**Java虚拟机**

目录

[自动内存管理机制 3](#_Toc35529801)

[概述 3](#_Toc35529802)

[运行时数据区域 3](#_Toc35529803)

[程序计数器（线程私有） 5](#_Toc35529804)

[Java虚拟机栈（线程私有） 6](#_Toc35529805)

[本地方法栈（私有） 9](#_Toc35529806)

[Java堆（全局共享） 9](#_Toc35529807)

[方法区（全局共享） 10](#_Toc35529808)

[直接内存 11](#_Toc35529809)

[HotSpot虚拟机对象探秘 11](#_Toc35529810)

[对象的创建 11](#_Toc35529811)

[1.判断对象对应的类是否加载、链接、初始化 12](#_Toc35529812)

[2.为对象分配内存 13](#_Toc35529813)

[3.处理并发安全问题 14](#_Toc35529814)

[4.初始化分配到的内存空间 14](#_Toc35529815)

[5.设置对象的对象头 15](#_Toc35529816)

[6.执行init方法进行初始化 15](#_Toc35529817)

[对象的布局 15](#_Toc35529818)

[对象头 16](#_Toc35529819)

[实例数据 16](#_Toc35529820)

[对齐填充 17](#_Toc35529821)

[对象的访问定位 17](#_Toc35529822)

[直接指针 18](#_Toc35529823)

[实战：OutOfMemoryError异常 19](#_Toc35529824)

[概念 19](#_Toc35529825)

[Java堆溢出 19](#_Toc35529826)

[虚拟机栈和本地方法栈溢出 20](#_Toc35529827)

[垃圾回收器与内存分配策略 24](#_Toc35529828)

[概述 24](#_Toc35529829)

[对象已死吗 24](#_Toc35529830)

[引用计数法 24](#_Toc35529831)

[可达性分析算法 25](#_Toc35529832)

[再谈引用 27](#_Toc35529833)

[回收方法区 28](#_Toc35529834)

[垃圾收集算法 29](#_Toc35529835)

[标记-清除算法 29](#_Toc35529836)

[复制算法 29](#_Toc35529837)

[标记-整理算法 30](#_Toc35529838)

[分代收集算法 31](#_Toc35529839)

[HotSpot的算法实现 34](#_Toc35529840)

[枚举根节点 34](#_Toc35529841)

[安全点 34](#_Toc35529842)

[安全区域 35](#_Toc35529843)

[垃圾收集器 35](#_Toc35529844)

[多线程垃圾回收器（吞吐量优先） 38](#_Toc35529845)

[内存分配与回收策略 43](#_Toc35529846)

[对象优先在 Eden 区分配 44](#_Toc35529847)

[大对象直接进入老年代 44](#_Toc35529848)

[长期存活对象将进入老年代 44](#_Toc35529849)

[动态对象年龄判定 45](#_Toc35529850)

[空间分配担保 45](#_Toc35529851)

[类文件结构 45](#_Toc35529852)

[概述 45](#_Toc35529853)

[Class类文件的结构 46](#_Toc35529854)

[魔数与 Class 文件的版本 47](#_Toc35529855)

[常量池 47](#_Toc35529856)

[访问标志 48](#_Toc35529857)

[字段表集合 49](#_Toc35529858)

[属性表集合 50](#_Toc35529859)

[虚拟机类加载机制 50](#_Toc35529860)

[概述 50](#_Toc35529861)

[类加载时机 51](#_Toc35529862)

[类加载过程 52](#_Toc35529863)

[加载 52](#_Toc35529864)

[验证 52](#_Toc35529865)

[准备 53](#_Toc35529866)

[解析 53](#_Toc35529867)

[初始化 53](#_Toc35529868)

[类加载器 54](#_Toc35529869)

[类与类加载器 54](#_Toc35529870)

[双亲委派模型 54](#_Toc35529871)

[类加载器的关系 55](#_Toc35529872)

[类加载器的隔离问题 56](#_Toc35529873)

[字节码执行引擎 57](#_Toc35529874)

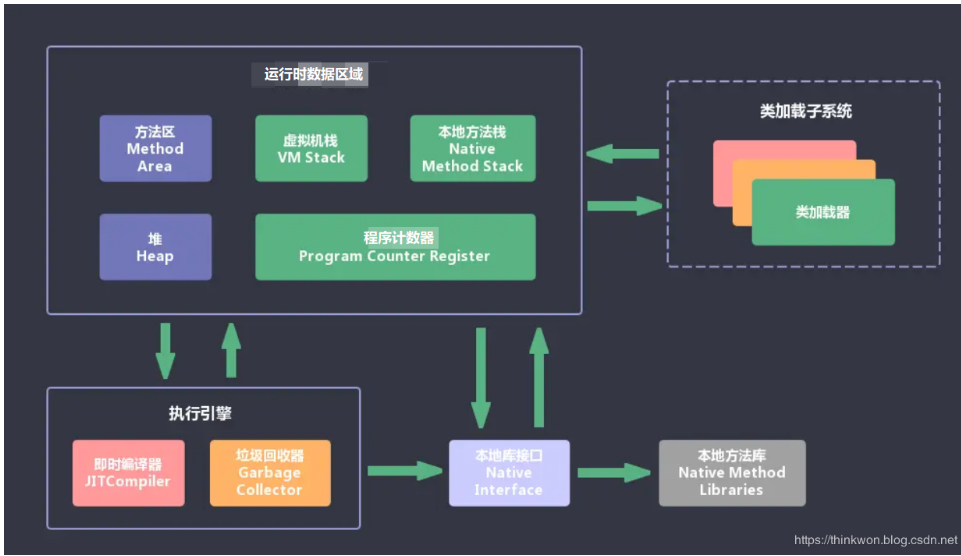
# 自动内存管理机制

## 概述

对于从事C、C++程序开发的开发人员来说，在内存管理领域，他们既是拥有最高权力的“皇帝”又是从事最基础工作的“劳动人民”——既拥有每一个对象的“所有权”，又担负着每一个对象生命开始到终结的维护责任。

对于Java程序员来说，在虚拟机自动内存管理机制的帮助下，不再需要为每一个new操作去写配对的delete/free代码，不容易出现内存泄漏和内存溢出问题，由虚拟机管理内存这一切看起来都很美好。不过，也正是因为Java程序员把内存控制的权力交给了Java虚拟机，一旦出现内存泄漏和溢出方面的问题，如果不了解虚拟机是怎样使用内存的，那么排查错误将会成为一项异常艰难的工作。

## 运行时数据区域



JVM包含两个子系统和两个组件，两个子系统为Class loader(类装载)、Execution engine(执行引擎)；两个组件为Runtime data area(运行时数据区)、Native Interface(本地接口)。

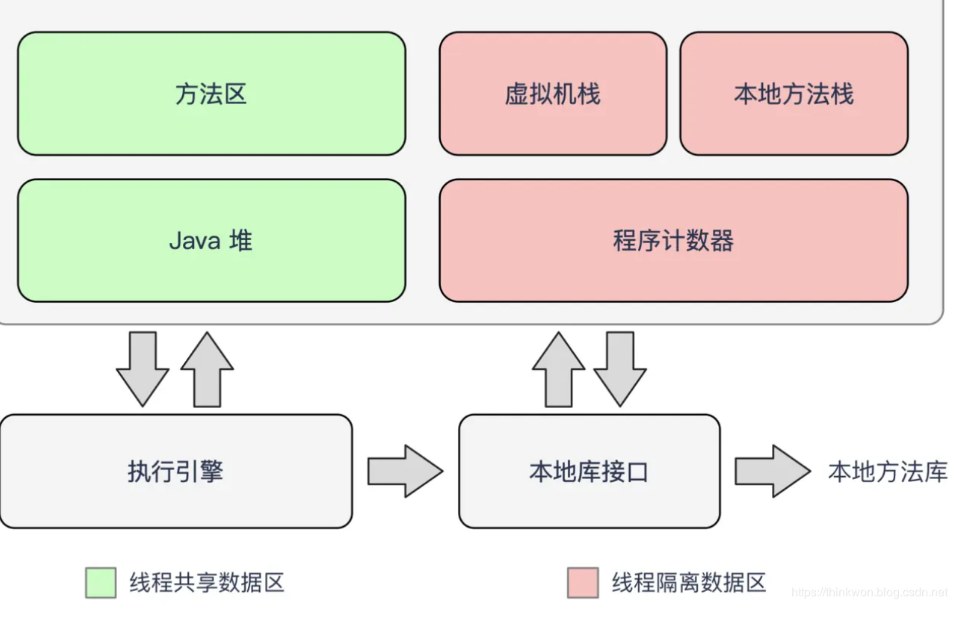
Class loader(类装载)：根据给定的全限定名类名(如：java.lang.Object)来装载class文件到Runtime data area中的method area。

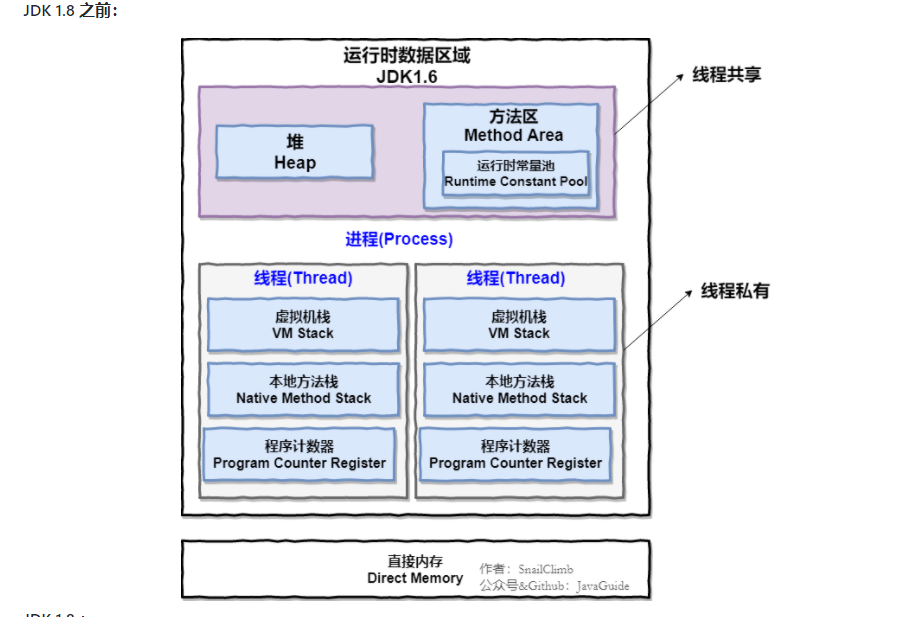
Execution engine（执行引擎）：执行classes中的指令。

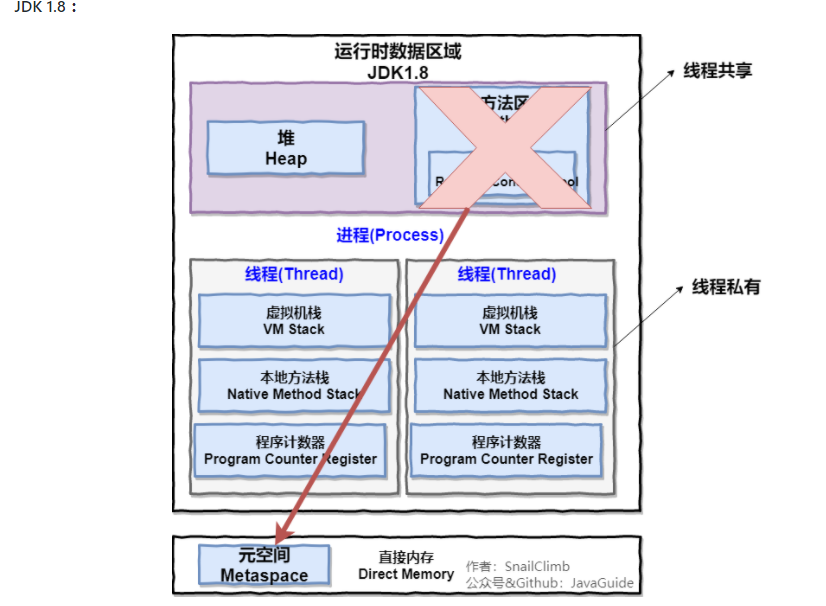
Native Interface(本地接口)：与native libraries交互，是其它编程语言交互的接口。

Runtime data area(运行时数据区域)：这就是我们常说的JVM的内存。

Java 虚拟机在执行 Java 程序的过程中会把它所管理的内存区域划分为若干个不同的数据区域。这些区域都有各自的用途，以及创建和销毁的时间，有些区域随着虚拟机进程的启动而存在，有些区域则是依赖线程的启动和结束而建立和销毁。Java 虚拟机所管理的内存被划分为如下几个区域：







### 程序计数器（线程私有）

1. 程序计数器是一块较小的内存区域，可以看做是当前线程所执行的字节码的行号指示器。在虚拟机的概念模型里，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，分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖这个计数器来完成。「属于线程私有的内存区域」
2. 在多线程的情况下，程序计数器用于记录当前线程执行的位置，从而当线程被切换回来的时候能够知道该线程上次运行到哪儿了。

附加：



### Java虚拟机栈（线程私有）

线程私有内存空间，它的生命周期和线程相同。线程执行期间，每个方法被执行时，都会创建一个栈帧（Stack Frame）用于存储局部变量表、操作栈、动态链接、方法出口等信息。每个方法从被调用到执行完成的过程，就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。「属于线程私有的内存区域」

注意：下面的内容为附加内容，对Java虚拟机栈进行详细说明，感兴趣的小伙伴可以有针对性的阅读

下面依次解释栈帧里的四种组成元素的具体结构和功能：

#### 局部变量表

局部变量表局部变量表是 Java 虚拟机栈的一部分，是一组变量值的存储空间，主要存放了编译器可知的各种数据类型（boolean、byte、char、short、int、float、long、double）、对象引用（reference 类型，它不同于对象本身，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）。用于存储方法参数和局部变量。 在 Class 文件的方法表的 Code 属性的 max\_locals 指定了该方法所需局部变量表的最大容量。

局部变量表在编译期间分配内存空间，可以存放编译期的各种变量类型：

基本数据类型 ：boolean, byte, char, short, int, float, long, double等8种；

对象引用类型 ：reference，指向对象起始地址的引用指针；不等同于对象本身，根据不同的虚拟机实现，它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者其他与此对象相关的位置

返回地址类型 ：returnAddress，返回地址的类型。指向了一条字节码指令的地址

变量槽(Variable Slot)：

变量槽是局部变量表的最小单位，规定大小为32位。对于64位的long和double变量而言，虚拟机会为其分配两个连续的Slot空间。

#### 操作数栈

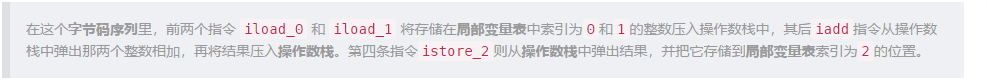
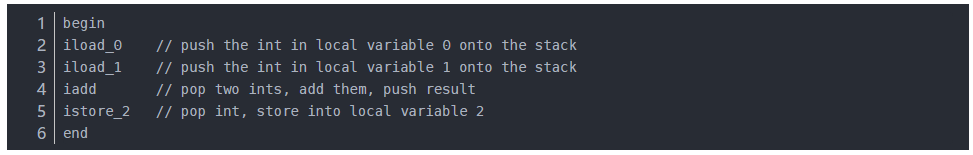
操作数栈（Operand Stack）也常称为操作栈，是一个后入先出栈。在 Class 文件的 Code 属性的 max\_stacks 指定了执行过程中最大的栈深度。Java虚拟机的解释执行引擎被称为基于栈的执行引擎 ，其中所指的栈就是指－操作数栈。

和局部变量表一样，操作数栈也是一个以32字长为单位的数组。

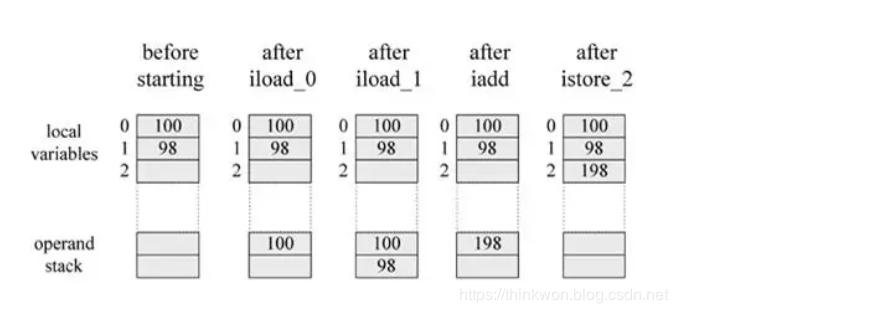
虚拟机在操作数栈中可存储的数据类型：int、long、float、double、reference和returnType等类型 (对于byte、short以及char类型的值在压入到操作数栈之前，也会被转换为int)。

和局部变量表不同的是，它不是通过索引来访问，而是通过标准的栈操作 — 压栈和出栈来访问。比如，如果某个指令把一个值压入到操作数栈中，稍后另一个指令就可以弹出这个值来使用。

虚拟机把操作数栈作为它的工作区——大多数指令都要从这里弹出数据，执行运算，然后把结果压回操作数栈。



下图详细表述了这个过程中局部变量表和操作数栈的状态变化(图中没有使用的局部变量表和操作数栈区域以空白表示)。



#### 动态链接

每个栈帧都包含一个指向运行时常量池中所属的方法引用，持有这个引用是为了支持方法调用过程中的动态链接。

Class文件的常量池中存在有大量的符号引用，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为参数。这些符号引用：

静态解析：一部分会在类加载阶段或第一次使用的时候转化为直接引用（如final、static域等），称为静态解析，

动态解析：另一部分将在每一次的运行期间转化为直接引用，称为动态链接。

方法返回地址

当一个方法开始执行以后，只有两种方法可以退出当前方法：

正常返回：当执行遇到返回指令，会将返回值传递给上层的方法调用者，这种退出的方式称为正常完成出口(Normal Method Invocation Completion)，一般来说，调用者的PC计数器可以作为返回地址。

异常返回：当执行遇到异常，并且当前方法体内没有得到处理，就会导致方法退出，此时是没有返回值的，称为异常完成出口(Abrupt Method Invocation Completion)，返回地址要通过异常处理器表来确定。

当一个方法返回时，可能依次进行以下3个操作：

恢复上层方法的局部变量表和操作数栈。

把返回值压入调用者栈帧的操作数栈。

将PC计数器的值指向下一条方法指令位置。

小结



### 本地方法栈（私有）

本地方法栈和Java虚拟机栈发挥的作用非常相似，主要区别是Java虚拟机栈执行的是Java方法服务，而本地方法栈执行Native方法服务(通常用C编写)。



### Java堆（全局共享）

对大多数应用而言，Java 堆是虚拟机所管理的内存中最大的一块，是被所有线程共享的一块内存区域，在虚拟机启动时创建。此内存区域的唯一作用就是存放对象实例，几乎所有的对象实例都是在这里分配的（不绝对，在虚拟机的优化策略下，也会存在栈上分配、标量替换的情况，后面的章节会详细介绍）。

