可观的执行成本。

第3章 垃圾收集器与内存分配策略

3.2 对象已死吗

3.2.1 引用计数算法

算法原理：给对象中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。

3.2.2 可达性分析算法

这个算法的基本思路是通过一系列的称为“GC Roots”的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链（Reference Chain），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。如图3-1所示。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。

GC Root Set

GC Roots

Object 1

Object 2

Object 3

Object 4

Object 5

Object 6

Object 7

仍然存活的对象

判定可回收的对象

图3-1 可达性分析算法判定对象是否可回收

在Java语言中，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（即一般说的Native方法）引用的对象。

3.2.3 再谈引用

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

* **强引用**就是指在程序代码之中普遍存在的，类似“Object obj = new Object()”这类的引用，只要强引用还存在，垃圾收集器永远不会回收掉被引用的对象。
* **软引用**是用来描述一些还有用但并非必需的对象。对于软引用关联着的对象，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。在JDK1.2之后，提供了SoftReference类来实现软引用。
* **弱引用**也是用来描述非必需对象的，但是它的强度比软引用更弱一些，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。当垃圾收集器工作时，无论当前内存是否足够，都会回收掉只被弱引用关联的对象。在JDK1.2之后，提供了WeakReference类来实现软引用。
* **虚引用**也称为幽灵引用或者幻影引用，它是最弱的一种引用关系。一个对象是否有虚引用的存在，完全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

3.2.4 生存还是死亡

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。当对象没有覆盖finalize()方法，或者finalize()方法已被虚拟机调用过，虚拟机将这两种情况都视为“没有必要执行”。

如果这个对象被判定为有必要执行finalize()方法，那么这个对象将会被设置在一个叫做F-Queue的队列之中，并在稍后由一个由虚拟机自动建立的、低优先级的Finalizer线程去执行它。这里所谓的“执行”是指虚拟机会触发这个方法，但并不承诺会等待它运行结束，这样做的原因是，如果一个对象在finalize()方法中执行缓慢，或者发生了死循环，将很可能会导致F-Queue队列中其他对象永久处于等待，甚至导致整个内存回收系统崩溃。finalize()方法是对象逃脱死亡命运的最后一次机会，稍后GC将对F-Queue中的对象进行第二次小规模的标记，如果对象要在finalize()中成功拯救自己——只要重新与引用链上的任何一个对象建立关联即可，譬如把自己赋值给某个类变量或者对象的成员变量，那么第二次标记时它将被移除出“即将回收”的集合；如果对象这时候还没有逃脱，那基本上它就真的被回收了。

3.2.5 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。回收废弃常量与回收Java堆中的对象非常相似。以常量池中字面量的回收为例，假如一个字符串“abc”已经进入了常量池中，但是当前系统没有任何一个String对象是叫做“abc”的，换句话说，就是没有任何String对象引用常量池中的“abc”常量，也没有其他地方引用了这个字面量，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。

类需要同时满足下面3个条件才能算是“无用”的类：

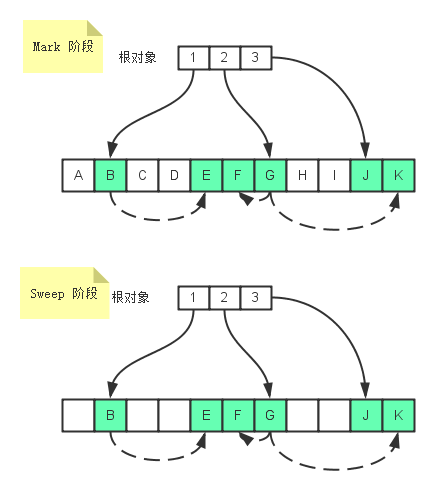
* 该类所有的实例都已经被回收，也就是Java堆中不存在该类的任何实例。
* 加载该类的ClassLoader已经被回收。
* 该类对应的java.lang.Class对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法。

虚拟机可以对满足上述3个条件的无用类进行回收，这里说的仅仅是“可以”，而并不是和对象一样，不使用了就必然会回收。

3.3 垃圾收集算法

3.3.1 标记-清除算法

标记清除（Mark-Sweep）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以后在程序运行过程中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出发另一次垃圾收集动作。标记清除法的执行过程如图3-2所示。

图3-2 标记清除法

3.3.2 复制算法

为了解决效率问题，一种称为“复制”的收集算法出现了，它将利用内存容量划分为大小相等的两块，每次只是用其中的一块。当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，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复杂情况，只要移动堆顶指针，按顺序分配内存即可，实现简单，运行高效。只是这种算法的代价是将内存缩小为原来的一半。

3.3.3 标记-整理算法

标记-整理算法在标记-清除算法基础上做了改进，标记阶段是相同的标记出所有需要回收的对象，在标记完成之后不是直接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，在移动过程中清理掉可回收的对象，这个过程叫做整理。

标记-整理算法相比标记-清除算法的优点是内存被整理以后不会产生大量不连续内存碎片问题。

3.3.4 分代收集算法

当前商业虚拟机基本上都是采用分代垃圾回收算法来回收垃圾，思想也很简单，就是根据对象的生命周期将内存划分，然后进行分区管理。

在Java虚拟机分代垃圾回收机制中，应用程序可用的堆空间可以分为新生代和老年代，新手代的对象98%是“朝生夕死”的，所以并不需要按照1:1的比例来划分内存空间，而是将内存分为一块较大Eden空间和两块较小的Survivor空间，每次使用Eden和其中一块Survivor。

**Eden**

**Young Generation**

**Old Generation**

**Survivor 1**

**Survivor 2**

图3-3 老年代和新生代

当回收时，将Eden和Survivor中还存活着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（复制算法）。HotSpot虚拟机默认Eden和Survivor的大小比例是8:1.

