jvm学习笔记

1 jvm简介

- java技术体系提供了完整的用于软件开发和跨平台的支持环境，并广泛应用于嵌入式系统、移动终端、企业服务器、大型机等场合。java实现了“一次编写，导出运行”的理想
- java代码是运行于java虚拟机上的，通过java虚拟机实现了跨平台，并且java虚拟机帮助程序员做一系列易出错的事务，比如内存管理

2 Java内存区域与内存溢出异常

2.1 运行时数据区域

- Java虚拟机在执行Java程序的过程中会把她锁管理的内存划分为若干个不同的数据区域，这些区域称为运行时数据区域。Java虚拟机所管理的内存包含以下几个运行时数据区域，如图2-1

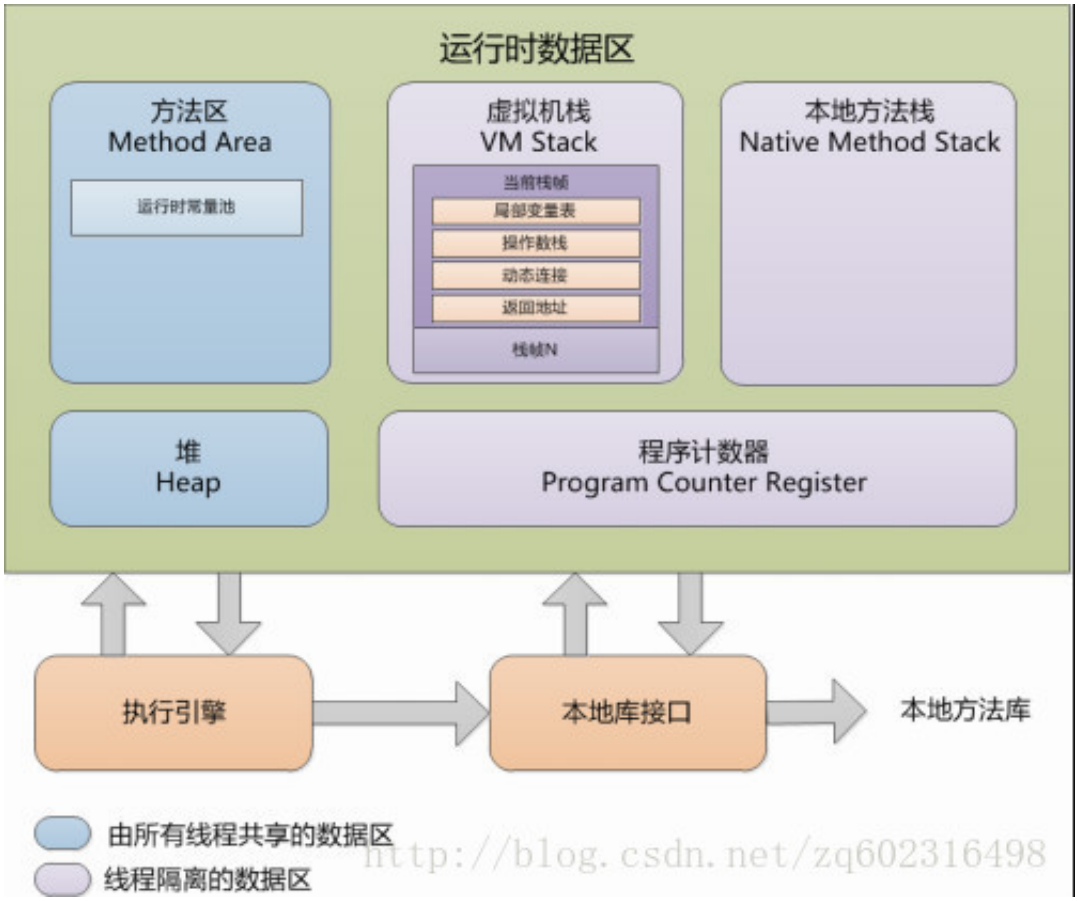

图2-1 Java虚拟机运行时数据区

2.1.1 程序计数器

程序计数器可看作当前线程所执行的字节码的行号指示器，是一块较小的内存空间，线程私有。

2.1.2 Java虚拟机栈

Java虚拟机栈也是线程私有的，生命周期与线程相同。虚拟机指的是Java方法执行模型：每个方法在执行的同时都会创建一个栈帧用于存储局部变量表、操作数栈、动态链接、方法出口等信息。

```
Java虚拟机规范中定义了两种异常
1. StackOverflowError:若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
2. OutOfMemoryError:若虚拟机栈可以动态扩展，但扩展时无法申请到足够的内存，就会抛出OutOfMemoryError异常
```