Java 堆是 GC 回收的主要区域，因此很多时候也被称为 GC 堆。

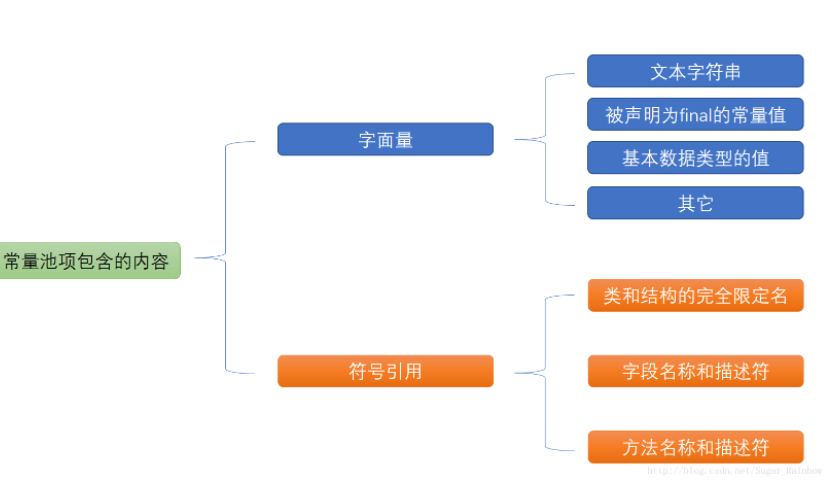
从内存回收的角度看，由于现在收集器基本都采用分代收集算法，所以在Java堆被划分成两个不同的区域：新生代 (Young Generation) 、老年代 (Old Generation) 。新生代 (Young) 又被划分为三个区域：一个Eden区和两个Survivor区 - From Survivor区和To Survivor区。不过无论如何划分，都与存放内容无关，无论哪个区域，存储的都仍然时对象实例，记你一步划分的目的是为了使JVM能够更好的管理堆内存中的对象，包括内存的分配以及回收。

从内存回收的角度看，线程共享的 Java 堆可能划分出多个线程私有的分配缓冲区（Thread Local Allocation Buffer，TLAB）。「属于线程共享的内存区域」

### 方法区（全局共享）

方法区和Java堆一样，为多个线程共享，它用于存储类信息、常量、静态常量和即时编译后的代码等数据。Non-Heap（非堆）「属于线程共享的内存区域」

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，还有一项信息就是常量池（Constant Pool Table），用于存放编译期生成的各种字面常量和符号引用，这部分内容会在类加载后进入方法区的运行时常量池。

下面信息为附加信息

HotSpot虚拟机中，将方法区称为“永久代”，本质上两者并不等价，仅仅是因为HotSpot虚拟机把GC分代收集扩展至方法区。

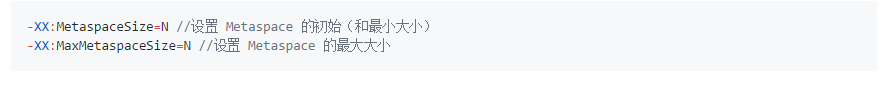
JDK 7的HotSpot中，已经将原本存放于永久代中的字符串常量池移出。

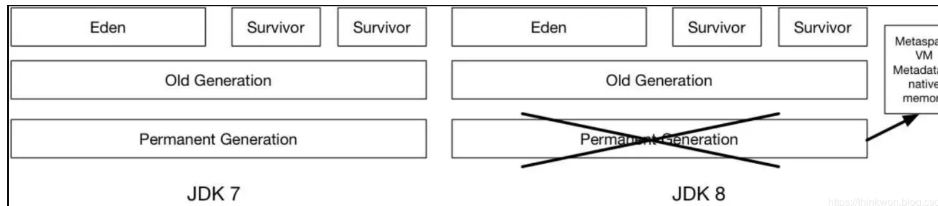
根据虚拟机规范的规定，当方法区无法满足内存分配需求时，将会抛出OutOfMemoryError异常。当常量池无法再申请到内存时也会抛出OutOfMemoryError异常。

JDK 1.8 之前永久代还没被彻底移除的时候通常通过下面这些参数来调节方法区大小



JDK 8的HotSpot中，已经将永久代废除，用元数据实现了方法区。元空间与永久代之间最大的区别在于：元空间并不在虚拟机中，而是使用本地内存。理论上取决于32位/64位系统可虚拟的内存大小。可见也不是无限制的，需要配置参数。





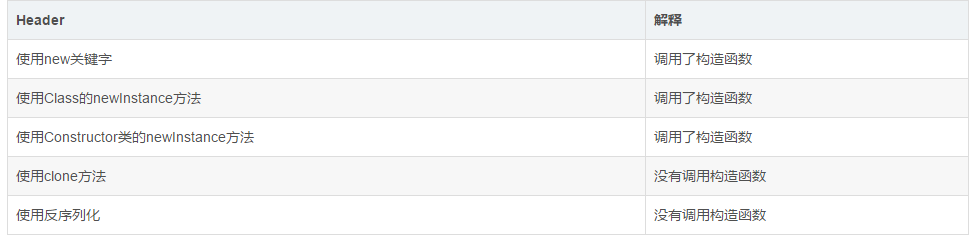
### 直接内存

直接内存（Direct Memory）并不是虚拟机运行时数据区的一部分，也不是 Java 虚拟机规范中定义的内存区域。Java 中的 NIO 可以使用 Native 函数直接分配堆外内存，通常直接内存的速度会优于Java堆内存，然后通过一个存储在 Java 堆中的 DiectByteBuffer 对象作为这块内存的引用进行操作。这样能在一些场景显著提高性能，对于读写频繁、性能要求高的场景，可以考虑使用直接内存，因为避免了在 Java 堆和 Native 堆中来回复制数据。直接内存不受 Java 堆大小的限制。

# HotSpot虚拟机对象探秘

## 对象的创建

说到对象的创建，首先让我们看看 Java 中提供的几种对象创建方式：



下面是对象创建的主要流程:



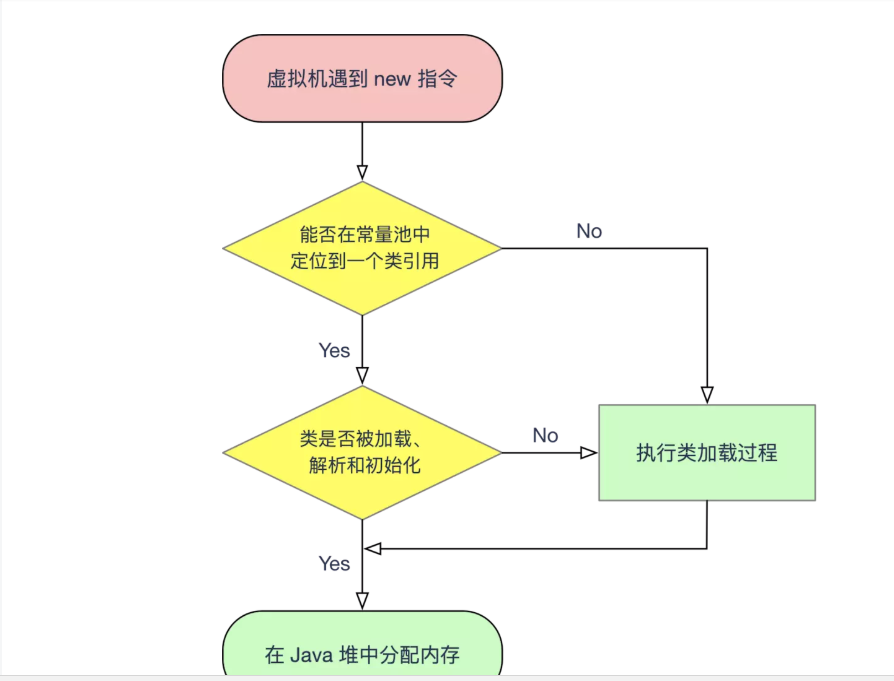
虚拟机遇到一条new指令时，先检查常量池是否已经加载相应的类，如果没有，必须先执行相应的类加载。类加载通过后，接下来分配内存。若Java堆中内存是绝对规整的，使用“指针碰撞“方式分配内存；如果不是规整的，就从空闲列表中分配，叫做”空闲列表“方式。划分内存时还需要考虑一个问题-并发，也有两种方式: CAS同步处理，或者本地线程分配缓冲(Thread Local Allocation Buffer, TLAB)。然后内存空间初始化操作，接着是做一些必要的对象设置(元信息、哈希码…)，最后执行<init>方法。

下面内容是对象创建的详细过程

对象的创建通常是通过new关键字创建一个对象的，当虚拟机接收到一个new指令时，它会做如下的操作。

### 1.判断对象对应的类是否加载、链接、初始化

虚拟机接收到一条new指令时，首先会去检查这个指定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类加载器加载、链接和初始化过。如果没有则先执行相应的类加载过程。



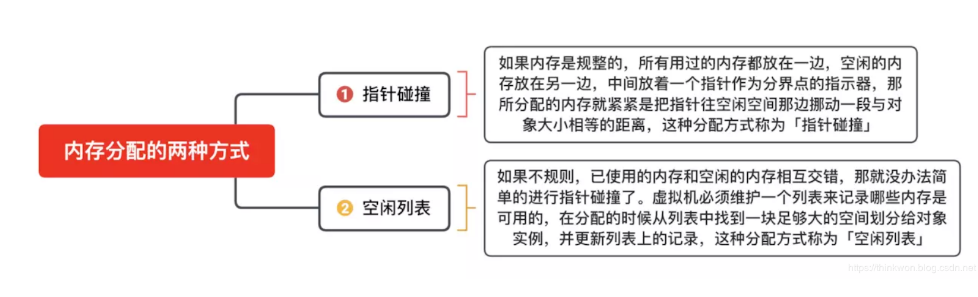
### 2.为对象分配内存

类加载完成后，接着会在Java堆中划分一块内存分配给对象。内存分配根据Java堆是否规整，有两种方式：

指针碰撞：如果Java堆的内存是规整，即所有用过的内存放在一边，而空闲的的放在另一边。分配内存时将位于中间的指针指示器向空闲的内存移动一段与对象大小相等的距离，这样便完成分配内存工作。

空闲列表：如果Java堆的内存不是规整的，则需要由虚拟机维护一个列表来记录那些内存是可用的，这样在分配的时候可以从列表中查询到足够大的内存分配给对象，并在分配后更新列表记录。

选择哪种分配方式是由 Java 堆是否规整来决定的，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。

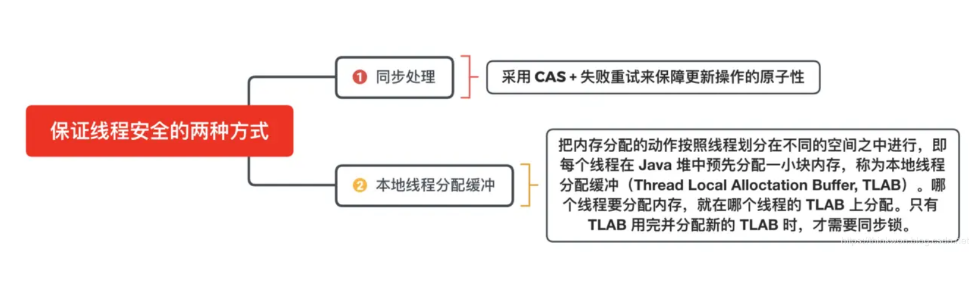


### 3.处理并发安全问题

对象的创建在虚拟机中是一个非常频繁的行为，哪怕只是修改一个指针所指向的位置，在并发情况下也是不安全的，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：

对分配内存空间的动作进行同步处理（采用 CAS + 失败重试来保障更新操作的原子性）；

把内存分配的动作按照线程划分在不同的空间之中进行，即每个线程在 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。通过-XX:+/-UserTLAB参数来设定虚拟机是否使用TLAB。



### 4.初始化分配到的内存空间

内存分配完后，虚拟机要将分配到的内存空间初始化为零值（不包括对象头）。如果使用了 TLAB，这一步会提前到 TLAB 分配时进行。这一步保证了对象的实例字段在 Java 代码中可以不赋初始值就直接使用。程序能访问到这些字段的数据类型所对应的零值。

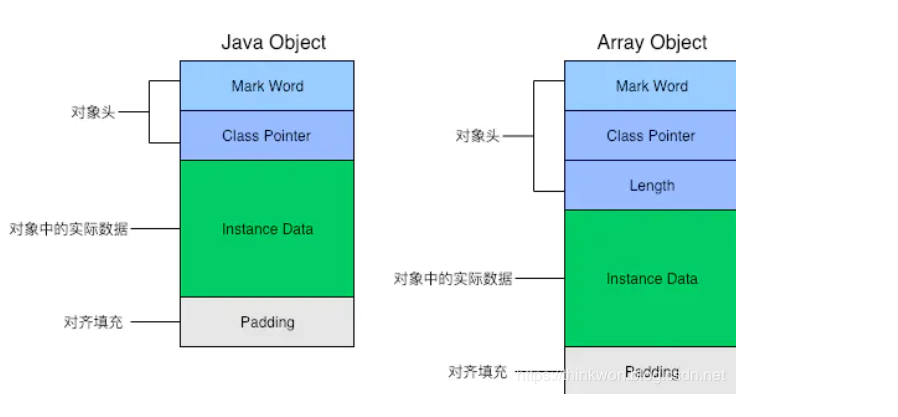
### 5.设置对象的对象头

接下来设置对象头（Object Header）信息，包括对象的所属类、对象的HashCode和对象的GC分代年龄等数据存储在对象的对象头中。。 这些信息存放在对象头中。 另外，根据虚拟机当前运行状态的不同，如是否启用偏向锁等，对象头会有不同的设置方式。

### 6.执行init方法进行初始化

执行init方法，初始化对象的成员变量、调用类的构造方法，这样一个对象就被创建了出来。

对象的内存布局



## 对象的布局

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（Header）、实例数据（Instance Data）和对齐填充（Padding）。

### 对象头

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（哈希码、GC 分代年龄、锁状态标志等等），**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那个类的实例。如果是数组对象，还有一个保存数组长度的空间。

Mark Word（运行时数据）：用于存储对象自身运行时的数据，如哈希码（hashCode）、GC分带年龄、线程持有的锁、偏向线程ID 等信息。在32位系统占4字节，在64位系统中占8字节；

HotSpot虚拟机对象头Mark Word在其他状态（轻量级锁定、重量级锁定、GC标记、可偏向）下对象的存储内容如下表所示：



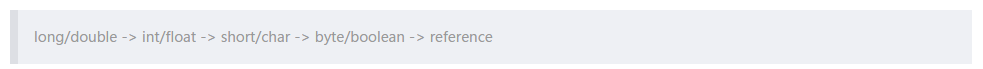
Class Pointer（类型指针）：用来指向对象对应的Class对象（其对应的元数据对象）的内存地址。在32位系统占4字节，在64位系统中占8字节；

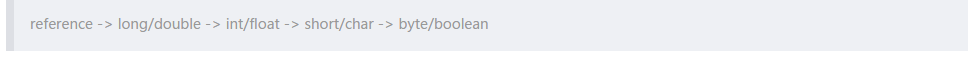
Length：如果是数组对象，还有一个保存数组长度的空间，占4个字节；

### 实例数据

实例数据 是对象真正存储的有效信息，无论是从父类继承下来的还是该类自身的，都需要记录下来，而这部分的存储顺序受虚拟机的分配策略和定义的顺序的影响。

默认分配策略：

如果设置了-XX:FieldsAllocationStyle=0（默认是1），那么引用类型数据就会优先分配存储空间：

结论：

分配策略总是按照字节大小由大到小的顺序排列，相同字节大小的放在一起。

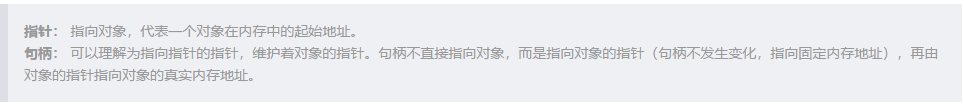
### 对齐填充

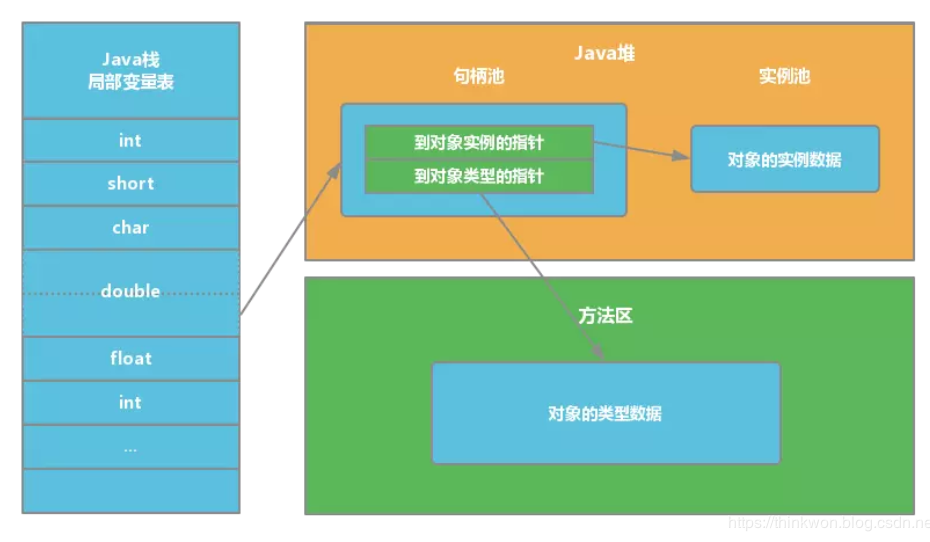
无特殊含义，不是必须存在的，仅作为占位符。

HotSpot虚拟机要求每个对象的起始地址必须是8字节的整数倍，也就是对象的大小必须是8字节的整数倍。而对象头部分正好是8字节的倍数（32位为1倍，64位为2倍），因此，当对象实例数据部分没有对齐的时候，就需要通过对齐填充来补全。

## 对象的访问定位

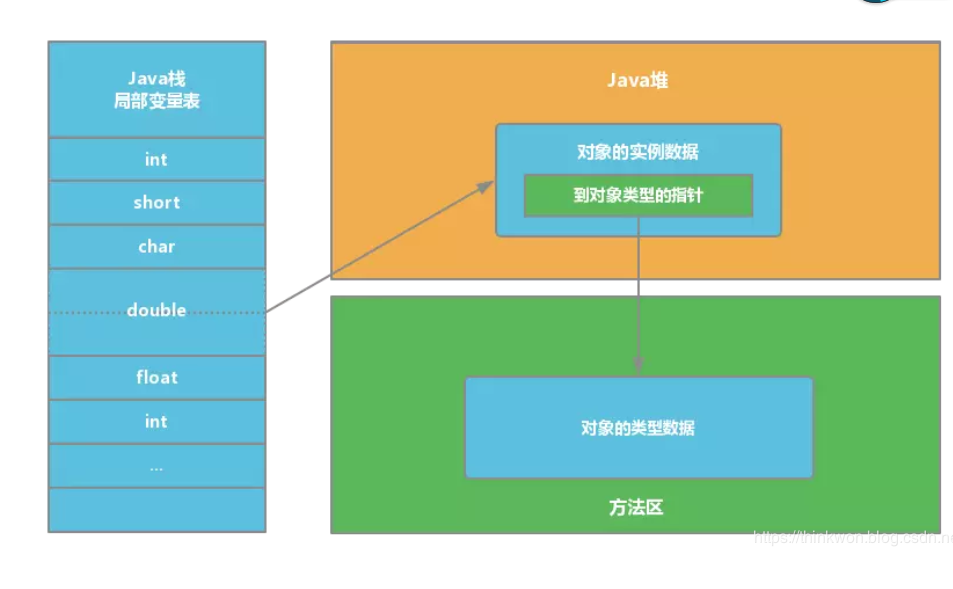
Java程序需要通过 JVM 栈上的引用访问堆中的具体对象。对象的访问方式取决于 JVM 虚拟机的实现。目前主流的访问方式有 句柄 和 直接指针 两种方式。

