《深入理解 Java 虚拟机》笔记

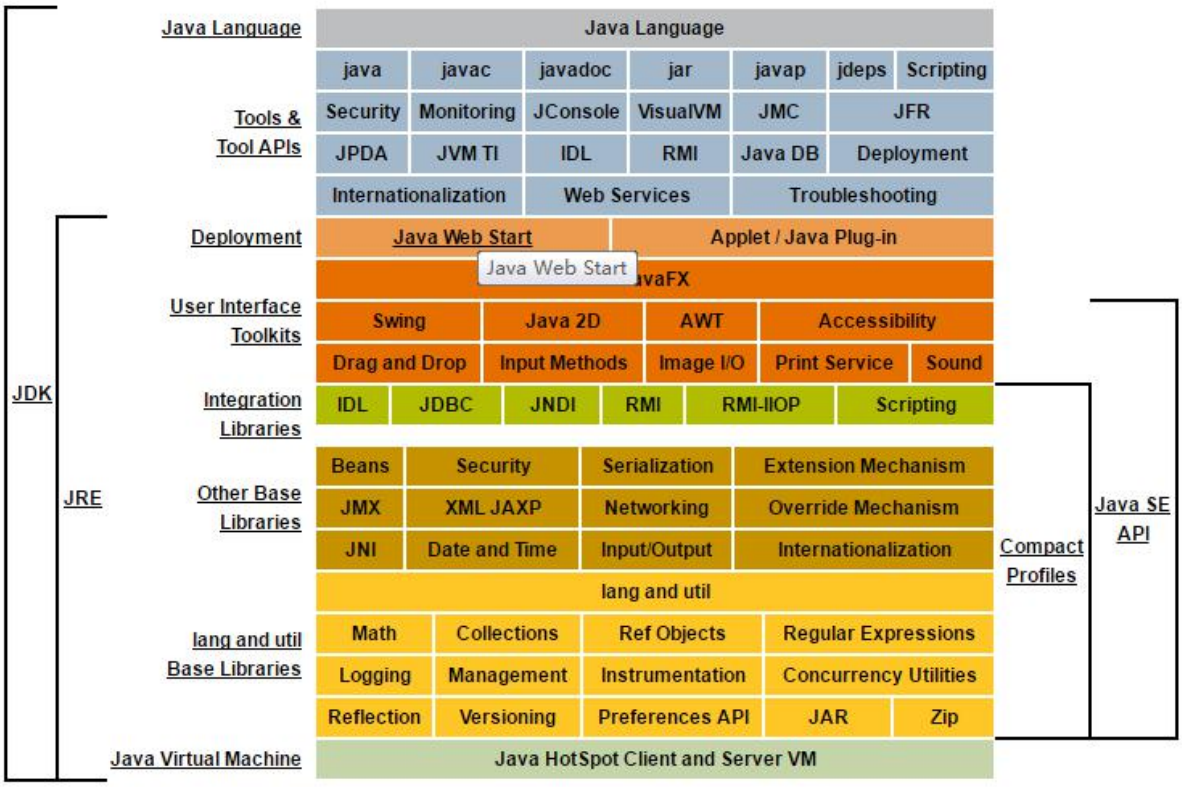
-- Dawei Min

1 走近 Java

1.1 JDK 与 JRE

把 **Java 程序设计语言、Java 虚拟机、Java API 类库**称为 **JDK**（Java Development Kit）, JDK 是用于支持 Java 程序开发的最小环境。

把 **Java API 类库中的 Java SE API 子集和 Java 虚拟机**称为 **JRE**（Java Runtime Environment）,JRE 是支持 Java 程序运行的标准环境。