2.1.3 本地方法栈

与Java虚拟机栈的作用相似，但本地方法栈为虚拟机使用到的Native方法服务，虚拟机栈为虚拟机执行的Java方法服务

2.1.4 Java堆

Java堆是jvm管理内存中最大的一块，是所有线程共享的一块内存区域，在虚拟机启动时创建。几乎所有的对象实例和数组都在这里分配。

2.1.5 方法区

方法区试线程共享的内存区域，用于存储一杯虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据

2.1.6 运行时常量池

运行时常量池是方法区的一部分，用于存放编译器生成的各种字面量和符号引用。

对象的访问定位

Java程序通过栈上的reference数据来操作堆上的具体对象。reference在Java虚拟机规范中只规定了一个指向对象的引用，没有定义如何去定位寻找对象的具体位置。目前主流方式通过句柄和直接指针两种。

```
1.句柄访问：Java堆中划分出一块内存作为句柄池，reference存储的就是对象句柄地址，句柄中包含了对象实例数据与类型数据各自的具体地址信息。
好处是reference存储的是稳定的句柄地址，在对象被移动时只会改变句柄池中的实例数据指针，而reference本身不需要改变
2.直接指针访问：reference存储的是对象地址。
好处是速度更快，节省了一次指针定位的时间开销
```

内存溢出

Java堆溢出

当出现OutOfMemeryError时，先分析内存泄露还是内存溢出。若是内存泄露，就通过GC Root的引用链找出GC为什么无法自动回收它们；若内存溢出可以考虑为虚拟机申请更多的内存

虚拟机栈和本地方法栈溢出

```
StackOverflowError：线程请求的栈深度大于虚拟机所允许的最大深度时抛出。使用-Xss参数减少栈内存，异常出现时输出到栈深度相应缩小
OutOfMemeoryError：虚拟机在扩展栈时无法申请到足够的内存时抛出
```

方法区和运行时常量池溢出

不断地生成常量，抛出OOM

3 垃圾收集器与内存分配策略

垃圾收集器（Garbage Collection，GC）

- 那些内存需要回收
- 什么时候回收
- 如何回收

对象已死吗

引用计数法

给对象添加引用计数器，当指向对象的引用为0时，对象可被回收

可达性分析

以GC Roots为起点，GC Roots不可达的对象，将被判定为可回收的对象。

> GC Roots的对象
>
> 1. 虚拟机栈（栈帧中的本地变量表）中引用的对象
> 2. 方法取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

引用

- 强引用：只要存在，对象就不会被GC回收
- 软引用：描述有用但非必需的对象。软引用的关联的对象，在系统发生内存溢出异常之前，将会把这些对象列进回收范围之中进行第二次回收
- 弱引用：描述非必需对象，关联的对象只能存活到下次垃圾回收之前

- 虚引用：虚引用的存在不会影响对象的生存时间，虚引用关联的唯一目的时能在该对象回收时收到一个系统通知 ##### finalize() 一个对象真正要被回收至少需要经历两次标记过程。若对象覆盖了finalize()方法且该方法没有被执行过，finalize()会被放入虚拟机的一个队列中执行，虚拟机不会等待它运行结束，对象可以在此函数里自救，但此函数每个对象只能触发一次。

垃圾回收算法

- 标记-清除算法
- 复制算法
- 标记-整理算法
- 分代收集算法

垃圾收集器

1. Serial收集器。单线程收集器，进行垃圾回收时，必须暂停其他线程
2. ParNew收集器。Serial的多线程版本
3. Parallel Scavenge收集器。新声代收集器，采用复制算法，并行。该收集器的关注点是尽可能达到可控制的吞吐量（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））。-XX:MaxGCPauseMillis控制最大垃圾收集停顿时间）（尽可能控制在此时间之内）；-XX:GCTimeRatio设置吞吐量大小
4. Serial Old收集器。老年代单线程收集器，使用标记-整理算法
5. Parallel Old收集器
6. CMS收集器。以获取最短停顿时间为目标的收集器。收集过程分四个步骤：初始标记、并发标记、重新标记、并发清除。

CMS被称为并发低停顿收集器，但也存在明显缺点：
1.对CPU资源非常敏感。CMS默认启动的垃圾回收的线程数是（CPU数量+3）/4。垃圾回收的线程在CPU数量大于4时约占25%，CPU数越少占比越高。
2.无法处理浮动垃圾，cms在并发清理阶段还会出现其他垃圾，这些垃圾在标记过后，是无法清除的，这些被称为浮动垃圾。因此需要预留一部分空间进行收集，当空间占用量达到阈值时进行回收，-XX:CMSInitatingOccupancyFraction用来设置这个启动阈值，百分比表示。JDK1.6中默认92%。当预留的内存不足时会出现“Concurrent Mode Failure”失败，此时临时启动Serial Old收集器重新收集。
3.CMS是基于标记-清楚算法实现的收集器，可能产生过多的空间碎片，会给大对象分配造成麻烦。CMS中提供了-XX:+UseCMSCompactAtFullCollection开关（默认开启），开启表示FullGC时进行内存碎片的合并整理。-XX：CMSFullGCsBeforeCompaction（默认0）用来设置执行多次不压缩的FullGC后，再执行一次压缩的FullGC。

