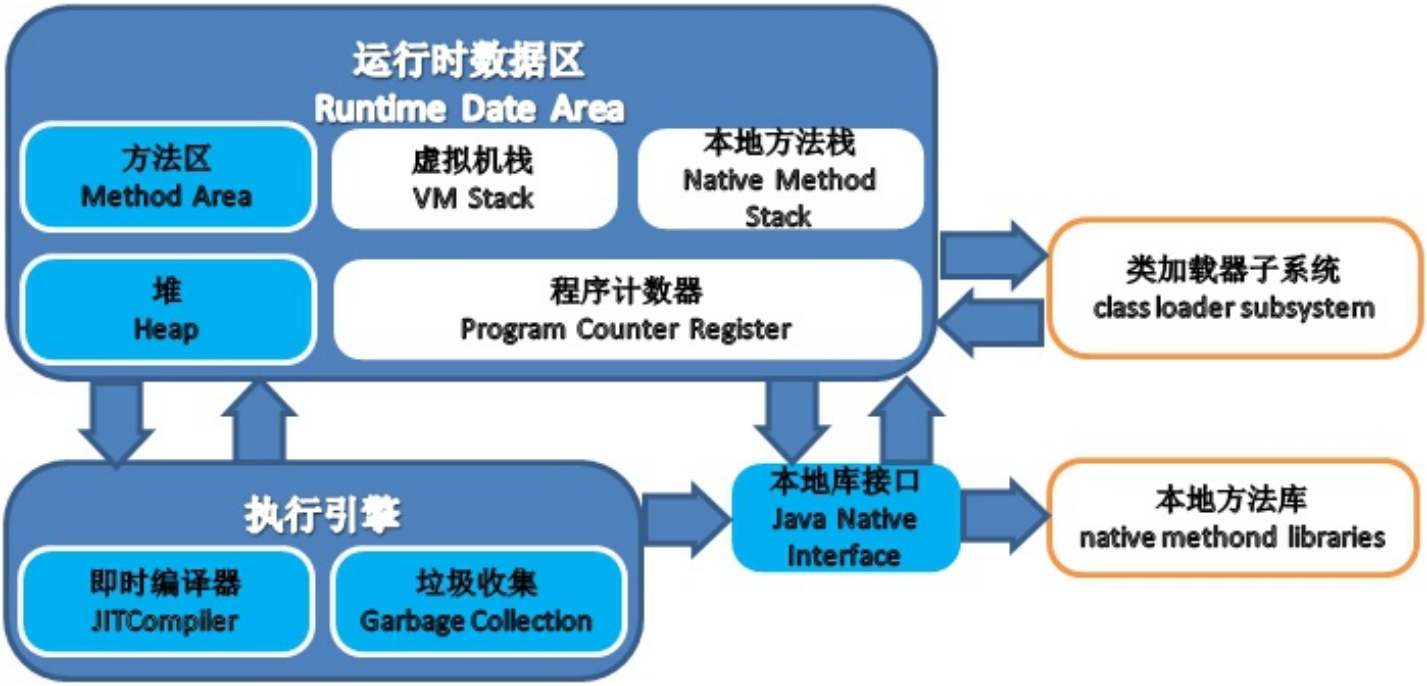
介绍

> JVM是Java Virtual Machine（Java虚拟机）的缩写，JVM是一种用于计算设备的规范，它是一个虚构出来的计算机，是通过在实际的计算机上仿真模拟各种计算机功能来实现的。

JVM运行时数据区

对于Java程序员来说，在虚拟机自动内存管理机制的帮助下，不再需要为每一个new操作去写对应的delete/free代码，不容易出现内存泄漏和内存溢出问题，由虚拟机管理内存。不过，也正是Java程序员把内存控制的权力交给了Java虚拟机，一旦出现内存泄漏和溢出方面的问题，如果不了解Java虚拟机是如何使用内存的，那么排查错误将会成为一项异常艰难的工作。



可以看出，JVM主要由类加载器子系统、运行时数据区（内存空间）、执行引擎以及与本地方法接口等组成。其中运行时数据区又由方法区、堆、Java栈、PC寄存器、本地方法栈组成。

程序计数器

记录当前线程所执行到的字节码的行号。

- 此内存区域是唯一一个在JVM上不会发生内存溢出异常（OutOfMemoryError）的区域。
- 线程私有的数据区

虚拟机栈

描述Java方法执行的内存模型。每个方法在执行的同时都会开辟一段内存区域用于存放方法运行时所需的数据，成为栈帧，一个栈帧包含如：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。
- 如果在动态扩展内存的时候无法申请到足够的内存，就会抛出OutOfMemoryError异常。
- 线程私有的数据区

