

* **程序计数器**（Program Counter Register）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
* 每条线程都有一个独立的程序计数器，各条线程之间计数器互不影响——“**线程私有**”
* 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**Native方法**，这个计数器值则为**空**
* **Java虚拟机栈**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**：**每个方法在执行的同时都会创建一个栈帧**（Stack Frame）用于存储**局部变量表**、**操作数栈**、**动态链接**、**方法出口**等信息。**每一个方法从调用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**
* **局部变量表**存放了**编译期可知**的各种**基本数据类型**（boolean、byte、char、short、int、float、long、double）、**对象引用**（reference类型，它不等同于对象本身，可能是一个指向对象起始地址的引用指针，也可能是指向一个代表对象的句柄或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）
* 64位长度的long和double类型的数据会占用2个局部变量空间（Slot），其余的数据类型只占用1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完全确定的，在方法运行期间不会改变局部变量表的大小**
* 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；如果虚拟机栈可以动态扩展，如果扩展时无法申请到足够的内存，就会抛出**OutOfMemoryError**异常
* **本地方法栈**（Native Method Stack）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**（也就是字节码）服务，而**本地方法栈**则为虚拟机使用到的**Native方法**服务

|  |  |  |  |
| --- | --- | --- | --- |
| **变量类型** | **存储空间（位）** | **变量类型** | **存储空间（位）** |
| boolean | 未知 | int | 32 |
| byte | 8 | float | 32 |
| char | 16 | long | 64 |
| short | 16 | double | 64 |

* **Java堆**（Java Heap）是Java虚拟机所管理的内存中最大的一块
* **所有的对象实例以及数组都要在堆上分配**，但是随着JIT（Just In Time）编译器的发展与逃逸分析技术逐渐成熟，**栈上分配**、**标量替换**优化技术将会导致一些微妙的变化发生，所有的对象都分配在堆上也渐渐变得不是那么“绝对”了
* 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
* -Xmx：最大堆容量 -Xms：初始堆容量
* 如果在堆中没有内存完成实例分配，并且堆也无法再扩展时，将会抛出**OutOfMemoryError**异常
* **方法区**（Method Area）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**等数据
* 方法区别名：非堆（Non-Heap）、永久代（Permanent Generation）
* -XX:MaxPermSize：最大方法区容量
* 方法区**不需要连续的内存**，且可以选择固定大小或者可扩展外，还**可以选择不实现垃圾收集**
* 当方法区无法满足内存分配需求时，将抛出**OutOfMemoryError**异常
* **运行时常量池**（Runtime Constant Pool）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（Constant Pool Table），用于**存放编译期生成的各种字面量和符号引用**
* 除了保存Class文件中描述的**符号引用**外，还会把翻译出来的**直接引用**也存储在运行时常量池中
* 运行时常量池具备**动态性**，Java语言并不要求常量一定只有编译期才能产生，也就是并非预置入Class文件中常量池的内容才能进入方法区运行时常量池，运行期间也可能将新的常量放入池中，如**String类的intern()方法**
* 当常量池无法再申请到内存时会抛出**OutOfMemoryError**异常
* 在JDK1.4中新加入了**NIO**类，引入了一种基于**通道（Channel）**与**缓冲区（Buffer）**的I/O方式，**它可以使用Native函数库直接分配堆外内存，然后通过一个存储在Java堆中的DirectByteBuffer对象作为这块内存的引用进行操作**
* 本机直接内存的分配不会受到Java堆大小的限制，但会受到本机总内存大小以及处理器寻址空间的限制。服务器管理员在配置虚拟机参数时，会根据实际内存设置-Xmx等参数信息，但经常忽略直接内存，使得各个内存区域总和大于物理内存限制，从而导致动态扩展时出现**OutOfMemoryError**异常。

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再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**。

在类加载检查通过后，接下来虚拟机将为新生对象分配内存。**对象所需内存的大小在类加载完成后便可完全确定**，为对象分配空间的任务等同于把一块确定大小的内存从Java堆中划分出来。假设Java堆中内存是绝对规整的，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界点的指示器，那所分配内存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，这种分配方式称为“**指针碰撞**”（Bump the Pointer）。如果Java堆中的内存并不是规整的，已使用的内存和空闲的内存相互交错，那就没有办法简单地进行指针碰撞了，虚拟机就必须维护一个列表，记录上哪些内存块是可用的，在分配的时候从列表中找到一块足够大的空间划分给对象实例，并更新列表上的记录，这种分配方式称为“**空闲列表**”（Free List）。选择哪种分配方式由Java堆是否规整决定，而Java堆是否规整又由**所采用的垃圾收集器是否带有压缩整理功能**决定。因此，在使用Serial、ParNew等带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。

