[1. java内存管理](#_Toc14077)

[1.1. 运行时内存区域](#_Toc29302)

[1.1.1. 程序计数器：](#_Toc29937)

[1.1.2. java虚拟机栈：](#_Toc10942)

[1.1.3. 本地方法栈](#_Toc32206)

[1.1.4. java堆](#_Toc25252)

[1.1.5. 方法区](#_Toc10172)

[1.1.6. 运行时常量池](#_Toc5274)

[1.1.7. 直接内存](#_Toc5314)

[1.2. java对象](#_Toc3703)

[1.2.1. 对象创建](#_Toc1918)

[1.2.2. 对象布局](#_Toc11234)

[1.2.3. 对象访问](#_Toc2685)

[2. 垃圾收集与内存分配](#_Toc23755)

[2.1. 判断对象是否存活](#_Toc26417)

[2.1.1. 引用计数算法：](#_Toc8440)

[2.1.2. 可达性分析算法：](#_Toc4586)

[2.2. 引用的性质](#_Toc10417)

[2.3. 垃圾收集算法](#_Toc6710)

[2.3.1. 标记-清除算法](#_Toc3084)

[2.3.2. 复制算法](#_Toc27533)

[2.3.3. 标记整理算法](#_Toc21871)

[2.3.4. 分代收集算法](#_Toc18502)

[2.4. HotSpot的垃圾收集算法实现](#_Toc16701)

[2.5. 垃圾收集器—（收集算法的实现）](#_Toc30997)

[2.5.1. Serial收集器](#_Toc25544)

[2.5.2. ParNew收集器](#_Toc15787)

[2.5.3. Parallel Scavenge收集器](#_Toc12288)

[2.5.4. Serial Old收集器](#_Toc12081)

[2.5.5. Parallel Old收集器](#_Toc27773)

[2.5.6. CMS（Concurrent Mark Sweep）收集器](#_Toc2992)

[2.5.7. G1收集器](#_Toc20934)

[2.6. 内存分配和回收策略](#_Toc22949)

[2.6.1. 对象优先在Eden分配](#_Toc9235)

[2.6.2. 大对象直接进入老年代](#_Toc3414)

[2.6.3. 长期存活的对象进入老年代](#_Toc27609)

[2.6.4. 动态对象年龄判定](#_Toc26013)

[2.6.5. 空间分配担保](#_Toc19415)

[3. Class类文件结构与字节码指令](#_Toc9288)

[3.1. Class类文件结构](#_Toc10375)

[3.2. 字节码指令](#_Toc9793)

[4. 类加载机制](#_Toc13344)

[4.1. 类加载的时机](#_Toc16913)

[4.2. 类加载的过程](#_Toc19331)

[4.2.1. 加载](#_Toc18313)

[4.2.2. 验证](#_Toc13268)

[4.2.3. 准备](#_Toc4493)

[4.2.4. 解析](#_Toc25427)

[4.2.5. 初始化](#_Toc14637)

[4.3. 类加载器](#_Toc21661)

[4.3.1. 类与类加载器](#_Toc11071)

[4.3.2. 双亲委派模型](#_Toc27827)

[5. 虚拟机的字节码指令执行引擎](#_Toc22461)

[5.1. 运行时栈帧结构](#_Toc18975)

[5.1.1. 局部变量表](#_Toc21347)

[5.1.2. 操作数栈](#_Toc25994)

[5.1.3. 动态连接](#_Toc30640)

[5.1.4. 方法返回地址](#_Toc12318)

[5.1.5. 附加信息](#_Toc8227)

[5.2. 方法调用](#_Toc4698)

[5.2.1. 解析](#_Toc2642)

[5.2.2. 分派](#_Toc13893)

[5.2.3. 动态类型语言支持](#_Toc14280)

[5.3. 基于栈的字节码解释执行引擎](#_Toc8280)

