

目录

目录	1
自动内存管理机制.....	3
Java 内存区域和内存溢出异常.....	3
运行时数据区域.....	3
程序计数器.....	3
Java 虚拟机栈.....	3
本地方法栈.....	4
Java 堆.....	4
方法区.....	4
运行时常量池.....	4
直接内存.....	5
对象访问.....	5
OOM 实战.....	6
堆溢出.....	6
虚拟机栈和本地方法栈溢出.....	6
运行时常量池溢出.....	6
方法区溢出.....	6
本机直接内存溢出.....	6
垃圾收集器和内存分配策略.....	7
概述.....	7
对象已死?	7
引用计数算法.....	7
根搜索算法.....	7
再谈引用.....	7
生存还是死亡.....	8
回收方法区.....	9
垃圾收集算法.....	9
标记-清除算法.....	9
复制算法.....	9
标记-整理算法	9
分代收集算法.....	10
垃圾收集器.....	10
Serial 收集器.....	10
ParNew 收集器	10
Parallel Scavenge 收集器.....	11
Serial Old 收集器	11
Parallel Old 收集器	11
CMS 收集器	11

G1 收集器	12
内存分配和回收策略.....	12
对象优先在 Eden 区域分配.....	12
大对象直接进入老年代.....	12
长期存活的对象进入老年代.....	13
动态对象年龄判定.....	13
空间分配担保.....	13
虚拟机性能监控和故障处理工具.....	13
JDK 的命令行工具.....	13
Jps: 虚拟机进程状况工具.....	13
Jstat: 虚拟机统计信息监视工具.....	14
Jinfo: Java 配置信息工具.....	15
Jmap: Java 内存映像工具.....	15
Jhat: 虚拟机堆转储快照分析工具	15
Jstack: 虚拟机堆栈分析工具	15
JDK 的可视化工具	16
虚拟机执行子系统.....	16
类文件结构.....	16
Class 类文件的结构.....	16
魔数和 class 文件的版本	17
常量池.....	18
访问标志.....	20
类索引、父类索引和接口索引	20
字段表集合.....	21
方法表集合.....	22
属性表集合.....	23
虚拟机类加载机制.....	29
类加载的时机.....	29
类加载的过程.....	30
加载.....	30
验证.....	30
准备.....	32
解析.....	32
初始化.....	36
类加载器.....	37
类和类加载器.....	37
双亲委派模式.....	37
虚拟机字节码执行引擎.....	38
运行时帧栈结构.....	38
局部变量表.....	39
操作数栈.....	39
动态连接.....	39
方法返回地址.....	39
方法调用.....	40

高效并发.....	40
Java 内存模型.....	40
主内存和工作内存.....	41
内存间交互操作.....	41
对于 <code>volatile</code> 变量的特殊规则.....	43
对于 <code>long</code> 和 <code>double</code> 型变量的特殊规则.....	44
原子性、可见性和有序性.....	44
先行发生原则.....	46

自动内存管理机制

Java 内存区域和内存溢出异常

运行时数据区域

JVM 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。这些区域有着各自的用途，以及创建和销毁的时间。

程序计数器

是一块较小的内存空间，它的作用可以看做是当前线程所执行的字节码的行号指示器。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。

由于 JVM 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实现的，在任何一个确定的时刻，一个处理器只会处理一条线程中的指令，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自己的程序计数器，各条线程之间的计数器互不影响，我们称这类内存区域为线程私有的内存区域。

如果线程正在执行的是一个 Java 方法，这个计数器记录的是正在执行的虚拟机字节码指令的地址，如果正在执行的是 Java Native 方法，这个计数器值则为空。这个内存区域是唯一一个在 JVM 规范中没有定义 OutOfMemoryError 情况的区域。

Java 虚拟机栈

Java 虚拟机栈同样是线程私有的，生命周期和线程相同，虚拟机栈描述的是 Java 方法执行的内存模型，每个方法被执行的同时都会创建一个帧栈，用于存储局部变量表、操作栈、动态链接、方法出口等信息。每一个方法被调用直至执行完成的过程，就对应着一个帧栈在虚拟机栈中从入栈到出栈的过程。

经常有人把 Java 内存分为堆内存和栈内存，这种分配方式比较粗糙，这里的栈内存指的就是虚拟机栈中的局部变量表部分。

局部变量表存放了编译期间可知的各种基本数据类型和对象引用和 `returnAddress` 类型。

其中 64 位长的 `long` 和 `double` 占用两个局部变量空间（slot），其余的数据类型占一个，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确定的，在方法运行期间不会改变局部变量表的大小。

JVM 规范对这个区域规定了两种异常：如果线程请求的栈深度大于虚拟机允许的栈深度，将抛出 `StackOverflowError` 异常；如果虚拟机允许动态扩展，当扩展无法申请到足够的内存时会抛出 `OutOfMemoryError` 异常。

本地方法栈

本地方法栈和虚拟机栈所发挥的作用是非常相似的，区别不过是虚拟机栈为虚拟机执行 Java 方法服务，而本地方法栈是对虚拟机使用到的 native 方法服务。JVM 规范对本地方法栈的实现没有做强制要求，本地方法栈也可以抛出虚拟机栈抛出的两种异常。

Java 堆

一般情况下，Java 堆是 Java 虚拟机所管理的内存中最大的一块，它是所有线程共享的，虚拟机启动时创建，此内存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。

Java 堆是垃圾收集器管理的主要区域，因此很多时候也被称为 GC 堆，现代的收集器基本都是采用的分代收集算法。所以 Java 堆还可以细分为：新生代和老生代，再细致一点有 Eden 区域、From Survivor 区域、To Survivor 空间等。

根据 JVM 的规范规定，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。如果堆中没有内存完成实例分配，就会抛出 `OutOfMemoryError` 异常。

方法区

也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数据。

JVM 规范对这个区域的限制比较宽松，除了和 Java 堆一样不需要连续的内存和可以选择固定大小或者可扩展外，还可以选择不实现垃圾回收。垃圾收集行为在这个区域是比较少出现的，这个区域的内存回收目标主要是针对常量池的回收和对类型的卸载。

当方法区无法完成内存分配时，就会抛出 `OutOfMemoryError` 异常。

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存放到方法区的运行时常量池中。

运行时常量池相对于 Class 文件常量池的另外一个重要特征是具备动态性。Java 语言并不要求常量一定只能在编译期产生，运行期间也可能将新的常量放入池中。这种特性被用的比较多的就是 String.intern() 方法。

常量池无法申请到内存时，也会抛出 OOM 异常。

直接内存

直接内存并不是虚拟机运行时数据区的一部分，也不是 JVM 规范中定义的内存区域，但是这部分区域也被频繁的使用，而且也会导致 OOM 出现。

和 NIO 有关，是使用 native 方法直接分配的一块堆外内存。

对象访问

对象访问在 Java 语言中无处不在，是最普通的程序行为，即便如此，也会涉及 Java 栈、Java 堆、方法区这三个最重要区域之间的关联关系。

例如 Object obj = new Object();

假设这句代码出现在方法体中，那么 “Object obj” 这部分的语义将会反映到 Java 栈的本地变量表中，作为一个 reference 类型的数据出现，而 “new Object()” 这部分语义将会放映到 Java 堆中，形成一块存储了 Object 类型所有实例数据值的结构化内存，根据具体类型以及虚拟机实现的对象内存布局的不同，这块内存的长度是不固定的，另外，在 Java 堆中还必须包含能查找到此对象类型数据（如对象类型、实现接口、父类、方法等）的地址信息，这些类型数据则存储在方法区中。

主流的对象访问方式有两种：

1. 句柄访问方式：Java 堆中将划分出一块区域作为句柄池，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和类型数据各自的具体地址信息。
2. 直接指针访问方式：reference 直接存储的就是对象的地址。

两种方式的优缺：使用句柄方式最大的好处就是 reference 存储的就是稳定的句柄地址，在对象被移动时只会改变句柄中的实例数据指针，而 reference 本身不需要被修改。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，这类积少成多最后也是一项可观的执行成本。

OOM 实战

堆溢出

要解决这个区域的异常，首先要分清楚是出现了内存泄露还是内存溢出。

如果是内存泄露，通过工具查看泄露对象到 **GC Roots** 的引用链，于是就可以找到泄露对象是通过怎样路径和 **GC Roots** 相关联并导致垃圾回收器无法回收它们，掌握了泄露对象的类型信息，以及 **GC Roots** 引用链的信息，就可以比较准确的定位出泄露代码的位置。

如果不存在泄露，那就应当检查虚拟机的堆参数，与机器物理内存比是否可以调大，从代码上检查是否存在某些对象生命周期过长，持有状态时间过长的情况，尝试减少程序运行时的内存消耗。

虚拟机栈和本地方法栈溢出

实验表明，在单个线程下，无论是由于帧栈太大，还是虚拟机栈容量太小，当内存无法分配时，都是抛出 **StackOverflowError** 异常。

如果测试时不局限于单线程，通过不断建立线程的方式倒是可以产生内存溢出代码，但是，这样产生的内存溢出异常和栈空间是否足够大没有任何联系，这种情况下，给每个线程的栈分配的内存越大，反而越容易产生内存溢出异常。

原因如下：操作系统分配给每个进程的内存是有限的，虚拟机提供了参数来控制 Java 堆和方法区这两部分内存的最大值，剩余的内存为操作系统给每个进程分配的内存减去最大堆内存减去方法区内存，程序计数器的内存可以忽略掉，如果虚拟机进程本身耗费的内存不算在内，剩下的内存就由虚拟机栈和本地方法栈瓜分了，每个线程分配到的栈容量越大，可以建立的线程数量自然就越少，建立线程时就越容易把剩下的内存耗光。

开发多线程应用时应该注意的一点就是：如果是建立多线程导致的内存溢出，在不能减少线程数或者更换 64 位虚拟机的情况下，就只能通过减少最大堆和减少栈容量来换取更多的线程。