图片来自 http://docs.oracle.com/javase/8/docs/

1.2 Java 虚拟机

Sun JDK 和 OpenJDK 中所带的虚拟机是 HotSpot VM。

可以参考的文档：http://docs.oracle.com/javase/8/docs/technotes/guides/vm/

1.3 Java 技术展望

模块化（提到 OSGI）、混合语言、多核并行（希望利用 GPU、APU 等）。

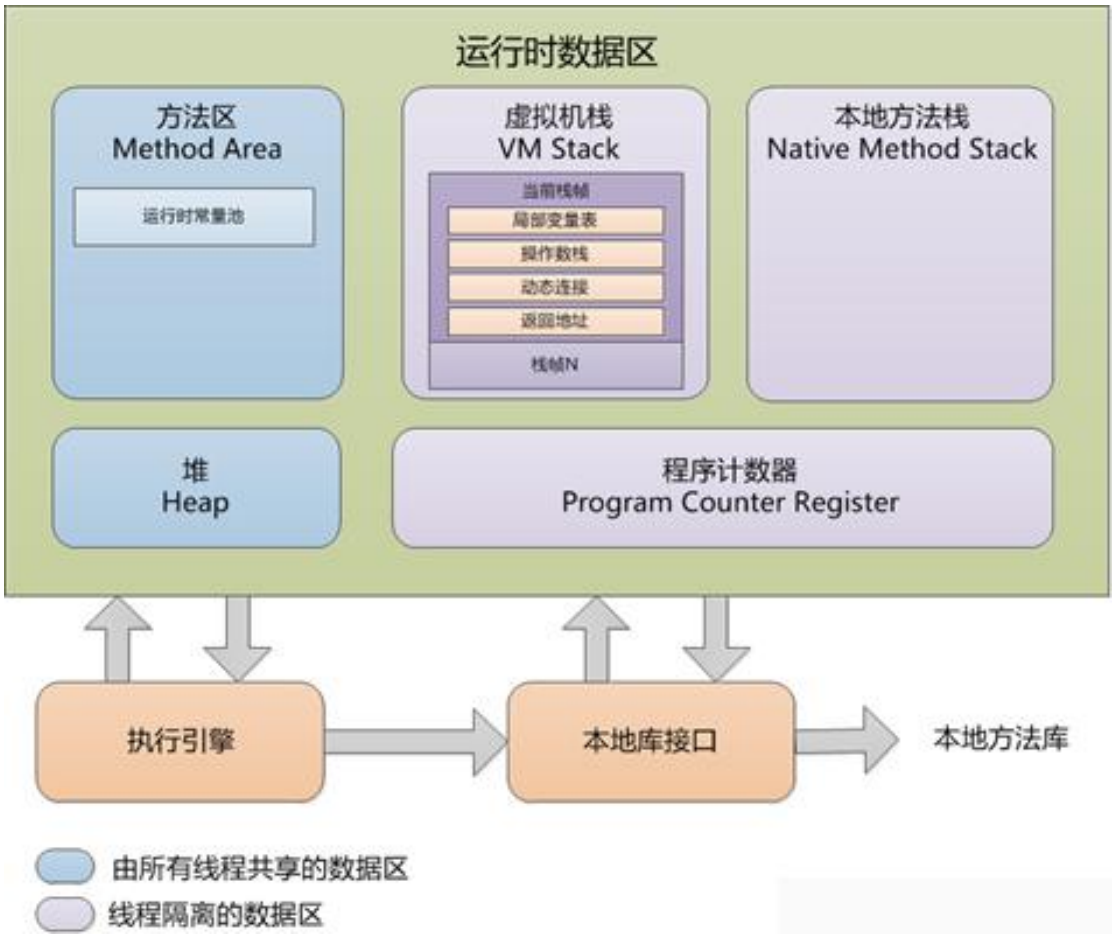
2 Java 内存区域与内存溢出异常

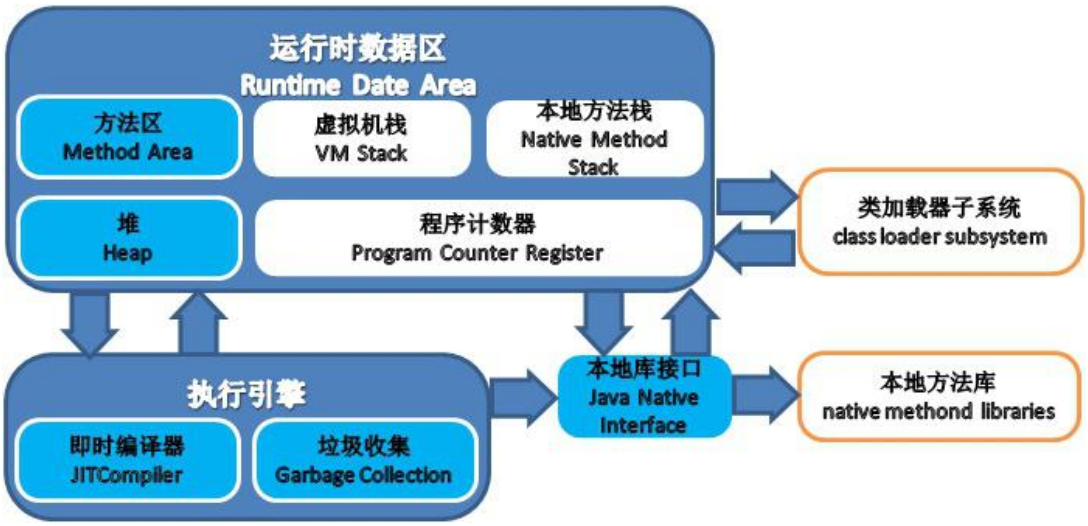
2.1 概述

Java 虚拟机内存自动管理利弊：

- 优点：不需要为每一个 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。
- 缺点：出现内存泄漏和溢出问题时，如果不了解虚拟机怎样使用内存，排查会比较艰难。

2.2 运行时数据区域





2.2.1 程序计数器

程序计数器（Program Counter Register）可以看成**当前线程**所执行的字节码的行号指示器。各个线程的程序计数器是相互独立的互不影响。

如果执行的是 Java 方法，计数器记录字节码指令地址；如果执行的是 Native 方法，计数器值为空（Undefined）。

程序计数器区域虚拟机规范中是唯一一个没有规定任何 OutOfMemoryError 情况的区域。

2.2.2 Java 虚拟机栈

Java 虚拟机栈（Java Virtual Machine Stacks）是线程私有的，描述的是 Java 方法执行的内存模型：每个方法在执行同时会创建一个栈帧（Stack Frame）用于存储局部变量表、操作数栈、动态链接、方法出口等信息。

如果**线程请求的栈深度大于虚拟机所允许的深度**，将抛出 StackOverflowError 异常；如果**虚拟机栈可以动态扩展（大部分虚拟都可以）并且扩展时无法申请到足够内存**，就会抛出 OutOfMemoryError 异常。

2.2.3 Java 堆

Java 堆（Java Heap）是被所有线程共享的一块内存区域，也是供所有类实例和数组对象分配内存的区域。

Java 堆的容量可以是固定大小的，也可以随着程序执行的需求动态扩展，并在不需要过多空间时自动收缩。Java 堆所使用的内存不需要保证是连续的。

如果实际所需的堆超过了自动内存管理系统能提供的最大容量，那 Java 虚拟

机将会抛出一个 OutOfMemoryError 异常。

```
Java 控制堆和非堆参数例子：
-vmargs -Xms128M -Xmx512M -XX:PermSize=64M -XX:MaxPermSize=128M
-vmargs 说明后面是 VM 的参数，所以后面的其实都是 JVM 的参数了
-Xms128m JVM 初始分配的堆内存
-Xmx512m JVM 最大允许分配的堆内存，按需分配
-XX:PermSize=64M JVM 初始分配的非堆内存
-XX:MaxPermSize=128M JVM 最大允许分配的非堆内存，按需分配
```