**JVM概述**

# java内存管理

## 运行时内存区域

### 程序计数器：

可以比作是当前线程所执行的字节码的行号指示器，字节码解释器通过这个计数器选取需要执行的字节码。是 唯一一个不会出现内存错误的区域。

### java虚拟机栈：

是方法执行的栈帧，用于存储局部变量表、操作数栈、方法出口等信息。也是线程私有的。

局部变量表存储了编译期可知的各种数据类型，所以局部变量表的内存空间在编译期就完成了分配，在运行时不会改变。

该内存区域会出现两种异常情况：

线程请求的栈深度大于虚拟机所允许的深度会抛出StackOverflowError

如果虚拟机栈允许动态扩展，当扩展到无法申请足够的内存时，会抛出OutOfMemoryError

### 本地方法栈

与java虚拟机栈作用相似，区别是为虚拟机使用到的Native方法服务。

HotSpot虚拟机的实现将该两个区域合二为一。

### java堆

线程共享的内存区域，由jvm启动时创建。用于存放对象和数组。

垃圾收集器管理的主要区域，可以细分为：

新生代：Eden、From Survivor、To Survivor。

老年代。

该区域无法再分配内存时，会抛出OutOfMemoryError。

### 方法区

线程共享的内存区域。用于存储类信息、常量、JIT编译后的代码等。也叫Non-Heap。

HotSpot使用永久代（Permanent Generation）实现方法区。

该区域无法再分配内存时，会抛出OutOfMemoryError。

### 运行时常量池

是方法区的一部分，用于存放编译期生成的字面量和符号引用。

具有动态性，String类的intern()可以在运行时把String对象放到常量池中。

会抛出OutOfMemoryError。

关于String数据类型：

通过字面量定义的String是存放在运行时常量池中。

通过new关键字创建的String是存放在堆中。调用intern()方法后，会存放到运行时常量池中，并返回指向常量池的引用。

通过+运算符定义的String是用StringBuilder对象创建的。

通过final修饰的String进行+运算时不会产生StringBuilder对象，而是直接被常量池中的存在的String替换掉。

### 直接内存

不属于jvm运行时数据区域，NIO使用Native函数库直接分配堆外内存。

会抛出OutOfMemoryError。

## java对象

### 对象创建

### 对象布局

### 对象访问

# 垃圾收集与内存分配

## 判断对象是否存活

### 引用计数算法：

给对象添加一个引用计数器。当对象被引用时，计数器值加1，对象引用失效时，计数器值减1。

有弊端，无法解决循环引用问题。

### 可达性分析算法：

HotSpot使用的对象存活判定算法。

通过一系列称为"GC Root"的对象作为起始点，开始向下搜索。经过的路径称为引用链，当一个对象到GC Root没有引用链时，说明这个对象不可用。

可作为GC Root的对象包括：

虚拟机栈中的引用类型变量

方法区中类静态属性的引用类型变量

方法区中运行时常量池的引用类型变量

Native方法

GC Root可以理解为一组必须活跃的引用。

GC主要管理的时堆内存中的对象。而虚拟机栈、本地方法栈、方法区这些区域不被GC所管理，所以这些区域里的对象被当做GC Root。

GC会收集那些不是GC Root而且不被GC Root引用的对象。

## 引用的性质

强引用：是程序中一般使用的引用类型。引用的对象是可达的，不会被回收。

软引用：一个对象持有软引用，当堆内存不足时，就会被回收。使用java.lang.ref.SoftReference类实现。

弱引用：在系统GC时，只要发现弱引用，不管堆内存使用情况如何，都会将对象进行回收。使用java.lang.ref.WeakReference类实现。

虚引用：一个持有虚引用的对象，和没有引用几乎一样，随时都可能被垃圾收集器回收。

finalize()方法：

1，当对象进行可达性分析后，没有与GC Root的引用链了，该对象会被第一次标记并判断是否覆盖了finalize方法，如果是会被放到F-Queue中。

2，finalize方法是对象避免被收集的最后一次机会，稍后GC会对F-Queue中的对象进行第二次标记。如果对象在finalize方法中重新与引用链上的对象建立了引用关系，比如用this赋值给了一个类的静态变量，第二次标记时会被移除"即将回收"的集合。如果没有建立引用关系，对象则会被回收 。

## 垃圾收集算法

### 标记-清除算法

实现：分两个阶段，首先标记出需要回收的对象，然后统一将所有标记出的对象进行回收。

有两个缺点：一是标记和清除都需要时间，导致回收效率不高。二是内存回收后产生不连续的内存碎片，当有大对象需要内存分配时，无法找到连续的内存空间，从而要提前触发一次垃圾回收动作。

后面的算法都是基于此算法并对其改进而产生的。

### 复制算法

实现：将内存分为相同大小的两部分，一部分用于对象分配，另一部分用于保留内存空间，当用于分配的内存空间用完时，将活着的对象复制到另一个内存空间内，然后对使用过的内存空间进行完整的一次清理。这样就不会产生内存碎片。