句柄访问

Java堆中划分出一块内存来作为句柄池，引用中存储对象的句柄地址，而句柄中包含了对象实例数据与对象类型数据各自的具体地址信息，具体构造如下图所示：

优势：引用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，而引用本身不需要修改。

### 直接指针

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

优势：速度更快，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非常可观的执行成本。HotSpot 中采用的就是这种方式。

# 实战：OutOfMemoryError异常

内存异常是我们工作当中经常会遇到问题，但如果仅仅会通过加大内存参数来解决问题显然是不够的，应该通过一定的手段定位问题，到底是因为参数问题，还是程序问题(无限创建，内存泄露)。定位问题后才能采取合适的解决方案，而不是一内存溢出就查找相关参数加大。

## 概念

在分析问题之前先给大家讲一讲排查内存溢出问题的方法，内存溢出时JVM虚拟机会退出，那么我们怎么知道JVM运行时的各种信息呢，Dump机制会帮助我们，可以通过加上VM参数-XX:+HeapDumpOnOutOfMemoryError让虚拟机在出现内存溢出异常时生成dump文件，然后通过外部工具(VisualVM)来具体分析异常的原因。

除了程序计数器外，Java虚拟机的其他运行时区域都有可能发生OutOfMemoryError的异常，下面分别给出验证：

## Java堆溢出

Java堆用来存储对象，因此只要不断创建对象，并保证 GC Roots 到对象之间有可达路径来避免垃圾回收机制清楚这些对象，那么当对象数量达到最大堆容量时就会产生 OOM。



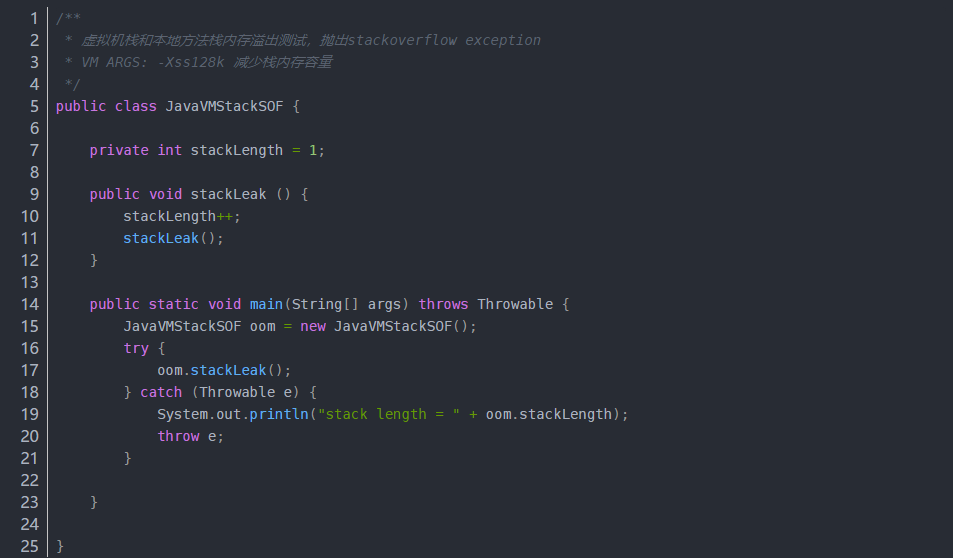
堆内存 OOM 是经常会出现的问题，异常信息会进一步提示 Java heap space

## 虚拟机栈和本地方法栈溢出

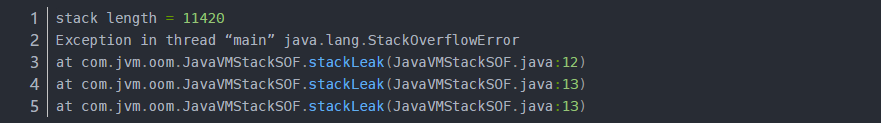
在 HotSpot 虚拟机中不区分虚拟机栈和本地方法栈，栈容量只由 -Xss 参数设定。关于虚拟机栈和本地方法栈，在 Java 虚拟机规范中描述了两种异常：

如果线程请求的栈深度大于虚拟机所允许的最大深度，将抛出 StackOverflowError 异常。

如果虚拟机在扩展栈时无法申请到足够的内存空间，则抛出 OutOfMemoryError 异常。



运行结果:



以上代码在单线程环境下，无论是由于栈帧太大还是虚拟机栈容量太小，当内存无法分配时，抛出的都是 StackOverflowError 异常。

如果测试环境是多线程环境，通过不断建立线程的方式可以产生内存溢出异常，代码如下所示。但是这样产生的 OOM 与栈空间是否足够大不存在任何联系，在这种情况下，为每个线程的栈分配的内存足够大，反而越容易产生OOM 异常。这点不难理解，每个线程分配到的栈容量越大，可以建立的线程数就变少，建立多线程时就越容易把剩下的内存耗尽。这点在开发多线程的应用时要特别注意。如果建立过多线程导致内存溢出，在不能减少线程数或更换64位虚拟机的情况下，只能通过减少最大堆和减少栈容量来换取更多的线程。

方法区和运行时常量池溢出

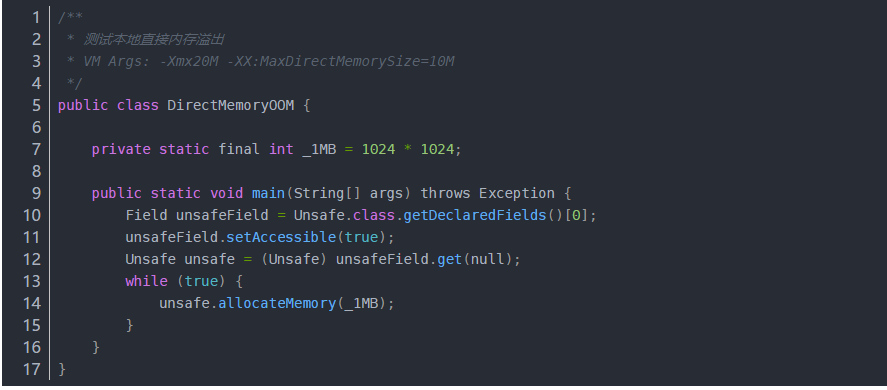
方法区用于存放Class的相关信息，对这个区域的测试，基本思路是运行时产生大量的类去填满方法区，直到溢出。使用CGLib实现。

方法区溢出也是一种常见的内存溢出异常，在经常生成大量Class的应用中，需要特别注意类的回收情况，这类场景除了使用了CGLib字节码增强和动态语言外，常见的还有JSP文件的应用(JSP第一次运行时要编译为Java类)、基于OSGI的应用等。

本机直接内存溢出

DirectMemory 容量可通过 -XX:MaxDirectMemorySize 指定，如不指定，则默认与Java堆最大值一样。测试代码使用了 Unsafe 实例进行内存分配。

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，如果发现 OOM 之后 Dump 文件很小，而程序直接或间接使用了NIO，那就可以考虑检查一下是不是这方面的原因。



# 垃圾回收器与内存分配策略

## 概述

说起垃圾收集（Garbage Collection，GC），大部分人都把这项技术当作Java语言的伴生产物。事实上，GC的历史比Java久远，1960年诞生于MIT的Lisp是第一门真正使用内存动态分配和垃圾收集技术的语言。当Lisp还在胚胎时期时，人们就在思考GC需要完成的3件事情：

哪些内存需要回收？

什么时候回收？

如何回收？

经过半个多世纪的发展，目前内存的动态分配与内存回收技术已经相当成熟，一切看起来都进入了"自动化"时代，那为什么我们还要去了解GC和内存分配？答案很简单：当需要排查各种内存溢出、内存泄漏问题时，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。

## 对象已死吗

垃圾收集器在做垃圾回收的时候，首先需要判定的就是哪些内存是需要被回收的，哪些对象是「存活」的，是不可以被回收的；哪些对象已经「死掉」了，需要被回收。

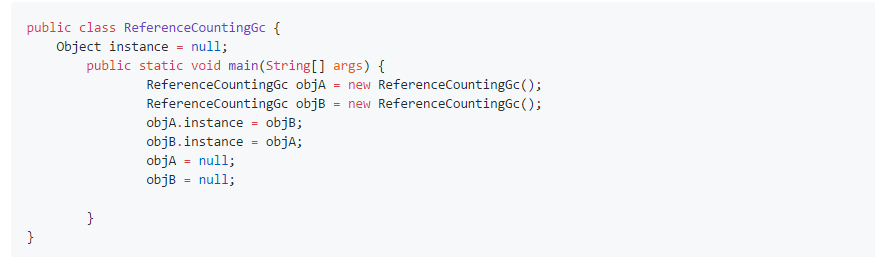
### 引用计数法

Java 堆 中每个具体对象（不是引用）都有一个引用计数器。当一个对象被创建并初始化赋值后，该变量计数设置为1。每当有一个地方引用它时，计数器值就加1。当引用失效时，即一个对象的某个引用超过了生命周期（出作用域后）或者被设置为一个新值时，计数器值就减1。任何引用计数为0的对象可以被当作垃圾收集。当一个对象被垃圾收集时，它引用的任何对象计数减1。

优点：

引用计数收集器执行简单，判定效率高，交织在程序运行中。对程序不被长时间打断的实时环境比较有利。

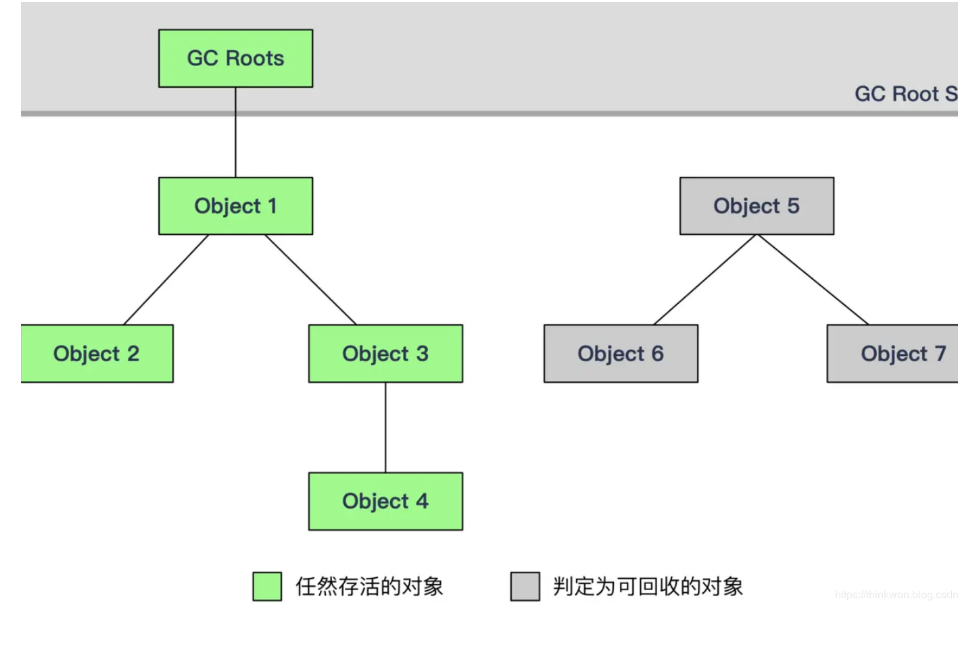
缺点：

难以检测出对象之间的循环引用。同时，引用计数器增加了程序执行的开销。所以Java语言并没有选择这种算法进行垃圾回收。

### 可达性分析算法

可达性分析算法又叫根搜索算法，该算法的基本思想就是通过一系列称为「GC Roots」的对象作为起始点，从这些起始点开始往下搜索，搜索所走过的路径称为引用链，当一个对象到 GC Roots 对象之间没有任何引用链的时候（不可达），证明该对象是不可用的，于是就会被判定为可回收对象。

如下图所示: Object5、Object6、Object7 虽然互有关联, 但它们到GC Roots是不可达的, 因此也会被判定为可回收的对象。



在 Java 中可作为 GC Roots 的对象包含以下几种：

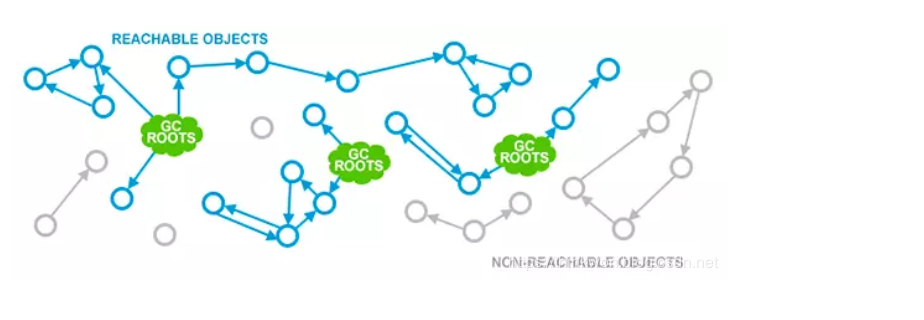
虚拟机栈（栈帧中的本地变量表）中引用的对象；

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（Native 方法）引用的对象。

JVM中用到的所有现代GC算法在回收前都会先找出所有仍存活的对象。可达性分析算法是从离散数学中的图论引入的，程序把所有的引用关系看作一张图。下图展示的JVM中的内存布局可以用来很好地阐释这一概念：



## 再谈引用

无论是通过引用计数器还是通过可达性分析来判断对象是否可以被回收都设计到「引用」的概念。在 Java 中，根据引用关系的强弱不一样，将引用类型划为强引用（Strong Reference）、软引用（Soft Reference）、弱引用（Weak Reference）和虚引用（Phantom Reference）。

**1．强引用（StrongReference）**

以前我们使用的大部分引用实际上都是强引用，这是使用最普遍的引用。如果一个对象具有强引用，那就类似于**必不可少的生活用品**，垃圾回收器绝不会回收它。当内存空间不足，Java 虚拟机宁愿抛出 OutOfMemoryError 错误，使程序异常终止，也不会靠随意回收具有强引用的对象来解决内存不足问题。

**2．软引用（SoftReference）**

如果一个对象只具有软引用，那就类似于**可有可无的生活用品**。如果内存空间足够，垃圾回收器就不会回收它，如果内存空间不足了，就会回收这些对象的内存。只要垃圾回收器没有回收它，该对象就可以被程序使用。软引用可用来实现内存敏感的高速缓存。

软引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被垃圾回收，JAVA 虚拟机就会把这个软引用加入到与之关联的引用队列中。

**3．弱引用（WeakReference）**

如果一个对象只具有弱引用，那就类似于**可有可无的生活用品**。弱引用与软引用的区别在于：只具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。不过，由于垃圾回收器是一个优先级很低的线程， 因此不一定会很快发现那些只具有弱引用的对象。

弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果弱引用所引用的对象被垃圾回收，Java 虚拟机就会把这个弱引用加入到与之关联的引用队列中。

**4．虚引用（PhantomReference）**

"虚引用"顾名思义，就是形同虚设，与其他几种引用都不同，虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收。

**虚引用主要用来跟踪对象被垃圾回收的活动**。

**虚引用与软引用和弱引用的一个区别在于：** 虚引用必须和引用队列（ReferenceQueue）联合使用。当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之关联的引用队列中。程序可以通过判断引用队列中是否已经加入了虚引用，来了解被引用的对象是否将要被垃圾回收。程序如果发现某个虚引用已经被加入到引用队列，那么就可以在所引用的对象的内存被回收之前采取必要的行动。

特别注意，在程序设计中一般很少使用弱引用与虚引用，使用软引用的情况较多，这是因为**软引用可以加速 JVM 对垃圾内存的回收速度，可以维护系统的运行安全，防止内存溢出（OutOfMemory）等问题的产生**。

#### 不可达的对象并非“非死不可”

一个对象是否应该在垃圾回收器在GC时回收，至少要经历两次标记过程。

第一次标记：如果对象在进行可达性分析后被判定为不可达对象，那么它将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。对象没有覆盖 finalize() 方法或者该对象的 finalize() 方法曾经被虚拟机调用过，则判定为没必要执行。

finalize()第二次标记：如果被判定为有必要执行 finalize() 方法，那么这个对象会被放置到一个 F-Queue 队列中，并在稍后由虚拟机自动创建的、低优先级的 Finalizer 线程去执行该对象的 finalize() 方法。但是虚拟机并不承诺会等待该方法结束，这样做是因为，如果一个对象的 finalize() 方法比较耗时或者发生了死循环，就可能导致 F-Queue 队列中的其他对象永远处于等待状态，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，如果对象要在 finalize() 中挽救自己，只要重新与 GC Roots 引用链关联上就可以了。这样在第二次标记时它将被移除「即将回收」的集合，如果对象在这个时候还没有逃脱，那么它基本上就真的被回收了。

### 回收方法区

前面介绍过，方法区在 HotSpot 虚拟机中被划分为永久代。在 Java 虚拟机规范中没有要求方法区实现垃圾收集，而且方法区垃圾收集的性价比也很低。

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**。

#### 废弃常量的回收

假如在常量池中存在字符串 "abc"，如果当前没有任何 String 对象引用该字符串常量的话，就说明常量 "abc" 就是废弃常量，如果这时发生内存回收的话而且有必要的话，"abc" 就会被系统清理出常量池。

#### 类回收

类的回收条件就比较苛刻了。要判定一个类是否可以被回收，要满足以下三个条件：

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

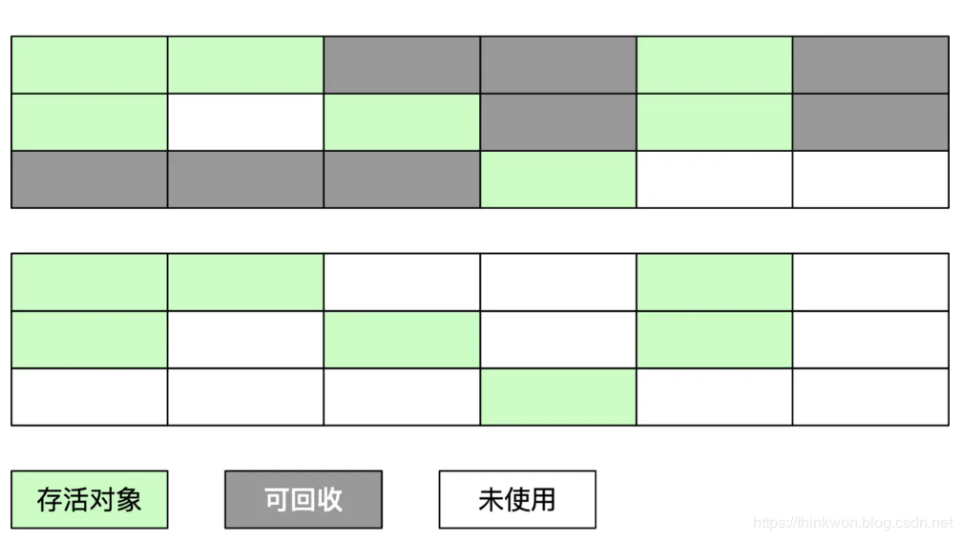
## 垃圾收集算法

### 标记-清除算法

该算法分为“标记”和“清除”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。它是最基础的收集算法，后续的算法都是对其不足进行改进得到。优点：实现简单，不需要对象进行移动。

