最近开始了全面的JAVA生态环境学习，因此，JVM的学习是必不可少的一个环节。和.NET的CLR一样，一起的JAVA应用均跑在JVM虚拟机上，不过相对我们只能干看看的CLR，JVM有很大的灵活性，可以通过配置优化JVM的性能，同时针对JVM相关的监控软件也非常丰富。这部分知识有一些晦涩，为了成为一名合格的JAVA程序员，再硬的骨头也要啃下来，由于自身对这部分没有特别多的心得，将借鉴参考资料中标注的文章进行学习。

首先通过数据类型来引入一个高级语言的核心概念，**堆和栈**。JAVA的基本类型包括：byte, short, int, long, returnAddress等，其存储在栈上；引用类型包括：类类型，接口类型和数组，其存储在堆上。在java中，一个线程就会有相应的线程栈与之对应，而堆则是所有线程共享的。栈是运行单位，因此里面存储的信息都是跟当前线程相关信息的，包括局部变量、程序运行状态、方法返回值等；而堆只负责存储对象信息。

之所以将对和栈分离，有如下几点原因：栈代表了逻辑处理，而堆代表数据，满足分治的思想；堆中的内容可以被多个栈共享，即提供数据交换的方式又节省空间；使得存储地址动态增长成为可能，相应栈中只需要记录堆中的一个地址即可；对面向对象的诠释，对象的属性就是数据，存放在堆中，对象的行为是运行逻辑，放在栈中；堆和栈中，栈是程序运行最根本的东西，程序运行可以没有堆，但不能没有栈，而堆是为栈进行数据存储服务的，就是一块共享的内存，这种思想使得垃圾回收成为可能。

**Java对象的大小**:一个空Object对象的大小是8byte，以及其地址空间4byte(32位)，比如对于int这个基础类型，其包装类型Integer大小为8+4=12byte，但由于java对象大小需要时8byte的倍数，因而为16byte，因此包装类型的消耗是基础类型的2倍。

**强引用**、软引用、弱引用和虚引用：强引用是一般虚拟机生成的引用，虚拟机严格的将通过它判断是否需要回收；软引用一般作为缓存使用，当内存紧张时，会对其进行回收；弱引用，也是作为缓存使用，不过一定会被垃圾回收。

**JVM的组成**，可以通过下图对其有个大体的了解。



Class Loader：加载大Class文件，该文件的格式由《JVM Specification》规定，包括父类，接口，版本，字段，方法等元数据信息。

Execution Engine：执行引擎也叫解释器，负责解释命令，提交OS执行。所谓的JIT指的就是提前将中间语言字节码转化为目标文件obj。

Native Interface:为了融合不同语言，java开辟了一块区域用于处理标记为native的代码，现在已很少使用。

Runtime data area：运行数据区是JVM的重点，所写的程序就被加载在这儿。

此外，jvm的寄存器包括：pc,java程序计数器；optop，指向操作数栈顶的指针；frame，指向当前执行方法的执行环境指针；vars，指向当前执行方法的局部变量区第一个变量的指针。

**JVM的内存管理**，所有的数据都是放在运行数据区，接下来介绍其中最复杂的栈(Stack)，也叫栈内存，是java程序的运行区，在线程创建时创建，它的生命周期跟随线程的生命周期，线程结束栈内存就释放，对于栈来说不存在垃圾回收。栈中的数据是以栈帧(Stack Frame)来存放的，其是一块内存区块，是一个有关方法和运行期数据的数据集，当方法A被调用时就产生一个栈帧F1，并压入到栈中，A方法又调用了B方法，于是产生的栈帧F2也被压入栈，执行完毕后，先弹出F2，再弹出F1，遵循“先进后出”原则，JAVA Stack的大体结构如下所示。