2.2.4 方法区

方法区（Method Area）与 Java 堆一样，是各个线程共享的内存区域，它用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译后的代码等数据。方法区别名叫做 **Non-Heap（非堆）**。

在 HotSpot 虚拟机上，很多人把方法区成为“永久代”（Permanent Generation）是因为 HotSpot 把 GC 分代收集扩展至方法区，用永久代实现方法区。这样做的好处是节省工作量，坏处是受到-XX:MaxPermSize 的限制，更容易遇到内存溢出问题。

运行时常量池（Runtime Constant Pool）是方法区的一部分。

如果方法区的内存空间不能满足内存分配请求，那 Java 虚拟机将抛出一个 OutOfMemoryError 异常。

2.2.5 本地方法栈

Java 虚拟机实现可能会使用到传统的栈（通常称之为“C Stacks”）来支持 native 方法（指使用 Java 以外的其他语言编写的方法）的执行，这个栈就是本地方法栈（Native Method Stack）。当 Java 虚拟机使用其他语言（例如 C 语言）来实现指令集解释器时，也会使用到本地方法栈。

如果线程请求分配的栈容量超过本地方法栈允许的最大容量时，Java 虚拟机将会抛出一个 StackOverflowError 异常。

如果本地方法栈可以动态扩展，并且扩展的动作已经尝试过，但是目前无法申请到足够的内存去完成扩展，或者在建立新的线程时没有足够的内存去创建对应的本地方法栈，那 Java 虚拟机将会抛出一个 OutOfMemoryError 异常。

该书提到直接内存（Direct Memory）,JDK1.4 中加入的 NIO（New Input/Output）引入了一种基于通道（Channel）与缓冲区（Buffer）的 I/O 方式，可以使用 Native

函数直接分配堆外内存，然后通过 Java 堆中 DirectByteBuffer 对象作为这块内存的引用进行操作。

2.3 HotSpot 虚拟机对象探秘

2.3.1 对象的创建

虚拟机遇到一条 new 指令时：

（1）检查这个指令的参数是否能在常量池中定位到一个类符号的引用。

（2）检查这个符号引用代表的类是否已被加载、解析和初始化过。如果没有必须先执行相应的类加载过程。

（3）虚拟机为新生对象分配内存。分配算法取决于垃圾收集器是否带有压缩整理功能。如果带有压缩整理功能（如 Serial、ParNew 等），Java 堆规整，采用“指针碰撞”（Bump the Pointer）,否则（如 CMS）Java 堆不规整,采用“空闲链表”Free List。

（4）将分配到的内存空间都初始化为零值（不包括对象头）。

（5）对对象头进行必要的设置（那个类的实例、如何找到类的元数据信息、对象的哈希吗、对象的 GC 分代年龄等）。

（6）调用对象的<init>方法。

2.3.2 对象的内存布局

在 HotSpot 虚拟机中，对象在内存中存储的布局可以分为 3 块区域：对象头（Header）、实例数据（Instance Data）和对齐填充（Padding）。

对象头：存储对象自身的运行时数据：Mark Word（在 32bit 和 64bit 虚拟机上长度分别为 32bit 和 64bit），包含如下信息：

（1）对象 HashCode

（2）对象 GC 分代年龄

（3）锁状态标志（轻量级锁、重量级锁）

（4）线程持有的锁（轻量级锁、重量级锁）

（5）偏向锁相关：偏向锁、自旋锁、轻量级锁以及其他的一些锁优化策略是 JDK1.加入的，这些优化使得 Synchronized 的性能与 ReentrantLock 的性能持平，在 Synchronized 可以满足要求的情况下，优先使用 Synchronized，除非是使用一些 ReentrantLock 独有的功能，例如指定时间等待等。

（6）类型指针：对象指向类元数据的指针（32bit-->32bit，64bit-->64bit(未开启压缩指针)，32bit(开启压缩指针)）。JVM 通过这个指针来确定这个对象是哪个类的实例（根据对象确定其 Class 的指针）