缺点：间接的将可用内存压缩为了原来的一半。

新生代的对象98%都是"朝生夕死"，所以新生代的垃圾收集算法都是用的复制算法。

新生代内存大小不是等分的，而是分为1个Eden区域2个Survivor区域。分配内存时只使用Eden区和1块Survivor区，当垃圾回收时，将活着的对象复制到另1块Survivor区，然后直接清理掉Eden区和用过的Survivor区。

HotSpot的Eden区和Survivor区默认比例是8：1，即新生代中可用内存空间为整个新生代的90%，另一块保留的Survivor区占10%。当保留的1块Survivor区域不够时，需要依赖老年代做内存担保。

### 标记整理算法

用于老年代的收集算法，因为老年代的对象存活率高，没有额外的内存空间做分配担保。

实现：分三个步骤，首先标记可回收的对象，标记过程和"标记-清除"一样，后续将存活的对象都向一端移动，然后将端边界以外的内存直接清理掉。

### 分代收集算法

主流虚拟机的垃圾收集实现。将前面的各种算法进行分代运用。

## HotSpot的垃圾收集算法实现

1，在枚举GC Roots中的引用链时，使用一组称为OopMap的数据结构，存储引用链，例如常量、静态属性或本地变量表的引用位置。GC扫描时可以直接获取引用链的信息。

2，HotSpot不会为每条指令生成OopMap，只在称为"安全点（SafePoint）"的位置记录这些信息。可以作为安全点的指令有方法调用、循环跳转、异常跳转等。

3，SafePoint机制保证了程序执行时，GC可以安全的进行。但是在程序"不执行"，即线程Sleep或Blocked状态时，HotSpot使用"安全区（Safe Region）"，让GC可以安全的进入并执行。

## 垃圾收集器—（收集算法的实现）

### Serial收集器

新生代的收集器。

使用单线程，并停止其他工作线程，进行垃圾收集。实现的是复制算法。

Serial收集器的工作方式简单高效，没有线程切换的开销。是JVM Client模式下的默认新生代收集器。

### ParNew收集器

新生代的收集器。是Serial收集器的多线程版本。

使用多线程，与老年代的CMS收集器配合使用。

使用-XX:+UseConcMarkSweepGC会默认指定，使用-XX:+UserPaNewGC可以强制指定。

在单CPU环境下没有Serial收集器效率高，在多CPU环境下效率很高。

使用-XX:ParallelGCThreads参数限制其启动的线程数。

### Parallel Scavenge收集器

新生代收集器。

和ParNew收集器相似，使用多线程，实现了复制算法。

可以控制程序运行的吞吐量。吞吐量=用户代码运行时间/(用户代码运行时间+垃圾收集时间)。

使用-XX:MaxGCPauseMillis参数控制垃圾收集停顿时间。值可以是大于0的毫秒数。

使用-XX:GCTimeRatio参数控制吞吐量大小。值是大于0，小于100的整数n。可以理解为垃圾收集占总时间的比率，相当于吞吐量的倒数，公式为1/(1+n)。

假设 GCTimeRatio 的值为 n，那么系统将花费不超过 1/(1+n) 的时间用于垃圾收集。

使用-XX:+UseAdaptiveSizePolicy开启GC自适应的调节策略。这个是与ParNew收集器的重要策略。前提是设置好了上面两个参数。

GC停顿时间短，可以提供良好的响应速度，适合与用户交互的程序。

高吞吐量可以提高CPU利用率、快速执行运算任务，适合后台运算的程序。

### Serial Old收集器

老年代收集器，Serial收集器的老年代版本。

使用单线程，实现了"标记-整理"算法。

主要用于JVM的Client模式。在Server模式下，一是可以和Parallel Scavenge配合使用，二是作为CMS收集器的后备预案，在并发收集发生Concurrent Mode Failure时使用。

### Parallel Old收集器

老年代收集器，Parallel Scavenge收集器的老年代版本。

使用多线程，实现了"标记-整理"算法。

JDK1.6之后提供的，可以与Parallel Scavenge收集器配合使用。适用于高吞吐量及CPU资源敏感的场景。

### CMS（Concurrent Mark Sweep）收集器

老年代收集器，目标是获取最短回收停顿时间。

使用单线程和多线程结合，实现了"标记-清除"算法。

运行分为4个步骤：

初始标记：使用单线程，标记与GC Root直接关联的对象。会Stop The World。

并发标记：进行GC Root Tracing。与用户线程并发执行。

重新标记：修正并发标记期间由于程序运行而产生变动的对象的标记记录。会Stop The World。

并发清除：与用户线程并发执行。

缺点：

1，对CPU资源非常敏感。在并发阶段会占用CPU资源，导致总吞吐量降低。

2，无法处理浮动垃圾。浮动垃圾是在CMS并发清理过程中由用户程序新产生的垃圾，它是在标记过程之后的垃圾，所以CMS当次不能回收他们。并发收集时，因为用户线程也在执行，所以CMS收集器不会等老年代完全填满后才进行收集，而是预留一定的内存空间供收集线程运作使用。