除如何划分可用空间之外，还有另外一个需要考虑的问题是对象创建在虚拟机中是非常频繁的行为，即使是仅仅修改一个指针所指向的位置，在并发情况下也并不是线程安全的，可能出现正在给对象A分配内存，指针还没来得及修改，对象B又同时使用了原来的指针来分配内存的情况。解决这个问题有两种方案，一种是对分配内存空间的动作进行同步处理——实际上虚拟机采用**CAS配上失败重试**的方式保证更新操作的原子性；另一种是**把内存分配的动作按照线程划分在不同的空间之中进行，即每个线程再Java堆中预先分配一小块内存，称为本地线程分配缓冲（Thread Local Allocation Buffer， TLAB）**。哪个线程要分配内存，就在哪个线程的TLAB上分配，**只有TLAB用完并分配新的TLAB时，才需要同步锁定**。虚拟机是否使用TLAB，可以通过-XX:+/-UseTLAB参数类设定。

接下来，虚拟机要对对象进行必要的设置，例如这个对象是哪个类的实例、如何才能找到**类的元数据信息**、**对象的哈希码**、**对象的GC分代年龄**等信息。这些信息存放在**对象的对象头**（Object Header）之中。根据虚拟机当前的运行状态的不同，如是否启用偏向锁等，对象头会有不同的设置。

在上面的工作都完成之后，从虚拟机的视角来看，一个新的对象已经产生了，但从Java程序的视角来看，对象创建才刚刚开始——**<init>**方法还没有执行，所有的字段都还为零。所以，一般来说（由字节码中是否跟随**invokespecial**指令所决定），执行new指令之后会接着执行<init>方法，把对象按照程序员的意愿进行初始化，这样一个真正可用的对象才算产生出来。

* 对象在内存中存储的布局可以分为3块区域：**对象头**（Header）、**实例数据**（Instance Data）和**对齐填充**（Padding）。
* **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**哈希码**（HashCode）、**GC分代年龄**、**锁状态标志**、**线程持有的锁**、**偏向线程ID**、**偏向时间戳**等，这部分数据的长度在32位和64位的虚拟机中分别为32bit和64bit，官方称它为“**Mark Word**”。Mark Word被设计成一个非固定的数据结构以便在极小的空间内存储尽量多的信息，它会根据对象的状态复用自己的存储空间。对象头的另外一部分是**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。**查找对象的元数据信息并不一定要经过对象本身**（句柄）。**如果对象是一个Java数组，那在对象头中还必须有一块用于记录数组长度的数据，因为虚拟机可以通过普通Java对象的元数据信息确定Java对象的大小，但是从数组的元数据中却无法确定数组的大小**。
* **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。**无论是从父类继承下来的，还是在子类中定义的，都需要记录起来**。
* **对齐填充**仅仅起着**占位符**的作用，因为对象的大小必须是8字节的整数倍。



* **句柄**访问：Java堆中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。在对象被移动时只会改变句柄的实例数据指针，而reference本身不需要修改。
* **直接指针**访问：Java堆对象的布局中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。速度更快，节省了一次指针定位的时间开销。
* 通过参数**-XX:+HeapDumpOnOutOfMemoryError**可以让虚拟机在出现内存溢出异常时Dump出当前的内存堆转储快照以便事后进行分析。

Java堆内存的OOM异常是实际应用中常见的内存溢出异常情况。当出现Java堆内存溢出时，异常堆栈信息“java.lang.OutOfMemoryError”会跟着进一步提示“Java heap space”。

要解决这个区域的异常，一般的手段是先通过内存映像分析工具对Dump出来的堆转储快照进行分析，重点是确认内存中的对象是否是必要的，也就是要先分清楚到底是出现了**内存泄漏**（Memory Leak）还是**内存溢出**（Memory Overflow）。