运行时常量池溢出

方法区溢出

方法区溢出是一种常见的内存溢出，一个类如果要被垃圾收集器回收掉，判定条件是十分苛刻的。

本机直接内存溢出

垃圾收集器和内存分配策略

概述

我们为什么要了解 GC 和内存分配？

当需要排查各种内存溢出、内存泄露问题时，当垃圾收集成为系统达到更高并发量的瓶颈时，我们就需要对这些自动化的技术实施必要的监控和调节。

我们所关注的回收内存是哪些？

主要是 Java 堆和方法区中的内存，一个接口的多个实现类需要的内存可能不一样，一个方法的多个分支需要的内存可能也不一样，我们只有在程序处于运行期间才能知道它会创建哪些对象，这部分内存的分配和回收都是动态的，垃圾回收器关注的内存是这一块内存。

对象已死？

如何确认一个对象的存活或者死亡？

引用计数算法

给对象添加一个引用计数器，每当有一个地方引用它时，计数器值就加 1，引用失效时，计数器值减 1，任何时刻计数器为 0 的对象就是不可能被引用的。

Java 语言中并没有选择这种计数方式，主要原因是它很难解决对象之间的相互循环引用问题。

根搜索算法

通过一系列的名为 GC Roots 的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径叫做引用链，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。

Java 语言中，可以作为 GC Roots 的对象包括以下几种：

1. 虚拟机栈（帧栈中的本地变量表）中引用的对象
2. 方法区中类静态属性引用的对象
3. 方法区中的常量引用的对象
4. 本地方法栈 JNI 的引用的对象。

再谈引用

Jdk1.2 之前，Java 中的引用定义的传统，如果 reference 类型数据中存储的数值代表的是另外一块内存的起始地址，就称这块内存代表着一个引用。

Jdk1.2 之后，Java 对引用的概念进行了扩展，将引用分为强引用、软引用、弱引用、虚引用。

强引用就是指在程序代码中普遍存在的，类似“Object obj = new Object()”这类的引用，只要强引用还存在，垃圾收集器就永远也不会回收掉被引用的对象。

软引用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内并进行第二次回收，如果回收后还是没有足够的内存，才会抛出内存溢出异常。

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还弱一点，被弱引用关联的对象只能生存到下一次垃圾收集之前，当垃圾收集器发生工作时，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。

虚引用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产生影响，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希望能在这个对象被收集器回收时收到一个系统通知。

生存还是死亡

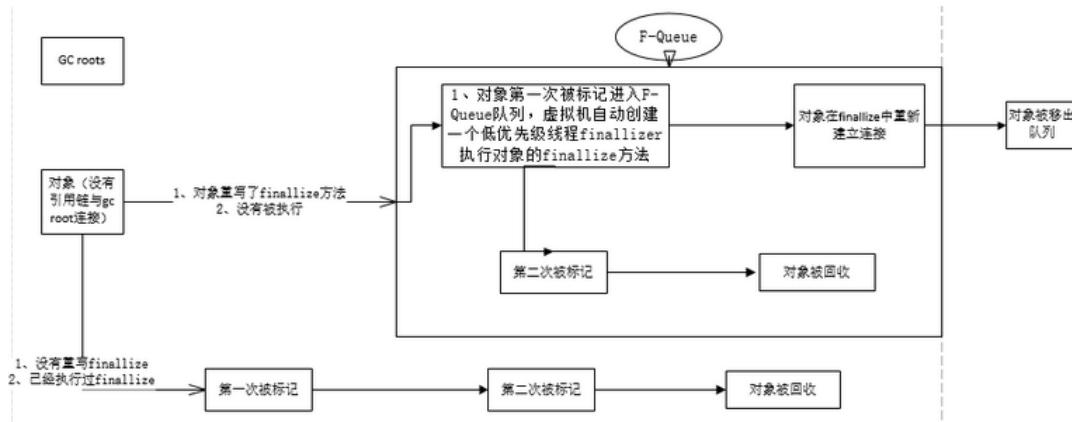
要宣告一个对象死亡，至少要经历两次标记过程，如果对象在根搜索后没有发现和 GC Roots 相连的引用链，那它将会被第一次标记并且进行筛选，筛选的条件是此对象是否有必要执行 `finalize()` 方法，当对象没有覆盖这个方法或者虚拟机已经执行过这个方法，虚拟机将这两种情况都视为没有必要执行，这时候对象会被进行第二次标记，紧接着被回收。

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放在一个名为 F-Queue 的队列中，并在稍后由一条虚拟机自动建立的、低优先级的 Finalize 线程去执行，这里所谓的执行是虚拟机会触发这个方法，但是并不承诺它会运行结束。这样做的原因是：如果一个对象在 `finalize()` 中执行缓慢或者发生了死循环，将很可能会导致 F-Queue 中的其他对象处在永久等待状态。Finalize() 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标记，对象如果要在 `finalize()` 中拯救自己，只要重新与引用链上的任何一个对象建立关联即可，那么在第二次标记后它将被移出“即将回收”的集合。

注意的是任何一个对象的 `finalize()` 方法都只会被系统调用一次。

`Finalize()` 方法能完成的功能，使用 `try-finally` 或者其他方式可以做的更好、更及时。

判定过程如下图：



回收方法区

方法区的垃圾回收效率是比较低的, JVM 规范也没有强制要求虚拟机在方法区实现垃圾回收。方法区的垃圾回收主要回收两部分内容: 废弃常量和无用的类。

回收废弃常量和回收 Java 堆中的对象非常类似。

判定一个类是否是无用的类的条件要苛刻很多。要满足如下三个条件:

1. 该类的所有实例都已经被回收, 也就是 Java 堆中不存在该类的任何实例
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

满足了上述条件, 仅仅是可以被回收, 是否回收可以通过 JVM 参数来控制。

在大量使用反射、动态代理、CGLIB 等框架的场景中, 以及动态生成 JSP 和 OSGI 这类频繁自定义 ClassLoader 的场景都需要虚拟机具备类卸载的功能, 以保证方法区不会溢出。

垃圾收集算法

标记-清除算法

算法分为标记和清除两个阶段, 首先标记出所有需要回收的对象, 在标记完成后统一回收掉所有被标记的对象。缺点有两个, 一个是效率问题, 标记和清除的效率都不高, 另外一个是空间问题, 标记结束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, 当程序在以后的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触发另一次垃圾收集动作。

复制算法

将可用的内存按照容量分成大小相等的两块, 每次只使用其中的一块, 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。缺点就是将内存缩小为原来的一半, 代价比较大。

现代的商业虚拟机都使用这种方式来收集新生代, 因为 98% 新生代的特点是朝生夕死, 所以不需要按照 1:1 的比例来划分内存, 而是将内存划分为一块较大的 Eden 区域和两块较小的 Survivor 区域, 当回收时, 将 Eden 和 Survivor 中还存活着的对象一次性拷贝到另外一块 Survivor 空间上, 最后清理掉 Eden 和刚才用过的 Survivor 空间。HotSpot 虚拟机中 Eden 和 Survivor 的默认比例大小是 8:1, 也就是每次新生代可用内存空间是新生代容量的 90%。98% 对象可回收只是一般场景下的数据, 我们无法保证每次回收都只有不多于 10% 的对象存活, 当 Survivor 空间不够时, 我们需要依赖其他内存(这里指老年代)进行分配担保。

标记-整理算法

复制收集算法在对象存活率较高时就需要执行较多的复制操作, 效率会变低, 更关键的

是，如果不想要浪费 50% 的空间，就需要有额外的空间进行分配担保，以应对被使用的内存 100% 对象都存活的极端情况。所以老年代一般不使用这种方法。

根据老年代的特点，有人提出了标记-整理算法，标记过程和标记-清除算法一样，但是后续步骤不是直接对可回收对象进行整理，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。

分代收集算法

根据对象的存活周期的不同将内存划分为几块，一般是把 Java 堆分成新生代和老生代，新生代使用复制算法，老生代使用标记-清除或者标记-整理算法。

垃圾收集器

如果说收集算法是内存回收的方法论，垃圾收集器就是内存回收的具体实现。

HotSpot 虚拟机中大致有如下 7 种作用于不同分代的垃圾收集器，注意的一个问题是没有任何所谓的最好的垃圾收集器，只有最合适的选择。

Serial 收集器

是一个单线程的垃圾收集器，在它进行垃圾回收时，必须暂停其他所有的工作线程，直到它收集结束。最大的缺点就是用户体验太差。

它目前仍然是虚拟机运行在 Client 模式下的默认新生代收集器，它有着优于其他垃圾收集器的地方，简单而高效，对于限定单个 CPU 的环境来说，Serial 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，在用户桌面应用场景中，停顿时间完全可以控制在 100ms 之内，这完全是可以接受的，所以 Serial 收集器对于运行在 Client 模式下的虚拟机来说是一个很好的选择。

ParNew 收集器

ParNew 收集器其实就是 Serial 收集器的多线程版本，除了使用多条线程进行垃圾回收之外，其余行为和 Serial 完全一样。

是运行在 Server 模式下的虚拟机中首选的新生代收集器，其中有一个与性能无关的原因就是：除了 Serial 收集器之外，只有它能和 CMS 收集器配合工作。

不幸的是，作为老生代的收集器，却无法和 jdk1.4 中已经存在的新生代收集器 Parallel Scavenge 配合工作，所以在 jdk1.5 中使用 CMS 来收集老生代的时候，新生代只能选择 ParNew 或者 Serial。

在单 CPU 的环境下，ParNew 不会有比 Serial 更好的效果。

Parallel Scavenge 收集器

也是一个新生代收集器，使用复制算法，并行的多线程收集器，它和 ParNew 有什么区别呢？

答案是关注点不同。CMS 等收集器的关注点是尽可能缩短垃圾收集时用户线程的停顿时间。而 Parallel Scavenge 的目的是达到一个可控制的吞吐量。吞吐量=运行用户代码时间/(运行用户代码时间+垃圾收集时间)。

