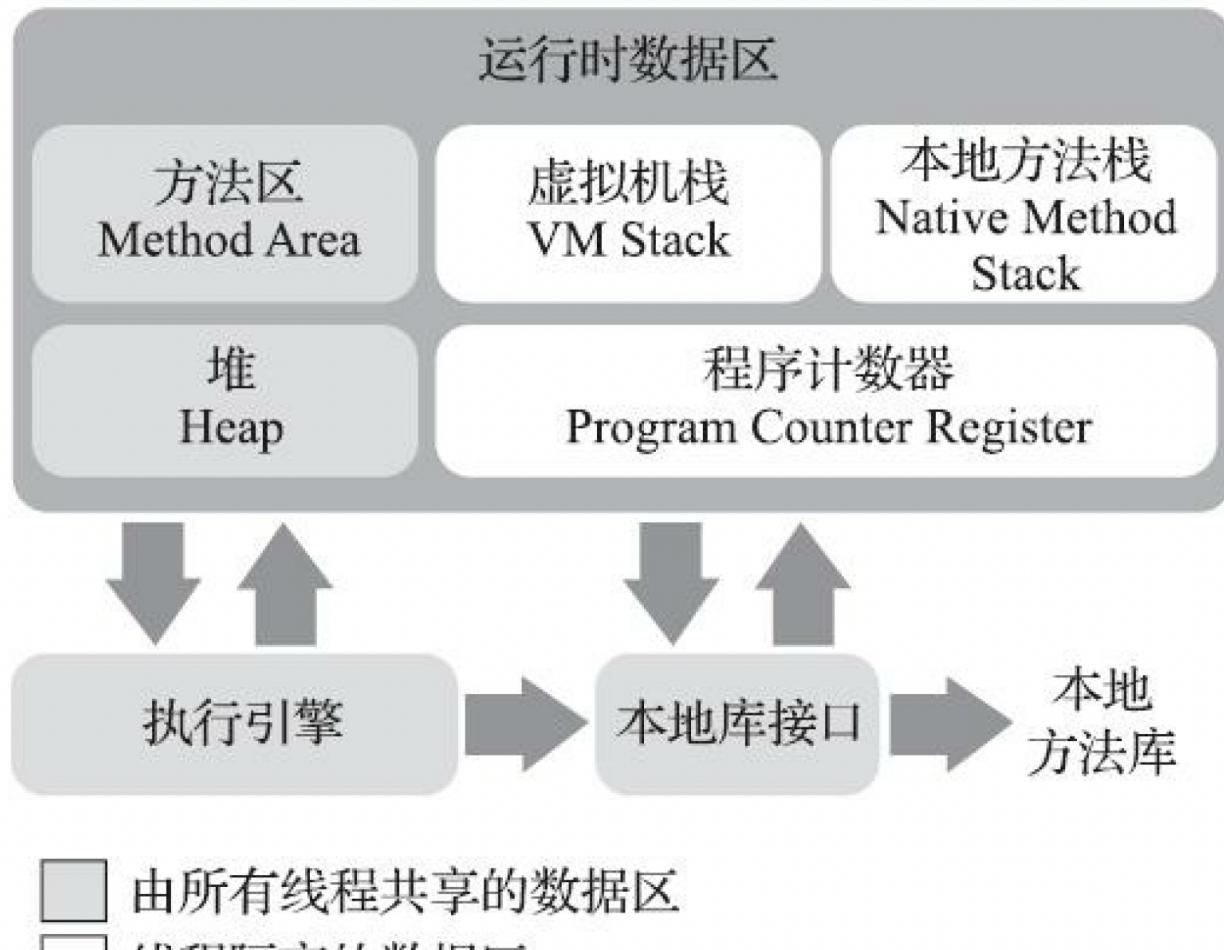


JVM笔记 -- 《深入理解java虚拟机：JVM高级特性与最佳实践》

第二章 java内存区域与内存溢出异常

运行时数据区域

内存数据区域结构



程序计数器 Program Counter Register (PCR)

- 定义与特点
 - 很小的内存区域
 - 当前线程所执行的字节码行号指示器 - 改变计数器数值选取下一行要执行的字节码指令
 - 线程私有 - 生命周期与线程相同
- 作用
 - 线程切换的时候，通过计数器能准确回到需要执行的字节码的位置，因此每个线程需要一个独立的PCR
- 数值如何改变

- 执行java程序 - 计数器数值是正要执行的字节码指令的地址；执行本地方法 - 计数器数值是空 (undefined)
- 异常状况
 - 唯一一个在《java虚拟机规范》没有规定任何**OutOfMemoryError**情况的区域

虚拟机栈 (JAVA Virtual Machine Stack)

- 定义与特点
 - 线程私有 - 生命周期与线程相同
 - java方法执行的线程内存模型
- 作用
 - 每个方法执行的时候，虚拟机会同步创建一个栈帧（stack frame），用来存储局部变量表、操作数栈、动态连接、方法出口等信息。每个方法调用都是一个栈帧在虚拟机栈中入栈和出栈的过程
 - 局部变量表
 - 提到栈的时候经常特指局部变量表
 - 存放**java**虚拟机基本数据类型、对象引用（句柄或者引用指针或其他跟对象位置有关的）、**returnAddress**类型（下一条字节码指令的地址）
 - 这些数据类型的存储空间在局部变量表中用局部变量槽（slot）来表示。除了64位的double和long一般占两个槽，其他都是一个槽
 - 局部变量表的内存空间在编译期间完成分配，因此进入方法时栈帧要分配给局部变量表的内存空间是完全确定的，运行期间也不会改变大小
- 异常状况
 - **StackOverflowError** - 线程请求的栈深度超过虚拟机允许的栈深度时
 - **OutOfMemoryError** - 如果虚拟机栈可以动态扩展，栈扩展申请不到足够内存时
 - HotSpot虚拟机不能动态扩展，因此不存在动态扩展申请不到足够内存抛出OOM异常，只可能是线程一开始申请栈空间就没申请到足够内存空间时抛出OOM异常

本地方法栈 (Native method stacks)

- 定义与特点
 - 线程私有 - 生命周期与线程相同
- 作用
 - 与虚拟机栈作用类似
 - 与虚拟机栈作用区别
 - 虚拟机栈为虚拟机执行java方法（字节码）服务
 - 本地方法栈为虚拟机执行本地方法服务
 - HotSpot虚拟机直接将本地方法栈与虚拟机栈合二为一，因为规范中没有强制规定
- 异常状况
 - 同虚拟机栈 - **OutOfMemoryError & StackOverflowError**

JAVA堆 (JAVA Heap)

- 定义与特点
 - 虚拟机管理的内存中最大的一块
 - 线程共享 - 虚拟机启动时创建
 - 垃圾回收器管理的内存区域 - GC堆

- 可以处于物理上不连续的内存空间中，但在逻辑上它应该被视为连续的
- 可以被实现成固定大小，或者可扩展，但当今主流java虚拟机都是按照可扩展实现的（设定参数-Xmx, -Xms）
- 作用
 - 存放实例对象
 - 回收内存角度 - 分成新生代、老年代、永久代、eden空间等等
 - 内存分配角度 - 分成若干个线程私有的分配缓冲区 (Thread Local Allocation Buffer, TLAB)
 - 以上角度的堆细分只是为了更高效地回收和分配内存，java堆唯一目的就是存放实例对象
- 异常状况
 - 没内存完成实例分配 & 无法扩展 - **OutOfMemoryError**

方法区 (Method Area)

- 定义与特点
 - 线程共享
 - 规范中描述为堆的一个逻辑部分，但有别名“非堆” (Non-Heap)
 - 不需要连续内存
 - 实现可以选择固定大小或者可扩展，甚至可以选择不实现垃圾收集
 - 内存回收一般针对常量池回收和对类型的卸载，但回收效果一般难令人满意，但有时这个区域回收确实必要
- 作用
 - 用于存储已被虚拟机加载的类型信息、常量、静态变量、即时编译器编译后的代码缓存等数据
- 异常状况
 - 无法满足新的内存分配 - **OutOfMemoryError**

运行时常量池 (Runtime Constant Pool)

- 定义与特点
 - 方法区的一部分
 - Class文件中存放除了类的版本、字段、方法、接口等描述信息外，还有一项信息是常量池表 (Constant Pool Table)，用于存放编译器生成的各种字面量和符号引用，类加载后这部分内容会存放到运行时常量池中
 - 动态性 - 运行期间也可以有新的常量放入池中 e.g. String类的intern()方法
- 作用
 - 类加载后存储Class文件中常量池内容（字面量、符号引用），一般来说也存储由符号引用翻译的直接引用，运行期间也有新的常量存入
- 异常状况
 - 无法申请到内存 - **OutOfMemoryError**

直接内存 (Direct Memory)

- 定义与特点
 - 不是虚拟机运行时数据区域一部分，也不是java虚拟机规范定义的内存区域
 - 但被频繁使用
 - 受到本机总内存大小（包括物理内存、SWAP分区或者分页文件）以及处理器选址空间的限制
- 作用
 - 可以被分配内存（属于对外内存）

- 异常状况

- 配置虚拟机参数时，会根据实际内存设置-Xmx等参数，但经常忽略直接内存，导致各个内存区域综合大于物理内存限制（包括物理和操作系统级的限制） - 动态扩展时 - **OutOfMemoryError**

HotSpot虚拟机对象探秘

对象创建

- 类加载检查

- 遇到一条字节码new指令时，首先检查该指令的参数是否能在常量池中定位到一个类的符号引用，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，若没有，则需要类加载过程的执行

- 新生对象分配内存

- 对象所需内存大小在类加载完成后完全确定
- 分配方法

- 堆内存规整 - 指针碰撞 (**Bump The Pointer**) - 只需要指针向空闲空间方向挪动一段与对象大小相等的距离
- 堆内存不规整 - 空闲列表维护 (**Free List**) - 记录哪些内存块可用，划出去给实例了就更新列表
- 使用哪种方法根据gc是否有空间压缩整理能力 (**Compact**)

- 并发如何保证线程安全

- 方法一：对分配内存空间的动作进行同步处理 - CAS + 失败重试
- 方法二：每个线程在堆上都有本地线程分配缓冲 (**TLAB**)，线程先在自己的本地缓冲区分配内存，不够了需要分配新的缓冲区才需要同步锁定 - 参数：-XX: +/-UseTLAB

- 分配的内存空间（不包括对象头）都初始化为零值

- 如果使用了TLAB，可以提前到TLAB分配时顺便初始化
- 保证了对象实例字段在java代码中可以不赋初始值即可直接使用，访问到字段的数据类型对应的零值

- 对对象必要的设置并存放在对象的对象头 (**Object Header**)

- 例如：哪个对象的实例，如何找到类的元数据信息，对象的hashcode，对象的gc分代年龄等

- (java程序视角) 执行class文件中的< init >()方法来初始化

- 虚拟机角度 - 前四个步骤已经产生新的对象；java程序视角 - 构造函数刚开始建造对象
- 一般来说，new指令之后执行< init >()方法，按照程序员的意愿对对象进行初始化，这样一个真正可用对象才完全被构造出来

对象内存布局(Hotspot)

- 对象在内存中的存储布局分三个部分

- 对象头 (**Header**)
- 实例数据 (**Instance data**)
- 对齐填充 (**Padding**)

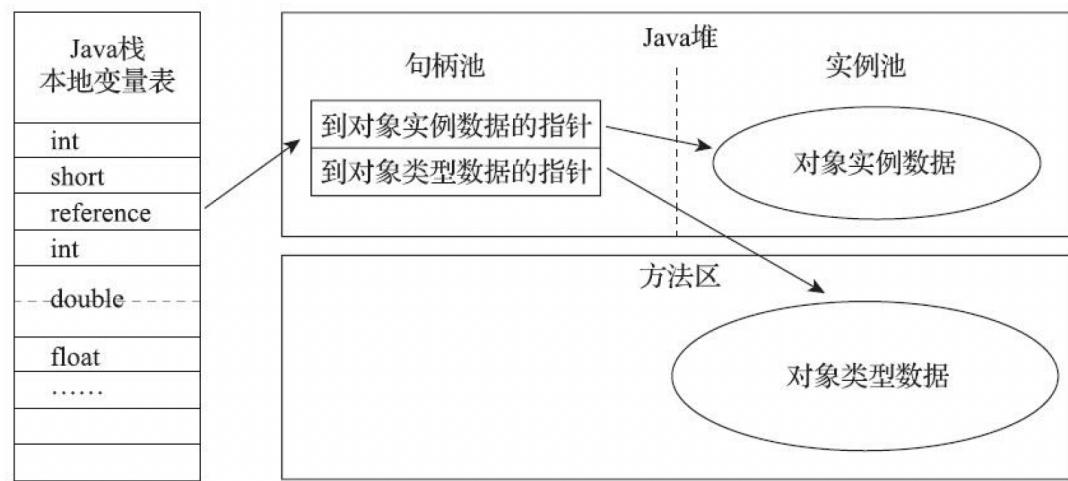
- 对象头 - 存储两类信息

- **Mark Word**
 - 用于存储对象自身的运行时数据，比如hashcode、gc分代年龄、锁状态信息、线程持有的锁、偏向线程id、偏向时间戳等
 - 动态定义的数据结构 - 在极小空间存储尽量多的数据，并且根据对象状态复用空间

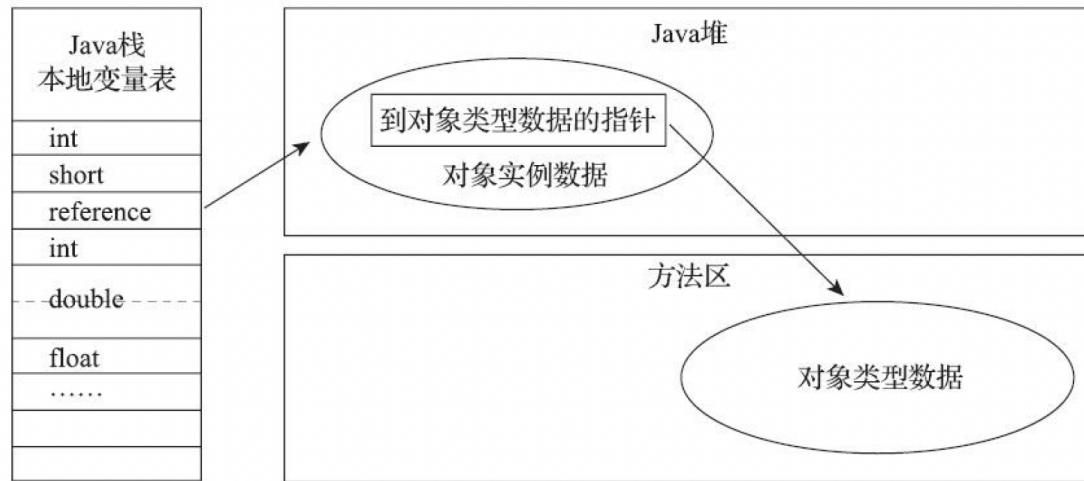
- 类型指针
 - 对象指向它的类型元数据的指针
 - 用来确认对象为哪个类的实例
- 如果对象是**java数组**, 还需一块记录数组长度的数据 - java对象元数据可以判断对象大小, 但如果数组长度不确定就无法推断数组大小
- 实例数据
 - 对象真正存储的有效数据 - 定义的各种类型的字段内容 (父类继承以及子类定义的)
 - 存储顺序 - 受虚拟机分配策略参数和字段在源码定义顺序影响
 - **Hotspot默认分配顺序** - 相同宽度字段分配到一起存放, 在该前提下, 父类中定义的变量出现在子类之前
- 对齐填充
 - 非必然存在, 仅仅是起占位符作用
 - HotSpot的自动内存管理系统要求对象起始地址必须是8字节的整数倍, 即任何对象大小都是8字节的整数倍, 所以对象头已经被精心设计成正好是8字节的倍数 (1-2倍), 如果实例数据部分没有对齐就用对齐填充来补全

对象访问定位

- 如何定位
 - java程序通过栈上的**reference**数据来操控堆的对象
- 实现**reference**来访问的方法
 - 句柄
 - java堆上可能划分出一块内存作为句柄池, reference存储的就是对象的句柄地址 - 包括: 对象实例数据地址 + 对象类型数据地址
 - 优势
 - reference存储的是稳定句柄池地址, 对象被移动时 (比如垃圾回收时) 不会改变 reference中的值, 只改变句柄里实例数据指针



- 直接指针
 - reference存储的直接就是对象实例数据地址, 对象内存布局里存有访问类型数据的相关信息
 - HotSpot使用该方法访问对象
 - 优势
 - 快捷, 省去一次指针定位的时间开销



第三章 垃圾收集器与内存分配策略

为什么需要了解垃圾收集和内存分配

- 垃圾收集GC需要完成三件事：哪些内存需回收？啥时候回收？如何回收？
- 当需要排查内存泄漏、溢出问题时，或者垃圾收集称为系统达到更高并发量的瓶颈中，我们就必须对其技术实施进行**必要的监控和调节**
- 不需要过多考虑回收的区域
 - 程序计数器、虚拟机栈、本地方法栈 - 生命周期与线程一样，栈帧大小一开始类结构确定时就已知，然后随方法入栈出栈，因此内存分配回收具有确定性，当方法结束或者线程结束，内存自然跟随回收
- **需要考虑垃圾回收的区域**
 - **java堆和方法区** - 处于运行期间，内存分配和回收是动态的 - **显著的不确定性**

如何判断对象死亡

- 死去 - 即不可能被任何途径使用的对象

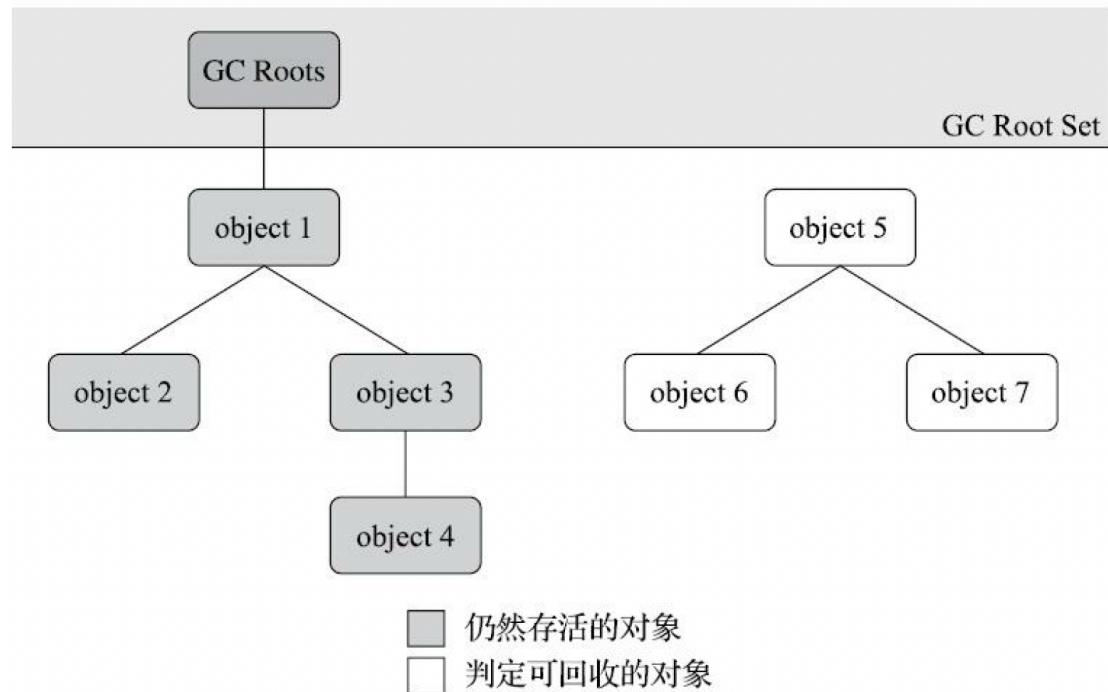
引用计数算法（Reference Counting）

- **原理**
 - 在对象中添加一个引用计数器，每当有一个地方引用它，计数器++；引用失败，计数器--；任何时刻计数器==0的对象不可能再被使用
- **优势**
 - 虽然占用额外内存空间计数，但原理简单，判定效率高，多数情况效果不错
- **劣势**
 - 需要考虑很多例外情况，必须配合大量额外处理才能保证正确地工作 - e.g.单纯引用计数很难解决对象间相互循环引用问题
- **应用**
 - 主流java虚拟机都不选择该算法

可达性分析算法（Reachability Analysis）

- **原理**

- 通过一系列称为“**GC Roots**”的根对象作为起始节点集，在这些节点开始，根据引用关系向下搜索，搜索过程所走过的路径称为“引用链”（**Reference Chain**），如果一个对象到gc roots间没有任何引用链相连，或者用图论的话说就是从**gc roots**到对象不可达，则该对象不可能再被使用
- e.g.



- 可作为gc roots的对象
 - 虚拟机栈（栈帧中的本地变量表）中引用的对象 - e.g. 各个线程被调用的方法堆栈中使用到的参数、局部变量、临时变量等
 - 方法区中类静态属性引用的对象 - e.g. java类的引用类型静态变量
 - 方法区常量引用的对象 - e.g. 字符串常量池（String Table）里的应用
 - 本地方法栈中JNI（native方法）引用的对象
 - Java虚拟机内部的引用 - e.g. 基本数据类型对应的Class对象，一些常驻异常对象（OutOfMemoryError、NullPointerException等），系统类加载器
 - 所有被同步锁（synchronized关键词）持有的对象
 - 反映java虚拟机内部情况的JMXBean、JVMTI中注册的回调、本地代码缓存等
- 根据所选垃圾收集器以及当前回收的内存区域不同，也有其他对象“临时性”加入gc roots集合，以保证可达性分析的正确性（比如分代收集和局部回收（partial gc），某个区域的对象可能被位于堆中其他区域对象所引用，所以需要加入gc roots集合中 - 跨代引用）
- 应用
 - 当前主流使用程序语言（java、c#等）内存管理系统子系统都用该算法
 - 为避免gc roots包含过多对象导致过度膨胀，实现时做了各种优化处理

引用

- JDK1.2之后，java将引用分为四种引用（强度依次减弱） - 强引用（**Strong Reference**），软引用（**Soft Reference**），弱引用（**Weak Reference**），和虚引用（**Phantom Reference**）（后三种都是jdk1.2之后提供了对应类来实现 - SoftReference、WeakReference、PhantomReference）
- 强引用
 - 最传统的引用定义 - `Object obj = new Object()`
 - 该引用存在，该被引用对象就永远不会被回收
- 软引用

- 还有用，但非必须对象
- 系统将要发生内存溢出异常前，把这些对象列进垃圾回收范围之中进行第二次回收，若垃圾回收后还是没有足够内存，才会抛出内存溢出异常
- 弱引用
 - 非必须对象，强度比软引用更弱
 - 被引用对象只能生存到下一次gc发生为止，gc一旦开始就会被回收
- 虚引用（幽灵引用or幻影引用）
 - 最弱
 - 对象添加虚引用完全无法左右它的生存时间，也无法通过该引用获得实例
 - 唯一目的只是该对象回收时收到一个系统通知

判定不可达后是否真正死亡（回收）

- 真正死亡需要至少经历两次标记过程
 - 第一次标记：对象进行可达性分析后发现没有与gc roots相连接的引用链
 - 第二次标记：先进行一次筛选，条件是该对象是否有必要执行**finalize()**方法
 - 假如对象没有覆盖**finalize()** or **finalize()**已经被虚拟机调用过 - 虚拟机视为“没有必要执行**finalize()**”
 - 反之，“有必要执行**finalize()**”，该对象放入一个名为**F-Queue**队列中，并在稍后由一条虚拟机自动建立、低调度优先级的Finalizer线程去执行它们的**finalize()**方法，该执行指的是虚拟机会触发这个方法开始运行，但不承诺一定等它运行结束
 - 原因：如果某对象**finalize()**执行缓慢，或者更极端发生了死循环，那**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子系统的崩溃。
 - 稍后收集器会对**F-Queue**的对象进行第二次小规模标记。若对象**finalize()**成功让自己和引用链上的任何对象建立关联即可拯救成功（e.g. 自己（`this`关键词）赋值给某个类变量或对象的成员变量），就会被第二次标记时被移出“即将回收”的集合。
 - **finalize()**是对象逃脱死亡最后的机会，一个对象自救的**finalize()**最多只会被系统自动调用一次。
- 两次标记后，在集合中的对象就基本上要真回收了
- 尽量避免使用**finalize()**
 - 运行代价高昂，不确定性大，无法保证各个对象的调用顺序，**官方声明不推荐使用 - try-finally**或其他可以做的更好、更及时

回收方法区

- 垃圾收集器可以选择不实现对方法区的回收或者部分回收，因为方法区回收“性价比”较低，在回收判断条件苛刻的情况下，回收效果不佳。
- 方法区回收有两个部分
 - 废弃常量
 - 条件相对简单
 - 回收判断条件：类似堆对象回收，若没有任何对象或者虚拟机其他地方引用该常量（比如常量池的某个字面量或者符号引用），该常量可以被系统清理出常量池
 - 不再使用的类型
 - 条件相对苛刻
 - 回收判断条件（要同时满足三条，满足后仅仅是“被允许对类进行回收”，而不是像对象一样没有引用就必然回收）：
 - 该类型所有实例都被回收了，堆里不存在该类以及任何派生子类的实例

- 加载该类的类加载器已经被回收，除非是经过精心设计的可替换类加载器的场景，比如OSGi、JSP的重加载等，否则通常很难达成
- 该类对应的`java.lang.Class`对象没有在任何地方被应用，无法在任何地方通过反射访问该类的方法

垃圾收集算法

- 根据配合判断对象死亡角度，垃圾收集算法划分为
 - 引用计数式垃圾收集（ReferenceCountingGC） - 直接垃圾收集
 - 追踪式垃圾收集（TracingGC） - 间接垃圾收集 - 属于主流Java虚拟机使用的垃圾收集算法