3.5 垃圾收集器

**Serial**

**ParNew**

**Parallel**

**Scavenge**

**CMS**

**Parallel Old**

**Serial Old**

**(MSC)**

**Yound generation**

**Tenured generation**

**G1**

图3-5 HotSpot虚拟机的垃圾收集器

3.5.1 Serial收集器

这个收集器是一个单线程的收集器，但它的“单线程”的意义并不仅仅说明它只会使用一个CPU或一条收集线程去完成垃圾收集工作，更重要的是在它进行垃圾收集时，必须暂停其他所有的工作线程，直到它收集结束。

用户线程1

CPU 0

用户线程2

CPU 1

用户线程3

CPU 2

用户线程4

CPU 3

**SafePoint**

GC 线程

新生代采取复制算法

老年代采取标记-整理算法

暂停所有用户线程

图3.6 Serial/Serial Old收集器运行示意图

Serial收集器优于其他收集器的地方：简单而高效（与其他收集器的单线程比），对于限定单个CPU的环境来说，Serial收集器由于没有线程交互的开销，专心做垃圾收集自然可以获得最高的单线程收集效率。

3.5.2 ParNew收集器

ParNew收集器其实就是Serial收集器的多线程版本，除了使用多线程进行垃圾收集之外，其余行为包括Serial收集器可用的所有控制参数、收集算法、Stop The World、对象分配规则、回收策略等都与Serial收集器完全一样。

用户线程1

CPU 0

用户线程2

CPU 1

用户线程3

CPU 2

用户线程4

CPU 3

**SafePoint**

GC 线程2

GC 线程1

GC 线程3

图3-7 ParNew/Serial Old收集器运行示意图

3.5.3 Parallel Scavenge收集器

Parallel Scavenge收集器是一个新生代收集器，它也是使用复制算法的收集器，又是并行的多线程收集器。

Parallel Scavenge收集器的特点是它的关注点与其他收集器不同，CMS等收集器的关注点是尽可能地缩短垃圾收集时用户线程的停顿时间，而Parallel Scavenge收集器的目标则是达到一个**可控制的吞吐量**。所谓吞吐量就是CPU用于运行用户代码的时间和CPU总消耗时间的比值。

Parallel Scavenge收集器提供了两个参数用于精确控制吞吐量，分别是控制最大垃圾收集停顿时间的-XX:MaxGCPauseMillis以及直接设置吞吐量大小的-XX:GCTimeRatio。

MaxGCPauseMillis参数允许的值是一个大于0的毫秒数，收集器将尽可能地保证内存回收花费的时间不超过设定值。GC停顿时间缩短是以牺牲吞吐量和新生代空间来换取的。

GCTimeRatio参数的值应当是一个大于0且小于100的整数，也就是垃圾收集时间占总时间的比率，相当于是吞吐量的倒数。

由于与吞吐量关系密切，Parallel Scavenge收集器也经常称为“吞吐量优先”收集器。

3.5.4 Serial Old收集器

Serial Old是Serial收集器的老年代版本，它同样是一个单线程收集器，使用“标记-整理”算法。这个收集器的主要意义也是在于给Client模式下的虚拟机使用。

3.5.5 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。

3.5.6 CMS收集器

CMS（Concurrent Mark Sweep）收集器是一种以获取**最短回收停顿时间**为目标的收集器。CMS收集器是基于“标记­清除”算法实现的，整个过程分为4个步骤：

* 初始标记（CMS initial Mark）
* 并发标记（CMS concurrent Mark）
* 重新标记（CMS remark）
* 并发清除（CMS concurrent sweep）

其中，初始标记、重新标记这两个步骤仍需要“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，这个阶段的停顿时间一般会比初始标记阶段稍长一些，但远比并发标记的时间短。

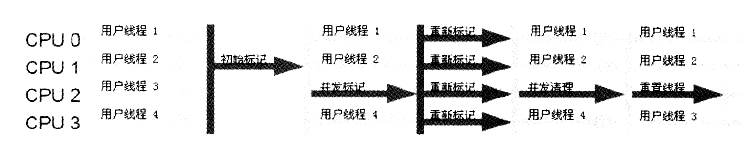
由于整个过程中耗时最长的并发标记和并发清除过程收集器线程都可以与用户线程一起工作，所以，从总体上来说，CMS收集器的内存回收过程是与用户线程一起并发执行的。

图3-10 CMS收集器运行示意图

优点：并发收集、低停顿。

缺点：

（1）CMS收集器对CPU资源非常敏感。

（2）CMS收集器无法处理浮动垃圾，可能出现“Concurrent Mode Failure”失败而导致另一次Full GC的产生。由于CMS并发清理阶段用户线程还在运行着，伴随程序运行自然就还会有新的垃圾不断产生，这一部分垃圾出现在标记过程之后，CMS无法在当次收集中处理掉它们，只好留待下一次GC时再清理掉。这一部分垃圾就称为“浮动垃圾”。

要是CMS运行期间预留的内存无法满足程序需要，就会出现一次“Concurrent Mode Failure”失败，这时虚拟机将启动后备预案：临时启用Serial Old收集器来重新进行老年代的垃圾收集，这样停顿时间就很长了。

（3）由于CMS是一款基于“标记—清除”算法实现的收集器，意味着收集结束后会有大量空间碎片产生。空间碎片过多时，将会给大对象分配带来很大麻烦，往往会出现老年代还有很大空间剩余，但是无法找到足够大的连续空间来分配当前对象，不得不提前触发一次Full GC。