回收算法的分类方式有很多，接下来通过一张表格对其进行一个简单的介绍。

|  |  |
| --- | --- |
| 算法类别 | 原理阐述 |
| 按基本回收策略分 | |
| 引用计数(Reference Counting) | 针对某个对象，其每有一个引用，即增加一个计数，删除一个就减少一个计数，垃圾回收时只收集计数为0的对象，缺点是无法处理循环引用的情况 |
| 标记-清除(Mark-Sweep) | 分为两个阶段，首先从引用根结点开始标识所有引用的对象，之后遍历整个堆，把未标记的对象删除，此算法需要暂停整个应用，同时会产生内存碎片 |
| 复制(Copying) | 把内存空间划分为2个相等区域，每次使用一个，当垃圾回收时，遍历当前使用区域，把使用中对象赋值到另一个区域，该复制操作成本较小。并可以进行内存整理，缺点是需要两倍的内存空间 |
| 标记-整理(Mark-Compact) | 该算法结合了“标记-清除”和“复制”的优点，第一阶段从根结点开始标记对象，第二阶段遍历整个堆，清除未标记对象并把存活对象压缩到堆的其中一块，按顺序排放，同时解决碎片和空间问题。 |
| 按分区对待的方式分 | |
| 增量收集(Incremental Collecting) | 实时垃圾回收，即在应用进行的同时进行 |
| 分代收集(Generational Collecting) | 基于对象生命周期分析得出的算法，把对象分为年轻代、年老代和持久代，对不同生命周期的对象使用不同的算法。 |

**垃圾回收的判断**：由于引用计数方式无法解决循环引用，因而实际上，回收算法都是从根结点出发，遍历整个对象引用，查找存活对象。搜索的起点为栈（例如java的Main函数）或者是运行时的寄存器，通过其代表的引用找到堆中对象，逐步迭代，直到以null引用或基本类型结束，该结果是一个对象树，回收器会对未在该树的对象进行回收。

**分代的概念**：由于不同对象的生命周期不同，根据其自己的特点采取不同的收集方式可以大幅提高回收效率。比如与业务相关的对象一般生命周期较长，而临时变量生命周期很短，通过分代，可以避免长生命周期的对象被遍历，以此来减少消耗。

**如何分代**：虚拟机分为年轻代(Young Generation)、年老代(Old Generation)和持久代(Permanent Generation)。所有新生成的对象首先是放在年轻代中，该代的目标就是尽快回收那些短生命周期的对象，其分为3个区，一个Eden区，两个Survior区。大部分对象在Eden区生成，当该区满时，将存活对象复制到Survivor区（两个中的一个），当该区也满了时，将存活对象复制到另一个Survivor，当这个Survivor也满了时，将从第一个Survivor区复制过来的并且还存活的对象复制到年老区Tenured，因此在年老区中主要存放生命周期较长的对象。而持久代，用于存放静态文件，如Java类、方法等。持久代对垃圾回收无显著影响，但App使用较多反射时，需要增加持久代的大小，通过设置-XX:MaxPermSize=<N>。接下来通过一张图，对该部分有个宏观的了解。



垃圾回收算法的触发：由于对象进行了分代处理，因此垃圾回收的区域和时间也有了不同，主要包括如下两种类型的GC。

**Scavenge GC**:一般当新对象生成，并且在Eden申请空间失败时，触发。将清除Eden区的非存活对象，并把存货对象移动到Survivor，然后整理两个Survivor区。该方式不会影响到老年代，此外，该GC推荐使用速度快，效率高的算法，使Eden区尽快空闲出来。

**Full GC**:对整个堆进行整理，包括Young、Tenured和Perm，因此为了提高系统性能，需要减少FullGC的次数。发生FullGC的场景有：年老代写满，持久代被写满和System.gc()被显示调用，上一次GC后Heap各域分配策略动态变化。