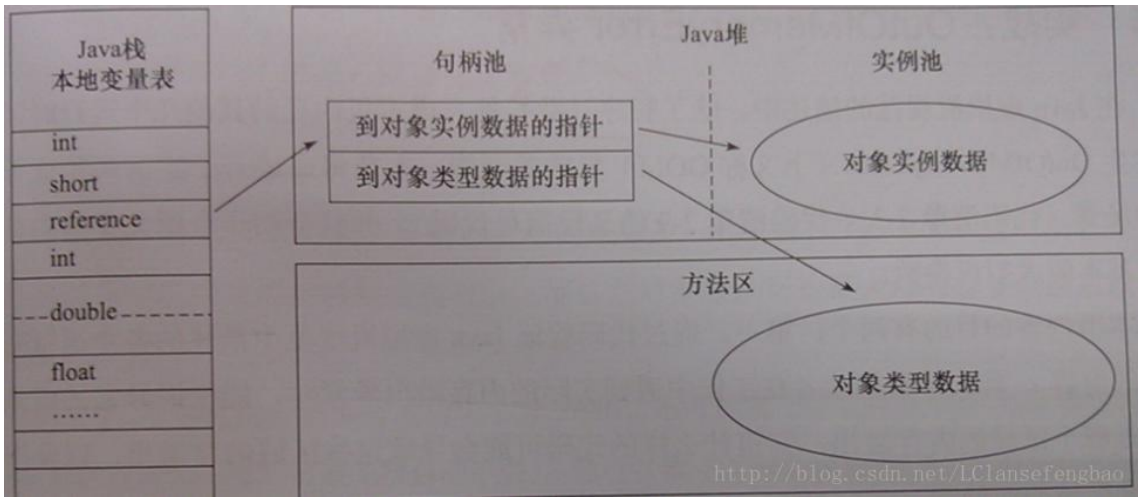
实例数据：对象真正存储的有效信息。

对齐填充：HotSpot VM 要求对象的大小必须是 8 的整数倍，若不是，需要补位对齐。

2.3.3 对象的访问定位

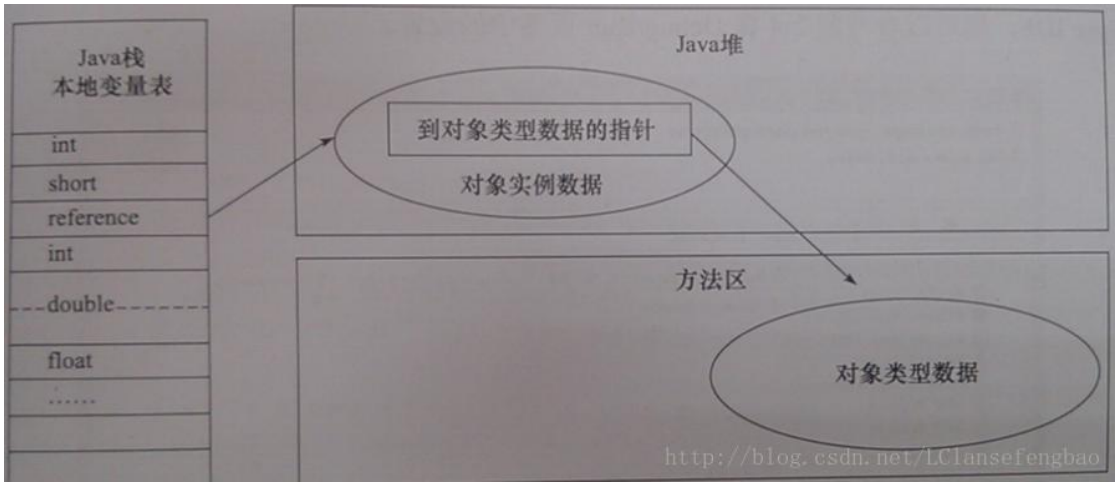
对象的访问方式有**使用句柄**和**直接指针**两种。

如果使用句柄访问的话，那么 Java 堆中将会划分出一块区域作为句柄池，reference 中存的就是句柄地址。句柄中包含了实例数据和类型数据各自的地址。



如果使用直接指针访问的的话，reference 中存的直接就是对象地址，对象中

放置类型数据的相关信息。



使用句柄的好处：对象被移动时只需改变句柄中地址，reference 本身不需要改变。使用直接指针访问的好处：速度更快，节省了一次指针定位的时间开销。

2.4 实战：OutOfMemoryError 异常

除了程序计数器外，虚拟机内存的其他几个运行时区域都有发生 OutOfMemoryError（简称 OOM）异常的可能。

2.4.1 Java 堆溢出

产生例子：不断创建对象，并保证 GC Roots 到对象之间有可达路径。

解决思路：利用工具对 Dump 出来的堆转储快照进行分析，重点确认内存中的对象是否必要，也就是要分清是出现内存泄漏（Memory Leak）还是内存溢出（Memory Overflow）。如果是内存泄漏，就使用工具查看泄漏对象到 GC Roots 的引用链，确定泄漏代码位置；如果是内存溢出则检查虚拟机的堆参数（-Xmx 与-Xms)。

2.4.2 虚拟机栈和本地方法栈溢出

在 HotSpot 虚拟机中并不区分虚拟机栈和本地方法栈，栈容量只由-Xss 参数设定，虽然-Xoss（设置本地方法栈大小）参数存在，但实际上是无效的。

实验结果表明：单个线程下，无论由于栈帧太大还是虚拟机容量太小，当内存无法分配时候，虚拟机抛出的都是 StackOverflowError 异常。

当持续创建线程时，会产生 OOM 异常。因为线程栈空间 ≈ 总内存（操作系统内存）-Xmx（最大堆容量）-MaxPermSize（最大方法区容量），所以可以通过减少“内存”（Xmx、MaxPermSize）来解决多线程创建造成的 OOM 异常。

2.4.3 方法区和运行时常量池溢出

JDK1.7 之后 String.intern()方法不再复制实例到到永久代中，只是在常量池中记录首次出现的实例引用。所以通过 String.intern()持续创建不会造成常量池 OOM。

方法区用于存放 Class 的相关信息，如类名、访问修饰符、字段描述、方法描述等。基本思路是运行时产生大量的类去填满方法区，直到溢出。

可能出现溢出的场景：Spring、Hibernate 使用 CGLib 对类进行增强、大量 JSP 或动态产生 JSP 文件的应用（JSP 第一次运行时需要编译为 Java 类）、基于 OSGI 的应用（即使是同一个类文件，被不同的类加载器加载也会视为不同的类）等。

2.4.4 本机直接内存溢出

DirectMemory 容量可以通过-XX:MaxDirectMemorySize 指定，如果不指定，则默认与 Java 堆最大值（-Xmx 指定）一样。

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，而程序中又直接或间接使用了 NIO，那就可以考虑检查一下是不是这方面的原因。

