第六章：Java 虚拟机

6.1 Java 虚拟机概念

1.什么是 Java 虚拟机?

Java 虚拟机是一个想象中的机器，在实际的计算机上通过软件模拟来实现。Java 虚拟机有自己想象的硬件，如处理器、堆栈、寄存器，还有相应的指令系统。打个比方来说，Java 程序是汽车，那么 Java 虚拟机就是道路。

那么为什么要用 Java 虚拟机呢？是为了实现 Java 跨平台的特性。

2.Java 虚拟机的生命周期

一个运行中的 Java 虚拟机有着一个清晰的任务：执行 Java 程序。程序开始执行时它才运行，程序结束时它就停止。每个 Java 程序会单独运行一个 Java 虚拟机。

（1）通过命令行启动 Java 虚拟机：java XXX（类名）

Java 虚拟机总是开始于一个 main()方法，这个方法必须是共有 public，返回 void，直接接受一个字符串数组。在程序执行时，必须给 Java 虚拟机知名这个包含有 main()方法的类名。

public static void main(String[] args)

main()方法是程序的起点，它被执行的线程初始化为程序的初试线程。程序中其它的县城都由它来启动。Java 中的线程分为两种：守护线程（daemon）和普通线程（no-daemon）。守护线程是 Java 虚拟机自己使用的线程，比如负责垃圾收集的线程。也可以把自己的程序设置为守护线程。**包含 main()方法的初试线程不是守护线程**。

只要 Java 虚拟机中有普通线程在执行，Java 虚拟机就不会停止。但是如果有足够的权限，就可以调用 exit()方法终止程序。

3.Java 虚拟机的体系结构

在 Java 虚拟机的规范中定义了一系列的子系统、内存区域、数据类型和使用指南。这些组件构成了 Java 虚拟机的内部结构。它们不仅仅为 Java 虚拟机的实现提供了清晰的内部结构，更严格规范了 Java 虚拟机实现的外部行为。

每个 Java 虚拟机都有一个类加载器子系统（class loader subsystem），负责加载程序中的类型（类 class 和借口 interface），并赋予唯一的名字。每一个 Java 虚拟机都有一个执行引擎（execution engine）负责执行被加载类中包含的指令。

4.Java 虚拟机的数据类型

数据类型和操作都在 Java 虚拟机规范中严格定义，都是确定的。Java 中数据类型分为原始数据类型（primitive types）和引用数据类型（reference type）。

原始数据类型如表 1.1 所示。

表 1.1 Java 中的原始数据类型

整型数据	byte
	short
	int
	long
浮点型数据	float
	double

布尔型	boolean
字符型	char

在 Java 虚拟机中还存在一个 java 不能使用的原始数据类型——返回值类型（return value）。这种类型被用来实现 java 程序中的 finally classes。

引用类型可能被创建为：类类型（class type）、接口类型（interface type）、数组类型（array type）。它们都引用被动态创建的对象。当引用类型引用 null 时，说明没有引用任何对象。

6.2 Java 虚拟机内存划分

划分如图 1 所示，左侧是方法区和堆这个区，右侧包括虚拟机栈、本地方法栈和程序计数器。左侧可以被称为线程共享区，右侧是线程私有区。

1.程序计数器

JVM 将这个技术看做当前线程执行某条字节码的行数，会根据计数器的值来选取需要执行的操作语句。这个属于线程私有，不可共享，如共享会导致计数混乱，无法准确执行当前线程需要执行的语句。**该区域不会出现任何 OutOfMemoryError 的情况**。

2.虚拟机栈

也就是通常说的栈内存。Java 中每一个方法从调用直到执行完成的过程，就对应着一个栈帧在虚拟机栈中入栈到出栈的过程。

如果线程请求的栈深度大于虚拟机所允许的深度，则抛出一个 StackOverflowError 异常；如果虚拟机栈可以动态扩展（当前大部分都可，只不过规范中也允许固定长度的），如果扩展时无法申请到足够内存，就会抛出 OutOfMemoryError 异常。

图 1

3.本地方法栈

本地方法栈用来执行本地方法（不一定是 Java 语言方法），抛出异常的情况和虚拟机栈一样。而虚拟基站用来**执行 Java 方法**。

4.堆

是 JVM 中内存最大、线程共享的一块区域。唯一的目的是存储对象实例。这里也是垃圾收集器主要收集的区域。由于现代垃圾收集器采用的是分代收集算法，所以 Java 堆也分为新生代和老年代。