3.5.7 G1收集器

**特点**：

* **并行与并发**：G1能充分利用多CPU、多核环境下的硬件优势，使用多个CPU来缩短Stop-The-World（停止Java线程）的时间，部分其他收集器原本需要停顿Java线程执行的GC动作，G1收集器仍然可以通过并发的方式让Java程序继续执行。
* **分代收集**：与其他收集器一样，分代概念在G1中依然得以保留。虽然G1可以不需要其他收集器配合就能独立管理整个GC堆，但它能够采用不同的方式去处理新创建的对象和已经存活了一段时间、熬过多次GC的旧对象以获取更好的收集效果。
* **空间整合**：与CMS的“标记—清理”算法不同，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）上来看是基于“复制”算法实现的，但无论如何，这两种算法都意味着G1运作期间不会产生内存空间碎片，收集后能提供规整的可用内存。这种特性有利于程序长时间运行，分配大对象时不会因为无法找到连续空间而提前触发下一次GC。
* **可预测的停顿**：这是G1相对于CMS的另一大优势，降低停顿时间是G1和CMS共同的关注点，但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，这几乎已经是实时Java（RTSJ）的垃圾收集器的特征了。

使用G1收集器时，Java堆的内存布局就与其他收集器有很大差别，它将整个Java堆划分为多个大小相等独立区域（Region），虽然还保留有新生代和老年代的概念，但新生代和老年代不再是物理隔离的了，它们都是一部分Region（不需要连续）的集合。

G1收集器之所以能建立可预测的停顿时间模型，是因为它可以有计划地避免在整个Java堆中进行全区域的垃圾收集。G1跟踪各个Region里面的垃圾堆积的价值大小（回收所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，优先回收价值最大的Region（Garbage-First）。这种使用Region划分内存空间以及由优先级的区域回收方式，保证了G1收集器在有限的时间内可以获取尽可能高的收集效率。

在G1收集器中，Region之间的对象引用以及其他收集器中的新生代和老年代之间的对象引用，虚拟机都是使用Remembered Set来避免全堆扫描的。G1中每个Region都有一个与之对应的Remembered Set，虚拟机发现程序在对Reference类型的数据进行写操作时，会产生一个Write Barrier暂时中断写操作，检查Reference引用的对象是否处于不同的Region之中（在分代的例子中就是检查是否老年代中的对象引用了新生代中的对象），如果是，便通过CardTable把相关引用信息记录到被引用对象所属的Region的Remembered Set即可保证不对全堆扫描也不会有遗漏。

如果不计算维护Remembered Set的操作，G1收集器的运作可划分为以下几个步骤：

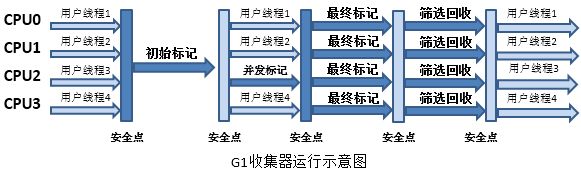
* **初始标记**：同CMS初始标记步骤
* **并发标记**：同CMS并发标记步骤
* **最终标记**：虚拟机将这段时间对象变化记录在线程Remembered Set Logs里面，最终标记阶段需要把Remembered Set Logs的数据合并到Remembered Set中，这阶段需要停顿线程，但是可并行执行。
* **筛选回收**：首先对各个Region的回收价值和成本进行排序，根据用户所期望的GC停顿时间来制定回收计划。

图3-11 G1收集器运行示意图

3.6 内存分配与回收策略

Java技术体系中所提倡的自动内存管理最终可以归结为自动化地解决了两个问题：给对象分配内存以及回收分配给对象的内存。

对象的内存分配，往大方向讲，就是在堆上分配，对象主要分配在新生代的Eden区上，如果启动了本地线程分配缓冲，将按线程优先在TLAB上分配。少数情况下也可能会直接分配在老年代中，分配规则并不是百分之百固定的，其细节取决于当前使用的是哪一种垃圾收集器组合，还有虚拟机中内存相关的参数的设置。

3.6.1 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，虚拟机将发生一次Minor GC。

* 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质，所以Minor GC非常频繁，一般回收速度也比较快。
* 老年代GC（Major GC/Full GC）：指发生在老年代的GC，出现了Major GC，经常会伴随至少一次的Minor GC（但非绝对的，在Parallel Scavenge收集器的收集策略里就有直接进行Major GC的策略选择过程）。Major GC的速度一般会比Minor GC慢10倍以上。

第6章 类文件结构

6.3 Class类文件结构

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，中间没有添加任何分隔符，这使得整个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位字节以上空间的数据项时，则会按照高位在前的方式分割成若干个8位字节进行存储**。

6.3.1 魔数与Class文件的版本

6.3.2 常量池

6.3.3 访问标志

6.3.4 类索引、父类索引与接口索引集合

6.3.5 字段表集合

6.3.6 方法表集合

6.3.7 属性表集合

第7章 虚拟机类加载机制

7.2 类加载的时机

类被加载到虚拟机内存中开始，到卸载出内存位置，它的整个生命周期包括：加载、验证、准备、解析、初始化、使用和卸载七个阶段。其中验证、准备、解析三个部分统称为连接。

**加载**

**验证**

**准备**

**解析**

**卸载**

**使用**

**初始化**

**连接**

图7-1 类的生命周期

加载、验证、准备、初始化和卸载这5个阶段的顺序是确定的，类的加载过程必须按照这种顺序按部就班的**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，这是为了支持java语言的运行时绑定。这些阶段通常都是互相交互地混合式进行的，通常会在一个阶段的过程中调用、激活另一个阶段。

对于初始化阶段，虚拟机规范严格规定了有且只有5种情况必须立即对类进行“初始化”（而，加载、验证、准备自然需要在此之前开始）：

（1）遇到new、getstatic、putstatic和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。

（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。（当一个接口初始化时，并不要求其父接口全部都完成了初始化，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）