缺点：标记、清除过程效率低，产生大量不连续的内存碎片，提高了垃圾回收的频率。

标记-清除算法的执行的过程如下图所示



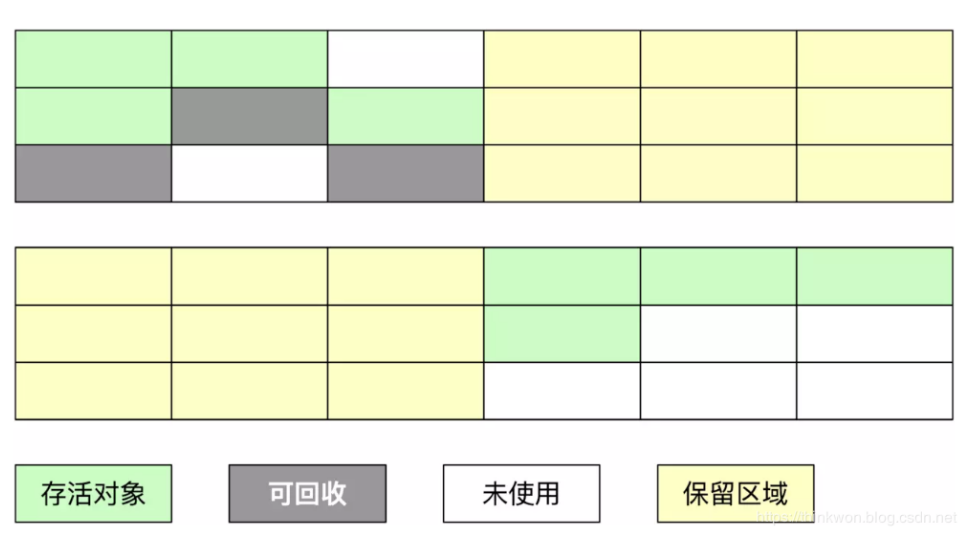
### 复制算法

为了解决标记-清除算法的效率不高的问题，产生了复制算法。它把内存空间划为两个相等的区域，每次只使用其中一个区域。垃圾收集时，遍历当前使用的区域，把存活对象复制到另外一个区域中，最后将当前使用的区域的可回收的对象进行回收。

优点：按顺序分配内存即可，实现简单、运行高效，不用考虑内存碎片。

缺点：可用的内存大小缩小为原来的一半，对象存活率高时会频繁进行复制。

复制算法的执行过程如下图所示



现在的商业虚拟机都采用这种算法来回收新生代，在 IBM 的研究中新生代中的对象 98% 都是「朝生夕死」，所以并不需要按照 1:1 的比例来划分空间，而是将内存分为一块较大的 Eden 空间和两块较小的 Survivor 空间，每次使用 Eden 和其中一块 Survivor。当回收时，将 Eden 和 Survivor 中还存活的对象一次性复制到另一块 Survivor 空间上，最后清理掉 Eden 和刚才用过的 Survivor 空间。 HotSpot 默认 Eden 和 Survivor 的大小比例是 8:1，也就是每次新生代中可用的内存为整个新生代容量的 90%（80%+10%），只有 10% 会被浪费。当然，98% 的对象可回收只是一般场景下的数据，我们没办法保证每次回收后都只有不多于 10% 的对象存活，当 Survivor 空间不够用时，需要依赖其它内存（这里指老年代）进行分配担保。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存活的对象时，这些对象将直接通过分配担保机制进入老年代。

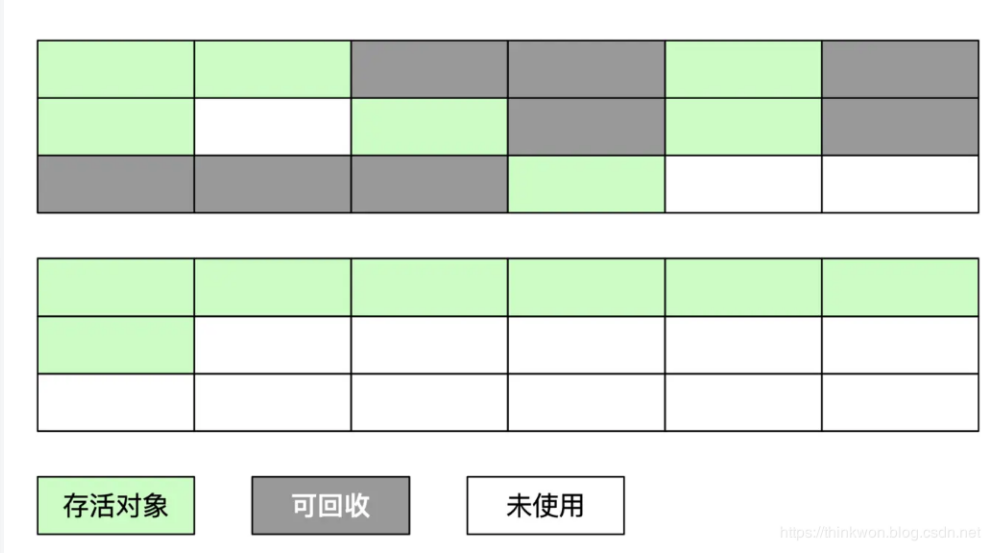
### 标记-整理算法

在新生代中可以使用复制算法，但是在老年代就不能选择复制算法了，因为老年代的对象存活率会较高，这样会有较多的复制操作，导致效率变低。标记-清除算法可以应用在老年代中，但是它效率不高，在内存回收后容易产生大量内存碎片。因此就出现了一种标记-整理算法（Mark-Compact）算法，与标记-整理算法不同的是，在标记可回收的对象后将所有存活的对象压缩到内存的一端，使他们紧凑的排列在一起，然后对端边界以外的内存进行回收。回收后，已用和未用的内存都各自一边。

优点：解决了标记-清理算法存在的内存碎片问题。

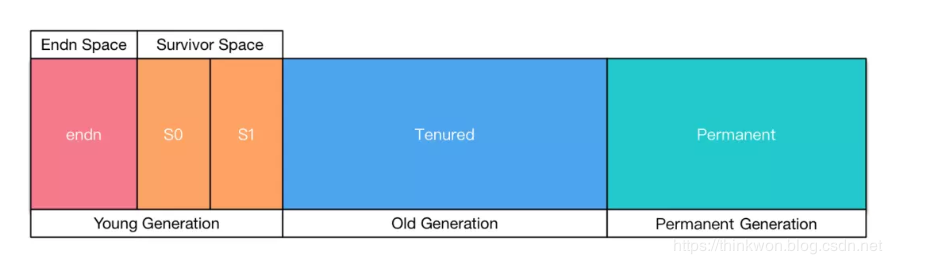
缺点：仍需要进行局部对象移动，一定程度上降低了效率。

标记-整理算法的执行过程如下图所示



### 分代收集算法

当前商业虚拟机都采用分代收集的垃圾收集算法。分代收集算法，顾名思义是根据对象的存活周期将内存划分为几块。一般包括年轻代、老年代 和 永久代，如图所示：



#### 新生代（Young generation）

绝大多数最新被创建的对象会被分配到这里，由于大部分对象在创建后会很快变得不可达，所以很多对象被创建在新生代，然后消失。对象从这个区域消失的过程我们称之为 minor GC。

新生代 中存在一个Eden区和两个Survivor区。新对象会首先分配在Eden中（如果新对象过大，会直接分配在老年代中）。在GC中，Eden中的对象会被移动到Survivor中，直至对象满足一定的年纪（定义为熬过GC的次数），会被移动到老年代。

可以设置新生代和老年代的相对大小。这种方式的优点是新生代大小会随着整个堆大小动态扩展。参数 -XX:NewRatio 设置老年代与新生代的比例。例如 -XX:NewRatio=8 指定 老年代/新生代 为8/1. 老年代 占堆大小的 7/8 ，新生代 占堆大小的 1/8（默认即是 1/8）。

例如：

老年代（Old generation）

对象没有变得不可达，并且从新生代中存活下来，会被拷贝到这里。其所占用的空间要比新生代多。也正由于其相对较大的空间，发生在老年代上的GC要比新生代要少得多。对象从老年代中消失的过程，可以称之为major GC（或者full GC）。

#### 永久代（permanent generation）

像一些类的层级信息，方法数据 和方法信息（如字节码，栈 和 变量大小），运行时常量池（JDK7之后移出永久代），已确定的符号引用和虚方法表等等。它们几乎都是静态的并且很少被卸载和回收，在JDK8之前的HotSpot虚拟机中，类的这些\*\*“永久的”\*\* 数据存放在一个叫做永久代的区域。

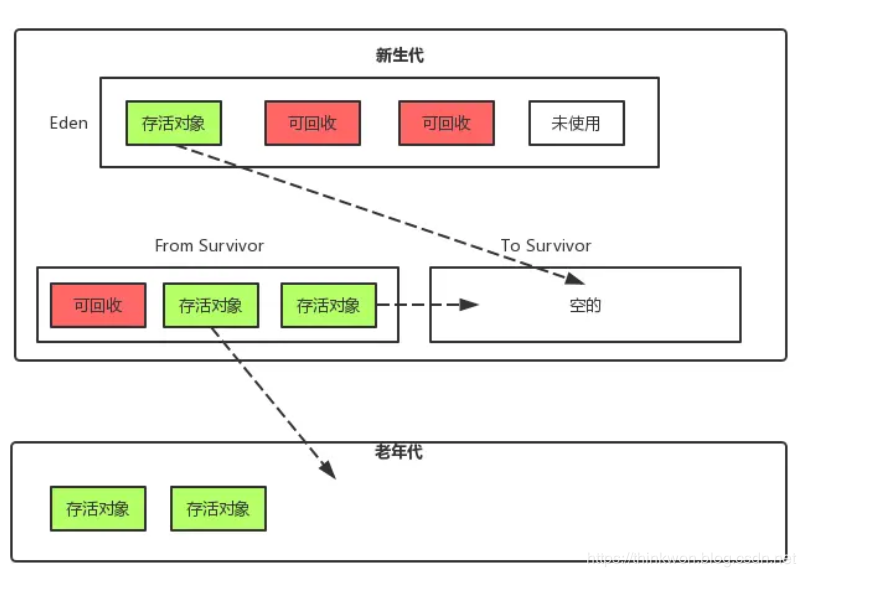
永久代一段连续的内存空间，我们在JVM启动之前可以通过设置-XX:MaxPermSize的值来控制永久代的大小。但是JDK8之后取消了永久代，这些元数据被移到了一个与堆不相连的称为元空间 (Metaspace) 的本地内存区域。

小结

当执行一次Minor Collection时，Eden空间的存活对象会被复制到To Survivor空间，并且之前经过一次Minor Collection并在From Survivor空间存活的仍年轻的对象也会复制到To Survivor空间。

有两种情况Eden空间和From Survivor空间存活的对象不会复制到To Survivor空间，而是晋升到老年代。一种是存活的对象的分代年龄超过-XX:MaxTenuringThreshold（用于控制对象经历多少次Minor GC才晋升到老年代）所指定的阈值。另一种是To Survivor空间容量达到阈值。

当所有存活的对象被复制到To Survivor空间，或者晋升到老年代，也就意味着Eden空间和From Survivor空间剩下的都是可回收对象，如下图所示。



这时GC执行Minor Collection，Eden空间和From Survivor空间都会被清空，而存活的对象都存放在To Survivor空间。

接下来将From Survivor空间和To Survivor空间互换位置，也就是此前的From Survivor空间成为了现在的To Survivor空间，每次Survivor空间互换都要保证To Survivor空间是空的，这就是复制算法在新生代中的应用。在老年代则采用了标记-压缩算法。

JDK8堆内存一般是划分为年轻代和老年代，不同年代 根据自身特性采用不同的垃圾收集算法。

对于新生代，每次GC时都有大量的对象死亡，只有少量对象存活。考虑到复制成本低，适合采用复制算法。因此有了From Survivor和To Survivor区域。

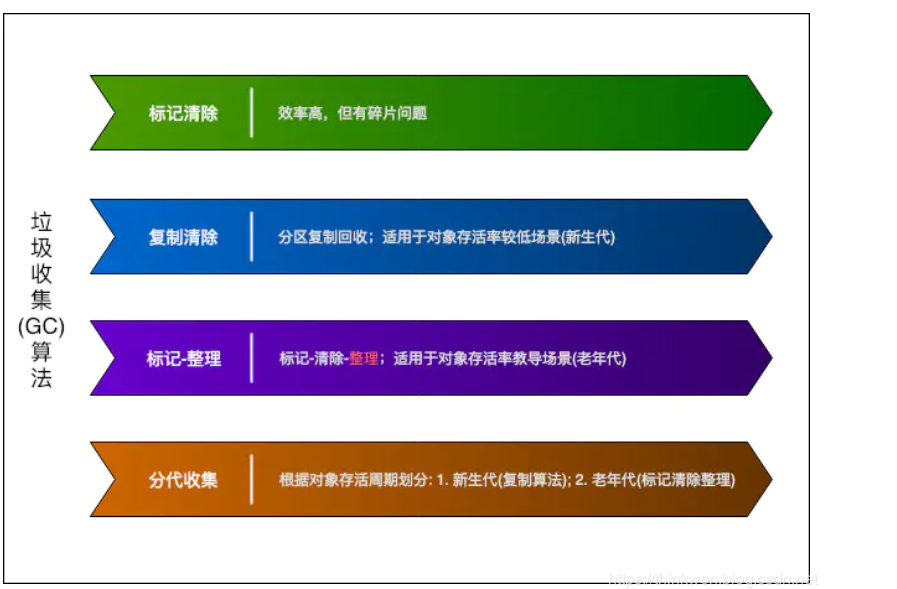
对于老年代，因为对象存活率高，没有额外的内存空间对它进行担保。因而适合采用标记-清理算法和标记-整理算法进行回收。

由于对象进行了分代处理，因此垃圾回收区域、时间也不一样。垃圾回收有两种类型，Minor GC 和 Full GC。

Minor GC：新生代垃圾收集。对新生代进行回收，不会影响到年老代。因为新生代的 Java 对象大多死亡频繁，所以 Minor GC 非常频繁，一般在这里使用速度快、效率高的算法，使垃圾回收能尽快完成。

Full GC：也叫 Major GC，对整个堆进行回收，包括新生代和老年代（JDK8 取消永久代）。由于Full GC需要对整个堆进行回收，所以比Minor GC要慢，因此应该尽可能减少Full GC的次数。它的收集频率较低，耗时较长。

垃圾收集算法小结



## HotSpot的算法实现

### 枚举根节点

可达性分析枚举GC Roots时 ，必须stop the world

目前JVM使用准确式GC，停顿时并不需要一个个检查，而是从预先存放的地方直接取。（HotSpot保存在OopMap数据结构中）

### 安全点

基于效率考虑，生成OopMap只会才特定的地方，称为安全点

安全点的选定方法

抢先式中断：现代JVM不采用

主动式中断：线程轮询安全点标识，然后挂起

### 安全区域

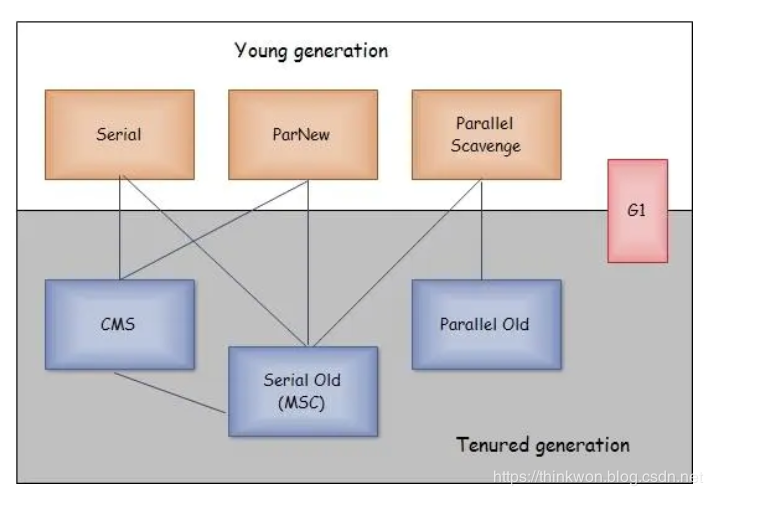
对于没有分配cpu的线程（sleep），安全点无法处理，由安全区域解决

安全区域指一段代码中引用关系不会发生变化

线程进入安全区域时，JVM发起GC就不用管这些线程，离开时需要检查GC是否完成，未完成就需要等待

## 垃圾收集器

如果说垃圾收集算法是内存回收的方法论，那么垃圾收集器就是内存回收的具体实现。下图展示了7种作用于不同分代的收集器，其中用于回收新生代的收集器包括Serial、PraNew、Parallel Scavenge，回收老年代的收集器包括Serial Old、Parallel Old、CMS，还有用于回收整个Java堆的G1收集器。不同收集器之间的连线表示它们可以搭配使用。



Serial收集器（复制算法): 新生代单线程收集器，标记和清理都是单线程，优点是简单高效；

ParNew收集器 (复制算法): 新生代收并行集器，实际上是Serial收集器的多线程版本，在多核CPU环境下有着比Serial更好的表现；

Parallel Scavenge收集器 (复制算法): 新生代并行收集器，追求高吞吐量，高效利用 CPU。吞吐量 = 用户线程时间/(用户线程时间+GC线程时间)，高吞吐量可以高效率的利用CPU时间，尽快完成程序的运算任务，适合后台应用等对交互相应要求不高的场景；

Serial Old收集器 (标记-整理算法): 老年代单线程收集器，Serial收集器的老年代版本；

Parallel Old收集器 (标记-整理算法)： 老年代并行收集器，吞吐量优先，Parallel Scavenge收集器的老年代版本；

CMS(Concurrent Mark Sweep)收集器（标记-清除算法）： 老年代并行收集器，以获取最短回收停顿时间为目标的收集器，具有高并发、低停顿的特点，追求最短GC回收停顿时间。

G1(Garbage First)收集器 (标记-整理算法)： Java堆并行收集器，G1收集器是JDK1.7提供的一个新收集器，G1收集器基于“标记-整理”算法实现，也就是说不会产生内存碎片。此外，G1收集器不同于之前的收集器的一个重要特点是：G1回收的范围是整个Java堆(包括新生代，老年代)，而前六种收集器回收的范围仅限于新生代或老年代。