3 垃圾收集器与内存分配策略

3.1 概述

GC 关注三件事：哪些内存需要回收？什么时候回收？如何回收？

了解 GC 和内存分配的意义：当需要排查内存溢出、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，就需要对这些“自动化”的技术实施必要的监控和调节。

“内存分配和回收”中讨论的内存指的是“Java 堆和方法区”，因为程序计数器、Java 虚拟机栈、本地方法栈这三个区域的内存分配和回收具备确定性。

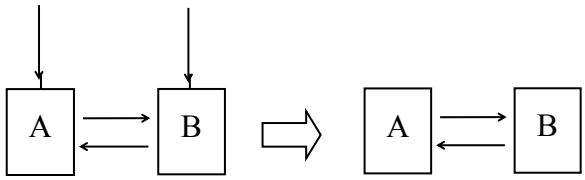
3.2 对象已死吗

3.2.1 引用计数法

引用计数法（Reference Counting）：给对象添加一个引用计数器，每当有一个地方引用它时，计数器值就加 1；当引用失效时，计数器值就减 1；任何时刻计数器值为 0 的对象不可能再被使用。

优点：实现简单，判断效率也很高。

缺点：很难解决对象之间相互循环引用的问题。



3.2.2 可达性分析算法

可达性分析（Reachability Analysis）:通过一系列称为“GC Roots”的对象作为起始点，从**这些节点**开始向下搜索，搜索走过的路径称为引用链（Reference Chain）,从 GC Roots 到某个对象不可达时，则这个对象是不可用的。

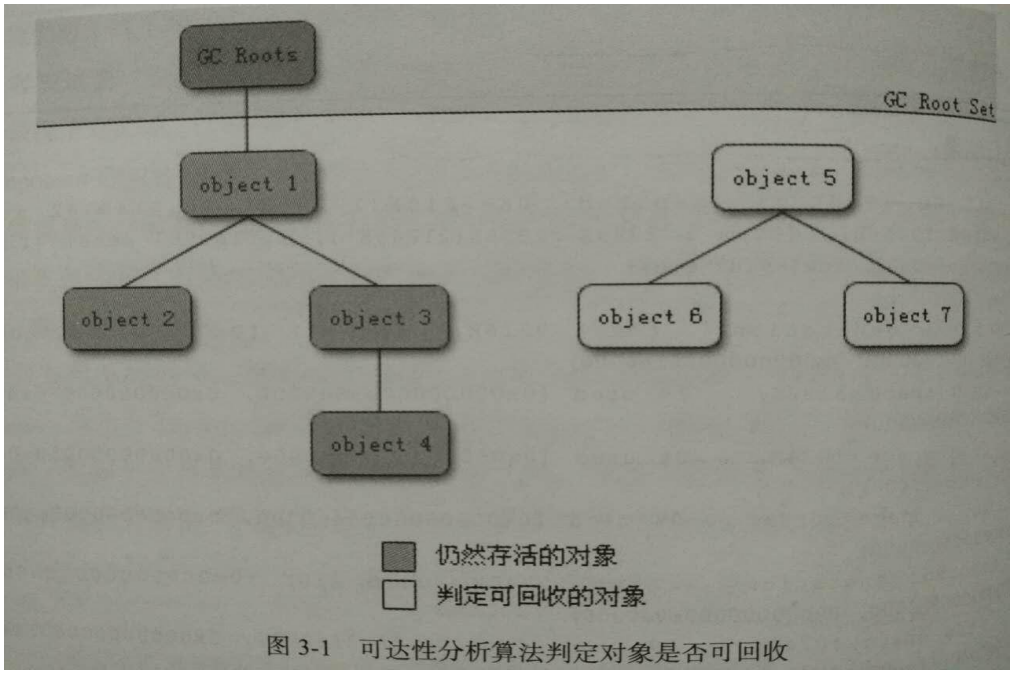


图 3-1　可达性分析算法判定对象是否可回收

在 Java 语言中可以作为 GC Roots 的对象包括以下几种：

（1）虚拟机栈（栈帧中的本地变量表）中引用的对象。

（2）方法区中类静态属性引用的对象。

（3）方法区中常量引用的对象。

（4）本地方法栈中 JNI（即一般说的 Native 方法）引用的对象。

3.2.3 再谈引用

希望能描述这样一类对象：当内存空间还足够的时候，则能保留在内存之中；如果内存空间在进行 GC 后还是很紧张，则可以抛弃这些对象。很多系统的缓存功能都符合这样的场景。

JDK1.2 之后，Java 对引用的概念进行了扩充。

引用类型	实现	存在时间
强引用	Object obj = new Object()	只要引用还存在就存在
软引用	SoftReference	直到将要发生 OOM 时
弱引用	WeakReference	下一次 GC 之前
虚引用	PhantomReference	对对象毫无影响，只是为了在对象回收时受到一个通知

3.2.4 生存还是死亡

要宣告一个对象的死亡，至少要经历两次标记过程。

第一次标记：如果从 GC Roots 不可达，则进行第一次标记并进行筛选，筛选的条件是此对象是否有必要执行 finalize()方法，当对象没有覆盖 finalize()或者已经被调用过，则没有必要再执行该方法。

第二次标记：需要执行 finalize()方法的对象会被放入 F-Queue 中，但是不承诺会全部执行.在执行 finalize()中，可以通过重新与引用链上的任何对象建立关系，从而逃脱“死亡”。稍后，GC 会对 F-Queue 中的对象进行第二次小规模标记。

3.2.5 回收方法区

方法区（或者 Hotspot 虚拟机中的永久代）主要回收两个部分的内容：废弃常量和无用的类。Java 虚拟机规范说过可以不要求虚拟机在方法区实现垃圾收集，而且在方法区中进行垃圾收集的“性价比”一般比较低。