分代收集理论（Generational Collection）

- 理论（实际是符合大多数程序运行实际情况的经验法则）
 - 当前商业虚拟机垃圾收集器大多都遵循该理论
 - 两个分代假说
 - 弱分代假说（Weak Generational Hypothesis）：绝大多数对象都是朝生夕死
 - 强分代假说（Strong Generational Hypothesis）：熬过越多次垃圾回收过程的对象越难以消化
 - 基于上述假说的设计原则
 - 收集器将Java堆划分为不同区域，将回收对象依据其年龄（熬过垃圾收集过程的次数）分配到不同区域存储
 - 好处
 - 省时间，低代价回收大量空间，空间获得有效利用
 - 分为不同区域后，垃圾收集器就可以每次只回收其中一个或某些部分的区域，才能针对不同区域安排与里面存储对象存亡特点相匹配的垃圾收集算法
 - 现在商用Java虚拟机，一般至少把堆划分为新生代（Young Generation）和老年代（Old Generation）
 - 每次有大批对象死在新生代，少部分存活下来的会逐步晋升到老年代存放
 - 第三条假说（经验法则）
 - 出现原因：对象间可能会存在跨代引用
 - 跨带引用可能引起的后果：假如只局限新生代的收集，但新生代对象可能被老年代对象引用，为了找出该区域存货对象，不得不在固定GC Roots之外，额外遍历整个老年代的对象老保证可达性分析的正确性 - 巨大性能负担。反之亦然。
 - 假说内容：跨代引用相对于同代引用仅占极少数
 - 由上述两个假说得到的隐含推论：存在互相引用关系的两个对象应该倾向于同时生存或者消亡 - 新生代对象有跨代引用 -> 老年代对象难消亡 -> 新生代对象因为该引用得以存活 -> 年龄增长，晋升老年代 -> 跨代引用消除
 - 根据第三条假说进行的针对跨代引用的设计原则
 - 只需在新生代上建立一个全局数据结构（记忆集Remembered Set），该结构把老年代划分为若干小块，标识出老年代哪一内存会存在跨代引用
 - 之后发生minor gc，只有包含跨代引用的小块内存的对象被放进gc roots里进行扫描
 - 好处
 - 虽然需要在对象改变引用关系（比如自己或者某个属性赋值）时维护记录数据的正确性而增加一点运行时开销（写屏障），但比收集时扫描整个老年代划算
 - 部分回收（Partial GC）
 - 目标不是完整收集整个Java堆的垃圾收集

■ 分类

- 新生代收集(Minor GC/Young GC)
- 老年代收集 (Major GC/Old GC) : 目前只有CMS收集器有单独收集老年代的行为
- 混合收集 (Mixed GC) : 收集整个新生代+部分老年代。目前只有G1收集器有这个行为
- 整堆收集 (Full GC) : 收集整个java堆和方法区

标记-清除算法 (Mark-Sweep)

• 原理

- 两个阶段-标记+清除：标记所有需要回收对象然后统一回收被标记对象/标记所有存活对象然后统一回收未被标记对象（标记过程-判定是否为垃圾/死亡的过程）

• 特点

- 最基础算法，后续算法都是以此为基础，改进其缺点
- 执行效率不稳定 - 如果有大量对象且大部分需要回收，必须进行大量标记和清除动作 - 对象越多，执行效率越低
- 内存空间碎片化问题 - 产生大量不连续内存碎片，导致之后运行过程要分配大对象时找不到足够的连续内存，不得不提前触发另一次垃圾收集动作

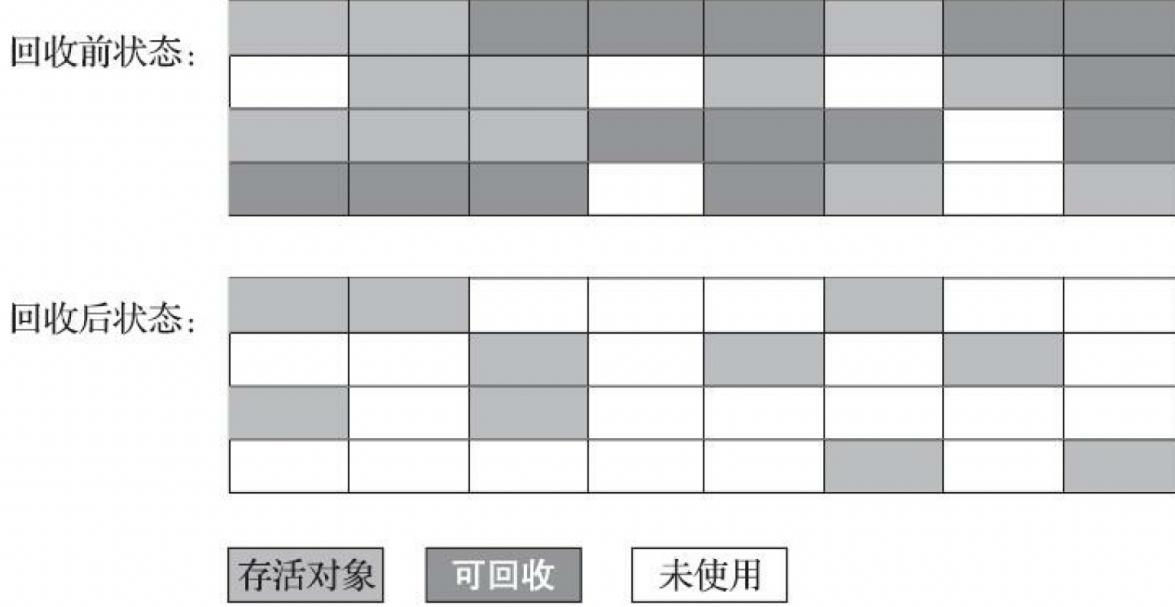


图3-2 “标记-清除”算法示意图

标记-复制算法 (复制算法)

• 半区复制算法 (Semispace Copying)

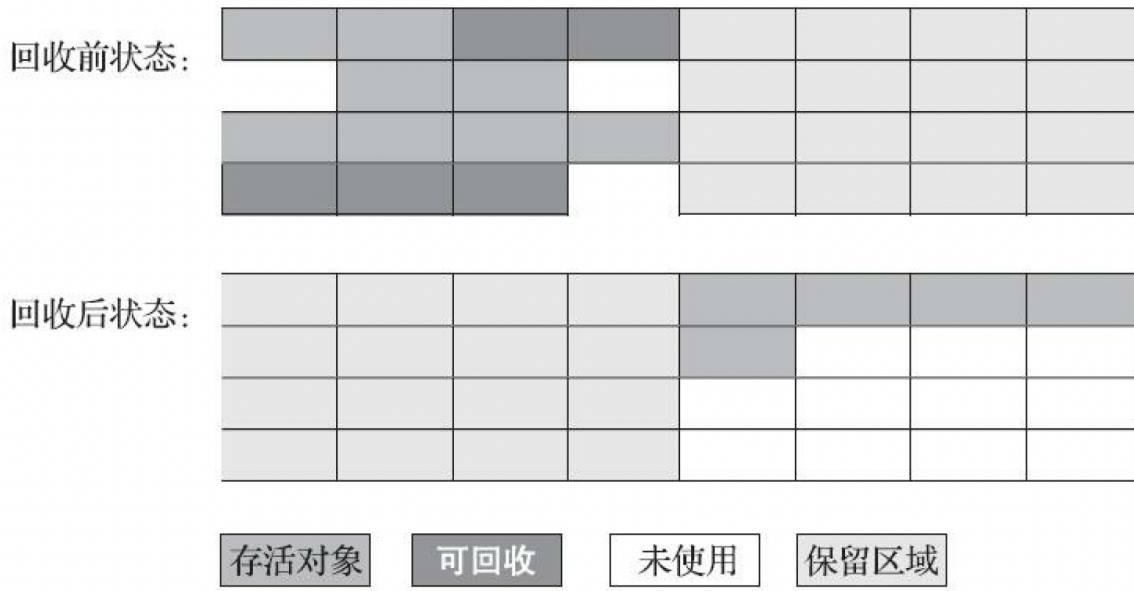
◦ 原理

- 内存容量分为大小相等两个部分，每次只用其中一块，当一块内存用完了，把存活对象对象复制到另一块上，再把用完的这一块一次清除

◦ 特点

- 多数对象存活 - 产生大量内存间复制开销
- 多数对象可回收 - 只需复制极少数存活对象
- 优势 - 只针对半区回收，内存分配时不用考虑空间碎片复杂情况，只需要移动堆顶指针即可按顺序分配 - 简单，运行高效

- 缺点 - 可用内存缩小为原来一半，空间浪费多



。

- 优化版半区复制分代策略 - Appel式回收

- 应用

- 现在商用虚拟机大多优先采用标记复制算法来回收新生代
- "Appel式回收"根据新生代"朝生夕死"提出的优化版半区复制策略，该策略被HotSpot的Serial、ParNew等新生代收集器采用来设计新生代内存布局

- 原理

- 新生代分为较大的Eden空间和两块较小的Survivor空间，每次分配内存只使用Eden和其中一块Survivor，发生gc后，将Eden和Survivor中存活对象一次性复制到另一块Survivor上，然后清除Eden和已用过的Survivor
- HotSpot虚拟机默认Eden: Survivor大小比例=8:1，即每次新生代可用内存位整个新生代容量的90%，只有10%的新生代空间被浪费
- 分配担保 (Handle Promotion)
 - 原因：没人能保证每次回收都只有不多于10%的对象存活
 - 当另一块Survivor空间无法容纳存放上一次新生代收集下来的存活对象，这些对象将需要依赖其他内存区域（实际一般为老年年代）来分配担保。一般是将这些对象通过分配担保机制直接进入老年年代，这对虚拟机是安全的

标记-整理算法 (Mark-Compact)

- 出现原因

- 标记-复制算法在对象存活率高时效率低，如果对象存活率过高，则需要额外空间进行分配担保。老年年代每次回收对象存活率都很高，因此老年年代一般不能直接使用该算法

- 原理

- 标记过程：与“标记-清除”算法一样
- 清除过程：让所有存活的对象都想内存空间一端移动，然后直接清理掉边界以外的内存

回收前状态:

回收后状态:

存活对象

可回收

未使用

-

• 特点

- 移动式算法 - 是否移动回收后的存活对象都存在弊端
- 如果移动整理存活对象 - 老年代每次回收都有大量存活对象 - 移动存活对象并更新所有引用这些对象的地方将会是极为负重的操作，并且移动对象必须全程暂停用户应用程序（标记清除也要STW，但时间可忽略） - "Stop the World" (STW) - 内存回收更复杂
- 如果不移动整理存活对象 - 空间碎片化问题 - 只能靠更为复杂的内存分配器和内存访问器解决（比如“分区空闲分配链表”来解决内存分配）。内存访问是用户程序最频繁的操作 - 这个环节如果有额外负担会直接影响应用程序吞吐量 - 内存分配更复杂
- 不移动对象停顿时间会更短，但从整个程序吞吐量来看，移动对象更划算 - 因为内存分配和访问比垃圾回收频率要高的多，即便内存回收速度提高了，内存分配和访问耗时也会让总吞吐量下降

• 应用 (HotSpot)

- 关注吞吐量收集器 - ParallelScavenge收集器
- 关注延迟收集器 - CMS收集器 - 基于标记清除算法
 - CMS用了一种“和稀泥式”解决方案 - 虚拟机平时多数用标记清除，直到内存空间碎片化程度影响到对象分配，再采用标记整理算法收集一次

HotSpot算法细节实现

- 实现虚拟机时对算法执行效率严格考量才能保证虚拟机高效运行

根节点枚举

- 消耗时间多：固定可作为**GC Roots**的节点主要在全局性的引用（比如常量和类静态属性）与执行上下文（比如栈帧的本地变量表），查找过程要高效很难
- 必须暂停用户线程：根节点枚举始终是必须在一个能保证一致性的快照中得以进行 - 枚举过程中根节点集合的对象引用关系不能再变化，不然分析准确性无法保证
- **HotSpot**高效进行对象引用查找的解决方法
 - 使用一组称为**OopMap**的数据结构，一旦类加载动作完成的时候，HotSpot会把对象内什么偏移量上是什么类型的数据计算出来，在即时编译过程中，也会在特定的位置记录下栈里和寄存器里哪些位置是引用，这样收集器扫描时可直接得到这些信息，不需要一个不漏从方法区等gc roots开始查找

安全点 (Safe Point)

- 作用
 - 有很多指令能导致OopMap内容（引用关系）发生变化，但为每一条指令生成对应OopMap很浪费空间
 - 需要解决如何停顿用户线程
- 定义
 - 在特定的位置记录下信息（OopMap），可以强制要求必须执行到达安全点后才能够暂停进行垃圾收集
- 选取原则
 - “是否具有让程序长时间执行的特质”为标准进行选定
 - “长时间执行”最明显特征 - 指令序列复用（比如方法调用、循环跳转、异常跳转等）
- 如何让垃圾收集发生时所有线程（这里不包括执行JNI调用的线程）都跑到最近的安全点停顿下来
 - 方案一：抢断式中断（Preemptive Suspension）：gc发生时，系统先把所有用户线程全部中断，若某线程不再安全点上，恢复它的执行直到跑到安全点再中断 - 现在几乎不用这个方案
 - 方案二：主动式中断（Voluntary Suspension）：gc发生时，不直接对线程操作，设置一个标志，各个线程执行过程中不停主动地轮询该标志，一旦发现中断标志为真就自己在最近的安全点上主动中断挂起 - 轮询标志地点和安全点重合 - 轮询操作频繁出现所以设计必须高效

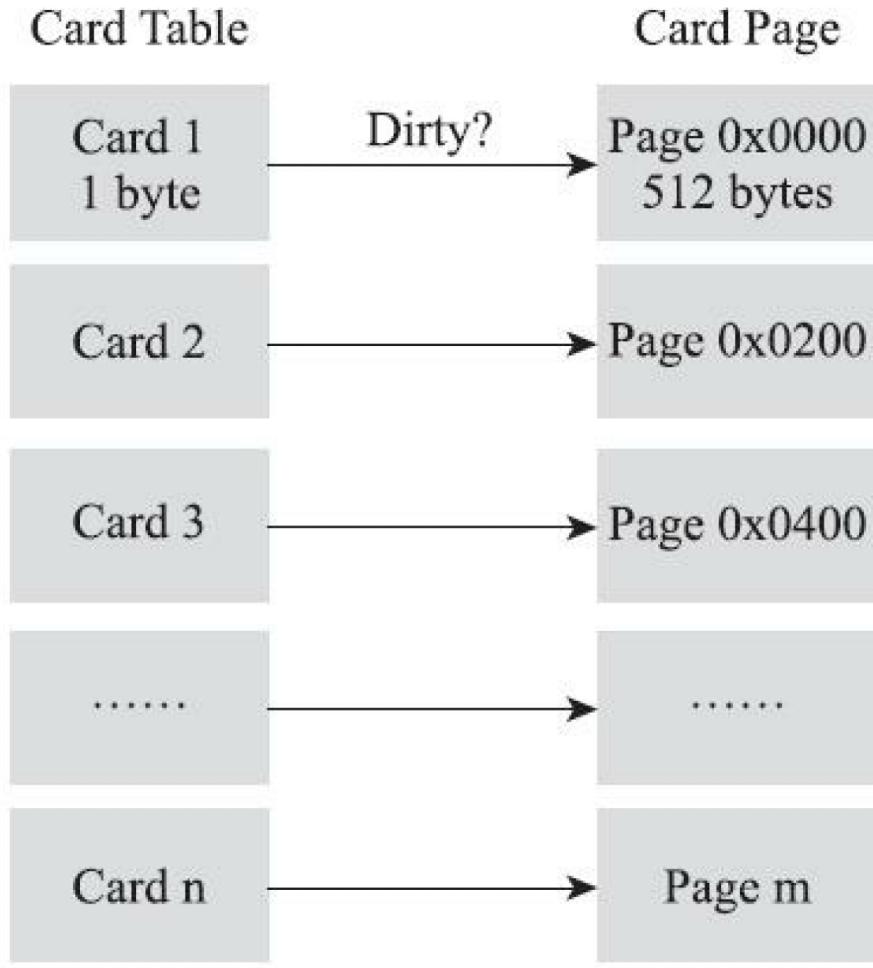
安全区域 (Safe Region)

- 作用
 - 如果程序“不执行”，比如Sleep或者blocked状态，线程无法相应虚拟机中断请求，无法走到安全点挂起自己，虚拟机也不可能持续等待
- 定义
 - 能够确保在某一段代码片段中，引用关系不会发生变化，因此在这个区域中任意地方开始垃圾收集都是安全的 - 被扩展拉伸的安全点
- 线程如何运行
 - 用户线程执行到安全区域的代码 - 标识自己已经进入安全区域 - gc时不必管理这些进区域的线程 - 离开安全区域时要检查虚拟机是否发成根节点枚举或者gc时需要的其他STW - 如果完成就线程正常执行 - 没完成就必须等待，直到收到可以离开区域的信号

记忆集与卡表

- 作用
 - 解决对象跨代引用问题，以及涉及Partial GC行为的垃圾收集器
- 记忆集
 - 定义
 - 用于记录从非收集区域指向收集区域的指针集合的抽象数据结构
 - 实现方法
 - 最简单实现 - 用非收集区域中所有含跨代引用的对象数组来实现这个数据结构 - 存储和维护成本过高
 - 用更粗犷的记录粒度类节省存储和维护成本 - 记录精度
 - 字长精度 - 机器字长
 - 对象精度
 - 卡精度 - 每个记录精确到一块内存区域，内有对象有跨代指针
 - 利用卡精度 - 卡表（Card Table）实现记忆集

- 最常用
- 最简单形式 - 一个字节数组 - HotSpot虚拟机也用该方法实现
- 字节数组中每一个元素对应着一块特定大小的内存块 - 卡页（Card Page）
 - 大小: 2的n次幂的字节数 - hotspot是 2^9 (512字节)
 - 只要卡页内有一个或者多个对象的字段有跨代指针, 就把对应卡表的数组元素的值标志为1: 元素变脏 (dirty), 没有就标0, gc的时候筛选出卡表中的变脏的元素 - 找到哪些卡页中有跨代指针 - 加入gc roots中扫描

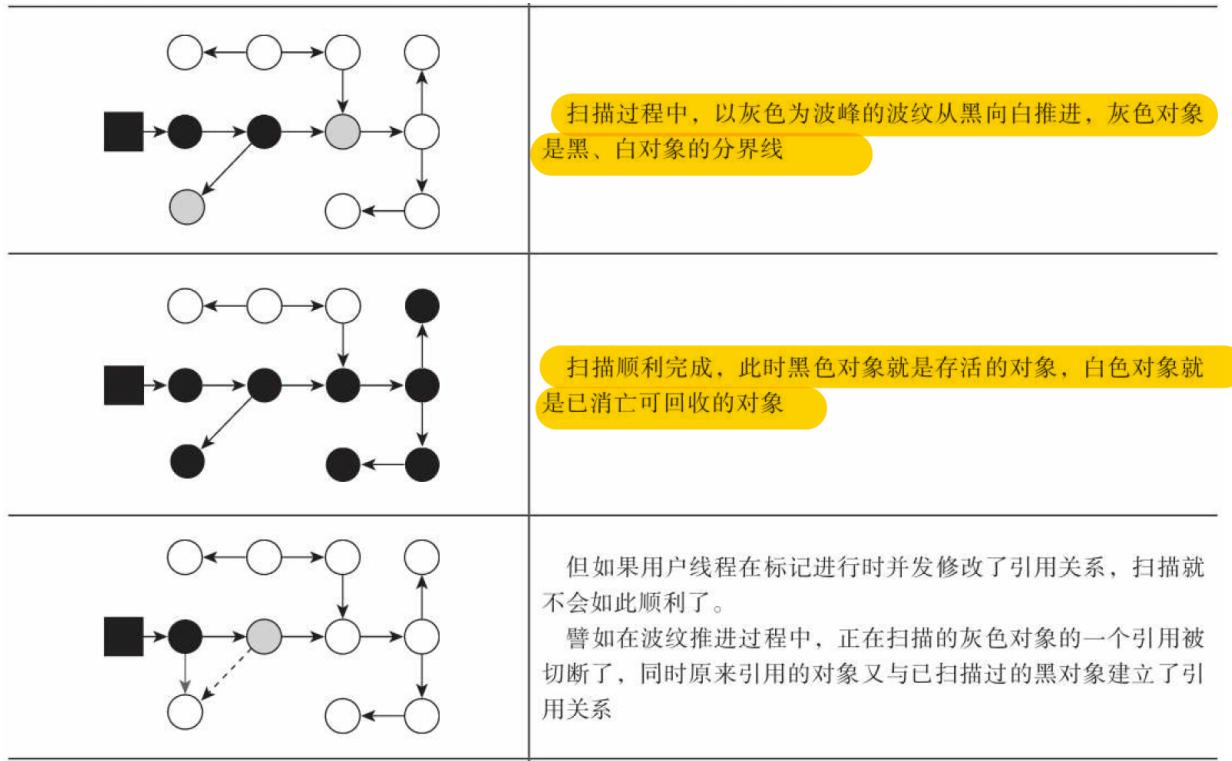


写屏障 (Write Barrier)

- 作用
 - HotSpot中来解决卡表元素如何维护问题 - 如何在对象赋值那一刻更新维护卡表来让卡页变脏
- 定义
 - 在虚拟机层面对“引用类型字段赋值”这个动作的AOP切面，在引用对象赋值时会产生一个环形 (Around) 通知，供程序执行额外的动作，也就是说赋值的前后都在写屏障的覆盖范围内
 - 写前屏障 (Pre-Write Barrier) : 赋值前的部分的写屏障
 - 写后屏障 (Post-Write Barrier) : 赋值后的部分的写屏障
 - 应用写屏障后，所有赋值操作生成相应的指令 - 添加更新卡表的操作
- 特点
 - 有一定开销，但划算
 - 高并发下面临“伪共享” (False Sharing) 问题 - 几个线程同时改变卡表元素
 - 解决: 有条件的写屏障 - 先检查卡表标记, 只有当该卡表元素未被标记过时才将起标记为脏

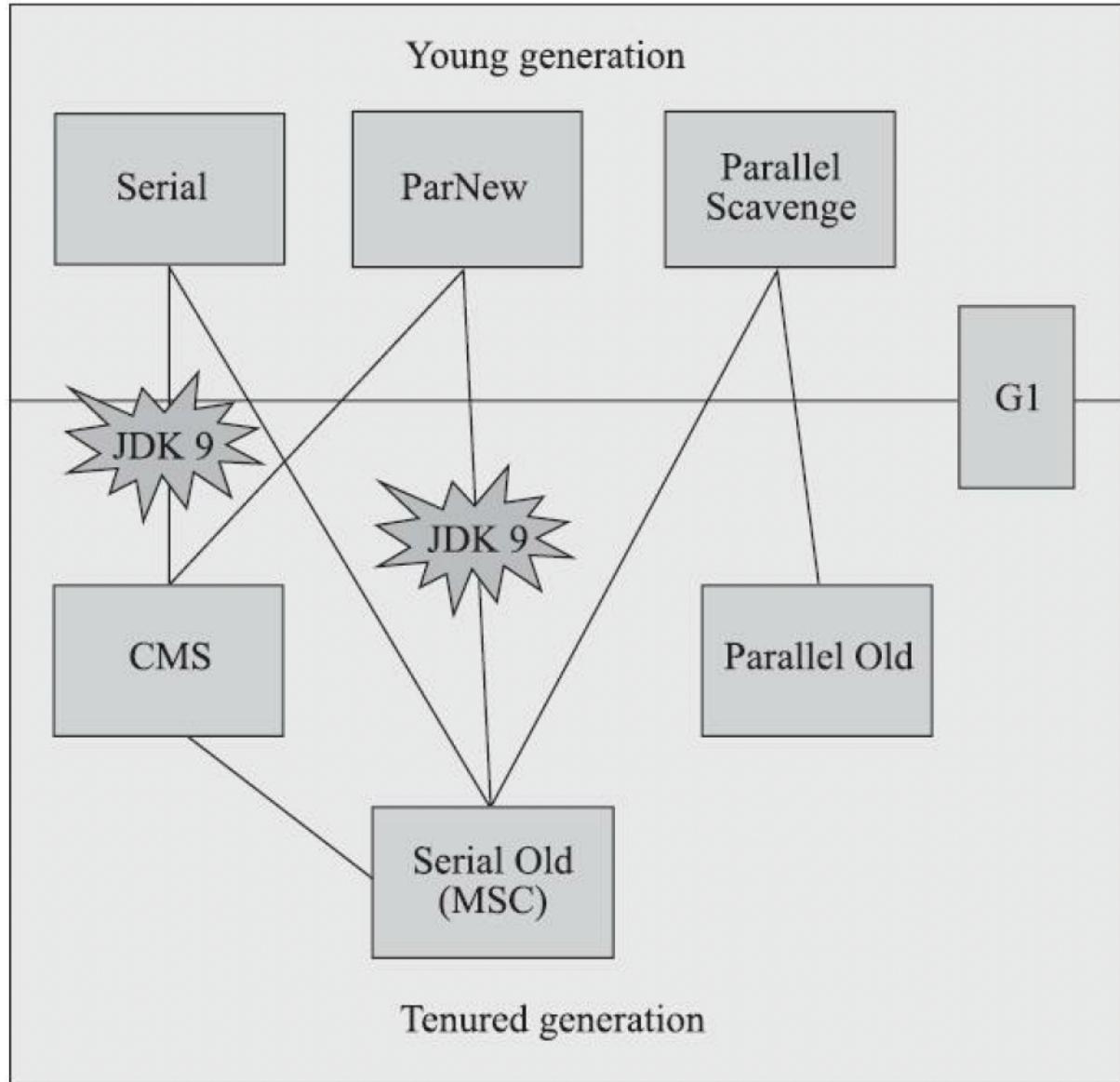
并发的可达性分析

- 为了解决或者降低用户线程停顿，可以考虑用户线程和gc roots遍历并发进行，但必须解决并发的可达性分析准确性问题
- 用三色标记来标记对象状态



- 如果可达性分析和用户线程并发工作，可达性分析一边标记，一边有用户线程在改变引用关系，导致有些对象会被标错（比如原本存活却被标记为消亡，很致命）
- 存活对象误标为消亡条件**
 - 赋值器插入一条或多条黑色到白色的引用
 - 赋值器删除全部从灰色到该白色的直接或间接引用
- 解决对象消失问题：只要破坏这两个条件任意一个即可**
 - 增量更新 (Incremental Update) - 破坏第一个条件**
 - 插入引用时把这个新插入的引用记录下来，并发扫描结束后再以这些记录过的引用关系中的黑色为根重新扫描一次 - 相当于黑色变回灰色
 - 原始快照 (Snapshot At The Beginning) - 破坏第二个条件**
 - 删除引用前把这个该删除引用记录下来，并发扫描结束后再以这些记录过的引用关系中的灰色为根重新扫描一次 - 相当于最终一定是按照最原始的样子扫描的
- 应用**
 - CMS - 基于增量更新的并发标记
 - G1、Shenandoah - 基于原始快照的并发标记
- 以上无论是对引用关系记录的插入还是删除，虚拟机记录操作通过写屏障实现