（4）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。

（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。

这5种场景中的行为称为对一个类的主动引用。除此之外，所有引用类的方式都不会触发初始化，称为被动引用。例如：

（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。

7.3 类加载的过程

7.3.1 加载

在加载阶段，虚拟机需要完成以下3件事情：

（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，作为方法区中这个类的各种数据的访问入口

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由虚拟机实现自行定义，虚拟机规范未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，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。

7.3.2 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。

验证阶段大致上会完成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。

**2.元数据验证**

第二阶段是对字节码描述的信息进行语义分析，以保证其描述的信息符合Java语言规范的要求。

**3.字节码验证**

第三阶段是整个验证过程中最复杂的一个阶段，主要目的是通过数据流和控制流分析，确定程序语义是合法的、符合逻辑的。在第二阶段对元数据信息中的数据类型做完校验后，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。

**4.符号引用验证**

最后一个阶段的检验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。符号引用验证可以看做是对类自身以外（常量池中的各种符号引用）的信息进行匹配性校验。

7.3.3 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，这些变量所使用的内存都将在**方法区**中进行分配。首先，这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，这里所说的初始值“通常情况”下是数据类的零值。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属性所指定的值，假设上面类变量value的定义变为：

public static final int value = 123;

编译时Javac将会为value生成ConstantValue属性，在准备阶段虚拟机就会根据ConstantValue的设置将value赋值为123。

7.3.4 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，在Class文件中它以CONSTANT\_Class\_info、CONSTANT\_Fieldref\_info、CONSTANT\_Methodref\_info等类型的常量出现。

* **符号引用**：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地定位到目标即可。符号引用与虚拟机实现的内存布局无关，引用的目标并不一定已经加载到内存中。各种虚拟机实现的内存布局可以各不相同，但是它们能接受的符号引用必须都是一致的，因为符号引用的字面量形式明确定义在Java虚拟机规范的Class文件格式中。
* **直接引用**：直接引用可以是直接指向目标的指针、相对偏移量或是一个能间接定位到目标的句柄。直接引用是和虚拟机实现的内存布局相关的，同一个符号引用在不同虚拟机实例上翻译出来的直接引用一般不会相同。如果有了直接引用，那引用的目标必定已经在内存中存在

**1.类或接口的解析**

假设当前代码所处的类为D，如果要把一个从未解析过的符号引用N解析为一个类或接口C的直接引用，那虚拟机完成整个解析的过程需要以下3个步骤：

（1）如果C不是一个数组类型，那虚拟机将会把代表N的全限定名传递给D的类加载器去加载这个类C。在加载过程中，由于元数据验证、字节码验证的需要，又可能触发其他相关类的加载动作，例如加载这个类的父类或实现的接口。一旦这个加载过程出现了任何异常，解析过程就宣告失败。

（2）如果C是一个数组类型，并且数组的元素类型为对象，也就是N的描述符会是类似“[Ljava/lang/Integer”的形式，那将会按照第1点的规则加载数组元素类型。如果N的描述符如前面所假设的形式，需要加载的元素类型就是”java.lang.Integer”，接着由虚拟机生成一个代表此数组维度和元素的数组对象。

（3）如果上面的步骤没有出现任何异常，那么C在虚拟机中实际上已经成为一个有效的类或接口了，但在解析完成之前还有进行符号引用验证，确认D是否具备对C的访问权限。如果发现不具备访问权限，将抛出java.lang.IllegalAccessError异常。

**2.字段解析**

要解析一个未被解析过的字段符号引用，首先将会对字段表内class\_index项中索引的CONSTANT\_Class\_info符号引用进行解析，也就是字段所属的类或接口的符号引用。如果在解析这个类或接口符号引用的过程中出现了任何异常，都会导致字段符号引用解析的失败。如果解析成功，那将这个字段所属的类或接口用C表示，虚拟机规范要求按照如下步骤对C进行后续字段的搜索。

（1）如果C本身就包含了简单名称和字段描述符都与目标相匹配的字段，则返回这个字段的直接引用，查找结束。

（2）否则，如果在C中实现了接口，将会按照继承关系从下往上递归搜索各个接口和它的父接口，如果接口中包含了简单名称和字段描述符都与目标相匹配的字段，则返回这个字段的直接引用，查找结束。

（3）否则，如果引用C不是java.lang.Object的话，将会按照继承关系从下往上递归搜索其父类，如果在父类中包含了简单名称和字段描述符都与目标相匹配的字段，则返回这个字段的直接引用，查找结束。

（4）否则，查找失败，抛出java.lang.NoSuchFieldError异常。

**3.类方法解析**

类方法解析的第一个步骤与字段解析一样，也需要先解析出类方法表的class\_index项中索引的方法所属的类或接口的符号引用，如果解析成功，我们依然用C表示这个类，接下来虚拟机将会按照如下步骤进行后续的类方法搜索。

（1）类方法和接口方法符号引用的常量类型定义是分开的，如果在类方法表中发现class\_index中索引的C是个接口，那就直接抛出IncompatibleClassChangeError异常。

（2）如果通过了第1步，在类C中查找是否有简单名称和描述符都与目标相匹配的方法，如果有则返回这个方法的直接引用，查找结束。

（3）否则，在类C的父类中递归查找是否有简单名称和描述符都与目标相匹配的方法，如果有则返回这个方法的直接引用，查找结束。

（4）否则，在类C实现的接口列表及它们的父接口之中递归查找是否有简单名称和描述符都与目标相匹配的方法，如果存在匹配的方法，说明类C是一个抽象类，这时查找结束，抛出AbstractMethodError异常