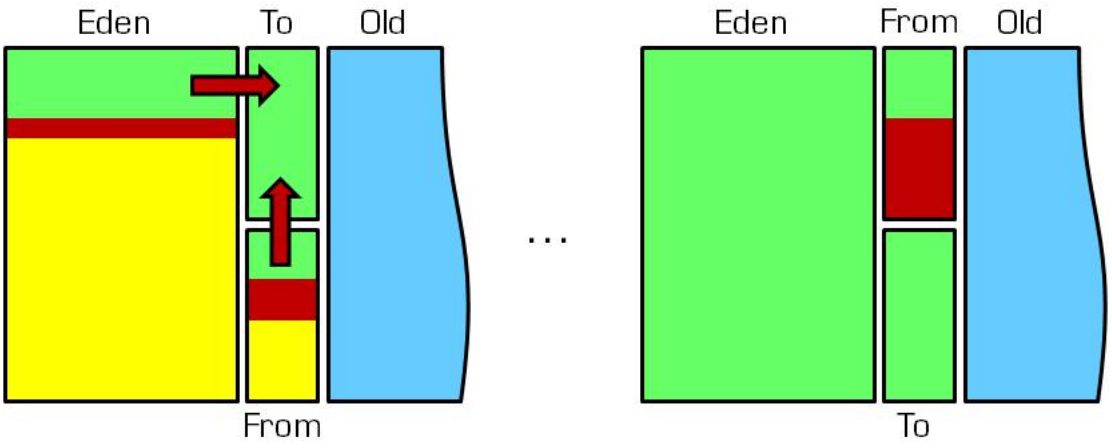
在大量使用反射、动态代理、GCLib 等 ByteCode 框架、动态生成 JSP 以及 OSGi 这类频繁自定义 ClassLoader 的场景都需要虚拟机具备类卸载的功能，以保证永久代不会溢出。

3.3 垃圾收集算法

介绍几种垃圾收集算法。

算法名称	算法过程	算法说明
标记-清除（Mark-Sweep）	分为“标记”和“清除”两阶段：（1）标记出需要回收的的对象。（2）在标记完成后统一回收。	效率问题：标记和清除两个过程效率都不高。空间问题：标记清除后会产生大量不连续的内存碎片。
复制（Copying）	将可用内存按容量划分为大小相等的两块，每次只使用其中一块。	将内存缩小为原来的一般，代价太高。
标记-整理(Mark-Compact)	（1）标记出需要回收的的对象。（2）在标记完成后将存活的对象都向一端移动，然后直接清理掉边界以外的内存。	由于年老代中对象存活率较高，所以在年老代中采用这种算法。
分代收集(Generational Collection)	把 Java 堆分为新生代和老年代。新生代使用“复制”算法，年老代使用“标记-清理”或者“标记-整理”算法。	

现代商业虚拟机**采用复制算法来回收新生代**。将内存划分为一块较大的 Eden 空间和两块较小的 Survivor 空间。当回收时，将 Eden 和 Survivor 中存活的对象一次地复制到另一块 Survivor 上，最后清理掉 Eden 和刚才用过的 Survivor 空间。HotSpot 中 Eden 和 Survivor 的比例是 8:1。当 Survivor 空间不够用时需要依赖其他内存（这里指年老代）进行分配担保（Handle Promotion），直接进入年老代。



3.4 HotSpot 的算法实现

3.4.1 枚举根节点

为了确保准确性，GC 进行时需要停顿所有 Java 执行线程（Sun 将这件事情

称为 “Stop The World”）。

虚拟机有办法得知哪些地方存着对象引用，在 HotSpot 的实现中，使用的是一组称为 OopMap 的数据结构。

3.4.2 安全点

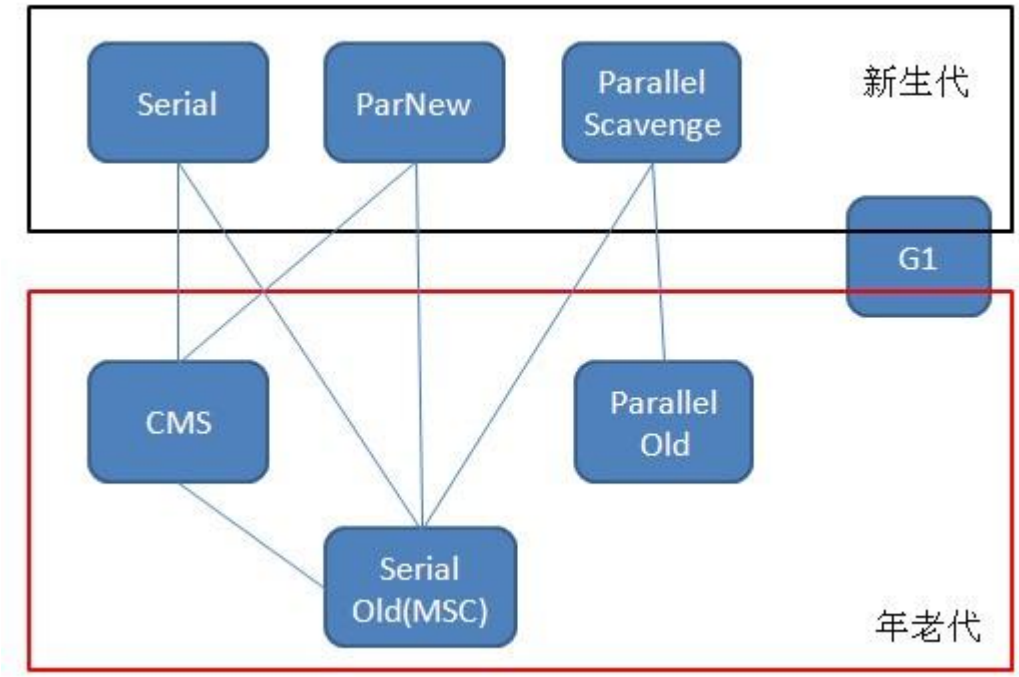
HotSpot 没有为每条指令都生成 OopMap，只是在“特定的位置”记录了这些信息，这些位置称为安全点（Safepoint）,即程序执行时只有在到达安全点时才能暂停，进行 GC。