停顿时间越短越适合和用户交互的程序，良好的响应速度能提升用户的体验，而高吞吐量能高效率的利用 CPU 时间，尽快的完成程序运算任务，主要适合在后台运算而不需要太多交互的任务。

这个收集器还有一个很大的特点就是拥有自适应调节策略，只要设置相应的 JVM 参数即可。这个开关打开后，就不需要指定新生代的大小、Eden 区域和 Survivor 区域的比例，晋升老年代对象年龄等细节参数了，虚拟机会根据当前系统的运行情况收集性能监控信息，动态调整这些参数以提供最合适的停顿时间或者最大的吞吐量。

Serial Old 收集器

是 Serial 收集器的老年代版本，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被 Client 模式下的虚拟机使用。如果在 Server 模式下，主要有两大用途：一个是在 jdk1.5 之前的版本中与 Parallel Scavenge 收集器搭配使用，另外一个就是作为 CMS 收集器的后备方案。

Parallel Old 收集器

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

注重吞吐量的场合中，可以优先考虑 Parallel Scavenge 收集器+Parallel Old 收集器。

CMS 收集器

CMS 收集器是一种以获取最短回收停顿时间为为目标的收集器。CMS 是基于标记-清除算法来实现的。

CMS 收集器的运作过程分为四个步骤：

1. 初始标记
2. 并发标记
3. 重新标记
4. 并发清除

初始标记和重新标记阶段仍然需要“stop the world”。初始标记仅仅是标记一下 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，而重新标记阶段就是为了修正并发标记阶段，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。

由于整个过程中最耗时的并发标记和并发清除过程是和用户线程一起运行的，所以总体来说，**CMS** 收集器的内存回收过程是和用户线程一起并发执行的。

CMS 收集器的三个缺点：

1. 对 CPU 资源十分敏感。
2. **CMS** 无法处理浮动垃圾
3. 收集结束后会产生大量的空间碎片

G1 收集器

是当前收集器技术发展的最前沿成果。它和 **CMS** 收集器相比有两个明显的改善：一是 **G1** 收集器基于标记-整理算法实现，也就是说它不会产生空间碎片，一是它可以准确的控制停顿，既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 M 毫秒，这几乎已经是实时 Java 的垃圾收集器的特征了。

G1 收集器可以在不牺牲吞吐量的前提下完成低停顿的内存回收，这是因为它能够极力避免全区域的垃圾回收，之前的垃圾收集器的回收范围都是整个新生代或者老生代。而 **G1** 将整个 Java 堆划分为多个大小固定的独立区域，并且追踪这些区域里面的垃圾堆积程度，在后台维护一个优先列表，每次根据允许的收集时间，优先回收垃圾最多的区域。

内存分配和回收策略

Java 技术体系中的自动内存管理最终解决了两个问题：给对象分配内存和回收掉分配给对象的内存。

对象优先在 Eden 区域分配

大多数情况下，对象在新生代的 **Eden** 区域分配，当 **Eden** 区域没有足够的空间进行分配时，虚拟机将发起一次 **Minor GC**，会尝试将 **Eden** 区域和一块 **Survivor** 区域仍然存活的对象移动到空闲 **Survivor** 区域中，如果空闲 **Survivor** 区域空间不够，会通过分配担保将仍然存活的对象移动到老生代。

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死的特征，所以 **Minor GC** 非常频繁，一般回收速度也比较快。

Major/Full GC：指发生在老生代的 GC，**Major GC** 的速度一般比 **Minor GC** 慢十倍以上。

大对象直接进入老年代

所谓大对象，就是需要大量连续内存空间的 Java 对象。虚拟机提供了一个-XX:PretenureSizeThreshold 参数，令大于这个设置值的对象直接在老年代中分配，这样做的

目的是避免在 **Eden** 区域及两个 **Survivor** 区域之间发生大量的内存拷贝。

这个参数只对 **Serial** 和 **ParNew** 两款收集器有效。

长期存活的对象进入老年代

虚拟机给每个对象设置了一个年龄计数器，如果对象在 **Eden** 区域出生并且经过第一次 **Minor GC** 之后仍然存活，并且能被 **Survivor** 区域容纳的话，将被移动到 **Survivor** 区域中，并将年龄设置为 1，对象在 **Survivor** 中每熬过一次 **Minor GC**，年龄就增加一岁，当它的年龄增加到一定程度，就会被晋升到老年代中。

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，年龄大于或者等于该年龄的对象就可以直接进入老生代。

空间分配担保

在发生 **Minor GC** 时，虚拟机会检测之前每次晋升到老年代的平均大小是否大于老年代的剩余空间大小，如果大于，则会直接进行一次 **Full GC**，如果小于，则查看参数是否担保失败，允许的话，只进行 **Minor GC**，否则也要改为 **Full GC**。

取平均值是一种动态概率的手段，仍然存在失败的风险，如果失败了，则就再进行一次 **Full GC**，虽然失败时绕的圈子很大，但是一般情况下还是打开允许担保失败的设置，防止 **Full GC** 太过频繁。

虚拟机性能监控和故障处理工具

JDK 的命令行工具

每次 **JDK** 版本更新，**bin** 目录下的命令行工具的数量和功能都会不知不觉的增加和增强。其中有一部分工具就是监视虚拟机和故障处理的工具，这些工具是 **sun** 公司作为礼物附赠给 **jdk** 使用者的。这些工具异常的稳定并且功能强大。这些工具的体积也都异常的小，因为它们大多是 **jdk/lib/tools.jar** 类库的一层包装，它们都是使用 **Java** 编写的。

Jps: 虚拟机进程状况工具

可以列出正在运行的虚拟机进程，并显示虚拟机执行主类的名称，以及这些进程的本地

虚拟机的唯一 ID (LVMID)。对于本地虚拟机进程来说，LVMID 和操作系统的进程 ID 是一致的。

Jps 还可以通过 RMI 协议查询开启了 RMI 服务的远程虚拟机进程状态。

Jps 命令的常用选项如下：

表 4-2 jps 工具主要选项

选项	作用
-q	只输出 LVMID，省略主类的名称
-m	输出虚拟机进程启动时传递给主类 main() 函数的参数
-l	输出主类的全名，如果进程执行的是 Jar 包，输出 Jar 路径
-v	输出虚拟机进程启动时 JVM 参数

Jstat：虚拟机统计信息监视工具

用于监视虚拟机各种运行状态信息的命令，它可以显示本地或远程虚拟机进程中的类装载、内存、垃圾收集、JIT 编译等运行数据，在没有 GUI 只有纯文本控制台环境的服务器上，它将是运行期定位虚拟机性能问题的首选工具。

Jstat 命令格式：

```
jstat [ option vmid [interval[s|ms] [count]] ]
```

如果是本地虚拟机进程，那么 lvmid 和 vmid 是一致的，如果是远程服务器，那么 vmid 的格式应当是：

```
[protocol://]lvmid[@hostname[:port]/servername]
```

参数 interval 和 count 代表查询间隔和次数。

选项 option 代表用户希望查询的虚拟机信息，主要分为三类：类装载、垃圾收集、即运行期编译状况。

常用选项如下：

表 4-3 jstat 工具主要选项

选 项	作 用
-class	监视类装载、卸载数量、总空间及类装载所耗费的时间
-gc	监视 Java 堆状况，包括 Eden 区、2 个 survivor 区、老年代、永久代等的容量、已用空间、GC 时间合计等信息
-gccapacity	监视内容与 -gc 基本相同，但输出主要关注 Java 堆各个区域使用到的最大和最小空间
-gcutil	监视内容与 -gc 基本相同，但输出主要关注已使用空间占总空间的百分比
-gccause	与 -gcutil 功能一样，但是会额外输出导致上一次 GC 产生的原因
-gcnew	监视新生代 GC 的状况
-genewcapacity	监视内容与 - gcnew 基本相同，输出主要关注使用到的最大和最小空间
-gcold	监视老年代 GC 的状况
-gcoldcapacity	监视内容与 - gcold 基本相同，输出主要关注使用到的最大和最小空间
-gcpermcapacity	输出永久代使用到的最大和最小空间
-compiler	输出 JIT 编译器编译过的方法、耗时等信息
-printcompilation	输出已经被 JIT 编译的方法

Jinfo: Java 配置信息工具

作用是实时的查看和调整虚拟机的各项参数。

Jps -v 可以查看虚拟机启动时显示指定的参数列表，如果想知道未被显示指定的参数的系统默认值，除了查找资料外，就只能使用 jinfo 的 -flag 选项进行查询。Jinfo 也可以使用 -sysprops 选项把 System.getProperties() 打印出来。

命令格式：

```
jinfo [ option ] pid
```

Jmap: Java 内存映像工具

用于生成堆转储快照。Jmap 的作用不仅仅是获得 dump 文件，它还可以查询 finalize 执行队列，Java 堆和永久代的详细信息，如空间使用率，当前使用的是哪一种垃圾收集器等。

jmap 命令格式：

```
jmap [ option ] vmid
```

option 选项的合法值与具体含义如表 4-4 所示。

表 4-4 jmap 工具主要选项

选 项	作 用
-dump	生成 Java 堆转储快照。格式为：-dump:[live,]format=b,file=<filename>，其中 live 子参数说明是否只 dump 出存活的对象
-finalizerinfo	显示在 F-Queue 中等待 Finalizer 线程执行 finalize 方法的对象。只在 Linux / Solaris 平台下有效
-heap	显示 Java 堆详细信息，如使用哪种回收器、参数配置、分代状况等。只在 Linux / Solaris 平台下有效
-histo	显示堆中对象统计信息，包括类、实例数量和合计容量
-permstat	以 ClassLoader 为统计口径显示永久代内存状态。只在 Linux / Solaris 平台下有效
-F	当虚拟机进程对 -dump 选项没有响应时，可使用这个选项强制生成 dump 快照。只在 Linux / Solaris 平台下有效