如果是内存泄漏，可进一步通过工具查看泄漏对象到GC Roots的引用链。于是就能找到泄漏对象是通过怎样的路径与GC Roots相关联并导致垃圾收集器无法自动回收它们的。掌握了泄漏对象的类型信息及GC Roots引用链的信息，就可以比较准确地定位出泄漏代码的位置。

如果不存在泄漏，就是内存中的对象确实都还必须存活着，那就应当检查虚拟机的堆参数（-Xmx与-Xms），与机器物理内存对比看是否还可以调大，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，尝试减少程序运行期的内存消耗。

* 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。在JDK1.6中，intern()方法会把首次遇到的字符串实例复制到永久代中，返回的也是永久代中这个字符串实例的引用。在JDK1.7中，intern()实现不会再复制实例，只是在常量池中记录首次出现的实例引用。
* 由DirectMemory导致的内存溢出，一个明显的特征是在Heap Dump文件中不会看见明显的异常，如果读者发现OOM之后**Dump文件很小**，而程序中又直接或间接使用了NIO，那就可以考虑检查一下是不是这方面的原因。
* 在Java语言中，可作为GC Roots的对象包括下面几种：（1）**虚拟机栈（栈帧中的本地变量表）中引用的对象**。（2）**方法区中类静态属性引用的对象**。（3）**方法区中常量引用的对象**。（4）**本地方法栈中JNI（Java Native Interface）（即一般说的Native方法）引用的对象**。
* **强引用**：Object obj = new Object() 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**：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。
* **弱引用**：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。
* **虚引用**：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得到一个系统通知。

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()方法已经被虚拟机调用过**，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，那么这个对象将会放置在一个叫做F-Queue的队列之中，并在稍后由一个虚拟机自动建立的、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。

* 任何一个对象的finalize()方法都只会被系统自动调用一次。
* 类需要同时满足下面3个条件才能算是“无用的类”：（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**。
* “**标记-清除**”算法：**效率**问题，标记和清除两个过程的效率都不高；**空间**问题，标记清除之后会产生大量不连续的内存碎片。
* **复制**算法：Eden：Survivor：Survivor=8：1：1。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**（Handle Promotion）。如果另外一块Survivor空间没有足够空间存放上一次新生代收集下来的存活对象时，这些对象将直接通过分配担保机制进入老年代。
* **为什么要有Survivor区**：当Eden区对象满时，会触发一次Minor GC，剩余存活的对象会进入Survivor区。而如果没有Survivor区，剩余存活的对象将会直接进入老年代。这样很快会导致老年代空间不够用，继而触发一次Major GC（由于发生Major GC的同时常常伴随着Minor GC，也可以看作是发生Full GC）。由于老年代空间一般是新生代的2倍，Major GC的耗时比起Minor GC是十分严重的。如果有了Survivor区，就可以保证对象只有在经过16次Minor GC之后才会进入老年代。
* **为什么要有两个Survivor区**：设置两个Survivor区最大的好处是解决了**碎片化**。假设现在只有一个Survivor区，我们来模拟一下流程：刚刚新建的对象在Eden区，当Eden区满的时候，触发一次Minor GC，Eden区中剩余的存活对象会被转移至Survivor区。但是，等到下一个Eden区又满的时候，如果只有一个Survivor区，那么此时Survivor区和Eden区都存放有一部分对象，如果此时将Eden区的存活对象硬塞到Survivor区，很明显这两部分对象所占有的内存是不连续的，也就导致了内存碎片化。
* “**标记-整理**”算法：标记过程仍然与“标记-清除”算法一样，但后续步骤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，然后直接清理掉端边界以外的内存。
* “**分代收集**”算法：**新生代用复制算法**，**老年代用“标记-清理”或“标记-整理”算法**。

枚举根节点时：**Stop The World**。当系统停顿下来之后，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，**虚拟机应当是有办法直接得知哪些地方存放着对象引用**。在HotSpot的实现中，是使用一组称为OopMap的数据结构来达到这个目的的。

HotSpot没有为每条指令都生成OopMap，只是在“特定的位置“记录了这些信息，这些位置称为**安全点**（SafePoint），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——因为每条指令执行的时间都非常短暂，程序不太可能因为指令流长度太长这个原因而过长时间运行，”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SafePoint。