经典垃圾收集器



- 两个收集器之间如果存在连线，说明可以搭配使用
- 不存在“万能”或最好的收集器，需要根据具体应用选择最合适的

Serial收集器

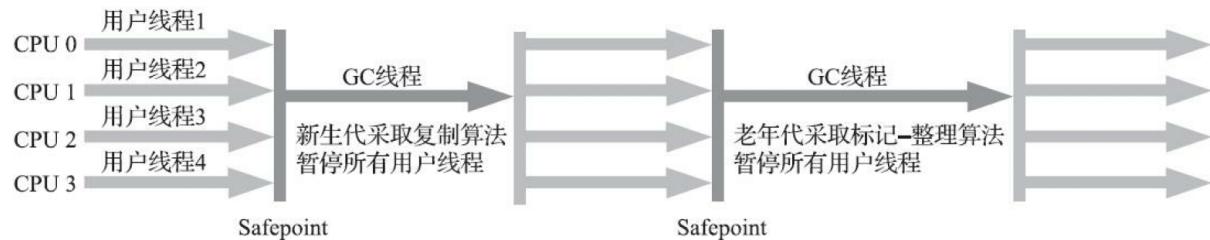


图3-7 Serial/Serial Old收集器运行示意图

- 特点
 - 最基础、历史最悠久的收集器
 - 基于标记复制算法
 - 单线程工作 - 不仅说明只会使用一个处理器或一条收集线程去完成垃圾收集工作，更强调它进行 gc 时，必须 **STW**（由虚拟机后台自动发起和完成），直到 gc 结束

- 简单高效（与其他收集器单线程相比）
- 资源受限环境下，所有收集器中额外内存消耗（Memory Footprint）最小的
- 对于单核处理器或者处理器核心数较少的环境里，Serial由于没有线程交互，专心做gc自然获得最高的单线程收集效率
- 应用**
 - 迄今为止，依旧是HotSpot虚拟机运行在客户端模式下的默认新生代收集器 - 是个很好的选择

ParNew收集器

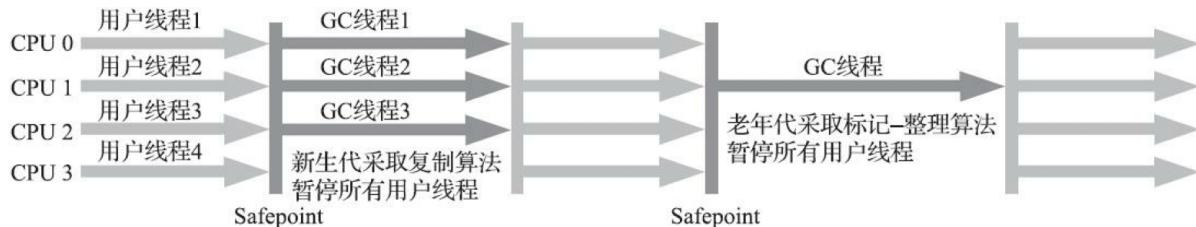


图3-8 ParNew/Serial Old收集器运行示意图

- 特点**
 - Serial收集器的多线程并行版本，除了同时使用多条线程进行gc之外，其余行为都与Serial完全一致 - 基于标记复制算法
 - 无过多创新
 - 单核处理比不过Serial
 - 可以被使用的处理器核心数量增加，ParNew对于gc的系统资源的高效利用还是有好处的 - 默认开启收集线程数=处理器核心数量，可以使用-XX:ParallelGCThreads参数来限制gc线程数
- 应用**
 - 不少运行在服务端模式下的HotSpot虚拟机，尤其是JDK7之前的遗留系统中首选的新一代收集器
 - 除了Serial收集器外，目前只有ParNew能与CMS收集器配合工作 - jdk5中CMS收集老年代时，新生代只能选择ParNew或者Serial
 - ParNew是激活CMS后（使用-XX: +UserConcMarkSweepGC）的默认新一代收集器，也可以使用-XX: +/-UseParNewGC来强制指定或者禁用它
 - G1出现后因为其不需要新一代收集器配合工作，jdk9开始ParNew+CMS组合不再是官方推荐的服务端模式下的收集器解决方案，希望完全被g1取代，甚至取消了ParNew+Serial Old以及Serial+CMS组合的支持（也没几个用），取消了-XX:+UseParNewGC参数，意味着ParNew+CMS只能相互搭配，相当于ParNew合并入CM，成为专门处理新世代的组成部分 - HotSpot第一款退出历史舞台的垃圾收集器
- 并行 (Parallel)**
 - 多条垃圾收集器线程之间的关系，说明同一时间有多条线程在协同工作，通常默认此时用户线程是处于等待状态
- 并发 (Concurrent)**
 - 垃圾收集器线程与用户线程之间的关系，说明同一时间垃圾收集器线程与用户线程都在运行；用户线程未被冻结，但垃圾收集器线程占用了一部分系统资源，所以应用程序处理吞吐量受一定影响

Parallel Scavenge收集器

- 特点**

- 新生代收集器，基于标记复制算法，能够多线程并行收集 - 表面特性类似ParNew
- 目标 - 达到一个可控制的吞吐量（Throughput）= 运行用户代码时间 / (运行用户代码时间 + 运行垃圾收集时间) - 吞吐量优先收集器
 - 高吞吐量可以最高效率利用处理器资源，尽快完成程序的运算任务，主要适合在后台运算而不需要太多交互的分析任务
 - 提供了两个用于精确控制吞吐量的参数
 - 控制最大垃圾收集停顿时间-XX:MaxGCPauseMillis参数
 - 设置大于0的毫秒数，收集器将尽力保证gc时间不超过这个设定值
 - gc停顿时间缩短是以牺牲吞吐量和新生代空间为代价换来的 - 停顿时间下降，吞吐量也下降
 - 直接设置吞吐量大小的-XX:GCTimeRatio参数
 - 值是大于0小于100的整数，垃圾收集时间占总时间的比率 (e.g. 19 -> 1/1+19 -> 最大垃圾收集时间占总时间的5%。默认值99 -> 1%)
- 自适应调节策略 (GC Ergonomics)
 - 该收集器还有一个参数-XX:+UseAdaptiveSizePolicy。激活之后不需要人工设置新生代大小 (-Xmn)、Eden与Survivor的比例 (-XX: SurvivorRatio)、晋升老年对象大小 (-XX:PretenureSizeThreshold) 等参数，虚拟机会根据当前系统运行情况收集性能监控信息，动态调整参数 - 提供最合适的停顿时间或最大吞吐量
 - 只需要设置基本内存数据 (如-Xmx最大堆)，然后使用-XX:MaxGCPauseMillis或者-XX:GCTimeRatio设置一个优化目标，剩下由虚拟机自动优化

Serial Old收集器

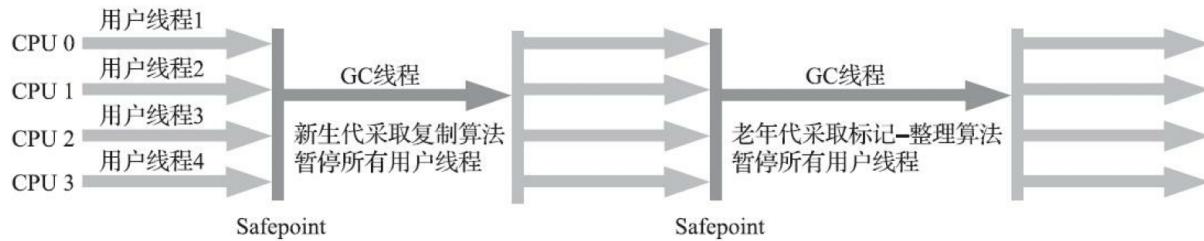


图3-7 Serial/Serial Old收集器运行示意图

- 特点
 - Serial收集器老年代版本
 - 单线程，基于标记整理算法
- 应用
 - 主要意义就是供客户端模式下的HotSpot虚拟机使用
 - 服务端模式下
 - jdk5以及之前与Parallel Scavenge搭配使用
 - 作为CMS发生失败的后备预案，在并发收集发生Concurrent Mode Failure时使用

Parallel Old收集器

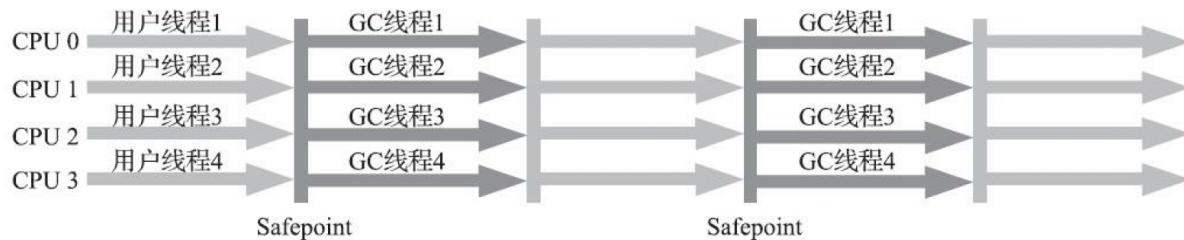


图3-10 Parallel Scavenge/Parallel Old收集器运行示意图

-
- **特点**
 - Parallel Scavenge收集器的老年代版本
 - 多线程并行收集，基于标记整理算法
- **应用**
 - 没有该收集器之前，之前Parallel Scavange只能选择和Serial Old（或者叫PS MarkSweep）搭配使用，Serial Old在服务端性能很拖累，因此Parallel Scavange很难发挥整体吞吐量最大化。现在，当注重吞吐量或处理器资源稀缺时，都可以考虑Parallel Scavange+Parallel Old组合搭配

CMS收集器

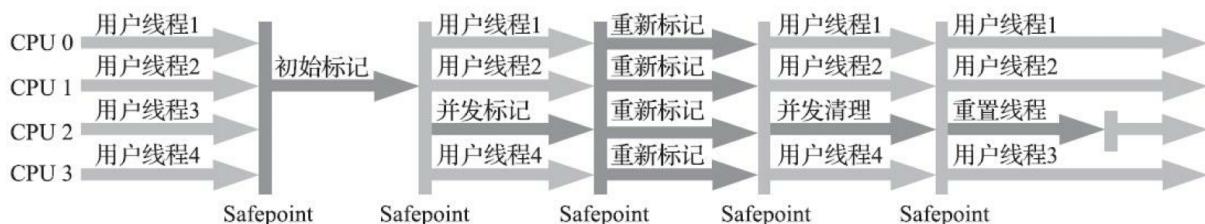


图3-11 Concurrent Mark Sweep收集器运行示意图

-
- **运行过程（四个步骤）**
 - 初始标记(CMS initial mark)
 - 需要STW
 - 仅仅标记一下GC Roots能直接关联到的对象
 - 速度很快
 - 并发标记(CMS concurrent mark)
 - 从GC ROOTS的直接关联对象开始遍历，与用户线程并发进行
 - 耗时长
 - 重新标记(CMS remark)
 - 需要STW
 - 修正并发标记期间因为用户程序导致的标记产生变动的那一部分对象标记记录 - 采用增量更新方案
 - 比初始标记略长，但远比并发标记短
 - 并发清除(CMS concurrent sweep)
 - 清除所有标记为死亡的对象
 - 不需要移动对象，所以可以与用户线程并发进行
 - 耗时长
- **特点**
 - 目标 - 获取最短gc停顿时间
 - 基于标记清除算法

- 并发收集 - 最耗时的是并发标记和并发清除但都和用户线程一起完成，总体看，内存回收与用户线程并发完成 - 并发低停顿收集器（Concurrent Low Pause Collector）
- 缺点
 - CMS对处理器资源非常敏感，并发阶段虽然不会暂停用户线程，但会占用一部分线程（或者说处理器的计算能力）导致应用程序变慢，降低总吞吐量，导致用户程序执行速度降低
 - 由于CMS无法处理“浮动垃圾”（Floating Garbage），导致可能出现“Concurrent Mode Failure”导致另一次STW的Full GC产生
 - 浮动垃圾：并发的两个阶段运行程序运行还会有垃圾对象产生，但这部分出现在标记过程以后，只能下次清理
 - 需要预留足够内存空间在并行的时候供用户线程使用，所以CMS不能等待老年代被填满了才gc，需要预留空间 - 可以设置参数-XX:CMSInitiatingOccupancyFraction的值来提高CMS触发百分比，降低内存回收频率，但太高会容易引发并发失败
 - 一旦预留的内存无法满足新的内存分配，就会出现并发失败Concurrent Mode Failure，就不得不启动后备预案：冻结线程执行，临时启用Serial Old来收集老年代，停顿时间变长。并发失败越多，性能大幅度下降
 - 基于标记清除算法导致大量空间碎片产生，因为无法分配大对象额导致不得不提前出发Full GC
 - 解决方法
 - -XX:+UseCMSCompactAtFullCollection开关参数（默认开启，jdk9之后废弃） - 不得不FullGC时开启碎片整理 - 要移动对象所以无法并发 - 停顿时间变长
 - -XX:CMSFullGCsBeforeCompaction（jdk9之后废弃） - 若干次（数量由参数决定）不整理空间的FullGC之后，下一次进入Full GC前先碎片整理（默认0）

Garbage First (G1) 收集器

- 运行步骤（大致分为四个步骤）
 - 初始标记(Initial Marking)
 - 只标记一下GC Roots直接关联的对象
 - 修改TAMS指针的值，让并发标记时用户线程能够在可用的Region中分配新对象
 - 短暂STW
 - 借用Minor GC的时候同步完成，所以实际该阶段无额外停顿
 - 并发标记(Concurrent Marking)
 - 从GC ROOTS开始进行可达性分析
 - 与用户线程并发进行，耗时长
 - 最终标记(Final Marking)
 - 短暂STW
 - 处理并发阶段产生的原始快照记录（SATB），进行修正
 - 多条收集器线程并行
 - 筛选回收(Live Data Counting)
 - 负责更新Region统计数据，对各个Region的回收价值和成本进行排序，根据用户期望停顿时间制定回收计划，自由选择任意多个Region构成回收集，把回收集的Region的存活对象复制到空的Region，清理掉旧的Region空间
 - 涉及到对象移动，必须STW
 - 多条收集器线程并行

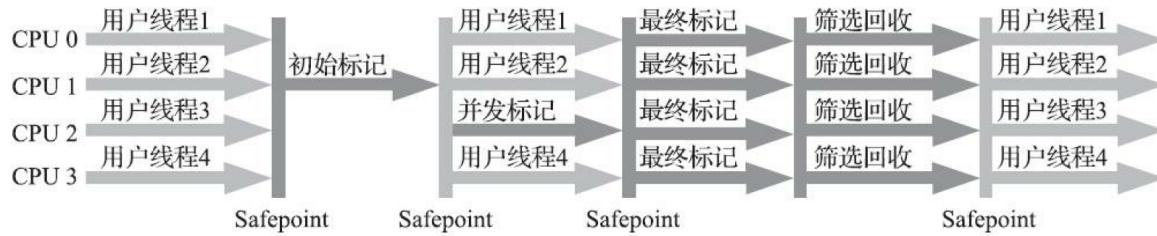
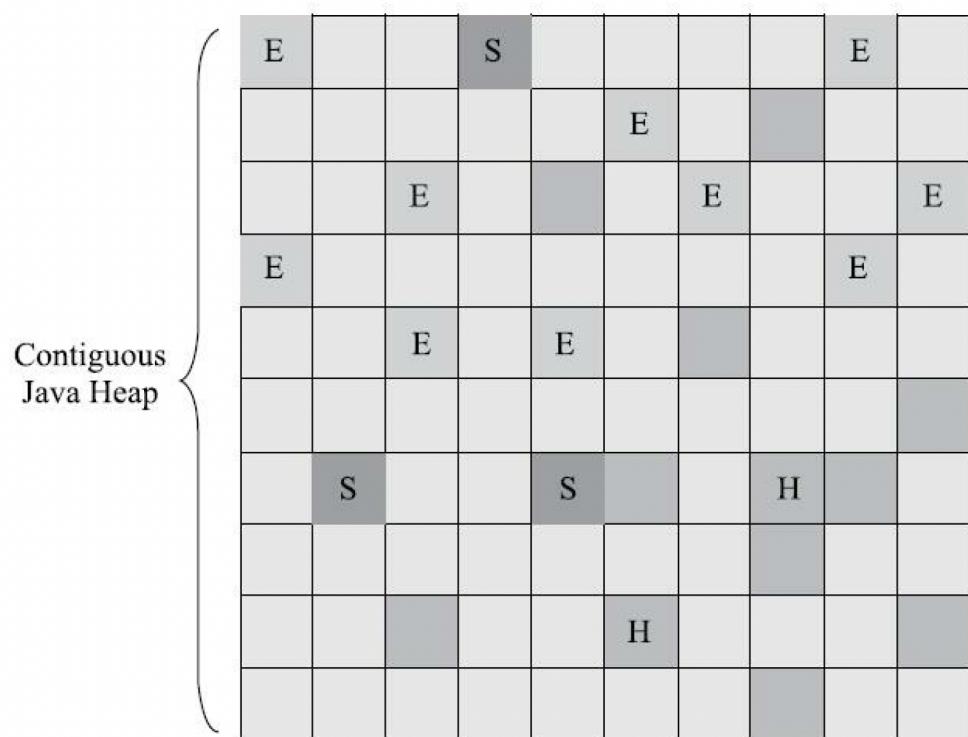


图3-13 G1收集器运行示意图

○

- 特点

- 里程碑式成果 - 开创了收集器面向局部收集的设计思路+基于Region的内存布局形式
- "全功能的垃圾收集器" (Fully-Featured Garbage Collector)
- 面向服务端应用
- 目标 - 希望能建立一个"停顿时间模型" (Pause Prediction Model) 的收集器 - 能够支持指定长度为M毫秒的时间片段内，消耗在垃圾收集上的时间大概率不超过N毫秒 - 可以由用户指定期望的停顿时间，从而可以在不同场景下取得关注吞吐量+延迟之间的平衡
- 实现
 - 实现目标关键1 - 基于Region的堆内存布局

图3-12 G1收集器Region分区示意图^[3]

-
- G1可以面向堆内存任何部分来组成回收集（Collection Set - CSet）进行回收，衡量标准不再是属于哪个分代，而是哪块内存存放的垃圾数量最多，回收收益最大 - Mixed GC
- 虽然也遵循分代收集理论，但G1不再坚持固定大小以及固定数量的分代区域划分，而是把连续的堆划分多个大小相等的独立区域（Region），每个区域根据需要可以使Eden、Survivor或者老年代空间，根据扮演的角色来采取不同策略处理 - 收集效果提高
- XX: G1HeapRegionSize设置Region大小 (1-32MB)，为2的n次幂
- Humongous区域**
 - Region中的特殊一类

- 存储大对象，当一个对象占Region一半大小时判定为大对象，就存放到N个连续Humongous Region之中
- 大多数情况视为老年代一部分
- 仍然保留新生代老年代概念，但这两个代都不再固定了，是一系列可以不连续的动态集合
- 实现目标关键2 - 具有优先级的区域回收方式
 - 能够建立可预测停顿时间模型的原因 - 将Region作为单次回收的最小单位，即每次收集到的内存空间都是Region大小的整数倍，就可以有计划地避免全区域垃圾收集
 - 根据各个Region里面的垃圾堆积的价值大小，即回收所获得的空间大小以及回收所需时间经验值，然后后台维护一个优先级列表，根据用户设置的允许停顿时间（-XX: MaxGCPauseMillis指定，默认值200ms），优先回收价值最大的Region - Garbage First名字由来
- 以上两点保证了g1能在有限时间内取得尽可能高的收集效率
- 实现细节
 - 如何解决跨Region引用对象
 - 使用记忆集避免全堆扫描
 - 每个region都要维护自己的记忆集，一般是记录别的region指向自己的指针以及是哪些卡页范围。在存储结构上本质是哈希表，key：别的region的起始地址，Value：一个集合，里面存储的元素是卡表的索引号 -- 一种双向卡表结构（卡表是“我指向谁”，这种结构还记录了“谁指向我”）比原来的卡表实现复杂
 - Region数量非常多 - G1有着更高的内存占用负担 - 至少要消耗大约相当于堆容量的10%-20%的额外内存来维持收集器工作
 - 并发标记阶段如何保证收集线程与用户线程互不干扰
 - G1采用修正引用关系变化导致标记的错误的方案 - 原始快照（SATB）
 - 并发时新对象的产生问题 - G1为每一个Region设计了两个名为TAMS（Top at Mark Start）的指针，Region中的一部分空间划分为并发时新对象分配，对象地址必须要在这两个指针位置之上，默认该地址以上的对象被隐式标记过，默认为存活而不回收
 - 如果并发时内存不够无法分配，也要STW来FullGC而延长回收
 - 如何建立可预测停顿时间模型
 - 以衰减均值（Decaying Average）为理论基础实现的 - 记录每个Region回收耗时、脏卡数量等可测量成本，分析得出平均值、标准差、置信度等统计信息。
 - 衰减均值比普通的平均值更受新数据影响，更准确代表“最近”平均状态，Region统计状态越新越能决定其回收价值
 - 通过以上信息预测，由哪些Region组成回收集，才可以不超过期望停顿时间的约束下获得最高收益
- 期望停顿时间必须符合实际（默认200ms），否则垃圾回收跟不上分配速度，导致频繁地引发FullGC而STW，降低性能，所以一般设置为一两百或者两三百都很合理
- 从G1开始，最先进的垃圾收集器设计导向都趋向于追求能应付应用的内存分配速率（Allocation Rate），不追求一次清理完Java堆
- 与CMS相比较，从最传统算法理论来看，G1更有发展潜力
 - G1整体看基于“标记-整理”算法，但从局部（两个Region之间）看又基于“标记复制”算法，这两种都意味着无内存碎片，不容易触发下一次gc
- 不过g1也无法做到压倒性优势，它的弱势也有不少，例如用户程序运行时，无论是为了垃圾收集产生的内存占用（Footprint）还是程序运行时额外执行负载（Overload）都比CMS高

- G1每个Region都需要维护一份卡表，实现很复杂，导致其记忆集（和其它内存消耗）可能会占堆容量的20%甚至更多；CMS就一份卡表，记录了老年代到新生代的引用，反过来则没有（新生代朝生夕死）
- 执行负载角度上，虽然在维护卡表上都是用写屏障，CMS用写后屏障；G1除了使用写后屏障来做同样的卡表维护（更繁琐），为了实现原始快照算法，使用写前屏障跟踪并发时的指针变化情况，因为其能减少并发标记与重新标记的消耗，避免最终标记像CMS那样过长，但给用户程序运行阶段带来额外负担；CMS的写屏障实现是直接的同步操作，而G1因为写屏障消耗更多运算资源，只能实现类似于消息队列的结构，写前写后要做的都放进队列里，再异步处理

- 应用

- JDK9发布后，G1宣告取代Parallel Scavenge+ParallelOld的组合，称为服务端模式下的默认垃圾收集器，CMS被沦为不推荐（Deprecate）
- G1和CMS从不同方面看各有利弊，需要具体场景来定量比较，实践经验来看，小内存应用CMS表现大概率优于G1，大内存应用上G1大多能发挥其优势，优劣势的java堆容量平衡点通常在6-8GB

低延迟垃圾收集器

Shenandoah收集器

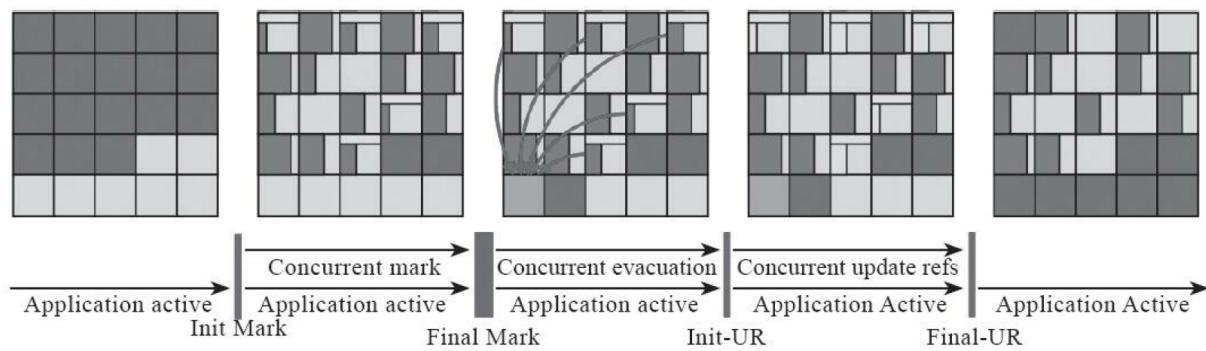


图3-16 Shenandoah收集器的工作过程

-

- 执行步骤

- 初始标记：与g1一样
- 并发标记：与g1一样
- 最终标记（Final Marking）：与g1一样，统计回收价值最高region组成回收集
- 并发清理（Concurrent Cleanup）：清理一个存活对象都没有的Region - Immediate Garbage Region
- 并发回收（Concurrent Evacuation）：（核心差异）并发，把回收集存活对象复制到空Region - 用读屏障和“Brooks Pointers”转发指针（对象最前面存的一个引用字段，一般引用自己，复制移动后改为新对象地址）解决移动后对象引用问题（可能引用还是旧地址无法瞬间改变）
- 初始引用更新（Initial Update Reference）：未有动作，仅建立线程集合点，短暂STW
- 并发引用更新（Concurrent Update Reference）：并发，按内存物理地址顺序线性搜索改新值
- 最终引用更新（Final Update Reference）：修改GC roots的引用，最后的STW
- 并发清理（Concurrent Cleanup）：回收集全部清理