本地方法栈

为JVM所调用到的Nativa即本地方法服务。

- 和虚拟机栈出现的异常很相像
- 线程私有的数据区

堆

所有线程共享一块内存区域，在虚拟机开启的时候创建。

- 存储对象实例，更好地分配内存
- 垃圾回收（GC）。堆是垃圾收集器管理的主要区域
- 如果堆上没有内存进行分配，并无法进行扩展时，将会抛出OutOfMemoryError异常

方法区

用于存储运行时常量池、已被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。

- 当方法区无法满足内存分配需求时，将抛出OutOfMemoryError异常
- 线程共享的区域

垃圾收集器

对象的存活?

- 引用计数法
- 可达性分析算法

```
虚拟机栈（栈帧中的本地变量表）中引用的对象
方法区中类静态属性引用的对象
方法区中常量引用的对象
本地方法栈中JIT（即一般说的Native方法）引用的对象
```

引用

思想:

```
食之无味，弃之可惜
```

java四种引用:

- 强引用
- 软引用
- 弱引用
- 虚引用

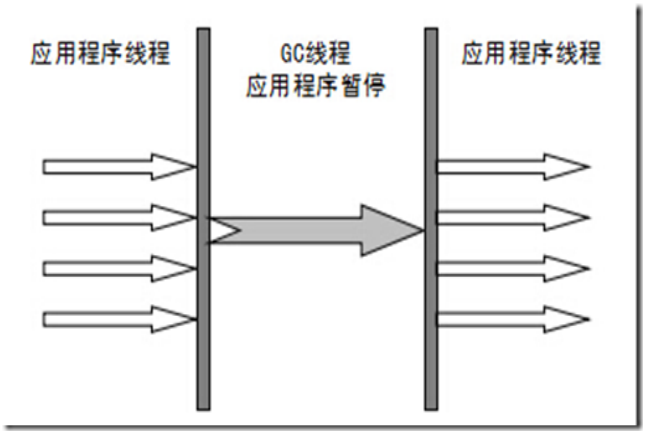
垃圾回收算法

- 标记-清除算法
- 复制算法
- 标记-整理算法
- 分代收集算法

收集器

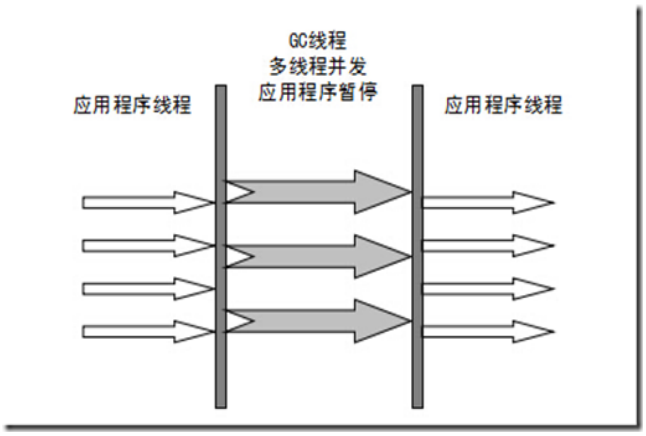
- Serial 收集器 (新生代)

暂停所有线程
复制算法



- ParNew 收集器 (新生代)

Serial 的多线程版本，使用多线程进行垃圾收集
复制算法



- Parallel Scavenge 收集器(新生代)

达到一个可控制的吞吐量
复制算法
最大垃圾收集停顿时间 -XX：MaxGCPauseMillis
直接设置吞吐量大小：-XX:GCTimeRatio

- Serial Old 收集器 (老年代)