如何在GC发生时让所有线程（这里不包括执行JNI调用的线程）都“跑“到最近的安全点上再停顿下来：**抢先式中断**和**主动式中断**。抢先式中断不需要线程的执行代码主动去配合，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，就恢复线程，让它”跑“到安全点上。主动式中断是当GC需要中断线程的时候，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，**轮询标志的地方和安全点是重合的，另外再加上创建对象需要分配内存的地方**。

**安全区域**是指在一段代码片段之中，引用关系不会发生变化。在这个区域中的任意地方开始GC都是安全的。在线程执行到Safe Region中的代码时，首先标识自己已经进入了Safe Region，那样，**当在这段时间里JVM要发起GC时，就不用管标识为Safe Region状态的线程了**。**在线程要离开Safe Region时，它要检查系统是否已经完成了根节点枚举（或者是整个GC过程）**，如果完成了，那线程就继续执行，否则它就必须等待直到收到可以安全离开Safe Region的信号为止。



* **Serial**收集器：单线程，它只会使用一个CPU或一条收集线程去完成垃圾收集工作。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。**复制算法**。
* **ParNew**收集器：Serial收集器的多线程版本，ParNew收集器在单CPU的环境中绝对不会有比Serial收集器更好的效果。
* **并行**：真正意义上的同时执行。
* **并发**：偏重于多个任务交替执行，而多个任务之间有可能还是串行的。
* **Parallel Scavenge**收集器：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收集器的目标是达到一个可控制的吞吐量（Throughput）。**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）**
* **Serial Old**收集器：Serial收集器的老年代版本。“**标记-整理**“算法。
* **Parallel Old**收集器：多线程，“**标记-整理**“算法。在注重吞吐量及CPU资源敏感的场合，都可以优先考虑Parallel Scavenge加Parallel Old收集器。
* **CMS**（Concurrent Mark Sweep）分为4个步骤：（1）**初始标记**，“Stop The World“，仅仅标记一下GC Roots能直接关联到的对象，速度快。（2）**并发标记**，进行GC Roots Tracing的过程，耗时。（3）**重新标记**，“Stop The World“，为了修正并发标记期间因用户程序继续运作而导致标记发生变动的那一部分对象的标记记录。（4）**并发清除**。
* CMS收集器的缺点：（1）**对CPU资源非常敏感**。当CPU在4个以上时，并发回收时垃圾收集线程不少于25%的CPU资源，并且随着CPU数量的增加而下降。但是当CPU不足4个（臂如2个）时，CMS对用户程序的影响就可能变得很大。（2）CMS收集器无法处理**浮动垃圾**（Floating Garbage），可能出现“**Concurrent Mode Failure**“失败而导致另一次Full GC的产生。由于在垃圾收集阶段用户线程还需要运行，需要预留有足够的内存空间给用户线程使用，因此**CMS收集器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**。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**。（3）**大量空间碎片的产生**。
* **G1**收集器的特点：（1）**并行与并发**。（2）**分代收集**。（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）来看是基于”复制“算法实现的。（4）**可预测的停顿**。
* G1将整个Java堆划分为**多个大小相等的独立区域（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的每个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把相关的引用信息记录到被引用对象所属的Region的Remembered Set之中**。当进行内存回收时，在GC根节点的枚举范围中加入Remembered Set即可保证不对全堆扫描也不会有遗漏。
* G1收集器的流程：（1）**初始标记**，标记一下GC Roots能直接关联到的对象。（2）**并发标记**，从GC Roots开始对堆中的对象进行可达性分析。（3）**最终标记**，“Stop The World”，可并行。（4）**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。
* 对象优先在Eden分配。
* 大对象直接进入老年代。
* 长期存活的对象进入老年代。对象在Survivor区中每“熬过”一次Minor GC，年龄就增加1岁，当它的年龄增加到一定程度（默认为15岁），就将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，年龄大于或等于该年龄的对象就可以直接进入老年代，无须等待MaxTenuringThreshold中要求的年龄。
* 空间分配担保。在发生Minor GC之前，虚拟机会先检查老年代最大可用的连续空间是否大于新生代所有对象总空间，如果这个条件成立，那么Minor GC可以确保是安全的。如果不成立，则虚拟机会查看HandlePromotionFailure设置值是否允许担保失败。如果允许，那么会继续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对象的平均大小，如果大于，将尝试着进行一次Minor GC，尽管这次Minor GC是有风险的；如果小于，或者HandlePromotionFailure设置不允许冒险，那这时也要改为进行一次Full GC。
* **新生代GC**（**Minor GC**）：发生在新生代的垃圾收集动作。
* **老年代GC**（**Major GC** / **Full GC**）：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。Major GC的速度一般会比Minor GC慢10倍以上。
* JDK开发团队选择采用Java代码来实现监控工具是有特别用意的：当应用程序部署到生产环境后，无论是直接接触物理服务器还是远程Telnet到服务器上都可能会受到限制。借助tools.jar类库里面的接口，我们可以直接在应用程序中实现功能强大的监控分析工具。