1. G1收集器

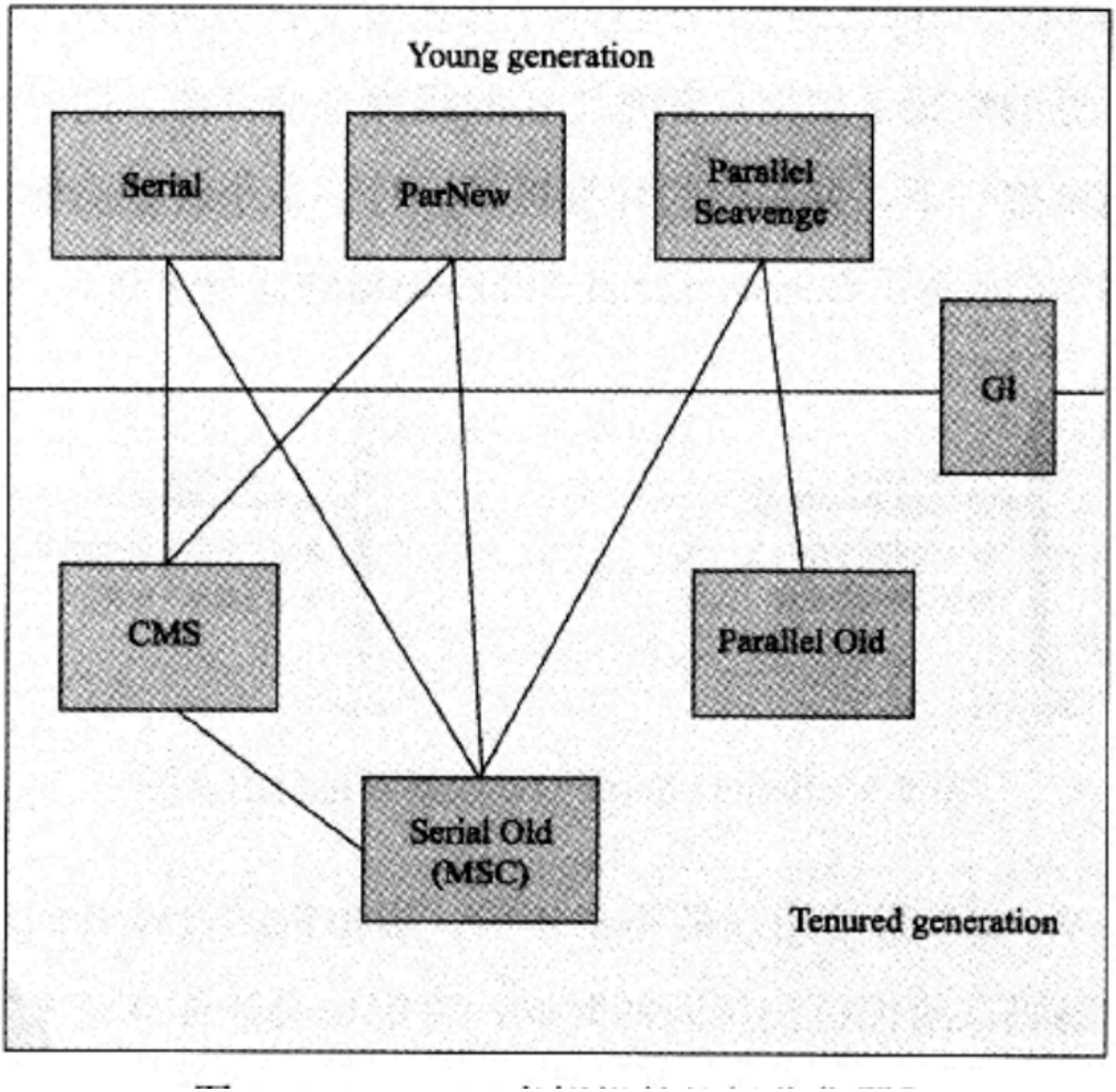

图3-1 HotSpot虚拟机的垃圾收集器

参考

1. 深入理解java虚拟机 第二版
2. https://www.jianshu.com/p/c69f9f7c273b