- 特性

- 非Oracle公司制作，属于OpenJDK
- 任何堆大小，**gc停顿时间限制在10ms以内**
- 相比于g1
 - 回收阶段能够进行**并发标记整理算法**
 - 默认不使用分代收集，**无分代**
 - **无复杂记忆集，改用连接矩阵（Connection Matrix）**，降低维护消耗和伪共享概率

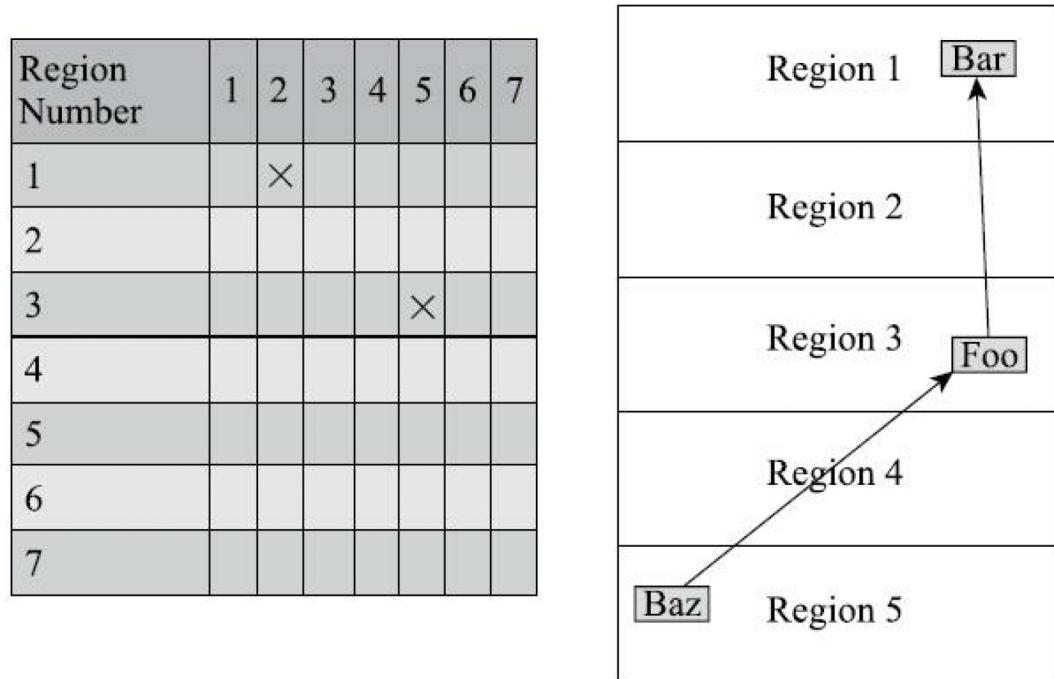


图3-15 Shenandoah收集器的连接矩阵示意图

ZGC收集器（Z Garbage Collector）

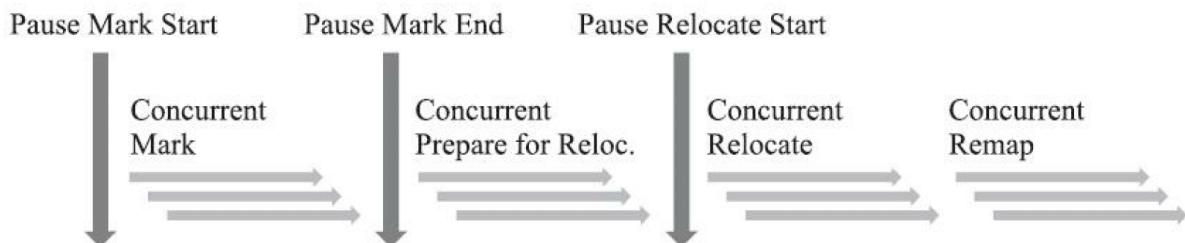


图3-22 ZGC运作过程

- 执行步骤
 - 并发标记（Concurrent Mark）：与g1无区别，前后有初始、最终标记（都有短暂STW）
 - 并发预备重分配：全部的Region都要扫描，建立重分配集，省去记忆集
 - 并发重分配：重分配，核心阶段，重分配集对象复制到新Region，重分配集的Region都维护一个转发表（Forward Table），记录旧对象到新对象转向关系，如果并发时访问旧对象，被预置的内存屏障截获，根据转发表转到新对象，并修改指针的值（指针的自愈，只有第一次需要转发）。因此Region复制完可以立马释放并保留转发表
 - 并发重映射（Concurrent Remap）：修正重分配集旧对象的所有引用，不迫切因为指针自愈，主要是加速并能释放转发表
- 特性
 - 低延迟垃圾收集器，与shenandoah目标一样
 - 基于Region内存布局，（暂时）不设分代，使用了读屏障、染色指针和内存多重映射等技术实现并发的标记-整理算法

- Region大小可分为不同容量，具有动态创建和销毁
- 染色指针（colored Pointer）：标记对象时，把标记信息记载引用对象的指针上
 - 不用修改引用即可清理region
 - 减少gc时内存屏障使用数量（尤其是写屏障维护引用）
- 无记忆集，也不用写屏障 - 限制了能承受的对象分配速率 - 浮动垃圾容易占领内存所以对象分配不能太高 - 只能将堆容量设大一点
- 支持“NUMA-Aware”内存分配

如何选择合适的垃圾收集器

Epsilon收集器

- 不能够垃圾收集，但能够对堆进行管理、对象分配等让虚拟机正常运行 - 最小化功能的实现
- 负载小，内存占用小
- 在一些短时间、小规模的服务形式中很适合，只要虚拟机正确分配内存，堆耗尽前退出

收集器权衡

- 衡量垃圾收集器的三个最重要的指标：内存占用（Footprint）、吞吐量（Throughput）、延迟（Latency） - 构成“不可能三角”。三个整体随着技术进步会越来越好，但同时满足三个完美表现的收集器很难实现甚至不可能，优秀收集器最多同时满足两项
- 需要关注三个因素
 - 应用程序的关注点
 - 吞吐量 - 数据分析、科学计算，目标尽快算出结果
 - 延迟 - SLA应用，延迟导致直接影响服务质量
 - 内存占用 - 客户端或者嵌入式应用
 - 运行应用的基础设施 - 硬件规格，处理器数量，分配内存大小，操作系统等
 - 使用JDK发行商和版本号 - ZingJDK/Zulu, OracleJDK, OpenJDK, OpenJ9或者其他？JDK对应哪个虚拟机规范？

虚拟机及垃圾收集器日志

- JDK9之后，HotSpot所有功能的日志都收敛到了“-Xlog”参数上
- `-Xlog[:[selector][:[output][:[decorators][:output-options]]]]`
- 最关键参数 - 选择器Selector，由标签（Tag）+日志级别（Level）组成
 - 标签可以理解为某个功能模块的名字，比如垃圾收集器标签为“gc”
 - 其他标签模块：`add, age, alloc, annotation, aot, arguments, attach, barrier, biasedlocking, blocks, bot, breakpoint, bytecode`
 - 日志级别从低到高：`Trace, Debug, Info, Warning, Error, Off`六种级别，决定了输出信息的详细程度，默认Info
- 修改器（Decorator）
 - 要求每行日志输出都附加额外内容，可支持信息包括
 - time:当前日期和时间
 - uptime:虚拟机启动到现在经过的时间，以秒为单位
 - timemillis:当前时间的毫秒数，相当于System.currentTimeMillis()的输出
 - uptimemillis:虚拟机启动到现在经过的毫秒数
 - timenanos:当前时间的纳秒数，相当于System.nanoTime()的输出

- **up timenanos**: 虚拟机启动到现在经过的纳秒数
- **pid**: 进程ID
- **tid**: 线程ID
- **level**: 日志级别
- **tags**: 日志输出的标签集
 - 默认值 `uptime`、`level`、`tags`
 - e.g. `[3.080s] [info] [gc,cpu] GC(5) User=0.03s Sys=0.00s Real=0.01s`

实战：内存分配与回收策略

对象优先在Eden分配

- 大多数情况下，对象先在新生代**Eden**区中分配，当**Eden**区没有足够空间分配时，虚拟机发起一次**Minor GC**

大对象直接进入老年代

- 大对象是指需要大量连续内存空间的Java对象，最典型的大对象就是很长的字符串或者元素数量庞大的数组
- 大对象容易在内存还有很多的情况下提前触发gc，复制对象时意味着高额内存复制开销
- HotSpot提供了-XX: PretenureSizeThreshold参数，制定大于该设置值的对象直接在老年代分配，目的是避免**Eden**区及两个**Survivor**区之间来回复制（该参数只能用在Serial和ParNew上）

长期存活的对象将进入老年代

- 虚拟机给每个对象定义了一个对象年龄（Age）计数器，存储在对象头中
- 在**Eden**诞生对象，经历一次**Minor gc**，如果存活并被**Survivor**容纳，就被移动到**Survivor**空间中，设定年龄为1岁
- 每熬一次**minor gc**增加1岁，当年龄增到一定程度（默认15），晋升到老年代
- 晋升年龄阈值可以设置参数-XX: MaxTenuringThreshold设置

动态对象年龄判定

- 如果**Survivor**空间中相同年龄所有对象大小总和大于**Survivor**空间的一半，年龄大于或等于该年龄对象就可以直接进入老年代，无需等到-XX: MaxTenuringThreshold要求的年龄

空间分配担保

- 发生**Minor GC**之前，虚拟机必须检查老年代最大可用连续空间是否大于新生代所有对象空间
 - 大于，那这一次**Minor GC**确保安全
 - 不大于，虚拟机查看-XX: HandlePromotionFailure参数是否设置允许担保失败（Handle Promotion Failure）
 - 允许
 - 检查老年代最大可用连续空间是否大于历次晋升到老年代对象的平均大小
 - 大于，可以尝试一次**Minor GC**（冒险）
 - 小于，只能进行一次**Full GC**
 - 不允许
 - 只能进行一次**Full GC**

- 标记复制算法，有可能出现大量对象在Minor GC之后仍存活，Survivor空间无法容纳，需要老年代进行分配担保，这些对象直接进入老年代，但前提是老年代有容纳这些对象的剩余空间，但每次回收时无法得知有多少对象存活，所以只能取每次回收晋升到老年代对象容量的平均大小作为经验值来比较老年代剩余空间
- 冒险在于，有可能出现存活对象远超历史平均值，导致担保失败，只能重新发起一次**Full GC**，停顿时间变长
- 通常还是为了避免Full GC过于频繁而打开-XX: HandlePromotionFailure参数
- **JDK 6 Update 24之后的规则：**只要老年代的连续空间大于新生代对象总大小或者历次晋升平均大小，就会进行**Minor GC**，否则Full GC (-XX: HandlePromotionFailure参数不再使用)

第四章&第五章 虚拟机性能监控、故障处理工具及内存调优

基础故障处理工具

- jps (JVM Process Status Tool) - **查询进程状况信息** - 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 (LVMID) - 使用频率最高
- jstat (JVM Statistics Monitoring Tool) - **监视虚拟机各种运行状态信息** (进程中的类加载、内存、垃圾收集、即时编译等) - 可以持续观察虚拟机内存中各分区的使用率和gc统计数据
- jinfo (Configuration Info for Java) - **实时查看和调整虚拟机各项参数**
- jmap (Memory Map for java) - **查看内存使用情况** - 生成堆转储快照 (**heap dump**或者**dump**文件)
- 还可以查询finalize执行队列、java堆和方法区的详细信息，如空间使用率、当前用的是哪种收集器等
- jhat (JVM Heap Analysis Tool) - **与jmap搭配使用，分析堆转储快照** - 内置一个微型HTTP/HTML服务器，**生成dump分析结果可以在浏览器查看** - 一般不直接用它
- jstack (Stack Trace for Java) - **用于生成虚拟机当前时刻的线程快照** (**threaddump**或者**javacore**文件) - **方法堆栈信息** - 可以用来迅速定位问题线程 - 可结合应用日志使用

可视化故障处理工具

- **JHSDDB** - 基于服务性代理的调试工具
- **JConsole** - 基于JMX (Java Management Extensions) 的可视化监视、管理工具，通过JMX的MBean对系统进行信息收集和参数动态调整
- **VisualVM** - 功能最强大的运行监视和故障处理程序之一 - **多合故障处理工具** - 对应用程序性能影响小，可以直接应用到生产环境 - 具备插件扩展功能
- **JMC (Java Mission Control) & JFR (Java Flight Recorder)** - **可持续在线的监控工具 & 可持续收集数据** (过程数据) - JMC可以分析本地应用以及连接远程ip使用，提供实时分析线程、内存、CPU、GC等信息的可视化界面 - 采取JMX协议与虚拟机通信，显示虚拟机MBean提供的数据，也是JFR的分析工具，展示其数据

JVM内存调优

- **主要目的**
 - **减少GC频率和Full GC次数**，过多会占用很多系统资源 (主要CPU)，影响系统吞吐量
- **使用JDK提供的内存查看工具，比如JConsole或者VisualVM**
- **调优过程**
 - 监视GC状态，使用各种JVM工具，查看当前日志，并分析当前堆内存快照和gc日志，根据实际情况来决定是否优化
 - 通过JMX的MBean或者jmap生成dump文件，使用VisualVM或者Eclipse自带的Mat分析dump文件

- 如果参数设置合理，没有超时日志，GC频率、耗时都不高则不用优化，**如果GC时间超过1秒或者频率过高，则必须优化**
- 调整**GC类型和内存分配**，使用一台或多台机器进行测试，进行性能比较，再做最后修改，通过不断试验和试错，分析并找到最合适的参数

调优命令

- Sun JDK监控和故障处理命令 - 见基础故障处理工具

调优工具

- **jdk自带监控工具**（下面这两都是java5开始jdk自带的监控管理控制台，可以对jvm内存、线程、类等监控）
 - JConsole
 - JVMTI
- **第三方工具**
 - Mat (Memory Analyzer Tool)
 - GChisto

第六章 类文件结构

概述

- 各种不同平台的java虚拟机，以及所有平台都统一支持的程序存储格式 - **字节码（Byte Code）是构成平台无关性的基石** - “一次编写，到处运行”
- 虚拟机另一种中立特性 - **语言无关性**越来越被重视
- 实现**语言无关性的基础**仍然是**虚拟机和字节码存储格式**
- java虚拟机不包括java语言在内的任何程序语言绑定，它只与“**Class文件**”这种特定的二进制文件格式所关联，Class文件包含了**java虚拟机指令集、符号表以及若干其他辅助信息**
- 基于安全方面考虑，Java虚拟机规范中要求在Class文件必须应用许多强制性的语法和结构化约束、但**图灵完备的字节码格式**，保障任意语言都可以表示为一个java虚拟机可以接受的有效Class文件，因此可以基于这些来创造其他生成Class文件的语言在虚拟机上运行
- 字节码指令所能提供的语言描述能力必须比那些语言足够强大，可以多条组合使用

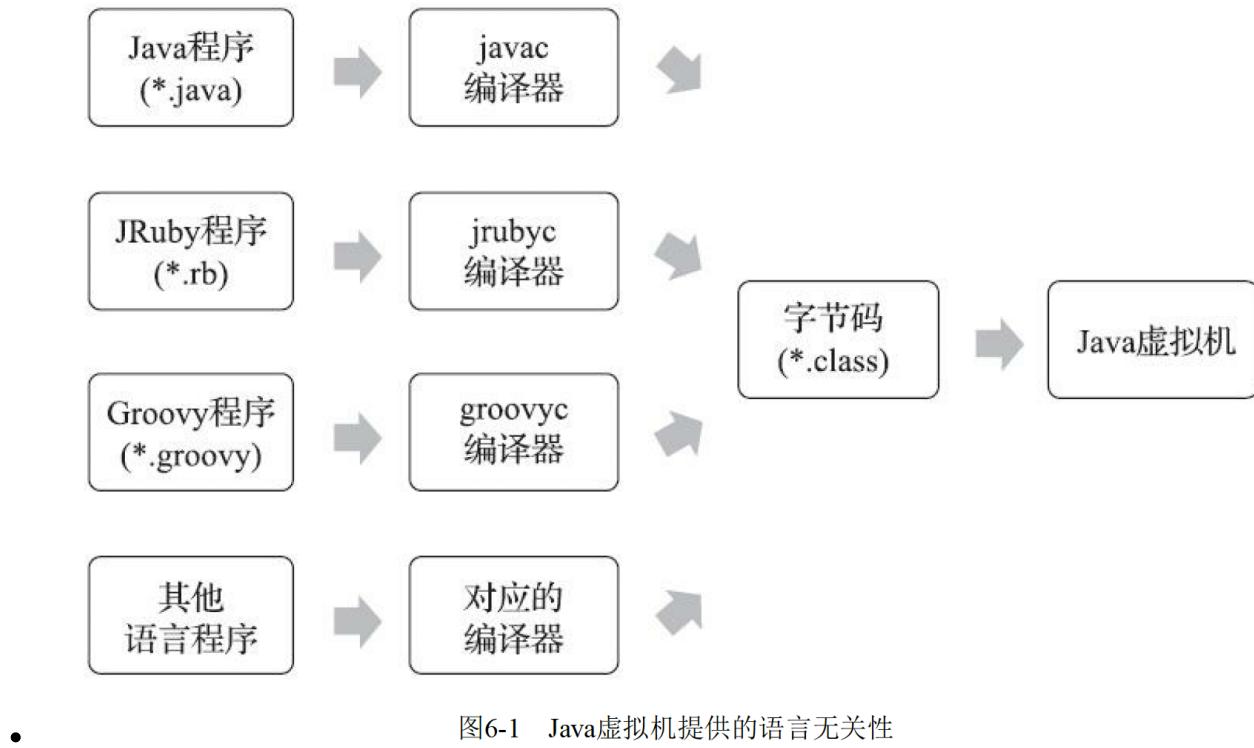


图6-1 Java虚拟机提供的语言无关性

Class类文件的结构

- 任何一个**Class**文件都对应着唯一的一个类或者接口的定义信息，但反过来，类或者接口并不一定都得定义在文件里，可以不需要以磁盘文件形式存在（比如可以动态生成类or接口，直接进入类加载器中）
- 定义**
 - Class文件是一组以8个字节为基础单位的二进制流，各个数据项目按照严格顺序紧凑排列在文件之中，中间没有任何分隔符 - 几乎全部都是必要数据，没有空隙 - 当需要占用8个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字节进行存储
- 数据结构**
 - 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，该结构只有**两种数据类型 - 无符号数 + 表**
 - 无符号数**
 - 基本数据类型**
 - u1、u2、u4、u8**来表示1、2、4、8个字节的无符号数
 - 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**
 - 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
 - 所有表命名习惯性以'**_info**'结尾
 - 用于描述有层次关系的复合结构的数据
- 整个Class文件本质上可以视为一张表，由以下所示数据项按照严顺序排列组成(16项)

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

- 集合
 - 无论是无符号数还是表，当需要描述同一类型但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，称这一系列连续的某一类型的数据为某一类型的“集合”
- 因为没有分隔符，所以这16个数据项无论顺序还是数量，甚至数据存储的字节序（Byte Ordering, Class文件中字节序为Big-Endian），都是被严格限定的，不许改变

- 魔数 (**Magic Number**) : 头四个字节被称为魔数
 - 唯一作用: 确定这个文件是否为虚拟机可接受的**Class文件**
 - 不用文件扩展名识别的原因: 基于安全性考虑, 扩展名容易随便改动
 - 文件格式制定者可以任选魔数值
 - **Class文件的魔数值为0xCAFEBABE**
- Class文件版本
 - 紧接着魔数的四个字节存储class文件的版本号
 - 次版本号 (**Minor Version**) : 占第5和第6字节
 - 主版本号 (**Major Version**) : 占第7和第8字节
 - java的版本号从45开始, JDK1.1之后每个jdk大版本发布主版本号向上+1 (JDK1.0-1.1使用了45.0-45.3)
 - 高版本jdk能向下兼容以前版本的**Class文件**, 但不能运行以后版本的**Class文件**, 虚拟机必须拒绝执行超过其版本号的**Class文件**
 - 次版本号从jdk1.2以后, 直到**jdk12**之前都未使用, 全部为0 - 直到**jdk12**出现, 一些新特性需要以“公测”形式放出, 所以重新启用副版本号 - 若**Class文件**使用了该版本**jdk**未列入正式特性清单的预览功能, 必须把本次版本号标志为65535

常量池

- 紧接主次版本号为常量池入口
- Class文件里的资源仓库 - 文件中与其他数据项关联最多的数据 - 占用**Class文件**空间最大的数据项目之一
- 表类型数据项目
- 常量池容量计数值 (**constant_pool_count**) : 常量池入口先放置一个u2类型的数据 - 常量池的常量数量不固定需要计数
 - 从1开始计数而不是0开始 e.g. 十六进制0x0016 - 十进制22 - 代表常量池有21项常量, 索引值范围1-21
 - 第0项常量空出来的原因: 如果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”, 可以把索引值设为0
 - 只有常量池容量计数从1开始, 其他比如接口索引集合、字段表集合、方法表集合等的容量计数都是0开始
- 常量池主要存放两大类常量: 字面量 (**Literal**) 和符号引用 (**Symbolic References**)
 - 字面量: 比较接近java语言的常量概念, 如**文本字符串**、**final的常量值**
 - 符号引用: 属于编译原理方面的概念, 主要包括以下几类常量
 - 被模块导出或开放的包 (**Package**)
 - 类和接口的全限定名 (**Fully Qualified Name**)
 - 字段的名称和描述符 (**Descriptor**)
 - 方法的名称和描述符
 - 方法句柄和方法类型 (**Method handle**、**Method Type**、**Invoke Dynamic**)
 - 动态调用点和动态常量 (**Dynamically-Computed Call Site**、**Dynamically-Computed Constant**)
 - java代码在进行**javac**编译时, 在虚拟机加载**Class文件**的时候进行动态连接 (第七章)。Class文件中不会保存各个方法、字段最终在内存中的布局信息, 这些字段、方法的符号引用不经过虚拟机在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。类加载时, 会从常量池获得对应的符号引用, 然后在类创建时解析、翻译到具体的内存地址之中
- 每个常量都是一个表, 一共有17种不同类型的常量 (不同的表结构)
 - 分类

- 11种最开始就有的
- 4种动态语言相关的（为了支持动态语言调用）
- 2种模块化相关的的（为了支持java模块化系统Jigsaw）
- 共同特点
 - 表结构起始第一位 - u1类型的标志位（tag），代表当前常量属于哪种常量类型，如下表所示

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
CONSTANT_MethodHandle_info	15	表示方法句柄
CONSTANT_MethodType_info	16	表示方法类型
CONSTANT_Dynamic_info	17	表示一个动态计算常量

(续)

类 型	标 志	描 述
CONSTANT_InvokeDynamic_info	18	表示一个动态方法调用点
CONSTANT_Module_info	19	表示一个模块
CONSTANT_Package_info	20	表示一个模块中开放或者导出的包

- 先查tag - 然后查找对应常量类型的表结构

- jdk的bin目录下有专门分析Class文件的字节码工具：javap

访问标志

- 紧接着两位就是访问标志（Access_flags） - u2类型
- 用于识别类或者接口层次的访问信息，包括Class是类还是接口；是否定义为public；是否定义为abstract；如果是类，是否声明final等。
- 一共有16个标志位可以使用，目前只定义了其中九个，没有使用到的标志位一律为0

第七章 虚拟机类加载机制

概述

- 定义
 - java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
- java语言里，类型加载、连接和初始化过程都是在程序运行期间完成的
 - 缺点：该策略导致java语言进行提前编译会面临额外困难，会让类加载增加性能开销

- 优点：提供了极高的扩展性和灵活性 - java天生可以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
 - e.g. 动态组装应用 - 编写一个面向接口的应用程序，可以等运行时在指定其实际实现类，用户可以通过java预置的或自定义类加载器，让某个本地的应用程序在运行时从网络或其他地方加载一个二进制流作为其程序代码的一部分
 - 以上应用方式目前广泛应用：Applet、JSP、OSGi技术

类加载时机

- 一个类型的生命周期（七个阶段）
 - 加载（Loading）
 - 验证（Verification）
 - 准备（Preparation）
 - 解析（Resolution）
 - 初始化（Initialization）
 - 使用（Using）
 - 卸载（Unloading）
- 验证、准备、解析三个部分统称为连接（Linking）