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

* **线程快照**就是当前虚拟机内每一条线程正在执行的方法堆栈的集合，生成线程快照的主要目的是定位线程出现长时间停顿的原因，如**线程间死锁**、**死循环**、**请求外部资源导致的长时间等待**等都是线程长时间停顿的常见原因。
* 任何一个Class文件都对应着唯一一个类或接口的定义信息，但反过来说，类或接口并不一定都得定义在文件里（譬如类或接口也可以通过类加载器直接生成）。
* Class文件是一组以8位字节为基础的二进制流，各个数据项目严格按照顺序紧凑地排列在Class文件之中，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时的必要数据，没有空隙存在。
* 根据Java虚拟机规范的规定，Class文件格式采用一种类似于C语言结构体的伪结构来存储数据，这种伪结构中只有两种数据类型：**无符号数**和**表**。
* **无符号数**属于基本的数据类型，以**u1**、**u2**、**u4**、**u8**来分别代表**1个字节**、**2个字节**、**4个字节**和**8个字节**的无符号数，无符号数可以用来描述**数字**、**索引引用**、**数量值**或者**按照UTF-8编码构成字符串值**。
* **表**是由多个**无符号数**或者其他**表**作为数据项构成的**复合**数据类型，所有表都习惯性地以“\_info“结尾。表用于描述有层次关系的复合结构的数据，整个Class文件本质上就是一张表。

|  |  |  |
| --- | --- | --- |
| 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 |

* 每个Class文件的头4个字节称为**魔数**（Magic Number），它的唯一作用是确定这个文件是否为一个能被虚拟机接受的Class文件。0x**CAFEBABY**
* 紧接着魔数的4个字节存储的是Class文件的版本号：第5和第6个字节是**次版本号**（Minor Version），第7和第8个字节是**主版本号**（Major Version）。

紧接着次主版本号之后的是常量池入口，常量池可以理解为Class文件之中的资源仓库，它是Class文件结构中与其他项目关联最多的数据类型，也是占用Class文件空间最大的数据项目之一，同时它还是在Class文件中第一个出现的表类型数据项目。

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一项u2类型的数据，代表常量池容量计数值（constant\_pool\_count）。与Java语言习惯不一样的是，这个**容量计数是从1而不是0开始的**。在Class文件格式规范制定之时，设计者将第0项常量空出来是有特殊考虑的，这样做的目的在于**满足后面某些指向常量池的索引值的数据在特定情况下需要表达“不引用任何一个常量池项目“的含义**，这种情况就可以把索引值置为0来标识。Class文件结构中只有常量池的容量计数是从1开始，对于其他集合类型，包括接口索引集合、字段表集合、方法表集合等的容量计数都与一般习惯相同，是从0开始的。

常量池中主要存放两大类常量：**字面量**（Literal）和**符号引用**（Symbolic Reference）。**字面量**比较接近于Java语言层面的常量概念，如**文本字符串**、**声明为final的常量值**等。而**符号引用**则属于编译原理方面的概念，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**

**Java代码在进行Javac编译时，并不像C和C++那样有“连接“这一步骤，而是在虚拟机加载Class文件的时候进行动态拼接**。在Class文件中不会保存各个方法、字段的最终内存布局信息，因此这些字段、方法的符号引用不经过运行期转换的话无法得到真正的内存入口地址，也就无法直接被虚拟机使用。**当虚拟机运行时，需要从常量池获得对应的符号引用，再在类创建时或运行时解析、翻译到具体的内存地址之中。**