3.4.3 安全区域

安全区域（Safe Region）可以看做是被扩展了的 Safepoint。

3.5 垃圾收集器

From: http://blog.csdn.net/chjttony/article/details/7883748



图中如果两个垃圾收集器直接有连线，则表明这两个垃圾收集器可以搭配使用。

3.5.1 Serial 垃圾收集器

Serial 是一个单线程的收集器，它不仅仅只会使用一个 CPU 或一条线程去完成垃圾收集工作，并且在进行垃圾收集的同时，必须暂停其他所有的工作线程，

直到垃圾收集结束。

Serial 垃圾收集器虽然在收集垃圾过程中需要暂停所有其他的工作线程，但是它简单高效，对于限定单个 CPU 环境来说，没有线程交互的开销，可以获得最高的单线程垃圾收集效率，因此 Serial 垃圾收集器依然是 **java 虚拟机运行在 Client 模式下默认的新生代垃圾收集器。**

3.5.2 ParNew 垃圾收集器

ParNew 垃圾收集器其实是 Serial 收集器的多线程版本，也使用复制算法，除了使用多线程进行垃圾收集之外，其余的行为和 Serial 收集器完全一样，ParNew 垃圾收集器在垃圾收集过程中同样也要暂停所有其他的工作线程。

ParNew 虽然是除了多线程外和 Serial 收集器几乎完全一样，但是 ParNew 垃圾收集器是**很多 java 虚拟机运行在 Server 模式下新生代的默认垃圾收集器。**

3.5.3 Parallel Scavenge 收集器

Parallel Scavenge 收集器也是一个新生代垃圾收集器，同样使用复制算法，也是一个多线程的垃圾收集器，它重点关注的是程序达到一个可控制的吞吐量（Thoughput，CPU 用于运行用户代码的时间/CPU 总消耗时间，即吞吐量=运行用户代码时间/(运行用户代码时间+垃圾收集时间)），高吞吐量可以最高效率地利用 CPU 时间，尽快地完成程序的运算任务，主要适用于在后台运算而不需要太多交互的任务。

Parallel Scavenge 是吞吐量优先的垃圾收集器，它还提供一个参数：-XX:+UseAdaptiveSizePolicy，这是个开关参数，打开之后就不需要手动指定新生代大小(-Xmn)、Eden 与 Survivor 区的比例(-XX:SurvivorRation)、新生代晋升年老代对象年龄(-XX:PretenureSizeThreshold)等细节参数，虚拟机会根据当前系统运行情况收集性能监控信息，动态调整这些参数以达到最大吞吐量，这种方式称为 GC 自适应调节策略，**自适应调节策略也是 ParallelScavenge 收集器与 ParNew 收集器的一个重要区别。**

3.5.4 Serial Old 收集器

Serial Old 是 Serial 垃圾收集器年老代版本，它同样是个单线程的收集器，使用标记-整理算法，**这个收集器也主要是运行在 Client 默认的 java 虚拟机默认的年老代垃圾收集器。**

在 Server 模式下，主要有两个用途：

a.在 JDK1.5 之前版本中与新生代的 Parallel Scavenge 收集器搭配使用。

b.作为年老代中使用 CMS 收集器的后备垃圾收集方案。

3.5.5 Parallel Old 收集器

Parallel Old 收集器是 Parallel Scavenge 的年老代版本，使用多线程的标记-整理算法，在 JDK1.6 才开始提供。

在 JDK1.6 之前，新生代使用 ParallelScavenge 收集器只能搭配年老代的 Serial Old 收集器，只能保证新生代的吞吐量优先，无法保证整体的吞吐量，Parallel Old 正是为了在年老代同样提供吞吐量优先的垃圾收集器，**如果系统对吞吐量要求比较高，可以优先考虑新生代 Parallel Scavenge 和年老代 Parallel Old 收集器的搭配策略。**

3.5.6 CMS 收集器

Concurrent mark sweep(CMS)收集器是一种年老代垃圾收集器，其最主要目标是获取最短垃圾回收停顿时间，和其他年老代使用标记-整理算法不同，它使用多线程的标记-清除算法。

最短的垃圾收集停顿时间可以为交互比较高的程序提高用户体验，**CMS 收集器是 Sun HotSpot 虚拟机中第一款真正意义上并发垃圾收集器**，它第一次实现了让垃圾收集线程和用户线程同时工作。

3.5.7 G1 收集器

Garbage first 垃圾收集器是目前垃圾收集器理论发展的最前沿成果，相比与 CMS 收集器，G1 收集器两个最突出的改进是：

a.基于标记-整理算法，不产生内存碎片。

b.可以非常精确控制停顿时间，在不牺牲吞吐量前提下，实现低停顿垃圾回收。

G1 收集器避免全区域垃圾收集，它把堆内存划分为大小固定的几个独立区域，并且跟踪这些区域的垃圾收集进度，同时在后台维护一个优先级列表，每次根据所允许的收集时间，优先回收垃圾最多的区域。