可以通过参数-Xmx（JVM 最大可用内存）和-Xms（JVM 初始内存）来调整堆内存，如果扩大至无法继续扩展时，会出现 OutOfMemoryError 的错误。

5.方法区

JVM 中内存共享的一片区域，用来存储类信息、常量、静态变量、class 文件。垃圾收集器也会对这部分区域进行回收，比如常量池的清理和类型的卸载等。内存不够用时，会出现 OutOfMemoryError 的错误

6.3 Java 虚拟机类加载机制

1.虚拟机类加载机制的概念

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，类型的加载和连接过程是在程序运行期间完成的。

2.类的生命周期

（1）加载。通过一个雷的全限定名来获取此类的二进制字节码，将这个字节码所代表的的静态存储结构转化为方法区的运行时数据结构。在 Java 堆中生成一个代表这个类的 Class 对象，作为方法区这些数据的访问入口。

（2）验证。做出如图 2 所示的验证。

图 2

（3）准备。准备阶段是正式为**类变量**分配内存并设置**初始值**的阶段。这些内存将在**方法区**中进行分配。但是如果类字段的字段属性表中存在 ConstantValue 属性，那么在准备阶段变量值就会被初始化为 ConstantValue 属性指定的值。

（4）解析。解析阶段是在虚拟机将常量池内的符号引用替换为直接引用的过程。

所谓符号引用，是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的定位到目标即可。符号引用与虚拟机实现的内存布局无关，引用的目标并不一定已经加载到内存中。

所谓直接引用是指直接指向目标的指针、相对偏移量或者一个能间接定位到目标的句柄。如果有了直接引用，那饮用的目标必定已经在内存中存在。

（5）初始化。初始化用到了以下方法:

图 3

①类的主动引用。遇到 new、getstatic、putstatic、invokestatic 着四条字节码执行时（使用 new 实例化对象的时候，读取或设置一个类的静态字段、调用一个类的静态方法）。

使用 java.lang.reflet 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，如果父类没有进行初始化，则先触发其父类初始化。

当虚拟机启动时，虚拟机会主动初始化 main()方法。

②类的被动引用。通过子类引用父类的静态字段，不会导致子类初始化（对于静态字段，只有直接定义这个字段的类才会初始化）。

通过数组定义类应用类：ClassA[] array=new ClassA[10]。触发了一个名为 ClassA 的类初始化。它是一个由虚拟机自动生成的，直接集成于 Object 的类，创建动作由字节码指令 newarray 触发。

常量会在编译阶段存入调用类的常量池。

（6）使用

（7）卸载

6.4 判断对象是否存活算法及对象引用

1.什么是垃圾回收?

当一个对象没有引用指向它时，这个对象就成为无用的内存，就必须进行回收，以便用于后续其它对象的内存分配。

2.垃圾回收算法

（1）考虑到垃圾回收是没有引用指向一个对象才回收，那么可以设置一个来得到引用指向对象的个数的变量来判断，若值为 0 则回收，由此引出了“**引用计数算法**”。

该算法实现简单，判断效率也很高，在大部分情况下是一个不错的算法。但是在 Java 中没有选用引用计数算法来管理内存，其中最主要的原因是它**很难解决对象之间相互循环引用**

的问题。例如：

ObjA obj = ObjB

ObjB obj = ObjA

这两个相互引用的指针都为 1，如果按照前面的算法的话，这两个就不能被回收。

（2）可达性分析算法（根搜索算法）

在主流商用程序语言（包括 Java、C#等），都是利用这个方法。它的基本思路是这样的：它通过一系列名为 GCroot（GC 即为 Garbage Collection）的对象作为起始点，GCroot 就是根节点，从这些节点开始往下搜索，搜索所走过的路径称为引用链。当一个对象到这个 GCroot 没有任何引用链相连时，就证明这个对象是不可用的，这个节点视为垃圾回收的对象。