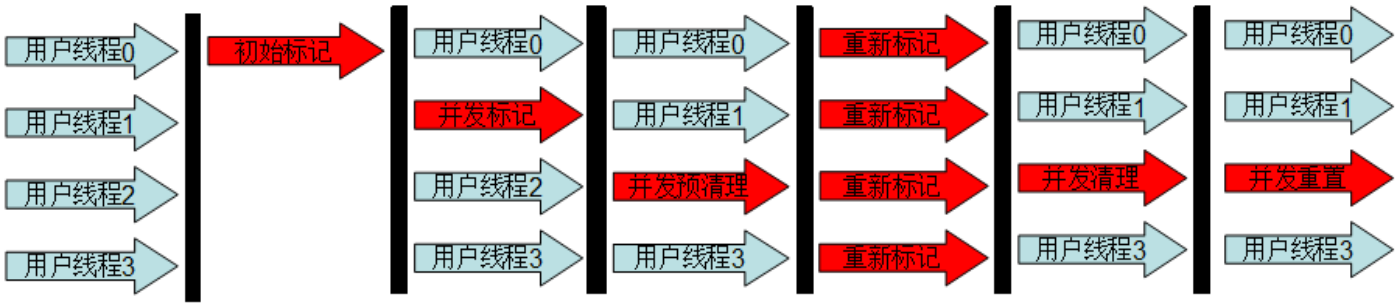
单线程的收集器
暂停所有线程
标记-整理算法

- Parallel Old 收集器 (老年代)

```
Parallel Scavenge收集器的老年代版本,
使用多线程
标记-整理”算法
```

- CMS 收集器 (老年代)

```
初始标记(STW initial mark)
并发标记(Concurrent marking)
并发预清理(Concurrent precleaning)
重新标记(STW remark)
并发清理(Concurrent sweeping)
并发重置(Concurrent reset)
```




```
cms对cpu资源非常的敏感 (cup数量+3)/4
cms无法回收浮动垃圾
cms会产生碎片

 – XX:CMSInitiatingOccupancyFraction

 - XX:+UseCMSCompactAtFullCollection
 - XX:CMSFullGCsBeforeCompaction 多少次cms之后，执行一次碎片的整理
```

```
[GC [1 CMS-initial-mark: 349375K(638976K)] 393510K(1007616K), 0.0424478 secs] [Times
[CMS-concurrent-mark-start]
[CMS-concurrent-mark: 0.216/0.216 secs] [Times: user=1.00 sys=0.13, real=0.22 secs]
[CMS-concurrent-preclean-start]
[CMS-concurrent-preclean: 0.004/0.004 secs] [Times: user=0.03 sys=0.00, real=0.01 se
[CMS-concurrent-abortable-preclean-start]
[CMS-concurrent-abortable-preclean: 4.238/5.134 secs] [Times: user=5.38 sys=0.78, re
[GC[YG occupancy: 77380 K (368640 K)]2017-05-18T20:32:26.429+0800: 447.591: [Rescan
[1 CMS-remark: 349375K(638976K)] 426755K(1007616K), 0.0718764 secs] [Times: user=0.1
[CMS-concurrent-sweep-start]
[CMS-concurrent-sweep: 0.256/0.256 secs] [Times: user=0.25 sys=0.00, real=0.26 secs]
[CMS-concurrent-reset-start]
[CMS-concurrent-reset: 0.003/0.003 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
```

内存分配策略

自动给对象分配内存以及回收分配的内存

简单常用的jvm参数

- -XX:+PrintGCDetails
- -Xms
- -Xmx
- -Xmn
- -XX:NewRatio
- -XX:SurvivorRatio
- -Xloggc
- -XX:+HeapDumpOnOutOfMemoryError
- -XX:HeapDumpPath
- -XX:MaxTenuringThreshold
- -XX:MetaspaceSize
- -XX:MaxMetaspaceSize

垃圾收集器常用的GC参数

- 与串行回收器相关的参数

```
-XX:+UseSerialGC:在新生代和老年代使用串行收集器
-XX:SurvivorRatio:设置eden区大小和survivor区大小的比例
-XX:PretenureSizeThreshold:设置大对象直接进入老年代的阈值。当对象的大小超过这个值时，将直接在老年代分配。
-XX:MaxTenuringThreshold:设置对象进入老年代的年龄的最大值。每一次Minor GC后，对象年龄就加1。任何大于这个年龄的对象，一定会进入老年代。
```

- 并行GC相关的参数

```
-XX:+UseParNewGC:在新生代使用并行收集器
-XX:+UseParallelOldGC:老年代使用并行回收收集器
-XX:ParallelGCThreads:设置用于垃圾回收的线程数。通常情况下可以和CPU数量相等，但在CPU数量较多的情况下，设置相对较小的数值也是合理的。
-XX:MaxGCPauseMillis:设置最大垃圾收集停顿时间。他的值是一个大于0的整数。收集器在工作时，会调整Java堆大小或者其他参数，尽可能把停顿时间控制在MaxGCPauseMillis以内。
-XX:GCTimeRatio:设置吞吐量大小。它是0-100的整数。假设GCTimeRatio的值为n，那么系统将花费不超过1/(1+n)的时间用于垃圾收集。
-XX:+UseAdaptiveSizePolicy:打开自适应GC策略。在这种模式下，新生代的大小、eden和survivor的比例、晋升老年代的对象年龄等参数会被自动调整，已达到在堆大小、吞吐量和停顿时间之间的平衡点。
```