常量池中每一项常量都是一个表，表开始的第一位是一个u1类型的标志位（tag），代表当前这个常量属于哪种类型。

由于Class文件中的方法、字段等都需要引用CONSTANT\_Utf8\_info型常量来描述名称，所以**CONSTANT\_Utf8\_info型常量的最大长度64KB也就是Java方法、字段名的最大长度**。

* 在常量池结束之后，紧接着的两个字节代表**访问标志**（access\_flags），这个标志用于识别一些类或接口层次的访问信息，包括：这个Class是类还是接口；是否定义为public类型；是否定义为abstract类型；如果是类的话，是否被声明为final。
* **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，它们各自指向一个类型为CONSTANT\_Class\_info的类描述符常量，通过CONSTANT\_Class\_info类型的常量中的索引值可以找到定义在CONSTANT\_Utf8\_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（interfaces\_count），表示索引表的容量。如果该类没有实现任何接口，则该计数器值为0，后面接口的索引表不再占用任何字节。
* **字段表**（field\_info）用于描述接口或者类中声明的变量。字段（field）包括类级变量以及实例级变量，但不包括方法内部声明的局部变量。**ACC\_FINAL**、**ACC\_VOLATILE**不能同时选择。接口之中的字段必须有**ACC\_PUBLIC**、**ACC\_STATIC**、**ACC\_FINAL**标志，这些都是由Java本身的语言规则所决定的。
* 根据**描述符**规则，**基本数据类型**（byte-B、char-C、boolean-Z、short-S、int-I、float-F、double-D、long-J）以及代表无返回值的void类型都用一个大写字母来表示，而对象类型则用字符L加对象的全限定名来表示。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，参数列表按照参数的严格顺序放在一组”()“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;“。
* 字段表集合中不会列出从超类或者父接口中继承而来的字段，但有可能列出原本Java代码之中不存在的字段，譬如在内部类中为了保持对外部类的访问性，会自动添加指向外部类实例的字段。另外，**在Java语言中字段是无法重载的，两个字段的数据类型、修饰符不管是否相同，都必须使用不一样的名称，但是对于字节码来讲，如果两个字段的描述符不一致，那字段重名就是合法的**。
* 方法里的代码，经过编译器编译成字节码指令后，存放在方法属性表集合中一个名为“Code“的属性里面，属性表是Class文件格式中最具扩展性的一种数据项目。
* 如果父类方法在子类中没有被重写（Override），方法表集合中就不会出现来自父类的方法信息，但可能出现由编译器自动添加的方法，如类构造器“<clinit>“方法和实例构造器”<init>“方法。Java代码的方法特征签名只包括了**方法名称**、**参数顺序**及**参数类型**，而字节码的特征签名还包括**方法返回值**以及**受查异常表**。
* 如果同时使用final和static来修饰一个变量，并且这个变量的数据类型是基本类型或者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，则将会选择在<clinit>方法中进行初始化。
* ConstantValue的属性值只能限于基本类型和String，因为此属性的属性值只是一个常量池的索引号，由于**Class文件格式的常量类型中只有与基本属性和字符串相对应的字面量**，所以计算ConstantValue属性想支持别的类型也无能为力。
* **StackMapTable**会在虚拟机类加载的字节码验证阶段被新类型检查验证器（Type Checker）使用，目的在于代替以前比较消耗性能的基于数据流分析的类型推导验证器。新的验证器在同样能保证Class文件合法性的前提下，省略了在运行期通过数据流分析去确认字节码的行为逻辑合法性的步骤，而是在编译阶段将一系列的验证类型（Verification Types）直接记录在Class文件之中，通过检查这些验证类型代替了类型推导过程，从而大幅提升了字节码验证的性能。StackMapTable属性中包含零至多个栈映射帧（Stack Map Frames），每个栈映射帧都显式或隐式地代表了一个字节码偏移量，用于表示执行到该字节码时局部变量表和操作数栈的验证类型。类型检查验证器会通过检查目标方法的局部变量和操作数栈所需要的类型来确定一段字节码指令是否符合逻辑约束。
* **BootstrapMethods**用于**保存invokedynamic指令引用的引导方法限定符**。如果某个类文件结构的常量池中曾经出现过CONSTANT\_InvokeDynamic\_info类型的常量，那么这个类文件的属性表中必须存在一个明确的BootstrapMethods属性，另外，即使CONSTANT\_InvokeDynamic\_info类型的常量在常量池中出现过多次，类文件的属性表中最多也只能有一个BootstrapMethods属性。
* 在Java虚拟机中，处理异常（catch语句）不是由字节码指令来实现的，而是采用**异常表**来实现的。