在 Java 语言里，可作为 GC Roots 对象的包括以下几种：

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方法区中的常量引用对象；

④本地方法栈中 JNI 的引用的对象。

3.对象引用的类型

（1）强引用

只要引用存在，垃圾回收器永远不会回收。即 Object obj = new Object();

obj 对象对后面的 new Object 有一个强引用，只有当 obj 这个引用被释放之后（例如将 obj=null），对象才会被释放掉。

（2）软引用

非必须引用，内存溢出前进行回收，可通过下面代码实现：

```
Object obj = new Object();
SoftReference<Object> sf =new SoftReference<Object>(obj); //obj 转换成一个软引用
obj = null;
sf.get();
```

软引用主要用于实现类似缓存的功能，在内存足够的情况下直接通过软引用取值，无需从繁忙的真实来源查询数据，这样能够提升速度；当内存不足时，自动删除这部分缓存数据，从真正的来源查询这些数据。

（3）弱引用

在第二次垃圾回收时回收，可以通过如下代码实现：

```
Object obj = new Object();
WeakReference<Object> wf =new WeakReference<Object>(obj);//obj 转换成一个弱引用
obj = null;
wf.get();//有时会返回 null
wf.isEnQueued();//是不是一个队列
```

弱引用主要用于监控对象是否已经被垃圾回收器标记为即将回收的垃圾，可以通过弱引用的 isEnQueued 方法返回对象是否被垃圾回收器所回收。

（4）虚引用（幽灵/幻影引用）

在垃圾回收时回收，无法通过引用取到对象之，可以通过如下代码实现：

```
Object obj = new Object();
PhantomReference<Object> pf =new PhantomReference<Object>(obj);//obj 转换成一个虚引用
obj = null;
```

```
pf.get();//永远返回 null
pf.isEnQueued();
```

主要用于检测对象是否已经从内存中删除。

6.5 分代垃圾回收

Java 语言没有显式的提供分配内存和删除内存的方法。一些开发人员将引用对象设置为 null 或者调用 System.gc()来释放内存，但后者会严重影响计算机的性能。

在 Java 中，由于开发人员没有在代码中显式删除内存，所以垃圾收集器会去发现不需要（垃圾）的对象，然后删除它们，释放内存。

分代垃圾收集器是按照下面两个假设创建的：

（1）绝大多数对象在短时间内变得不可达；

（2）只有少两年老对象引用年轻对象。

年轻代：新创建的对象都存放在这里。因为大多数对象很快变得不可达，所以大多数对象在年轻代中创建，然后消失。当对象从这块内存区域消失时，我们说发生了一次“minor GC”。

老年代：没有变得不可达，存活下来的年轻代对象被复制到这里。这块内存区域一般大于年轻代，因为它具有更大的规模，GC 发生的次数比在年轻代更少。对象从老年代消失时，我们说“major GC”或者“Full GC”发生了。

年轻代组成部分：

年轻代总共有 3 块空间，1 块为 Eden 区，2 块为 Survivor 区。各个空间的执行顺序如下：

1.绝大多数新创建的对象分配在 Eden 区。

2.在 Eden 区发生一次 GC 后，存活的对象转移到其中一个 Survivor 区。

3.一旦一个 Survivor 区满，存活的对象移动到另外一个 Survivor 区。然后之前那个空间已满区将置为空，没有任何数据。

4.经过重复多次这样的步骤后依然存活的对象呗移到老年区。

6.6 典型的垃圾收集算法

1.Mark-Sweep（标记-清除）算法

如图 4 所示，灰色是存活对象，绿色是未使用，黑色是可回收。

图 4

这是最基础的算法，之所以说最基础是因为其容易实现，思想最为简单。标记-清除算法分为两个阶段：标记阶段和清除阶段。标记阶段的任务是标记出所有需要被回收的对象，清除阶段就是回收被标记的对象所占用的空间。但是它容易产生内存碎片，碎片太多会导致后续过程中需要为大对象分配空间时无法找到足够的空间而提前触发新的一次垃圾收集动作。

2.Copying（复制）算法

如图 5 所示，它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。当这一块内存用完了，就将还存活着的对象复制到另外一块上面，然后再把以使用的内存空间一次清理掉，这样一来就不容易出现内存碎片问题。

这种算法虽然实现简单、运行高效，但是却对内存空间的使用作出了高昂的代价。

图 5

3.Mark-Compact（标记-整理）算法

如图 6 所示，为了充分利用内存空间，解决 Copying 算法缺陷，提出了该算法，该算法标记阶段和 Mark-Sweep 一样，但是在完成标记之后，它不是直接清理可回收对象，而是将存活对象都向一端移动，然后清理掉端边界以外的内存。