接下来通过一个表格来连接不同的收集器的优缺点。

|  |  |
| --- | --- |
| 收集器名称 | 诠释 |
| 串性收集 | 使用单线程处理所有垃圾回收，简单高效，适合数据量小的场景。通过-XX：+**UseSerialGC**打开 |
| 并行收集 | 对年轻代进行并行垃圾回收，因此可以减少垃圾回收时间，使用-XX:+**UseParallelGC**打开。  可以对老年代进行并行收集，默认使用单线程垃圾回收，使用-XX:+UseParallelOldGC打开  使用-XX:ParallelGCThreads=<N>设置并行垃圾回收的线程数，此值可以和机器处理器数相等  通过-XX:MaxGCPauseMillis=<N>设置最大垃圾回收暂定时间  通过-XX:GCTimeRatio=<N>垃圾回收时间与非垃圾回收时间的比值，那么1/(1+N)即为当先系统的吞吐量，N默认值为99，即1%时间用于垃圾回收 |
| 并发收集 | 前两者在垃圾回收时，应用会有明显的暂停，该方式可以减少该影响，保证大部分工作并发进行（应用不停止），适合中大规模应用，使用-XX:+**UseConcMarkSweepGC**打开，由于并发收集比较复杂，接下来介绍几个基本概念。  浮动垃圾：由于在应用运行时进行垃圾回收，所有有些垃圾可能在垃圾回收进行完成时产生，这样就造成了“Floating Garbage”，这些垃圾需要在下次垃圾回收周期才能回收，所以并发收集器需要保留20%的预留空间用于这些浮动垃圾。  Concurrent Mode Failure:由于在垃圾回收时系统运行，需要保证有足够空间给程序使用，否则堆满时，会发生“并发模式失败”，整个应用暂停，进行垃圾回收。可以通过设置-XX:CMSInitiatingOccupancyFraction=<N>指定还有多少神域堆空间时开始执行并发收集 |

**新一代的垃圾回收算法**（Garbage First, G1）：该算法是为大型应用准备的，支持很大的堆和高吞吐量。该算法简单来说，就是把整个堆划分为一个个等大小的区域。内存的回收和划分都以region为单位，同时汲取CMS特点，把垃圾回收过程分为几个阶段。G1在扫描了region以后，对其中的活跃对象的大小进行排序，首先会收集那些活跃对象小的region，以便快速回收空间，因为活跃对象小，里面可以认为多数都是垃圾，所有这种方式被称为Garbage First，即垃圾优先回收，整个垃圾回收过程包含如下几个步骤。

**初始标记**(Initial Marking)：G1对于每个region都保存了两个标记用的bitmap，一个为previous marking bitmap，一个next marking bitmap，bitmap中包含了一个bit的地址信息指向对象的起始点。在开始标记前，首先并发的清空next marking bitmap，然后停止所有应用线程，并扫描标识出每个region中root可直接访问的对象，将region的top值放入next top at mark start（TAMS），之后恢复所有线程。

**并发标记**(Concurrent Marking)：按照之前的标记扫描对象，以标识这些对象的下层对象的活跃状态，将在此期间使用线程并发修改的先关记录写入remembered set logs中，新创建的对象则放入比top值更高的地址区间中，这些新创建的对象默认状态即为活跃的，同时修改top值。

**最终标记暂停**(Final Marking Pause)：当应用线程的remembered set logs未满时，是不会放入filled RS buffers中的，因此需要在此步骤中处理remembered set logs并修改相应的remembered set。

**存活对象计算并清除**(Live Data Counting and Cleanup)：该步骤的触发依赖内存空间是否达到H（H=（1-h）\*HeapSize, h为JVM Heap大小的百分比阀值）。

JVM的相关配置项非常的多，首先通过一个通用的配置理解堆相关的配置。

|  |
| --- |
| Java –Xmx3550m –Xms3550m –Xmn2g –Xss128k –XX:NewRatio=4 –XX:SurvivorRatio=4 –xx:MaxPermSize=64m –XX:MaxTenuringThreshold=0 |