Jhat: 虚拟机堆转储快照分析工具

分析 jmap 生成的堆转储快照，不推荐使用这种方式。

Jstack: 虚拟机堆栈分析工具

用于生成虚拟机当前时刻的线程快照，线程快照就是虚拟机内每一条线程正在执行的方法堆栈的集合，生成线程快照的目的就是定位线程出现长时间停顿的原因

jstack 命令格式：

```
jstack [ option ] vmid
```

option 选项的合法值与具体含义如表 4-5 所示。

表 4-5 jstack 工具的主要选项

选项	作用
-F	当正常输出的请求不被响应时，强制输出线程堆栈
-l	除堆栈外，显示关于锁的附加信息
-m	如果调用到本地方法的话，可以显示 C/C++ 的堆栈

JDK 的可视化工具

熟悉 JConsole 和 VisualVM 的使用。

虚拟机执行子系统

类文件结构

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的排列在 Class 文件中，中间没有添加任何分隔符，这使得 Class 文件中存储的基本都是程序运行所需要的必要数据，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。

Class 文件的结构中只有两种数据类型：无符号数和表。

无符号数属于基本的数据类型，以 u1、u2、u4、u8 分别来代表一个字节、两个字节、四个字节、八个字节的无符号数，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。

表是由多个无符号数或者其他表作为数据项构成的复合数据类型，所有表都习惯以“_info”结尾，整个 class 文件本质上就是一张表，如下图：

表 6-1 Class 文件格式

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_count	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是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，这时候称这一系列连续的某一类型的数据为某一类型的集合。下面我们来看一下各个数据项的含义。

魔数和 class 文件的版本

每个 class 文件的头四个字节称为魔数，唯一作用是确定这个文件是否是一个能够被虚拟机接受的 class 文件。

紧接着魔数的四个字节是 class 文件的版本号，5、6 字节代表的是次版本号，7、8 字节代表的是主版本号，Java 的版本号从 45 开始，jdk1.1 之后的每个 jdk 大版本发布主版本号向上加 1，高版本的 jdk 能向下兼容低版本的 class 文件，但是不能运行以后版本的 class 文件。

主流的 jdk 版本编译器输出的默认可支持的 class 文件版本号：

表 6-2 Class 文件版本号

编译器版本	-target 参数	十六进制版本号	十进制版本号
JDK 1.1.8	不能带 target 参数	00 03 00 2D	45.3
JDK 1.2.2	不带（默认为 -target 1.1）	00 03 00 2D	45.3
JDK 1.2.2	-target 1.2	00 00 00 2E	46.0
JDK 1.3.1_19	不带（默认为 -target 1.1）	00 03 00 2D	45.3
JDK 1.3.1_19	-target 1.3	00 00 00 2F	47.0
JDK 1.4.2_10	不带（默认为 -target 1.2）	00 00 00 2E	46.0
JDK 1.4.2_10	-target 1.4	00 00 00 30	48.0
JDK 1.5.0_11	不带（默认为 -target 1.5）	00 00 00 31	49.0
JDK 1.5.0_11	-target 1.4 -source 1.4	00 00 00 30	48.0
JDK 1.6.0_01	不带（默认为 -target 1.6）	00 00 00 32	50.0
JDK 1.6.0_01	-target 1.5	00 00 00 31	49.0
JDK 1.6.0_01	-target 1.4 -source 1.4	00 00 00 30	48.0
JDK 1.7.0	不带（默认为 -target 1.7）	00 00 00 33	51.0
JDK 1.7.0	-target 1.6	00 00 00 32	50.0
JDK 1.7.0	-target 1.4 -source 1.4	00 00 00 30	48.0

常量池

紧接着主次版本号之后的是常量池入口， class 文件中出现的第一个表结构。

由于常量池中常量的数量不确定的，所以在常量池的入口需要放置一项 u2 类型的数据，代表常量池容量计数值。这个容量计数是从 1 开始的，例如常量池容量为 0x0016，即十进制的 22，那么常量池中有 21 项常量，索引从 1-21。之所以把 0 空出来是有原因的，这样做是为了满足后面某些指向常量池的索引值的数据在某些特定的情况下需要表达“不需要引用任何一个常量池项目”的意思。Class 文件结构中只有常量池的容量计数从 1 开始，其他的集合类型都是从 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 Java 语言层面的常量概念，如文本字符串、被声明为 final 的常量值等。

符号引用包括了下面三类常量：

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

常量池的每一项常量都是一个表，共有 11 种结构各不相同的表结构数据。这些表的第一位都是一个 u1 类型的标志位，代表这个常量属于哪种常量类型。11 种常量类型所代表的含义如下：

表 6-3 常量池的项目类型

类 型	标 志	描 述
CONSTANT_Utf8_info	1	UTF-8 编码的字符串
CONSTANT_Integer_info	3	整型字面量
CONSTANT_Float_info	4	浮点型字面量
CONSTANT_Long_info	5	长整型字面量
CONSTANT_Double_info	6	双精度浮点型字面量
CONSTANT_Class_info	7	类或接口的符号引用
CONSTANT_String_info	8	字符串类型字面量
CONSTANT_Fieldref_info	9	字段的符号引用
CONSTANT_Methodref_info	10	类中方法的符号引用
CONSTANT_InterfaceMethodref_info	11	接口中方法的符号引用
CONSTANT_NameAndType_info	12	字段或方法的部分符号引用

常量池中的 11 种数据类型的结构总表

常量	项 目	类 型	描 述
CONSTANT_Utf8_info	tag	u1	值为 1
	length	u2	UTF-8 编码的字符串占用了字节数
	bytes	u1	长度为 length 的 UTF-8 编码的字符串
CONSTANT_Integer_info	tag	u1	值为 3
	bytes	u4	按照高位在前存储的 int 值
CONSTANT_Float_info	tag	u1	值为 4
	bytes	u4	按照高位在前存储的 float 值
CONSTANT_Long_info	tag	u1	值为 5
	bytes	u8	按照高位在前存储的 long 值
CONSTANT_Double_info	tag	u1	值为 6
	bytes	u8	按照高位在前存储的 double 值
CONSTANT_Class_info	tag	u1	值为 7
	index	u2	指向全限定名常量项的索引
CONSTANT_String_info	tag	u1	值为 8
	index	u2	指向字符串字面量的索引
CONSTANT_Fieldref_info	tag	u1	值为 9
	index	u2	指向声明字段的类或接口描述符 CONSTANT_Class_info 的索引项
	index	u2	指向字段描述符 CONSTANT_NameAndType 的索引项

常量	项目	类型	描述
CONSTANT_Methodref_info	tag	u1	值为 10
	index	u2	指向声明方法的类描述符 CONSTANT_Class_info 的索引项
	index	u2	指向名称及类型描述符 CONSTANT_NameAndType 的索引项
CONSTANT_InterfaceMethodref_info	tag	u1	值为 11
	index	u2	指向声明方法的接口描述符 CONSTANT_Class_info 的索引项
	index	u2	指向名称及类型描述符 CONSTANT_NameAndType 的索引项
CONSTANT_NameAndType_info	tag	u1	值为 12
	index	u2	指向该字段或方法名称常量项的索引
	index	u2	指向该字段或方法描述符常量项的索引

访问标志

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的访问信息。例如：这个 class 是类还是接口，是否定义为 public 类型等等。具体的标志位及含义如下：

表 6-7 访问标志

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，JDK 1.2 之后编译出来的类的这个标志为真
ACC_INTERFACE	0x0200	标识这是一个接口
ACC_ABSTRACT	0x0400	是否为 abstract 类型，对于接口或抽象类来说，此标志值为真，其他类值为假
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

类索引、父类索引和接口索引

类索引和父类索引都是一个 u2 类型的数据，而接口索引集合是一组 u2 类型的数据集合，class 文件由这三项数据来确定一个类的继承关系。

类索引、父类索引和接口索引集合都按顺序排列在访问标志之后，类索引和父类索引用两个 u2 类型的索引值表示，他们各自指向一个类型为 CONSTANT_Class_info 的类描述符常量，通过这个类型常量中的索引值可以找到定义在 CONSTANT_Utf-8_info 类型的常量中的全限定名字符串。

对于接口索引集合，入口的第一项-u2 类型的数据为接口计数器，表示索引表的容量，

如果该类没有实现任何接口，那么该计数器值为 0，后面的接口的索引表不再占用任何字节。

字段表集合

字段表用于描述接口或类中声明的变量。变量包括了类级变量或者实例变量，但不包括在方法内部声明的变量。字段表的结构如下：

表 6-8 字段表结构

类 型	名 称	数 量
u2	access_flags	1
u2	name_index	1
u2	descriptor_index	1
u2	attributes_count	1
attribute_info	attributes	attributes_count

字段修饰符放在 access_flags 项目中，它和类中的 access_flags 项目非常类似，都是一个 u2 的数据类型，可以设置的标志位和含义如下：

表 6-9 字段访问标志

标志名称	标志值	含 义
ACC_PUBLIC	0x0001	字段是否 public
ACC_PRIVATE	0x0002	字段是否 private
ACC_PROTECTED	0x0004	字段是否 protected
ACC_STATIC	0x0008	字段是否 static
ACC_FINAL	0x0010	字段是否 final
ACC_VOLATILE	0x0040	字段是否 volatile
ACC_TRANSIENT	0x0080	字段是否 transient
ACC_SYNTHETIC	0x1000	字段是否由编译器自动产生的
ACC_ENUM	0x4000	字段是否 enum

“全限定名”、“简单名称”、“描述符”的概念。

全限定名就是把类全名中的.换成/。例如 org/fenixsoft/clazz/TestClass。简单名称就是指没有类型和参数修饰的方法或字段名称，例如 private int m 的简单名称就是 m。描述符和前两个比起来复杂了一些，描述符的作用是用来描述字段的数据类型、方法的参数列表、返回值。根据描述符规则，基本数据类型及代表无返回值的 void 类型都用一个大写字母来表示，而对象类型则用 L 加上对象的全限定名来表示。