附加：以下为垃圾回收器的详细说明，可以选择性阅读

#### 垃圾回收器分类标准

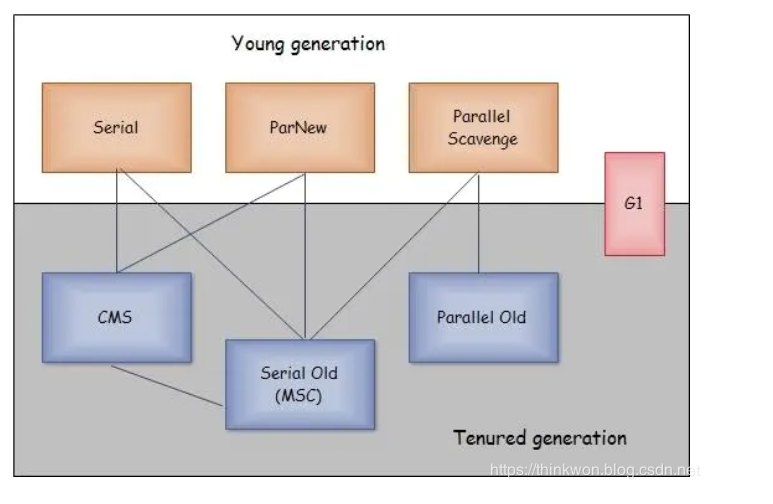


#### 七种垃圾回收器概述

在 JVM 中，具体实现有 Serial、ParNew、Parallel Scavenge、CMS、Serial Old（MSC）、Parallel Old、G1 等。在下图中，你可以看到 不同垃圾回收器 适合于 不同的内存区域，如果两个垃圾回收器之间 存在连线，那么表示两者可以 配合使用。

如果当 垃圾回收器 进行垃圾清理时，必须 暂停 其他所有的 工作线程，直到它完全收集结束。我们称这种需要暂停工作线程才能进行清理的策略为 Stop-the-World。以上回收器中， Serial、ParNew、Parallel Scavenge、Serial Old、Parallel Old 均采用的是 Stop-the-World 的策略。

图中有 7 种不同的 垃圾回收器，它们分别用于不同分代的垃圾回收。



新生代回收器：Serial、ParNew、Parallel Scavenge

老年代回收器：Serial Old、Parallel Old、CMS

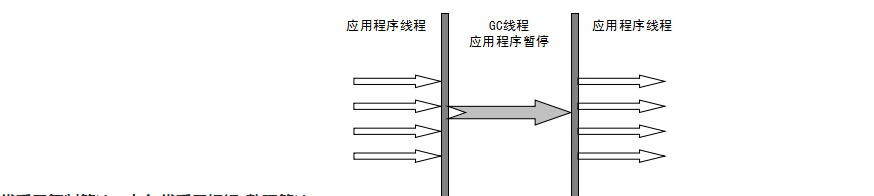
整堆回收器：G1

两个 垃圾回收器 之间有连线表示它们可以 搭配使用，可选的搭配方案如下：

单线程垃圾回收器

#### Serial（-XX:+UseSerialGC）

Serial 回收器是最基本的 新生代 垃圾回收器，是 单线程 的垃圾回收器。由于垃圾清理时，Serial 回收器 不存在 线程间的切换，因此，特别是在单 CPU 的环境下，它的 垃圾清除效率 比较高。对于 Client 运行模式的程序，选择 Serial 回收器是一个不错的选择。

Serial 新生代回收器 采用的是 复制算法。

#### Serial Old（-XX:+UseSerialGC）

Serial Old 回收器是 Serial 回收器的 老生代版本，属于 单线程回收器，它使用 标记-整理 算法。对于 Server 模式下的虚拟机，在 JDK1.5 及其以前，它常与 Parallel Scavenge 回收器配合使用，达到较好的 吞吐量，另外它也是 CMS 回收器在 Concurrent Mode Failure 时的 后备方案。

Serial 回收器和 Serial Old 回收器的执行效果如下：

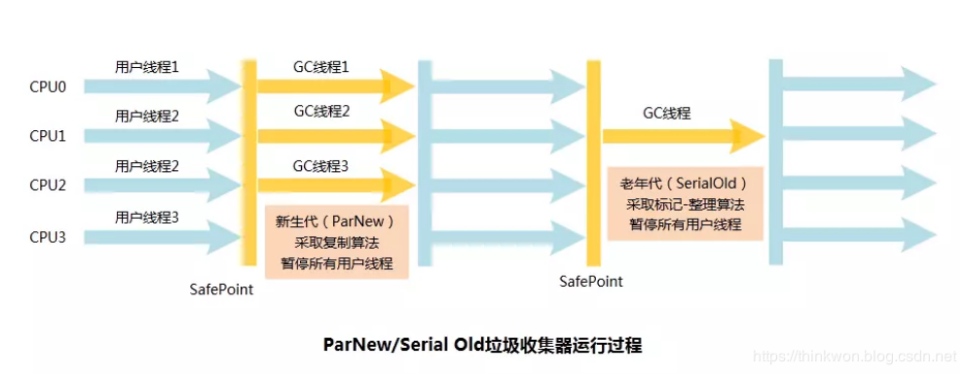


Serial Old 老年代回收器 采用的是 标记 - 整理算法。

### 多线程垃圾回收器（吞吐量优先）

#### ParNew（-XX:+UseParNewGC）

ParNew 回收器是在 Serial 回收器的基础上演化而来的，属于 Serial 回收器的 多线程版本，同样运行在 新生代区域。在实现上，两者共用很多代码。在不同运行环境下，根据 CPU 核数，开启 不同的线程数，从而达到 最优 的垃圾回收效果。对于那些 Server 模式的应用程序，如果考虑采用 CMS 作为 老生代回收器 时，ParNew 回收器是一个不错的选择。



ParNew 新生代回收器 采用的是 复制算法。

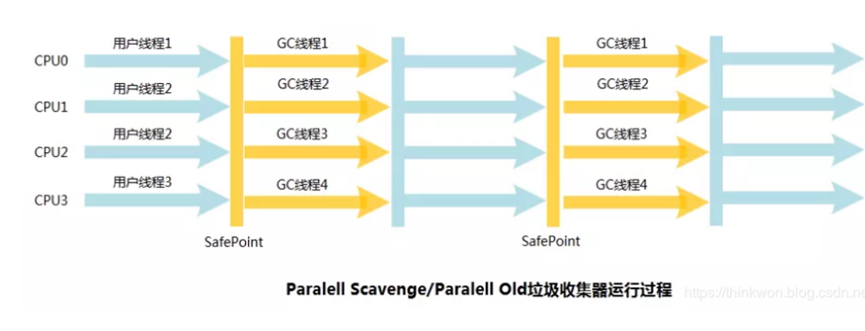
#### Parallel Scavenge（-XX:+UseParallelGC）

和 ParNew 回收一样，Parallel Scavenge 回收器也是运行在 新生代区域，属于 多线程 的回收器。但不同的是，ParNew 回收器是通过控制 垃圾回收 的 线程数 来进行参数调整，而 Parallel Scavenge 回收器更关心的是 程序运行的吞吐量。即一段时间内，用户代码 运行时间占 总运行时间 的百分比。

Parallel Scavenge 新生代回收器 采用的是 复制算法。

Parallel Old（-XX:+UseParallelOldGC）

Parallel Old 回收器是 Parallel Scavenge 回收器的 老生代版本，属于 多线程回收器，采用 标记-整理算法。Parallel Old 回收器和 Parallel Scavenge 回收器同样考虑了 吞吐量优先 这一指标，非常适合那些 注重吞吐量 和 CPU 资源敏感 的场合。

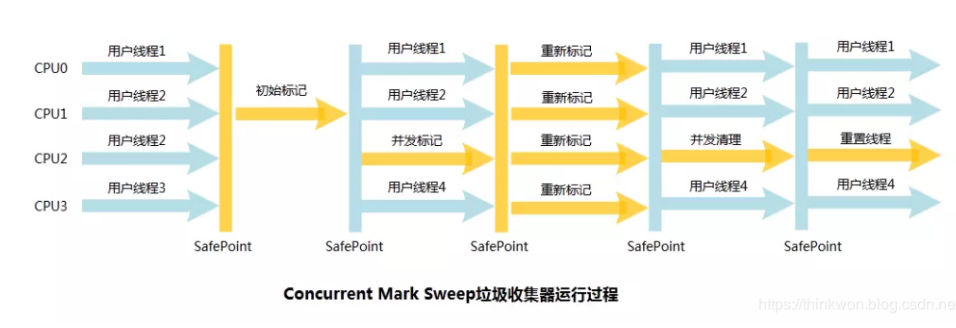


Parallel Old 老年代回收器 采用的是 标记 - 整理算法。

其他的回收器（停顿时间优先）

#### CMS（-XX:+UseConcMarkSweepGC）

CMS（Concurrent Mark Sweep） 回收器是在 最短回收停顿时间 为前提的回收器，属于 多线程回收器，采用 标记-清除算法。



相比之前的回收器，CMS 回收器的运作过程比较复杂，分为四步：

初始标记（CMS initial mark）

初始标记 仅仅是标记 GC Roots 内 直接关联 的对象。这个阶段 速度很快，需要 Stop the World。

并发标记（CMS concurrent mark）

并发标记 进行的是 GC Tracing，从 GC Roots 开始对堆进行 可达性分析，找出 存活对象。

重新标记（CMS remark）

重新标记 阶段为了 修正 并发期间由于 用户进行运作 导致的 标记变动 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，也需要 Stop The World。

并发清除（CMS concurrent sweep）

并发清除 阶段会清除垃圾对象。

初始标记（CMS initial mark）和 重新标记（CMS remark）会导致 用户线程 卡顿，Stop the World 现象发生。

在整个过程中，CMS 回收器的 内存回收 基本上和 用户线程 并发执行，如下所示：

由于 CMS 回收器 并发收集、停顿低，因此有些地方成为 并发低停顿回收器（Concurrent Low Pause Sweep Collector）。

CMS 回收器的缺点：

CMS回收器对CPU资源非常依赖

CMS 回收器过分依赖于 多线程环境，默认情况下，开启的 线程数 为（CPU 的数量 + 3）/ 4，当 CPU 数量少于 4 个时，CMS 对 用户查询 的影响将会很大，因为他们要分出一半的运算能力去 执行回收器线程；

CMS回收器无法清除浮动垃圾

由于 CMS 回收器 清除已标记的垃圾 （处于最后一个阶段）时，用户线程 还在运行，因此会有新的垃圾产生。但是这部分垃圾 未被标记，在下一次 GC 才能清除，因此被成为 浮动垃圾。

由于 内存回收 和 用户线程 是同时进行的，内存在被 回收 的同时，也在被 分配。当 老生代 中的内存使用超过一定的比例时，系统将会进行 垃圾回收；当 剩余内存 不能满足程序运行要求时，系统将会出现 Concurrent Mode Failure，临时采用 Serial Old 算法进行 清除，此时的 性能 将会降低。

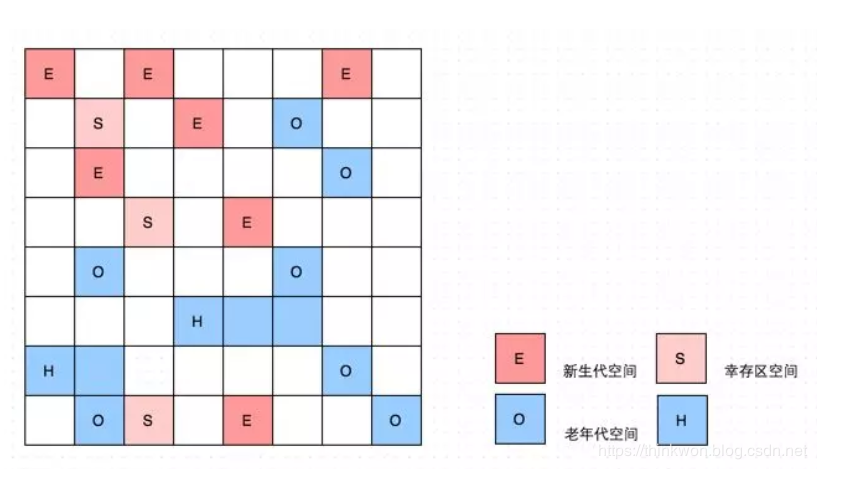
垃圾收集结束后残余大量空间碎片

CMS 回收器采用的 标记清除算法，本身存在垃圾收集结束后残余 大量空间碎片 的缺点。CMS 配合适当的 内存整理策略，在一定程度上可以解决这个问题。

#### G1回收器（垃圾区域Region优先）

G1 是 JDK 1.7 中正式投入使用的用于取代 CMS 的 压缩回收器。它虽然没有在物理上隔断 新生代 与 老生代，但是仍然属于 分代垃圾回收器。G1 仍然会区分 年轻代 与 老年代，年轻代依然分有 Eden 区与 Survivor 区。

G1 首先将 堆 分为 大小相等 的 Region，避免 全区域 的垃圾回收。然后追踪每个 Region 垃圾 堆积的价值大小，在后台维护一个 优先列表，根据允许的回收时间优先回收价值最大的 Region。同时 G1采用 Remembered Set 来存放 Region 之间的 对象引用 ，其他回收器中的 新生代 与 老年代 之间的对象引用，从而避免 全堆扫描。G1 的分区示例如下图所示：



这种使用 Region 划分 内存空间 以及有 优先级 的区域回收方式，保证 G1 回收器在有限的时间内可以获得尽可能 高的回收效率。

G1 和 CMS 运作过程有很多相似之处，整个过程也分为 4 个步骤：

1.初始标记（CMS initial mark）

初始标记 仅仅是标记 GC Roots 内 直接关联 的对象。这个阶段 速度很快，需要 Stop the World。

2.并发标记（CMS concurrent mark）

并发标记 进行的是 GC Tracing，从 GC Roots 开始对堆进行 可达性分析，找出 存活对象。

3.重新标记（CMS remark）

重新标记 阶段为了 修正 并发期间由于 用户进行运作 导致的 标记变动 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，也需要 Stop The World。

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。这个阶段可以与用户程序一起 并发执行，但是因为只回收一部分 Region，时间是用户可控制的，而且停顿 用户线程 将大幅提高回收效率。

与其它 GC 回收相比，G1 具备如下 4 个特点：

并行与并发

使用多个 CPU 来缩短 Stop-the-World 的 停顿时间，部分其他回收器需要停顿 Java 线程执行的 GC 动作，G1 回收器仍然可以通过 并发的方式 让 Java 程序继续执行。

分代回收

与其他回收器一样，分代概念 在 G1 中依然得以保留。虽然 G1 可以不需要 其他回收器配合 就能独立管理 整个GC堆，但它能够采用 不同的策略 去处理 新创建的对象 和 已经存活 一段时间、熬过多次 GC 的旧对象，以获取更好的回收效果。新生代 和 老年代 不再是 物理隔离，是多个 大小相等 的独立 Region。

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 算法实现的回收器。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，回收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 连续内存空间 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。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 垃圾回收 上的时间不得超过 N 毫秒。（后台维护的 优先列表，优先回收 价值大 的 Region）。

## 内存分配与回收策略

所谓自动内存管理，最终要解决的也就是内存分配和内存回收两个问题。前面我们介绍了内存回收，这里我们再来聊聊内存分配。

对象的内存分配通常是在 Java 堆上分配（随着虚拟机优化技术的诞生，某些场景下也会在栈上分配，后面会详细介绍），对象主要分配在新生代的 Eden 区，如果启动了本地线程缓冲，将按照线程优先在 TLAB 上分配。少数情况下也会直接在老年代上分配。总的来说分配规则不是百分百固定的，其细节取决于哪一种垃圾收集器组合以及虚拟机相关参数有关，但是虚拟机对于内存的分配还是会遵循以下几种「普世」规则：

### 对象优先在 Eden 区分配

多数情况，对象都在新生代 Eden 区分配。当 Eden 区分配没有足够的空间进行分配时，虚拟机将会发起一次 Minor GC。如果本次 GC 后还是没有足够的空间，则将启用分配担保机制在老年代中分配内存。

这里我们提到 Minor GC，如果你仔细观察过 GC 日常，通常我们还能从日志中发现 Major GC/Full GC。

Minor GC 是指发生在新生代的 GC，因为 Java 对象大多都是朝生夕死，所有 Minor GC 非常频繁，一般回收速度也非常快；

Major GC/Full GC 是指发生在老年代的 GC，出现了 Major GC 通常会伴随至少一次 Minor GC。Major GC 的速度通常会比 Minor GC 慢 10 倍以上。

### 大对象直接进入老年代

所谓大对象是指需要大量连续内存空间的对象，频繁出现大对象是致命的，会导致在内存还有不少空间的情况下提前触发 GC 以获取足够的连续空间来安置新对象。

前面我们介绍过新生代使用的是标记-清除算法来处理垃圾回收的，如果大对象直接在新生代分配就会导致 Eden 区和两个 Survivor 区之间发生大量的内存复制。因此对于大对象都会直接在老年代进行分配。

### 长期存活对象将进入老年代

虚拟机采用分代收集的思想来管理内存，那么内存回收时就必须判断哪些对象应该放在新生代，哪些对象应该放在老年代。因此虚拟机给每个对象定义了一个对象年龄的计数器，如果对象在 Eden 区出生，并且能够被 Survivor 容纳，将被移动到 Survivor 空间中，这时设置对象年龄为 1。对象在 Survivor 区中每「熬过」一次 Minor GC 年龄就加 1，当年龄达到一定程度（默认 15） 就会被晋升到老年代。

### 动态对象年龄判定

为了更好的适应不同程序的内存情况，虚拟机并不是永远要求对象的年龄必需达到某个固定的值（比如前面说的 15）才会被晋升到老年代，而是会去动态的判断对象年龄。如果在 Survivor 区中相同年龄所有对象大小的总和大于 Survivor 空间的一半，年龄大于等于该年龄的对象就可以直接进入老年代。

### 空间分配担保

在新生代触发 Minor GC 后，如果 Survivor 中任然有大量的对象存活就需要老年队来进行分配担保，让 Survivor 区中无法容纳的对象直接进入到老年代。在发生Minior 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HandlePermotionFailure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，将会尝试进行一次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，那这时也要进行一次Full GC。

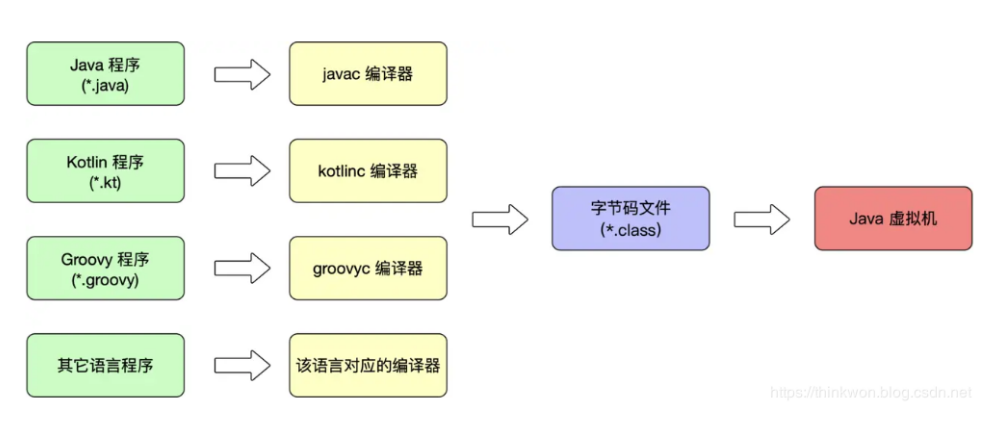
# 类文件结构

## 概述

我们所编写的每一行代码，要在机器上运行，最终都需要编译成二进制的机器码 CPU 才能识别。但是由于虚拟机的存在，屏蔽了操作系统与 CPU 指令集的差异性，类似于 Java 这种建立在虚拟机之上的编程语言通常会编译成一种中间格式的文件来存储，比如我们今天要聊的字节码（ByteCode）文件。