-Xmx3550:设置JVM最大可用内存为3550M

-Xms3550:设置JVM的初始内存为3550M，此值可以与最大内存一致，避免每次垃圾回收后JVM重新分配内存

-Xmn2g:设置年纪代大小为2G，整个堆大小=年轻代大小+年老代大小+持久代大小。持久代默认大小为64m，所有增加年轻代会减少年老代大小，因此此值非常重要，推荐为整个堆大小的3/8

-Xss128k:设置线程的堆栈大小，默认为1M,实际中需要根据应用进行调整，一般OS推荐的线程数为3000-5000。

-XX:NewRatio=4:设置年轻代与老年代的比值，即年亲代占年老代的1/4。

-XX:SurvivorRatio=4:设置年轻代中Eden区域Survivor区的大小比值，设置为4，即两个Survior区与一个Eden区的比值为2:4。

-XX:MaxPermSize=64m:设置持久代大小为64m

-XX:MaxTenuringThreshold=0:设置垃圾最大年龄，如果设置为0，则年轻代将不经过Survivor区，直接进入老年代，适合老年代较多的场景。

接下里介绍吞吐量优先的并行收集器和响应时间优先的并发收集器。Tip:这类应用推荐将年轻代设置的尽可能的大，尤其是吞吐量大的应用。

并行收集器

|  |
| --- |
| java -Xmx3550m -Xms3550 –Xmn2g –Xss128k –XX:+UseParallelGC –XX:ParallelGCThreads=20  –XX:+UseParallelOldGC -XX:MaxGCPauseMillis=100 –XX:UseAdaptiveSizePolicy |

-XX:+UseParallelGC:选择年轻代的垃圾收集器为并行收集器

-XX:ParallelGCThreads=20:设置并行收集器的线程数，最好和处理器数目一致

-XX:+UseParallelOldGC:配置年老代垃圾收集方式为并行收集

-XX:MaxGCPauseMillis=100:设置每次年轻代垃圾回收的最长时间，如果满足，则自动调整年亲代大小以满足此值。

-XX:+UseAdaptiveSizePolicy:设置此选项后，并行收集器自动选择年轻代区大小和相应Survivor区比例，推荐一直打开。

并发收集器

|  |
| --- |
| java -Xmx3550m -Xms3550 –Xmn2g –Xss128k –XX:ParallelGCThreads=20 -XX:+UseConcMarkSweepGC  –XX:+UseParNewGC -XX:CMSFullGCBeforeCompaction=5 –XX:UseCMSCompactAtFullCollection |

-XX:+UseConcMarkSweepGC(CMS):设置年老代为并发收集

-XX:+UseParNewGC:设置年轻代为并行收集，可以与CMS收集同时进行，现有版本无需设置

-XX:CMSFullGCBeforeCompaction=5:设置运行多少次GC后对内存空间进行压缩、整理

-XX:UseCMSCompactAtFullCollection：打开年老代的压缩，可以消除碎片但会影响性能

此外，还有一些展示GC辅助信息的配置： -XX：PrintGC, -XX:+PrintGCDetails, -XX:PrintGCTimeStamps, Xloggc:filename。

**Java内存模型**：不同的平台，内存模型是不一样的，但jvm内存模型规范是统一的，java多线程并发问题都会反映在java的内存模型上，所谓线程安全就是要控制多个线程对某个资源的有序访问和修改。总结的Java的内存模型，需要注意2个主要问题：**可见性和有序性**。

Tip:这部分内容理解起来有一定难度，需要多复习。

**可见性**：多个线程之间是不能相互传递数据通信的，它们之间的沟通需要通过共享变量。Java内存模型规定了jvm有主内存，主内存是多个线程共享的，当new一个对象时，也是被分配子啊**主内存**中的，每个线程都有自己的工作内存，工作内存存储了主存的某些对象的副本。当线程操作某个对象时，其执行顺序为：从主内复制变量当前工作内存(read and load)；执行代码，改变共享变量值(use and assign)；用工作内存数据刷新主存相关内容(store and write)。JVM规范定义了线程对主存的操作指令：read,load,use,assign,store,write。当一个共享变量在多个线程的工作内存中都有副本时，如果一个线程修改了这个共享变量，那么其他线程应该可以看到这个被修改后的值，这就是多线程的可见性问题。