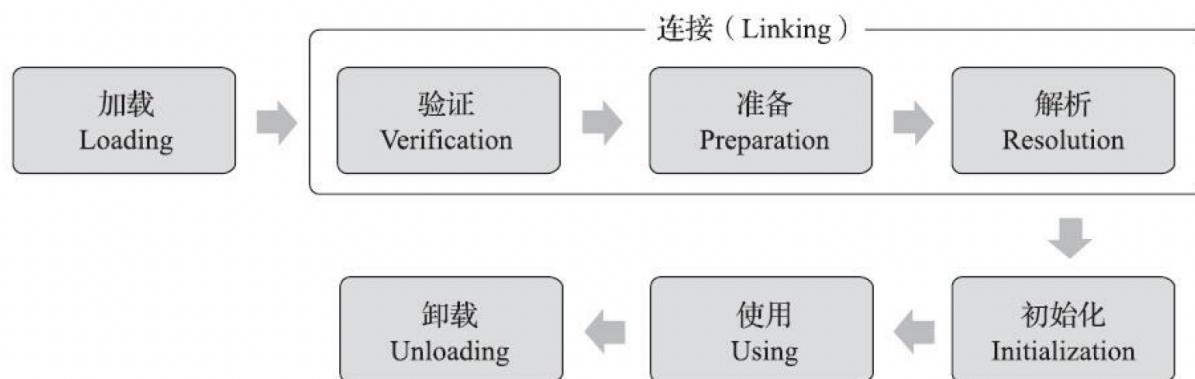


图7-1 类的生命周期

- 加载、验证、准备、初始化、卸载五个阶段顺序确定，加载过程必须按部就班地“开始”
- 解析阶段不一定：在某些情况下可以在初始化之后再开始，为了支持JAVA语言的运行时绑定特性（动态绑定或晚期绑定）
- 不是按部就班地“完成”而是“开始”，这些阶段通常互相交叉混合进行，会在一个阶段执行的过程中调用、激活下一个阶段
- 加载阶段 - 虚拟机规范没有强制约束
- 初始化阶段 - 虚拟机规范严格规定有且只有六个情况如果类没初始化过必须立即对类进行“初始化”（前面阶段自然需要在此之前开始）
 - 遇到new、getstatic、putstatic或invokestatic这四条字节码指令时
 - 典型场景使用这四条指令
 - 使用new关键词实例化对象
 - 读取或设置一个类型的静态字段（被final修饰、已在编译期把结果放入常量池的静态字段除外）
 - 调用一个类型的静态方法
 - 使用java.lang.reflect包的方法对类型进行反射调用的时候
 - 初始化类时发现其父类还没有过初始化时，先初始化父类
 - 虚拟机启动时，需要指定一个要执行的主类（包含main()方法那个类），先初始化该主类

- 当使用JDK7新加入的动态语言支持时，如果一个`java.lang.invoke.MethodHandle`实例最后的解析结果为`REF_getStatic`、`REF_putStatic`、`REF_invokeStatic`、`REF_newInvokeSpecial`四种类型的方法句柄，并且这个方法句柄对应的类没有进行过初始化时，先触发初始化
- 当一个接口中定义了jdk8新加入的默认方法（`default`修饰的接口方法）时，如果这个接口的实现类发生了初始化，那该接口要在其之前被初始化
- 主动引用：以上六种场景的行为被称为对一个类型进行主动引用
- 被动引用：除此之外所有引用类型的方式都不会触发初始化，被称为被动引用
 - 通过子类引用父类静态字段，不会导致子类初始化（父类直接定义了这个静态字段，所以会初始化）
 - 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
- 接口初始化
 - 接口也有初始化过程
 - 接口不能使用静态语句块“`static{}`”，但编译期仍然会为接口生成`<clinit>()`类构造器，用于初始化接口中所定义的成员变量
 - 与类真正有所区别的是六种中的第三类：接口初始化时，并不要求其父接口全部初始化完成，只有真正使用到父接口的时候（如引用接口中定义的常量）才会初始化

类加载过程

加载阶段

- 目的：加载阶段，虚拟机需要完成三件事
 - 通过一个类的全限定名来获取定义此类的二进制字节流（获取）
 - 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（转化）
 - 在堆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（生成）
- 这三点定义不是特别具体，所以虚拟机实现和java应用灵活度很大
 - “获取”这条规则，没有说一定要从某个`Class`文件或者指定位置中获取，所以来源可以非常灵活 - zip、网络、运行时计算生成、由其他文件生成、从数据库读取、从加密文件获取等
- 相对于类加载过程的其他阶段，非数组类型的加载阶段（主要说“获取”）是可控性最强的阶段
 - 加载阶段可以使用虚拟机里内置的引导类加载器来完成
 - 也可以使用用户自定义的类加载器去完成 - 可以用自定义的类加载器去控制字节流获取方式（重写一个类加载器的`findClass()`或`loadClass()`方法） - 实现自主获取运行代码的动态性
- 数组类型本身不通过类加载创建，由虚拟机直接在内存中动态构造出来，但数组类的元素类型（Element Type - 去掉所有维度的类型）最终还是要靠类加载器来完成加载
- 数组类C创建过程的规则
 - 如果数组的组件类型(Component Type，指的是数组去掉一个维度的类型，注意和前面的元素类型区分开来)是引用类型，那就递归采用本节中定义的加载过程去加载这个组件类型，数组C将被标识在加载该组件类型的类加载器的类名称空间上(这点很重要，在7.4节会介绍，一个类型必须与类加载器一起确定唯一性)
 - 如果数组的组件类型不是引用类型(例如`int[]`数组的组件类型为`int`)，Java虚拟机将会把数组C标记为与引导类加载器关联
 - 数组类的可访问性与它的组件类型的可访问性一致，如果组件类型不是引用类型，它的数组类的可访问性将默认为`public`，可被所有的类和接口访问到

验证阶段

- 连接阶段第一步
- 目的
 - 确保Class文件的字节流中包含的信息符合《java虚拟机规范》的全部约束要求，保证运行后不会危害虚拟机自身安全
- 为确保严谨性，验证阶段工作量在虚拟机类加载过程占了很大比重
- 大致会完成四个阶段的检验动作
 - 文件格式验证
 - 验证字节流是否符合Class文件格式规范，并能被当前版本的虚拟机处理
 - 可能包括下面这些验证点
 - 规定魔数是否在开头
 - 主次版本号是否在当前虚拟机接受范围内
 - 常量池的常量是否有不被支持的常量类型（检查tag）
 - 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UTF-8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
 - ...
 - 主要目的：保证输入的字节流能正确地解析并存储于方法区之内，格式上符合描述一个java类型信息的要求
 - 基于二进制字节流进行
 - 通过这个阶段后，该字节流才能被允许进入虚拟机内存的方法区中进行存储，后面三个验证阶段全部基于方法区的存储结构上进行
 - 元数据验证
 - 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
 - 可能包括的验证点
 - 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)
 - ...
 - 主要目的：对累的元数据信息进行语义校验，保证不存在与《java语言规范》定义相悖的元数据信息
 - 字节码验证
 - 验证最复杂的阶段
 - 主要目的：通过数据流分析和控制流分析，确定程序语义是合法的、符合逻辑的。该阶段对类的方法体（Class文件中的Code属性）进行校验分析，保证该类的方法在运行时不会危害虚拟机安全
 - 可能校验点
 - 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，例如不会出现类似于“在操作栈放置了一个int类型的数据，使用时却按long类型来加载入本地变量表中”这样的情况
 - 保证任何跳转指令都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总是有效的，例如子类赋给父类安全，父类赋给子类或者毫不相干类不安全

- ...
 - 即便通过字节码验证这种非常严密的检查，仍然无法保证方法体安全
 - 为避免执行时间过长，jdk6之后尽量把校验辅助措施移到javac编译器里
 - 具体做法：给方法体Code属性的属性表新增加一个项名为“**StackMapTable**”的新属性 - 描述了方法体所有的基本块开始时本地变量表和操作栈应有的状态
 - **java**虚拟机只需要检查这个新属性的记录是否合法即可，类型推导转为类型检查，从而省去大量校验时间
- 符号引用验证
 - 发生阶段：虚拟机将符号引用转化为直接引用时，该转化动作在解析阶段发生
 - 对类自身以外（常量池中的各种符号引用）的各类信息进行匹配性校验 - 给类是否缺少或者禁止访问它依赖的某些外部类、方法、字段等资源
 - 可能得校验点
 - 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
 - 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、方法的可访问性(private、protected、public、package)是否可被当前类访问
 - ...
 - 主要目的：确保解析行为能正常执行
 - 如果无法通过该校验，Java虚拟机会抛出
 - java.lang.IncompatibleClassChangeError**的子类异常，比如**java.lang.IllegalAccessException**、**java.lang.NoSuchFieldError**、**java.lang.NoSuchMethodError**等
- 校验阶段重要、但非必须执行，有些代码可能已经被反复使用过和验证过，生产环境实施阶段可以考虑使用-Xverify: none参数来关闭大部分类校验措施，缩减类加载时间

准备阶段

- 目的
 - 正式为类中定义的变量（静态变量）分配内存并设置类变量初始值的阶段
- 概念上，这些变量所使用的内存都应当在方法区中进行分配，但方法区本身是一个逻辑上的区域，jdk7之前HotSpot用永久代实现方法区，则符合这个逻辑概念；**jdk8之后**，类变量会随着**Class**对象一起存放在**java堆**里，这时候“类变量在方法区”仅是逻辑概念
- 两个注意点
 - 内存分配只是包括类变量，不包括实例变量
 - 准备阶段下初始化“通常情况”下是数据类型的零值，而不是代码里写的赋的值。因为此时并没有开始执行任何java方法，赋值为指定值的**putstatic**指令是程序被编译后，存放于类构造器**<clinit>**方法之中，所以该赋值必须等到类的初始化阶段才会被执行

表7-1 基本数据类型的零值

数据类型	零 值	数据类型	零 值
int	0	boolean	false
long	0L	float	0.0f
short	(short) 0	double	0.0d
char	'\u0000'	reference	null
byte	(byte) 0		

- 特殊情况：如果类字段的字段属性表中存在**ConstantValue**属性，那在准备阶段变量值就会被初始化为**ConstantValue**属性所指定的初始值（主要就是**final**修饰的静态变量，也就

是常量，编译时javac会为它生成ConstantValue属性，在准备阶段虚拟机就会根据ConstantValue的设置将它赋值为指定的值)

解析阶段

- 目的
 - java虚拟机将常量池内的符号引用替换为直接引用的过程
- 符号引用 (Symbolic Reference)
 - 定义
 - 以一组符号来描述所引用的目标，符号可以是任何形式的字面量，只要使用时能无歧义地定位到目标即可
 - 特点
 - 与虚拟机实现的内存布局无关
 - 引用目标不一定是加载到虚拟机内存中的内容
 - 虚拟机内存布局可以不一样，但能接受的符号引用必须一样
- 直接引用 (Direct References)
 - 定义
 - 可以直接指向目标的指针、相对偏移量或者是一个能间接定位到目标的句柄
 - 特点
 - 和虚拟机实现的内存布局直接相关
 - 同一个符号引用在不同虚拟机实例上翻译出来的直接引用一般不会相同
 - 如果有了直接引用，那引用目标必定已经在虚拟机内存中存在
- 解析时机
 - 虚拟机实现可以根据需要自行判断，到底是在类被加载器加载时就对常量池中的符号引用进行解析，还是等到一个符号引用将要被使用前才去解析它
 - 对方法或者字段的访问，也会在解析阶段中对他们的可访问性(public、protected、private、< package >)进行检查
- 对同一个符号引用进行多次解析请求
 - 除了invokedynamic指令以外，虚拟机实现可以对第一次解析的结果进行缓存，比如运行时直接引用常量池中的记录，并把常量标识为已解析状态，从而避免重复解析
 - 同一实体，符号引用解析成功一次，后面应该次次成功；第一次失败了，后面的解析请求应该次次收到相同异常，哪怕后面该符号成功加载
 - 对于invokedynamic指令，当遇到一次该指令触发解析了一个符号引用时，后面的invokedynamic指令解析结果不一定相同
 - 该指令目的就是用于支持动态语言，它对应的引用称为动态调用点限定符(Dynamically-Computed Call Site Specifier)，这里的“动态”含义是必须等到程序实际运行到这条指令，解析动作才会开始，而其他触发解析的指令都是静态的，可以实现为提前解析
- 解析动作主要针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用点限定符7类符号引用进行，分别对应常量池的8种常量类型，后4种都是和动态语言支持相关

初始化阶段

- 除了加载阶段应用程序可以通过自定义类加载器的方式局部参与外，其余动作都完全由java虚拟机来主导控制，直到初始化阶段，虚拟机才真正开始执行类中编写的java程序代码，主导权移交给应用程序
- 目的
 - 根据程序员通过程序编码制定的主观计划去初始化类变量和其他资源
 - 执行类构造器< clinit >()方法的过程

- <clinit>()方法的产生
 - <clinit>()方法不是java代码中直接编写的方法，是javac编译器自动生成物
 - 由编译器自动收集类中的所有类变量的赋值动作和静态语句块（static{}）中的语句合并产生的
 - 收集顺序：由语句在源文件中出现的顺序决定，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
 - 该方法与类的构造函数（即在虚拟机视角中的实例构造器<init>()方法）不同，他不需要显式地调用父类构造器，因为子类的该方法执行前，父类的<clinit>()方法已经执行完毕。因此在Java虚拟机中第一个被执行的()方法的类型肯定是java.lang.Object
 - <clinit>()方法对于类或接口来说并不是必需的，如果一个类中没有静态语句块，也没有对变量的赋值操作，那么编译器可以不为这个类生成<clinit>()方法
 - 接口中不能使用静态语句块，但仍然有变量初始化的赋值操作，因此接口与类一样都会生成<clinit>()方法。但接口执行的<clinit>()方法不需要先执行父接口的<clinit>()方法，因为只有当父接口中定义的变量被使用时，父接口才会被初始化
 - 接口的实现类在初始化时也一样不会执行接口的<clinit>()方法
- 同一个类加载器，一个类型只会被初始化一次

类加载器

- 加载阶段，设计者故意将“获取”部分这个动作放到java虚拟机外部实现，以便应用程序自行决定如何去获取所需的类 - 该代码称为类加载器（Class Loader）

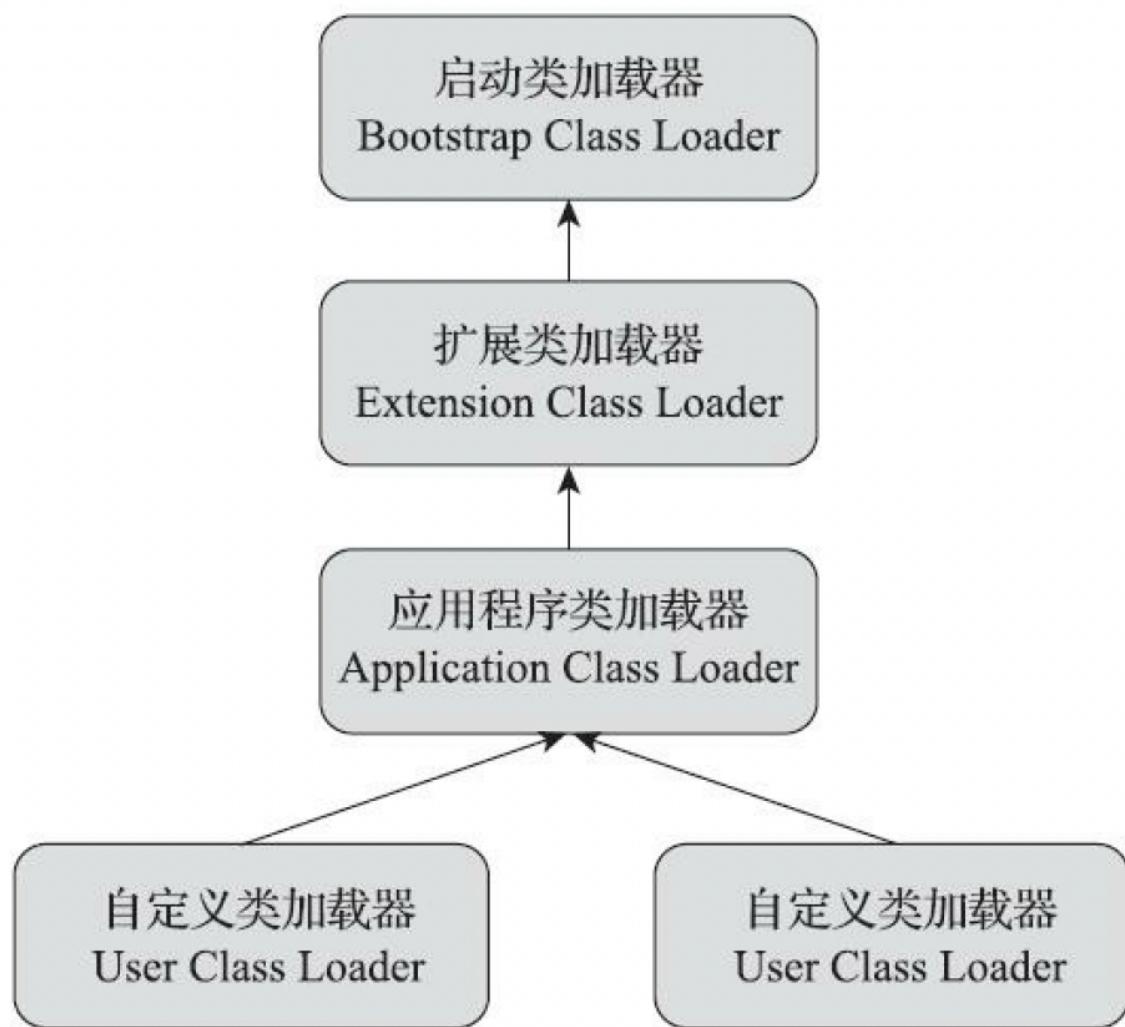
类与类加载器

- 类加载器的作用
 - 对于任意一个类，必须由加载它的类加载器和这个类本身一起共同确立其在Java虚拟机中的唯一性，每一个类加载器，都拥有一个独立的类名称空间
 - 比较两个类是否“相等”，只有在这两个类是由同一个类加载器加载的前提下才有意义
 - 即使这两个类来源于同一个Class文件，被同一个Java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

- 按java虚拟机角度分类类加载器
 - 一种是启动类加载器（Bootstrap ClassLoader），用c++实现，是虚拟机自身一部分
 - 另一种就是其他的类加载器，由java实现，独立于虚拟机之外，全部继承自抽象类 `java.lang.ClassLoader`
- 按java开发人员角度，类加载器分类则更细致
- 类加载构架
 - 自JDK 1.2以来，Java一直保持着三层类加载器、双亲委派的类加载架构
- 三层类加载器
 - 启动类加载器（Bootstrap Class Loader）
 - 负责加载存放在 <JAVA_HOME>\lib 目录，或者被-Xbootclasspath参数所指定的路径中存放的，而且是Java虚拟机能够识别的（按照文件名识别，如rt.jar、tools.jar，名字不符合的类库即使放在lib目录中也不会被加载）类库加载到虚拟机的内存中
 - 启动类加载器无法被Java程序直接引用，用户在编写自定义类加载器时，如果需要把加载请求委派给引导类加载器去处理，直接使用null代替
 - 扩展类加载器（Extension Class Loader）

- 在类sun.misc.Launcher\$ExtClassLoader中以Java代码的形式实现
- 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所有的类库
- 一种java系统类库的扩展机制
- 允许用户将具有通用性的类库放置在ext目录里以扩展Java SE的功能，在JDK9之后，这种扩展机制被模块化带来的天然的扩展能力所取代
- 直接用java实现 - 可以直接在程序中使用该加载器加载class文件
- 应用程序类加载器（Application Class Loader）
 - 由sun.misc.Launcher\$AppClassLoader来实现，用java代码
 - 是ClassLoader类中的getSystemClassLoader()方法的返回值 - 被称为“系统类加载器”
 - 负责加载用户类路径(ClassPath)上所有的类库，可以直接在代码中使用这个类加载器
 - 若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
- 双亲委派模型（Parent Delegation Model）



-
- 自定义类加载器
 - 可以扩展功能，如增加除了磁盘位置之外的Class文件来源，或者通过类加载器实现类的隔离、重载功能等
- 定义
 - 图中展现的类加载器之间“通常”的协作关系
 - 除了启动类加载器外，其他类加载器都拥有自己的父类加载器 - 非继承关系，通常使用组合（Composition）关系来复用父加载器的代码
- 工作过程

- 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最顶层的启动类加载器中，只有当父加载器反馈自己无法完成这个加载请求(它的搜索范围中没有找到所需的类)时，子加载器才会尝试自己去完成加载
- 好处
 - Java中的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
 - e.g.类**java.lang.Object**，它存放在rt.jar之中，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最顶端的启动类加载器进行加载，因此Object类在程序的各种类加载器环境中都能够保证是同一个类。反之，如果没有使用双亲委派模型，都由各个类加载器自行去加载的话，如果用户自己也编写了一个名为java.lang.Object的类，并放在程序的ClassPath中，那系统中就会出现多个不同的Object类，Java类型体系中最基础的行为也就无从保证，应用程序将会变得一片混乱
- 模型的程序实现

代码清单7-10 双亲委派模型的实现

```

protected synchronized Class<?> loadClass(String name, boolean resolve) throws ClassNotFoundException
{
    // 首先，检查请求的类是否已经被加载过了
    Class c = findLoadedClass(name);
    if (c == null) {
        try {
            if (parent != null) {
                c = parent.loadClass(name, false);
            } else {
                c = findBootstrapClassOrNull(name);
            }
        } catch (ClassNotFoundException e) {
            // 如果父类加载器抛出ClassNotFoundException
            // 说明父类加载器无法完成加载请求
        }
        if (c == null) {
            // 在父类加载器无法加载时
            // 再调用本身的findClass方法来进行类加载
            c = findClass(name);
        }
    }
    if (resolve) {
        resolveClass(c);
    }
}

return c;
}

```

微信 642620018，获取慕课网、极客时间、腾讯课堂、网易云课堂全套课程

- 过程
 - 先检查请求加载的类型是否被加载过
 - 若没有
 - 有父加载器
 - 调用父加载器的**loadClass ()** 方法
 - 没有父加载器==null
 - 默认使用启动类加载器作为父加载器
 - 假如父类加载器加载失败，抛出**ClassNotFoundException**异常，就调用自己的**findClass ()** 尝试进行加载

破坏双亲委派模型

- 第一次破坏
 - 发生在双亲委派模型出现之前 - jdk1.2以前
 - 面对已经存在的用户自定义类加载器的代码，只能在引入该模型时做妥协，为了兼容这些代码，无法再以技术手段避免**loadClass()**被子类覆盖的可能性，只能在JDK1.2之后的

java.lang.ClassLoader中添加一个新的**protected方法findClass()**, 并引导用户编写的类加载逻辑时尽可能去重写这个方法, 而不是在**loadClass()**中编写代码

- 按照**loadClass()**方法的逻辑, 如果父类加载失败, 会自动调用自己的**findClass()**方法来完成加载, 这样既不影响用户按照自己的意愿去加载类, 又可以保证新写出来的类加载器是符合双亲委派规则

- 第二次破坏

- 由模型自身的缺陷导致: 双亲委派很好地解决了各个类加载器协作时基础类型的一致性问题(越基础的类由越上层的加载器进行加载), “基础”总是被用户代码继承、调用, 但问题是如果基础类型又要调用回用户代码就没辙了
- 解决方法
 - 线程上下文类加载器 (**Thread Context ClassLoader**) - 这个类加载器可以通过**java.lang.Thread**类的**setContextClassLoader()**方法进行设置, 如果创建线程时还未设置, 它将会从父线程中继承一个, 如果在应用程序的全局范围内都没有设置过的话, 那这个类加载器默认就是应用程序类加载器
 - 使用该加载器可以让父类加载器去请求子类加载器完成类加载 - 逆向使用类加载器 - 违背双亲委派模型
 - JNDI服务就是使用了这个加载器去加载所需的SPI服务代码

- 第三次破坏

- OSGi为了实现模块热部署 (Hot Deployment) 即能再不重启的情况下替换java应用程序, 改变了类加载器的机制
- 关键是它自定义的类加载器机制的实现, 每一个程序模块 (OSGi中称为Bundle) 都有一个自己的类加载器, 当需要更换一个Bundle时, 就把Bundle连同类加载器一起换掉以实现代码的热替换
- 在OSGi环境下, 类加载器不再双亲委派模型推荐的树状结构, 而是进一步发展为更加复杂的网状结构
- 收到类加载请求时, OSGi按照七个步骤的顺序进行类搜索, 只有前两个符合双亲委派, 后面的类查找都是在平级的类加载器中进行的

Java模块化系统 (Java Platform Module System, JPMPS)

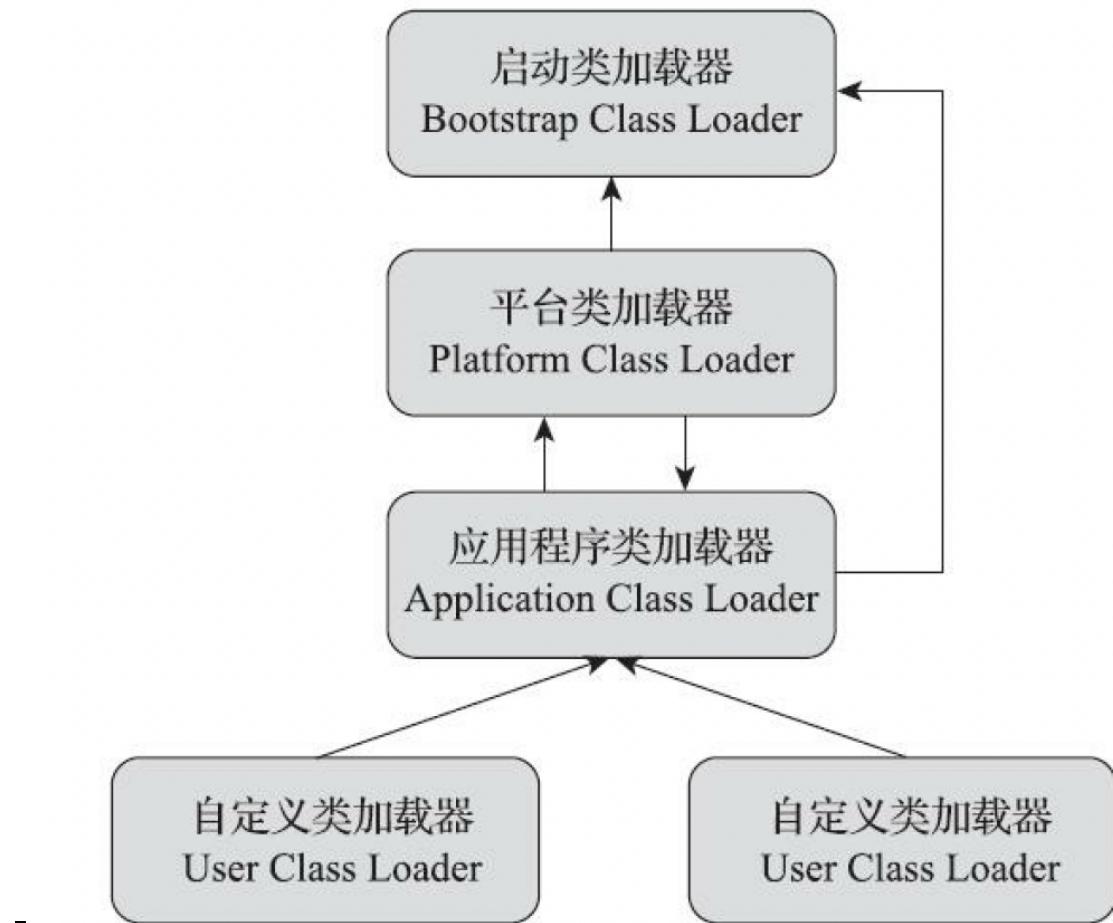
- jdk9引进
- **关键目标: 可配置的封装隔离机制**
- 对类加载架构做出了相应的变动
- 除了代码, **java模块定义**包含内容
 - 依赖其他模块的列表
 - 导出的包列表, 即其他模块可以使用的列表
 - 开放的包列表, 即其他模块可反射访问模块的列表
 - 使用的服务列表
 - 提供服务的实现列表
- 首先解决**jdk9之前的基于类路径 (ClassPath)** 来查找依赖的可靠性问题
 - 以前类加载缺少运行时依赖 - 只能等到该类加载、连接时才报运行异常
 - 现在模块声明了对其他的模块的显式依赖, 因此虚拟机启动时验证依赖关系在运行期是否完备, 缺失就直接启动失败
 - 更精细的可访问性控制