无关性的基石

Java 虚拟机的设计者在设计之初就考虑并实现了其它语言在 Java 虚拟机上运行的可能性。所以并不是只有 Java 语言能够跑在 Java 虚拟机上，时至今日诸如 Kotlin、Groovy、Jython、JRuby 等一大批 JVM 语言都能够在 Java 虚拟机上运行。它们和 Java 语言一样都会被编译器编译成字节码文件，然后由虚拟机来执行。所以说类文件（字节码文件）具有语言无关性。



## Class类文件的结构

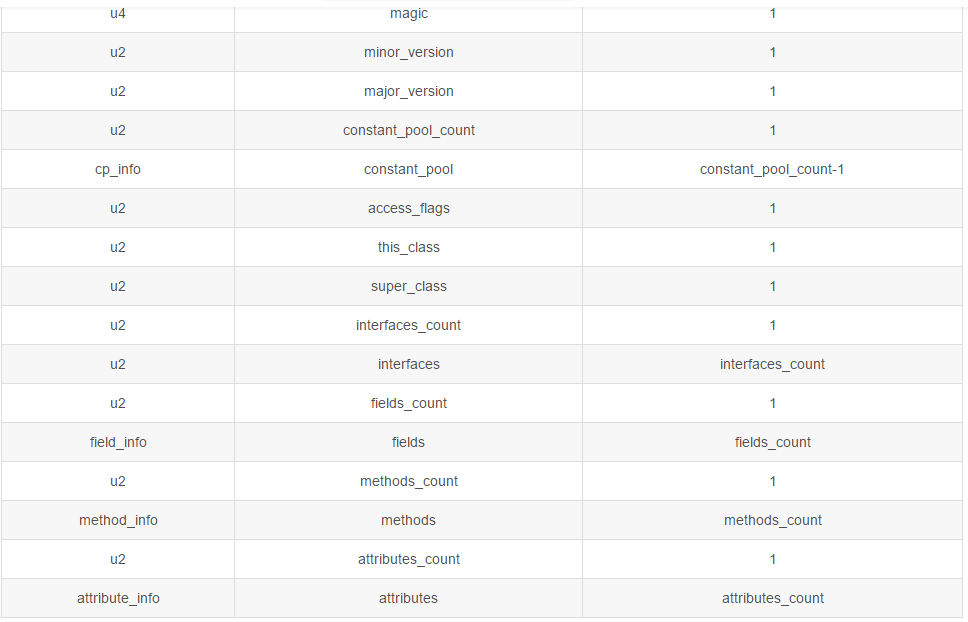
Class 文件是一组以 8 位字节为基础单位的二进制流，各个数据严格按照顺序紧凑的排列在 Class 文件中，中间无任何分隔符，这使得整个 Class 文件中存储的内容几乎全部都是程序运行的必要数据，没有空隙存在。当遇到需要占用 8 位字节以上空间的数据项时，会按照高位在前的方式分割成若干个 8 位字节进行存储。

Java 虚拟机规范规定 Class 文件格式采用一种类似与 C 语言结构体的伪结构体来存储数据，这种伪结构体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 编码结构构成的字符串值。

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 文件就是一张表，它由下表中所示的数据项构成。

类型 名称 数量



Class 文件中存储的字节严格按照上表中的顺序紧凑的排列在一起。哪个字节代表什么含义，长度是多少，先后顺序如何都是被严格限制的，不允许有任何改变。

### 魔数与 Class 文件的版本

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收的 Calss 文件。之所以使用魔数而不是文件后缀名来进行识别主要是基于安全性的考虑，因为文件后缀名是可以随意更改的。Class 文件的魔数值为「0xCAFEBABE」。

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 两个字节是次版本号（Minor Version），第 7 和第 8 个字节是主版本号（Major Version）。高版本的 JDK 能够向下兼容低版本的 Class 文件，虚拟机会拒绝执行超过其版本号的 Class 文件。

### 常量池

主版本号之后是常量池入口，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，同是它还是 Class 文件中第一个出现的表类型数据项目。

因为常量池中常量的数量是不固定的，所以在常量池入口需要放置一个 u2 类型的数据来表示常量池的容量「constant\_pool\_count」，和计算机科学中计数的方法不一样，这个容量是从 1 开始而不是从 0 开始计数。之所以将第 0 项常量空出来是为了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，这种情况可以把索引值置为 0 来表示。

Class 文件结构中只有常量池的容量计数是从 1 开始的，其它集合类型，包括接口索引集合、字段表集合、方法表集合等容量计数都是从 0 开始。

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，如字符串、声明为 final 的常量值等。

符号引用属于编译原理方面的概念，包括了以下三类常量：

类和接口的全限定名

字段的名称和描述符

方法的名称和描述符

### 访问标志

紧接着常量池之后的两个字节代表访问标志（access\_flag），这个标志用于识别一些类或者接口层次的访问信息，包括这个 Class 是类还是接口；是否定义为 public 类型；是否定义为 abstract 类型；如果是类的话，是否被申明为 final 等。具体的标志位以及标志的含义见下表：

access\_flags 中一共有 16 个标志位可以使用，当前只定义了其中的 8 个，没有使用到的标志位要求一律为 0。

类索引、父类索引与接口索引集合

类索引（this\_class）和父类索引（super\_class）都是一个 u2 类型的数据，而接口索引集合（interfaces）是一组 u2 类型的数据集合，Class 文件中由这三项数据来确定这个类的继承关系。

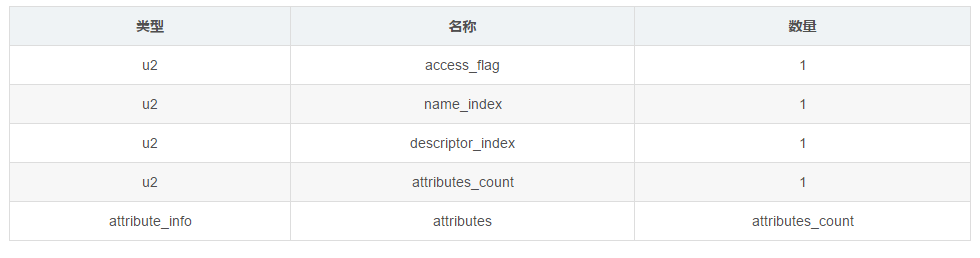
类索引用于确定这个类的全限定名

父类索引用于确定这个类的父类的全限定名

接口索引集合用于描述这个类实现了哪些接口

### 字段表集合

字段表集合（field\_info）用于描述接口或者类中声明的变量。字段（field）包括类变量和实例变量，但不包括方法内部声明的局部变量。下面我们看看字段表的结构：

字段修饰符放在 access\_flags 中，它与类中的 access\_flag 非常相似，都是一个 u2 的数据类型。

方法表集合

Class 文件中对方法的描述和对字段的描述是完全一致的，方法表中的结构和字段表的结构一样。

因为 volatile 关键字和 transient 关键字不能修饰方法，所以方法表的访问标志中没有 ACC\_VOLATILE 和 ACC\_TRANSIENT。与之相对的，synchronizes、native、strictfp 和 abstract 关键字可以修饰方法，所以方法表的访问标志中增加了 ACC\_SYNCHRONIZED、ACC\_NATIVE、ACC\_STRICTFP 和 ACC\_ABSTRACT 标志。

对于方法里的代码，经过编译器编译成字节码指令后，存放在方法属性表中一个名为「Code」的属性里面。

### 属性表集合

在 Class 文件、字段表、方法表中都可以携带自己的属性表（attribute\_info）集合，用于描述某些场景专有的信息。

属性表集合不像 Class 文件中的其它数据项要求这么严格，不强制要求各属性表的顺序，并且只要不与已有属性名重复，任何人实现的编译器都可以向属性表中写入自己定义的属性信息，Java 虚拟机在运行时会略掉它不认识的属性。

字节码指令简介

感兴趣的小伙伴可以自行阅读《深入理解Java虚拟机》

公有设计和私有实现

感兴趣的小伙伴可以自行阅读《深入理解Java虚拟机》

Class文件结构的发展

感兴趣的小伙伴可以自行阅读《深入理解Java虚拟机》

# 虚拟机类加载机制

## 概述

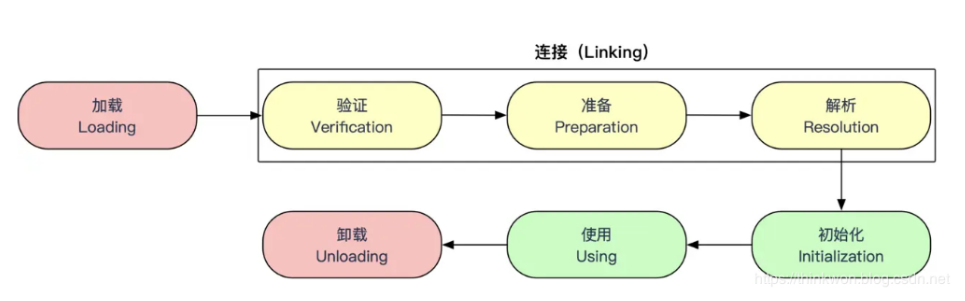
我们的源代码经过编译器编译成字节码之后，最终都需要加载到虚拟机之后才能运行。虚拟机把描述类的数据从 Class 文件加载到内存，并对数据进行校验、转换解析和初始化，最终形成可以被虚拟机直接使用的 Java 类型，这就是虚拟机的类加载机制。

与编译时需要进行连接工作的语言不同，Java 语言中类的加载、连接和初始化都是在程序运行期间完成的，这种策略虽然会让类加载时增加一些性能开销，但是会为 Java 应用程序提供高度的灵活性，Java 里天生可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的特点实现的。

例如，一个面向接口的应用程序，可以等到运行时再指定实际的实现类；用户可以通过 Java 预定义的和自定义的类加载器，让一个本地的应用程序运行从网络上或其它地方加载一个二进制流作为程序代码的一部分。

## 类加载时机

类从被虚拟机从加载到卸载，整个生命周期包含：加载（Loading）、验证（Verification）、准备（Preparation）、解析（Resolution）、初始化（Initialization）、使用（Using）、卸载（Unloading）7 个阶段。其中验证、准备、解析 3 个部分统称为连接（Linking）。这 7 个阶段的发生顺序如下图：



上图中加载、验证、准备、初始化和卸载 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「注意，这里说的是按部就班的开始，并不要求前一阶段执行完才能进入下一阶段」，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 的动态绑定。

虚拟机规范中对于什么时候开始类加载过程的第一节点「加载」并没有强制约束。但是对于「初始化」阶段，虚拟机则是严格规定了有且只有以下 5 种情况，如果类没有进行初始化，则必须立即对类进行「初始化」（加载、验证、准备自然需要在此之前开始）：

遇到 new、getstatic、putstatic 或 invokestatic 这 4 条字节码指令；

使用 java.lang.reflect 包的方法对类进行反射调用的时候；

当初始化一个类的时候，发现其父类还没有进行初始化的时候，需要先触发其父类的初始化；

当虚拟机启动时，用户需要指定一个要执行的主类，虚拟机会先初始化这个类；

当使用 JDK 1.7 的动态语言支持时，如果一个 java.lang.invoke.MethodHandle 实例最后的解析结果 REF\_getStatic、REF\_putStatic、REF\_invokeStatic 的方法句柄，并且这个方法句柄所对应的类没有初始化。

「有且只有」以上 5 种场景会触发类的初始化，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。比如如下几种场景就是被动引用：

通过子类引用父类的静态字段，不会导致子类的初始化；

通过数组定义来引用类，不会触发此类的初始化；

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；

## 类加载过程

### 加载

这里的「加载」是指「类加载」过程的一个阶段。在加载阶段，虚拟机需要完成以下 3 件事：

通过一个类的全限定名来获取定义此类的二进制字节流；

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；

在内存中生成一个代表这个类的 java.lang.Class 对象，作为方法区这个类的各种数据的访问入口。

### 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下面 4 个阶段的检验动作：

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，并且能够被当前版本的虚拟机处理。验证点主要包括：是否以魔数 0xCAFEBABE 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；Class 文件中各个部分及文件本身是否有被删除的或者附加的其它信息等等。

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，这个阶段的验证点包括：这个类是否有父类；这个类的父类是否继承了不允许被继承的类；如果这个类不是抽象类，是否实现了其父类或者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矛盾等等。

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。

符号引用验证：最后一个阶段的校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）的形象进行匹配性校验。

### 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进行分配。这个阶段中有两个容易产生混淆的概念需要强调下：

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」下是数据类型的零值。假设一个类变量的定义为public static int value = 123; 那么变量 value 在准备阶段过后的初始值为 0 而不是 123，因为这个时候尚未执行任何 Java 方法，而把 value 赋值为 123 的 putstatic 指令是程序被编译之后，存放于类构造器 () 方法之中，所以把 value 赋值为 123 的动作将在初始化阶段才会执行。

这里提到，在「通常情况」下初始值是零值，那相对的会有一些「特殊情况」：如果类字段的字段属性表中存在 ConstantsValue 属性，那在准备阶段变量 value 就会被初始化为 ConstantValue 属性所指的值。假设上面的类变量 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。

### 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。前面提到过很多次符号引用和直接引用，那么到底什么是符号引用和直接引用呢？

符号引用（Symbolic Reference）：符号引用以一组符号来描述所引用的目标，符号可以上任何形式的字面量，只要使用时能无歧义地定位到目标即可。

直接引用（Direct Reference）：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。

### 初始化

类初始化阶段是类加载过程中的最后一步，前面的类加载过程中，除了在加载阶段用户应用程序可以通过自定义类加载器参与之外，其余动作完全是由虚拟机主导和控制的。到了初始化阶段，才真正开始执行类中定义的 Java 程序代码。初始阶段是执行类构造器 <clinit> () 方法的过程。

## 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 Java 虚拟机外部去实现，以便让应用程序自己决定如何去获取所需要的类。实现这个动作的代码模块称为「类加载器」。

类加载器：类加载器负责加载程序中的类型（类和接口），并赋予唯一的名字予以标识。

### 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这个类本身一同确立其在 Java 虚拟机的唯一性，每个类加载器都拥有一个独立的类名称空间。也就是说：比较两个类是否「相等」，只要在这两个类是由同一个类加载器加载的前提下才有意义，否则，即使这两个类来源于同一个 Class 文件，被同一个虚拟机加载，只要加载它们的类加载器不同，那这两个类就必定不相等。

### 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：一种是启动类加载器（Bootstrap ClassLoader），这个类加载器使用 C++ 来实现，是虚拟机自身的一部分；另一种就是所有其他的类加载器，这些类加载器都由 Java 来实现，独立于虚拟机外部，并且全都继承自抽象类 java.lang.ClassLoader。

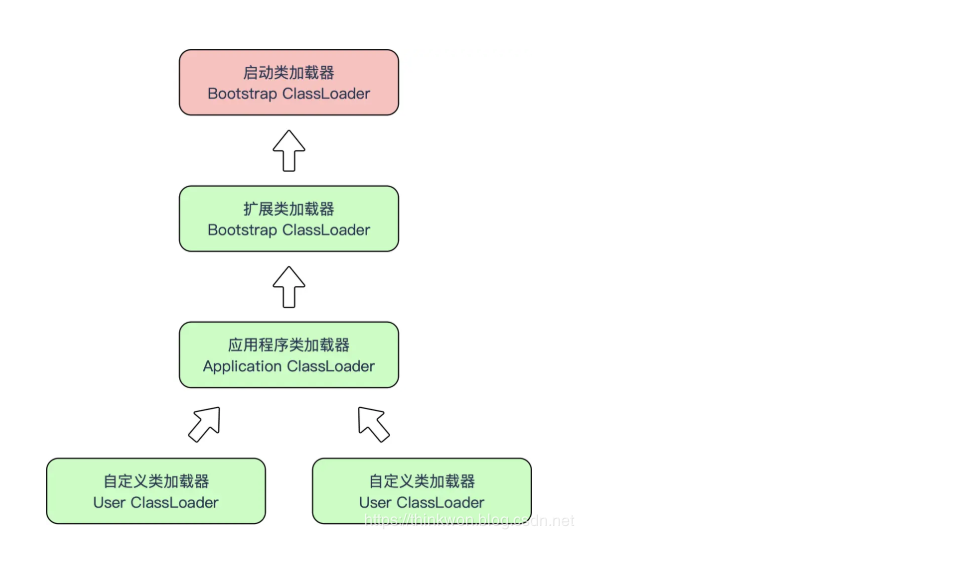
从 Java 开发者的角度来看，类加载器可以划分为：

启动类加载器（Bootstrap ClassLoader）：这个类加载器负责将存放在 <JAVA\_HOME>\lib 目录中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，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启动类加载器，纳智捷使用 null 代替即可；

扩展类加载器（Extension ClassLoader）：这个类加载器由 sun.misc.Launcher$ExtClassLoader 实现，它负责加载 <JAVA\_HOME>\lib\ext 目录中，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；

应用程序类加载器（Application ClassLoader）：这个类加载器由 sun.misc.Launcher$App-ClassLoader 实现。getSystemClassLoader() 方法返回的就是这个类加载器，因此也被称为系统类加载器。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类加载器。

我们的应用程序都是由这 3 种类加载器互相配合进行加载的，在必要时还可以自己定义类加载器。它们的关系如下图所示：



上图中所呈现出的这种层次关系，称为类加载器的双亲委派模型（Parents Delegation Model）。双亲委派模型要求除了顶层的启动类加载器以外，其余的类加载器都应当有自己的父类加载器。

### 类加载器的关系

Bootstrap Classloader 是在Java虚拟机启动后初始化的。

Bootstrap Classloader 负责加载 ExtClassLoader，并且将 ExtClassLoader的父加载器设置为 Bootstrap Classloader

Bootstrap Classloader 加载完 ExtClassLoader 后，就会加载 AppClassLoader，并且将 AppClassLoader 的父加载器指定为 ExtClassLoader。

类加载器的作用



类加载器的特点

层级结构：Java里的类装载器被组织成了有父子关系的层级结构。Bootstrap类装载器是所有装载器的父亲。

代理模式： 基于层级结构，类的代理可以在装载器之间进行代理。当装载器装载一个类时，首先会检查它在父装载器中是否进行了装载。如果上层装载器已经装载了这个类，这个类会被直接使用。反之，类装载器会请求装载这个类

可见性限制：一个子装载器可以查找父装载器中的类，但是一个父装载器不能查找子装载器里的类。

不允许卸载：类装载器可以装载一个类但是不可以卸载它，不过可以删除当前的类装载器，然后创建一个新的类装载器装载。

### 类加载器的隔离问题

每个类装载器都有一个自己的命名空间用来保存已装载的类。当一个类装载器装载一个类时，它会通过保存在命名空间里的类全局限定名(Fully Qualified Class Name) 进行搜索来检测这个类是否已经被加载了。