4.Generational Collection（分代收集）算法

JVM 中最常用的算法。在第 5 节已经讲过。

目前大部分垃圾收集器对于新生代都采取 Copying 算法，因为新生代中每次垃圾回收都要回收大部分对象，也就是说需要复制的操作次数比较少，但是实际中并不是按照 1:1 来划分新生代空间，而是如第 5 节所述一个 Eden 空间和两个 Survivor 空间。

老年代是每次回收指挥手少量对象，因此一般使用 Mark-Compact 算法。

注意：在堆区之外还有一个代是**永久代（Permanent Generation）**，它用来存储 class 类、常量、方法描述等。对于永久代的回收主要回收两部分内容：**废弃常量**和**无用的类**。

图 6

6.7 典型的垃圾收集器

1.Serial/Serial Old 收集器

它是最基本最古老的收集器，是一个单线程收集器，并且在它进行垃圾收集时，必须暂停所有用户线程。Serial 收集器是针对新生代的收集器，采用 copying 算法，serial old 收集器针对老年代收集，采用 Mark-Compact 算法。它有点是简单高效，缺点是给用户带来停顿。

2.ParNew

ParNew 是 Serial 收集器的多线程版本，使用多个线程进行垃圾收集。如图 7 所示。它除了使用多条线程进行垃圾收集之外，其余行为包括 Serial 收集器可用的所有控制参数（例如：-XX:SurvivorRatio、-XX:PretenureSizeThreshold、-XX:HandlePromotionFailure 等）、收集算法、Stop The World、对象分配规则、回收策略等都与 Serial 收集器一致。

新生代并行，老年代串行；新生代复制算法、老年代标记-压缩。

图 7

3.Parallel Scavenge

Parallel Scavenge 收集器类似 ParNew 收集器，它是一个新生代的多线程收集器（并行收集器），它在回收期间不需要暂停其他用户线程。Parallel 收集器更关注系统的**吞吐量**。可以通过参数来打开**自适应调节策略，**虚拟机会根据当前系统的运行情况收集性能监控信息，动态调整这些参数以提供最合适的停顿时间或最大的吞吐量；也可以通过参数控制 GC 的时间不大于多少毫秒或者比例；新生代复制算法、老年代标记-压缩。

4.Parallel Old 收集器

Parallel Old 是 Parallel Scavenge 收集器的老年代版本，使用多线程和“标记－整理”算法。这个收集器是在 JDK 1.6 中才开始提供。

参数控制： -XX:+UseParallelOldGC 使用 Parallel 收集器+ 老年代并行。

5.CMS 收集器

如图 8 所示，CMS（Concurrent Mark Sweep）收集器是一种以获取最短回收停顿

时间为目标的收集器。目前很大一部分的Java应用都集中在互联网站或B/S系统的服务端上，这类应用尤其重视服务的响应速度，希望系统停顿时间最短，以给用户带来较好的体验。

从名字（包含“Mark Sweep”）上就可以看出CMS收集器是基于“标记-清除”算法实现的，它的运作过程相对于前面几种收集器来说要更复杂一些，整个过程分为 4 个步骤，包括：

①初始标记（CMS initial mark）

②并发标记（CMS concurrent mark）

③重新标记（CMS remark）

④并发清除（CMS concurrent sweep）

其中初始标记、重新标记这两个步骤仍然需要“Stop The World”。初始标记仅仅只是标记一下GC Roots能直接关联到的对象，速度很快，并发标记阶段就是进行GC Roots Tracing的过程，而重新标记阶段则是为了修正并发标记期间，因用户程序继续运作而导致标记产生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。

由于整个过程中耗时最长的并发标记和并发清除过程中，收集器线程都可以与用户线程一起工作，所以总体上来说，CMS收集器的内存回收过程是与用户线程一起并发地执行。**老年代收集器**（新生代使用ParNew）

优点:**并发收集、低停顿**

缺点：**产生大量空间碎片、并发阶段会降低吞吐量。**

图 8

6.G1 收集器

G1 是目前技术发展的最前沿成果之一，HotSpot开发团队赋予它的使命是未来可以替换掉JDK1.5 中发布的CMS收集器。与CMS收集器相比G1 收集器有以下特点：