Java虚拟机可以支持方法级的同步和方法内部一段指令序列的同步，这两种同步结构都是使用**管程**（Monitor）来支持的。

方法级的同步是隐式地，即**无须通过字节码指令来控制，它实现在方法调用和返回操作之中**。虚拟机可以从方法常量池的方法表结构中的ACC\_SYNCHRONIZED访问标志得知一个方法是否声明为同步方法。当方法调用时，调用指令将会检查方法的ACC\_SYNCHRONIZED访问标志是否被设置。如果设置了，执行线程就要求先成功持有管程，然后才能执行方法，最后当方法完成（无论是正常完成还是非正常完成）时释放管程。在方法执行期间，执行线程持有了管程，其他任何线程都无法再获取到同一个管程。**如果一个同步方法执行期间抛出了异常，并且在方法内部无法处理此异常，那么这个同步方法所持有的管程将在异常抛到同步方法之外时自动释放**。

同步一段指令集序列通常是由Java语言中的synchronized语句块来表示的，Java虚拟机的指令集中有**monitorenter**和**monitorexit**两条指令来支持synchronized关键字的语义。编译器必须确保无论方法通过何种方式完成，方法中调用过的每条monitorenter指令都必须执行其对应的monitorexit指令，而无论这个方法是正常结束还是异常结束。

* Java虚拟机的2种实现形式：（1）**将输入的Java虚拟机代码在加载和执行时翻译成另外一种虚拟机的指令集**。（2）**将输入的Java虚拟机代码在加载或执行时翻译成宿主机CPU的本地指令集（JIT代码生成技术）**。
* 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，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。

Java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“：

（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，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.MethodHandler实例最后的解析结果是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。

* 许多新生的Java语法特性，都是靠编译器的“语法糖”来实现，而不是依赖虚拟机的底层改进来支持，Java中即时编译器在运行期的优化过程对于程序运行来说更重要，而前端编译器在编译期的优化过程对于程序编码来说关系更加密切。