JVM 及 Dalvik 对类唯一的识别是 ClassLoader id + PackageName + ClassName，所以一个运行程序中是有可能存在两个包名和类名完全一致的类的。并且如果这两个类不是由一个 ClassLoader 加载，是无法将一个类的实例强转为另外一个类的，这就是 ClassLoader 隔离性。

为了解决类加载器的隔离问题，JVM引入了双亲委托机制。

双亲委派模型的工作过程是这样的：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，只有当父类加载器反馈自己无法完成这个类加载请求（它的搜索范围中没有找到所需的类）时，子加载器才会尝试自己去加载。

这样做的好处就是 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。例如 java.lang.Object，它放在 rt.jar 中，无论哪一个类加载器要加载这个类，最终都是委派给处于模型顶端的启动类加载器来加载，因此 Object 类在程序的各种类加载器环境中都是同一个类。相反，如果没有使用双亲委派模型，由各个类加载器自行去加载的话，如果用户自己编写了一个称为 java.lang.Object 的类，并放在程序的 ClassPath 中，那系统中将会出现多个不同的 Object 类，Java 类型体系中最基本的行为也就无法保证了。

双亲委派模型对于保证 Java 程序运行的稳定性很重要，但它的实现很简单，实现双亲委派模型的代码都集中在 java.lang.ClassLoader 的 loadClass() 方法中，逻辑很清晰：先检查是否已经被加载过，若没有则调用父类加载器的 loadClass() 方法，若父加载器为空则默认使用启动类加载器作为父加载器。如果父类加载失败，抛出 ClassNotFoundException 异常后，再调用自己的 findClass() 方法进行加载。

关于类文件结构和类加载就通过连续的两篇文章介绍到这里了，下一篇我们来聊聊「虚拟机的字节码执行引擎」。

破坏双亲委派模型

感兴趣的小伙伴可以自行阅读《深入理解Java虚拟机》

# 字节码执行引擎

## 概述

执行引擎是 Java 虚拟机最核心的组成部分之一。「虚拟机」是相对于「物理机」的概念，这两种机器都有代码执行的能力，区别是物理机的执行引擎是直接建立在处理器、硬件、指令集和操作系统层面上的，而虚拟机执行引擎是由自己实现的，因此可以自行制定指令集与执行引擎的结构体系，并且能够执行那些不被硬件直接支持的指令集格式。

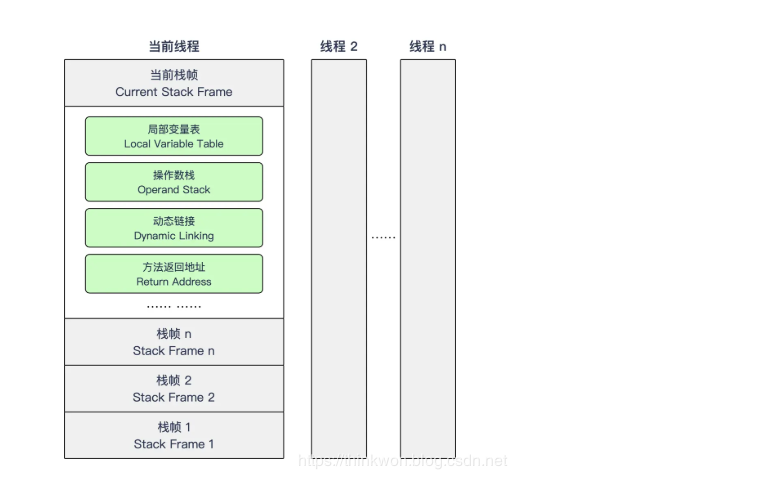
在 Java 虚拟机规范中制定了虚拟机字节码执行引擎的概念模型，这个概念模型成为各种虚拟机执行引擎的统一外观（Facade）。在不同的虚拟机实现里，执行引擎在执行 Java 代码的时候可能会有解释执行（通过解释器执行）和编译执行（通过即时编译器产生本地代码执行）两种方式，也可能两者都有，甚至还可能会包含几个不同级别的编译器执行引擎。但从外观上来看，所有 Java 虚拟机的执行引擎是一致的：输入的是字节码文件，处理过程是字节码解析的等效过程，输出的是执行结果。

## 运行时栈帧结构

栈帧（Stack Frame）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、操作数栈、动态链接和方法返回地址等信息。每一个方法从调用开始到执行完成的过程，都对应着一个栈帧在虚拟机栈里从入栈到出栈的过程。

每一个栈帧都包括了局部变量表、操作数栈、动态链接、方法返回地址和一些额外的附加信息。在编译程序代码时，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，而仅仅取决于具体的虚拟机实现。

一个线程中的方法调用链可能会很长，很多方法都处于执行状态。对于执行引擎来说，在活动线程中，只有位于栈顶的栈帧才是有效的，称为当前栈帧（Current Stack Frame），与这个栈帧相关联的方法成为当前方法。执行引擎运行的所有字节码指令对当前栈帧进行操作，在概念模型上，典型的栈帧结构如下图：



## 局部变量表

局部变量表（Local Variable Table）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 程序中编译为 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。

### 操作数栈

操作数栈（Operand Stack）是一个后进先出栈。同局部变量表一样，操作数栈的最大深度也在编译阶段写入到 Code 属性的 max\_stacks 数据项中。操作数栈的每一个元素可以是任意的 Java 数据类型，包括 long 和 double。32 位数据类型所占的栈容量为 1，64 位数据类型所占的栈容量为 2。在方法执行的任何时候，操作数栈的深度都不会超过 max\_stacks 数据项中设定的最大值。

一个方法刚开始执行的时候，该方法的操作数栈是空的，在方法的执行过程中，会有各种字节码指令往操作数栈中写入和提取内容，也就是入栈和出栈操作。

### 动态链接

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链接（Dynamic Linking）。Class 文件的常量池中存在大量的符号引用，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第一次使用时转化为直接引用，这种转化成为静态解析。另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。

### 方法返回地址

当一个方法开始执行后，只有两种方式可以退出这个方法。

一种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方法的调用者，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称为正常完成出口。

另一种退出方式是，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。这种称为异常完成出口。一个方法使用异常完成出口的方式退出，是不会给上层调用者产生任何返回值的。

无论采用何种退出方式，在方法退出后都需要返回到方法被调用的位置，程序才能继续执行，方法返回时可能需要在栈帧中保存一些信息，用来恢复它的上层方法的执行状态。一般来说，方法正常退出时，调用者的 PC 计数器的值可以作为返回地址，栈帧中很可能会保存这个计数器值。而方法异常退出时，返回地址是要通过异常处理器表来确定的，栈帧中一般不会保存这部分信息。

方法退出的过程实际上就等同于把当前栈帧出栈，因此退出时可能执行的操作有：恢复上次方法的局部变量表和操作数栈，把返回值（如果有的话）压入调用者栈帧的操作数栈中，调整 PC 计数器的值以指向方法调用指令后面的一条指令等。

### 附加信息

虚拟机规范允许具体的虚拟机实现增加一些规范里没有描述的信息到栈帧中，例如与调试相关的信息，这部分信息完全取决于具体的虚拟机实现。实际开发中，一般会把动态连接、方法返回地址与其他附加信息全部归为一类，成为栈帧信息。

### 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。

在程序运行时，进行方法调用是最为普遍、频繁的操作。前面说过 Class 文件的编译过程是不包含传统编译中的连接步骤的，一切方法调用在 Class 文件里面存储的都只是符号引用，而不是方法在运行时内存布局中的入口地址（相当于之前说的直接引用）。这个特性给 Java 带来了更强大的动态扩展能力，但也使得 Java 方法调用过程变得相对复杂起来，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。

### 解析

所有方法调用中的目标方法在 Class 文件里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，并且这个方法的调用版本在运行期是不可改变的。话句话说，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（Resolution）。

Java 语言中符合「编译器可知，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者别的方式重写其它版本，因此它们都适合在类加载阶段解析。

与之相应的是，在 Java 虚拟机里提供了 5 条方法调用字节码指令，分别是：

invokestatic：调用静态方法；

invokespecial：调用实例构造器 方法、私有方法和父类方法；

invokevirtual：调用所有虚方法；

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。

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直接引用。这些方法可以称为非虚方法，与之相反，其它方法称为虚方法（final 方法除外）。

Java 中的非虚方法除了使用 invokestatic、invokespecial 调用的方法之外还有一种，就是被 final 修饰的方法。虽然 final 方法是使用 invokevirtual 指令来调用的，但是由于它无法被覆盖，没有其它版本，所以也无需对方法接受者进行多态选择，又或者说多态选择的结果肯定是唯一的。在 Java 语言规范中明确说明了 final 方法是一种非虚方法。

解析调用一定是个静态过程，在编译期间就能完全确定，在类装载的解析阶段就会把涉及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。而分派（Dispatch）调用则可能是静态的也可能是动态的，根据分派依据的宗量数可分为单分派和多分派。这两类分派方式的两两组合就构成了静态单分派、静态多分派、动态单分派、动态多分派 4 种分派组合情况，下面我们再看看虚拟机中的方法分派是如何进行的。

## 分派

面向对象有三个基本特征，封装、继承和多态。这里要说的分派将会揭示多态特征的一些最基本的体现，如「重载」和「重写」在 Java 虚拟机中是如何实现的？虚拟机是如何确定正确目标方法的？

### 静态分派

在开始介绍静态分派前我们先看一段代码。



运行后这段程序的输出结果如下：

Hello, guy!

Hello, guy!

1

2

稍有经验的 Java 程序员都能得出上述结论，但为什么我们传递给 sayHello() 方法的实际参数类型是 Man 和 Woman，虚拟机在执行程序时选择的却是 Human 的重载呢？要理解这个问题，我们先弄清两个概念。

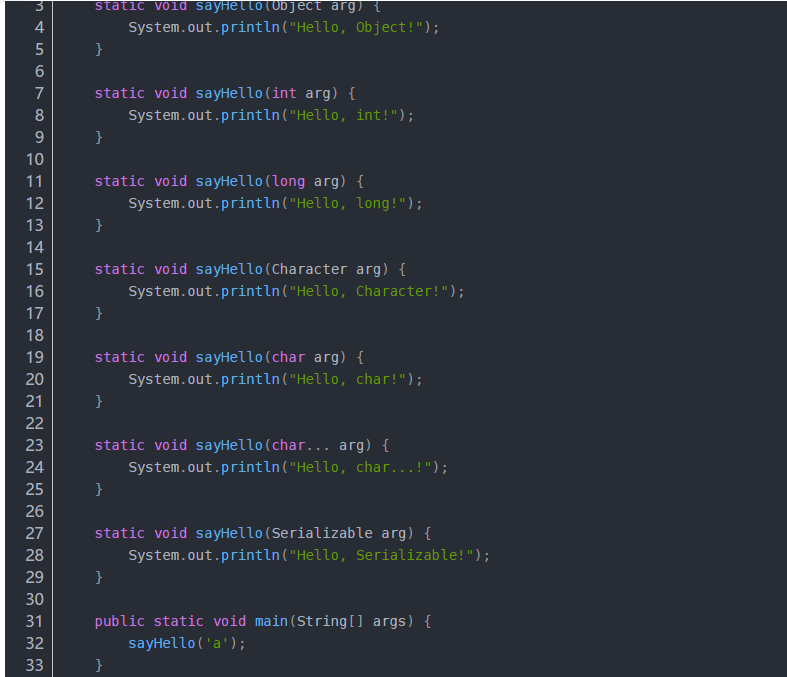
Human man = new Man();

1

上面这段代码中的「Human」称为变量的静态类型（Static Type），或者叫做外观类型（Apparent Type），后面的「Man」称为变量为实际类型（Actual Type），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发生在使用时，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。

弄清了这两个概念，再来看 StaticDispatch 类中 main() 方法里的两次 sayHello() 调用，在方法接受者已经确定是对象「dispatch」的前提下，使用哪个重载版本，就完全取决于传入参数的数量和数据类型。代码中定义了两个静态类型相同但是实际类型不同的变量，但是虚拟机（准确的说是编译器）在重载时是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期可知的，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，所以选择了 sayHello(Human) 作为调用目标，并把这个方法的符号引用写到 man() 方法里的两条 invokevirtual 指令的参数中。

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。静态分派发生在编译阶段，因此确定静态分派的动作实际上不是由虚拟机来执行的。

另外，编译器虽然能确定方法的重载版本，但是很多情况下这个重载版本并不是「唯一」的，因此往往只能确定一个「更加合适」的版本。产生这种情况的主要原因是字面量不需要定义，所以字面量没有显示的静态类型，它的静态类型只能通过语言上的规则去理解和推断。下面的代码展示了什么叫「更加合适」的版本。

上面代码的运行结果为：

Hello, char!

1

这很好理解，‘a’ 是一个 char 类型的数据，自然会寻找参数类型为 char 的重载方法，如果注释掉 sayHello(chat arg) 方法，那么输出结果将会变为：

Hello, int!

1

这时发生了一次类型转换， ‘a’ 除了可以代表一个字符，还可以代表数字 97，因为字符 ‘a’ 的 Unicode 数值为十进制数字 97，因此参数类型为 int 的重载方法也是合适的。我们继续注释掉 sayHello(int arg) 方法，输出变为：

Hello, long!

1

这时发生了两次类型转换，‘a’ 转型为整数 97 之后，进一步转型为长整型 97L，匹配了参数类型为 long 的重载方法。我们继续注释掉 sayHello(long arg) 方法，输出变为：

Hello, Character!

1

这时发生了一次自动装箱， ‘a’ 被包装为它的封装类型 java.lang.Character，所以匹配到了类型为 Character 的重载方法，继续注释掉 sayHello(Character arg) 方法，输出变为：

Hello, Serializable!

1

这里输出之所以为「Hello, Serializable!」，是因为 java.lang.Serializable 是 java.lang.Character 类实现的一个接口，当自动装箱后发现还是找不到装箱类，但是找到了装箱类实现了的接口类型，所以紧接着又发生了一次自动转换。char 可以转型为 int，但是 Character 是绝对不会转型为 Integer 的，他只能安全的转型为它实现的接口或父类。Character 还实现了另外一个接口 java.lang.Comparable，如果同时出现两个参数分别为 Serializable 和 Comparable 的重载方法，那它们在此时的优先级是一样的。编译器无法确定要自动转型为哪种类型，会提示类型模糊，拒绝编译。程序必须在调用时显示的指定字面量的静态类型，如：sayHello((Comparable) ‘a’)，才能编译通过。继续注释掉 sayHello(Serializable arg) 方法，输出变为：

Hello, Object!

1

这时是 char 装箱后转型为父类了，如果有多个父类，那将在继承关系中从下往上开始搜索，越接近上层的优先级越低。即使方法调用的入参值为 null，这个规则依然适用。继续注释掉 sayHello(Serializable arg) 方法，输出变为：

Hello, char...!

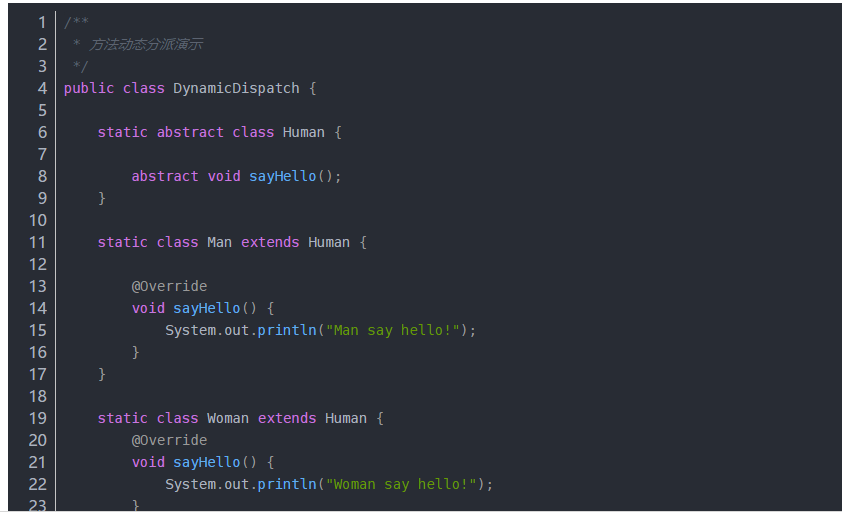
1

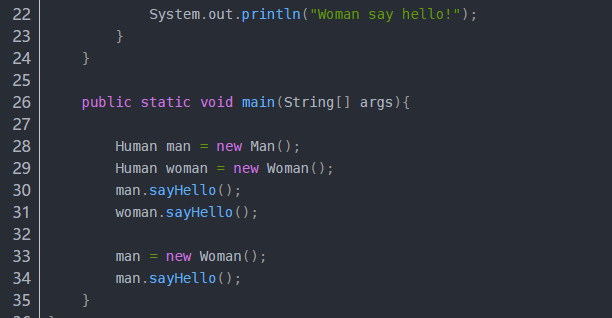
7 个重载方法以及被注释得只剩一个了，可见变长参数的重载优先级是最低的，这时字符 ‘a’ 被当成了一个数组元素。

前面介绍的这一系列过程演示了编译期间选择静态分派目标的过程，这个过程也是 Java 语言实现方法重载的本质。

### 动态分派

动态分派和多态性的另一个重要体现「重写（Override）」有着密切的关联，我们依旧通过代码来理解什么是动态分派。





代码执行结果：

Man say hello!

Woman say hello!

Woman say hello!

1

2

3

对于上面的代码，虚拟机是如何确定要调用哪个方法的呢？显然这里不再通过静态类型来决定了，因为静态类型同样都是 Human 的两个变量 man 和 woman 在调用 sayHello() 方法时执行了不同的行为，并且变量 man 在两次调用中执行了不同的方法。导致这个结果的原因是因为它们的实际类型不同。对于虚拟机是如何通过实际类型来分派方法执行版本的，这里我们就不做介绍了，有兴趣的可以去看看原著。

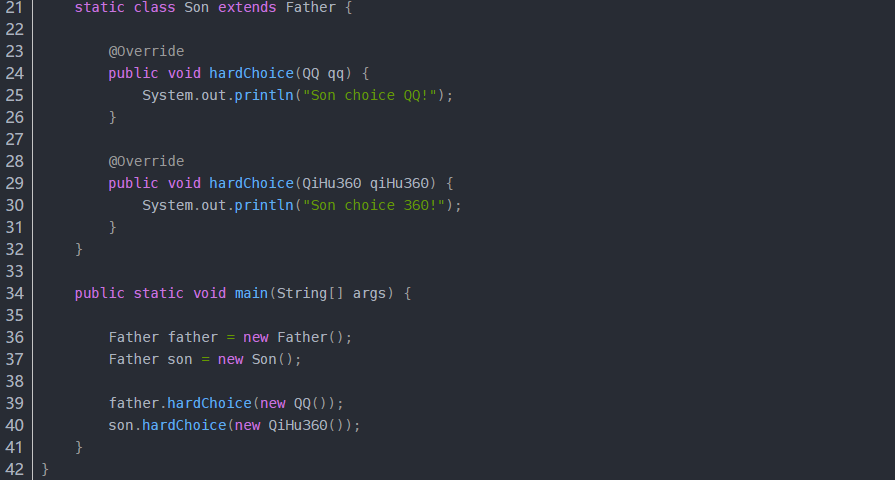
我们把这种在运行期根据实际类型来确定方法执行版本的分派称为动态分派。

### 单分派和多分派

方法的接收者和方法的参数统称为方法的宗量，这个定义最早来源于《Java 与模式》一书。根据分派基于多少宗量，可将分派划分为单分派和多分派。

单分派是根据一个宗量来确定方法的执行版本；多分派则是根据多余一个宗量来确定方法的执行版本。

我们依旧通过代码来理解(代码以著名的 3Q 大战作为背景)：



代码输出结果：

Father choice QQ!