使用-XX:CMSInitialtingOccupancyFraction参数调整触发内存回收的内存的百分比。

设置的太高会导致预留内存无法满足并发收集线程的需要，导致出现"Concurrent Mode Failure"。出现"Concurrent Mode Failure"后，JVM会启用Serial Old收集器来收集老年代的内存。

3，基于"标记-清除"的算法会产生大量的内存碎片，导致分配大对象时，找不到连续内存空间而提前触发一次Full GC。

使用-XX:+UseCMSCompactAtFullCollection参数可以让CMS在将要Full GC时开启内存碎片的合并整理过程，因为这个过程时无法并发的，所以后面进行Full GC时停顿时间会变长。

使用-XX:CMSFullGCsBeforeCompaction参数可以设置带压缩的Full GC的频率，即执行n次不压缩的Full GC后，进行一次带压缩的Full GC。默认为0，表示每次Full GC都进行碎片整理（压缩）。

### G1收集器

新生代和老年代的收集器，目标是替换掉jdk1.5中的CMS收集器。

特点：

1，并行和并发。利用多CPU的硬件条件，需要停顿执行GC的线程，通过并发的方式让用户程序继续执行。

2，分代收集。G1收集器不需要和其他收集器配合使用，仍然采用不同的方式处理新创建的对象、存活一段时间的对象、熬过多次GC的对象。

3，空间整理。整体采用"标记-整理"算法，局部（两个Region之间）采用"复制"算法。所以不会产生内存空间碎片，分配大对象时也不会因为无法找到连续内存空间而提前触发GC。

4，可预测的停顿。G1除了追求低停顿外，还能建立可预测的停顿时间模型。因为它可以有计划的避免在java堆中全区域的垃圾收集。

G1收集器把java堆分成多个大小独立的Region，所以新生代和老年代就是一部分Region的集合。

运行步骤：

1，初始标记

2，并发标记

3，最终标记

4，筛选回收

## 内存分配和回收策略

自动内存管理归结为自动化了解决了两个问题：给对象分配内存和回收分配给对象的内存。

### 对象优先在Eden分配

大多数情况下，对象在Eden区分配。Eden区内存空间不足时，JVM会发起一次Minor GC—试图把Eden区的对象放到Survivor区。Survivor区内存空间不足时，通过分配担保机制将对象转移到老年代中。

### 大对象直接进入老年代

参数-XX:PretenureSizeThreshold可以设置大于这个值的大对象直接分配到老年代。该参数只对Serial和ParNew收集器有效。

### 长期存活的对象进入老年代

如果对象在Eden区被分配，并经过第一次Minor GC后仍然存活，并且被Survivor容纳后，会被放到Survivor中。并且对象年龄被设为1。对象在Survivor区熬过一次Minor GC后，年龄就会加1，当它的年龄增加到一定程度时（默认15），就会被晋升到老年代中。

参数-XX:MaxTenuringThreshold可以设置对象晋升老年代的年龄阈值。

### 动态对象年龄判定

JVM并不是一定根据MaxTenuringThreshold的值将对象晋升到老年代的。如果Survivor区的所有年龄相同的对象的大小总和大于Survivor区域的一半，那么所有大于或等于该年龄的对象也会在GC时直接进入到老年代。

### 空间分配担保

在Minor GC之前，jvm会检查老年代的最大连续可用内存空间是否大于新生代对象的总内存空间，如果条件成立 ，可以确保Minor GC是安全的。如果条件不成立，jvm会查看HandlePromotionFailure设置的值是否允许担保失败，如果允许，会继续检查老年代的最大连续可用内存空间是否大于历次晋升到老年代对象的的平均大小，如果大于，则尝试进行一次Minor GC。如果小于或者HandlePromotionFailure设置不允许担保失败，则会进行一次Full GC。