模块兼容性

- 模块路径
 - 为了能够兼容传统的类路径查找机制，jdk9提出了与“类路径”相对应的模块路径（ModulePath）
 - 某个类库到底是模块还是传统jar包，取决于放在哪个路径 - 放在类路径就当做传统jar包，放在模块路径就当做模块
- 访问规则
 - JAR文件在类路径的访问规则：所有类路径下的JAR文件及其他资源文件，都被视为自动打包在一个匿名模块(Unnamed Module)里，这个匿名模块几乎是没有任何隔离的，它可以看到和使用类路径上所有的包、JDK系统模块中所有的导出包，以及模块路径上所有模块中导出的包
 - 模块在模块路径的访问规则：模块路径下的具名模块(Named Module)只能访问到它依赖定义中列明依赖的模块和包，匿名模块里所有的内容对具名模块来说都是不可见的，即具名模块看不见传统JAR包的内容
 - JAR文件在模块路径的访问规则：如果把一个传统的、不包含模块定义的JAR文件放置到模块路径中，它就会变成一个自动模块(Automatic Module)。尽管不包含module-info.class，但自动模块将默认依赖于整个模块路径中的所有模块，因此可以访问到所有模块导出的包，自动模块也默认导出自己所有的包
- java模块化系统目前不支持在模块定义中加入版本号来管理和约束依赖，本身也不支持多版本号概念和版本选择功能

模块化下的类加载器

- 为了保证兼容性并没有根本性改变类加载器的机制，但也做了如下变化
 - 扩展类加载器被平台类加载器（Platform Class Loader）取代
 - 既然整个jdk都基于模块化进行构建(原来的rt.jar和tools.jar被拆分成数十个JMOD文件)，其中的Java类库就已天然地满足了可扩展的需求，那自然无须再保留<JAVA_HOME>\lib\ext目录
 - 平台类加载器和应用程序类加载器都不在派生自java.net.URLClassLoader
 - 现在启动类加载器、平台类加载器、应用程序类加载器全都继承于`jdk.internal.loader.BuiltinClassLoader`，实现了新的模块化构架下如何从模块加载的逻辑，以及模块中资源可访问性的处理
 - 启动类加载器现在是在Java虚拟机内部和Java类库共同协作实现的类加载器，尽管有了`BootClassLoader`这样的Java类，但保持兼容，所有在获取启动类加载器的场景（譬如`Object.class.getClassLoader()`）中仍然会返回null
 - 虽然仍保持以前的类加载器构架，但委派关系发生了变动



- 当平台及应用程序类加载器收到类加载请求，在委派给父加载器加载前，要先判断该类是否能够归属到某一个系统模块中，如果可以找到这样的归属关系，就要优先委派给负责那个模块的加载器完成加载 - 第四次破坏
- 模块化系统明确规定了三个类加载器负责各自加载的模块，即归属关系

第八章 虚拟机字节码执行引擎

概述

- 执行引擎是java虚拟机核心的组成部分之一
- 虚拟机的执行引擎由软件自行实现，可以不受物理条件制约地定制指令集与执行引擎的结构体系，可执行不被硬件直接支持的指令集格式
- java虚拟机规范规定了java虚拟机字节码执行引擎的概念模型 - 统一外观（Facade）
- 执行字节码时，通常有解释执行（解释器执行）和编译执行（即时编译器产生本地代码执行），也可能两者兼备，也可能同时包含几个不同级别的即时编译器一起工作
- 但输入、输出是一致的
 - 输入 - 字节码二进制流
 - 处理过程 - 字节码解析执行的等效过程
 - 输出 - 执行结果

运行时栈帧结构

- java虚拟机以方法为最基本执行单元
- “栈帧”（Stack Frame）
 - 定义

- 用于支持虚拟机进行方法调用和方法执行背后的数据结构
- 虚拟机栈的栈元素
- 存储内容
 - 局部变量表、操作数栈、动态连接、方法返回地址、附加信息等信息
- 特点
 - 一个方法从调用开始到执行结束的过程对应着一个栈帧在虚拟机栈里面入栈到出栈的过程
 - 每一个栈帧都包含上面提到的存储内容
 - 栈帧中各部分需要多大已经在编译时被分析计算出来并写入到方法表的Code属性之中 - 栈帧需要分配多少内存，仅仅取决于程序源码和具体虚拟机实现的栈内存布局形式，不受运行期数据影响
- 当前栈帧（Current Stack Frame）：对于执行引擎来说，在活动线程中，只有位于栈顶的方法是在运行和生效的 - 当前栈帧。执行引擎所运行的所有字节码指令都只针对当前栈帧进行操作
- 当前方法（Current Method）：与当前栈帧所关联的方法

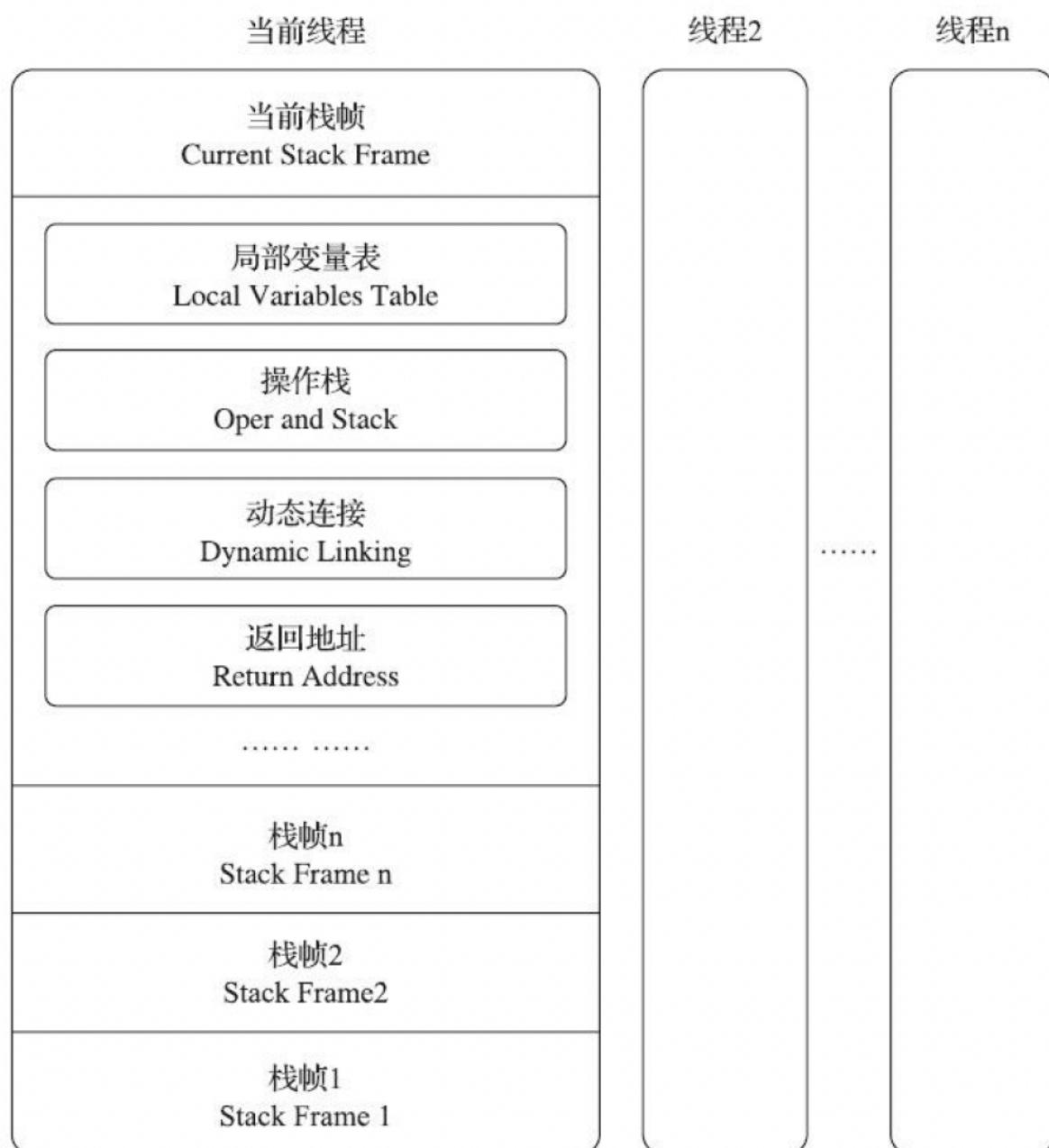


图8-1 栈帧的概念结构

局部变量表 (Local Variables Table)

- 定义
 - 一组变量值的存储空间，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分配的局部变量表的最大容量
- 存储形式
 - 容量以变量槽(Variable Slot)为最小单位
 - 一个变量槽可以存放一个32位以内的数据类型，java中占用不超过32位存储空间的数据类型有boolean、byte、char、short、int、float、reference和returnAddress这8种
 - 第7种reference
 - 表示一个对象实例的引用
 - 虚拟机通过它至少要做到两件事
 - 根据引用直接或间接地查找到对象在Java堆中的数据存放的起始地址或索引
 - 根据引用直接或间接地查找到对象所属数据类型在方法区中的存储的类型信息
 - 第8种returnAddress：目前很少见，指向了一条字节码指令的地址
 - 对于64位的数据类型 (long & double)，Java虚拟机会以高位对齐的方式为其分配两个连续的变量槽空间
- 使用
 - 通过索引定位来使用
 - 索引范围：从0开始至局部变量表最大的变量槽数量
 - 32位数据类型的变量 - 索引N表示使用了第N个变量槽
 - 64位数据类型的变量 - 第N和第N+1个变量槽（不允许任何方式访问其中某一个）
 - 实参到形参的传递：当一个方法被调用，会使用局部变量表来完成参数值到参数变量列表的传递过程
 - 如果是实例方法（无static），第0位索引变量槽默认是用于传递方法所属对象实例的引用，在方法中可以通过**关键字“this”**来访问到这个隐含的参数
 - 其余参数则按照参数表顺序排列，占用从1开始
 - 参数表分配完毕后，再根据方法体内部定义的变量顺序和作用域分配其余的变量槽
 - 变量槽可以重用，如果当前字节码PC计数器的值已经超出了某个变量的作用域，那这个变量对应的变量槽就可以交给其他变量来重用
 - 如果一个局部变量定义了但没有赋初始值，它是完全不能使用的

操作数栈 (Operand Stack)

- 操作栈，后入先出 (LIFO)
- 最大深度在编译的时候被写入Code属性的max_stacks数据项之中
- 存储形式
 - 每一个元素都可以是long和double在内的任意数据类型
 - 32位 - 栈容量为1
 - 64位 - 栈容量为2
- 使用
 - 方法开始执行的时候操作数栈为空
 - 执行过程中 - 出栈和入栈操作
 - 操作数栈元素的数据类型必须与字节码指令的序列严格匹配
- 概念模型中，两个不同栈帧作为不同方法的虚拟机栈的元素，完全相互独立；大多虚拟机实现时，进行了优化，让下面栈帧的部分操作数栈与上面栈帧的部分局部变量表重叠在一起，这样可以省一些空

间，也可以进行方法调用时直接共用一部分数据，无需进行额外的参数赋值传递

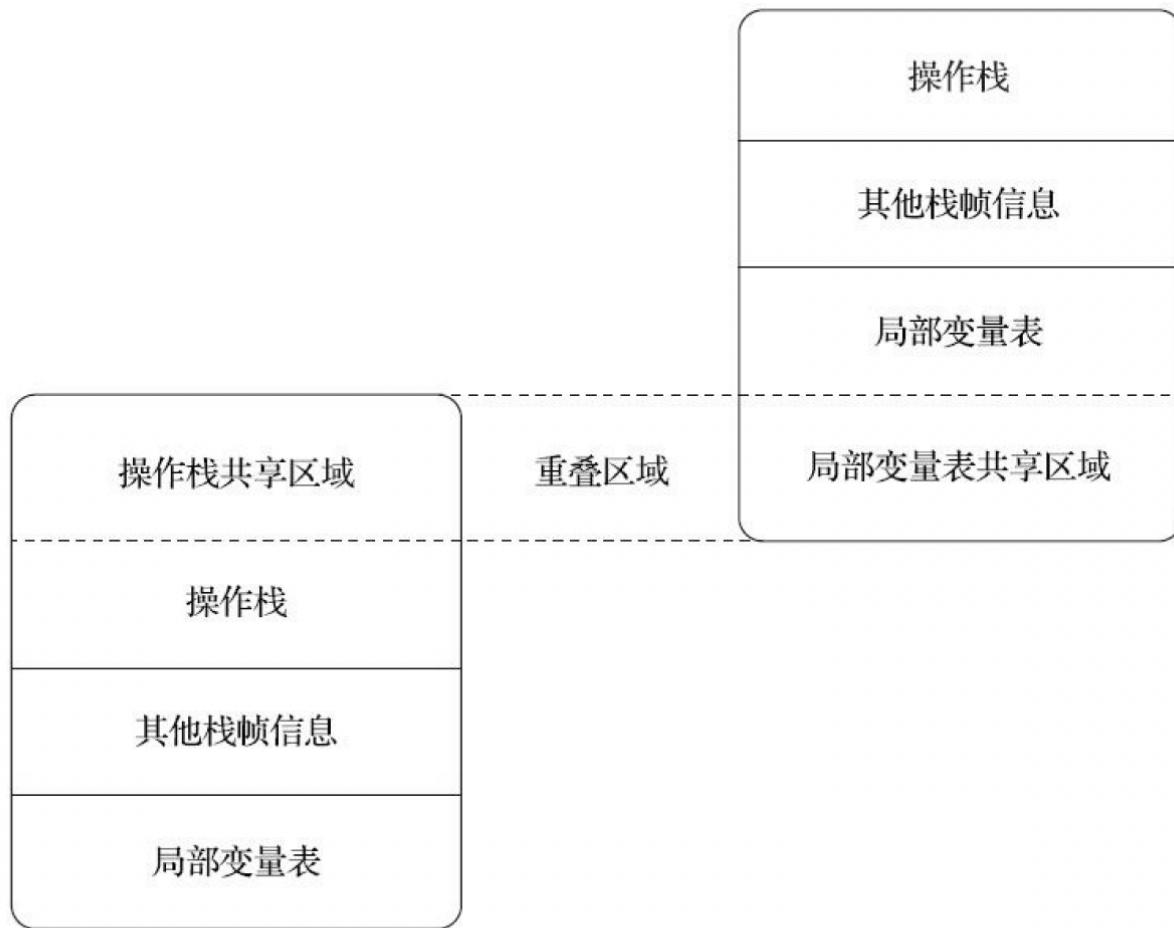


图8-2 两个栈帧之间的数据共享

动态连接

- 每个栈帧都包含一个指向运行时常量池中该栈帧所属方法的引用，持有这个引用是为了支持方法调用过程中的动态连接(Dynamic Linking)
- Class文件的常量池中存有大量的符号引用，字节码中的方法调用指令就以常量池里指向方法的符号引用作为参数
- 这些符号引用一部分会在类加载阶段或者第一次使用的时候就被转化为直接引用，这种转化被称为静态解析;另外一部分将在每一次运行期间都转化为直接引用，这部分就称为动态连接

方法返回地址

- 方法退出
 - “正常调用完成”(Normal Method Invocation Completion)
 - 执行引擎遇到任意一个方法返回的字节码指令，这时候可能会有返回值传递给上层的方法调用者(调用当前方法的方法称为调用者或者主调方法)，方法是否有返回值以及返回值的类型将根据遇到何种方法返回指令来决定
 - “异常调用完成(Abrupt Method Invocation Completion)”
 - 在方法执行的过程中遇到了异常，并且这个异常没有在方法体内得到妥善处理。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，不返回任何返回值
- 方法返回

- 无论采用何种退出方式，在方法退出之后都必须返回到最初方法被调用时的位置，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，用来帮助恢复它的上层主调方法的执行状态
- 一般来说，方法正常退出时，主调方法的PC计数器的值就可以作为返回地址，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，栈帧中就一般不会保存这部分信息
- 实际上等同于把当前栈帧出栈，因此退出时可能执行的操作有：恢复上层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

- 具体虚拟机实现时可增加一些规范里没有描述的信息到栈里，比如与调试、性能收集相关的信息
- 一般把动态连接、方法返回地址、附加信息统称为栈帧信息

方法调用

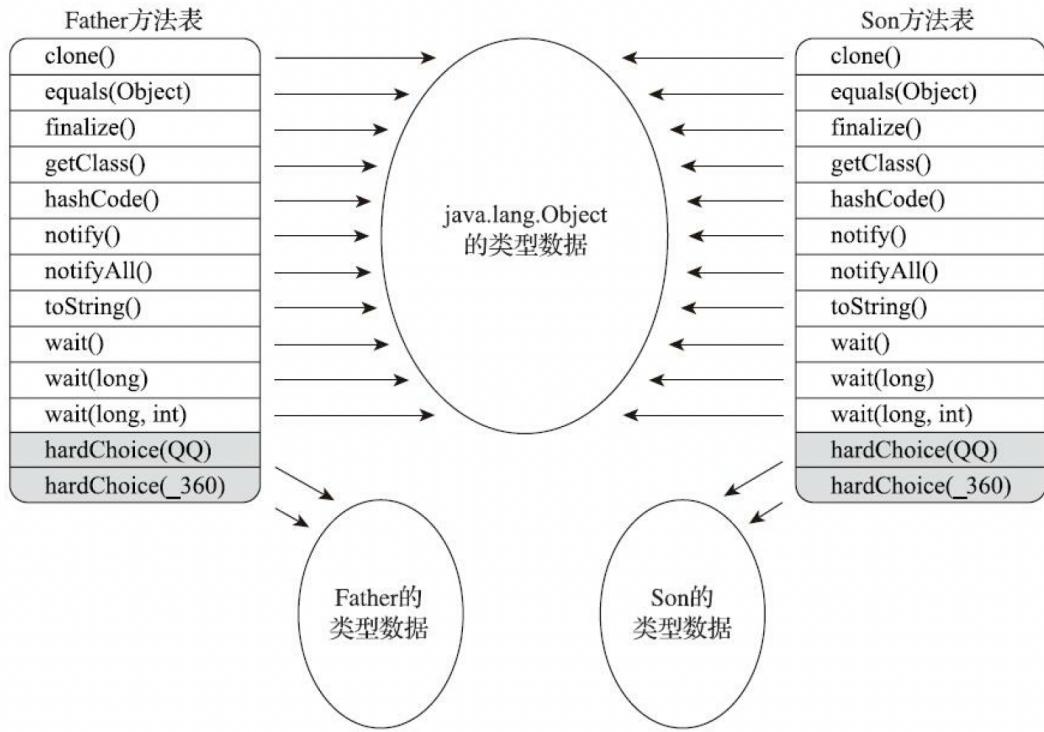
- 唯一任务：确认被调用方法的版本（调哪一个）
- 一切方法调用的目标方法在Class文件里面存储的都只是符号引用，而不是方法在实际运行时内存布局中的入口地址(也就是之前说的直接引用)

解析(Resolution)

- 定义
 - 该方法调用指的是，类加载的解析阶段，会把部分符号引用转化为直接引用，这部分的前提：方法在程序真正运行之前有一个可确定的调用版本，在运行期间不可变 - 编译器可知，运行期不可变
- 分类
 - 主要分为静态方法和私有方法两大类
 - 前者与类型直接关联
 - 后者外部不可访问
- 方法调用字节码指令
 - invokestatic - 用于调用静态方法
 - invokespecial - 用于调用实例构造器<init>()方法、私有方法和父类中的方法
 - invokevirtual - 用于调用所有的虚方法
 - invokeinterface - 用于调用接口方法，会在运行时再确定一个实现该接口的对象
 - 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
- 前4种指令 - 分派逻辑都固化在Java虚拟机内部
- invokedynamic指令 - 分派逻辑是由用户设定的引导方法来决定的
- 只要能被**invokestatic**和**invokespecial**指令调用的方法，都可以在解析阶段中确定唯一的调用版本
 - Java语言里符合这个条件的方法共有静态方法、私有方法、实例构造器、父类方法4种，再加上被final修饰的方法(尽管它使用**invokevirtual**指令调用)，
 - 这些方法统称为**“非虚方法”(Non-Virtual Method)，与之相反，其他方法就被称为“虚方法”(Virtual Method)**
- 解析调用一定是静态过程

分派 (Dispatch)

- 可能是静态也可以是动态
- 按照分配依据的宗量数分单分派和多分派
- 这两类分派方式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
- 静态分派
 - e.g. `Human man = new Man()`
 - “Human” - “静态类型” (**Static Type**)，或者叫“外观类型” (Apparent Type)
 - “Man” - “实际类型” (Actual Type) **或者叫“运行时类型” (Runtime Type)
 - 静态类型的变化仅仅在使用时发生，变量本身的静态类型不会被改变，并且最终的静态类型是在编译期可知的
 - 实际类型变化的结果在运行期才可确定
 - 虚拟机（或者准确地说是编译器）在重载时是通过参数的静态类型而不是实际类型作为判断依据，从而决定哪个重载版本，把这个方法的符号引用写进`invokevirtual`指令的参数中
 - 所有依赖静态类型来决定方法执行版本的分派动作，都属于静态分派
 - 应用：方法重载，但有时候重载版本也不是唯一，只是选择相对合理的版本，因为字面量的模糊性
- 动态分派
 - 与重写密切相关
 - `invokevirtual`指令的运行时解析过程
 - 找到操作数栈顶的第一个元素所指向的对象的实际类型，记作C
 - 如果在类型C中找到与常量中的描述符和简单名称都相符的方法，则进行访问权限校验，如果通过则返回这个方法的直接引用，查找过程结束；不通过则返回`java.lang.IllegalAccessError`异常
 - 否则，按照继承关系从下往上依次对C的各个父类进行第二步的搜索和验证过程
 - 如果始终没有找到合适的方法，则抛出`java.lang.AbstractMethodError`异常
 - 运行期间根据实际类型确认方法执行版本的分派过程称为动态分派
 - 实现
 - 需要运行时在接收者类型的方法元数据中搜索合适的目标方法 - 繁琐
 - 常见的优化手段：为类型在方法区中建立一个虚方法表（Virtual Method Table, vtable），用其索引来代替元数据查找以提高性能
 - 对应的，`invokeinterface`执行时也用到接口方法表 - Interface Method Table (itable)
 - 虚方法表存放着各个方法的实际入口地址
 - 子类重写了方法，子类的地址会替换为子类实现版本的入口地址
 - 子类没有重写，地址入口和父类相同方法的入口一致
 - 为了程序实现方便，具有相同签名的方法，在父类、子类的虚方法表中都应当具有同样的索引序号，这样当类型变换时，仅需要变更查找的虚方法表，就可以从不同的虚方法表中按索引转换出所需的入口地址
 - 虚方法表一般在类加载的连接阶段进行初始化，准备了类的变量初始值后，虚拟机会把该类的虚方法表也一同初始化完毕



- 单分派和多分派

- 方法的宗量：方法的接收者+方法的参数
- 单分派：根据一个宗量对目标方法进行选择
- 多分派：根据多个宗量对目标方法进行选择
- 如今的Java语言是静态多分派、动态单分派的语言

第十二章 Java内存模型与线程

概述

- 多任务处理的原因
 - 计算机运算速度与存储和通信子系统速度差距过大，时间浪费在磁盘I/O，网络通信或数据库访问上，太浪费运算性能，所以要多任务处理
 - 一个服务端同时对多个客户端提供服务是现在的具体的并发应用场景，每秒事务处理数（TPS）衡量服务性能，该值与程序并发能力密切相关，并发越协调效率越高TPS越高

硬件的效率与一致性

- 处理器与内存速度的矛盾
 - 运算任务执行肯定要与内存交互，也就是I/O，但速度又太慢了
 - 解决：系统添加一层或者多层读写速度接近于处理器运算速度的高速缓存（Cache）来作为内存与处理器之前的缓冲 - 基于高速缓存的存储交互
 - 运算需要的数据复制到缓存，运算能快速进行，运算结束后再把缓存同步回内存
- 缓存一致性问题（Cache Coherence）
 - 多路处理器系统中，每个处理器都有自己的高速缓存，又都共享同一main memory（共享内存多核系统 Shared Memory Multiprocessors System），运算涉及到同一块主内存就可能导致各自缓存数据不一致
 - 解决：各个处理器访问缓存时都遵循一些协议，读写根据协议操作（这类协议有MSI、MESI（Illinois Protocol）、MOSI、Synapse、Firefly及Dragon Protocol）

- 内存模型

- 特定操作协议下，对特定的内存或高速缓存进行读写访问的过程抽象
- 不同架构物理机器可以拥有不一样的内存模型，java虚拟机也有自己的内存模型
- 目的：关注虚拟机吧变量值存储内存和从内存中取出变量值的底层细节