（5）否则，宣告方法查找失败，抛出NoSuchMethodError。

**4 接口方法解析**

类似于类方法解析。

7.3.5 初始化

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主观计划去初始化类变量和其他资源，或者从另一个角度来表达：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{}块）中的语句合并产生的，编译器收集的顺序是由语句在源文件中出现的顺序所决定的，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。

代码清单7-5 非法向前引用变量

|  |
| --- |
| public class Test {  static {  i = 0; // 给变量赋值可以正常通过  System.out.println(i); // ” Cannot reference a field before it is defined”  }  static int i = 1;  } |

* <clinit>()方法与类的构造器（或者说实例构造器<init>()方法）不同，它不需要显示地调用父类构造器，虚拟机会保证在子类<clinit>()方法执行之前，父类的<clinit>()方法已经执行完毕。因此在虚拟机中第一个被执行的<clinit>()方法的类肯定是java.lang.Object。

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。

代码清单7-6 <clinit>()方法执行顺序

|  |
| --- |
| **public** **class** ParentStaticInitFirst {  **static** **class** Parent {  **public** **static** **int** *i* = 1;  **static** {  *i* = 2;  }  }  **static** **class** Sub **extends** Parent {  **public** **static** **int** *j* = *i*;  }    **public** **static** **void** main(String[] args) {  System.***out***.println(Sub.*j*);  }  } |

* <clinit>()对于类或接口来说并不是必需的，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
* 接口中不能使用静态语句块，但仍然有变量初始化的赋值操作，因此接口与类一样都会生成<clinit>()。但接口与类不同的是，执行接口的<clinit>()不需要先执行父接口的<clinit>()。只有当父接口中定义的变量使用时，父接口才会初始化。另外，接口的实现类在初始化时也一样不会执行接口的<clinit>()。
* 虚拟机会保证一个类的<clinit>()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，那么只会有一个线程去执行这个类的<clinit>()，其他线程都需要阻塞等待，知道活动线程执行<clinit>()完毕。如果在一个类的<clinit>()中有耗时很长的操作，就可能造成多个线程阻塞。

7.4 类加载器

7.4.1 类与类加载器

类加载器虽然只用于实现类的加载动作，但它在Java程序中起到的作用却远远不限于类加载阶段。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，每一个类加载器，都拥有一个独立的类名称空间。

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：一种启动类加载器，这个类加载器使用C++语言实现，是虚拟机自身的一部分；另一种就是所有其他的类加载器，这些类加载器都是有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。

* **启动类加载器**：这个类加载器负责将存放在lib目录下的，或者被-Xbootclasspath

参数所指定的路径中的，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。

* **扩展类加载器**：这个加载器由sun.misc.Launcher$ExtClassLoader实现，它负责加载lib/ext目录中的，或者被java.ext.dirs系统变量指定的路径中的所有类库，开发者可以直接使用扩展类加载器。
* **应用程序类加载器**：这个类加载器由sun.misc.Launcher$APPClassLoader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，一般情况下这个就是程序中默认的加载器。

**启动类加载器**

**扩展类加载器**

**应用程序类加载器**

**应用程序类加载器**

**应用程序类加载器**

图7-2 类加载器双亲委派模型

双亲委派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。这里类加载器之间的父子关系一般不会以继承的关系实现，而是都使用组合关系来复用父加载器的代码。

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，子加载器才会尝试去加载。

使用双亲委派模型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。

第8章 虚拟机字节码执行引擎

8.2 运行时栈帧结构

栈帧是用于支持虚拟机进行方法调用和方法执行的数据结构，它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、动态连接和方法返回地址等信息。每一个方法从调用至执行完成的过程，都对应着一个栈帧在虚拟机里面从入栈到出栈的过程。

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，而仅仅取决于具体地虚拟机实现。

**……**

**当前栈帧**

局部变量表

操作栈

动态连接

返回地址

**栈帧1**

**栈帧2**

**当前线程**

**线程2**

**线程n**

图8-1 栈帧的概念结构

8.2.1 局部变量表

局部变量表是一组变量值存储空间，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，就在方法的Code属性的max\_locals数据项中确定了该方法所需要分配的局部变量表的最大容量。

局部变量表的容量以变量槽(Slot)为最小单位。

在方法执行时，虚拟机是使用局部变量表完成参数值到参数变量列表的传递过程的，如果执行的是实例方法，那局部变量表中第0位索引的Slot默认是用于传递方法所属对象实例的引用，在方法中可以通过关键字“this”来访问到这个隐含的参数。其余参数则按照参数表顺序排列，占用从1开始的局部变量Slot，参数表分配完毕后，再根据方法体内部定义的变量顺序和作用域分配其余的Slot。

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，方法体中定义的变量，其作用域并不一定会覆盖整个方法体，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，那这个对应的Slot就可以交给其他变量使用。

8.2.2 操作数栈

操作数栈也称为操作栈，它是一个后入先出栈。同局部变量表一样，操作数栈的最大深度也在编译的时候写入到Code属性的max\_stacks数据项中。

8.2.3 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，持有这个引用是为了支持方法调用过程中的动态连接。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，这部分称为动态连接。

8.2.4 方法返回地址

当一个方法开始执行时，只有两种方式可以退出这个方法。第一种方式是执行引擎遇到任意一个方法返回的字节码指令，这时候可能会有返回值传递给上层的方法调用者，是否有返回值和返回值的类型将根据遇到何种方法返回指令来决定，这种退出方法的方式成为正常完成出口（Normal Method Invocation Completion）。

另外一种退出方式是，在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，无论是Java虚拟机内部产生的异常，还是代码中使用athrow字节码指令产生的异常，只要在本方法的异常表中没有搜索到匹配的异常处理器，就会导致方法退出，这种退出方法的方式称为异常完成出口（Abrupt Method Invocation Completion）。这种退出方式是不会给它的上层调用者产生任何返回值的。