# Class类文件结构与字节码指令

## Class类文件结构

字节码文件是由编译器编译后产生的，与平台无关的二进制文件。虚拟机可以载入和执行字节码文件，实现了程序的"一次编写，到处运行"。

java虚拟机能够接受的字节码文件就是Class文件。

Class文件包含了java虚拟机指令集和符号表以及其他信息。java语言中的各种变量、关键字和运算符号的语义最终都是由字节码指令组合完成的。

Class文件是以8位字节为基础单位的二进制文件。

Class文件中有两种数据结构：无符号数和表。

无符号数由若干个字节组成，用于描述数字、索引引用、数量值或者字符串值。

表通常是由多个无符号数或者其他表作为数据项而构成的复合数据类型。都以"\_info"结尾。整个Class文件本质上就是一张表。

常量池（cp\_info）是Class文件中的第一个表类型的数据项目。常量池中主要存放两类常量：字面量和符号引用。

字面量接近于java语言层面的概念，例如String、final声明的常量等。

符号引用是编译原理方面的概念，具体常量池中就是：1，类和接口的全限定名；2，字段的名称和限定符；3，方法的名称和限定符。

Class文件中不保存方法、字段的内存布局信息，jvm是通过常量池中的符号引用，在类创建时或运行时解析、翻译到具体的内存地址中的。

属性表（attribute\_info）集合是在Class文件、字段表（field\_info）、方法表（method\_info）中都携带的信息。

Code属性：

方法体中的代码经过javac编译器处理后，最终变为字节码指令存储在Code属性 中。

Args\_size默认等于1是因为方法的局部变量表中至少会存在一个指向当前对象实例的局部变量this。如果方法声明为static，则等于0。

LineNumberTable属性：

用于描述java源代码和字节码的行号。javac的-g:none和-g:lines选项来取消或要求生成此项信息。

LocalVariableTable属性：

用于描述栈帧中局部变量表和java源代码中定义的变量之间的关系。javac的-g:none或-g:vars选项来取消或要求生成生成此项信息。

## 字节码指令

# 类加载机制

java语言里，类型的加载、连接和初始化过程，都是在运行期间完成的（解释型）。这是和那些编译期进行连接工作的语言的不同之处（编译型）。

java语言里，动态扩展的语言特性就是依赖运行期动态加载和动态连接这个特点实现的。例如Applet、JSP、OSGI等技术。

## 类加载的时机

类的生命周期包括：加载、验证、准备、解析、初始化、使用、卸载。类的加载过程必须按照这个顺序按部就班地开始。

解析阶段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（动态绑定）。

加载的时机jvm没有规定，但是初始化的时机，jvm则严格规定了有且只有5种情况必须对类进行"初始化"：

1，遇到new、getstatic、putstatic、invokestatic这个4个字节码指令时。分别对应java代码中的使用new实例化对象、获取类的静态字段（被final修饰的常量除外）、设置类的静态字段、执行类的静态方法。

2，使用java.lang.reflect包的方法对类进行反射调用的时候，如果类没有被初始化过，要触发类的初始化。

3，初始化一个类时，如果其父类还没初始化，要先触发其父类的初始化。

4，当jvm启动时，用户需要指定一个执行的主类—包含main方法的类。jvm要先初始化主类。

5，java7中，如果一个java.lang.invoke.MethodHandle实例最后的解析结果为REF\_getStatic、REF\_putStatic、REF\_invokeStatic的方法句柄，并且该方法句柄对应的类没有进行初始化，需要先触发对应类的初始化。

这五种情况称为对一个类进行主动引用。其他情况都不会触发初始化，称为被动引用。

引用常量不会初始化类的原因：在编译阶段通过常量传播优化将常量值存放到类的常量池中，对常量的引用就是类对自身常量池的引用。

## 类加载的过程

### 加载

此阶段，jvm要做3件事情：

1）根据类的全限定名获取定义了此类的二进制字节流（不仅是Class文件）。

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此类的java.lang.class对象，作为方法区这个类的各种数据的访问入口。

此阶段可以由开发者自定义类加载器—重写一个类加载器的loadClass()方法，去控制字节流的获取方式。

生成Class对象不是在堆中，而是在方法区中。

### 验证

此阶段的目的是保证Class文件的字节流中包含的信息符合jvm的要求，并且不会危害jvm自身的安全。

包括符号引用验证。

### 准备

此阶段是为类变量分配内存空间并设置类变量初始值。

这些类变量（static修饰的变量）使用的内存空间都在方法区中分配。不要和实例变量混淆。实例变量是在对象实例化时和对象一起分配在堆中。

### 解析

此阶段是将常量池中的符号引用替换为直接引用的过程。

### 初始化

此阶段会初始化类变量和其他资源，也可以当做是执行<clinit>()方法的过程。

<clinit>()方法的特点：

1）<clinit>()方法是编译器把类中的类变量赋值动作和静态语句块中的语句合并而产生的。

2）<clinit>()与类的构造函数（实例构造器<init>()）不同。<clinit>()方法不需要显示的调用父类构造器，jvm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。