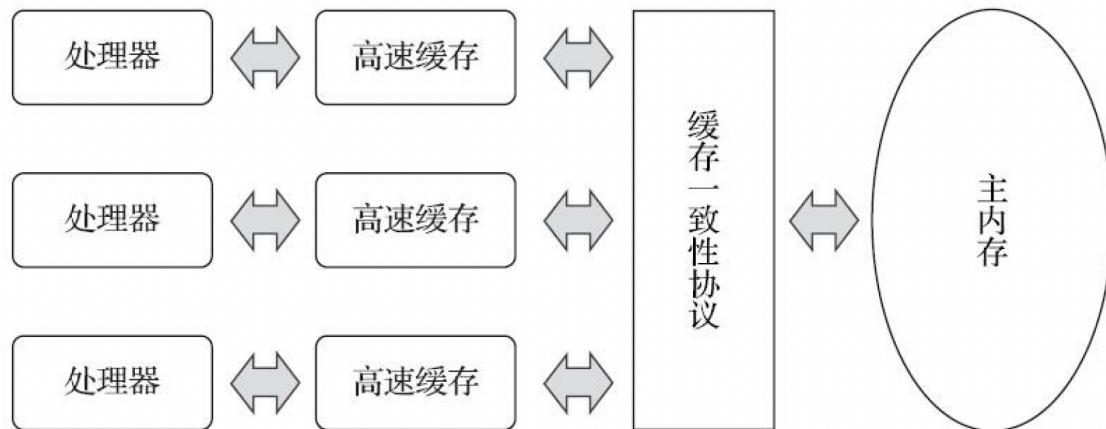


图12-1 处理器、高速缓存、主内存间的交互关系

。

- 乱序执行优化 (Out of Order Execution)

- 处理器在计算之后将乱序执行的结果重组，保证结果与顺序执行的结果是一致的 -- 为了让处理器运算单元充分利用而可能出现的处理器优化
- Java虚拟机的即时编译器种也有指令重排序 (Instruction Reorder) 优化

Java内存模型

主内存与工作内存

- 所有变量存在主内存 (Main Memory) 中 (物理上仅是虚拟机内存一部分) -- java堆中的对象实例数据部分，直接对应物理硬件的内存
- 每条线程有自己的工作内存 (Working Memory) -- 虚拟机栈的部分区域，优先存于寄存器和高速缓存中。保存了该线程使用的变量主内存副本，所有操作必须都在工作内存之中，不能直接读写主内存的数据，不同线程之间无法直接访问对方工作内存中的变量，线程间变量值的传递需要通过主内存

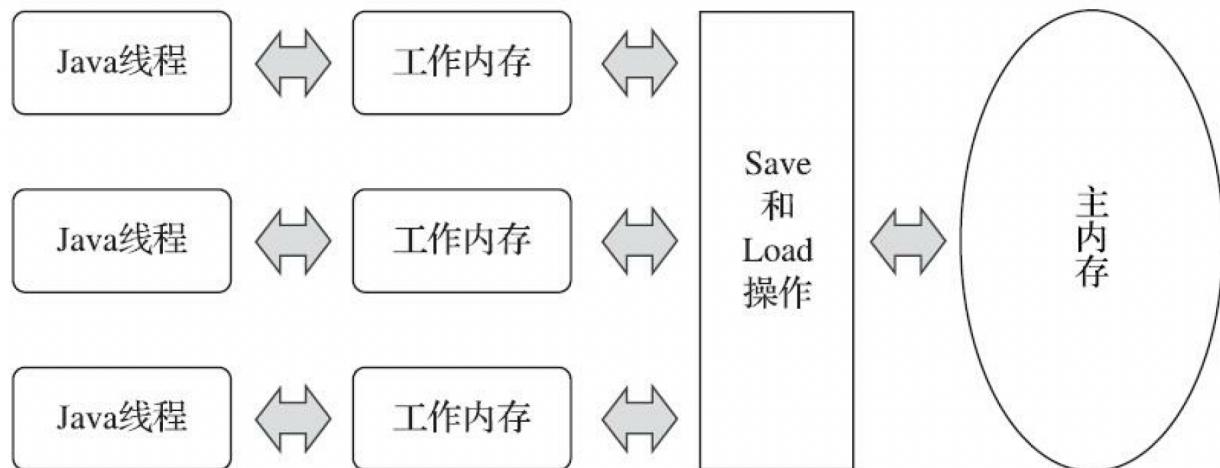


图12-2 线程、主内存、工作内存三者的交互关系 (请与图12-1对比)

- reference指向的对象可以被共享，但reference本身在栈的局部变量表中是线程私有的
- 对象的引用、对象某个字段可能被复制，但虚拟机不会把整个对象复制给工作内存

内存间的交互操作

- java内存模型定义了**8种操作**（原子的不可分的，double和long允许load store read write在某些平台有例外）来完成变量在主内存和工作内存之间的交互
 - **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的值赋给工作内存的变量，每当虚拟机遇到一个给变量赋值的字节码指令时执行这个操作。
 - **store**（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。

- **其他规则**

- read&load 和 store&write必须顺序执行，但不要求连续执行，中间可以穿插别的指令
- 不允许read和load、store和write操作之一单独出现
- 不允许一个线程丢弃它最近的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、store操作之前，必须先执行assign和load操作。一个变量在同一个时刻只允许一条线程对其进行lock操作，但lock操作可以被同一条线程重复执行多次，多次执行lock后，只有执行相同次数的unlock操作，变量才会被解锁。
- 如果对一个变量执行lock操作，那将会清空工作内存中此变量的值，在执行引擎使用这个变量前，需要重新执行load或assign操作以初始化变量的值。
- 如果一个变量事先没有被lock操作锁定，那就不允许对它执行unlock操作，也不允许去unlock一个被其他线程锁定的变量。
- 对一个变量执行unlock操作之前，必须先把此变量同步回主内存中（执行store、write操作）。

对于volatile型变量的特殊规则

- Java虚拟机提供的最轻量级的同步机制
- 具有volatile的变量的两个特性
 - 保证变量对所有线程的可见性
 - 一条线程修改了这个变量的值，新值对于其他线程说是可以立即得知的
 - 变量值在线程之间传递时均需要通过主内存来完成
 - 各线程工作内存中不存在一致性问题（因为每次使用前都要刷新），但java的运算操作不是原子操作，导致volatile变量运算并发下一样不安全（即便是编译只有一条字节码，也不意味着该指令是院子操作，因为可能转换出很多机器码指令）
 - 必须符合以下两个规则的运算，否则就得用加锁来保证原子性

- 运算结果并不依赖变量的当前值，或者能够确保只有单一的线程修改变量的值。
- 变量不需要与其他的状态变量共同参与不变约束。

◦ 禁止指令重排序优化

- 普通变量只能保证结果正确不保证顺序一致（“线程内表现为串行的语义”（Within-Thread As-If-Serial Semantics））
- volatile避免了指令重排序
- 原理
 - 一个“lock addl\$0x0, (%esp)”操作，这个操作的作用相当于一个内存屏障（Memory Barrier或Memory Fence，指重排序时不能把后面的指令重排序到内存屏障之前的位置，只有一个处理器访问内存时，并不需要内存屏障；但如果有两个或更多处理器访问同一块内存，且其中有一个在观测另一个，就需要内存屏障来保证一致性了。）
 - lock前缀，它的作用是将本处理器的缓存写入了内存，该写入动作也会引起别的处理器或者别的内核无效化（Invalidate）其缓存，这种操作相当于对缓存中的变量做了一次前面介绍Java内存模式中所说的“store和write”操作。所以通过这样一个空操作，可让前面volatile变量的修改对其他处理器立即可见。
 - lock addl\$0x0, (%esp)指令把修改同步到内存时，意味着所有之前的操作都已经执行完成，这样便形成了“指令重排序无法越过内存屏障”的效果。

◦ 效率

- 性能总体优于锁，但现在锁有很多优化
- 与普通变量读操作没太大区别，写操作略慢，因为要插入内存屏障指令
- 与锁的选择：volatile寓意是否满足使用场景的需求

◦ 内存模型下的特殊规则：假定T表示一个线程，V和W分别表示两个volatile型变量，那么在进行read、load、use、assign、store和write操作时需要满足如下规则

- 线程T对变量V的use动作可以认为是和线程T对变量V的load、read动作相关联的，必须连续且一起出现。 - 使用前必须刷新主内存最新值，保证其他线程能看到修改
- 线程T对变量V的assign动作可以认为是和线程T对变量V的store、write动作相关联的，必须连续且一起出现 - 修改后必须同步到主内存，保证其他线程能看到修改
- 假定动作A是线程T对变量V实施的use或assign动作，假定动作F是和动作A相关联的load或store动作，假定动作P是和动作F相应的对变量V的read或write动作；与此类似，假定动作B是线程T对变量W实施的use或assign动作，假定动作G是和动作B相关联的load或store动作，假定动作Q是和动作G相应的对变量W的read或write动作。如果A先于B，那么P先于Q - volatile修饰的变量不会被指令重排序优化，保证代码的执行顺序与程序的顺序相同。

针对long和double型变量的特殊规则

- “long和double的非原子性协定”（Non-Atomic Treatment of double and long Variables）
 - 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32位的操作来进行 - 允许虚拟机实现自行选择是否要保证64位数据类型的load、store、read和write这四个操作的原子性
 - 一般64位Java虚拟机很难出现读半个值的非原子性访问状况，但32位有这个风险 - hotspot增加了一个实验性的参数-XX: +AlwaysAtomicAccesses
 - double一般不用担心因为处理器有专门处理浮点数据的浮点运算器
 - 一般不用特意把long和double变量专门声明为volatile

原子性、可见性、有序性 - 内存模型就是看并发时如何处理这三个特征

◦ 原子性（Atomicity）

- 六个原子性变量操作 - 保证基本数据类型访问具备原子性 (long和double处理见上面)
- 可实现的关键字:
 - synchronized同步块 - 包含了高层次字节码monitoreenter和monitorexit隐式调用 - 具备原子性
- 可见性 (**Visibility**)
 - 一个线程修改共享变量, 其他线程能立刻获知
 - 实现: 修改后同步回主内存, 其他线程读取前从主内存刷新 - 需要主内存作为传递媒介
 - 可实现的关键字:
 - synchronized - unlock必须同步回主内存
 - final - 一旦初始化完毕, 并且构造器没有把this的引用传递出去, 其他线程可见final值
- 有序性 (**Ordering**)
 - “线程内似表现为串行的语义” (Within-Thread As-If-SerialSemantics) - 本线程内都有序
 - “指令重排序”现象和“工作内存与主内存同步延迟”现象 - 线程观察另一线程都无序
 - 可实现关键词:
 - synchronized - 同一时刻只有一个线程, 同步块只能串行
 - volatile - 禁止指令重排序语义

先行发生原则 (**Happens-Before**)

- 目的
 - 用于判断线程是否存在竞争, 这样可以避免繁琐的volatile或锁等
- 定义
 - 两个操作的偏序关系 - A先发生于B, 意思说A产生的影响会被B观察 (在B操作之前)
- 天然的先行发生关系 - 不需要同步器协助 - 有且只有下面几条
 - 程序次序规则 (**Program Order Rule**) : 在一个线程内, 按照控制流顺序, 书写在前面的操作先行发生于书写在后面的操作。这里说的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。
 - 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。
 - 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()方法是否结束、Thread::isAlive()的返回值等手段检测线程是否已经终止执行。
 - 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。
- 时间先后顺序与先行发生原则之间没有因果关系
 - 时间上先发生不代表先行发生 - 不同线程下
 - 先行发生也不代表时间上先发生 - 指令重排序
 - 一般不要受时间顺序干扰, 判断并发必须以先行发生原则为准

Java与线程

线程的实现

- 线程：java里面进行处理器资源调度的最基本单位 - 可以使进程的资源分配和执行调度分开，各新城共享进程资源，又可以独立调度
- Thread类与大部分java类库API显著差别 - 关键方法都被声明Native - 无法使用平台无关手段实现方法
- 实现线程主要的三个方法
 - 内核线程实现（1:1实现）
 - 内核线程（Kernel-Level Thread, KLT）
 - 直接由操作系统内核（Kernel）支持的线程，可以由内核完成线程切换，内核通过操作调度器（Scheduler）对线程进行调度，负责将现场任务映射到各个处理器上，每个内核线程可以视为内核的一个分身 - 实现多线程处理，该内核称为多线程内核（Multi-Threads Kernel）
 - 程序不直接使用内核线程，使用它的高级接口 - 轻量级进程（Light Weight Process LWP）（通常讲的线程）
 - 每个LWP由一个KLT支持，先有KLT再有LWP - 1:1线程模型
 - 每一个轻量级进程都成为一个独立的调度单元

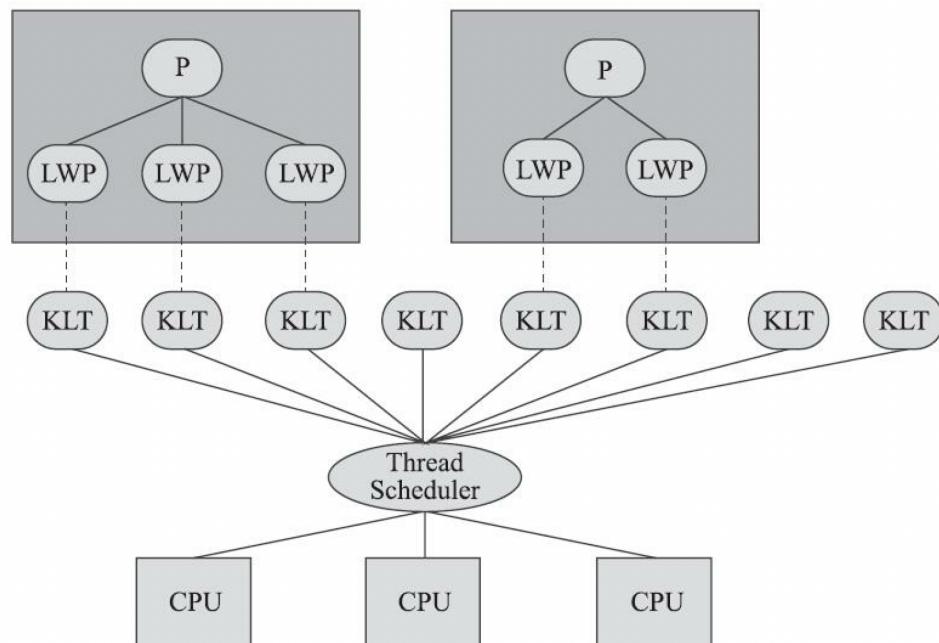


图12-3 轻量级进程与内核线程之间1: 1的关系

- 局限性
 - 基于内核线程实现导致线程操作都需要系统调用 - 调用代价高，需要用户态（User Mode）和内核态（Kernel Mode）来回切换
 - 一个LWP对应一个KLT，需要占用一定的内核资源（比如它的栈空间） - LWP数量有限
- 使用用户线程实现（1: N实现）
 - 用户线程（User Thread UT）
 - 广义：不是内核线程都属于用户线程的一种，因此轻量级进程也属于用户线程，但建立在内核之上受限制而不具备用户线程优点
 - 狹义：完全建立在用户空间的线程库上，用户线程的建立、同步、销毁和调度完全在用户态中完成，不需要内核的帮助，实现得到就不需要切换内核态
- 优点

- 快速低消耗
- 支持规模更大的线程数量

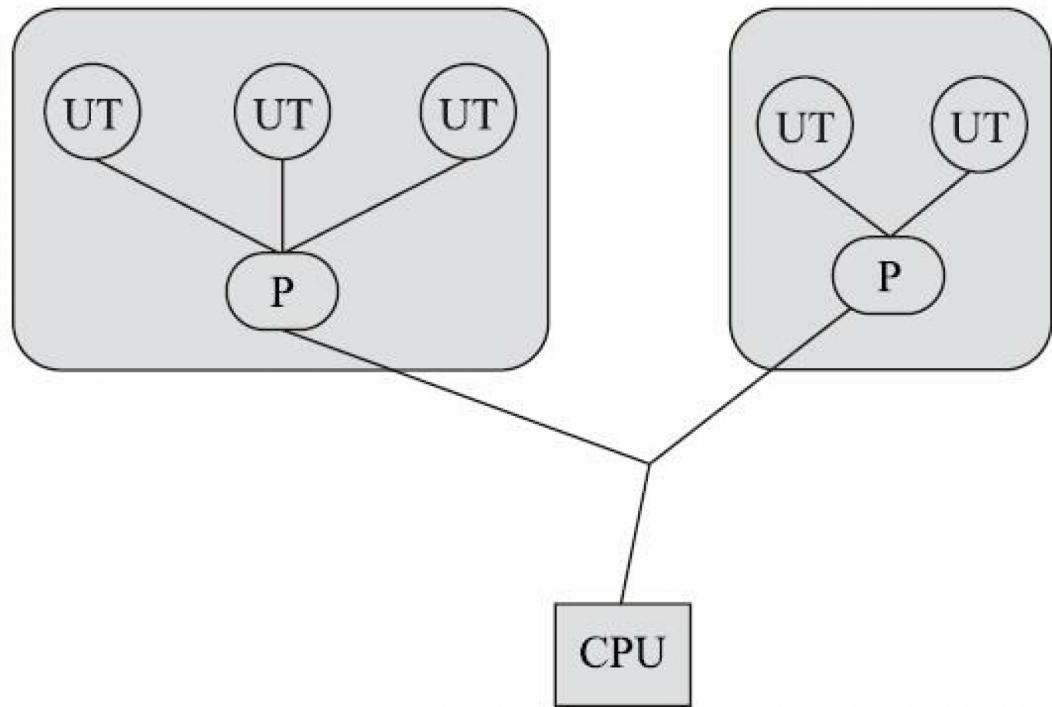


图12-4 进程与用户线程之间1: N的关系

- 局限性
 - 线程操作无法用内核调度（操作系统只把处理器资源分配到进程），必须用户程序自己处理各种问题，难度很大实现复杂，一般应用程序不倾向用用户线程，但近年以高并发为卖点的语言又开始使用用户线程了
- 使用用户线程加轻量级进程混合实现（N: M实现）
 - 用户线程和轻量级进程都有
 - 用户线程还是完全建立在用户空间中，操作廉价，规模大
 - 轻量级进程作为用户线程和内核线程的桥梁，可以使用内核提供的调度功能和处理器映射，用户线程系统调度由轻量级进程完成，解决很多复杂问题（比如进程阻塞）

- 因为用户线程与轻量级进程数量比不定，是N: M关系

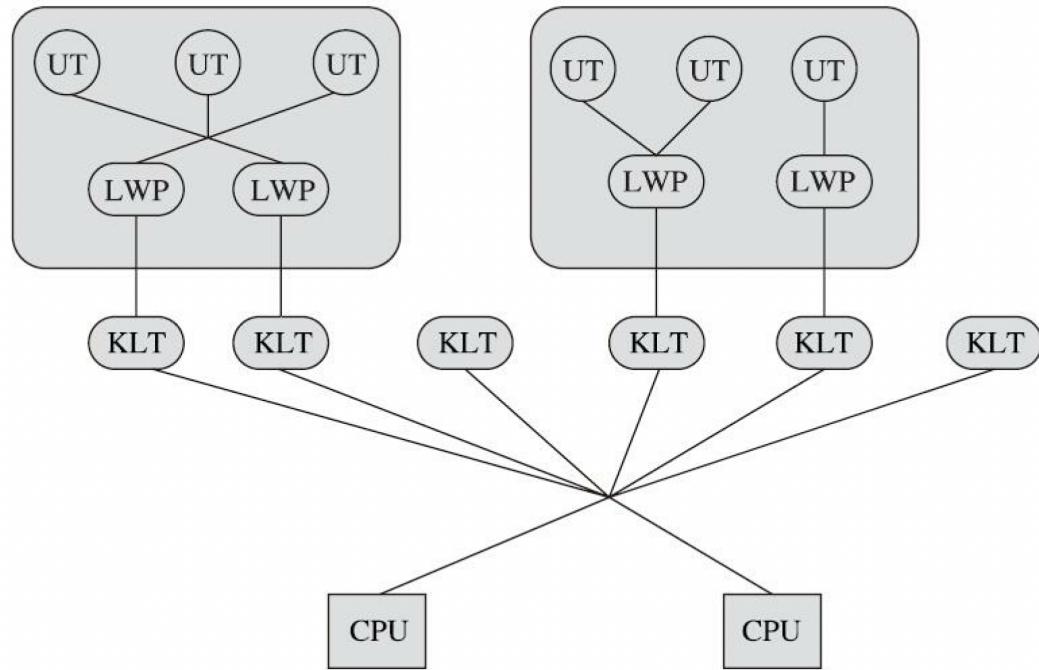


图12-5 用户线程与轻量级进程之间M: N的关系

- 许多Unix系列操作系统，如Solaris、HP-UX都提供了该线程模型实现，因此应用该模型比较容易
 - java线程实现**
 - jdk1.3开始，主流java虚拟机线程模型普遍替换为内核线程模型
 - 操作系统支持怎样的线程模型很大程度影响虚拟机线程怎么映射，因此不同平台很难统一

java线程调度

- 线程调度
 - 系统为线程分配处理器使用权的过程
 - 两种主要方式
 - 协同式线程调度 (Cooperative Threads-Scheduling)**
 - 线程执行时间由线程本身控制，完成工作执行后，主动通知系统切换到另一个线程
 - 优点：实现简单
 - 缺点：线程执行时间不可控，可能会一直阻塞导致程序无法进行
 - 抢占式线程调度 (Preemptive Threads-Scheduling)**
 - 每个线程由系统分配时间，切换不由线程决定 - 系统可控，不会出现一个线程阻塞整个进程
 - java线程调度使用抢占式**
 - 如何建议操作系统分配给某些线程多一点时间
 - 线程优先级**
 - 10个级别(Thread.MIN_PRIORITY - Thread.MAX_PRIORITY)
 - 不稳定，最终的调度还是由操作系统说了算
 - 操作系统的优先级不一定和java线程的优先级一一对应
 - 如果java线程优先级少就可能出现几个线程优先级对应到同一个操作系统优先级的情况
 - 优先级可能会被系统自行改变 - 不要依赖优先级

状态转换

- java定义了6种线程状态，特定条件可以互相转换
 - 新建（New）：创建后尚未启动的线程处于这种状态。
 - 运行（Runnable）：包括操作系统线程状态中的Running和Ready，也就是处于此状态的线程有可能正在执行，也有可能正在等待着操作系统为它分配执行时间。
 - 无限期等待（Waiting）：处于这种状态的线程不会被分配处理器执行时间，它们要等待被其他线程显式唤醒。以下方法会让线程陷入无限期的等待状态：
 - 没有设置Timeout参数的Object::wait()方法；
 - 没有设置Timeout参数的Thread::join()方法；
 - LockSupport::park()方法。
 - 限期等待（Timed Waiting）：处于这种状态的线程也不会被分配处理器执行时间，不过无须等待被其他线程显式唤醒，在一定时间之后它们会由系统自动唤醒。以下方法会让线程进入限期等待状态：
 - Thread::sleep()方法；
 - 设置了Timeout参数的Object::wait()方法；
 - 设置了Timeout参数的Thread::join()方法；
 - LockSupport::parkNanos()方法；
 - LockSupport::parkUntil()方法。
 - 阻塞（Blocked）：线程被阻塞了，“阻塞状态”与“等待状态”的区别是“阻塞状态”在等待着获取到一个排它锁，这个事件将在另外一个线程放弃这个锁的时候发生；而“等待状态”则是在等待一段时间，或者唤醒动作的发生。在程序等待进入同步区域的时候，线程将进入这种状态。
 - 结束（Terminated）：已终止线程的线程状态，线程已经结束执行。

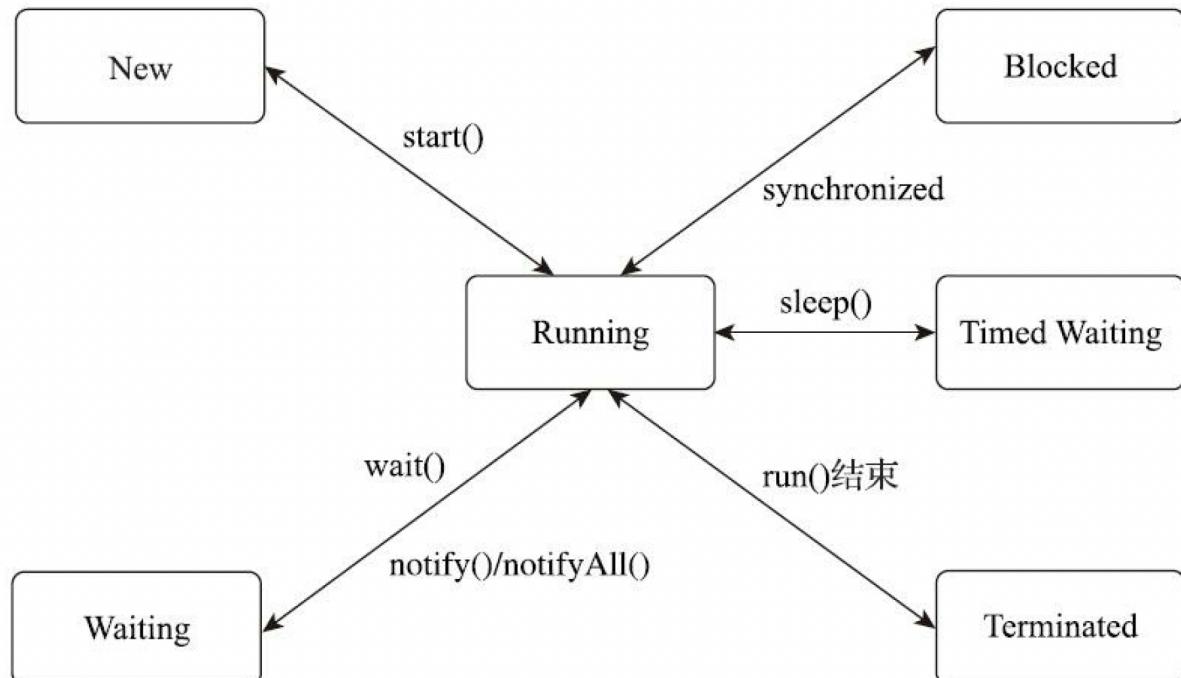


图12-6 线程状态转换关系

Java与协程

内核线程的局限

- 现代B/S系统中一次对外部业务请求的响应，往往需要分布在不同机器上的大量服务共同协作来实现，这种服务细分的架构在减少单个服务复杂度、增加复用性的同时，也不可避免地增加了服务的数量，缩短了留给每个服务的响应时间。这要求每一个服务都必须在极短的时间内完成计算，这样组合多个服务的总耗时才不会太长；也要求每一个服务提供者都要能同时处理数量更庞大的请求，这样才不会出现请求由于某个服务被阻塞而出现等待。
- 目前主流java并发机制用的1:1内核线程模型，但切换、调度成本高，线程数量有限，无法适应上面的架构趋势