表 6-10 描述符标识字符含义

标识字符	含 义
B	基本类型 byte
C	基本类型 char
D	基本类型 double
F	基本类型 float
I	基本类型 int
J	基本类型 long
S	基本类型 short
Z	基本类型 boolean
V ^①	特殊类型 void
L	对象类型, 如 Ljava/lang/Object;

对于数组类型, 每一维度将使用一个前置的 “[” 字符来描述, 如一个定义为 “java.lang.String[][]” 类型的二维数组, 将被记录为: “[Ljava/lang/String;”, 一个整型数组 “int[]” 将被记录为 “[I”。

用描述符来描述方法时, 按照先参数列表, 后返回值的顺序描述, 参数列表按照

① void 类型在虚拟机规范之中单独列出为 “VoidDescriptor”, 笔者为了结构统一, 将其列在基本数据类型中一起描述。

152 第三部分 虚拟机执行子系统

参数的严格顺序放在一组小括号 “()” 之内。如方法 void inc() 的描述符为 “()V”, 方法 java.lang.String toString() 的描述符为 “()Ljava/lang/String;”, 方法 int indexOf(char[] source, int sourceOffset, int sourceCount, char[] target, int targetOffset, int targetCount, int fromIndex) 的描述符为 “[CII[CIII]I”。

方法表集合

表 6-11 方法表结构

类 型	名 称	数 量
u2	access_flags	1
u2	name_index	1
u2	descriptor_index	1
u2	attributes_count	1
attribute_info	attributes	attributes_count

表 6-12 方法访问标志

标志名称	标志值	含 义
ACC_PUBLIC	0x0001	方法是否为 public
ACC_PRIVATE	0x0002	方法是否为 private
ACC_PROTECTED	0x0004	方法是否为 protected
ACC_STATIC	0x0008	方法是否为 static
ACC_FINAL	0x0010	方法是否为 final
ACC_SYNCHRONIZED	0x0020	方法是否为 synchronized
ACC_BRIDGE	0x0040	方法是否是由编译器产生的桥接方法
ACC_VARARGS	0x0080	方法是否接受不定参数
ACC_NATIVE	0x0100	方法是否为 native
ACC_ABSTRACT	0x0400	方法是否为 abstract
ACC_STRICT	0x0800	方法是否为 strictfp
ACC_SYNTHETIC	0x1000	方法是否是由编译器自动产生的

方法里的 Java 代码，存放在方法属性表集合中一个名为 `Code` 的属性里面。

Java 中，如果要重载父类的方法，除了要与父类方法拥有相同的简单名称外，还要求必须拥有一个和原方法不同的特征签名，特征签名就是一个方法中各个参数在常量池中的字段符号引用的集合。也就是因为返回值不会包含在特征签名之中，所以 Java 中无法通过返回值的不同对一个已有的方法进行重载。但是在 class 文件格式中，特征签名的范围更大一些，只要描述符不完全一致两个方法就可以共存。

属性表集合

在 class 文件结构中，字段表、方法表都可以携带自己的属性表集合，以用于描述某些场景专有的信息。虚拟机规范定义好的能识别的属性如下：

表 6-13 虚拟机规范预定义的属性

属性名称	使用位置	含 义
Code	方法表	Java 代码编译成的字节码指令
ConstantValue	字段表	final 关键字定义的常量值
Deprecated	类、方法表、字段表	被声明为 deprecated 的方法和字段
Exceptions	方法表	方法抛出的异常
InnerClasses	类文件	内部类列表
LineNumberTable	Code 属性	Java 源码的行号与字节码指令的对应关系
LocalVariableTable	Code 属性	方法的局部变量描述
SourceFile	类文件	源文件名称
Synthetic	类、方法表、字段表	标识方法或字段为编译器自动生成的

属性表的结构如下：

表 6-14 属性表结构

类 型	名 称	数 量
u2	attribute_name_index	1
u2	attribute_length	1
u1	info	attribute_length

Code 属性

Java 程序方法体里的代码经过 javac 编译器编译处理之后，最终变成字节码指令存储在 Code 属性中。并非所有的方法表都必须存在这个属性，例如抽象类和接口。如果有 Code 属性的话，结构如下：

表 6-15 Code 属性表的结构

类 型	名 称	数 量
u2	attribute_name_index	1
u4	attribute_length	1
u2	max_stack	1
u2	max_locals	1
u4	code_length	1
u1	code	code_length
u2	exception_table_length	1
exception_info	exception_table	exception_table_length
u2	attributes_count	1
attribute_info	attributes	attributes_count

Exceptions 属性

列举出方法中可能抛出的受检查异常。也就是方法 throws 关键字后面举出的异常。

表 6-17 属性表结构

类 型	名 称	数 量
u2	attribute_name_index	1
u4	attribute_length	1
u2	number_of_exceptions	1
u2	exception_index_table	number_of_exceptions

此属性中的 number_of_exceptions 项表示方法可能抛出 number_of_exceptions 种受查异常，每一种受查异常使用一个 exception_index_table 项表示，exception_index_table 是一个指向常量池中 CONSTANT_Class_info 型常量的索引，代表了该受查异常的类型。

LineNumberTable 属性

LineNumberTable 属性用于描述 Java 源码行号与字节码行号（字节码的偏移量）之间的对应关系。它并不是运行时必需的属性，但默认会生成到 Class 文件之中，可以在 Javac 中使用 -g:none 或 -g:lines 选项来取消或要求生成这项信息。如果选择不生成 LineNumberTable 属性，对程序运行产生的最主要的影响就是在抛出异常时，堆栈中将不会显示出错的行号，并且在调试程序的时候无法按照源码来设置断点。LineNumberTable 属性的结构如表 6-18 所示。

表 6-18 LineNumberTable 属性结构

类 型	名 称	数 量
u2	attribute_name_index	1
u4	attribute_length	1
u2	line_number_table_length	1
line_number_info	line_number_table	line_number_table_length

line_number_table 是一个数量为 line_number_table_length、类型为 line_number_info 的集合，line_number_info 表包括了 start_pc 和 line_number 两个 u2 类型的数据项，前者是字节码行号，后者是 Java 源码行号。

LocalVariable 属性

LocalVariableTable 属性用于描述栈帧中局部变量表中的变量与 Java 源码中定义的变量之间的关系，它不是运行时必需的属性，默认也不会生成到 Class 文件之中，可以在 Javac 中使用 -g:none 或 -g:vars 选项来取消或要求生成这项信息。如果没有生成这项属性，最大的影响就是当其他人引用这个方法时，所有的参数名称都将丢失，IDE 可能会使用诸如 arg0、arg1 之类的占位符来代替原有的参数名，这对程序运行没有影响，但是会给代码编写带来较大的不便，而且在调试期间调试器无法根据参数名称从运行上下文中获得参数值。LocalVariableTable 属性的结构如表 6-19 所示。

表 6-19 LocalVariableTable 属性结构

类 型	名 称	数 量
u2	attribute_name_index	1
u4	attribute_length	1
u2	local_variable_table_length	1
local_variable_info	local_variable_table	local_variable_table_length

其中 local_variable_info 项目代表了一个栈帧与源码中的局部变量的关联，结构如表 6-20 所示。

start_pc 和 length 属性分别代表了这个局部变量的生命周期开始的字节码偏移量及其作用范围覆盖的长度，两者结合起来就是这个局部变量在字节码之中的作用域范围。

name_index 和 descriptor_index 都是指向常量池中 CONSTANT_Utf8_info 型常量的索引，分别代表了局部变量的名称及该局部变量的描述符。

表 6-20 local_variable_info 项目结构

类 型	名 称	数 量
u2	start_pc	1
u2	length	1
u2	name_index	1
u2	descriptor_index	1
u2	index	1

SourceFile 属性

SourceFile 属性用于记录生成这个 Class 文件的源码文件名称。这个属性也是可选的，可以使用 Javac 的 -g:none 或 -g:source 选项来关闭或要求生成这项信息。在 Java 中，对于大多数的类来说，类名和文件名是一致的，但是有一些特殊情况（如内部类）例外。如果不生成这项属性，当抛出异常时，堆栈中将不会显示出错误代码所属的文件名。这个属性是一个定长的属性，其结构如表 6-21 所示。

表 6-21 SourceFile 属性结构

类 型	名 称	数 量
u2	attribute_name_index	1
u4	attribute_length	1
u2	sourcefile_index	1

sourcefile_index 数据项是指向常量池中 CONSTANT_Utf8_info 型常量的索引，常量值是源码文件的文件名。

ConstantValue 属性

ConstantValue 属性的作用是通知虚拟机自动为静态变量赋值。只有被 static 关键字修饰的变量（类变量）才可以使用这项属性。在 Java 程序里面类似“int x = 123”和“static int x = 123”这样的变量定义是非常常见的事情，但虚拟机对这两种变量赋值的方式和时刻都有所不同。对于非 static 类型的变量（也就是实例变量）的赋值是在实例构造器 <init> 方法中进行的；而对于类变量，则有两种方式可以选择：赋值在类构造器 <clinit> 方法中进行，或者使用 ConstantValue 属性来赋值。目前 Sun Java 编译器的选择是：如果同时使用 final 和 static 来修饰一个变量（或者说常量更贴切），并且这个变量的数据类型是基本类型或 java.lang.String 的话，就生成 ConstantValue 属性来进行初始化，如果这个变量没有被 final 修饰，或者并非基本类型及字符串，则选择在 <clinit> 方法中进行初始化。

表 6-22 ConstantValue 属性结构

类 型	名 称	数 量
u2	attribute_name_index	1
u4	attribute_length	1
u2	constantvalue_index	1

从数据结构中可以看出 ConstantValue 属性是一个定长属性，它的 attribute_length 数据项值必须固定为 2。constantvalue_index 数据项代表了常量池中一个字面量常量的引用，根据字段类型的不同，字面量可以是 CONSTANT_Long_info、CONSTANT_Float_info、CONSTANT_Double_info、CONSTANT_Integer_info 和 CONSTANT_String_info 常量中的一种。