- 与CMS回收期相关的参数

```
-XX:+UseConcMarkSweepGC:新生代使用并行收集器，老年代使用CMS+串行收集器
-XX:ParallelCMSThreads:设定CMS的线程数量
-XX:CMSInitiatingOccupancyFraction:设置CMS收集器在老年代空间被使用多少后触发，默认为68%
-XX:+UseCMSCompactAtFullCollection:设置CMS收集器完成垃圾收集后是否要进行一次内存碎片的整理
-XX:CMSFullGCsBeforeCompaction:设定进行多少次CMS垃圾回收后，进行一次内存压缩
```

分配策略

- 对象优先分配在Eden区
- 大对象直接进入老年代
- 长期存活的对象将进入老年代
- 空间分配担保

- 新生代GC(Minor GC)

```
当Eden区满时，触发Minor GC
```

- 老年代GC(Full GC)

```
System.gc()方法的调用
老年代代空间不足
永生区空间不足
CMS GC时出现promotion failed和concurrent mode failure
堆中分配很大的对象
```

虚拟机性能监控与故障处理工具

命令行工具

- `jps` ： JVM Process Status Tool，显示指定系统内所有HotSpot虚拟机进程

```
-q 不输出类名、Jar名和传入main方法的参数
-m 输出传入main方法的参数
-l 输出main类或Jar的全限名
-v 输出传入JVM的参数
```

- `jstat` : JVM Statistics Minitoring Tool，用于收集HotSpot虚拟机各方面的运行数据

```
-class  监视类装载、卸载数量、总空间及类装载所耗费的时间
-gc  监视Java堆状况，包括Eden区、2个Survivor区、老年代、永久代等的容量
-gccapacity 监视内容与-gc基本相同，但输出主要关注Java堆各个区域使用到的最大和最小空间
-gcutil 监视内容与-gc基本相同，但输出主要关注已使用空间占总空间的百分比
-gccause    与-gcutil功能一样，但是会额外输出导致上一次GC产生的原因
-gcnew  监视新生代GC的状况
-gcnewcapacity  监视内容与-gcnew基本相同，输出主要关注使用到的最大和最小空间
-gcold  监视老年代GC的状况
-gcoldcapacity  监视内容与--gcold基本相同，输出主要关注使用到的最大和最小空间
```

- `jinfo` :Java配置信息工具

```
-flag< name >: 打印指定java虚拟机的参数值。
-flag [+|-]< name >: 设置或取消指定java虚拟机参数的布尔值。
-flag < name >=< value >: 设置指定java虚拟机的参数的值
```

- `jmap` ： Java内存映像工具

```
-dump    生成Java堆转储快照。格式为：-dump:[live,]format=b,file=<filename>，其中live
子参数说明是否只dump出存活的对象
-heap    显示Java堆详细信息，如使用哪种回收器、参数配置、分代状况等。
-histo   显示堆中对象统计信息，包括类、实例数量和合计容量
```

- `jstack`： Java堆栈跟踪工具

```
-F 没有相应的时候强制打印栈信息
-l 长列表. 打印关于锁的附加信息
```

- `jhat`： 虚拟机堆转储快照分析工具(不推荐)

JDK的可视化工具

- JConsole
- VisualVM

实战

分析堆内存OOM

-Xmx256m -Xms256m -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/Users/heganzhi/JavaHeapTest.dump

```
public class JavaHeapTest {
    public final static int OUTOFMEMORY = 900000000;

    private String oom;

    private int length;

    StringBuffer tempOOM = new StringBuffer();

    public JavaHeapTest(int leng) {
        this.length = leng;

        int i = 0;
        while (i < leng) {
            i++;
            try {
                tempOOM.append("a");
            } catch (OutOfMemoryError e) {
               e.printStackTrace();
               break;
            }
        }
        this.oom = tempOOM.toString();

    }

    public String getOom() {
        return oom;
    }

    public int getLength() {
        return length;
    }

    public static void main(String[] args) {
        JavaHeapTest javaHeapTest = new JavaHeapTest(OUTOFMEMORY);
        System.out.println(javaHeapTest.getOom().length());
    }

}
```

分析线程状态和cpu情况

```
public class JtackCase {
    public static Executor executor = Executors.newFixedThreadPool(5);
    public static Object lock = new Object();

    public static void main(String[] args) {
        Task task1 = new Task();
        Task task2 = new Task();
        executor.execute(task1);
        executor.execute(task2);

    }

    static class Task implements Runnable{

        @Override
        public void run() {
            synchronized (lock){
                try {
                   cal();
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }

        public void cal(){
            int i = 0;
            while (true){
                i++;
            }

        }
    }
}
```