协程的复苏

- 为什么内核线程调度切换成本高
 - 调度成本主要是用户态和和心态之间的状态转换，开销来自于响应中断、保护、恢复执行现场的成本
 - 物理硬件的各种存储设备和寄存器是被操作系统内所有线程共享的资源，当中断发生，从线程A切换到线程B去执行之前，操作系统首先要把线程A的上下文数据妥善保管好，然后把寄存器、内存分页等恢复到线程B挂起时候的状态，这样线程B被重新激活后才能仿佛从来没有被挂起过
 - 需要数据在寄存器、缓存中的来回拷贝
 - 用户线程也无法省掉这部分开销，但可以通过编写来缩减开销
- 协程 (**Coroutine**)
 - 用户线程设计为协同式调度，所以叫协程
 - 有栈协程 (**Stackfull Coroutine**) - 线程做了完整的调用栈的保护、恢复工作
 - 无栈协程 (**Stackless Coroutine**) - 本质是一种有限状态机，状态保存在闭包里，自然比有栈协程回复调用栈要轻量，但功能相对有限
 - 优势
 - 比传统内核轻量
 - 局限性
 - 需要在应用层面实现的内容（调用栈、调度器）特别多

java的解决方案

- 有栈协程特例 - 纤程 (**Fiber**) - 重新提供对用户线程的支持
- Java虚拟机提供了两个并发编程模型，可以同时使用
- 在新并发模型下，一段使用纤程并发的代码会被分为两部分——执行过程 (**Continuation**) 和调度器 (**Scheduler**)。执行过程主要用于维护执行现场，保护、恢复上下文状态，而调度器则负责编排所有要执行的代码的顺序。将调度程序与执行过程分离的好处是，用户可以选择自行控制其中的一个或者多个，而且Java中现有的调度器也可以被直接重用

第十三章 线程安全与锁优化

线程安全定义

- 当多个线程同时访问一个对象时，如果不用考虑这些线程在运行时环境下的调度和交替执行，也不需要进行额外的同步，或者在调用方进行任何其他的协调操作，调用这个对象的行为都可以获得正确的结果，被称为这个对象是线程安全的
- 代码本身封装了所有必要的正确性保障手段（如互斥同步等），另调用者无需关心多线程下的调用问题，更无需自己实现任何措施来保证多线程环境下的正确调用

java语言中的线程安全

- 按照线程安全的“安全程度”由强至弱来排序，我们可以将java语言中各种操作共享的数据分为五类
 - 不可变 (Immutable)
 - 对象一定是线程安全的
 - 一个不可变对象正确构建处理（无this引用逃逸），其外部可见状态永远不会改变，永远都不会看到它在多个线程之中处于不一致的状态
 - 典型：final
 - 基本数据类型：加入final不可变
 - 对象：暂时没有提供方法支持，需要对象自行保证其行为不会对其状态产生任何影响
 - 实现方法之一：所有状态变量声明为final
 - 常见不可变类型：String, 枚举类型, java.lang.Number部分子类（Long和Double等数值包装类型、BigInteger、BigDecimal等大数据类型，但是Number子类下的原子类AtomicInteger和AtomicLong则是可变的）
 - 绝对线程安全
 - 要完全满足之前给的定义，相当严格
 - java api标注的自己是线程安全的类大多不是绝对线程安全
 - 举例
 - Java.util.Vector所有方法被synchronized修饰，保证了原子性可见性有序性，但多线程环境下如果不在方法调用端做额外的同步措施，使用它仍然不安全。如果要保证绝对的线程安全，必须在它内部维护互斥一致性快照访问，每次对其中元素进行改动都要产生新的快照，这也付出的时间空间成本太大
 - 相对线程安全
 - 通常意义上说的线程安全，保证对对象单次操作时线程安全的
 - 特定顺序的连续调用需要调用端使用额外同步手段来保证调用的正确性
 - 常见声称线程安全的类
 - Vector、HashTable、Collections的synchronizedCollection () 方法包装的集合等
 - 线程兼容
 - 对象本身并不是线程安全的，但是可以通过在调用端正确地使用同步手段来保证对象在并发环境中可以安全地使用
 - 我们平常说一个类不是线程安全的，通常就是指这种情况
 - Java类库API中大部分的类都是线程兼容的
 - 常见
 - 前面的Vector和HashTable相对应的集合类ArrayList和HashMap等。
 - 线程对立
 - 线程对立是指不管调用端是否采取了同步措施，都无法在多线程环境中并发使用代码
 - Java语言天生就支持多线程的特性，线程对立这种排斥多线程的代码是很少出现的，而且通常都是有害的，应当尽量避免
 - 举例
 - Thread类的suspend()和resume()方法
 - 如果有两个线程同时持有一个线程对象，一个尝试去中断线程，一个尝试去恢复线程，在并发进行的情况下，无论调用时是否进行了同步，目标线程都存在死锁风险
 - System.setIn()、System.setOut()和System.runFinalizersOnExit()等

线程安全的实现方法

- 互斥同步 (Mutual Exclusion & Synchronization) (阻塞同步 - blocking synchronization)
 - 最常见最主要的并发正确性保障手段
 - 定义：并发时，保证共享数据同时刻只有一条（或者是一些，当使用信号量的时候）线程使用
 - 互斥是实现同步的手段，**临界区 (CriticalSection)，互斥量 (Mutex) 和信号量 (Semaphore) **都是常见的互斥实现方式
 - 互斥量主要用于保护临界区，确保共享资源不会被多个线程同时访问；临界区是代码段的概念，需要通过互斥量等机制来实现互斥；信号量是一种更通用的同步机制，不仅可以用于互斥，还可以用于同步，并且可以控制多个线程对资源的访问
 - 最基本的互斥同步手段 - **synchronized**关键字
 - 块结构 (Block Structured) 的同步语法
 - 经过javac编译后，同步块前后形成**monitorenter**和**monitorexit**两个字节码指令，俩指令都需要reference参数指明锁定解锁对象。
 - 明确指定对象了就用那个对象的reference，如果没有指定就看它修饰的方法类型，实例方法则是代码里的对象实例，类方法则是对应的Class对象
 - 工作过程
 - 执行**monitorenter**来尝试获取锁
 - 没锁定或者当前线程持有了锁 - 锁计数器++
 - 执行**monitorexit** - 锁计数器--
 - 获取锁失败 - 阻塞，直到锁被释放
 - 同步块对同一个线程可以重入，反复进入不会锁死自己
 - 持有锁线程执行完毕释放锁之前，其他线程无条件阻塞
 - 由执行成本来看，属于重量级 (Heavy Weight) 操作
 - java使用的是映射到操作系统的原生内核线程上，转换用户态和核心态成本大 - 虚拟机在进一步优化，比如阻塞前计入自旋等待过程来避免频繁切入核心态
 - Lock接口
 - java.util.concurrent包（简称J.U.C）中的**java.util.concurrent.locks.Lock**接口
 - 能够以非块结构 (Non-Block Structured) 来实现互斥同步，在类库层面去实现同步
 - 重入锁(ReentrantLock)
 - Lock接口最常见的一种实现
 - 可以重入
 - 与synchronized很相似，但在基础上增加了一些高级功能，主要三项
 - 等待可中断
 - 阻塞的线程可以选择放弃等待先去处理其他事情
 - 可实现公平锁
 - 多个线程等待同一个锁时，必须按照申请锁的时间顺序来获得锁，非公平锁就无法保证这一点（比如synchronized）
 - ReentrantLock默认非公平，但可以通过构造函数来设置公平锁
 - 公平锁会影响性能，明显影响吞吐量
 - 锁可以绑定多个条件
 - ReentrantLock可以同时绑定多个Condition对象，只要多次调用**newCondition()**方法即可
 - synchronized可以使用**wait()**以及**notify ()** 或者**notifyAll ()** 来配合实现一个隐含的条件
 - Lock和synchronized对比
 - jdk6之后加入了大量针对synchronized的优化，两个性能现在基本持平了
 - 若两个都满足时还是建议使用**synchronized**

- java语法层面同步，足够清晰，足够简单
 - lock必须保证finally块释放锁，否则一旦在同步保护的代码块中抛出异常，则有可能永远不会释放该锁，必须由程序员自主保证
 - synchronized虚拟机能保证异常的时候锁能自动释放
 - java虚拟机更容易优化synchronized
 - Java虚拟机可以在线程和对象的元数据记录synchronized的锁的相关信息
 - 另一种常见实现 - 重入读写锁 (ReentrantReadWriteLock)
- 非阻塞同步 (**Non-Blocking Synchronization**) - 无锁编程 (**Lock-Free**)
 - 互斥同步缺点
 - 悲观并发策略，认为不做措施就会有问题，所以数据竞争与否都会进行加锁 - 开销大（用户态核心态转换、维护锁计数器、检查是否有阻塞的线程需要被唤醒等）
 - 基于冲突检测的乐观并发策略
 - 原理
 - 不管有没有风险，先去操作
 - 无竞争就操作成功，有竞争再进行补偿措施
 - 最常见的补偿措施 - 不断重试，直到没有竞争为止
 - 必须靠硬件指令集的发展来实现该策略 - 因为必须保证操作和冲突检测两个步骤的原子性，保证某些需要多次操作行为可以只通过一条处理器指令就能完成
 - 常见指令
 - 测试并设置 (Test-and-Set) ;
 - 获取并增加 (Fetch-and-Increment) ;
 - 交换 (Swap) ;
 - 比较并交换 (**Compare-and-Swap**, 下文称CAS) ;
 - 加载链接/条件储存 (Load-Linked/Store-Conditional, 下文称LL/SC) 。
 - java主要靠cas操作
 - cas指令
 - 工作流程
 - 三个操作数：内存位置（理解为内存地址，V）、旧预期值（A）、准备设置的新值（B）
 - 当且仅当V符合A时，处理器才会用B更新V的值，否则不执行更新
 - 无论是否更新V，都会返回V的旧值
 - 以上操作属于原子操作
 - jdk5之后类库才有cas操作 - **sun.misc.Unsafe类的compareAndSwapInt()和compareAndSwapLong () **等几个方法包提供
 - HotSpot虚拟机在内部对这些方法做了特殊处理，即时编译出来的结果就是一条平台相关的处理器cas指令，可以认为是无条件内联
 - jdk9之后类库才在VarHandle类里开放了面向用户程序使用的cas操作
 - 举例
 - **AtomicInteger** - 里面的incrementAndGet方法具有原子性，在无限循环中不断尝试将一个比当前值大一的新值赋给自己，失败了就说明旧值改变了，再次循环操作直到成功
 - 缺陷
 - ABA问题：原值A，后来改为B，再后来又改回A，cas误以为它从来没有改变过
 - 解决方案：JUC提供了一个AtomicStampedReference原子引用类，可以通过控制变量值的版本来保证cas正确性 - 但其实很鸡肋，因为aba问题一般不影响并发，一定要解决不如用传统互斥同步

- 无同步方案
 - 有些代码天生线程安全
 - 两类
 - 可重入代码（Reentrant Code） - 纯代码（Pure code）
 - 代码执行任何时刻都可中断，转去执行另外一段代码（包括递归调用它本身），控制权回来后源代码执行不会出错，结果也不错
 - 多线程上下文语境里，可重入代码是线程安全代码的一个真子集 - 所有可重入代码都是线程安全的，但并不是所有线程安全代码都是可重入的
 - 不依赖全局变量、存储在堆上的数据和公用的系统资源，用到的状态量都由参数中传入
 - 不调用非可重入方法
 - 判断方法：返回结果可预测，每次输入相同数据都能返回相同结果
 - 线程本地存储（Thread local storage）
 - 如果一段代码的数据必须与其它代码共享，看能否把这段代码保证在同一线程执行，可以则保证了数据可见范围限制在同一线程内
 - 应用
 - 大部分使用消费队列的架构模式（比如生产者-消费者模式），把消费过程限制在一个线程中消费完
 - 最重要的实例就是**经典Web交互模型中“一个请求对应一个服务器线程”（Thread-per-Request）**的处理方式
 - 实现
 - 可以通过`java.lang.ThreadLocal`类来实现线程本地存储的功能。每一个线程的Thread对象中都有一个`ThreadLocalMap`对象，这个对象存储了一组以`ThreadLocal.threadLocalHashCode`为键，以本地线程变量为值的K-V值对，`ThreadLocal`对象就是当前线程的`ThreadLocalMap`的访问入口，每一个`ThreadLocal`对象都包含了一个独一无二的`threadLocalHashCode`值，使用这个值就可以在线程K-V值对中找回对应的本地线程变量。

锁优化

自旋锁与自适应自旋

- 出现原因
 - 为了优化阻塞对性能的影响
- 定义
 - 多个线程并行，让后面请求锁的等一下，但不放弃处理器执行时间，看看持有锁的线程是否很快释放锁，一般让线程执行一个忙循环（自旋）
- jdk6默认开启
- 性能
 - 如果所占用时间很短，自旋等待效果非常好，但占的时间太长就会白白消耗处理器资源 - 必须有一定的限度，如果自旋超过了一定次数就应当正常挂起 - 默认十次
- 自适应自旋
 - 自旋时间不固定，而是由前一次在同一锁上的自选时间及锁的拥有者的状态来决定
 - 自旋成功，进而允许自旋等待更久；自旋很少成功获得锁，则以后可能直接省略自旋过程，避免浪费处理器资源

锁消除

- 定义
 - 虚拟机即时编译器在运行时，对一些代码要求同步，但是对被检测到不可能存在共享数据竞争的锁进行消除
- 主要判定依据
 - 来源于逃逸分析的数据支持 - 如果判断到一段代码中，在堆上的所有数据都不会逃逸出去被其他线程访问到，那就可以把它们当作栈上数据对待，认为它们是线程私有的，同步加锁自然就无须再进行。
- 有许多同步措施并不是程序员自己加入的，同步的代码在Java程序中出现的频繁程度也许超过了大部分人的想象。

锁粗化

- 如果一系列的连续操作都对同一个对象反复加锁和解锁，甚至加锁操作是出现在循环体之中的，那即使没有线程竞争，频繁地进行互斥同步操作也会导致不必要的性能损耗。
- 如果虚拟机探测到有这样一串零碎的操作都对同一个对象加锁，将会把加锁同步的范围扩展（粗化）到整个操作序列的外部

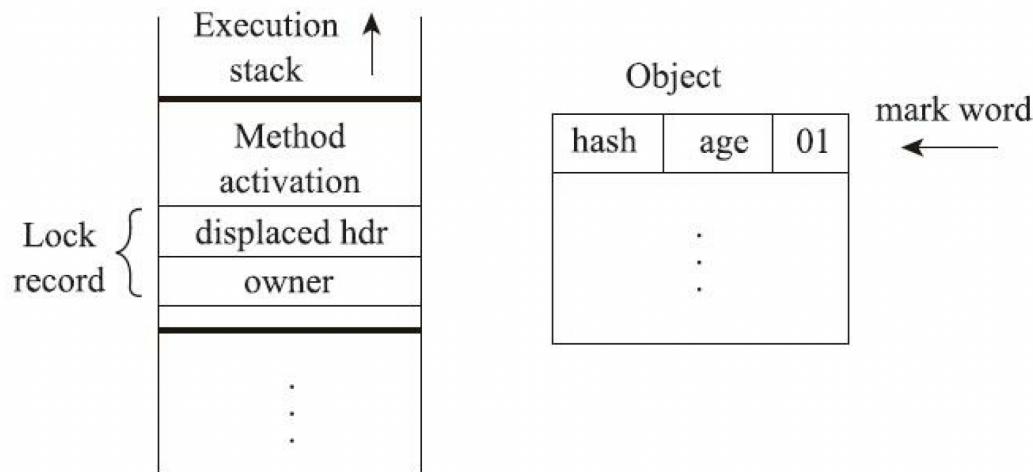
轻量级锁

- 出现原因
 - 在没有多线程竞争的前提下，减少传统的重量级锁使用操作系统互斥量产生的性能消耗
- 原理
 - **mark word**
 - Hotspot虚拟机对象头(object header)分两部分 = 运行时数据 + 类型数据 + (数组的数组长度)
 - 运行时数据那部分，包含hashcode、gc分代年龄等，在32位或者64位虚拟机中分别会占用32个或64个bit - **Mark Word** - 非固定动态数据结构，极小空间存更多信息还会复用空间
 - mark word其中两个bit用来存储锁标志位，还有一个bit的偏向标记

表13-1 HotSpot虚拟机对象头Mark Word

锁状态	32bit				
	25bit		4bit	1bit	2bit
	23bit	2bit		偏向模式	标志位
未锁定	对象哈希码		分代年龄	0	01
轻量级锁定	指向调用栈中锁记录的指针				
重量级锁定 (锁膨胀)	指向重量级锁的指针				
GC 标记	空				11
可偏向	线程 ID	Epoch	分代年龄	1	01

- 工作过程

图13-3 轻量级锁CAS操作之前堆栈与对象的状态^[1]

- 在代码即将进入同步块的时候，如果此同步对象没有被锁定（锁标志位为“01”状态），虚拟机首先将在当前线程的栈帧中建立一个名为锁记录（Lock Record）的空间，用于存储锁对象目前的Mark Word的拷贝（官方为这份拷贝加了一个Displaced前缀，即Displaced Mark Word）
- 然后，虚拟机将使用**CAS操作**尝试把对象的Mark Word更新为指向Lock Record的指针。如果这个更新动作成功了，即代表该线程拥有了这个对象的锁，并且对象Mark Word的锁标志位（Mark Word的最后两个比特）将转变为“00”，表示此对象处于轻量级锁定状态。
- 如果这个更新操作失败了，那就意味着至少存在一条线程与当前线程竞争获取该对象的锁。虚拟机首先会检查对象的Mark Word是否指向当前线程的栈帧，如果是，说明当前线程已经拥有了这个对象的锁，那直接进入同步块继续执行就可以了；否则就说明这个锁对象已经被其他线程抢占了
- 如果出现两条以上的线程争用同一个锁的情况，那轻量级锁就不再有效，必须要膨胀为重量级锁，锁标志的状态值变为“10”，此时Mark Word中存储的就是指向重量级锁（互斥量）的指针

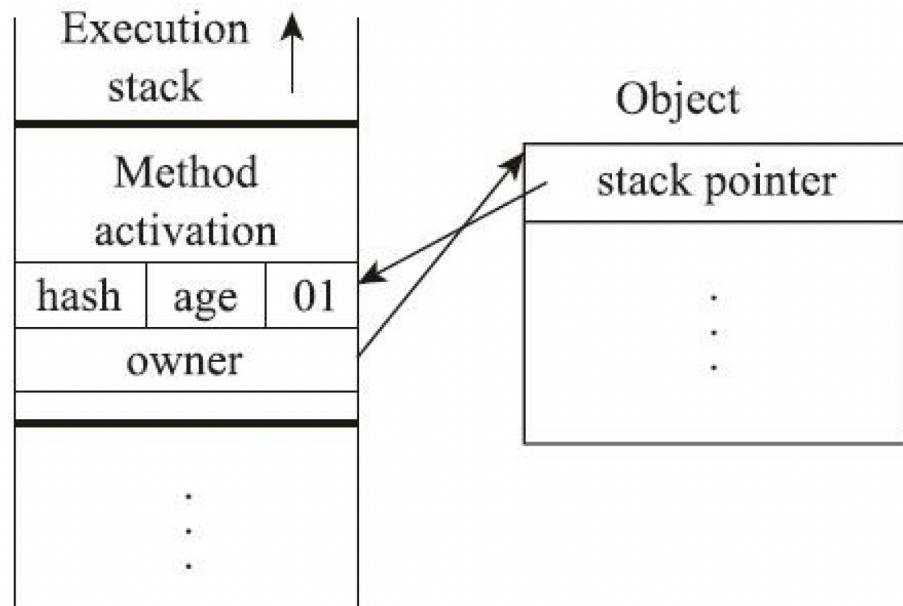


图13-4 轻量级锁CAS操作之后堆栈与对象的状态

- 解锁过程

- Mark Word仍然指向线程的锁记录，那就用**CAS操作**把对象当前的**Mark Word**和线程中复制的**Displaced Mark Word**替换回来。假如能够成功替换，那整个同步过程就顺利完成了；如果替换失败，则说明有其他线程尝试过获取该锁，就要在释放锁的同时，唤醒被挂起的线程。

- 同步性能
 - 没有竞争时，通过cas操作可以成功避免使用互斥量的开销
 - 存在竞争，除了互斥量开销外，额外发生了cas操作的开销，反而比重量级锁更慢

偏向锁

- 目的
 - 目的是消除数据在无竞争情况下的同步原语，进一步提高程序的运行性能，在无竞争的情况下把整个同步都消除掉，连**CAS操作都不去做了**。
- 定义
 - 这个锁会偏向于第一个获得它的线程，如果在接下来的执行过程中，该锁一直没有被其他的线程获取，则持有偏向锁的线程将永远不需要再进行同步。
- 原理
 - 假设当前虚拟机启用了偏向锁（启用参数-XX: +UseBiased Locking，这是自JDK 6起HotSpot虚拟机的默认值），那么当锁对象第一次被线程获取的时候，虚拟机将会把对象头中的标志位设置为“01”、把偏向模式设置为“1”，表示进入偏向模式
 - 同时使用**CAS操作**把获取到这个锁的线程的ID记录在对象的**Mark Word**之中。如果CAS操作成功，持有偏向锁的线程以后每次进入这个锁相关的同步块时，虚拟机都可以不再进行任何同步操作（例如加锁、解锁及对Mark Word的更新操作等）。
 - 一旦出现另外一个线程去尝试获取这个锁的情况，偏向模式就马上宣告结束。根据锁对象目前是否处于被锁定的状态决定是否撤销偏向（偏向模式设置为“0”），撤销后标志位恢复到未锁定（标志位为“01”）或轻量级锁定（标志位为“00”）的状态

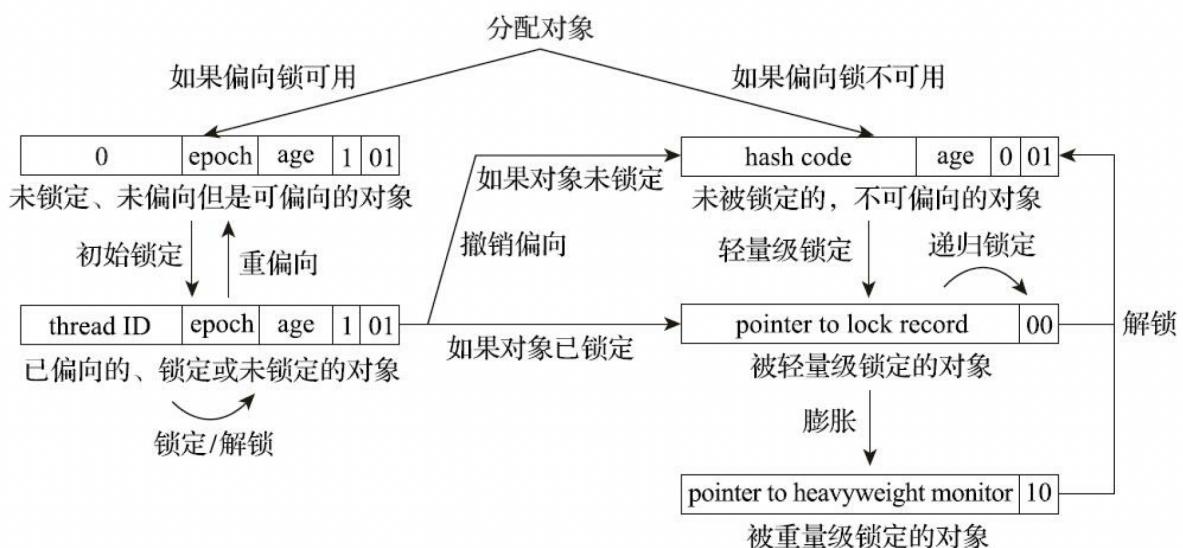


图13-5 偏向锁、轻量级锁的状态转化及对象Mark Word的关系

- 用来存储线程id，那hashcode怎么办
 - `Object::hashCode()`方法，返回的是对象的一致性哈希码（Identity Hash Code），这个值是能强制保证不变的，它通过在对象头中存储计算结果来保证第一次计算之后，再次调用该方法取到的哈希码值永远不会再发生改变。
 - 因此，当一个对象已经计算过一致性哈希码后，它就再也无法进入偏向锁状态了

- 而当一个对象当前正处于偏向锁状态，又收到需要计算其一致性哈希码请求时，它的偏向状态会被立即撤销，并且锁会膨胀为重量级锁
- 在重量级锁的实现中，对象头指向了重量级锁的位置，代表重量级锁的ObjectMonitor类里有字段可以记录非加锁状态（标志位为“01”）下的Mark Word，其中自然可以存储原来的哈希码。
- 注意：如果hashCode方法重写了，就没有这个情况了

- 性能

- 可以提高带有同步但无竞争的程序性能
- 但如果程序中大多数的锁都总是被多个不同的线程访问，那偏向模式就是多余的
- 在具体问题具体分析的前提下，有时候使用参数-XX: -UseBiasedLocking来禁止偏向锁优化反而可以提升性能。