InnerClasses 属性

InnerClasses 属性用于记录内部类与宿主类之间的关联。如果一个类中定义了内部类，那编译器将会为它及它所包含的内部类生成 InnerClasses 属性。属性的结构如表 6-23 所示。

表 6-23 InnerClasses 属性结构

类 型	名 称	数 量
u2	attribute_name_index	1
u4	attribute_length	1
u2	number_of_classes	1
inner_classes_info	inner_classes	number_of_classes

数据项 number_of_classes 代表需要记录多少个内部类信息，每一个内部类的信息都由一个 inner_classes_info 表进行描述。inner_classes_info 表的结构如表 6-24 所示。

表 6-24 inner_classes_info 表的结构

类 型	名 称	数 量
u2	inner_class_info_index	1
u2	outer_class_info_index	1
u2	inner_name_index	1
u2	inner_class_access_flags	1

inner_class_info_index 和 outer_class_info_index 都是指向常量池中 CONSTANT_Class_info 型常量的索引，分别代表了内部类和宿主类的符号引用。

inner_name_index 是指向常量池中 CONSTANT_Utf8_info 型常量的索引，代表这个内部类的名称，如果是匿名内部类，则这项值为 0。

inner_class_access_flags 是内部类的访问标志，类似于类的 access_flags，它的取值范围如表 6-25 所示。

Deprecated 属性和 Synthetic 属性

Deprecated 和 Synthetic 两个属性都属于标志类型的布尔属性，只存在有和没有的区别，没有属性值的概念。

Deprecated 属性用于表示某个类、字段或方法，已经被程序作者定为不再推荐使用，它可以通过在代码中使用 @deprecated 注释进行设置。

Synthetic 属性代表此字段或方法并不是由 Java 源码直接产生的，而是由编译器自行添加的，在 JDK 1.5 之后，标识一个类、字段或方法是编译器自动产生的，也可以设置它们访问标志中的 ACC_SYNTHETIC 标志位，其中最典型的例子就是 Bridge Method。所有由非用户代码产生的类、方法及字段都应当至少设置 Synthetic 属性和 ACC_SYNTHETIC 标志位中的一项，唯一的例外是实例构造器 “<init>” 方法和类构造器 “<clinit>” 方法。

Deprecated 和 Synthetic 属性的结构非常简单，如表 6-26 所示。

表 6-26 Deprecated 及 Synthetic 属性的结构

类型	名称	数量
u2	attribute_name_index	1
u4	attribute_length	1

其中 attribute_length 数据项的值必须为 0x00000000，因为没有任何属性值需要设置。

虚拟机类加载机制

虚拟机把描述类的 class 文件加载到内存，并对数据进行校验、转化解析和初始化，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。

Java 语言中，类型的加载和连接过程都是在程序运行期间完成的，这样会在类加载时增加一点开销，但是却为 Java 应用程序提供了高的灵活性，Java 中天生可以动态扩展的语言特性就是依赖运行期间动态加载和动态连接过程来实现的。。

类加载的时机

类从被加载到虚拟机内存开始，到被卸载出内存，它的整个生命周期包括了加载、验证、准备、解析、初始化、使用和卸载七个阶段，其中验证、准备、解析三个阶段被称为连接。

加载、验证、准备、初始化和卸载五个阶段的顺序是确定的，解析阶段的顺序是不确定的，在某些情况下可以在初始化阶段之后才开始，这是为了支持 Java 语言的运行时绑定。

关于类的初始化，JVM 规范规定了有且只有四种情况必须立刻对类进行初始化（而加载、验证、准备阶段必须在之前开始）。

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，则需要触发其初始化，生成这四条字节码指令的常见 Java 场景是，使用 new 关键字实例化对象的时候，读取或者设置一个类的静态字段（final 修饰的除外），以及调用一个类的静态方法的时候。
2. 使用 java.lang.reflect 包对类进行反射操作的时候。
3. 当初始化一个类的时候，如果发现这个类的父类还没有进行初始化，则需要对其父类先进行初始化。
4. 当虚拟机启动的时候，虚拟机会首先初始化 main 方法所在的那个类。

接口也有初始化过程，但是接口中不能使用 static 语句块，但是编译器仍然会为接口生成类构造器，用于初始化接口所定义的成员变量，接口和类真正有区别的是前面讲述的有且仅有场景中的第三种，当一个接口初始化时，不要求其父接口已经初始化完成了，只有当真正用到父接口的时候，才会初始化。

类加载的过程

加载

加载阶段，虚拟机需要完成三件事情：

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换为方法区的运行时数据结构。
3. 在 Java 堆中生成一个 `java.lang.Class` 对象，作为方法区这些数据的访问入口。

这个阶段是开发期可控性最强的阶段，因为加载阶段既可以使用系统提供的类加载器完成，也可以使用用户自定义的类加载器来完成，开发人员可以通过自定义类加载器去控制字节流的获取方式。加载阶段和连接阶段的部分内容是交叉进行的，加载阶段尚未完成，连接阶段可能已经开始。但是这两个阶段的开始时间仍然保持着先后顺序。

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，并且不会危及虚拟机自身的安全。这个阶段大致都会完成下面四个阶段的检验过程：

文件格式验证

第一阶段主要验证 `class` 文件看是否符合 `class` 文件格式的规范，并且能被当前的虚拟机版本处理。这一阶段可能包含如下验证点：

- 是否以魔数 0xCAFEBAE 开头。
- 主、次版本号是否在当前虚拟机处理范围之内。
- 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）。
- 指向常量的各种索引值中是否有指向不存在的常量或不符合类型的常量。
- `CONSTANT_Utf8_info` 型的常量中是否有不符合 UTF8 编码的数据。
- `Class` 文件中各个部分及文件本身是否有被删除的或附加的其他信息。
-

经过了这个阶段的验证之后，字节流才会进入内存的方法区中进行存储，所以后面的三个验证阶段全都是基于方法区的存储结构进行的。

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，以保证其描述的信息符合 Java 语言规范的要求，这个阶段可能包括的验证点如下：

- 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）。
- 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）。
- 如果这个类不是抽象类，是否实现了其父类或接口之中要求实现的所有方法。
- 类中的字段、方法是否与父类产生了矛盾（例如覆盖了父类的 `final` 字段，或者出现不符合规则的方法重载，例如方法参数都一致，但返回值类型却不同等）。
-

第二个阶段主要是对类的元数据信息进行语义校验，保证不存在不符合 Java 语义规范的元数据信息。

字节码验证

本阶段是最复杂的一个验证阶段，主要工作是进行工作流和控制流的分析，这阶段将对类的方法体进行校验分析，这阶段的任务是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行为。例如：

- 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，例如不会出现类似这样的情况：在操作栈中放置了一个 `int` 类型的数据，使用时却按 `long` 类型来加载入本地变量表中。
- 保证跳转指令不会跳转到方法体以外的字节码指令上。
- 保证方法体中的类型转换是有效的，例如可以把一个子类对象赋值给父类数据类型，这是安全的，但是把父类对象赋值给子类数据类型，甚至把对象赋值给与它

① 源码位置：`hotspot\src\share\vm\classfile\classFileParser.cpp`。

◆ 第三部分 虚拟机执行子系统

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。

□

由于数据验证的高复杂性，虚拟机设计团队为了避免将过多的时间消耗在字节码验证阶段，在 jdk1.6 之后的 `javac` 编译器中进行了一项优化，给方法体的 `Code` 属性的属性表中增

加了一项名为“`StackMapTable`”的属性，这项属性描述了方法体中所有的基本块开始时本地变量表和操作栈应有的状态，这可以将字节码验证的类推导转变为类型检查从而节省一些时间，`jdk1.7`之后，使用类型检查来完成数据流分析是唯一的选择。

符号引用验证

最后一个阶段的验证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个阶段-解析阶段中完成，符号引用验证可以看做是对类自身以外的信息进行匹配性的校验，通常需要校验以下内容：

- 符号引用中通过字符串描述的全限定名是否能找到对应的类。

-
- ① 停机问题就是判断任意一个程序是否会在有限的时间之内结束运行的问题。如果这个问题可以在有限的时间之内解决，则可以有一个程序判断其本身是否会停机并做出相反的行为。这时候显然不管停机问题的结果是什么都不会符合要求。所以这是一个不可解的问题。具体的证明过程可参考：[http://zh.wikipedia.org/zh/ 停机问题。](http://zh.wikipedia.org/zh/)
-

- 在指定类中是否存在符合方法的字段描述符及简单名称所描述的方法和字段。
- 符号引用中的类、字段和方法的访问性（`private`、`protected`、`public`、`default`）是否可被当前类访问。
-

符号引用验证的目的是确保解析动作能正常执行，如果无法通过符号引用验证，将会抛出对应的异常。

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时候，这些内存都将在方法区中进行分配。两个注意点：一是仅仅是类变量是在方法区中完成分配，实例变量是随着对象的创建在堆内存中分配，二是这里的初始值通常是指数据类型的零值。

那么什么情况下不赋零值呢？

如果类字段的属性表中存在 `ConstantValue` 属性，则将会在准备阶段把变量的值初始化为 `ConstantValue` 指定的值。

解析

解析阶段是将常量池中的符号引用转换为直接引用的过程，在解析阶段符号引用和直接引用有什么关联呢？

符号引用：符号引用以一组符号来描述引用的目标，符号可以是以任何形式的字面量，只

要使用时能够无歧义的定位到目标即可，符号引用与虚拟机的内存布局无关，引用的目标并不一定加载到了内存中。

直接引用：直接引用是可以直接指向目标的指针、相对偏移量或是一个间接定位到目标的句柄，直接引用和虚拟机实现的内存布局相关，而一个符号引用在不同的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，那么引用的目标一定在内存中。

解析动作主要针对类或接口、字段、类方法、接口方法四类符号引用进行。

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包括以下 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，由于无数据验证、字节码验证的需要，又将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，解析过程就将宣告失败。
- 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要进行符号引用验证，确认 C 是否具备对 D 的访问权限。如果发现不具备访问权限，将抛出 java.lang.IllegalAccessError 异常。