区域划分和优先级区域回收机制，确保 G1 收集器可以在有限时间获得最高的垃圾收集效率。

3.6 内存分配与回收策略

3.6.1 对象优先在 Eden 分配

大多数情况下，对象在新生代 Eden 分配。当 Eden 区没有足够空间进行分配时，虚拟机将发起一次 Minor GC。

触发 JVM 进行 Full GC 的情况及应对策略：
http://blog.csdn.net/chenleixing/article/details/46706039

3.6.2 大对象直接进入年老代

虚拟机提供了一个-XX:PretenureSizeThreshold 参数，令大于这个设置值的对象直接在老年代分配。注意：这个参数只对 Serial 和 ParNew 两款收集器有效，Parallel Scavenge 收集器不认识这个参数。

3.6.3 长期存活的对象将进入老年代

3.6.4 动态对象年龄判定

3.6.5 空间分配担保

4 虚拟机性能监控与故障处理工具

4.1 概述

研究数据包括：运行日志、异常堆栈、GC 日志、线程快照（threaddump / javacore 文件）、堆转储快照（heapdump / hprof 文件）等。

4.2 JDK 命令行工具

JDK 中的工具大多是 jdk/lib/tools.jar 类库的一层薄包装而已。借助 tools.jar 我们可以直接在应用程序中实现功能强大的监控分析功能，但 tools.jar 不是 Java 标准的 API，如果引入这个类库，程序就只能运行与 Sun Hotspot 上面。

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

4.3 JDK 的可视化工具

4.3.1 JConsole：Java 监视与管理控制台

JConsole 可以检测到对应线程的死锁。

4.3.2 VisualVM：多合一故障处理工具

除了 JConsole 中的一些基本功能外，VisualVM 还可以下载插件、生成和浏览堆转储快照、分析程序性能、BTrace 动态日志跟踪。

5 调优案例分析与实战（略）

6 类文件结构

字节码（ByteCode）是构成平台无关性的基石。此外，虚拟机的另一种中立特性——语言无关性正越来越被开发者所重视。

整个 Class 文件本质上就是一张表。

7 虚拟机类加载机制

7.1 概述

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。

7.2 类加载的时机

类的生命周期如下图所示。

图 7-1　类的生命周期

虚拟机规范严格规定了有且只有 5 种情况必须立即对类进行“初始化”（而加载、验证、准备自然要在此之前开始）：

1）使用 new 实例化对象、读取一个类的静态字段（被 final 修饰、已在编译期把结果放入常量次的静态字段除外）、调用一个类的静态方法的时候。

2）反射调用的时候。

3）初始化类点的时候，父类未初始化的要初始化。

4）虚拟机启动时要初始化主类。

5）使用 JDK1.7 的动态语言支持时。

7.3 类加载的过程

7.3.1 加载

在加载阶段，虚拟机需要完成以下 3 件事情：

1）通过一个类的全称限定名来获取定义此类的二进制字节流。

2）将这个字节流所代表的静态存储结构转换为方法区的运行时数据结构。

3）在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。

7.3.2 验证

验证大致会完成下面 4 个阶段的检验动作：

1）文件格式验证：验证字节流是否符合 Class 文件格式的规范，并且能被当前版本的虚拟机处理。

2）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。

3）字节码的验证：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。

4）符号引用校验：发生在将符号引用转换为直接引用的时候，这个转化动作将在连接的第三阶段——解析阶段中发生。

7.3.3 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中进行分配。注意：这里是类变量（static 修饰的变量），而且初始值“通畅情况下”（非 final）是 0 值，因为复制代码还没有编译。

7.3.4 解析

将常量池内的符号引用替换为直接引用的过程。

7.3.5 初始化

初始化阶段是执行类构造器<clinit>()方法的过程。

<clinit>()方法是由编译器自动收集类中的所有类变量的赋值动作和静态语句块（static{}块）中的语句合并产生的。

7.4 类加载器

加载器在类层次划分、OSGi、热部署、代码加密等领域大放异彩。

下图展示的类加载器之间的这种层次关系，称为双亲委派模型。双亲委派模型要求除了顶层的启动类加载器外，其余的类加载器都应当有自己的父类加载器（一般是组合而不是继承关系）。

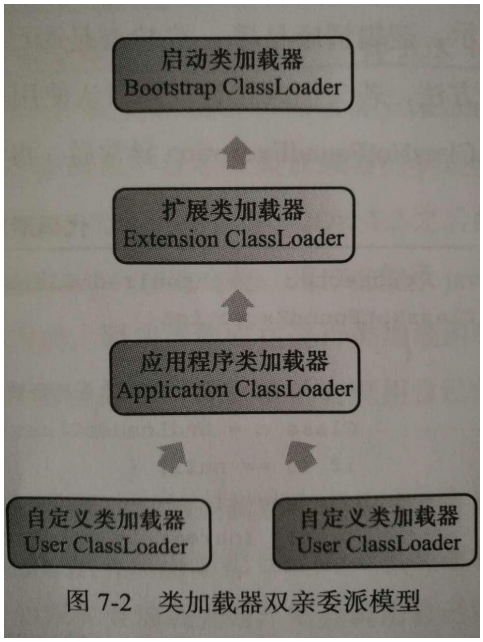


图 7-2　类加载器双亲委派模型

8 虚拟机字节码执行引擎

8.1 概述

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（通过即时逼哪一期产生本地代码执行）两种选择。

8.2 运行时栈帧结构

图 8-1 栈帧的概念结构

略

9 类加载及执行子系统的案例与实战（略）

10 早期（编译期）优化（略）

11 晚期（运行期）优化（略）

12 Java 内存模型与线程

13 线程安全与锁优化