**有序性**：线程在引用变量时不能直接从主内存中引用，如果线程工作内存中没有该变量，则会从主内存中拷贝一个副本到工作内存中，这个过程为read-load，完成后线程会引用该副本。当同一线程再度引用该字段时，就有可能重新从主内存中获取变量副本(read-load-use)，也有可能直接引用原来的副本(use)，也就是说read,load,use顺序可以有JVM实现系统决定。线程不能直接为主存中字段赋值，它会将值指定给工作内存中的副本变量(assign)，完成后这个变量副本会同步到主存储去(store-write)，至于何时同步过去，也有JVM决定。为了这部分操作的有序性，需要使用synchronized关键字，可以将方法变为同步方法public synchronized void add()，也可以增加同步变量static Object lock=new Object()，然后synchronized（lock）。每个锁对象都有两个队列，一个是就绪队列，一个是阻塞队列，就绪队列存储了将要获得锁的线程，阻塞队列存储被阻塞的线程，当一个线程被唤醒（nitify）后，才能进入到就绪队列，等待cpu调度。例如，当一个线程a第一次执行account.add方法是，jvm会检查锁对象account的就绪队列是否已经有线程在等待，如果有说明account被占用，此时是第一次运行，因此account就绪队列为空，所以线程a获得锁，执行方法。如果恰好这是线程b要执行account.withdraw方法，由于线程a获得的锁还未释放，因此b要进入account的就绪队列，等得到锁再执行。

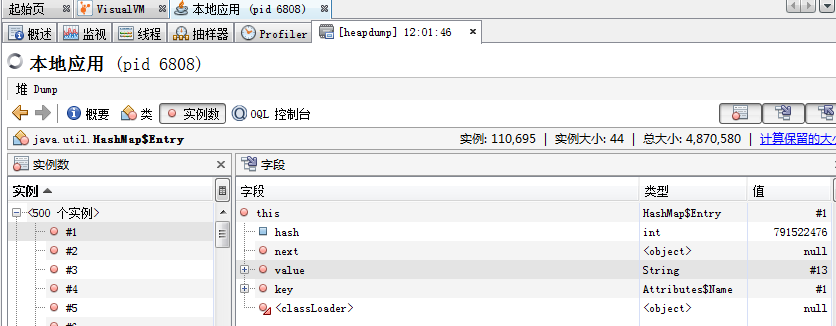
简单来说，一个线程执行临界区代码过程为：获得同步锁李晴空工作内存；从主存拷贝变量副本到工作内存；对这些变量进行计算；将变量从工作内存写回到主存；释放锁。

**生产者-消费者模型**：这是一个非常经典的线程同步模型，有时不光需要保证多个线程多一个共享资源操作的互斥性，往往多个线程见都是有协作的，一个简单的例子如下所示。

|  |
| --- |
| public class Plate {  private List<Object> eggs = new ArrayList<Object>();  public synchronized Object getEgg() {  if (eggs.size() == 0) {  try {  wait();  } catch (InterruptedException e) {  }  }  Object egg = eggs.get(0);  eggs.clear();  notify();  return egg;  }    public synchronized void putEgg(Object egg) {  if (eggs.size() > 0) {  try {  wait();  } catch (InterruptedException e) {  }  }  eggs.add(egg);  notify();  }  } |

**Volatile关键字**：volatile是java的一种轻量级同步手段，它只提供多线程内存的可见性，不保证执行的有序性。其意义在于，任何线程对volatile修饰的变量进行修改，都会马上被其他线程读取到，因为直接操作主存，没有线程对工作内存和主存同步。其使用场景为：对变量的写操作不依赖于当前值；该变量没有包含在具有其他变量的不定式中。