字段的解析

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^①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。如果解析成功完成，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上往下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 C 不是 java.lang.Object 的话，将会按照继承关系从上往下递归搜索其父类，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。

如果查找过程成功返回了引用，将会对这个字段进行权限验证，如果发现不具备对字段的访问权限，将抛出 java.lang.IllegalAccessError 异常。

在实际应用中，虚拟机的编译器实现可能会比上述规范要求得更加严格一些，如果有一个同名字段同时出现在 C 的接口和父类中，或者同时在自己或父类的多个接口中出现，那编译器将可能拒绝编译。在代码清单 7-4 中，如果注释了 Sub 类中的“`public static int A = 4;`”，接口与父类同时存在字段 A，那编译器将提示“`The field Sub.A is ambiguous`”。并且会拒绝编译这段代码。

类方法的解析

类方法解析的第一个步骤与字段解析一样，也是需要先解析出类方法表的 `class_index`^① 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 `java.lang.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，这时候查找结束，抛出 `java.lang.AbstractMethodError` 异常。
- 5) 否则，宣告方法查找失败，抛出 `java.lang.NoSuchMethodError`。

最后，如果查找过程成功返回了直接引用，将会对这个方法进行权限验证；如果发现不具备对此方法的访问权限，将抛出 `java.lang.IllegalAccessError` 异常。

接口方法的解析

接口方法也是需要先解析出接口方法表的 class_index^①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，依然用 C 表示这个接口，接下来虚拟机将会按照如下步骤进行后续的接口方法搜索：

- 1) 与类方法解析相反，如果在接口方法表中发现 class_index 中的索引 C 是个类而不是接口，那就直接抛出 java.lang.IncompatibleClassChangeError 异常。
- 2) 否则，在接口 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，在接口 C 的父接口中递归查找，直到 java.lang.Object 类（查找范围会包括 Object 类）为止，看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，宣告方法查找失败，抛出 java.lang.NoSuchMethodError 异常。

由于接口中的所有方法都默认是 public 的，所以不存在访问权限的问题，因此接口方法的符号解析应当不会抛出 java.lang.IllegalAccessError 异常。

初始化

初始化是类加载过程的最后一步，前面提到的类加载过程中，除了加载阶段用户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控制和引导，到了初始化阶段，才真正开始执行类中定义的 Java 程序代码。在初始化阶段，根据程序员通过制定的主观计划去初始化类变量和其他资源，换个角度讲，初始化阶段是执行类的 <clinit>() 方法过程。

<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中的语句合并产生的，静态语句块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。

<clinit>() 方法和类的构造函数不同，它不需要显示的调用父类 <clinit>() 方法，虚拟机会保证子类的 <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执行 <clinit>() 方法的类一定是 java.lang.Object。

由于父类的 <clinit>() 方法先执行，所以父类中定义的静态语句块将会优先于子类的变量赋值操作。

<clinit>() 方法对于类或者接口来说不是必须的。

接口虽然没有静态语句块，但是接口也有对类变量的赋值操作，所以接口也有 <clinit>() 方法，接口和类不一样的地方是，执行接口的 <clinit>() 方法不需要先执行父接口的 <clinit>() 方法，除非父接口中定义的变量被使用，才会执行父接口的 <clinit>() 方法。

虚拟机保证类的 <clinit>() 是线程同步的。

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 JVM 外部去实现，以便让应用程序自己去决定如何获取所需要的类，实现这个动作的代码模块叫做类加载器。

类和类加载器

Java 中，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 JVM 中的唯一性。通俗地讲，比较两个类是否相等，只有在它们是由同一个类加载器加载的前提下才有意义。

这里的相等，包括 `equals` 方法，`isAssignableFrom` 方法以及 `isInstance` 方法。如果不注意类加载器的结果，那么可能会出现某些具有迷惑性的结果。

双亲委派模式

站在 JVM 的角度上讲，只有两种类型的类加载器，一种是启动类加载器，这个类加载器使用 c++ 实现，是虚拟机自身的一部分，另外一种就是所有其他的类加载器，这些类加载器都由 Java 语言实现，独立于虚拟机外部，并且全部继承自抽象类 `java.lang.ClassLoader`。

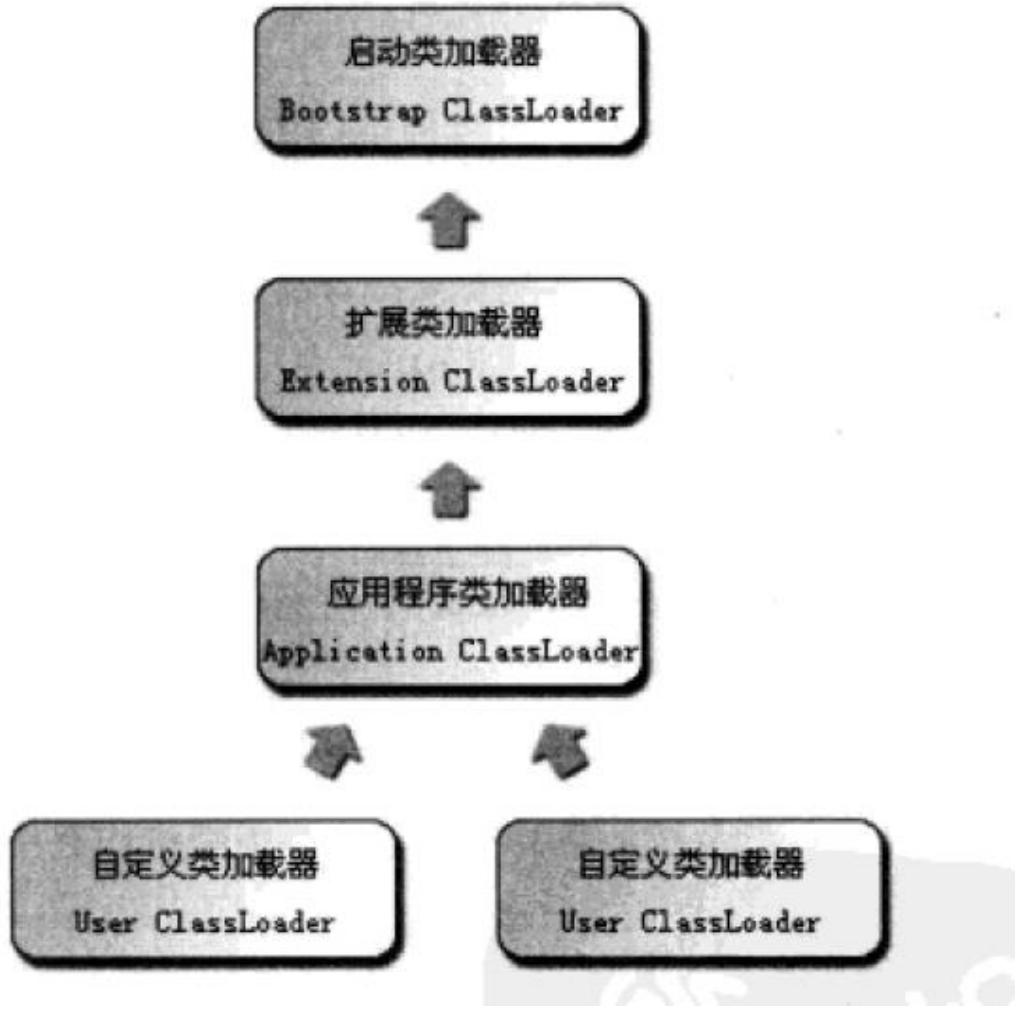
从 Java 程序员的角度来看，类加载器大致可以分为以下三种：

1. 启动类加载器：这个类加载器负责将存放在 `JAVA_HOME/lib` 目录下的，或者被 `-Xbootclasspath` 参数所指定的路径中的，并且是被虚拟机识别的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。

2. 扩展类加载器：它负责加载 `JAVA_HOME/lib/ext` 目录下的，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：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，它负责加载用户类路径上所指定的类库，开发者可以直接使用这个类加载器，一般情况下这个就是程序中的默认类加载器。

我们的应用程序是由这三种类加载器相互配合进行加载的，如果有必要，还可以加入自己的类加载器。这些类加载器之间的关系一般如图：



图中所展示的这种类加载器之间的层次关系，就称为类加载器的双亲委派模式。双亲委派模式要求除了顶层的启动类加载器之外，其他每个类加载器都要有自己的父类加载器，这些类加载器之间的父子关系一般都使用组合的方式来实现复用。

双亲委派模式的工作过程：如果一个类加载器收到了类加载器的请求，它首先不会自己尝试去加载这个类，而是把这个请求委派给它的父类加载器去完成，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时，子加载器才会尝试加载。

双亲委派模式的好处：显而易见的一个好处就是 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，例如类 rt.jar，无论哪个类加载器要加载这个类，最终都是委派给启动类加载器去完成加载。对于 Java 程序的稳定运作很重要。

虚拟机字节码执行引擎

运行时帧栈结构

帧栈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的虚拟机栈中的栈元素。帧栈存储了局部表、操作数栈、动态连接和方法返回值等信息。

一个帧栈需要分配多少内存，在编译程序代码的时候就确定了，并且写入到方法表的

`Code` 属性中。

活动线程中，只有栈顶的栈帧是有有效的。

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 程序被编译成 class 文件的时候，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最大局部变量表的容量。

局部变量表的容量以 `slot` 为基本单位。

虚拟机通过索引定位的方式使用局部变量表，索引值的范围是从 0 开始到局部变量表的最大 `slot` 数量。如果是 32 位数据类型的变量，索引 n 就代表使用第 n 个 `slot`，如果是 64 位数据类型的变量，索引 n 就代表使用第 n 和 n+1 个 `slot`。