8.3 方法调用

方法调用并不等同于方法执行，方法调用阶段唯一的任务就是确定被调用方法的版本（即调用哪一个方法），暂时还不涉及方法内部的具体运行过程。

8.3.1 解析

所有方法调用中的目标方法在Class文件里面都是一个常量池中的符号引用，在类加载的解析阶段，会将其中的一部分符号引用转化为直接引用，这种解析能成立的前提是：方法在程序真正运行之前就有一个可确定的调用版本，并且方法的调用版本在运行期是不可改变的。换句话说，调用目标在程序代码写好、编译器进行编译时就必须确定下来。这类方法的调用称为解析。

在Java语言中符合“编译期可知，运行期不可变”这个要求的方法，主要包括静态方法和私有方法两大类，前者与类型直接关联，后者在外部不可被访问，这两种方法各自的特点决定了它们都不可能通过继承或别的方式重写其他版本，因此它们都适合在类加载阶段进行解析。

与之相对应的是，在Java虚拟机里面提供了5条方法调用字节码指令：

* **invokestatic**：调用静态方法
* **invokespecial**：调用实例构造器<init>方法、私有方法和父类方法。
* **invokevirtual**：调用所有的虚方法。
* **invokeinterface**：调用接口方法，会在运行时再确定一个实现此接口的对象。
* **invokedynamic**：先在运行时动态解析出调用点限定符所引用的方法，然后再执行该方法，在此之前的4条调用指令，分派逻辑是固化在Java虚拟机内部的，而invokedynamic指令的分派逻辑是由用户所设定的引导方法决定的。

只要能被invokestatic和invokespecial指令调用的方法，都可以在解析阶段中确定唯一的调用版本，符合这个条件的有静态方法、私有方法、实例构造器、父类方法4类，它们在加载的时候就会把符号引用解析为该方法的直接引用。这些方法可以称为非虚方法，与之相反，其他方法称为虚方法（除去final方法）。

8.3.2 分派

**1.静态分派**

代码清单8-6 方法静态分派演示

|  |
| --- |
| **public** **class** StaticDispatch {  **static** **abstract** **class** Human {}  **static** **class** Man **extends** Human {}  **static** **class** Woman **extends** Human {}    **public** **void** sayHello(Human guy) {  System.***out***.println("Hello, guy!");  }    **public** **void** sayHello(Man guy) {  System.***out***.println("Hello, gentleman!");  }    **public** **void** sayHello(Woman guy) {  System.***out***.println("Hello, lady!");  }    **public** **static** **void** main(String[] args) {  Human man = **new** Man();  Human woman = **new** Woman();  StaticDispatch sd = **new** StaticDispatch();  sd.sayHello(man);  sd.sayHello(woman);  }  } |

运行结果：

*Hello, guy!*

*Hello, guy!*

我们把上面代码中的“Human”称为变量的静态类型（Static Type），或者叫做外观类型，后面的“Man”则称为变量的实际类型，静态类型和实际类型在程序中都可以发生一些变化，区别是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，并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。例如下面的代码：

// 实际类型变化

Human man = **new** Man();

Human woman = **new** Woman();

StaticDispatch sd = **new** StaticDispatch();

// 静态类型变化

sd.sayHello((Man)man);

sd.sayHello((Woman)woman);

虚拟机在重载时时通过参数的静态类型而不是实际类型作为判定依据的。并且静态类型是编译器可知的，因此，在编译阶段，Javac编译器会根据参数的静态类型决定使用哪个重载版本。

所有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派。静态分派的典型应用是方法重载。

**2 动态分派**

invokevirtual指令的运行时解析过程大致分为以下几个步骤：

（1）找到操作数栈顶的第一个元素所指向的对象的实际类型，记作C。

（2）如果在类型C中找到与常量中的描述符和简单名称都相符的方法，则进行访问权限校验，如果通过则返回这个方法的直接引用，查找过程结束；如果不通过，则放回java.lang.IllegalAccessError。

（3）否则，按照继承关系从下往上依次对C的各个父类进行第2步的搜索和验证过程。

（4）如果始终没有找到合适的方法，则抛出java.lang.AbstractMethodError异常。

由于invokevirtual指令执行的第一步就是在运行期确定接收者的实际类型，所以两次调用中的invokevirtual指令把常量池中的类方法符号引用解析到了不同的直接引用上，这个过程就是Java语言中方法重写的本质。我们把这种在运行期根据实际类型确定方法执行版本的分派过程称为动态分派。

**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。

**4.虚拟机动态分派的实现**

由于动态分派是非常频繁的动作，而且动态分派的方法版本选择过程需要运行时在类

继承自

Object

的方法

choose(Fruit)

choose(Meat)

Father方法表

继承自

Object

的方法

choose(Fruit)

choose(Meat)

Son方法表

clone()

clone()

wait()

wait()

图8-3 方法表结构

的方法元数据中搜索合适的目标方法，因此在虚拟机的实际实现中基于性能的考虑，大部分实现都不会真正地进行如此频繁的搜索。最常用的“稳定优化”手段就是为类在方法区中建立虚方法表（Virtual Method Table，也称为vtable，与此对应的，在invokeinterface执行时也会用到接口方法表——Interface Method Table，简称itable），使用虚方法表索引来代替元数据查找以提高性能。

虚方法表中存放着各个方法的实际入口地址。如果某个方法在子类中没有被重写，那子类的虚方法表里面的地址入口和父类相同方法的地址入口是一致的，都指向父类的实现入口。如果子类中重写了这个方法，子类方法表中的地址将会是替换为指向子类实现版本的入口地址。

为了实现上的方便，具有相同签名的方法，在父类、子类的虚方法表中都应当具有一样的索引序号，这样当类型转换时，仅需要变更查找的方法表，就可以从不同的虚方法表中按索引转换出所需的入口地址。