3）父类的<clinit>()方法先执行，也意味着父类的static代码块会优先于子类的变量赋值操作。

4）<clinit>方法对于类和接口来说并不是一定会生成的。前提是类中没有static代码块也没有变量赋值操作。

5）jvm会保证<clinit>方法在多线程环境下，会被正确的加锁、同步。

## 类加载器

虚拟机设计者把加载阶段的"获取二进制字节流"的动作放到了jvm外部去实现，这个动作的代码模块称为"类加载器"

### 类与类加载器

对于任意一个类，都需要由加载它的类加载器和类本身一起确定其在jvm中的唯一性。

### 双亲委派模型

以jvm的角度来讲，在jvm中只有两种类加载器：

一是启动类加载器—Bootstrap ClassLoader，它由c++实现，是jvm的一部分。

二是所有其他的类加载器，这些类由java语言实现，独立于jvm，并且全继承自抽象类java.lang.ClassLoader。

以java开发人员的角度来讲，java程序会使用到以下3种系统提供的类加载器：

1）启动类加载器—Bootstrap ClassLoader

负责将<JAVA\_HOME>\lib目录、-Xbootclasspath参数指定的路径中的类库加载到虚拟机内存中。

此加载器不能直接被java程序引用。

2）扩展类加载器—Extension ClassLoader

负责加载<JAVA\_HOME>\lib\ext目录、java.ext.dirs系统变量指定路径中的类库。

由sun.misc.Launcher$ExtClassLoader实现，java程序可以直接使用此加载器。

3）应用程序类加载器—Application ClassLoader

负责加载用户类路径（ClassPath）上所指定的类库。

由sun.misc.Launcher$AppClassLoader实现，java程序可以直接使用此加载器。

# 虚拟机的字节码指令执行引擎

字节码指令执行引擎是具体的虚拟机实现的，属于虚拟机的一部分。

## 运行时栈帧结构

栈帧是支持虚拟机进行方法调用和方法执行的数据结构，是运行时数据区中的虚拟机栈的栈元素。每一个栈帧都包含了局部变量表、操作数栈、动态连接、方法返回地址等信息。

执行引擎的所有字节码指令都只针对当前的栈帧进行操作。

### 局部变量表

用于存储方法参数和方法内部定义的局部参数。

### 操作数栈

用于字节码指令往操作数栈中写入和提取内容，也就是出栈/入栈操作。

### 动态连接

每个栈帧中都包含一个指向运行时常量池中的该栈帧所属方法的引用，用于支持方法调用过程中的动态连接。

### 方法返回地址

第一种方法退出方式是执行引擎遇到方法返回的字节码指令，此时可能会有返回值传递给上层方法调用者。

第二种方法退出方式是方法执行过程中遇到了异常，并且异常没有在方法内处理。

无论哪种方式，方法退出之后，都需要返回到方法被调用的地方。

一般情况下，正常方法退出时，调用者的程序计数器值可以作为返回地址，战阵中可能会保存这个计数器值。而方法异常退出时，返回地址是通过异常处理器表来确定的，栈帧中一般不保存这个信息。

### 附加信息

不再详述

## 方法调用

方法调用不等同于方法执行。

方法调用中的目标方法，在Class文件中存储的只是符号引用，而不是方法在实际运行时内存布局中的入口地址（直接引用），需要在类加载期间，甚至到运行期间才能确定目标方法的直接引用。这个特性给java带来了动态扩展能力。

### 解析

在类加载的解析阶段，会将一部分符号引用转化为直接引用。

解析成立的前提是这些目标方法“编译期可知，运行期不可变”。

符合条件的方法包括：静态方法、私有方法、实例构造器、父类方法4类。这些方法由invokestatic和invokespecial指令调用。这种调用称为“解析调用”。

其他指令的调用方式称为“分派调用”。

### 分派

1. 静态分派

典型应用是方法重载。在编译阶段，javac会根据参数的静态类型决定使用哪个重载版本。

静态分派的动作不是由虚拟机来执行的。

1. 动态分派

典型应用是方法重写。

1. 单分派或多分派
2. 虚拟机动态分派的实现

### 动态类型语言支持

## 基于栈的字节码解释执行引擎