局部变量表的 `slot` 是可重用的。

局部变量不像前面介绍的类变量那样有准备阶段，局部变量如果定义了但是没有赋初始值是不能使用的。

操作数栈

方法的执行过程中，会有各种字节码指令向操作数栈写入和提取内容。

操作数栈中元素的数据类型必须和字节码指令的序列严格匹配。

动态连接

每个帧栈都包含一个指向运行时常量池中该帧栈所属方法的引用，持有这个引用是为了支持方法调用过程中的动态连接。

方法返回地址

当一个方法被执行时，有两个方式退出这个方法，第一种方式是执行引擎遇到任何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由遇到何种方法返回指令来决定，这种退出方法叫做正常完成出口。

另外一种退出方式是在执行过程中遇到了异常，并且这个异常没有在方法体中进行处理。这种退出方式叫做异常完成出口，一个方法使用异常完成出口的方式退出，是不会给他的上层调用者产生任何返回值的。

方法调用

解析

所有方法调用中的目标方法在 `class` 文件中都是一个常量池的符号引用，在类加载的解析阶段，会将其中的一部分符号引用转换为直接引用，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，并且这个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。

符合这类要求的方法主要有静态方法、私有方法、实例构造器、父类方法和 `final` 修饰的方法。

解析调用一定是个静态的过程，在编译期间就完全确定，在类装载的解析阶段就会把设计的符号引用全部转变为可确定的直接引用，不会延迟到运行期再去完成。

分派

静态分派

在编译阶段，`javac` 编译器根据参数的静态类型来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，都称为静态分派。静态分派的典型应用就是方法重载。

动态分派

在运行期根据实际类型确定方法执行版本的分派过程叫做动态分派。Java 重写方法的本质。

高效并发

Java 内存模型

JVM 规范试图定义一种 Java 内存模型来屏蔽掉各种硬件和操作系统的内存访问差异，以实现让 Java 程序在各种平台下都能达到一致的并发效果。

主内存和工作内存

Java 内存模型的主要目的就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中和从内存中取出变量这样的底层细节。这里的变量和 Java 编程语言中的变量有区别，它包括了实例字段、静态字段和构成数组对象的元素，但是不包括局部变量和方法参数，因为后者是线程私有的，不存在竞争问题。

Java 内存模型规定了所有的变量存储在主内存，每条线程都有自己的工作内存，线程的工作内存中保存了被该线程用到的变量的主内存副本拷贝，线程对变量的所有操作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。不同线程之间也不能直接访问对方工作内存中的变量，线程间变量值的传递需要通过主内存来完成。三者的关系图大致如下：

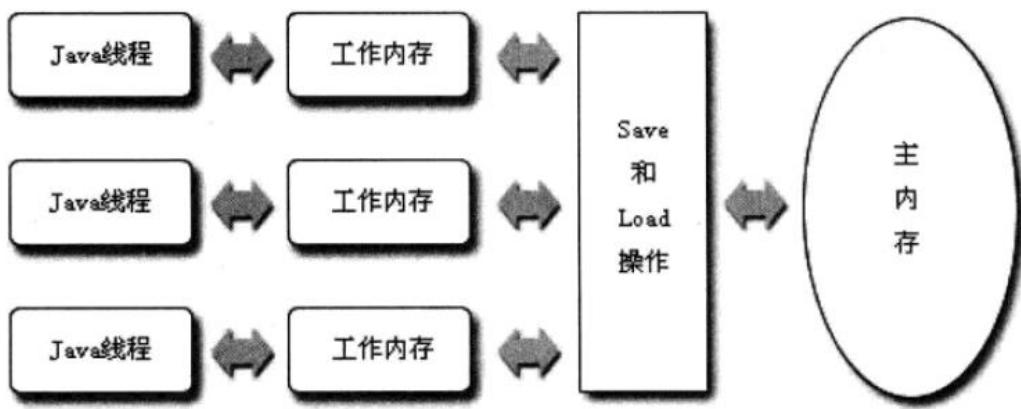


图 12-2 线程、主内存、工作内存三者的交互关系（请与图 12-1 对比）

内存间交互操作

关于主内存和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，一个变量如何从工作内存回写到主内存，Java 内存模型定义了以下 8 种操作来完成。

- ❑ **lock (锁定)**：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 (解锁)**：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** (读取)：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** (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。
- **use** (使用)：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** (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。
- **store** (存储)：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** (写入)：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。

如果需要把一个变量从主内存拷贝到工作内存，那就要按照顺序执行 **read** 和 **load** 操作，如果要把工作内存的变量同步到主内存，就要按照顺序执行 **store** 和 **write** 操作，Java 内存模型还规定了在使用上述 8 中操作时必须遵守的一些规定：

- 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中。
- 一个新的变量只能在主内存中“诞生”，不允许在工作内存中直接使用一个未被初始化（**load** 或 **assign**）的变量，换句话说就是对一个变量实施 **use** 和 **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，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 sotre 和 write 操作）。

对于 volatile 变量的特殊规则

当一个变量被定义成 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”指当一条线程修改了这个变量的值，新值对于其他线程来讲是可以立即得知的，而普通变量不能做到这一点，变量值在线程间传递均需要通过主内存来完成，如：线程 A 修改一个普通变量的值，然后向主内存进行回写，另外一条线程 B 在线程 A 回写完成了之后再从主内存进行读取操作，新变量的值才会对线程 B 可见。

由于 volatile 变量只能保证可见性，在不符合以下两条运算规则的场景中，我们仍然要通过加锁来保证原子性。

1. 运算结果不依赖于当前的值，或者能够保证单一的线程修改这个值
2. 变量不需要和其他的状态变量共同参与不变约束

使用 volatile 的第二个作用就是防止指令字节码重排序优化，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得正确的结果，而不能保证变量赋值操作顺序和程序代码中的顺序一致。

对于 long 和 double 型变量的特殊规则

Java 内存模型要求 lock、unlock、read、load、assign、use、store 和 write 这八个操作都具有原子性，但是对于 64 位的数据类型（long 和 double），在模型中特别定义了一条宽松的规定：允许虚拟机将没有被 volatile 修饰的 64 位数据的读写操作划分为两次

328 ◊ 第五部分 高效并发

32 位的操作来进行，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 这四个操作的原子性，这点就是所谓的 long 和 double 的非原子性协定（Nonatomic Treatment of double and long Variables）。

如果有多个进程共享一个并未声明为 volatile 的 long 或 double 类型的变量，并且同时对它们进行读取和修改操作，那么某些线程可能会读取到一个既非原值，也不是其他线程修改值的代表了“半个变量”的数值。

不过这种读取到“半个变量”的情况非常罕见，因为 Java 内存模型虽然允许虚拟机不把 long 和 double 变量的读写实现成原子操作，但允许虚拟机选择把这些操作实现为具有原子性的操作，而且还“强烈建议”虚拟机这样实现。在实际开发中，目前各种平台下的商用虚拟机几乎都选择把 64 位数据的读写操作作为原子操作来对待，因此我们在编写代码时一般不需要将用到的 long 和 double 变量专门声明为 volatile。

原子性、可见性和有序性

Java 内存模型是围绕着在并发过程中如何处理原子性、可见性、有序性这三个特征展开的。

原子性 (Atomicity)：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 这六个，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（long 和 double 的非原子性协定例外，笔者的观点是知道这件事情就可以了，无须太过在意这些几乎不会发生的例外情况）。

如果应用场景需要一个更大范围的原子性保证（经常会遇到），Java 内存模型还提供了 lock 和 unlock 操作来满足这种需求，尽管虚拟机未把 lock 和 unlock 操作直接开放给用户使用，但是却提供了更高层次的字节码指令 monitoreenter 和 monitorexit 来隐式地使用这两个操作，这两个字节码指令反映到 Java 代码中就是同步块——synchronized 关键字，因此在 synchronized 块之间的操作也具备原子性。

可见性 (Visibility)：可见性就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。上文在讲解 volatile 变量的时候我们已详细讨论过这一点。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。因此我们可以说 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，Java 还有两个关键字能实现可见性，它们是 synchronized 和 final。同步块的可见性是由“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 和 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被初始化完成，并且构造器没有把“this”的引用传递出去（this 引用逃逸是一件很危险的事情，其他线程有可能通过这个引用访问到“初始化了一半”的对象），那么在其他线程中就能看见 final 字段的值。如代码清单 12-5 所示，变量 i 与 j 都具备可见性，它们无须同步就能被其他线程正确地访问。

有序性 (Ordering)：Java 内存模型的有序性在前面讲解 volatile 时也详细地讨论过了，Java 程序中天然的有序性可以总结为一句话：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。前半句是指“线程内表现为串行的语义”(Within-Thread As-If-Serial Semantics)，后半句是指“指令重排序”现象和“工作内存与主内存同步延迟”现象。

Java 语言提供了 volatile 和 synchronized 两个关键字来保证线程之间操作的有序性，volatile 关键字本身就包含了禁止指令重排序的语义，而 synchronized 则是由“一个变量

330 ◆ 第五部分 高效并发

在同一个时刻只允许一条线程对其进行 lock 操作”这条规则获得的，这个规则决定了持有同一个锁的两个同步块只能串行地进入。

先行发生原则

下面是 Java 内存模型下一些“天然的”先行发生关系，这些先行发生关系无须任何同步器协助就已经存在，可以在编码中直接使用。如果两个操作之间的关系不在此列，并且无法从下列规则推导出来的话，它们就没有顺序性保障，虚拟机可以对它们进行随意地重排序。

- **程序次序规则 (Program Order Rule)**：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 (Monitor Lock Rule)**：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 (Volatile Variable Rule)**：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 Start Rule)**：Thread 对象的 start() 方法先行发生于此线程的每一个动作。

- **线程终止规则** (Thread Termination Rule)：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.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** (Thread Interruption Rule)：对线程 interrupt() 方法的调用先行发生于被中断线程的代码检测到中断事件的发生，可以通过 Thread.interrupted() 方法检测到是否有中断发生。
- **对象终结规则** (Finalizer Rule)：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** (Transitivity)：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。