方法表一般在类加载的连接阶段进行初始化，准备类的变量初始值后，虚拟机会把该类的方法表也初始化完毕。

8.4 基于栈的字节码解释执行引擎

8.4.1 解释执行

图8-4 编译过程

**程序源码**

**词法分析**

**单词流**

**语法分析**

**解释执行**

**解释器**

**指令流(OP)**

**抽象语法树**

**目标代码**

**生成器**

**中间代码(OP)**

**优化器**

Java语言中，Javac编译器完成了程序代码经过词法分析、语法分析到抽象语法树，再遍历语法树生成线性的字节码指令流的过程。因为这一部分动作是在Java虚拟机之外进行的，而解释器在虚拟机的内部，所以Java程序的编译就是半独立的实现。

8.4.2 基于栈的指令集与基于寄存器的指令集

Java编译器输出的指令流，基本上（部分字节码指令会带有参数）是一种基于栈的指令集架构，指令流中的指令大部分都是零地址指令，它们依赖操作数栈进行工作。

基于栈的指令集主要的优点就是可移植，寄存器由硬件直接提供，程序直接依赖这些硬件寄存器则不可避免地要受到硬件的约束。其次，栈架构的指令集还有一些其他的优点，如代码相对更加紧凑（字节码中每个字节就对应一条指令，而多地址指令集中还需要存放参数）、编译器实现更加简单等。

栈架构指令集的主要缺点是执行速度相对来说稍慢一些。

第12章 Java内存模型与线程

12.3 Java内存模型

12.3.1 主内存和工作内存

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。此处的变量与Java编程中所说的变量有所区别，它包括了实例字段、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，因为后者是线程私有的，不会被共享，自然就不会存在竞争问题。

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（类似于处理器高速缓存），线程的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。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图12-2所示。

Java线程

工作内存

Save

和

Load

操作

Java线程

工作内存

Java线程

工作内存

图12-2 线程、主内存、工作内存三者的交互关系

这里所讲的主内存、工作内存与Java内存区域中的Java堆、栈、方法区等并不是同一个层次的内存划分，这两者基本上是没有关系的，如果两者一定要勉强对应起来，那从变量、主内存、工作内存的定义来看，主内存主要对应于Java堆中的对象实例数据部分，而工作内存则对应于虚拟机栈中的部分区域。从更低层次上说，主内存就直接对应于物理硬件的内存，而为了获取更好的运行速度，虚拟机（甚至是硬件系统本身的优化措施）可能会让工作内存优先存储于寄存器和高速缓存中，因为程序运行时主要访问读写的是工作内存。

12.3.2 内存间交互操作

关于主内存与工作内存之间具体的交互协议，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回主内存之类的实现细节，Java内存模型中定义了以下8中操作来完成，虚拟机实现时必须保证下面提及的每一种操作都是原子的、不可再分的：

* **lock**：作用于主内存的变量，它把一个变量标识为一条线程独占的状态。
* **unlock**：作用于主内存的变量，它把一个处于锁定状态的变量释放出来，释放后的变量才可以被其他线程锁定。
* **read**：作用于主内存的变量，它把一个变量的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。
* **load**：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。
* **use**：作用于工作内存的变量，它把工作内存中一个变量的值传递给执行引擎，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将会执行这个操作。
* **assign**：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。
* **store**：作用于工作内存的变量，它把工作内存中一个变量的值传送到主内存中，以便随后的write操作使用。
* **write**：作用于主内存的变量，它把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。

工作内存

load

read

主内存

use

assign

**执行引擎**

sotre

write

图12-3 变量操作示意图

如果要把一个变量从主内存复制到工作内存，那就要顺序地执行read和load操作，如果要把变量从工作内存同步回主内存，就要顺序地执行store和write操作。注意，Java内存模型只要求上述两个操作必须按顺序执行，而没有保证是连续执行。除此之外，Java内存模型还规定了在执行上述8种基本操作时必须满足如下规则：

* 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。
* 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回主内存。
* 不允许一个线程无原因地（没有发生过任何assign操作）把数据从线程的工作内存同步回主内存。
* 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的变量，换句话说，就是对一个变量实施use、store操作之前，必须先执行过了assign和load操作。
* 一个变量在同一个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。
* 如果对一个变量执行lock操作，那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。
* 如果对一个变量没有被lock操作锁定，那就不允许对它执行unlock操作，也不允许去unlock一个被其他线程锁定住的变量。
* 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（执行store、witter操作）。

12.3.3 对于volatile型变量的特殊规则

当一个变量定义为volatile之后，它将具备两种特性，第一是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。而普通变量做不到这一点，普通变量的值在线程间传递均需要通过主内存来完成。

由于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中，我们仍然要通过加锁（使用synchronized或java.util.concurrent中的原子类）来保证原子性。

* 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。
* 变量不需要与其他的状态变量共同参与不变约束。

而在像如下的代码清单12-3所示的这类场景就很适合使用volatile变量来控制并发，当shutdown()方法被调用时，能保证所有线程中执行的doWork()方法都立即停下来。

代码清单12-3 volatile的使用场景

|  |
| --- |
| **volatile** **boolean** shutdownRequested;    **public** **void** shutdown() {  shutdownRequested = **true**;  }    **private** **void** doWork() {  **while** (!shutdownRequested) {  // do stuff  }  } |

使用volatile变量的第二个语义是**禁止指令重排序优化**，普通的变量仅仅会保证在该方法的执行过程中所有依赖赋值结果的地方都能获取到正确的结果，而不能保证变量赋值操作的顺序与程序代码中的执行顺序一致。因为在一个线程的方法执行过程中无法感知到这点，这也就是Java内存模型中描述的所谓的“线程内表现为串行的语义”。