Son choice 360!

1

2

我们先来看看编译阶段编译器的选择过程，也就是静态分派过程。这个时候选择目标方法的依据有两点：一是静态类型是 Father 还是 Son；二是方法入参是 QQ 还是 QiHu360。因为是根据两个宗量进行选择的，所以 Java 语言的静态分派属于多分派。

再看看运行阶段虚拟机的选择过程，也就是动态分派的过程。在执行 son.hardChoice(new QiHu360()) 时，由于编译期已经确定目标方法的签名必须为 hardChoice(QiHu360)，这时参数的静态类型、实际类型都不会对方法的选择造成任何影响，唯一可以影响虚拟机选择的因数只有此方法的接收者的实际类型是 Father 还是 Son。因为只有一个宗量作为选择依据，所以 Java 语言的动态分派属于单分派。

综上所述，Java 语言是一门静态多分派、动态单分派的语言。

动态类型语言支持

感兴趣的小伙伴可以自行阅读《深入理解Java虚拟机》

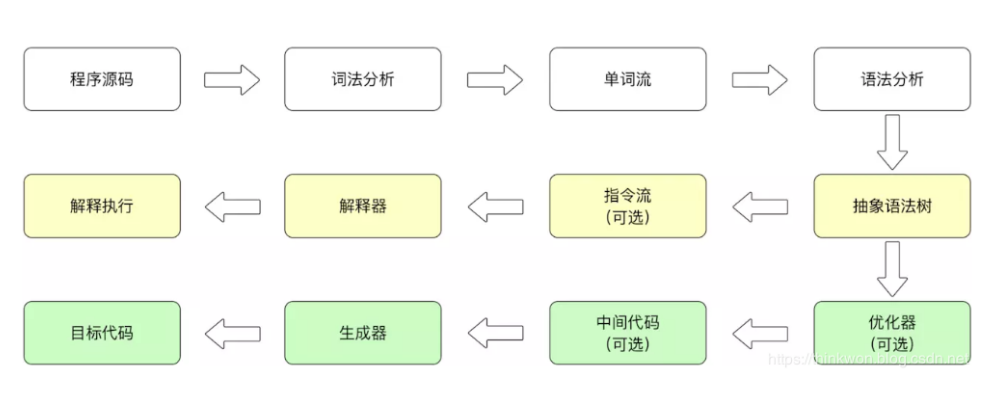
## 基于栈的字节码解释执行引擎

虚拟机如何调用方法已经介绍完了，下面我们来看看虚拟机是如何执行方法中的字节码指令的。

### 解释执行

Java 语言常被人们定义成「解释执行」的语言，但随着 JIT 以及可直接将 Java 代码编译成本地代码的编译器的出现，这种说法就不对了。只有确定了谈论对象是某种具体的 Java 实现版本和执行引擎运行模式时，谈解释执行还是编译执行才会比较确切。

无论是解释执行还是编译执行，无论是物理机还是虚拟机，对于应用程序，机器都不可能像人一样阅读、理解，然后获得执行能力。大部分的程序代码到物理机的目标代码或者虚拟机执行的指令之前，都需要经过下图中的各个步骤。下图中最下面的那条分支，就是传统编译原理中程序代码到目标机器代码的生成过程；中间那条分支，则是解释执行的过程。



如今，基于物理机、Java 虚拟机或者非 Java 的其它高级语言虚拟机的语言，大多都会遵循这种基于现代编译原理的思路，在执行前先对程序源代码进行词法分析和语法分析处理，把源代码转化为抽象语法树。对于一门具体语言的实现来说，词法分析、语法分析以至后面的优化器和目标代码生成器都可以选择独立于执行引擎，形成一个完整意义的编译器去实现，这类代表是 C/C++。也可以为一个半独立的编译器，这类代表是 Java。又或者把这些步骤和执行全部封装在一个封闭的黑匣子中，如大多数的 JavaScript 执行器。

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。

许多 Java 虚拟机的执行引擎在执行 Java 代码的时候都有解释执行（通过解释器执行）和编译执行（通过即时编译器产生本地代码执行）两种选择。而对于最新的 Android 版本的执行模式则是 AOT + JIT + 解释执行，关于这方面我们后面有机会再聊。

### 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，基本上是一种基于栈的指令集架构。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的要受到硬件约束。栈架构的指令集还有一些其他优点，比如相对更加紧凑（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）、编译实现更加简单（不需要考虑空间分配的问题，所有空间都是在栈上操作）等。

栈架构指令集的主要缺点是执行速度相对来说会稍慢一些。所有主流物理机的指令集都是寄存器架构也从侧面印证了这一点。

虽然栈架构指令集的代码非常紧凑，但是完成相同功能需要的指令集数量一般会比寄存器架构多，因为出栈、入栈操作本身就产生了相当多的指令数量。更重要的是，栈实现在内存中，频繁的栈访问也意味着频繁的内存访问，相对于处理器来说，内存始终是执行速度的瓶颈。由于指令数量和内存访问的原因，所以导致了栈架构指令集的执行速度会相对较慢。

正是基于上述原因，Android 虚拟机中采用了基于寄存器的指令集架构。不过有一点不同的是，前面说的是物理机上的寄存器，而 Android 上指的是虚拟机上的寄存器。

基于栈的解释器执行过程

感兴趣的小伙伴可以自行阅读《深入理解Java虚拟机》

类加载及执行子系统的案例与实战

# DK 监控和故障处理工具总结

## JDK 命令行工具

这些命令在 JDK 安装目录下的 bin 目录下：

* **jps** (JVM Process Status）: 类似 UNIX 的 ps 命令。用户查看所有 Java 进程的启动类、传入参数和 Java 虚拟机参数等信息；
* **jstat**（ JVM Statistics Monitoring Tool）: 用于收集 HotSpot 虚拟机各方面的运行数据;
* **jinfo** (Configuration Info for Java) : Configuration Info forJava,显示虚拟机配置信息;
* **jmap** (Memory Map for Java) :生成堆转储快照;
* **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):生成虚拟机当前时刻的线程快照，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。

### jps:查看所有 Java 进程

jps(JVM Process Status) 命令类似 UNIX 的 ps 命令。

jps：显示虚拟机执行主类名称以及这些进程的本地虚拟机唯一 ID（Local Virtual Machine Identifier,LVMID）。jps -q ：只输出进程的本地虚拟机唯一 ID。



jps -v：输出虚拟机进程启动时 JVM 参数。

jps -m：输出传递给 Java 进程 main() 函数的参数。

### jstat: 监视虚拟机各种运行状态信息

jstat（JVM Statistics Monitoring Tool） 使用于监视虚拟机各种运行状态信息的命令行工具。 它可以显示本地或者远程（需要远程主机提供 RMI 支持）虚拟机进程中的类信息、内存、垃圾收集、JIT 编译等运行数据，在没有 GUI，只提供了纯文本控制台环境的服务器上，它将是运行期间定位虚拟机性能问题的首选工具。

**jstat 命令使用格式：**

jstat -<option> [-t] [-h<lines>] <vmid> [<interval> [<count>]]

比如 jstat -gc -h3 31736 1000 10表示分析进程 id 为 31736 的 gc 情况，每隔 1000ms 打印一次记录，打印 10 次停止，每 3 行后打印指标头部。

**常见的 option 如下：**

* jstat -class vmid ：显示 ClassLoader 的相关信息；
* jstat -compiler vmid ：显示 JIT 编译的相关信息；
* jstat -gc vmid ：显示与 GC 相关的堆信息；
* jstat -gccapacity vmid ：显示各个代的容量及使用情况；
* jstat -gcnew vmid ：显示新生代信息；
* jstat -gcnewcapcacity vmid ：显示新生代大小与使用情况；
* jstat -gcold vmid ：显示老年代和永久代的信息；
* jstat -gcoldcapacity vmid ：显示老年代的大小；
* jstat -gcpermcapacity vmid ：显示永久代大小；
* jstat -gcutil vmid ：显示垃圾收集信息；

另外，加上 -t参数可以在输出信息上加一个 Timestamp 列，显示程序的运行时间。

## jmap:生成堆转储快照



访问 <http://localhost:7000/>

### jstack :生成虚拟机当前时刻的线程快照

jstack（Stack Trace for Java）命令用于生成虚拟机当前时刻的线程快照。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.

生成线程快照的目的主要是定位线程长时间出现停顿的原因，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导致线程长时间停顿的原因。线程出现停顿的时候通过jstack来查看各个线程的调用堆栈，就可以知道没有响应的线程到底在后台做些什么事情，或者在等待些什么资源。

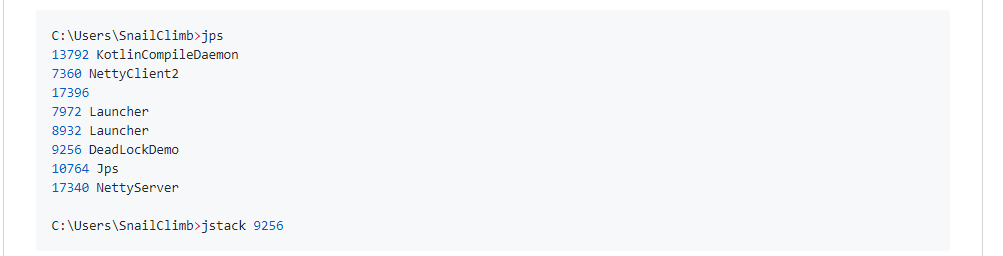
**下面是一个线程死锁的代码。我们下面会通过 jstack 命令进行死锁检查，输出死锁信息，找到发生死锁的线程。**





线程 A 通过 synchronized (resource1) 获得 resource1 的监视器锁，然后通过 Thread.sleep(1000);让线程 A 休眠 1s 为的是让线程 B 得到执行然后获取到 resource2 的监视器锁。线程 A 和线程 B 休眠结束了都开始企图请求获取对方的资源，然后这两个线程就会陷入互相等待的状态，这也就产生了死锁。

**通过 jstack 命令分析：**



输出的部分内容如下：

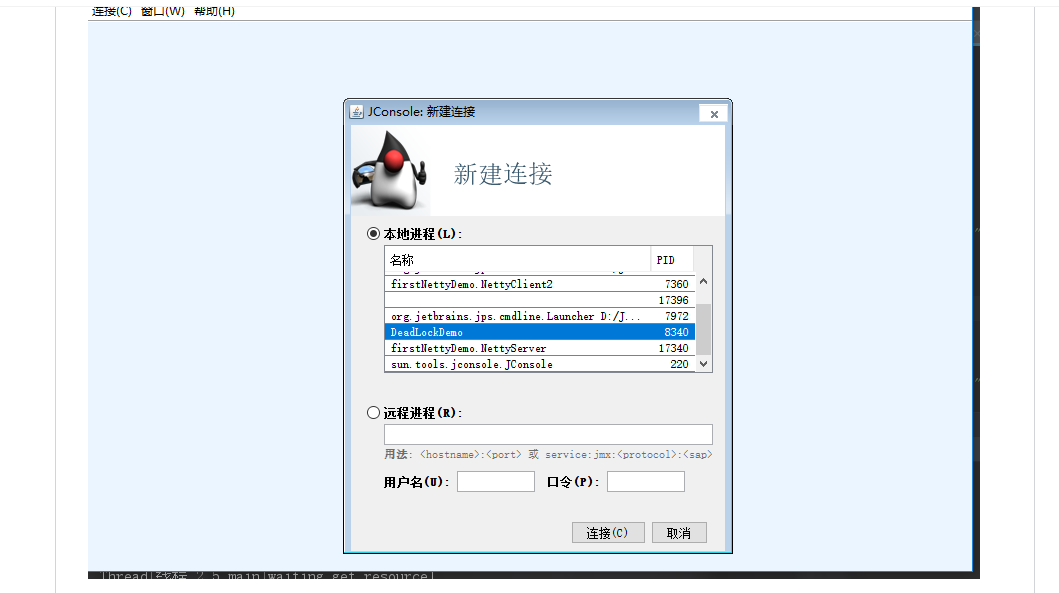


可以看到 jstack 命令已经帮我们找到发生死锁的线程的具体信息。

## JDK 可视化分析工具

### JConsole:Java 监视与管理控制台

JConsole 是基于 JMX 的可视化监视、管理工具。可以很方便的监视本地及远程服务器的 java 进程的内存使用情况。你可以在控制台输出console命令启动或者在 JDK 目录下的 bin 目录找到jconsole.exe然后双击启动。



**连接 Jconsole**

如果需要使用 JConsole 连接远程进程，可以在远程 Java 程序启动时加上下面这些参数:

-Djava.rmi.server.hostname=外网访问 ip 地址

-Dcom.sun.management.jmxremote.port=60001 //监控的端口号

-Dcom.sun.management.jmxremote.authenticate=false //关闭认证

-Dcom.sun.management.jmxremote.ssl=false

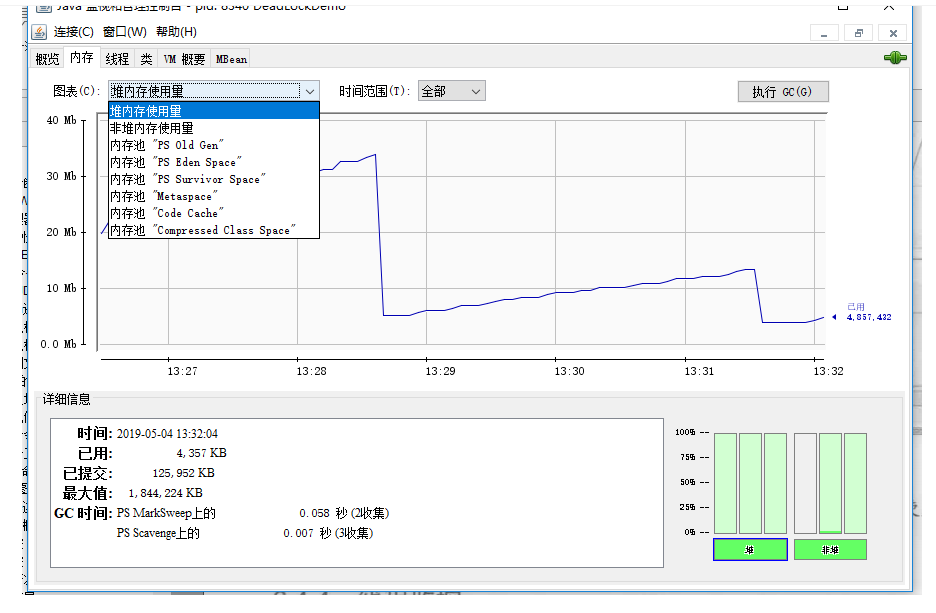
在使用 JConsole 连接时，远程进程地址如下：

外网访问 ip 地址:60001

**查看 Java 程序概况**



**内存监控**

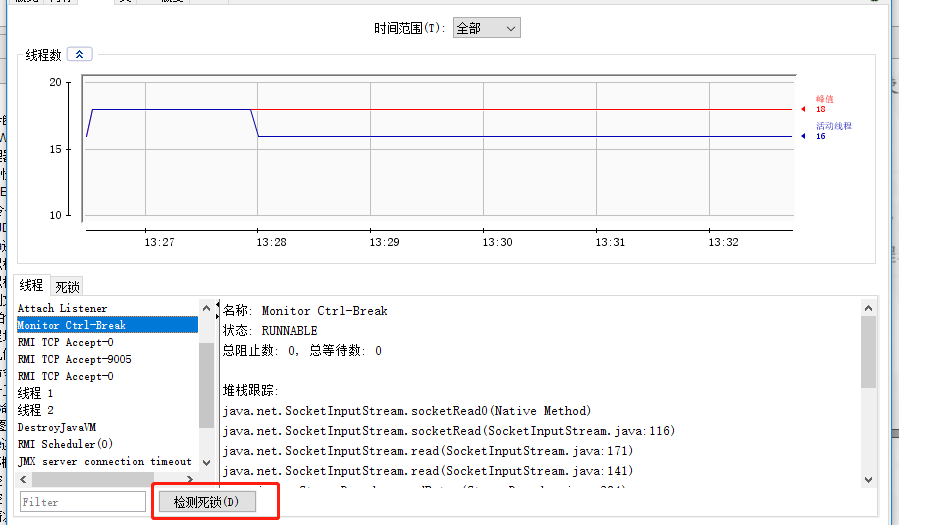


JConsole 可以显示当前内存的详细信息。不仅包括堆内存/非堆内存的整体信息，还可以细化到 eden 区、survivor 区等的使用情况，如下图所示。

点击右边的“执行 GC(G)”按钮可以强制应用程序执行一个 Full GC。

* **新生代 GC（Minor GC）**:指发生新生代的的垃圾收集动作，Minor GC 非常频繁，回收速度一般也比较快。
* **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（并非绝对），Major GC 的速度一般会比 Minor GC 的慢 10 倍以上。

**线程监控**



类似我们前面讲的 jstack 命令，不过这个是可视化的。

最下面有一个"检测死锁 (D)"按钮，点击这个按钮可以自动为你找到发生死锁的线程以及它们的详细信息 。

## Visual VM:多合一故障处理工具

VisualVM 提供在 Java 虚拟机 (Java Virutal Machine, JVM) 上运行的 Java 应用程序的详细信息。在 VisualVM 的图形用户界面中，您可以方便、快捷地查看多个 Java 应用程序的相关信息。Visual VM 官网：<https://visualvm.github.io/> 。Visual VM 中文文档:<https://visualvm.github.io/documentation.html>。

下面这段话摘自《深入理解 Java 虚拟机》。

VisualVM（All-in-One Java Troubleshooting Tool）是到目前为止随 JDK 发布的功能最强大的运行监视和故障处理程序，官方在 VisualVM 的软件说明中写上了“All-in-One”的描述字样，预示着他除了运行监视、故障处理外，还提供了很多其他方面的功能，如性能分析（Profiling）。VisualVM 的性能分析功能甚至比起 JProfiler、YourKit 等专业且收费的 Profiling 工具都不会逊色多少，而且 VisualVM 还有一个很大的优点：不需要被监视的程序基于特殊 Agent 运行，因此他对应用程序的实际性能的影响很小，使得他可以直接应用在生产环境中。这个优点是 JProfiler、YourKit 等工具无法与之媲美的。

VisualVM 基于 NetBeans 平台开发，因此他一开始就具备了插件扩展功能的特性，通过插件扩展支持，VisualVM 可以做到：

* **显示虚拟机进程以及进程的配置、环境信息（jps、jinfo）。**
* **监视应用程序的 CPU、GC、堆、方法区以及线程的信息（jstat、jstack）。**
* **dump 以及分析堆转储快照（jmap、jhat）。**
* **方法级的程序运行性能分析，找到被调用最多、运行时间最长的方法。**
* **离线程序快照：收集程序的运行时配置、线程 dump、内存 dump 等信息建立一个快照，可以将快照发送开发者处进行 Bug 反馈。**
* **其他 plugins 的无限的可能性......**