**JVM调用工具**：常见的包括Jconsole、JProfile和VisualVM，推荐使用VisualVM。所有的调优都源于对线上应用的监控和分析,主要需要观察内存的释放情况、集合类检查、对象树等。如下图所示，通过查看集合实例的情况来分析。通过这**类堆信息**查看，可以分析出年老代年轻代划分是否合理、内存是否泄漏、垃圾回收算法是否合适等问题。



此外，还可以通过线程监控了解系统的线程数量和线程的状态，是否死锁等；通过抽样器查看CPU和内存热点的情况；通过快照来了解不同时刻相关状态的差异。

**内存泄漏**的检查：内存泄漏一般可以理解为系统资源在错误使用的情况下，导致使用完毕的资源无法回收，从而导致新的资源分配请求无法完成，引起系统错误。其常见场景为：年老代堆空间被占满（java.lang.OutOfMemoryError:Java heap space），可以通过堆大小的变化发现问题；持久代被占满(java.lang.OutOfMemoryError:PermGen space)，在大量使用反射时会出现；堆栈溢出(java.lang.StackOverflowError)，一般因为错误的递归和循环造成；线程堆栈满(Fatal:Stack size too small)，可以通过修改-Xss解决，不过还是主要注意是否是因为线程栈过深造成；系统内存被占满(java.lang.OutOfMemoryError:unable to create new native thread)，由于OS没有足够的资源来产生线程造成的，可以考虑减少单个线程的消耗或重新设计这部分程序。

**常见问题**

1.堆和栈的区别：堆是存放对象的，但是对象内临时变量是存在栈内存中的。栈是跟随线程的，有线程就有栈，堆是跟随JVM的，有JVM就有堆内存。

2.堆内存中到底存在什么：对象，包括对象变量和对象方法。

3.类变量和实例变量有什么区别：静态变量(有static修饰)是类变量，非静态变量是实例变量。静态变量存在方法区中，实例变量存在堆内存中。有个说法是类变量是在JVM启动时就初始化好了，其实不对。

4.Java的方法到底是传值还是传引用：都不是，而是以传值的方式传递地址，具体的说就是原始数据类型传递的值，引用类型传递的地址。对于原始数据类型，JVM的处理方法是从Method Area或Heap中拷贝到Stack，然后运行Frame中方法，运行完毕再将变量拷贝回去。

5.为什么会产生OutOfMemory：原因是Heap内存中没有可用空间了或永久区满了，有时会发现对象不多仍出现该情况，一般是由继承层次过多造成，因为Heap中产生的对象都是先产生父类，然后产生子类。

6.为什么会产生StackOverFlowError：因为线程把栈空间消耗完了，一般都是递归函数造成的。

7.JVM中那些共享的，那些是私有的:Heap和Method Area是共享的，其他都是私有的。

8.还有那些需要注意的补充概念：常量池(constant pool)，按照顺序存放程序中的常量，且进行索引编号，默认0到127放在常量池，string也是；安全管理器(Security Manager)，提供java运行期的安全控制，类加载器只有在通过认证后才能加载class文件；方法索引表(Methods table)，记录每个method的地址信息，Stack和Heap中的地址指针其实指向Methods table的地址。

9.为什么不能调用System.gc():因为该操作会进行Full GC并停止所有活动。

10.CGLib是什么：用于Spring和Hibernate等技术对类进行增强时，其可以直接操作字节码动态生成Class文件。

参考资料:

1. 周志明. 深入理解Java虚拟机：JVM高级特性与最佳实践（第2版）[M]. 北京:机械工业出版社, 2013.
2. JVM调优[EB/OL]. http://hllvm.group.iteye.com/group/wiki/?category\_id=301.