12.3.5 原子性、可见性与有序性

**原子性**：由Java内存模型来直接保证的原子性变量操作包括read、load、assign、use、store和write，我们大致可以认为基本数据类型的访问读写是具备原子性的。

**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回主内存，在变量读取前从主内存刷新变量值这种依赖主内存作为传递媒介的方式来实现可见性的，无论是普通变量还是volatile变量都是如此，普通变量与volatile变量的区别是，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刷新。因此，可以说volatile保证了多线程操作时变量的可见性，而普通变量则不能保证这一点。

除了volatile之外，synchronized和final也能实现可见性。同步快的可见性是由“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（执行store、write操作）”这条规则获得的，而final关键字的可见性是指：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”的引用传递出去，那在其他线程中就能看见final字段的值。

有序性：如果在本线程内观察，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。

12.3.6 先行发生原则

如果说操作A先行发生于操作B，其实就是说在发生操作B之前，操作A产生的影响能被操作B观察到，“影响”包括修改了内存中共享变量的值、发送了消息、调用了方法等。

* 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。准确地说，应该是控制流顺序而不是程序代码顺序，因为要考虑分支、循环等结构。
* 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”是指时间上的先后顺序。
* 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。
* 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。
* 线程终止操作：线程中的所有操作先行都发生于对此线程的终止检测，我们可以通过Thread.join()方法结束、Thread.isAlive()的返回值等手段检测到线程已经终止执行。
* 线程中断规则：对现场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.interrupted()方法检测到是否有中断发生。
* 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。
* 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。

12.4 Java线程

12.4.1 线程的实现

**1.使用内核线程实现**

1:1

**2.使用用户线程实现**

1：N

**3.使用用户线程加轻量级进程混合实现**

12.4.2 Java线程调度

线程调度是指系统为线程分配处理器使用权的过程，主要调度方式有两种，分别是协同式线程调度（Cooperative Threads-Scheduling）和抢占式线程调度（Preemptive）。

如果使用协同式调度的多线程系统，线程的执行时间由线程本身来控制，线程把自己的工作执行完了之后，要主动通知系统切换到另外一个线程上。协同式多线程的最大好处是实现简单，而且由于线程要把自己的事情干完后才会进行线程切换，切换操作对线程自己是可知的，所以没有什么线程同步的问题。它的坏处也很明显：线程执行时间不可控制，甚至如果一个线程编写有问题，一直不告知系统进行线程切换，那么程序就会一直阻塞在那里。

如果使用抢占式调度的多线程系统，那么每个线程将由系统来分配执行时间，线程的切换不由线程本身来决定（在Java中，Thread.yield()可以让出执行时间，但是要获取执行时间的话，线程本身是没有什么办法的）。在这种实现线程调度的方式下，线程的执行时间是系统可控的，也不会有一个线程导致整个线程阻塞的问题。

12.4.3 状态切换

Java语言定义了5种线程状态：

* 新建(New)：创建后尚未启动的线程处于这种状态。
* 运行（Runnable）：Runnable包括了操作系统线程状态中的Running和Ready，也就是处于此状态的线程有可能正在执行，也有可能正在等待着CPU为它分配执行时间。
* 无限期等待（Waiting）：处于这种状态的线程不会被分配CPU执行时间，它们要等待被其他线程显式地唤醒。以下方法会让线程陷入无限期的等待状态：
* 没有设置Timeout参数的Object.wait()方法。
* 没有设置Timeout参数的Thread.join()方法。
* LockSupport.park()方法。
* 限期等待（Timed Waiting）：处于这种状态的线程也不会被分配CPU执行时间，不过无需等待被其他线程显式地唤醒，在一定时间之后它们会由系统自动唤醒。以下方法会让线程进入限期等待状态：
* Thread.sleep()方法。
* 设置了Timeout参数的Object.wait()方法。
* 设置了Timeout参数的Thread.join()方法。
* LockSupport.parkNanos()方法
* LockSupport.parkUntil()方法
* 阻塞（Blocked）：线程被阻塞了，“阻塞状态”与“等待状态”的区别是：“阻塞状态”在等待着获取到一个排他锁，这个事件将在另外一个线程放弃这个锁的时候发生；而“等待状态”则是在等待一段时间，或者唤醒动作的发生。在程序等待进入同步区域的时候，线程将进入这种状态。
* 结束（Terminated）：已终止线程的线程状态，线程已经结束执行。

上述5种状态在遇到特定事件发生的时候将会互相转换，它们的转换关系如图12-6所示。

图12-6 线程状态转换关系

run()结束

start()

synchoronized

sleep()

notify()/notifyAll()

wait()

**Running**

**New**

**Waiting**

**Terminated**

**Timed Waiting**

**Blocked**

13 线程安全与锁优化

13.2 线程安全

13.2.1 Java语言中的线程安全

线程安全安装“安全程度”由强至弱来排序，我们可以将Java语言中各种操作共享的数据分为以下5类：不可变、绝对线程安全、相对线程安全、线程兼容和先对立。

**1.不可变**

Java语言中，如果共享数据是一个基本数据内省，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。如果共享数据是一个对象，那就需要保证对象的行为不会对其状态产生任何影响才行。

**2.绝对线程安全**

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。

**3.相对线程安全**

相对现场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地连续调用，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语言中，大部分的线程安全类都属于这种类型，例如Vector、HashTable、Collections的synchronizedCollection()方法包装的集合等。

**4.线程兼容**

线程兼容是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。

**5.线程对立**

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。

13.2.2 线程安全的实现方法

**1.互斥同步（Mutual Exclusion & Synchronization）**

同步是指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。因此在这4个字面，互斥是因，同步是果；互斥是方法，同步是目的。

monitorenter、monitorexit