1. 空间整合，G1 收集器采用标记整理算法，不会产生内存空间碎片。分配大对象时不会因为无法找到连续空间而提前触发下一次GC。

2. 可预测停顿，这是G1 的另一大优势，降低停顿时间是G1 和CMS的共同关注点，但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N毫秒的时间片段内，消耗在垃圾收集上的时间不得超过N毫秒，这几乎已经是实时Java（RTSJ）的垃圾收集器的特征了。

上面提到的垃圾收集器，收集的范围都是整个新生代或者老年代，而G1 不再是这样。使用G1 收集器时，Java堆的内存布局与其他收集器有很大差别，它将整个Java堆划分为多个大小相等的独立区域（Region），虽然还保留有新生代和老年代的概念，但新生代和老年代不再是物理隔阂了，它们都是一部分（可以不连续）Region的集合。

G1 的新生代收集跟ParNew类似，当新生代占用达到一定比例的时候，开始出发收集。和CMS类似，G1 收集器收集老年代对象会有短暂停顿。如图 9 所示。

图 9

收集步骤：

1、标记阶段，首先初始标记(Initial-Mark),这个阶段是停顿的(Stop the World Event)，并且会触发一次普通Mintor GC。对应GC log:GC pause (young) (inital-mark)

2、Root Region Scanning，程序运行过程中会回收survivor区(存活到老年代)，这一过程必须在young GC之前完成。

3、Concurrent Marking，在整个堆中进行并发标记(和应用程序并发执行)，此过程可能被young GC中断。在并发标记阶段，若发现区域对象中的所有对象都是垃圾，那个这个区

域会被立即回收(图中打X)。同时，并发标记过程中，会计算每个区域的对象活性(区域中存活对象的比例)。

图 10

4、Remark，再标记，会有短暂停顿(STW)。再标记阶段是用来收集 并发标记阶段 产生新的垃圾(并发阶段和应用程序一同运行)；G1 中采用了比CMS更快的初始快照算法:snapshot-at-the-beginning (SATB)。

5、Copy/Clean up，多线程清除失活对象，会有STW。G1 将回收区域的存活对象拷贝到新区域，清除Remember Sets，并发清空回收区域并把它返回到空闲区域链表中。

图 11

6、复制/清除过程后。回收区域的活性对象已经被集中回收到深蓝色和深绿色区域。

图 12

对常用的收集器组合的总结见表 6.1 所示。

表 6.1 常用的收集器组合

	新生代 GC 策略	年老代 GC 策略	说明
1	Serial	Serial Old	Serial 和 Serial Old 都是单线程进行 GC，特点就是 GC 时暂停所有应用线程。
2	Serial	CMS+Serial Old	CMS（Concurrent Mark Sweep）是并发 GC，实现 GC 线程和应用线程并发工作，不需要暂停所有应用线程。另外，当 CMS 进行 GC 失败时，会自动使用 Serial Old 策略进行 GC。
3	ParNew	CMS	使用-XX:+UseParNewGC 选项来开启。ParNew 是 Serial 的并行版本，可以指定 GC 线程数，默认 GC 线程数为 CPU 的数量。可以使用-XX:ParallelGCThreads 选项指定 GC 的线程数。 如果指定了选项-XX:+UseConcMarkSweepGC 选项，则新生代默认使用 ParNew GC 策略。
4	ParNew	Serial Old	使用-XX:+UseParNewGC 选项来开启。新生代使用 ParNew GC 策略，年老代默认使用 Serial Old GC 策略。
5	Parallel Scavenge	Serial Old	Parallel Scavenge 策略主要是关注一个可控的吞吐量：应用程序运行时间 /（应用程序运行时间 + GC 时间），可见这会使得 CPU 的利用率尽可能的高，适用于后台持久运行的应用程序，而不适用于交互较多的应用程序。
6	Parallel Scavenge	Parallel Old	Parallel Old 是 Serial Old 的并行版本
7	G1GC	G1GC	-XX:+UnlockExperimentalVMOptions -XX:+UseG1GC #开启 -XX:MaxGCPauseMillis =50 #暂停时

			间目标 -XX:GCPauseIntervalMillis =200 #暂停间隔目标 -XX:+G1YoungGenSize=512m #年轻代大小 -XX:SurvivorRatio=6 #幸存区比例