* 编译过程大致分为3个过程，分别是：（1）**解析与填充符号表过程**。（2）**插入式注解处理器的注解处理过程**。（3）**分析与字节码生成过程**。
* **解析**步骤包括了经典程序编译原理中的**词法分析**和**语法分析**两个过程。词法分析是将源代码的字符流转变为**标记**（Token）集合，单个字符是程序编写过程的最小元素，而标记则是编译过程的最小元素，**关键字**、**变量名**、**字面量**、**运算符**都可以成为标记。语法分析是根据Token序列构造抽象语法树的过程，**抽象语法树**（Abstract Syntax Tree）是一种用来描述程序代码语法结构的树形表示方式，语法树的每一个节点都代表着程序代码中的一个语法结构，例如**包**、**类型**、**修饰符**、**运算符**、**接口**、**返回值**甚至**代码注释**都可以是一个语法结构。经过这个步骤之后，编译器就基本不会再对源码文件进行操作了，**后续的操作都建立在抽象语法树之上**。
* 完成了词法分析和语法分析之后，下一步就是**填充符号表**的过程。符号表（Symbol Table）是一组符号地址和符号信息构成的表格。符号表中所登记的信息在编译的不同阶段都要用到。**在语义分析中，符号表所登记的内容将用于语义检查**（如检查一个名字的使用和原先的说明是否一致）和产生中间代码。**在目标代码生成阶段，当对符号名进行地址分配时，符号表是地址分配的依据**。
* JDK1.6提供了一组插入式注解处理器的标准API在编译期间对注解进行处理，我们可以把它看做是一组编译器的插件，在这些插件里面，可以读取、修改、添加抽象语法树中的任意元素。**如果这些插件在处理注解期间对语法树进行了修改，编译器将回到解析及填充符号表的过程重新处理，直到所有插入式注解处理器都没有再对语法树进行修改为止**。
* **语法树能表示一个结构正确的源程序的抽象，但无法保证源程序是符合逻辑的**。而**语义分析的主要任务是对结构上正确的源程序进行上下文有关性质的审查**，如进行类型审查。
* **语义分析**过程分为**标注检查**以及**数据及控制流分析**两个步骤。
* **标注检查**步骤检查的内容诸如**变量使用前是否已被声明**、**变量与赋值之间的数据类型是否能够匹配**等。在标注检查中，还有一个重要的动作称为**常量折叠**，在代码里面定义“a=1+2”比起直接定义“a=3”，并不会增加程序运行期哪怕一个CPU指令的运算量。
* **数据及控制流分析**是对程序上下文逻辑更进一步的验证，它可以检查出诸如**程序局部变量在使用前是否有赋值**、**方法的每条路径是否都有返回值**、**是否所有的受查异常都被正确处理了**等问题。编译时期的数据及控制流分析与类加载时的数据及控制流分析的目的基本上是一致的，但校验范围有所区别，有一些校验项只有在编译期或运行期才能进行。
* 局部变量与字段（实例变量、类变量）是有区别的，它在常量池中没有CONSTANT\_Fieldref\_info符号引用，自然就没有访问标志（Access\_Flags）的信息，甚至可能连名称都不会保留下来，自然在Class文件中不可能知道一个局部变量是不是声明为final了。因此，**将局部变量声明为final，对运行期是没有影响的，变量的不变性仅仅由编译器在编译期间保障**。
* 虚拟机在运行时不支持变长参数、自动装箱/拆箱等语法，它们在编译阶段还原回简单的基础语法结构，这个过程称为**解语法糖**。
* **字节码生成**是Javac编译过程的最后一个阶段，不仅仅把前面各个步骤所生成的信息（语法树、符号表）转化成字节码写到磁盘中，编译器还进行了少量的代码添加和转换工作。例如，实例构造器<init>和类构造器<clinit>就是在这个阶段添加到语法树之中的，这两个构造器的产生过程实际上是一个代码收敛的过程，编译器会把语句块（对于实例构造器而言是“{}”块，对于类构造器而言是“static{}”块）、变量初始化（实例变量和类变量）、调用父类的实例构造器（仅仅是实例构造器，<clinit>方法中无须调用父类的<clinit>方法，虚拟机会自动保证父类构造器的执行）等操作收敛到<init>和<clinit>方法之中，并且保证一定是按**先执行父类的实例构造器，然后初始化变量，最后执行语句块**的顺序进行。
* Java语言中的泛型只在程序源码中存在，在编译后的字节码文件中，就已经替换为原来的原生类型（Raw Type，也称为裸类型）了，并且在相应的地方插入了强制转型代码，因此，对于运行期的Java语言来说，ArrayList<Integer>与ArrayList<String>即使同一个类，所以泛型技术实际上是Java语言的一颗语法糖，Java语言中的泛型实现方法称为**类型擦除**，基于这种方法实现的泛型称为伪泛型。
* 方法重载要求方法具备不同的特征签名，返回值并不包含在方法的特征签名之中，所以返回值不参与重载选择，但是在Class文件格式之中，只要描述符不是完全一致的两个方法就可以共存。也就是说，两个方法如果有相同的名称和特征签名，但返回值不同，那它们也可以合法地共存于一个Class文件中。
* Signature的作用是存储一个方法在**字节码层面的特征签名**，这个属性中保存的参数类型并不是原生类型，而是包括了**参数化类型**的信息。
* 在Java代码中的方法特征签名只包括了**方法名称**、**参数顺序**及**参数类型**，而在字节码中的特征签名还包括**方法返回值**及**受查异常表**。
* 擦除法所谓的擦除，仅仅是**对方法的Code属性中的字节码进行擦除**，实际上**元数据还是保留了泛型信息**，这也是**我们能通过反射手段取得参数化类型**的根本依据。
* **包装类的“==”运算在不遇到算术运算的情况下不会自动拆箱**，以及它们**equals()方法不处理数据转型的关系**。
* 编译器并非一个个地编译Java文件，而是**将所有编译单元的语法树顶级节点输入到待处理列表后再进行编译**，因此各个文件之间能够互相提供符号信息。